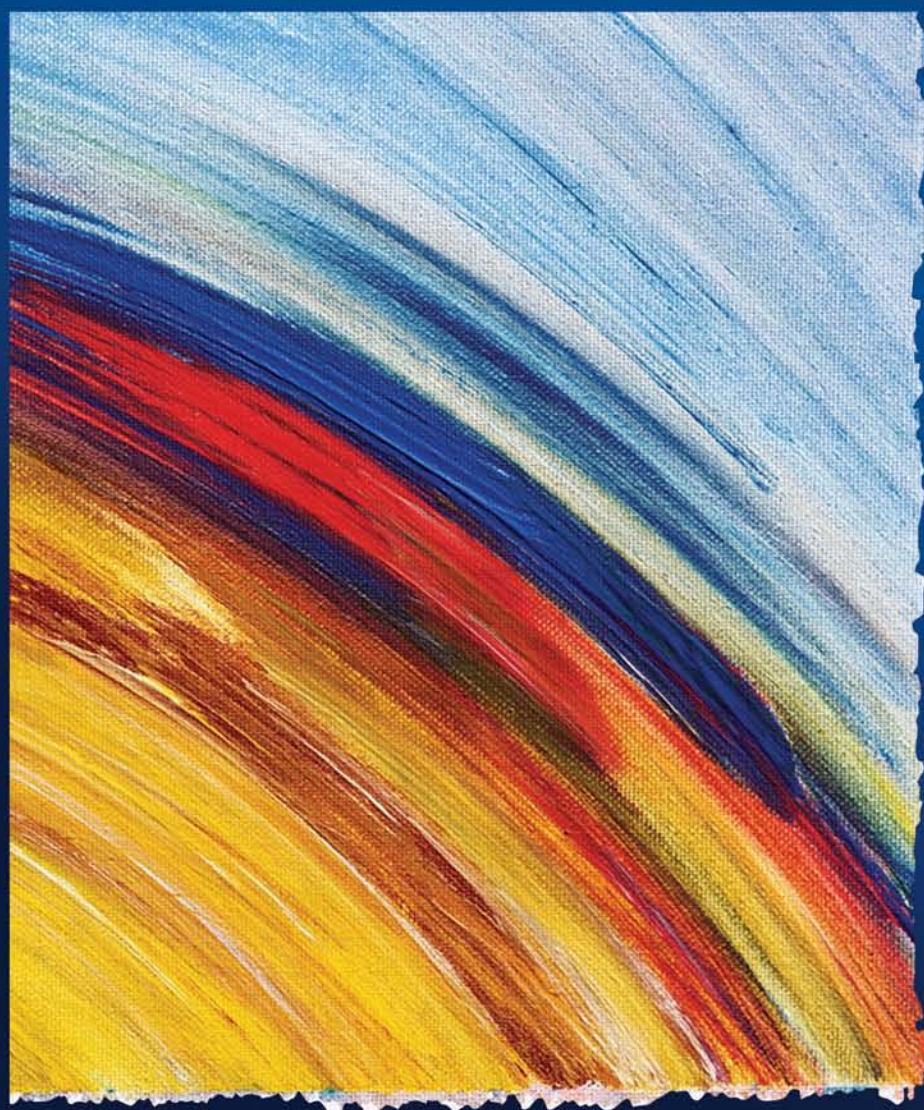


Roger S. Pressman • Bruce R. Maxim

Engenharia de Software

UMA ABORDAGEM PROFISSIONAL

8ª EDIÇÃO



**Mc
Graw
Hill
Education**



**Adquirido em
www.mercadolivre.com.br**

**Vendedor: GFS-USA
Clique aqui para obter novos títulos**



P934e Pressman, Roger S.
Engenharia de software : uma abordagem profissional
recurso eletrônico / Roger S. Pressman, Bruce R. Maxim ;
tradução: João Eduardo Nóbrega Tortello ; revisão técnica:
Reginaldo Arakaki, Julio Arakaki, Renato Manzan de
Andradel. – 8. ed. – Porto Alegre : AMGH, 2016.

Editado como livro impresso em 2016.
ISBN 978-85-8055-534-9

1. Engenharia de software. 2. Gestão de projetos de
softwares. I. Maxim, Bruce R. II. Título.

CDU 004.41

ROGER S. PRESSMAN, Ph.D.
BRUCE R. MAXIM, Ph.D.

Engenharia de Software

UMA ABORDAGEM PROFISSIONAL

8ª EDIÇÃO

Revisão técnica:

Reginaldo Arakaki

Professor Doutor do Departamento de Engenharia de Computação
e Sistemas Digitais da EPUSP

Julio Arakaki

Professor Doutor do Departamento de Computação da PUC-SP

Renato Manzan de Andrade

Doutor pelo Departamento de Engenharia de Computação
e Sistemas Digitais da EPUSP

Versão impressa
desta obra: 2016



AMGH Editora Ltda.

2016

Obra originalmente publicada sob o título
Software Engineering: A Practitioner's Approach, 8th Edition
ISBN 978-0-07-802212-8

Edição original em língua inglesa copyright ©2015, McGraw-Hill Global Education Holdings, LLC., New York, New York 10121. Todos os direitos reservados.
Tradução para língua portuguesa copyright ©2016, AMGH Editora Ltda., uma empresa do Grupo A Educação S.A. Todos os direitos reservados.

Tradução da 8ª edição: *João Eduardo Nóbrega Tortello*

Textos traduzidos por *Ariovaldo Griesi* e *Mario Moro Fecchio* da 7ª edição da obra foram utilizados nesta nova edição.

Gerente editorial: *Arysinha Jacques Affonso*

Colaboraram nesta edição:

Editora: *Mariana Belloli*

Leitura final: *Frank Holbach Duarte*

Capa: *Kaéle Finalizando Ideias*, arte sobre capa original

Editoração eletrônica: *Techbooks*

Reservados todos os direitos de publicação, em língua portuguesa, à
AMGH EDITORA LTDA., uma parceria entre GRUPO A EDUCAÇÃO S.A.
e MCGRAW-HILL EDUCATION
Av. Jerônimo de Ornelas, 670 – Santana
90040-340 – Porto Alegre – RS
Fone: (51) 3027-7000 Fax: (51) 3027-7070

Unidade São Paulo
Av. Embaixador Macedo Soares, 10.735 – Pavilhão 5 – Cond. Espace Center
Vila Anastácio – 05095-035 – São Paulo – SP
Fone: (11) 3665-1100 Fax: (11) 3667-1333

SAC 0800 703-3444 – www.grupoa.com.br

É proibida a duplicação ou reprodução deste volume, no todo ou em parte, sob quaisquer formas ou por quaisquer meios (eletrônico, mecânico, gravação, fotocópia, distribuição na Web e outros), sem permissão expressa da Editora.

IMPRESSO NO BRASIL
PRINTED IN BRAZIL

Para minhas netas, Lily e Maya, que já entendem a importância do software, mesmo ainda estando na pré-escola.

Roger S. Pressman

Em memória de meus queridos pais, que me ensinaram desde cedo que buscar uma boa educação era bem mais importante do que ir atrás de dinheiro.

Bruce R. Maxim

Os autores

Roger S. Pressman é consultor e autor reconhecido internacionalmente. Por mais de quatro décadas, trabalhou como engenheiro de software, gerente, professor, escritor, consultor e empresário.

É presidente da R. S. Pressman & Associates, Inc., uma empresa de consultoria especializada em ajudar empresas a estabelecerem práticas eficazes de engenharia de software, e com a qual desenvolveu um conjunto de técnicas e ferramentas que melhoram a prática de engenharia de software. É também o fundador da Teslaccessories, LLC, uma *start-up* do ramo da manufatura especializada em produtos para o veículo elétrico Tesla Model S.

É autor de nove livros, incluindo dois romances, e ensaios técnicos e gerenciais. Participou da comissão editorial dos periódicos *IEEE Software* e *The Cutter IT Journal* e foi editor da coluna “Manager” na *IEEE Software*.

É palestrante renomado, apresentando-se nas principais conferências do setor, entre elas a International Conference on Software Engineering. Foi associado da ACM e da IEEE.

Bruce R. Maxim trabalhou como engenheiro de software, gerente de projeto, professor, escritor e consultor por mais de 30 anos. Sua pesquisa inclui engenharia de software, interação humano-computador, projeto de games, mídia social, inteligência artificial e educação em ciência da computação.

É professor adjunto de Ciência da Computação e Informação na Universidade de Michigan-Dearborn. Organizou o GAME Lab na Faculdade de Engenharia e Ciência da Computação. Publicou vários artigos sobre algoritmos de computador para animação, desenvolvimento de jogos e educação em engenharia. É coautor de um texto introdutório de sucesso sobre ciência da computação. Como parte de seu trabalho na UM-Dearborn, supervisionou centenas de projetos de desenvolvimento de software industrial.

Sua experiência profissional inclui o gerenciamento de sistemas de informações de pesquisa em uma faculdade de Medicina, direção educacional da computação em um campus de Medicina e atuação como programador estatístico. Trabalhou como diretor-chefe de tecnologia em uma empresa de desenvolvimento de games.

Recebeu vários prêmios por distinção no ensino e um por notável serviço comunitário. É associado da Association of Computing Machinery, IEEE Computer Society, American Society for Engineering Education, Society of Women Engineers e International Game Developers Association.

Prefácio

Quando um software é bem-sucedido – ou seja, atende às necessidades dos usuários, opera perfeitamente durante um longo período de tempo, é fácil de modificar e mais fácil ainda de utilizar –, ele pode mudar, e de fato muda, as coisas para melhor. Entretanto, quando um software é falho – quando seus usuários estão insatisfeitos, quando é propenso a erros, quando é difícil de modificar e mais difícil ainda de utilizar –, coisas desagradáveis podem acontecer, e de fato acontecem. Todos queremos construir software que facilite o trabalho, evitando as falhas que se escondem nos esforços malsucedidos. Para termos êxito, precisamos de disciplina no projeto e na construção do software. Precisamos de uma abordagem de engenharia.

Já faz quase três décadas e meia que a 1ª edição deste livro foi escrita. Durante esse período, a engenharia de software evoluiu de algo obscuro, praticado por um número relativamente pequeno de adeptos, para uma disciplina de engenharia legítima. Hoje, ela é reconhecida como um assunto digno de pesquisa séria, estudo metódico e debates acalorados. Em toda a indústria, o engenheiro de software tomou o lugar do programador como cargo mais procurado. Modelos de processos de software e métodos de engenharia de software, bem como ferramentas de software vêm sendo adotados com sucesso em um amplo espectro de segmentos industriais.

Embora gerentes e profissionais da área reconheçam a necessidade de uma abordagem mais disciplinada ao software, eles continuam a discutir a maneira como essa disciplina deve ser aplicada. Muitos profissionais e empresas desenvolvem software de forma desordenada, mesmo ao construir sistemas dirigidos às mais avançadas tecnologias. Grande parte dos profissionais e estudantes não está ciente dos métodos modernos, e, como consequência, a qualidade do software que produzem é afetada. Também continuam a discussão e a controvérsia sobre a real natureza da abordagem de engenharia de software. A engenharia de software é um estudo repleto de contrastes. A postura mudou, e progressos foram feitos; porém, falta muito para essa disciplina atingir a maturidade.

A 8ª Edição. O objetivo desta 8ª edição é ser um guia para uma disciplina de engenharia em fase de amadurecimento. Assim como as edições que a precederam, ela é voltada tanto para estudantes no final do curso de graduação ou no primeiro ano de pós-graduação quanto para profissionais da área.

A 8ª edição é muito mais do que uma simples atualização. O livro foi revisado e reestruturado para melhorar seu fluxo pedagógico e enfatizar novos e importantes processos e práticas da engenharia de software. Seus 39 capítulos estão organizados em cinco partes. Essa organização divide melhor os assuntos e ajuda os professores que não tenham tempo para concluir o livro inteiro em um semestre.

A Parte I, *O processo de software*, apresenta diferentes visões de processo de software, considerando todos os modelos importantes e contemplando o debate entre as filosofias de processos ágeis e prescritivos. A Parte II, *Modelagem*, fornece métodos de projeto e análise com ênfase em técnicas orientadas a objetos e modelagem UML. Também é considerado o projeto baseado em padrões, bem como o projeto para aplicações Web e aplicativos móveis. A Parte III, *Gestão da qualidade*, apresenta conceitos, procedimentos, técnicas e métodos que permitem que uma equipe de software avalie a qualidade do software, revise produtos gerados por engenharia de software, realize procedimentos para a garantia de qualidade de software (SQA) e aplique estratégias e táticas de teste eficaz. Além disso, são considerados também métodos de modelagem e verificação formais. A Parte IV, *Gerenciamento de projetos de software*, aborda tópicos relevantes para os que planejam, gerenciam e controlam um projeto de desenvolvimento de software. A Parte V, *Tópicos avançados*, considera o aperfeiçoamento de processos de software e tendências da engenharia de software.

Preservando a tradição das edições anteriores, são usadas caixas de texto para apresentar as atribuições de uma equipe (fictícia) de desenvolvimento de software e fornecer conteúdo complementar sobre métodos e ferramentas relevantes para os tópicos do capítulo. Pequenos textos na margem lateral das páginas enriquecem ainda mais o conteúdo; eles seguem esta convenção:

Ponto-chave: textos em preto e negrito apresentam conceitos ou ideias importantes destacados do texto.

“Citação”: textos entre aspas e em itálico trazem comentários e frases pertinentes ao assunto discutido.

Pergunta: textos em azul e negrito trazem questões que ajudam a fixar o conteúdo apresentado.

WebRef: textos em fonte comum apresentam links para sites e outras fontes eletrônicas interessantes.

Aviso: textos em azul e itálico informam os pontos que o leitor deve dar mais atenção.

A organização em partes permite agrupar o conteúdo em tópicos considerando o tempo disponível e a necessidade dos alunos. Um curso de um semestre pode ser baseado em uma ou mais das cinco partes. Um curso de pesquisa em engenharia de software selecionaria capítulos de todas as partes. Um curso de engenharia de software que enfatize a análise e o projeto selecionaria tópicos das Partes I e II. Um curso de engenharia de software voltado para testes selecionaria tópicos das Partes I e III, com uma breve incursão na Parte II. Um “curso de gerenciamento” enfatizaria as Partes I e IV. Organizada dessa maneira, a 8ª edição oferece ao professor diferentes opções didáticas.

Recursos para estudantes e profissionais Resumos dos capítulos (em inglês) estão disponíveis para download no site do Grupo A. Acesse o site, www.grupoa.com.br, cadastre-se gratuitamente, encontre a página do livro por meio do campo de busca e clique no link Conteúdo Online para fazer o download.

Recursos para professores Professores podem fazer download de material complementar exclusivo (em inglês) no site do Grupo A. Acesse o site,

www.grupoa.com.br, cadastre-se gratuitamente como professor, encontre a página do livro por meio do campo de busca e clique no link Material para o Professor. O material disponível (em inglês) inclui:

- Guia do Instrutor;
- Mais de 700 *slides* em PowerPoint®;
- Testes, questões para exame e soluções;
- Modelo de arquitetura para o sistema *CasaSegura*;
- Modelo UML para o sistema *CasaSegura*.

O site da edição em inglês deste livro oferece uma ampla gama de recursos para estudantes, professores e profissionais – alguns de acesso livre, outros de acesso restrito. Caso tenha interesse em explorar esses recursos, acesse **www.mhhe.com/engineering/pressman/**. Como toda fonte baseada na Web, esse endereço eletrônico e o conteúdo lá disponível estão sujeitos a serem retirados da Web a qualquer momento. A gestão desse conteúdo é feita exclusivamente pelos autores e pela editora original, logo, o Grupo A não se responsabiliza pela disponibilização do conteúdo caso o site do livro deixe de existir.

Agradecimentos. Agradecimentos especiais a Tim Lethbridge, da Universidade de Ottawa, que nos ajudou no desenvolvimento de exemplos em UML e OCL e do estudo de caso que acompanha este livro, bem como a Dale Skrien, do Colby College, que desenvolveu o tutorial UML do Apêndice 1. Sua ajuda e comentários foram inestimáveis.

Além disso, gostaríamos de agradecer a Austin Krauss, engenheiro de software sênior da Treyarch, por dar ideais sobre desenvolvimento de software no setor de games. Também gostaríamos de agradecer aos revisores da 8ª edição: Manuel E. Bermudez, Universidade da Flórida; Scott DeLoach, Kansas State University; Alex Liu, Michigan State University; e Dean Mathias, Utah State University. Seus comentários minuciosos e suas críticas construtivas nos ajudaram a tornar este livro muito melhor.

Agradecimentos especiais. BRM: Estou muito agradecido pela oportunidade de trabalhar com Roger na 8ª edição deste livro. Enquanto estava trabalhando no livro, meu filho, Benjamin, lançou seu primeiro aplicativo móvel e minha filha, Katherine, iniciou sua carreira de designer de interiores. Estou muito contente de ver os adultos que se tornaram. Agradeço muito à minha esposa, Norma, pelo apoio entusiasmado que me deu quando preenchi meu tempo livre com o trabalho nesta obra.

RSP: Assim como as edições deste livro evoluíram, meus filhos, Mathew e Michael, cresceram e se tornaram homens. Sua maturidade, caráter e sucesso me inspiraram. Nada mais me deixou tão orgulhoso. Agora eles têm suas próprias filhas, Maya e Lily, que dão início a mais uma geração. As duas já são especialistas em dispositivos de computação móvel. Por fim, à minha esposa, Barbara, meu amor, agradeço por ter tolerado as várias horas que dediquei ao trabalho e por ter me incentivado a fazer mais uma edição “do livro”.

*Roger S. Pressman
Bruce R. Maxim*

Esta página foi deixada em branco intencionalmente.

Sumário

CAPÍTULO 1 A natureza do software 1

- 1.1 A natureza do software 3
 - 1.1.1 Definição de software 4
 - 1.1.2 Campos de aplicação de software 6
 - 1.1.3 Software legado 7
- 1.2 A natureza mutante do software 9
 - 1.2.1 WebApps 9
 - 1.2.2 Aplicativos móveis 9
 - 1.2.3 Computação em nuvem 10
 - 1.2.4 Software para linha de produtos (de software) 11
- 1.3 Resumo 11

Problemas e pontos a ponderar 12

Leituras e fontes de informação complementares 12

CAPÍTULO 2 Engenharia de software 14

- 2.1 Definição da disciplina 15
- 2.2 O processo de software 16
 - 2.2.1 A metodologia do processo 17
 - 2.2.2 Atividades de apoio 18
 - 2.2.3 Adaptação do processo 18
- 2.3 A prática da engenharia de software 19
 - 2.3.1 A essência da prática 19
 - 2.3.2 Princípios gerais 21
- 2.4 Mitos do desenvolvimento de software 23
- 2.5 Como tudo começa 26
- 2.6 Resumo 27

Problemas e pontos a ponderar 27

Leituras e fontes de informação complementares 27

PARTE I O processo de software 29

CAPÍTULO 3 Estrutura do processo de software 30

- 3.1 Um modelo de processo genérico 31
- 3.2 Definição de uma atividade metodológica 32
- 3.3 Identificação de um conjunto de tarefas 34
- 3.4 Padrões de processo 34
- 3.5 Avaliação e aperfeiçoamento de processos 37
- 3.6 Resumo 38

Problemas e pontos a ponderar 38

Leituras e fontes de informação complementares 39

CAPÍTULO 4 Modelos de processo 40

- 4.1 Modelos de processo prescritivo 41
 - 4.1.1 O modelo cascata 41
 - 4.1.2 Modelos de processo incremental 43
 - 4.1.3 Modelos de processo evolucionário 44
 - 4.1.4 Modelos concorrentes 49
 - 4.1.5 Um comentário final sobre processos evolucionários 51
- 4.2 Modelos de processo especializado 52
 - 4.2.1 Desenvolvimento baseado em componentes 52
 - 4.2.2 O modelo de métodos formais 53
 - 4.2.3 Desenvolvimento de software orientado a aspectos 54
- 4.3 O processo unificado 55
 - 4.3.1 Um breve histórico 56
 - 4.3.2 Fases do processo unificado 56
- 4.4 Modelos de processo pessoal e de equipe 58
 - 4.4.1 Processo de Software Pessoal 59
 - 4.4.2 Processo de Software de Equipe 60
- 4.5 Tecnologia de processos 61
- 4.6 Produto e processo 62
- 4.7 Resumo 63
- Problemas e pontos a ponderar 64
- Leituras e fontes de informação complementares 65

CAPÍTULO 5 Desenvolvimento ágil 66

- 5.1 O que é agilidade? 68
- 5.2 Agilidade e o custo das mudanças 68
- 5.3 O que é processo ágil? 69
 - 5.3.1 Princípios da agilidade 70
 - 5.3.2 A política do desenvolvimento ágil 71
- 5.4 Extreme programming – XP (Programação Extrema) 72
 - 5.4.1 O processo XP 72
 - 5.4.2 Industrial XP 76
- 5.5 Outros modelos de processos ágeis 77
 - 5.5.1 Scrum 78
 - 5.5.2 Método de Desenvolvimento de Sistemas Dinâmicos (DSDM) 79
 - 5.5.3 Modelagem Ágil (AM) 80
 - 5.5.4 Processo Unificado Ágil 82
- 5.6 Um conjunto de ferramentas para o processo ágil 83
- 5.7 Resumo 84
- Problemas e pontos a ponderar 85
- Leituras e fontes de informação complementares 85

CAPÍTULO 6 Aspectos humanos da engenharia de software 87

- 6.1 Características de um engenheiro de software 88
- 6.2 A psicologia da engenharia de software 89
- 6.3 A equipe de software 90
- 6.4 Estruturas de equipe 92
- 6.5 Equipes ágeis 93
 - 6.5.1 A equipe ágil genérica 93
 - 6.5.2 A equipe XP 94
- 6.6 O impacto da mídia social 95

6.7	Engenharia de software usando a nuvem	97
6.8	Ferramentas de colaboração	98
6.9	Equipes globais	99
6.10	Resumo	100
	Problemas e pontos a ponderar	101
	Leituras e fontes de informação complementares	102

PARTE II Modelagem 103

CAPÍTULO 7 Princípios que orientam a prática 104

7.1	Conhecimento da engenharia de software	105
7.2	Princípios fundamentais	106
7.2.1	Princípios que orientam o processo	106
7.2.2	Princípios que orientam a prática	107
7.3	Princípios das atividades metodológicas	109
7.3.1	Princípios da comunicação	110
7.3.2	Princípios do planejamento	112
7.3.3	Princípios da modelagem	114
7.3.4	Princípios da construção	121
7.3.5	Princípios da disponibilização	124
7.4	Formas de trabalhar	126
7.5	Resumo	127
	Problemas e pontos a ponderar	128
	Leituras e fontes de informação complementares	129

CAPÍTULO 8 Entendendo os requisitos 131

8.1	Engenharia de requisitos	132
8.2	Estabelecimento da base de trabalho	138
8.2.1	Identificação de envolvidos	139
8.2.2	Reconhecimento de diversos pontos de vista	139
8.2.3	Trabalho em busca da colaboração	140
8.2.4	Questões iniciais	140
8.2.5	Requisitos não funcionais	141
8.2.6	Rastreabilidade	142
8.3	Levantamento de requisitos	142
8.3.1	Coleta colaborativa de requisitos	143
8.3.2	Aplicação da qualidade por QFD (Quality Function Deployment)	146
8.3.3	Cenários de uso	146
8.3.4	Artefatos do levantamento de requisitos	147
8.3.5	Levantamento de requisitos ágil	148
8.3.6	Métodos orientados a serviços	148
8.4	Desenvolvimento de casos de uso	149
8.5	Construção do modelo de análise	154
8.5.1	Elementos do modelo de análise	154
8.5.2	Padrões de análise	157
8.5.3	Engenharia de requisitos ágil	158
8.5.4	Requisitos de sistemas autoadaptativos	158
8.6	Negociação de requisitos	159
8.7	Monitoramento de requisitos	160
8.8	Validação dos requisitos	161

8.9	Evite erros comuns	162
8.10	Resumo	162
	Problemas e pontos a ponderar	163
	Leituras complementares e outras fontes de informação	164

CAPÍTULO 9 Modelagem de requisitos: métodos baseados em cenários 166

9.1	Análise de requisitos	167
9.1.1	Filosofia e objetivos gerais	168
9.1.2	Regras práticas para a análise	168
9.1.3	Análise de domínio	169
9.1.4	Abordagens de modelagem de requisitos	171
9.2	Modelagem baseada em cenários	173
9.2.1	Criação de um caso de uso preliminar	173
9.2.2	Refinamento de um caso de uso preliminar	176
9.2.3	Criação de um caso de uso formal	177
9.3	Modelos UML que complementam o caso de uso	179
9.3.1	Desenvolvimento de um diagrama de atividades	179
9.3.2	Diagramas de raias	180
9.4	Resumo	182
	Problemas e pontos a ponderar	182
	Leituras e fontes de informação complementares	183

CAPÍTULO 10 Modelagem de requisitos: métodos baseados em classes 184

10.1	Identificação de classes de análise	185
10.2	Especificação de atributos	188
10.3	Definição das operações	189
10.4	Modelagem classe-responsabilidade-colaborador	192
10.5	Associações e dependências	198
10.6	Pacotes de análise	199
10.7	Resumo	200
	Problemas e pontos a ponderar	201
	Leituras e fontes de informação complementares	201

CAPÍTULO 11 Modelagem de requisitos: comportamento, padrões e WebApps/aplicativos móveis 202

11.1	Criação de um modelo comportamental	203
11.2	Identificação de eventos com o caso de uso	203
11.3	Representações de estados	204
11.4	Padrões para a modelagem de requisitos	207
11.4.1	Descoberta de padrões de análise	208
11.4.2	Exemplo de padrão de requisitos: Atuador-Sensor	209
11.5	Modelagem de requisitos para WebApps e aplicativos móveis	213
11.5.1	Que nível de análise é suficiente?	214
11.5.2	Entrada da modelagem de requisitos	214
11.5.3	Saída da modelagem de requisitos	215
11.5.4	Modelo de conteúdo	216
11.5.5	Modelo de interação para WebApps e aplicativos móveis	217
11.5.6	Modelo funcional	218
11.5.7	Modelo de configuração para WebApps	219
11.5.8	Modelo de navegação	220

11.6 Resumo 221

Problemas e pontos a ponderar 222

Leituras e fontes de informação complementares 222

CAPÍTULO 12 Conceitos de projeto 224**12.1 Projeto no contexto da engenharia de software 225****12.2 O processo de projeto 228**

12.2.1 Diretrizes e atributos da qualidade de software 228

12.2.2 A evolução de um projeto de software 230

12.3 Conceitos de projeto 231

12.3.1 Abstração 232

12.3.2 Arquitetura 232

12.3.3 Padrões 233

12.3.4 Separação por interesses (por afinidades) 234

12.3.5 Modularidade 234

12.3.6 Encapsulamento de informações 235

12.3.7 Independência funcional 236

12.3.8 Refinamento 237

12.3.9 Aspectos 237

12.3.10 Refatoração 238

12.3.11 Conceitos de projeto orientado a objetos 238

12.3.12 Classes de projeto 239

12.3.13 Inversão da dependência 241

12.3.14 Projeto para teste 242

12.4 O modelo de projeto 243

12.4.1 Elementos de projeto de dados 244

12.4.2 Elementos do projeto de arquitetura 244

12.4.3 Elementos do projeto de interface 245

12.4.4 Elementos do projeto de componentes 247

12.4.5 Elementos do projeto de implantação 247

12.5 Resumo 249

Problemas e pontos a ponderar 250

Leituras e fontes de informação complementares 250

CAPÍTULO 13 Projeto de arquitetura 252**13.1 Arquitetura de software 253**

13.1.1 O que é arquitetura? 253

13.1.2 Por que a arquitetura é importante? 254

13.1.3 Descrições de arquitetura 255

13.1.4 Decisões de arquitetura 256

13.2 Gêneros de arquitetura 257**13.3 Estilos de arquitetura 258**

13.3.1 Uma breve taxonomia dos estilos de arquitetura 258

13.3.2 Padrões de arquitetura 263

13.3.3 Organização e refinamento 263

13.4 Considerações sobre a arquitetura 264**13.5 Decisões sobre a arquitetura 266****13.6 Projeto de arquitetura 267**

13.6.1 Representação do sistema no contexto 267

13.6.2 Definição de arquétipos 269

13.6.3 Refinamento da arquitetura em componentes 270

13.6.4 Descrição das instâncias do sistema 272

13.6.5	Projeto de arquitetura para aplicações web (WebApps)	273
13.6.6	Projeto de arquitetura para aplicativos móveis	274
13.7	Avaliação das alternativas de projeto de arquitetura	274
13.7.1	Linguagens de descrição da arquitetura	276
13.7.2	Revisões da arquitetura	277
13.8	Lições aprendidas	278
13.9	Revisão de arquitetura baseada em padrões	278
13.10	Verificação de conformidade da arquitetura	279
13.11	Agilidade e arquitetura	280
13.12	Resumo	282
	Problemas e pontos a ponderar	282
	Leituras e fontes de informação complementares	283

CAPÍTULO 14 Projeto de componentes 285

14.1	O que é componente?	286
14.1.1	Uma visão orientada a objetos	286
14.1.2	A visão tradicional	288
14.1.3	Uma visão relacionada a processos	291
14.2	Projeto de componentes baseados em classes	291
14.2.1	Princípios básicos de projeto	292
14.2.2	Diretrizes para o projeto de componentes	295
14.2.3	Coesão	296
14.2.4	Acoplamento	298
14.3	Condução de projetos de componentes	299
14.4	Projeto de componentes para WebApps	305
14.4.1	Projeto de conteúdo para componentes	305
14.4.2	Projeto funcional para componentes	306
14.5	Projeto de componentes para aplicativos móveis	306
14.6	Projeto de componentes tradicionais	307
14.7	Desenvolvimento baseado em componentes	308
14.7.1	Engenharia de domínio	308
14.7.2	Qualificação, adaptação e composição de componentes	309
14.7.3	Divergência arquitetural	311
14.7.4	Análise e projeto para reutilização	312
14.7.5	Classificação e recuperação de componentes	312
14.8	Resumo	313
	Problemas e pontos a ponderar	315
	Leituras e fontes de informação complementares	316

CAPÍTULO 15 Projeto de interfaces do usuário 317

15.1	As regras de ouro	318
15.1.1	Deixar o usuário no comando	318
15.1.2	Reduzir a carga de memória do usuário	319
15.1.3	Tornar a interface consistente	321
15.2	Análise e projeto de interfaces	322
15.2.1	Modelos de análise e projeto de interfaces	322
15.2.2	O processo	323
15.3	Análise de interfaces	325
15.3.1	Análise de usuários	325
15.3.2	Análise e modelagem de tarefas	326
15.3.3	Análise do conteúdo exibido	331
15.3.4	Análise do ambiente de trabalho	331

- 15.4 Etapas no projeto de interfaces 332
 - 15.4.1 Aplicação das etapas para projeto de interfaces 332
 - 15.4.2 Padrões de projeto de interfaces do usuário 334
 - 15.4.3 Questões de projeto 335
- 15.5 Projeto de interfaces para WebApps e aplicativos móveis 337
 - 15.5.1 Princípios e diretrizes para projeto de interfaces 337
 - 15.5.2 Fluxo de trabalho de projeto de interfaces para WebApps e aplicativos móveis 341
- 15.6 Avaliação de projeto 342
- 15.7 Resumo 344

Problemas e pontos a ponderar 345

Leituras e fontes de informação complementares 346

CAPÍTULO 16 Projeto baseado em padrões 347

- 16.1 Padrões de projeto 348
 - 16.1.1 Tipos de padrões 349
 - 16.1.2 Frameworks 351
 - 16.1.3 Descrição de padrões 352
 - 16.1.4 Linguagens e repositórios de padrões 353
- 16.2 Projeto de software baseado em padrões 354
 - 16.2.1 Contexto do projeto baseado em padrões 354
 - 16.2.2 Pense em termos de padrões 354
 - 16.2.3 Tarefas de projeto 356
 - 16.2.4 Construção de uma tabela para organização de padrões 358
 - 16.2.5 Erros comuns de projeto 359
- 16.3 Padrões de arquitetura 359
- 16.4 Padrões de projeto de componentes 360
- 16.5 Padrões de projeto para interfaces do usuário 362
- 16.6 Padrões de projeto para WebApps 364
 - 16.6.1 Foco do projeto 365
 - 16.6.2 Granularidade do projeto 365
- 16.7 Padrões para aplicativos móveis 366
- 16.8 Resumo 367

Problemas e pontos a ponderar 368

Leituras e fontes de informação complementares 369

CAPÍTULO 17 Projeto de WebApps 371

- 17.1 Qualidade de projeto em WebApps 372
- 17.2 Objetivos de projeto 374
- 17.3 Uma pirâmide de projeto para WebApps 375
- 17.4 Projeto de interfaces para WebApp 376
- 17.5 Projeto estético 377
 - 17.5.1 Questões de layout 378
 - 17.5.2 Questões de design gráfico 378
- 17.6 Projeto de conteúdo 379
 - 17.6.1 Objetos de conteúdo 379
 - 17.6.2 Questões de projeto de conteúdo 380
- 17.7 Projeto de arquitetura 381
 - 17.7.1 Arquitetura de conteúdo 381
 - 17.7.2 Arquitetura de uma WebApp 384
- 17.8 Projeto da navegação 385
 - 17.8.1 Semântica de navegação 385
 - 17.8.2 Sintaxe de navegação 387

17.9	Projeto em nível de componentes	387
17.10	Resumo	388
	Problemas e pontos a ponderar	389
	Leituras e fontes de informação complementares	389

CAPÍTULO 18 Projeto de aplicativos móveis 391

18.1	Os desafios	392
18.1.1	Considerações sobre o desenvolvimento	392
18.1.2	Considerações técnicas	393
18.2	Desenvolvimento de aplicativos móveis	395
18.2.1	Qualidade do aplicativo móvel	397
18.2.2	Projeto de interface de usuário	398
18.2.3	Aplicativos sensíveis ao contexto	398
18.2.4	Lições aprendidas	400
18.3	Projeto de aplicativos móveis – boas práticas	401
18.4	Ambientes de mobilidade	403
18.5	A nuvem	405
18.6	A aplicabilidade da engenharia de software convencional	407
18.7	Resumo	408
	Problemas e pontos a ponderar	409
	Leituras e fontes de informação complementares	410

PARTE III Gestão da qualidade 411

CAPÍTULO 19 Conceitos de qualidade 412

19.1	O que é qualidade?	413
19.2	Qualidade de software	414
19.2.1	Dimensões de qualidade de Garvin	415
19.2.2	Fatores de qualidade de McCall	416
19.2.3	Fatores de qualidade ISO 9126	418
19.2.4	Fatores de qualidade desejados	418
19.2.5	A transição para uma visão quantitativa	420
19.3	O dilema da qualidade do software	420
19.3.1	Software “bom o suficiente”	421
19.3.2	Custo da qualidade	422
19.3.3	Riscos	424
19.3.4	Negligência e responsabilidade civil	425
19.3.5	Qualidade e segurança	425
19.3.6	O impacto das ações administrativas	426
19.4	Alcançando a qualidade de software	427
19.4.1	Métodos de engenharia de software	427
19.4.2	Técnicas de gerenciamento de software	427
19.4.3	Controle de qualidade	428
19.4.4	Garantia da qualidade	428
19.5	Resumo	428
	Problemas e pontos a ponderar	429
	Leituras e fontes de informação complementares	430

CAPÍTULO 20 Técnicas de revisão 431

- 20.1 Impacto de defeitos de software nos custos 432
- 20.2 Amplificação e eliminação de defeitos 433
- 20.3 Métricas de revisão e seu emprego 435
 - 20.3.1 Análise de métricas 435
 - 20.3.2 Eficácia dos custos de revisões 436
- 20.4 Revisões: um espectro de formalidade 438
- 20.5 Revisões informais 439
- 20.6 Revisões técnicas formais 441
 - 20.6.1 A reunião de revisão 441
 - 20.6.2 Relatório de revisão e manutenção de registros 442
 - 20.6.3 Diretrizes de revisão 442
 - 20.6.4 Revisões por amostragem 444
- 20.7 Avaliações post-mortem 445
- 20.8 Resumo 446
- Problemas e pontos a ponderar 446
- Leituras e fontes de informação complementares 447

CAPÍTULO 21 Garantia da qualidade de software 448

- 21.1 Plano de fundo 449
- 21.2 Elementos de garantia da qualidade de software 450
- 21.3 Processos da SQA e características do produto 452
- 21.4 Tarefas, metas e métricas da SQA 452
 - 21.4.1 Tarefas da SQA 453
 - 21.4.2 Metas, atributos e métricas 454
- 21.5 Abordagens formais da SQA 455
- 21.6 Estatística da garantia da qualidade de software 456
 - 21.6.1 Um exemplo genérico 456
 - 21.6.2 Seis Sigma para engenharia de software 458
- 21.7 Confiabilidade de software 459
 - 21.7.1 Medidas de confiabilidade e disponibilidade 459
 - 21.7.2 Segurança do software 460
- 21.8 Os padrões de qualidade ISO 9000 461
- 21.9 O plano de SQA 463
- 21.10 Resumo 463
- Problemas e pontos a ponderar 464
- Leituras e fontes de informação complementares 464

CAPÍTULO 22 Estratégias e teste de software 466

- 22.1 Uma abordagem estratégica do teste de software 466
 - 22.1.1 Verificação e validação 467
 - 22.1.2 Organizando o teste de software 468
 - 22.1.3 Estratégia de teste de software – visão global 469
 - 22.1.4 Critérios para conclusão do teste 472
- 22.2 Problemas estratégicos 472
- 22.3 Estratégias de teste para software convencional 473
 - 22.3.1 Teste de unidade 473
 - 22.3.2 Teste de integração 475
- 22.4 Estratégias de teste para software orientado a objetos 481
 - 22.4.1 Teste de unidade em contexto orientado a objetos 481
 - 22.4.2 Teste de integração em contexto orientado a objetos 481

22.5	Estratégias de teste para WebApps	482
22.6	Estratégias de teste para aplicativos móveis	483
22.7	Teste de validação	483
22.7.1	Critérios de teste de validação	484
22.7.2	Revisão da configuração	484
22.7.3	Testes alfa e beta	484
22.8	Teste de sistema	486
22.8.1	Teste de recuperação	486
22.8.2	Teste de segurança	486
22.8.3	Teste por esforço	487
22.8.4	Teste de desempenho	487
22.8.5	Teste de disponibilização	488
22.9	A arte da depuração	488
22.9.1	O processo de depuração	489
22.9.2	Considerações psicológicas	490
22.9.3	Estratégias de depuração	490
22.9.4	Correção do erro	493
22.10	Resumo	493
	Problemas e pontos a ponderar	494
	Leituras e fontes de informação complementares	494

CAPÍTULO 23 Teste de aplicativos convencionais 496

23.1	Fundamentos do teste de software	497
23.2	Visões interna e externa do teste	499
23.3	Teste caixa-branca	500
23.4	Teste do caminho básico	500
23.4.1	Notação de grafo de fluxo	500
23.4.2	Caminhos de programa independentes	502
23.4.3	Derivação de casos de teste	504
23.4.4	Matrizes de grafos	506
23.5	Teste de estrutura de controle	507
23.6	Teste caixa-preta	509
23.6.1	Métodos de teste baseados em grafos	509
23.6.2	Particionamento de equivalência	511
23.6.3	Análise de valor limite	512
23.6.4	Teste de matriz ortogonal	513
23.7	Teste baseado em modelos	516
23.8	Teste da documentação e dos recursos de ajuda	516
23.9	Teste para sistemas em tempo real	517
23.10	Padrões para teste de software	519
23.11	Resumo	520
	Problemas e pontos a ponderar	521
	Leituras e fontes de informação complementares	521

CAPÍTULO 24 Teste de aplicações orientadas a objeto 523

24.1	Ampliando a visão do teste	524
24.2	Teste de modelos de análise e de projeto orientados a objetos	525
24.2.1	Exatidão dos modelos de OOA e OOD	525
24.2.2	Consistência dos modelos orientados a objetos	526

24.3	Estratégias de teste orientado a objetos	528
24.3.1	Teste de unidade em contexto orientado a objetos	528
24.3.2	Teste de integração em contexto orientado a objetos	529
24.3.3	Teste de validação em contexto orientado a objetos	529
24.4	Métodos de teste orientados a objetos	529
24.4.1	As implicações dos conceitos de orientação a objetos no projeto de casos de teste	530
24.4.2	Aplicabilidade dos métodos convencionais de projeto de casos de teste	531
24.4.3	Teste baseado em falhas	531
24.4.4	Projeto de teste baseado em cenários	532
24.5	Métodos de teste aplicáveis no nível de classe	532
24.5.1	Teste aleatório para classes orientadas a objetos	532
24.5.2	Teste de partição em nível de classe	533
24.6	Projeto de caso de teste entre classes	534
24.6.1	Teste de múltiplas classes	534
24.6.2	Testes derivados de modelos comportamentais	536
24.7	Resumo	537
	Problemas e pontos a ponderar	538
	Leituras e fontes de informação complementares	538

CAPÍTULO 25 Teste de aplicações para Web 540

25.1	Conceitos de teste para WebApps	541
25.1.1	Dimensões da qualidade	541
25.1.2	Erros em um ambiente WebApp	542
25.1.3	Estratégia de teste	543
25.1.4	Planejamento de teste	543
25.2	O processo de teste – uma visão geral	544
25.3	Teste de conteúdo	545
25.3.1	Objetivos do teste de conteúdo	545
25.3.2	Teste de banco de dados	546
25.4	Teste da interface do usuário	549
25.4.1	Estratégia de teste de interface	549
25.4.2	Teste de mecanismos de interface	549
25.4.3	Teste da semântica da interface	551
25.4.4	Testes de usabilidade	552
25.4.5	Testes de compatibilidade	553
25.5	Teste no nível de componente	554
25.6	Testes de navegação	555
25.6.1	Teste da sintaxe de navegação	556
25.6.2	Teste da semântica de navegação	556
25.7	Teste de configuração	558
25.7.1	Tópicos no lado do servidor	558
25.7.2	Tópicos no lado do cliente	559
25.8	Teste de segurança	559
25.9	Teste de desempenho	560
25.9.1	Objetivos do teste de desempenho	561
25.9.2	Teste de carga	562
25.9.3	Teste de esforço	562
25.10	Resumo	563
	Problemas e pontos a ponderar	564
	Leituras e fontes de informação complementares	565

CAPÍTULO 26 Teste de aplicativos móveis 567

- 26.1 Diretrizes de teste 568
 - 26.2 As estratégias de teste 569
 - 26.2.1 As estratégias convencionais são adequadas? 570
 - 26.2.2 A necessidade de automação 571
 - 26.2.3 Construção de uma matriz de teste 572
 - 26.2.4 Teste de esforço (stress) 573
 - 26.2.5 Testes em um ambiente de produção 573
 - 26.3 Considerações sobre o espectro da interação do usuário 574
 - 26.3.1 Teste de gestos 575
 - 26.3.2 Entrada e reconhecimento de voz 576
 - 26.3.3 Entrada por teclado virtual 577
 - 26.3.4 Alertas e condições extraordinárias 577
 - 26.4 Teste além de fronteiras 578
 - 26.5 Problemas do teste em tempo real 578
 - 26.6 Ferramentas e ambientes de teste 579
 - 26.7 Resumo 581
- Problemas e pontos a ponderar 582
- Leituras e fontes de informação complementares 582

CAPÍTULO 27 Engenharia de segurança 584

- 27.1 Análise dos requisitos de segurança 585
 - 27.2 Segurança e privacidade em um mundo online 586
 - 27.2.1 Mídia social 587
 - 27.2.2 Aplicativos móveis 587
 - 27.2.3 Computação em nuvem 587
 - 27.2.4 A Internet das coisas 588
 - 27.3 Análise da engenharia de segurança 588
 - 27.3.1 Levantamento de requisitos de segurança 589
 - 27.3.2 Modelagem de segurança 590
 - 27.3.3 Projeto de medidas 591
 - 27.3.4 Verificações de exatidão 591
 - 27.4 Garantia da segurança 592
 - 27.4.1 O processo da garantia da segurança 592
 - 27.4.2 Organização e gerenciamento 593
 - 27.5 Análise de risco de segurança 594
 - 27.6 A função das atividades da engenharia de software convencional 595
 - 27.7 Verificação de sistemas confiáveis 597
 - 27.8 Resumo 599
- Problemas e pontos a ponderar 599
- Leituras e fontes de informação complementares 600

CAPÍTULO 28 Modelagem formal e verificação 601

- 28.1 Estratégia sala limpa 602
- 28.2 Especificação funcional 604
 - 28.2.1 Especificação caixa-preta 605
 - 28.2.2 Especificação caixa de estado 606
 - 28.2.3 Especificação caixa-clara 607
- 28.3 Projeto sala limpa 607
 - 28.3.1 Refinamento do projeto 608
 - 28.3.2 Verificação de projeto 608

- 28.4 Teste sala limpa 610
 - 28.4.1 Teste de uso estatístico 610
 - 28.4.2 Certificação 612
 - 28.5 Reconsideração dos métodos formais 612
 - 28.6 Conceitos de métodos formais 615
 - 28.7 Argumentos alternativos 618
 - 28.8 Resumo 619
- Problemas e pontos a ponderar 620
- Leituras e fontes de informação complementares 621

CAPÍTULO 29 Gestão de configuração de software 623

- 29.1 Gestão de configuração de software 624
 - 29.1.1 Um cenário SCM 625
 - 29.1.2 Elementos de um sistema de gestão de configuração 626
 - 29.1.3 Referenciais 626
 - 29.1.4 Itens de configuração de software 628
 - 29.1.5 Gestão de dependências e alterações 629
 - 29.2 O repositório de SCM 630
 - 29.2.1 Características gerais e conteúdo 630
 - 29.2.2 Características da SCM 631
 - 29.3 O processo SCM 632
 - 29.3.1 Identificação de objetos na configuração de software 633
 - 29.3.2 Controle de versão 634
 - 29.3.3 Controle de alterações 635
 - 29.3.4 Gestão de impacto 638
 - 29.3.5 Auditoria de configuração 639
 - 29.3.6 Relatório de status 639
 - 29.4 Gestão de configuração para WebApps e aplicativos móveis 640
 - 29.4.1 Problemas predominantes 641
 - 29.4.2 Objetos de configuração 642
 - 29.4.3 Gestão de conteúdo 643
 - 29.4.4 Gestão de alterações 646
 - 29.4.5 Controle de versão 648
 - 29.4.6 Auditoria e relatório 649
 - 29.5 Resumo 650
- Problemas e pontos a ponderar 651
- Leituras e fontes de informação complementares 651

CAPÍTULO 30 Métricas de produto 653

- 30.1 Framework para métricas de produto 654
 - 30.1.1 Medidas, métricas e indicadores 654
 - 30.1.2 O desafio das métricas de produto 655
 - 30.1.3 Princípios da medição 656
 - 30.1.4 Medição de software orientada a metas 656
 - 30.1.5 Atributos de métricas de software eficazes 657
- 30.2 Métricas para o modelo de requisitos 659
 - 30.2.1 Métricas baseadas em função 659
 - 30.2.2 Métricas para qualidade de especificação 662
- 30.3 Métricas para o modelo de projeto 663
 - 30.3.1 Métricas de projeto de arquitetura 663
 - 30.3.2 Métricas para projeto orientado a objetos 666
 - 30.3.3 Métricas orientadas a classes – o conjunto de métricas CK 667

30.3.4	Métricas orientadas a classes – o conjunto de métricas MOOD	670
30.3.5	Métricas orientadas a objeto propostas por Lorenz e Kidd	671
30.3.6	Métricas de projeto em nível de componente	671
30.3.7	Métricas orientadas a operação	671
30.3.8	Métricas de projeto de interface de usuário	672
30.4	Métricas de projeto para WebApps e aplicativos móveis	672
30.5	Métricas para código-fonte	675
30.6	Métricas para teste	676
30.6.1	Métricas de Halstead aplicadas ao teste	676
30.6.2	Métricas para teste orientado a objetos	677
30.7	Métricas para manutenção	678
30.8	Resumo	679
	Problemas e pontos a ponderar	680
	Leituras e fontes de informação complementares	680

PARTE IV Gerenciamento de projetos de software 683

CAPÍTULO 31 Conceitos de gerenciamento de projeto 684

31.1	O espectro de gerenciamento	685
31.1.1	As pessoas	685
31.1.2	O produto	686
31.1.3	O processo	686
31.1.4	O projeto	686
31.2	As pessoas	687
31.2.1	Os envolvidos	687
31.2.2	Líderes de equipe	688
31.2.3	A equipe de software	689
31.2.4	Equipes ágeis	691
31.2.5	Questões de comunicação e coordenação	693
31.3	O produto	693
31.3.1	Escopo do software	693
31.3.2	Decomposição do problema	694
31.4	O processo	694
31.4.1	Combinando o produto e o processo	695
31.4.2	Decomposição do processo	695
31.5	O projeto	697
31.6	O princípio W ⁵ HH	698
31.7	Práticas vitais	699
31.8	Resumo	699

Problemas e pontos a ponderar 700

Leituras e fontes de informação complementares 701

CAPÍTULO 32 Métricas de processo e de projeto 703

32.1	Métricas no domínio do processo e do projeto	704
32.1.1	Métricas de processo e aperfeiçoamento do processo de software	704
32.1.2	Métricas de projeto	707
32.2	Medição de software	708
32.2.1	Métricas orientadas a tamanho	709
32.2.2	Métricas orientadas a função	710
32.2.3	Harmonizando métricas LOC e FP	711

32.2.4	Métricas orientadas a objetos	713
32.2.5	Métricas orientadas a casos de uso	714
32.2.6	Métricas de projeto de WebApp	714
32.3	Métricas para qualidade de software	716
32.3.1	Medição da qualidade	717
32.3.2	Eficiência na remoção de defeitos	718
32.4	Integração de métricas dentro do processo de software	719
32.4.1	Argumentos favoráveis às métricas de software	720
32.4.2	Estabelecimento de um referencial	720
32.4.3	Coleta, cálculo e avaliação de métricas	721
32.5	Métricas para empresas pequenas	721
32.6	Estabelecimento de um programa de métricas de software	722
32.7	Resumo	724
Problemas e pontos a ponderar		724
Leituras e fontes de informação complementares		725

CAPÍTULO 33 Estimativas de projeto de software 727

33.1	Observações sobre as estimativas	728
33.2	O processo de planejamento do projeto	729
33.3	Escopo e viabilidade do software	730
33.4	Recursos	731
33.4.1	Recursos humanos	731
33.4.2	Recursos de software reutilizáveis	732
33.4.3	Recursos ambientais	732
33.5	Estimativa do projeto de software	733
33.6	Técnicas de decomposição	734
33.6.1	Dimensionamento do software	734
33.6.2	Estimativa baseada em problema	735
33.6.3	Um exemplo de estimativa baseada em LOC	736
33.6.4	Um exemplo de estimativa baseada em FP	738
33.6.5	Estimativa baseada em processo	739
33.6.6	Um exemplo de estimativa baseada em processo	740
33.6.7	Estimativa com casos de uso	740
33.6.8	Um exemplo de estimativa usando pontos de caso de uso	742
33.6.9	Harmonizando estimativas	742
33.7	Modelos empíricos de estimativa	743
33.7.1	A estrutura dos modelos de estimativa	744
33.7.2	O modelo COCOMO II	744
33.7.3	A equação do software	744
33.8	Estimativa para projetos orientados a objetos	746
33.9	Técnicas de estimativa especializadas	746
33.9.1	Estimativa para desenvolvimento ágil	746
33.9.2	Estimativa para projetos de WebApps	747
33.10	A decisão fazer/comprar	748
33.10.1	Criação de uma árvore de decisões	749
33.10.2	Terceirização	750
33.11	Resumo	752
Problemas e pontos a ponderar		752
Leituras e fontes de informação complementares		753

CAPÍTULO 34 Cronograma de projeto 754

- 34.1 Conceitos básicos 755
- 34.2 Cronograma de projeto 757
 - 34.2.1 Princípios básicos 758
 - 34.2.2 Relação entre pessoas e esforço 759
 - 34.2.3 Distribuição de esforço 760
- 34.3 Definição de um conjunto de tarefas para o projeto de software 761
 - 34.3.1 Um exemplo de conjunto de tarefas 762
 - 34.3.2 Refinamento das tarefas principais 763
- 34.4 Definição de uma rede de tarefas 764
- 34.5 Cronograma 765
 - 34.5.1 Gráfico de Gantt 766
 - 34.5.2 Acompanhamento do cronograma 767
 - 34.5.3 Acompanhamento do progresso de um projeto orientado a objetos 768
 - 34.5.4 Cronograma para projetos de WebApps e aplicativos móveis 769
- 34.6 Análise de valor agregado 772
- 34.7 Resumo 774
- Problemas e pontos a ponderar 774
- Leituras e fontes de informação complementares 776

CAPÍTULO 35 Gestão de riscos 777

- 35.1 Estratégias de risco reativas *versus* proativas 778
- 35.2 Riscos de software 778
- 35.3 Identificação do risco 780
 - 35.3.1 Avaliação do risco geral do projeto 781
 - 35.3.2 Componentes e fatores de risco 782
- 35.4 Previsão de risco 782
 - 35.4.1 Desenvolvimento de uma tabela de riscos 783
 - 35.4.2 Avaliação do impacto do risco 785
- 35.5 Refinamento do risco 787
- 35.6 Mitigação, monitoramento e gestão de riscos (RMMM) 788
- 35.7 O plano RMMM 790
- 35.8 Resumo 792
- Problemas e pontos a ponderar 792
- Leituras e fontes de informação complementares 793

CAPÍTULO 36 Manutenção e reengenharia 795

- 36.1 Manutenção de software 796
- 36.2 Suportabilidade do software 798
- 36.3 Reengenharia 798
- 36.4 Reengenharia de processo de negócio 799
 - 36.4.1 Processos de negócio 799
 - 36.4.2 Um modelo de BPR 800
- 36.5 Reengenharia de software 802
 - 36.5.1 Um modelo de processo de reengenharia de software 802
 - 36.5.2 Atividades de reengenharia de software 803
- 36.6 Engenharia reversa 805
 - 36.6.1 Engenharia reversa para entender os dados 807
 - 36.6.2 Engenharia reversa para entender o processamento 807
 - 36.6.3 Engenharia reversa das interfaces de usuário 808

36.7	Reestruturação	809
36.7.1	Reestruturação de código	809
36.7.2	Reestruturação de dados	810
36.8	Engenharia direta	811
36.8.1	Engenharia direta para arquiteturas cliente-servidor	812
36.8.2	Engenharia direta para arquiteturas orientadas a objetos	813
36.9	Aspectos econômicos da reengenharia	813
36.10	Resumo	814
Problemas e pontos a ponderar		815
Leituras e fontes de informação complementares		816

PARTE V Tópicos avançados 817

CAPÍTULO 37 Melhoria do processo de software 818

37.1	O que é SPI?	819
37.1.1	Abordagens para SPI	819
37.1.2	Modelos de maturidade	821
37.1.3	A SPI é para todos?	822
37.2	O processo de SPI	823
37.2.1	Avaliação e análise de lacunas	823
37.2.2	Educação e treinamento	825
37.2.3	Seleção e justificação	825
37.2.4	Instalação/migração	826
37.2.5	Mensuração	827
37.2.6	Gestão de riscos para SPI	827
37.3	O CMMI	828
37.4	People-CMM	832
37.5	Outros frameworks SPI	832
37.6	Retorno sobre investimento em SPI	834
37.7	Tendências da SPI	835
37.8	Resumo	836
Problemas e pontos a ponderar		837
Leituras e fontes de informação complementares		837

CAPÍTULO 38 Tendências emergentes na engenharia de software 839

38.1	Evolução da tecnologia	840
38.2	Perspectivas para uma verdadeira disciplina de engenharia	841
38.3	Observação de tendências na engenharia de software	842
38.4	Identificação das “tendências leves”	843
38.4.1	Gestão da complexidade	845
38.4.2	Software aberto	846
38.4.3	Requisitos emergentes	846
38.4.4	O mix de talentos	847
38.4.5	Blocos básicos de software	847
38.4.6	Mudança na percepção de “valor”	848
38.4.7	Código aberto	848
38.5	Rumos da tecnologia	849
38.5.1	Tendências de processo	849
38.5.2	O grande desafio	851
38.5.3	Desenvolvimento colaborativo	852
38.5.4	Engenharia de requisitos	852

38.5.5	Desenvolvimento de software controlado por modelo	853
38.5.6	Projeto pós-moderno	854
38.5.7	Desenvolvimento guiado por teste	854
38.6	Tendências relacionadas a ferramentas	855
38.7	Resumo	857
	Problemas e pontos a ponderar	857
	Leituras e fontes de informação complementares	858

CAPÍTULO 39 Comentários finais 860

39.1	A importância do software – revisitada	861
39.2	Pessoas e a maneira como constroem sistemas	861
39.3	Novos modos de representar a informação	863
39.4	A visão no longo prazo	864
39.5	A responsabilidade do engenheiro de software	865
39.6	Comentário final de RSP	867

APÊNDICE 1 Introdução à UML 869

APÊNDICE 2 Conceitos de orientação a objetos 891

APÊNDICE 3 Métodos formais 899

REFERÊNCIAS 909

ÍNDICE 933

A natureza do software

Depois de me mostrar a construção mais recente de um dos games de tiro em primeira pessoa mais populares do mundo, o jovem desenvolvedor riu. “Você não joga, né?”, ele perguntou.

Eu sorri. “Como adivinhou?”

O jovem estava de bermuda e camiseta. Sua perna balançava para cima e para baixo como um pistão, queimando a tensa energia que parecia ser comum entre seus colegas.

“Porque, se jogasse”, ele disse, “estaria muito mais empolgado. Você acabou de ver nosso mais novo produto, algo que nossos clientes matariam para ver... sem trocadilhos”.

Estávamos na área de desenvolvimento de uma das empresas de games mais bem-sucedidas do planeta. Ao longo dos anos, as gerações anteriores do game que ele demonstrou venderam mais de 50 milhões de cópias e geraram uma receita de bilhões de dólares.

“Então, quando essa versão estará no mercado?”, perguntei.

Ele encolheu os ombros. “Em cerca de cinco meses. Ainda temos muito trabalho a fazer”. Ele era responsável pela jogabilidade e pela funcionalidade de inteligência artificial de um aplicativo que abrangia mais de três milhões de linhas de código.

“Vocês usam técnicas de engenharia de software?”, perguntei, meio que esperando sua risada e sua resposta negativa.

Conceitos-chave

aplicativos móveis	9
campos de aplicação	6
computação em nuvem	10
curvas de defeitos	5
deterioração	5
linha de produtos	11
software, definição de	4
software legado	7
software, natureza do	3
software, perguntas sobre	4
WebApps	9

PANORAMA

O que é? Software de computador é o produto que profissionais de software desenvolvem e ao qual dão suporte no longo prazo. Abrange programas executáveis em um computador de qualquer porte ou arquitetura, conteúdos (apresentados à medida que os programas são executados), informações descritivas tanto na forma impressa (*hard copy*) quanto na virtual, abrangendo praticamente qualquer mídia eletrônica.

Quem realiza? Os engenheiros de software criam e dão suporte a ele, e praticamente todos que têm contato com o mundo industrializado o utilizam, direta ou indiretamente.

Por que é importante? Porque afeta quase todos os aspectos de nossa vida e se difundiu no comércio, na cultura e em nossas atividades cotidianas.

Quais são as etapas envolvidas? Os clientes e outros envolvidos expressam a necessidade pelo software de computador, os engenheiros constroem o produto de software e os usuários o utilizam para resolver um problema específico ou para tratar de uma necessidade específica.

Qual é o artefato? Um programa de computador que funciona em um ou mais ambientes específicos e atende às necessidades de um ou mais usuários.

Como garantir que o trabalho foi realizado corretamente? Se você é engenheiro de software, aplique as ideias contidas no restante deste livro. Se for usuário, conheça sua necessidade e seu ambiente e escolha uma aplicação que seja a mais adequada a ambos.

Ele fez uma pausa e pensou por uns instantes. Então, lentamente, fez que sim com a cabeça. “Adaptamos às nossas necessidades, mas, claro, usamos”.

“Onde?”, perguntei, sondando. “Geralmente, nosso problema é traduzir os requisitos que os criativos nos dão”. “Os criativos?”, interrompi. “Você sabe, os caras que projetam a história, os personagens, todas as coisas que tornam o jogo um sucesso. Temos de pegar o que eles nos dão e produzir um conjunto de requisitos técnicos que nos permita construir o jogo.”

“E depois os requisitos são fixados?”

Ele encolheu os ombros. “Precisamos ampliar e adaptar a arquitetura da versão anterior do jogo e criar um novo produto. Temos de criar código a partir dos requisitos, testá-lo com construções diárias e fazer muitas coisas que seu livro recomenda.”

“Conhece meu livro?” Eu estava sinceramente surpreso. “Claro, usei na faculdade. Há muita coisa lá.”

“Falei com alguns de seus colegas aqui e eles são mais céticos a respeito do material de meu livro.”

Ele franziu as sobrancelhas. “Olha, não somos um departamento de TI nem uma empresa aeroespacial, então, temos de adaptar o que você defende. Mas o resultado final é o mesmo – precisamos criar um produto de alta qualidade, e o único jeito de conseguirmos isso sempre é adaptar nosso próprio subconjunto de técnicas de engenharia de software.”

“E como seu subconjunto mudará com o passar dos anos?”

Ele fez uma pausa como se estivesse pensando no futuro. “Os games vão se tornar maiores e mais complexos, com certeza. E nossos cronogramas de desenvolvimento vão ser mais apertados, à medida que a concorrência surgir. Lentamente, os próprios jogos nos obrigarão a aplicar um pouco mais de disciplina de desenvolvimento. Se não fizermos isso, estaremos mortos.”

“Ideias e descobertas tecnológicas são os mecanismos que impulsionam o crescimento econômico.”

Wall Street Journal

Software de computador continua a ser a tecnologia mais importante no cenário mundial. E é também um ótimo exemplo da lei das consequências não intencionais. Há 60 anos, ninguém poderia prever que o software se tornaria uma tecnologia indispensável para negócios, ciência e engenharia; que software viabilizaria a criação de novas tecnologias (por exemplo, engenharia genética e nanotecnologia), a extensão de tecnologias existentes (por exemplo, telecomunicações) e a mudança radical nas tecnologias mais antigas (por exemplo, a mídia); que software se tornaria a força motriz por trás da revolução do computador pessoal; que aplicativos de software seriam comprados pelos consumidores com seus smartphones; que o software evoluiria lentamente de produto para serviço, à medida que empresas de software “sob encomenda” oferecessem funcionalidade imediata (*just-in-time*), via um navegador Web; que uma empresa de software se tornaria maior e mais influente do que todas as empresas da era industrial; que uma vasta rede comandada por software evoluiria e modificaria tudo: de pesquisa em bibliotecas a compras feitas pelos consumidores, de discursos políticos a comportamentos de namoro entre jovens e adultos não tão jovens.

Ninguém poderia prever que o software seria incorporado a sistemas de todas as áreas: transportes, medicina, telecomunicações, militar, industrial, entretenimento, máquinas de escritório... a lista é quase infindável. E se você

acredita na lei das consequências não intencionais, há muitos efeitos que ainda não somos capazes de prever.

Também ninguém poderia prever que milhões de programas de computador teriam de ser corrigidos, adaptados e ampliados à medida que o tempo passasse. A realização dessas atividades de “manutenção” absorve mais pessoas e recursos do que todo o esforço aplicado na criação de um novo software.

À medida que aumenta a importância do software, a comunidade da área tenta criar tecnologias que tornem mais fácil, mais rápido e mais barato desenvolver e manter programas de computador de alta qualidade. Algumas dessas tecnologias são direcionadas a um campo de aplicação específico (por exemplo, projeto e implementação de sites); outras são focadas em um campo de tecnologia (por exemplo, sistemas orientados a objetos ou programação orientada a aspectos); e outras ainda são de bases amplas (por exemplo, sistemas operacionais como o Linux). Entretanto, nós ainda temos de desenvolver uma tecnologia de software que faça tudo isso – e a probabilidade de surgir tal tecnologia no futuro é pequena. Ainda assim, as pessoas apostam seus empregos, seu conforto, sua segurança, seu entretenimento, suas decisões e suas próprias vidas em software. Tomara que estejam certas.

Este livro apresenta uma estrutura que pode ser utilizada por aqueles que desenvolvem software – pessoas que devem fazê-lo corretamente. A estrutura abrange um processo, um conjunto de métodos e uma gama de ferramentas que chamaremos de *engenharia de software*.

1.1 A natureza do software

Hoje, o software tem um duplo papel. Ele é um produto e, ao mesmo tempo, o veículo para distribuir um produto. Como produto, fornece o potencial computacional representado pelo hardware ou, de forma mais abrangente, por uma rede de computadores que podem ser acessados por hardware local. Seja residindo em um celular, seja em um tablet, em um computador de mesa ou em um mainframe, o software é um transformador de informações – produzindo, gerenciando, adquirindo, modificando, exibindo ou transmitindo informações que podem ser tão simples quanto um único bit ou tão complexas quanto uma apresentação multimídia derivada de dados obtidos de dezenas de fontes independentes. Como veículo de distribuição do produto, o software atua como a base para o controle do computador (sistemas operacionais), a comunicação de informações (redes) e a criação e o controle de outros programas (ferramentas de software e ambientes).

O software distribui o produto mais importante de nossa era – a *informação*. Ele transforma dados pessoais (por exemplo, transações financeiras de um indivíduo) de modo que possam ser mais úteis em determinado contexto; gerencia informações comerciais para aumentar a competitividade; fornece um portal para redes mundiais de informação (Internet) e os meios para obter informações sob todas as suas formas. Também propicia um veículo que pode ameaçar a privacidade pessoal e é uma porta que permite a pessoas mal-intencionadas cometer crimes.

Software é tanto um produto quanto um veículo que distribui um produto.

"Software é um lugar onde sonhos são plantados e pesadelos são colhidos, um pântano abstrato e místico onde demônios terríveis competem com mágicas panaceias, um mundo de lobisomens e balas de prata."

Brad J. Cox

O papel do software passou por uma mudança significativa no decorrer da metade final do século passado. Aperfeiçoamentos significativos no desempenho do hardware, mudanças profundas nas arquiteturas computacionais, um vasto aumento na capacidade de memória e armazenamento e uma ampla variedade exótica de opções de entrada e saída; tudo isso resultou em sistemas computacionais mais sofisticados e complexos. Sofisticação e complexidade podem produzir resultados impressionantes quando um sistema é bem-sucedido; porém, também podem trazer enormes problemas para aqueles que precisam desenvolver e projetar sistemas robustos.

Atualmente, uma enorme indústria de software tornou-se fator dominante nas economias do mundo industrializado. Equipes de especialistas em software, cada qual concentrando-se numa parte da tecnologia necessária para distribuir uma aplicação complexa, substituíram o programador solitário de antigamente. Ainda assim, as questões levantadas por esse programador solitário continuam as mesmas hoje, quando os modernos sistemas computacionais são desenvolvidos:¹

- Por que a conclusão de um software leva tanto tempo?
- Por que os custos de desenvolvimento são tão altos?
- Por que não conseguimos encontrar todos os erros antes de entregarmos o software aos clientes?
- Por que gastamos tanto tempo e esforço realizando a manutenção de programas existentes?
- Por que ainda temos dificuldades de medir o progresso de desenvolvimento e a manutenção de um software?

Essas e muitas outras questões demonstram a preocupação com o software e a maneira como é desenvolvido – uma preocupação que tem levado à adoção da prática da engenharia de software.

1.1.1 Definição de software

Hoje, a maior parte dos profissionais e muitos outros integrantes do público em geral acham que entendem de software. Mas será que entendem mesmo?

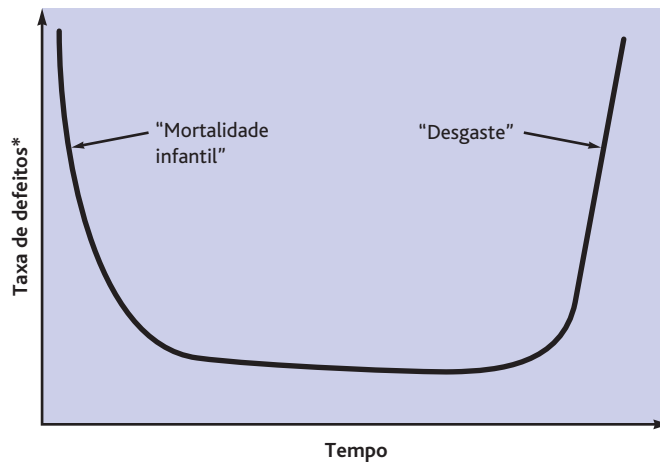
Uma descrição de software em um livro-texto poderia ser a seguinte:

Software consiste em: (1) instruções (programas de computador) que, quando executadas, fornecem características, funções e desempenho desejados; (2) estruturas de dados que possibilitam aos programas manipular informações adequadamente; e (3) informação descritiva, tanto na forma impressa quanto na virtual, descrevendo a operação e o uso dos programas.

Sem dúvida, poderíamos dar outras definições mais completas, mas, provavelmente, uma definição mais formal não melhoraria, consideravelmente, a compreensão do que é software.

Como devemos definir software?

¹ Em um excelente livro de ensaio sobre o setor de software, Tom DeMarco [DeM95] contesta. Segundo ele: "Em vez de perguntar por que software custa tanto, precisamos começar perguntando: 'O que fizemos para que o software atual custe tão pouco?'" A resposta a essa pergunta nos ajudará a continuar com o extraordinário nível de realização que tem distinguido a indústria de software".



Caso queira reduzir a deterioração do software, terá de fazer um projeto de software melhor (Capítulos 12 a 18).

FIGURA 1.1 Curva de defeitos para hardware.

Para conseguir isso, é importante examinar as características do software que o tornam diferenciado de outras coisas que os seres humanos constroem. Software é mais um elemento de sistema lógico do que físico. Portanto, o software tem uma característica fundamental que o torna consideravelmente diferente do hardware: *software não “se desgasta”*.

A Figura 1.1 representa a taxa de defeitos em função do tempo para hardware. Essa relação, normalmente denominada “curva da banheira”, indica que o hardware apresenta taxas de defeitos relativamente altas no início de sua vida (geralmente, atribuídas a defeitos de projeto ou de fabricação); os defeitos são corrigidos; e a taxa cai para um nível estável (felizmente, bastante baixo) por certo período. Entretanto, à medida que o tempo passa, a taxa aumenta novamente, conforme os componentes de hardware sofrem os efeitos cumulativos de poeira, vibração, impactos, temperaturas extremas e vários outros fatores maléficos do ambiente. Resumindo, o hardware começa a *se desgastar*.

Software não é suscetível aos fatores maléficos do ambiente que fazem com que o hardware se desgaste. Portanto, teoricamente, a curva da taxa de defeitos para software deveria assumir a forma da “curva idealizada”, mostrada na Figura 1.2. Defeitos ainda não descobertos irão resultar em altas taxas logo no início da vida de um programa. Entretanto, esses serão corrigidos, e a curva se achata, como mostrado. A curva idealizada é uma simplificação grosseira de modelos de defeitos reais para software. Porém, a implicação é clara: software não se desgasta. Mas *deteriora!*

Essa aparente contradição pode ser elucidada pela curva real apresentada na Figura 1.2. Durante sua vida², o software passará por alterações. À

* N. de R.T.: Os defeitos do software nem sempre se manifestam como falha, geralmente devido a tratamentos dos erros decorrentes desses defeitos pelo software. Esses conceitos serão mais detalhados e diferenciados nos capítulos sobre qualidade. Neste ponto, optou-se por traduzir *failure rate* por taxa de defeitos, sem prejuízo para a assimilação dos conceitos apresentados pelo autor neste capítulo.

² De fato, desde o momento em que o desenvolvimento começa, e muito antes de a primeira versão ser entregue, podem ser solicitadas mudanças por uma variedade de diferentes envolvidos.

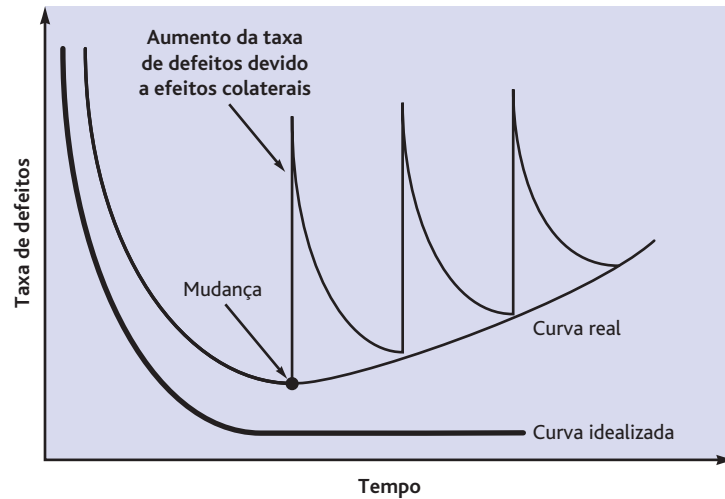


FIGURA 1.2 Curva de defeitos para software.

Os métodos de engenharia de software tentam reduzir ao máximo a magnitude das elevações (picos) e a inclinação da curva real da Figura 1.2.

medida que elas ocorram, é provável que sejam introduzidos erros, fazendo com que a curva de taxa de defeitos se acentue, conforme mostrado na “curva real” (Figura 1.2). Antes que a curva possa retornar à taxa estável original, outra alteração é requisitada, fazendo com que a curva se acentue novamente. Lentamente, o nível mínimo da taxa começa a aumentar – o software está deteriorando devido à modificação.

Outro aspecto do desgaste ilustra a diferença entre hardware e software. Quando um componente de hardware se desgasta, ele é substituído por uma peça de reposição. Não existem peças de reposição de software. Cada defeito de software indica um erro no projeto ou no processo pelo qual o projeto foi traduzido em código de máquina executável. Portanto, as tarefas de manutenção de software, que envolvem solicitações de mudanças, implicam complexidade consideravelmente maior do que a de manutenção de hardware.

1.1.2 Campos de aplicação de software

Atualmente, sete grandes categorias de software apresentam desafios contínuos para os engenheiros de software:

Software de sistema Conjunto de programas feito para atender a outros programas. Certos softwares de sistema (por exemplo, compiladores, editores e utilitários para gerenciamento de arquivos) processam estruturas de informação complexas; porém, determinadas.³ Outras aplicações de sistema (por exemplo, componentes de sistema operacional, drivers, software de rede, processadores de telecomunicações) processam dados amplamente indeterminados.

³ Um software é *determinado* se a ordem e o *timing* (periodicidade, frequência, medidas de tempo) de entradas, processamento e saídas forem previsíveis. É *indeterminado* se a ordem e o *timing* de entradas, processamento e saídas não puderem ser previstos antecipadamente.

Software de aplicação Programas independentes que solucionam uma necessidade específica de negócio. Aplicações nessa área processam dados comerciais ou técnicos de uma forma que facilite operações comerciais ou tomadas de decisão administrativas/técnicas.

Software de engenharia/científico Uma ampla variedade de programas de “cálculo em massa” que abrangem astronomia, vulcanologia, análise de estresse automotivo, dinâmica orbital, projeto auxiliado por computador, biologia molecular, análise genética e meteorologia, entre outros.

Software embarcado Residente num produto ou sistema e utilizado para implementar e controlar características e funções para o usuário e para o próprio sistema. Executa funções limitadas e específicas (por exemplo, controle do painel de um forno micro-ondas) ou fornece função significativa e capacidade de controle (por exemplo, funções digitais de automóveis, tal como controle do nível de combustível, painéis de controle e sistemas de freio).

Software para linha de produtos Projetado para prover capacidade específica de utilização por muitos clientes diferentes. Software para linha de produtos pode se concentrar em um mercado hermético e limitado (por exemplo, produtos de controle de inventário) ou lidar com consumidor de massa.

Aplicações Web/aplicativos móveis Esta categoria de software voltada às redes abrange uma ampla variedade de aplicações, contemplando aplicativos voltados para navegadores e software residente em dispositivos móveis.

Software de inteligência artificial Faz uso de algoritmos não numéricos para solucionar problemas complexos que não são passíveis de computação ou de análise direta. Aplicações nessa área incluem: robótica, sistemas especialistas, reconhecimento de padrões (de imagem e de voz), redes neurais artificiais, prova de teoremas e jogos.

Milhões de engenheiros de software em todo o mundo trabalham arduamente em projetos de software em uma ou mais dessas categorias. Em alguns casos, novos sistemas estão sendo construídos, mas, em muitos outros, aplicações já existentes estão sendo corrigidas, adaptadas e aperfeiçoadas. Não é incomum um jovem engenheiro de software trabalhar em um programa mais velho do que ele! Gerações passadas de pessoal de software deixaram um legado em cada uma das categorias discutidas. Espera-se que o legado a ser deixado por esta geração facilite o trabalho dos futuros engenheiros de software.

Uma das mais abrangentes bibliotecas de shareware/freeware (software compartilhado/livre) pode ser encontrada em shareware.cnet.com.

“Para mim, o computador é a ferramenta mais extraordinária que já inventamos. É o equivalente de uma bicicleta para nossas mentes.”

Steve Jobs

1.1.3 Software legado

Centenas de milhares de programas de computador caem em um dos sete amplos campos de aplicação discutidos na subseção anterior. Alguns deles são software de ponta – recém-lançados para indivíduos, indústria e governo. Outros programas são mais antigos – em alguns casos *muito mais* antigos.

Esses programas mais antigos – frequentemente denominados *software legado* – têm sido foco de contínua atenção e preocupação desde os anos 1960. Dayani-Fard e seus colegas [Day99] descrevem software legado da seguinte maneira:

Sistemas de software legado... foram desenvolvidos décadas atrás e têm sido continuamente modificados para se adequar às mudanças dos requisitos de negócio e a plataformas computacionais. A proliferação de tais sistemas está causando dores de cabeça para grandes organizações que os consideram dispendiosos de manter e arriscados de evoluir.

Liu e seus colegas [Liu98] ampliam essa descrição, observando que “muitos sistemas legados permanecem dando suporte para funções de negócio vitais e são ‘indispensáveis’ para o mesmo”. Por isso, um software legado é caracterizado pela longevidade e criticidade de negócios.

O que faço caso encontre um sistema legado de baixa qualidade?

Infelizmente, de vez em quando uma característica adicional está presente em software legado – a *baixa qualidade*.⁴ Às vezes, os sistemas legados têm projetos inextensíveis, código de difícil entendimento, documentação deficiente ou inexistente, casos de teste e resultados que nunca foram documentados, um histórico de alterações mal gerenciado – a lista pode ser bastante longa. Ainda assim, esses sistemas dão suporte a “funções vitais de negócio e são indispensáveis para ele”. O que fazer?

A única resposta adequada talvez seja: *não faça nada*, pelo menos até que o sistema legado tenha que passar por alguma modificação significativa. Se o software legado atende às necessidades de seus usuários e funciona de forma confiável, ele não está “quebrado” e não precisa ser “consertado”. Entretanto, com o passar do tempo, esses sistemas evoluem devido a uma ou mais das razões a seguir:

Que tipos de mudanças são feitas em sistemas legados?

- O software deve ser adaptado para atender às necessidades de novos ambientes ou de novas tecnologias computacionais.
- O software deve ser aperfeiçoado para implementar novos requisitos de negócio.
- O software deve ser expandido para torná-lo capaz de funcionar com outros bancos de dados ou com sistemas mais modernos.
- O software deve ser rearquitetado para torná-lo viável dentro de um ambiente computacional em evolução.

Todo engenheiro de software deve reconhecer que modificações são naturais. Não tente combatê-las.

Quando essas modalidades de evolução ocorrem, um sistema legado deve passar por reengenharia (Capítulo 36) para que permaneça viável no futuro. O objetivo da engenharia de software moderna é “elaborar metodologias baseadas na noção de evolução”; isto é, na noção de que os sistemas de software modificam-se continuamente, novos sistemas são construídos a partir dos antigos e... todos devem agir em grupo e cooperar uns com os outros” [Day99].

⁴ Nesse caso, a qualidade é julgada em termos da engenharia de software moderna – um critério um tanto injusto, já que alguns conceitos e princípios da engenharia de software moderna talvez não tenham sido bem entendidos na época em que o software legado foi desenvolvido.

1.2 A natureza mutante do software

A evolução de quatro categorias amplas de software domina o setor. Ainda assim, há pouco mais de uma década essas categorias estavam em sua infância.

1.2.1 WebApps

Nos primórdios da World Wide Web (por volta de 1990 a 1995), os *sites* eram formados por nada mais do que um conjunto de arquivos de hipertexto linkados e que apresentavam informações usando texto e gráficos limitados. Com o tempo, o crescimento da linguagem HTML, via ferramentas de desenvolvimento (por exemplo, XML, Java), tornou possível aos engenheiros da Internet oferecerem capacidade computacional juntamente com as informações. Nasceram, então, os *sistemas e aplicações baseados na Web*⁵ (refiro-me a eles coletivamente como *WebApps*).

Atualmente, as WebApps evoluíram para sofisticadas ferramentas computacionais que não apenas oferecem funções especializadas (*stand-alone functions*) ao usuário, como também foram integradas aos bancos de dados corporativos e às aplicações de negócio.

Há uma década, as WebApps “envolvem(iam) uma mistura de publicação impressa e desenvolvimento de software, de marketing e computação, de comunicações internas e relações externas e de arte e tecnologia” [Pow98]. Mas, hoje, fornecem potencial de computação total em muitas das categorias de aplicação registradas na seção 1.1.2.

No decorrer da década passada, tecnologias Semantic Web (muitas vezes denominadas Web 3.0) evoluíram para sofisticadas aplicações corporativas e para o consumidor, as quais abrangem “bancos de dados semânticos [que] oferecem novas funcionalidades que exigem links Web, representação [de dados] flexível e APIs de acesso externo” [Hen10]. Sofisticadas estruturas de dados relacionais levarão a WebApps inteiramente novas que permitirão acessar informações díspares de maneiras inéditas.

“Quando notarmos qualquer tipo de estabilidade, a Web terá se transformado em algo completamente diferente.”

Louis Monier

1.2.2 Aplicativos móveis

O termo *aplicativo* evoluiu para sugerir software projetado especificamente para residir em uma plataforma móvel (por exemplo, iOS, Android ou Windows Mobile). Na maioria dos casos, os aplicativos móveis contêm uma interface de usuário que tira proveito de mecanismos de interação exclusivos fornecidos pela plataforma móvel, da interoperabilidade com recursos baseados na Web que dão acesso a uma grande variedade de informações relevantes ao aplicativo e de capacidades de processamento local que coletam, analisam e formatam as informações de forma mais conveniente para a plataforma. Além disso, um aplicativo móvel fornece recursos de armazenamento persistente dentro da plataforma.

⁵ No contexto deste livro, o termo *aplicação Web (WebApp)* engloba tudo, de uma simples página Web que possa ajudar um cliente a processar o pagamento do aluguel de um automóvel a um amplo site que forneça serviços de viagem completos para executivos e turistas. Dentro dessa categoria estão sites completos, funcionalidade especializada dentro de sites e aplicações para processamento de informações residentes na Internet ou em uma intranet ou extranet.



FIGURA 1.3 Arquitetura lógica da computação em nuvem [Wik13].

Qual a diferença entre uma WebApp e um aplicativo móvel?

É importante reconhecer que existe uma diferença sutil entre aplicações web móveis e aplicativos móveis. Uma *aplicação web móvel* (WebApp) permite que um dispositivo móvel tenha acesso a conteúdo baseado na web por meio de um navegador especificamente projetado para se adaptar aos pontos fortes e fracos da plataforma móvel. Um *aplicativo móvel* pode acessar diretamente as características do hardware do dispositivo (por exemplo, acelerômetro ou localização por GPS) e, então, fornecer os recursos de processamento e armazenamento local mencionados anteriormente. Com o passar do tempo, essa diferença entre WebApps móveis e aplicativos móveis se tornará indistinta, à medida que os navegadores móveis se tornarem mais sofisticados e ganharem acesso ao hardware e às informações em nível de dispositivo.

1.2.3 Computação em nuvem

A *computação em nuvem* abrange uma infraestrutura ou “ecossistema” que permite a qualquer usuário, em qualquer lugar, utilizar um dispositivo de computação para compartilhar recursos computacionais em grande escala. A arquitetura lógica global da computação em nuvem está representada na Figura 1.3.

De acordo com a figura, os dispositivos de computação residem fora da nuvem e têm acesso a uma variedade de recursos dentro dela. Esses recursos abrangem aplicações, plataformas e infraestrutura. Em sua forma mais simples, um dispositivo de computação externa acessa a nuvem por meio de um navegador Web ou software semelhante. A nuvem dá acesso a dados residentes nos bancos de dados e em outras estruturas de dados. Além disso, os dispositivos podem acessar aplicativos executáveis, que podem ser usados no lugar das aplicações residentes no dispositivo de computação.

A implementação da computação em nuvem exige o desenvolvimento de uma arquitetura que contenha serviços de front-end e de back-end. O *front-end* inclui o dispositivo cliente (usuário) e o software aplicativo (por exemplo, um navegador) que permite o acesso ao back-end. O *back-end* inclui servidores e recursos de computação relacionados, sistemas de armazenamento de dados (por exemplo, bancos de dados), aplicativos residentes no servidor e servidores administrativos que utilizam middleware para coordenar e monitorar o tráfego, estabelecendo um conjunto de protocolos de acesso à nuvem e aos seus recursos residentes [Str08].

A arquitetura da nuvem pode ser segmentada para dar acesso a uma variedade de diferentes níveis de acesso público total para arquiteturas de nuvem privadas, acessíveis somente a quem tenha autorização.

1.2.4 Software para linha de produtos (de software)

O Software Engineering Institute define uma *linha de produtos de software* como “um conjunto de sistemas de software que compartilham um conjunto comum de recursos gerenciados, satisfazendo as necessidades específicas de um segmento de mercado ou de uma missão em particular, desenvolvidos a partir de um conjunto comum de itens básicos, contemplando uma forma prescrita” [SEI13]. De certa maneira, a noção de linha de produtos de software relacionados não é nova, mas a ideia de uma linha de produtos de software – todos desenvolvidos com a mesma aplicação subjacente e com as mesmas arquiteturas de dados, sendo todos implementados com um conjunto de componentes de software reutilizáveis em toda a linha de produtos – proporciona um potencial significativo para a engenharia.

Uma linha de produtos de software compartilha um conjunto de itens que incluem requisitos (Capítulo 8), arquitetura (Capítulo 13), padrões de projeto (Capítulo 16), componentes reutilizáveis (Capítulo 14), casos de teste (Capítulos 22 e 23) e outros produtos de trabalho para a engenharia de software. Basicamente, uma linha de produtos de software resulta no desenvolvimento de muitos produtos projetados tirando proveito dos atributos comuns a tudo que é feito dentro da linha de produtos.

1.3 Resumo

Software é o elemento-chave na evolução de produtos e sistemas baseados em computador e é uma das mais importantes tecnologias no cenário mundial. Ao longo dos últimos 50 anos, o software evoluiu de uma ferramenta especializada em análise de informações e resolução de problemas para uma indústria

propriamente dita. Mesmo assim, ainda temos problemas para desenvolver software de boa qualidade dentro do prazo e orçamento estabelecidos.

Software – programas, dados e informações descritivas – contemplam uma ampla gama de áreas de aplicação e tecnologia. O software legado continua a representar desafios especiais àqueles que precisam fazer sua manutenção.

A natureza do software é mutante. As aplicações e os sistemas baseados na Internet passaram de simples conjuntos de conteúdo informativo para sofisticados sistemas que apresentam funcionalidade complexa e conteúdo multimídia. Embora essas WebApps possuam características e requisitos exclusivos, elas não deixam de ser um tipo de software. Os aplicativos móveis apresentam novos desafios, à medida que migram para uma ampla variedade de plataformas. A computação em nuvem transformará o modo de distribuir software e o seu ambiente. O software para linha de produtos oferece eficiências em potencial na maneira de construir software.

Problemas e pontos a ponderar

1.1 Dê no mínimo mais cinco exemplos de como a lei das consequências não intencionais se aplica a software de computador.

1.2 Forneça uma série de exemplos (positivos e negativos) que indiquem o impacto do software em nossa sociedade.

1.3 Dê suas próprias respostas para as cinco perguntas feitas no início da Seção 1.1. Discuta-as com seus colegas.

1.4 Muitas aplicações modernas mudam frequentemente – antes de serem apresentadas ao usuário e depois da primeira versão ser colocada em uso. Sugira algumas maneiras de construir software para impedir a deterioração decorrente de mudanças.

1.5 Considere as sete categorias de software apresentadas na Seção 1.1.2. Você acha que a mesma abordagem em relação à engenharia de software pode ser aplicada a cada uma delas? Justifique sua resposta.

Leituras e fontes de informação complementares⁶

Literalmente milhares de livros são escritos sobre software. A grande maioria trata de linguagens de programação ou aplicações de software, porém poucos tratam do software em si. Pressman e Herron (*Software Shock*, Dorset House, 1991) apresentaram uma discussão preliminar (dirigida ao grande público) sobre software e a maneira como os profissionais o desenvolvem. O best-seller de Negroponte (*Being Digital*, Alfred A. Knopf, 1995) dá uma visão geral da computação e seu impacto global no século 21. DeMarco (*Why Does Software Cost So Much?* Dorset House, 1995) produziu um conjunto de divertidos e perspicazes ensaios sobre software e o processo pelo qual ele é desenvolvi-

⁶ A seção *Leitura e fontes de informação complementares* ao final de cada capítulo apresenta uma visão geral de publicações que podem ajudar a expandir o seu entendimento dos principais tópicos apresentados no capítulo. Criamos um site completo (em inglês) para dar suporte a este livro, no endereço www.mhhe.com/pressman. Entre os muitos temas tratados no site estão recursos de engenharia de software, capítulo por capítulo, até informações baseadas na Web que podem complementar o material apresentado em cada capítulo. Dentro desses recursos existe um link para a Amazon.com para cada livro mencionado nesta seção.

do. Ray Kurzweil (*How to Create a Mind*, Viking, 2013) discute como em breve o software imitará o pensamento humano e levará a uma “singularidade” na evolução de humanos e máquinas.

Keeves (*Catching Digital*, Business Infomedia Online, 2012) discute como os líderes empresariais devem se adaptar, à medida que o software evolui em um ritmo cada vez maior. Minasi (*The Software Conspiracy: Why Software Companies Put out Faulty Products, How They Can Hurt You, and What You Can Do*, McGraw-Hill, 2000) argumenta que o “flagelo moderno” dos bugs de software pode ser eliminado e sugere maneiras para concretizar isso. Eubanks (*Digital Dead End: Fighting for Social Justice in the Information Age*, MIT Press, 2011) e Compaine (*Digital Divide: Facing a Crisis or Creating a Myth*, MIT Press, 2001) defendem que a “separação” entre aqueles que têm acesso a fontes de informação (por exemplo, a Web) e aqueles que não o têm está diminuindo, à medida que avançamos na primeira década deste século. Kuniavsky (*Smart Things: Ubiquitous Computing User Experience Design*, Morgan Kaufman, 2010), Greenfield (*Everyware: The Dawning Age of Ubiquitous Computing*, New Riders Publishing, 2006) e Loke (*Context-Aware Pervasive Systems: Architectures for a New Breed of Applications*, Auerbach, 2006) introduzem o conceito de software “aberto” e preveem um ambiente sem fio no qual o software deve se adaptar às exigências que surgem em tempo real.

Uma ampla variedade de fontes de informação que discutem a natureza do software está disponível na Internet. Uma lista atualizada de referências relevantes (em inglês) para o processo do software pode ser encontrada no site: **www.mhhe.com/pressman**.

2

Engenharia de software

Conceitos-chave

atividades de apoio	18
camadas	15
CasaSegura	26
engenharia de software, definição	15
metodologia	17
mitos de software	23
prática	19
princípios	21
princípios gerais	21
processo de software	16
solução de problemas	19

Para desenvolver software que esteja preparado para enfrentar os desafios do século 21, devemos admitir alguns fatos:

- Software está profundamente incorporado em praticamente todos os aspectos de nossas vidas e, conseqüentemente, o número de pessoas interessadas nos recursos e nas funções oferecidas por determinada aplicação¹ tem crescido significativamente. *Depreende-se, portanto, que é preciso fazer um esforço conjunto para compreender o problema antes de desenvolver uma solução de software.*
- Os requisitos de tecnologia da informação demandados por pessoas, empresas e órgãos governamentais estão mais complexos a cada ano. Atualmente, equipes grandes desenvolvem programas de computador que, antigamente, eram desenvolvidos por um só indivíduo. Software sofisticado, outrora implementado em um ambiente computacional independente e previsível, hoje está incorporado em tudo, de produtos eletrônicos de consumo a equipamentos médicos e sistemas de armamento. *Depreende-se, portanto, que projetar se tornou uma atividade essencial.*

PANORAMA

ticas) e um leque de ferramentas que possibilitam aos profissionais desenvolverem software de altíssima qualidade.

Quem realiza? Os engenheiros de software aplicam o processo de engenharia de software.

Por que é importante? A engenharia de software é importante porque nos capacita para o desenvolvimento de sistemas complexos dentro do prazo e com alta qualidade. Ela impõe disciplina a um trabalho que pode se tornar caótico, mas também permite que as pessoas produzam software de computador adaptado à sua abordagem, da maneira mais conveniente às suas necessidades.

O que é? A engenharia de software abrange um processo, um conjunto de métodos (práticas)

Quais são as etapas envolvidas? Cria-se software para computadores da mesma forma que qualquer produto bem-sucedido: aplicando-se um processo adaptável e ágil que conduza a um resultado de alta qualidade, atendendo às necessidades daqueles que usarão o produto. Aplica-se uma abordagem de engenharia de software.

Qual é o artefato? Do ponto de vista de um engenheiro de software, software é um conjunto de programas, conteúdo (dados) e outros artefatos. Porém, do ponto de vista do usuário, o artefato consiste em informações resultantes que, de alguma forma, tornam a vida dele melhor.

Como garantir que o trabalho foi realizado corretamente? Leia o restante deste livro, escolha as ideias aplicáveis ao software que você desenvolver e use-as em seu trabalho.

¹ Neste livro, mais adiante, chamaremos tais pessoas de “envolvidos”.

- Pessoas, negócios e governos dependem, cada vez mais, de software para a tomada de decisões estratégicas e táticas, assim como para controle e para operações cotidianas. Se o software falhar, as pessoas e as principais empresas poderão ter desde pequenos inconvenientes até falhas catastróficas. *Depreende-se, portanto, que um software deve apresentar qualidade elevada.*
- À medida que o valor de uma aplicação específica aumenta, a probabilidade é de que sua base de usuários e longevidade também cresçam. À medida que sua base de usuários e seu tempo em uso forem aumentando, a demanda por adaptação e aperfeiçoamento também vai aumentar. *Depreende-se, portanto, que um software deve ser passível de manutenção.*

Entenda o problema antes de elaborar uma solução.

Qualidade e facilidade de manutenção são resultantes de um projeto bem feito.

Essas simples constatações nos conduzem a uma só conclusão: *software, em todas as suas formas e em todos os seus campos de aplicação, deve passar pelos processos de engenharia.* E isso nos leva ao tema principal deste livro – *engenharia de software.*

2.1 Definição da disciplina

O IEEE IEE93a¹ elaborou a seguinte definição para engenharia de software:

Engenharia de software: (1) A aplicação de uma abordagem sistemática, disciplinada e quantificável no desenvolvimento, na operação e na manutenção de software; isto é, a aplicação de engenharia ao software. (2) O estudo de abordagens como definido em (1).

Como definimos engenharia de software?

Entretanto, uma abordagem “sistemática, disciplinada e quantificável” aplicada por uma equipe de desenvolvimento de software pode ser pesada para outra. Precisamos de disciplina, mas também precisamos de adaptabilidade e agilidade.

A engenharia de software é uma tecnologia em camadas. Como ilustra a Figura 2.1, qualquer abordagem de engenharia (inclusive engenharia de software) deve estar fundamentada em um comprometimento organizacional com a qualidade. A gestão da qualidade total Seis Sigma e filosofias similares² promovem uma cultura de aperfeiçoamento contínuo de processos, e é essa cultura que, no final das contas, leva ao desenvolvimento de abordagens cada vez mais eficazes na engenharia de software. A pedra fundamental que sustenta a engenharia de software é o foco na qualidade.

A base da engenharia de software é a camada de *processos*. O processo de engenharia de software é a liga que mantém as camadas de tecnologia coesas e possibilita o desenvolvimento de software de forma racional e dentro do prazo. O processo define uma metodologia que deve ser estabelecida para a entrega efetiva de tecnologia de engenharia de software. O processo de software constitui a base para o controle do gerenciamento de projetos de software e estabelece o contexto no qual são aplicados métodos técnicos, são produzidos

A engenharia de software engloba um processo, métodos de gerenciamento e desenvolvimento de software, bem como ferramentas.

² A gestão da qualidade e as metodologias relacionadas são discutidas ao longo da Parte III deste livro.

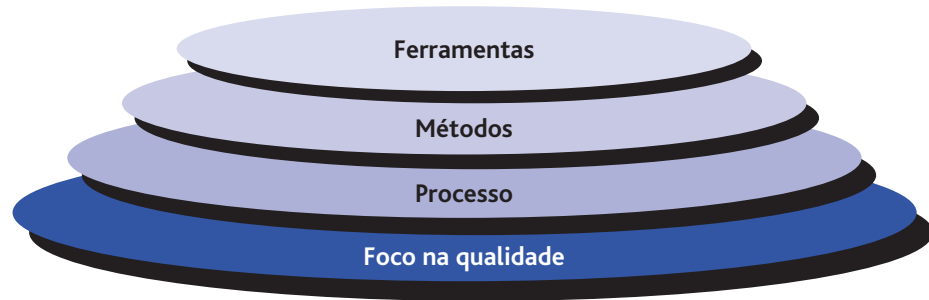


FIGURA 2.1 Camadas da engenharia de software.

artefatos (modelos, documentos, dados, relatórios, formulários etc.), são estabelecidos marcos, a qualidade é garantida e mudanças são geridas de forma apropriada.

Os *métodos* da engenharia de software fornecem as informações técnicas para desenvolver software. Os métodos envolvem uma ampla variedade de tarefas, que incluem: comunicação, análise de requisitos, modelagem de projeto, construção de programa, testes e suporte. Os métodos da engenharia de software se baseiam em um conjunto de princípios básicos que governam cada área da tecnologia e incluem atividades de modelagem e outras técnicas descritivas.

As *ferramentas* da engenharia de software fornecem suporte automatizado ou semiautomatizado para o processo e para os métodos. Quando as ferramentas são integradas, de modo que as informações criadas por uma ferramenta possam ser utilizadas por outra, é estabelecido um sistema para o suporte ao desenvolvimento de software, denominado *engenharia de software com o auxílio do computador*.

CrossTalk é um jornal que divulga informações práticas a respeito de processo, métodos e ferramentas. Pode ser encontrado no endereço: www.stsc.hill.af.mil.

2.2 O processo de software

Quais são os elementos de um processo de software?

Processo é um conjunto de atividades, ações e tarefas realizadas na criação de algum artefato. Uma *atividade* se esforça para atingir um objetivo amplo (por exemplo, comunicar-se com os envolvidos) e é utilizada independentemente do campo de aplicação, do tamanho do projeto, da complexidade dos esforços ou do grau de rigor com que a engenharia de software será aplicada. Uma *ação* (por exemplo, projeto de arquitetura) envolve um conjunto de tarefas que resultam em um artefato de software fundamental (por exemplo, um modelo arquitetural). Uma *tarefa* se concentra em um objetivo pequeno, porém bem-definido (por exemplo, realizar um teste de unidades), e produz um resultado tangível.

"Um processo define quem está fazendo o quê, quando e como para atingir determinado objetivo."

Ivar Jacobson,
Grady Booch e
James Rumbaugh

No contexto da engenharia de software, um processo *não* é uma prescrição rígida de como desenvolver um software. Ao contrário, é uma abordagem adaptável que possibilita às pessoas (a equipe de software) realizar o trabalho de selecionar e escolher o conjunto apropriado de ações e tarefas. A intenção é a de sempre entregar software dentro do prazo e com qualidade suficiente para satisfazer àqueles que patrocinaram sua criação e àqueles que vão utilizá-lo.

2.2.1 A metodologia do processo

Uma *metodologia (framework) de processo* estabelece o alicerce para um processo de engenharia de software completo por meio da identificação de um pequeno número de *atividades metodológicas* aplicáveis a todos os projetos de software, independentemente de tamanho ou complexidade. Além disso, a metodologia de processo engloba um conjunto de *atividades de apoio (umbrella activities)* aplicáveis a todo o processo de software. Uma metodologia de processo genérica para engenharia de software compreende cinco atividades:

Comunicação. Antes que qualquer trabalho técnico possa começar, é de importância fundamental se comunicar e colaborar com o cliente (e outros envolvidos).³ A intenção é entender os objetivos dos envolvidos para o projeto e reunir requisitos que ajudem a definir os recursos e as funções do software.

Planejamento. Qualquer jornada complicada pode ser simplificada com auxílio de um mapa. Um projeto de software é uma jornada complicada, e a atividade de planejamento cria um “mapa” que ajuda a guiar a equipe na sua jornada. O mapa – denominado plano de projeto de software – define o trabalho de engenharia de software, descrevendo as tarefas técnicas a serem conduzidas, os riscos prováveis, os recursos que serão necessários, os produtos resultantes a ser produzidos e um cronograma de trabalho.

Modelagem. Independentemente de ser um paisagista, um construtor de pontes, um engenheiro aeronáutico, um carpinteiro ou um arquiteto, trabalha-se com modelos todos os dias. Cria-se um “esboço” para que se possa ter uma ideia do todo – qual será o seu aspecto em termos de arquitetura, como as partes constituintes se encaixarão e várias outras características. Se necessário, refina-se o esboço com mais detalhes, numa tentativa de compreender melhor o problema e como resolvê-lo. Um engenheiro de software faz a mesma coisa, criando modelos para entender melhor as necessidades do software e o projeto que vai atender a essas necessidades.

Construção. O que se projeta deve ser construído. Essa atividade combina geração de código (manual ou automatizada) e testes necessários para revelar erros na codificação.

Entrega. O software (como uma entidade completa ou como um incremento parcialmente concluído) é entregue ao cliente, que avalia o produto entregue e fornece feedback, baseado na avaliação.

Essas cinco atividades metodológicas genéricas podem ser utilizadas para o desenvolvimento de programas pequenos e simples, para a criação de aplicações para a Internet e para a engenharia de grandes e complexos sistemas baseados em computador. Os detalhes do processo de software serão bem diferentes em cada caso, mas as atividades metodológicas permanecerão as mesmas.

Quais são as cinco atividades genéricas de uma metodologia de processo?

“Einstein afirmou que deve haver uma explicação simplificada da natureza, pois Deus não é caprichoso ou arbitrário. Tal fé não conforta o engenheiro de software. Grande parte da complexidade com a qual terá de lidar é arbitrária.”

Fred Brooks

³ *Envolvido* é qualquer pessoa que tenha interesse no êxito de um projeto – executivos, usuários, engenheiros de software, pessoal de suporte etc. Rob Thomsett ironiza: “Envolvido (*stakeholder*) é uma pessoa que segura (*hold*) uma estaca (*stake*) grande e pontiaguda... Se você não cuidar de seus envolvidos, sabe bem onde essa estaca vai parar”.

Para muitos projetos de software, as atividades metodológicas são aplicadas iterativamente conforme o projeto se desenvolve. Ou seja, comunicação, planejamento, modelagem, construção e entrega são aplicados repetidamente, sejam quantas forem as iterações do projeto. Cada iteração produzirá um *incremento de software* que disponibilizará uma parte dos recursos e das funcionalidades do software. A cada incremento, o software se torna cada vez mais completo.

2.2.2 Atividades de apoio

As atividades metodológicas do processo de engenharia de software são complementadas por diversas *atividades de apoio*. De modo geral, as atividades de apoio são aplicadas por todo um projeto de software e ajudam uma equipe de software a gerenciar e a controlar o andamento, a qualidade, as alterações e os riscos. As atividades de apoio típicas são:

Atividades de apoio ocorrem ao longo do processo de software e se concentram, principalmente, em gerenciamento, acompanhamento e controle do projeto.

Controle e acompanhamento do projeto – possibilita que a equipe avalie o progresso em relação ao plano do projeto e tome as medidas necessárias para cumprir o cronograma.

Administração de riscos – avalia riscos que possam afetar o resultado ou a qualidade do produto/projeto.

Garantia da qualidade de software – define e conduz as atividades que garantem a qualidade do software.

Revisões técnicas – avaliam artefatos da engenharia de software, tentando identificar e eliminar erros antes que se propaguem para a atividade seguinte.

Medição – define e coleta medidas (do processo, do projeto e do produto). Auxilia na entrega do software de acordo com os requisitos; pode ser usada com as demais atividades (metodológicas e de apoio).

Gerenciamento da configuração de software – gerencia os efeitos das mudanças ao longo do processo.

Gerenciamento da capacidade de reutilização – define critérios para a reutilização de artefatos (inclusive componentes de software) e estabelece mecanismos para a obtenção de componentes reutilizáveis.

A adaptação do processo de software é essencial para o sucesso de um projeto.

Preparo e produção de artefatos de software – engloba as atividades necessárias para criar artefatos como, por exemplo, modelos, documentos, logs, formulários e listas.

Cada uma dessas atividades de apoio será discutida em detalhes mais adiante.

2.2.3 Adaptação do processo

Anteriormente, declaramos que o processo de engenharia de software não é rígido nem deve ser seguido à risca. Mais do que isso, ele deve ser ágil e adaptável (ao problema, ao projeto, à equipe e à cultura organizacional). Portanto, o processo adotado para determinado projeto pode ser muito diferente daquele adotado para outro. Entre as diferenças, temos:

- Fluxo geral de atividades, ações e tarefas e suas interdependências.
- Até que ponto as ações e tarefas são definidas dentro de cada atividade da metodologia.
- Até que ponto artefatos de software são identificados e exigidos.
- Modo de aplicar as atividades de garantia da qualidade.
- Modo de aplicar as atividades de acompanhamento e controle do projeto.
- Grau geral de detalhamento e rigor da descrição do processo.
- Grau de envolvimento com o projeto (por parte do cliente e de outros envolvidos).
- Nível de autonomia dada à equipe de software.
- Grau de prescrição da organização da equipe.

A Parte I deste livro examina o processo de software com um grau de detalhamento considerável.

"Sinto que uma receita consiste em apenas um tema com o qual um cozinheiro inteligente pode brincar, cada vez com uma variação."

Madame Benoit

2.3 A prática da engenharia de software

A Seção 2.2 apresentou uma introdução a um modelo de processo de software genérico, composto por um conjunto de atividades que estabelecem uma metodologia para a prática da engenharia de software. As atividades genéricas da metodologia – **comunicação, planejamento, modelagem, construção e entrega** –, bem como as atividades de apoio, estabelecem um esquema para o trabalho da engenharia de software. Mas como a prática da engenharia de software se encaixa nisso? Nas seções seguintes, você vai adquirir um conhecimento básico dos princípios e conceitos genéricos que se aplicam às atividades de uma metodologia.⁴

Diversas citações instigantes sobre a prática da engenharia de software podem ser encontradas em www.literate-programming.com.

2.3.1 A essência da prática

No livro clássico *How to Solve It*, escrito antes de os computadores modernos existirem, George Polya [Pol45] descreveu em linhas gerais a essência da solução de problemas e, conseqüentemente, a essência da prática da engenharia de software:

1. *Compreender o problema* (comunicação e análise).
2. *Planejar uma solução* (modelagem e projeto de software).
3. *Executar o plano* (geração de código).
4. *Examinar o resultado para ter precisão* (testes e garantia da qualidade).

No contexto da engenharia de software, essas etapas de bom senso conduzem a uma série de questões essenciais [adaptado de Pol45]:

Compreenda o problema. Algumas vezes é difícil de admitir; porém, a maioria de nós é arrogante quando um problema nos é apresentado. Ouvimos por

Pode-se afirmar que a abordagem de Polya é simplesmente uma questão de bom senso. É verdade. Mas é espantoso como o bom senso é tão pouco usado no mundo do software.

O elemento mais importante para se entender um problema é escutar.

⁴ Você deve rever seções relevantes contidas neste capítulo à medida que discutirmos os métodos de engenharia de software e as atividades de apoio específicas mais adiante neste livro.

alguns segundos e então pensamos: “Ah, sim, estou entendendo, vamos começar a resolver este problema”. Infelizmente, compreender nem sempre é assim tão fácil. Vale a pena despende um pouco de tempo respondendo a algumas perguntas simples:

- *Quem tem interesse na solução do problema?* Ou seja, quem são os envolvidos?
- *Quais são as incógnitas?* Que dados, funções e recursos são necessários para resolver apropriadamente o problema?
- *O problema pode ser compartimentalizado?* É possível representá-lo em problemas menores que talvez sejam mais fáceis de ser compreendidos?
- *O problema pode ser representado graficamente?* É possível criar um modelo analítico?

Planeje a solução. Agora você entende o problema (ou assim pensa) e não vê a hora de começar a codificar. Antes de fazer isso, relaxe um pouco e faça um pequeno projeto:

- *Você já viu problemas semelhantes anteriormente?* Existem padrões que são reconhecíveis em uma possível solução? Existe algum software que implemente os dados, as funções e as características necessárias?
- *Algum problema semelhante já foi resolvido?* Em caso positivo, existem elementos da solução que podem ser reutilizados?
- *É possível definir subproblemas?* Em caso positivo, existem soluções aparentes e imediatas para eles?
- *É possível representar uma solução de maneira que conduza a uma implementação efetiva?* É possível criar um modelo de projeto?

“Há um grão de descoberta na solução de qualquer problema.”

George Polya

Leve o plano adiante. O projeto elaborado que criamos serve como um mapa para o sistema que se quer construir. Podem surgir desvios inesperados, e é possível que se descubra um caminho ainda melhor à medida que se prossiga; porém, o “planejamento” permitirá que continuemos sem nos perder.

- *A solução é adequada ao plano?* O código-fonte pode ser atribuído ao modelo de projeto?
- *Todas as partes componentes da solução estão provavelmente corretas?* O projeto e o código foram revistos ou, melhor ainda, provas da correção foram aplicadas ao algoritmo?

Examine o resultado. Não se pode ter certeza de que uma solução seja perfeita; porém, pode-se assegurar que um número de testes suficiente tenha sido realizado para revelar o maior número de erros possível.

- *É possível testar cada parte componente da solução?* Foi implementada uma estratégia de testes razoável?
- *A solução produz resultados adequados aos dados, às funções e às características necessários?* O software foi validado em relação a todas as solicitações dos envolvidos?

Não é surpresa que grande parte dessa metodologia consista no bom senso. De fato, é possível afirmar que uma abordagem de bom senso à engenharia de software jamais o levará ao mau caminho.

2.3.2 Princípios gerais

O dicionário define a palavra *princípio* como “uma importante afirmação ou lei básica em um sistema de pensamento”. Ao longo deste livro serão discutidos princípios em vários níveis de abstração. Alguns se concentram na engenharia de software como um todo, outros consideram uma atividade de metodologia genérica específica (por exemplo, **comunicação**) e outros ainda destacam as ações de engenharia de software (por exemplo, projeto de arquitetura) ou tarefas técnicas (por exemplo, redigir um cenário de uso). Independentemente do seu nível de enfoque, os princípios ajudam a estabelecer um modo de pensar para a prática segura da engenharia de software. Esta é a razão por que são importantes.

David Hooker [Hoo96] propôs sete princípios que se concentram na prática da engenharia de software como um todo. Eles são reproduzidos nos parágrafos a seguir:⁵

Primeiro princípio: *a razão de existir*

Um sistema de software existe por um motivo: *agregar valor para seus usuários*. Todas as decisões devem ser tomadas com esse princípio em mente. Antes de especificar um requisito de um sistema, antes de indicar alguma parte da funcionalidade de um sistema, antes de determinar as plataformas de hardware ou os processos de desenvolvimento, pergunte a si mesmo: “Isso realmente agrega valor real ao sistema?”. Se a resposta for “não”, não o faça. Todos os demais princípios se apoiam neste primeiro.

Segundo princípio: *KISS (Keep It Simple, Stupid!, ou seja: não complique!)*

O projeto de software não é um processo casual. Existem muitos fatores a considerar em qualquer trabalho de projeto. *Todo projeto deve ser o mais simples possível, mas não simplista*. Esse princípio contribui para um sistema mais fácil de compreender e manter. Isso não significa que características, até mesmo as internas, devam ser descartadas em nome da simplicidade. De fato, os projetos mais elegantes normalmente são os mais simples. Simples também não significa “gambiarra”. Na verdade, muitas vezes são necessárias muitas reflexões e trabalho em várias iterações para simplificar. A contrapartida é um software mais fácil de manter e menos propenso a erros.

Terceiro princípio: *mantenha a visão*

Uma visão clara é essencial para o sucesso. Sem ela, um projeto se torna ambíguo. Sem uma integridade conceitual, corre-se o risco de transformar o projeto em uma colcha de retalhos de projetos incompatíveis, unidos por parafusos inadequados... Comprometer a visão arquitetural de um sistema de software debilita e até poderá destruir sistemas bem projetados. Ter um ar-

Antes de iniciar um projeto, certifique-se de que o software tem um propósito para a empresa e de que seus usuários reconhecem seu valor.

“Há certa majestade na simplicidade, que está muito acima de toda a excentricidade do saber.”

**Alexandre Pope
(1688-1744)**

⁵ Reproduzido com a permissão do autor [Hoo96]. Hooker define padrões para esses princípios em <http://c2.com/cgi/wiki?SevenPrinciplesOfSoftwareDevelopment>.

Um software de valor mudará ao longo de sua vida útil. Por essa razão, ele deve ser desenvolvido para fácil manutenção.

quiteto responsável e capaz de manter a visão clara e de reforçar a adequação ajuda a assegurar o êxito de um projeto.

Quarto princípio: *o que um produz outros consomem*

Raramente um sistema de software de qualidade industrial é construído e utilizado de forma isolada. De uma maneira ou de outra, alguém mais vai usar, manter, documentar ou, de alguma forma, depender da capacidade de entender seu sistema. Portanto, *sempre especifique, projete e implemente ciência de que mais alguém terá de entender o que você está fazendo*. O público para qualquer produto de desenvolvimento de software é potencialmente grande. Especifique tendo como objetivo os usuários. Projete tendo em mente os implementadores. Codifique se preocupando com aqueles que deverão manter e ampliar o sistema. Alguém terá de depurar o código que você escreveu, e isso o torna um usuário de seu código. Facilitando o trabalho de todas essas pessoas, você agrega maior valor ao sistema.

Quinto princípio: *esteja aberto para o futuro*

Um sistema com tempo de vida mais longo tem mais valor. Nos ambientes computacionais de hoje, em que as especificações mudam de um instante para outro, e as plataformas de hardware se tornam rapidamente obsoletas, a vida de um software, em geral, é medida em meses. Contudo, os verdadeiros sistemas de software com “qualidade industrial” devem durar muito mais. Para serem bem-sucedidos nisso, esses sistemas precisam estar prontos para se adaptar a essas e outras mudanças. Sistemas que obtêm sucesso são aqueles que foram projetados dessa forma desde seu princípio. *Jamais faça projetos limitados*. Sempre pergunte “e se” e prepare-se para todas as respostas possíveis, criando sistemas que resolvam o problema geral, não apenas o específico.⁶ Isso muito provavelmente conduziria à reutilização de um sistema inteiro.

Sexto princípio: *planeje com antecedência, visando a reutilização*

A reutilização economiza tempo e esforço.⁷ Alcançar um alto grau de reutilização é indiscutivelmente a meta mais difícil de ser atingida ao se desenvolver um sistema de software. A reutilização de código e projetos tem sido proclamada como uma grande vantagem do uso de tecnologias orientadas a objetos. Contudo, o retorno desse investimento não é automático. Aproveitar as possibilidades de reutilização – oferecidas pela programação orientada a objetos (ou convencional) – exige planejamento e capacidade de fazer previsões. Existem várias técnicas para levar a cabo a reutilização em cada um dos níveis do processo de desenvolvimento do sistema... *Planejar com anteceden-*

⁶ Esse conselho pode ser perigoso se levado ao extremo. Projetar para o “problema geral” algumas vezes exige comprometer o desempenho e pode tornar ineficientes as soluções específicas.

⁷ Embora isso seja verdade para aqueles que reutilizam o software em projetos futuros, a reutilização poderá ser cara para aqueles que precisarem projetar e desenvolver componentes reutilizáveis. Estudos indicam que o projeto e o desenvolvimento de componentes reutilizáveis podem custar de 25 a 200% mais do que o próprio software. Em alguns casos, o diferencial de custo não pode ser justificado.

cia para a reutilização reduz o custo e aumenta o valor tanto dos componentes reutilizáveis quanto dos sistemas aos quais eles serão incorporados.

Sétimo princípio: *pense!*

Este último princípio é, provavelmente, o mais menosprezado. *Pensar bem e de forma clara antes de agir quase sempre produz melhores resultados.* Quando se analisa alguma coisa, provavelmente ela sairá correta. Ganha-se também conhecimento de como fazer correto novamente. Se você realmente analisar algo e mesmo assim o fizer da forma errada, isso se tornará uma valiosa experiência. Um efeito colateral da análise é aprender a reconhecer quando não se sabe algo, e até que ponto poderá buscar o conhecimento. Quando a análise clara faz parte de um sistema, seu valor aflora. Aplicar os seis primeiros princípios exige intensa reflexão, para a qual as recompensas em potencial são enormes.

Se todo engenheiro de software e toda a equipe de software simplesmente seguissem os sete princípios de Hooker, muitas das dificuldades enfrentadas no desenvolvimento de complexos sistemas baseados em computador seriam eliminadas.

2.4 Mitos do desenvolvimento de software

Os mitos criados para o desenvolvimento de software – crenças infundadas sobre o software e sobre o processo utilizado para criá-lo – remontam aos primórdios da computação. Os mitos possuem uma série de atributos que os tornam insidiosos. Por exemplo, eles parecem ser, de fato, afirmações sensatas (algumas vezes contendo elementos de verdade), têm uma sensação intuitiva e frequentemente são promulgados por praticantes experientes “que entendem do riscado”.

Atualmente, a maioria dos profissionais versados na engenharia de software reconhece os mitos por aquilo que eles representam – atitudes enganosas que provocaram sérios problemas tanto para gerentes quanto para praticantes da área. Entretanto, antigos hábitos e atitudes são difíceis de ser modificados, e resquícios de mitos de software permanecem.

Mitos de gerenciamento. Gerentes com responsabilidade sobre software, assim como gerentes da maioria das áreas, frequentemente estão sob pressão para manter os orçamentos, evitar deslizamentos nos cronogramas e elevar a qualidade. Como uma pessoa que está se afogando e se agarra a uma tábua, um gerente de software muitas vezes se agarra à crença em um mito do software para aliviar a pressão (mesmo que temporariamente).

Software Project Managers Network, em www.spmn.com, pode ajudá-lo a refutar esses e outros mitos.

Mito: *Já temos um livro cheio de padrões e procedimentos para desenvolver software. Ele não supriria meu pessoal com tudo que precisam saber?*

Realidade: O livro com padrões pode muito bem existir, mas ele é realmente utilizado? Os praticantes da área estão cientes de que ele existe? Esse livro reflete a prática moderna da engenharia de software? É completo? É adaptável? Está otimi-

zado para melhorar o tempo de entrega, mantendo ainda o foco na qualidade? Em muitos casos, a resposta para todas essas perguntas é “não”.

Mito: *Se o cronograma atrasar, poderemos acrescentar mais programadores e ficar em dia (algumas vezes denominado conceito da “horda mongol”).*

Realidade: O desenvolvimento de software não é um processo mecânico como o de uma fábrica. Nas palavras de Brooks [Bro95]: “acrescentar pessoas em um projeto de software atrasado só o tornará mais atrasado ainda”. A princípio, essa declaração pode parecer absurda. No entanto, o que ocorre é que, quando novas pessoas entram, as que já estavam terão de gastar tempo situando os recém-chegados, reduzindo, consequentemente, o tempo destinado ao desenvolvimento produtivo. Pode-se adicionar pessoas, mas somente de forma planejada e bem coordenada.

Mito: *Se eu decidir terceirizar o projeto de software, posso simplesmente relaxar e deixar a outra empresa realizá-lo.*

Realidade: Se uma organização não souber gerenciar e controlar projetos de software, ela vai, invariavelmente, enfrentar dificuldades ao terceirizá-los.

Mitos dos clientes. O cliente solicitante do software computacional pode ser uma pessoa na mesa ao lado, um grupo técnico do andar de baixo, de um departamento de marketing/vendas ou uma empresa externa que contratou o projeto. Em muitos casos, o cliente acredita em mitos sobre software porque gerentes e profissionais da área pouco fazem para corrigir falsas informações. Mitos conduzem a falsas expectativas (do cliente) e, em última instância, à insatisfação com o desenvolvedor.

Esforce-se ao máximo para compreender o que deve ser feito antes de começar. Você pode não chegar a todos os detalhes, mas, quanto mais souber, menor será o risco.

Mito: *Uma definição geral dos objetivos é suficiente para começar a escrever os programas – podemos preencher os detalhes posteriormente.*

Realidade: Embora nem sempre seja possível uma definição ampla e estável dos requisitos, uma definição de objetivos ambígua é a receita para um desastre. Requisitos não ambíguos (normalmente derivados iterativamente) são obtidos somente pela comunicação contínua e eficaz entre cliente e desenvolvedor.

Mito: *Os requisitos de software mudam continuamente, mas as mudanças podem ser facilmente assimiladas, pois o software é flexível.*

Realidade: É verdade que os requisitos de software mudam, mas o impacto da mudança varia dependendo do momento em que foi introduzida. Quando as mudanças dos requisitos são solicitadas cedo (antes de o projeto ou de a codificação te-

rem começado), o impacto sobre os custos é relativamente pequeno.⁸ Entretanto, conforme o tempo passa, ele aumenta rapidamente – recursos foram comprometidos, uma estrutura de projeto foi estabelecida e mudar pode causar uma revolução que exija recursos adicionais e modificações fundamentais no projeto.

Mitos dos profissionais da área. Mitos que ainda sobrevivem entre os profissionais da área têm resistido por mais de 60 anos de cultura da programação. Durante seus primórdios, a programação era vista como uma forma de arte. Hábitos e atitudes antigos são difíceis de perder.

Mito: *Uma vez que o programa foi feito e colocado em uso, nosso trabalho está terminado.*

Realidade: Uma vez alguém já disse que “o quanto antes se começar a codificar, mais tempo levará para terminar”. Levantamentos indicam que entre 60 e 80% de todo o esforço será despendido após a entrega do software ao cliente pela primeira vez.

Mito: *Até que o programa esteja “em execução”, não há como avaliar sua qualidade.*

Realidade: Um dos mecanismos de garantia da qualidade de software mais eficientes pode ser aplicado a partir da concepção de um projeto – *a revisão técnica*. Os revisores de software (descritos no Capítulo 20) são “filtros de qualidade”, considerados mais eficientes do que os testes feitos para encontrar certas classes de defeitos de software.

Mito: *O único produto passível de entrega é o programa em funcionamento.*

Realidade: Um programa funcionando é somente uma parte de uma configuração de software que inclui muitos elementos. Uma variedade de artefatos (por exemplo, modelos, documentos, planos) constitui uma base para uma engenharia bem-sucedida e, mais importante, uma orientação para suporte de software.

Mito: *A engenharia de software nos fará criar documentação volumosa e desnecessária e, invariavelmente, vai nos retardar.*

Realidade: O objetivo da engenharia de software não é criar documentos. É criar um produto de qualidade. Uma qualidade melhor leva à redução do retrabalho. E retrabalho reduzido resulta em tempos de entrega menores.

Atualmente, muitos profissionais de software reconhecem a falácia dos mitos que acabamos de descrever. Estar ciente das realidades do software é o primeiro passo para buscar soluções práticas na engenharia de software.

Toda vez que pensar “não temos tempo para engenharia de software”, pergunte a si mesmo: “teremos tempo para fazer de novo?”.

⁸ Muitos engenheiros de software têm adotado uma abordagem “ágil” que favorece as alterações incrementais, controlando, com isso, seu impacto e seu custo. Os métodos ágeis são discutidos no Capítulo 5.

2.5 Como tudo começa

Todo projeto de software é motivado por alguma necessidade de negócios – a necessidade de corrigir um defeito em uma aplicação existente; a necessidade de adaptar um “sistema legado” a um ambiente de negócios em constante transformação; a necessidade de ampliar as funções e os recursos de uma aplicação existente ou a necessidade de criar um novo produto, serviço ou sistema.

No início de um projeto de software, a necessidade do negócio é, com frequência, expressa informalmente como parte de uma simples conversa. A conversa apresentada no quadro a seguir é típica.



Como começa um projeto

Cena: Sala de reuniões da CPI Corporation, empresa (fictícia) que fabrica produtos de consumo para uso doméstico e comercial.

Atores: Mal Golden, gerente sênior, desenvolvimento do produto; Lisa Perez, gerente de marketing; Lee Warren, gerente de engenharia; Joe Camalleri, vice-presidente executivo, desenvolvimento de negócios.

Conversa:

Joe: Lee, ouvi dizer que o seu pessoal está trabalhando em algo. Do que se trata? Um tipo de caixa sem fio de uso amplo e genérico?

Lee: Trata-se de algo bem legal... aproximadamente do tamanho de uma caixa de fósforos, conectável a todo tipo de sensor, como uma câmera digital – ou seja, se conecta a quase tudo. Usa o protocolo sem fio 802.11n. Permite que acessemos saídas de dispositivos sem o emprego de fios. Acreditamos que nos levará a uma geração inteiramente nova de produtos.

Joe: Você concorda, Mal?

Mal: Sim. Na verdade, com as vendas tão baixas neste ano, precisamos de algo novo. Lisa e eu fizemos uma pequena pesquisa de mercado e acreditamos que conseguimos uma linha de produtos que poderá ser ampla.

Joe: Ampla em que sentido?

CASASEGURA⁹

Mal (evitando comprometimento direto): Conte sobre nossa ideia, Lisa.

Lisa: Trata-se de uma geração completamente nova na linha de “produtos de gerenciamento doméstico”. Chamamos esses produtos de *CasaSegura*. Eles usam uma nova interface sem fio e oferecem a pequenos empresários e proprietários de residências um sistema que é controlado por seus PCs, envolvendo segurança doméstica, sistemas de vigilância, controle de eletrodomésticos e dispositivos. Por exemplo, seria possível diminuir a temperatura do aparelho de ar condicionado enquanto você está voltando para casa, esse tipo de coisa.

Lee (reagindo sem pensar): O departamento de engenharia fez um estudo de viabilidade técnica dessa ideia, Joe. É possível fazê-lo com baixo custo de fabricação. A maior parte dos componentes do hardware é encontrada no mercado. O software é um problema, mas não é nada que não possamos resolver.

Joe: Interessante. Mas eu perguntei qual é o ponto principal.

Mal: PCs e tablets estão em mais de 70% dos lares nos EUA. Se pudermos acertar no preço, essa aplicação poderá ser excepcional. Ninguém mais tem nosso dispositivo sem fio... ele é exclusivo! Estaremos dois anos à frente de nossos concorrentes... e as receitas? Algo em torno de 30 a 40 milhões no segundo ano...

Joe (sorrindo): Vamos levar isso adiante. Estou interessado.

Exceto por uma rápida referência, o software mal foi mencionado como parte da conversa. Ainda assim, o software vai decretar o sucesso ou o fracasso da linha de produtos *CasaSegura*. O trabalho de engenharia só terá êxito se o software do *CasaSegura* tiver êxito. O mercado só vai aceitar o produto se o

⁹ O projeto *CasaSegura* será usado ao longo deste livro para ilustrar o funcionamento interno de uma equipe de projeto à medida que ela constrói um produto de software. A empresa, o projeto e as pessoas são fictícios, porém as situações e os problemas são reais.

software incorporado atender adequadamente às necessidades (ainda não declaradas) do cliente. Acompanharemos a evolução da engenharia do software *CasaSegura* em vários dos capítulos que estão por vir.

2.6 Resumo

A engenharia de software engloba processos, métodos e ferramentas que possibilitam a construção de sistemas complexos baseados em computador dentro do prazo e com qualidade. O processo de software incorpora cinco atividades estruturais: comunicação, planejamento, modelagem, construção e entrega, e elas se aplicam a todos os projetos de software. A prática da engenharia de software é uma atividade de resolução de problemas que segue um conjunto de princípios básicos.

Inúmeros mitos em relação ao software continuam a levar gerentes e profissionais para o mau caminho, mesmo com o aumento do conhecimento coletivo sobre software e das tecnologias necessárias para construí-los. À medida que for aprendendo mais sobre a engenharia de software, você começará a compreender por que esses mitos devem ser derrubados toda vez que nos depararmos com eles.

Problemas e pontos a ponderar

2.1. A Figura 2.1 coloca as três camadas de engenharia de software acima de uma camada intitulada “foco na qualidade”. Isso implica um programa de qualidade organizacional como o de gestão da qualidade total. Pesquise um pouco a respeito e crie um sumário dos princípios básicos de um programa de gestão da qualidade total.

2.2. A engenharia de software é aplicável na construção de WebApps? Em caso positivo, como poderia ser modificada para atender às características únicas das WebApps?

2.3. À medida que o software invade todos os setores, riscos ao público (devido a programas com imperfeições) passam a ser uma preocupação cada vez maior. Crie um cenário o mais catastrófico possível, porém realista, em que a falha de um programa de computador poderia causar um grande dano em termos econômicos ou humanos.

2.4. Descreva uma metodologia de processo com suas próprias palavras. Ao afirmarmos que atividades de modelagem se aplicam a todos os projetos, isso significa que as mesmas tarefas são aplicadas a todos os projetos, independentemente de seu tamanho e complexidade? Explique.

2.5. As atividades de apoio ocorrem ao longo do processo de software. Você acredita que elas são aplicadas de forma homogênea ao longo do processo ou algumas delas são concentradas em uma ou mais atividades da metodologia?

2.6. Acrescente mais dois mitos à lista apresentada na Seção 2.4. Declare também a realidade que acompanha o mito.

Leituras e fontes de informação complementares

O estado atual da engenharia de software e do processo de software pode ser mais bem determinado a partir de publicações como *IEEE Software*, *IEEE Computer*, *CrossTalk* e *IEEE*

Transactions on Software Engineering. Periódicos do setor, como *Application Development Trends* e *Cutter IT Journal*, normalmente contêm artigos sobre tópicos da engenharia de software. A disciplina é “sintetizada” todos os anos no *Proceeding of the International Conference on Software Engineering*, patrocinado pelo IEEE e ACM, e é discutida de forma aprofundada em periódicos como *ACM Transactions on Software Engineering and Methodology*, *ACM Software Engineering Notes* e *Annals of Software Engineering*. Dezenas de milhares de páginas Web são dedicadas à engenharia de software e ao processo de software.

Nos últimos anos, foram publicados vários livros sobre o processo de desenvolvimento de software e sobre a engenharia de software. Alguns fornecem uma visão geral de todo o processo, ao passo que outros se aprofundam em tópicos específicos importantes, em detrimento dos demais. Entre as ofertas mais populares (além deste livro, é claro!), temos:

SWEBOK: Guide to the Software Engineering Body of Knowledge,¹⁰ IEEE, 2013, consulte: <http://www.computer.org/portal/web/swebok>

Andersson, E., et al., *Software Engineering for Internet Applications*, MIT Press, 2006.

Braude, E. e M. Bernstein, *Software Engineering: Modern Approaches*, 2ª ed., Wiley, 2010.

Christensen, M. e R. Thayer, *A Project Manager's Guide to Software Engineering Best Practices*, IEEE-CS Press (Wiley), 2002.

Glass, R., *Fact and Fallacies of Software Engineering*, Addison-Wesley, 2002.

Hussain, S., *Software Engineering*, I K International Publishing House, 2013.

Jacobson, I., *Object-Oriented Software Engineering: A Use Case Driven Approach*, 2ª ed., Addison-Wesley, 2008.

Jalote, P., *An Integrated Approach to Software Engineering*, 3ª ed., Springer, 2010.

Pfleeger, S., *Software Engineering: Theory and Practice*, 4ª ed., Prentice Hall, 2009.

Schach, S., *Object-Oriented and Classical Software Engineering*, 8ª ed., McGraw-Hill, 2010.

Sommerville, I., *Software Engineering*, 9ª ed., Addison-Wesley, 2010.

Stober, T. e U. Hansmann, *Agile Software Development: Best Practices for Large Development Projects*, Springer, 2009.

Tsui, F. e O. Karam, *Essentials of Software Engineering*, 2ª ed., Jones & Bartlett Publishers, 2009.

Nygard (*Release It!: Design and Deploy Production-Ready Software*, Pragmatic Bookshelf, 2007), Richardson e Gwaltney (*Ship it! A Practical Guide to Successful Software Projects*, Pragmatic Bookshelf, 2005) e Humble e Farley (*Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, Addison-Wesley, 2010) apresentam uma ampla coleção de diretrizes úteis, aplicáveis à atividade de entrega.

Ao longo das últimas décadas foram publicados diversos padrões de engenharia de software pelo IEEE, pela ISO e suas organizações de padronização. Moore (*The Road Map to Software Engineering: A Standards-Based Guide*, IEEE Computer Society Press [Wiley], 2006) disponibiliza uma pesquisa útil sobre padrões relevantes e como aplicá-los em projetos reais.

Uma ampla variedade de fontes de informação sobre engenharia de software e o processo de software está disponível na Internet. Uma lista atualizada de referências relevantes (em inglês) para o processo para o software pode ser encontrada no site: www.mhhe.com/pressman.

¹⁰ Disponível gratuitamente em <<http://www.computer.org/portal/web/swebok/htmlformat>>.

PARTE



O processo de software

Nesta parte do livro, você vai aprender sobre o processo que fornece uma metodologia para a prática da engenharia de software. Estas questões são tratadas nos capítulos que seguem:

- O que é um processo de software?
- Quais são as atividades metodológicas genéricas presentes em todos os processos de software?
- Como os processos são modelados e o que são padrões de processo?
- O que são modelos de processo prescritivo e quais são seus pontos fortes e fracos?
- Por que *agilidade* é um lema no trabalho da engenharia de software moderna?
- O que é desenvolvimento de software ágil e como ele se diferencia dos modelos de processos mais tradicionais?

Respondidas essas questões, você estará mais bem preparado para compreender o contexto no qual a prática da engenharia de software é aplicada.

3

Estrutura do processo de software

Conceitos-chave

aperfeiçoamento de processos	37
avaliação de processos	37
conjunto de tarefas	34
fluxo de processo	31
modelo de processo genérico	31
padrões de processo.....	34

Em um livro fascinante que apresenta a visão de um economista sobre software e engenharia de software, Howard Baetjer Jr. [Bae98] comenta o processo de software:

Porque o software, como todo capital, é conhecimento incorporado, e porque esse conhecimento é, inicialmente, disperso, tácito, latente e, em considerável medida, incompleto, o desenvolvimento de software é um processo de aprendizado social. Esse processo é um diálogo no qual o conhecimento, que deverá se tornar o software, é coletado, reunido e incorporado ao software. O processo possibilita a interação entre usuários e projetistas, entre usuários e ferramentas em evolução e entre projetistas e ferramentas em evolução (tecnologia). Trata-se de um processo iterativo no qual a própria ferramenta em evolução serve como meio de comunicação, com cada nova rodada do diálogo extraindo mais conhecimento útil das pessoas envolvidas.

De fato, construir software é um processo de aprendizado social iterativo e o resultado, algo que Baetjer denominaria “capital de software”, é a incorporação do conhecimento coletado, filtrado e organizado à medida que se desenvolve o processo.

Mas, do ponto de vista técnico, o que é exatamente um processo de software? No contexto deste livro, definimos *processo de software* como uma metodologia para as atividades, ações e tarefas necessárias para desenvolver

PANORAMA

previsíveis – um roteiro que ajude a criar um resultado de alta qualidade e dentro do prazo estabelecido. O roteiro é denominado “processo de software”.

Quem realiza? Engenheiros de software e seus gerentes adaptam o processo às suas necessidades e então o seguem. Os solicitantes do software têm um papel a desempenhar no processo de definição, construção e teste do software.

Por que é importante? Porque propicia estabilidade, controle e organização para uma atividade que pode se tornar bastante caótica sem controle. Entretanto, uma abordagem de engenharia de software moderna deve ser “ágil”. Deve demandar apenas atividades, controles e artefatos que sejam apropriados para a equipe do projeto e para o produto a ser produzido.

O que é? Quando se elabora um produto ou sistema, é importante seguir uma série de passos

Quais são as etapas envolvidas? O processo adotado depende do software a ser desenvolvido. Determinado processo pode ser adequado para um software do sistema de voo de uma aeronave, enquanto um processo totalmente diferente pode ser indicado para a criação de um site.

Qual é o artefato? Do ponto de vista de um engenheiro de software, os artefatos são os programas, os documentos e os dados produzidos pelas atividades e tarefas definidas pelo processo.

Como garantir que o trabalho foi realizado corretamente? Há muitos mecanismos de avaliação que permitem às empresas determinarem o nível de “maturidade” de seu processo de software. Entretanto, a qualidade, o cumprimento de prazos e a viabilidade em longo prazo do produto que se desenvolve são os melhores indicadores da eficácia do processo utilizado.

um software de alta qualidade. “Processo” é sinônimo de “engenharia de software”? A resposta é “sim e não”. Um processo de software define a abordagem adotada conforme um software é elaborado pela engenharia. Entretanto, a engenharia de software também engloba tecnologias que fazem parte do processo – métodos técnicos e ferramentas automatizadas.

Mais importante, a engenharia de software é realizada por pessoas criativas e com amplo conhecimento, e que devem adaptar um processo de software maduro de modo que fique adequado aos produtos desenvolvidos e às demandas de seu mercado.

3.1 Um modelo de processo genérico

No Capítulo 2, processo foi definido como um conjunto de atividades de trabalho, ações e tarefas realizadas quando algum artefato de software deve ser criado. Cada uma dessas atividades, ações e tarefas se aloca dentro de uma metodologia ou modelo que determina sua relação com o processo e umas com as outras.

O processo de software está representado esquematicamente na Figura 3.1. De acordo com a figura, cada atividade metodológica é composta por um conjunto de ações de engenharia de software. Cada ação é definida por um *conjunto de tarefas*, o qual identifica as tarefas de trabalho a ser completadas, os artefatos de software que serão produzidos, os fatores de garantia da qualidade que serão exigidos e os marcos utilizados para indicar progresso.

Como discutido no Capítulo 2, uma metodologia de processo genérica para engenharia de software estabelece cinco atividades metodológicas: **comunicação, planejamento, modelagem, construção e entrega**. Além disso, um conjunto de atividades de apoio é aplicado ao longo do processo, como o acompanhamento e controle do projeto, a administração de riscos, a garantia da qualidade, o gerenciamento das configurações, as revisões técnicas, entre outras.

Um aspecto importante do processo de software ainda não foi discutido. Esse aspecto – chamado *fluxo de processo* – descreve como são organizadas as atividades metodológicas, bem como as ações e tarefas que ocorrem dentro de cada atividade em relação à sequência e ao tempo, como ilustrado na Figura 3.2.

Um *fluxo de processo linear* executa cada uma das cinco atividades metodológicas em sequência, começando com a comunicação e culminando com a entrega (Figura 3.2a). Um *fluxo de processo iterativo* repete uma ou mais das atividades antes de prosseguir para a seguinte (Figura 3.2b). Um *fluxo de processo evolucionário* executa as atividades de forma “circular”. Cada volta pelas cinco atividades conduz a uma versão mais completa do software (Figura 3.2c). Um *fluxo de processo paralelo* (Figura 3.2d) executa uma ou mais atividades em paralelo com outras (por exemplo, a modelagem para um aspecto do software poderia ser executada em paralelo com a construção de outro aspecto do software).

A hierarquia de trabalho técnico, dentro do processo de software, consiste em atividades e ações abrangentes compostas por tarefas.

O que é fluxo de processo?

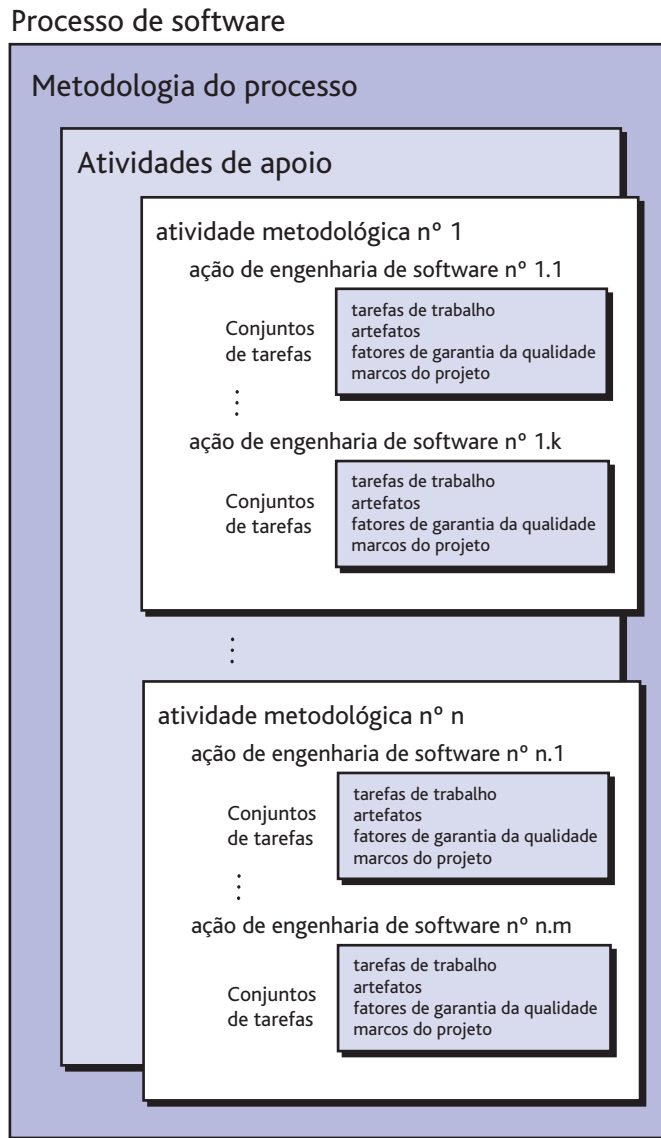


FIGURA 3.1 Uma metodologia do processo de software.

3.2 Definição de uma atividade metodológica

"Se o processo estiver correto, os resultados falarão por si mesmos."

Takashi Osada

Como uma atividade metodológica é modificada de acordo com as alterações da natureza do projeto?

Embora cinco atividades metodológicas tenham sido descritas e tenha-se fornecido uma definição básica de cada uma delas no Capítulo 2, uma equipe de software precisa de muito mais informações antes de poder executar qualquer uma das atividades como parte do processo de software. Assim, enfrenta-se uma questão-chave: *Quais ações são apropriadas para uma atividade metodológica, uma vez fornecidos a natureza do problema a ser solucionado, as características das pessoas que farão o trabalho e os envolvidos no projeto?*

Para um pequeno projeto de software solicitado por uma única pessoa (em um local distante) com requisitos simples e objetivos, a atividade de **comunicação** pode se resumir a pouco mais de um telefonema ou email para o

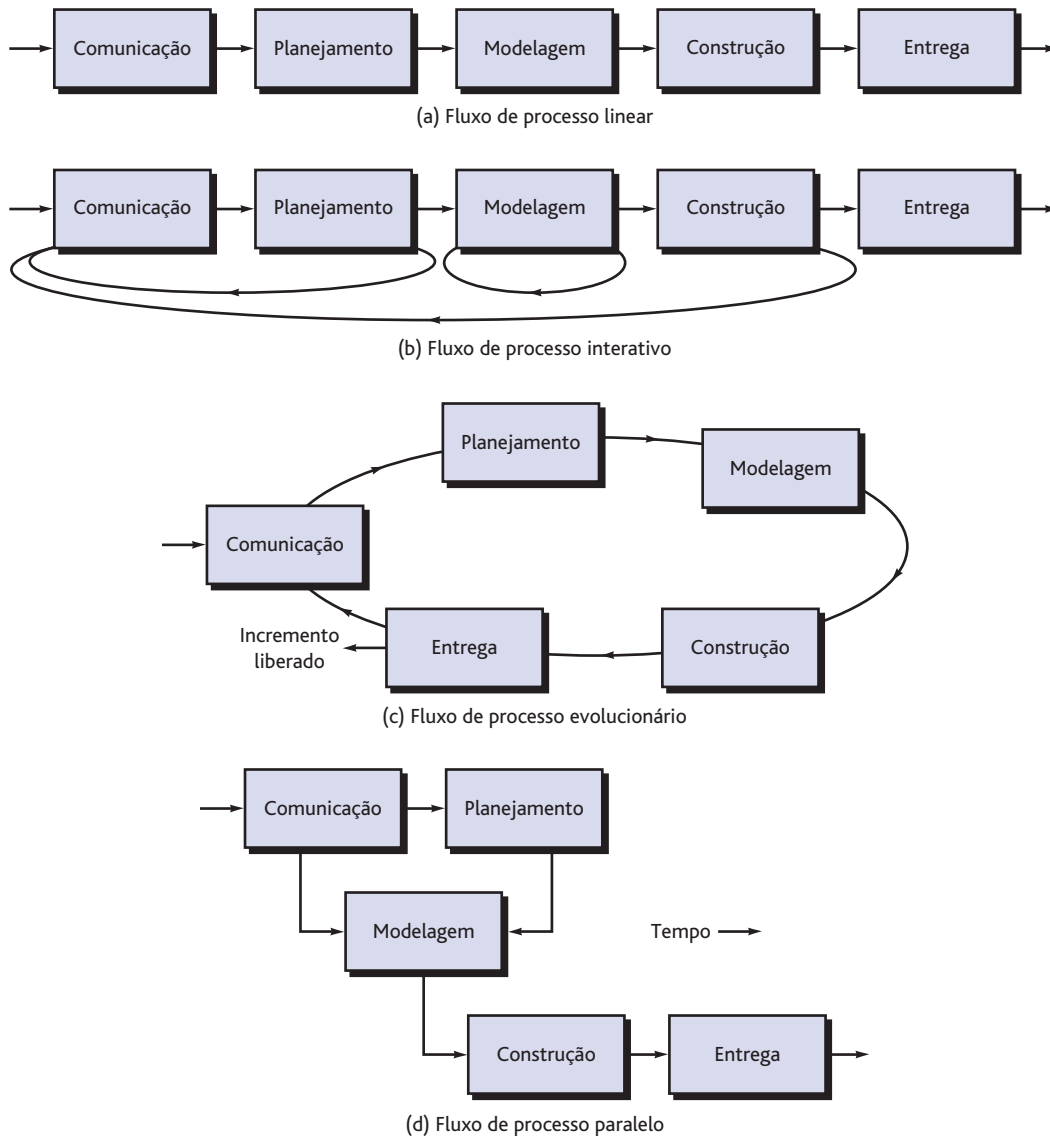


FIGURA 3.2 Fluxo de processo.

envolvido. Portanto, a única ação necessária é uma *conversa telefônica*, e as tarefas de trabalho (o *conjunto de tarefas*) que essa ação envolve são:

1. Contatar o envolvido via telefone.
2. Discutir os requisitos e gerar anotações.
3. Organizar as anotações em uma breve relação de requisitos, por escrito.
4. Enviar um email para o envolvido para revisão e aprovação.

Se o projeto fosse consideravelmente mais complexo, com muitos envolvidos, cada qual com um conjunto de requisitos diferentes (por vezes conflitantes), a atividade de comunicação poderia ter seis ações distintas (descritas no Capítulo 8): *concepção*, *levantamento*, *elaboração*, *negociação*, *especificação* e *validação*. Cada uma dessas ações de engenharia de software conteria muitas tarefas de trabalho e uma série de diferentes artefatos.

Projetos diferentes exigem conjuntos de tarefas diferentes. A equipe de software escolhe o conjunto de tarefas baseada no problema e nas características do projeto.

INFORMAÇÕES

**Conjunto de tarefas**

Um conjunto de tarefas define o trabalho a ser feito para atingir os objetivos de uma ação de engenharia de software. Por exemplo, *levantamento* (mais comumente denominada “levantamento de requisitos”) é uma importante ação de engenharia de software que ocorre durante a atividade de **comunicação**. A meta do levantamento de requisitos é compreender o que os vários envolvidos esperam do software a ser desenvolvido.

Para um projeto pequeno e relativamente simples, o conjunto de tarefas para levantamento dos requisitos seria semelhante a este:

1. Fazer uma lista dos envolvidos no projeto.
2. Fazer uma reunião informal com todos os envolvidos.
3. Solicitar a cada envolvido uma lista com as características e funções necessárias.
4. Discutir sobre os requisitos e elaborar uma lista final.
5. Organizar os requisitos por grau de prioridade.
6. Destacar pontos de incertezas.

Para um projeto de software maior e mais complexo, é preciso usar um conjunto diferente de tarefas. Esse conjunto pode incluir as seguintes tarefas de trabalho:

1. Fazer uma lista dos envolvidos no projeto.
2. Entrevistar separadamente cada um dos envolvidos para levantamento geral de suas expectativas e necessidades.

3. Fazer uma lista preliminar das funções e características, com base nas informações fornecidas pelos envolvidos.
4. Agendar uma série de reuniões facilitadoras para especificação de aplicações.
5. Realizar reuniões.
6. Incluir cenários informais de usuários como parte de cada reunião.
7. Refinar os cenários de usuários, com base no feedback dos envolvidos.
8. Fazer uma lista revisada dos requisitos dos envolvidos.
9. Empregar técnicas de implantação de funções de qualidade para estabelecer graus de prioridade dos requisitos.
10. Agrupar os requisitos de modo que possam ser entregues em incrementos.
11. Fazer um levantamento das limitações e restrições que serão aplicadas ao sistema.
12. Discutir sobre os métodos para validação do sistema.

Esses dois conjuntos de tarefas atingem o objetivo do “levantamento de requisitos”; porém, são bem diferentes quanto ao seu grau de profundidade e formalidade. A equipe de software deve escolher o conjunto de tarefas que possibilite atingir o objetivo de cada ação, mantendo, inclusive, a qualidade e a agilidade.

3.3 Identificação de um conjunto de tarefas

Voltando novamente à Figura 3.1, cada ação de engenharia de software (por exemplo, *levantamento*, uma ação associada à atividade de **comunicação**) pode ser representada por vários e diferentes *conjuntos de tarefas* – constituídos por uma gama de tarefas de trabalho de engenharia de software, artefatos relacionados, fatores de garantia da qualidade e marcos do projeto.

Deve-se escolher um conjunto de tarefas mais adequado às necessidades do projeto e às características da equipe. Isso significa que uma ação de engenharia de software pode ser adaptada às necessidades específicas do projeto de software e às características da equipe.

3.4 Padrões de processo

O que é padrão de processo?

Toda equipe de desenvolvimento encontra problemas à medida que avança no processo de software. Seria útil se soluções comprovadas estivessem pronta-

mente à disposição da equipe, de modo que os problemas pudessem ser localizados e resolvidos rapidamente. Um *padrão de processo*¹ descreve um problema de processo encontrado durante o trabalho de engenharia de software, identificando o ambiente onde foi encontrado e sugerindo uma ou mais soluções comprovadas para o problema. Em termos mais genéricos, um padrão de processo fornece um modelo [Amb98] – um método consistente para descrever soluções de problemas no contexto do processo de software. Combinando padrões, uma equipe conseguirá solucionar problemas e elaborar um processo que melhor atenda às necessidades de um projeto.

Padrões podem ser definidos em qualquer nível de abstração.² Em alguns casos, um padrão poderia ser utilizado para descrever um problema (e sua solução) associado ao modelo de processo completo (por exemplo, prototipação). Em outras situações, os padrões podem ser usados para descrever um problema (e sua solução) associado a uma atividade metodológica (por exemplo, **planejamento**) ou uma ação dentro de uma atividade metodológica (por exemplo, estimativa de custos do projeto).

Ambler [Amb98] propôs um modelo para descrever um padrão de processo:

Nome do padrão. O padrão deve receber um nome que o descreva no contexto do processo de software (por exemplo, **RevisõesTécnicas**).

Forças. Ambiente onde se encontram o padrão e as questões que tornam visível o problema e que poderiam afetar sua solução.

Tipo. É especificado o tipo de padrão. Ambler sugere três tipos:

1. **Padrão de estágio** – define um problema associado a uma atividade metodológica para o processo. Como uma atividade metodológica envolve várias ações e tarefas de trabalho, um padrão de estágio engloba vários padrões de tarefas (veja o próximo padrão) relevantes ao estágio (atividade metodológica). Um exemplo de padrão de estágio poderia ser **EstabelecimentoDeComunicação**. Esse padrão poderia incorporar o padrão de tarefas **LevantamentoDeRequisitos** e outros.
2. **Padrão de tarefas** – define um problema associado a uma ação de engenharia de software ou tarefa de trabalho relevante para a prática de engenharia de software bem-sucedida (por exemplo, **LevantamentoDeRequisitos** é um padrão de tarefas).
3. **Padrão de fases** – define a sequência das atividades metodológicas que ocorrem dentro do processo, mesmo quando o fluxo geral de atividades é iterativo por natureza. Um exemplo de padrão de fases seria **ModeloEspiral** ou **Prototipação**.³

"A repetição de padrões é algo bem diferente da repetição de partes. De fato, as partes diferentes serão únicas, pois os padrões são os mesmos."

Christopher Alexander

Um modelo de padrões propicia um meio consistente para descrever um padrão.

¹ Uma discussão detalhada sobre padrões é apresentada no Capítulo 11.

² Os padrões são aplicáveis a várias atividades de engenharia de software. Padrões de análise, de projeto e de testes são discutidos nos Capítulos 11, 13, 15, 16 e 20. Padrões e "antipadrões" para atividades de gerenciamento de projetos são discutidos na Parte IV deste livro.

³ Esses padrões de fases são discutidos no Capítulo 4.

"Achamos que desenvolvedores de software não percebem uma realidade essencial: a maioria das organizações não sabe o que faz. Elas acham que sabem, mas não sabem."

Tom DeMarco

Contexto inicial. Descreve as condições sob as quais o padrão se aplica. Antes do início do padrão: (1) Que atividades organizacionais ou relacionadas à equipe já ocorreram? (2) Qual é o estado inicial do processo? (3) Que informação de engenharia de software ou de projeto já existe?

Por exemplo, o padrão **Planejamento** (um padrão de estágio) exige que: (1) clientes e engenheiros de software tenham estabelecido uma comunicação colaborativa; (2) tenha ocorrido a finalização bem-sucedida de uma série de padrões de tarefas [especificados] para o padrão **Comunicação**; e (3) sejam conhecidos o escopo e as restrições do projeto, bem como os requisitos básicos do negócio.

Problema. O problema específico a ser resolvido pelo padrão.

Solução. Descreve como implementar o padrão de forma bem-sucedida. Esta seção descreve como o estado inicial do processo (que existe antes de o padrão ser implementado) é modificado como consequência do início do padrão. Descreve também como as informações de engenharia de software ou de projeto que se encontram à disposição antes do início do padrão são transformadas como consequência da execução bem-sucedida do padrão.

Contexto resultante. Descreve as condições que resultarão assim que o padrão tiver sido implementado com êxito. Após a finalização do padrão: (1) Quais atividades organizacionais ou relacionadas à equipe devem ter ocorrido? (2) Qual é o estado de saída do processo? (3) Quais informações de engenharia de software ou de projeto foram desenvolvidas?

Padrões relativos. Fornecem uma lista de todos os padrões de processo que estão diretamente relacionados ao processo em questão. Essa lista pode ser representada de forma hierárquica ou em alguma outra forma com diagramas. Por exemplo, o padrão de estágio **Comunicação** envolve os padrões de tarefas: **EquipeDeProjeto**, **DiretrizesColaborativas**, **IsolamentoDoEscopo**, **LevantamentoDeRequisitos**, **DescriçãoDasRestrições** e **CriaçãoDeCenários**.

Usos conhecidos e exemplos. Indicam as instâncias específicas a que o padrão é aplicável. Por exemplo, **Comunicação** é obrigatória no início de todo projeto de software, é recomendável ao longo de todo o projeto de software e é obrigatória assim que a atividade **Entrega** estiver em andamento.

Padrões de processo propiciam um mecanismo eficaz para a localização de problemas associados a qualquer processo de software. Os padrões permitem que se desenvolva uma descrição do processo de forma hierárquica que se inicia com nível alto de abstração (um padrão de fases). A descrição é então refinada em um conjunto de padrões de estágio que descreve atividades metodológicas e são ainda mais refinadas, de uma forma hierárquica, em padrões de tarefa mais detalhados para cada padrão de estágio. Uma vez que os padrões de processos tenham sido desenvolvidos, eles poderão ser reutilizados na definição de variantes do processo – isto é, um modelo de processo personalizado pode ser definido por uma equipe de software usando os padrões como blocos de construção para o modelo do processo.

Muitos recursos sobre padrões de processo podem ser encontrados em www.ambyssoft.com/processPatternsPage.html.

INFORMAÇÕES

**Um exemplo de padrão de processo**

O padrão de processo resumido mostrado a seguir descreve uma abordagem que pode ser aplicada quando os envolvidos têm uma ideia geral do que precisa ser feito, mas estão incertos quanto aos requisitos específicos do software.

Nome do padrão. Requisitos Imprecisos

Intuito. Esse padrão descreve uma abordagem voltada à construção de um modelo (um protótipo) passível de ser avaliado iterativamente pelos envolvidos, em um esforço para identificar ou solidificar requisitos do software.

Tipo. Padrão de fase.

Contexto inicial. As seguintes condições devem ser atendidas antes de iniciar esse padrão: (1) envolvidos identificados; (2) forma de comunicação entre envolvidos e equipe de software já determinada; (3) principal problema de software a ser resolvido já identificado pelos envolvidos; (4) compreensão inicial do escopo do projeto, dos requisitos de negócio básicos e das restrições do projeto já atingida.

Problema. Os requisitos são vagos ou inexistentes, ainda assim há o reconhecimento claro de que existe um problema

a ser solucionado e ele deve ser identificado utilizando-se uma solução de software. Os envolvidos não sabem o que querem, ou seja, eles não conseguem descrever os requisitos de software em detalhe.

Solução. Uma descrição do processo de prototipação poderia ser apresentada nesta etapa – e está disponível posteriormente, na Seção 4.1.3.

Contexto resultante. Um protótipo de software que identifique os requisitos básicos (por exemplo, modos de interação, características computacionais, funções de processamento) é aprovado pelos envolvidos. Em seguida, (1) o protótipo pode evoluir por uma série de incrementos para se tornar o software de produção ou (2) o protótipo pode ser descartado, e o software de produção ser construído usando-se algum outro padrão de processos.

Padrões relacionados. Os seguintes padrões estão relacionados a esse padrão: **Comunicação Com O Cliente, Projeto Iterativo, Desenvolvimento Iterativo, Avaliação Do Cliente, Extração De Requisitos.**

Usos conhecidos e exemplos. A prototipação é recomendada quando os requisitos são incertos.

3.5 Avaliação e aperfeiçoamento de processos

A existência de um processo de software não garante que o software será entregue dentro do prazo, que estará de acordo com as necessidades do cliente ou que apresentará características técnicas que resultarão em qualidade de longo prazo (Capítulo 19). Os padrões de processo devem ser combinados com uma prática de engenharia de software confiável (Parte II deste livro). Além disso, o próprio processo pode ser avaliado para que esteja de acordo com um conjunto de critérios de processo básicos, comprovados como essenciais para uma engenharia de software bem-sucedida.⁴

Ao longo das últimas décadas foi proposta uma série de diferentes abordagens de avaliação e aperfeiçoamento dos processos de software:

SCAMPI (Standard CMMI Assessment Method for Process Improvement) (Método Padrão CMMI de Avaliação para Aperfeiçoamento de Processo da CMMI) – fornece um modelo de avaliação do processo de cinco etapas, contendo cinco fases: início, diagnóstico, estabelecimento, atuação e aprendizado. O método SCAMPI usa o CMMI da SEI como base para avaliação [SEI00].

A avaliação tenta compreender o atual estado do processo de software com o intuito de aperfeiçoá-lo.

⁴ A CMMI [CMM07] da SEI descreve, de forma extremamente detalhada, as características de um processo de software e os critérios para o êxito de um processo.

"As empresas de software têm mostrado grandes deficiências em tirar proveito das experiências adquiridas com projetos executados."

NASA

CBA IPI (CMM-Based Appraisal for Internal Process Improvement) (Avaliação para Aperfeiçoamento do Processo Interno baseada na CMM) – fornece uma técnica de diagnóstico para avaliar a maturidade relativa de uma organização de software; usa a CMM da SEI como base para a avaliação [Dun01].

SPICE (ISO/IEC15504) – padrão que define um conjunto de requisitos para avaliação do processo de software. A finalidade do padrão é auxiliar as organizações no desenvolvimento de uma avaliação objetiva da eficácia de um processo qualquer de software [ISO08].

ISO 9001:2000 para Software – padrão genérico aplicável a qualquer organização que queira aperfeiçoar a qualidade global de produtos, sistemas ou serviços fornecidos. Portanto, o padrão é aplicável diretamente a organizações e empresas de software [Ant06].

Uma discussão mais detalhada sobre métodos de avaliação de software e aperfeiçoamento de processo é apresentada no Capítulo 37.

3.6 Resumo

Um modelo de processo genérico para engenharia de software consiste em um conjunto de atividades metodológicas e de apoio, ações e tarefas a realizar. Cada modelo de processo, entre os vários existentes, pode ser descrito por um fluxo de processo diferente – uma descrição de como as atividades metodológicas, ações e tarefas são organizadas, sequencial e cronologicamente. Padrões de processo são utilizados para resolver problemas comuns encontrados no processo de software.

Problemas e pontos a ponderar

3.1. Na introdução deste capítulo, Baetjer observa: "O processo oferece interação entre usuários e projetistas, entre usuários e ferramentas em evolução e entre projetistas e ferramentas de tecnologia em evolução". Liste cinco perguntas que (1) os projetistas deveriam fazer aos usuários, (2) os usuários deveriam fazer aos projetistas, (3) os usuários deveriam fazer a si mesmos sobre o produto de software a ser desenvolvido, (4) os projetistas deveriam fazer a si mesmos sobre o produto de software a ser construído e sobre o processo que será usado para construí-lo.

3.2. Discuta as diferenças entre os vários fluxos de processo descritos na Seção 3.1. Consegue identificar tipos de problemas que poderiam ser aplicáveis a cada um dos fluxos genéricos descritos?

3.3. Tente desenvolver um conjunto de ações para a atividade de comunicação. Selecione uma ação e defina um conjunto de tarefas para ela.

3.4. Durante a **comunicação**, um problema comum ocorre ao encontrarmos dois envolvidos com ideias conflitantes sobre como o software deve ser. Isto é, há requisitos mutuamente conflitantes. Desenvolva um padrão de processo (que seja um padrão de estágio) usando o modelo apresentado na Seção 3.4 que trata desse problema e sugira uma abordagem eficaz para ele.

Leituras e fontes de informação complementares

A maioria dos livros-texto sobre engenharia de software considera os modelos de processo com certo nível de detalhe. Livros de Sommerville (*Software Engineering*, 9ª ed., Addison-Wesley, 2010), Schach (*Object-Oriented and Classical Software Engineering*, 8ª ed., McGraw-Hill, 2010) e Pfleeger e Atlee (*Software Engineering: Theory and Practice*, 4ª ed., Prentice Hall, 2009) consideram os paradigmas tradicionais e discutem suas vantagens e desvantagens. Munch e seus colegas (*Software Process Definition and Management*, Springer, 2012) apresentam uma visão do processo e do produto da engenharia de software e de sistemas. Glass (*Facts and Fallacies of Software Engineering*, Prentice Hall, 2002) fornece uma visão simples e pragmática do processo de engenharia de software. Embora não seja especificamente dedicado a processos, Brooks (*The Mythical Man-Month*, 2ª ed., Addison-Wesley, 1995) apresenta conhecimentos sábios de projeto antigo que têm tudo a ver com processos.

Firesmith e Henderson-Sellers (*The OPEN Process Framework: An Introduction*, Addison-Wesley, 2001) apresentam um quadro geral para a criação de “processos de software flexíveis e que, ainda assim, não deixam de ser disciplinados” e discutem atributos e objetivos dos processos. Madachy (*Software Process Dynamics*, Wiley-IEEE, 2008) fala sobre técnicas de modelagem que possibilitam a análise dos elementos técnicos e sociais relacionados do processo de software. Sharpe e McDermott (*Workflow Modeling: Tools for Process Improvement and Application Development*, 2ª ed., Artech House, 2008) apresentam ferramentas para modelagem de processos de software e de negócios.

Uma ampla variedade de fontes de informação sobre engenharia de software e o processo de software está disponível na Internet. Uma lista atualizada de referências relevantes (em inglês) para o processo para o software pode ser encontrada no site: www.mhhe.com/pressman.

4

Modelos de processo

Conceitos-chave

desenvolvimento baseado em componentes	52
desenvolvimento de software orientado a aspectos	54
ferramentas de modelagem de processos	62
modelo cascata	41
modelo de métodos formais	53
modelo evolucionário	45
modelo espiral	47
modelo V	42
modelos concorrentes	49
modelos de processo incremental	43
processo de software de equipe	60
processo de software pessoal	59
processo unificado	55
prototipação	45
tecnologia de processos	61

Originalmente, os modelos de processo foram propostos para trazer ordem ao caos existente na área de desenvolvimento de software. A história demonstra que esses modelos fazem uma considerável contribuição à estrutura utilizável no trabalho de engenharia de software e fornecem um roteiro razoavelmente eficaz para as equipes de software. Entretanto, o trabalho de engenharia de software e os produtos gerados permanecem “à beira do caos”.

Em um intrigante artigo sobre o estranho relacionamento entre ordem e caos no mundo do software, Nogueira e seus colegas [Nog00] afirmam que

O limiar do caos é definido como “um estado natural entre ordem e caos, um grande comprometimento entre estrutura e surpresa”. [Kau95] O limiar do caos pode ser visualizado como um estado instável, parcialmente estruturado... Instável porque é constantemente atraído para o caos ou para a ordem absoluta.

Tendemos a pensar que a ordem é o estado ideal da natureza. Isso pode ser um erro. Pesquisas... defendem a teoria de que a operação longe do equilíbrio gera criatividade, processos auto-organizados e lucros crescentes [Roo96]. Ordem absoluta implica ausência de variabilidade, o que poderia ser uma vantagem em ambientes imprevisíveis. A mudança ocorre quando existe uma estrutura que permite que a mudança seja organizada, mas tal estrutura não deve ser tão rígida a ponto de impedir que a mudança ocorra. Por outro lado, caos em demasia pode

PANORAMA

O que é? Um modelo de processo fornece um guia específico para o trabalho de engenharia de software. Ele define o fluxo de todas as atividades, ações e tarefas, o grau de iteração, os artefatos e a organização do trabalho a ser feito.

Quem realiza? Os engenheiros de software e seus gerentes adaptam um modelo de processo às suas necessidades e então o seguem. Os solicitantes do software têm um papel a desempenhar no processo de definição, construção e teste do software.

Por que é importante? Porque o processo propicia estabilidade, controle e organização para uma atividade que pode, sem controle, tornar-se bastante caótica. Entretanto, uma abordagem de engenharia de software moderna deve ser “ágil”. Deve demandar apenas atividades, controles e produtos

de trabalho que sejam apropriados para a equipe do projeto e para o produto a ser gerado.

Quais são as etapas envolvidas? O modelo de processo fornece os “passos” necessários para realizar um trabalho de engenharia de software disciplinado.

Qual é o artefato? Do ponto de vista de um engenheiro de software, o artefato é uma descrição personalizada das atividades e tarefas definidas pelo processo.

Como garantir que o trabalho foi realizado corretamente? Há muitos mecanismos de avaliação de processos de software que possibilitam às organizações determinar o nível de “maturidade” de seu processo de software. Entretanto, a qualidade, o cumprimento de prazos e a viabilidade em longo prazo do produto que se desenvolve são os melhores indicadores da eficácia do processo utilizado.

impossibilitar a coordenação e a coerência. A falta de estrutura nem sempre implica desordem.

As implicações filosóficas desse argumento são importantes para a engenharia de software. Cada modelo de processo descrito neste capítulo tenta encontrar um equilíbrio entre a necessidade de pôr ordem em um mundo caótico e a de ser adaptável quando as coisas mudam constantemente.

A finalidade dos modelos de processo é tentar reduzir o caos presente no desenvolvimento de novos produtos de software.

4.1 Modelos de processo prescritivo

Um *modelo de processo prescritivo*¹ concentra-se em estruturar e ordenar o desenvolvimento de software. As atividades e tarefas ocorrem sequencialmente, com diretrizes de progresso definidas. Mas os modelos prescritivos são adequados para o mundo do software que se alimenta de mudanças? Se rejeitarmos os modelos de processo tradicionais (e a ordem implícita) e os substituímos por algo menos estruturado, tornaremos impossível atingir a coordenação e a coerência no trabalho de software?

Não há respostas fáceis para essas questões, mas existem alternativas disponíveis para os engenheiros de software. Nas próximas seções, examinamos a abordagem dos processos prescritivos, nos quais a ordem e a consistência do projeto são questões predominantes. Chamamos esses processos de “prescritivos” porque prescrevem um conjunto de elementos de processo – atividades metodológicas, ações de engenharia de software, tarefas, artefatos, garantia da qualidade e mecanismos de controle de mudanças para cada projeto. Cada modelo de processo também prescreve um fluxo de processo (também denominado *fluxo de trabalho*) – ou seja, a forma pela qual os elementos do processo estão relacionados.

Todos os modelos de processo de software podem acomodar as atividades metodológicas genéricas descritas nos Capítulos 2 e 3; porém, cada um deles dá uma ênfase diferente a essas atividades e define um fluxo de processo que invoca cada atividade metodológica (bem como tarefas e ações de engenharia de software) de forma diversa.

Um premiado “jogo de simulação de processos”, que inclui os mais importantes modelos de processo prescritivos, pode ser encontrado em: <http://www.ics.uci.edu/~emilyo/SimSE/downloads.html>.

4.1.1 O modelo cascata

Há casos em que os requisitos de um problema são bem compreendidos – quando o trabalho flui da comunicação à disponibilização de modo relativamente linear. Essa situação ocorre algumas vezes quando adaptações ou aperfeiçoamentos bem-definidos precisam ser feitos em um sistema existente (por exemplo, uma adaptação em software contábil exigida devido a mudanças nas normas governamentais). Pode ocorrer também em um número limitado de novos esforços de desenvolvimento, mas apenas quando os requisitos estão bem definidos e são razoavelmente estáveis.

Os modelos de processo prescritivo definem um conjunto prescrito de elementos de processo e um fluxo de trabalho de processo previsível.

¹ Os modelos de processo prescritivos são, algumas vezes, conhecidos como modelos de processo “tradicionais”.

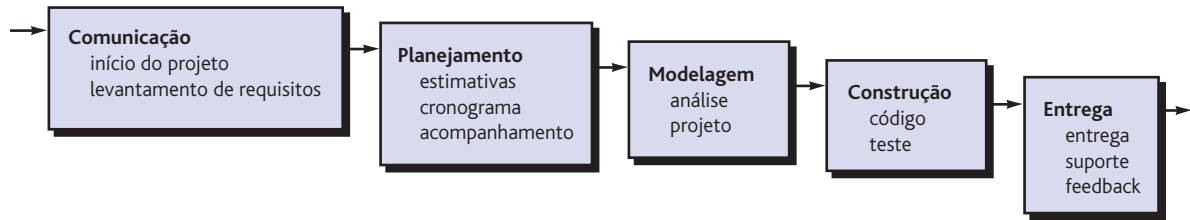


FIGURA 4.1 O modelo cascata.

O modelo V ilustra como as ações de verificação e validação estão associadas a ações de engenharia anteriores.

O *modelo cascata*, algumas vezes chamado *ciclo de vida clássico*, sugere uma abordagem sequencial² e sistemática para o desenvolvimento de software, começando com a especificação dos requisitos do cliente, avançando pelas fases de planejamento, modelagem, construção e disponibilização, e culminando no suporte contínuo do software concluído (Figura 4.1).

Uma variação na representação do modelo cascata é denominada *modelo V*. Representado na Figura 4.2, o modelo V [Buc99] descreve a relação entre ações de garantia da qualidade e ações associadas a comunicação, modelagem e atividades de construção iniciais. À medida que a equipe de software desce em direção ao lado esquerdo do V, os requisitos básicos do problema são refinados em representações cada vez mais detalhadas e técnicas do problema e de sua solução. Uma vez gerado o código, a equipe passa para o lado direito do V, basicamente realizando uma série de testes (ações de garantia da qualidade) que validam cada um dos modelos criados à medida que a equipe desce pelo lado esquerdo.³ Na realidade, não há nenhuma diferença fundamental entre o ciclo de vida clássico e o modelo V. O modelo V oferece uma maneira de visualizar como as ações de verificação e validação são aplicadas a um trabalho de engenharia anterior.

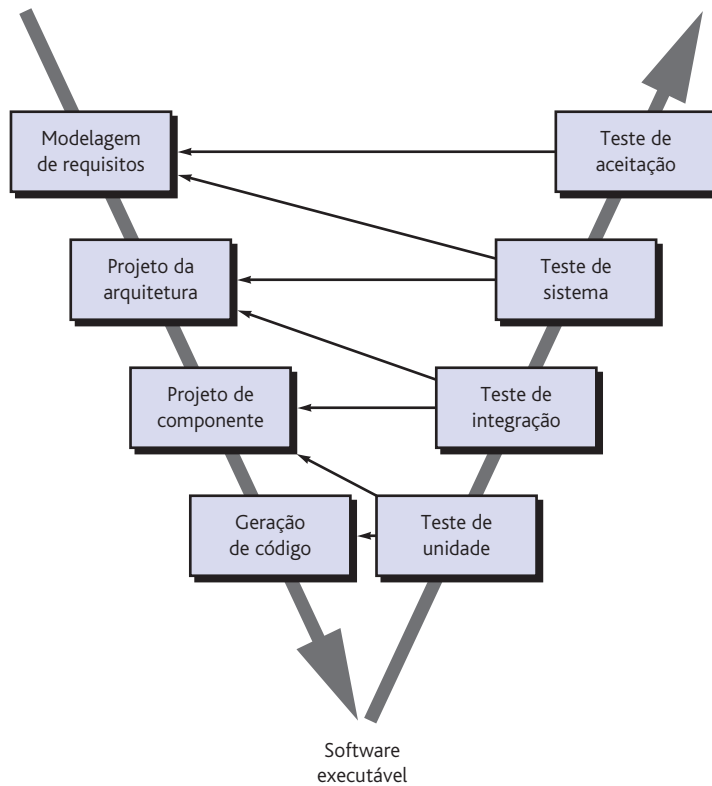
O modelo cascata é o paradigma mais antigo da engenharia de software. Entretanto, ao longo das últimas quatro décadas, as críticas a este modelo de processo fizeram até mesmo seus mais árdios defensores questionarem sua eficácia [Han95]. Entre os problemas às vezes encontrados quando se aplica o modelo cascata, temos:

Por que algumas vezes o modelo cascata falha?

1. Projetos reais raramente seguem o fluxo sequencial proposto pelo modelo. Embora o modelo linear possa conter iterações, ele o faz indiretamente. Como consequência, mudanças podem provocar confusão à medida que a equipe de projeto prossegue.
2. Frequentemente, é difícil para o cliente estabelecer explicitamente todas as necessidades. O modelo cascata exige isso e tem dificuldade para adequar a incerteza natural existente no início de muitos projetos.
3. O cliente deve ter paciência. Uma versão operacional do(s) programa(s) não estará disponível antes de estarmos próximos ao final do projeto. Um

² Embora o modelo cascata proposto por Winston Royce [Roy70] previsse os “*feedback loops*”, a vasta maioria das organizações que aplica esse modelo de processo os trata como se fossem estritamente lineares.

³ Na Parte III deste livro, é apresentada uma discussão detalhada sobre ações de garantia da qualidade.

**FIGURA 4.2** Modelo V.

erro grave, se não detectado até o programa operacional ser revisto, pode ser desastroso.

Em uma interessante análise de projetos reais, Bradac [Bra94] descobriu que a natureza linear do ciclo de vida clássico conduz a “estados de bloqueio”, nos quais alguns membros da equipe de projeto têm de aguardar outros completarem tarefas dependentes. O tempo gasto na espera pode exceder o tempo gasto em trabalho produtivo! O estado de bloqueio tende a prevalecer no início e no final de um processo sequencial linear.

Hoje, o trabalho com software tem um ritmo acelerado e está sujeito a uma cadeia de mudanças intermináveis (em características, funções e conteúdo de informações). O modelo cascata é frequentemente inadequado para esse trabalho. Entretanto, ele pode servir como um modelo de processo útil em situações nas quais os requisitos são fixos e o trabalho deve ser realizado até sua finalização de forma linear.

“Muito frequentemente, o trabalho de software segue a primeira lei do ciclismo: não importa aonde se esteja indo, é sempre ladeira acima e contra o vento.”

Autor desconhecido

4.1.2 Modelos de processo incremental

Há várias situações em que os requisitos iniciais do software são razoavelmente bem definidos; entretanto, o escopo geral do trabalho de desenvolvimento, impede o uso de um processo puramente linear. Pode ser necessário o rápido fornecimento de determinado conjunto funcional aos usuários para, somente após esse fornecimento, refinar e expandir sua funcionalidade em versões de software posteriores. Em tais casos, pode-se optar por

O modelo incremental libera uma série de versões, denominadas incrementos, que oferecem, progressivamente, maior funcionalidade ao cliente à medida que cada incremento é entregue.

Seu cliente exige a entrega em uma data impossível de atender. Sugira entregar um ou mais incrementos nessa data e o restante do software (incrementos adicionais) posteriormente.

Os modelos de processo evolucionário produzem uma versão cada vez mais completa do software a cada iteração.

um modelo de processo projetado para desenvolver o software de forma incremental.

O modelo incremental combina os fluxos de processo linear e paralelo dos elementos, discutidos no Capítulo 3. Na Figura 4.3, o modelo incremental aplica sequências lineares de forma escalonada, à medida que o tempo vai avançando. Cada sequência linear produz “incrementos” entregáveis do software [McD93].

Por exemplo, um software de processamento de textos desenvolvido com o emprego do paradigma incremental, poderia liberar funções básicas de gerenciamento de arquivos, edição e produção de documentos no primeiro incremento; recursos mais sofisticados de edição e produção de documentos no segundo; revisão ortográfica e gramatical no terceiro; e, finalmente, recursos avançados de formatação (layout) de página no quarto incremento. Deve-se notar que o fluxo de processo para qualquer incremento pode incorporar o paradigma da prototipação, discutido na próxima subseção.

Quando se utiliza um modelo incremental, frequentemente o primeiro incremento é um *produto essencial*. Ou seja, os requisitos básicos são atendidos; porém, muitos recursos complementares (alguns conhecidos, outros não) ainda não são entregues. Esse produto essencial é utilizado pelo cliente (ou passa por uma avaliação detalhada). Como resultado do uso e/ou avaliação, é desenvolvido um planejamento para o incremento seguinte. O planejamento já considera a modificação do produto essencial para melhor se adequar às necessidades do cliente e à entrega de recursos e funcionalidades adicionais. Esse processo é repetido após a liberação de cada incremento até que seja gerado o produto completo.

4.1.3 Modelos de processo evolucionário

Como todos os sistemas complexos, software evolui ao longo do tempo. Conforme o desenvolvimento do projeto avança, os requisitos do negócio e do produto frequentemente mudam, tornando inadequado seguir um planejamento em linha reta de um produto final. Prazos apertados, determinados pelo mer-

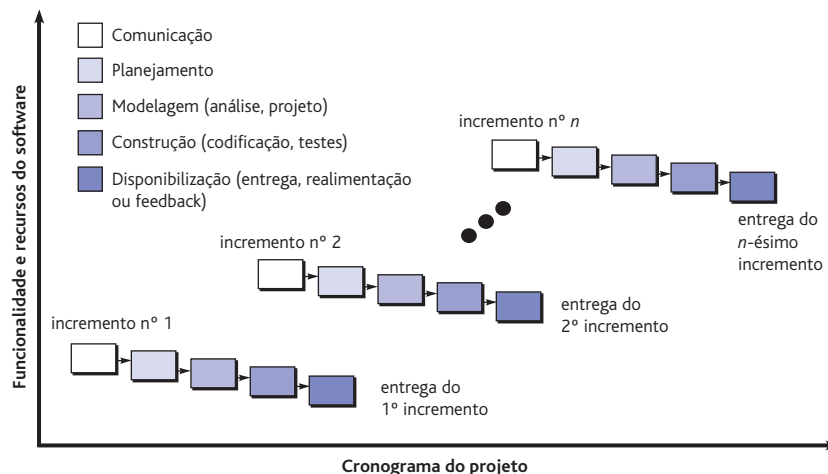


FIGURA 4.3 O modelo incremental.

cado, tornam impossível concluir um produto de software abrangente, porém uma versão limitada tem de ser introduzida para aliviar e/ou atender às pressões comerciais ou da concorrência. Um conjunto do produto essencial ou dos requisitos do sistema está bem compreendido; entretanto, detalhes de extensões do produto ou do sistema ainda devem ser definidos. Em situações como essa ou similares, faz-se necessário um modelo de processo que tenha sido projetado especificamente para desenvolver um produto que cresce e muda.

Modelos evolucionários são iterativos. Apresentam características que possibilitam desenvolver versões cada vez mais completas do software. Nos parágrafos seguintes, são apresentados dois modelos comuns de processos evolucionários.

Prototipação. Frequentemente, o cliente define uma série de objetivos gerais para o software, mas não identifica, detalhadamente, os requisitos para funções e recursos. Em outros casos, o desenvolvedor se encontra inseguro quanto à eficiência de um algoritmo, quanto à adaptabilidade de um sistema operacional ou quanto à forma em que deve ocorrer a interação homem-máquina. Em situações como essas, e em muitas outras, o *paradigma da prototipação* pode ser a melhor abordagem.

Embora a prototipação possa ser utilizada como um modelo de processo isolado (*stand-alone process*), ela é mais comumente utilizada como uma técnica a ser implementada no contexto de qualquer um dos modelos de processo citados neste capítulo. Independentemente da forma como é aplicado, quando os requisitos estão obscuros, o paradigma da prototipação auxilia os envolvidos a compreender melhor o que está para ser construído.

O paradigma da prototipação (Figura 4.4) começa com a comunicação. Faz-se uma reunião com os envolvidos para definir os objetivos gerais do software, identificar os requisitos já conhecidos e esquematizar quais áreas necessitam, obrigatoriamente, de uma definição mais ampla. Uma iteração de prototipação é planejada rapidamente e ocorre a modelagem (na forma de um

"Planeje jogar algo fora. Você vai fazer isso de qualquer maneira. Sua escolha consistirá em decidir se deve tentar ou não vender aos clientes o que foi descartado."

Frederick P. Brooks

Quando seu cliente tiver uma necessidade legítima, mas sem a mínima ideia em relação aos detalhes, faça um protótipo como uma primeira etapa.

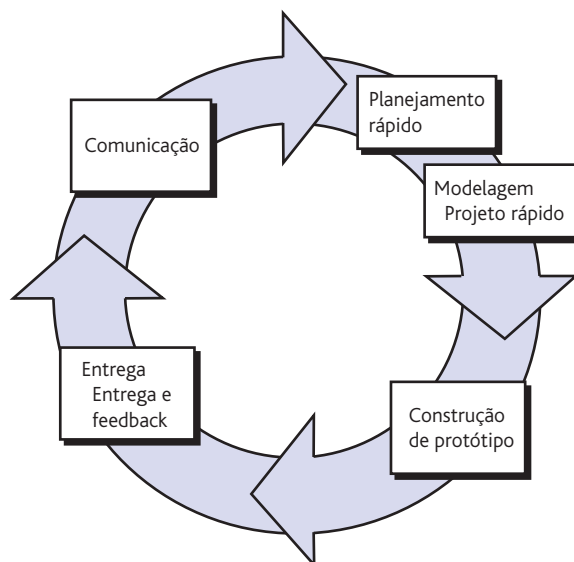


FIGURA 4.4 O paradigma da prototipação.

“projeto rápido”). Um projeto rápido se concentra em uma representação dos aspectos do software que serão visíveis para os usuários (por exemplo, o layout da interface com o usuário ou os formatos de exibição na tela). O projeto rápido leva à construção de um protótipo. O protótipo é entregue e avaliado pelos envolvidos, os quais fornecem feedback que é usado para refinar ainda mais os requisitos. A iteração ocorre conforme se ajusta o protótipo às necessidades de vários envolvidos e, ao mesmo tempo, possibilita a melhor compreensão das necessidades que devem ser atendidas.

Na sua forma ideal, o protótipo atua como um mecanismo para identificar os requisitos do software. Caso seja necessário desenvolver um protótipo operacional, pode-se utilizar partes de programas existentes ou aplicar ferramentas que possibilitem gerar rapidamente tais programas operacionais.

O que fazer com o protótipo quando este já serviu ao propósito descrito anteriormente? Brooks [Bro95] fornece uma resposta:

Na maioria dos projetos, o primeiro sistema dificilmente é útil. Pode ser lento demais, grande demais, estranho em sua utilização ou as três coisas juntas. Não há alternativa, a não ser começar de novo – ressentido, porém mais esperto – e desenvolver uma versão reformulada na qual esses problemas são resolvidos.

O protótipo pode servir como “o primeiro sistema”. Aquele que Brooks recomenda que se jogue fora. Porém, essa pode ser uma visão idealizada. Embora alguns protótipos sejam construídos como “descartáveis”, outros são evolucionários, no sentido de que evoluem lentamente até se transformarem no sistema real.

Tanto os envolvidos quanto os engenheiros de software gostam do paradigma da prototipação. Os usuários podem ter uma ideia prévia do sistema final, ao passo que os desenvolvedores passam a desenvolver algo imediatamente. Entretanto, a prototipação pode ser problemática pelas seguintes razões:

1. Os envolvidos enxergam o que parece ser uma versão operacional do software, ignorando que o protótipo é mantido de forma não organizada e que, na pressa de fazer com que ele se torne operacional, não se considera a qualidade global do software, nem sua manutenção em longo prazo. Quando informados de que o produto deve ser reconstruído para que altos níveis de qualidade possam ser mantidos, os envolvidos protestam e solicitam que “umas poucas correções” sejam feitas para tornar o protótipo um produto operacional. Frequentemente, a gerência do desenvolvimento de software aceita.
2. O engenheiro de software, com frequência, assume compromissos de implementação para conseguir que o protótipo entre em operação rapidamente. Um sistema operacional ou uma linguagem de programação inadequada podem ser utilizados simplesmente porque se encontram à disposição e são conhecidos; um algoritmo ineficiente pode ser implementado simplesmente para demonstrar capacidade. Após um tempo, é possível se acomodar com tais escolhas e esquecer todas as razões pelas quais eram inadequadas. Uma escolha longe da ideal acaba se tornando parte do sistema.

Resista à pressão de transformar um protótipo grosseiro em um produto final. Quase sempre a qualidade fica comprometida.

CASASEGURA

**Seleção de um modelo de processo, parte 1**

Cena: Sala de reuniões da equipe de engenharia de software da CPI Corporation, empresa (fictícia) que fabrica produtos de consumo para uso doméstico e comercial.

Atores: Lee Warren, gerente de engenharia; Doug Miller, gerente de engenharia de software; Jamie Lazar, membro da equipe de software; Vinod Raman, membro da equipe de software; e Ed Robbins, membro da equipe de software.

Conversa:

Lee: Recapitulando. Discuti bastante sobre a linha de produtos *CasaSegura*, da forma como a visualizamos no momento. Sem dúvida, temos muito trabalho a fazer para definir as coisas, mas eu gostaria que vocês comessem a pensar em como vão abordar a parte do software desse projeto.

Doug: Acho que fomos bastante desorganizados em nossa abordagem de software no passado.

Ed: Eu não sei, Doug, nós sempre conseguimos entregar o produto.

Doug: É, mas não sem grande sofrimento, e esse projeto parece ser maior e mais complexo do que qualquer outro que já fizemos.

Jamie: Não parece assim tão difícil, mas eu concordo... a abordagem improvisada que adotamos em projetos anteriores não dará certo neste caso, principalmente se tivermos um cronograma muito apertado.

Doug (sorrindo): Quero ser um pouco mais profissional em nossa abordagem. Participei de um curso rápido na semana passada e aprendi bastante sobre engenharia de software... bom conteúdo. Precisamos de um processo aqui.

Jamie (franzindo a testa): Minha função é desenvolver programas, não ficar mexendo em papéis.

Doug: Dê uma chance antes de dizer não. Eis o que quero dizer. (Doug prossegue descrevendo a metodologia de processo descrita no Capítulo 3 e os modelos de processo prescritivo apresentados até agora.)

Doug: De qualquer forma, parece-me que um modelo linear não é adequado para nós... ele presume que temos todos os requisitos antecipadamente e, conhecendo este lugar, isso é pouco provável.

Vinod: Isso mesmo, e parece orientado demais à tecnologia da informação... provavelmente bom para construir um sistema de controle de estoque ou algo parecido, mas certamente não é adequado para o *CasaSegura*.

Doug: Concordo.

Ed: Essa abordagem de prototipação me parece boa. Bastante parecida com o que fazemos aqui.

Vinod: Isso é um problema. Estou preocupado que ela não nos dê estrutura suficiente.

Doug: Não se preocupe. Temos várias opções e quero que vocês escolham o que for melhor para a equipe e para o projeto.

Embora possam ocorrer problemas, a prototipação pode ser um paradigma eficiente para a engenharia de software. O segredo é definir as regras do jogo logo no início: ou seja, todos os envolvidos devem concordar que o protótipo é construído para servir como um mecanismo para definição de requisitos e depois é descartado (pelo menos em parte); o software final será arquitetado visando à qualidade.

Modelo espiral. Originalmente proposto por Barry Boehm [Boe88], o *modelo espiral* é um modelo de processo de software evolucionário que une a natureza iterativa da prototipação aos aspectos sistemáticos e controlados do modelo cascata. Tem potencial para o rápido desenvolvimento de versões cada vez mais completas do software. Boehm [Boe01a] descreve o modelo da seguinte maneira:

O modelo espiral de desenvolvimento é um gerador de *modelos de processos* dirigidos a *riscos* e é utilizado para guiar a engenharia de sistemas com muito software, que ocorre de forma concorrente e tem vários envolvidos. Possui duas características principais que o distinguem. A primeira consiste em uma estratégia *cíclica* voltada para ampliar, de forma incremental, o grau de definição e a imple-

O modelo espiral pode ser adaptado para ser aplicado ao longo de todo o ciclo de vida de uma aplicação, desde o desenvolvimento de conceitos até sua manutenção.

mentação de um sistema, enquanto diminui o grau de risco do mesmo. A segunda característica é que possui uma série de *marcos de pontos-âncora* para garantir o comprometimento dos envolvidos quanto à busca de soluções de sistema que sejam mutuamente satisfatórias e viáveis.

Com o modelo espiral, o software será desenvolvido em uma série de versões evolucionárias. Nas primeiras iterações, a versão pode consistir em um modelo ou em um protótipo. Já nas iterações posteriores, são produzidas versões cada vez mais completas do sistema que passa pelo processo de engenharia.

O modelo espiral é dividido em um conjunto de atividades metodológicas definidas pela equipe de engenharia de software. A título de ilustração, utilizam-se as atividades metodológicas genéricas discutidas anteriormente.⁴ Cada uma dessas atividades representa um segmento do caminho espiral ilustrado na Figura 4.5. Assim que esse processo evolucionário começa, a equipe de software realiza atividades indicadas por um circuito em torno da espiral, no sentido horário, começando pelo seu centro. Os riscos (Capítulo 35) são levados em conta à medida que cada revolução é realizada. *Marcos de pontos-âncora* – uma combinação de artefatos e condições satisfeitas ao longo do trajeto da espiral – são indicados para cada passagem evolucionária.

O primeiro circuito em volta da espiral pode resultar no desenvolvimento de uma especificação de produto; passagens subsequentes em torno da espiral podem ser usadas para desenvolver um protótipo e, então, progressivamente, versões cada vez mais sofisticadas do software. Cada passagem pela região de planejamento resulta em ajustes no planejamento do projeto. Custo e cronograma são ajustados de acordo com o feedback (a realimentação) obtido do cliente após a entrega. Além disso, o gerente de projeto faz um ajuste no número de iterações planejadas para concluir o software.

Informações úteis sobre o modelo espiral podem ser obtidas em: www.sei.cmu.edu/publications/documents/00.reports/00sr008.html.

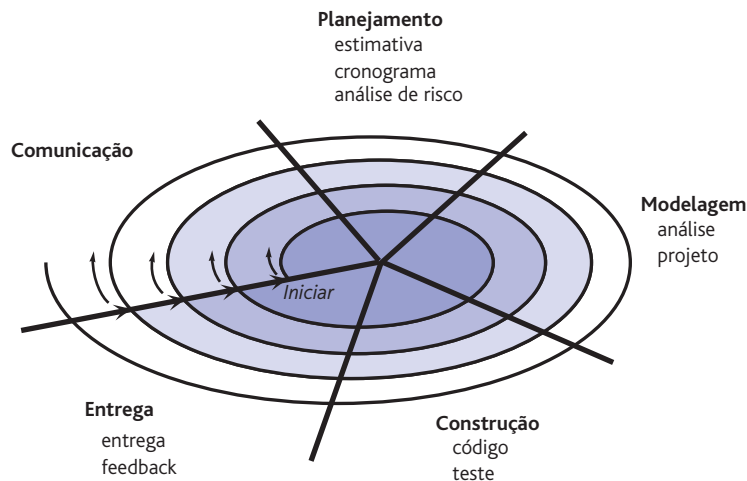


FIGURA 4.5 Modelo espiral típico.

⁴ O modelo espiral discutido nesta seção é uma variação do modelo proposto por Boehm. Para mais informações sobre o modelo espiral original, consulte [Boe88]. Um material mais recente sobre o modelo espiral de Boehm pode ser encontrado em [Boe98].

Diferentemente de outros modelos de processo, que terminam quando o software é entregue, o modelo espiral pode ser adaptado para ser aplicado ao longo da vida do software. Logo, o primeiro circuito em torno da espiral pode representar um “projeto de desenvolvimento de conceitos” que começa no núcleo da espiral e continua por várias iterações⁵ até que o desenvolvimento de conceitos esteja concluído. Se o conceito for desenvolvido para ser um produto final, o processo prossegue na espiral pelas “bordas” e um “novo projeto de desenvolvimento de produto” se inicia. O novo produto evoluirá, passando por iterações em torno da espiral. Mais tarde, uma volta em torno da espiral pode ser usada para representar um “projeto de aperfeiçoamento do produto”. Basicamente, a espiral, quando caracterizada dessa maneira, permanece em operação até que o software seja retirado. Há casos em que o processo fica inativo; porém, toda vez que uma mudança é iniciada, começa no ponto de partida apropriado (por exemplo, aperfeiçoamento do produto).

O modelo espiral é uma abordagem realista para o desenvolvimento de sistemas e de software em larga escala. Como o software evolui à medida que o processo avança, o desenvolvedor e o cliente compreendem e reagem melhor aos riscos em cada nível evolucionário. Esse modelo usa a prototipação como mecanismo de redução de riscos e, mais importante, torna possível a aplicação da prototipação em qualquer estágio do processo evolutivo do produto. Ele mantém a abordagem em etapas, de forma sistemática, sugerida pelo ciclo de vida clássico, mas a incorpora em uma metodologia iterativa que reflete mais realisticamente o mundo real. O modelo espiral exige consideração direta dos riscos técnicos em todos os estágios do projeto e, se aplicado apropriadamente, reduz os riscos antes de se tornarem problemáticos.

Como outros paradigmas, esse modelo não é uma panaceia. Pode ser difícil convencer os clientes (particularmente em situações contratuais) de que a abordagem evolucionária é controlável. Ela exige considerável especialização na avaliação de riscos e depende dessa especialização para seu sucesso. Se um risco muito importante não for descoberto e administrado, sem dúvida ocorrerão problemas.

4.1.4 Modelos concorrentes

O *modelo de desenvolvimento concorrente*, por vezes chamado de *engenharia concorrente*, possibilita à equipe de software representar elementos concorrentes e iterativos de qualquer um dos modelos de processo descritos neste capítulo. Por exemplo, a atividade de modelagem definida para o modelo espiral é realizada invocando uma ou mais destas ações de engenharia de software: prototipação, análise e projeto.⁶

⁵ As setas que apontam para dentro ao longo do eixo, separando a região de *disponibilização* da região de *comunicação*, indicam potencial para iteração local ao longo do mesmo trajeto da espiral.

⁶ Deve-se notar que a análise e o projeto são tarefas complexas que exigem discussão substancial. A Parte II deste livro considera esses tópicos em detalhes.

Se a gerência quiser um desenvolvimento com orçamento fixo (geralmente uma péssima ideia), a espiral pode ser um problema. À medida que cada circuito for completado, o custo do projeto será repetidamente revisado.

“Estou tão perto, mas apenas o amanhã guia o meu caminho.”

**Dave Matthews
Band**

Os planos de projeto devem ser considerados documentos vivos; o progresso deve ser avaliado frequentemente e revisado para levar em conta as alterações.



Seleção de um modelo de processo, parte 2

Cena: Sala de reuniões do grupo de engenharia de software da CPI Corporation, empresa (fictícia) que fabrica produtos de consumo de uso doméstico e comercial.

Atores: Lee Warren, gerente de engenharia; Doug Miller, gerente de engenharia de software; Vinod e Jamie, membros da equipe de engenharia de software.

Conversa: (Doug descreve as opções do processo evolucionário.)

Jamie: Agora estou vendo algo de que gosto. Faz sentido uma abordagem incremental, e eu realmente gosto do fluxo dessa coisa de modelo espiral. Isso tem a ver com a realidade.

Vinod: Concordo. Entregamos um incremento, aprendemos com o feedback do cliente, reformulamos e, então, entrega-

mos outro incremento. Também se encaixa na natureza do produto. Podemos colocar alguma coisa no mercado rapidamente e, depois, acrescentar funcionalidade a cada versão, digo, incremento.

Lee: Espere um pouco. Você disse que reformulamos o plano a cada volta na espiral, Doug? Isso não é tão legal; precisamos de um plano, um cronograma e temos de nos ater a ele.

Doug: Essa linha de pensamento é antiga, Lee. Como o pessoal disse, temos de manter os pés no chão. Acho que é melhor ir ajustando o planejamento à medida que formos aprendendo mais e as mudanças forem sendo solicitadas. É muito mais realista. Para que serve um plano se não para refletir a realidade?

Lee (franzindo a testa): Suponho que esteja certo, porém... a alta direção não vai gostar disso... querem um plano fixo.

Doug (sorrindo): Então, você terá que reeducá-los, meu amigo.

A Figura 4.6 mostra um exemplo de abordagem de modelagem concorrente. Uma atividade – **modelagem** – poderia estar em qualquer um dos estados⁷ observados em qualquer momento determinado. Similarmente, outras atividades, ações ou tarefas (por exemplo, **comunicação** ou **construção**) podem ser representadas de maneira análoga. Todas as atividades de engenharia de software existem simultaneamente, porém estão em diferentes estados.

Por exemplo, no início de um projeto, a atividade de comunicação (não mostrada na figura) completou sua primeira iteração e se encontra no estado **aguardando modificações**. A atividade de modelagem (que se encontrava no estado **nenhum**, enquanto a comunicação inicial era concluída) agora faz uma transição para o estado **em desenvolvimento**. Entretanto, se o cliente indicar que devem ser feitas mudanças nos requisitos, a atividade de modelagem passa do estado **em desenvolvimento** para o estado **aguardando modificações**.

A modelagem concorrente define uma série de eventos que vão disparar transições de um estado para outro para cada uma das atividades, ações ou tarefas da engenharia de software. Por exemplo, durante os estágios iniciais do projeto (uma ação de engenharia de software importante que ocorre durante a atividade de modelagem), uma inconsistência no modelo de requisitos não é descoberta. Isso gera o evento *correção do modelo de análise*, que vai disparar a ação de análise de requisitos, passando do estado **concluído** para o estado **aguardando modificações**.

A modelagem concorrente se aplica a todos os tipos de desenvolvimento de software e fornece uma imagem precisa do estado atual de um projeto. Em vez de limitar as atividades, ações e tarefas da engenharia de software a uma sequência de eventos, ela define uma rede de processos. Cada atividade, ação ou tarefa na rede existe simultaneamente com outras atividades, ações

O modelo concorrente é, com frequência, mais adequado para projetos de engenharia de produto nos quais diferentes equipes de engenharia estão envolvidas.

"Em todo processo há um cliente, pois, sem cliente, um processo deixa de ter sentido."

V. Daniel Hunt

⁷ Um *estado* é algum modo do comportamento observável externamente.

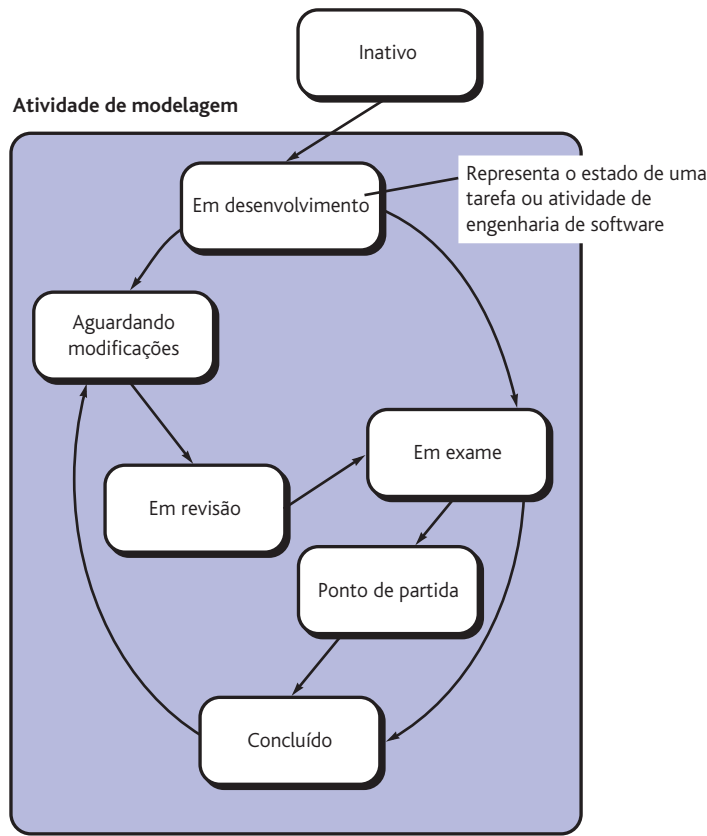


FIGURA 4.6 Um elemento do modelo de processo concorrente.

ou tarefas. Eventos gerados em um ponto da rede de processos disparam transições entre os estados associados a cada atividade.

4.1.5 Um comentário final sobre processos evolucionários

Conforme já mencionado, software moderno é caracterizado por contínuas modificações, prazos muito apertados e por uma ênfase na satisfação do cliente-usuário. Em muitos casos, o tempo de colocação de um produto no mercado é o requisito mais importante a ser gerenciado. Se o momento oportuno de entrada no mercado for perdido, o projeto de software pode ficar sem sentido.⁸

Os modelos de processo evolucionário foram concebidos para lidar com essas questões, mas mesmo assim, como uma classe genérica de modelos de processo, apresentam seus pontos fracos. Esses pontos fracos foram resumidos por Nogueira e seus colegas [Nog00]:

Apesar das inquestionáveis vantagens dos processos de software evolucionários, temos algumas preocupações. A primeira delas é que a prototipação (e outros

⁸ É importante notar, entretanto, que ser o primeiro a chegar ao mercado não é sinônimo de sucesso. Na verdade, muitos produtos de software bem-sucedidos foram o segundo ou até mesmo o terceiro a chegar ao mercado (aprendendo com os erros dos outros que o antecederam).

processos evolucionários mais sofisticados^l traz um problema para o planejamento do projeto, devido ao número incerto, de ciclos necessários para construir o produto...

A segunda é que os processos de software evolucionários não estabelecem a velocidade máxima da evolução. Se as evoluções ocorrerem em uma velocidade excessivamente rápida, sem um período de acomodação, é certo que o processo cairá no caos. Por outro lado, se a velocidade for muito lenta, então a produtividade pode ser afetada...

A terceira é que os processos de software [evolucionários] devem se concentrar mais na flexibilidade e na extensibilidade do que na alta qualidade. Essa afirmação parece assustadora.

Quais são os possíveis pontos fracos dos modelos de processo evolucionário?

Realmente, um processo de software que prioriza flexibilidade, extensibilidade e velocidade de desenvolvimento acima da alta qualidade parece assustador. Ainda assim, essa ideia foi proposta por renomados especialistas em engenharia de software (como, por exemplo, [You95], [Bac97]).

O objetivo dos modelos evolucionários é desenvolver software⁹ de alta qualidade de modo iterativo ou incremental. Entretanto, é possível usar um processo evolucionário para enfatizar a flexibilidade, a extensibilidade e a velocidade de desenvolvimento. O desafio para as equipes de software e seus gerentes será estabelecer um equilíbrio apropriado entre esses parâmetros críticos de projeto e produto e a satisfação dos clientes (o árbitro final da qualidade de um software).

4.2 Modelos de processo especializado

Os modelos de processo especializado incluem muitas das características de um ou mais dos modelos tradicionais apresentados nas seções anteriores. Eles tendem a ser aplicados quando se opta por uma abordagem de engenharia de software especializada ou definida de forma restrita.¹⁰

4.2.1 Desenvolvimento baseado em componentes

Componentes de software comercial de prateleira ou COTS (commercial off-the-shelf), desenvolvidos para serem oferecidos como produtos, disponibilizam a funcionalidade almejada com interfaces bem definidas que permitem que o componente seja integrado ao software a ser desenvolvido. O *modelo de desenvolvimento baseado em componentes* incorpora muitas das características do modelo espiral. É evolucionário por natureza [Nie92], demandando uma abordagem iterativa para a criação de software. O modelo de desenvolvi-

Informações úteis sobre desenvolvimento baseado em componentes podem ser encontradas em: www.cbd-hq.com.

⁹ Neste contexto, a qualidade de software é definida de forma bastante abrangente para englobar não apenas a satisfação dos clientes, mas também uma série de critérios técnicos, discutidos na Parte II deste livro.

¹⁰ Em alguns casos, esses modelos de processo especializado podem ser mais bem definidos como um conjunto de técnicas, ou uma “metodologia”, para alcançar uma meta de desenvolvimento de software específica. Entretanto, eles realmente implicam um processo.

mento baseado em componentes compreende aplicações de componentes de software previamente empacotados.

As atividades de modelagem e construção começam com a identificação de componentes candidatos. Esses componentes podem ser projetados como módulos de software convencionais, como classes orientadas a objeto ou pacotes¹¹ de classes. Seja qual for a tecnologia usada para criar os componentes, o modelo de desenvolvimento baseado em componentes incorpora as seguintes etapas (implementadas usando-se uma abordagem evolucionária):

1. Produtos baseados em componentes disponíveis são pesquisados e avaliados para o campo de aplicação em questão.
2. Itens de integração de componentes são considerados.
3. Uma arquitetura de software é projetada para acomodar os componentes.
4. Os componentes são integrados à arquitetura.
5. Testes completos são realizados para garantir a funcionalidade adequada.

O modelo de desenvolvimento baseado em componentes leva à reutilização de software, e a capacidade de reutilização oferece aos engenheiros de software diversas vantagens mensuráveis, como a redução no tempo do ciclo de desenvolvimento e nos custos do projeto, caso a reutilização de componentes se torne parte da cultura de sua organização. O desenvolvimento baseado em componentes é discutido em mais detalhes no Capítulo 14.

4.2.2 O modelo de métodos formais

O *modelo de métodos formais* inclui um conjunto de atividades que conduzem à especificação matemática formal do software. Eles permitem especificar, desenvolver e verificar um sistema baseado em computador pela aplicação de uma notação matemática rigorosa. Uma variação dessa abordagem, chamada *engenharia de software sala limpa (cleanroom)* [Mil87, Dye92], é aplicada atualmente por algumas empresas de desenvolvimento de software.

O uso de métodos formais (Apêndice 3) durante o desenvolvimento oferece um mecanismo de eliminação de muitos dos problemas difíceis de serem superados com o uso de outros paradigmas de engenharia de software. Ambiguidade, incompletude e inconsistência podem ser descobertas e corrigidas mais facilmente – não por meio de uma revisão local, mas devido à aplicação de análise matemática. Quando são utilizados métodos formais durante o projeto, eles servem como base para verificar a programação e, portanto, possibilitam a descoberta e a correção de erros que, de outra forma, poderiam passar despercebidos.

Embora não seja uma das abordagens mais adotadas, o modelo de métodos formais oferece a promessa de software sem defeitos. Ainda assim, há

¹¹ Conceitos de orientação a objetos são discutidos no Apêndice 2 e usados ao longo da Parte II deste livro. Neste contexto, uma classe engloba um conjunto de dados e os procedimentos que os processam. Pacote de classes é um conjunto de classes relacionadas que operam juntas para alcançar algum resultado final.

Se métodos formais são capazes de demonstrar correção de software, por que não são amplamente utilizados?

motivos para preocupação quanto à sua aplicabilidade em um ambiente de negócios:

- Atualmente, o desenvolvimento de modelos formais consome muito tempo e dinheiro.
- Como poucos desenvolvedores de software possuem formação e experiência necessárias para aplicação dos métodos formais, é necessário treinamento extensivo.
- É difícil usar os modelos como meio de comunicação com clientes tecnicamente despreparados (não sofisticados tecnicamente).

Apesar dessas preocupações, a abordagem dos métodos formais tem conquistado adeptos entre os desenvolvedores de software que precisam desenvolver software com fator crítico de segurança (como, por exemplo, os desenvolvedores de sistemas de voo para aeronaves e equipamentos médicos), bem como entre desenvolvedores que sofreriam pesadas sanções econômicas se ocorressem erros no software.

4.2.3 Desenvolvimento de software orientado a aspectos

Uma ampla variedade de recursos e informações sobre AOP pode ser encontrada em: aosd.net.

Para qualquer processo de software escolhido, os desenvolvedores de software complexo invariavelmente implementam um conjunto de recursos, funções e conteúdo localizados. Essas características de software localizadas são modeladas como componentes (por exemplo, classes orientadas a objetos) e, em seguida, construídas dentro do contexto da arquitetura do sistema. Conforme os sistemas baseados em computadores se tornam mais sofisticados (e complexos), certas preocupações – propriedades exigidas pelo cliente ou áreas de interesse técnico – se estendem por toda a arquitetura. Algumas preocupações são propriedades de alto nível de um sistema (por exemplo, segurança, tolerância a falhas). Outras afetam funções (por exemplo, a aplicação de regras de negócio), enquanto outras são sistêmicas (por exemplo, sincronização de tarefas ou gerenciamento de memória).

A AOSD define “aspectos” que representam preocupações do cliente que transcendem várias funções, recursos e informações do sistema.

Quando as preocupações transcendem várias funções, recursos e informações do sistema, elas costumam ser chamadas de *preocupações transversais*. Os *requisitos de aspectos* definem as preocupações transversais que têm impacto em toda a arquitetura do software. O *desenvolvimento de software orientado a aspectos* (AOSD, *aspect-oriented software development*), também conhecido como *programação orientada a aspectos* (AOP, *aspect-oriented programming*) ou *engenharia de componentes orientada a aspectos* (AOCE, *aspect-oriented component engineering*), é um paradigma de engenharia de software relativamente novo que oferece uma abordagem metodológica e de processos para definir, especificar, projetar e construir *aspectos* – “mecanismos além das sub-rotinas e herança para localizar a expressão de uma preocupação cruzada” [Elr01].

Um processo orientado a aspectos distinto ainda não atingiu a maturidade. Entretanto, é provável que um processo desses adote características tanto dos modelos de processo evolucionário quanto de processo concorrente. O modelo evolucionário é apropriado quando os aspectos são identificados e, então, construídos. A natureza paralela do desenvolvimento concorrente é

FERRAMENTAS DO SOFTWARE

**Gerenciamento de processos**

Objetivo: Ajudar na definição, execução e gerenciamento de modelos de processos prescritivos.

Mecanismos: As ferramentas de gerenciamento de processo possibilitam que uma organização ou equipe de software defina um modelo de processo de software completo (atividades metodológicas, ações, tarefas, pontos de verificação para garantia da qualidade, marcos e artefatos). Além disso, tais ferramentas fornecem um guia, conforme os engenheiros de software realizam o trabalho técnico, e também propiciam um modelo para os gerentes, os quais têm o dever de acompanhar e controlar o processo de software.

Ferramentas representativas:¹²

GDPA, um conjunto de ferramentas para definição de processos de pesquisa, desenvolvido na Universidade de Bremen, na Alemanha (www.informatik.unibremen.de/uniform/gdpa/home.htm), fornece uma grande quantidade de funções de gerenciamento e modelagem de processos.

ALM Studio, desenvolvida pela Kovair Corporation (<http://www.kovair.com/>), engloba um conjunto de ferramentas para definição de processo, gerenciamento de requisitos, solução de problemas, planejamento e acompanhamento de projetos.

ProVision BPMx, desenvolvida pela OpenText (<http://bps.opentext.com/>), é representativa de muitas ferramentas que auxiliam na definição de processo e automação de fluxo de trabalho.

Uma lista valiosa de muitas ferramentas diferentes associadas ao processo de software pode ser encontrada em www.computer.org/portal/web/swbok/html/ch10.

essencial, porque os aspectos são criados de modo independente dos componentes de software localizados, apesar disso, os aspectos têm impacto direto sobre esses componentes. Portanto, é essencial estabelecer comunicação assíncrona entre as atividades de processos de software aplicadas à engenharia e à construção de aspectos e componentes.

Deixamos a discussão detalhada sobre desenvolvimento de software orientado a aspectos para livros dedicados ao assunto. Se tiver mais interesse, consulte [Ras11], [Saf08], [Cla05], [Fil05], [Jac04] e [Gra03].

4.3 O processo unificado

No livro que deu origem ao *Processo Unificado* (PU), Ivar Jacobson, Grady Booch e James Rumbaugh [Jac99] discutem a necessidade de um processo de software “dirigido a casos de uso, centrado na arquitetura, iterativo e incremental”:

Hoje, o software tende em direção a sistemas maiores e mais complexos. Isso se deve, em parte, ao fato de que os computadores se tornam mais potentes a cada ano, aumentando a expectativa dos usuários em relação a eles. Essa tendência também tem sido influenciada pelo uso crescente da Internet para troca de todos os tipos de informação... Nosso apetite por software cada vez mais sofisticado aumenta à medida que tomamos conhecimento de como um produto pode ser aperfeiçoado de uma versão para a seguinte. Queremos software que seja cada

¹² A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

vez mais adaptado às nossas necessidades, mas isso, por sua vez, simplesmente torna o software mais complexo. Em suma, queremos cada vez mais.

Sob certos aspectos, o Processo Unificado é uma tentativa de aproveitar os melhores recursos e características dos modelos tradicionais de processo de software, mas caracterizando-os de modo a implementar muitos dos melhores princípios do desenvolvimento ágil de software (Capítulo 5). O Processo Unificado reconhece a importância da comunicação com o cliente e de métodos racionalizados para descrever a visão do cliente sobre um sistema (os casos de uso).¹³ Ele enfatiza o importante papel da arquitetura de software e “ajuda o arquiteto a manter o foco nas metas corretas, tais como compreensibilidade, confiança em mudanças futuras e reutilização” [Jac99]. Ele sugere um fluxo de processo iterativo e incremental, proporcionando a sensação evolucionária que é essencial no desenvolvimento de software moderno.

4.3.1 Um breve histórico

No início dos anos 1990, James Rumbaugh [Rum91], Grady Booch [Boo94] e Ivar Jacobson [Jac92] começaram a trabalhar em um “método unificado” que combinaria as melhores características de cada um de seus métodos individuais de análise e projeto orientados a objetos e adotariam características adicionais propostas por outros especialistas (por exemplo, [Wir90]) em modelagem orientada a objetos. O resultado foi a UML – uma *linguagem de modelagem unificada* que contém uma notação robusta para a modelagem e o desenvolvimento de sistemas orientados a objetos. Por volta de 1997, a UML tornou-se um padrão da indústria para o desenvolvimento de software orientado a objetos.

A UML é usada ao longo da Parte II deste livro para representar tanto modelos de projeto quanto de requisitos. O Apêndice 1 apresenta um tutorial introdutório para aqueles que não conhecem as regras básicas de notações e de modelagem UML. Uma apresentação completa da UML fica reservada a livros-texto dedicados ao assunto. Livros recomendados estão relacionados no Apêndice 1.

4.3.2 Fases do processo unificado¹⁴

No Capítulo 3, discutimos cinco atividades metodológicas genéricas, expondo que elas poderiam ser usadas para descrever qualquer modelo de processo de software. O Processo Unificado não é exceção. A Figura 4.7 descreve as “fases” do PU e as relaciona com as atividades genéricas discutidas no Capítulo 1 e anteriormente neste capítulo.

¹³ Um *caso de uso* (Capítulo 8) é uma narrativa textual ou modelo que descreve uma função ou recurso de um sistema do ponto de vista do usuário. Ele é escrito pelo usuário e serve como base para a criação de um modelo de análise mais amplo.

¹⁴ O Processo Unificado é, algumas vezes, chamado de *Processo Unificado Racional* (RUP, *Rational Unified Process*) em homenagem à Rational Corporation (posteriormente adquirida pela IBM), uma das primeiras colaboradoras para o desenvolvimento e refinamento do PU e desenvolvedora de ambientes completos (ferramentas e tecnologia) que dão suporte ao processo.

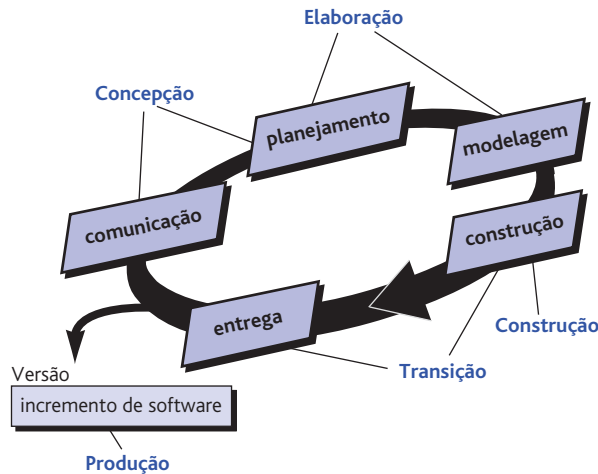


FIGURA 4.7 O Processo Unificado.

A *fase de concepção* do PU inclui a atividade de comunicação com o cliente e a de planejamento. A partir da colaboração com os envolvidos, identificam-se as necessidades de negócio para o software, propõe-se uma arquitetura rudimentar para o sistema e desenvolve-se um planejamento para a natureza iterativa e incremental do projeto decorrente. Requisitos de negócio fundamentais são descritos em um conjunto de casos de uso preliminares (Capítulo 8), descrevendo quais recursos e funções cada categoria principal de usuário deseja. Até esse ponto, a arquitetura nada mais é do que um esquema provisório dos principais subsistemas e das funções e recursos que os compõem. Posteriormente, a arquitetura será refinada e expandida para um conjunto de modelos que representarão diferentes visões do sistema. O planejamento identifica recursos, avalia os principais riscos, define um cronograma e estabelece uma base para as fases que serão aplicadas à medida que o incremento de software for desenvolvido.

A *fase de elaboração* inclui as atividades de comunicação e de modelagem do modelo de processo genérico (Figura 4.7). A elaboração refina e expande os casos de uso preliminares, desenvolvidos como parte da fase de concepção, e amplia a representação arquitetural, incluindo cinco diferentes visões do software: modelo de caso de uso, modelo de análise, modelo de projeto, modelo de implementação e modelo de disponibilização. Em alguns casos, a elaboração gera uma “base de arquitetura executável” [Arl02], consistindo em um sistema executável “de degustação”.¹⁵ Essa base demonstra a viabilidade da arquitetura, mas não oferece todos os recursos e funções necessárias para usar o sistema. Além disso, no auge da fase de elaboração, o plano é revisado cuidadosamente para assegurar que escopo, riscos e datas de entrega permaneçam adequados. Normalmente, as modificações no planejamento são feitas nessa oportunidade.

A *fase de construção* do PU é idêntica à atividade de construção definida para o processo de software genérico. Tendo como entrada o modelo de arqui-

Em seu intento, as *fases* do Processo Unificado são similares às atividades metodológicas genéricas definidas neste livro.

Uma interessante abordagem sobre o PU, no contexto do desenvolvimento ágil, pode ser encontrada em www.ambysoft.com/unifiedprocess/agileUP.html.

¹⁵ É importante observar que a base da arquitetura não é um protótipo, já que não é descartada. Ao contrário, a base ganha corpo durante a fase seguinte do PU.

tetura, a fase de construção desenvolve ou adquire componentes de software; esses componentes farão com que cada caso de uso se torne operacional para os usuários. Para tanto, os modelos de análise e de projeto, iniciados durante a fase de elaboração, são concluídos para refletir a versão final do incremento de software. Então, implementam-se, no código-fonte, todos os recursos e funções necessários e exigidos para o incremento de software (isto é, para a versão). À medida que os componentes são implementados, desenvolvem-se e executam-se testes¹⁶ de unidades para cada um deles. Além disso, realizam-se atividades de integração (montagem de componentes e testes de integração). Os casos de uso são utilizados para se obter um pacote de testes de aceitação, executados antes do início da fase seguinte do PU.

A *fase de transição* do PU abrange os últimos estágios da atividade de construção genérica e a primeira parte da atividade de emprego genérico: entrega e feedback. Entrega-se o software aos usuários para testes beta, e o feedback dos usuários relata defeitos e mudanças necessárias. Além disso, a equipe de software elabora material com as informações de apoio (por exemplo, manuais para o usuário, guias para solução de problemas, procedimentos de instalação) necessárias para o lançamento da versão. Na conclusão da fase de transição, o incremento torna-se uma versão utilizável do software.

A *fase de produção* do PU coincide com a atividade de entrega do processo genérico. Durante essa fase, monitora-se o uso contínuo do software, disponibiliza-se suporte para o ambiente (infraestrutura) operacional, realizam-se e avaliam-se relatórios de defeitos e solicitações de mudanças.

É provável que, ao mesmo tempo em que as fases de construção, transição e produção estejam sendo conduzidas, já se tenha iniciado o incremento de software seguinte. Isso significa que as cinco fases do PU não ocorrem em sequência, mas sim de forma concomitante e escalonada.

Um fluxo de trabalho de engenharia de software é distribuído ao longo de todas as fases do PU. No contexto do PU um *fluxo de trabalho* é análogo a um conjunto de tarefas (descrito no Capítulo 3). Ou seja, um fluxo de trabalho identifica as tarefas exigidas para realizar uma importante ação de engenharia de software e os artefatos produzidos como consequência da conclusão bem-sucedida das tarefas. Deve-se notar que nem toda tarefa identificada para um fluxo de trabalho do PU é conduzida em todos os projetos de software. A equipe adapta o processo (ações, tarefas, subtarefas e artefatos) para ficar de acordo com suas necessidades.

4.4 Modelos de processo pessoal e de equipe

"Pessoas bem-sucedidas simplesmente desenvolveram o hábito de realizar coisas que as fracassadas não vão fazer."

Dexter Yager

O melhor processo de software é aquele próximo às pessoas que realizarão o trabalho. Se um modelo de processo de software for desenvolvido em nível corporativo ou organizacional, ele somente será eficaz se for passível de grandes adaptações, a fim de atender às necessidades da equipe de projeto (aquela que está efetivamente realizando o trabalho de engenharia de software). Em

¹⁶ Uma discussão abrangente sobre testes de software (inclusive *testes de unidades*) é apresentada nos Capítulos 22 a 26.

um cenário ideal, seria desenvolvido um processo que melhor se adequasse às suas necessidades e, simultaneamente, atendesse às necessidades mais amplas da equipe e da empresa. Outra opção é a equipe criar seu próprio processo para atender, ao mesmo tempo, às necessidades mais específicas dos indivíduos e às necessidades mais amplas da organização. Watts Humphrey ([Hum05] e [Hum00]) diz que é possível criar um “processo de software pessoal” e/ou um “processo de software de equipe”. Ambos exigem trabalho árduo, treinamento e coordenação, mas são alcançáveis.¹⁷

4.4.1 Processo de Software Pessoal

Todo desenvolvedor utiliza algum processo para construir software. Esse processo pode ser nebuloso ou específico; pode mudar diariamente; não ser eficiente, efetivo ou bem-sucedido; porém, realmente existe um “processo”. Watts Humphrey [Hum05] sugere que, para modificar um processo pessoal ineficaz, um profissional deve passar por quatro fases, cada uma exigindo treinamento e orquestração cuidadosa. O *Processo de Software Pessoal* (PSP, *Personal Software Process*) enfatiza a medição pessoal, tanto do artefato de software gerado quanto da qualidade resultante dele. Além disso, responsabiliza o profissional pelo planejamento de projetos (por exemplo, estimativa de custos e cronograma) e lhe permite controlar a qualidade de todos os artefatos de software desenvolvidos. O modelo PSP define cinco atividades estruturais:

Planejamento. Isola os requisitos e desenvolve as estimativas de tamanho e recursos. Além disso, faz-se uma estimativa dos defeitos (o número de defeitos estimado para o trabalho). Registram-se todas as métricas em formulários ou planilhas. Por último, identificam-se as tarefas de desenvolvimento e faz-se um cronograma para o projeto.

Projeto de alto nível. Especificações externas são desenvolvidas para cada componente a ser construído e um projeto de componentes é elaborado. Quando há incerteza, constroem-se protótipos. Todos os problemas são registrados e monitorados.

Revisão de projeto de alto nível. Métodos de verificação formais (Apêndice 3) são aplicados para revelar erros no projeto. São mantidas métricas para resultados de trabalho e tarefas importantes.

Desenvolvimento. O projeto em nível de componentes é refinado e revisado. Um código é gerado, revisado, compilado e testado. São mantidas métricas para resultados de trabalho e tarefas importantes.

Autópsia. Usando as medidas e métricas coletadas (trata-se de um volume de dados substancial que deve ser analisado estatisticamente), é determinada a eficácia do processo. Medidas e métricas devem guiar as mudanças no processo, de modo a melhorar sua eficiência.

Uma grande quantidade de recursos para PSP pode ser encontrada em <http://www.sei.cmu.edu/tsp/tools/academic/>.

Quais atividades metodológicas são utilizadas durante o PSP?

O PSP enfatiza a necessidade de registrar e analisar tipos de erros cometidos para que se possa elaborar estratégias para eliminá-los.

¹⁷ Vale notar que os defensores do desenvolvimento ágil de software (Capítulo 5) também afirmam que o processo deve ficar próximo à equipe. Eles propõem um método alternativo para conseguir isso.

No PSP é enfatizada a necessidade de identificar erros precocemente e, tão importante quanto, compreender os tipos de erro que provavelmente ocorrerão. Isso é obtido por meio de uma rigorosa atividade de avaliação em todos os artefatos de software gerados.

O PSP é uma abordagem disciplinada e baseada em métricas para a engenharia de software que pode causar um choque cultural em muitos profissionais. Entretanto, quando apresentado de forma apropriada aos engenheiros de software [Hum96], a melhoria resultante na produtividade da engenharia e na qualidade de software é significativa [Fer97]. Apesar disso, não foi amplamente adotado pelo setor. Os motivos, infelizmente, têm mais a ver com a natureza humana e com a inércia organizacional do que com os pontos fortes e fracos da abordagem PSP. Esse processo é intelectualmente desafiador e exige um nível de comprometimento (por parte dos profissionais e de seus gerentes) que nem sempre é possível alcançar. O período de treinamento é relativamente longo; e os custos de treinamento, altos. O nível de medição exigido é culturalmente difícil para muitos profissionais da área de software.

Pode ser utilizado como um processo de software eficaz no nível pessoal? A resposta é um inequívoco “sim”. Porém, mesmo se não adotado em sua totalidade, muitos dos conceitos de aperfeiçoamento do processo pessoal que o PSP introduz são importantes e vale a pena aprendê-los.

4.4.2 Processo de software de equipe

Como muitos projetos de software para nível industrial são conduzidos por uma equipe de profissionais, Watts Humphrey estendeu as lições aprendidas com a introdução do PSP e propôs um *Processo de Software de Equipe* (TSP, *Team Software Process*). O objetivo do TSP é criar uma equipe de projetos “autodirigida” que se organize por si mesma para produzir software de alta qualidade. Humphrey [Hum98] define os seguintes objetivos para o TSP:

- Criar equipes autodirigidas que planejem e acompanhem seu próprio trabalho, estabeleçam metas e sejam proprietárias de seus processos e planos. As equipes poderão ser puras ou equipes de produto integradas (IPTs, integrated product teams) com cerca de 3 a 20 engenheiros.
- Mostrar aos gerentes como treinar e motivar suas equipes e como ajudá-las a manter alto desempenho.
- Acelerar o aperfeiçoamento dos processos de software, tornando o comportamento CMM¹⁸ nível 5 algo normal e esperado.
- Fornecer orientação para melhorias a organizações com elevado grau de maturidade.
- Facilitar o ensino universitário de habilidades de trabalho em equipe de nível industrial.

Uma equipe autodirigida tem um bom entendimento de suas metas e objetivos globais; define papéis e responsabilidades para cada um dos membros; monitora dados quantitativos do projeto (produtividade e qualidade); identi-

Informações sobre a formação de equipes com alto desempenho empregando TSP e PSP podem ser obtidas em www.sei.cmu.edu/tsp/.

Para formar uma equipe autodirigida, é preciso haver boa colaboração interna e boa comunicação externa.

¹⁸ O Modelo de Maturidade de Capacidade (CMM, Capability Maturity Model), uma medida da eficiência de um processo de software, é discutido no Capítulo 37.

fica um processo de equipe que seja apropriado para o projeto em questão e uma estratégia para implementação do processo; define padrões locais que sejam aplicáveis ao trabalho de engenharia da equipe; avalia continuamente os riscos e reage a eles; e, por fim, acompanha, gerencia e gera relatórios sobre a situação do projeto.

O TSP define as seguintes atividades metodológicas: *lançamento do projeto*, *projeto de alto nível*, *implementação*, *integração e testes* e *autópsia*. Assim como seus equivalentes no PSP (note que a terminologia é ligeiramente diferente), essas atividades capacitam a equipe a planejar, projetar e construir software de maneira disciplinada, ao mesmo tempo em que mede quantitativamente o processo e o produto. A autópsia representa o estágio para melhorias dos processos.

Esse processo faz uso de uma grande variedade de roteiros (*scripts*), formulários e padrões que servem para orientar os membros da equipe em seu trabalho. Os roteiros definem atividades de processo específicas (isto é, lançamento do projeto, projeto, implementação, integração e testes do sistema, autópsia) e outras funções de trabalho mais detalhadas (por exemplo, planejamento do desenvolvimento, desenvolvimento de requisitos, gerenciamento das configurações de software, teste de unidade) que fazem parte do processo de equipe.

O TSP reconhece que as melhores equipes de software são autogeridas.¹⁹ Seus membros estabelecem os objetivos do projeto, adaptam o processo para atender suas necessidades, controlam o cronograma e, através de medições e análise das métricas coletadas, trabalham continuamente para aperfeiçoar sua abordagem à engenharia de software.

Assim como o PSP, o TSP é uma abordagem rigorosa da engenharia de software que fornece benefícios distintos e quantificáveis para a produtividade e para a qualidade. A equipe deve se comprometer totalmente com o processo e deve passar por treinamento consciente para assegurar que a abordagem seja aplicada adequadamente.

Os roteiros (*scripts*) do TSP definem os elementos e as atividades realizadas no transcorrer do processo.

4.5 Tecnologia de processos

Um ou mais dos modelos de processo discutidos nas seções anteriores devem ser adaptados para emprego por uma equipe de software. Para tanto, foram desenvolvidas *ferramentas de tecnologia de processos*, com o objetivo de auxiliar organizações de software a analisar seus processos atuais, organizar tarefas de trabalho, controlar e monitorar o progresso, bem como administrar a qualidade técnica.

As ferramentas de tecnologia de processos permitem que uma empresa de software construa um modelo automatizado da metodologia de processos, dos conjuntos de tarefas e das atividades de apoio, discutidos no Capítulo 3. O modelo, normalmente representado como uma rede, pode, então, ser analisado para determinar o fluxo de trabalho típico e examinar estruturas de processo alternativas que possam levar à redução de custos e tempo de desenvolvimento.

¹⁹ No Capítulo 5, discutimos a importância das equipes “auto-organizadas” como um elemento-chave no desenvolvimento de software ágil.

FERRAMENTAS DO SOFTWARE



Ferramentas de modelagem de processos

Objetivo: Quando uma organização trabalha para aprimorar um processo de negócio (ou de software), ela precisa, primeiramente, compreendê-lo. As ferramentas de modelagem de processos (também chamadas ferramentas de *tecnologia de processos* ou ferramentas de *gerenciamento de processos*) são usadas para representar elementos-chave de um processo a fim de que possa ser mais bem compreendido. Essas ferramentas podem também oferecer “links” para descrições de processos, ajudando os envolvidos no processo a compreender as ações e tarefas necessárias para realizá-lo. As ferramentas de modelagem de processos fornecem links para outras ferramentas que oferecem suporte para atividades de processos definidas.

Mecanismos: As ferramentas nesta categoria permitem a uma equipe de desenvolvimento definir os elementos de um modelo único de processo (ações, tarefas, artefatos, pontos

de garantia da qualidade de software), dar orientação detalhada sobre o conteúdo ou descrição de cada elemento de um processo e, então, gerenciar o processo conforme ele for conduzido. Em alguns casos, as ferramentas de tecnologia de processos incorporam tarefas padronizadas de gerenciamento de projeto, como estimativa de custos, cronograma, acompanhamento e controle.

Ferramentas representativas:²⁰

Igrafx Process Tools – ferramentas que capacitam uma equipe a mapear, medir e modelar o processo de software (<http://www.igrafx.com/>)

Adeptia BPM Server – projetada para gerenciar, automatizar e otimizar processos de negócio (www.adeptia.com)

ALM Studio Suite – conjunto de seis ferramentas com forte ênfase no gerenciamento das atividades de comunicação e modelagem (<http://www.kovair.com/>)

Uma vez criado um processo aceitável, outras ferramentas de tecnologia de processos poderão ser usadas para alocar, monitorar e até mesmo controlar todas as atividades, ações e tarefas de engenharia de software definidas como parte do modelo de processo. Cada membro da equipe poderá usar essas ferramentas para desenvolver uma lista de controle das tarefas a serem realizadas, dos artefatos de software a serem gerados e das atividades de garantia da qualidade a serem realizadas. A ferramenta de tecnologia de processos também pode ser usada para coordenar o uso de outras ferramentas de engenharia de software apropriadas para determinada tarefa.

4.6 Produto e processo

Se o processo for fraco, certamente o produto final sofrerá consequências. Porém, uma confiança excessiva e obsessiva no processo é igualmente perigosa. Em um breve artigo, escrito muitos anos atrás, Margaret Davis [Dav95a] tece comentários atemporais sobre a dualidade produto e processo:

A cada 10 anos (com uma margem de erro de cinco anos), aproximadamente, a comunidade de software redefine “o problema”, mudando seu foco de itens do produto para itens do processo. Assim, adotamos linguagens de programação estruturada (produto), seguidas por métodos de análise estruturada (processo), seguidos pelo encapsulamento de dados (produto), seguido pela ênfase atual no Modelo de Maturidade de Capacitação de Desenvolvimento de Software (processo), do Software Engineering Institute [seguido por métodos orientados a objetos, seguido pelo desenvolvimento de software ágil.

²⁰ A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

Embora a tendência natural de um pêndulo seja a de repousar em um ponto intermediário entre dois extremos, o foco da comunidade de software muda constantemente, pois uma nova força é aplicada quando a última oscilação falha. Essas oscilações causam danos para si mesmos e para o ambiente externo, confundindo o profissional típico de software, mudando radicalmente o que significava desempenhar bem seu trabalho. Essas oscilações também não resolvem “o problema”, pois estão fadadas ao insucesso, enquanto produto e processo forem tratados como formando uma dicotomia (divisão de um conceito em dois elementos, em geral, contrários) em vez de uma dualidade (coexistência de dois princípios).

Na comunidade científica, há precedentes da tendência de adotar noções de dualidade quando, nas observações, as contradições não podem ser explicadas completamente nem por uma nem por outra teoria que competem entre si. A natureza dual da luz, parecendo ser simultaneamente partícula e onda, foi aceita desde os anos 1920, quando Louis de Broglie a propôs. Pelas observações feitas dos artefatos de software e de seu desenvolvimento, fica demonstrada a existência de uma dualidade fundamental entre produto e processo. Jamais poderemos destrinchar ou compreender o artefato completo, seu contexto, uso, significado e valor se o enxergarmos apenas como um processo ou como um produto.

Todas as atividades humanas podem ser um processo, mas todos se sentem valorizados quando tais atividades se tornam uma representação ou um exemplo, sendo utilizadas ou apreciadas por mais de uma pessoa, repetidamente, ou então utilizadas num contexto não imaginado. Ou seja, extraímos sentimentos de satisfação na reutilização de nossos produtos, seja por nós mesmos, seja por outros.

Assim, enquanto a assimilação rápida das metas de reutilização no desenvolvimento de software aumenta potencialmente a satisfação dos profissionais de software, também aumenta a urgência da aceitação da dualidade produto e processo. Enxergar um artefato reutilizável apenas como um produto ou apenas como um processo obscurece o contexto e as maneiras de usá-lo – ou obscurece o fato de que cada uso resulta em um produto que, por sua vez, será utilizado como entrada em alguma outra atividade de desenvolvimento de software. Adotar uma dessas visões em detrimento da outra reduz dramaticamente as oportunidades de reutilização e, portanto, perde-se a oportunidade de aumentar a satisfação no trabalho.

As pessoas obtêm satisfação tanto do processo criativo quanto do produto final. Um artista sente prazer tanto com suas pinceladas quanto com o resultado final de seu quadro. Um escritor sente prazer tanto com a procura da metáfora apropriada quanto com o livro finalizado. Como profissional de software criativo, você também deve extrair tanta satisfação do processo quanto do produto final. A dualidade produto e processo é um elemento importante para manter pessoas criativas engajadas, à medida que a engenharia de software continua a evoluir.

4.7 Resumo

Os modelos de processo prescritivos são aplicados há anos, na tentativa de organizar e estruturar o desenvolvimento de software. Cada um desses modelos sugere um fluxo de processo um pouco diferente, mas todos realizam o

mesmo conjunto de atividades metodológicas genéricas: comunicação, planejamento, modelagem, construção e disponibilização.

Os modelos de processo sequenciais, como os modelos cascata e V, são os mais antigos paradigmas da engenharia de software. Eles sugerem um fluxo de processos linear que, muitas vezes, é inadequado para os sistemas modernos (por exemplo, alterações contínuas, sistemas em evolução, prazos apertados). Entretanto, eles têm aplicabilidade em situações em que os requisitos são bem definidos e estáveis.

Modelos de processo incremental são iterativos por natureza e produzem rapidamente versões operacionais do software. Modelos de processos evolucionários reconhecem a natureza iterativa e incremental da maioria dos projetos de engenharia de software e são projetados para se adequar às mudanças. Esses modelos, como prototipação e o modelo espiral, produzem rapidamente artefatos de software incrementais (ou versões operacionais do software). Podem ser adotados para serem aplicados por todas as atividades de engenharia de software – desde o desenvolvimento de conceitos até a manutenção do sistema em longo prazo.

O modelo de processo concorrente permite que uma equipe de software represente elementos iterativos e concorrentes de qualquer modelo de processo. Modelos especializados incluem o modelo baseado em componentes (que enfatiza a montagem e a reutilização de componentes), o modelo de métodos formais (que estimula uma abordagem matemática para o desenvolvimento e a verificação de software) e o modelo orientado a aspectos (que considera preocupações transversais que se estendem por toda a arquitetura do sistema). O Processo Unificado é um processo de software “dirigido a casos de uso, centrado na arquitetura, iterativo e incremental”, desenvolvido como uma metodologia para os métodos e ferramentas da UML.

Foram propostos modelos pessoais e em equipe para o processo de software. Ambos enfatizam medida, planejamento e autodireção como ingredientes importantes para um processo de software bem-sucedido.

Problemas e pontos a ponderar

- 4.1. Dê três exemplos de projetos de software que seriam suscetíveis ao modelo cascata. Seja específico.
- 4.2. Dê três exemplos de projetos de software que seriam suscetíveis ao modelo de prototipação. Seja específico.
- 4.3. Quais adaptações de processo seriam necessárias caso o protótipo fosse se transformar em um sistema ou produto a ser entregue?
- 4.4. Dê três exemplos de projetos de software que seriam suscetíveis ao modelo incremental. Seja específico.
- 4.5. À medida que se desloca para fora ao longo do fluxo de processo em espiral, o que pode ser dito em relação ao software que está sendo desenvolvido ou sofrendo manutenção?
- 4.6. É possível combinar modelos de processo? Em caso positivo, dê um exemplo.

- 4.7. O modelo de processo concorrente define um conjunto de “estados”. Descreva, com suas próprias palavras, o que esses estados representam e, em seguida, indique como entram em cena no modelo de processos concorrentes.
- 4.8. Quais são as vantagens e desvantagens de desenvolver software cuja qualidade é “boa o suficiente”? Ou seja, o que acontece quando enfatizamos a velocidade de desenvolvimento em detrimento da qualidade do produto?
- 4.9. Dê três exemplos de projetos de software que seriam suscetíveis ao modelo baseado em componentes. Seja específico.
- 4.10. É possível provar que um componente de software e até mesmo um programa inteiro está correto. Então, por que todo mundo não faz isso?
- 4.11. Processo Unificado e UML são a mesma coisa? Justifique sua resposta.

Leituras e fontes de informação complementares

A maioria dos livros sobre engenharia de software mencionados na seção *Leituras e fontes de informação complementares* do Capítulo 2 trata dos modelos de processo prescritivo com algum detalhe.

Cynkovic e Larsson (*Building Reliable Component-Based Systems*, Addison-Wesley, 2002) e Heineman e Council (*Component-Based Software Engineering*, Addison-Wesley, 2001) descrevem o processo exigido para implementar sistemas baseados em componentes. Jacobson e Ng (*Aspect-Oriented Software Development with Use Cases*, Addison-Wesley, 2005) e Filman e seus colegas (*Aspect-Oriented Software Development*, Addison-Wesley, 2004) discutem a natureza única do processo orientado a aspectos. Monin e Hinchey (*Understanding Formal Methods*, Springer, 2003) apresentam uma interessante introdução e Boca e seus colegas (*Formal Methods*, Springer, 2009) discutem o estado atual e novos rumos.

Livros de Kenett e Baker (*Software Process Quality: Management and Control*, Marcel Dekker, 1999) e Chrissis, Konrad e Shrum (*CMMI for Development: Guidelines for Process Integration and Product Improvement*, 3ª ed., Addison-Wesley, 2011) consideram como o gerenciamento da qualidade e o projeto de processos estão intimamente ligados.

Além do livro seminal de Jacobson, Rumbaugh e Booch sobre o Processo Unificado [Jac99], livros como os de Shuja e Krebs (*IBM Rational Unified Process Reference and Certification Guide*, IBM Press, 2008), Arlow e Neustadt (*UML 2 and the Unified Process*, Addison-Wesley, 2005), Kroll e Kruchten (*The Rational Unified Process Made Easy*, Addison-Wesley, 2003) e Farve (*UML and the Unified Process*, IRM Press, 2003) fornecem excelentes informações complementares. Gibbs (*Project Management with the IBM Rational Unified Process*, IBM Press, 2006) fala sobre o gerenciamento de projetos no contexto do PU. Dennis, Wixom e Tegarden (*Systems Analysis and Design with UML*, 4ª ed., Wiley, 2012) aborda programação e modelagem de processo de negócio relacionado ao PU.

Uma ampla gama de fontes de informação sobre modelos de processo de software se encontra à disposição na Internet. Uma lista atualizada de referências relevantes (em inglês) para o processo para o software pode ser encontrada no site: www.mhhe.com/pressman.

5

Desenvolvimento ágil

Conceitos-chave

agilidade.....	68
Agile Alliance.....	70
custo da alteração	68
Método de Desenvolvimento de Sistemas Dinâmicos (DSDM, Dynamic Systems Development Method) ..	79
princípios da agilidade ...	70
processo ágil	69
Processo Unificado Ágil ..	82
testes de aceitação.....	75

Em 2001, Kent Beck e outros 16 renomados desenvolvedores, autores e consultores da área de software [Bec01] (batizados de “Agile Alliance” – “Aliança dos Ágeis”) assinaram o “Manifesto para o Desenvolvimento Ágil de Software” (“Manifesto for Agile Software Development”). Ele declarava:

Ao desenvolver e ajudar outros a desenvolver software, desvendamos formas melhores de desenvolvimento. Por meio deste trabalho passamos a valorizar:

Indivíduos e interações acima de processos e ferramentas

Software operacional acima de documentação completa

Colaboração dos clientes acima de negociação contratual

Respostas a mudanças acima de seguir um plano

Ou seja, embora haja valor nos itens à direita, valorizaremos os da esquerda mais ainda.

PANORAMA

desenvolvimento. A filosofia defende a satisfação do cliente e a entrega incremental antecipada; equipes de projeto pequenas e altamente motivadas; métodos informais; artefatos de engenharia de software mínimos; e, acima de tudo, simplicidade no desenvolvimento geral. Os princípios de desenvolvimento priorizam a entrega mais do que a análise e o projeto (embora essas atividades não sejam desencorajadas); também priorizam a comunicação ativa e contínua entre desenvolvedores e clientes.

Quem realiza? Os engenheiros de software e outros envolvidos no projeto (gerentes, clientes, usuários) trabalham conjuntamente em uma equipe ágil – uma equipe que se auto-organiza e que controla seu próprio destino. Uma equipe ágil acelera a comunicação e a colaboração entre todos os participantes (que estão ao seu serviço).

Por que é importante? O ambiente moderno dos sistemas e dos produtos da área é acelerado e está em constante mudança. A engenharia de software ágil constitui uma alternativa

O que é? A engenharia de software ágil combina filosofia com um conjunto de princípios de desenvolvimento.

razoável para a engenharia convencional voltada para certas classes de software e para certos tipos de projetos. Ela tem se mostrado capaz de entregar sistemas corretos rapidamente.

Quais são as etapas envolvidas? O desenvolvimento ágil poderia ser mais bem denominado “engenharia de software flexível”. As atividades metodológicas básicas – comunicação, planejamento, modelagem, construção e entrega – permanecem. Entretanto, elas se transformam em um conjunto de tarefas mínimas que impulsiona a equipe para o desenvolvimento e para a entrega (pode-se levantar a questão de que isso é feito em detrimento da análise do problema e do projeto de soluções).

Qual é o artefato? Tanto o cliente quanto o engenheiro têm o mesmo parecer: o único artefato realmente importante consiste em um “incremento de software” operacional que seja entregue, adequadamente, na data combinada.

Como garantir que o trabalho foi realizado corretamente? Se a equipe ágil concorda que o processo funciona e essa equipe produz incrementos de software passíveis de entrega e que satisfaçam o cliente, então, o trabalho está correto.

Um manifesto normalmente é associado a um movimento político emergente: ataca a velha guarda e sugere uma mudança revolucionária (espera-se que para melhor). De certa forma, é exatamente disso que trata o desenvolvimento ágil.

Embora as ideias fundamentais que norteiam o desenvolvimento ágil tenham estado conosco por muitos anos, apenas há menos de duas décadas se consolidaram como um “movimento”. Em essência, métodos ágeis¹ se desenvolveram em um esforço para sanar fraquezas reais e perceptíveis da engenharia de software convencional. O desenvolvimento ágil oferece benefícios importantes; no entanto, não é indicado para todos os projetos, produtos, pessoas e situações. Também *não* é a antítese da prática de engenharia de software confiável e pode ser aplicado como uma filosofia geral para todos os trabalhos de software.

Na economia moderna, frequentemente é difícil ou impossível prever como um sistema computacional (por exemplo, um aplicativo móvel) vai evoluir com o tempo. As condições de mercado mudam rapidamente, as necessidades dos usuários se alteram, e novas ameaças competitivas surgem sem aviso. Em muitas situações, não se conseguirá definir os requisitos completamente antes que se inicie o projeto. É preciso ser ágil o suficiente para dar uma resposta a um ambiente de negócios fluido.

Fluidez implica mudança, e mudança é cara – particularmente se for sem controle e mal gerenciada. Uma das características mais convincentes da metodologia ágil é sua habilidade de reduzir os custos da mudança no processo de software.

Será que isso significa que o reconhecimento dos desafios apresentados pela realidade moderna faz que valiosos princípios, conceitos, métodos e ferramentas da engenharia de software sejam descartados? Absolutamente não! Como todas as disciplinas de engenharia, a engenharia de software continua a evoluir. Ela pode ser adaptada facilmente aos desafios apresentados pela demanda por agilidade.

Em um texto instigante sobre desenvolvimento de software ágil, Alistair Cockburn [Coc02] argumenta que o modelo de processo prescritivo, apresentado no Capítulo 4, tem uma falha essencial: *esquece-se das fraquezas das pessoas que desenvolvem o software*. Os engenheiros de software não são robôs. Eles apresentam grande variação nos estilos de trabalho; diferenças significativas no nível de habilidade, criatividade, organização, consistência e espontaneidade. Alguns se comunicam bem na forma escrita, outros não. Cockburn afirma que os modelos de processos podem “lidar com as fraquezas comuns das pessoas com disciplina e/ou tolerância” e que a maioria dos modelos de processos prescritivos opta por disciplina. Segundo ele: “Como a coerência nas ações é uma fraqueza humana, as metodologias com disciplina elevada são frágeis”.

Para que funcionem, os modelos de processos devem fornecer um mecanismo realista que estimule a disciplina necessária ou, então, devem ter características que apresentem “tolerância” com as pessoas que realizam trabalhos de engenharia de software. Invariavelmente, práticas tolerantes são mais

Desenvolvimento ágil não significa que nenhum documento é criado; significa que apenas os documentos que vão ser consultados mais adiante no processo de desenvolvimento são criados.

“Agilidade: 1. Todo o resto: 0.”

Tom DeMarco

¹ Os métodos ágeis são, algumas vezes, conhecidos como *métodos light* ou *métodos enxutos* (*lean methods*).

facilmente adotadas e sustentadas pelas pessoas envolvidas, porém (como o próprio Cockburn admite) podem ser menos produtivas. Como a maioria das coisas na vida, deve-se considerar os prós e os contras.

5.1 O que é agilidade?

Afinal, o que é agilidade no contexto da engenharia de software? Ivar Jacobson [Jac02a] apresenta uma discussão útil:

Atualmente, *agilidade* se tornou a palavra da moda quando se descreve um processo de software moderno. Todo mundo é ágil. Uma equipe ágil é aquela rápida e capaz de responder de modo adequado às mudanças. Mudança tem tudo a ver com desenvolvimento de software. Mudança no software que está sendo criado, mudança nos membros da equipe, mudança devido a novas tecnologias, mudanças de todos os tipos que poderão ter um impacto no produto que está em construção ou no projeto que cria o produto. Suporte à mudança deve ser incorporado a tudo o que fazemos em software, algo que abraçamos porque é o coração e a alma do software. Uma equipe ágil reconhece que o software é desenvolvido por indivíduos trabalhando em equipes e que as habilidades dessas pessoas, suas capacidades em colaborar estão no cerne do sucesso do projeto.

De acordo com Jacobson, a difusão da mudança é o principal condutor para a agilidade. Os engenheiros de software devem ser rápidos, caso queiram assimilar as rápidas mudanças que Jacobson descreve.

Entretanto, agilidade é mais do que uma resposta à mudança. Ela abrange também a filosofia proposta no manifesto citado no início deste capítulo. Ela incentiva a estruturação e as atitudes em equipe que tornam a comunicação mais fácil (entre membros da equipe, entre o pessoal ligado à tecnologia e o pessoal da área comercial, entre os engenheiros de software e seus gerentes). Enfatiza a entrega rápida do software operacional e diminui a importância dos artefatos intermediários (nem sempre um bom negócio); aceita o cliente como parte da equipe de desenvolvimento e trabalha para eliminar a atitude de “nós e eles” que continua a impregnar muitos projetos de software; reconhece que o planejamento em um mundo incerto tem seus limites e que o plano (roteiro) de projeto deve ser flexível.

A agilidade pode ser aplicada a qualquer processo de software. Entretanto, para alcançá-la, é essencial que o processo seja projetado de modo que a equipe possa adaptar e alinhar (racionalizar) tarefas; possa conduzir o planejamento, compreendendo a fluidez de uma metodologia de desenvolvimento ágil; possa eliminar tudo, exceto os artefatos essenciais, conservando-os enxutos; e enfatize a estratégia de entrega incremental, conseguindo entregar ao cliente, o mais rapidamente possível, o software operacional para o tipo de produto e ambiente operacional.

5.2 Agilidade e o custo das mudanças

O pensamento convencional em desenvolvimento de software (baseada em décadas de experiência) é que os custos de mudanças aumentam de forma não linear conforme o projeto avança (Figura 5.1, curva em preto contínua). É relativamente

Não cometa o erro de supor que a agilidade lhe dará licença para abreviar soluções. Processo é um requisito, e disciplina é essencial.

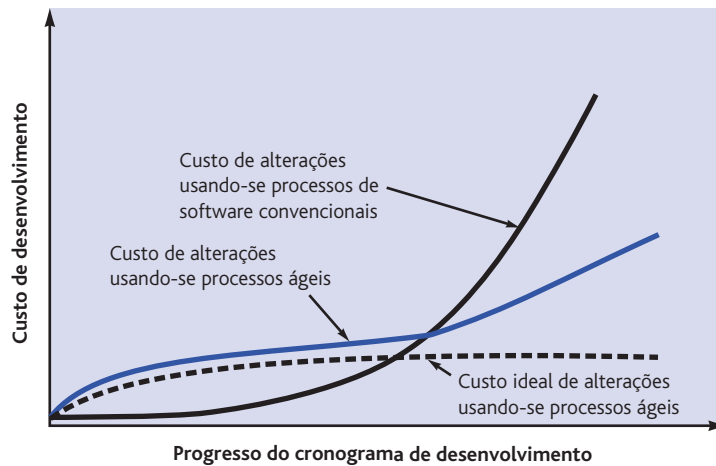


FIGURA 5.1 Custos de alterações como uma função do tempo em desenvolvimento.

fácil acomodar uma mudança quando a equipe de software está reunindo requisitos (no início de um projeto). Talvez seja necessário alterar um detalhamento do uso, ampliar uma lista de funções ou editar uma especificação por escrito. Os custos desse trabalho são mínimos, e o tempo demandado não afetará negativamente o resultado do projeto. Mas, se adiantarmos alguns meses, o que aconteceria? A equipe está em meio aos testes de validação (que ocorrem relativamente no final do projeto), e um importante envolvido está solicitando uma mudança funcional grande. A mudança exige uma alteração no projeto da arquitetura do software, projeto e desenvolvimento de três novos componentes, modificações em outros cinco componentes, projeto de novos testes e assim por diante. Os custos crescem rapidamente, e o tempo e os custos necessários para assegurar que a mudança seja feita sem efeitos colaterais inesperados não serão insignificantes.

Os proponentes da agilidade (por exemplo, [Bec00], [Amb04]) argumentam que um processo ágil bem elaborado “achata” o custo da curva de mudança (Figura 5.1, curva em linha azul), permitindo que uma equipe de software assimile as alterações, realizadas posteriormente em um projeto de software, sem um impacto significativo nos custos ou no tempo. Já foi mencionado que o processo ágil envolve entregas incrementais. O custo das mudanças é atenuado quando a entrega incremental é associada a outras práticas ágeis, como testes contínuos de unidades e programação em pares (discutida mais adiante neste capítulo). Há evidências [Coc01a] que sugerem ser possível alcançar redução significativa nos custos de alterações, embora haja um debate contínuo sobre qual o nível em que a curva de custos se torna “achatada”.

“A agilidade é dinâmica, de conteúdo específico, abrange mudanças agressivas e é orientada ao crescimento.”

**Steven Goldman
et al.**

Um processo ágil reduz o custo das alterações porque o software é entregue (liberado) de forma incremental e as alterações podem ser mais bem controladas dentro de incrementais.

5.3 O que é processo ágil?

Qualquer processo ágil de software é caracterizado de uma forma que trate de uma série de preceitos-chave [Fow02] acerca da maioria dos projetos de software:

1. É difícil prever quais requisitos de software vão persistir e quais sofrerão alterações. É igualmente difícil prever de que maneira as prioridades do cliente sofrerão alterações conforme o projeto avança.

Uma vasta coleção de artigos sobre processo ágil pode ser encontrada em <http://www.agilemodeling.com/>.

2. Para muitos tipos de software, o projeto e a construção são intercalados. Ou seja, ambas as atividades devem ser realizadas em conjunto para que os modelos de projeto sejam provados conforme são criados. É difícil prever quanto de trabalho de projeto será necessário antes que a sua construção (desenvolvimento) seja implementada para avaliar o projeto.
3. Análise, projeto, construção (desenvolvimento) e testes não são tão previsíveis (do ponto de vista de planejamento) quanto gostaríamos que fosse.

Dados esses três preceitos, surge uma importante questão: como criar um processo capaz de administrar a *imprevisibilidade*? A resposta, conforme já observado, está na adaptabilidade do processo (alterar rapidamente o projeto e as condições técnicas). Portanto, um processo ágil deve ser *adaptável*.

Adaptação contínua sem progressos, entretanto, de pouco adianta. Um processo ágil de software deve adaptar *de modo incremental*. Para conseguir uma adaptação incremental, a equipe ágil precisa de feedback do cliente (de modo que as adaptações apropriadas possam ser feitas). Um catalisador eficaz para o feedback do cliente é um protótipo operacional ou parte de um sistema operacional. Dessa forma, deve-se instituir uma *estratégia de desenvolvimento incremental*. Os *incrementos de software* (protótipos executáveis ou partes de um sistema operacional) devem ser entregues em curtos períodos de tempo, de modo que as adaptações acompanhem o mesmo ritmo das mudanças (imprevisibilidade). Essa abordagem iterativa capacita o cliente a avaliar o incremento de software regularmente, fornecer o feedback necessário para a equipe de software e influenciar as adaptações feitas no processo para incluir o feedback adequadamente.

5.3.1 Princípios da agilidade

A Agile Alliance (consulte [Agi03], [Fow01]) estabelece 12 princípios para alcançar a agilidade:

1. A maior prioridade é satisfazer o cliente com entrega adiantada e contínua de software funcionando.
2. Aceite bem os pedidos de alterações, mesmo com o desenvolvimento adiantado. Os processos ágeis se aproveitam das mudanças para a vantagem competitiva do cliente.
3. Entregue software em funcionamento frequentemente, de algumas semanas a alguns meses, dando preferência a intervalos mais curtos.
4. O pessoal do comercial e os desenvolvedores devem trabalhar em conjunto diariamente ao longo de todo o projeto.
5. Construa projetos em torno de pessoas motivadas. Dê a elas o ambiente e o apoio necessários e acredite que elas farão o trabalho corretamente.
6. O método mais eficiente e efetivo de transmitir informações para e dentro de uma equipe de desenvolvimento é uma conversa aberta, presencial.
7. Software em funcionamento é a principal medida de progresso.

Embora processos ágeis considerem as alterações, examinar as razões para tais mudanças ainda continua sendo importante.

Software ativo é importante, mas não se deve esquecer que também deve apresentar uma série de atributos de qualidade, incluindo confiabilidade, usabilidade e facilidade de manutenção.

8. Os processos ágeis promovem desenvolvimento sustentável. Proponentes, desenvolvedores e usuários devem estar aptos a manter um ritmo constante indefinidamente.
9. Atenção contínua para com a excelência técnica e para com bons projetos aumenta a agilidade.
10. Simplicidade – a arte de maximizar o volume de trabalho não realizado – é essencial.
11. As melhores arquiteturas, requisitos e projetos surgem de equipes auto-organizadas.
12. Em intervalos regulares, a equipe se avalia para ver como pode se tornar mais eficiente, então, sintoniza e ajusta seu comportamento de acordo.

Nem todo modelo de processo ágil aplica esses 12 princípios atribuindo-lhes pesos iguais, e alguns modelos preferem ignorar (ou pelo menos subestimam) a importância de um ou mais desses princípios. Entretanto, os princípios definem um *espírito ágil* mantido em cada um dos modelos de processo apresentados neste capítulo.

5.3.2 A política do desenvolvimento ágil

Tem havido debates consideráveis (algumas vezes acirrados) sobre os benefícios e a aplicabilidade do desenvolvimento de software ágil, em contraposição aos processos de engenharia de software mais convencionais. Jim Highsmith [Hig02a] (em tom jocoso) estabelece extremos ao caracterizar o sentimento do grupo pró-agilidade (“os agilistas”). “Os metodologistas tradicionais são um bando de ‘pés na lama’ que preferem produzir documentação sem falhas em vez de um sistema que funcione e atenda às necessidades do negócio”. Em um contraponto, ele apresenta (mais uma vez em tom jocoso) a posição do grupo da engenharia de software tradicional: “Os metodologistas de pouco peso, quer dizer, os metodologistas ‘ágeis’ são um bando de hackers pretensiosos que vão acabar tendo uma grande surpresa ao tentarem transformar seus brinquedinhos em software de porte empresarial”.

Como todo argumento sobre tecnologia de software, o debate sobre metodologia corre o risco de descambar para uma guerra santa. Se for deflagrada uma guerra, a racionalidade desaparecerá, e crenças, em vez de fatos, orientarão a tomada de decisão.

Ninguém é contra a agilidade. A verdadeira questão é: qual a melhor maneira de atingi-la? Igualmente importante é: como desenvolver software que atenda às necessidades atuais dos clientes e que apresente características de qualidade que o permitam ser estendido e ampliado para responder às necessidades dos clientes no longo prazo?

Não há respostas absolutas para nenhuma dessas perguntas. Mesmo na própria escola ágil, existem vários modelos de processos propostos (Seção 5.4), cada um com uma abordagem sutilmente diferente a respeito do problema da agilidade. Em cada modelo existe um conjunto de “ideias” (os agilistas relutam em chamá-las “tarefas de trabalho”) que representam um afastamento significativo da engenharia de software tradicional. E, ainda assim, muitos conceitos

Você não tem de escolher entre agilidade ou engenharia de software. Em vez disso, defina uma abordagem de engenharia de software que seja ágil.

ágeis são apenas adaptações de bons conceitos da engenharia de software. Conclusão: pode-se ganhar muito considerando o que há de melhor nas duas escolas e praticamente nada denegando uma ou outra abordagem.

Caso se interesse mais, consulte [Hig01], [Hig02a] e [DeM02], em que é apresentado um resumo interessante a respeito de outras importantes questões técnicas e políticas.

5.4 Extreme programming – XP (Programação Extrema)

Um premiado “jogo de simulação de processos”, que inclui um módulo de processo XP, pode ser encontrado em: <http://www.ics.uci.edu/~emilyo/SimSE/downloads.html>.

Para ilustrar um processo ágil de forma um pouco mais detalhada, vamos dar uma visão geral da *Extreme Programming – XP (Programação Extrema)*, a abordagem mais amplamente utilizada para desenvolvimento de software ágil. Embora os primeiros trabalhos sobre os conceitos e métodos associados à XP tenham ocorrido no final dos anos 1980, o trabalho seminal sobre o tema foi escrito por Kent Beck [Bec04a]. Uma variante da XP, denominada *Industrial XP (IXP)*, refina a XP para aplicar processo ágil especificamente em grandes organizações [Ker05].

5.4.1 O processo XP

O que é uma “história” XP?

A Extreme Programming (Programação Extrema) emprega uma metodologia orientada a objetos (Apêndice 2) como seu paradigma de desenvolvimento e envolve um conjunto de regras e práticas constantes no contexto de quatro atividades metodológicas: planejamento, projeto, codificação e testes. A Figura 5.2 ilustra o processo XP e destaca alguns conceitos e tarefas-chave associados a cada uma das atividades metodológicas. As atividades-chave da XP são sintetizadas nos parágrafos a seguir.

Planejamento. A atividade de planejamento (também chamada de *o jogo do planejamento*) se inicia com *ouvir* – uma atividade de levantamento de requisi-

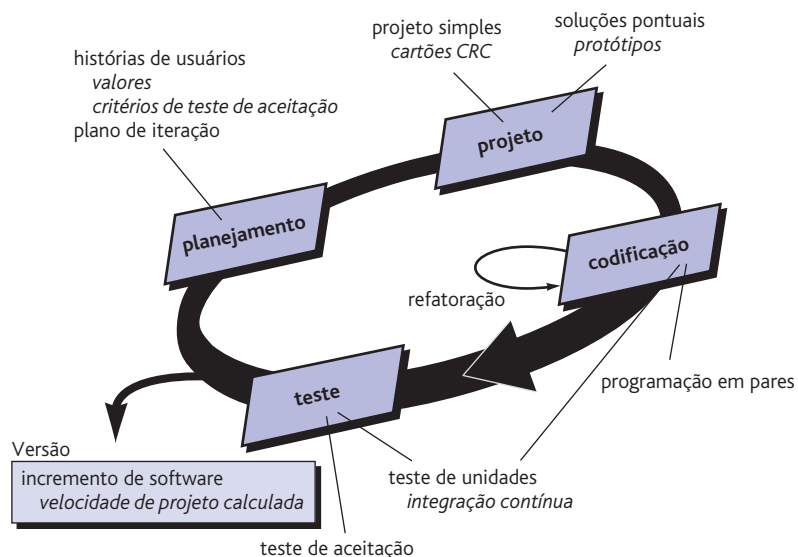


FIGURA 5.2 O processo da Extreme Programming (XP).

tos que capacita os membros técnicos da equipe XP a entender o ambiente de negócios do software e permite obter uma percepção ampla sobre os resultados solicitados, fatores principais e funcionalidade. A atividade de ouvir conduz à criação de um conjunto de “histórias” (também denominadas *histórias de usuários*) que descreve o resultado, as características e a funcionalidade solicitados para o software a ser construído. Cada *história* (similar aos casos de uso descritos no Capítulo 8) é escrita pelo cliente e é colocada em uma ficha. O cliente atribui um *valor* (uma prioridade) à história baseando-se no valor de negócio global do recurso ou função.² Os membros da equipe XP avaliam, então, cada história e atribuem um *custo* – medido em semanas de desenvolvimento – a ela. Se a história exigir, por estimativa, mais do que três semanas de desenvolvimento, é solicitado ao cliente que ele a divida em histórias menores, e a atribuição de valor e custo ocorre novamente. É importante notar que podem ser escritas novas histórias a qualquer momento.

Clientes e desenvolvedores trabalham juntos para decidir como agrupar histórias para a versão seguinte (o próximo incremento de software) a ser desenvolvida pela equipe XP. Conseguindo chegar a um *compromisso* básico (concordância sobre quais histórias serão incluídas, data de entrega e outras questões de projeto) para uma versão, a equipe XP ordena as histórias a ser desenvolvidas em uma de três formas: (1) todas serão implementadas imediatamente (em um prazo de poucas semanas), (2) as histórias de maior valor serão deslocadas para cima no cronograma e implementadas primeiro ou (3) as histórias de maior risco serão deslocadas para cima no cronograma e implementadas primeiro.

Depois de a primeira versão do projeto (também denominada incremento de software) ter sido entregue, a equipe XP calcula a velocidade do projeto. De forma simples, a *velocidade do projeto* é o número de histórias de clientes implementadas durante a primeira versão. Assim, a velocidade do projeto pode ser utilizada para (1) ajudar a estimar as datas de entrega e o cronograma para versões subsequentes e (2) determinar se foi assumido um compromisso exagerado para todas as histórias ao longo de todo o projeto de desenvolvimento. Se ocorrer um exagero, o conteúdo das versões é modificado – ou as datas finais de entrega são alteradas.

Conforme o trabalho de desenvolvimento prossegue, o cliente pode acrescentar histórias, mudar o valor de uma já existente, dividir algumas ou eliminá-las. Em seguida, a equipe XP reconsidera todas as versões remanescentes e modifica seus planos de forma correspondente.

Projeto. O projeto XP segue rigorosamente o princípio KISS (*keep it simple, stupid!*, ou seja, não complique!). É sempre preferível um projeto simples a uma representação mais complexa. Como acréscimo, o projeto oferece um guia de implementação para uma história à medida que é escrita – nada mais, nada menos. O projeto de funcionalidade extra (pelo fato de o desenvolvedor supor que ela será necessária no futuro) é desestimulado.³

Um “jogo de planejamento” XP bastante interessante pode ser encontrado em: <http://csis.pace.edu/~bergin/xp/planninggame.html>.

A velocidade do projeto é uma medida sutil da produtividade de uma equipe.

A XP tira a ênfase da importância do projeto. Nem todos concordam. De fato, há ocasiões em que o projeto deve ser enfatizado.

² O valor de uma história também pode depender da presença de outra história.

³ Tais diretrizes de projeto devem ser seguidas em todos os métodos de engenharia de software, apesar de ocorrerem situações em que terminologia e notação sofisticadas possam constituir obstáculo para a simplicidade.

Técnicas de refatoração e ferramentas podem ser encontradas em: www.refactoring.com.

A refatoração aprimora a estrutura interna de um projeto (ou código-fonte) sem alterar sua funcionalidade ou comportamento externos.

Informações úteis sobre a XP podem ser obtidas em www.xprogramming.com.

A XP estimula o uso de cartões CRC (Capítulo 10) como um mecanismo eficaz para pensar o software em um contexto orientado a objetos. Os cartões CRC (classe-responsabilidade-colaborador) identificam e organizam as classes orientadas a objetos⁴ relevantes para o incremento de software corrente. A equipe XP conduz o exercício de projeto usando um processo semelhante ao descrito no Capítulo 10. Os cartões CRC são o único artefato de projeto produzido como parte do processo XP.

Se for encontrado um problema de projeto difícil, como parte do projeto de uma história, a XP recomenda a criação imediata de um protótipo operacional dessa parte do projeto. Denominada *solução pontual*, o protótipo do projeto é implementado e avaliado. O objetivo é reduzir o risco para quando a verdadeira implementação iniciar e validar as estimativas originais para a história contendo o problema de projeto.

A XP estimula a *refatoração* – uma técnica de construção que também é uma técnica de projeto. Fowler [Fow00] descreve a refatoração da seguinte maneira:

Refatoração é o processo de alterar um sistema de software de modo que o comportamento externo do código não se altere, mas a estrutura interna se aprimore. É uma forma disciplinada de organizar código le modificar/simplificar o projeto interno que minimiza as chances de introdução de bugs. Em resumo, ao se refatorar, se está aperfeiçoando o projeto de codificação depois de este ter sido feito.

Como o projeto XP praticamente não usa notação e produz poucos artefatos, quando produz, além dos cartões CRC e soluções pontuais, o projeto é visto como algo transitório que pode e deve ser continuamente modificado conforme a construção prossegue. O objetivo da refatoração é controlar essas modificações, sugerindo pequenas mudanças de projeto “capazes de melhorá-lo radicalmente” [Fow00]. Deve-se observar, no entanto, que o esforço necessário para a refatoração pode aumentar significativamente à medida que o tamanho de uma aplicação cresce.

Um aspecto central na XP é o de que a elaboração do projeto ocorre tanto antes *quanto depois* de se ter iniciado a codificação. Refatoração significa que o “projetar” é realizado continuamente enquanto o sistema estiver em elaboração. Na realidade, a própria atividade de desenvolvimento guiará a equipe XP quanto ao aprimoramento do projeto.

Codificação. Depois de desenvolvidas as histórias, e de o trabalho preliminar de elaboração do projeto ter sido feito, a equipe *não* passa para a codificação, mas sim desenvolve uma série de testes de unidades que exercitarão cada uma das histórias a ser incluída na versão corrente (incremento de software).⁵

⁴ As classes orientadas a objetos são discutidas no Apêndice 2, no Capítulo 10 e ao longo da Parte II deste livro.

⁵ Essa abordagem equivale a saber as perguntas de uma prova antes de começar a estudar. Torna o estudo muito mais fácil, permitindo que se concentre a atenção apenas nas perguntas que serão feitas.

Uma vez criado o teste de unidades⁶, o desenvolvedor poderá se concentrar melhor no que deve ser implementado para ser aprovado no teste. Nada estranho é adicionado (KISS). Estando o código completo, ele pode ser testado em unidade imediatamente e, dessa forma, fornecer feedback para os desenvolvedores instantaneamente.

Um conceito-chave na atividade de codificação (e um dos mais discutidos aspectos da XP) é a *programação em pares*. A XP recomenda que duas pessoas trabalhem juntas em uma mesma estação de trabalho para criar código para uma história. Isso fornece um mecanismo para solução de problemas em tempo real (duas cabeças normalmente funcionam melhor do que uma) e garantia da qualidade em tempo real (o código é revisto à medida que é criado). Ele também mantém os desenvolvedores concentrados no problema em questão. Na prática, cada pessoa assume um papel ligeiramente diferente. Por exemplo, uma pessoa poderia pensar nos detalhes da codificação de determinada parte do projeto, enquanto outra assegura que padrões de codificação (uma parte exigida pela XP) sejam seguidos ou que o código para a história passará no teste de unidades desenvolvido para validação do código em relação à história.⁷

À medida que a dupla de programadores conclui o trabalho, o código que desenvolveram é integrado ao trabalho de outros. Em alguns casos, isso é realizado diariamente por uma equipe de integração. Em outros, a dupla de programadores é responsável pela integração. A estratégia de “integração contínua” ajuda a evitar problemas de compatibilidade e de interface, além de criar um ambiente “teste da fumaça” (Capítulo 22) que ajuda a revelar erros precocemente.

Testes. Os testes de unidades criados devem ser implementados usando-se uma metodologia que os capacite a ser automatizados (assim, poderão ser executados fácil e repetidamente). Isso estimula uma estratégia de testes de regressão (Capítulo 22) toda vez que o código for modificado (o que é frequente, dada a filosofia de refatoração da XP).

Como os testes de unidades individuais são organizados em um “conjunto de testes universal” [Wel99], os testes de integração e validação do sistema podem ocorrer diariamente. Isso dá à equipe XP uma indicação contínua do progresso e também permite lançar alertas logo no início, caso as coisas não andem bem. Wells [Wel99] afirma: “Corrigir pequenos problemas em intervalos de poucas horas leva menos tempo do que corrigir problemas enormes próximo ao prazo de entrega”.

Os *testes de aceitação* da XP, também denominados *testes de cliente*, são especificados pelo cliente e mantêm o foco nas características e na funcionalidade do sistema total que são visíveis e que podem ser revistas pelo cliente. Os testes de aceitação são obtidos de histórias de usuários implementadas como parte de uma versão do software.

O que é programação em pares?

Muitas equipes de software são constituídas por individualistas. É preciso mudar tal cultura para que a programação em pares funcione efetivamente.

Como são usados os testes de unidade na XP?

Os testes de aceitação da XP são elaborados com base nas histórias de usuários.

⁶ O teste de unidades, discutido detalhadamente no Capítulo 22, concentra-se em um componente de software individual, exercitando a interface, a estrutura de dados e a funcionalidade do componente, em uma tentativa de que se revelem erros pertinentes ao componente.

⁷ A programação em pares se tornou tão difundida em toda a comunidade do software, que o tema virou manchete no *The Wall Street Journal* [Wal12].

Que novas práticas são acrescentadas à XP para elaborar a IXP?

"Habilidade consiste no que se é capaz de fazer. Motivação determina o que você faz. Atitude determina quão bem você faz."

Lou Holtz

5.4.2 Industrial XP

Joshua Kerievsky [Ker05] descreve a *Industrial Extreme Programming* (IXP, Programação Extrema Industrial) da seguinte maneira: "A IXP é uma evolução orgânica da XP. Ela é imbuída do mesmo espírito minimalista, centrado no cliente e orientado a testes da XP. Difere da XP original principalmente por sua maior inclusão do gerenciamento, por seu papel expandido para os clientes e por suas práticas técnicas atualizadas". A IXP incorpora seis novas práticas desenvolvidas para ajudar a garantir que um projeto XP funcione com êxito em empreendimentos significativos em uma grande organização:

Avaliação imediata. A equipe IXP verifica se todos os membros da comunidade de projeto (por exemplo, envolvidos, desenvolvedores, gerentes) estão a bordo, têm o ambiente correto estabelecido e entendem os níveis de habilidade envolvidos.

Comunidade de projeto. A equipe IXP determina se as pessoas certas, com as habilidades e o treinamento corretos, estão prontas para o projeto. A "comunidade" abrange tecnólogos e outros envolvidos.

Mapeamento do projeto. A própria equipe IXP avalia o projeto para determinar se ele se justifica em termos de negócios e se vai ultrapassar as metas e objetivos globais da organização.

Gerenciamento orientado a testes. A equipe IPX estabelece uma série de "destinos" mensuráveis [Ker05] que avaliam o progresso até a data e, então, define mecanismos para determinar se estes foram atingidos ou não.

Retrospectivas. Uma equipe IXP conduz uma revisão técnica especializada (Capítulo 20) após a entrega de um incremento de software. Denominada *retrospectiva*, a revisão examina "problemas, eventos e lições aprendidas" [Ker05] ao longo do processo de incremento de software e/ou do desenvolvimento da versão completa do software.

Aprendizagem contínua. A equipe IXP é estimulada (e possivelmente incentivada) a aprender novos métodos e técnicas que possam levar a um produto de qualidade mais alta.

CASASEGURA



Considerando o desenvolvimento de software ágil

Cena: Escritório de Doug Miller.

Atores: Doug Miller, gerente de engenharia de software; Jamie Lazar, membro da equipe de software; Vinod Raman, membro da equipe de software.

Conversa:
(Batendo à porta, Jamie e Vinod entram na sala de Doug.)

Jamie: Doug, você tem um minuto?

Doug: Com certeza, Jamie, o que há?

Jamie: Estivemos pensando a respeito da discussão de ontem sobre processos... sabe, que processo vamos escolher para o *CasaSegura*.

Doug: E?

Vinod: Eu estava conversando com um amigo de outra empresa e ele me falou sobre Extreme Programming. É um modelo de processo ágil... já ouviu falar?

Doug: Sim, algumas coisas boas, outras ruins.

Jamie: Bem, pareceu muito bom para nós. Permite que se desenvolva software rapidamente, usa algo chamado programação em pares para fazer verificações de qualidade em tempo real... é bem legal, eu acho.

Doug: Realmente, apresenta um monte de ideias muito boas. Gosto do conceito de programação em pares, por exemplo, e da ideia de que os envolvidos devam fazer parte da equipe.

Jamie: Hã? Quer dizer que o pessoal de marketing trabalhará conosco na equipe de projeto?

Doug (confirmando com a cabeça): Eles estão envolvidos, não?

Jamie: Jesus... eles vão solicitar alterações a cada cinco minutos.

Vinod: Não necessariamente. Meu amigo me disse que existem formas de se “abarcara” as mudanças durante um projeto XP.

Doug: Então, meus amigos, vocês acham que deveríamos usar a XP?

Jamie: Definitivamente vale considerar.

Doug: Concordo. E mesmo que optássemos por um modelo incremental, não há razão para não podermos incorporar muito do que a XP tem a oferecer.

Vinod: Doug, mas antes você disse “algumas coisas boas, outras ruins”. Quais são as coisas ruins?

Doug: O que não me agrada é a maneira como a XP dá menos importância à análise e ao projeto... diz mais ou menos que a codificação é onde a ação está...

(Os membros da equipe se entreolham e sorriem.)

Doug: Então vocês concordam com a metodologia XP?

Jamie (falando por ambos): Escrever código é o que fazemos, chefe!

Doug (rindo): É verdade, mas eu gostaria de vê-los perdendo um pouco menos de tempo codificando para depois re-codificar e dedicando um pouco mais de tempo analisando o que precisa ser feito e projetando uma solução que funcione.

Vinod: Talvez possamos ter as duas coisas, agilidade com um pouco de disciplina.

Doug: Acho que sim, Vinod. Na realidade, tenho certeza disso.

Além das seis novas práticas apresentadas, a IXP modifica várias práticas XP existentes e redefine certas funções e responsabilidades para torná-las mais receptivas para projetos importantes de grandes empresas. Para uma discussão mais ampla sobre a IXP, visite <http://industrialxp.org>.

5.5 Outros modelos de processos ágeis

A história da engenharia de software é recheada de metodologias e descrições de processos, métodos e notações de modelagem, ferramentas e tecnologias obsoletas. Todas atingiram certa notoriedade e foram ofuscadas por algo novo e (supostamente) melhor. Com a introdução de uma ampla variedade de modelos de processos ágeis – todos disputando aceitação pela comunidade de desenvolvimento de software –, o movimento ágil está seguindo o mesmo caminho histórico.⁸

Conforme citado na última seção, o modelo mais utilizado entre os modelos de processos ágeis é o Extreme Programming (XP). Porém, muitos outros têm sido propostos e encontram-se em uso no setor. Nesta seção, apresentamos um breve panorama de quatro métodos ágeis comuns: Scrum, DSSD, Modelagem Ágil (AM) e Processo Unificado Ágil (AUP).

“Nossa profissão troca de metodologias como uma garota de 14 anos troca de roupas.”

**Stephen Hawrysh e
Jim Ruprecht**

⁸ Isso não é algo ruim. Antes que um ou mais modelos ou métodos sejam aceitos como um padrão, todos devem competir para conquistar os corações e mentes dos engenheiros de software. Os “vencedores” evoluem e se transformam nas boas práticas, enquanto os “perdedores” desaparecem ou se fundem aos modelos vencedores.

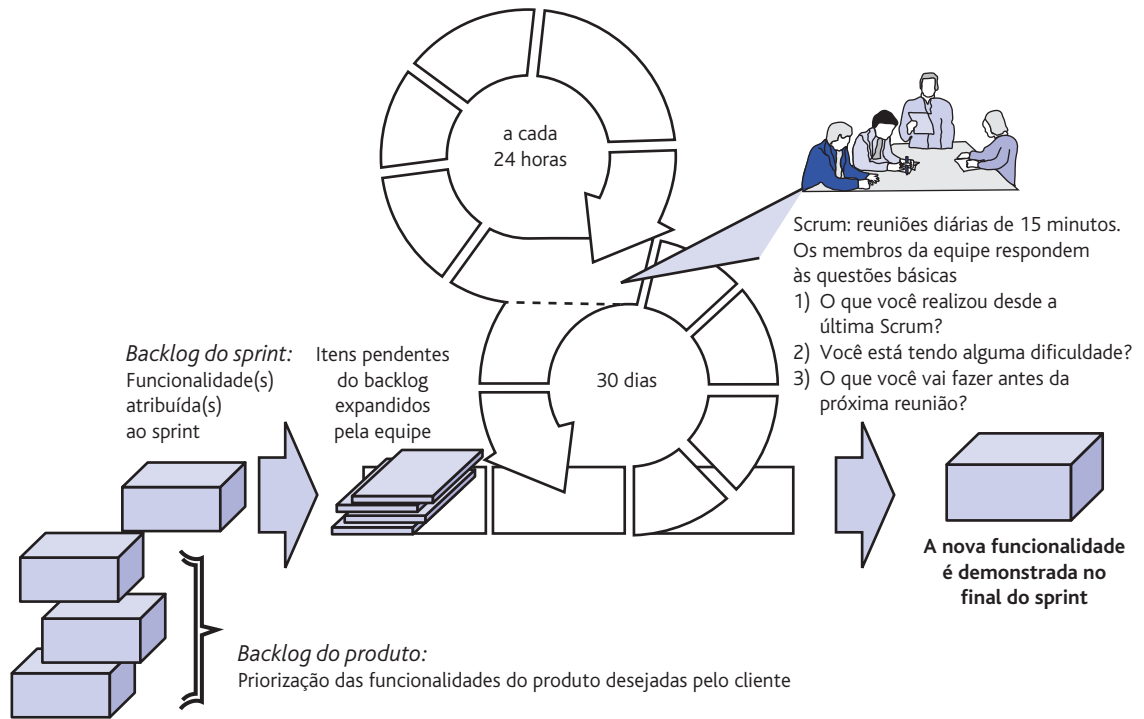


FIGURA 5.3 Fluxo do processo Scrum.

5.5.1 Scrum

Informações e recursos úteis sobre o Scrum podem ser encontrados em www.controlchaos.com.

Scrum (o nome provém de uma atividade que ocorre durante a partida de rugby)⁹ é um método de desenvolvimento ágil de software concebido por Jeff Sutherland e sua equipe de desenvolvimento no início dos anos 1990. Mais recentemente, Schwaber e Beedle [Sch01b] realizaram desenvolvimentos adicionais nos métodos Scrum.

Os princípios do Scrum são coerentes com o manifesto ágil e são usados para orientar as atividades de desenvolvimento dentro de um processo que incorpora as seguintes atividades metodológicas: requisitos, análise, projeto, evolução e entrega. Em cada atividade metodológica, ocorrem tarefas realizadas dentro de um padrão de processo (discutido no parágrafo a seguir) chamado *sprint*. O trabalho realizado dentro de um sprint (o número de sprints necessários para cada atividade metodológica varia dependendo do tamanho e da complexidade do produto) é adaptado ao problema em questão e definido, e muitas vezes modificado em tempo real, pela equipe Scrum. O fluxo geral do processo Scrum está ilustrado na Figura 5.3.

O Scrum enfatiza o uso de um conjunto de padrões de processos de software [Noy02] que provaram ser eficazes para projetos com prazos de entrega apertados, requisitos mutáveis e urgência do negócio. Cada um desses padrões de processos define um conjunto de atividades de desenvolvimento:

⁹ Um grupo de jogadores faz uma formação em torno da bola, e seus companheiros de equipe trabalham juntos (às vezes, de forma violenta!) para avançar com a bola em direção ao fundo do campo.

Backlog – uma lista com prioridades dos requisitos ou funcionalidades do projeto que fornecem valor comercial ao cliente. Os itens podem ser adicionados a esse registro a qualquer momento (é assim que as alterações são introduzidas). O gerente de produto avalia o registro e atualiza as prioridades conforme solicitado.

Sprints – consistem em unidades de trabalho solicitadas para atingir um requisito estabelecido no registro de trabalho (backlog) e que precisa ser ajustado dentro de um prazo já fechado (janela de tempo)¹⁰ (tipicamente 30 dias). Alterações (por exemplo, itens do registro de trabalho – *backlog work items*) não são introduzidas durante execução de urgências (sprint). Portanto, o sprint permite que os membros de uma equipe trabalhem em um ambiente de curto prazo, porém estável.

Reuniões Scrum – são reuniões curtas (tipicamente 15 minutos), realizadas diariamente pela equipe Scrum. São feitas três perguntas-chave que são respondidas por todos os membros da equipe [Noy02]:

- O que você realizou desde a última reunião de equipe?
- Quais obstáculos está encontrando?
- O que planeja realizar até a próxima reunião da equipe?

Um líder de equipe, chamado *Scrum master*, conduz a reunião e avalia as respostas de cada integrante. A reunião Scrum, realizada diariamente, ajuda a equipe a revelar problemas em potencial o mais cedo possível. Ela também leva à “socialização do conhecimento” [Bee99] e, portanto, promove uma estrutura de equipe auto-organizada.

Demos – entrega do incremento de software ao cliente para que a funcionalidade implementada possa ser demonstrada e avaliada por ele. É importante notar que a demo pode não ter toda a funcionalidade planejada, mas sim funções que possam ser entregues no prazo estipulado.

Beedle e seus colegas [Bee99] apresentam uma ampla discussão sobre esses padrões: “O Scrum pressupõe a existência do caos...”. Os padrões de processos do Scrum capacitam uma equipe de software a trabalhar com sucesso em um mundo onde é impossível eliminar a incerteza.

5.5.2 Método de Desenvolvimento de Sistemas Dinâmicos (DSDM)

O *Método de Desenvolvimento de Sistemas Dinâmicos* (DSDM, *Dynamic Systems Development Method*) [Sta97] é uma abordagem de desenvolvimento de software ágil que “oferece uma metodologia para construir e manter sistemas que satisfaçam restrições de prazo apertado por meio do uso da prototipação incremental em um ambiente de projeto controlado” [CCS02]. A filosofia DSDM baseia-se em uma versão modificada do princípio de Pareto – 80% de uma aplicação pode ser entregue em 20% do tempo que levaria para entregar a aplicação completa (100%).

O Scrum engloba um conjunto de padrões de processos enfatizando prioridades de projeto, unidades de trabalho compartimentalizadas, comunicação e feedback frequente por parte dos clientes.

Recursos úteis para o DSDM podem ser encontrados em www.dsdm.org.

¹⁰ *Janela de tempo* (*time box*) é um termo de gerenciamento de projetos (consulte a Parte IV deste livro) que indica um período de tempo destinado para cumprir alguma tarefa.

O DSDM é um processo de software iterativo em que cada iteração segue a regra dos 80%. Ou seja, somente o trabalho suficiente é requisitado para cada incremento, para facilitar o movimento para o próximo incremento. Os detalhes restantes podem ser concluídos depois, quando outros requisitos do negócio forem conhecidos ou alterações tiverem sido solicitadas e acomodadas.

O DSDM Consortium (www.dsdm.org) é um grupo mundial de empresas-membro que assume coletivamente o papel de “mantenedor” do método. Esse consórcio definiu um modelo de processos ágeis, chamado *ciclo de vida DSDM*, que começa com um *estudo de viabilidade*, o qual estabelece os requisitos básicos e as restrições do negócio, e é seguido por um *estudo do negócio*, o qual identifica os requisitos de função e informação. Então, o DSDM define três diferentes ciclos iterativos:

O DSDM é uma metodologia de processos que pode adotar a tática de outra metodologia ágil, como a XP.

Iteração de modelos funcionais – produz um conjunto de protótipos incrementais que demonstram funcionalidade para o cliente. (Observação: todos os protótipos DSDM são feitos com a intenção de que evoluam para a aplicação final entregue ao cliente.) Durante esse ciclo iterativo, o objetivo é reunir requisitos adicionais ao se obter feedback dos usuários, à medida que eles testam o protótipo.

Iteração de projeto e desenvolvimento – revê os protótipos desenvolvidos durante a iteração de modelos funcionais para assegurar-se de que cada um tenha passado por um processo de engenharia para capacitá-los a oferecer, aos usuários, valor de negócio em termos operacionais. Em alguns casos, a iteração de modelos funcionais e a iteração de projeto e desenvolvimento ocorrem ao mesmo tempo.

Implementação – coloca a última versão do incremento de software (um protótipo “operacionalizado”) no ambiente operacional. Deve-se notar que: (1) o incremento pode não estar 100% completo ou (2) alterações podem vir a ser solicitadas conforme o incremento seja alocado. Em qualquer um dos casos, o trabalho de desenvolvimento do DSDM continua, retornando-se à atividade de iteração do modelo funcional.

O DSDM pode ser combinado com a XP (Seção 5.4) para fornecer uma abordagem combinada que defina um modelo de processos confiável (o ciclo de vida do DSDM) com as práticas básicas (XP) necessárias para construir incrementos de software.

5.5.3 Modelagem Ágil (AM)

Muita informação sobre a modelagem ágil pode ser encontrada em: www.agilemodeling.com.

Existem muitas situações em que engenheiros de software têm de desenvolver sistemas grandes e críticos para o negócio. O escopo e a complexidade desses sistemas devem ser modelados de modo que (1) todas as partes envolvidas possam entender melhor quais requisitos devem ser atingidos, (2) o problema possa ser subdividido eficientemente entre as pessoas que têm de solucioná-lo e (3) a qualidade possa ser avaliada enquanto se está projetando e desenvolvendo o sistema. Porém, em alguns casos pode ser desencorajador gerenciar o volume de notação exigido, o grau de formalismo sugerido, o mero tamanho dos modelos para grandes projetos e a dificuldade em manter o(s) modelo(s) à

medida que ocorrem mudanças. Existe uma metodologia ágil para a modelagem de engenharia de software que possa fornecer algum alívio?

No “The Official Agile Modeling Site”, Scott Ambler [Amb02a] descreve *modelagem ágil* (AM) da seguinte maneira:

Modelagem ágil (AM) consiste em uma metodologia prática, voltada para a modelagem e documentação de sistemas baseados em software. Simplificando, modelagem ágil consiste em um conjunto de valores, princípios e práticas voltados para a modelagem do software que podem ser aplicados a um projeto de desenvolvimento de software de forma leve e eficiente. Os modelos ágeis são mais eficientes do que os tradicionais pelo fato de serem simplesmente bons, pois não têm a obrigação de ser perfeitos.

A modelagem ágil adota todos os valores coerentes com o manifesto ágil. Sua filosofia reconhece que uma equipe ágil deve ter a coragem de tomar decisões que possam causar a rejeição de um projeto e sua refatoração. A equipe também deve ter humildade para reconhecer que os profissionais de tecnologia não possuem todas as respostas e que os experts em negócios e outros envolvidos devem ser respeitados e integrados ao processo.

Embora a AM sugira uma ampla variedade de princípios de modelagem “básicos” e “suplementares”, os que a tornam única são [Amb02a]:

Modelar com um objetivo. O desenvolvedor que utilizar a AM deve ter um objetivo antes de criar o modelo (por exemplo, comunicar informações ao cliente ou ajudar a compreender melhor algum aspecto do software). Uma vez identificado o objetivo, ficará mais evidente o tipo de notação a ser utilizado e o nível de detalhamento necessário.

Usar vários modelos. Há muitos modelos e notações diferentes que podem ser usados para descrever software. Para a maioria dos projetos, somente um subconjunto é essencial. A AM sugere que, para propiciar a percepção necessária, cada modelo deve apresentar um aspecto diferente do sistema e devem ser usados somente aqueles que valorizem esses modelos para o público pretendido.

Viajar leve. Conforme o trabalho de engenharia de software prossegue, conserve apenas aqueles modelos que terão valor no longo prazo e desfaca-se do restante. Todo artefato mantido deve sofrer manutenção à medida que mudanças ocorram. Isso representa trabalho que retarda a equipe. Ambler [Amb02a] observa que “Toda vez que se opta por manter um modelo, troca-se a agilidade pela conveniência de ter aquela informação acessível para a equipe de uma forma abstrata (já que, potencialmente, aumenta a comunicação dentro da equipe, assim como com os envolvidos no projeto)”.

Conteúdo é mais importante do que a representação. A modelagem deve transmitir informações para seu público pretendido. Um modelo sintaticamente perfeito que transmita pouco conteúdo útil não possui tanto valor quanto aquele com notações falhas que, no entanto, fornece conteúdo valioso para seu público-alvo.

Conhecer os modelos e as ferramentas utilizadas para criá-los. Compreenda os pontos fortes e fracos de cada modelo e as ferramentas usadas para criá-lo.

“Um dia, estava em uma farmácia tentando achar um remédio para resfriado... Não foi fácil. Havia uma parede inteira de produtos. Fica-se lá procurando: ‘Bem, este tem ação imediata, mas este outro tem efeito mais duradouro...’ O que é mais importante, o presente ou o futuro?”

Jerry Seinfeld

“Viajar leve” é uma filosofia apropriada para todo o trabalho de engenharia de software. Construa apenas os modelos que forneçam valor... nem mais, nem menos.

Adaptar localmente. A modelagem deve ser adaptada às necessidades da equipe ágil.

Um segmento de vulto da comunidade da engenharia de software adotou a linguagem de modelagem unificada (Unified Modeling Language, UML)¹¹ como o método preferido para análise representativa e para modelos de projeto. O Processo Unificado (Capítulo 4) foi desenvolvido para fornecer uma metodologia para a aplicação da UML. Scott Ambler [Amb06] desenvolveu uma versão simplificada do UP que integra sua filosofia de modelagem ágil.

5.5.4 Processo Unificado Ágil

O *Processo Unificado Ágil* (AUP, *Agile Unified Process*) adota uma filosofia “serial para o que é amplo” e “iterativa para o que é particular” [Amb06] no desenvolvimento de sistemas computadorizados. Adotando as atividades em fases UP clássicas – concepção, elaboração, construção e transição –, o AUP fornece uma camada serial (isto é, uma sequência linear de atividades de engenharia de software) que permite à equipe visualizar o fluxo do processo geral de um projeto de software. Entretanto, dentro de cada atividade, a equipe itera para alcançar a agilidade e entregar incrementos de software significativos para os usuários o mais rápido possível. Cada iteração AUP trata das seguintes atividades [Amb06]:

- *Modelagem.* Representações UML do universo do negócio e do problema são criadas. Entretanto, para permanecerem ágeis, esses modelos devem ser “suficientemente bons e adequados” [Amb06] para possibilitar que a equipe prossiga.
- *Implementação.* Os modelos são traduzidos em código-fonte.
- *Testes.* Como a XP, a equipe projeta e executa uma série de testes para descobrir erros e assegurar que o código-fonte se ajuste aos requisitos.
- *Entrega.* Como a atividade de processo genérica discutida no Capítulo 3, neste contexto a entrega se concentra no fornecimento de um incremento de software e na obtenção de feedback dos usuários.
- *Configuração e gerenciamento de projeto.* No contexto do AUP, gerenciamento de configuração (Capítulo 29) refere-se a gerenciamento de alterações, de riscos e de controle de qualquer artefato¹² persistente que sejam produzidos por uma equipe. O gerenciamento de projeto monitora e controla o progresso de uma equipe e coordena suas atividades.
- *Gerenciamento do ambiente.* Coordena a infraestrutura de processos que inclui padrões, ferramentas e outras tecnologias de suporte disponíveis para a equipe.

¹¹ Um breve tutorial sobre a UML é apresentado no Apêndice 1.

¹² *Artefato persistente* é um modelo ou documento ou pacote de testes produzido pela equipe que será mantido por um período de tempo indeterminado. Não será descartado quando o incremento de software for entregue.

Embora o AUP tenha conexões históricas e técnicas com a linguagem de modelagem unificada, é importante notar que a modelagem UML pode ser usada com qualquer modelo de processo ágil descrito neste capítulo.

FERRAMENTAS DO SOFTWARE



Engenharia de requisitos

Objetivo: O objetivo das ferramentas de desenvolvimento ágil é auxiliar em um ou mais aspectos do desenvolvimento ágil, com ênfase em facilitar a geração rápida de software operacional. Essas ferramentas também podem ser usadas quando forem aplicados modelos de processos prescritivos (Capítulo 4).

Mecanismos: O mecanismo das ferramentas é variado. Em geral, conjuntos de ferramentas ágeis englobam suporte automatizado para o planejamento de projetos, desenvolvimento de casos de uso, reunião de requisitos, projeto rápido, geração de código e testes.

Ferramentas representativas:¹³

Observação: como o desenvolvimento ágil é um tópico importante, a maioria dos fornecedores de ferramentas

de software tende a vender ferramentas que aceitam a metodologia ágil. As ferramentas aqui mencionadas têm características que as tornam particularmente úteis para projetos ágeis.

OnTime, desenvolvida pela Axosoft (www.axosoft.com), fornece suporte para gerenciamento de processo ágil para uma variedade de atividades técnicas dentro do processo.

Ideogramic UML, desenvolvida pela Ideogramic (<http://ideogramic-uml.software.informer.com/>), é um conjunto de ferramentas UML desenvolvido para uso em processo ágil.

Together Tool Set, distribuída pela Borland (www.borland.com), fornece uma mala de ferramentas que dão suporte para muitas atividades técnicas na XP e em outros processos ágeis.

5.6 Um conjunto de ferramentas para o processo ágil

Alguns proponentes da filosofia ágil argumentam que as ferramentas de software automatizadas (por exemplo, ferramentas para projetos) deveriam ser vistas como um suplemento secundário para as atividades, e não como fundamental para o sucesso da equipe. Entretanto, Alistair Cockburn [Coc04] sugere que ferramentas podem trazer vantagens e que “equipes ágeis enfatizam o uso de ferramentas que permitem o fluxo rápido de compreensão. Algumas dessas ferramentas são sociais, iniciando-se até no estágio de contratação de pessoal. Algumas são tecnológicas, auxiliando equipes distribuídas a simular sua presença física. Muitas são físicas, permitindo sua manipulação em workshops.”

“Ferramentas” voltadas para a comunicação e para a colaboração são, em geral, de baixa tecnologia e incorporam qualquer mecanismo (“proximidade física, quadros brancos, papéis para pôster, fichas e lembretes adesivos” [Coc04] ou modernas técnicas de rede social) que forneça informações e coordenação entre desenvolvedores ágeis. A comunicação ativa é obtida por meio de dinâmicas de grupo (por exemplo, programação em pares), enquanto a comunicação passiva é obtida por meio dos “irradiadores de informações” (por exemplo, um display de um painel fixo que apresente o status geral dos diferentes componentes de um incremento). As ferramentas de gerenciamento de projeto dão pouca ênfase ao diagrama de Gantt e o substituem por gráficos de valores ganhos ou “gráficos de testes criados e cruzados com os anteriores... outras ferramentas

O “conjunto de ferramentas” que suporta os processos ágeis se concentra mais nas questões pessoais do que nas questões tecnológicas.

¹³ A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

ágeis são utilizadas para otimizar o ambiente no qual a equipe ágil trabalha (por exemplo, áreas de reunião mais eficientes), ampliar a cultura da equipe promovendo interações sociais (por exemplo, equipes próximas umas das outras), dispositivos físicos (por exemplo, lousas eletrônicas) e melhoria do processo (por exemplo, programação em pares ou janela de tempo)” [Coc04].

Algumas dessas coisas são realmente ferramentas? Serão, caso facilitem o trabalho desenvolvido por um membro da equipe ágil e venham a aprimorar a qualidade do produto final.

5.7 Resumo

Em uma economia moderna, as condições de mercado mudam rapidamente, as necessidades do cliente e do usuário evoluem e novos desafios competitivos surgem sem aviso. Os profissionais têm de assumir uma abordagem de engenharia de software que permita que permaneçam ágeis – definindo processos que sejam manipuláveis, adaptáveis e sem excessos, somente com o conteúdo essencial que possa se adequar às necessidades do mundo dos negócios moderno.

Uma filosofia ágil para a engenharia de software enfatiza quatro elementos-chave: a importância das equipes que se auto-organizam, que têm controle sobre o trabalho por elas realizado; a comunicação e a colaboração entre os membros da equipe e entre os desenvolvedores e seus clientes; o reconhecimento de que as mudanças representam oportunidades; e a ênfase na entrega rápida do software para satisfazer o cliente. Os modelos de processos ágeis foram feitos para tratar de todas essas questões.

Extreme Programming (XP) é o processo ágil mais amplamente utilizado. Organizada em quatro atividades metodológicas – planejamento, projeto, codificação e testes – a XP sugere várias técnicas poderosas e inovadoras que possibilitam a uma equipe ágil criar versões de software com frequência, propiciando recursos e funcionalidade descritos previamente e priorizados pelos envolvidos.

Outros modelos de processos ágeis também enfatizam a colaboração humana e a auto-organização das equipes, mas definem suas próprias atividades metodológicas e selecionam diferentes pontos de ênfase. Por exemplo, o Scrum enfatiza o uso de um conjunto de padrões de software que se mostrou eficaz para projetos com cronogramas apertados, requisitos mutáveis e aspectos críticos de negócio. Cada padrão de processo define um conjunto de tarefas de desenvolvimento e permite à equipe Scrum construir um processo que se adapte às necessidades do projeto. O método de desenvolvimento de sistemas dinâmicos (DSDM) defende o uso de um cronograma de tempos definidos (janela de tempo) e sugere que apenas o trabalho suficiente seja requisitado para cada incremento de software, para facilitar o movimento em direção ao incremento seguinte. A modelagem ágil (AM) afirma que modelagem é essencial para todos os sistemas, mas a complexidade, tipo e tamanhos de um modelo devem ser balizados pelo software a ser construído. O processo unificado ágil (AUP) adota a filosofia do “serial para o que é amplo” e “iterativa para o que é particular” para o desenvolvimento de software.

Problemas e pontos a ponderar

- 5.1. Releia o “Manifesto for Agile Software Development” no início deste capítulo. Você consegue exemplificar uma situação em que um ou mais dos quatro “valores” poderiam levar a equipe a ter problemas?
- 5.2. Descreva agilidade (para projetos de software) com suas próprias palavras.
- 5.3. Por que um processo iterativo facilita o gerenciamento de mudanças? Todos os processos ágeis discutidos neste capítulo são iterativos? É possível concluir um projeto com apenas uma iteração e ainda assim permanecer ágil? Justifique suas respostas.
- 5.4. Cada um dos processos ágeis poderia ser descrito usando-se as atividades metodológicas genéricas citadas no Capítulo 3? Construa uma tabela que associe as atividades genéricas às atividades definidas para cada processo ágil.
- 5.5. Tente elaborar mais um “princípio de agilidade” que ajudaria uma equipe de engenharia de software a se tornar mais adaptável.
- 5.6. Escolha um princípio de agilidade citado na Seção 5.3.1 e tente determinar se cada um dos modelos de processos apresentados neste capítulo demonstra o princípio. (Observação: apresentamos apenas uma visão geral desses modelos de processos; portanto, talvez não seja possível determinar se um princípio foi ou não tratado por um ou mais dos modelos, a menos que você pesquise mais a respeito, o que não é exigido neste problema).
- 5.7. Por que os requisitos mudam tanto? Afinal de contas, as pessoas não sabem o que elas querem?
- 5.8. A maior parte dos modelos de processos ágeis recomenda comunicação face a face. Mesmo assim, hoje em dia os membros de uma equipe de software e seus clientes podem estar geograficamente separados uns dos outros. Você acredita que isso implique que a separação geográfica seja algo a ser evitado? Você é capaz de imaginar maneiras de superar esse problema?
- 5.9. Escreva uma história de usuário XP que descreva o recurso “sites favoritos” ou “favoritos” disponível na maioria dos navegadores Web.
- 5.10. O que é uma solução pontual na XP?
- 5.11. Descreva com suas próprias palavras os conceitos de refatoração e programação em pares da XP.
- 5.12. Usando a planilha de padrões de processos apresentada no Capítulo 3, desenvolva um padrão de processo para qualquer um dos padrões Scrum da Seção 5.5.1.
- 5.13. Visite o site Official Agile Modeling e faça uma lista completa de todos os princípios básicos e complementares do AM.
- 5.14. O conjunto de ferramentas proposto na Seção 5.6 oferece suporte a muitos dos aspectos “menos prioritários” dos métodos ágeis. Como a comunicação é tão importante, recomende um conjunto de ferramentas real que poderia ser usado para melhorar a comunicação entre os envolvidos de uma equipe ágil.

Leituras e fontes de informação complementares

A filosofia geral e os princípios subjacentes do desenvolvimento de software ágil são considerados em profundidade em muitos dos livros citados neste capítulo. Além disso, livros de Pichler (*Agile Project Management with Scrum: Creating Products that Customers Love*, Addison-Wesley, 2010), Highsmith (*Agile Project Management: Creating Innovative*

Products, 2ª ed. Addison-Wesley, 2009), Shore e Chromatic (*The Art of Agile Development*, O'Reilly Media, 2008), Hunt (*Agile Software Construction*, Springer, 2005) e Carmichael e Haywood (*Better Software Faster*, Prentice Hall, 2002) trazem discussões interessantes sobre o tema. Aguanno (*Managing Agile Projects*, Multi-Media Publications, 2005) e Larman (*Agile and Iterative Development: A Manager's Guide*, Addison-Wesley, 2003) apresentam uma visão geral sobre gerenciamento e consideram as questões envolvidas no gerenciamento de projetos. Highsmith (*Agile Software Development Ecosystems*, Addison-Wesley, 2002) retrata uma pesquisa de princípios, processos e práticas ágeis. Uma discussão que vale a pena sobre o delicado equilíbrio entre agilidade e disciplina é fornecida por Booch e seus colegas (*Balancing Agility and Discipline*, Addison-Wesley, 2004).

Martin (*Clean Code: A Handbook of Agile Software Craftsmanship*, Prentice-Hall, 2009) enumera os princípios, padrões e práticas necessários para desenvolver “código limpo” em um ambiente de engenharia de software ágil. Leffingwell (*Agile Software Requirements: Lean Requirements Practices for Teams, Programs, and the Enterprise*, Addison-Wesley, 2011; e *Scaling Software Agility: Best Practices for Large Enterprises*, Addison-Wesley, 2007) discute estratégias para dar maior corpo às práticas ágeis para poderem ser usadas em grandes projetos. Lippert e Rook (*Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*, Wiley, 2006) discutem o uso da refatoração quando aplicada a sistemas grandes e complexos. Stamelos e Sftesos (*Agile Software Development Quality Assurance*, IGI Global, 2007) trazem técnicas SQA que estão em conformidade com a filosofia ágil.

Foram escritos dezenas de livros sobre Extreme Programming ao longo da última década. Beck (*Extreme Programming Explained: Embrace Change*, 2ª ed., Addison-Wesley, 2004) ainda é o tratado de maior autoridade sobre o tema. Além disso, Jeffries e seus colegas (*Extreme Programming Installed*, Addison-Wesley, 2000), Succi e Marchesi (*Extreme Programming Examined*, Addison-Wesley, 2001), Newkirk e Martin (*Extreme Programming in Practice*, Addison-Wesley, 2001) e Auer e seus colegas (*Extreme Programming Applied: Play to Win*, Addison-Wesley, 2001) fornecem uma discussão básica da XP, juntamente com uma orientação sobre como melhor aplicá-la. McBreen (*Questioning Extreme Programming*, Addison-Wesley, 2003) adota uma visão crítica em relação à XP, definindo quando e onde ela é apropriada. Uma análise aprofundada da programação em pares é apresentada por McBreen (*Pair Programming Illuminated*, Addison-Wesley, 2003).

Kohut (*Professional Agile Development Process: Real World Development Using SCRUM*, Wrox, 2013), Rubin (*Essential Scrum: A Practical Guide to the Most Popular Agile Process*, Addison-Wesley, 2012), Larman e Vodde (*Scaling Lean and Agile Development: Thinking and Organizational Tools for Large Scale Scrum*, Addison-Wesley, 2008) e Schwaber (*The Enterprise and Scrum*, Microsoft Press, 2007) discutem o uso de Scrum para projetos que têm grande impacto comercial. Os detalhes práticos do Scrum são debatidos por Cohn (*Succeeding with Agile*, Addison-Wesley, 2009) e Schwaber e Beedle (*Agile Software Development with SCRUM*, Prentice-Hall, 2001). Tratados úteis sobre o DSDM foram escritos pelo DSDM Consortium (*DSDM: Business Focused Development*, 2ª ed., Pearson Education, 2003) e Stapleton (*DSDM: The Method in Practice*, Addison-Wesley, 1997).

Livros de Ambler e Lines (*Disciplined Agile Delivery: A Practitioner's Guide to Agile Delivery in the Enterprise*, IBM Press, 2012) e Poppendieck e Poppendieck (*Lean Development: An Agile Toolkit for Software Development Managers*, Addison-Wesley, 2003) dão diretrizes para gerenciar e controlar projetos ágeis. Ambler e Jeffries (*Agile Modeling*, Wiley, 2002) discutem a AM com certa profundidade.

Uma grande variedade de fontes de informação sobre desenvolvimento de software ágil está disponível na Internet. Uma lista atualizada de referências relevantes (em inglês) para o processo ágil pode ser encontrada no site: www.mhhe.com/pressman.

Aspectos humanos da engenharia de software

6

Em uma edição especial da *IEEE Software*, os editores convidados [deS09] fizeram a seguinte observação:

A engenharia de software tem uma fartura de técnicas, ferramentas e métodos projetados para melhorar tanto o processo de desenvolvimento de software quanto o produto final. Aprimoramentos técnicos continuam a surgir e a gerar resultados animadores. No entanto, software não é simplesmente um produto de soluções técnicas adequadas aplicadas a hábitos técnicos inadequados. Software é desenvolvido por pessoas, usado por pessoas e dá suporte à interação entre pessoas. Assim, características, comportamento e cooperação humanos são fundamentais no desenvolvimento prático de software.

Ao longo dos capítulos posteriores a este, vamos discutir as “técnicas, ferramentas e métodos” que resultarão na criação de um produto de software bem-sucedido. Mas, antes disso, é fundamental entender que, sem pessoas habilitadas e motivadas, o sucesso é improvável.

Conceitos-chave

ambientes de desenvolvimento colaborativo (CDEs)	98
atributos da equipe	90
computação em nuvem	97
equipe consistente	90
equipe XP	94
equipes ágeis	93
equipes globais	99
estruturas de equipe	92
mídia social	95
papéis	89
características	88
psicologia	89
toxicidade de equipe	91

PANORAMA

O que é? Todos nós temos a tendência de nos dedicarmos à linguagem de programação mais recente, aos melhores e novos métodos de projeto, ao processo ágil mais moderno ou à impressionante ferramenta de software recém lançada. Mas no frigir dos ovos são *pessoas* que constroem software de computador. E, por isso, os aspectos humanos da engenharia de software frequentemente têm tanto a ver com o sucesso de um projeto quanto a melhor e mais recente tecnologia.

Quem realiza? Indivíduos e equipes realizam o trabalho de engenharia de software. Em alguns casos, apenas uma pessoa é responsável pela maior parte do trabalho, mas no caso de produção de software em nível industrial, uma equipe de pessoas o realiza.

Por que é importante? Uma equipe de software só será bem-sucedida se sua dinâmica estiver correta. Às vezes, os engenheiros de software têm a reputação de não trabalhar bem com outras pessoas. Na verdade, é fundamental que os engenheiros de software de uma equipe trabalhem bem com seus colegas e com outros envolvidos no produto a ser construído.

Quais são as etapas envolvidas? Primeiramente, é preciso entender as características pessoais de um engenheiro de software bem-sucedido e, então, tentar imitá-las. Em seguida, você deve compreender a complexa psicologia do trabalho de engenharia de software para que possa navegar por um projeto sem riscos. Então, precisa entender a estrutura e a dinâmica de uma equipe de software, pois a engenharia de software baseada no trabalho em equipe é comum em um cenário industrial. Por fim, você deve compreender o impacto das mídias sociais, da nuvem e de outras ferramentas colaborativas.

Qual é o artefato? Uma melhor compreensão das pessoas, do processo e do produto final.

Como garantir que o trabalho foi realizado corretamente? Passe um tempo observando como os engenheiros de software bem-sucedidos fazem seu trabalho e ajuste sua abordagem para tirar proveito dos pontos positivos do projeto deles.

"A maioria dos bons programadores faz seu trabalho não porque espera pagamento ou bajulação pública, mas porque é divertido programar."

Linus Torvalds

Quais são as características pessoais de um engenheiro de software competente?

6.1 Características de um engenheiro de software

Então você quer ser engenheiro de software? Obviamente, precisa dominar o material técnico, aprender e aplicar as habilidades exigidas para entender o problema, projetar uma solução eficaz, construir o software e testá-lo com a finalidade de produzir a mais alta qualidade possível. Você precisa gerenciar mudanças, comunicar-se com os envolvidos e usar ferramentas adequadas nas situações apropriadas. Tudo isso é discutido com detalhes mais adiante neste livro.

Mas existem outras coisas igualmente importantes – os aspectos humanos que o tornarão um engenheiro de software competente. Erdogmus [Erd09] identifica sete características pessoais que estão presentes quando um engenheiro de software demonstra comportamento “super profissional”.

Um engenheiro de software competente tem um senso de *responsabilidade individual*. Isso implica a determinação de cumprir suas promessas para colegas, para os envolvidos e para a gerência. Significa que ele fará o que precisar ser feito, quando for necessário, executando um esforço adicional para a obtenção de um resultado bem-sucedido.

Um engenheiro de software competente tem *consciência aguçada* das necessidades dos outros membros de sua equipe, dos envolvidos que solicitaram uma solução de software para um problema existente e dos gerentes que têm controle global sobre o projeto que vai gerar essa solução. É capaz de observar o ambiente em que as pessoas trabalham e de adaptar seu comportamento a ele e às próprias pessoas.

Um engenheiro de software competente é *extremamente honesto*. Se ele vê um projeto falho, aponta os defeitos de maneira construtiva, mas honesta. Se instado a distorcer fatos sobre cronogramas, recursos, desempenho ou outras características do produto ou projeto, opta por ser realista e sincero.

Um engenheiro de software competente mostra *resiliência sob pressão*. Conforme mencionamos anteriormente no livro, a engenharia de software está sempre à beira do caos. A pressão (e o caos que pode resultar) vem em muitas formas – mudanças nos requisitos e nas prioridades, envolvidos ou colegas exigentes, um gerente irrealista ou autoritário. Mas um engenheiro de software competente é capaz de suportar a pressão de modo que seu desempenho não seja prejudicado.

Um engenheiro de software competente tem *elevado senso de lealdade*. De boa vontade, compartilha os créditos com seus colegas. Tenta evitar conflitos de interesse e nunca age no sentido de sabotar o trabalho dos outros.

Um engenheiro de software competente mostra *atenção aos detalhes*. Isso não significa obsessão com a perfeição, mas sugere que ele considera atentamente as decisões técnicas que toma diariamente, em comparação com critérios mais amplos (por exemplo, desempenho, custo, qualidade) que foram estabelecidos para o produto e para o projeto.

Por último, um engenheiro de software competente é pragmático. Reconhece que a engenharia de software não é uma religião na qual devem ser seguidas regras dogmáticas, mas sim uma disciplina que pode ser adaptada de acordo com as circunstâncias.

6.2 A psicologia da engenharia de software

Em um artigo seminal sobre a psicologia da engenharia de software, Bill Curtis e Diane Walz [Cur90] sugerem um modelo comportamental em camadas para desenvolvimento de software (Figura 6.1). No nível individual, a psicologia da engenharia de software se concentra no reconhecimento do problema a ser resolvido, nas habilidades exigidas para solucionar o problema e na motivação necessária para concluir a solução dentro das restrições estabelecidas pelas camadas externas do modelo. Nos níveis da equipe e do projeto, dinâmicas de grupo se tornam o fator dominante. Aqui, a estrutura da equipe e fatores sociais governam o sucesso. A comunicação, a colaboração e a coordenação do grupo são tão importantes quanto as habilidades dos membros individuais da equipe. Nas camadas externas, o comportamento organizacional governa as ações da empresa e sua resposta para o meio empresarial.

No nível das equipes, Sawyer e seus colegas [Saw08] sugerem que elas frequentemente estabelecem fronteiras artificiais que reduzem a comunicação e, como consequência, a eficácia da equipe. Sugerem ainda um conjunto de “papéis que ultrapassam fronteiras”, que permite que os membros de uma equipe de software transponham eficazmente suas fronteiras. Os papéis a seguir podem ser atribuídos explicitamente ou evoluir naturalmente.

- *Embaixador* – representa a equipe para a clientela de fora, com o objetivo de negociar tempo e recursos e obter o retorno dos envolvidos.
- *Patrulha* – cruza a fronteira da equipe para reunir informações organizacionais. “O patrulhamento pode incluir o mapeamento de mercados externos, busca de novas tecnologias, identificação de atividades relevantes fora da equipe e descoberta de focos de concorrência em potencial” [Saw08].

Quais papéis os membros de uma equipe de software desempenham?

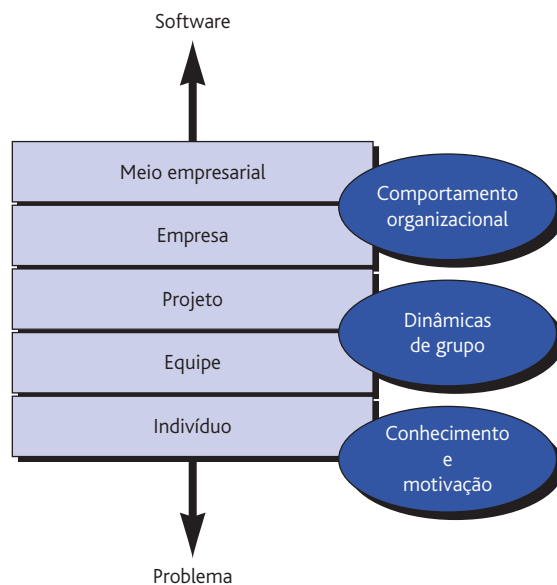


FIGURA 6.1 Um modelo comportamental em camadas para engenharia de software (adaptado de [Cur90]).

- *Guarda* – protege o acesso aos artefatos e outras informações da equipe.
- *Sentinela* – controla o fluxo de informações enviadas para a equipe pelos envolvidos e por outros.
- *Coordenador* – concentra-se na comunicação horizontal entre a equipe e dentro da organização (por exemplo, discutindo um problema de projeto específico com um grupo de especialistas da organização).

6.3 A equipe de software

Em seu livro clássico, *Peopleware*, Tom DeMarco e Tim Lister [DeM98] discutem a coesão de uma equipe de software:

Há uma tendência em se utilizar a palavra *equipe* de forma constante e vaga na área de negócios, denominando qualquer grupo de profissionais designados para trabalharem juntos de “equipe”. Entretanto, muitos deles não se assemelham a equipes. Não há uma definição comum de sucesso nem um espírito de equipe identificável. O que falta é um fenômeno que se denomina *consistência*.

O que é uma equipe “consistente”?

Uma equipe consistente é um grupo de pessoas tão coesas, que o todo é maior que a soma das partes...

Quando uma equipe começa a ser consistente, a probabilidade de sucesso aumenta muito. A equipe pode se tornar imbatível, um rolo compressor de sucesso... Não é preciso gerenciá-la do modo tradicional e, com certeza, não precisará ser motivada. Ela adquire velocidade e ímpeto.

DeMarco e Lister sustentam que os membros de equipes consistentes são significativamente mais produtivos e mais motivados do que a média. Compartilham de um objetivo comum, de uma cultura comum e, em muitos casos, de um senso de pertencimento a uma equipe de elite que os torna únicos.

Não existe nenhum método infalível para se criar uma equipe consistente. Porém, existem atributos normalmente encontrados em equipes de software eficazes.¹ Miguel Carrasco [Car08] sugere que uma equipe de software eficiente deve estabelecer um *senso de propósito*. Por exemplo, se todos os membros da equipe concordam que o objetivo dela é desenvolver software que vai transformar uma categoria de produto e, como consequência, transformar sua empresa em líder do setor, eles têm um forte senso de propósito. Uma equipe eficaz também deve incorporar um *senso de envolvimento* que permita a cada membro sentir que suas qualidades e contribuições são valiosas.

Uma equipe eficaz deve promover um *senso de confiança*. Os engenheiros de software da equipe devem confiar nas habilidades e na competência de seus colegas e gerentes. A equipe deve estimular um *senso de melhoria*, refletindo periodicamente em sua abordagem de engenharia de software e buscando maneiras de melhorar seu trabalho.

¹ Bruce Tuckman observa que as equipes bem-sucedidas passam por quatro fases (Formação, Ataque, Regulamentação e Execução) no caminho para se tornarem produtivas (<http://www.realsoftwaredevelopment.com/7-key-attributes-of-high-performance-software-development-teams/>).

As equipes de software mais eficazes são diversificadas, no sentido de combinarem uma variedade de diferentes qualidades. Técnicos altamente capacitados são complementados por membros que podem ter menos base técnica, mas compreendem melhor as necessidades dos envolvidos.

Porém, nem todas as equipes são eficazes e nem todas são consistentes. Na verdade, muitas sofrem do que Jackman [Jac98] denomina de “toxicidade de equipe”. Ela define cinco fatores que “promovem um ambiente em equipe potencialmente tóxico”: uma atmosfera de trabalho frenética; alto grau de frustração que causa atrito entre os membros da equipe; um processo de software fragmentado ou coordenado de forma deficiente; uma definição nebulosa dos papéis dentro da equipe de software; e contínua e repetida exposição a falhas.

Para evitar um ambiente de trabalho frenético, a equipe deve ter acesso a todas as informações exigidas para cumprir a tarefa. As principais metas e objetivos, uma vez definidos, não devem ser modificados, a menos que seja absolutamente necessário. Uma equipe pode evitar frustrações se lhe for oferecida, tanto quanto possível, responsabilidade para tomada de decisão. Um processo inapropriado (por exemplo, tarefas onerosas ou desnecessárias ou artefatos mal selecionados) pode ser evitado por meio da compreensão do produto a ser desenvolvido, das pessoas que realizam o trabalho e pela permissão para que a equipe selecione o modelo do processo. A própria equipe deve estabelecer seus mecanismos de responsabilidades (revisões técnicas² são excelentes meios para conseguir isso) e definir uma série de abordagens corretivas quando um membro falhar em suas atribuições. E, por fim, a chave para evitar uma atmosfera de derrota consiste em estabelecer técnicas baseadas no trabalho em equipe voltadas para realimentação (feedback) e solução de problemas.

Somando-se às cinco toxinas descritas por Jackman, uma equipe de software frequentemente despende esforços com as diferentes características de seus membros. Uns são extrovertidos; outros, introvertidos. Uns coletam informações intuitivamente, destilando conceitos amplos de fatos disparatados. Outros processam informações linearmente, coletando e organizando detalhes minuciosos dos dados fornecidos. Alguns se sentem confortáveis tomando decisões apenas quando um argumento lógico e ordenado for apresentado. Outros são intuitivos, acostumados a tomar decisões baseadas em percepções. Certos desenvolvedores querem um cronograma detalhado, preenchido por tarefas organizadas que os tornem aptos a ter proximidade com elementos do projeto. Outros, ainda, preferem um ambiente mais espontâneo, no qual resultados e questões abertas são aceitáveis. Alguns trabalham arduamente para conseguir que as etapas sejam concluídas bem antes da data estabelecida, evitando, portanto, estresse na medida em que a data-limite se aproxima, enquanto outros são energizados pela correria em fazer até o último minuto da data-limite. Reconhecer as diferenças humanas, junto com outras diretrizes apresentadas nesta seção, proporciona uma maior probabilidade de se criar equipes consistentes.

Uma equipe de software eficaz é diversificada, preenchida por pessoas que têm senso de propósito, envolvimento, confiança e melhoria.

Por que as equipes não conseguem ser consistentes?

“Nem todo grupo é uma equipe, nem toda equipe é eficaz.”

Glenn Parker

² Revisões técnicas são tratadas em detalhes no Capítulo 20.

6.4 Estruturas de equipe

A melhor estrutura de equipe depende do estilo de gerenciamento das organizações, da quantidade de pessoas na equipe e seus níveis de habilidade e do grau de dificuldade geral do problema. Mantei [Man81] descreve vários fatores que devem ser considerados ao planejarmos a estrutura da equipe de engenharia de software: dificuldade do problema a ser resolvido; “tamanho” do programa (ou programas) resultante em linhas de código ou pontos de função;³ tempo que a equipe irá permanecer reunida (tempo de vida da equipe); até que ponto o problema pode ser modularizado; qualidade e confiabilidade exigidas do sistema a ser construído; rigidez da data de entrega; e grau de sociabilidade (comunicação) exigida para o projeto.

Constantine [Con93] sugere quatro “paradigmas organizacionais” para equipes de engenharia de software:

Quais opções temos ao definir a estrutura de uma equipe de software?

Quais fatores devem ser considerados ao se escolher a estrutura de uma equipe de software?

“Se deseja ser incrementalmente melhor, seja competitivo. Se deseja ser exponencialmente melhor, seja cooperativo.”

Autor desconhecido

1. O *paradigma fechado* estrutura uma equipe em termos de uma hierarquia de autoridade tradicional. Tais equipes podem trabalhar bem em produção de software bastante similar a esforços já feitos no passado, mas se mostrarão menos propícias a ser inovadoras trabalhando sob o paradigma fechado.
2. O *paradigma randômico* estrutura uma equipe de forma mais livre e depende da iniciativa individual de seus membros. Quando for necessária uma inovação ou um avanço tecnológico, as equipes que seguem o paradigma randômico se destacarão. Mas essas equipes podem brigar quando for exigido um “desempenho ordenado”.
3. O *paradigma aberto* procura estruturar a equipe de maneira que consiga alguns dos controles associados ao paradigma fechado, mas também muito da inovação que ocorre ao se usar o paradigma randômico. O trabalho é feito de forma colaborativa, com forte comunicação e tomada de decisão baseada no consenso – características marcantes das equipes de paradigma aberto. As estruturas das equipes de paradigmas abertos são bem adequadas para a solução de problemas complexos, mas não conseguem desempenhar tão eficientemente quanto outras equipes.
4. O *paradigma sincronizado* baseia-se na compartimentalização natural de um problema e organiza os membros da equipe para trabalhar nas partes do problema com pouca comunicação entre si.

Como um comentário histórico final, uma das mais antigas organizações de equipe de software foi uma estrutura de paradigma fechado, denominada originalmente *equipe com um programador-chefe* (principal). Essa estrutura foi primeiramente proposta por Harlan Mills e descrita por Baker [Bak72]. O núcleo da equipe era composto de um *engenheiro sênior* (o programador-chefe), que planejava, coordenava e fazia a revisão de todas as atividades técnicas da equipe, o *peçoal técnico* (normalmente de duas a cinco pessoas), que conduzia as atividades de análise e de desenvolvimento, e um *engenheiro re-*

³ Linhas de código (LOC, lines of code) e pontos de função são medidas do tamanho de um programa de computador e são discutidas no Capítulo 33.

serva que dava suporte ao engenheiro sênior em suas atividades e que podia substituí-lo com perdas mínimas de continuidade. O programador-chefe (principal) podia ter a seu dispor um ou mais especialistas (por exemplo, perito em telecomunicações, desenvolvedor de banco de dados), uma equipe de suporte (por exemplo, codificadores técnicos, pessoal de escritório) e um bibliotecário de software.

Como contraponto à estrutura da equipe de programadores-chefe, o paradigma randômico de Constantine [Con93] sugere a primeira equipe criativa, cuja abordagem de trabalho poderia ser mais bem denominada *anarquia inovadora*. Embora a abordagem de espírito livre para o trabalho de software seja atraente, a energia da criatividade direcionada para uma equipe de alta performance deve ser o objetivo central de uma organização de engenharia de software.

CASASEGURA



Estrutura da equipe

Cena: Escritório de Doug Miller antes do início do projeto do software *CasaSegura*.

Atores: Doug Miller (gerente da equipe de engenharia de software do *CasaSegura*) e Vinod Raman, Jamie Lazar e outros membros da equipe.

Conversa:

Doug: Vocês deram uma olhada no informativo preliminar do *CasaSegura* que o departamento de marketing preparou?

Vinod (balançando afirmativamente a cabeça e olhando para seus companheiros de equipe): Sim, mas temos muitas dúvidas.

Doug: Vamos deixar isso de lado por um momento. Gostaria de conversar sobre como vamos estruturar a equipe, quem será responsável pelo quê...

Jamie: Estou totalmente de acordo com a filosofia ágil, Doug. Acho que devemos ser uma equipe auto-organizada.

Vinod: Concordo. Devido ao cronograma apertado e ao grau de incertezas e pelo fato de todos sermos realmente competentes (risos), parece ser o caminho certo a tomar.

Doug: Tudo bem por mim, mas vocês conhecem o procedimento.

Jamie (sorridente e falando ao mesmo tempo): Tomamos decisões táticas sobre quem faz o que e quando, mas é nossa responsabilidade ter o produto pronto sem atraso.

Vinod: E com qualidade.

Doug: Exatamente. Mas lembrem-se de que há restrições. O marketing define os incrementos de software a serem desenvolvidos – consultando-nos, é claro.

Jamie: E...?

6.5 Equipes ágeis

Ao longo da última década, o desenvolvimento de software ágil (Capítulo 5) tem sido indicado como o antídoto para muitos problemas que se alastraram nas atividades de projeto de software. Relembrando, a filosofia ágil enfatiza a satisfação do cliente e a entrega prévia incremental de software, pequenas equipes de projeto altamente motivadas, métodos informais, mínimos artefatos de engenharia de software e total simplicidade de desenvolvimento.

6.5.1 A equipe ágil genérica

A pequena e altamente motivada equipe de projeto, também denominada *equipe ágil*, adota muitas das características das equipes de software bem-sucedidas, discutidas na seção anterior, e evita muito das toxinas geradoras

Uma equipe ágil é auto-organizada e tem autonomia para planejar e tomar decisões técnicas.

"Propriedades coletivas nada mais são do que uma instância da ideia de que os produtos deveriam ser atribuídos à equipe (ágil), não a indivíduos que compõem a equipe."

Jim Highsmith

Simplifique sempre que puder, mas reconheça que uma "refabricação" (retrabalho, redesenvolvimento) contínua pode absorver tempo e recursos significativos.

de problemas. Entretanto, a filosofia ágil enfatiza a competência individual (membro da equipe) combinada com a colaboração em grupo como fatores críticos de sucesso para a equipe. Cockburn e HighSmith [Coc01a] observam isso ao escreverem:

Se as pessoas do projeto forem boas o suficiente, podem usar praticamente qualquer processo e cumprir sua missão. Se não forem boas o suficiente, nenhum processo irá reparar a sua inadequação. "Pessoas são o trunfo do processo" é uma forma de dizer isso. Entretanto, falta de suporte ao desenvolvedor e ao usuário pode acabar com um projeto – "política é o trunfo de pessoas". Suporte inadequado pode fazer com que até mesmo os bons fracassem na realização de seus trabalhos.

Para uso das competências de cada membro da equipe, para fomentar colaboração efetiva ao longo do projeto, equipes ágeis são auto-organizadas. Uma equipe auto-organizada não mantém, necessariamente, uma estrutura de equipe única, mas usa elementos da aleatoriedade de Constantine, paradigmas abertos e de sincronicidade discutidos na Seção 6.2.

Muitos modelos ágeis de processo (por exemplo, Scrum) dão à equipe ágil autonomia para gerenciar o projeto e tomar as decisões técnicas necessárias à conclusão do trabalho. O planejamento é mantido em um nível mínimo, e a equipe tem a permissão para escolher sua própria abordagem (por exemplo, processo, método, ferramentas), limitada somente pelos requisitos de negócio e pelos padrões organizacionais. Conforme o projeto prossegue, a equipe se auto-organiza, concentrando-se em competências individuais para maior benefício do projeto em um determinado ponto do cronograma. Para tanto, uma equipe ágil pode fazer reuniões de equipe diariamente a fim de coordenar e sincronizar as atividades que devem ser realizadas naquele dia.

Com base na informação obtida durante essas reuniões, a equipe adapta sua abordagem para incrementar o trabalho. A cada dia que passa, auto-organizações contínuas e colaboração conduzem a equipe em direção a um incremento de software completo.

6.5.2 A equipe XP

Beck [Bec04a] define um conjunto de cinco *valores* que estabelecem as bases para todo trabalho realizado como parte da programação extrema (XP) – comunicação, simplicidade, feedback (realimentação ou retorno), coragem e respeito. Cada um desses valores é usado como um direcionamento para as atividades, ações e tarefas específicas da XP.

Para conseguir a *comunicação* efetiva entre a equipe ágil e outros envolvidos (por exemplo, estabelecer as funcionalidades necessárias para o software), a XP enfatiza a colaboração estreita, embora informal (verbal), entre clientes e desenvolvedores, o estabelecimento de metáforas⁴ eficazes para comunicar conceitos importantes, feedback (realimentação) contínuo e evita documentação volumosa como um meio de comunicação.

⁴ No contexto da XP, *metáfora* é "uma história que todos – clientes, programadores e gerentes – podem contar sobre como o sistema funciona" [Bec04a].

Para obter a *simplicidade*, a equipe ágil projeta apenas para as necessidades imediatas, em vez de considerar as necessidades futuras. O objetivo é criar um projeto simples que possa ser facilmente implementado em código. Se o projeto tiver que ser melhorado, ele poderá ser *refabricado*⁵ mais tarde.

O *feedback* provém de três fontes: do próprio software implementado, do cliente e de outros membros da equipe de software. Por meio da elaboração do projeto e da implementação de uma estratégia de testes eficaz (Capítulos 22 a 26), o software (via resultados de testes) propicia um feedback para a equipe ágil. A equipe faz uso do *teste unitário* como tática de teste principal. À medida que cada classe é desenvolvida, a equipe desenvolve um teste unitário para testar cada operação de acordo com a funcionalidade especificada. À medida que um incremento é entregue a um cliente, as *histórias de usuários* ou *casos de uso* (Capítulo 9) implementados pelo incremento são utilizados para realizar os testes de aceitação. O grau em que o software implementa o produto, a função e o comportamento do caso em uso é uma forma de feedback. Por fim, conforme novas necessidades surgem como parte do planejamento iterativo, a equipe dá ao cliente um rápido feedback referente ao impacto nos custos e no cronograma.

Beck [Bec04a] afirma que a obediência estrita a certas práticas da XP exige *coragem*. Uma palavra melhor poderia ser *disciplina*. Por exemplo, frequentemente, há uma pressão significativa para a elaboração do projeto pensando em futuros requisitos. A maioria das equipes de software sucumbe, argumentando que “projetar para amanhã” poupará tempo e esforço no longo prazo. Uma equipe XP deve ter disciplina (coragem) para projetar para hoje, reconhecendo que as necessidades futuras podem mudar significativamente, exigindo, conseqüentemente, um retrabalho substancial em relação ao projeto e ao código implementado.

Ao buscar cada um desses valores, a equipe XP fomenta *respeito* entre seus membros, entre outros envolvidos e os membros da equipe e, indiretamente, para o próprio software. Conforme consegue entregar com sucesso incrementos de software, a equipe desenvolve cada vez mais respeito pelo processo XP.

“A XP é a resposta para a pergunta: ‘Qual é o mínimo possível que se pode realizar e mesmo assim desenvolver um software excelente?’.”

Anônimo

6.6 O impacto da mídia social

E-mail, mensagens de texto e videoconferência se tornaram atividades onipresentes no trabalho de engenharia de software. Mas, na verdade, esses mecanismos de comunicação nada mais são do que substitutos ou suplementos modernos para o contato face a face. A mídia social é diferente.

⁵ A refabricação permite que o engenheiro de software aperfeiçoe a estrutura interna de um projeto (ou código-fonte) sem alterar sua funcionalidade ou comportamento externos. Basicamente, a refabricação pode ser usada para melhorar a eficiência, a legibilidade ou o desempenho de um projeto ou o código que implementa um projeto.

Begel [Beg10] e seus colegas tratam do crescimento e da aplicação de mídia social na engenharia de software ao escreverem:

Os processos sociais em torno do desenvolvimento de software são... altamente dependentes da capacidade dos engenheiros de encontrar e associar-se a pessoas que compartilhem objetivos semelhantes e habilidades complementares para harmonizar a comunicação e as preferências de cada membro da equipe, colaborar e coordenar durante todo o ciclo de vida do software e defender o sucesso de seu produto no mercado.

De certa forma, essa “associação” pode ser tão importante quanto a comunicação face a face. O valor da mídia social cresce à medida que o tamanho da equipe aumenta, e é ainda mais ampliado quando a equipe está geograficamente dispersa.

Primeiro, é definida uma rede social para um projeto de software. Usando a rede, a equipe de software pode se basear na experiência coletiva de seus membros, dos envolvidos, dos técnicos, especialistas e outros executivos que tenham sido convidados a participar da rede (se for privada) ou de qualquer parte interessada (se for pública). E isso pode acontecer quando surgir uma questão, uma dúvida ou um problema. Existem várias formas diferentes de mídia social, e cada uma ocupa seu lugar no trabalho de engenharia de software.

Um *blog* pode ser usado para postar uma série de artigos breves, descrevendo aspectos importantes de um sistema ou expressando opiniões sobre recursos ou funções que ainda precisam ser desenvolvidos. Também é importante observar que “as empresas de software frequentemente usam blogs para compartilhar informações técnicas e opiniões com seus funcionários e, muito proveitosamente, com seus clientes, tanto internos quanto externos” [Beg10].

Microblogs (como o Twitter) permitem que um membro de uma rede de engenharia de software poste mensagens breves para seus seguidores. Como as mensagens são instantâneas e podem ser lidas a partir de todas as plataformas móveis, a dispersão da informação se dá quase em tempo real. Isso permite a uma equipe de software convocar uma reunião inesperada caso surja uma questão, solicitar ajuda especializada se ocorrer um problema ou informar os envolvidos sobre algum aspecto do projeto.

Fóruns online dirigidos permitem aos participantes postar perguntas, opiniões, estudos de caso ou qualquer outra informação relevante. Uma pergunta técnica pode ser postada e, em poucos minutos, frequentemente várias “respostas” estão disponíveis.

Sites de rede social (como Facebook, LinkedIn) permitem ligações por laços de amizade entre desenvolvedores de software e técnicos próximos. Isso permite aos “amigos” em um site de rede social conhecer amigos de amigos que podem ter conhecimento ou especialidade relacionada ao domínio de aplicação ou problema a ser resolvido. Redes privadas especializadas, baseadas no paradigma das redes sociais, podem ser usadas dentro de uma organização.

Grande parte das mídias sociais permite a formação de “comunidades” de usuários com interesses semelhantes. Por exemplo, uma comunidade de engenheiros de software especializados em sistemas embarcados em tempo real poderia ser uma maneira interessante para uma pessoa ou equipe que esteja

“Se conteúdo é rei, a conversa é rainha.”

John Munsell

trabalhando nessa área estabelecer relações que melhorariam seu trabalho. À medida que uma comunidade cresce, os participantes discutem tendências tecnológicas, cenários de aplicação, novas ferramentas e outros conhecimentos da engenharia de software. Por fim, os *sites de social bookmarking* (como Delicious, Stumble, CiteULike) permitem a um engenheiro ou equipe de software recomendar recursos baseados na Web que podem ser interessantes para uma comunidade de mídia social de pessoas com a mesma opinião.

É muito importante mencionar que questões de privacidade e segurança não devem ser desprezadas ao se usar mídia social no trabalho de engenharia de software. Grande parte do trabalho realizado por engenheiros de software pode ser propriedade de seus empregadores, e a divulgação poderia ser muito prejudicial. Por isso, os benefícios da mídia social devem ser ponderados em relação à possibilidade de revelação descontrolada de informações privativas.

6.7 Engenharia de software usando a nuvem

A computação em nuvem oferece um mecanismo de acesso a todos os produtos, artefatos e informações relacionados ao projeto da engenharia de software. Ela funciona em qualquer lugar e elimina a dependência de dispositivos, o que já foi uma restrição para muitos projetos de software. Também permite que os membros de uma equipe de software façam testes de baixo risco e independentes de plataforma de novas ferramentas de software e forneçam feedback sobre elas. Permite ainda novas possibilidades para a distribuição e testes de software beta. Ela oferece o potencial de abordagens aprimoradas de gerenciamento de conteúdo e configuração (Capítulo 29).

Como a computação em nuvem pode fazer essas coisas, ela tem o potencial de influenciar o modo como os engenheiros de software organizam suas equipes, como realizam seu trabalho, como se comunicam e se associam e o modo de gerenciar projetos de software. As informações de engenharia de software desenvolvidas por um membro da equipe podem estar instantaneamente disponíveis para todos os membros, independentemente da plataforma que os outros estejam usando ou de sua localização.

Basicamente, a dispersão da informação é significativamente acelerada e ampliada. Isso muda a dinâmica da engenharia de software e pode ter um impacto profundo em seus aspectos humanos.

Porém, a computação em nuvem em um ambiente de engenharia de software tem seus riscos [The13]. A nuvem é dispersa por muitos servidores, e a arquitetura e os serviços frequentemente estão fora do controle de uma equipe de software. Como consequência, existem vários pontos de falha, apresentando riscos à confiabilidade e à segurança. À medida que o número de serviços fornecidos pela nuvem aumenta, a complexidade relativa do ambiente de desenvolvimento de software também aumenta. Cada um desses serviços funciona bem com outros serviços, possivelmente de outros fornecedores? Isso apresenta um risco à capacidade de operação conjunta para serviços da nuvem. Por último, se a nuvem se torna o ambiente de desenvolvimento, os serviços devem enfatizar a usabilidade e o desempenho. Às vezes, esses atributos entram em conflito com a segurança, privacidade e confiabilidade.

"Eles não a chamam mais de Internet, chamam de computação em nuvem. Não vou mais me opor ao nome. Chame-a como quiser."

Larry Ellison

A nuvem é um poderoso repositório de informações sobre engenharia de software, mas você deve considerar as questões de controle de alteração discutidas no Capítulo 29.

Porém, do ponto de vista humano, a nuvem oferece bem mais benefícios do que riscos para os engenheiros de software. Dana Gardner [Gar09] resume os benefícios (com um alerta):

Tudo que está relacionado aos aspectos sociais ou colaborativos do desenvolvimento de software serviu bem para a nuvem. Gerenciamento de projeto, cronogramas, listas de tarefas (checklists), requisitos e gerenciamento de defeitos; tudo convém, pois estão no grupo principal das funções, no qual a comunicação é essencial para manter os projetos sincronizados e todos os membros da equipe – onde quer que estejam – literalmente no mesmo canal. Evidentemente, há uma importante advertência aqui – se sua empresa projeta software embarcado nos produtos, ele não é um bom candidato para a nuvem: imagine pôr as mãos nos planos de projeto da Apple para a próxima versão do iPhone.

Como Gardner declara, uma das principais vantagens da nuvem é sua capacidade de melhorar os “aspectos sociais e colaborativos do desenvolvimento de software”. Na próxima seção, você vai saber um pouco mais sobre ferramentas colaborativas.

6.8 Ferramentas de colaboração

Fillipo Lanubile e seus colegas [Lan10] sugerem que os ambientes de desenvolvimento de software (SDEs, software development environments) do último século se transformaram em *ambientes de desenvolvimento colaborativo* (CDEs; *collaborative development environments*).⁶ Eles declaram:

Ferramentas são essenciais para a colaboração entre os membros da equipe, permitindo a facilitação, automação e controle do processo de desenvolvimento inteiro. O suporte de ferramentas adequado é particularmente necessário na engenharia de software global, pois a distância agrava os problemas de coordenação e controle, direta ou indiretamente, por meio de seus efeitos negativos sobre a comunicação.

Muitas das ferramentas usadas em um CDE não são diferentes das usadas para ajudar nas atividades de engenharia de software discutidas nas Partes II, III e IV do livro. Mas um CDE digno de consideração também fornece um conjunto de serviços especificamente projetados para melhorar o trabalho colaborativo [Fok10]. Esses serviços incluem:

- Um repositório com a identificação clara do projeto que permita a equipe de projeto armazenar todos os artefatos e outras informações de um modo que melhore a segurança e a privacidade, permitindo o acesso apenas a pessoas autorizadas.
- Um *calendário* para coordenar reuniões e outros eventos do projeto.
- *Modelos* que permitam aos membros da equipe criar artefatos com aparência e estrutura padronizados.
- *Suporte de métricas* que monitorem as contribuições de cada membro da equipe de maneira quantitativa.

Quais serviços genéricos são encontrados em ambientes de desenvolvimento colaborativo?

⁶ O termo *ambiente de desenvolvimento colaborativo* (CDE) foi cunhado por Grady Booch [Boo02].

- *Análise de comunicação* que monitore a comunicação entre a equipe e isole padrões que possam significar problemas ou questões que precisam ser resolvidos.
- *Agrupamento de artefatos* que organize os artefatos e outros produtos do projeto de uma maneira que responda a perguntas como: “O que causou uma mudança em particular, quem conhece um produto específico que possivelmente deve ser consultado sobre alterações nele e como o trabalho de um membro [equipe] poderia afetar o de outras pessoas?” [Fok10].

FERRAMENTAS DO SOFTWARE



Ambientes de desenvolvimento colaborativo

Objetivo: À medida que o desenvolvimento de software se torna global, as equipes de software precisam de mais ferramentas. Elas precisam de um conjunto de serviços que permitam aos membros da equipe colaborar de forma local e em longas distâncias.

Mecanismos: Ferramentas e serviços dessa categoria permitem a uma equipe estabelecer mecanismos para trabalho colaborativo. Um CDE implementará muitos ou todos os serviços descritos na Seção 6.6, ao mesmo tempo fornecendo acesso às ferramentas de engenharia de software convencionais

para gerenciamento de processo (Capítulo 4) discutidas ao longo deste livro.

Ferramentas representativas:⁷

GForge – um ambiente colaborativo que contém recursos de gerenciamento de projeto e código (<http://gforge.com/gf/>).

OneDesk – oferece um ambiente colaborativo que cria e gerencia uma área de trabalho de projetos para desenvolvedores e envolvidos (www.onedesk.com).

Rational Team Concert – um sistema detalhado de gerenciamento de ciclo de vida colaborativo (<http://www-01.ibm.com/software/rational/products/rtc/>).

6.9 Equipes globais

No campo do software, globalização significa mais do que a transferência de bens e serviços entre fronteiras internacionais. Nas últimas décadas, foi construído um crescente número de importantes produtos de software, por equipes muitas vezes sediadas em diferentes países. Essas equipes de desenvolvimento de software global (GSD, global software development) têm muitas das características de uma equipe de software convencional (Seção 6.4), mas uma equipe GSD tem outros desafios exclusivos, que incluem coordenação, colaboração, comunicação e tomada de decisão especializada. Abordagens de coordenação, colaboração e comunicação foram discutidas anteriormente neste capítulo. A tomada de decisão em todas as equipes de software é complicada por quatro fatores [Gar10]:

- Complexidade do problema.
- Incerteza e risco associados à decisão.
- A lei das consequências não intencionais (isto é, uma decisão associada ao trabalho tem um efeito colateral sobre outro objetivo do projeto).
- Diferentes visões do problema que levam a diferentes conclusões sobre o caminho a seguir.

“Cada vez mais, em qualquer empresa, os gerentes lidam com diferentes culturas. As empresas estão se tornando globais, mas as equipes estão sendo divididas e espalhadas por todo o planeta.”

**Carlos Ghosn,
Nissan**

⁷ A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

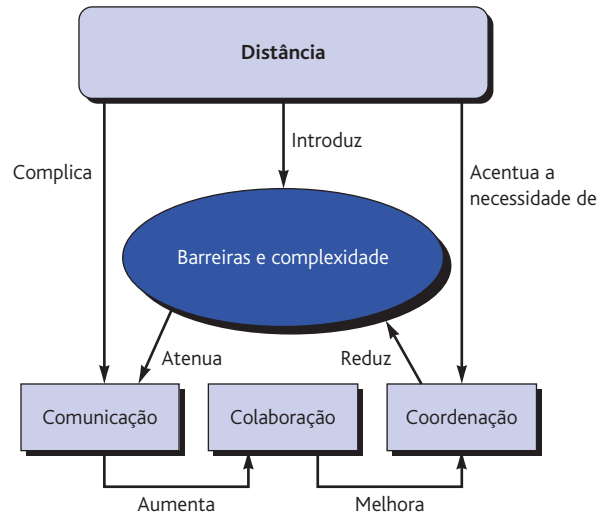


FIGURA 6.2 Fatores que afetam uma equipe GSD (adaptado de [Cas06]).

Para uma equipe GSD, os desafios associados à coordenação, colaboração e comunicação podem ter um efeito profundo na tomada de decisão. A Figura 6.2 ilustra o impacto da distância sobre os desafios enfrentados por uma equipe GSD. A distância complica a comunicação, mas, ao mesmo tempo, acentua a necessidade de coordenação. A distância também introduz barreiras e complexidade, motivadas por diferenças culturais. Barreiras e complexidade enfraquecem a comunicação (isto é, a relação sinal-ruído diminui). Os problemas inerentes a essa dinâmica podem resultar em um projeto instável.

Embora não exista nenhuma solução milagrosa que possa corrigir totalmente as relações indicadas na Figura 6.2, o uso de CDEs eficazes (Seção 6.6) pode ajudar a reduzir o impacto da distância.

6.10 Resumo

Um engenheiro de software de sucesso precisa ter habilidades técnicas. Além disso, deve assumir a responsabilidade por seus compromissos, estar ciente das necessidades de seus colegas, ser honesto em sua avaliação do produto e do projeto, mostrar resiliência sob pressão, tratar seus colegas de modo correto e mostrar atenção aos detalhes.

A psicologia da engenharia de software engloba o conhecimento e a motivação individuais, a dinâmica de grupo de uma equipe de software e o comportamento organizacional da empresa. Para melhorar a comunicação e a colaboração, os membros de uma equipe de software podem assumir papéis que ultrapassam fronteiras.

Uma equipe de software bem-sucedida (“consistente”) é mais produtiva e motivada do que a média. Para ser eficaz, uma equipe de software deve ter senso de propósito, envolvimento, confiança e melhoria. Além disso, a equipe deve evitar a “toxicidade” caracterizada por uma atmosfera de trabalho frenética e frustrante, um processo de software inadequado, uma definição nebulosa dos papéis na equipe e contínua exposição a falhas.

Existem muitas estruturas de equipe diferentes. Algumas equipes são organizadas hierarquicamente, enquanto outras preferem uma estrutura livre que conte com a iniciativa individual. As equipes ágeis endossam a filosofia ágil e geralmente têm mais autonomia do que as equipes de software mais convencionais. As equipes ágeis enfatizam a comunicação, simplicidade, feedback, coragem e respeito.

A mídia social está se tornando parte de muitos projetos de software. Blogs, microblogs, fóruns e recursos de rede social ajudam a formar uma comunidade de engenharia de software que se comunica e é coordenada de modo mais eficaz.

A computação em nuvem tem o potencial de influenciar o modo como os engenheiros de software organizam suas equipes, como realizam seu trabalho, como se comunicam e se associam e o modo de gerenciar projetos de software. Em situações nas quais a nuvem pode melhorar os aspectos sociais e colaborativos do desenvolvimento de software, seus benefícios superam em muito os riscos.

Os ambientes de desenvolvimento colaborativo contêm vários serviços que melhoram a comunicação e a colaboração de uma equipe de software. Esses ambientes são particularmente úteis para desenvolvimento de software global, em que a separação geográfica pode criar barreiras para uma engenharia de software bem-sucedida.

Problemas e pontos a ponderar

- 6.1 Com base em sua própria observação de pessoas que são excelentes desenvolvedoras de software, cite três qualidades da personalidade que pareçam ser comuns entre elas.
- 6.2 Como você pode ser “extremamente honesto” e ainda não ser percebido (pelos outros) como insultante ou agressivo?
- 6.3 Como uma equipe de software constrói “fronteiras artificiais” que reduzem sua capacidade de se comunicar com outros?
- 6.4 Escreva um breve cenário descrevendo cada um dos “papéis que ultrapassam fronteiras” estudadas na Seção 6.2.
- 6.5 Na Seção 6.3, mencionamos que senso de propósito, envolvimento, confiança e melhoria são atributos fundamentais para equipes de software eficazes. Quem é responsável por instilar esses atributos quando uma equipe é formada?
- 6.6 Qual dos quatro paradigmas organizacionais para equipes (Seção 6.4) você acha o mais eficaz (a) para o departamento de TI em uma grande companhia de seguros; (b) para um grupo de engenharia de software em um importante fornecedor militar; (c) para um grupo de software que constrói jogos de computador; (d) para uma grande empresa de software? Explique o motivo das escolhas que você fez.
- 6.7 Se você tivesse de escolher um atributo de uma equipe ágil que a tornasse diferente de uma equipe de software convencional, qual seria?
- 6.8 Das formas de mídia social descritas para o trabalho de engenharia de software na Seção 6.6, qual você acha a mais eficaz e por quê?
- 6.9 Escreva um cenário no qual os membros da equipe do *CasaSegura* utilizam uma ou mais formas de mídia social como parte de seu projeto de software.
- 6.10 Atualmente, a nuvem é um dos conceitos mais badalados no mundo da computação. Descreva como ela pode agregar valor para uma organização de engenharia de software, com referência específica aos serviços especialmente destinados a melhorar o trabalho de engenharia de software.

6.11 Pesquise uma das ferramentas de CDE mencionadas no quadro da Seção 6.8 (ou uma ferramenta designada por seu professor) e prepare uma breve apresentação de seus recursos para sua classe.

6.12 Com referência à Figura 6.2, por que a distância complica a comunicação? Por que acentua a necessidade de coordenação? Por que tipos de barreiras e complexidade são introduzidos pela distância?

Leituras e fontes de informação complementares

Embora muitos livros tenham tratado dos aspectos humanos da engenharia de software, dois deles podem ser legitimamente chamados “clássicos”. Jerry Weinberg (*The Psychology of Computer Programming*, Silver Anniversary Edition, Dorset House, 1998) foi o primeiro a considerar a psicologia das pessoas que constroem software de computador. Tom DeMarco e Tim Lister (*Peopleware: Productive Projects and Teams*, 2ª ed., Dorset House, 1999) argumentam que os principais desafios no desenvolvimento de software são humanos, não técnicos.

Observações interessantes sobre os aspectos humanos da engenharia de software também foram feitas por Mantle e Lichty (*Managing the Unmanageable: Rules, Tools, and Insights for Managing Software People and Teams*, Addison-Wesley, 2012), Fowler (*The Passionate Programmer*, Pragmatic Bookshelf, 2009), McConnell (*Code Complete*, 2ª ed., Microsoft Press, 2004), Brooks (*The Mythical Man-Month*, 2ª ed., Addison-Wesley, 1999) e Hunt e Thomas (*The Pragmatic Programmer*, Addison-Wesley, 1999). Tomayko e Hazzan (*Human Aspects of Software Engineering*, Charles River Media, 2004) tratam da psicologia e da sociologia da engenharia de software, com ênfase em XP.

Os aspectos humanos do desenvolvimento ágil foram tratados por Rasmussen (*The Agile Samurai*, Pragmatic Bookshelf, 2010) e Davies (*Agile Coaching*, Pragmatic Bookshelf, 2010). Importantes aspectos das equipes ágeis são considerados por Adkins (*Coaching Agile Teams*, Addison-Wesley, 2010) e Derby, Larsen e Schwaber (*Agile Retrospectives: Making Good Teams Great*, Pragmatic Bookshelf, 2006).

A solução de problemas é uma atividade exclusivamente humana e é tratada em livros de Adair (*Decision Making and Problem Solving Strategies*, Kogan Page, 2010), Roam (*Unfolding the Napkin*, Portfolio Trade, 2009) e Wananabe (*Problem Solving 101*, Portfolio Hardcover, 2009).

Diretrizes para facilitar a colaboração dentro de uma equipe de software são apresentadas por Tabaka (*Collaboration Explained*, Addison-Wesley, 2006). Rosen (*The Culture of Collaboration*, Red Ape Publishing, 2009), Hansen (*Collaboration*, Harvard Business School Press, 2009) e Sawyer (*Group Genius: The Creative Power of Collaboration*, Basic Books, 2007) apresentam estratégias e diretrizes práticas para melhorar a colaboração em equipes técnicas.

Promover a inovação humana é o tema de livros de Gray, Brown e Macanufo (*Game Storming*, O'Reilly Media, 2010), Duggan (*Strategic Intuition*, Columbia University Press, 2007) e Hohmann (*Innovation Games*, Addison-Wesley, 2006).

Uma visão geral do desenvolvimento de software global é apresentada por Ebert (*Global Software and IT: A Guide to Distributed Development, Projects, and Outsourcing*, Wiley-IEEE Computer Society Press, 2011). Mite e seus colegas (*Agility Across Time and Space: Implementing Agile Methods in Global Software Projects*, Springer, 2010) editaram uma antologia que trata do uso de equipes ágeis no desenvolvimento global.

Uma ampla variedade de fontes de informação que discutem os aspectos humanos da engenharia de software está disponível na Internet. Uma lista atualizada de referências relevantes (em inglês) para o processo de software pode ser encontrada no site: www.mhhe.com/pressman.

PARTE



Modelagem

Nesta parte do livro, você vai aprender os princípios, conceitos e métodos usados para criar modelos de requisitos e de projeto de alta qualidade. Estas questões são tratadas nos capítulos que seguem:

- Quais conceitos e princípios orientam a prática da engenharia de software?
- O que é engenharia de requisitos e quais são os conceitos subjacentes que levam a uma análise de requisitos adequada?
- Como é criado o modelo de requisitos e quais são seus elementos?
- Quais são os elementos de um bom projeto?
- Como o projeto arquitetural estabelece uma estrutura para todas as demais ações de projeto e quais são os modelos utilizados?
- Como projetar componentes de software de alta qualidade?
- Quais conceitos, modelos e métodos são aplicados quando uma interface para o usuário é projetada?
- O que é projeto baseado em padrões?
- Quais estratégias e métodos especializados são usados para projetar WebApps?
- Quais estratégias e métodos especializados são usados para projetar aplicativos móveis?

Assim que essas perguntas forem respondidas, você estará mais bem preparado para a prática da engenharia de software.

7 Princípios que orientam a prática

Conceitos-chave

prática	104
princípios fundamentais	106
princípios da codificação	122
princípios da comunicação	110
princípios da modelagem	114
princípios da modelagem viva	120
princípios da disponibilização	124
princípios da modelagem de projetos	117
princípios da modelagem de requisitos	116
princípios do planejamento	112
princípios de testes	123
processo	106

Em um livro que pesquisa o cotidiano e as reflexões dos engenheiros de software, Ellen Ullman [Ull97] descreve um pouco do cotidiano à medida que relata os pensamentos de um profissional que está sob pressão:

Não tenho ideia de que horas são. Não há janelas neste escritório, nem relógio, somente o LED vermelho de um micro-ondas piscando 12:00, 12:00, 12:00, 12:00. Joel e eu estamos programando há dias. Temos um defeito, um demônio teimoso... Assim, o pisca-pisca vermelho que não mostra a hora parece correto, como se estivesse lendo nossos cérebros em voz alta, que, de alguma forma, acabaram sincronizados na mesma intermitência do LED...

Em que estamos trabalhando?... Os detalhes me escapam neste instante. Podemos estar ajudando os pobres ou ajustando uma série de rotinas de baixo nível para verificar os bits de um protocolo de distribuição de dados. Não me importo. Deveria me importar; em outro momento da minha existência – talvez quando sairmos desta sala cheia de computadores –, vou me importar muitíssimo com o porquê, para quem e para qual finalidade estou desenvolvendo software. Mas, neste exato momento, não. Ultrapassei uma camada na qual o mundo real e seus usuários não mais importam. Sou um engenheiro de software...

PANORAMA

tos, métodos e ferramentas que devem ser considerados no planejamento e desenvolvimento de um software. Princípios que direcionam a ação estabelecem a infraestrutura a partir da qual a engenharia de software é conduzida.

Quem realiza? Praticantes (engenheiros de software) e seus coordenadores (gerentes) desenvolvem uma variedade de tarefas de engenharia de software.

Por que é importante? O processo de software propicia, a todos os envolvidos na criação de um sistema computacional ou produto para computador, um roteiro para conseguir chegar a um destino com sucesso. A prática fornece o detalhamento necessário para seguir ao longo da estrada; aponta onde estão as pontes, as barreiras, as bifurcações; auxilia a compreender os conceitos e princípios que devem ser entendidos e seguidos para que se dirija com rapidez e segurança; e orienta quanto a como dirigir, onde diminuir e aumentar a

velocidade. No contexto da engenharia de software, a prática consiste no que se realiza diariamente, à medida que o software evolui da ideia para a realidade.

Quais são as etapas envolvidas? Aplicam-se três elementos práticos, independentemente do modelo de processo escolhido. São eles: princípios, conceitos e métodos. Um quarto elemento de prática – ferramentas – sustenta a aplicação dos métodos.

Qual é o artefato? A prática engloba as atividades técnicas que produzem todos os artefatos definidos pelo modelo de processo de software escolhido.

Como garantir que o trabalho foi realizado corretamente? Primeiro, compreenda completamente os princípios aplicados ao trabalho (por exemplo, projeto) que está sendo realizado no momento. Em seguida, esteja certo de que escolheu um método apropriado, certifique-se de ter compreendido como aplicá-lo, use ferramentas automatizadas quando forem apropriadas à tarefa e seja inflexível quanto à necessidade de técnicas para garantir a qualidade dos artefatos produzidos.

Uma imagem sombria da engenharia de software, sem dúvida, mas muitos leitores deste livro se identificarão com ela.

As pessoas que criam software praticam a arte ou o ofício ou a disciplina¹ que é a engenharia de software. Mas em que consiste a “prática” de engenharia de software? Genericamente, *prática* é um conjunto de conceitos, princípios, métodos e ferramentas aos quais um engenheiro de software recorre diariamente. A prática permite que coordenadores (gerentes) gerenciem os projetos e que engenheiros de software criem programas de computador. A prática preenche um modelo de processo de software com os recursos técnicos e de gerenciamento necessários para realizar o trabalho. Transforma uma abordagem desfocada e confusa em algo mais organizado, mais efetivo e mais propenso a ser bem-sucedido.

Diversos aspectos da engenharia de software serão examinados ao longo do livro. Neste capítulo, nosso foco estará nos princípios e conceitos que norteiam a prática da engenharia de software em geral.

7.1 Conhecimento da engenharia de software

Em um editorial publicado na *IEEE Software*, Steve McConnell [McC99] fez o seguinte comentário:

Muitos desenvolvedores (de software) veem o conhecimento da engenharia de software quase que exclusivamente como um conhecimento de tecnologias específicas: Java, Perl, HTML, C++, Linux, Windows NT e assim por diante. Ter conhecimento dos detalhes tecnológicos específicos é necessário para programar. Se alguém o contrata para desenvolver um programa em C++, você tem de saber algo sobre C++ para fazer seu programa funcionar.

É comum as pessoas dizerem que o conhecimento sobre desenvolvimento de software dura três anos: metade do que você precisa saber hoje estará obsoleto daqui a três anos. No que diz respeito ao conhecimento de tecnologia, isso provavelmente está. Mas há outro tipo de conhecimento sobre desenvolvimento de software – um tipo visto como “princípios da engenharia de software” – que não tem três anos de duração. Tais princípios provavelmente servirão ao programador profissional durante toda a sua carreira.

McConnell continua afirmando que a base do conhecimento em engenharia de software (por volta do ano 2000) evoluiu para uma “essência estável” que ele estimou representar cerca de “75% do conhecimento necessário para desenvolver um sistema complexo”. No entanto, no que consiste essa essência estável?

Desde então, temos visto a evolução de novos sistemas operacionais, como iOS ou Android, e linguagens, como Java, Python e C#.

Mas, como indica McConnell, princípios essenciais – ideias elementares que guiam engenheiros de software em seus trabalhos – ainda fornecem uma infraestrutura a partir da qual os modelos de engenharia de software, métodos e ferramentas podem ser aplicados e avaliados.

¹ Certos autores argumentam que um desses termos exclui os outros. Na realidade, a engenharia de software se aplica aos três.

7.2 Princípios fundamentais

"Teoricamente, não há diferença entre a teoria e a prática. Porém, na prática, ela existe."

**Jan van de
Snepscheut**

A engenharia de software é norteada por um conjunto de princípios fundamentais que ajudam na aplicação de um processo de software significativo e na execução de métodos de engenharia de software eficazes. No nível do processo, os princípios fundamentais estabelecem uma infraestrutura filosófica que guia uma equipe de software à medida que desenvolve atividades de apoio e estruturais, navega no fluxo do processo e cria um conjunto de artefatos de engenharia de software. Quanto ao nível relativo à prática, os princípios estabelecem uma série de valores e regras que servem como guia ao se analisar um problema, projetar uma solução, implementar e testar uma solução e, por fim, disponibilizar o software para a sua comunidade de usuários.

No Capítulo 2, identificamos uma série de princípios fundamentais que abrange o processo e a prática da engenharia de software: (1) fornecer valor aos usuários, (2) simplificar, (3) manter a visão (do produto e do projeto), (4) reconhecer que outros usuários consomem (e devem entender) o que se produz, (5) manter abertura para o futuro, (6) planejar antecipadamente para a reutilização e (7) raciocinar! Embora tais princípios gerais sejam importantes, são caracterizados em um nível tão elevado de abstração que, às vezes, torna-se difícil a tradução para a prática diária da engenharia de software. Nas subseções seguintes, há um maior detalhamento dos princípios essenciais que conduzem o processo e a prática.

7.2.1 Princípios que orientam o processo

Na Parte I deste livro, discutimos a importância do processo de software e descrevemos os diferentes modelos de processos propostos para o trabalho de engenharia de software. Independentemente de um modelo ser linear ou iterativo, prescritivo ou ágil, ele pode ser caracterizado usando-se uma metodologia de processo genérica aplicável a todos os modelos de processo. O conjunto de princípios fundamentais apresentados a seguir pode ser aplicado à metodologia e, por extensão, a todos os processos de software.

Todo projeto e toda equipe são únicos. Isso significa que se deve adaptar o processo para que melhor se ajuste às suas necessidades.

Princípio 1. *Seja ágil.* Não importa se o modelo de processo que você escolheu é prescritivo ou ágil, os princípios básicos do desenvolvimento ágil devem comandar sua abordagem. Todo aspecto do trabalho deve enfatizar a economia de ações – mantenha a abordagem técnica tão simples quanto possível, mantenha os produtos tão concisos quanto possível e tome decisões localmente sempre que possível.

Princípio 2. *Concentre-se na qualidade em todas as etapas.* A condição final de toda atividade, ação e tarefa do processo deve se concentrar na qualidade do produto.

Princípio 3. *Esteja pronto para adaptações.* Processo não é uma experiência religiosa e não há espaço para dogmas. Quando necessário, adapte sua abordagem às restrições impostas pelo problema, pelas pessoas e pelo próprio projeto.

Princípio 4. Monte uma equipe eficiente. O processo e a prática de engenharia de software são importantes, mas o fator mais importante são as pessoas. Forme uma equipe que se organize automaticamente, que tenha confiança e respeito mútuos.²

Princípio 5. Estabeleça mecanismos para comunicação e coordenação. Os projetos falham devido à omissão de informações importantes e/ou devido a falha dos envolvidos, na coordenação de seus esforços para criar um produto final bem-sucedido. Esses são itens de gerenciamento e devem ser tratados.

"A verdade da questão é que sempre sabemos a coisa certa a ser feita. A parte difícil é fazê-la."

General H. Norman Schwarzkopf

Princípio 6. Gerencie mudanças. A abordagem pode ser tanto formal quanto informal, entretanto devem ser estabelecidos mecanismos para gerenciar a maneira como as mudanças serão solicitadas, avaliadas, aprovadas e implementadas.

Princípio 7. Avalie os riscos. Uma série de coisas pode dar errada quando um software é desenvolvido. É essencial estabelecer planos de contingência. Alguns desses planos de contingência formarão a base das tarefas de engenharia de segurança (Capítulo 27).

Princípio 8. Gere artefatos que forneçam valor para outros. Crie apenas artefatos que proporcionarão valor para outro processo, atividades, ações ou tarefas. Todo artefato produzido como parte da prática da engenharia de software será repassado para alguém. Uma lista de funções e características requisitadas será repassada para a pessoa (ou pessoas) que vai desenvolver um projeto, o projeto será repassado para aqueles que gerarão o código e assim por diante. Certifique-se de que o artefato contenha a informação necessária, sem ambiguidades ou omissões.

A Parte IV deste livro enfoca o projeto, os fatores de gerenciamento do processo e aborda os aspectos variados de cada um desses princípios com certos detalhes.

7.2.2 Princípios que orientam a prática

A prática da engenharia de software tem um objetivo primordial único: entregar dentro do prazo, com alta qualidade, o software operacional contendo funções e características que satisfaçam as necessidades de todos os envolvidos. Para atingir esse objetivo, deve-se adotar um conjunto de princípios fundamentais que orientem o trabalho técnico. Tais princípios são importantes, independentemente do método de análise ou projeto aplicado, das técnicas de desenvolvimento (por exemplo, linguagem de programação, ferramentas para automação) escolhidas ou da abordagem de verificação e validação utilizada. O cumprimento de princípios fundamentais a seguir é essencial para a prática de engenharia de software:

² As características das equipes de software eficientes foram discutidas no Capítulo 6.

Princípio 1. *Divida e conquiste.* De forma mais técnica, a análise e o projeto sempre devem enfatizar a *separação por interesses** (SoCs, *separation of concerns*). Um problema será mais fácil de resolver se for subdividido em conjuntos de interesses. Na forma ideal, cada interesse fornece uma funcionalidade distinta a ser desenvolvida e, em alguns casos, validada, independentemente de outros negócios.

Princípio 2. *Compreenda o uso da abstração.* Em essência, abstrair é simplificar algum elemento complexo de um sistema comunicando o significado em uma única frase. Quando se usa a abstração *planilha*, presume-se que se compreenda o que vem a ser uma planilha, a estrutura geral de conteúdo que uma planilha apresenta e as funções típicas que podem ser aplicadas a ela. Na prática de engenharia de software, usam-se muitos níveis diferentes de abstração, cada um incorporando ou implicando um significado que deve ser comunicado. No trabalho de análise de projeto, uma equipe de software normalmente inicia com modelos que representam altos níveis de abstração (por exemplo, planilha) e, aos poucos, refina tais modelos em níveis de abstração mais baixos (por exemplo, uma *coluna* ou uma função de *soma*).

Joel Spolsky [Spo02] sugere que “todas as abstrações não triviais são, até certo ponto, frágeis”. O objetivo de uma abstração é eliminar a necessidade de comunicar detalhes. Algumas vezes, efeitos problemáticos surgem devido ao “vazamento” desses detalhes. Sem o entendimento dos detalhes, a causa de um problema não poderá ser facilmente diagnosticada.

Princípio 3. *Esforce-se pela consistência.* Seja criando um modelo de análise, desenvolvendo um projeto de software, gerando código-fonte ou criando casos de teste, o princípio da consistência sugere que um contexto conhecido facilita o uso do software. Consideremos, por exemplo, o projeto de uma interface para o usuário de uma WebApp. A colocação padronizada do menu de opções, o uso padronizado de um esquema de cores e de ícones identificáveis colaboram para uma interface ergonomicamente boa.

Princípio 4. *Concentre-se na transferência de informações.* Software trata da transferência de informações: do banco de dados para um usuário, de um sistema judiciário para uma aplicação na Web (WebApp), do usuário para uma interface gráfica (GUI), de um sistema operacional de um componente de software para outro; a lista é quase infinita. Em todos os casos, a informação flui por meio de uma interface, e, como consequência, há a possibilidade de erros, omissões e ambiguidade. A implicação desse princípio é que se deve prestar especial atenção à análise, ao projeto, construção e testes das interfaces.

Princípio 5. *Construa software que apresente modularidade efetiva.* A separação por interesse (Princípio 1) estabelece uma filosofia para software. A *modularidade* fornece um mecanismo para colocar a filosofia em prá-

* N. de R.T.: A palavra *concern* foi traduzida como interesse ou afinidade. Dividir é uma técnica utilizada para lidar com complexidade. Uma das estratégias para dividir é separar usando critérios como interesses ou afinidade, tanto no domínio do problema quanto no domínio da tecnologia do software.

tica. Qualquer sistema complexo pode ser dividido em módulos (componentes), porém a boa prática de engenharia de software demanda mais do que isso. A modularidade deve ser *efetiva*. Isto é, cada módulo deve se concentrar exclusivamente em um aspecto bem restrito do sistema – deve ser coeso em sua função e/ou apresentar conteúdo bem preciso. Além disso, os módulos devem ser interconectados de uma maneira relativamente simples – cada módulo deve apresentar baixo acoplamento com outros módulos, fontes de dados e outros aspectos ambientais.

Princípio 6. Verifique os padrões. Brad Appleton [App00] propõe que:

O objetivo dos padrões é criar uma fonte literária para ajudar os desenvolvedores de software a solucionar problemas recorrentes, encontrados ao longo de todo o desenvolvimento de software. Os padrões ajudam a criar uma linguagem que pode ser compartilhada de tal forma que possa transmitir conteúdos e experiências acerca dos problemas e de suas soluções. Codificar formalmente tais soluções e as suas relações permite que se armazene com sucesso a base do conhecimento, a qual define nossa compreensão sobre boas arquiteturas, correspondendo às necessidades de seus usuários.

Padrões de projeto podem ser utilizados em problemas mais amplos de engenharia e de integração de sistemas, permitindo que os componentes de sistemas complexos evoluam independentemente.

Princípio 7. Quando possível, represente o problema e sua solução sob perspectivas diferentes. Ao analisar um problema e sua solução sob uma série de perspectivas diferentes, é mais provável que se obtenha uma melhor visão e, assim, os erros e omissões sejam revelados. Por exemplo, um modelo de requisitos pode ser representado usando um ponto de vista orientado a cenários, um ponto de vista orientado a classes ou um ponto de vista comportamental (Capítulos 9 a 11). Cada um deles fornece uma perspectiva diferente do problema e de seus requisitos.

Princípio 8. Lembre-se de que alguém fará a manutenção do software. No longo prazo, à medida que defeitos forem descobertos, o software será corrigido, adaptado de acordo com as alterações de seu ambiente e estendido conforme solicitação de novas funcionalidades por parte dos envolvidos. As atividades de manutenção podem ser facilitadas se for aplicada uma prática de engenharia de software consistente ao longo do processo.

Esses princípios não constituem tudo que é necessário para a construção de um software de alta qualidade, mas estabelecem uma base para os métodos de engenharia de software discutidos neste livro.

Use padrões (Capítulo 16) para armazenar conhecimento e experiência para as futuras gerações de engenheiros de software.

Evite a falta de visão. Examine um problema a partir de diferentes perspectivas. Você descobrirá aspectos que, de outra forma, ficariam ocultos.

7.3 Princípios das atividades metodológicas

Nas próximas seções, serão apresentados princípios que têm forte influência sobre o sucesso de cada atividade metodológica genérica definida como parte do processo de software. Em muitos casos, os princípios discutidos para cada

atividade metodológica são aprimoramentos dos princípios apresentados na Seção 7.2. Eles são apenas princípios fundamentais se situam em um nível de abstração mais baixo.

7.3.1 Princípios da comunicação

Antes que os requisitos dos clientes sejam analisados, modelados ou especificados, eles devem ser coletados por meio da atividade de comunicação. Um cliente apresenta um problema que pode ser resolvido por uma solução baseada em computador. Você responde ao pedido de ajuda. A comunicação acabou de começar. Entretanto, o percurso da comunicação até o entendimento costuma ser acidentado.

A comunicação efetiva (entre parceiros técnicos, com o cliente, com outros parceiros envolvidos e com gerentes de projetos) constitui uma das atividades mais desafiadoras. Nesse contexto, discutem-se princípios aplicados na comunicação com o cliente. Entretanto, muitos desses princípios são aplicados a todas as formas de comunicação.

Antes de se comunicar, assegure-se de compreender o ponto de vista alheio e suas necessidades. Saiba ouvir.

Princípio 1. Ouça. Concentre-se mais em ouvir do que em se preocupar com respostas. Peça esclarecimento, se necessário, e evite interrupções constantes. Nunca se mostre contestador, tanto em palavras quanto em ações (por exemplo, revirar olhos ou balançar a cabeça) enquanto uma pessoa estiver falando.

Princípio 2. Prepare-se antes de se comunicar. Dedique tempo para compreender o problema antes de se reunir com outras pessoas. Se necessário, faça algumas pesquisas para entender o jargão da área de negócios em questão. Caso seja sua a responsabilidade conduzir uma reunião, prepare uma agenda com antecedência.

Princípio 3. Alguém deve facilitar a atividade. Toda reunião de comunicação deve ter um líder (um facilitador) para manter a conversa direcionada e produtiva, mediar qualquer conflito que ocorra e garantir que outros princípios sejam seguidos.

Princípio 4. Comunicar-se pessoalmente é melhor. No entanto, costuma ser mais produtivo quando alguma outra representação da informação relevante está presente. Por exemplo, um participante pode fazer um desenho ou um esboço de documento que servirá como foco para a discussão.

Princípio 5. Anote e documente as decisões. As coisas tendem a cair no esquecimento. Algum participante desta reunião deve servir como “gravador” e anotar todos os pontos e decisões importantes.

Princípio 6. Esforce-se para conseguir colaboração. Colaboração e consenso ocorrem quando o conhecimento coletivo dos membros da equipe é usado para descrever funções e características do produto ou sistema. Cada pequena colaboração servirá para estabelecer confiança entre os membros e chegar a um objetivo comum.

Princípio 7. Mantenha o foco; crie módulos para a discussão. Quanto mais pessoas envolvidas, maior a probabilidade de a discussão saltar de um tó-

“Perguntas e respostas simples são o caminho mais curto para a maioria das perplexidades.”

Mark Twain

pico a outro. O facilitador deve manter a conversa modular, abandonando um assunto somente depois de ele ter sido resolvido (veja, entretanto, o Princípio 9).

Princípio 8. *Faltando clareza, desenhe.* A comunicação verbal flui até certo ponto. Um esboço ou um desenho pode permitir maior clareza quando palavras são insuficientes.

Princípio 9. (a) *Uma vez de acordo, siga em frente.* (b) *Se não chegar a um acordo, siga em frente.* (c) *Se uma característica ou função não estiver clara e não puder ser elucidada no momento, siga em frente.* A comunicação, assim como qualquer outra atividade da engenharia de software, toma tempo. Em vez de ficar interagindo indefinidamente, os participantes precisam reconhecer que muitos assuntos exigem discussão (veja o Princípio 2) e que “seguir em frente” é, algumas vezes, a melhor maneira de ser ágil na comunicação.

Princípio 10. *Negociação não é uma competição nem um jogo. Funciona melhor quando as duas partes saem ganhando.* Há muitas ocasiões em que é necessário negociar funções e características, prioridades e prazos de entrega. Se a equipe interagiu adequadamente, todas as partes envolvidas têm um objetivo comum. Mesmo assim, a negociação exigirá compromisso de todos.

O que acontecerá caso não se consiga chegar a um acordo com um cliente sobre alguma questão relativa ao projeto?

INFORMAÇÕES



A diferença entre clientes e usuários

Os engenheiros de software se comunicam com muitos e diferentes envolvidos, mas clientes e usuários têm o maior impacto sobre o trabalho técnico que virá a seguir. Em alguns casos, cliente e usuário são a mesma pessoa, mas, para muitos projetos, são pessoas que trabalham para gerentes diferentes em organizações diferentes.

Cliente é a pessoa ou grupo que: (1) originalmente requisita o software a ser construído, (2) define os objetivos gerais do negócio para o software, (3) fornece os requisitos

básicos do produto e (4) coordena os recursos financeiros para o projeto. Em uma negociação de sistemas ou de produtos, o cliente, com frequência, é o departamento de marketing. Em um ambiente de tecnologia da informação (TI), o cliente pode ser um departamento ou componente da empresa.

Usuário é uma pessoa ou grupo que (1) vai realmente usar o software construído para atingir algum propósito de negócio e (2) vai definir os detalhes operacionais do software de modo que o objetivo seja alcançado.

CASASEGURA



Erros de comunicação

Cena: Local de trabalho da equipe de engenharia de software

Atores: Jamie Lazar, membro da equipe de software; Vinod Raman, membro da equipe de software; Ed Robbins, membro da equipe de software.

Conversa:

Ed: O que você ouviu falar sobre o projeto *CasaSegura*?

Vinod: A reunião inicial está marcada para a próxima semana.

Jamie: Já andei investigando, mas não deu muito certo.

Ed: O que você quer dizer?

Jamie: Bem, liguei para Lisa Perez. Ela é a “mandachuva” do marketing dessa história.

Vinod: E...?

Jamie: Eu queria que ela me desse informações sobre as características e funções do *CasaSegura*... esse tipo de coisa. Mas, em vez disso, ela começou a me fazer perguntas sobre sistemas de segurança, sistemas de vigilância... Não sou especialista na área.

Vinod: O que isso lhe diz?

(Jamie encolhe os ombros.)

Vinod: Isso quer dizer que o marketing precisará de nós como consultores e que é melhor nos prepararmos nessa

área de produto antes da primeira reunião. Doug disse que quer que ‘colaboremos’ com nosso cliente; portanto, é melhor aprendermos como fazer isso.

Ed: Provavelmente teria sido melhor ter dado uma passada em seu escritório. Simples telefonemas não funcionam bem para esse tipo de coisa.

Jamie: Vocês dois estão certos. Temos que agir em conjunto, ou nossos primeiros contatos serão difíceis.

Vinod: Vi o Doug lendo um livro sobre “engenharia de requisitos”. Aposto que ele enumera alguns princípios de comunicação eficaz. Vou pedi-lo emprestado amanhã mesmo.

Jamie: Boa ideia... depois você poderá nos ensinar.

Vinod (sorrindo): É isso aí.

7.3.2 Princípios do planejamento

“Na preparação para a batalha sempre achei que os planos fossem inúteis, mas planejamento é indispensável.”

General Dwight D. Eisenhower

A atividade de comunicação ajuda a definir as metas e os objetivos gerais (sujeitos a mudanças à medida que o tempo passa, é claro). Entretanto, compreender essas metas e objetivos não é o mesmo que definir um plano para alcançá-los. A atividade de planejamento engloba um conjunto de técnicas e práticas gerenciais que permitem à equipe de software definir um roteiro à medida que segue na direção de seus objetivos estratégicos e táticos.

Por mais que se tente, é impossível prever com exatidão como um projeto de software vai evoluir. Não há uma maneira simples de determinar quais problemas técnicos não previstos serão encontrados, quais erros ocorrerão ou quais itens de negócio sofrerão mudanças. Ainda assim, uma boa equipe deve planejar sua abordagem.

Existem muitas filosofias diferentes de planejamento.³ Algumas pessoas são “minimalistas”, afirmando que alterações evidenciam, frequentemente, a necessidade de um plano detalhado. Outras são “tradicionalistas”, afirmando que o plano fornece um roteiro eficaz e que, quanto mais detalhes são apresentados, menos probabilidade a equipe terá de se perder. Outras ainda são “agilistas”, afirmando que um rápido “planejamento do jogo” pode ser necessário, mas que o roteiro surgirá como “verdadeiro trabalho” no início do software.

O que fazer? Em muitos projetos, planejamento em excesso representa consumo de tempo sem resultado produtivo (mudanças demais); entretanto, pouco planejamento é uma receita para o caos. Como a maioria das coisas na vida, o planejamento deveria ser conduzido de forma moderada, o suficiente para servir de guia para a equipe – nem mais, nem menos. Indepen-

Um excelente banco de informações sobre planejamento e gerenciamento de projetos pode ser encontrado em www.4pm.com/repository.htm.

³ Na Parte IV do livro é apresentada uma discussão detalhada sobre planejamento e gerenciamento de projetos de software.

dentemente do rigor com o qual o planejamento é feito, os seguintes princípios sempre se aplicam:

Princípio 1. *Entenda o escopo do projeto.* É impossível usar um mapa se você não sabe aonde está indo. O escopo indica um destino para a equipe de software.

Princípio 2. *Inclua os envolvidos na atividade de planejamento.* Os envolvidos definem prioridades e estabelecem as restrições de projeto. Para adequar essas características, os engenheiros muitas vezes devem negociar a programação de entrega, cronograma e outras questões relativas ao projeto.

Princípio 3. *Reconheça que o planejamento é iterativo.* Um plano de projeto jamais é gravado em pedra. Depois que o trabalho se inicia, muito provavelmente ocorrerão alterações. Consequentemente, o plano deverá ser ajustado para incluir as alterações. Além do mais, os modelos de processos incremental e iterativo exigem replanejamento após a entrega de cada incremento de software, de acordo com os feedbacks recebidos dos usuários.

Princípio 4. *Faça estimativas baseadas no que conhece.* O objetivo da estimativa é dar indicações de esforço, custo e prazo para a realização, com base na compreensão atual do trabalho a ser realizado. Se a informação for vaga ou não confiável, as estimativas serão igualmente não confiáveis.

Princípio 5. *Considere os riscos ao definir o plano.* Caso tenha identificado riscos de alto impacto e alta probabilidade, um planejamento de contingência será necessário. Além disso, o plano de projeto (inclusive o cronograma) deve ser ajustado para incluir a possibilidade de um ou mais desses riscos ocorrerem. Leve em conta a provável exposição decorrente de perdas ou comprometimentos de bens do projeto.

Princípio 6. *Seja realista.* As pessoas não trabalham 100% de todos os dias. Sempre há interferência de ruído em qualquer comunicação humana. Omissões e ambiguidades são fatos da vida. Mudanças ocorrem. Até mesmo os melhores engenheiros de software cometem erros. Essas e outras realidades devem ser consideradas ao se estabelecer um plano de projeto.

Princípio 7. *Ajuste particularidades ao definir o plano.* *Particularidades* referem-se ao nível de detalhamento introduzido conforme o plano de projeto é desenvolvido. Um plano com alto grau de particularidade fornece considerável detalhamento de tarefas planejadas para incrementos em intervalos relativamente curtos para que o rastreamento e controle ocorram com frequência. Um plano com baixo grau de particularidade resulta em tarefas mais amplas para intervalos maiores. Em geral, particularidades variam de altas para baixas, conforme o cronograma de projeto se distancia da data atual. Nas semanas ou meses seguintes, o projeto pode ser planejado com detalhes significativos. As atividades que não serão realizadas por muitos meses não exigem alto grau de particularidade (muito pode ser alterado).

"O sucesso deve-se mais ao bom senso do que à genialidade."

An Wang

O termo *particularidade* refere-se a detalhes por meio dos quais alguns elementos do planejamento são representados ou conduzidos.

Princípio 8. Defina como pretende garantir a qualidade. O plano deve determinar como a equipe pretende garantir a qualidade. Se forem necessárias revisões técnicas⁴, deve-se agendá-las. Se a programação em pares for utilizada (Capítulo 5), isso deve estar definido explicitamente dentro do plano.

Princípio 9. Descreva como acomodar as alterações. Mesmo o melhor planejamento pode ser prejudicado por alterações sem controle. Deve-se identificar como as alterações serão integradas ao longo do trabalho de engenharia. Por exemplo, o cliente pode solicitar uma alteração a qualquer momento? Se for solicitada uma mudança, a equipe é obrigada a implementá-la imediatamente? Como é avaliado o impacto e o custo de uma alteração?

Princípio 10. Verifique o plano com frequência e faça os ajustes necessários. Os projetos de software atrasam uma vez ou outra. Portanto, é bom verificar diariamente seu progresso, procurando áreas ou situações problemáticas, nas quais o que foi programado não está em conformidade com o trabalho realizado. Ao surgir um descompasso, deve-se ajustar o plano adequadamente.

Para máxima eficiência, todos da equipe de software devem participar da atividade de planejamento. Somente assim os membros estarão engajados com o plano.

7.3.3 Princípios da modelagem

Criam os modelos para entender melhor o que será construído. Quando a entidade for algo físico (por exemplo, um edifício, um avião, uma máquina), podemos construir um modelo que seja idêntico na forma e no formato, porém em menor escala. Entretanto, quando a entidade a ser construída for software, nosso modelo deve assumir uma forma diferente. Ele deve ser capaz de representar as informações que o software transforma, a arquitetura e as funções que permitem a transformação, as características que os usuários desejam e o comportamento do sistema à medida que a transformação ocorra. Os modelos devem cumprir esses objetivos em diferentes níveis de abstração – primeiro, descrevendo o software do ponto de vista do cliente e, depois, em um nível mais técnico.

No trabalho de engenharia de software, podem ser criadas duas classes de modelos: de requisitos e de projeto. Os *modelos de requisitos* (também denominados *modelos de análise*) representam os requisitos dos clientes, descrevendo o software em três domínios diferentes: o domínio da informação, o domínio funcional e o domínio comportamental. Os *modelos de projeto* representam características do software que ajudam os desenvolvedores a construí-lo com eficiência: a arquitetura, a interface do usuário e os detalhes dos componentes.

Os modelos de análise representam os requisitos dos clientes. Os modelos de projeto oferecem uma especificação concreta para a construção do software.

⁴ As revisões técnicas são discutidas no Capítulo 20.

Em seu livro sobre modelagem ágil, Scott Ambler e Ron Jeffries [Amb02b] estabelecem um conjunto de princípios de modelagem⁵ destinados àqueles que usam o modelo de processos ágeis (Capítulo 5), mas que também podem ser usados por todos os engenheiros de software que executam ações e tarefas de modelagem:

Princípio 1. *O objetivo principal da equipe de software é construir software, não criar modelos.* Agilidade significa entregar software ao cliente no menor prazo possível. Os modelos que fazem isso acontecer são criações valiosas; entretanto, os que retardam o processo ou oferecem pouca novidade devem ser evitados.

Princípio 2. *Seja objetivo – não crie mais modelos do que precisa.* Todo modelo criado deve ser atualizado quando ocorrem alterações. E, mais importante, todo modelo novo demanda tempo que poderia ser despendido em construção (codificação e testes). Portanto, crie somente modelos que facilitem mais e diminuam o tempo para a construção do software.

Princípio 3. *Esforce-se ao máximo para produzir o modelo mais simples possível.* Não exagere no software [Amb02b]. Mantendo-se modelos simples, o software resultante também será simples. O resultado será um software mais fácil de ser integrado, testado e mantido. Além disso, modelos simples são mais fáceis de compreender e criticar, resultando em uma forma contínua de feedback (realimentação) que otimiza o resultado final.

O objetivo de qualquer modelo é transmitir informações. Para tanto, use um formato consistente. Considere o fato de não estar lá para explicá-lo. Ele deve ser autoexplicativo.

Princípio 4. *Construa modelos que facilitem alterações.* Considere que os modelos mudarão, mas, ao considerar tal fato, não seja relapso. Por exemplo, uma vez que os requisitos serão alterados, há uma tendência de dar pouca atenção a seus modelos. Por quê? Porque se sabe que mudarão de qualquer forma. O problema dessa atitude é que, sem um modelo de requisitos razoavelmente completo, criar-se-á um projeto (modelo de projeto) que invariavelmente vai deixar de lado funções e características importantes.

Princípio 5. *Estabeleça um propósito claro para cada modelo.* Toda vez que criar um modelo, pergunte se há motivo para tanto. Se você não for capaz de dar justificativas sólidas para a existência do modelo, não desperdice tempo com ele.

Princípio 6. *Adapte os modelos que desenvolveu ao sistema à disposição.* Talvez seja necessário adaptar a notação ou as regras do modelo ao aplicativo; por exemplo, um aplicativo de videogame pode exigir uma técnica de modelagem diferente daquela utilizada em um software embarcado e de tempo real que controla o motor de um automóvel.

⁵ Os princípios citados nesta seção foram resumidos e reescritos de acordo com os propósitos do livro.

Princípio 7. *Crie modelos úteis, mas esqueça a construção de modelos perfeitos.* Ao construir modelos de requisitos e de projetos, um engenheiro de software atinge um ponto de retornos decrescentes. Isto é, o esforço necessário para fazer o modelo absolutamente completo e internamente consistente não vale os benefícios resultantes. Estaríamos sugerindo que a modelagem deve ser descuidada ou de baixa qualidade? A resposta é: não. Mas a modelagem deve ser conduzida tendo-se em vista as próximas etapas de engenharia de software. Iterar indefinidamente para tornar um modelo “perfeito” não supre a necessidade de agilidade.

Princípio 8. *Não se torne dogmático quanto à sintaxe do modelo. Se ela consegue transmitir o conteúdo, a representação é secundária.* Embora todos os integrantes de uma equipe devam tentar usar uma notação consistente durante a modelagem, a característica mais importante do modelo reside em transmitir informações que possibilitem a próxima tarefa de engenharia. Se um modelo viabilizar isso com êxito, a sintaxe incorreta pode ser perdoada.

Princípio 9. *Se os instintos dizem que um modelo não está correto, mesmo parecendo correto no papel, provavelmente há motivos para se preocupar.* Se você for um engenheiro experiente, confie em seus instintos. O trabalho com software nos ensina muitas lições – muitas das quais em um nível subconsciente. Se algo lhe diz que um modelo de projeto parece falho, embora não haja provas explícitas, há motivos para dedicar tempo extra, examinando o modelo ou desenvolvendo outro diferente.

Princípio 10. *Obtenha feedback o quanto antes.* Todo modelo deve ser revisado pelos membros da equipe de software. O objetivo das revisões é proporcionar feedback que seja usado para corrigir erros de modelagem, alterar interpretações errôneas e adicionar características ou funções omitidas inadvertidamente.

Princípios da modelagem de requisitos. Nas últimas três décadas, inúmeros métodos de modelagem de requisitos foram desenvolvidos. Pesquisadores identificaram problemas de análise de requisitos e suas causas e desenvolveram uma série de notações de modelagem e de “heurísticas” correspondentes para resolvê-los. Cada um dos métodos de análise tem um ponto de vista particular, mas todos estão inter-relacionados por princípios operacionais:

“O primeiro problema do engenheiro em qualquer situação de projeto é descobrir qual é realmente o problema.”

Autor desconhecido

Princípio 1. *O universo de informações de um problema deve ser representado e compreendido.* O universo de informações engloba os dados constantes no sistema (do usuário, de outros sistemas ou dispositivos externos), os dados que fluem para fora do sistema (via interface do usuário, interfaces de rede, relatórios, gráficos e outros meios) e a armazenagem de dados que coleta e organiza objetos de dados persistentes (isto é, dados que são mantidos permanentemente).

Princípio 2. *As funções executadas pelo software devem ser definidas.* As funções do software oferecem benefício direto aos usuários e também su-

porte interno para fatores visíveis aos usuários. Algumas funções transformam dados que fluem no sistema. Em outros casos, as funções exercem certo nível de controle sobre o processamento interno do software ou sobre elementos de sistema externo. As funções podem ser descritas em diferentes níveis de abstração, desde afirmação geral até uma descrição detalhada dos elementos de processo que devem ser requisitados.

Princípio 3. O comportamento do software (consequência de eventos externos) deve ser representado. O comportamento de um software é comandado por sua interação com o ambiente externo. Dados fornecidos pelos usuários, informações referentes a controles provenientes de um sistema externo ou dados de monitoramento coletados de uma rede fazem o software se comportar de maneira específica.

Princípio 4. Os modelos que representam informação, função e comportamento devem ser divididos de modo a revelar detalhes em camadas (ou de maneira hierárquica). A modelagem de requisitos é a primeira etapa da solução de um problema de engenharia de software. Permite que se entenda melhor o problema e se estabeleçam bases para a solução (projeto). Os problemas complexos são difíceis de resolver em sua totalidade. Por essa razão, deve-se usar a estratégia dividir-e-conquistar. Um problema grande e complexo é dividido em subproblemas até que cada um seja relativamente fácil de ser compreendido. Esse conceito é denominado *fracionamento* ou *separação por interesse* e é uma estratégia-chave na modelagem de requisitos.

Princípio 5. A análise deve partir da informação essencial para os detalhes da implementação. A modelagem de análise se inicia pela descrição do problema sob o ponto de vista do usuário. A “essência” do problema é descrita sem levar em consideração como será implementada uma solução. Por exemplo, um jogo de videogame exige que o jogador “instrua” seu protagonista sobre qual direção seguir para continuar, conforme se envolve em situações perigosas. Essa é a essência do problema. O detalhamento da implementação (em geral descrito como parte do modelo de projeto) indica como a essência (do software) será implementada. No caso do videogame, talvez fosse usada entrada de voz. Por outro lado, poderia ser digitado um comando no teclado, um joystick (ou mouse) poderia ser apontado em uma direção específica, um dispositivo sensível ao movimento poderia ser agitado no ar ou poderia ser usado um dispositivo que lê diretamente os movimentos do corpo do jogador.

Ao aplicar esses princípios, o engenheiro de software aborda um problema de forma sistemática. Mas como os princípios são aplicados na prática? Essa pergunta será respondida nos Capítulos 8 a 11.

Princípios da modelagem de projetos. A modelagem de projetos de software equivale às plantas de uma casa feitas por um arquiteto. Ela começa pela representação do todo a ser construído (por exemplo, uma representação tridimensional da casa) e, gradualmente, concentra-se nos detalhes, oferecendo um roteiro para a sua construção (por exemplo, a estrutura do encanamento).

A modelagem de análise se concentra em três atributos de software: informações a serem processadas, função a ser entregue e comportamento a ser apresentado.

"Primeiramente, verifique se o projeto é inteligente e preciso. Feito isso, persista. Não desista do seu propósito por causa de uma rejeição."

William Shakespeare

Comentários mais específicos sobre o processo dos projetos, juntamente com um debate sobre sua estética, podem ser encontrados em <http://www.gobookee.net/search.php?q=aabyan+design+aesthetics>.

De modo similar, a modelagem de projetos fornece uma variedade de diferentes enfoques do sistema.

Existem muitos métodos para identificar os vários elementos de um projeto. Alguns são voltados a dados, permitindo que a estrutura de dados determine a arquitetura do programa e dos componentes de processamento resultantes. Outros são voltados para padrões, usando informações a respeito do domínio do problema (da modelagem dos requisitos) para desenvolver os estilos arquiteturais e os padrões de processamento. Outros, ainda, são voltados a objetos, usando os objetos do domínio do problema como determinantes para a criação dos métodos e das estruturas de dados que os manipularão. Ainda assim, todos englobam uma série de princípios de projeto que podem ser aplicados independentemente do método empregado:

Princípio 1. O projeto deve ser alinhado para o modelo de requisitos. O modelo de requisitos descreve a área de informação do problema, funções visíveis ao usuário, desempenho do sistema e um conjunto de classes de requisitos que empacota objetos de negócios com os métodos a que servem. O modelo de projeto traduz essa informação em uma arquitetura, um conjunto de subsistemas que implementam funções mais amplas e um conjunto de componentes que são a concretização das classes de requisitos. Os elementos da modelagem de projetos devem ser alinhados para a modelagem de requisitos.

Princípio 2. Sempre considere a arquitetura do sistema a ser construído. A arquitetura de software (Capítulo 13) é a espinha dorsal do sistema a ser construído. Afeta interfaces, estruturas de dados, desempenho e fluxo de controle de programas, a maneira pela qual os testes podem ser conduzidos, a manutenção do sistema realizada e muito mais. Por todas essas razões, o projeto deve começar com as considerações arquiteturais. Só depois de a arquitetura ter sido estabelecida devem ser considerados os elementos relativos aos componentes.

Princípio 3. O projeto de dados é tão importante quanto o projeto das funções de processamento. O projeto de dados é um elemento essencial do projeto da arquitetura. A forma como os objetos de dados são percebidos no projeto não pode ser deixada ao acaso. Um projeto de dados bem estruturado ajuda a simplificar o fluxo do programa e torna mais fácil a elaboração do projeto e a implementação dos componentes de software, tornando mais eficiente o processamento como um todo.

Princípio 4. As interfaces (tanto internas quanto externas) devem ser projetadas com cuidado. A forma como os dados fluem entre os componentes de um sistema tem muito a ver com a eficiência do processamento, com a propagação de erros e com a simplicidade do projeto. Uma interface bem elaborada facilita a integração e auxilia o responsável pelos testes quanto à validação das funções dos componentes.

Princípio 5. O projeto da interface do usuário deve ser voltado às necessidades do usuário, mas ele sempre deve enfatizar a facilidade de uso. A interface do usuário é a manifestação visível do software. Não importa quão sofisticadas sejam as funções internas, quão amplas sejam as estruturas

de dados, quão bem projetada seja a arquitetura; um projeto de interface deficiente leva à percepção de que um software é “ruim”.

Princípio 6. O projeto no nível de componentes deve ser funcionalmente independente. Independência funcional é uma medida para a “mentalidade simplificada” de um componente de software. A funcionalidade entregue por um componente deve ser coesa – isto é, concentrar-se em uma, e somente uma, função ou subfunção.⁶

Princípio 7. Os componentes devem ser relacionados livremente tanto entre componentes quanto com o ambiente externo. O relacionamento é obtido de várias maneiras – via interface de componentes, por meio de mensagens, por meio de dados em geral. À medida que o nível de correlação aumenta, a tendência para a propagação do erro também aumenta, e a manutenção geral do software decresce. Portanto, a dependência entre componentes deve ser mantida o mais baixo possível.

Princípio 8. Representações de projetos (modelos) devem ser de fácil compreensão. A finalidade dos projetos é transmitir informações aos desenvolvedores que farão a codificação, àqueles que irão testar o software e a outros que possam vir a dar manutenção futuramente. Se o projeto for de difícil compreensão, não servirá como meio de comunicação efetivo.

Princípio 9. O projeto deve ser desenvolvido iterativamente. A cada iteração, o projetista deve se esforçar para obter maior grau de simplicidade. Como todas as atividades criativas, a elaboração de um projeto ocorre de forma iterativa. As primeiras iterações são realizadas para refinar o projeto e corrigir erros; entretanto, as iterações finais devem dirigir esforços para tornar o projeto tão simples quanto possível.

Princípio 10. A criação de um modelo de projeto não exclui uma abordagem ágil. Alguns proponentes do desenvolvimento de software ágil (Capítulo 5) insistem em que o código é a única documentação de projeto necessária. Contudo, o objetivo de um modelo de projeto é ajudar outros que deverão manter e evoluir o sistema. É extremamente difícil entender o objetivo de nível mais alto de um trecho de código ou suas interações com outros módulos em um moderno ambiente de execução (*run-time*) e multiprocessos (*multithread*).

Embora a documentação de código possa ser útil, muitas vezes é difícil manter o código e suas descrições consistentes. O modelo de projeto é vantajoso porque é criado em um nível de abstração isento de detalhes técnicos desnecessários e é fortemente acoplado aos conceitos e requisitos da aplicação.

Informações de projeto complementares podem incorporar a razão de ser de um projeto, incluindo as descrições de alternativas arquiteturais rejeitadas. Essas informações podem ser necessárias para ajudar a compreender claramente a grande quantidade de código. Além disso, pode ajudar a manter a consistência quando forem necessárias decisões de projeto mais refinadas.

“As diferenças não são insignificantes – são muito semelhantes às diferenças entre Salieri e Mozart. Cada vez mais estudos demonstram que os melhores projetistas produzem estruturas mais rápidas, menores, mais simples, mais claras e com menos esforço.”

Frederick P. Brooks

⁶ Mais informações a respeito de coesão podem ser encontradas no Capítulo 12.

Esse tipo de especificação arquitetural também pode ajudar diferentes envolvidos no sistema a se comunicar com a equipe de projeto e a comunicação entre os próprios componentes da equipe.

Com exceção dos sistemas relativamente pequenos, cujos protótipo e experimentação podem ser feitos rapidamente, fazer um projeto de alto nível usando somente código-fonte seria imprudente. A documentação de projeto ágil acompanha o projeto e o desenvolvimento passo a passo. Para evitar desperdício, o esforço despendido nesses documentos deve ser proporcional à estabilidade do projeto. Nos primeiros estágios do projeto, as descrições devem ser adequadas para se comunicar com os envolvidos. Quanto mais estável o projeto, mais abrangentes as descrições. Uma abordagem poderia ser usar ferramentas de modelagem de projeto, as quais produzem modelos executáveis que podem ser normalmente avaliados de maneira ágil.

Quando esses princípios são bem aplicados, cria-se um projeto com fatores de qualidade tanto externos quanto internos [Mye78]. *Fatores externos de qualidade* são as propriedades que podem ser rapidamente notadas pelos usuários (por exemplo, velocidade, confiabilidade, correção, usabilidade). Os *fatores internos de qualidade* são de extrema importância para os engenheiros de software, pois conduzem a um projeto de alta qualidade do ponto de vista técnico. Para obter fatores de qualidade internos, o projetista deve entender sobre conceitos básicos de projeto (Capítulo 12).

Princípios da modelagem viva. Breu [Bre10] descreve os *modelos vivos* como um paradigma que combina o desenvolvimento baseado em modelos⁷ com o gerenciamento e operação de sistemas orientados a serviços.⁸ Os modelos vivos dão suporte à cooperação entre todos os envolvidos no projeto, fornecendo abstrações baseadas em modelo apropriadas, que descrevem as interdependências entre os elementos do sistema. Oito princípios são fundamentais para o estabelecimento de um ambiente de modelos vivos:

Princípio 1. Os modelos centrados nos envolvidos devem ter como alvos os envolvidos específicos e suas tarefas. Isso significa que os envolvidos têm permissão para operar nos modelos em um nível de abstração adequado e que os níveis mais baixos ficam ocultos para eles. Por exemplo, o diretor de informática se preocupa com os processos comerciais, enquanto o responsável pelos testes precisa formular casos de teste no nível dos requisitos.

Princípio 2. Modelos e código devem ser fortemente acoplados. Se a meta principal é um sistema operável, qualquer modelo que não reflita esse sistema é inútil. Isso significa que o código e o modelo precisam estar em estados semelhantes. Ferramentas podem ser usadas para dar suporte ao vínculo entre modelos e código.

Princípio 3. Deve ser estabelecido um fluxo de informações bidirecional entre os modelos e o código. Deve ser permitida a propagação de altera-

⁷ O desenvolvimento baseado em modelos (também denominado *engenharia orientada a modelos*) constrói modelos de domínio que representam aspectos específicos de uma área de aplicação.

⁸ Um *sistema orientado a serviços* empacota funcionalidade de software na forma de serviços que são acessíveis por meio de uma infraestrutura de rede.

ções dentro do modelo, do código e do sistema operável, quando necessárias. Tradicionalmente, as alterações feitas em nível de código são refletidas no sistema em execução. Também é importante ter essas alterações de código refletidas no modelo.

Princípio 4. Deve ser criada uma visão comum do sistema. O metamodelo de um sistema define processos comerciais e objetos de informação na camada de gerenciamento de TI, serviços em execução e nós físicos na camada de operações de sistemas e uma visão dos requisitos na camada de engenharia de software. As associações no metamodelo do sistema descrevem dependências dos processos e objetos comerciais na camada de tecnologia.

Princípio 5. As informações do modelo devem ser persistentes para permitir o monitoramento de alterações no sistema. O modelo do sistema descreve o estado atual do sistema em todos os níveis de abstração. A evolução do sistema pode ser descrita e documentada como uma sequência de visões rápidas do modelo do sistema.

Princípio 6. Deve ser verificada a consistência das informações em todos os níveis do modelo. A verificação das restrições do modelo e a recuperação de informações de estado são dois serviços importantes exigidos para apoiar a tomada de decisão dos envolvidos. Por exemplo, um arquiteto de software talvez precise verificar se cada serviço no nível dos requisitos tem um serviço correspondente no nível da arquitetura.

Princípio 7. Cada elemento do modelo tem atribuídos os direitos e responsabilidades dos envolvidos. Cada envolvido é responsável por um subconjunto identificado dos elementos do modelo. Cada subconjunto do modelo é domínio de um envolvido. Isso significa que cada elemento do modelo tem acesso a informações que descrevem as ações que cada envolvido pode executar no elemento.

Princípio 8. Devem ser representados os estados de vários elementos do modelo. Assim como o estado da computação é definido pelos valores armazenados por variáveis-chave durante a execução, o estado de cada elemento do modelo pode ser definido pelos valores atribuídos aos seus atributos.

"Por muito tempo em minha vida fui um observador em relação ao mundo do software, espiando furtivamente os códigos poluídos de outras pessoas. Ocasionalmente, encontro uma joia verdadeira, um programa bem estruturado, escrito em um estilo consistente, livre de gambiarras, desenvolvido de modo que cada componente seja simples, organizado e projetado de forma que o produto possa ser facilmente alterado."

David Parnas

7.3.4 Princípios da construção

A atividade de construção abrange um conjunto de tarefas de codificação e testes que gera um software operacional pronto para ser entregue ao cliente e ao usuário. Na moderna atividade de engenharia de software, a codificação pode ser: (1) a criação direta do código-fonte na linguagem de programação (por exemplo, Java), (2) a geração automática de código-fonte usando uma representação intermediária semelhante a um projeto do componente a ser construído (por exemplo, Enterprise Architect)⁹ ou então (3) a geração automática de código executável usando uma linguagem de programação de quarta geração (por exemplo, Visual C#).

⁹ Enterprise Architect é uma ferramenta criada pela Sparx Systems <http://www.sparxsystems.com/products/ea/index.html>

O enfoque inicial dos testes é voltado para componentes, com frequência denominado *teste de unidade*. Outros níveis de testes incluem: (1) *teste de integração* (realizado à medida que o sistema é construído), (2) *teste de validação*, que avalia se os requisitos foram atendidos pelo sistema completo (ou incremento de software), e (3) *teste de aceitação* conduzido pelo cliente, com o intuito de empregar todos os fatores e funções requisitados. A série de princípios e conceitos fundamentais a seguir é aplicável à codificação e aos testes.

Princípios da codificação. Os princípios que regem a codificação são intimamente alinhados ao estilo, às linguagens e aos métodos de programação. Entretanto, há um número de princípios fundamentais que podem ser estabelecidos:

Não desenvolva um programa elegante que resolva o problema errado. Preste particular atenção ao primeiro princípio da preparação.

Princípios da preparação: Antes de escrever uma linha de código, certifique-se de que

- Compreendeu bem o problema a ser solucionado.
- Compreendeu bem os princípios e conceitos básicos sobre o projeto.
- Escolheu uma linguagem de programação adequada às necessidades do software a ser desenvolvido e ao ambiente em que ele vai operar.
- Selecionou um ambiente de programação que forneça ferramentas para tornar seu trabalho mais fácil.
- Elaborou um conjunto de testes de unidade que serão aplicados assim que o componente codificado estiver completo.

Princípios da codificação: Ao começar a escrever código

- Restrinja seus algoritmos seguindo a prática de programação estruturada [Boh00].
- Pense na possibilidade de usar programação em pares.
- Selecione estruturas de dados que atendam às necessidades do projeto.
- Domine a arquitetura de software e crie interfaces coerentes com ela.
- Mantenha a lógica condicional tão simples quanto possível.
- Crie “loops” agrupados de tal forma que testes sejam facilmente aplicáveis.
- Escolha denominações de variáveis significativas e obedeça a outros padrões de codificação locais.
- Escreva código que se documente automaticamente.
- Crie uma disposição (layout) visual (por exemplo, recuos e linhas em branco) que auxilie a compreensão.

Princípios da validação: Após completar a primeira etapa de codificação, certifique-se de

- Aplicar uma revisão de código quando for apropriado.
- Realizar testes de unidades e corrigir erros ainda não identificados.
- Refabricar o código.

Foram escritos mais livros sobre programação (codificação) e sobre os princípios e conceitos que a guiam do que sobre qualquer outro tópico do processo de software. Livros sobre o tema incluem trabalhos antigos sobre estilo

de programação [Ker78], construção prática de software [McC04], pérolas da programação [Ben99], a arte de programar [Knu98], elementos da programação pragmática [Hun99] e muitos, muitos outros assuntos. Uma discussão ampla sobre esses princípios e conceitos foge dos objetivos deste livro. Caso haja maior interesse, examine uma ou mais das referências indicadas.

Princípios de testes. Em um livro clássico sobre testes de software, Glen Myers [Mye79] estabelece regras que podem servir bem como objetivos da atividade de testes:

- Teste é um processo de executar um programa com o intuito de encontrar um erro.
- Um bom pacote de testes é aquele que tem alta probabilidade de encontrar um erro ainda não descoberto.
- Um teste bem-sucedido é aquele que revela um novo erro.

Para alguns desenvolvedores, esses objetivos representam uma mudança radical de ponto de vista. Eles vão contra a visão comumente difundida de que um teste bem-sucedido é aquele em que nenhum erro é encontrado. Seu objetivo é o de projetar testes que descubram, sistematicamente, diferentes classes de erros, consumindo o mínimo de esforço e tempo.

Se testes forem conduzidos com sucesso (de acordo com os objetivos listados previamente), vão encontrar erros no software. Como benefício secundário, os testes demonstram que as funções do software estão funcionando de acordo com as especificações e que os requisitos relativos ao desempenho e ao comportamento parecem estar sendo atingidos. Os dados coletados durante os testes fornecem um bom indício da confiabilidade do software, assim como um indício da qualidade do software como um todo. Entretanto, os testes não são capazes de mostrar a ausência de erros e defeitos, podendo apenas mostrar que erros e defeitos de software estão presentes. É importante manter essa afirmação (bastante pessimista) em mente enquanto os testes são aplicados.

Davis [Dav95b] sugere um conjunto de princípios de testes¹⁰ que foram adaptados para este livro; Everett e Meyer [Eve09] sugerem princípios adicionais:

Princípio 1. *Todos os testes devem estar alinhados com os requisitos do cliente.*¹¹ O objetivo do teste de software é descobrir erros. Constata-se que os efeitos mais críticos do ponto de vista do cliente são aqueles que conduzem a falhas no programa quanto a seus requisitos.

Princípio 2. *Os testes devem ser planejados muito antes de serem iniciados.* O planejamento dos testes (Capítulo 22) pode começar assim que o modelo de requisitos estiver concluído. A definição detalhada dos casos de teste pode começar assim que o modelo de projeto tenha sido solidifi-

Uma ampla variedade de links para padrões de codificação pode ser encontrada no site <http://www.literateprogramming.com/links.html>.

Quais são os objetivos dos testes de software?

Em um contexto mais amplo de projeto de software, lembre-se de que iniciamos "no geral", focalizando a arquitetura de software, e terminamos "no particular", focalizando os componentes. Para testes, simplesmente invertemos o foco e testamos nosso desenvolvimento.

¹⁰ Apenas um pequeno subconjunto dos princípios de testes de David é citado aqui. Para maiores informações, veja [Dav95b].

¹¹ Esse princípio refere-se a *testes funcionais*; por exemplo, testes que se concentram em requisitos. Os *testes estruturais* (testes que se concentram nos detalhes lógicos ou arquiteturais) não podem se referir a requisitos específicos diretamente.

cado. Portanto, todos os testes podem ser planejados e projetados antes que qualquer codificação tenha sido gerada.

Princípio 3. O princípio de Pareto se aplica a testes de software. Neste contexto, o princípio de Pareto indica que 80% de todos os erros revelados durante testes provavelmente estarão em aproximadamente 20% de todos os componentes do programa. O problema, evidentemente, consiste em isolar os componentes suspeitos e testá-los por completo.

Princípio 4. Os testes devem começar “no particular” e progredir para o teste “no geral”. Os primeiros testes planejados e executados geralmente se concentram nos componentes individuais. À medida que os testes progridem, o enfoque muda para tentar encontrar erros em grupos de componentes integrados e, posteriormente, no sistema inteiro.

Princípio 5. Testes exaustivos são impossíveis. A quantidade de trocas de direção, mesmo para um programa de tamanho moderado, é excepcionalmente grande. Por essa razão, é impossível executar todas as rotas durante os testes. O que é possível, no entanto, é cobrir adequadamente a lógica do programa e garantir que todas as condições referentes ao projeto no nível de componentes sejam exercidas.

Princípio 6. Aplique a cada módulo do sistema um teste equivalente à sua densidade de defeitos esperada. Frequentemente, esses são os módulos mais recentes ou os que são menos compreendidos pelos desenvolvedores.

Princípio 7. Técnicas de testes estáticos podem gerar resultados importantes. Mais de 85% dos defeitos de software são originados na sua documentação (requisitos, especificações, ensaios de código e manuais de usuário) [Jon91]. Testar a documentação do sistema pode ser vantajoso.

Princípio 8. Rastreie defeitos e procure padrões em falhas descobertas pelos testes. O total dos defeitos descobertos é um bom indicador da qualidade do software. Os tipos de defeitos descobertos podem ser uma boa medida da estabilidade do software. Padrões de defeitos encontrados com o passar do tempo podem projetar os números de falhas esperadas.

Princípio 9. Inclua casos de teste que demonstrem que o software está se comportando corretamente. À medida que os componentes do software vão sendo mantidos ou adaptados, interações inesperadas causam efeitos colaterais involuntários em outros componentes. É importante ter um conjunto de casos de teste de regressão (Capítulo 22) pronto para verificar o comportamento do sistema depois que alterações forem aplicadas a um produto de software.

7.3.5 Princípios da disponibilização

Como mencionado, na Parte I deste livro, a disponibilização envolve três ações: entrega, suporte e feedback. Porque os modernos modelos de processos de software serem, em sua natureza, evolutivos ou incrementais, a disponibilização não ocorre imediatamente, mas sim em diversas vezes, à medida que o software avança para sua conclusão. Cada ciclo de entrega propicia ao

cliente e ao usuário um incremento de software operacional que fornece fatores e funcionalidades utilizáveis. Cada ciclo de suporte fornece assistência humana e documentação para todas as funcionalidades e fatores introduzidos durante todos os ciclos de disponibilização até o presente. Cada ciclo de feedback fornece à equipe de software um importante roteiro que resulta em alteração de funcionalidades, elementos e abordagem adotados para o próximo incremento.

A entrega de um incremento de software representa um marco importante para qualquer projeto de software. Um conjunto de princípios essenciais deve ser seguido enquanto a equipe se prepara para a entrega de um incremento:

Princípio 1. *As expectativas do cliente para o software devem ser gerenciadas.* Muitas vezes, o cliente espera mais do que a equipe havia prometido entregar e ocorre a decepção. Isso resulta em feedback não produtivo e arruína o moral da equipe. Em seu livro sobre gerenciamento de expectativas, Naomi Karten [Kar94] afirma: “O ponto de partida para administrar expectativas consiste em se tornar mais consciente sobre como e o que vai comunicar”. Ela sugere que um engenheiro de software deve ser cauteloso em relação ao envio de mensagens conflituosas ao cliente (por exemplo, prometer mais do que pode entregar racionalmente no prazo estabelecido ou entregar mais do que o prometido para determinado incremento e, em seguida, menos do que prometera para o próximo).

Certifique-se de que seu cliente saiba o que esperar antes da entrega de um incremento de software. Caso contrário, com certeza ele contará com mais do que você poderá entregar.

Princípio 2. *Um pacote de entrega completo deve ser montado e testado.* Todo software executável, arquivo de dados de suporte, documento de suporte e outras informações relevantes deve ser completamente montado e testado com usuários reais em uma versão beta. Todos os roteiros de instalação e outros itens operacionais devem ser aplicados inteiramente no maior número possível de configurações computacionais (isto é, hardware, sistemas operacionais, dispositivos periféricos, disposições de rede).

Princípio 3. *É preciso estabelecer uma estrutura de suporte antes da entrega do software.* Um usuário conta com o recebimento de informações precisas e com responsabilidade caso surja um problema. Se o suporte for local, ou, pior ainda, inexistente, imediatamente o cliente ficará insatisfeito. O suporte deve ser planejado, seus materiais devem estar preparados, e mecanismos para manutenção de registros apropriados devem estar determinados para que a equipe de software possa oferecer uma avaliação de qualidade das formas de suporte solicitadas.

Princípio 4. *Material instrucional adequado deve ser fornecido aos usuários.* A equipe de software deve entregar mais do que o software em si. Auxílio em treinamento de forma adequada (se solicitado) deve ser desenvolvido; orientações quanto a problemas inesperados devem ser oferecidas; quando necessário, é importante publicar uma descrição sobre as “diferenças existentes no incremento de software”.¹²

¹² Durante a atividade de comunicação, a equipe de software deve determinar quais recursos de apoio os usuários desejam.

Princípio 5. *Software com bugs deve ser primeiramente corrigido e, depois, entregue.* Sob a pressão do prazo, muitas empresas de software entregam incrementos de baixa qualidade, notificando o cliente de que os bugs “serão corrigidos na próxima versão”. Isso é um erro. Há um ditado no mercado de software: “Os clientes esquecerão a entrega de um produto de alta qualidade em poucos dias, mas jamais esquecerão os problemas causados por um produto de baixa qualidade. O software os lembra disso todos os dias”.

O software entregue traz vantagens ao usuário, mas também fornece feedback útil para a equipe de software. À medida que um incremento é colocado em uso, os usuários devem ser encorajados a tecer comentários sobre recursos e funcionalidades, facilidade de uso, confiabilidade, preocupações com a segurança e quaisquer outras características apropriadas.

7.4 Formas de trabalhar

“O engenheiro ideal é um composto... Não é um cientista, não é um matemático, não é um sociólogo ou escritor; mas pode utilizar o conhecimento e as técnicas de qualquer uma ou de todas essas disciplinas para resolver os problemas de engenharia.”

N. W. Dougherty

Iskold [Isk08] escreve que a qualidade do software tem de se tornar o diferenciador competitivo entre as empresas de software. Conforme aprendemos no Capítulo 6, os aspectos humanos da engenharia de software são tão importantes quanto qualquer outra área da tecnologia. Por isso, é interessante examinar as peculiaridades e os hábitos de trabalho que parecem ser compartilhados entre os engenheiros de software bem-sucedidos. Dentre os mais importantes estão os desejos de refabricar continuamente o projeto e o código, de usar ativamente padrões de projeto comprovados, de adquirir componentes reutilizáveis quando possível, de se concentrar na usabilidade, de desenvolver aplicações que possam ser mantidas, de empregar a linguagem de programação mais conveniente para a aplicação e de construir software utilizando práticas de projeto e teste comprovadas.

Além das peculiaridades e hábitos de trabalho básicos, Iskold [Isk08] sugere 10 conceitos que transcendem linguagens de programação e tecnologias específicas. Alguns deles formam o conhecimento prévio necessário para se apreciar o papel da engenharia de software no processo de software.

1. **Interfaces.** Interfaces simples e conhecidas são menos propensas a erros do que interfaces complexas ou exclusivas.
2. **Convenções e modelos.** Convenções de atribuição de nomes e modelos de software são boas maneiras de se comunicar com um número maior de desenvolvedores e usuários.
3. **Disposição em camadas.** Disposição em camadas é o segredo das abstrações de dados e de programação. Ela permite a separação de conceitos de projeto e detalhes da implementação e, ao mesmo tempo, reduz a complexidade do projeto de software.
4. **Complexidade do algoritmo.** Os engenheiros de software devem ser capazes de apreciar a elegância e as características de desempenho dos algoritmos, mesmo ao escolher um entre rotinas de biblioteca. Escrever código simples e legível frequentemente é uma boa maneira de garantir eficiência de tempo e espaço de uma aplicação.

5. *Uso de hash.* Hashes são importantes para o armazenamento e a recuperação eficiente de dados. Também podem ser importantes como uma maneira de alocar dados igualmente entre computadores em um banco de dados da nuvem.
6. *Uso de cache.* Os engenheiros de software precisam estar à vontade com os prós e contras associados ao fornecimento de acesso rápido a um subconjunto dos dados, armazenando-os na memória do computador e não em dispositivos de armazenamento secundários. Pode ocorrer um *desarte (thrashing)* quando dados mutuamente dependentes não estão na memória ao mesmo tempo. Os aplicativos podem ficar lentos quando novas informações precisam ir para a memória (por exemplo, reproduzir *cutscenes* em um videogame em tempo real).
7. *Concorrência.* A disponibilidade difundida de computadores com vários processadores e ambientes de programação multithread gera desafios para a engenharia de software.
8. *Computação na nuvem.* A computação na nuvem fornece web services e dados poderosos e prontamente acessíveis para plataformas de computação de todos os tipos.
9. *Segurança.* Proteger a confidencialidade e a integridade dos bens do sistema deve ser a preocupação de todo profissional de informática.
10. *Bancos de dados relacionais.* Os bancos de dados relacionais são essenciais no processo de armazenamento e recuperação de informações. É importante saber minimizar a redundância dos dados e maximizar a velocidade de recuperação.

Em muitos casos, alguns bons engenheiros de software trabalhando de forma “inteligente” podem ser mais produtivos do que grupos muito maiores. Um bom engenheiro de software deve saber quais princípios, práticas e ferramentas usar, quando usar e por que são necessárias.

7.5 Resumo

A prática de engenharia de software envolve princípios, conceitos, métodos e ferramentas aplicados por engenheiros da área ao longo de todo o processo de desenvolvimento. Cada projeto de engenharia de software é diferente. Ainda assim, um conjunto de princípios genéricos se aplica ao processo como um todo e à prática de cada atividade metodológica, seja qual for o projeto ou o produto.

Um conjunto de princípios fundamentais auxilia na aplicação de um processo de software significativo e na execução de métodos eficazes de engenharia de software. No nível do processo, os princípios fundamentais estabelecem uma base filosófica que orienta a equipe durante essa fase de desenvolvimento. Quanto ao nível da prática, os princípios estabelecem uma série de valores e regras que servem como guia para analisar um problema, projetar, implementar e testar uma solução e, por fim, disponibilizar o software para a sua comunidade de usuários.

Os princípios de comunicação focam na necessidade de reduzir ruído e aumentar a dimensão conforme o diálogo entre o desenvolvedor e o cliente progride. Ambas as partes devem colaborar para que ocorra a melhor comunicação.

Os princípios de planejamento proporcionam roteiros para a construção do melhor mapa para a jornada rumo a um sistema ou produto completo. O plano pode ser projetado para um único incremento de software ou definido para o projeto inteiro. Independentemente da situação, deve indicar o que será feito, quem o fará e quando o trabalho estará concluído.

A modelagem abrange tanto análise quanto projeto, descrevendo representações do software que se tornam progressivamente mais detalhadas. O objetivo dos modelos é solidificar a compreensão do trabalho a ser feito e providenciar orientação técnica aos implementadores do software. Os princípios de modelagem servem como infraestrutura para os métodos e para a notação utilizada para criar representações do software.

A construção incorpora um ciclo de codificação e testes no qual o código-fonte de um componente é gerado e testado. Os princípios de codificação definem ações genéricas que devem ocorrer antes da codificação ser feita, enquanto está sendo criada e após estar concluída. Embora haja muitos princípios de testes, apenas um é dominante: teste consiste em um processo de execução de um programa com o intuito de encontrar um erro.

A disponibilização ocorre à medida que cada incremento de software é apresentado ao cliente e engloba a entrega, o suporte e o feedback. Os princípios fundamentais para a entrega consideram o gerenciamento das expectativas dos clientes e o fornecimento ao cliente de informações de suporte apropriadas sobre o software. O suporte exige preparação antecipada. O feedback permite ao cliente sugerir mudanças que tenham valor agregado e fornecer ao desenvolvedor informações para o próximo ciclo de engenharia de software.

Problemas e pontos a ponderar

- 7.1 Uma vez que o foco na qualidade demanda recursos e tempo, é possível ser ágil e ainda assim manter o foco na qualidade?
- 7.2 Dos oito princípios básicos que orientam um processo (discutidos na Seção 7.2.1), qual você acredita ser o mais importante?
- 7.3 Descreva o conceito de *separação por interesses* com suas próprias palavras.
- 7.4 Um importante princípio de comunicação afirma “prepare-se antes de se comunicar”. Como essa preparação se manifesta no trabalho prévio que você realiza? Quais artefatos podem resultar da preparação antecipada?
- 7.5 Pesquise sobre “facilitação” para a atividade de comunicação (use as referências fornecidas ou outras) e prepare um conjunto de passos concentrados somente em facilitação.
- 7.6 Em que a comunicação ágil difere da tradicional em engenharia de software? Em que ela é similar?
- 7.7 Por que é necessário “seguir em frente”?
- 7.8 Pesquise sobre “negociação” para a atividade de comunicação e prepare uma série de etapas concentrando-se apenas na negociação.

- 7.9 Descreva o que significa *particularidade* no contexto do cronograma de um projeto.
- 7.10 Por que os modelos são importantes no trabalho de engenharia de software? Eles são sempre necessários? Existem qualificadores para sua resposta sobre necessidade?
- 7.11 Quais são os três “domínios” considerados durante a modelagem de requisitos?
- 7.12 Tente acrescentar mais um princípio àqueles determinados para a codificação na Seção 7.3.4.
- 7.13 Em que consiste um teste bem-sucedido?
- 7.14 Você concorda ou discorda da seguinte afirmação: “Uma vez que vários incrementos são entregues ao cliente, por que se preocupar com a qualidade nos incrementos iniciais? Podemos corrigir os problemas em iterações posteriores”. Justifique sua resposta.
- 7.15 Por que o feedback é importante para uma equipe de software?

Leituras e fontes de informação complementares

A comunicação com o cliente é uma atividade crítica na engenharia de software, embora poucos profissionais invistam tempo em ler sobre isso. Withall (*Software Requirements Patterns*, Microsoft Press, 2007) apresenta diversos padrões úteis que tratam de problemas de comunicação. van Lamsweerde (*Requirement Engineering: From System Goals to UML Models to Software Specifications*, Wiley, 2009) e Sutliff (*User-Centered Requirements Engineering*, Springer, 2002) se concentram expressivamente nos desafios relacionados à comunicação.

Livros de Karten (*Changing How You Manage and Communicate Change*, IT Governance Publishing, 2009), Weigers (*Software Requirements*, 2ª ed., Microsoft Press, 2003), Pardee (*To Satisfy and Delight Your Customer*, Dorset House, 1996) e Karten [Kar94] dão uma excelente visão sobre métodos para interação eficiente com os clientes. Embora a obra não se concentre em software, Hooks e Farry (*Customer-Centered Products*, American Management Association, 2000) apresentam úteis diretrizes genéricas para a comunicação com os clientes. Young (*Project Requirements: A Guide to Best Practices*, Management Concepts, 2006; e *Effective Requirements Practices*, Addison-Wesley, 2001) enfatiza uma “equipe conjunta” entre clientes e desenvolvedores que desenvolvem requisitos de forma colaborativa. Hull, Jackson e Dick (*Requirements Engineering*, Springer, 3ª ed., 2010) e Somerville e Kotonya (*Requirements Engineering: Processes and Techniques*, Wiley, 1998) discutem técnicas e conceitos de “suscitação”, bem como outros princípios de engenharia de requisitos.

Conceitos e princípios de comunicação e planejamento são considerados em vários livros de gerenciamento de projetos. Contribuições úteis para o gerenciamento de projetos incluem livros de Juli (*Leadership Principles for Project Success*, CRC Press, 2012), West e seus colegas (*Project Management for IT Related Projects*, British Informatics Society, 2012), Wysocki (*Effective Project Management: Agile, Adaptive, Extreme*, 5ª ed., Wiley, 2009), Hughes (*Software Project Management*, 5ª ed., McGraw-Hill, 2009), Bechtold (*Essentials of Software Project Management*, 2ª ed., Management Concepts, 2007), Leach (*Lean Project Management: Eight Principles for Success*, BookSurge Publishing, 2006) e Stellman e Greene (*Applied Software Project Management*, O'Reilly Media, 2005).

Davis [Dav95b] compilou um excelente conjunto de princípios de engenharia de software. Além disso, praticamente todo livro sobre engenharia de software contém uma discussão proveitosa sobre conceitos e princípios para análise, projeto e testes. Entre os livros mais amplamente usados (além deste livro, é claro!), temos:

Abran, A. e J. Moore, *SWEBOK: Guide to the Software Engineering Body of Knowledge*, IEEE, 2002.¹³

Pfleeger, S., *Software Engineering: Theory and Practice*, 4ª ed., Prentice Hall, 2009.

Schach, S., *Object-Oriented and Classical Software Engineering*, McGraw-Hill, 8ª ed., 2010.

Sommerville, I., *Software Engineering*, 9ª ed., Addison-Wesley, 2010.

Esses livros também apresentam uma discussão detalhada sobre princípios de modelagem e construção.

Princípios de modelagem são tratados em muitos textos dedicados à análise de requisitos e/ou projeto de software. Livros de Lieberman (*The Art of Software Modeling*, Auerbach, 2007), Rosenberg e Stephens (*Use Case Driven Object Modeling with UML: Theory and Practice*, Apress, 2007), Roques (*UML in Practice*, Wiley, 2004), Penker e Eriksson (*Business Modeling with UML: Business Patterns at Work*, Wiley, 2001) discutem os princípios e métodos de modelagem.

A obra de Norman (*The Design of Everyday Things*, Basic Books, 2002) é uma leitura obrigatória para todo engenheiro de software que pretenda realizar trabalhos de projeto. Winograd e seus colegas (*Bringing Design to Software*, Addison-Wesley, 1996) editaram um excelente conjunto de artigos que tratam das questões práticas no projeto de software. Constantine e Lockwood (*Software for Use*, Addison-Wesley, 1999) apresentam os conceitos associados ao “projeto centrado no usuário”. Tognazzini (*Tog on Software Design*, Addison-Wesley, 1995) apresenta uma interessante discussão filosófica sobre a natureza do projeto. Stahl e seus colegas (*Model-Driven Software Development: Technology, Engineering*, Wiley, 2006) discutem os princípios do desenvolvimento dirigido por modelos. Halladay (*Principle-Based Refactoring*, Principle Publishing, 2012) considera oito princípios de projeto fundamentais e identifica 50 regras para refabricação.

Centenas de livros tratam de um ou mais elementos da atividade de construção. Kernighan e Plauger [Ker78] escreveram um clássico sobre estilo de programação, McConnell [McC04] apresenta diretrizes pragmáticas para a construção prática de software, Bentley [Ben99] sugere uma ampla variedade de pérolas da programação, Knuth [Knu98] escreveu uma série clássica de três volumes sobre a arte de programar e Hunt [Hun99] sugere diretrizes de programação pragmáticas.

Myers e seus colegas (*The Art of Software Testing*, 3ª ed., Wiley, 2011) fizeram uma revisão importante de seu texto clássico e discutem vários princípios de testes importantes. Livros de *How Google Tests Software*, Addison-Wesley, 2012), Perry (*Effective Methods for Software Testing*, 3ª ed., Wiley, 2006) e Whittaker (*How to Break Software*, Addison-Wesley, 2002), Kaner e seus colegas (*Lessons Learned in Software Testing*, Wiley, 2001) e Marick (*The Craft of Software Testing*, Prentice-Hall, 1997) apresentam importantes conceitos e princípios de testes e bastante orientação pragmática.

Uma ampla gama de fontes de informação sobre a prática da engenharia de software se encontra à disposição na Internet. Uma lista atualizada de referências relevantes (em inglês) para a prática da engenharia de software pode ser encontrada no site: **www.mhhe.com/pressman**.

¹³ Disponível gratuitamente em <http://www.computer.org/portal/web/swebok/v3guide>.

Entendendo os requisitos

8

Entender os requisitos de um problema está entre as tarefas mais difíceis enfrentadas por um engenheiro de software. Quando você pensa nisso pela primeira vez, entender claramente a engenharia de requisitos não parece assim tão difícil. Afinal, o cliente não sabe o que é necessário? Os usuários não deveriam ter um bom entendimento das características e funções que serão vantajosas? Surpreendentemente, em muitos casos a resposta a essas perguntas é “não”. E mesmo se os clientes e usuários fossem explícitos quanto às suas necessidades, elas mudariam ao longo do projeto.

No prefácio de um livro de Ralph Young [You01] sobre práticas de requisitos eficazes, um de nós [RSP] escreveu:

É o seu pior pesadelo. Um cliente entra em seu escritório, senta-se, olha diretamente nos seus olhos e diz: “Eu sei que você pensa que entendeu o que eu disse, mas o que você não entende é que aquilo que eu disse não era o que eu quis dizer”. Invariavelmente, isso acontece no final do projeto, após compromissos de prazos de entrega terem sido estabelecidos, reputações estarem em risco e muito dinheiro estar em jogo.

Conceitos-chave

artefatos.....	147
casos de uso.....	149
colaboração.....	140
concepção.....	133
disponibilização da função de qualidade.....	146
elaboração.....	135
engenharia de requisitos.....	132
especificação.....	135
gestão de requisitos.....	138
levantamento.....	134
levantamento de requisitos.....	143
monitoramento de requisitos.....	160
negociação.....	135
negociação de requisitos.....	159

PANORAMA

O que é? Antes de iniciar qualquer trabalho técnico, é uma boa ideia criar um conjunto de requisitos para todas as tarefas de engenharia. As tarefas levam a um entendimento de qual será o impacto do software sobre o negócio, o que o cliente quer e como os usuários vão interagir com o software.

Quem realiza? Engenheiros de software (algumas vezes conhecidos no mundo da TI como engenheiros de sistemas ou “analistas”) e outros envolvidos no projeto (gerentes, clientes e usuários), todos participam da engenharia de requisitos.

Por que é importante? Projetar e construir um programa de computador elegante que resolva o problema errado não atende às necessidades de ninguém. É por isso que é importante entender o que o cliente quer antes de começar a projetar e construir um sistema baseado em computador.

Quais são as etapas envolvidas? A engenharia de requisitos começa com a concepção (uma tarefa que define a abrangência e a natureza do problema a ser resolvido). Ela prossegue para o levantamento (uma tarefa de investigação que ajuda

os envolvidos a definir o que é necessário) e, então, para a elaboração (na qual os requisitos básicos são refinados e modificados). À medida que os envolvidos definem o problema, ocorre a negociação (quais são as prioridades, o que é essencial, quando é necessário?). Por fim, o problema é especificado de algum modo e, então, é revisado ou validado para garantir que o seu entendimento sobre o problema e o dos envolvidos coincidam.

Qual é o artefato? O objetivo da engenharia de requisitos é fornecer a todas as partes um entendimento escrito do problema. Isso pode ser obtido por meio de uma série de artefatos: cenários de uso, listas de funções e características, modelos de análise ou uma especificação.

Como garantir que o trabalho foi realizado corretamente? Os artefatos da engenharia de requisitos são revisados com os envolvidos para garantir que aquilo que você entendeu é realmente aquilo que eles queriam dizer. Um alerta: mesmo depois de todas as partes terem entrado em acordo, as coisas vão mudar e continuarão mudando ao longo do projeto.

padrões de análise	157
partes envolvidas	139
pontos de vista	139
validação	136
validação de requisitos	161

Quem trabalhou na área de software e sistemas por mais do que alguns poucos anos já viveu esse pesadelo; mesmo assim, poucos aprendem a livrar-se dele. Passamos por muitas dificuldades ao tentar extrair os requisitos de nossos clientes. Temos dificuldades para entender as informações obtidas. Normalmente, registramos os requisitos de forma desorganizada e investimos pouco tempo verificando aquilo que registramos. Deixamos que as mudanças nos controlem, em vez de estabelecermos mecanismos para controlar as mudanças. Em suma, não conseguimos estabelecer uma base sólida para o sistema ou software. Todos esses problemas são desafiadores. Quando combinados, o panorama é assustador até mesmo para os gerentes e profissionais mais experientes. Mas soluções existem.

É possível afirmar que as técnicas que discutiremos neste capítulo não são a verdadeira “solução” para os desafios que acabamos de citar. Entretanto, elas fornecem uma abordagem (para compor uma estratégia) confiável para lidar com esses desafios.

8.1 Engenharia de requisitos

“A parte mais difícil ao se construir um sistema de software é decidir o que construir. Nenhuma parte do trabalho afeta tanto o sistema resultante se for feita a coisa errada. Nenhuma outra parte é mais difícil de consertar depois.”

Fred Brooks

A engenharia de requisitos estabelece uma base sólida para o projeto e para a construção. Sem ela, o software resultante tem grande probabilidade de não atender às necessidades do cliente.

Projetar e construir software é desafiador, criativo e divertido. Na verdade, construir software é tão envolvente, que muitos desenvolvedores querem começar imediatamente, antes de terem um entendimento claro daquilo que é necessário. Eles argumentam que as coisas ficarão mais claras à medida que forem construindo o software, que os envolvidos no projeto serão capazes de entender a necessidade apenas depois de examinar as primeiras iterações do software, que as coisas mudam tão rápido que qualquer tentativa de entender os requisitos de forma detalhada será perda de tempo, que o primordial é produzir um programa que funcione e que todo o resto é secundário. O que torna esses argumentos tentadores é que eles contêm elementos de verdade.¹ Porém, eles apresentam pontos fracos e podem levar um projeto ao fracasso.

O amplo espectro de tarefas e técnicas que levam a um entendimento dos requisitos é chamado de *engenharia de requisitos*. Do ponto de vista do processo de software, a engenharia de requisitos é uma ação de engenharia de software importante que se inicia durante a atividade de comunicação e continua na de modelagem. Ela deve ser adaptada às necessidades do processo, do projeto, do produto e das pessoas que estão realizando o trabalho.

A engenharia de requisitos constrói uma ponte entre o projeto e a construção, mas onde começa essa ponte? Alguém pode argumentar que ela tem sua base nos envolvidos no projeto (por exemplo, gerentes, clientes e usuários), em que é definida a necessidade do negócio, são descritos cenários de usuários, delineadas funções e recursos e identificadas restrições de projeto. Outros poderiam sugerir que ela se inicia com uma definição mais abrangente do sistema, em que o software é apenas um componente do domínio do sistema mais abrangente. Porém, independentemente do ponto de partida, a jornada pela ponte nos leva bem à frente no projeto, permitindo que examinemos o contexto do trabalho de software a ser realizado; as necessidades específicas

¹ Isto é particularmente verdade para pequenos projetos (com entregas em menos de um mês) com esforços simples. À medida que o software cresce em complexidade e tamanho, essas argumentações podem falhar.

a que o projeto e a construção devem atender; as prioridades que orientam a ordem na qual o trabalho deve ser concluído; e as informações, funções e comportamentos que terão um impacto profundo no projeto resultante.

Na última década, houve muitas mudanças tecnológicas que influenciam o processo da engenharia de requisitos [Wev11]. A computação onipresente permite que a tecnologia computacional seja integrada a muitos objetos do cotidiano. Quando esses objetos são interligados em rede, permitem a criação de perfis de usuário mais completos, com as respectivas preocupações com a privacidade e a segurança.

A ampla disponibilidade de aplicativos no mercado eletrônico levará a requisitos mais diversificados por parte dos envolvidos. Eles podem personalizar um produto para cumprir certos requisitos específicos, aplicáveis apenas a um pequeno subconjunto dos usuários. À medida que os ciclos de desenvolvimento dos produtos é reduzido, existem pressões para otimizar a engenharia de requisitos para que os produtos cheguem ao mercado mais rapidamente. Mas o problema fundamental permanece o mesmo: obter resposta oportuna e rápida, precisa e estável dos envolvidos.

A engenharia de requisitos abrange sete tarefas distintas: concepção, levantamento,* elaboração, negociação, especificação, validação e gestão. É importante notar que algumas delas ocorrem em paralelo e que todas são adaptadas às necessidades do projeto.

Concepção. Como um projeto de software é iniciado? Existe um evento que se torna o catalisador para um novo produto ou sistema de computador ou a necessidade evolui ao longo do tempo? Não há resposta definitiva para essas perguntas. Em alguns casos, uma conversa informal basta para precipitar um trabalho de engenharia de software. Entretanto, em geral, a maioria dos projetos começa quando é identificada a necessidade do negócio ou é descoberto um novo serviço ou mercado potencial. Envolvidos da comunidade de negócios (por exemplo, gerentes comerciais, pessoal de marketing, gerentes de produto) definem um plano de negócio para a ideia, tentam identificar o tamanho do mercado, fazem uma análise de viabilidade aproximada e identificam uma descrição operacional da abrangência do projeto. Todas as informações estão sujeitas a mudanças, porém é suficiente para suscitar discussões com a organização² de engenharia de software. Na concepção do projeto,³ estabelecemos um entendimento básico do problema, as pessoas que querem uma solução, a natureza da solução desejada e a eficácia da comunicação e colaboração preliminares entre os demais envolvidos e a equipe de software.

É esperado realizar um pouco de projeto durante o trabalho de levantamento de requisitos e um pouco de trabalho de levantamento de requisitos durante o projeto.

"As sementes das principais catástrofes de software normalmente são semeadas nos três primeiros meses do projeto."

Caper Jones

* N. de R.T.: O termo *elicitation* corresponde a uma investigação, aqui colocado como levantamento para entrevistar, assistir, entender os fluxos impactantes de negócios, de experiência de usuário, de integração técnica e outros aspectos restritivos que direcionam os requisitos do projeto.

² Se é para desenvolver um sistema baseado em computador, a discussão começa com um processo de engenharia de sistema. Para uma discussão detalhada sobre engenharia de sistemas, visite o site que acompanha este livro: www.mhhe.com/pressman.

³ O Processo Unificado (UP, Unified Process) visto no Cap. 4 define uma fase de concepção mais detalhada que compreende as atividades relacionadas a concepção, levantamento e elaboração discutidas neste capítulo.

Levantamento. Certamente parece bastante simples – perguntar ao cliente, aos usuários e aos demais envolvidos quais são os objetivos para o sistema ou produto, o que deve ser obtido, como o sistema ou produto atende às necessidades da empresa e, por fim, como o sistema ou produto deve ser utilizado no dia a dia. Mas isso não é simples – na verdade, é muito difícil.

Uma parte importante do levantamento é estabelecer metas de negócios [Cle10]. Sua tarefa é mobilizar os envolvidos e estimulá-los a compartilhar suas metas honestamente. Uma vez capturadas as metas, deve ser estabelecido um mecanismo de atribuição de prioridades, podendo ser criado um raciocínio lógico para a possível arquitetura do projeto (que atenda às metas dos envolvidos).

INFORMAÇÕES



Engenharia de requisitos orientada a metas

Uma *meta* é um objetivo de longo prazo que um sistema ou produto deve alcançar. As metas podem tratar de aspectos funcionais ou não funcionais (por exemplo, confiabilidade, segurança, usabilidade etc.). Frequentemente, representam uma boa maneira de explicar os requisitos aos envolvidos e, uma vez estabelecidas, podem ser usadas para gerenciar conflitos entre eles.

Modelos de objeto (Capítulos 10 e 11) e requisitos podem ser extraídos sistematicamente das metas. Um gráfico de metas, mostrando vínculos entre elas, pode fornecer certo

grau de rastreabilidade (Seção 8.2.6) entre as preocupações estratégicas de alto nível e os detalhes técnicos de baixo nível. As metas devem ser especificadas com precisão e servir de base para a elaboração, verificação/validação, gerenciamento de conflitos, negociação, explicação e evolução dos requisitos.

Muitas vezes, os conflitos detectados nos requisitos são resultado de desacordos presentes nas próprias metas. A solução de conflitos é conseguida por meio da negociação de um conjunto de metas mutuamente consensuais, coerentes entre si e com os desejos dos envolvidos. Uma discussão mais completa sobre metas e engenharia de requisitos encontra-se em um artigo de Lamsweweerde [LaM01b].

Por que é difícil entender claramente o que o cliente deseja?

Christel e Kang [Cri92] identificaram uma série de problemas que são encontrados durante o levantamento. *Problemas de escopo* ocorrem quando os limites do sistema são definidos de forma precária ou quando os clientes e usuários especificam detalhes técnicos desnecessários que podem confundir, em vez de esclarecer, os objetivos globais do sistema. *Problemas de entendimento* são encontrados quando clientes e usuários não estão completamente certos do que precisam, têm um entendimento inadequado das capacidades e limitações de seus ambientes computacionais, não possuem um entendimento completo do domínio do problema, têm problemas para transmitir suas necessidades, omitem informações que acreditam ser “óbvias”, especificam requisitos conflitantes com as necessidades de outros clientes e usuários ou especificam requisitos ambíguos ou impossíveis de ser testados. *Problemas de volatilidade* ocorrem quando os requisitos mudam com o passar do tempo. Para ajudar a superar esses problemas, devemos abordar a atividade de levantamento de requisitos de forma organizada.

Elaboração. As informações obtidas do cliente durante a concepção e o levantamento são expandidas e refinadas durante a elaboração. Essa tarefa con-

centra-se no desenvolvimento de um modelo de requisitos refinado (Capítulos 9 a 11) que identifique os diversos aspectos da função, do comportamento e das informações do software.

A elaboração é guiada pela criação e pelo refinamento de cenários que descrevem como o usuário (e outros atores) vão interagir com o sistema. Cada cenário de usuário é analisado para extrair classes de análise – entidades do domínio de negócio visíveis para o usuário. Os atributos de cada classe de análise são definidos, e os serviços⁴ exigidos de cada uma são identificados. As relações e a colaboração entre as classes são identificadas, e uma variedade de diagramas suplementares é produzida.

Negociação. Não é raro clientes e usuários pedirem mais do que é possível, dados os recursos limitados do negócio. Também é relativamente comum diferentes clientes ou usuários proporem necessidades conflitantes, argumentando que sua versão é “essencial para nossas necessidades especiais”.

É preciso conciliar esses conflitos por meio de um processo de negociação. Devemos solicitar a clientes, usuários e outros envolvidos para que ordenem seus requisitos e discutam sua prioridade. Usando uma abordagem iterativa que priorize os requisitos, avalie seus custos e riscos e trate dos conflitos internos, os requisitos são eliminados, combinados e/ou modificados, de modo que cada parte atinja certo nível de satisfação.

Especificação. No contexto de sistemas baseados em computador (e software), o termo *especificação* assume diferentes significados para diferentes pessoas. Especificação pode ser um documento por escrito, um conjunto de modelos gráficos, um modelo matemático formal, um conjunto de cenários de uso, um protótipo ou qualquer combinação dos fatores citados.

Alguns sugerem que um “modelo padrão” [Som97] deve ser desenvolvido e utilizado para a especificação, argumentando que ele leva a requisitos que são apresentados de forma consistente e, portanto, mais compreensível. Entretanto, algumas vezes é necessário permanecer flexível quando uma especificação precisa ser desenvolvida. Para sistemas grandes, um documento escrito, combinando descrições em linguagem natural e modelos gráficos, pode ser a melhor abordagem. Entretanto, talvez sejam necessários apenas cenários de uso para produtos ou sistemas menores que residam em ambientes técnicos bem compreendidos.

A elaboração é uma coisa boa, porém é preciso saber quando parar. O segredo é descrever o problema de maneira que estabeleça uma base sólida para o projeto. Caso passe desse ponto, você estará realizando um projeto.

Em uma negociação efetiva não existem ganhadores, nem perdedores. Ambos os lados ganham, pois é consolidado um “acordo” que ambas as partes aceitam.

A formalidade e o formato de uma especificação variam com o tamanho e a complexidade do software a ser construído.

⁴ O serviço manipula os dados encapsulados pela classe. Os termos operação e método também são usados neste contexto. Se os conceitos sobre orientação a objetos não são familiares a você, uma introdução básica pode ser vista no Apendice 2.



Modelo de especificação de requisitos de software

Uma *especificação de requisitos de software* (SRS, *software requirements specification*) é um artefato criado quando uma descrição detalhada de todos os aspectos do software a ser construído deve ser especificada antes de o projeto começar. É importante notar que uma SRS formal nem sempre é por escrito. Na realidade, há várias ocasiões em que o esforço gasto em uma SRS talvez fosse mais bem aproveitado em outras atividades de engenharia de software. Entretanto, quando um software for desenvolvido por terceiros, quando a falta de uma especificação criar graves problemas de negócio ou quando um sistema for extremamente complexo ou crítico para o negócio, será justificável uma SRS.

Karl Wiegers [Wie03], da Process Impact Inc., desenvolveu uma planilha bastante útil (disponível em www.processimpact.com/process_assets/srs_template.doc) que pode servir como diretriz para aqueles que precisam criar uma SRS completa. Uma descrição geral por tópicos é apresentada a seguir:

Sumário

Histórico de Revisão

1. Introdução
 - 1.1 Finalidade
 - 1.2 Convenções do documento
 - 1.3 Público-alvo e sugestões de leitura
 - 1.4 Escopo do projeto
 - 1.5 Referências

2. Descrição geral
 - 2.1 Perspectiva do produto
 - 2.2 Características do produto
 - 2.3 Classes e características do usuário
 - 2.4 Ambiente operacional
 - 2.5 Restrições de projeto e implementação
 - 2.6 Documentação para usuários
 - 2.7 Hipóteses e dependências
3. Características do sistema
 - 3.1 Características do sistema 1
 - 3.2 Características do sistema 2 (e assim por diante)
4. Requisitos de interfaces externas
 - 4.1 Interfaces do usuário
 - 4.2 Interfaces de hardware
 - 4.3 Interfaces de software
 - 4.4 Interfaces de comunicação
5. Outros requisitos não funcionais
 - 5.1 Requisitos de desempenho
 - 5.2 Requisitos de segurança – privacidade
 - 5.3 Requisitos de segurança – integridade
 - 5.4 Atributos de qualidade de software
6. Outros requisitos

Apêndice A: Glossário

Apêndice B: Modelos de análise

Apêndice C: Lista de problemas

Uma descrição detalhada de cada tópico de SRS pode ser obtida fazendo-se o download da planilha SRS na URL citada anteriormente neste quadro.

Uma preocupação fundamental durante a validação de requisitos é a consistência. Use o modelo de análise para garantir que os requisitos foram declarados de forma consistente.

Validação. Os artefatos produzidos pela engenharia de requisitos têm sua qualidade avaliada durante a etapa de validação. A validação de requisitos examina a especificação⁵ para garantir que todos os requisitos de software tenham sido declarados de forma não ambígua; que as inconsistências, omissões e erros tenham sido detectados e corrigidos; e que os artefatos estejam de acordo com os padrões estabelecidos para o processo, projeto e produto.

O principal mecanismo de validação de requisitos é a revisão técnica (Capítulo 20). A equipe de revisão que valida os requisitos é formada por engenheiros de software, clientes, usuários e outros envolvidos que examinam a especificação em busca de erros no conteúdo ou na interpretação, de áreas em que talvez sejam necessários esclarecimentos, de informações faltantes, de inconsistências (um problema grave quando são criados produtos ou sistemas grandes), de requisitos conflitantes ou de requisitos irreais (inatingíveis).

⁵ Não esqueça que a natureza da especificação vai variar para cada projeto. Em alguns casos, “especificação” é um conjunto de cenários de usuário e um pouco mais. Em outros, pode ser um documento que contém cenários, modelos e descrições escritas.

Para ilustrar alguns dos problemas que ocorrem durante a validação de requisitos, considere duas necessidades aparentemente inofensivas:

- O software deve ser amigável.
- A probabilidade de invasão não autorizada e bem-sucedida ao banco de dados deve ser menor do que 0,0001.

O primeiro requisito é vago demais para os desenvolvedores testarem ou avaliarem. O que exatamente significa “amigável”? Para sua validação, isso precisa ser quantificado ou qualificado de algum modo.

O segundo requisito tem um elemento quantitativo (“menor do que 0,0001”), mas o teste de invasões será difícil e demorado. Esse nível de segurança é realmente garantido pelo aplicativo? Outros requisitos complementares associados à segurança (por exemplo, proteção com senha, protocolo de interação especializado) podem substituir o requisito quantitativo mencionado?

Glinz [Gli09] escreve que requisitos qualitativos precisam ser representados de uma maneira que comuniquem o melhor valor. Isso significa avaliar o risco (Capítulo 35) de distribuir um sistema que não cumpra os requisitos qualitativos dos envolvidos e tente mitigar esse risco a um custo mínimo. Quanto mais crítico é o requisito qualitativo, maior é a necessidade de expressá-lo em termos quantificáveis. Os requisitos qualitativos menos críticos podem ser expressos em termos gerais. Em alguns casos, um requisito qualitativo geral pode ser verificado com uma técnica qualitativa (por exemplo, pesquisa junto ao usuário ou lista de controle). Em outras situações, os requisitos qualitativos podem ser verificados usando-se uma combinação de avaliações qualitativas e quantitativas.

INFORMAÇÕES



Lista de controle para validação de requisitos

Muitas vezes é útil examinar cada requisito em relação a um conjunto de perguntas contidas em uma lista de controle. A seguir, um pequeno subconjunto do que poderia ser aplicado:

- Os requisitos estão expressos de forma clara? Eles podem ser mal interpretados?
- A fonte (por exemplo, uma pessoa, uma regulamentação, um documento) do requisito foi identificada? A declaração final do requisito foi examinada pela fonte original ou com ela?
- O requisito está limitado em termos quantitativos?
- Quais outros requisitos se relacionam a este requisito? Eles estão claramente indicados por meio de uma matriz de referência cruzada ou algum outro mecanismo?
- O requisito viola quaisquer restrições do domínio do sistema?
- O requisito pode ser testado? Em caso positivo, podemos especificar testes (algumas vezes denominados critérios de validação) para verificar o requisito?
- O requisito pode ser rastreado por algum modelo de sistema que tenha sido criado?
- O requisito pode ser rastreado pelos objetivos globais do sistema/produto?
- A especificação está estruturada de forma que leve ao fácil entendimento, referência e tradução em artefatos mais técnicos?
- Criou-se um índice para a especificação?
- Os requisitos associados ao desempenho, ao comportamento e às características operacionais foram declarados de maneira clara? Quais requisitos parecem estar implícitos?

Gestão de requisitos. Os requisitos para sistemas baseados em computador mudam, e o desejo de mudar os requisitos persiste ao longo da vida de um sistema. A gestão de requisitos é um conjunto de atividades que ajuda a equipe de projeto a identificar, controlar e acompanhar as necessidades e suas mudanças à medida que o projeto prossegue.⁶ Muitas dessas atividades são idênticas às técnicas de gerenciamento de configurações de software (SCM, software configuration management) discutidas no Capítulo 29.

FERRAMENTAS DO SOFTWARE



Engenharia de requisitos

Objetivo: As ferramentas de engenharia de requisitos auxiliam no levantamento, na modelagem e na gestão, bem como na validação de requisitos.

Mecanismos: O mecanismo das ferramentas é variado. Em geral, as ferramentas de engenharia de requisitos constroem uma grande variedade de modelos gráficos (por exemplo, UML) que representam os aspectos informativos, funcionais e comportamentais de um sistema. Esses modelos formam a base para todas as demais atividades no processo de software.

Ferramentas representativas:⁷

Uma lista relativamente abrangente (e atualizada) de ferramentas de engenharia de requisitos pode ser encontrada no site Volvere Requirements, em www.volere.co.uk/tools.htm. As ferramentas de modelagem de requisitos são discu-

tadas nos Capítulos 9 e 10. As ferramentas citadas a seguir se concentram na gestão de requisitos.

EasyRM, desenvolvida pela Cybernetic Intelligence GmbH (<http://www.visuresolutions.com/visure-requirements-software>), Visure Requirements é uma solução de ciclo de vida de engenharia de requisitos flexível e completa, suportando captura, análise, especificação, validação e verificação, gestão e reutilização de requisitos.

Rational RequisitePro, desenvolvida pela Rational Software (www-03.ibm.com/software/products/us/en/reqpro), permite aos usuários construir um banco de dados de requisitos, representar as relações entre os requisitos e organizar, priorizar e rastrear requisitos.

Muitas outras ferramentas de gestão de requisitos podem ser encontradas no site da Volvere, citado anteriormente, no seguinte endereço www.jiludwig.com/Requirements_Management_Tools.html.

8.2 Estabelecimento da base de trabalho

Em um ambiente ideal, os envolvidos e os engenheiros de software trabalham juntos na mesma equipe.⁸ Nesses casos, fazer a engenharia de requisitos é apenas uma questão de ter conversas proveitosas com membros bem conhecidos da equipe. A realidade, entretanto, muitas vezes é bastante diferente.

Cliente(s) ou usuários podem estar em cidades ou países diferentes, podem ter apenas uma vaga ideia daquilo que é necessário, podem ter opiniões conflitantes sobre o sistema a ser construído, podem ter conhecimento técnico limitado ou, quem sabe, pouco tempo para interagir com o engenheiro que está fazendo o levantamento de requisitos. Nenhuma dessas situações é dese-

⁶ Gerenciamento formal de requisitos é iniciado somente para projetos grandes que têm centenas de requisitos. Para projetos pequenos, essa função da engenharia de requisitos é consideravelmente menos formal.

⁷ As ferramentas citadas aqui constituem uma amostra desta categoria. Os nomes das marcas são marcas registradas de seus respectivos fabricantes.

⁸ Esta abordagem é altamente recomendada para projetos que adotam a filosofia de desenvolvimento ágil.

jável; contudo, todas são relativamente comuns, e muitas vezes você é forçado a trabalhar de acordo com as limitações impostas pela situação.

Nas seções a seguir, discutiremos as etapas necessárias para estabelecer as bases para o entendimento dos requisitos de software – para que o projeto possa ser iniciado de modo que avance na direção de uma solução bem-sucedida.

8.2.1 Identificação de envolvidos

Sommerville e Sawyer [Som97] definem *envolvidos* (*stakeholder*) como “qualquer pessoa que se beneficie de forma direta ou indireta do sistema que está sendo desenvolvido”. Já identificamos os envolvidos “de sempre”: gerentes de operações, gerentes de produto, pessoal de marketing, clientes internos e externos, usuários, consultores, engenheiros de produto, engenheiros de software, engenheiros de suporte e manutenção e outros. Cada envolvido tem uma visão diferente do sistema, obtém diferentes benefícios quando o sistema é desenvolvido com êxito e está sujeito a diferentes riscos caso o trabalho de desenvolvimento venha a fracassar.

No início, devemos criar uma lista das pessoas que vão contribuir com sugestões à medida que os requisitos forem obtidos (Seção 8.3). A lista inicial crescerá à medida que os envolvidos forem contatados, pois, para cada um deles, será feita a pergunta: “Com quem mais você acha que eu devo falar?”.

Envolvido é qualquer um que tenha interesse direto ou que se beneficie do sistema a ser desenvolvido.

8.2.2 Reconhecimento de diversos pontos de vista

Como há muitos envolvidos diferentes, os requisitos do sistema serão explorados sob vários pontos de vista. Por exemplo, o grupo de marketing está interessado nas funções e recursos que vão instigar o mercado potencial, facilitando a venda do novo sistema. Os gerentes comerciais estão interessados em um conjunto de recursos que possa ser construído dentro do orçamento e que estarão prontos para atender às oportunidades de ingresso no mercado definidas. Os usuários podem querer recursos que sejam conhecidos deles e que sejam fáceis de aprender e usar. Os engenheiros de software podem estar preocupados com funções invisíveis aos envolvidos não técnicos, mas que possibilitam uma infraestrutura que dê suporte a um maior número de funções e recursos comercializáveis. Os engenheiros de suporte talvez se concentrem na facilidade de manutenção do software.

Cada uma dessas partes (e outras) contribuirá com informações para o processo de engenharia de requisitos. À medida que as informações dos diversos pontos de vista são coletadas, requisitos emergentes talvez sejam inconsistentes ou entrem em conflito uns com os outros. As informações de todos os envolvidos (inclusive os requisitos inconsistentes e conflitantes) devem ser classificadas, de maneira que permita aos tomadores de decisão escolher um conjunto internamente consistente de requisitos para o sistema.

Várias coisas podem dificultar a obtenção de requisitos para software que satisfaçam seus usuários: metas de projeto imprecisas, diferentes prioridades dos envolvidos, suposições não mencionadas, pessoas envolvidas interpretando significados de formas diferentes e requisitos expressos de um modo que

“Coloque três envolvidos em uma sala e pergunte a eles que tipo de sistema desejam. Provavelmente você obterá quatro ou mais opiniões diferentes.”

Autor desconhecido

os torna difíceis de verificar [Ale11]. O objetivo da engenharia de requisitos eficaz é eliminar ou pelo menos reduzir esses problemas.

8.2.3 Trabalho em busca da colaboração

Se cinco envolvidos estiverem ligados a um projeto de software, talvez tenhamos cinco (ou mais) opiniões diferentes sobre o conjunto de requisitos apropriado. Nos capítulos anteriores, vimos que os clientes (e outros envolvidos) devem colaborar entre si (evitando insignificantes lutas internas pelo poder) e com os profissionais de engenharia de software, caso se queira obter um sistema bem-sucedido. Mas como a colaboração é obtida?

A função de um engenheiro de requisitos é identificar áreas em comum (requisitos com os quais todos os envolvidos concordam) e áreas de conflito ou inconsistência (requisitos desejados por um envolvido, mas que estão em conflito com os de outro envolvido). É claro que é a última categoria que representa um desafio.

INFORMAÇÕES



Utilização de "pontos de prioridade"

Um modo de resolver requisitos conflitantes e, ao mesmo tempo, entender a importância relativa de todas as necessidades é usar um esquema de "votação" baseado nos *pontos de prioridade*. Todos os envolvidos recebem certo número de pontos de prioridade que podem ser "gastos" em um número qualquer de requisitos. É apresenta-

da uma lista de requisitos e cada envolvido indica a importância relativa de cada um deles (sob o seu ponto de vista), gastando um ou mais pontos de prioridade nele. Pontos gastos não podem ser reutilizados. Uma vez que os pontos de prioridade de um envolvido tenham se esgotado, ele não tem como votar ou priorizar. O total de pontos dados por todos os envolvidos a cada requisito dá um quadro comparativo (*ranking*) da importância global de cada requisito.

Colaboração não significa necessariamente que os requisitos são definidos por um comitê. Em muitos casos, os envolvidos colaboram dando suas visões dos requisitos, mas um "defensor dos projetos" forte (por exemplo, um gerente comercial ou um técnico sênior) pode tomar a decisão final sobre quais requisitos serão cortados.

8.2.4 Questões iniciais

As perguntas feitas na concepção do projeto devem ser "livres de contexto" [Gau89]. O primeiro conjunto de perguntas livres de contexto foca no cliente e outros envolvidos, nos benefícios e nas metas de projeto globais. Por exemplo, poderíamos perguntar:

- Quem está por trás da solicitação deste trabalho?
- Quem vai usar a solução?
- Qual será o benefício econômico de uma solução bem-sucedida?
- Há outra fonte para a solução de que você precisa?

"É melhor conhecer algumas perguntas do que todas as respostas."

James Thurber

Essas perguntas ajudam a identificar todos os envolvidos interessados no software a ser criado. Além disso, identificam o benefício mensurável de uma implementação bem-sucedida e possíveis alternativas para o desenvolvimento de software personalizado.

O próximo conjunto de perguntas permite entender melhor o problema e permite que o cliente expresse suas percepções sobre uma solução:

- Como você caracterizaria uma “boa” saída, que seria gerada por uma solução bem-sucedida?
- Qual(is) problema(s) esta solução vai resolver?
- Você poderia me indicar (ou descrever) o ambiente de negócios em que a solução será usada?
- Aspectos ou restrições de desempenho afetam a maneira com que a solução será abordada?

Quais perguntas vão ajudá-lo a obter um entendimento preliminar do problema?

O conjunto final de perguntas concentra-se na eficiência da atividade de comunicação em si. Gause e Weinberg [Gau89] chamam esse conjunto de “metaperguntas” e propõem a seguinte lista (sintetizada):

- Você é a pessoa correta para responder a estas perguntas? Suas respostas são “oficiais”?
- Minhas perguntas são relevantes para o problema que você tem?
- Estaria eu fazendo perguntas demais?
- Alguma outra pessoa poderia me prestar informações adicionais?
- Deveria eu perguntar-lhe algo mais?

“Aquele que faz uma pergunta é tolo por cinco minutos; aquele que não a faz é tolo para sempre.”

Provérbio chinês

Essas (e outras) perguntas ajudarão “quebrar o gelo” e iniciar o processo de comunicação que é essencial para o êxito do levantamento. Uma reunião no formato perguntas e respostas, entretanto, não é uma abordagem que tem obtido grande sucesso. Na realidade, a sessão de perguntas e respostas deveria ser usada apenas no primeiro encontro e, depois, ser substituída pelo formato de levantamento de requisitos que combina elementos de resolução de problemas, negociação e especificação. Uma abordagem desse tipo é apresentada na Seção 8.3.

8.2.5 Requisitos não funcionais

Um *requisito não funcional* (NFR, *nonfunctional requirement*) pode ser descrito como um atributo de qualidade, de desempenho, de segurança ou como uma restrição geral em um sistema. Frequentemente, os envolvidos têm dificuldade de articulá-los. Chung [Chu09] sugere a existência de uma ênfase demasiada em relação à funcionalidade do software, embora o software talvez não seja útil ou aproveitável sem as necessárias características não funcionais.

Na Seção 8.3.2, discutimos uma técnica chamada *disponibilização da função de qualidade* (QFD). A disponibilização da função de qualidade tenta traduzir as necessidades ou metas não mencionadas do cliente em requisitos do

sistema. Os requisitos não funcionais frequentemente são listados separadamente em uma especificação de requisitos de software.

Como auxiliar da QFD, é possível definir uma abordagem de duas fases [Hne11] que pode ajudar uma equipe de software e outros envolvidos na identificação de requisitos não funcionais. Durante a primeira fase, é estabelecido um conjunto de diretrizes de engenharia de software para o sistema a ser construído. Isso inclui diretrizes de melhor prática, mas também trata do estilo arquitetural (Capítulo 13) e do uso de padrões de projeto (Capítulo 16). Então, é feita uma lista de NFRs (por exemplo, requisitos que tratam da usabilidade, teste, segurança ou manutenção). Uma tabela simples lista NFRs como *rótulos de coluna* e as diretrizes de engenharia de software como *rótulos de linha*. Uma matriz de relações compara cada diretriz com todas as outras, ajudando a equipe a avaliar se cada par de diretrizes é *complementar*, *sobreposto*, *conflitante* ou *independente*.

Na segunda fase, a equipe prioriza cada requisito não funcional, criando um conjunto homogêneo de requisitos não funcionais, usando um conjunto de regras [Hne11] que estabelecem quais diretrizes vão ser implementadas e quais vão ser rejeitadas.

8.2.6 Rastreabilidade

Rastreabilidade é um termo da engenharia de software que se refere a mapeamentos documentados entre os artefatos de engenharia de software (por exemplo, requisitos e casos de teste). Uma *matriz de rastreabilidade* permite a um engenheiro de requisitos representar a relação entre os requisitos e outros artefatos da engenharia de software. As linhas da matriz de rastreabilidade são rotuladas com os nomes dos requisitos, e as colunas podem ser rotuladas com o nome de um artefato da engenharia de software (por exemplo, um elemento do projeto ou um caso de teste). Uma célula da matriz é marcada para indicar a presença de um vínculo entre as duas.

As matrizes de rastreabilidade podem dar suporte a uma variedade de atividades de desenvolvimento de engenharia. Elas podem propiciar continuidade para os desenvolvedores à medida que um projeto passa de uma fase para outra, independentemente do modelo de processo que esteja sendo usado. Muitas vezes, as matrizes de rastreabilidade podem ser usadas para garantir que os artefatos de engenharia consideram todos os requisitos.

À medida que o número de requisitos e o número de artefatos aumentam, torna-se cada vez mais difícil manter a matriz de rastreabilidade atualizada. Contudo, é importante criar algumas maneiras de monitorar o impacto e a evolução dos requisitos do produto [Got11].

8.3 Levantamento de requisitos

O levantamento de requisitos (também chamado *elicitação de requisitos*) combina elementos de solução de problemas, elaboração, negociação e especificação. Para estimular uma abordagem colaborativa e orientada a equipes em relação ao levantamento de requisitos, os envolvidos trabalham juntos para identificar o problema, propor elementos da solução, negociar

diferentes abordagens e especificar um conjunto preliminar de requisitos da solução [Zah90].⁹

8.3.1 Coleta colaborativa de requisitos

Muitas abordagens para a coleta colaborativa de requisitos foram propostas. Cada uma faz uso de um cenário ligeiramente diferente, porém todas aplicam alguma variação das seguintes diretrizes básicas:

- As reuniões (reais ou virtuais) são conduzidas por e com a participação tanto dos engenheiros de software quanto de outros envolvidos.
- São estabelecidas regras para preparação e participação.
- É sugerida uma agenda suficientemente formal para cobrir todos os pontos importantes; porém, suficientemente informal para estimular o fluxo livre de ideias.
- Um “facilitador” (pode ser um cliente, um desenvolvedor ou uma pessoa de fora) dirige a reunião.
- É utilizado um “mecanismo de definições” (planilhas, *flip charts*, adesivos de parede ou um boletim eletrônico, salas de bate-papo ou fóruns virtuais).

O objetivo é identificar o problema, propor elementos da solução, negociar diferentes abordagens e especificar um conjunto preliminar de requisitos da solução em uma atmosfera que seja propícia para o cumprimento da meta.

Durante a concepção, um “pedido de produto” de uma ou duas páginas é gerado (Seção 8.2). São escolhidos local, hora e data para a reunião; é escolhido um facilitador; e os membros da equipe de software e de outros departamentos envolvidos são convidados a participar. A solicitação de produto é distribuída a todos os participantes antes da data da reunião.

Como exemplo,¹⁰ considere o trecho de uma solicitação de produto redigida por uma pessoa do marketing envolvida no projeto *CasaSegura*. Essa pessoa escreve a seguinte narrativa sobre a função de segurança domiciliar que faz parte do *CasaSegura*:

Nossa pesquisa indica que o mercado para sistemas de gestão domiciliar está crescendo a taxas de 40% ao ano. A primeira função do *CasaSegura* a lançarmos no mercado deveria ser a função de segurança domiciliar. A maioria das pessoas está familiarizada com “sistemas de alarme”, então, isso seria algo fácil de vender.

A função de segurança domiciliar protegeria e/ou reconheceria uma série de “situações” indesejáveis, como invasão, incêndio, inundação, níveis de monóxido de carbono e outras. Ela vai usar nossos sensores sem fio para detectar cada situação, pode ser programada pelo proprietário e ligará automaticamente para um órgão de vigilância quando uma situação for detectada.

Quais são as diretrizes básicas para conduzir uma reunião de coleta colaborativa de requisitos?

Joint Application Development (JAD) é uma técnica popular para levantamento de requisitos. Uma excelente descrição sobre ela pode ser encontrada em www.carolla.com/wp-jad.htm.

⁹ Esta abordagem é chamada algumas vezes de *técnica de especificação simplificada de aplicação* (FAST, *facilitated application specification technique*).

¹⁰ Este exemplo (com variações e ampliações) é usado para ilustrar muitos métodos de engenharia de software em muitos dos capítulos que se seguem. Como exercício, vale a pena conduzir a sua própria reunião de levantamento de requisitos e desenvolver uma lista para este exemplo.

Se um sistema ou produto vai atender a muitos usuários, esteja absolutamente certo de que os requisitos foram extraídos de uma amostra representativa deles. Se apenas um usuário definiu todos os requisitos, o risco de não aceitação é grande.

"Fatos não deixam de existir por serem ignorados."

Aldous Huxley

Evite o impulso de detonar a sugestão de um cliente classificando-a como "muito cara" ou "inviável". O objetivo aqui é negociar uma lista que seja aceitável para todos. Para tanto, você deve ser receptivo a novas ideias.

Na realidade, outras pessoas contribuiriam para essa narrativa durante a reunião para levantamento de requisitos, e um número consideravelmente maior de informações ficaria disponível. Contudo, mesmo com essas informações adicionais, a ambiguidade está presente, provavelmente existem omissões e podem ocorrer erros. Por enquanto, a "descrição funcional" anterior será suficiente.

Ao rever a solicitação de produto nos dias que antecedem a reunião, é pedida a cada participante uma lista de objetos que fazem parte do ambiente que cerca o sistema, outros que devem ser produzidos pelo sistema e aqueles usados pelo sistema para desempenhar suas funções. Além disso, é pedido a cada um outra lista de serviços (processos ou funções) que manipulam ou interagem com os objetos. Por fim, também são desenvolvidas listas de restrições (por exemplo, custo, dimensões, regras comerciais) e de critérios de desempenho (por exemplo, velocidade, precisão). Os participantes são informados que as listas não precisam ser exaustivas, mas devem refletir a percepção do sistema de cada pessoa.

Entre os objetos descritos para o *CasaSegura*, poderíamos ter o painel de controle, os detectores de fumaça, os sensores para janelas e portas, os detectores de movimento, um alarme, um evento (por exemplo, um sensor foi ativado), um display, um PC, os números de telefone, uma ligação telefônica e assim por diante. A lista de serviços poderia incluir *configurar* o sistema, *acionar* o alarme, *monitorar* os sensores, *ligar* para o telefone, *programar* o painel de controle e *ler* o display (note que os serviços atuam sobre os objetos). De maneira similar, cada participante vai criar listas de restrições (por exemplo, o sistema tem de reconhecer quando os sensores não estão operando, ser amigável, conectar-se diretamente a uma linha telefônica comum) e de critérios de desempenho (por exemplo, um evento de sensor seria reconhecido em um intervalo de um segundo e seria implementado um esquema de prioridade para os eventos).

As listas de objetos poderiam ser fixadas nas paredes da sala de reuniões utilizando-se folhas de papel grandes coladas nas paredes com fitas adesivas ou escritas em um mural. Também poderiam ser postadas em um fórum do grupo, em um site interno ou colocadas em um ambiente de rede social para revisão antes da reunião. De modo ideal, cada entrada deveria ser capaz de ser manipulada separadamente, de modo que as listas pudessem ser combinadas; as entradas, excluídas; e as adições, feitas. Nesse estágio, críticas e polêmicas são estritamente proibidas.

Depois que as listas forem apresentadas, o grupo cria uma lista única, eliminando entradas repetidas e acrescentando ideias novas que surjam durante a discussão, mas não excluindo nada. Segue-se então uma discussão (coordenada pelo facilitador). A lista combinada é reduzida, ampliada ou redigida de outra maneira para refletir apropriadamente o produto ou sistema a ser desenvolvido. O objetivo é criar uma lista consensual de objetos, serviços, restrições e desempenho para o sistema a ser construído.

Em muitos casos, um objeto ou serviço descrito em uma lista exigirá mais explicações. Para isso, os envolvidos desenvolvem *miniespecificações* para as entradas nas listas ou criam um caso de uso (Seção 8.4) que envolva o objeto ou

serviço. Por exemplo, a miniespecificação para o objeto **Painel de Controle** do *CasaSegura* poderia ser:

O painel de controle é a unidade que pode ser montada na parede, com tamanho aproximado de 230 x 130 mm. O painel de controle tem conectividade sem fio a sensores e a um PC. A interação com o usuário ocorre por um teclado numérico contendo 12 teclas. Um display colorido OLED de 75 x 75 mm fornece o feedback do usuário. O software fornece prompts interativos, eco e funções similares.

As miniespecificações são apresentadas a todos os envolvidos para discussão. Acréscimos, supressões e mais detalhes são feitos. Em alguns casos, o desenvolvimento de miniespecificações vai revelar novos objetos, serviços, restrições ou requisitos de desempenho a ser acrescentados às listas originais. Durante todas as discussões, a equipe pode levantar um assunto que não pôde ser resolvido durante a reunião. É mantida uma *lista de questões pendentes* para que essas ideias sejam trabalhadas posteriormente.



Realização de uma reunião de levantamento de requisitos

Cena: Uma sala de reunião. A primeira reunião para levantamento de requisitos está em andamento.

Atores: Jamie Lazar, membro da equipe de software; Vinod Raman, membro da equipe de software; Ed Robbins, membro da equipe de software; Doug Miller, gerente da engenharia de software; três membros do Depto. de Marketing; um representante da Engenharia de Produto e um facilitador.

Conversa:

Facilitador (apontando para uma lousa branca): Portanto, esta é a lista atual de objetos e serviços para a função segurança domiciliar.

Representante do Depto. de Marketing: Achamos que ela cobre quase todas as funcionalidades.

Vinod: Alguém não mencionou que eles queriam que todas as funcionalidades do *CasaSegura* pudessem ser acessadas via Internet? Isso incluiria a função de segurança domiciliar, não?

Representante do Depto. de Marketing: Sim, você está certo... Teremos de acrescentar essa funcionalidade e os objetos apropriados.

Facilitador: Isso também não acrescentaria restrições?

Jamie: Sim, técnicas e jurídicas.

Representante da Engenharia de Produto: O que você quer dizer?

Jamie: É melhor termos certeza de que um intruso não conseguirá invadir o sistema, desarmá-lo e roubar o local ou coisa pior. Uma grande responsabilidade sobre nós.

Doug: Uma grande verdade.

Representante do Depto. de Marketing: Mas ainda assim precisamos disso... apenas certifiquem-se de impedir uma invasão.

Ed: É fácil falar, o duro é fazer...

Facilitador (interrompendo): Não gostaria de discutir esta questão agora. Anotemos a questão para ser trabalhada no futuro e prossigamos.

(Doug, atuando como o secretário da reunião, faz uma anotação.)

Facilitador: Tenho a impressão de que ainda há mais coisas a serem consideradas aqui.

(O grupo gasta os 20 minutos seguintes refinando e expandindo os detalhes da função segurança domiciliar.)

Muitas preocupações dos envolvidos (por exemplo, precisão, acessibilidade dos dados, segurança) formam a base para os requisitos não funcionais do sistema (Seção 8.2). À medida que os envolvidos declaram essas preocupações, os engenheiros de software devem considerá-las no contexto do sistema a ser

construído. Dentre as perguntas que devem ser respondidas [Lag10], estão as seguintes:

- Podemos construir o sistema?
- Esse processo de desenvolvimento nos permitirá superar nossos concorrentes no mercado?
- Existem recursos adequados para construir e manter o sistema proposto?
- O desempenho do sistema vai atender às necessidades de nossos clientes?

As respostas dessas e de outras perguntas vão evoluir com o passar do tempo.

8.3.2 Aplicação da qualidade por QFD (Quality Function Deployment)

O QFD define necessidades de modo a maximizar a satisfação do cliente.

Todo mundo quer implementar um monte de requisitos fascinantes; porém, tenha cuidado. É assim que o "surgimento descontrolado de novos requisitos" se estabelece. Por outro lado, requisitos fascinantes levam a um produto revolucionário (ou disruptivo)!

Informações úteis sobre o QFD podem ser obtidas em www.qfdi.org.

A *disponibilização da função de qualidade* (QFD, *quality function deployment*) é uma técnica de gestão da qualidade que traduz as necessidades do cliente para requisitos técnicos do software. A QFD “concentra-se em maximizar a satisfação do cliente por meio do processo de engenharia de software” [Zul92]. Para tanto, enfatiza o entendimento daquilo que é valioso para o cliente e emprega esses valores ao longo do processo de engenharia.

No contexto da QFD, *requisitos normais* identificam os objetivos e metas declarados para um produto ou sistema durante as reuniões com o cliente. Se esses requisitos estiverem presentes, o cliente ficará satisfeito. Os *requisitos esperados* estão implícitos no produto ou sistema e podem ser tão básicos, que o cliente não os declara explicitamente. Sua ausência será causa de grande insatisfação. *Requisitos fascinantes* vão além da expectativa dos clientes e demonstram ser muito satisfatórios quando presentes.

Embora os conceitos de QFD possam ser aplicados ao longo de todo o processo de software [Par96a], técnicas de QFD específicas são aplicáveis à atividade de levantamento de requisitos. O QFD usa observação e entrevistas com clientes, pesquisas e exame de dados históricos (por exemplo, relatórios de problemas) como evidências para a atividade de levantamento de requisitos. Esses dados são então transformados em uma tabela de requisitos – denominada *tabela da voz do cliente* –, revisada com o cliente e outros envolvidos. Uma série de diagramas, matrizes e métodos de avaliação é então usada para extrair os requisitos esperados e para tentar obter os requisitos fascinantes [Aka04].

8.3.3 Cenários de uso

À medida que os requisitos são reunidos, uma visão geral das funções e características começa a se materializar. Entretanto, é difícil passar para atividades mais técnicas da engenharia de software até que se entenda como tais funções e características serão usadas por diferentes classes de usuários. Para tanto, os desenvolvedores e usuários podem criar um conjunto de cenários que identifique um roteiro de uso para o sistema a ser construído. Os cenários, normalmente chamados de *casos de uso* [Jac92], fornecem uma descrição de como o sistema será utilizado. Casos de uso são discutidos de forma mais detalhada na Seção 8.4.

CASASEGURA

**Desenvolvimento de um cenário de uso preliminar**

Cena: Sala de reuniões, onde prossegue a primeira reunião de levantamento de requisitos.

Atores: Jamie Lazar, membro da equipe de software; Vinod Raman, membro da equipe de software; Ed Robbins, membro da equipe de software; Doug Miller, gerente da engenharia de software; três membros do Depto. de Marketing; um representante da Engenharia de Produto; e um facilitador.

Conversa:

Facilitador: Andamos conversando sobre a segurança de acesso à funcionalidade do CasaSegura que poderá ser acessada via Internet. Gostaria de tentar algo. Vamos desenvolver um cenário de uso para acesso à função segurança domiciliar.

Jamie: Como?

Facilitador: Podemos fazer isso de várias maneiras, mas, por enquanto, gostaria de manter as coisas informais. Conte (ele aponta para um representante do Depto. de Marketing) como você imagina o acesso ao sistema.

Representante do Depto. de Marketing: Hum... bem, esse é o tipo de coisa que eu faria se estivesse fora e tivesse que deixar alguém em casa, por exemplo, uma empregada ou um encanador, que não teriam o código de acesso.

Facilitador (sorrindo): Esse é o motivo... diga como você faria isso realmente.

Representante do Depto. de Marketing: Hum... a primeira coisa de que precisaria seria um PC. Entraria em um site que manteríamos para todos os usuários do CasaSegura. Forneceria meu nome de usuário e...

Vinod (interrompendo): A página teria de ser segura e criptografada para garantir que estamos seguros e...

Facilitador (interrompendo): Essas são informações adequadas, Vinod, porém técnicas. Concentremo-nos apenas em como o usuário usará essa capacidade, certo?

Vinod: Sem problemas.

Representante do Depto. de Marketing: Bem, como estava dizendo, entraria em um site e forneceria meu nome de usuário e dois níveis de senhas.

Jamie: O que acontece se eu esquecer minha senha?

Facilitador (interrompendo): Excelente observação, Jamie, mas não tratemos disso agora. Faremos um registro de sua observação e a chamaremos de exceção. Tenho certeza de que existirão outras.

Representante do Depto. de Marketing: Após introduzir as senhas, uma tela representando todas as funções do CasaSegura aparecerá. Eu selecionaria a função de segurança domiciliar. O sistema poderia solicitar que eu verificasse quem sou eu, digamos, perguntando meu endereço ou telefone ou algo do gênero. Em seguida, ele exibiria uma imagem do painel de controle do sistema de segurança com uma lista das funções que eu poderia executar – armar o sistema, desarmá-lo, desarmar um ou mais sensores. Suponho que ele também me permitiria reconfigurar zonas de segurança e outros itens similares, mas não tenho certeza disso.

(Enquanto o representante do Depto. de Marketing continua falando, Doug faz anotações de maneira ininterrupta; essas formam a base para o primeiro cenário de uso informal. Como alternativa, poderia ser solicitado ao representante do Depto. de Marketing que escrevesse o cenário, mas isso seria feito fora da reunião.)

8.3.4 Artefatos do levantamento de requisitos

Os artefatos produzidos pelo levantamento de requisitos vão variar dependendo do tamanho do sistema ou produto a ser construído. Para a maioria dos sistemas, entre os artefatos, temos: (1) uma declaração de necessidade e viabilidade, (2) uma declaração da abrangência do sistema ou produto com escopo limitado, (3) uma lista de clientes, usuários e outros envolvidos que participaram do levantamento de requisitos, (4) uma descrição do ambiente técnico do sistema, (5) uma lista dos requisitos (preferivelmente organizada por função) e as restrições do domínio que se aplicam a cada um, (6) um conjunto de cenários de utilização dando uma ideia do uso do sistema ou produto sob diferentes condições operacionais e (7) quaisquer protótipos desenvolvidos para definir melhor os requisitos. Cada um desses artefatos é revisado por todas as pessoas que participaram do levantamento de requisitos.

Quais informações são produzidas como consequência do levantamento de requisitos?

Jornadas de usuário são o modo de documentar os requisitos levantados a partir dos clientes nos modelos de processo ágeis.

O que é serviço no contexto dos métodos orientados a serviços?

O levantamento de requisitos para métodos orientados a serviços refina os serviços prestados por um aplicativo. Um *ponto de contato* representa uma oportunidade para o usuário interagir com o sistema a fim de receber um serviço desejado.

8.3.5 Levantamento de requisitos ágil

No contexto de um processo ágil, o levantamento de requisitos é feito pedindo-se aos envolvidos para que criem *jornadas de usuário**. Cada jornada de usuário descreve um requisito simples do sistema, redigido do ponto de vista do usuário. As jornadas de usuário podem ser escritas em cartões de anotação pequenos, tornando fácil para os desenvolvedores escolher e gerenciar um subconjunto dos requisitos a implementar no próximo incremento do produto. Os proponentes afirmam que usar cartões de anotação redigidos na própria linguagem do usuário permite aos desenvolvedores mudar o foco da comunicação com os envolvidos para os requisitos selecionados, em vez de usar sua própria pauta [Mai10a].

Embora a abordagem ágil para o levantamento de requisitos seja atraente para muitas equipes de software, os críticos alegam que uma consideração das metas comerciais globais e dos requisitos não funcionais são incompletas. Em alguns casos, são necessários retrabalhos para incorporar questões de desempenho e segurança. Além disso, histórias de usuário podem não fornecer uma base suficiente para a evolução do sistema com o passar do tempo.

8.3.6 Métodos orientados a serviços

O desenvolvimento orientado a serviços vê um sistema como um agrupamento de serviços. Um *serviço* pode ser algo “tão simples como fornecer uma única função, por exemplo, um mecanismo baseado em pedido/resposta que fornece uma série de números aleatórios, ou um agrupamento de elementos complexos, como a API de Web service” [Mic12].

No desenvolvimento orientado a serviços, o levantamento de requisitos se concentra na definição dos serviços a serem prestados por um aplicativo. Como uma metáfora, considere o serviço fornecido quando você visita um hotel de luxo. Um porteiro cumprimenta os hóspedes. Um manobrista estaciona os carros deles. O recepcionista faz o check-in dos hóspedes. Um mensageiro cuida das malas. O concierge ajuda o hóspede nas acomodações. Cada contato ou *ponto de contato* entre um hóspede e um funcionário do hotel é programado para melhorar a visita ao hotel e representa um serviço oferecido.

A maioria dos métodos de projeto de serviço enfatiza a compreensão do cliente, o pensamento criativo e a rápida construção de soluções [Mai10b]. Para atingir essas metas, o levantamento de requisitos pode incluir estudos etnográficos¹¹, oficinas de inovação e protótipos iniciais de alta fidelidade. As técnicas de levantamento de requisitos também devem obter informações sobre a marca e sua percepção por parte dos envolvidos. Além de estudar como a marca é utilizada pelos clientes, os analistas precisam de estratégias para descobrir e documentar requisitos relativos às qualidades desejadas das novas experiências do usuário. As jornadas de usuário são úteis sob esse aspecto.

* N. de R.T.: Também é comum o uso do termo “história de usuário”.

¹¹ Estudar o comportamento do usuário no ambiente em que o software proposto será usado.

Os requisitos para pontos de contato devem ser caracterizados de uma maneira que indique a realização dos requisitos globais do serviço. Isso sugere que cada requisito deve ser rastreável até um serviço específico.

8.4 Desenvolvimento de casos de uso

Em uma obra que discute como redigir casos de uso eficazes, Alistair Cockburn [Coc01b] observa que “um caso de uso captura um contrato... [que] descreve o comportamento do sistema sob várias condições, à medida que o sistema responde a uma solicitação de um de seus envolvidos...”. Basicamente, um caso de uso conta uma jornada estilizada sobre como um usuário (desempenhando um de uma série de papéis possíveis) interage com o sistema sob um conjunto de circunstâncias específicas. A jornada poderia ser um texto narrativo, uma descrição geral das tarefas ou interações, uma descrição baseada em modelos ou uma representação esquemática. Independentemente de sua forma, um caso de uso representa o software ou o sistema do ponto de vista do usuário.

O primeiro passo ao escrever um caso de uso é definir o conjunto de “atores” envolvidos na história. *Atores* são as diferentes pessoas (ou dispositivos) que usam o sistema ou produto no contexto da função e do comportamento a ser descritos. Os atores representam os papéis que pessoas (ou dispositivos) desempenham enquanto o sistema opera. Definido de maneira um pouco mais formal, ator é qualquer coisa que se comunica com o sistema ou produto e que é externa ao sistema em si. Todo ator possui uma ou mais metas ao usar o sistema.

É importante notar que ator e usuário não são necessariamente a mesma coisa. O usuário típico poderia desempenhar inúmeros papéis diferentes ao usar um sistema, ao passo que o ator representa uma classe de entidades externas (normalmente, mas não sempre, pessoas) que desempenham apenas um papel no contexto do caso de uso. Como exemplo, consideremos um operador de máquina (um usuário) que interage com o computador de controle de uma célula de fabricação contendo uma série de robôs e máquinas, comandada por controle numérico. Após uma revisão cuidadosa dos requisitos, o software para o computador de controle exige quatro modos (papéis) diferentes para interação: modo de programação, modo de teste, modo de monitoramento e modo de diagnóstico. Portanto, podem ser definidos quatro atores: programador, testador, monitorador e diagnosticador. Em alguns casos, o operador da máquina pode desempenhar todos esses papéis. Em outros, pessoas diferentes poderiam desempenhar o papel de cada ator.

Como o levantamento de requisitos é uma atividade evolutiva, nem todos os atores são identificados durante a primeira iteração. É possível identificar atores primários [Jac92] durante a primeira iteração e atores secundários quando mais fatos são aprendidos sobre o sistema. Os *primários* interagem para atingir a função necessária e obter o benefício desejado do sistema. Eles trabalham com o software direta e frequentemente. Os *secundários* dão suporte ao sistema para que os primários possam realizar seu trabalho.

Os casos de uso são definidos sob o ponto de vista de um ator. Ator é um papel que as pessoas (usuários) ou dispositivos desempenham ao interagir com o software.

Um excelente artigo sobre casos de uso pode ser baixado de www.ibm.com/developerworks/webservices/library/codesign7.html.

O que preciso saber para desenvolver um caso de uso eficaz?

Depois de identificados os atores, os casos de uso podem ser desenvolvidos. Jacobson [Jac92] sugere algumas perguntas¹² que devem ser respondidas por um caso de uso:

- Quem é o ator primário e quem é (são) o(s) ator(es) secundário(s)?
- Quais são as metas do ator?
- Que condições devem existir antes de uma jornada começar?
- Que tarefas ou funções principais são realizadas pelo ator?
- Que exceções poderiam ser consideradas à medida que uma jornada é descrita?
- Quais são as variações possíveis na interação do ator?
- Que informações de sistema o ator adquire, produz ou modifica?
- O ator terá de informar o sistema sobre mudanças no ambiente externo?
- Que informações o ator deseja do sistema?
- O ator gostaria de ser informado sobre mudanças inesperadas?

Relembrando os requisitos básicos do *CasaSegura*, definimos quatro atores: **proprietário** (um usuário), **gerente de ativação** (provavelmente a mesma pessoa que o **proprietário**, porém desempenhando um papel diferente), **sensores** (dispositivos conectados ao sistema) e o **subsistema de monitoramento e resposta** (a estação central que monitora a função de segurança domiciliar do *CasaSegura*). Para os propósitos deste exemplo, consideramos apenas o ator **proprietário**. O ator **proprietário** interage com a função de segurança doméstica de diferentes maneiras, usando o painel de controle de alarme ou um PC. O proprietário (1) digita uma senha para permitir todas as outras interações, (2) consulta sobre o status de uma zona de segurança, (3) consulta sobre o status de um sensor, (4) pressiona o botão de pânico em caso de emergência e (5) ativa/desativa o sistema de segurança.

Considerando a situação em que o proprietário do imóvel usa o painel de controle, o caso de uso básico para ativação do sistema é o seguinte:¹³

1. O proprietário olha o painel de controle do *CasaSegura* (Figura 8.1) para determinar se o sistema está pronto para entrada. Se o sistema não estiver pronto, será mostrada uma mensagem *não disponível* no display LCD e o proprietário terá de fechar manualmente as janelas para que a mensagem *não disponível* desapareça. [A mensagem *não disponível* significa que um sensor está aberto; isto é, que uma porta ou janela está aberta.]
2. O proprietário usa o teclado numérico para introduzir uma senha de quatro dígitos. A senha é comparada com a senha válida armazenada no sistema. Se estiver incorreta, o painel de controle emitirá um bipe uma vez e se reiniciará

¹² As perguntas de Jacobson foram ampliadas para fornecer uma visão mais completa do conteúdo do caso de uso.

¹³ Note que este caso de uso difere daquele na qual o sistema é acessado pela Internet. Neste caso, a interação ocorre por meio do painel de controle, e não da interface de um desktop ou de um dispositivo móvel.

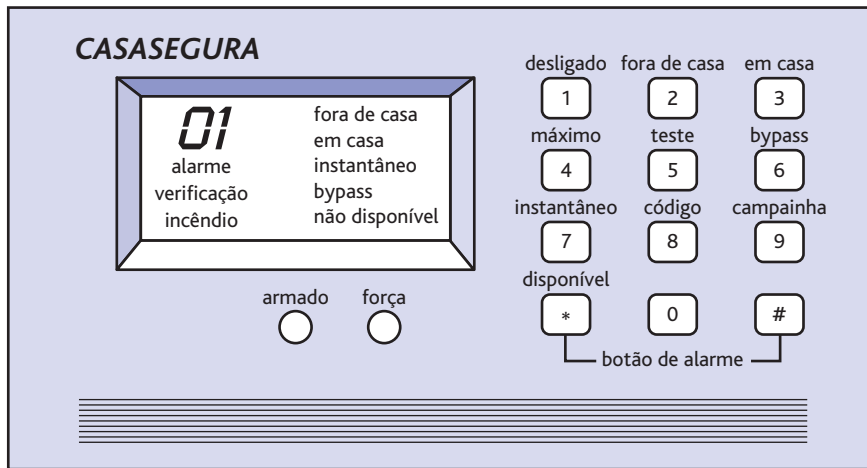


FIGURA 8.1 Painel de controle do CasaSegura.*

automaticamente na espera de entrada adicional. Se a senha estiver correta, o painel de controle aguarda por novas ações.

3. O proprietário seleciona e digita *em casa* ou *fora de casa* (consulte a Figura 8.1) para ativar o sistema. *Em casa* ativa apenas sensores periféricos (os sensores para detecção de movimento interno são desativados). *Fora de casa* ativa todos os sensores.
4. Quando ocorre a ativação, uma luz de alarme vermelha pode ser observada pelo proprietário.

O caso de uso básico apresenta uma história detalhada que descreve a interação entre o ator e o sistema.

Em muitas ocasiões, os casos de uso são mais elaborados para dar um nível de detalhes consideravelmente maior sobre a interação. Por exemplo, Cockburn [Coc01b] sugere o seguinte modelo para descrições detalhadas de casos de uso:

Caso de uso:	<i>IniciarMonitoramento</i>
Ator primário:	Proprietário.
Meta no contexto:	Ativar o sistema para monitoramento dos sensores quando o proprietário deixa a casa ou nela permanece.
Precondições:	O sistema foi programado para uma senha e para reconhecer vários sensores.
Disparador:	O proprietário decide “acionar” o sistema, isto é, ativar as funções de alarme.

Muitas vezes, os casos de uso são redigidos informalmente. Entretanto, use o modelo aqui mostrado para garantir que você tratou de todas as questões-chave.

* N. de R.T.: O termo “disponível” pode ser entendido como “pronto”: não disponível – não pronto; disponível – pronto.

Cenário:

1. Proprietário: observa o painel de controle
2. Proprietário: introduz a senha
3. Proprietário: seleciona “em casa” ou “fora de casa”
4. Proprietário: observa a luz de alarme vermelha para indicar que o *CasaSegura* foi armado

Exceções:

1. O painel de controle encontra-se no estado *não disponível*: o proprietário verifica todos os sensores para determinar quais estão abertos; fechando-os.
2. Senha incorreta (o painel de controle emite um bipe): o proprietário introduz novamente a senha, desta vez correta.
3. Senha não reconhecida: o subsistema de monitoramento e resposta deve ser contatado para reprogramar a senha.
4. É selecionado *em casa*: o painel de controle emite dois bipes, e uma luz de *em casa* é acesa; os sensores periféricos são ativados.
5. É selecionado *fora de casa*: o painel de controle emite três bipes, e uma luz de *fora de casa* é acesa; todos os sensores são ativados.

Prioridade: Essencial, deve ser implementada

Quando disponível: Primeiro incremento

Frequência de uso: Várias vezes por dia

Canal com o ator: Via interface do painel de controle

Atores secundários: Técnico de suporte, sensores

Canais com os atores secundários:

Técnico de suporte: linha telefônica

Sensores: interfaces *hardwired* e de radiofrequência

Questões em aberto:

1. Existiria um modo de ativar o sistema sem o uso de uma senha ou com uma senha abreviada?
2. Deveria o painel de controle exibir outras mensagens de texto?
3. Quanto tempo o proprietário tem para introduzir a senha a partir do instante em que a primeira tecla é pressionada?
4. Existe alguma maneira de desativar o sistema antes de ser realmente ativado?

Casos de uso para outras interações do **proprietário** seriam desenvolvidos de maneira semelhante. É importante revisar cada caso com cuidado. Se algum elemento da interação for ambíguo, é provável que uma revisão do caso de uso indique um problema.

CASASEGURA



Desenvolvimento de um diagrama de caso de uso de alto nível

Cena: Sala de reuniões, onde prossegue a reunião para levantamento de requisitos.

Atores: Jamie Lazar, membro da equipe de software; Vinod Raman, membro da equipe de software; Ed Robbins, membro da equipe de software; Doug Miller, gerente da engenharia de software; três membros do Depto. de Marketing; um representante da Engenharia de Produto; e um facilitador.

Conversa:

Facilitador: Conversamos por um bom tempo sobre a funcionalidade de segurança domiciliar do *CasaSegura*. Durante o intervalo, esbocei um diagrama de caso de uso para sintetizar os cenários importantes que fazem parte desta função. Deem uma olhada.

(Todos os participantes observam a Figura 8.2.)

Jamie: Estou apenas começando a aprender a notação UML.¹⁴ Então, a função de segurança domiciliar é representada pelo retângulo grande com as elipses em seu interior? E as elipses representam os casos de uso que redigimos?

Facilitador: Isso. E as figuras de bonecos representam atores – as pessoas ou coisas que interagem com o sistema conforme descrito pelo caso de uso... Ah, eu uso o quadrado legendado para representar um ator que não é uma pessoa... Neste caso, sensores.

Doug: Isso é permitido na UML?

Facilitador: Permissão não é o problema. O ponto é comunicar a informação. Eu acho enganoso o uso de uma figura de boneco para representar um dispositivo. Portanto, adaptei um pouco as coisas. Não creio que isso vai criar problemas.

Vinod: Certo, temos narrativas de casos de uso para cada uma das elipses. Precisamos desenvolver narrativas baseadas nos modelos detalhados sobre os quais li a respeito?

Facilitador: Provavelmente, mas isso pode esperar até que tenhamos considerado outras funções do *CasaSegura*.

Representante do Depto. de Marketing: Espere um pouco. Fiquei observando este diagrama e de repente me dei conta de que deixamos passar algo.

Facilitador: Ah é? Diga o que deixamos passar.

(A reunião continua.)

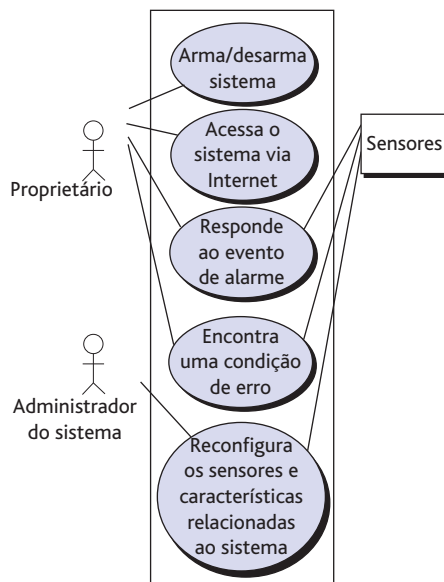


FIGURA 8.2 Diagrama de caso de uso em UML para a função de segurança domiciliar do *CasaSegura*.

¹⁴ Um resumo breve sobre a notação UML está no Apêndice 1.

FERRAMENTAS DO SOFTWARE

**Desenvolvimento de caso de uso**

Objetivo: Auxiliar no desenvolvimento de casos de uso por meio de modelos e mecanismos automatizados para avaliar a clareza e a consistência.

Mecanismos: O mecanismo das ferramentas é variado. Em geral, as ferramentas de caso de uso fornecem modelos em que se pode preencher os espaços em branco para criar casos de uso efetivos. A maior parte da funcionalidade de caso de uso está embutida em um conjunto de funções de engenharia de requisitos mais abrangentes.

Ferramentas representativas:¹⁵

A grande maioria das ferramentas de modelagem de análise baseadas em UML oferece recursos de texto e gráficos para o desenvolvimento e a modelagem de casos de uso.

Objects by Design (www.objectsbydesign.com/tools/uml-tools_byCompany.html) oferece um grande número de links para ferramentas desse tipo.

8.5 Construção do modelo de análise¹⁶

O objetivo do modelo de análise é fornecer uma descrição dos domínios informacional, funcional e comportamental necessários para um sistema baseado em computador. O modelo é modificado dinamicamente à medida que você aprende mais sobre o sistema a ser construído e outros envolvidos adquirem um melhor entendimento sobre aquilo que realmente querem. Por essa razão, o modelo de análise é uma reprodução dos requisitos em determinado momento. É esperado que ele mude.

À medida que o modelo de análise evoluir, certos elementos se tornarão relativamente estáveis, fornecendo uma sólida base para as tarefas posteriores do projeto. Entretanto, outros elementos do modelo podem ser mais voláteis, indicando que os envolvidos ainda não têm um entendimento completo dos requisitos do sistema. O modelo de análise e os métodos usados para construí-lo são apresentados em detalhe nos Capítulos 9 a 11. Apresentamos uma visão geral nas seções a seguir.

8.5.1 Elementos do modelo de análise

Há várias maneiras de examinar os requisitos para um sistema baseado em computador. Alguns profissionais de software argumentam que é melhor selecionar um modo de representação (por exemplo, o caso de uso) e aplicá-lo. Outros acreditam que vale a pena usar uma série de modos de representação para representar o modelo de análise. Modos de representação diferentes nos forçam a considerar os requisitos de diferentes pontos de vista – uma abordagem com maior probabilidade de revelar omissões, inconsistências e ambiguidades. Um conjunto de elementos genéricos é comum à maioria dos modelos de análise.

É sempre uma boa ideia fazer os envolvidos participarem ativamente. Uma das melhores formas para isso é pedir a cada envolvido para escrever casos de uso que descrevam como o software será utilizado.

¹⁵ As ferramentas indicadas não representam o endosso do autor; apenas uma amostra de ferramentas desta categoria. Os nomes das ferramentas são marcas registradas dos respectivos fornecedores.

¹⁶ No livro, os termos são sinônimos: modelo de análise e modelo de requisitos. Ambos referem-se à representação do domínio de informações, de funções e de comportamento que descrevem os requisitos do problema.

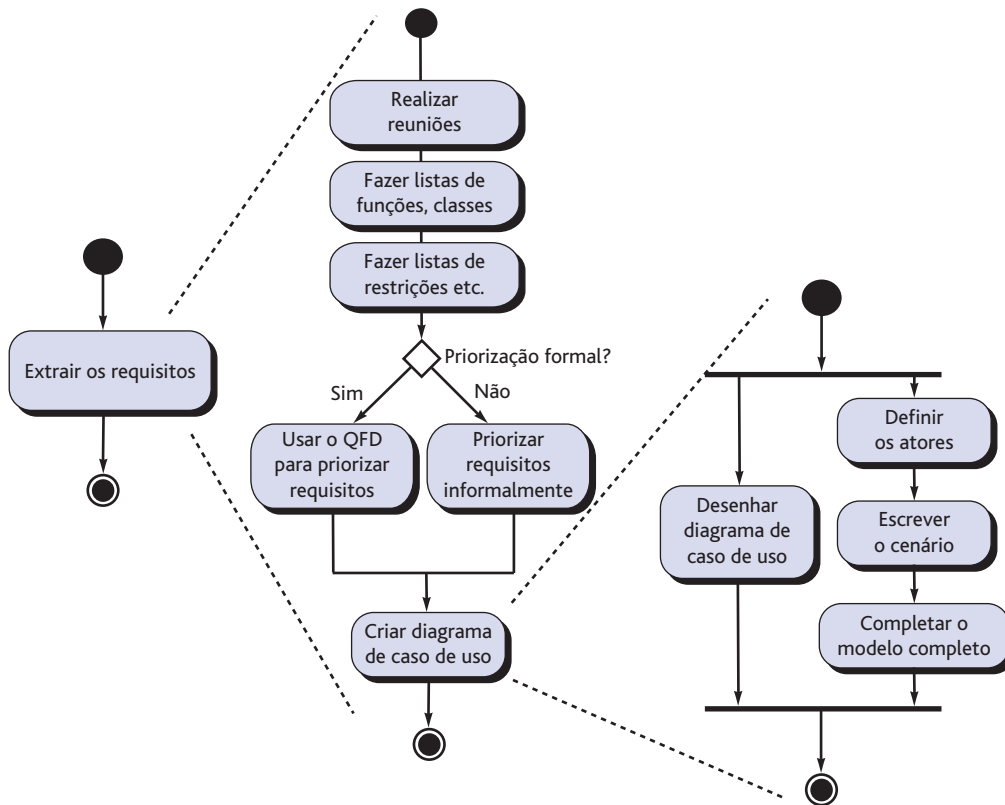


FIGURA 8.3 Diagramas de atividades UML para levantamento de requisitos.

Elementos baseados em cenários. O sistema é descrito sob o ponto de vista do usuário usando uma abordagem baseada em cenários. Por exemplo, casos de uso básicos (Seção 8.4) e seus diagramas (Figura 8.2) evoluem para casos de uso mais elaborados baseados em modelos. Elementos do modelo de requisitos baseados em cenários são, em geral, a primeira parte do modelo a ser desenvolvido. Como tal, servem como entrada para a criação de outros elementos de modelagem. A Figura 8.3 mostra um diagrama¹⁷ de atividades em UML para o levantamento de requisitos e representa-os utilizando casos de uso. São mostrados três níveis da elaboração, culminando em uma representação baseada em cenários.

Elementos baseados em classes. Cada cenário de uso implica um conjunto de objetos manipulados à medida que um ator interage com o sistema. Esses objetos são categorizados em classes – um conjunto de coisas que possuem atributos similares e comportamentos comuns. Por exemplo, um diagrama de classes UML pode ser utilizado para representar uma classe **Sensor** para a função de segurança do *CasaSegura* (Figura 8.4). Note que o diagrama enumera os atributos dos sensores (por exemplo, nome, tipo) e as operações (por exemplo, *identificar*, *habilitar*) que podem ser aplicadas para modificar tais atributos. Além dos diagramas de classes, outros elementos de modelagem de

Uma maneira de isolar classes é procurar substantivos descritivos em um texto de caso de uso. Pelo menos alguns dos substantivos serão candidatos a classes. Veja mais sobre isso no Capítulo 12.

¹⁷ Um breve tutorial sobre a UML é apresentado no Apêndice 1 para aqueles que não estão familiarizados com sua notação.

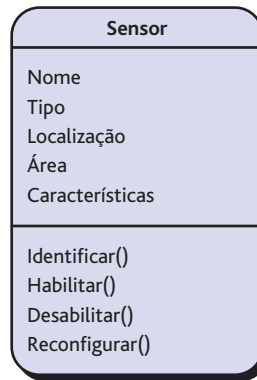


FIGURA 8.4 Diagrama de classes para sensor.

análise descrevem o modo pelo qual as classes colaboram entre si e os relacionamentos e interações entre as classes. Estes são discutidos de forma mais detalhada no Capítulo 10.

Elementos comportamentais. O comportamento de um sistema baseado em computadores pode ter um efeito profundo sobre o projeto escolhido e a abordagem de implementação aplicada. Portanto, o modelo de análise deve fornecer elementos de modelagem que descrevam comportamento.

O *diagrama de estados* é um método para representar o comportamento de um sistema por meio da representação de seus estados e dos eventos que fazem com que o sistema mude de estado. *Estado* é qualquer modo de comportamento observável. Além disso, o diagrama de estados indica as ações (por exemplo, ativação de processos) tomadas em decorrência de determinado evento.

Para ilustrar o uso de um diagrama de estados, considere o software embarcado no painel de controle do *CasaSegura* responsável pela leitura das entradas feitas pelos usuários. A Figura 8.5 mostra um diagrama de estados UML simplificado.

Além das representações comportamentais do sistema como um todo, o comportamento de classes individuais também pode ser modelado. Uma discussão mais aprofundada sobre modelagem comportamental é apresentada no Capítulo 11.

Um estado é um modo de comportamento observável externamente. Estímulos externos provocam transições entre estados.

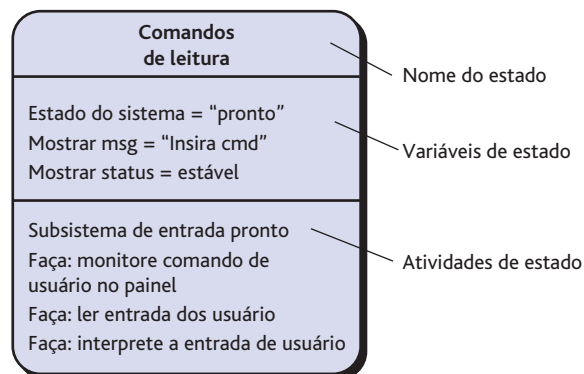


FIGURA 8.5 Notação de um diagrama de estados UML.

CASASEGURA

**Modelagem comportamental preliminar**

Cena: Sala de reuniões, onde prossegue a primeira reunião para levantamento de requisitos.

Atores: Jamie Lazar, membro da equipe de software; Vinod Raman, membro da equipe de software; Ed Robbins, membro da equipe de software; Doug Miller, gerente da engenharia de software; três membros do Depto. de Marketing; um representante da Engenharia de Produto; e um facilitador.

Conversa:

Facilitador: Estamos prestes a finalizar nossa discussão sobre a funcionalidade segurança domiciliar do *CasaSegura*. Antes de fazê-lo, gostaria de discutir o comportamento da função.

Representante do Depto. de Marketing: Não entendi o que você quis dizer com comportamento.

Ed (sorrindo): Trata-se de dar ao produto um "tempo de espera", caso ele se comporte mal.

Facilitador: Não exatamente. Permita-me explicar.

(O facilitador explica os fundamentos da modelagem comportamental à equipe de levantamento de requisitos.)

Representante do Depto. de Marketing: Isso me parece um tanto técnico. Não estou certo se serei de alguma ajuda aqui.

Facilitador: Certamente que você pode ajudar. Que comportamento você observa do ponto de vista de usuário?

Representante do Depto. de Marketing: Bem, o sistema estará *monitorando* os sensores. *Lendo* comandos do proprietário. *Mostrando* seu estado.

Facilitador: Viu, você pode ajudar.

Jamie: Ele também vai *indagar* o PC para determinar se há qualquer entrada proveniente dele, por exemplo, acesso baseado em Internet ou informações de configuração.

Vinod: Sim, de fato, configurar o sistema é um estado em si.

Doug: Pessoal, vocês estão enrolando. Pensem um pouco mais... Existe uma maneira de colocar esta coisa em um diagrama?

Facilitador: Existe, mas adiemos para logo depois da reunião.

8.5.2 Padrões de análise

Qualquer pessoa que tenha feito engenharia de requisitos em mais do que uns poucos projetos de software começa a perceber a recorrência de certos problemas em todos os projetos em um domínio de aplicação específico.¹⁸ Esses *padrões de análise* [Fow97] sugerem soluções (por exemplo, uma classe, função ou comportamento) no campo de aplicação que podem ser reutilizadas na modelagem de muitas aplicações.

Geyer-Schulz e Hahsler [Gey01] sugerem dois benefícios que podem estar associados ao uso de padrões de análise:

Primeiro, os padrões de análise aceleram o desenvolvimento de modelos de análise abstratos que capturam os principais requisitos do problema concreto, fornecendo modelos de análise reutilizáveis com exemplos, bem como uma descrição das vantagens e limitações. Em segundo lugar, os padrões de análise facilitam a transformação do modelo de análise em um modelo de projeto, sugerindo padrões de projeto e soluções confiáveis para problemas comuns.

Os padrões de análise são integrados ao modelo de análise por meio da referência ao nome do padrão. Eles também são armazenados em um

Se quiser obter soluções para requisitos de cliente mais rapidamente e fornecer abordagens comprovadas à sua equipe, use padrões de análise.

¹⁸ Em alguns casos, problemas são recorrentes independentemente do domínio do problema. Por exemplo, as características e funções usadas para o problema de interface do usuário ocorrem de modo similar em vários domínios de problemas.

repositório, de modo que os engenheiros de requisitos podem usar recursos de busca para encontrá-los e reutilizá-los. Informações sobre um padrão de análise (e outros tipos de padrões) são apresentadas em um modelo padrão [Gey01],¹⁹ discutido de maneira mais detalhada no Capítulo 16. Exemplos de padrões de análise e uma discussão mais ampla deste tópico são apresentados no Capítulo 11.

8.5.3 Engenharia de requisitos ágil

O objetivo da engenharia de requisitos ágil é transferir as ideias dos envolvidos para a equipe de software, em vez de criar artefatos de análise abrangentes. Em muitas situações, os requisitos não são predefinidos, mas surgem no início de cada iteração do produto. Quando a equipe ágil adquire um conhecimento de alto nível dos recursos críticos de um produto, as jornadas de uso (Capítulo 5) relevantes para o próximo incremento são refinadas. O processo ágil estimula a identificação e a implementação antecipadas dos recursos de prioridade mais alta do produto. Isso permite a criação e os testes antecipados de protótipos funcionais.

A engenharia de requisitos ágil trata de questões importantes, comuns em projetos de software: alta volatilidade dos requisitos, conhecimento incompleto da tecnologia de desenvolvimento e clientes incapazes de articular suas visões, até verem um protótipo em funcionamento. O processo ágil intercala atividades de engenharia e projeto de requisitos.

8.5.4 Requisitos de sistemas autoadaptativos

Os *sistemas autoadaptativos*²⁰ podem se reconfigurar, aumentar sua funcionalidade, proteger-se, recuperar-se de falhas e fazer tudo isso enquanto ocultam de seus usuários a maior parte de sua complexidade interna [Qur09]. Os requisitos adaptativos documentam a variabilidade necessária para os sistemas autoadaptativos. Isso significa que um requisito deve abranger a noção de variabilidade ou flexibilidade e, ao mesmo tempo, especificar um aspecto funcional ou qualitativo do produto de software. A variabilidade pode incluir incerteza de tempo, diferenças de perfil de usuário (por exemplo, usuários *versus* administradores de sistema), mudanças de comportamento baseadas no domínio do problema (por exemplo, comercial ou educacional) ou comportamentos predefinidos explorando bens do sistema.

A captura de requisitos adaptativos enfoca as mesmas questões utilizadas pela engenharia de requisitos de sistemas mais convencionais. Contudo, uma variabilidade significativa pode estar presente ao se responder a cada uma dessas questões. Quanto mais variáveis as respostas, mais complexo precisará ser o sistema resultante para acomodar os requisitos.

Quais são as características de um sistema autoadaptativo?

¹⁹ Uma variedade de templates de padrões foi proposta na literatura. Veja [Fow97], [Gam95], [Yac03] e [Bus07], entre muitas outras fontes.

²⁰ Um exemplo de sistema autoadaptativo é um aplicativo baseado em geolocalização: o seu comportamento muda em função da localização do dispositivo móvel na qual ele está embarcado.

8.6 Negociação de requisitos

Em um contexto de engenharia de requisitos ideal, as tarefas de início, levantamento e elaboração determinam os requisitos do cliente com detalhes suficientes para prosseguir nas atividades de engenharia de software subsequentes. Infelizmente, isso raramente acontece. Na realidade, talvez você tenha de iniciar uma *negociação* com um ou mais envolvidos. Na maioria dos casos, solicita-se aos envolvidos contrabalançar funcionalidade, desempenho e outras características do produto ou sistema em função do custo e tempo para chegar ao mercado. O intuito da negociação é desenvolver um plano de projeto que atenda às necessidades dos envolvidos e, ao mesmo tempo, reflita as restrições do mundo real (por exemplo, tempo, pessoal, orçamento) impostas à equipe de software.

As melhores negociações buscam ao máximo um resultado “ganha-ganha”.²¹ Ou seja, os envolvidos ganham obtendo um sistema ou produto que satisfaz a maioria de suas necessidades e você (como membro da equipe de software) ganha trabalhando com prazos de entrega e orçamentos reais e atingíveis.

Boehm [Boe98] define um conjunto de atividades de negociação no início de cada iteração do processo de software. Mais do que uma simples atividade de comunicação com o cliente, são definidas as seguintes atividades:

1. Identificação dos principais envolvidos no sistema ou subsistema.
2. Determinação das “condições de ganho” dos envolvidos.
3. Negociação das condições de ganho dos envolvidos para ajustar-se a um conjunto de condições de ganho mútuo para todos os envolvidos (inclusive a equipe de software).

O êxito na concretização dessas etapas atinge um resultado em que todos saem ganhando, critério-chave para prosseguir nas atividades de engenharia de software subsequentes.

“Acordo é a arte de dividir um bolo de tal forma que cada um acredite que ficou com a fatia maior.”

Ludwig Erhard

Um breve artigo sobre negociação para requisitos de software pode ser baixado de www.alexander-egyed.com/publications/Software_Requirements_Negotiation-Some_Lessons_Learned.html.



A arte da negociação

Aprender a negociar eficazmente pode ser bem útil para toda a sua vida técnica e pessoal. Vale muito considerar as seguintes diretrizes:

1. *Reconhecer que não se trata de uma competição.* Para se ter sucesso, ambas as partes têm de ter a sensação de que ganharam ou atingiram algum objetivo. Ambas terão de ceder.
2. *Planejar uma estratégia.* Decidir o que você quer obter, o que a outra parte quer obter e como você vai fazer para que ambas aconteçam.

INFORMAÇÕES

3. *Ouvir ativamente.* Não pense em sua resposta enquanto a outra parte estiver falando. Ouça-a. É provável que você tome conhecimento de algo que irá ajudá-lo a negociar melhor sua posição.
4. *Concentrar-se nos interesses da outra parte.* Caso queira evitar conflitos, não assuma posições rígidas.
5. *Não deixar a negociação ir para o lado pessoal.* Concentre-se no problema que precisa ser resolvido.
6. *Ser criativo.* Não tenha medo de inovar, caso se encontre em um impasse.
7. *Estar preparado para se comprometer.* Uma vez que se tenha chegado a um acordo, seja objetivo; comprometa-se e vá adiante.

²¹ Vários livros foram escritos sobre as habilidades de negociação (p. ex., [Fis11], [Lew09], [Rai06]). É uma das mais importantes habilidades que você pode desenvolver. Leia um desses livros.

Fricker [Fri10] e seus colegas sugerem substituir a transferência tradicional de especificações de requisitos para as equipes de software por um processo de comunicação bidirecional chamado *handshake* (aperto de mão). No handshake, a equipe de software propõe soluções para os requisitos, descreve seu impacto e comunica seus objetivos para representantes do cliente. Os representantes examinam as soluções propostas, concentrando-se nos recursos ausentes e buscando esclarecimento de requisitos novos. Os requisitos serão considerados *bons o suficiente* se os clientes aceitarem a solução proposta.

O handshake permite delegar requisitos detalhados para as equipes de software. As equipes precisam levantar os requisitos dos clientes (por exemplo, usuários do produto e especialistas na área), melhorando, com isso, a aceitação do produto. O handshake tende a melhorar a identificação, a análise e a escolha de variantes, além de promover uma negociação de ganho mútuo (ganha-ganha/*win-win*).

CASASEGURA



O início de uma negociação

Cena: Sala de Lisa Perez após a primeira reunião para levantamento de requisitos.

Atores: Doug Miller, gerente de engenharia de software, e Lisa Perez, gerente de marketing.

Conversa:

Lisa: Bem, ouvi dizer que a primeira reunião correu muito bem.

Doug: Na verdade, correu, sim. Você mandou bons representantes para a reunião... Eles realmente contribuíram.

Lisa (sorrindo): É, na verdade eles me contaram que se engajaram no processo e que não foi uma "atividade de torrar os miolos".

Doug (rindo): Tomarei cuidado para não usar jargão técnico na próxima vez que eu visitar... Veja, Lisa, acho que vamos ter problemas para entregar toda a funcionalidade para o sistema de segurança domiciliar nas datas que sua gerência está propondo. Eu sei, é prematuro, porém já andei fazendo um planejamento preliminar e...

Lisa (franzindo a testa): Temos de ter isso até aquela data, Doug. De que funcionalidade você está falando?

Doug: Presumo que consigamos ter a funcionalidade de segurança domiciliar completa até a data-limite, mas teremos de postergar o acesso via Internet para a segunda versão.

Lisa: Doug, é o acesso via Internet que dá todo o charme ao *CasaSegura*. Vamos criar toda a campanha de marketing em torno disso. Temos de ter esse acesso!

Doug: Entendo sua situação, realmente. O problema é que, para lhes dar o acesso via Internet, teremos de construir e ter funcionando um site totalmente seguro. Isso demanda tempo e pessoal. Também precisaremos desenvolver muita funcionalidade adicional para a primeira versão... Não acredito que possamos fazer isso com os recursos que temos.

Lisa (ainda franzindo a testa): Entendo, mas temos de descobrir uma maneira de ter tudo isso pronto. É crítico para as funções de segurança domiciliar e para outras funções também... Estas últimas poderão esperar até a próxima versão... Concordo com isso.

Lisa e Doug parecem ter chegado a um impasse, mas eles têm de negociar uma solução para o problema. Poderiam os dois "ganhar" neste caso? Fazendo o papel de mediador, o que você sugeriria?

8.7 Monitoramento de requisitos

Atualmente, o desenvolvimento incremental é comum. Isso significa que casos de uso evoluem, novos casos de teste são desenvolvidos para cada novo incremento do software e ocorre uma contínua integração do código-fonte ao longo de um projeto. O *monitoramento de requisitos* pode ser extremamente útil,

quando o desenvolvimento incremental é usado. Ele abrange cinco tarefas: (1) a *depuração distribuída* revela erros e determina suas causas, (2) a *verificação em tempo de execução* determina se o software atende à sua especificação, (3) a *validação em tempo de execução* estima se o software em evolução atende às metas do usuário, (4) o *monitoramento da atividade comercial* avalia se um sistema satisfaz as metas comerciais e (5) a *evolução e o projeto colaborativo* fornecem informações para os envolvidos à medida que o sistema evolui.

O desenvolvimento incremental implica a necessidade de validação incremental. O monitoramento de requisitos dá suporte à validação contínua por analisar modelos de meta do usuário em relação ao sistema em uso. Por exemplo, um sistema de monitoramento poderia avaliar continuamente a satisfação do usuário e usar feedback para guiar aprimoramentos incrementais [Rob10].

8.8 Validação de requisitos

À medida que os elementos do modelo de requisitos são criados, eles são examinados em termos de inconsistência, omissões e ambiguidade. Os requisitos representados pelo modelo são priorizados pelos envolvidos e agrupados em pacotes de requisitos que serão implementados como incrementos de software. Uma revisão do modelo de requisitos trata das seguintes questões:

- Todos os requisitos estão de acordo com os objetivos globais para o sistema ou produto?
- Todos os requisitos foram especificados no nível de abstração apropriado? Ou seja, algum dos requisitos fornece um nível de detalhe técnico inadequado no atual estágio?
- O requisito é realmente necessário ou representa um recurso adicional que talvez não seja essencial para o objetivo do sistema?
- Cada um dos requisitos é limitado e sem ambiguidade?
- Cada um dos requisitos possui atribuição? Ou seja, uma fonte (em geral, um indivíduo específico) é indicada para cada requisito?
- Algum dos requisitos conflita com outros requisitos?
- Cada um dos requisitos é atingível no ambiente técnico que vai abrigar o sistema ou produto?
- Cada um dos requisitos pode ser testado, uma vez implementado?
- O modelo de requisitos reflete, de forma apropriada, a informação, função e comportamento do sistema a ser construído?
- O modelo de requisitos foi “dividido” para expor progressivamente informações mais detalhadas sobre o sistema?
- Padrões de requisitos foram utilizados para simplificar o modelo de requisitos? Todos os padrões foram validados adequadamente? Todos os padrões estão de acordo com os requisitos do cliente?

Ao revisar os requisitos, que perguntas devo fazer?

Essas e outras perguntas devem ser levantadas e respondidas para garantir que o modelo de requisitos reflita de maneira precisa as necessidades do envolvido e forneça uma base sólida para o projeto.

8.9 Evite erros comuns

Buschmann [Bus10] descreve três erros relacionados que devem ser evitados quando uma equipe de software faz engenharia de requisitos. Ele os chama de: *recursite*, *flexibilitite* e *desempenhite*.

Recursite descreve a prática de trocar a cobertura funcional pela qualidade global do sistema. Em algumas organizações, há a tendência de equiparar a quantidade de funções entregues no menor tempo possível com a qualidade global do produto final. Em parte, isso é impulsionado por envolvidos comerciais que acham que mais é melhor. Também há a tendência de desenvolvedores de software que querem implementar as funções fáceis rapidamente, sem pensar em sua qualidade. A realidade é que uma das causas mais comuns de falha de projeto de software é a falta de qualidade operacional – *não* a ausência de funcionalidade. Para evitar essa armadilha, inicie uma discussão (com outros envolvidos) sobre as principais funções exigidas pelo sistema e certifique-se de que cada função entregue apresente todos os atributos de qualidade necessários.

Flexibilitite acontece quando engenheiros de software sobrecarregam o produto com recursos de adaptação e configuração. Sistemas excessivamente flexíveis são difíceis de configurar e apresentam desempenho operacional ruim. Isso pode ser um sintoma de abrangência mal definida do sistema. A causa-raiz, entretanto, pode ser desenvolvedores que usam a flexibilidade como desculpa para a incerteza. Em vez de tomar decisões de projeto rígidas no início, eles fornecem “ganchos” para permitir a adição de recursos não planejados. O resultado é um sistema “flexível” desnecessariamente complexo, mais difícil de testar e cujo gerenciamento é mais desafiador.

Desempenhite ocorre quando os desenvolvedores de software se concentram demasiadamente no desempenho do sistema à custa de atributos de qualidade, como facilidade de manutenção, confiabilidade ou segurança. As características de desempenho do sistema devem ser determinadas como parte de uma avaliação dos requisitos não funcionais do software. O desempenho deve estar de acordo com a necessidade comercial de um produto e ser compatível com as outras características do sistema.

8.10 Resumo

As tarefas da engenharia de requisitos são conduzidas para estabelecer uma base sólida para o projeto e a construção. A engenharia de requisitos ocorre durante as atividades de comunicação com o cliente e de modelagem que são definidas para o processo genérico de software. Os membros da equipe de software conduzem sete funções de engenharia de requisitos – concepção, levantamento, elaboração, negociação, especificação, validação e gestão.

Na concepção do projeto, os envolvidos estabelecem os requisitos básicos do problema, definem restrições de projeto predominantes e abordam as principais características e funções que precisam estar presentes para que o sistema cumpra seus objetivos. Essas informações são refinadas e expandidas durante o levantamento – uma atividade de reunião de requisitos que faz uso

de reuniões com a participação de um facilitador, do QFD e do desenvolvimento de cenários de uso.

A elaboração expande ainda mais os requisitos em um modelo – um conjunto de elementos comportamentais orientados a fluxos e baseados em cenários, classes e atividades. O modelo pode fazer referência a padrões de análise, características do domínio do problema recorrentes em diferentes aplicações.

À medida que os requisitos são identificados e o modelo de análise é criado, a equipe de software e outros envolvidos no projeto negociam a prioridade, a disponibilidade e o custo relativo de cada requisito. O objetivo dessa negociação é desenvolver um plano de projeto realista. Além disso, todos os requisitos e o modelo de análise como um todo são validados em relação às necessidades do cliente para garantir que o sistema correto será construído.

Problemas e pontos a ponderar

8.1 Por que um número muito grande de desenvolvedores de software não dedica muita atenção à engenharia de requisitos? Existiria alguma circunstância em que poderíamos deixá-la de lado?

8.2 Você foi incumbido de extrair os requisitos de um cliente que lhe diz que está muito ocupado para poder atendê-lo. O que você deve fazer?

8.3 Discuta alguns dos problemas que ocorrem quando os requisitos têm de ser obtidos de três ou quatro clientes diferentes.

8.4 Por que dizemos que o modelo de análise representa uma reprodução de um sistema em determinado momento?

8.5 Suponhamos que você tenha convencido o cliente (você é um excelente vendedor) a concordar com todas as suas exigências como desenvolvedor. Isso o torna um mestre da negociação? Por quê?

8.6 Desenvolva pelo menos três “perguntas livres de contexto” que você faria a um envolvido durante a atividade de concepção.

8.7 Desenvolva um “kit” de levantamento de requisitos. O kit deve incluir um conjunto de diretrizes para realizar uma reunião para levantamento de requisitos e materiais que podem ser utilizados para facilitar a criação de listas e quaisquer outros itens que poderiam ajudar na definição dos requisitos.

8.8 Seu professor vai dividir a classe em grupos de quatro ou seis alunos. Metade do grupo vai desempenhar o papel do departamento de marketing e a outra fará o papel da engenharia de software. Sua tarefa é definir os requisitos para a função de segurança do *CasaSegura* descrita neste capítulo. Realize uma reunião para levantamento de requisitos usando as diretrizes apresentadas neste capítulo.

8.9 Desenvolva um caso de uso completo para uma das atividades a seguir:

- a. Fazer um saque em um caixa eletrônico.
- b. Usar seu cartão de débito para uma refeição em um restaurante.
- c. Comprar ações usando uma conta de corretagem online.
- d. Procurar livros (sobre um assunto específico) usando uma livraria online.
- e. Uma atividade especificada pelo seu professor.

8.10 O que representam as “exceções” nos casos de uso?

8.11 Escreva uma jornada de usuário para uma das atividades listadas na questão 8.9.

- 8.12 Considere o caso de uso criado na questão 8.9 e escreva um requisito não funcional para a aplicação.
- 8.13 Descreva com suas próprias palavras o que é um *padrão de análise*.
- 8.14 Usando o modelo apresentado na Seção 8.5.2, sugira um ou mais padrões de análise para os seguintes campos de aplicação:
- Software contábil.
 - Software de e-mail.
 - Navegadores para a Internet.
 - Software de processamento de texto.
 - Software para criação de sites.
 - Um campo de aplicação especificado pelo seu professor.
- 8.15 Qual o significado de *ganha-ganha* no contexto das negociações durante uma atividade de engenharia de requisitos?
- 8.16 O que você acha que acontece quando uma validação de requisitos revela um erro? Quem será envolvido na correção do erro?
- 8.17 Quais cinco tarefas compõem um programa de monitoramento de requisitos abrangente?

Leituras complementares e outras fontes de informação

Pelo fato de ser crucial para o sucesso na criação de qualquer sistema complexo baseado em computadores, a engenharia de requisitos é discutida em uma ampla variedade de livros. Chemuturi (*Requirements Engineering and Management for Software Development Projects*, Springer, 2013) apresenta importantes aspectos da engenharia de requisitos. Pohl e Rupp (*Requirements Engineering Fundamentals*, Rocky Nook, 2011) apresentam princípios e conceitos básicos, e Pohl (*Requirements Engineering*, Springer, 2010) oferece uma visão mais detalhada de todo o processo de engenharia de requisitos. Young (*The Requirements Engineering Handbook*, Artech House Publishers, 2003) apresenta uma discussão aprofundada das tarefas da engenharia de requisitos.

Beaty e Chen (*Visual Models for Software Products Best Practices*, Microsoft Press, 2012), Robertson (*Mastering the Requirements Process: Getting Requirements Right*, 3ª ed., Addison-Wesley, 2012), Hull e seus colegas (*Requirements Engineering*, 3ª ed., Springer-Verlag, 2010), Bray (*An Introduction to Requirements Engineering*, Addison-Wesley, 2002), Arlow (*Requirements Engineering*, Addison-Wesley, 2001), Gilb (*Requirements Engineering*, Addison-Wesley, 2000), Graham (*Requirements Engineering and Rapid Development*, Addison-Wesley, 1999) e Sommerville e Kotonya (*Requirement Engineering: Processes and Techniques*, Wiley, 1998) são apenas alguns exemplos de muitos livros dedicados ao assunto. Wiegers (*More About Software Requirements*, Microsoft Press, 2010) aborda várias técnicas práticas para o gerenciamento e o levantamento de requisitos.

Uma visão da engenharia de requisitos baseada em padrões é descrita por Withall (*Software Requirement Patterns*, Microsoft Press, 2007). Ploesch (*Contracts, Scenarios and Prototypes*, Springer-Verlag, 2004) discute técnicas avançadas para desenvolvimento de requisitos de software. Windle e Abreo (*Software Requirements Using the Unified Process*, Prentice Hall, 2002) discutem a engenharia de requisitos no contexto do Processo Unificado e da notação da UML. Alexander e Steven (*Writing Better Requirements*, Addison-Wesley, 2002) fornecem um breve conjunto de diretrizes para redação clara dos requisitos, representando-os como cenários e revisando o resultado final.

A modelagem de casos de uso muitas vezes impulsiona a criação de todos os demais aspectos do modelo de análise. O assunto é debatido exaustivamente por Rosenberg e Stephens (*Use Case Driven Object Modeling with UML: Theory and Practice*, Apress, 2007), Denny (*Succeeding with Use Cases: Working Smart to Deliver Quality*, Addison-Wesley, 2005), Alexander e Maiden (eds.) (*Scenarios, Stories, Use Cases: Through the Systems Development Life-Cycle*, Wiley, 2004), Leffingwell e seus colegas (*Managing Software Requirements: A Use Case Approach*, 2ª ed., Addison-Wesley, 2003) apresentam um proveitoso conjunto das melhores práticas para levantamento de requisitos.

Uma discussão sobre requisitos ágeis pode ser encontrada em livros de Adzic (*Specification by Example: How Successful Teams Deliver the Right Software*, Manning Publications, 2011), Leffingwell (*Agile Requirements: Lean Requirements for Teams, Programs, and Enterprises*, Addison-Wesley, 2011), Cockburn (*Agile Software Development: The Cooperative Game*, 2ª ed., Addison-Wesley, 2006) e Cohn (*User Stories Applied: For Agile Software Development*, Addison-Wesley, 2004).

Uma ampla gama de fontes de informação sobre análise e engenharia de requisitos se encontra à disposição na Internet. Uma lista atualizada de referências relevantes (em inglês) para a prática da engenharia e análise de requisitos pode ser encontrada no site: www.mhhe.com/pressman.

9

Modelagem de requisitos: métodos baseados em cenários

Conceitos-chave

análise de domínio	169
análise de requisitos	167
casos de uso	173
casos de uso formais	177
diagrama de atividade	179
diagrama de raias	180
exceção de caso de uso	177
modelagem baseada em cenários	173
modelagem de requisitos	171
modelos UML	179

No nível técnico, a engenharia de software começa com uma série de tarefas de modelagem que conduzem à especificação dos requisitos e à representação do projeto para o software a ser construído. O modelo de requisitos¹ – na verdade, um conjunto de modelos – é a primeira representação técnica de um sistema.

Em um livro seminal sobre métodos de modelagem de requisitos, Tom DeMarco [DeM79] descreve o processo da seguinte maneira:

Reverendo os problemas e as falhas reconhecidos da fase de análise, sugiro que precisamos fazer os seguintes acréscimos ao nosso conjunto de metas para a fase de análise. Os produtos da análise devem apresentar grande facilidade em sua manutenção. Isso se aplica particularmente ao Documento-Alvo (especificação de

PANORAMA

sariamente a melhor maneira de representar os requisitos de um software. A modelagem de requisitos usa uma combinação das formas textual e diagramática para representar os requisitos de maneira relativamente fácil de entender e, mais importante ainda, simples, para fazer a revisão em termos de correção, completude e consistência.

Quem realiza? Um engenheiro de software (às vezes chamado de analista) constrói o modelo usando os requisitos extraídos do cliente.

Por que é importante? Para validar os requisitos do software, é preciso examiná-los segundo uma série de pontos de vista. Neste capítulo consideraremos a modelagem de requisitos sob a perspectiva baseada em cenários e examinaremos como a UML pode ser usada para complementar os cenários. Nos Capítulos 10 e 11, conheceremos outras “dimensões” do modelo de requisitos. Por examinar várias dimensões diferentes, aumenta-se a probabilidade de se encontrar erros, inconsistências vêm à tona e omissões são reveladas.

O que é? A palavra escrita é um maravilhoso veículo de comunicação; porém, não é neces-

Quais são as etapas envolvidas? A modelagem baseada em cenários representa o sistema sob o ponto de vista do usuário. Com a construção de um modelo baseado em cenários, é possível compreender melhor como o usuário interage com o software, revelando as principais funções e características exigidas pelos envolvidos.

Qual é o artefato? A modelagem baseada em cenários produz uma representação textual denominada “caso de uso”. O caso de uso descreve uma interação específica que pode ser de natureza informal (uma simples narrativa) ou mais estruturada e formal. O caso de uso pode ser complementado com vários diagramas UML diferentes, sobrepondo uma visão mais procedural da interação.

Como garantir que o trabalho foi realizado corretamente? Os artefatos do modelamento de requisitos devem ser revisados em termos de correção, completude e consistência. Devem refletir os requisitos de todos os envolvidos e estabelecer uma base a partir da qual o projeto pode ser conduzido.

¹ Nas edições anteriores deste livro, foi usado o termo *modelo de análise*, em vez de *modelo de requisitos*. Nesta edição, decidimos usar ambos os termos para representar a atividade de modelagem que define vários aspectos do problema a ser resolvido. *Análise* é a ação que ocorre à medida que requisitos são obtidos.

requisitos de software. Problemas de tamanho devem ser tratados por meio de um método efetivo de fracionamento. Uma especificação escrita como se fosse um romance vitoriano está acabada. Elementos gráficos têm de ser usados sempre que possível. Temos de diferenciar as considerações lógicas [essenciais] das físicas [implementação]... No mínimo, precisamos de... Algo que nos ajude a subdividir nossos requisitos e documentar essa subdivisão antes da especificação... Alguns meios de acompanhar e avaliar interfaces... Novas ferramentas para descrever lógica e política, algo melhor do que um texto narrativo.

Embora DeMarco tenha escrito sobre os atributos da modelagem de análise há mais de três décadas, seus comentários ainda se aplicam aos métodos e à notação da modelagem de requisitos modernos.

9.1 Análise de requisitos

A análise de requisitos resulta na especificação das características operacionais do software, indica a interface do software com outros elementos do sistema e estabelece restrições a que o software deve atender. Permite ainda que você (independentemente de ser chamado de *engenheiro de software*, *analista* ou *modelador*) amplie os requisitos básicos estabelecidos durante as tarefas de concepção, levantamento e negociação, que são parte da engenharia de requisitos (Capítulo 8).

A ação da modelagem de requisitos resulta em um ou mais dos seguintes tipos de modelos:

- *Modelos baseados em cenários* de requisitos do ponto de vista de vários “atores” do sistema.
- *Modelos orientados a classes* que representam classes orientadas a objetos (atributos e operações) e a maneira como as classes colaboram para atender aos requisitos do sistema.
- *Modelos comportamentais e baseados em padrões* que representam como o software se comporta em consequência de “eventos” externos.
- *Modelos de dados* que representam o domínio de informações para o problema.
- *Modelos orientados a fluxos* que representam os elementos funcionais do sistema e como eles transformam os dados à medida que se movem pelo sistema.

Esses modelos dão ao projetista de software informações que podem ser transformadas em projetos de arquitetura, de interfaces e de componentes. Por fim, o modelo de requisitos (e a especificação de requisitos de software) fornecem ao desenvolvedor e ao cliente os meios para verificar a qualidade assim que o software é construído.

Neste capítulo, vamos nos concentrarmos na *modelagem baseada em cenários* – uma técnica que está ficando cada vez mais popular em toda a comunidade de engenharia de software. Nos Capítulos 10 e 11 trataremos dos modelos baseados em classes e os modelos comportamentais. No decorrer da década passada, a modelagem de fluxos e de dados foram menos utilizadas,

“Qualquer ‘visão’ individual de requisitos é insuficiente para entender ou descrever o comportamento desejado de um sistema complexo.”

Alan M. Davis

O modelo de análise e a especificação de requisitos são um meio de avaliar a qualidade assim que o software for construído.

enquanto os métodos baseados em cenários e em classes, complementados com abordagens comportamentais e técnicas baseadas em padrões, tornaram-se mais populares.²

"Requisito não é sinônimo de arquitetura. Requisito não é sinônimo de projeto nem de interface do usuário. Requisito é sinônimo de necessidade."

**Andrew Hunt e
David Thomas**

O modelo de análise deve descrever o que o cliente quer e estabelecer uma base para o projeto, bem como definir uma meta para avaliação.

9.1.1 Filosofia e objetivos gerais

Na modelagem de requisitos, o foco principal está no *que* e não no *como*. Que interação com o usuário ocorre em dada circunstância, quais objetos o sistema manipula, que funções o sistema deve executar, quais comportamentos o sistema apresenta, que interfaces são definidas e quais restrições se aplicam?³

Em capítulos anteriores, citamos que a especificação de requisitos completa talvez não seja possível nesse estágio. O cliente pode estar inseguro daquilo que é precisamente necessário para certos aspectos do sistema. O desenvolvedor pode não estar seguro se determinada estratégia vai cumprir as funções e o desempenho esperados de modo adequado. Isso é mitigado em favor de uma abordagem iterativa para a análise e modelagem de requisitos. O analista deve modelar aquilo que é conhecido e usar o modelo como base para o projeto de incremento do software.⁴

O modelo de requisitos deve alcançar três objetivos principais: (1) descrever o que o cliente solicita, (2) estabelecer uma base para a criação de um projeto de software e (3) definir um conjunto de requisitos que possa ser validado assim que o software estiver construído. O modelo de análise preenche a lacuna entre uma descrição sistêmica que descreve o sistema como um todo ou a funcionalidade de negócio que é atingida aplicando-se software, hardware, dados, pessoal e outros elementos de sistema e um projeto de software (Capítulos 12 a 18) que descreve a arquitetura, a interface do usuário e a estrutura em termos de componentes do software. Essa relação está ilustrada na Figura 9.1.

É importante notar que todos os elementos do modelo de requisitos estão diretamente associados a partes do modelo do projeto. Nem sempre é possível estabelecer uma divisão clara das tarefas de análise e de projeto entre essas duas importantes atividades de modelagem. Invariavelmente, algum projeto ocorre como parte da análise, e alguma análise será realizada durante o projeto.

9.1.2 Regras práticas para a análise

Arlow e Neustadt [Arl02] sugerem algumas regras práticas que deveriam ser seguidas na criação do modelo de análise:

² Nossa apresentação de modelagem orientada a fluxos e de modelagem de dados foi omitida nesta edição. Contudo, muitas informações sobre esses métodos de modelagem de requisitos mais antigos podem ser encontradas na Web. Caso tenha interesse, faça uma pesquisa usando a frase "análise estruturada".

³ Deve-se notar que, à medida que os clientes se tornam tecnologicamente mais sofisticados, há uma tendência a especificar o *como* e também o *quê*. Entretanto, o enfoque principal deve permanecer no *quê*.

⁴ Como alternativa, a equipe de software poderia optar por criar um protótipo (Capítulo 4) em uma tentativa de entender melhor os requisitos do sistema.

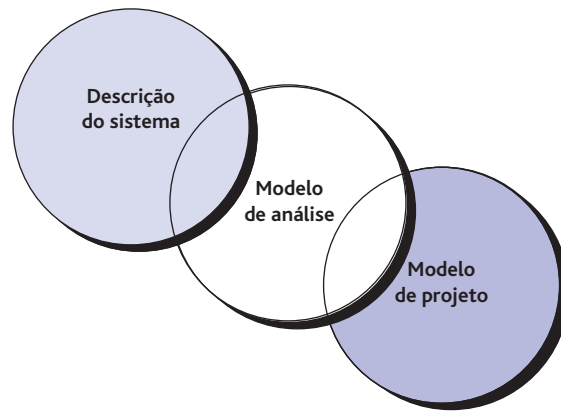


FIGURA 9.1 O modelo de requisitos como uma ponte entre a descrição do sistema e o modelo de projeto.

- *O modelo deve se concentrar nos requisitos visíveis no domínio do problema ou do negócio. O nível de abstração deve ser relativamente elevado. “Não se perca nos detalhes” [Arl02] que tentam explicar como o sistema vai funcionar.*
- *Cada elemento do modelo de requisitos deve contribuir para o entendimento geral dos requisitos de software e fornecer uma visão do domínio de informação, função e comportamento do sistema.*
- *Postergue considerações de infraestrutura e outros modelos não funcionais até a fase de projeto. Ou seja, talvez seja preciso um banco de dados, porém as classes necessárias para sua implementação, as funções necessárias para acessar o banco de dados e o comportamento que será apresentado à medida que ele for usado devem ser considerados apenas depois de a análise do domínio do problema ter sido concluída.*
- *Minimize o acoplamento do sistema. É importante representar as relações entre as classes e funções. Entretanto, se o nível de “interconexão” for extremamente alto, deve-se esforçar para reduzi-lo.*
- *Certifique-se de que o modelo de requisitos agrega valor para todos os envolvidos. Cada participante tem um uso próprio para o modelo. Por exemplo, os envolvidos no negócio devem usar o modelo para validar os requisitos; os projetistas devem usar o modelo como base para o projeto; o pessoal da Garantia da Qualidade (QA) deve usar o modelo para ajudar no planejamento de testes de aceitação.*
- *Mantenha o modelo o mais simples possível. Não adicione diagramas quando não acrescentam nenhuma informação nova. Não utilize formas de notação complexas quando uma lista simples já bastaria.*

Existem diretrizes básicas que possam nos guiar enquanto realizamos atividades de análise de requisitos?

“Problemas que valem a pena ser atacados demonstram seu valor contra-atacando.”

Piet Hein

9.1.3 Análise de domínio

Na discussão sobre engenharia de requisitos (Capítulo 8), destacamos que frequentemente ocorre a recorrência de padrões de análise em muitas aplicações em um domínio de aplicação específico. Se esses padrões são definidos e categorizados de maneira que permita seu reconhecimento e sua aplicação

Recursos úteis para análise de domínio e muitos outros tópicos podem ser encontrados em <http://www.sei.cmu.edu/>.

para resolver problemas comuns, a criação do modelo de análise deve ser acelerada. Mais importante ainda, a probabilidade de aplicação de padrões de projeto e de componentes de software executáveis cresce drasticamente. Isso melhora o tempo de colocação do produto no mercado e reduz os custos de desenvolvimento.

Como os padrões e as classes de análise são inicialmente reconhecidos? Quem os define, os classifica e os prepara para uso em projetos subsequentes? As respostas a essas questões caem na análise de domínio. Firesmith [Fir93] descreve análise de domínio da seguinte maneira:

A análise de domínio não examina uma aplicação específica, mas sim o domínio em que a aplicação reside. O intuito é identificar elementos comuns para solução de problemas que sejam aplicáveis a todas as aplicações no domínio.

Análise de domínio de um software é a identificação, a análise e a especificação de requisitos comuns de um campo de aplicação específico, tipicamente para reutilização em vários projetos dentro desse campo de aplicação... [A análise de domínio orientada a objetos é] a identificação, a análise e a especificação de capacidades comuns reutilizáveis dentro de um campo de aplicação específico, em termos de objetos, classes, componentes e frameworks comuns.

O “domínio de aplicação específico” pode ir da aviãoica a sistemas bancários, de games multimídia a software embarcado em equipamentos médicos. O objetivo da análise de domínio é simples: encontrar ou criar as classes de análise e/ou padrões de análise amplamente aplicáveis de modo que possam ser reutilizados.⁵

Usando a terminologia apresentada anteriormente neste livro, a análise de domínio pode ser vista como uma atividade universal para o processo de software. Queremos dizer com isso que a análise de domínio é uma atividade contínua da engenharia de software que não está ligada a nenhum projeto de software específico. De certo modo, o papel de um analista de domínio é semelhante ao papel de um mestre-ferramenteiro em um ambiente de manufatura pesada. O trabalho do mestre-ferramenteiro é projetar e construir ferramentas que possam ser usadas por muitas pessoas que fazem trabalhos semelhantes, mas não necessariamente iguais. O papel da análise de domínio⁶ é descobrir e definir padrões de análise, classes de análise e informações relacionadas que possam ser usados por várias pessoas que trabalham em aplicações similares, mas não necessariamente iguais.

A Figura 9.2 [Arn89] mostra entradas e saídas fundamentais para o processo de análise de domínio. São consultadas fontes de conhecimento do domínio para identificar objetos que possam ser reutilizados no domínio.

⁵ Uma visão complementar da análise de domínio “envolve a modelagem do domínio de forma que os engenheiros de software e outros envolvidos possam entender melhor sobre ela... Nem todas as classes de domínio resultam necessariamente no desenvolvimento de classes reutilizáveis...” [Let03a].

⁶ Não parta do pressuposto de que, se um analista de domínio está envolvido no trabalho, um engenheiro de software não precisa entender o domínio de aplicação. Todos os membros da equipe de software precisam ter algum conhecimento do domínio em que o software será inserido.

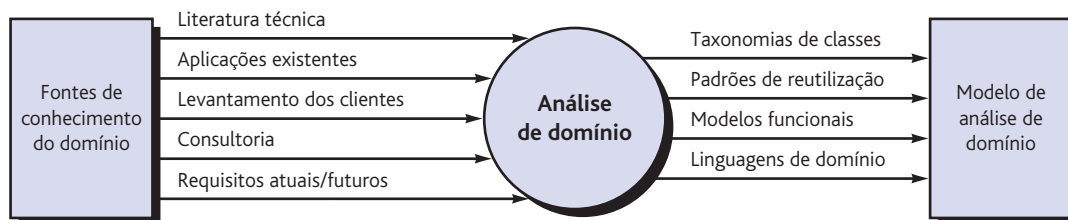


FIGURA 9.2 Entrada e saída da análise de domínio.

CASASEGURA

**Análise de domínio**

Cena: Sala de Doug Miller, após uma reunião com o pessoal de marketing.

Atores: Doug Miller, gerente de engenharia de software e Vinod Raman, membro da equipe de engenharia de software.

Conversa:

Doug: Preciso de você para um projeto especial, Vinod. Vou ter de tirá-lo das reuniões de levantamento de requisitos.

Vinod (com cara de contrariado): Que pena. Esse formato funciona... estava conseguindo algo com ele. O que houve?

Doug: Jamie e Ed lhe substituirão. De qualquer forma, o Marketing insiste que liberemos a capacidade de acesso à Internet com a função de segurança domiciliar na primeira versão do CasaSegura. Estamos com a corda no pescoço nessa... não temos tempo ou gente suficiente; portanto, temos de resolver os dois problemas – a interface PC e a interface Web – de uma só vez.

Vinod (parecendo confuso): Não sabia que o plano havia sido estabelecido... nós ainda nem terminamos de fazer o levantamento de requisitos.

Doug (com um sorriso amarelo): Eu sei, mas os prazos são tão apertados que decidi começar a traçar uma estratégia com o Marketing agora mesmo... de qualquer maneira, revisaremos qualquer plano provisório assim que tivermos todas as informações obtidas nas reuniões de levantamento de requisitos.

Vinod: OK, o que houve? O que você quer que eu faça?

Doug: Você sabe o que é “análise de domínio”?

Vinod: Mais ou menos. Primeiramente, devo procurar padrões similares nas aplicações que fazem os mesmos tipos de coisas que a que estou construindo. Se possível, você “rouba” então os padrões e os reutiliza em seu trabalho.

Doug: Não sei se gosto da palavra *roubar*, mas basicamente você entendeu a questão. O que eu gostaria que você fizesse é que começasse a pesquisar interfaces de usuário existentes de sistemas que controlam algo como o CasaSegura. Quero que você proponha um conjunto de padrões e classes de análise que possam ser comuns tanto para a interface baseada em PCs que ficarão instalados nos domicílios quanto para a interface baseada em navegadores que pode ser acessada via Internet.

Vinod: Podemos poupar tempo fazendo com que eles sejam os mesmos... por que não fazemos simplesmente isso?

Doug: Ah... É bom ter pessoas que pensam como você. Esse é o ponto – podemos poupar tempo e esforço se ambas as interfaces forem praticamente idênticas, implementadas com o mesmo código, blá, blá, blá, em que o Marketing tanto insiste.

Vinod: Então, você quer o quê? Classes, padrões de análise, padrões de projeto?

Doug: Todos eles. Nada formal neste momento. Quero apenas ter alguma vantagem inicial em nossa análise interna e trabalho de projeto.

Vinod: Pesquisarei nossa biblioteca de classes e verei o que conseguimos. Também usarei um modelo de padrões que vi em um livro alguns meses atrás.

Doug: Ótimo. Mãos à obra!

9.1.4 Abordagens de modelagem de requisitos

Uma visão da modelagem de requisitos, chamada de *análise estruturada*, considera os dados e os processos que transformam os dados como entidades separadas. Os objetos de dados são modelados de maneira a definir seus atributos e relações. Processos que manipulam objetos de dados são modelados para mostrar como transformam os dados à medida que objetos de dados fluem através do sistema.

"Análise é frustrante, cheia de complexos relacionamentos interpessoais, indefinida e difícil. Resumindo: é fascinante. Uma vez fisgado por ela, os antigos e fáceis prazeres da construção de um sistema jamais serão suficientes para satisfazê-lo novamente."

Tom DeMarco

Quais pontos de vista diferentes podem ser usados para descrever o modelo de requisitos?

Outra abordagem de modelagem de análise, denominada *análise orientada a objetos*, concentra-se na definição de classes e na maneira como colaboram entre si para atender aos requisitos dos clientes. A UML e o Processo Unificado (Capítulo 4) são predominantemente orientados a objetos.

Nesta edição do livro, optamos por enfatizar os elementos da análise orientada a objetos conforme é modelada usando-se UML. Nosso objetivo é sugerir uma combinação de representações que forneça aos envolvidos o melhor modelo de requisitos de software e a ligação mais eficaz para o projeto de software.

Cada elemento do modelo de requisitos (Figura 9.3) apresenta o problema segundo um ponto de vista. Os elementos baseados em cenários representam como o usuário interage com o sistema e a sequência específica de atividades que ocorrem à medida que o software é utilizado. Os elementos baseados em classes modelam os objetos que o sistema vai manipular, as operações que serão aplicadas aos objetos para efetuar a manipulação, as relações (algumas hierárquicas) entre os objetos e as colaborações que ocorrem entre as classes definidas. Os elementos comportamentais representam como eventos externos mudam o estado do sistema ou as classes que nele residem. Por fim, elementos orientados a fluxos representam o sistema como uma transformação de informações, indicando como os objetos de dados são transformados à medida que fluem pelas várias funções do sistema.

A modelagem de análise leva à obtenção de um ou mais desses elementos. Entretanto, o conteúdo específico de cada elemento (os diagramas usados para construir o elemento e o modelo) pode diferir de projeto para projeto. Como já citado várias vezes neste livro, a equipe de software tem de se esforçar para mantê-lo o mais simples possível. Devem ser utilizados apenas os elementos da modelagem que agregam valor ao modelo.

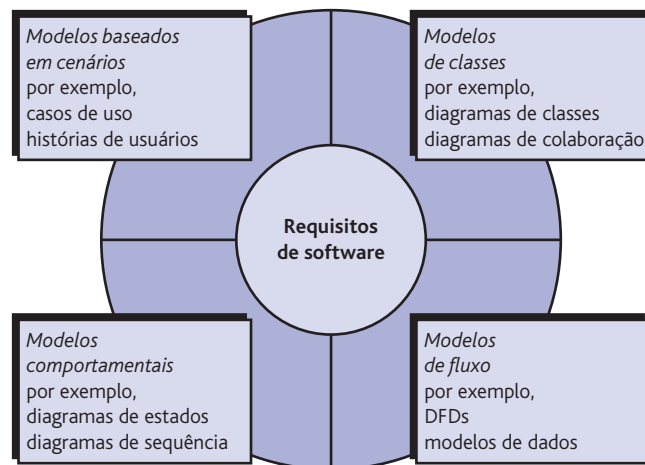


FIGURA 9.3 Elementos do modelo de análise.

9.2 Modelagem baseada em cenários

Embora haja muitas maneiras de medir o sucesso de um sistema ou produto baseado em computador, a satisfação do usuário está no topo da lista. Se você entender como os usuários (e outros atores) querem interagir com um sistema, sua equipe de software estará mais capacitada a caracterizar, de maneira apropriada, os requisitos e a construir modelos de análise e projeto proveitosos. Portanto, a modelagem de requisitos com UML⁷ começa com a criação de cenários na forma de casos de uso, diagramas de atividades e diagramas de raias.

9.2.1 Criação de um caso de uso preliminar

Alistair Cockburn caracteriza um caso de uso como um “contrato de comportamento” [Coc01b]. Como discutido no Capítulo 8, o “contrato” define a maneira como um ator⁸ usa um sistema baseado em computadores para atingir alguma meta. Basicamente, um caso de uso captura as interações que ocorrem entre produtores e consumidores de informação e o sistema em si. Nesta seção, examinaremos como os casos de uso são desenvolvidos como parte da atividade da modelagem de análise.⁹

No Capítulo 8, observamos que um caso de uso descreve um cenário de uso específico em uma linguagem simples sob o ponto de vista de um ator definido. Mas como sabemos (1) sobre o que escrever, (2) quanto escrever a respeito, (3) com que nível de detalhamento fazer uma descrição e (4) como organizar a descrição? Essas são as questões que devem ser respondidas nas situações em que os casos de uso devem ser valorizados como uma ferramenta da modelagem de requisitos.

Sobre o que escrever? As duas primeiras tarefas da engenharia de requisitos – concepção e levantamento – fornecem as informações necessárias para começarmos a escrever casos de uso. As reuniões para levantamento de requisitos, a disponibilização da função de qualidade (QFD) e outros mecanismos de engenharia de requisitos são utilizados para identificar os envolvidos, definir o escopo do problema, especificar as metas operacionais globais, estabelecer as prioridades, descrever todos os requisitos funcionais conhecidos e descrever os itens (objetos) manipulados pelo sistema.

Para começar a desenvolver um conjunto de casos de uso, enumere as funções ou atividades realizadas por um ator específico. Podemos obtê-las de uma lista de funções dos requisitos do sistema, por meio de conversas com envolvidos ou por meio de uma avaliação de diagramas de atividades (Seção 9.3.1) desenvolvidas como parte da modelagem de análise.

“[Casos de uso] são apenas uma ferramenta para definir o que existe fora do sistema (atores) e o que deve ser realizado pelo sistema (casos de uso).”

Ivar Jacobson

Em algumas situações, os casos de uso se tornam o mecanismo dominante da engenharia de requisitos. Entretanto, isso não significa que devemos descartar outros métodos de modelagem, quando forem apropriados.

⁷ A UML será usada como notação para modelagem ao longo deste livro. O Apêndice 1 apresenta um breve tutorial para aqueles que talvez não conheçam a notação básica da UML.

⁸ Ator não é uma pessoa específica, mas sim um papel que uma pessoa (ou dispositivo) desempenha em um contexto específico. Um ator “invoca o sistema para que este realize um de seus serviços” [Coc01b].

⁹ Os casos de uso são uma parte importante da modelagem de análise para as interfaces do usuário. O projeto e a análise de interfaces são discutidos em detalhes no Capítulo 15.



Desenvolvimento de outro cenário preliminar de usuário

Cena: Uma sala de reuniões, durante a segunda reunião para levantamento de requisitos.

Atores: Jamie Lazar, membro da equipe de software; Ed Robbins, membro da equipe de software; Doug Miller, gerente da engenharia de software; três membros do Depto. de Marketing; um representante da Engenharia de Produto; e um facilitador.

Conversa:

Facilitador: É hora de começarmos a falar sobre a função de vigilância do CasaSegura. Vamos desenvolver um cenário de usuário para acesso à função de vigilância.

Jamie: Quem desempenha o papel do ator nisso?

Facilitador: Acredito que a Meredith (uma pessoa do Marketing) venha trabalhando nessa funcionalidade. Por que você não desempenha o papel?

Meredith: Você quer fazer da mesma forma que fizemos da última vez, não é mesmo?

Facilitador: Correto... da mesma forma.

Meredith: Bem, obviamente a razão para a vigilância é permitir ao proprietário do imóvel verificar a casa enquanto ele se encontra fora, gravar e reproduzir imagens de vídeo que são capturadas... Esse tipo de coisa.

Ed: Usaremos compactação para armazenamento de vídeo?

Facilitador: Boa pergunta, Ed, mas vamos postergar essas questões de implementação por enquanto. Meredith?

Meredith: Certo, então basicamente há duas partes para a função de vigilância... A primeira configura o sistema, inclusive desenhando uma planta – precisamos de ferramentas para ajudar o proprietário do imóvel a fazer isso – e a segunda parte é a própria função de vigilância. Como o layout faz parte da atividade de configuração, vou me concentrar na função de vigilância.

Facilitador (sorrindo): Você tirou as palavras da minha boca.

Meredith: Hummm... quero ter acesso à função de vigilância via PC ou via Internet. Sinto que o acesso via Internet seria mais utilizado. De qualquer maneira, quero exibir visões de câmeras em um PC e controlar o deslocamento e a ampliação de imagens de determinada câmera. Especifico a câmera selecionando-a na planta da casa. Quero, de forma seletiva, gravar imagens geradas por câmeras e reproduzi-las. Também quero ser capaz de bloquear o acesso a uma ou mais câmeras com uma senha específica. Também quero a opção de ver pequenas janelas que mostrem visões de todas as câmeras e então escolher uma que deseje ampliar.

Jamie: Estas são chamadas de visões em miniatura.

Meredith: Certo, então eu quero visões em miniatura de todas as câmeras. Também quero que a interface para a função de vigilância tenha o mesmo aspecto de todas as demais do CasaSegura. Quero que ela seja intuitiva, significando que não vou precisar ler todo o manual para usá-la.

Facilitador: Bom trabalho. Agora, vamos nos aprofundar um pouco mais nessa função...

A função de vigilância domiciliar do CasaSegura (subsistema) discutida no quadro anterior identifica as seguintes funções (uma lista resumida) realizadas pelo ator **proprietário**:

- Selecionar câmera a ser vista.
- Solicitar imagens em miniatura de todas as câmeras.
- Exibir imagens das câmeras em uma janela de um PC.
- Controlar deslocamento e ampliação de uma câmera específica.
- Gravar, de forma seletiva, imagens geradas pelas câmeras.
- Reproduzir as imagens geradas pelas câmeras.
- Acessar a vigilância por câmeras via Internet.

À medida que as conversas com o envolvido (que desempenha o papel de proprietário de um imóvel) forem avançando, a equipe de levantamento de requisitos desenvolve casos de uso para cada uma das funções citadas. Em geral, os casos de uso são escritos primeiramente de forma narrativa informal. Caso seja necessária maior formalidade, o mesmo caso de uso é reescrito usando

um formato estruturado similar àquele proposto no Capítulo 8 e reproduzido mais adiante, nesta seção, na forma de um quadro.

Para fins de ilustração, consideremos a função *acessar a vigilância por câmeras via Internet – exibir visões das câmeras* (AVC-EVC). O envolvido que faz o papel do ator **proprietário** escreveria a seguinte narrativa:

Caso de uso: Acessar a vigilância por câmeras via Internet – exibir visões das câmeras (AVC-EVC)

Ator: proprietário

Se eu estiver em um local distante, posso usar qualquer PC com navegador apropriado para entrar no site *Produtos do CasaSegura*. Introduzo meu ID de usuário e dois níveis de senhas e, depois de validado, tenho acesso a toda funcionalidade para o meu sistema *CasaSegura* instalado. Para acessar a visão de câmera específica, seleciono “vigilância” nos botões das principais funções mostradas. Em seguida, seleciono “escolha uma câmera”, e a planta da casa é mostrada. Depois, seleciono a câmera em que estou interessado. Como alternativa, posso ver, simultaneamente, imagens em miniatura de todas as câmeras, selecionando “todas as câmeras” como opção de visualização. Depois de escolher uma câmera, seleciono “visualização”, e uma visualização com um quadro por segundo aparece em uma janela de visualização identificada pelo ID de câmera. Se quiser trocar de câmera, seleciono “escolha uma câmera”, e a janela de visualização original desaparece e a planta da casa é mostrada novamente. Em seguida, seleciono a câmera em que estou interessado. Surge uma nova janela de visualização.

Uma variação de um caso de uso narrativo apresenta a interação na forma de uma sequência ordenada de ações de usuário. Cada ação é representada como uma sentença declarativa. Voltando à função AVC-EVC, poderíamos escrever:

Caso de uso: Acessar a vigilância por câmeras via Internet – exibir visões das câmeras (AVC-EVC)

Ator: proprietário

1. O proprietário do imóvel faz o login no site *Produtos do CasaSegura*.
2. O proprietário do imóvel introduz seu ID de usuário.
3. O proprietário do imóvel introduz duas senhas (cada uma com pelo menos oito caracteres).
4. O sistema mostra os botões de todas as principais funções.
5. O proprietário seleciona a “vigilância” por meio dos botões das funções principais.
6. O proprietário seleciona “escolher uma câmera”.
7. O sistema mostra a planta da casa.
8. O proprietário seleciona um ícone de câmera da planta da casa.
9. O proprietário seleciona o botão “visualização”.
10. O sistema mostra uma janela de visualização identificada pelo ID de câmera.
11. O sistema mostra imagens de vídeo na janela de visualização a uma velocidade de um quadro por segundo.

“Os casos de uso podem ser utilizados em vários processos [de software]. Nosso processo favorito é o iterativo e dirigido por riscos.”

**Geri Schneider e
Jason Winters**

É importante notar que essa apresentação sequencial não leva em consideração quaisquer interações alternativas (a narrativa flui de forma mais natural e representa um número pequeno de alternativas). Casos de uso desse tipo são algumas vezes conhecidos como *cenários primários* [Sch98a].

9.2.2 Refinamento de um caso de uso preliminar

A descrição de interações alternativas é essencial para um completo entendimento da função a ser descrita por um caso de uso. Portanto, cada etapa no cenário primário é avaliada fazendo-se as seguintes perguntas [Sch98a]:

Como examinar sequências de ações alternativas ao desenvolver um caso de uso?

- *O ator pode fazer algo diferente neste ponto?*
- *Existe a possibilidade de o ator encontrar alguma condição de erro neste ponto?* Em caso positivo, qual seria?
- *Existe a possibilidade de o ator encontrar algum outro tipo de comportamento neste ponto (por exemplo, comportamento que é acionado por algum evento fora do controle do ator)?* Em caso positivo, qual seria?

As respostas a essas perguntas levam à criação de um conjunto de *cenários secundários* que fazem parte do caso de uso original, mas representam comportamento alternativo. Consideremos, por exemplo, as etapas 6 e 7 do cenário primário apresentado anteriormente:

6. O proprietário seleciona “escolher uma câmera”.
7. O sistema mostra a planta da casa.

O ator pode fazer algo diferente neste ponto? A resposta é “sim”. Referindo-se à narrativa que flui naturalmente, o ator poderia optar por ver, simultaneamente, imagens em miniatura de todas as câmeras. Portanto, um cenário secundário poderia ser “Visualizar imagens em miniatura para todas as câmeras”.

Existe a possibilidade de o ator encontrar alguma condição de erro neste ponto? Qualquer número de condições de erro pode ocorrer enquanto um sistema baseado em computadores opera. Nesse contexto, consideramos condições de erro apenas aquelas que provavelmente são resultado direto da ação nas etapas 6 ou 7. Novamente, a resposta à pergunta é “sim”. Uma planta com ícones de câmera talvez jamais tenha sido configurada. Portanto, selecionar “escolher uma câmera” resulta em uma condição de erro: “Não há nenhuma planta configurada para este imóvel”.¹⁰ Essa condição de erro se torna um cenário secundário.

Existe a possibilidade de o ator encontrar algum outro tipo de comportamento neste ponto? Novamente, a resposta à pergunta é “sim”. Enquanto as etapas 6 e 7 ocorrem, o sistema pode encontrar uma condição de alarme. Isso resultaria no sistema exibindo uma notificação de alerta especial (tipo, local, ação do sistema) e dando ao ator uma série de opções relevantes à natureza

¹⁰ Nesse caso, outro ator, o **administrador do sistema**, teria de configurar a planta da casa, instalar e inicializar (por exemplo, atribuir um ID de equipamento) todas as câmeras e testar cada uma delas para ter certeza de que elas podem ser acessadas pelo sistema e pela planta da casa.

do alerta. Como o cenário secundário pode ocorrer a qualquer momento para praticamente todas as interações, ele não fará parte do caso de uso **AVC-EVC**. Em vez disso, seria desenvolvido um caso de uso distinto – **Condição de alarme encontrada** – e referido a partir de outros casos de uso, conforme a necessidade.

Essas situações descritas são caracterizadas como exceções do caso de uso. Uma *exceção* descreve uma situação (seja ela uma condição de falha ou uma alternativa escolhida pelo ator) que faz com que o sistema exiba um comportamento um tanto diferente.

Cockburn [Coc01b] recomenda uma sessão de *brainstorming* para obter um conjunto de exceções relativamente completo para cada caso de uso. Além das três perguntas genéricas sugeridas anteriormente, as questões a seguir também devem ser exploradas:

- *Existem casos em que ocorre alguma “função de validação” durante esse caso de uso?* Isso implica que a função de validação é chamada e poderia ocorrer uma condição de erro.
- *Existem casos em que uma função de suporte (ou ator) parará de responder apropriadamente?* Por exemplo, uma ação de usuário aguarda uma resposta, porém a função que deve responder entra em condição de *time-out*.
- *Existe a possibilidade de um fraco desempenho do sistema resultar em ações do usuário inesperadas ou impróprias?* Por exemplo, uma interface baseada na Web responde de forma muito lenta, fazendo com que um usuário selecione um botão de processamento várias vezes seguidas. Essas seleções entram em fila de forma inapropriada e, por fim, geram uma condição de erro.

A lista de extensões desenvolvidas a partir das perguntas e respostas deve ser “racionalizada” [Co01b] por meio dos seguintes critérios: deve ser indicada uma exceção no caso de uso se o software for capaz de detectar a condição descrita e, em seguida, tratar a condição assim que esta for detectada. Em alguns casos, uma exceção vai precipitar o desenvolvimento de outro caso de uso (para tratar da condição percebida).

9.2.3 Criação de um caso de uso formal

Às vezes, os casos de uso informais apresentados na Seção 9.2.1 são suficientes para a modelagem de requisitos. Entretanto, quando um caso de uso envolve uma atividade crítica ou descreve um conjunto complexo de etapas com um número significativo de exceções, uma abordagem mais formal talvez seja mais desejável.

O caso de uso **AVC-EVC** mostrado no quadro segue uma descrição geral típica para casos de uso formais. O *objetivo no contexto* identifica o escopo geral do caso de uso.

A *precondição* descreve aquilo que é conhecido como verdadeiro antes de o caso de uso ser iniciado. O *disparador* identifica o evento ou a condição que “faz com que o caso de uso seja iniciado” [Coc01b]. O *cenário* enumera as ações específicas que o ator deve executar e as respostas apropriadas do sis-

O que é uma exceção de caso de uso e como determino quais são as exceções prováveis?

tema. As *exceções* identificam as situações reveladas à medida que o caso de uso preliminar é refinado (Seção 9.2.2). Cabeçalhos adicionais poderão ou não ser acrescentados e são relativamente autoexplicativos.

CASASEGURA



Modelo de caso de uso para vigilância

Caso de uso: Acessar a vigilância por câmeras via Internet – exibir visões das câmeras (AVC-EVC)

Iteração: 2, última modificação: 14 de janeiro feita por V. Raman.

Ator primário: Proprietário.

Meta no contexto: Visualizar imagens de câmera espalhadas pela casa a partir de qualquer ponto remoto via Internet.

Precondições: O sistema deve estar totalmente configurado; devem ser obtidos ID de usuário e senhas apropriadas.

Disparador: O proprietário do imóvel decide fazer uma inspeção na casa enquanto se encontra fora.

Cenário:

1. O proprietário do imóvel faz o login no site *Produtos do CasaSegura*.
2. O proprietário do imóvel introduz seu ID de usuário.
3. O proprietário do imóvel introduz duas senhas (cada uma com pelo menos oito caracteres).
4. O sistema mostra os botões de todas as principais funções.
5. O proprietário seleciona a "vigilância" por meio dos botões das funções principais.
6. O proprietário seleciona "escolher uma câmera".
7. O sistema mostra a planta da casa.
8. O proprietário seleciona um ícone de câmera da planta da casa.
9. O proprietário seleciona o botão "visualização".
10. O sistema mostra uma janela de visualização identificada pelo ID de câmera.
11. O sistema mostra imagens de vídeo na janela de visualização a uma velocidade de um quadro por segundo.

Exceções:

1. O ID ou senhas são incorretos ou não foram reconhecidos – veja o caso de uso **Validar ID e senhas**.
2. A função de vigilância não está configurada para este sistema – o sistema mostra a mensagem de erro apropriada; veja o caso de uso **Configurar função de vigilância**.
3. O proprietário seleciona "Visualizar as imagens em miniatura para todas as câmeras" – veja o caso de uso **Visualizar as imagens em miniatura para todas as câmeras**.
4. A planta não está disponível ou não foi configurada – exibir a mensagem de erro apropriada e ver o caso de uso **Configurar planta da casa**.
5. É encontrada uma condição de alarme – veja o caso de uso **Condição de alarme encontrada**.

Prioridade: Prioridade moderada, a ser implementada após as funções básicas.

Quando disponível: Terceiro incremento.

Frequência de uso: Rara.

Canal com o ator: Via navegador instalado em PC e conexão de Internet.

Atores secundários: Administrador do sistema, câmeras.

Canais com os atores secundários:

1. Administrador do sistema: sistema baseado em PCs.
2. Câmeras: conectividade sem fio.

Questões em aberto:

1. Quais mecanismos protegem o uso não autorizado deste recurso por parte de funcionários do *Produtos do CasaSegura*?
2. A segurança é suficiente? Acessar este recurso de forma não autorizada representaria uma invasão de privacidade importante.
3. A resposta do sistema via Internet seria aceitável dada a largura de banda exigida para visualizações de câmeras?
4. Vamos desenvolver um recurso para fornecer vídeo a uma velocidade de quadros por segundo maior quando as conexões de banda larga estiverem disponíveis?

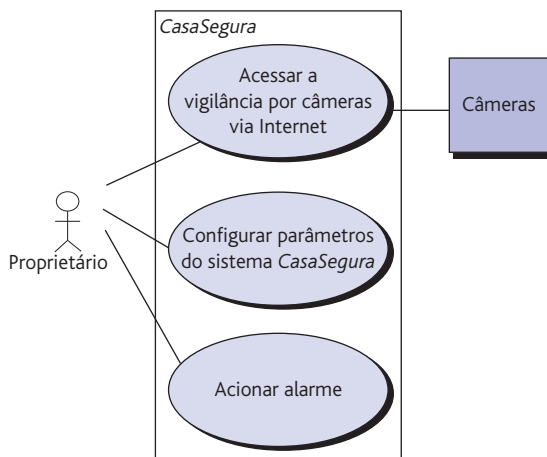


FIGURA 9.4 Diagrama de caso de uso preliminar para o sistema *CasaSegura*.

Em muitos casos, não há necessidade de criar uma representação gráfica de um cenário de uso, mas a representação diagramática pode facilitar a compreensão, particularmente quando o cenário é complexo. Conforme já citado neste livro, a UML oferece recursos de diagramação de casos de uso. A Figura 9.4 representa um diagrama de caso de uso preliminar para o produto *CasaSegura*. Cada caso de uso é representado por uma elipse. Nesta seção foi discutido apenas o caso de uso **AVC-EVC**.

Toda notação de modelagem tem suas limitações, e o caso de uso não é uma exceção. Assim como qualquer outra forma de descrição escrita, a qualidade de um caso de uso depende de seu(s) autor(es). Se a descrição não for clara, o caso de uso pode ser enganoso ou ambíguo. Um caso de uso que se concentra nos requisitos funcionais e comportamentais geralmente é inadequado para requisitos não funcionais. Para situações em que o modelo de análise deve ter muitos detalhes e precisão (por exemplo, sistemas com segurança crítica), um caso de uso talvez não seja suficiente.

Entretanto, a modelagem baseada em cenários é apropriada para a grande maioria de todas as situações com as quais você vai se deparar como engenheiro de software. Se desenvolvido apropriadamente, o caso de uso pode trazer grandes benefícios como ferramenta de modelagem.

Quando termina a atividade de criação de casos de uso? Para uma proveitosa discussão sobre esse assunto, consulte ootips.org/use-cases-done.html.

9.3 Modelos UML que complementam o caso de uso

Há muitas situações de modelagem de requisitos em que um modelo baseado em texto – mesmo um tão simples quanto um caso de uso – pode não fornecer informações de maneira clara e concisa. Nesses casos, pode-se optar por uma ampla variedade de modelos gráficos da UML.

9.3.1 Desenvolvimento de um diagrama de atividades

Um diagrama de atividades UML complementa o caso de uso por meio de uma representação gráfica do fluxo de interação em um cenário específico. Semelhante ao fluxograma, um diagrama de atividades usa retângulos com cantos

arredondados para representar determinada função do sistema, setas para representar o fluxo através do sistema, losangos de decisão para representar uma decisão com ramificação (cada seta saindo do losango é identificada) e linhas horizontais cheias indicam as atividades paralelas que estão ocorrendo. Um diagrama de atividades para o caso de uso **AVC-EVC** aparece na Figura 9.5. É importante observar que o diagrama de atividades acrescenta outros detalhes não mencionados (mas implícitos) pelo caso de uso. Por exemplo, um usuário poderia tentar introduzir **ID de usuário** e **senha** um número limitado de vezes. Isso é representado pelo losango de decisão abaixo de “Prompt para reintrodução de dados”.

9.3.2 Diagramas de raias

O *diagrama de raias* UML é uma variação útil do diagrama de atividades, permitindo representar o fluxo de atividades descrito pelo caso de uso e indicar qual ator (se existirem vários atores envolvidos em um caso de uso específico) ou classe de análise (Capítulo 10) tem a responsabilidade pela ação descrita

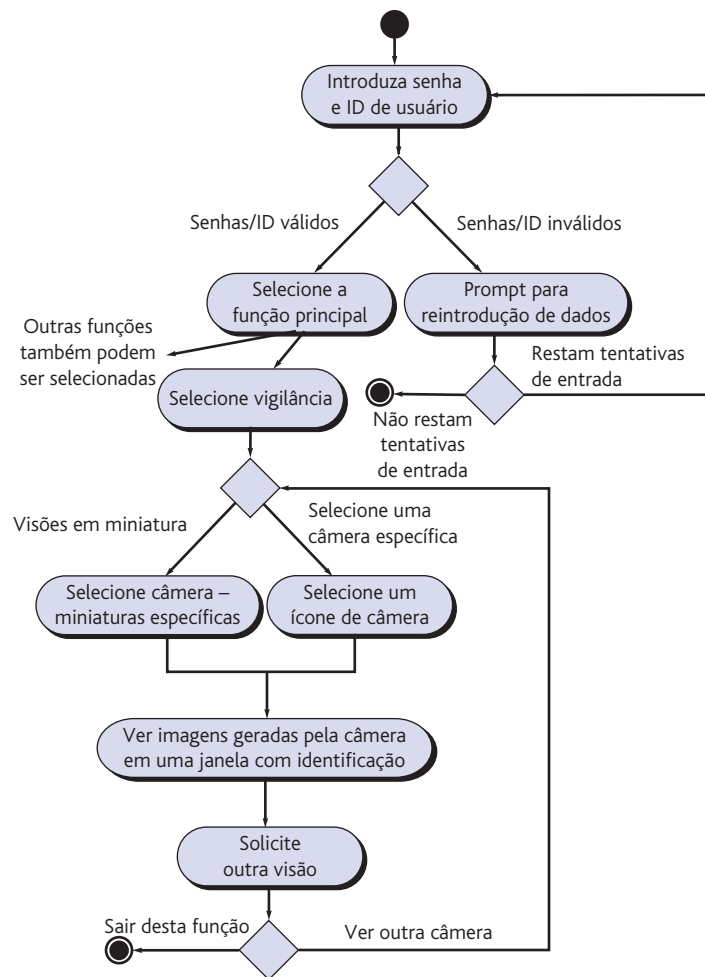


FIGURA 9.5 Diagrama de atividades para a função “Acessar a vigilância por câmeras via Internet – exibir visões das câmeras”.

por um retângulo de atividade. As responsabilidades são representadas como segmentos paralelos que dividem o diagrama verticalmente, como as raia de uma piscina.

Três classes de análise – **Proprietário**, **Câmera** e **Interface** – têm responsabilidades diretas ou indiretas no diagrama de atividades representado na Figura 9.5. Na Figura 9.6, o diagrama de atividades é rearranjado de forma que as atividades associadas a determinada classe de análise caiam na raia da referida classe. Por exemplo, a classe **Interface** representa a interface do usuário conforme vista pelo proprietário. O diagrama de atividades indica dois prompts que são a responsabilidade da interface – “prompt para reintrodução de dados” e “prompt para outra visão”. Esses prompts e as decisões associadas a eles caem na raia da **Interface**. Entretanto, as setas saem dessa raia e voltam para a raia do **Proprietário**, onde ocorrem as ações do proprietário do imóvel.

Um diagrama de raia UML representa o fluxo de ações e decisões e indica quais atores realizam cada uma delas.

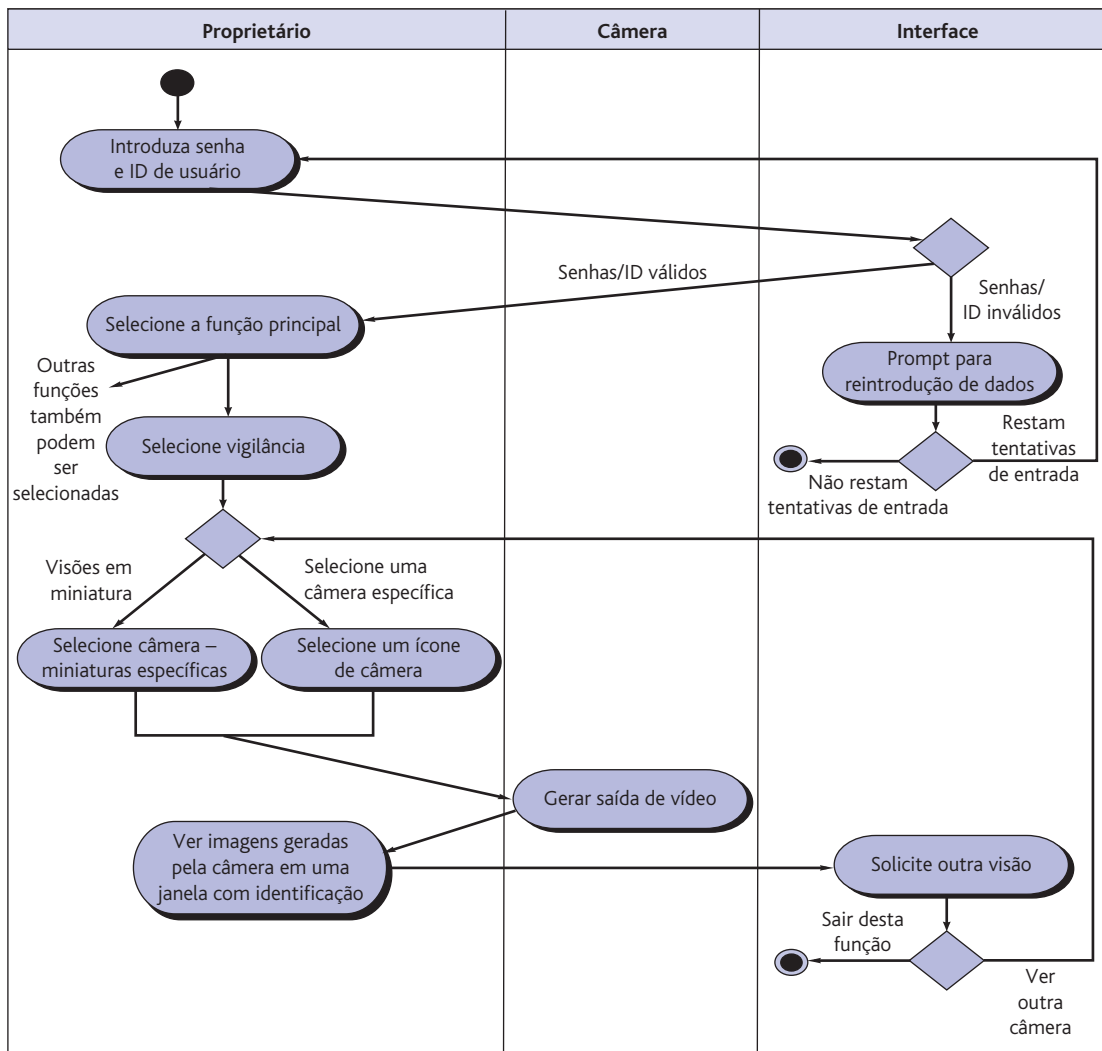


FIGURA 9.6 Diagrama de raia para a função “Acessar a vigilância por câmeras via Internet – exibir visões das câmeras”.

"Um bom modelo orienta nossas ideias, enquanto um ruim as distorce."

Brian Marick

Os casos de uso e os diagramas de atividades e de raia são orientados a procedimentos. Eles representam o modo como vários atores chamam funções específicas (ou outras etapas procedurais) para atender aos requisitos do sistema. Porém, uma visão procedural dos requisitos representa apenas uma dimensão de um sistema. Nos Capítulos 10 e 11, examinamos outras dimensões da modelagem de requisitos.

9.4 Resumo

O objetivo da modelagem de requisitos é criar várias representações que descrevam aquilo que o cliente exige; estabelecer uma base para a criação de um projeto de software; e definir um conjunto de requisitos que possam ser validados assim que o software for construído. O modelo de requisitos preenche a lacuna entre uma descrição sistêmica que define o sistema como um todo ou a funcionalidade de negócio e um projeto de software que descreve a arquitetura da aplicação de software, a interface do usuário e a estrutura em nível de componentes.

Os modelos baseados em cenários representam os requisitos de software sob o ponto de vista do usuário. O caso de uso – uma narrativa ou descrição dirigida por modelos de uma interação entre um ator e o software – é o principal elemento da modelagem. Obtido durante o levantamento de requisitos, ele define as etapas fundamentais para uma função ou interação específica. O grau de formalidade dos casos de uso e dos detalhes varia, porém o resultado final fornece a entrada necessária para as demais atividades da modelagem de análise. Os cenários também podem ser descritos usando-se um diagrama de atividades – uma representação gráfica do tipo fluxograma que representa o fluxo de processamento dentro de um cenário específico. Os diagramas de raia ilustram como o fluxo de processamento é alocado a vários atores ou classes.

Problemas e pontos a ponderar

- 9.1. Existe a possibilidade de começar a codificar logo depois de um modelo de requisitos ser criado? Justifique sua resposta e, em seguida, argumente ao contrário.
- 9.2. Uma regra prática para análise é que o modelo “deve se concentrar nos requisitos visíveis dentro do domínio do negócio ou do problema”. Que tipos de requisitos *não* são visíveis nesses domínios? Dê alguns exemplos.
- 9.3. Qual o propósito da análise de domínio? Como está relacionada com o conceito de padrões de requisitos?
- 9.4. É possível desenvolver um modelo de análise eficaz sem desenvolver todos os quatro elementos da Figura 9.3? Explique.
- 9.5. O departamento de obras públicas de uma grande cidade decidiu desenvolver um sistema de tapa-buracos (PHTRS, pothole tracking and repair system) baseado na Web. Segue uma descrição:

Os cidadãos podem entrar em um site e relatar o local e a gravidade dos buracos. À medida que são relatados, os buracos são registrados em um “sistema

de reparos do departamento de obras públicas” e recebem um número identificador, armazenado pelo endereço (nome da rua), tamanho (em uma escala de 1 a 10), localização (no meio da rua, meio-fio etc.), bairro (determinado com base no endereço) e prioridade para o reparo (determinada segundo o tamanho do buraco). Os dados de solicitação de trabalho são associados a cada buraco e incluem a localização e o tamanho do buraco, número de identificação da equipe de obras, o número de operários na equipe, equipamento alocado, horas usadas para o reparo, estado do buraco (trabalho em andamento, reparado, reparo temporário, não reparado), quantidade de material de preenchimento utilizado e custo do reparo (calculado com base nas horas utilizadas, no número de pessoas, no material e no equipamento usado). Por fim, é criado um arquivo de danos para armazenar informações sobre o dano relatado devido ao buraco e que inclui o nome, endereço e telefone do cidadão, tipo de dano e custo financeiro do dano. O PHTRS é um sistema online; todas as consultas devem ser feitas interativamente.

Desenhe um diagrama de caso de uso UML para o sistema PHTRS. Você terá de fazer uma série de suposições sobre a maneira como um usuário interage com esse sistema.

9.6. Escreva dois ou três casos de uso que definam os papéis de vários atores no PHTRS descritos no Problema 9.5.

9.7. Desenvolva um diagrama de atividades para um aspecto do PHTRS.

9.8. Desenvolva um diagrama de raias para um ou mais aspectos do PHTRS.

Leituras e fontes de informação complementares

Os casos de uso podem servir como base para todas as abordagens de modelagem de requisitos. O tema é discutido em detalhes por Gomaa (*Software Modeling: UML, Use Case, Patterns, and Architecture*, Cambridge University Press, 2011), Rosenberg e Stephens (*Use Case Driven Object Modeling with UML: Theory and Practice*, Apress, 2007), Denny (*Succeeding with Use Cases: Working Smart to Deliver Quality*, Addison-Wesley, 2005), Alexander e Maiden (Eds.) (*Scenarios, Stories, Use Cases: Through the Systems Development Life-Cycle*, Wiley, 2004), Bittner e Spence (*Use Case Modeling*, Addison-Wesley, 2002), Cockburn (Coc01b) e outras referências citadas no Capítulo 8.

Técnicas de modelagem UML que podem ser aplicadas tanto para análise como para projeto são discutidas por Dennis e seus colegas (*Systems Analysis and Design with UML Version 2.0*, 4ª ed., Wiley, 2012), O'Docherty (*Object-Oriented Analysis and Design: Understanding System Development with UML 2.0*, Wiley, 2005), Arlow e Neustadt (*UML 2 and the Unified Process*, 2ª ed., Addison-Wesley, 2005), Roques (*UML in Practice*, Wiley, 2004), Larman (*Applying UML and Patterns*, 2ª ed., Prentice Hall, 2001) e Rosenberg e Scott (*Use Case Driven Object Modeling with UML*, Addison-Wesley, 1999).

Alguns livros sobre requisitos incluem Robertson e Robertson (*Mastering the Requirements Process: Getting Requirements Right*, 3ª ed., Addison-Wesley, 2012), Hull, Jackson e Dick (*Requirements Engineering*, 3ª ed., Springer, 2010) e Alexander e Beus-Dukic (*Discovering Requirements: How to Specify Products and Services*, Wiley, 2009). Uma ampla gama de fontes de informação sobre modelagem de requisitos se encontra à disposição na Internet.

Uma lista atualizada de referências relevantes (em inglês) para a modelagem de análise pode ser encontrada no site: www.mhhe.com/pressman.

10 Modelagem de requisitos: métodos baseados em classes

Conceitos-chave

análise sintática	185
associações	198
atributos	188
classes de análise	185
colaborações	195
dependências	198
modelagem CRC	192
operações	189
pacotes de análise	199
responsabilidades	193

Quando introduzidos pela primeira vez, no início dos anos 1990, os métodos baseados em classes para modelagem de requisitos eram frequentemente classificados como *análise orientada a objetos*. Embora tenham sido apresentados vários métodos baseados em classes e representações, Coad e Yourdon [Coa91] observaram uma característica universal em todos eles:

[Todos os métodos orientados a objetos são] baseados nos conceitos que aprendemos no jardim de infância: objetos e atributos, totalidades e partes, classes e membros.

Os métodos baseados em classes para modelagem de requisitos utilizam esses conceitos comuns para produzir uma representação de uma aplicação que possa ser entendida por envolvidos sem conhecimento técnico. À medida que o modelo de requisitos é refinado e se expande, ele evolui para uma especificação que pode ser usada pelos engenheiros de software na criação do projeto de software.

A modelagem baseada em classes representa os objetos que o sistema vai manipular; as operações (também chamadas de *métodos* ou *serviços*) que vão ser aplicadas aos objetos para efetuar a manipulação; as relações (algumas hierárquicas) entre os objetos; e as colaborações que ocorrem entre as classes

PANORAMA

O que é? Quase sempre os problemas de software podem ser caracterizados considerando-se um conjunto de objetos interagentes, cada um representando algo de interesse dentro de um sistema. Cada objeto se torna membro de uma classe de objetos. Cada objeto é descrito pelo seu estado – os atributos dos dados que descrevem o objeto. Tudo isso pode ser representado com métodos de modelagem de requisitos baseados em classes.

Quem realiza? Um engenheiro de software (às vezes chamado de “analista”) constrói o modelo baseado em classes usando os requisitos extraídos do cliente.

Por que é importante? Um modelo de requisitos baseado em classes utiliza objetos extraídos na visão do cliente de uma aplicação ou sistema. O modelo representa uma visão do sistema simples para o cliente. Portanto, ele pode ser prontamente avaliado pelo cliente, resultando em feedback útil no menor tempo possível. Posteriormente, à medida que o modelo é refinado, ele se torna a base do projeto de software.

Quais são as etapas envolvidas? A modelagem baseada em classes define objetos, atributos e relações. Um conjunto de heurísticas simples pode ser desenvolvido para extrair objetos e classes do enunciado de um problema, representando-os então em formas baseadas em texto e/ou esquemáticas. Assim que os modelos preliminares forem criados, eles são refinados e analisados para avaliar sua clareza, completude e consistência.

Qual é o artefato? Uma ampla variedade de formas textuais e esquemáticas pode ser escolhida para o modelo de requisitos. Cada uma dessas representações dá uma visão de um ou mais dos elementos do modelo.

Como garantir que o trabalho foi realizado corretamente? Os artefatos do modelamento de requisitos devem ser revistos em termos de correção, completude e consistência. Devem refletir os requisitos de todos os envolvidos e estabelecer uma base a partir da qual o projeto pode ser conduzido.

definidas. Os elementos de um modelo baseado em classes são: classes e objetos, atributos, operações, modelos CRC (classe-responsabilidade-colaborador), diagramas de colaboração e pacotes. As seções a seguir apresentam uma série de diretrizes informais que auxiliarão na identificação e na representação desses elementos.

10.1 Identificação de classes de análise

Se olhar para uma sala, você verá que existe um conjunto de objetos físicos que podem ser facilmente identificados, classificados e definidos (em termos de atributos e operações). Porém, quando você “olha em torno” do espaço de problema de uma aplicação de software, as classes (e objetos) podem ser mais difíceis de compreender.

Podemos começar a identificar classes examinando os cenários de uso desenvolvidos como parte do modelo de requisitos (Capítulo 9) e realizando uma “análise sintática” [Abb83] dos casos de uso desenvolvidos para o sistema a ser construído. As classes são determinadas sublinhando-se cada substantivo ou frase nominal (que contém substantivos) e introduzindo-as em uma tabela simples. Os sinônimos devem ser anotados. Se for preciso que a classe (substantivo) implemente uma solução, então ela faz parte do espaço de soluções; caso contrário, se for necessária uma classe apenas para descrever uma solução, ela faz parte do espaço de problemas.

O que devemos procurar uma vez que todos os substantivos tenham sido isolados? As *classes de análise* se manifestam de uma das seguintes maneiras:

- *Entidades externas* (por exemplo, outros sistemas, dispositivos, pessoas) que produzem ou consomem informações a ser usadas por um sistema baseado em computadores.
- *Coisas* (por exemplo, relatórios, exibições, letras, sinais) que fazem parte do domínio de informações para o problema.
- *Ocorrências ou eventos* (por exemplo, uma transferência de propriedades ou a finalização de uma série de movimentos de robô) que ocorrem no contexto da operação do sistema.
- *Papéis* (por exemplo, gerente, engenheiro, vendedor) desempenhados pelas pessoas que interagem com o sistema.
- *Unidades organizacionais* (por exemplo, divisão, grupo, equipe) relevantes para uma aplicação.
- *Locais* (por exemplo, chão de fábrica ou área de carga) que estabelecem o contexto do problema e a função global do sistema.
- *Estruturas* (por exemplo, sensores, veículos de quatro rodas ou computadores) que definem uma classe de objetos ou classes de objetos relacionadas.

Essa categorização é apenas uma das muitas que foram propostas.¹ Por exemplo, Budd [Bud96] sugere uma taxonomia de classes que inclui *produto-*

“O problema realmente difícil é descobrir, logo no início, quais são os objetos [classes] corretos.”

Carl Argila

De que maneira as classes de análise se manifestam como elementos do espaço de soluções?

¹ Outra classificação importante, definindo classes de entidades, de contorno e de controle, é discutida na Seção 10.5.

res (fontes) e *consumidores* (reservatórios) de dados, *gerenciadores de dados*, *classes de visualização* ou *de observação* e *classes auxiliares*.

Também é importante destacar o que as classes ou objetos não são. Em geral, uma classe jamais deve ter um “nome procedural imperativo” [Cas89]. Por exemplo, se os desenvolvedores de software de um sistema de imagens para aplicação em medicina definiram um objeto com o nome **InverterImagem** ou até mesmo **InversãoDeImagem**, eles estariam cometendo um erro sutil. A **Imagem** obtida do software poderia, obviamente, ser uma classe (é algo que faz parte do domínio de informações). A inversão da imagem é uma operação aplicada ao objeto. É provável que a inversão seja definida como uma operação para o objeto **Imagem**, mas não seja definida como uma classe separada com a conotação “inversão de imagem”. Conforme Cashman [Cas89] diz, “O intuito da orientação a objetos é encapsular, mas, ainda, manter separados os dados e as operações sobre os dados”.

Para ilustrar como as classes de análise poderiam ser definidas durante os estágios iniciais da modelagem, considere uma análise sintática (os substantivos são sublinhados; os verbos, colocados em *itálico*) para uma narrativa² de processamento da função de segurança domiciliar do *CasaSegura*.

A função de segurança domiciliar do CasaSegura *permite* que o proprietário do imóvel *configure* o sistema de segurança quando ele é *instalado*, *monitore* todos os sensores conectados ao sistema de segurança e *interaja* com o proprietário do imóvel por meio da Internet, de um PC ou de um painel de controle.

Durante a instalação, o PC do CasaSegura é usado para *programar* e *configurar* o sistema. É atribuído um número e um tipo a cada sensor, é programada uma senha-mestra para *armar* e *desarmar* o sistema e são *introduzidos* número(s) de telefone para *disar* quando um evento de sensor ocorre.

Quando um evento de sensor é *reconhecido*, o software *aciona* um alarme audível ligado ao sistema. Após um tempo de retardo especificado pelo proprietário do imóvel durante as atividades de configuração do sistema, o software *disca* um número de telefone de um serviço de monitoramento, *fornece informações* sobre o local, *relatando* a natureza do evento detectado. O número de telefone será *discado novamente* a cada 20 segundos até que *seja completada* a ligação.

O proprietário do imóvel *recebe* informações de segurança por meio de um painel de controle, do PC ou de um navegador, coletivamente denominados interface. A interface *mostra* mensagens de aviso e informações sobre o estado do sistema no painel de controle, no PC ou na janela do navegador. A interação com o proprietário do imóvel acontece da seguinte forma...

A análise sintática não é infalível, mas pode proporcionar um excelente passo inicial, caso esteja havendo dificuldades para definir objetos de dados e as transformações que neles operam.

² A narrativa de processamento é similar ao caso de uso ou jornada de usuário em estilo, porém com uma finalidade diferente. A descrição provida pela narrativa são das funções a serem desenvolvidas. A busca por substantivos para identificar as classes candidatas pode ser feita para todos casos de usos desenvolvidos no levantamento de requisitos.

Extraindo os substantivos, podemos propor uma série de possíveis classes:

Classe em potencial	Classificação geral
proprietário	papel ou entidade externa
sensor	entidade externa
painel de controle	entidade externa
instalação	ocorrência
sistema (também conhecido como sistema de segurança)	coisa
número, tipo	não objetos, atributos do sensor
senha-mestra	coisa
número de telefone	coisa
evento de sensor	ocorrência
alarme audível	entidade externa
serviço de monitoramento	unidade organizacional ou entidade externa

A lista seguiria até que todos os substantivos contidos na narrativa de processamento tivessem sido considerados. Observe que chamamos cada entrada da lista de objeto “em potencial”. Temos de considerar cada um deles até que uma decisão final seja tomada.

Coad e Yourdon [Coa91] sugerem seis características de seleção que deveriam ser usadas à medida que se considera cada classe em potencial para inclusão no modelo de análise:

1. *Informações retidas.* A classe em potencial será útil durante a análise apenas se as informações sobre ela tiverem de ser lembradas para que o sistema possa funcionar.
2. *Serviços necessários.* A classe em potencial deve ter um conjunto de operações identificáveis capazes de modificar, de alguma forma, o valor de seus atributos.
3. *Atributos múltiplos.* Durante a análise de requisitos, o foco deve ser nas informações “importantes”; uma classe com um único atributo poderia, na verdade, ser útil durante o projeto; porém, provavelmente seria mais bem representada na forma de atributo de outra classe durante a atividade de análise.
4. *Atributos comuns.* Um conjunto de atributos pode ser definido para a classe em potencial e esses atributos se aplicam a todas as instâncias da classe.
5. *Operações comuns.* Um conjunto de operações pode ser definido para a classe em potencial e tais operações se aplicam a todas as instâncias da classe.
6. *Requisitos essenciais.* Entidades externas que aparecem no espaço do problema e produzem ou consomem informações essenciais à operação de qualquer solução para o sistema, quase sempre serão definidas como classes no modelo de requisitos. Para ser considerada uma classe legítima para inclusão no modelo de requisitos, um objeto em potencial deve satisfazer todas (ou quase todas) essas características. A decisão para inclusão de classes em potencial no modelo de análise é um tanto subjetiva, e uma avaliação posterior poderia fazer com que um objeto fosse descartado ou reintegrado.

Como podemos determinar se uma classe em potencial deve, de fato, se tornar uma classe de análise?

“As classes lutam. Algumas triunfam, outras são eliminadas.”

Mao Zedong

Entretanto, a primeira etapa da modelagem baseada em classes é definir as classes e decisões (mesmo aquelas subjetivas). Com isso em mente, devemos aplicar as características de seleção da lista de possíveis classes do *CasaSegura*:

Classe em potencial	Número característico que se aplica
proprietário	rejeitado: 1, 2 falham, embora 6 se aplique
sensor	aceito: todos se aplicam
painel de controle	aceito: todos se aplicam
instalação	rejeitado
sistema (também conhecido como sistema de segurança)	aceito: todos se aplicam
número, tipo	rejeitado: 3 falha, atributos de sensor
senha-mestra	rejeitado: 3 falha
número de telefone	rejeitado: 3 falha
evento de sensor	aceito: todos se aplicam
alarme audível	aceito: 2, 3, 4, 5, 6 se aplicam
serviço de monitoramento	rejeitado: 1, 2 falham, embora 6 se aplique

Deve-se notar o seguinte: (1) a lista anterior não é definitiva, outras classes talvez tenham de ser acrescentadas para completar o modelo; (2) algumas das classes em potencial rejeitadas se tornarão atributos para as que foram aceitas (por exemplo, número e tipo são atributos de **Sensor** e senha-mestra e número de telefone talvez se tornem atributos de **Sistema**); (3) enunciados do problema diferentes talvez provoquem decisões “aceito ou rejeitado” diferentes (por exemplo, se cada proprietário de um imóvel tivesse uma senha individual ou fosse identificado pelo seu padrão de voz, a classe **Proprietário** satisfaria as características 1 e 2 e teriam sido aceitas).

10.2 Especificação de atributos

Atributos representam o conjunto de objetos de dados que definem completamente a classe no contexto do problema.

Atributos descrevem uma classe selecionada para inclusão no modelo de análise. Em essência, são os atributos que definem uma classe – que esclarecem o que a classe representa no contexto do espaço de problemas. Por exemplo, se fôssemos construir um sistema que registrasse estatísticas dos jogadores do campeonato de beisebol profissional, os atributos da classe **Jogador** seriam bem diferentes dos atributos da mesma classe quando usados no contexto do sistema de aposentadoria dos jogadores profissionais de beisebol. No primeiro, atributos como nome, posição, média de rebatidas, porcentagem de voltas completas em torno do campo, número de anos jogados e número de jogos podem ser relevantes. No outro caso, alguns desses atributos seriam relevantes, mas outros seriam substituídos (ou ampliados) por atributos como salário médio, crédito faltante para aposentadoria plena, opções escolhidas para o plano de aposentadoria, endereço para correspondência e outros similares.

Para criarmos um conjunto de atributos que fazem sentido para uma classe de análise, devemos estudar cada caso de uso e escolher as “coisas” que “pertencem” àquela classe. Além disso, a pergunta a seguir deve ser respondida para cada uma das classes: *Quais dados (compostos e/ou elementares) definem completamente esta classe no contexto do problema à mão?”*.

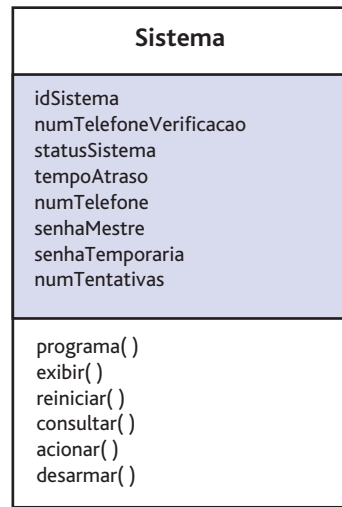


FIGURA 10.1 Diagrama de classes para a classe Sistema.

A título de ilustração, vamos considerar a classe **Sistema** definida para o *CasaSegura*. O proprietário de um imóvel pode configurar a função de segurança para que ela reflita informações sobre os sensores, sobre o tempo de resposta do alarme, sobre ativação/desativação, sobre identificação e assim por diante. Podemos representar esses dados compostos da seguinte maneira:

informação de identificação = ID do sistema + número de telefone para verificação + estado do sistema

informação da resposta do alarme = tempo de retardo + número de telefone

informação de ativação/desativação = senha-mestra + número de tentativas permitidas + senha temporária

Os dados à direita do sinal de igual poderiam ser definidos de forma mais ampla até um nível elementar; porém, para nossos propósitos, eles constituem uma lista de atributos adequada para a classe **Sistema** (a parte sombreada da Figura 10.1).

Os sensores fazem parte do sistema global *CasaSegura* e ainda não estão listados como dados ou atributos na Figura 10.1. **Sensor** já foi definido como uma classe, e vários objetos **Sensor** serão associados à classe **Sistema**. Em geral, evitamos definir um item como um atributo, caso mais de um dos itens deva ser associado à classe.

10.3 Definição das operações

Operações definem o comportamento de um objeto. Embora existam muitos tipos de operações, em geral elas podem ser divididas em quatro grandes categorias: (1) operações que manipulam dados (por exemplo, adição, eliminação, reformatação, seleção), (2) operações que efetuam um cálculo, (3) operações que pesquisam o estado de um objeto e (4) operações que monitoram um objeto quanto às ocorrências de um evento de controle. Tais funções são realizadas operando-se sobre atributos e/ou associações (Seção 10.5). Consequentemen-

Ao definir operações para uma classe de análise, concentre-se no comportamento orientado ao problema em vez de nos comportamentos necessários para a implementação.

te, uma operação deve ter “conhecimento” da natureza dos atributos e associações das classes.

Como primeira iteração na obtenção de um conjunto de operações para uma classe de análise, podemos estudar novamente uma narrativa de processamento (ou caso de uso) e escolher aquelas operações que pertencem de forma adequada à classe. Para tanto, a análise sintática é mais uma vez estudada e os verbos são isolados. Alguns dos verbos serão as operações legítimas, e elas podem ser facilmente associadas a uma classe específica. Por exemplo, da narrativa de processamento do *CasaSegura* apresentada anteriormente neste capítulo, vemos que “é atribuído um número e tipo aos sensores” ou “uma senha-mestra é *programada* para *armar* e *desarmar* o sistema”. Essas frases indicam uma série de coisas:

- Que uma operação *assign()* é relevante para a classe **Sensor**.
- Que uma operação *program()* será aplicada à classe **Sistema**.
- Que *arm()* e *disarm()* são operações que se aplicam à classe **Sistema**.

Após uma investigação mais apurada, é provável que a operação *program()* seja dividida em uma série de suboperações mais específicas, exigidas para configurar o sistema. Por exemplo, *program()* implica a especificação de números de telefone, a configuração de características do sistema (por exemplo, criar a tabela de sensores, introduzir características do alarme) e a introdução de senha(s). Mas, por enquanto, especificaremos *program()* como uma única operação.

Além da análise sintática, pode-se ter uma melhor visão sobre outras operações considerando-se a comunicação que ocorre entre os objetos. Os objetos se comunicam passando mensagens entre si. Antes de prosseguirmos com a especificação de operações, exploraremos essa questão com mais detalhes.

CASASEGURA



Modelos de classe

Cena: Sala do Ed, quando começa o modelamento da análise.

Atores: Jamie, Vinod e Ed – todos membros da equipe de engenharia de software do *CasaSegura*.

Conversa:

Ed vem trabalhando na extração de classes do modelo de casos de uso para o AVC-EVC (apresentado em um quadro anterior deste capítulo) e está mostrando a seus colegas as classes que extraiu.

Ed: Então, quando o proprietário de um imóvel quiser escolher uma câmera, terá de selecioná-la na planta da casa. Defini uma classe **Planta**. Aqui está o diagrama. (Eles observam a Figura 10.2.)

Jamie: Então, **Planta** é um objeto formado de paredes, portas, janelas e câmeras. É isso que estas linhas com identificação significam, certo?

Ed: Isso mesmo, elas são chamadas “associações”. Uma classe está associada a outra de acordo com as associações que eu mostrei. [As associações são discutidas na Seção 10.5.]

Vinod: Então, a planta real é feita de paredes e contém câmeras e sensores colocados nessas paredes. Como a planta da casa sabe onde colocar esses objetos?

Ed: Ela não sabe, mas as outras classes sim. Veja os atributos em, digamos, **TrechoParede**, que é usada para construir uma parede. O trecho de parede tem coordenadas de início e fim, e a operação *draw()* faz o resto.

Jamie: E o mesmo acontece com as janelas e portas. Parece-me que câmera possui alguns atributos extras.

Ed: Exatamente, preciso deles para fornecer informações sobre deslocamento e ampliação de imagens.

Vinod: Tenho uma pergunta. Por que a câmera possui um ID, mas os demais não? Percebi que você tem um atributo chamado **próximaParede**. Como **TrechoParede** saberá qual será a parede seguinte?

Ed: Boa pergunta, mas, como dizem, essa é uma decisão de projeto e, portanto, vou adiá-la até...

Jamie: Dá um tempo... aposto que você já bolou isso.

Ed (sorrindo timidamente): É verdade, vou usar uma estrutura de listas que irei modelar quando chegarmos ao

projeto. Se ficarmos muito presos à questão de separar análise e projeto, o nível de detalhe que tenho aqui poderia ser duvidoso.

Jamie: Para mim parece muito bom, mas tenho algumas outras perguntas. (Jamie faz perguntas que resultam em pequenas modificações.)

Vinod: Você tem cartões CRC para cada um dos objetos? Em caso positivo, deveríamos simular seus papéis, apenas para ter certeza de que nada tenha escapado.

Ed: Não estou bem certo de como fazê-lo.

Vinod: Não é difícil e realmente vale a pena. Vou lhes mostrar.

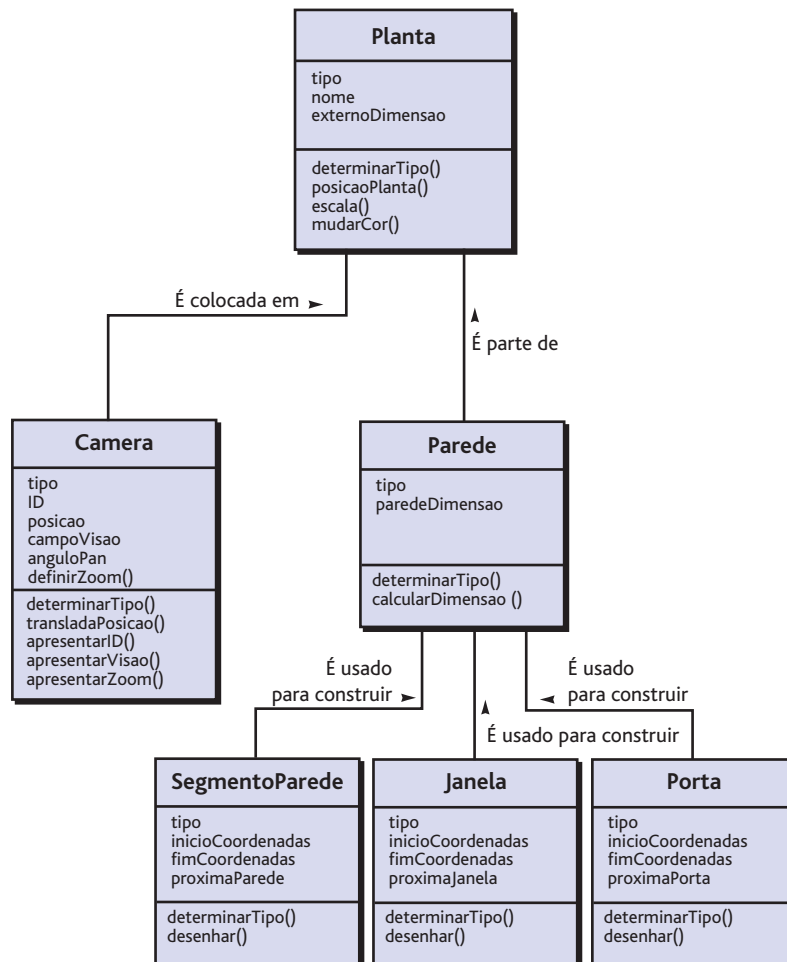


FIGURA 10.2 Diagrama de classes para Planta (FloorPlan) (veja a discussão no quadro).

10.4 Modelagem classe-responsabilidade-colaborador

"Um dos propósitos dos cartões CRC é fazer as falhas aparecerem logo, com frequência e de forma barata. É muito mais barato rasgar um monte de cartões do que reorganizar um grande volume de código-fonte."

C. Horstmann

Uma excelente discussão sobre esses tipos de classes pode ser encontrada em <http://www.oracle.com/technetwork/developer-tools/jdev/gettingstartedwithumlclassmodeling-130316.pdf>.

A *modelagem CRC (classe-responsabilidade-colaborador)* [Wir90] é uma maneira simples de identificar e organizar as classes relevantes para os requisitos do sistema ou produto. Ambler [Amb95] descreve a modelagem CRC da seguinte maneira:

Um modelo CRC é um conjunto de fichas-padrão que representam classes. Os cartões são divididos em três seções. Ao longo da parte superior do cartão escrevemos o nome da classe. No corpo do cartão enumeramos as responsabilidades da classe no lado esquerdo e os colaboradores no lado direito.

Na realidade, o modelo CRC pode fazer uso de fichas reais ou virtuais. O objetivo é desenvolver uma representação organizada das classes. *Responsabilidades* são os atributos e as operações relevantes para a classe. Em outras palavras, responsabilidade é "qualquer coisa que a classe sabe ou faz" [Amb95]. *Colaboradores* são as classes necessárias para fornecer a uma classe as informações necessárias para completar uma responsabilidade. Em geral, *colaboração* implica em uma solicitação de informações ou uma solicitação de alguma ação.

Um cartão CRC simples para a classe **Planta** está ilustrado na Figura 10.3. A lista de responsabilidades mostrada no cartão CRC é preliminar e sujeita a acréscimos ou modificações. As classes **Parede** e **Câmera** são indicadas ao lado da responsabilidade que vai exigir sua colaboração.

Classes. No início deste capítulo foram apresentadas diretrizes básicas para a identificação de classes e objetos. A taxonomia dos tipos de classe da Seção 10.1 pode ser estendida considerando-se as seguintes categorias:

- *Classes de entidade*, também denominadas classes de *modelo* ou *do negócio*, são extraídas diretamente do enunciado do problema (por exemplo, **Planta** e **Sensor**). Normalmente, essas classes representam

Classe: Planta	
Descrição	
Responsabilidade:	Colaborador:
Define o nome/tipo da planta	
Gerencia o posicionamento na planta	
Amplia/reduz a planta para exibição	
Amplia/reduz a planta para exibição	
Incorpora paredes, portas e janelas	Parede
Mostra a posição das câmeras de vídeo	Câmera

FIGURA 10.3 Um modelo de cartão CRC.

coisas que vão ser armazenadas em um banco de dados e persistem por toda a duração da aplicação (a não ser que sejam especificamente excluídas).

- *Classes de fronteira* são usadas para criar a interface (por exemplo, tela interativa ou relatórios impressos) que o usuário vê e interage à medida que o software é usado. Os objetos de entidades contêm informações importantes para os usuários, mas não são exibidos. Classes de fronteira são projetadas com a responsabilidade de controlar a maneira como os objetos de entidades são representados para os usuários. Por exemplo, uma classe de fronteira chamada **JanelaCâmera** teria a responsabilidade de exibir imagens de câmeras de vigilância do sistema *CasaSegura*.
- *Classes de controle* gerenciam uma “unidade de trabalho” do início ao fim. Isto é, as classes de controle podem ser desenvolvidas para gerenciar: (1) a criação ou a atualização de objetos de entidades, (2) a instanciação de objetos de fronteira à medida que forem obtendo informações dos objetos de entidades, (3) comunicação complexa entre conjuntos de objetos, (4) validação de dados transmitidos entre objetos ou entre o usuário e a aplicação. Em geral, as classes de controle não são consideradas até que a atividade de projeto tenha sido iniciada.

Responsabilidades. Diretrizes básicas para a identificação das responsabilidades (atributos e operações) foram apresentadas nas Seções 10.2 e 10.3. Wirfs-Brock e seus colegas [Wir90] sugerem cinco diretrizes para alocação de responsabilidades às classes:

1. **A inteligência do sistema deve ser distribuída pelas classes para melhor atender às necessidades do problema.** Toda aplicação engloba certo grau de inteligência; aquilo que o sistema sabe e aquilo que pode fazer. Essa inteligência pode ser distribuída pelas classes em uma série de maneiras. Classes “burras” (aquelas com poucas responsabilidades) podem ser modeladas para atuar como serventes para algumas poucas classes “inteligentes” (aquelas com muitas responsabilidades). Embora essa abordagem faça com que o fluxo de controle em um sistema se torne simples, ela apresenta algumas desvantagens: concentra toda a inteligência em algumas poucas classes, tornando as mudanças mais difíceis, e tende a exigir mais classes e, portanto, um esforço de desenvolvimento maior.

Se a inteligência do sistema for distribuída de forma mais homogênea pelas classes de uma aplicação, cada objeto conhecerá e fará apenas algumas poucas coisas (que em geral são bem focadas), e a coesão do sistema aumentará.³ Isso aumenta a facilidade de manutenção do software e reduz o impacto dos efeitos colaterais devido a mudanças.

“Os objetos podem ser classificados cientificamente em três grandes categorias: aqueles que não funcionam, aqueles que quebram e aqueles que se perdem.”

Russell Baker

Quais diretrizes podem ser aplicadas para alocar responsabilidades às classes?

³ Coesão é um conceito de projeto discutido no Capítulo 12.

Para determinar se a inteligência do sistema está distribuída de maneira apropriada, as responsabilidades indicadas em cada cartão CRC modelo devem ser avaliadas para determinar se alguma classe tem uma lista extraordinariamente longa de responsabilidades. Isso indica uma concentração de inteligência.⁴ Além disso, as responsabilidades para cada classe deveriam exibir o mesmo nível de abstração. Por exemplo, entre as operações listadas para uma classe agregada denominada **ContaCorrente**, um revisor indica duas responsabilidades: *saldo-da-conta* e *dar-baixa-cheques-compensados*. A primeira operação (responsabilidade) implica em um procedimento lógico e matemático razoavelmente complexo. A segunda é uma atividade administrativa simples. Como essas duas operações não se encontram no mesmo nível de abstração, *dar-baixa-cheques-compensados* deve ser colocada dentro das responsabilidades de **EntradaCheques**, uma classe que é englobada pela classe agregada **ContaCorrente**.

2. **Cada responsabilidade deve ser declarada da forma mais genérica possível.** Essa diretriz implica que as responsabilidades gerais (tanto atributos quanto operações) devem estar no topo da hierarquia de classes (por elas serem genéricas, serão aplicáveis a todas as subclasses).
3. **As informações e o comportamento relativos a elas devem residir na mesma classe.** Isso atende ao princípio da orientação a objetos denominado *encapsulamento*. Os dados e os processos que manipulam os dados devem ser empacotados como uma unidade coesa.
4. **As informações sobre um item devem estar em uma única classe e não distribuída por várias classes.** Uma única classe deve assumir a responsabilidade pelo armazenamento e manipulação de um tipo de informação específico. Tal responsabilidade não deve, em geral, ser compartilhada por uma série de classes. Se as informações forem distribuídas, o software se tornará mais difícil de ser mantido e testado.
5. **Quando adequado, as responsabilidades devem ser compartilhadas entre classes relacionadas.** Há muitos casos em que uma série de objetos relacionados deve apresentar o mesmo comportamento ao mesmo tempo. Como exemplo, consideremos um game que precise exibir as seguintes classes: **Jogador**, **CorpoJogador**, **BraçosJogador**, **PernasJogador**, **CabeçaJogador**. Cada uma dessas classes possui seus próprios atributos (por exemplo, posição, orientação, cor, velocidade) e todas têm de ser atualizadas e exibidas à medida que o usuário manipula um joystick. Consequentemente, as responsabilidades *atualizar* e *exibir* devem ser compartilhadas por cada um dos objetos citados. **Jogador** sabe quando algo mudou e quando é necessário *atualizar*. A classe colabora com os demais objetos para atingir uma nova posição ou orientação, porém cada objeto controla sua própria visualização.

⁴ Em tais situações, pode ser necessário quebrar uma classe em múltiplas classes ou subsistemas completos, de modo a distribuir a inteligência de modo mais eficiente.

Colaborações. As classes cumprem suas responsabilidades de duas formas: (1) uma classe pode usar suas próprias operações para manipular seus próprios atributos, cumprindo, portanto, determinada responsabilidade ou (2) uma classe pode colaborar com outras classes. Wirfs-Brock e seus colegas [Wir90] definem as colaborações da seguinte forma:

As colaborações representam solicitações de um cliente a um servidor no cumprimento de uma responsabilidade do cliente. Colaboração é a materialização do contrato entre o cliente e o servidor. Dizemos que um objeto colabora com outro se, para cumprir uma responsabilidade, precisa enviar mensagens ao outro objeto. Uma colaboração simples flui em uma direção – representando uma solicitação do cliente ao servidor. Do ponto de vista do cliente, cada uma de suas colaborações está associada a determinada responsabilidade implementada pelo servidor.

As colaborações são identificadas determinando se uma classe pode ou não cumprir cada responsabilidade por si só. Caso não possa, ela precisa interagir com outra classe – e ocorre a colaboração.

Como exemplo, consideremos a função de segurança domiciliar do *CasaSegura*. Como parte do procedimento de ativação, o objeto **PainelDeControle** deve determinar se algum sensor está aberto. A responsabilidade chamada *determinar-estado-sensor()* é definida. Se existirem sensores abertos, **PainelDeControle** tem de ativar um atributo de estado para “não preparado”. As informações dos sensores podem ser adquiridas de cada objeto **Sensor**. Consequentemente, a responsabilidade *determinar-estado-sensor()* pode ser cumprida apenas se **PainelDeControle** trabalhar em colaboração com **Sensor**.

Para ajudar na identificação dos colaboradores, podemos examinar três relações genéricas diferentes entre as classes [Wir90]: (1) a relação *faz-parte-de*, (2) a relação *tem-conhecimento-de* e (3) a relação *depende-de*. Cada uma dessas três relações genéricas é considerada resumidamente nos parágrafos a seguir.

Todas as classes que fazem parte de uma classe agregada são interligadas à classe agregada por meio de uma relação *faz-parte-de*. Considerando as classes definidas para o videogame citado anteriormente, a classe **CorpoJogador** *faz-parte-de* **Jogador**, assim como **BraçosJogador**, **PernasJogador** e **CabeçaJogador**. Em UML, tais relações são representadas na forma da agregação mostrada na Figura 10.4.

Quando uma classe tem de adquirir informações de outra classe, é estabelecida a relação *tem-conhecimento-de*. A responsabilidade *determinar-estado-sensor()* citada anteriormente é um exemplo de relação *tem-conhecimento-de*.

A relação *depende-de* implica que duas classes têm uma dependência que não é conseguida por *tem-conhecimento-de* ou *faz-parte-de*. Por exemplo, **CabeçaJogador** sempre tem de estar interligada a **CorpoJogador** (a menos que o videogame seja particularmente violento), embora cada objeto pudesse existir sem o conhecimento direto do outro. Um atributo do objeto **CabeçaJogador** chamado posição-central é determinado a partir da posição central do **CorpoJogador**. Essa informação é obtida por meio de um terceiro objeto, **Jogador**, que a adquire de **CorpoJogador**. Então, **CabeçaJogador** *depende-de* **CorpoJogador**.

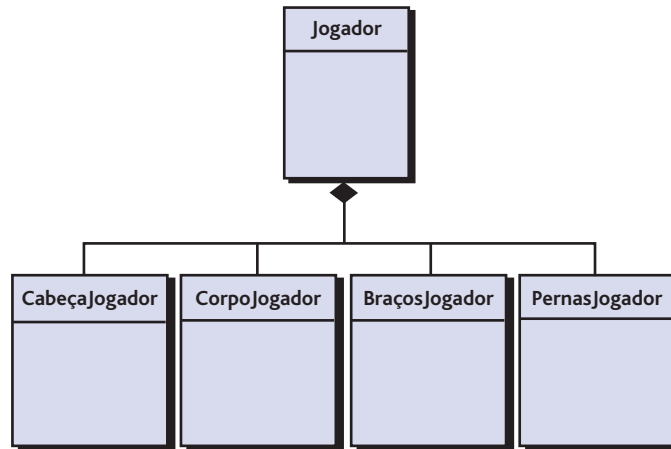


FIGURA 10.4 Uma classe composta de agregação.

Em todos os casos, o nome da classe colaboradora é registrado no cartão CRC modelo, próximo à responsabilidade que gerou a colaboração. Consequentemente, o cartão contém uma lista de responsabilidades e as colaborações correspondentes que permitem que as responsabilidades sejam cumpridas (Figura 10.3).

Quando um modelo CRC completo tiver sido desenvolvido, representantes dos envolvidos poderão revisar o modelo usando a seguinte abordagem [Amb95]:

1. Todos os participantes da revisão (do modelo CRC) recebem um subconjunto dos cartões CRC. Os cartões que colaboram devem ser separados (nenhum revisor deve ter dois cartões que colaboram).
2. Todos os cenários de uso (e diagramas de casos de uso correspondentes) devem ser organizados em categorias.
3. O líder da revisão lê o caso de uso pausadamente. À medida que o líder da revisão chega a um objeto com nome, ele passa uma ficha para a pessoa que está com o cartão da classe correspondente. Por exemplo, um caso de uso para o *CasaSegura* conteria a seguinte narrativa:

O proprietário do imóvel observa o painel de controle do *CasaSegura* para determinar se o sistema está pronto para operar. Se o sistema não estiver pronto, o proprietário do imóvel deve fechar manualmente as janelas/portas de modo que o indicador pronto esteja presente. [Um indicador não preparado implica que um sensor está aberto, isto é, que uma porta ou janela está aberta.]

Quando o líder da revisão chega em “painel de controle”, na narrativa de caso de uso, a ficha é passada para a pessoa que está com o cartão **PainelDeControle**. A frase “implica que um sensor está aberto” exige que o cartão contenha a responsabilidade que irá validar essa implicação (a responsabilidade *determinar-estado-sensor()* realiza isso). Próximo à responsabilidade no cartão se encontra o colaborador **Sensor**. A ficha é então passada para o objeto **Sensor**.

4. Quando a ficha é passada, solicita-se ao portador do cartão da classe que descreva as responsabilidades anotadas no cartão. O grupo deter-

mina se uma (ou mais) das responsabilidades satisfaz à necessidade do caso de uso.

- Se as responsabilidades e colaborações anotadas nos cartões não puderem satisfazer o caso de uso, são feitas modificações nos cartões. Elas podem incluir a definição das novas classes (e os cartões CRC correspondentes) ou a especificação de responsabilidades novas ou revisadas ou de colaborações em cartões existentes.

Esse *modus operandi* prossegue até que o caso de uso seja finalizado. Quando todos os casos de uso (ou diagramas de caso de uso) tiverem sido revisados, a modelagem de requisitos continua.



Modelos CRC

Cena: Sala do Ed, quando se inicia a modelagem de requisitos.

Atores: Vinod e Ed – membros da equipe de engenharia de software do *CasaSegura*.

Conversa:

Vinod decidiu mostrar a Ed como desenvolver cartões CRC por meio de um exemplo.

Vinod: Enquanto você trabalhava na função de vigilância e Jamie estava envolvido com a função de segurança, trabalhei na função de administração domiciliar.

Ed: E em que ponto você se encontra? O Marketing vive mudando de ideia.

Vinod: Aqui está o primeiro exemplo do caso de uso da função inteira... refinamos um pouco, mas deve dar uma visão geral...

Caso de uso: A função de administração domiciliar do *CasaSegura*.

Narrativa: Queremos usar a interface de administração domiciliar em um PC ou via Internet para controlar dispositivos eletrônicos que possuem controladores de interface sem fio. O sistema deve permitir que eu ligue e desligue luzes específicas, controle aparelhos que estiverem conectados a uma interface sem fio, configure o meu sistema de aquecimento e ar-condicionado para as temperaturas que eu desejar. Para tanto, quero escolher os dispositivos com base na planta da casa. Cada dispositivo tem de ser identificado na planta da casa. Como recurso opcional, quero controlar todos os dispositivos audiovisuais – áudio, televisão, DVD, gravadores digitais e assim por diante.

Com uma única seleção, quero ser capaz de configurar a casa inteira para várias situações. Uma delas é em casa, a outra é fora de casa, a terceira é viagem de fim de semana e a quarta é viagem prolongada. Todas essas situações terão

ajustes de configuração aplicados a todos os dispositivos. Nos estados viagem de fim de semana e viagem prolongada, o sistema deve acender e apagar as luzes da casa em intervalos aleatórios (para parecer que alguém está em casa) e controlar o sistema de aquecimento e ar-condicionado. Devo ser capaz de cancelar essas configurações via Internet com a proteção de uma senha apropriada...

Ed: O pessoal do hardware já bolou todas as interfaces sem fio?

Vinod (sorrindo): Eles estão trabalhando nisso; digamos que não há nenhum problema. De qualquer forma, extraí um monte de classes para a administração domiciliar e podemos usar uma delas como exemplo. Vamos usar a classe **InterfaceAdministraçãoDomiciliar**.

Ed: Certo... então, as responsabilidades são... os atributos e as operações para a classe, e as colaborações são as classes para as quais as responsabilidades apontam.

Vinod: Pensei que você não entendesse sobre CRC.

Ed: Um pouquinho, mas vá em frente.

Vinod: Então, aqui está minha definição de classe para **InterfaceAdministraçãoDomiciliar**.

Atributos:

opcoesPainel – contém informações sobre os botões que permitem ao usuário escolher a funcionalidade.

situacaoPainel – contém informações sobre os botões que permitem ao usuário escolher a situação.

planta – o mesmo que o objeto de vigilância, porém este aqui mostra os dispositivos.

dispositivosIcones – informações sobre os ícones representando luzes, aparelhos, HVAC etc.

dispositivosPainel – simulação do painel de controle de dispositivos ou de aparelhos; possibilita o controle.

CASASEGURA

Operações:

apresentarControle(), *selecionarControle()*, *apresentarSituacao()*, *selecionarSituacao()*, *acessarPlanta()*, *selecionarDispositivoVcone()*, *apresentarDispositivoPainel()*, *acessarDispositivoPainel()*,...

Classe: InterfaceAdministraçãoDomiciliar

Responsabilidade	Colaborador
<i>apresentarControle()</i>	opcoesPainel (classe)
<i>selecionarControle()</i>	opcoesPainel (classe)
<i>apresentarSituacao()</i>	situacaoPainel (classe)
<i>selecionarSituacao()</i>	situacaoPainel (classe)
<i>acessarPlanta()</i>	Planta (classe)...
...	

Ed: Então, quando a operação *acessarPlanta()* for chamada, ela colaborará com o objeto **Planta** exatamente como aquela que desenvolvemos para a função de vigilância. Espere, tenho uma descrição dela aqui comigo. (Eles observam a Figura 10.2.)

Vinod: Exatamente. E se quiséssemos rever o modelo de classes inteiro, poderíamos começar com este cartão, depois ir para o cartão do colaborador e a partir deste ponto para um dos colaboradores do colaborador e assim por diante.

Ed: Uma boa forma de descobrir omissões ou erros.

Vinod: Sim.

Uma associação define uma relação entre classes. A multiplicidade define quantos de uma classe estão relacionados com quantos de outra classe.

O que é um estereótipo?

10.5 Associações e dependências

Muitas vezes, duas classes de análise estão, de algum modo, relacionadas. Na UML, essas relações são chamadas de *associações*. Novamente na Figura 10.2, a classe **Planta** é definida identificando-se um conjunto de associações entre **Planta** e duas outras classes, **Câmera** e **Parede**. A classe **Parede** é associada a três classes que permitem que uma parede seja construída, **TrechoParede**, **Janela** e **Porta**.

Em alguns casos, uma associação pode ser mais bem definida indicando-se a *multiplicidade*. Referindo-nos à Figura 10.2, um objeto **Parede** é construído por meio de um ou mais objetos **TrechoParede**. Além disso, o objeto **Parede** pode conter nenhum ou alguns objetos **Janela** e nenhum ou alguns objetos **Porta**. Essas restrições de multiplicidade são ilustradas na Figura 10.5, em que “um ou mais” é representado por 1..*; e “nenhum ou alguns”, por 0..*. Na UML, o asterisco indica um limite superior infinito no intervalo.⁵

Em muitos casos, existe uma relação cliente/servidor entre duas classes de análise. Em tais casos, uma classe cliente depende da classe servidora de alguma forma e é estabelecida uma *relação de dependência*. Dependências são definidas por um estereótipo. *Estereótipo* é um “mecanismo de extensibilidade” [Arl02] dentro da UML que nos permite definir um elemento de modelagem especial cuja semântica é definida de forma personalizada. Na UML os estereótipos são representados entre << >> (por exemplo, <<stereotype>>).

Como exemplo de uma dependência simples no sistema de vigilância *CasaSegura*, um objeto **Câmera** (neste caso, a classe servidora) fornece uma imagem de vídeo para um objeto **ExibirJanela** (neste caso, a classe cliente). A

⁵ Outras relações de multiplicidade – um para um, um para muitos, muitos para muitos, um para um intervalo especificado com limites inferior e superior e outras – podem ser indicadas como parte de uma associação.

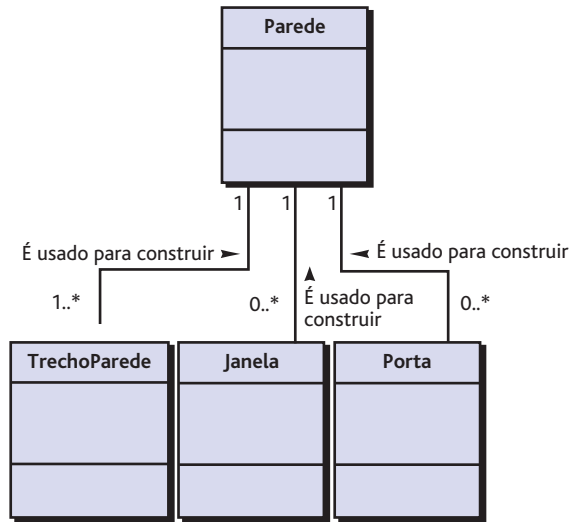


FIGURA 10.5 Multiplicidade.

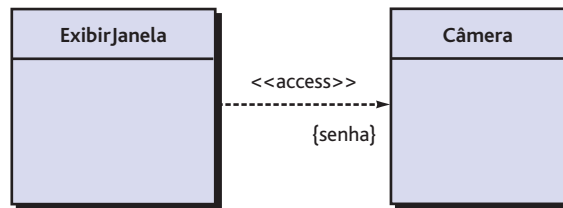


FIGURA 10.6 Dependências.

relação entre esses dois objetos não é uma associação simples, embora não exista uma associação de dependência. Em um caso de uso escrito para vigilância (não mostrado), você toma conhecimento de que uma senha especial tem de ser fornecida para visualizar posições de câmera específicas. Uma maneira de conseguir isso é fazer com que **Câmera** solicite uma senha e então dê permissão para **ExibirJanela** produzir a exibição de vídeo. Isso pode ser representado como mostra a Figura 10.6 em que `<<access>>` implica que o uso da saída de câmera é controlado por uma senha especial.

10.6 Pacotes de análise

Uma parte importante da modelagem de análise é a categorização. Vários elementos do modelo de requisitos (por exemplo, casos de uso, classes de análise) são categorizados de maneira a serem empacotados como um agrupamento – denominado *pacote de análise* – que recebe um nome representativo.

Para ilustrar o uso de pacotes de análise, considere o game apresentado anteriormente. À medida que o modelo de análise para o game é desenvolvido, um número maior de classes é extraído. Algumas focam o ambiente do jogo – as cenas que o usuário vê à medida que o jogo se desenrola. Classes como **Árvore**, **Paisagem**, **Rodovia**, **Parede**, **Ponte**, **Prédio** e **EfeitoVisual** poderiam cair dentro dessa categoria. Outras focam os personagens do jogo, des-

Um pacote é usado para montar um conjunto de classes relacionadas.

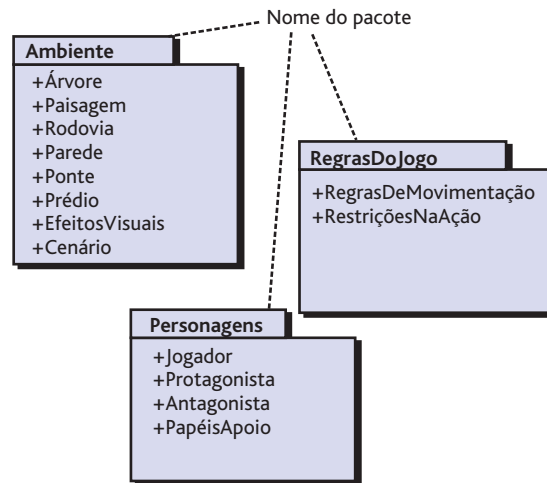


FIGURA 10.7 Pacotes.

crevendo suas características físicas, ações e restrições. Poderiam ser definidas classes como **Jogador** (descrita anteriormente), **Protagonista**, **Antagonista** e **PapéisApoio**. Outras, ainda, descrevem as regras do jogo – como um jogador navega pelo ambiente. Classes como **RegrasDeMovimentação** e **RestriçõesNaAção** são candidatas aqui. Poderiam existir muitas outras categorias. Essas classes podem ser representadas como classes de análise, conforme mostra a Figura 10.7.

O sinal de mais antes do nome da classe de análise nos pacotes indica que as classes têm visibilidade pública e são acessíveis a partir de outros pacotes. Embora não sejam mostrados na figura, outros símbolos podem anteceder um elemento dentro de um pacote. Um sinal de menos indica que um elemento está oculto para todos os demais pacotes e um símbolo # indica que um elemento é acessível apenas para pacotes contidos em determinado pacote.

10.7 Resumo

A modelagem baseada em classes usa informações extraídas de casos de uso e outras descrições da aplicação, escritas para identificar classes de análise. Uma análise sintática pode ser usada para extrair classes, atributos e operações candidatos com base em narrativas textuais. São definidos critérios para a definição de uma classe.

Um conjunto de cartões classe-responsabilidade-colaborador pode ser usado para definir relações entre classes. Além disso, uma variedade de notações da modelagem UML pode ser aplicada para definir hierarquias, relações, associações, agregações e dependências entre as classes. Pacotes de análise são usados para categorizar e agrupar classes de maneira que as tornem mais administráveis para grandes sistemas.

Problemas e pontos a ponderar

10.1 Você foi incumbido de construir um dos seguintes sistemas:

- a. Um sistema de matrícula em cursos baseado em rede para a sua universidade.
- b. Um sistema de processamento de pedidos baseado na Web para uma loja de informática.
- c. Um sistema de faturas simples para um pequeno negócio.
- d. Um livro de receitas baseado na Internet que é embutido em forno elétrico ou micro-ondas.

Escolha o sistema em que tiver interesse e desenvolva uma narrativa de processamento. Em seguida, use a técnica de análise sintática para identificar objetos e classes candidatos.

10.2 Desenvolva um conjunto de operações para serem usadas dentro das classes identificadas no Problema 10.1.

10.3 Desenvolva um modelo de classe para o sistema PHTRS apresentado no Problema 9.5.

10.4 Escreva um caso de uso baseado em modelos para o sistema de administração domiciliar *CasaSegura* descrito informalmente no quadro após a Seção 10.4.

10.5 Desenvolva um conjunto completo de cartões CRC sobre o produto ou sistema que você escolheu como parte do Problema 10.1.

10.6 Realize uma revisão dos cartões CRC com seus colegas. Quantas classes, responsabilidades e colaboradores adicionais são acrescentados como consequência da revisão?

10.7 O que é um pacote de análise e como poderia ser usado?

Leituras e fontes de informação complementares

Conceitos baseados em classes gerais são discutidos por Weisfeld (*The Object-Oriented Thought Process*, 4ª ed., Addison-Wesley, 2013). Métodos de modelagem baseados em classes são discutidos em livros de Bennet e Farmer (*Object-Oriented Systems Analysis and Design Using UML*, McGraw-Hill, 2010), Ashrafi e Ashrafi (*Object-Oriented Systems Analysis and Design*, Prentice Hall, 2008), Booch (*Object-Oriented Analysis and Design with Applications*, 3ª ed., Addison-Wesley, 2007), George e seus colegas (*Object-Oriented Systems Analysis and Design*, 2ª ed., Prentice Hall, 2006), O'Docherty (*Object-Oriented Analysis and Design*, Wiley, 2005), Satzinger *et al.* (*Object-Oriented Analysis and Design with the Unified Process*, Course Technology, 2004), Stumpf e Teague (*Object-Oriented Systems Analysis and Design with UML*, Prentice Hall, 2004).

Técnicas de modelagem com UML, que podem ser aplicadas tanto em análise quanto em projeto, são discutidas por Dennis e seus colegas (*Systems Analysis and Design with UML Version 2.0*, Wiley, 4ª ed., 2012), Ramnath e Dathan (*Object-Oriented Analysis and Design*, Springer, 2011), Bennett e Farmer (*Object-Oriented Systems Analysis and Design Using UML*, McGraw-Hill, 4ª ed., 2010). Larman (*Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*, Dohrling Kindersley, 2008), Rosenberg e Stephens (*Use Case Driven Object Modeling with UML Theory and Practice*, Apress, 2007) e Arlow e Neustadt (*UML 2 and the Unified Process*, 2ª ed., Addison-Wesley, 2005) tratam da definição de classes de análise no contexto da UML.

Uma ampla gama de fontes de informação sobre métodos baseados em classes se encontra à disposição na Internet. Uma lista atualizada de referências relevantes (em inglês) para a modelagem de análise pode ser encontrada no site: www.mhhe.com/pressman.

11

Modelagem de requisitos: comportamento, padrões e WebApps/aplicativos móveis

Conceitos-chave

diagramas de sequência.....	205
diagramas de estados.....	204
eventos.....	203
modelo comportamental.....	203
modelo de conteúdo.....	216
modelo de interação.....	217
modelo de navegação.....	220
modelo funcional.....	218
modelo de configuração.....	219
padrões de análise.....	208
representações de estados.....	204

Após nossa discussão sobre modelos baseados em cenários e em classes, nos Capítulos 9 e 10, podemos nos perguntar: “Essas representações de modelagem de requisitos não são suficientes?”

A única resposta adequada é: “Depende”.

Para alguns tipos de software, o caso de uso poderia ser a única representação de modelagem de requisitos exigida. Para outros, é escolhida uma abordagem orientada a objetos, e modelos baseados em classes poderiam ser desenvolvidos. Porém, em outras situações, requisitos de aplicação complexos poderiam exigir um exame de como uma aplicação se comporta como consequência de eventos externos; se o conhecimento do domínio existente pode ser adaptado ao problema atual; ou, no caso de sistemas e aplicações baseadas na Web (WebApps) ou aplicativos móveis, como o conteúdo e a funcionalidade combinados poderiam proporcionar a um usuário a capacidade de navegar com êxito por uma aplicação para atingir metas de utilização.

PANORAMA

O que é? Neste capítulo, você vai conhecer outras dimensões do modelo de requisitos – modelos comportamentais, padrões e considerações sobre análise de requisitos especiais que entram em cena quando WebApps são desenvolvidas. Cada uma dessas representações de modelagem complementa os modelos baseados em cenário e em classes discutidos nos Capítulos 9 e 10.

Quem realiza? Um engenheiro de software (às vezes denominado analista) constrói o modelo usando os requisitos extraídos de vários envolvidos.

Por que é importante? Sua visão sobre os requisitos do software aumenta em proporção direta ao número de diferentes dimensões da modelagem de requisitos. Embora talvez você não tenha tempo, recursos ou aptidão para desenvolver cada representação sugerida neste capítulo e nos Capítulos 9 a 11, deve reconhecer que cada abordagem de modelagem oferece uma forma diferente de visualizar o problema. Como consequência, você (e outros envolvidos) estará mais bem preparado para avaliar se o que deve ser feito foi ou não especificado de maneira apropriada.

Quais são as etapas envolvidas? A modelagem comportamental representa os estados do sistema e suas classes e o impacto dos eventos sobre esses estados. A modelagem baseada em padrões faz uso do conhecimento do domínio existente para facilitar a análise de requisitos. Os modelos de requisitos para WebApp são especialmente adaptados para a representação dos requisitos relacionados ao conteúdo, interação, funcionalidade e configuração.

Qual é o artefato? Uma ampla variedade de formas textuais e esquemáticas pode ser escolhida para o modelo de requisitos. Cada uma dessas representações dá uma visão de um ou mais elementos do modelo.

Como garantir que o trabalho foi realizado corretamente? Os artefatos da modelagem de requisitos devem ser revisados em termos de correção, completude e consistência. Devem refletir os requisitos de todos os envolvidos e estabelecer uma base a partir da qual o projeto pode ser conduzido.

11.1 Criação de um modelo comportamental

A notação de modelagem discutida nos capítulos anteriores representa elementos estáticos do modelo de requisitos. É chegado o momento de fazermos uma transição para o comportamento dinâmico do sistema ou produto. Para tanto, precisamos representar o comportamento do sistema em função do tempo e eventos específicos.

O *modelo comportamental* indica como o software vai responder a estímulos ou eventos externos. Para criá-lo, devemos executar as seguintes etapas: (1) avaliar todos os casos de uso para entender completamente a sequência de interação dentro do sistema, (2) identificar eventos que controlam a sequência de interação e entender como esses eventos se relacionam com objetos específicos, (3) criar uma sequência para cada caso de uso, (4) criar um diagrama de estado para o sistema e (5) examinar o modelo comportamental para verificar exatidão e consistência. Cada uma dessas etapas é discutida nas seções a seguir.

Como modelar a reação do software a algum evento externo?

11.2 Identificação de eventos com o caso de uso

No Capítulo 9, vimos que o caso de uso representa uma sequência de atividades que envolvem atores e o sistema. Em geral, um evento ocorre toda vez que o sistema e um ator trocam informações. Um evento *não* é a informação que foi trocada, mas sim o fato de que houve uma troca de informações.

Um caso de uso é examinado para encontrar pontos de troca de informação. Para ilustrarmos, reconsideremos o caso de uso de uma parte da função de segurança do *CasaSegura*.

O proprietário usa o teclado numérico para introduzir uma senha de quatro dígitos. A senha é comparada com a senha válida armazenada no sistema. Se a senha for incorreta, o painel de controle emitirá um bipe e reiniciará para receber novas entradas. Se a senha for correta, o painel de controle aguarda as próximas ações.

Os trechos sublinhados do cenário do caso de uso indicam eventos. Deve-se identificar um ator para cada evento; as informações trocadas devem ser indicadas e quaisquer condições ou restrições devem ser enumeradas.

Como um exemplo típico de evento, consideremos o trecho sublinhado do caso de uso “proprietário usa o teclado numérico para digitar uma senha de quatro dígitos”. No contexto do modelo de requisitos, o objeto **Proprietário**¹ transmite um evento para o objeto **PainelDeControle**. O evento poderia ser chamado *entrada de senha*. As informações transferidas são os quatro dígitos que constituem a senha, mas essa não é parte essencial do modelo comportamental. É importante notar que alguns eventos têm um impacto explícito no fluxo de controle do caso de uso, ao passo que outros não têm impacto direto no fluxo. Por exemplo, o evento *entrada de senha* não muda explicitamente o

¹ Neste exemplo, supomos que cada usuário (proprietário) que interage com o *CasaSegura* tem uma senha de identificação e, portanto, é um objeto legítimo.

fluxo de controle do caso de uso, mas os resultados do evento *comparação de senha* (derivado da interação “senha é comparada com a senha válida armazenada no sistema”) terá um impacto explícito no fluxo de controle e de informações do software *CasaSegura*.

Uma vez que todos os eventos tenham sido identificados, eles são alocados aos objetos envolvidos. Os objetos podem ser responsáveis pela geração de eventos (por exemplo, **Proprietário** gera um evento *entrada de senha*) ou reconhecem eventos que ocorreram em algum outro ponto (por exemplo, **PainelDeControle** reconhece o resultado binário do evento *comparação de senha*).

11.3 Representações de estados

No contexto da modelagem comportamental, devem ser consideradas duas caracterizações de estados distintas: (1) o estado de cada classe à medida que o sistema executa sua função e (2) o estado do sistema observado de fora, à medida que o sistema executa sua função.

O estado de uma classe pode assumir tanto características passivas quanto ativas [Cha93]. *Estado passivo* é o estado atual de todos os atributos de um objeto. Por exemplo, o estado passivo da classe **Jogador** (na aplicação de videogame discutida no Capítulo 10) incluiria atributos referentes à posição e orientação atual do **Jogador**, bem como outros recursos do **Jogador** relevantes ao jogo (por exemplo, um atributo que indique os desejos mágicos restantes). O *estado ativo* de um objeto indica seu estado atual à medida que passa por uma transformação ou processamento contínuo. A classe **Jogador** poderia ter os seguintes estados ativos: *em movimento*, *em repouso*, *machucado*, *em tratamento*, *preso*, *perdido* e assim por diante. Deve acontecer um *evento* (algumas vezes denominado *gatilho*) para forçar um objeto a fazer uma transição de um estado ativo para outro.

Nos próximos parágrafos serão discutidas duas representações comportamentais distintas. A primeira indica como determinada classe muda de estado com base em eventos externos, e a segunda mostra o comportamento do software em função do tempo.

Diagramas de estados para classes de análise. O componente de um modelo comportamental é um diagrama de estados em UML² que representa estados ativos para cada uma das classes e para os eventos (gatilhos) que provocam mudanças entre esses estados ativos. A Figura 11.1 ilustra um diagrama de estados para o objeto **PainelDeControle** na função de segurança do *CasaSegura*.

Cada seta da Figura 11.1 representa a transição de um estado ativo de um objeto para outro. As identificações em cada seta representam o evento que dispara a transição. Embora o modelo de estados ativos dê uma vi-

O sistema possui estados que representam um comportamento específico observável externamente; uma classe possui estados que representam seu comportamento à medida que o sistema executa suas funções.

² Caso não conheça UML, o Apêndice 1 apresenta uma breve introdução a essa importante notação de modelagem.

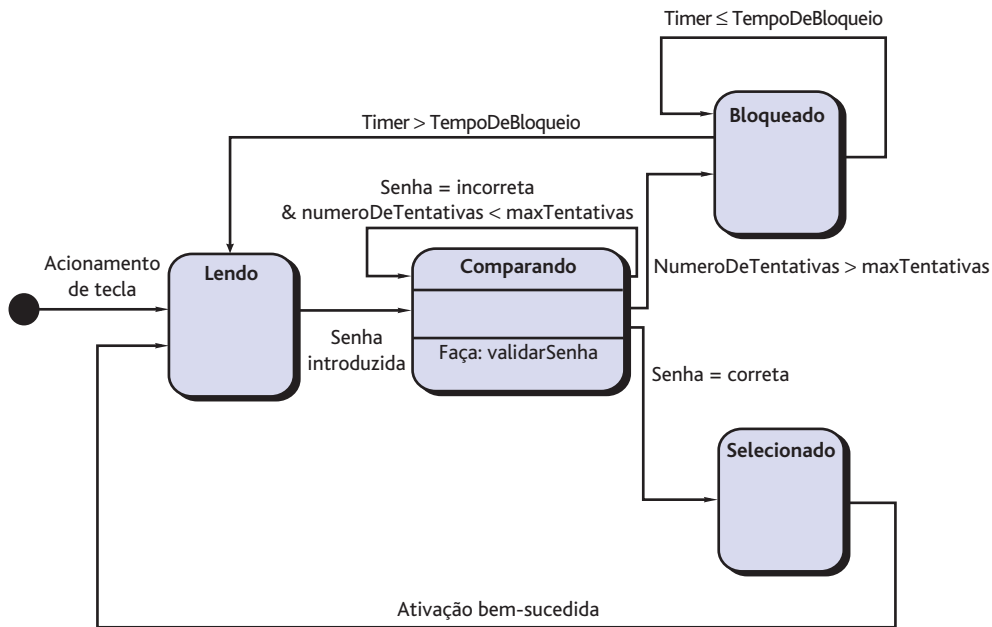


FIGURA 11.1 Diagrama de estados para a classe PaineleControle.

são proveitosa sobre a “história de vida” de um objeto, é possível especificar informações adicionais para uma maior profundidade no entendimento do comportamento de um objeto. Além de especificarmos o evento que faz com que a transição ocorra, podemos especificar um guarda e uma ação [Cha93]. *Guarda* é uma condição booleana que deve ser satisfeita para que a transição ocorra. Por exemplo, o guarda para a transição do estado “lendo” para o estado “comparando” na Figura 11.1 pode ser determinado examinando-se o caso de uso:

se (entrada de senha = 4 dígitos) então *compare* com a senha armazenada

Em geral, o guarda para uma transição normalmente depende do valor de um ou mais atributos de um objeto. Em outras palavras, o guarda depende do estado passivo do objeto.

Uma *ação* ocorre concomitantemente com a transição de estado ou como consequência dele e geralmente envolve uma ou mais operações (responsabilidades) do objeto. Por exemplo, a ação ligada ao evento *entrada de senha* (Figura 11.1) é uma operação chamada *validarSenha()* que acessa um objeto **senha** e realiza uma comparação dígito por dígito para validar a senha digitada.

Diagramas de sequência. O segundo tipo de representação comportamental, denominado *diagrama de sequência* em UML, indica como os eventos provocam transições de objeto para objeto. Uma vez que os eventos tenham sido identificados pelo exame de um caso de uso, o modelador cria um diagrama de sequência – uma representação de como os eventos provocam o fluxo de um objeto para outro em função do tempo. Em resumo, o diagrama

Diferentemente de um diagrama de estados, que representa comportamento sem citar as classes envolvidas, um diagrama de sequência representa comportamento descrevendo como as classes passam de um estado para outro.

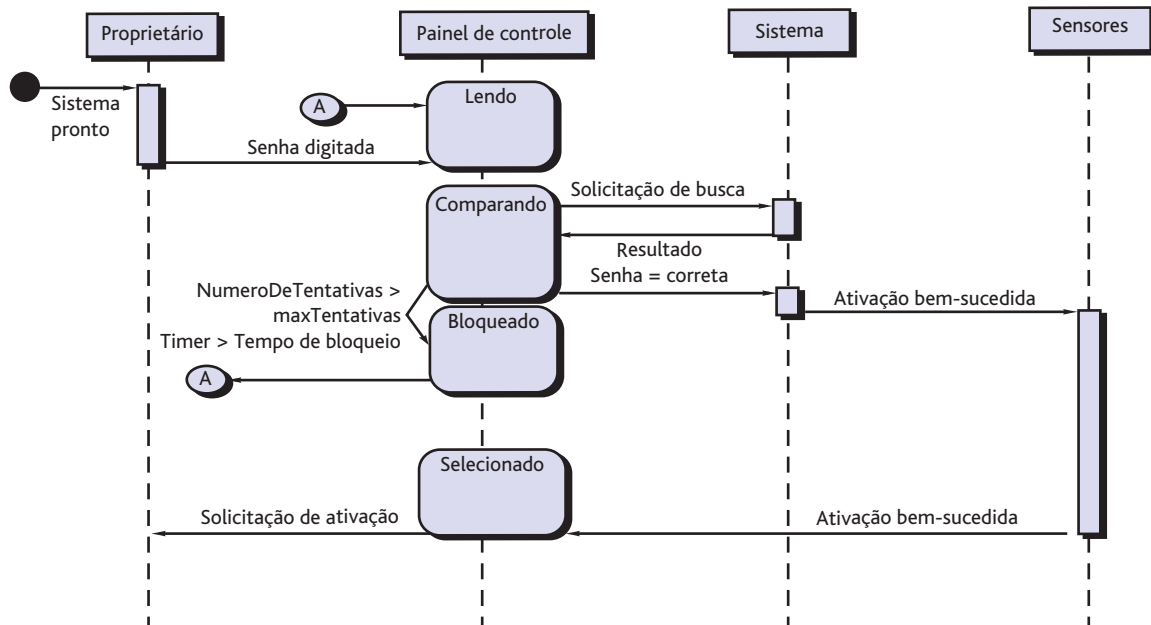


FIGURA 11.2 Diagrama de sequência (parcial) para a função de segurança do CasaSegura.

de sequência é uma versão abreviada do caso de uso. Ele representa classes-chave e os eventos que fazem com que o comportamento flua de classe para classe.

A Figura 11.2 ilustra um diagrama de sequência parcial para a função de segurança do CasaSegura. Cada uma das setas representa um evento (derivado de um caso de uso) e indica como o evento canaliza o comportamento entre os objetos do CasaSegura. O tempo é medido verticalmente (de cima para baixo) e os retângulos estreitos na vertical representam o tempo gasto no processamento de uma atividade. Os estados poderiam ser mostrados ao longo de uma linha do tempo vertical.

O primeiro evento, *sistema pronto*, é derivado do ambiente externo e canaliza comportamento para o objeto **Proprietário**. O proprietário do imóvel digita uma senha. Um evento *solicitação de busca* é passado para **Sistema**, o qual busca uma senha em um banco de dados simples e retorna um *resultado* (*encontrada* ou *não encontrada*) para **PainelDeControle** (agora no estado *comparando*). Uma senha válida resulta em um evento *senha=correta* para **Sistema**, o qual ativa **Sensores** com um evento *solicitação de ativação*. Por fim, o controle retorna ao proprietário com o evento *ativação bem-sucedida*.

Assim que um diagrama de sequência completo tiver sido desenvolvido, todos os eventos que provocam transições entre objetos do sistema podem ser reunidos em um conjunto de eventos de entrada e de saída (de um objeto). Essas informações são úteis na criação de um projeto eficaz para o sistema a ser construído.

FERRAMENTAS DO SOFTWARE

**Modelagem de análise generalizada em UML**

Objetivo: As ferramentas de modelagem de análise fornecem a capacidade de desenvolver modelos baseados em cenários, modelos baseados em classes e modelos comportamentais usando a notação UML.

Mecanismos: As ferramentas nesta categoria suportam todo o conjunto de diagramas UML necessários para construir um modelo de análise (as ferramentas também dão suporte à modelagem de projeto). Além da diagramação, elas (1) realizam verificação de consistência e correção para todos os diagramas UML, (2) fornecem links para projeto e geração de código, (3) constroem um banco de dados que permite o gerenciamento e a avaliação de modelos UML grandes, necessários para sistemas complexos.

Ferramentas representativas:³

As seguintes ferramentas dão suporte a todo o conjunto de diagramas UML necessários para a modelagem de análise:

ArgoUML é uma ferramenta com código-fonte aberto disponível em argouml.tigris.org.

Enterprise Architect, desenvolvida pela Sparx Systems (www.sparxsystems.com.au).

PowerDesigner, desenvolvida pela Sybase (www.sybase.com).

Rational Rose, desenvolvida pela IBM (Rational) (www-01.ibm.com/software/rational/).

Rational System Architect, desenvolvida pela Popkin Software, agora pertencente à IBM (<http://www-01.ibm.com/software/awdtools/systemarchitect/>).

UML Studio, desenvolvida pela Pragsoft Corporation (www.pragsoft.com).

Visio, desenvolvida pela Microsoft (<http://office.microsoft.com/en-gb/visio/>).

Visual UML, desenvolvida pela Visual Object Modelers (www.visualuml.com).

11.4 Padrões para a modelagem de requisitos

Padrões de software constituem um mecanismo para capturar conhecimento do domínio acumulado, para permitir que seja reaplicado quando um novo problema for encontrado. Em alguns casos, o conhecimento do domínio é aplicado a um novo problema no mesmo domínio de aplicação. Em outros, o conhecimento do domínio capturado por um padrão pode ser aplicado por analogia a um domínio de aplicação completamente diferente.

O autor original de um padrão de análise não “cria” o padrão, mas o *descobre* à medida que o fluxo de trabalho de levantamento de requisitos é conduzido. Assim que o padrão tiver sido descoberto, é documentado descrevendo-se “explicitamente o problema geral ao qual o padrão se aplica, a solução recomendada, hipóteses e restrições do emprego do padrão na prática e, em geral, algumas outras informações sobre o padrão, como a motivação e as forças que orientam seu uso, discussão sobre as vantagens e desvantagens do padrão, bem como referências para alguns exemplos conhecidos do uso desse padrão em aplicações práticas” [Dev01].

No Capítulo 8, apresentamos a noção de padrões de análise e indicamos que eles representam algo (por exemplo, uma classe, uma função, um comportamento) dentro do domínio de aplicação, que pode ser reutilizado ao se fazer

³ A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

a modelagem de requisitos para uma aplicação dentro de um domínio.⁴ Os padrões de análise são armazenados em um repositório para que os membros da equipe de software possam usar recursos de pesquisa para encontrá-los e reutilizá-los. Assim que um padrão apropriado tiver sido selecionado, ele é integrado ao modelo de requisitos por meio de referência ao nome do padrão.

11.4.1 Descoberta de padrões de análise

O modelo de requisitos compreende uma grande variedade de elementos: baseados em cenários (casos de uso), baseados em classes (objetos e classes) e comportamentais (eventos e estados). Cada um deles representa o problema segundo uma perspectiva e cada um proporciona a descoberta de padrões que poderiam ocorrer em um domínio de aplicação ou, por analogia, em domínios de aplicação diferentes.

O elemento mais básico na descrição de um modelo de requisitos é o caso de uso. No contexto desta discussão, um conjunto coerente de casos de uso poderia servir como base para a descoberta de um ou mais padrões de análise. Um *padrão de análise semântica* (SAP, *semantic analysis pattern*) “descreve um pequeno conjunto de casos de uso coerentes que, juntos, descrevem uma aplicação genérica básica” [Fer00].

Consideremos o seguinte caso de uso preliminar para um software necessário para controlar e monitorar um sensor de proximidade e uma câmera de visualização real para um automóvel:

Caso de uso: *Monitorar deslocamento em marcha à ré*

Descrição: Quando o veículo é colocado em *marcha à ré*, o software de controle habilita um alimentador de vídeo por meio de uma câmera de vídeo colocada atrás do painel de comandos. O software de controle sobrepõe uma variedade de linhas de orientação e distâncias na tela do painel de comandos para que o condutor do veículo possa ser orientado à medida que se desloca em *marcha à ré*. O software de controle também monitora um sensor de proximidade para determinar se um objeto se encontra ou não a 3 m da traseira do veículo. Ele vai frear o veículo automaticamente se o sensor de proximidade indicar um objeto a x metros da traseira do veículo, em que x é determinado com base na velocidade do veículo.

Esse caso de uso implica uma funcionalidade muito variada que poderia ser refinada e elaborada (em um conjunto de casos de uso coerentes) durante o levantamento de requisitos e a modelagem. Independentemente do nível de elaboração alcançado, os casos de uso sugerem um SAP simples, porém de ampla aplicação – o monitoramento e controle de sensores e atuadores baseado em software em um sistema físico. Nesse caso, os “sensores” fornecem informações sobre proximidade e informações de vídeo. O “atuador” é o sistema de frenagem do veículo (acionado caso um objeto esteja próximo ao veículo). Porém, em um caso mais genérico, descobre-se um padrão de ampla aplicação.

⁴ Uma discussão aprofundada do uso de padrões durante o projeto de software é apresentada no Capítulo 16.

CASASEGURA

**Descoberta de um padrão de análise**

Cena: Sala de reuniões, durante uma reunião da equipe.

Atores: Jamie Lazar, membro da equipe de software; Ed Robins, membro da equipe de software; Doug Miller, gerente de engenharia de software

Conversa:

Doug: Como está indo a modelagem dos requisitos da rede de sensores para o projeto do *CasaSegura*?

Jamie: Não conheço muito bem o funcionamento de sensores, mas acho que consigo resolver isso.

Doug: Há algo que possamos fazer para ajudá-lo?

Jamie: Seria muito mais fácil se eu já tivesse construído um sistema como esse antes.

Doug: É verdade.

Ed: Estou achando que essa é uma situação na qual podemos encontrar um padrão de análise para nos ajudar a modelar esses requisitos.

Doug: Se conseguirmos encontrar o padrão correto, não reinventaremos a roda.

Jamie: Parece bom, na minha opinião. Como começamos?

Ed: Temos acesso a um repositório que contém um grande número de padrões de análise e de projeto. Basta procurar-mos padrões com objetivos que correspondam às nossas necessidades.

Doug: Pode ser que isso dê certo. O que acha, Jamie?

Jamie: Se Ed puder me ajudar a começar, vou cuidar disso hoje mesmo.

Diferentes domínios de aplicação para monitorar sensores e controlar atuadores físicos necessitam de software. Um padrão de análise que descreva requisitos genéricos para essa capacidade poderia ser amplamente utilizado. O padrão, denominado **Atuador-Sensor**, seria aplicável como parte do modelo de requisitos do *CasaSegura* e é discutido na Seção 11.4.2.

11.4.2 Exemplo de padrão de requisitos: Atuador-Sensor⁵

Um dos requisitos da funcionalidade de segurança do *CasaSegura* é a capacidade de monitorar sensores de segurança (por exemplo, sensores contra roubos, sensores de fogo, fumaça ou CO, sensores de água). Extensões para o *CasaSegura* baseadas na Internet exigirão a capacidade de controlar o movimento (por exemplo, deslocamento horizontal e vertical, ampliação/redução) de uma câmera de segurança no interior de uma residência. A implicação: o software *CasaSegura* deve gerenciar vários sensores e “atuadores” (por exemplo, mecanismos de controle de câmera).

Konrad e Cheng [Kon02] sugeriram um padrão de requisitos chamado **Atuador-Sensor** que fornece uma orientação útil para modelar esses requisitos no software *CasaSegura*. Uma versão simplificada do padrão **Atuador-Sensor**, originalmente desenvolvido para aplicações na indústria automobilística, é apresentada a seguir.

Nome do Padrão. Atuador-Sensor

Intuito. Especificar vários tipos de sensores e atuadores em um sistema embarcado.

⁵ Esta seção foi adaptada de [Kon02] com a permissão dos autores.

Motivo. Os sistemas embarcados normalmente possuem vários tipos de sensores e atuadores. Esses sensores e atuadores estão todos direta ou indiretamente conectados a uma unidade de controle. Embora muitos sensores e atuadores pareçam bem diferentes, seu comportamento é suficientemente parecido para estruturá-los em um padrão. O padrão mostra como especificar os sensores e atuadores para um sistema, inclusive atributos e operações. O padrão **Atuador-Sensor** usa um mecanismo *pull* (solicitação explícita de informação) para **PassiveSensors** (SensoresPassivos) e um mecanismo *push* (difusão de informação) para os **ActiveSensors** (SensoresAtivos).

Restrições

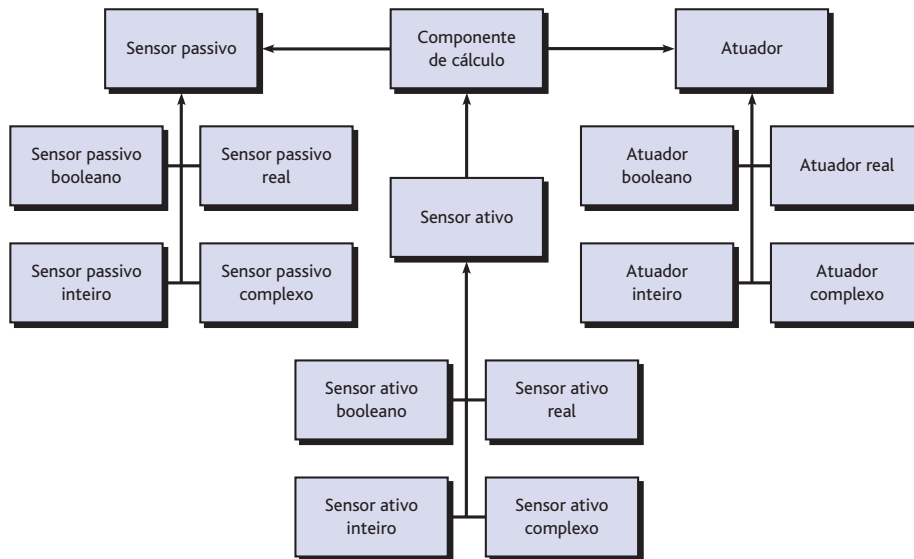
- Cada sensor passivo deve ter algum método para ler a entrada e atributos que representam o valor do sensor.
- Cada sensor ativo deve ter capacidades para transmitir mensagens de atualização quando seu valor muda.
- Cada sensor ativo deve enviar um *senal de vida*, uma mensagem de estado emitida em determinado intervalo de tempo, para detectar defeitos.
- Cada atuador deve ter algum método para chamar a resposta apropriada, determinada por **ComputingComponent** (ComponenteDeRealização).
- Cada sensor e atuador deve ter uma função implementada para verificar seu próprio estado de operação.
- Cada sensor e atuador deve ser capaz de testar a validade dos valores recebidos ou enviados e estabelecer seu estado de operação caso os valores se encontrem fora das especificações.

Aplicabilidade. Útil em qualquer sistema em que vários sensores e atuadores estão presentes.

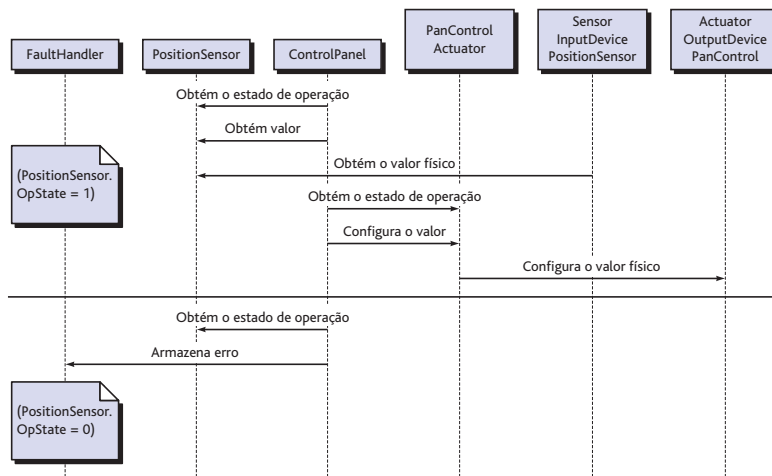
Estrutura. Um diagrama de classes UML para o padrão **Atuador-Sensor** aparece na Figura 11.3. **Atuador**, **PassiveSensor** e **ActiveSensor** são classes abstratas e grafadas em itálico. Existem quatro tipos diferentes de sensores e atuadores nesse padrão. As classes **Boolean**, **Integer** e **Real** representam os tipos mais comuns de sensores e atuadores. As classes complexas são sensores ou atuadores que usam valores que não podem ser facilmente representados em termos de tipos de dados primitivos, como um dispositivo de radar. Não obstante, tais dispositivos ainda devem herdar a interface das classes abstratas, pois devem ter funcionalidades básicas como consulta aos estados de operação.

Comportamento. A Figura 11.4 apresenta um diagrama de sequência UML para um exemplo do padrão **Atuador-Sensor**, conforme poderia ser aplicado à função do *CasaSegura* que controla o posicionamento (por exemplo, deslocamentos, ampliação/redução) de uma câmera de segurança. Nesse caso, **ControlPanel**⁶ (PainelDeControle) consulta um sensor (um sensor de posição passivo) e um atuador (controle de deslocamentos) para verificar o estado de operação para fins de diagnóstico antes de ler ou configurar um valor. *SetPhy-*

⁶ O padrão original usa o termo genérico **ComputingComponent**.

**FIGURA 11.3** Diagrama de sequência UML para o padrão Atuador-Sensor.

Fonte: Adaptado de [Kon02] com permissão

**FIGURA 11.4** Diagrama de classes UML para o padrão Atuador-Sensor.

Fonte: Reproduzido de [Kon02] com permissão.

sicalValue (AjustarValorFísico) e *GetPhysicalValue* (ObterValorFísico) não são mensagens entre objetos. Em vez disso, descrevem a interação entre os dispositivos físicos do sistema e seus equivalentes em software. Na parte inferior do diagrama, abaixo da linha horizontal, **PositionSensor** (SensorPosicional) informa que o estado da operação é zero. **ComputingComponent** envia, então, o código de erro devido a uma falha no sensor de posição para **FaultHandler** (ControleDeFalhas), que decidirá como esse erro afeta o sistema e que medidas são necessárias. Ele obtém os dados dos sensores e calcula a resposta necessária para os atuadores.

Participantes. Essa seção de descrição dos padrões “elencas as classes/objetos incluídos no padrão de requisitos” [Kon02] e descreve as responsabilidades de cada classe/objeto (Figura 11.3). A seguir, temos uma lista resumida:

- **PassiveSensor abstract:** Define uma interface para sensores passivos.
- **PassiveBooleanSensor:** Define sensores passivos booleanos.
- **PassiveIntegerSensor:** Define sensores passivos inteiros.
- **PassiveRealSensor:** Define sensores passivos reais.
- **ActiveSensor abstract:** Define uma interface para sensores ativos.
- **ActiveBooleanSensor:** Define sensores ativos booleanos.
- **ActiveIntegerSensor:** Define sensores ativos inteiros.
- **ActiveRealSensor:** Define sensores ativos reais.
- **Actuator abstract:** Define uma interface para atuadores.
- **BooleanActuator:** Define atuadores booleanos.
- **IntegerActuator:** Define atuadores inteiros.
- **RealActuator:** Define atuadores reais.
- **ComputingComponent:** A parte principal do controlador; obtém os dados dos sensores e calcula a resposta necessária para os atuadores.
- **ActiveComplexSensor:** Os sensores ativos complexos possuem a funcionalidade básica da classe abstrata **ActiveSensor**; porém, métodos e atributos adicionais mais elaborados precisam ser especificados.
- **PassiveComplexSensor:** Os sensores passivos complexos possuem a funcionalidade básica da classe abstrata **PassiveSensor**; porém, métodos e atributos adicionais mais elaborados precisam ser especificados.
- **ComplexActuator:** Os atuadores complexos também possuem a funcionalidade básica da classe abstrata **Actuator**; porém, métodos e atributos adicionais mais elaborados precisam ser especificados.

Colaborações. Essa seção descreve como os objetos e as classes interagem entre si e como cada uma delas cumpre suas obrigações.

- Quando **ComputingComponent** precisa atualizar o valor de um **PassiveSensor**, consulta os sensores, solicitando o valor pelo envio de uma mensagem apropriada.
- **ActiveSensors** não são consultados. Eles iniciam a transmissão de valores de sensor para a unidade de cálculo usando o método apropriado para configurar o valor no **ComputingComponent**. Eles enviam um sinal pelo menos uma vez durante um intervalo de tempo especificado para atualizar seus registros de horas com o horário do relógio do sistema.
- Quando **ComputingComponent** precisa configurar o valor de um atuador, envia o valor para o atuador.

- **ComputingComponent** pode consultar e configurar o estado da operação dos sensores e atuadores usando os métodos apropriados. Se for constatado que um estado da operação é zero, o erro é enviado a **FaultHandler**, uma classe que contém métodos para tratar mensagens de erro como, por exemplo, acionar um mecanismo de recuperação mais elaborado ou um dispositivo de backup. Se a recuperação não for possível, o sistema poderá usar apenas o último valor conhecido do sensor ou o valor padrão.
- **ActiveSensors** oferece métodos para acrescentar ou remover os endereços ou intervalos de endereços dos componentes que querem receber as mensagens no caso de mudança de um valor.

Consequências

1. As classes de sensores e atuadores possuem uma interface comum.
2. Os atributos de classes podem ser acessados apenas por meio de mensagens, e a classe decide se aceita ou não uma mensagem. Por exemplo, se o valor de um atuador estiver configurado acima de seu valor máximo, a classe do atuador talvez não aceite a mensagem – ou então use um valor máximo padrão.
3. A complexidade do sistema é potencialmente reduzida devido à uniformidade das interfaces para atuadores e sensores.

A descrição dos padrões de requisitos também poderia fornecer referências a outros padrões de projeto e de requisitos relacionados.

11.5 Modelagem de requisitos para WebApps e aplicativos móveis⁷

Os desenvolvedores de WebApps e aplicativos móveis frequentemente ficam céticos quando a ideia de análise de requisitos é sugerida. “Afim de contas”, argumentam eles, “nosso processo de desenvolvimento deve ser ágil, e a análise toma muito tempo. Ela vai nos atrasar quando precisamos apenas projetar e construir a aplicação”.

A análise de requisitos realmente leva tempo, porém resolver o problema errado leva mais tempo ainda. A pergunta para cada desenvolvedor de WebApp e aplicativos móveis é simples: você tem certeza de que entendeu os requisitos do problema ou produto? Se a resposta for um inequívoco “sim”, talvez seja possível pular a fase de modelagem de requisitos; porém, se a resposta for “não”, a modelagem de requisitos deve ser realizada.

⁷ Partes desta seção foram adaptadas de Pressman e Lowe [Pre08] com permissão.

11.5.1 Que nível de análise é suficiente?

O grau com o qual a modelagem de requisitos para WebApps e aplicativos móveis é enfatizado depende dos seguintes fatores relacionados ao tamanho: (1) o tamanho e a complexidade do incremento da aplicação, (2) o número de envolvidos (a análise pode ajudar a identificar requisitos conflitantes provenientes de diferentes fontes), (3) o tamanho da equipe de desenvolvimento de aplicativos, (4) por quanto tempo os membros da equipe trabalharam juntos (a análise pode ajudar a desenvolver um entendimento comum do projeto) e (5) até que ponto o sucesso da organização depende diretamente do sucesso da aplicação.

O contrário dos pontos anteriores é que, à medida que um projeto se torna menor, que o número de envolvidos diminui, que a equipe se torna mais coesa, e a aplicação, menos crítica, é sensato aplicar uma abordagem de análise menos rígida.

Embora seja uma boa ideia analisar os requisitos do problema ou produto *antes* de começar o projeto, não é verdade que *toda* análise deve preceder *todo* projeto. Na verdade, o projeto de uma parte específica da aplicação exige apenas uma análise dos requisitos que afetam somente essa parte. Um exemplo para o *CasaSegura*: sem dúvida, poderíamos projetar toda a estética do site (layouts, esquemas de cores etc.) sem ter analisado os requisitos funcionais para recursos de comércio eletrônico. Precisaríamos analisar apenas a parte do problema relevante ao trabalho de projeto do incremento a ser entregue.⁸

11.5.2 Entrada da modelagem de requisitos

Uma versão ágil do processo de software genérico discutido no Capítulo 5 pode ser aplicada quando WebApps ou aplicativos móveis são criados. O processo incorpora uma atividade de comunicação que identifica envolvidos e categorias de usuário, o contexto de negócio, metas de aplicação e de informação definidas, requisitos gerais do produto e cenários de uso – as informações se tornam entrada para a modelagem de requisitos. Estas são representadas na forma de descrições em linguagem natural, descrições gerais, esboços e outras representações de informação.

A análise captura essas informações, estrutura-as usando um esquema de representação formalmente definido (onde apropriado) e depois produz como saída modelos mais rigorosos. O modelo de requisitos fornece uma indicação detalhada da verdadeira estrutura do problema e dá uma visão da forma da solução.

A função ACS-EVC (vigilância por meio de câmeras) do *CasaSegura* foi apresentada no Capítulo 9. Quando foi introduzida, essa função parecia relativamente clara e foi descrita com certo nível de detalhamento como parte de um caso de uso (Seção 9.2.1). Entretanto, um reexame do caso de uso poderia revelar informações ausentes, ambíguas ou confusas.

Alguns aspectos das informações faltantes emergiriam naturalmente durante o projeto. Exemplos poderiam incluir o layout específico dos botões de função, seu aspecto estético, o tamanho das visões instantâneas, o posiciona-

⁸ Em situações nas quais o projeto de uma parte de uma aplicação terá impacto em outras partes dela, a abrangência da análise deve ser ampliada.

mento das visões das câmeras e da planta do imóvel ou até mesmo minúcias como o comprimento máximo e mínimo das senhas. Alguns desses aspectos são decisões de projeto (como o layout dos botões), e outros são requisitos (como o tamanho das senhas) que, fundamentalmente, não influenciam os trabalhos iniciais do projeto.

Porém, algumas informações faltantes poderiam realmente influenciar o próprio projeto como um todo e estar mais relacionadas a um entendimento real dos requisitos. Por exemplo:

- Q1: Qual a resolução de vídeo fornecida pelas câmeras do *CasaSegura*?
- Q2: O que acontece se uma condição de alarme for encontrada enquanto a câmera estiver sendo monitorada?
- Q3: Como o sistema manipula câmeras que possam ser deslocadas e ampliadas/reduzidas?
- Q4: Que informações deveriam ser fornecidas juntamente com a visão das câmeras? (Por exemplo, posição? Hora/data? Último acesso anterior?)

Nenhuma dessas perguntas foi identificada ou considerada no desenvolvimento inicial do caso de uso, muito embora as respostas possam ter um efeito significativo em diferentes aspectos do projeto.

Consequentemente, podemos concluir que, embora a atividade de comunicação forneça uma boa fundamentação para o entendimento, a análise de requisitos refina esse entendimento por meio de interpretações adicionais. À medida que a estrutura do problema é delineada como parte do modelo de requisitos, invariavelmente surgem perguntas. São essas perguntas que preenchem as lacunas – ou, em alguns casos, realmente nos ajudam a encontrar as lacunas.

Em suma, as entradas para o modelo de requisitos serão as informações coletadas durante a atividade de comunicação – qualquer uma, desde um e-mail informal até uma descrição de projeto detalhada contendo cenários de uso e especificações de produto completas.

11.5.3 Saída da modelagem de requisitos

A análise de requisitos fornece um mecanismo disciplinado para representar e avaliar o conteúdo e a função de uma aplicação, os modos de interação que os usuários vão encontrar e o ambiente e a infraestrutura em que a WebApp ou aplicativo móvel reside.

Cada uma dessas características pode ser representada como um conjunto de modelos que permitem analisar os requisitos da aplicação de maneira estruturada. Embora os modelos específicos dependam em grande parte da natureza da aplicação, há cinco classes principais de modelos:

- **Modelo de conteúdo** – identifica o espectro completo do conteúdo a ser fornecido pela aplicação. Conteúdo pode ser texto, gráficos, imagens, vídeo e áudio.
- **Modelo de interações** – descreve a maneira como os usuários interagem com a aplicação.

- **Modelo funcional** – define as operações que serão aplicadas para manipular o conteúdo e descreve outras funcionalidades de processamento independentes do conteúdo, mas necessárias para o usuário.
- **Modelo de navegação** – define a estratégia geral de navegação para a aplicação.
- **Modelo de configuração** – descreve o ambiente e a infraestrutura na qual a aplicação reside.

Podemos desenvolver cada um desses modelos usando um esquema de representação (em geral chamado “linguagem”) que permite que seu intuito e estrutura sejam comunicados e avaliados facilmente entre os membros da equipe de engenharia e outros envolvidos. Como consequência, várias questões fundamentais (por exemplo, erros, omissões, inconsistências, sugestões para aperfeiçoamento ou modificações, pontos a ser esclarecidos) são identificadas e posteriormente trabalhadas.

11.5.4 Modelo de conteúdo

O modelo de conteúdo contém elementos estruturais que dão uma visão importante dos requisitos de conteúdo para uma aplicação. Eles englobam objetos de conteúdo e todas as *classes de análise* – entidades visíveis para os usuários, criadas ou manipuladas à medida que o usuário interage com a aplicação por meio de um navegador ou dispositivo móvel.⁹

O conteúdo pode ser desenvolvido antes da implementação da aplicação, enquanto ela está sendo construída ou bem depois de estar em funcionamento. Em todos os casos, ela é incorporada à estrutura geral da aplicação via referência navegacional. Um *objeto de conteúdo* pode ser uma descrição textual de um produto, um artigo descrevendo um evento de noticiário, uma representação gráfica de dados recuperados (por exemplo, preço de ações como uma função do tempo), uma fotografia de ação tirada durante um evento esportivo, a resposta de um usuário em um fórum de discussão, uma animação do logotipo de uma empresa, um breve vídeo de apresentação ou um áudio agregado a um conjunto de slides de uma apresentação. Os objetos de conteúdo poderiam ser armazenados como arquivos distintos ou obtidos dinamicamente de um banco de dados. Poderiam ser incorporados diretamente em páginas Web exibidas na tela de um dispositivo móvel. Em outras palavras, um objeto de conteúdo é qualquer informação coesa que deve ser apresentada a um usuário.

Os objetos de conteúdo podem ser determinados diretamente dos casos de uso, examinando-se a descrição do cenário para referências diretas e indiretas ao conteúdo. Por exemplo, uma WebApp que oferecesse suporte ao *CasaSegura* seria estabelecida em www.casaseguragarantida.com. Um caso de uso, *Comprando Componentes Específicos do CasaSegura*, descreve o cenário necessário para adquirir um componente do *CasaSegura* e contém a seguinte frase:

Serei capaz de obter informações descritivas e de preços para cada componente do produto.

⁹ As classes de análise foram discutidas no Capítulo 10.

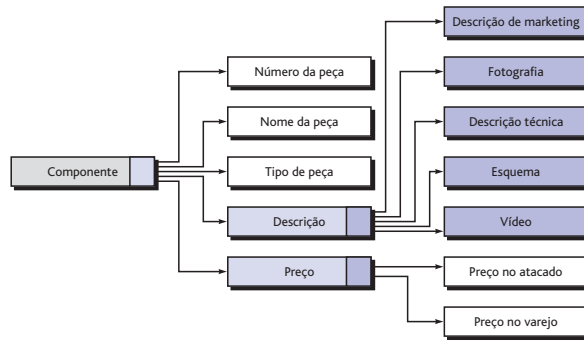


FIGURA 11.5 Árvore de dados para um componente de www.casaseguragarantida.com.

O modelo de conteúdo deve ser capaz de descrever o objeto de conteúdo **Componente**. Em muitos casos, uma simples lista dos objetos de conteúdo, com uma breve descrição de cada objeto, é suficiente para definir os requisitos para o conteúdo a ser projetado e implementado. Entretanto, em alguns casos, o modelo de conteúdo pode se beneficiar de uma análise mais rica que ilustre graficamente as relações entre os objetos de conteúdo e/ou a hierarquia do conteúdo mantido por uma WebApp.

Por exemplo, consideremos a *árvore de dados* [Sri01] criada para um componente de www.casaseguragarantida.com ilustrado na Figura 11.5. A árvore representa uma hierarquia de informações usadas para descrever um componente. Dados simples ou compostos (um ou mais valores de dados) são representados com retângulos de fundo branco. Os objetos de conteúdo são representados como retângulos sombreados. Na figura, a descrição é definida por cinco objetos de conteúdo (os retângulos sombreados). Em alguns casos, um ou mais desses objetos seriam ainda mais refinados, à medida que a árvore de dados fosse se expandindo.

Uma árvore de dados pode ser criada para qualquer conteúdo composto de vários objetos de conteúdo e dados. A árvore de dados é desenvolvida em uma tentativa de definir relações hierárquicas entre os objetos de conteúdo e de fornecer um meio para revisar o conteúdo de modo que omissões e inconsistências sejam reveladas antes de o projeto começar. Além disso, a árvore de dados serve como base para o projeto do conteúdo.

11.5.5 Modelo de interação para WebApps e aplicativos móveis

A grande maioria das WebApps e aplicativos móveis possibilita um “diálogo” entre o usuário e a funcionalidade, o conteúdo e o comportamento de uma aplicação. Esse diálogo pode ser descrito por meio de um modelo de *interações* que pode ser composto de um ou mais dos seguintes elementos: (1) casos de uso, (2) diagramas de sequência, (3) diagramas de estados¹⁰ e/ou (4) protótipos de interfaces do usuário.

Muitas vezes, um conjunto de casos de uso¹¹ já basta para descrever a interação em um nível de análise (um maior refinamento e detalhes serão introdu-

¹⁰ Diagramas de sequência e de estado são modelados usando-se a notação da UML.

¹¹ Casos de uso são descritos em detalhes no Capítulo 9.

zidos durante a fase de projeto). Entretanto, quando a sequência de interação for complexa e envolver várias classes de análise ou muitas tarefas, às vezes vale a pena representá-la usando uma forma esquemática mais rigorosa.

O layout da interface do usuário, o conteúdo que ela apresenta, os mecanismos de interação que implementa e a estética geral da conexão do usuário com a aplicação têm muito a ver com a satisfação do usuário e com o êxito geral da aplicação. Embora seja possível argumentar que a criação de um protótipo de interface do usuário seja uma atividade de projeto, é uma boa ideia realizá-la durante a criação do modelo de análise. O quanto antes uma representação física de uma interface do usuário puder ser revisada, maior será a probabilidade de que os usuários obterão aquilo que realmente desejam. O projeto de interfaces do usuário é discutido em detalhes no Capítulo 15.

Como as ferramentas para construção de WebApps e aplicativos móveis são muitas, relativamente baratas e funcionalmente poderosas, é melhor criar o protótipo de interface utilizando-as. O protótipo deve implementar os principais links de navegação e representar o layout geral da tela em grande parte da mesma forma que será construído. Por exemplo, se o intuito é oferecer cinco funções principais de sistema para o usuário, o protótipo deve representá-las, já que o usuário vai vê-las assim que entrar na aplicação. Serão fornecidos links gráficos? Onde será exibido o menu de navegação? Que outras informações o usuário verá? Perguntas como essas devem ser respondidas pelo protótipo.

11.5.6 Modelo funcional

Muitas WebApps oferecem uma ampla gama de funções computacionais e manipuladoras que podem ser associadas diretamente ao conteúdo (seja usando-o, seja produzindo-o) e que frequentemente são sua meta principal de interação com o usuário. Os aplicativos móveis tendem a ser mais específicos e a fornecer um conjunto mais limitado de funções computacionais e de manipulação. Independentemente da quantidade de funcionalidade, os requisitos funcionais devem ser analisados e, quando necessário, modelados.

O *modelo funcional* lida com dois elementos de processamento da aplicação, cada um representando um nível diferente de abstração procedural: (1) funcionalidade observável pelo usuário, fornecida pela aplicação aos usuários e (2) as operações contidas nas classes de análise que implementam comportamentos associados à classe.

A funcionalidade observável pelos usuários engloba quaisquer funções de processamento iniciadas diretamente por eles. Por exemplo, um aplicativo móvel financeiro poderia implementar uma variedade de funções financeiras (por exemplo, cálculo de pagamento de hipoteca). Essas funções poderiam, na verdade, ser implementadas usando-se operações dentro das classes de análise; porém, do ponto de vista do usuário, a função (mais precisamente, os dados fornecidos pela função) é o resultado visível.

Em um nível de abstração procedural mais baixo, o modelo de requisitos descreve o processamento a ser realizado pelas operações das classes de análise. Essas operações manipulam atributos de classes e estão envolvidas, já que as classes colaboram entre si para cumprir determinado comportamento exigido.

Independentemente do nível de abstração procedural, o diagrama de atividades UML pode ser utilizado para representar detalhes do processamento. No nível de análise, os diagramas de atividades devem ser usados apenas onde a funcionalidade é relativamente complexa. Grande parte da complexidade de muitas WebApps e aplicativos móveis ocorre não na funcionalidade fornecida, mas na natureza das informações que podem ser acessadas e nas maneiras pelas quais podem ser manipuladas.

Um exemplo de funcionalidade relativamente complexa para **www.casaseguragarantida.com** é tratado por um caso de uso intitulado *Obter recomendações para a disposição dos sensores no espaço disponível*. O usuário já desenvolveu um layout para o espaço a ser monitorado e, nesse caso de uso, escolhe esse layout e solicita posições recomendadas para os sensores de acordo com o layout. **www.casaseguragarantida.com** responde com uma representação gráfica do layout, com informações adicionais sobre os pontos recomendados para posicionamento dos sensores. A interação é bastante simples, o conteúdo é ligeiramente mais complexo, porém a funcionalidade subjacente é muito sofisticada. O sistema deve empreender uma análise relativamente complexa do layout do recinto para determinar o conjunto ótimo de sensores. Ele tem de examinar as dimensões dos ambientes, a posição das portas e janelas e coordená-las com as capacidades e especificações dos sensores. Tarefa nada fácil! Um conjunto de diagramas de atividades pode ser usado para descrever o processamento para esse caso de uso.

O segundo exemplo é o caso de uso *Controlar câmeras*. Nele, a interação é relativamente simples, porém há grande risco de funcionalidade complexa, dado que essa operação “simples” exige comunicação complexa com os dispositivos localizados remotamente e acessíveis via Internet. Outra possível complicação está relacionada com a negociação do controle, quando várias pessoas autorizadas tentam monitorar e/ou controlar um único sensor ao mesmo tempo.

A Figura 11.6 ilustra um diagrama de atividades para a operação *assumirControleDaCamera()* que faz parte da classe de análise **Câmera** usada no caso de uso *Controlar câmeras*. Deve-se notar que duas operações adicionais são chamadas dentro do fluxo procedural: *solicitarBloqueioCamera()*, que tenta bloquear a câmera para esse usuário, e *obterUsuarioCameraAtual()*, que recupera o nome do usuário que está controlando a câmera no momento. Os detalhes construtivos indicando como essas operações são chamadas, bem como os detalhes da interface para cada operação, não são considerados até que o projeto da WebApp seja iniciado.

Uma ampliação da funcionalidade da WebApp *CasaSegura* poderia ocorrer com o desenvolvimento de um aplicativo móvel que desse acesso ao sistema *CasaSegura* a partir de um smartphone ou tablet. Os requisitos de conteúdo e funcionais de um aplicativo móvel para o *CasaSegura* poderiam ser semelhantes a um subconjunto daqueles fornecidos pela WebApp, mas requisitos de interface e segurança específicos precisariam ser estabelecidos.

11.5.7 Modelo de configuração para WebApps

Em alguns casos, o modelo de configuração nada mais é do que uma lista de atributos no servidor e no cliente. Entretanto, para aplicações mais comple-

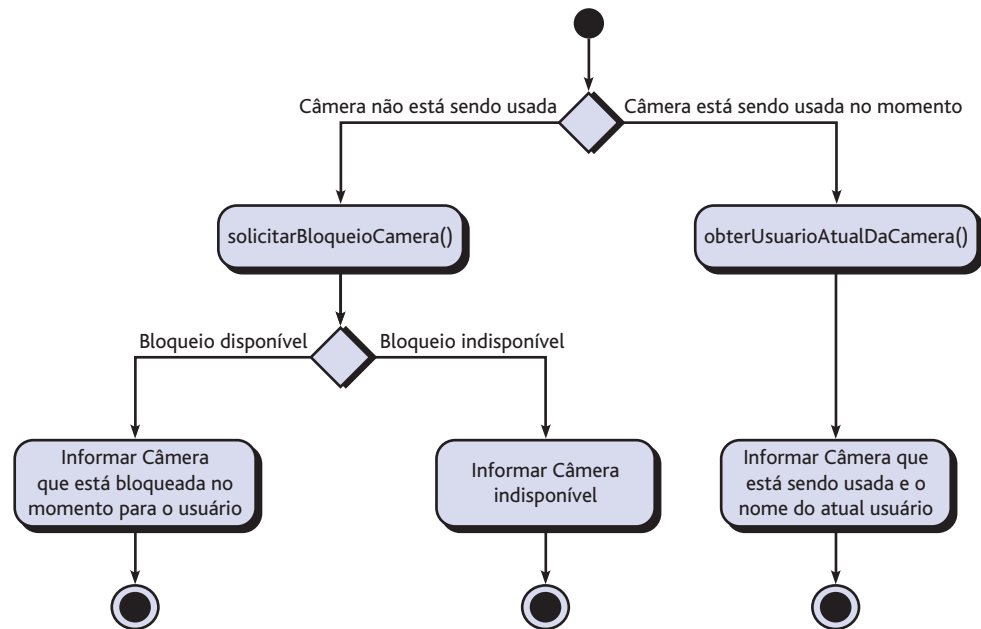


FIGURA 11.6 Diagrama de atividades para a operação `assumirControleDaCamera()`.

xas, uma série de detalhes de configuração (por exemplo, distribuição da carga entre vários servidores, arquiteturas de caching, bancos de dados remotos, vários servidores atendendo a vários objetos) poderia ter um impacto na análise e no projeto. O *diagrama de implantação* em UML pode ser utilizado em situações em que complexas arquiteturas de configuração têm de ser consideradas.

Para **www.casaseguragarantida.com**, a funcionalidade e conteúdo público devem ser especificados como acessíveis a todos os principais clientes Web (isto é, aqueles com 1% de participação no mercado ou mais). Ao contrário, pode ser aceitável restringir a funcionalidade de monitoramento e controle mais complexa (que poderia ser acessada somente pelos usuários **Proprietário**) para um conjunto de clientes menor. Para um aplicativo móvel, a implementação poderia se limitar aos três principais sistemas operacionais móveis. O modelo de configuração para **www.casaseguragarantida.com** também especificará a interoperabilidade com aplicações de monitoramento e bancos de dados de produtos existentes.

11.5.8 Modelo de navegação

Na maioria dos aplicativos móveis residentes em plataformas de smartphone, em geral a navegação fica restrita a listas de botões e menus baseados em ícones relativamente simples. Além disso, a profundidade de navegação (isto é, o número de níveis na hierarquia de hipermídia) é relativamente baixa. Por esses motivos, a modelagem da navegação é relativamente simples.

Para WebApps e para um crescente número de aplicativos móveis para tablets, a modelagem da navegação é mais complexa e muitas vezes considera como cada categoria de usuário vai navegar de um elemento (por exemplo, objeto de conteúdo) a outro da WebApp. A mecânica de navegação é definida

como parte do projeto. Nesse estágio, devemos nos concentrar nos requisitos gerais de navegação. As seguintes perguntas devem ser consideradas:

- Devem certos elementos ser mais fáceis de ser acessados (exigir um número de passos de navegação menor) do que outros? Qual a prioridade para a apresentação?
- Devem certos elementos ser enfatizados para forçar os usuários a navegar em sua direção?
- Como os erros de navegação devem ser tratados?
- Deve a navegação para grupos de elementos relacionados ter maior prioridade do que a navegação para um elemento específico?
- A navegação deve ser obtida via links, via acesso baseado em pesquisa ou por outros meios?
- Devem certos elementos ser apresentados aos usuários no contexto de ações de navegação prévias?
- Deve o log de navegação ser mantido para os usuários?
- Deve existir um menu ou mapa de navegação completo (em vez de um simples link “de retorno” ou ponteiro indicando direção) em qualquer ponto da interação de um usuário?
- Deve o projeto de navegação ser dirigido pelos comportamentos de usuário mais comumente esperados ou pela importância percebida dos elementos definidos da WebApp?
- Pode um usuário “armazenar” sua navegação prévia pela WebApp para agilizar seu uso futuro?
- Para qual categoria de usuário a navegação otimizada deve ser desenvolvida?
- Como os links externos à WebApp devem ser tratados? Sobrepondo a janela do navegador existente? Como uma nova janela do navegador? Como um quadro separado?

Essas e muitas outras perguntas devem ser feitas e respondidas como parte da análise de navegação.

Você e outros envolvidos também devem determinar os requisitos gerais da navegação. Por exemplo, será fornecido um “mapa do site” para dar aos usuários uma visão geral de toda a estrutura da WebApp? Será possível um usuário fazer um “tour guiado” que destacará os elementos mais importantes (objetos de conteúdo e funções) disponíveis? Um usuário será capaz de acessar objetos de conteúdo ou funções baseadas em atributos definidos daqueles elementos (por exemplo, um usuário poderá acessar todas as fotografias de determinado prédio ou a todas as funções que possibilitem o cálculo de peso)?

11.6 Resumo

A modelagem comportamental durante a análise de requisitos representa o comportamento dinâmico do software. O modelo comportamental usa entrada de elementos baseados em cenários ou em classes para representar os

estados das classes de análise e do sistema como um todo. Para tanto, são identificados os estados, são definidos os eventos que fazem uma classe (ou o sistema) passar por uma transição de um estado para outro e também são identificadas as ações que ocorrem à medida que acontece a transição. Os diagramas de estados e os diagramas de sequência são a notação utilizada para modelagem comportamental.

Os padrões de análise permitem que o engenheiro de software use conhecimento do domínio existente para facilitar a criação de um modelo de requisitos. Um padrão de análise descreve uma função ou recurso de software específico que pode ser descrito por meio de um conjunto de casos de uso coerente. Ele especifica o intuito do padrão, o motivo para seu emprego, restrições que limitam seu uso, sua aplicabilidade em vários domínios de problemas, a estrutura geral do padrão, seu comportamento e colaborações, bem como outras informações complementares.

A modelagem de requisitos para aplicativos móveis e WebApps pode usar a maioria (se não todos) dos elementos de modelagem discutidos neste livro. Entretanto, tais elementos são aplicados em um conjunto de modelos especializados que tratam o conteúdo, interação, função, navegação e a configuração cliente/servidor em que o aplicativo móvel ou a WebApp reside.

Problemas e pontos a ponderar

- 11.1 Existem dois tipos de “estados” que os modelos comportamentais são capazes de representar. Quais são eles?
- 11.2 Como um diagrama de sequência difere de um diagrama de estado? Em que são similares?
- 11.3 Sugira três padrões de requisitos para um celular moderno e redija uma breve descrição de cada um deles. Podem esses padrões ser utilizados para outros dispositivos? Cite um exemplo.
- 11.4 Escolha um dos padrões que você desenvolveu no Problema 11.3 e faça uma descrição de padrão relativamente completa, similar em conteúdo e estilo àquela apresentada na Seção 11.4.2.
- 11.5 Que nível de modelagem de análise você acredita que seria necessária para **www.casaseguragarantida.com**? Seria necessário cada um dos tipos de modelos descritos na Seção 11.5.3?
- 11.6 Qual o objetivo do modelo de interações para uma WebApp?
- 11.7 Pode-se afirmar que um modelo funcional para WebApp deve ser postergado até a fase de projeto. Apresente os prós e os contras desse argumento.
- 11.8 Qual o objetivo de um modelo de configuração?
- 11.9 Em que o modelo de navegação difere do modelo de interações?

Leituras e fontes de informação complementares

A modelagem comportamental apresenta uma visão dinâmica importante do comportamento de um sistema. Livros de Samek (*Practical UML Statecharts in C/C++: Event Driven Programming for Embedded Systems*, CRC Press, 2008), Wagner e seus colegas (*Modeling Software with Finite State Machines: A Practical Approach*, Auerbach, 2006) e Boerger e Staerk (*Abstract State Machines*, Springer, 2003) apresentam uma discussão abrangente sobre diagramas de estado e outras representações comportamentais. Gomes e Fernandez (*Behavioral Modeling for Embedded Systems and Technologies*, Information Science Reference, 2009) editaram uma antologia que trata de técnicas de modelagem comportamental para sistemas embarcados.

A maioria dos livros escritos que abordam o tema padrões de software enfoca o projeto de software. Contudo, livros de Vaughn (*Implementing Domain-Driven Design*, Addison-Wesley, 2013), Whithall (*Software Requirement Patterns*, Microsoft Press, 2007), Evans (*Domain-Driven Design*, Addison-Wesley, 2003) e Fowler ([Fow03] e [Fow97]) tratam especificamente de padrões de análise.

Um tratado aprofundado sobre a modelagem de análise para WebApps é apresentado por Pressman e Lowe [Pre08]. Livros de Rossi e seus colegas (*Web Engineering: Modeling and Implementing Web Applications*, Springer, 2010) e Neil (*Mobile Design Pattern Gallery: UI Patterns*, O'Reilly, 2012) discutem o uso de padrões no desenvolvimento de aplicativos. Artigos contidos em uma antologia editada por Murugesan e Desphande (*Web Engineering: Managing Diversity and Complexity of Web Application Development*, Springer, 2001) discutem vários aspectos dos requisitos de uma WebApp. Além desses, o *Proceedings of the International Conference on Web Engineering*, publicado anualmente, trata regularmente de questões referentes à modelagem de requisitos.

Uma ampla gama de fontes de informação sobre modelagem de requisitos se encontra à disposição na Internet. Uma lista atualizada de referências relevantes (em inglês) para a modelagem de análise pode ser encontrada no site: www.mhhe.com/pressman.

12

Conceitos de projeto

Conceitos-chave

abstração	232
arquitetura	232
aspectos	237
atributos de qualidade	229
bom projeto	228
coesão	236
diretrizes de qualidade	228
encapsulamento	
de informações	235
independência	
funcional	236
modularidade	234

O projeto de software* abrange o conjunto de princípios, conceitos e práticas que levam ao desenvolvimento de um sistema ou produto de alta qualidade. Os princípios de projeto estabelecem uma filosofia que guia o trabalho que você deve desempenhar. Os conceitos de projeto devem ser entendidos antes que a mecânica da prática de projeto possa ser aplicada, e a prática de projeto em si conduz à criação de várias representações do software que servem como um guia para a atividade de construção que se segue.

A atividade de projeto é crucial para uma engenharia de software bem sucedida. No início dos anos 1990, Mitch Kapor, o criador do Lotus 1-2-3, apresentou um “manifesto do projeto de software” no *Dr. Dobbs Journal*. Ele escreveu:

PANORAMA

O que é? Projeto é o que quase todo engenheiro quer fazer. É o lugar onde a criatividade impera — onde os requisitos dos envolvidos, as necessidades da aplicação e as considerações técnicas se juntam na formulação de um produto ou sistema. O projeto cria uma representação ou modelo do software, mas, diferentemente do modelo de requisitos (que se concentra na descrição do “que” é para ser feito: dos dados, função e comportamento necessários), o modelo de projeto indica “como” fazer, fornecendo detalhes sobre a arquitetura de software, estruturas de dados, interfaces e componentes fundamentais para implementar o sistema.

Quem realiza? Os engenheiros de software conduzem cada uma das tarefas de projeto.

Por que é importante? O projeto permite que se modele o sistema ou produto a ser construído. O modelo pode ser avaliado em termos de qualidade e aperfeiçoado antes de o código ser gerado, ou de os testes serem realizados, ou de os usuários se envolverem em grande número. Projeto é o lugar onde a qualidade do software é estabelecida.

Quais são as etapas envolvidas? O projeto representa o software de várias formas diferentes. Primeiramente, a arquitetura do sistema ou do produto tem de ser representada. Em seguida, são modeladas as interfaces que conectam o software aos usuários, a outros sistemas e a dispositivos, bem como a seus próprios componentes internos. Por fim, os componentes de software usados para construir o sistema são projetados. Cada uma dessas visões representa uma ação de projeto diferente, mas todas devem estar de acordo com um conjunto de conceitos básicos de projeto que orientam o trabalho de projeto de software.

Qual é o artefato? Um modelo de projeto que engloba representações de arquitetura, de interface, em nível de componentes e de utilização, é o principal artefato gerado durante o projeto de software.

Como garantir que o trabalho foi realizado corretamente? O modelo de projeto é avaliado pela equipe de software em um esforço para determinar se ele contém erros, inconsistências ou omissões, se existem alternativas melhores e se o modelo pode ser implementado de acordo com as restrições, prazo e orçamento estabelecidos.

* N. de R.T.: O termo para “projeto” em inglês é “design”. Ele se refere à atividade de engenharia cujo foco é definir “como” os requisitos estabelecidos do projeto devem ser implementados no software. É uma fase que se apresenta de maneira similar nas diversas especializações da engenharia, como civil, naval, química e mecânica.

O que é projeto? É onde você fica com um pé em dois mundos – o mundo da tecnologia e o mundo das pessoas e dos propósitos do ser humano –, e você tenta unir os dois... O crítico da arquitetura romano Vitruvius lançou a noção de que prédios bem projetados eram aqueles que apresentavam solidez, comodidade e deleite. O mesmo poderia ser dito em relação a software de boa qualidade. *Solidez*: um programa não deve apresentar nenhum bug que impeça seu funcionamento. *Comodidade*: um programa deve ser adequado aos propósitos para os quais foi planejado. *Deleite*: a experiência de usar o programa deve ser prazerosa. Temos aqui os princípios de uma teoria de projeto de software.

padrões.....	233
processo de projeto	228
projeto de dados.....	244
projeto de software	230
projeto orientado a objetos	238
refatoração	238
refinamento gradual....	237
separação por interesses.....	234

O objetivo da atividade de projetar é gerar um modelo ou representação que apresente solidez, comodidade e deleite. Para tanto, temos de praticar a diversificação e, depois, a convergência. Belady [Bel81] afirma que “diversificação é a aquisição de um repertório de alternativas, a matéria-prima do projeto: componentes, soluções de componentes e conhecimento, todos contidos em catálogos, livros-textos e na mente”. Assim que esse conjunto diversificado de informações for montado, temos de escolher elementos do repertório que atendam aos requisitos definidos pela engenharia de requisitos e pelo modelo de análise (Capítulos 8 a 11). À medida que isso ocorre, são consideradas e rejeitadas alternativas e convergimos para “uma configuração particular de componentes e, portanto, para a criação do produto final” [Bel81].

Diversificação e convergência combinam intuição e julgamento baseado na experiência de construção de entidades similares, um conjunto de princípios e/ou heurística que orientam a maneira como o modelo evolui, um conjunto de critérios que permitem avaliar a qualidade e um processo de iteração que, ao fim, leva a uma representação final do projeto.

O projeto de software muda continuamente à medida que novos métodos, melhor análise e entendimento mais abrangente evoluem.¹ Mesmo hoje em dia, a maioria das metodologias de projeto de software carece da profundidade, flexibilidade e natureza quantitativa que normalmente estão associadas às disciplinas mais clássicas de engenharia de projeto. Entretanto, existem efetivamente métodos para projeto de software, critérios para qualidade de projeto estão disponíveis e notação de projeto pode ser aplicada. Neste capítulo, exploraremos os conceitos e princípios fundamentais aplicáveis a todos os projetos de software, os elementos do modelo de projeto e o impacto dos padrões no processo de projeto. Nos Capítulos 12 a 18 apresentaremos uma série de métodos de projeto de software à medida que são aplicados ao projeto da arquitetura, de interfaces e de componentes, bem como as metodologias de projeto baseado em padrões e orientado para a Web.

“O milagre mais comum da engenharia de software é a transição da análise para o projeto e do projeto para o código.”

Richard Dué

12.1 Projeto no contexto da engenharia de software

O projeto de software está no núcleo técnico da engenharia de software e é aplicado seja qual for o modelo de processos de software utilizado. Iniciando

¹ Os leitores curiosos sobre a filosofia do projeto de software talvez se interessem pela intrigante discussão de Philippe Kruchten a respeito do projeto “pós-moderno” [Kru05].

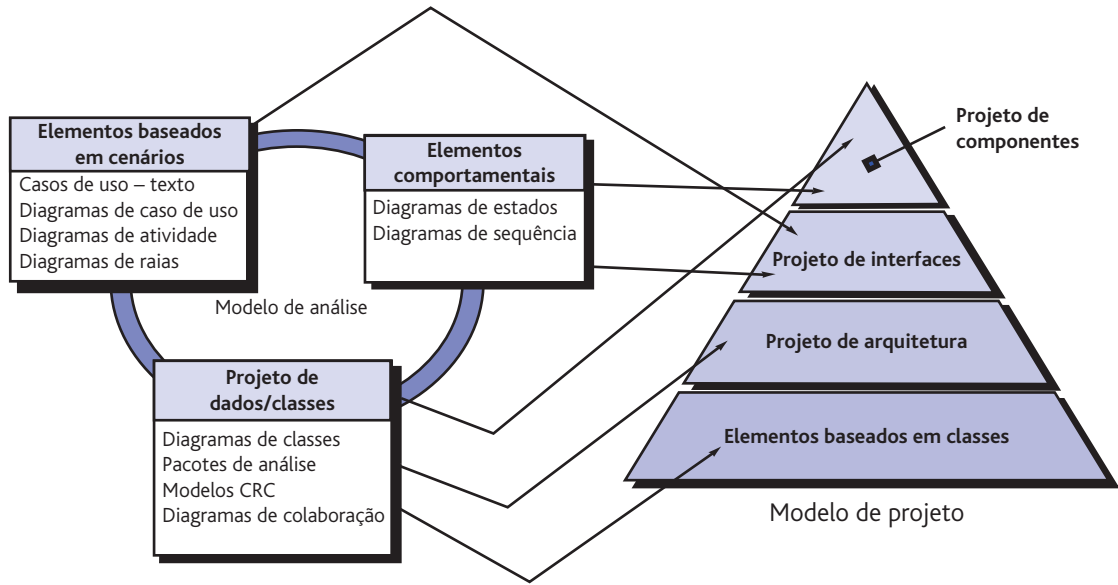


FIGURA 12.1 Transformando o modelo de requisitos no modelo de projeto.

assim que os requisitos de software tiverem sido analisados e modelados, o projeto de software é a última ação da engenharia de software na atividade de modelagem e prepara o cenário para a **construção** (geração de código e testes).

Cada elemento do modelo de requisitos (Capítulos 9 a 11) fornece as informações necessárias para criar os quatro modelos de projeto exigidos por uma especificação completa. O fluxo de informações durante o projeto de software está ilustrado na Figura 12.1. O modelo de requisitos, manifestado por elementos baseados em cenários, baseados em classes, orientado a fluxos e comportamentais, alimenta a tarefa de projeto. Usando a notação de projeto e os métodos de projeto discutidos em capítulos posteriores, são gerados um projeto de dados/classes, um projeto de arquitetura, um projeto de interfaces e um projeto de componentes.

O projeto de dados/classes transforma os modelos de classes (Capítulo 10) em realizações de classes de projeto e nas estruturas de dados dos requisitos necessárias para implementar o software. Os objetos e as relações definidos nos cartões CRC e no conteúdo detalhado dos dados representados por atributos de classes e outra notação fornecem a base para a realização do projeto de dados. Parte do projeto de classes pode ocorrer com o projeto da arquitetura de software. O projeto de classe mais detalhado ocorre à medida que cada componente de software é projetado.

O projeto de arquitetura define as relações entre os principais elementos estruturais do software, os estilos arquiteturais e padrões de projeto (Capítulo 13) que podem ser usados para satisfazer os requisitos definidos para o sistema e as restrições que afetam o modo como a arquitetura pode ser implementada [Sha96]. A representação do projeto de arquitetura – a organização da solução técnica de um sistema baseado em computador – é derivada do modelo de requisitos.

O projeto de software sempre deve começar levando em consideração os dados – a base para todos os demais elementos do projeto. Depois de estabelecida a base, a arquitetura tem de ser extraída. Só então devem se realizar outras tarefas de projeto.

O projeto de interfaces descreve como o software se comunica com sistemas que operam em conjunto e com as pessoas que o utilizam. Uma interface implica fluxo de informações (por exemplo, dados e/ou controle) e em um tipo de comportamento específico. Consequentemente, modelos comportamentais e de cenários de uso fornecem grande parte das informações necessárias para o projeto de interfaces.

O projeto de componentes transforma elementos estruturais da arquitetura de software em uma descrição procedural dos componentes de software. As informações obtidas dos modelos baseados em classes e dos modelos comportamentais servem como base para o projeto de componentes.

Durante o projeto, tomamos decisões que, em última análise, afetarão o sucesso da construção do software e, igualmente importante, a facilidade de manutenção do software. Mas por que o projeto é tão importante?

A importância do projeto de software pode ser definida em uma única palavra – *qualidade*. Projeto é a etapa em que a qualidade é incorporada à engenharia de software. O projeto nos fornece representações do software que podem ser avaliadas em termos de qualidade. Projeto é a única maneira pela qual podemos transformar precisamente os requisitos dos envolvidos em um produto ou sistema de software finalizado. O projeto de software serve como base para todas as atividades de apoio e da engenharia de software que se seguem. Sem um projeto, corremos o risco de construir um sistema instável – um sistema que falhará quando forem feitas pequenas alterações, um sistema que talvez seja difícil de ser testado, um sistema cuja qualidade não pode ser avaliada até uma fase avançada do processo de software, quando o tempo está se esgotando e muito dinheiro já foi gasto.

"Há duas maneiras de construir um projeto de software. Uma delas é fazê-lo tão simples que, obviamente, não existirão deficiências; a outra maneira é fazê-lo tão complicado que não há nenhuma deficiência óbvia. O primeiro método é bem mais difícil".

C. A. R. Hoare



Projeto versus codificação

Cena: Sala do Ed, enquanto a equipe se prepara para transformar os requisitos em projeto.

Atores: Jamie, Vinod e Ed – todos eles membros da equipe de engenharia de software do CasaSegura.

Conversa:

Jamie: Sabe, Doug [o gerente da equipe] está obcecado pelo projeto. Tenho de ser honesto: o que eu realmente adoro fazer é programar. Dê-me C++ ou Java e ficarei feliz.

Ed: Que nada... você gosta de projetar.

Jamie: Você não está me ouvindo, programar é o canal.

Vinod: Acredito que o que o Ed quis dizer é que, na verdade, você não gosta de programar; você gosta de projetar e expressar isso em forma de código de programa. Código é a linguagem que você usa para representar um projeto.

Jamie: E o que há de errado nisso?

Vinod: Nível de abstração.

Jamie: Hã?

Ed: Uma linguagem de programação é boa para representar detalhes, como estruturas de dados e algoritmos, mas não é tão boa para representar a colaboração componente-componente ou a arquitetura... Coisas do tipo.

Vinod: E uma arquitetura desordenada pode arruinar até mesmo o melhor código.

Jamie (pensando por uns instantes): Então, você está dizendo que não posso representar arquitetura no código... Isso não é verdade.

Vinod: Certamente você pode envolver arquitetura no código, mas, na maioria das linguagens de programação, é bem difícil ter uma visão geral rápida da arquitetura examinando-se o código.

Ed: E é isso que queremos antes de começar a programar.

Jamie: Certo, talvez projetar e programar sejam coisas diferentes, mas ainda assim prefiro programar.

CASASEGURA

12.2 O processo de projeto

O projeto de software é um processo iterativo por meio do qual os requisitos são traduzidos em uma “planta” para a construção do software. Inicialmente, a planta representa uma visão holística do software. O projeto é representado em uma abstração de alto nível – um nível que pode ser associado diretamente ao objetivo específico do sistema e aos requisitos mais detalhados de dados, funcionalidade e comportamento. À medida que ocorrem as iterações do projeto, o refinamento subsequente leva a representações do projeto em níveis de abstração cada vez mais baixos. Estes ainda podem ser associados aos requisitos, mas a conexão é mais sutil.

12.2.1 Diretrizes e atributos da qualidade de software

“Escrever um trecho de código inteligente que funcione é uma coisa; projetar algo que possa dar suporte a negócios duradouros é outra totalmente diferente.”

C. Ferguson

Ao longo do processo de projeto, a qualidade do projeto que evolui é avaliada com uma série de revisões técnicas, discutidas no Capítulo 20. McGlaughlin [McG91] sugere três características que servem como guia para a avaliação de um bom projeto:

- O projeto deve implementar todos os requisitos explícitos contidos no modelo de requisitos e deve acomodar todos os requisitos implícitos desejados pelos envolvidos.
- O projeto deve ser um guia legível e compreensível para aqueles que geram código e para aqueles que testam e, subsequentemente, dão suporte ao software.
- O projeto deve dar uma visão completa do software, tratando os domínios de dados, funcional e comportamental do ponto de vista da implementação.

Cada uma dessas características é, na verdade, uma meta do processo de projeto. Mas como cada uma é alcançada?

Diretrizes de qualidade. Para avaliar a qualidade da representação de um projeto, você e outros membros da equipe de software devem estabelecer critérios técnicos para um bom projeto. Na Seção 12.3, discutimos conceitos de projeto que também servem como critérios de qualidade de software. Por enquanto, consideremos as seguintes diretrizes:

1. Um projeto deve exibir uma arquitetura que (1) foi criada usando estilos ou padrões de arquitetura reconhecíveis, (2) seja composta por componentes que apresentam boas características de projeto (discutidas mais adiante neste capítulo) e (3) possa ser implementada de uma forma evolucionária², facilitando, portanto, a implementação e os testes.
2. Um projeto deve ser modular; ou seja, o software deve ser dividido logicamente em elementos ou subsistemas, de modo que seja fácil de testar e manter.
3. Um projeto deve conter representações distintas de: dados, arquitetura, interfaces e componentes.

Quais são as características de um bom projeto?

“Projetar não é apenas o que parece e se tem vontade. Projetar é como a coisa funciona.”

Steve Jobs

² Para sistemas menores, algumas vezes o projeto pode ser desenvolvido linearmente.

4. Um projeto deve levar a estruturas de dados adequadas às classes a ser implementadas e baseadas em padrões de dados reconhecíveis.
5. Um projeto deve levar a componentes que apresentem características funcionais independentes (baixo acoplamento).
6. Um projeto deve levar a interfaces que reduzam a complexidade das conexões entre os componentes e o ambiente externo (encapsulamento).
7. Um projeto deve ser obtido usando-se um método repetível, isto é, dirigido por informações obtidas durante a análise de requisitos de software.
8. Um projeto deve ser representado usando-se uma notação que comunique seu significado eficientemente.

Essas diretrizes não são atingidas por acaso. Elas são alcançáveis por meio da aplicação de princípios de projeto fundamentais, de metodologia sistemática e de revisão.

Atributos de qualidade. A Hewlett-Packard [Gra87] desenvolveu um conjunto de atributos de qualidade de software ao qual foi atribuído o acrônimo FURPS: *functionality* (funcionalidade), *usability* (usabilidade), *reliability* (confiabilidade), *performance* (desempenho) e *supportability* (facilidade de suporte). Os atributos de qualidade FURPS representam uma meta para todo projeto de software:

- A *funcionalidade* é avaliada pela observação do conjunto de características e capacidades do programa, a generalidade das funções entregues e a segurança do sistema como um todo.

“Qualidade não é algo que se coloque sobre os assuntos e objetos como um enfeite em uma árvore de Natal.”

Robert Pirsig

INFORMAÇÕES



Avaliação da qualidade do projeto – a revisão técnica

O projeto é importante porque permite à equipe de software avaliar a qualidade³ do software antes de ser implementado – em um momento em que é fácil e barato corrigir erros, omissões ou inconsistências. Mas como avaliar a qualidade durante um projeto? O software não pode ser testado, pois não existe nenhum software executável para testar. O que fazer?

Durante um projeto, a qualidade é avaliada realizando-se uma série de revisões técnicas (*technical reviews*, TRs). As TRs são discutidas em detalhes no Capítulo 20⁴, mas vale fazer um resumo neste momento. A revisão técnica é uma reunião conduzida por membros da equipe de software. Normalmente duas, três ou quatro pessoas parti-

cipam, dependendo do escopo das informações de projeto a ser revisadas. Cada pessoa desempenha um papel: um *líder de revisão* planeja a reunião, estabelece uma agenda e conduz a reunião; o *registrador* toma notas de modo que nada seja perdido; o *produtor* é a pessoa cujo artefato (por exemplo, o projeto de um componente de software) está sendo revisado. Antes de uma reunião, cada pessoa da equipe de revisão recebe uma cópia do artefato do projeto para ler, procurando encontrar erros, omissões ou ambiguidades. Quando a reunião começa, o intuito é perceber todos os problemas do artefato, de modo que possam ser corrigidos antes da implementação começar. A TR dura, normalmente, entre 60 e 90 minutos. Na conclusão, a equipe de revisão determina se outras ações são necessárias por parte do produtor antes de o artefato do projeto ser aprovado como parte do modelo de projeto final.

³ Os fatores de qualidade discutidos no Capítulo 30 podem ajudar a equipe de revisão à medida que avalia a qualidade.

⁴ Pense na possibilidade de consultar o Capítulo 20 neste momento. As revisões técnicas são parte crítica do processo de projeto e um importante mecanismo para atingir qualidade em um projeto.

Os projetistas de software tendem a se concentrar no problema a ser resolvido. Não se esqueça de que os atributos de qualidade FURPS sempre fazem parte do problema. Eles têm de ser considerados.

- A *usabilidade* é avaliada considerando-se fatores humanos (Capítulos 6 e 15), estética, consistência e documentação como um todo.
- A *confiabilidade* é avaliada medindo-se a frequência e a gravidade das falhas, a precisão dos resultados gerados, o tempo médio entre defeitos (MTTF, mean-time-to-failure), a capacidade de se recuperar de uma falha e a previsibilidade do programa.
- O *desempenho* é medido usando-se a velocidade de processamento, o tempo de resposta, o consumo de recursos, vazão (*throughput*) e eficiência.
- A *facilidade de suporte* combina a capacidade de estender o programa (extensibilidade), a adaptabilidade e a reparabilidade. Esses três atributos representam um termo mais comum, *facilidade de manutenção* – e, além disso, a facilidade de realizar testes, a compatibilidade, a facilidade de configurar (a habilidade de organizar e controlar elementos da configuração do software, Capítulo 29), a facilidade com a qual um sistema pode ser instalado, bem como a facilidade com a qual os problemas podem ser localizados.

Nem todo atributo de qualidade de software tem o mesmo peso à medida que o projeto é desenvolvido. Uma aplicação poderia enfatizar a funcionalidade com ênfase especial na segurança. Outra poderia demandar desempenho com particular ênfase na velocidade de processamento. Um terceiro enfoque poderia ser na confiabilidade. Independentemente do peso dado, é importante notar que esses atributos de qualidade devem ser considerados quando o projeto se inicia, e *não* após o projeto estar concluído e a construção tiver começado.

12.2.2 A evolução de um projeto de software

"Um projetista sabe que seu projeto atingiu a perfeição não quando não resta mais nada a ser acrescentado, mas quando não há mais nada a ser eliminado."

Antoine de St-Exupéry

A evolução de um projeto de software é um processo contínuo que já atinge mais de seis décadas. Os trabalhos iniciais concentravam-se em critérios para o desenvolvimento de programas modulares [Den73] e de métodos para refinamento das estruturas de software de uma forma *top-down* "estruturada" ([Wir71], [Dah72], [Mil72]). Metodologias de projeto mais recentes (por exemplo, [Jac92], [Gam95]) propuseram uma abordagem orientada a objetos para a derivação do projeto. A ênfase mais recente em projeto de software tem sido na arquitetura de software [Kru06] e nos padrões de projeto que podem ser utilizados para implementar arquiteturas de software e níveis de abstração de projeto mais baixos (por exemplo, [Hol06] [Sha05]). Tem crescido a ênfase em métodos orientados a aspectos (por exemplo, [Cla05], [Jac04]), no desenvolvimento dirigido a modelos [Sch06] e dirigido a testes [Ast04], que enfatizam técnicas para se atingir uma modularidade e uma estrutura arquitetural mais eficiente nos projetos criados.

Uma série de métodos de projetos, decorrentes dos trabalhos citados, está sendo aplicada em toda a indústria. Assim como os métodos de análise apresentados nos Capítulos 9 a 11, cada método de projeto de software introduz heurísticas e notação únicas, bem como uma visão um tanto provinciana daquilo que caracteriza a qualidade de um projeto. Mesmo assim, todos os métodos possuem uma série de características comuns: (1) um mecanismo

Quais características são comuns a todos os métodos de projeto?

para a transformação do modelo de requisitos em uma representação de projeto, (2) uma notação para representar componentes funcionais e suas interfaces, (3) heurística para refinamento e divisão e (4) diretrizes para avaliação da qualidade.

Independentemente do método de projeto utilizado, devemos aplicar um conjunto de conceitos básicos ao projeto de dados, de arquitetura, de interface e de componentes. Tais conceitos são considerados nas seções a seguir.

CONJUNTO DE TAREFAS



Conjunto de tarefas genéricas para projeto

1. Examinar o modelo do domínio de informação e projetar estruturas de dados apropriadas para objetos de dados e seus atributos.
2. Usar o modelo de análise, selecionar um estilo de arquitetura (padrão) apropriado ao software.
3. Dividir o modelo de análise em subsistemas de projeto e alocá-los na arquitetura:
 - Certificar-se de que cada subsistema seja funcionalmente coeso.
 - Projetar interfaces de subsistemas.
 - Alocar classes ou funções de análise para cada subsistema.
4. Criar um conjunto de classes ou componentes de projeto:
 - Transformar a descrição de classes de análise em uma classe de projeto.
 - Verificar cada classe de projeto em relação aos critérios de projeto; considerar questões de herança.
 - Definir métodos e mensagens associadas a cada classe de projeto.
5. Avaliar e selecionar padrões de projeto para uma classe ou um subsistema de projeto.
6. Rever as classes de projeto e revisar quando necessário.
5. Projetar qualquer interface necessária para sistemas ou dispositivos externos.
6. Projetar a interface do usuário:
 - Revisar os resultados da análise de tarefas.
 - Especificar a sequência de ações baseando-se nos cenários de usuário.
 - Criar um modelo comportamental da interface.
 - Definir objetos de interface e mecanismos de controle.
 - Rever o projeto de interfaces e revisar quando necessário.
7. Conduzir o projeto de componentes.
 - Especificar todos os algoritmos em um nível de abstração relativamente baixo.
 - Refinar a interface de cada componente.
 - Definir estruturas de dados dos componentes.
 - Revisar cada componente e corrigir todos os erros descobertos.
8. Desenvolver um modelo de implantação.

12.3 Conceitos de projeto

O conjunto de conceitos fundamentais de projeto de software evoluiu ao longo da história da engenharia de software. Embora o grau de interesse nesses conceitos tenha variado ao longo dos anos, todos resistiram ao tempo. Esses conceitos fornecem ao projetista de software uma base a partir da qual podem ser aplicados métodos de projeto mais sofisticados. Cada um deles ajuda a definir critérios que podem ser usados para dividir o software em componentes individuais, separar os detalhes da estrutura de dados de uma representação conceitual do software e estabelecer critérios uniformes que definam a qualidade técnica de um projeto de software.

M. A. Jackson [Jac75] certa vez disse: “O princípio da sabedoria para um engenheiro de software é reconhecer a diferença entre fazer um programa funcionar e fazer com que ele funcione corretamente”. Nas seções a

seguir, apresentamos uma visão geral dos conceitos de projeto de software fundamentais que fornecem o framework necessário para que “ele funcione corretamente”.

“Abstração é uma das maneiras fundamentais como nós, seres humanos, lidamos com a complexidade.”

Grady Booch

Como projetista, trabalhe arduamente para extrair tanto as abstrações procedurais quanto as de dados que atendam ao problema em questão. Se elas puderem atender a um domínio inteiro dos problemas, melhor ainda.

Uma discussão aprofundada sobre arquitetura de software pode ser encontrada em www.sei.cmu.edu/ata/ata_init.html.

12.3.1 Abstração

Quando se considera uma solução modular para qualquer problema, muitos níveis de abstração podem ser colocados. No nível de abstração mais alto, uma solução é expressa em termos abrangentes, usando a linguagem do domínio do problema. Em níveis de abstração mais baixos, é fornecida uma descrição mais detalhada da solução. A terminologia do domínio do problema é associada à terminologia de implementação para definir uma solução. Por fim, no nível de abstração mais baixo, a solução técnica do software é expressa de maneira que possa ser implementada diretamente.

Conforme diferentes níveis de abstração são alcançados, a combinação de abstrações procedurais e de dados é usada. Uma *abstração procedural* refere-se a uma sequência de instruções que possuem uma função específica e limitada. O nome de uma abstração procedural indica sua função, porém os detalhes específicos são omitidos. Um exemplo de abstração procedural seria a palavra *abrir* para uma porta. *Abrir* implica uma longa sequência de etapas procedurais (por exemplo, dirigir-se até a porta, alcançar e agarrar a maçaneta, girar a maçaneta e puxar a porta, afastar-se da porta em movimento etc.).⁵

A *abstração de dados* é um conjunto de dados com nome que descreve um objeto de dados. No contexto da abstração procedural *abrir*, podemos definir uma abstração de dados chamada **porta**. Assim como qualquer objeto de dados, a abstração de dados para **porta** englobaria um conjunto de atributos que descrevem a porta (por exemplo, tipo de porta, mudar de direção, mecanismo de abertura, peso, dimensões). Daí decorre que a abstração *abrir* faria uso de informações contidas nos atributos da abstração de dados **porta**.

12.3.2 Arquitetura

Arquitetura de software refere-se “à organização geral do software e aos modos pelos quais ela disponibiliza integridade conceitual para um sistema” [Sha95a]. Em sua forma mais simples, arquitetura é a estrutura ou a organização de componentes de programa (módulos), a maneira como esses componentes interagem e a estrutura de dados que são usados pelos componentes. Em um sentido mais amplo, entretanto, os componentes podem ser generalizados para representar os principais elementos de um sistema e suas interações.

Uma das metas do projeto de software é derivar um quadro da arquitetura de um sistema. Esse quadro representa a organização a partir da qual atividades mais detalhadas de projeto são conduzidas. Um conjunto de padrões de arquitetura permite a um engenheiro de software reutilizar conceitos em nível de projeto.

⁵ Deve-se notar, entretanto, que um conjunto de operações pode ser substituído por outro, desde que a função implicada pela abstração procedural permaneça a mesma. Consequentemente, as etapas necessárias para implementar *abrir* mudariam drasticamente se a porta fosse automática e ligada a um sensor.

Shaw e Garlan [Sha95a] descrevem um conjunto de propriedades que devem ser especificadas como parte de um projeto de arquitetura. *Propriedades estruturais* definem “os componentes de um sistema (por exemplo, módulos, objetos, filtros) e a maneira como esses componentes são empacotados e interagem entre si”. *Propriedades extra-funcionais* tratam da maneira como o projeto da arquitetura atinge os requisitos de desempenho, capacidade, confiabilidade, segurança, adaptabilidade e outras características do sistema. *Famílias de sistemas relacionados* “exploram padrões reusáveis comumente encontrados no projeto de famílias de sistemas similares”.

Dada a especificação dessas propriedades, o projeto da arquitetura pode ser representado usando-se um ou mais modelos diferentes [Gar95]. Os *modelos estruturais* representam a arquitetura como um conjunto organizado de componentes de programa. *Modelos de framework* aumentam o nível de abstração do projeto, tentando identificar frameworks (padrões) de projeto de arquitetura reutilizáveis, encontrados em tipos de aplicações similares. Os *modelos dinâmicos* tratam dos aspectos comportamentais da arquitetura do programa, indicando como a estrutura ou configuração do sistema pode mudar em função de eventos externos. Os *modelos de processos* concentram-se no projeto do processo técnico ou do negócio a que o sistema deve atender. Por fim, os *modelos funcionais* podem ser utilizados para representar a hierarquia funcional de um sistema.

Diferentes *linguagens de descrição de arquitetura* (*architectural description languages*, ADLs) foram desenvolvidas para representar esses modelos [Sha95b]. Embora diversas ADLs tenham sido propostas, a maioria fornece mecanismos para descrever componentes de sistema e a maneira pela qual eles estão conectados entre si.

Note que há certo debate em torno do papel da arquitetura no projeto. Alguns pesquisadores argumentam que a obtenção da arquitetura de software deve ser separada do projeto e ocorre entre as ações da engenharia de requisitos e ações de projeto mais convencionais. Outros acreditam que a obtenção da arquitetura é parte do processo de projeto. A maneira como a arquitetura de software é caracterizada e seu papel no projeto são discutidos no Capítulo 13.

“Arquitetura de software é o artefato resultante do desenvolvimento que dá o maior retorno sobre o investimento em relação à qualidade, prazos e custo.”

Len Bass et al.

“Cada padrão descreve um problema que ocorre repetidamente em nosso ambiente e, então, descreve o núcleo da solução para esse problema de forma que podemos usar a solução milhões de vezes sem jamais fazê-lo da mesma forma.”

Christopher Alexander

12.3.3 Padrões

Brad Appleton define *padrão de projeto* da seguinte maneira: “Padrão é parte de um conhecimento consolidado já existente que transmite a essência de uma solução comprovada para um problema recorrente em certo contexto, em meio a preocupações concorrentes” [App00]. Em outras palavras, um padrão de projeto descreve uma estrutura de projeto que resolve uma categoria de problemas de projeto em particular, em um contexto específico e entre “forças” que direcionam a maneira como o padrão é aplicado e utilizado.

O objetivo de cada padrão de projeto é fornecer uma descrição que permita a um projetista determinar (1) se o padrão se aplica ou não ao trabalho em questão, (2) se o padrão pode ou não ser reutilizado (e, portanto, poupar tempo) e (3) se o padrão pode servir como um guia para desenvolver um pa-

drão similar, mas funcional ou estruturalmente diferente. Os padrões de projeto são discutidos de forma detalhada no Capítulo 16.

12.3.4 Separação por interesses (por afinidades)

A *separação por interesses* é um conceito de projeto [Dij82] que sugere que qualquer problema complexo pode ser tratado mais facilmente se for subdividido em trechos a ser resolvidos e/ou otimizados independentemente. *Interesse* se manifesta como uma característica ou comportamento especificado como parte do modelo de requisitos do software. Por meio da separação por interesses em blocos menores e, portanto, mais administráveis, um problema toma menos tempo para ser resolvido.

A complexidade percebida de dois problemas, quando estes são combinados, normalmente é maior do que soma da complexidade percebida quando cada um deles é tomado separadamente. Isso nos leva a uma estratégia dividir-para-conquistar – é mais fácil resolver um problema complexo quando o subdividimos em partes gerenciáveis. Isso tem implicações importantes em relação à modularidade do software.

A separação por interesses se manifesta em outros conceitos relacionados ao projeto: modularidade, aspectos, independência funcional e refinamento. Cada um será discutido nas subseções a seguir.

12.3.5 Modularidade

Modularidade é a manifestação mais comum da separação por interesses. O software é dividido em componentes separadamente especificados e localizáveis, algumas vezes denominados *módulos*, que são integrados para satisfazer os requisitos de um problema.

Já foi dito que “modularidade é o único atributo de software que possibilita um programa ser intelectualmente gerenciável” [Mye78]. Software monolítico (um grande programa composto de um único módulo) não pode ser facilmente entendido por um engenheiro de software. O número de caminhos de controle, abrangência de referências, número de variáveis e complexidade geral tornaria o entendimento quase impossível. Em quase todos os casos, devemos dividir o projeto em vários módulos para facilitar a compreensão e, conseqüentemente, reduzir o custo necessário para construir o software.

Retomando a nossa discussão sobre separação por interesses, é possível concluir que, se subdividirmos o software indefinidamente, o esforço exigido para desenvolvê-lo será muito pequeno! Infelizmente, outros fatores entram em jogo, invalidando essa conclusão. Na Figura 12.2, o esforço (custo) para desenvolver um módulo de software individual realmente diminui à medida que o número total de módulos cresce. Dado o mesmo conjunto de requisitos, mais módulos significa um tamanho individual menor. Entretanto, à medida que o número de módulos aumenta, o esforço (custo) associado à integração dos módulos também cresce. Essas características levam a um custo total ou curva de esforço mostrada na figura. Existe um número M de módulos que resultaria em um custo de desenvolvimento mínimo, porém não temos a sofisticação suficiente para prever M com certeza.

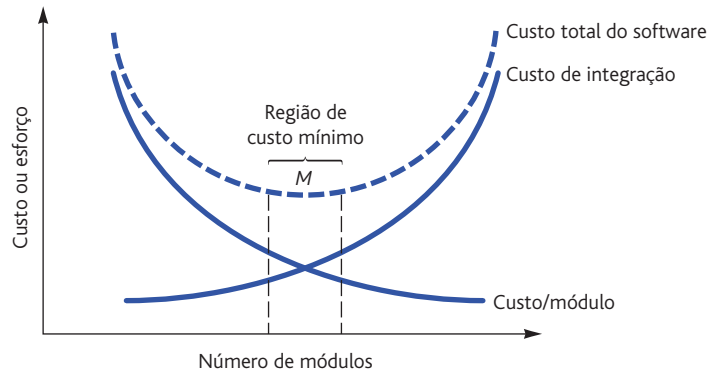


FIGURA 12.2 Modularidade e custo do software.

As curvas da Figura 12.2 nos dão uma útil orientação qualitativa quando a modularidade é considerada. Devemos modularizar, mas tomar cuidado para permanecer nas vizinhanças de M . Devemos evitar modularizar a menos ou a mais. Mas como saber a vizinhança de M ? Quão modular devemos fazer com que um software seja? As respostas dessas perguntas exigem entendimento de outros conceitos de projeto, considerados posteriormente neste capítulo.

Modularizamos um projeto (e o programa resultante) de modo que o desenvolvimento possa ser planejado mais facilmente, incrementos de software possam ser definidos e entregues, as mudanças possam ser mais facilmente acomodadas, os testes e a depuração possam ser conduzidos de forma mais eficaz e a manutenção no longo prazo possa ser realizada sem efeitos colaterais graves.

12.3.6 Encapsulamento* de informações

O conceito de modularidade conduz a uma questão fundamental: “Como decompor uma solução de software para obter o melhor conjunto de módulos?”. O princípio de *encapsulamento de informações* [Par72] sugere que os módulos sejam “caracterizados por decisões de projeto que ocultem (cada uma delas) todas as demais”. Em outras palavras, os módulos devem ser especificados e projetados de modo que as informações (algoritmos e dados) contidas em um módulo sejam inacessíveis por parte de outros módulos que não necessitam de tais informações, disponibilizando apenas os itens que interessam aos outros módulos.

Encapsulamento implica que uma modularidade efetiva pode ser obtida por meio da definição de um conjunto de módulos independentes que passam entre si apenas as informações necessárias para realizar determinada função do software. A abstração ajuda a definir as entidades procedurais (ou informativas) que constituem o software. O encapsulamento define e impõe restrições de acesso, tanto a detalhes procedurais em um módulo quanto a qualquer estrutura de dados local usada pelo módulo [Ros75].

Qual o número exato de módulos para um sistema?

O intuito do encapsulamento de informações é esconder os detalhes das estruturas de dados e do processamento procedural que estão por trás da interface de acesso a um módulo. Os detalhes não precisam ser conhecidos por usuários do módulo.

* N. de R.T.: Encapsulamento é uma técnica de engenharia amplamente utilizada. Por exemplo, um componente de hardware digital é projetado da seguinte forma: esconde aspectos que não interessam aos demais componentes e publica aspectos úteis aos demais componentes.

O uso de encapsulamento de informações como critério de projeto para sistemas modulares fornece seus maiores benefícios quando são necessárias modificações durante os testes e, posteriormente, durante a manutenção do software. Como a maioria dos detalhes procedurais e de dados são ocultos para outras partes do software, erros introduzidos inadvertidamente durante a modificação em um módulo têm menor probabilidade de se propagar para outros módulos ou locais dentro do software.

12.3.7 Independência funcional

O conceito de independência funcional é resultado direto da separação por interesses, da modularidade e dos conceitos de abstração e encapsulamento de informações. Em artigos marcantes sobre projeto de software, Wirth [Wir71] e Parnas [Par72] tratam de técnicas de refinamento que aumentam a independência entre módulos. Um trabalho posterior de Stevens, Myers e Constantine [Ste74] solidificou o conceito.

A independência funcional é atingida desenvolvendo-se módulos com função “única” e com “aversão” à interação excessiva com outros módulos. Em outras palavras, devemos projetar software de modo que cada módulo atenda a um subconjunto específico de requisitos e tenha uma interface simples, quando vista de outras partes da estrutura do programa.

Por que a independência é importante? Software com efetiva modularidade, isto é, com módulos independentes, é mais fácil de ser desenvolvido, pois a função pode ser compartimentalizada; e as interfaces, simplificadas (considere as consequências de quando o desenvolvimento é conduzido por uma equipe). Módulos independentes são mais fáceis de ser mantidos (e testados), pois efeitos colaterais provocados por modificação no código ou projeto são limitados, a propagação de erros é reduzida e módulos reutilizáveis são possíveis. Em suma, a independência funcional é a chave para um bom projeto, e projeto é a chave para a qualidade de um software.

A independência é avaliada por dois critérios qualitativos: coesão e acoplamento. A *coesão* indica a robustez funcional relativa de um módulo. O *acoplamento* indica a interdependência relativa entre os módulos.

A coesão é uma extensão natural do conceito do encapsulamento de informações descrito na Seção 12.3.6. Um módulo coeso realiza uma única tarefa, exigindo pouca interação com outros componentes em outras partes de um programa. De forma simples, um módulo coeso deve (de maneira ideal) fazer apenas uma coisa. Embora você sempre deva tentar ao máximo obter uma alta coesão (funcionalidade única), muitas vezes é necessário e recomendável fazer com que um componente de software realize várias funções. Entretanto, componentes “esquizofrênicos” (módulos que realizam muitas funções não relacionadas) devem ser evitados caso se queira um bom projeto.

O acoplamento é uma indicação da interconexão entre os módulos em uma estrutura de software. Ele depende da complexidade da interface entre os módulos, do ponto onde é feito o acesso a um módulo e dos dados que passam pela interface. Em projeto de software, você deve se esforçar para obter o menor grau de acoplamento possível. A conectividade simples entre módulos

Por que devemos nos esforçar para criar módulos independentes?

Coesão é uma indicação qualitativa do grau com o qual um módulo se concentra em fazer apenas uma coisa.

Acoplamento é uma indicação qualitativa do grau com o qual um módulo está conectado a outros módulos e com o mundo externo.

resulta em software mais fácil de ser compreendido e menos sujeito à “reação em cadeia” [Ste74], provocada quando ocorrem erros, em um ponto, que se propagam por todo o sistema.

12.3.8 Refinamento

Refinamento gradual é uma estratégia de projeto descendente (*top-down*) proposta originalmente por Niklaus Wirth [Wir71]. Uma aplicação é desenvolvida refinando-se sucessivamente níveis de detalhes procedurais. É desenvolvida uma hierarquia através da decomposição de uma declaração macroscópica da função (uma abstração procedural) de forma gradual até que as instruções da linguagem de programação sejam atingidas.

Refinamento é, na verdade, um processo de *elaboração*. Começamos com um enunciado da função (ou descrição de informações) definida em um nível de abstração alto. O enunciado descreve a função ou informações conceitualmente, mas não fornece nenhuma indicação do funcionamento interno da função ou da estrutura interna das informações. Em seguida, elaboramos a declaração original, fornecendo cada vez mais detalhes à medida que ocorre cada refinamento (elaboração) sucessivo.

Abstração e refinamento são conceitos complementares. A abstração nos permite especificar procedimentos e dados internamente, mas suprime a necessidade de que “estranhos” tenham de conhecer detalhes de baixo nível. O refinamento nos ajuda a revelar detalhes menores à medida que o projeto avança. Ambos os conceitos permitem que criemos um modelo de projeto completo à medida que o projeto evolui.

12.3.9 Aspectos

À medida que a análise de requisitos ocorre, um conjunto de “interesses (ou afinidades)” se revela. Entre esses interesses “temos os requisitos, os casos de uso, as características, as estruturas de dados, questões de qualidade de serviço, variações, limites de propriedade intelectual, colaborações, padrões e contratos” [AOS07]. De maneira ideal, um modelo de requisitos pode ser organizado de forma a permitir isolar grupos de interesses (requisitos) para que possam ser considerados independentemente. Na prática, entretanto, alguns desses interesses abrangem o sistema inteiro e não podem ser facilmente dividido em compartimentos.

Quando um projeto se inicia, os requisitos são refinados em uma representação de projeto modular. Consideremos dois requisitos, *A* e *B*. O requisito *A* intersecciona o requisito *B* “se tiver sido escolhida uma decomposição [refinamento] de software em que *B* não pode ser satisfeito sem levar em conta *A*” [Ros04].

Por exemplo, considere dois requisitos para a WebApp **www.casaseguragarantida.com**. O requisito *A* é descrito por meio do caso de uso AVC-EVC discutido no Capítulo 9. O refinamento de um projeto poderia se concentrar nos módulos que permitiriam a um usuário registrado acessar imagens de vídeo de câmeras distribuídas em um ambiente. O requisito *B* é um requisito de segurança genérico que afirma que *um usuário registrado tem de ser validado antes de usar www.casaseguragarantida.com*. Esse requisito se aplica a todas

Existe uma tendência a ir imediatamente até o último detalhe, pulando as etapas de refinamento. Isso induz a erros e omissões e torna o projeto muito mais difícil de ser revisado. Realize o refinamento gradual.

“É difícil ler de cabo a rabo um livro sobre princípios de mágica sem, de tempos em tempos, dar uma olhada na capa para ter certeza de que não é um livro sobre projeto de software.”

Bruce Tognazzini

Preocupação transversal é alguma característica do sistema que se aplica a vários requisitos diferentes.

as funções disponíveis para usuários registrados do *CasaSegura*. À medida que ocorre o refinamento de projeto, A^* é uma representação de projeto para o requisito A , e B^* é uma representação de projeto para o requisito B . Consequentemente, A^* e B^* são representações de interesses, e B^* tem intersecção com A^* .

Aspecto é uma representação de um interesse em comum. Portanto, a representação de projeto, B^* , do requisito *um usuário registrado tem de ser validado antes de usar www.casaseguragarantida.com*, é um aspecto da WebApp *CasaSegura*. É importante identificar aspectos de modo que o projeto possa acomodá-los apropriadamente à medida que ocorrem o refinamento e a modularização. Em um contexto ideal, um aspecto é implementado como um módulo (componente) separado, em vez de trechos de software “espalhados” ou “emaranhados” em vários componentes [Ban06a]. Para tanto, a arquitetura de projeto deve oferecer suporte para um mecanismo de definição de aspectos – um módulo que possibilite que um interesse seja implementado e atenda aos demais interesses que ele interseccione.

12.3.10 Refatoração

Excelentes recursos de refatoração podem ser encontrados em www.refactoring.com.

Uma importante atividade sugerida por diversos métodos ágeis (Capítulo 5), a *refatoração* é uma técnica de reorganização que simplifica o projeto (ou código) de um componente sem mudar sua função ou comportamento. Fowler [Fow00] define refatoração da seguinte maneira: “Refatoração é o processo de mudar um sistema de software de tal forma que não altere o comportamento externo do código [projeto], embora melhore sua estrutura interna”.

Uma série de padrões de refatoração podem ser encontrados em <http://c2.com/cgi/wiki?RefactoringPatterns>.

Quando um software é refatorado, o projeto existente é examinado em termos de redundância, elementos de projeto não utilizados, algoritmos ineficientes ou desnecessários, estruturas de dados mal construídas ou inadequadas ou qualquer outra falha de projeto que possa ser corrigida para produzir um projeto melhor. Por exemplo, uma primeira iteração de projeto poderia gerar um componente que apresentasse baixa coesão (realizar três funções que possuem apenas relação limitada entre si). Após cuidadosa consideração, talvez decidamos que o componente deve ser refatorado em três componentes distintos, cada um apresentando alta coesão. O resultado será um software mais fácil de integrar, testar e manter.

Embora a intenção da refatoração seja modificar o código de uma forma que não altere seu comportamento externo, podem ocorrer (e realmente ocorrem) efeitos colaterais involuntários. Como consequência, são utilizadas ferramentas de refatoração [Soa10] para analisar as alterações automaticamente e “gerar um conjunto de testes adequado para detectar mudanças de comportamento”.

12.3.11 Conceitos de projeto orientado a objetos

O paradigma orientado a objetos (OO, object oriented) é amplamente utilizado na engenharia de software moderna. O Apêndice 2 é dirigido àqueles que talvez não conheçam os conceitos de projeto OO, como classes e objetos, herança, mensagens e polimorfismo, entre outros.



Conceitos de projeto

Cena: Sala do Vinod, quando começa a modelagem de projetos.

Atores: Jamie, Vinod e Ed – todos eles membros da equipe de engenharia de software do *CasaSegura*. Também participa Shakira, novo membro da equipe.

Conversa:

Os quatro membros da equipe acabaram de voltar de um seminário intitulado “Aplicação de Conceitos Básicos de Projeto”, ministrado por um professor de computação.

Vinod: Vocês tiveram algum proveito do seminário?

Ed: Já conhecia grande parte do que foi falado, mas não é má ideia ouvir novamente, suponho.

Jamie: Quando era aluno de Ciências da Computação, nunca entendi realmente por que o encapsulamento de informações era tão importante como diziam.

Vinod: Porque... basicamente... é uma técnica para reduzir a propagação de erros em um programa. Na verdade, a independência funcional também faz a mesma coisa.

Shakira: Eu não fiz Ciências da Computação; portanto um monte de coisas que o professor mencionou é novidade para mim. Sou capaz de gerar bom código e rapidamente. Não vejo por que isso é tão importante.

Jamie: Vi seu trabalho, Shak, e sabe de uma coisa, você já faz naturalmente grande parte do que foi dito... É por isso que seus projetos e códigos funcionam.

Shakira (sorrindo): Bem, sempre tento subdividir o código, mantê-lo concentrado em algo, manter as interfaces simples e restritas, reutilizar código sempre que posso... Esse tipo de coisa.

Ed: Modularidade, independência funcional, encapsulamento, padrões... está vendo?

Jamie: Ainda me lembro do primeiro curso de programação que fiz... Eles nos ensinaram a refinar o código iterativamente.

Vinod: O mesmo pode ser aplicado ao projeto, sabe.

Jamie: Os únicos conceitos que ainda não havia ouvido falar foram “aspectos” e “refatoração”.

Shakira: Isso é usado em *Extreme Programming*, acho que foi isso que ele disse.

Ed: Sim. Não é muito diferente do refinamento, apenas que você o faz depois que o projeto ou código está concluído. Acontece uma espécie de otimização no software, se você quer saber.

Jamie: Retornemos ao projeto *CasaSegura*. Imagino que devamos colocar esses conceitos em nossa lista de controle de revisão à medida que desenvolvermos o modelo de projeto para o *CasaSegura*.

Vinod: Concordo. Mas tão importante quanto, vamos todos nos comprometer a pensar nelas enquanto desenvolvemos o projeto.

12.3.12 Classes de projeto

O modelo de análise define um conjunto de classes de análise (Capítulo 10). Cada uma dessas classes descreve algum elemento do domínio do problema, concentrando-se nos aspectos do problema visíveis ao usuário. O nível de abstração de uma classe de análise é relativamente alto.

À medida que o modelo de projeto evoluir, definiremos um conjunto de *classes de projeto* que refinem as classes de análise, fornecendo detalhes de projeto que permitirão implementar as classes, e implementaremos uma infraestrutura de software que suporte a solução do negócio. Podem ser desenvolvidos cinco tipos diferentes de classes de projeto, cada um deles representando uma camada diferente da arquitetura de projeto [Amb01]. *Classes de interfaces do usuário* definem todas as abstrações necessárias para a interação humano-computador (*human-computer interaction*, HCI) e, em muitos casos, implementam a HCI no contexto de uma metáfora. *Classes de domínio de negócio* identificam os atributos e serviços (métodos) necessários para implementar algum elemento do domínio de negócio definido por uma ou mais classes de análise. *Classes de processos* implementam as abstrações de aplicação

Quais tipos de classes o projetista cria?

de baixo nível necessárias para a completa gestão das classes de domínio de negócio. *Classes persistentes* representam repositórios de dados (por exemplo, um banco de dados) que persistirá depois da execução do software. *Classes de sistema* implementam funções de gerenciamento e controle de software que permitem ao sistema operar e comunicar em seu ambiente computacional e com o mundo exterior.

À medida que a arquitetura se forma, o nível de abstração é reduzido, enquanto cada classe de análise (Capítulo 10) é transformada em uma representação de projeto. As classes de análise representam objetos de dados (e serviços associados aplicados a eles), usando o jargão do domínio de negócio. Classes de projeto apresentam um detalhe significativamente mais técnico como um guia para a implementação.

Arlow e Neustadt [Arl02] sugerem que cada classe de projeto seja revista para garantir que seja “bem formada”. Eles definem quatro características de uma classe de projeto bem formada:

O que é uma classe de projeto “bem formada”?

Completa e suficiente. Uma classe de projeto deve ser o encapsulamento completo de todos os atributos e métodos exigidos por uma classe (com base em uma interpretação reconhecível do nome da classe). Por exemplo, a classe **Cena** definida para software de edição de vídeo estará completa apenas se contiver todos os atributos e métodos que podem ser razoavelmente associados à criação de uma cena de vídeo. A suficiência garante que a classe de projeto contenha somente os métodos necessários para atingir a finalidade da classe, nem mais nem menos.

Primitivismo. Os métodos associados a uma classe de projeto deveriam se concentrar na realização de um serviço para a classe. Assim que o serviço tivesse sido implementado com um método, a classe não deveria realizar a mesma coisa de outra maneira. Por exemplo, a classe **VideoClipe** para um software de edição de vídeo poderia ter atributos para indicar os pontos de início e fim do clipe (observe que uma fita virgem carregada no sistema talvez seja mais longa do que o clipe utilizado). Os métodos, *estabelecerPontoInício()* e *estabelecerPontoFinal()*, fornecem os únicos meios para estabelecer os pontos de início e fim do clipe.

Alta coesão. Uma classe de projeto coesa tem um conjunto de responsabilidades pequeno e focado – e, de forma resoluta, aplica atributos e métodos para implementar essas responsabilidades. Por exemplo, a classe **VideoClipe** poderia conter um conjunto de métodos para editar o videoclipe. Contanto que cada método se concentre somente em atributos associados ao videoclipe, a coesão é mantida.

Baixo acoplamento. No modelo de projeto, é necessário que as classes de projeto colaborem umas com as outras. No entanto, a colaboração deve ser mantida em um nível mínimo aceitável. Se um modelo de projeto for altamente acoplado (todas as classes de projeto colaboram com todas as demais classes de projeto), o sistema é difícil de implementar, testar e manter ao longo do tempo. Em geral, classes de projeto em um subsistema devem ter apenas um conhecimento limitado das outras classes. Essa

restrição, chamada *Lei de Demeter* [Lie03], sugere que um método deve enviar mensagens apenas para métodos de classes vizinhas.⁶

CASASEGURA



Refinamento de uma classe de análise em uma classe de projeto

Cena: Sala do Ed, quando começa o modelamento de projetos.

Atores: Vinod e Ed – membros da equipe de engenharia de software do *CasaSegura*.

Conversa:

[Ed está trabalhando na classe **Planta** (veja discussão no quadro da Seção 10.3 e a Figura 10.2) e a refinou para o modelo de projeto.]

Ed: Então, você se lembra da classe **Planta**, certo? Ela é usada como parte das funções de vigilância e gestão da casa.

Vinod (acenando afirmativamente): É isso mesmo, parece que nós a usamos como parte de nossas discussões CRC para gestão da casa.

Ed: Usamos. De qualquer maneira, estou refinando-a para o projeto. Gostaria de mostrar como realmente implementaremos a classe **Planta**. Minha ideia é implementá-la como um conjunto de listas ligadas [uma estrutura de dados específica]. Então... eu tinha de refinar a classe de análise **Planta** (Figura 10.2) e, na verdade, simplificá-la.

Vinod: A classe de análise mostrava coisas apenas no domínio do problema; bem, na verdade, na tela do computador, que era visível para o usuário, certo?

Ed: Isso, mas, para a classe de projeto **Planta**, tive de acrescentar algumas coisas específicas da implementação. Precisava mostrar que **Planta** é uma agregação de segmentos – daí a classe **Segmento** – e que a classe **Segmento** é composta por listas de segmentos de parede, janelas, portas e assim por diante. A classe **Câmera** colabora com **Planta**, e, obviamente, podem existir muitas câmeras na planta.

Vinod: Ufa, vejamos uma figura dessa nova classe de projeto **Planta**.

[Ed mostra a Vinod o desenho apresentado na Figura 12.3.]

Vinod: Certo, vejo o que você está tentando fazer. Isso lhe permite modificar facilmente a planta, pois novos itens podem ser acrescentados ou eliminados da lista – a agregação – sem quaisquer problemas.

Ed (acenando afirmativamente): É isso aí, acho que vai funcionar.

Vinod: Eu também.

12.3.13 Inversão da dependência

A estrutura de muitas arquiteturas de software mais antigas é hierárquica. No topo da arquitetura, componentes de “controle” contam com componentes “de trabalho” de nível mais baixo para executar várias tarefas coesas. Consideremos um programa simples, com três componentes. O objetivo do programa é ler as teclas pressionadas no teclado e imprimir o resultado em uma impressora. Um módulo de controle, *C*, coordena dois outros módulos – um módulo leitor de toque de tecla, *R*, e um módulo que escreve em uma impressora, *W*.

O projeto do programa é acoplado, pois *C* é altamente dependente de *R* e *W*. Para eliminar o nível de dependência existente, os módulos “de trabalho” *R* e *W* devem ser invocados a partir do módulo de controle *S* por meio de abstrações. Na engenharia de software orientada a objetos, as abstrações são implementadas como classes abstratas, **R*** e **W***. Essas classes abstratas poderiam, então, ser usadas para invocar classes de trabalho que executariam qualquer função de leitura e escrita. Portanto, uma classe **copiar**, *C*, invoca

O que é o “princípio da inversão da dependência”?

⁶ Uma maneira menos formal de expressar a Lei de Demeter seria: “Cada unidade deve conversar apenas com seus amigos; não converse com estranhos”.

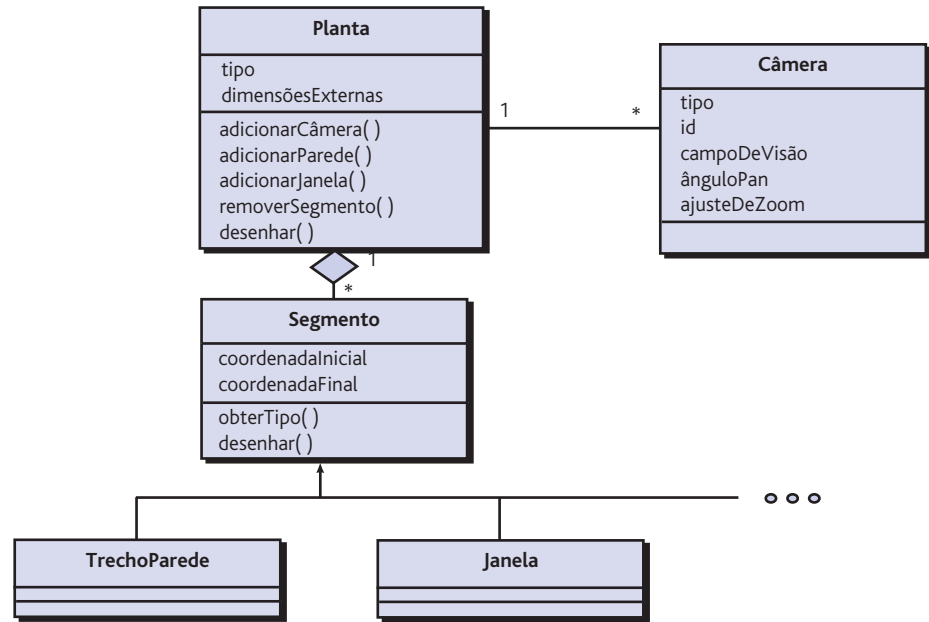


FIGURA 12.3 Classe de projeto para Planta e agregação composta para a classe (consulte a discussão no quadro).

classes abstratas, R^* e W^* , e a classe abstrata aponta para a classe de trabalho apropriada (por exemplo, em um contexto, a classe R^* poderia apontar para uma operação *ler()* dentro de uma classe **teclado** e, em outro, para uma operação *ler()* dentro de uma classe **sensor**). Essa abordagem reduz o acoplamento e melhora a facilidade de realizar testes de um projeto.

O exemplo discutido no parágrafo anterior pode ser generalizado com o princípio da inversão da dependência [Obj10], que diz: *Módulos de alto nível (classes) não devem depender [diretamente] de módulos de baixo nível. Ambos devem depender de abstrações. As abstrações não devem depender de detalhes. Os detalhes devem depender de abstrações.*

12.3.14 Projeto para teste

“Teste rápido, falhe rápido, ajuste rápido.”

Tom Peters

Como no dilema do ovo e da galinha, existe um permanente debate sobre quem nasceu primeiro, se foi o projeto de software ou o projeto de casos de teste. Rebecca Wirfs-Brock [Wir09] escreve:

Os defensores do desenvolvimento orientado a testes (*test-driven development*, TDD) escrevem os testes antes de implementar qualquer outro código. Eles levam a sério o credo de Tom Peters, que diz “Teste rápido, falhe rápido, ajuste rápido”. Os testes guiam seus projetos ao implementarem concisamente ciclos acelerados do tipo “escrever código de teste – falhar no teste – escrever código suficiente para passar – e, então, passar no teste”.

Porém, se o projeto vem primeiro, então ele (e o código) deve ser desenvolvido com *costuras* – locais no projeto detalhado onde é possível “inserir código de teste que investigue o estado de seu software em execução” e/ou “isolar o código sob teste de seu ambiente de produção para que se possa experimentá-lo em um contexto de teste controlado” [Wir09].

Algumas vezes chamadas “ganchos de teste”, as costuras devem ser projetadas conscientemente no nível do componente. Para isso, um projetista deve pensar nos testes que vão ser realizados para experimentar o componente. Conforme diz Wirfs-Brock: “Em resumo, você precisa fornecer funcionalidades de teste adequadas – fatorar seu projeto de um modo que permita ao código de teste interrogar e controlar o sistema em execução”.

12.4 O modelo de projeto

O modelo de projeto pode ser visto em duas dimensões, conforme ilustrado na Figura 12.4. A *dimensão processo* indica uma evolução do modelo de projeto à medida que as tarefas de projeto são executadas como parte do processo do software. A *dimensão abstração* representa o nível de detalhe à medida que cada elemento do modelo de análise é transformado em um equivalente de projeto e, então, refinado iterativamente. Na figura, a linha tracejada indica o limite entre os modelos de análise e de projeto. Em alguns casos, é possível ter uma clara distinção entre os modelos de análise e de projeto. Em outros, o modelo de análise vai lentamente se misturando ao de projeto, e essa distinção clara é menos evidente.

Os elementos do modelo de projeto usam vários dos diagramas UML⁷ utilizados no modelo de análise. A diferença é que esses diagramas são refinados e elaborados como parte do projeto; são fornecidos detalhes mais específicos à implementação e enfatizados a estrutura e o estilo da arquitetura, os compo-

O modelo de projeto possui quatro elementos principais: dados, arquitetura, componentes e interface.

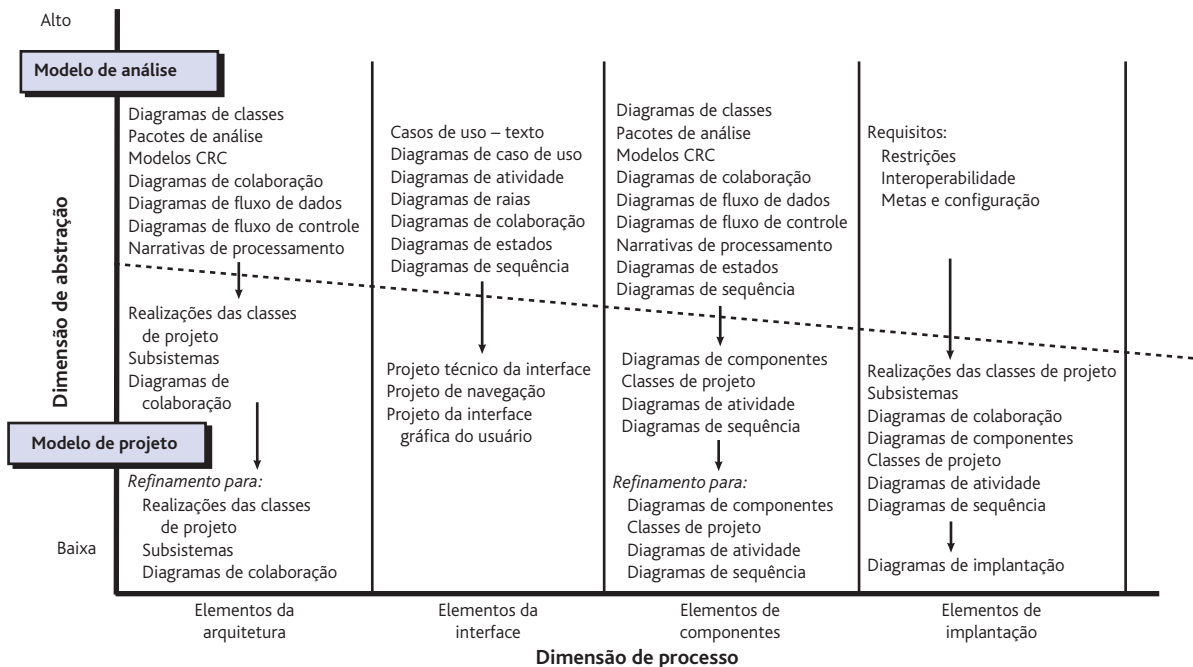


FIGURA 12.4 Dimensões do modelo de projeto.

⁷ O Apêndice 1 apresenta um tutorial sobre conceitos básicos e notação da UML.

“Questões como se o projeto é necessário ou acessível não vêm ao caso: o projeto é inevitável. A alternativa para um bom projeto é um projeto ruim, e não nenhum projeto em absoluto.”

Douglas Martin

No nível da arquitetura (aplicação), o projeto de dados se concentra em arquivos ou bancos de dados; no nível dos componentes, o projeto de dados considera as estruturas de dados necessárias para implementar os objetos de dados locais.

“Você pode usar uma borracha, enquanto ainda estiver na prancheta, ou uma marreta depois, na obra.”

Frank Lloyd Wright

nentes que residem nessa arquitetura, bem como as interfaces entre os componentes e com o mundo exterior.

Entretanto, os elementos de modelo indicados ao longo do eixo horizontal nem sempre são desenvolvidos de maneira sequencial. Na maioria dos casos, um projeto de arquitetura preliminar prepara o terreno e é seguido pelos projetos de interfaces e projeto de componentes, que normalmente ocorrem em paralelo. O modelo de implantação em geral é retardado até que o projeto tenha sido completamente desenvolvido.

Podemos aplicar padrões de projeto (Capítulo 16) em qualquer ponto durante o projeto. Estes possibilitam a utilização de conhecimentos adquiridos em projetos anteriores para problemas de domínios específicos encontrados e solucionados por outros.

12.4.1 Elementos de projeto de dados

Assim como ocorre com outras atividades da engenharia de software, o projeto de dados (também conhecido como *arquitetura de dados*) cria um modelo de dados e/ou informações que é representado em um nível de abstração elevado (a visão do cliente/usuário dos dados). Esse modelo é, então, refinado em representações cada vez mais específicas da implementação que podem ser processadas pelo sistema baseado em computador. Em muitas aplicações de software, a arquitetura dos dados terá uma profunda influência sobre a arquitetura do software que deve processá-los.

A estrutura de dados sempre foi uma parte importante do projeto de software. No nível dos componentes de programa, o projeto das estruturas de dados e os algoritmos associados necessários para manipulá-los são essenciais para a criação de aplicações de alta qualidade. No nível da aplicação, a transformação de um modelo de dados (obtido como parte da engenharia de requisitos) em um banco de dados é fundamental para atingir os objetivos de negócio de um sistema. No nível de negócio, o conjunto de informações armazenadas em bancos de dados diferentes e reorganizadas em um “depósito de dados” possibilita o *data mining* ou descoberta de conhecimento que pode ter um impacto no sucesso do negócio em si. Em qualquer caso, o projeto de dados desempenha um papel importante. O projeto de dados é discutido de forma mais detalhada no Capítulo 13.

12.4.2 Elementos do projeto de arquitetura

O *projeto de arquitetura* para software é o equivalente à planta baixa de uma casa. A planta baixa representa a distribuição dos cômodos; seus tamanhos, formas e relações entre si e as portas e janelas que possibilitam o deslocamento para dentro e para fora dos cômodos. A planta baixa nos dá uma visão geral da casa. Os elementos de projeto de arquitetura nos dão uma visão geral do software.

O modelo de arquitetura [Sha96] é obtido de três fontes: (1) informações sobre o domínio de aplicação do software a ser construído; (2) elementos específicos do modelo de requisitos, como os casos de uso ou as classes de análise, suas relações e colaborações para o problema em questão; e (3) a disponibilidade de estilos de arquitetura (Capítulo 13) e padrões (Capítulo 16).

O projeto dos elementos de arquitetura é normalmente representado como um conjunto de subsistemas interligados, em geral derivados dos pacotes de análise contidos no modelo de requisitos. Cada subsistema pode ter sua própria arquitetura (por exemplo, uma interface gráfica do usuário poderia ser estruturada de acordo com um estilo de arquitetura preexistente para interfaces do usuário). Técnicas para obtenção de elementos específicos do modelo de arquitetura são apresentadas no Capítulo 13.

12.4.3 Elementos do projeto de interface

O projeto de interface para software é análogo a um conjunto de desenhos detalhados (e especificações) para portas, janelas e ligações externas de uma casa. Basicamente, os desenhos detalhados (e especificações) para portas, janelas e ligações externas nos notificam como as coisas e as informações fluem para dentro e para fora da casa e no interior dos cômodos que fazem parte da planta. Os elementos de projeto de interfaces para software representam fluxos de informação que entram e saem de um sistema e como são transmitidos entre os componentes definidos como parte da arquitetura.

Há três importantes elementos de projeto de interfaces: (1) a interface do usuário (UI, user interface), (2) interfaces externas para outros sistemas, dispositivos, redes ou outros produtores ou consumidores de informação e (3) interfaces internas entre vários componentes do projeto. Esses elementos do projeto de interfaces possibilitam que o software se comunique externamente e que a comunicação interna e a colaboração entre os componentes preencham a arquitetura de software.

O projeto da UI (cada vez mais chamado *projeto de usabilidade*) é uma importante ação da engenharia de software e é considerado em detalhes no Capítulo 15. O projeto de usabilidade incorpora elementos estéticos (por exemplo, layout, cor, imagens, mecanismos de interação), elementos ergonômicos (por exemplo, o layout e o posicionamento de informações, metáforas, navegação da UI) e elementos técnicos (por exemplo, padrões UI, componentes reutilizáveis). Em geral, a UI é um subsistema exclusivo da arquitetura da aplicação geral.

O projeto de interfaces externas exige informações definitivas sobre a entidade para as quais as informações são enviadas ou recebidas. Em todos os casos, essas informações devem ser coletadas durante a engenharia de requisitos (Capítulo 8) e verificadas assim que o projeto de interface for iniciado.⁸ O projeto de interfaces externas deve incorporar verificação de erros e características de segurança apropriadas.

O projeto de interfaces internas está intimamente ligado ao projeto dos componentes (Capítulo 14). As realizações de projeto das classes de análise representam todas as operações e os esquemas de troca de mensagens necessários para permitir a comunicação e a colaboração entre as operações em várias classes. Cada mensagem deve ser desenvolvida para acomodar a transferência de informações exigidas e os requisitos funcionais específicos da operação solicitada.

"O público está mais acostumado com projetos ruins do que com projetos bons. Ele está, de fato, condicionado a preferir projetos ruins, pois é com isso que convive. O novo representa uma ameaça; o antigo, um sentimento de tranquilidade."

Paul Rand

Há três partes para o elemento de projeto de interfaces: a interface do usuário, interfaces com os sistemas externos à aplicação e interfaces com componentes internos à aplicação.

"De tempos em tempos, dê uma volta, relaxe um pouco, de modo que, ao voltar para o trabalho, seu julgamento seja mais seguro. Afaste-se um pouco; o trabalho parecerá menor e grande parte dele poderá ser vista com um pequeno exame. A falta de harmonia e proporção serão visualizadas mais facilmente."

Leonardo DaVinci

⁸ As características das interfaces podem mudar ao longo do tempo. Consequentemente, é papel do projetista garantir que a especificação para uma interface seja precisa e completa.

Em alguns casos, uma interface é modelada de forma bastante parecida com a de uma classe. Na UML, a interface é definida da seguinte maneira [OMG03a]: “Interface é um especificador para as operações [públicas] visíveis externamente de uma classe, componente ou outro classificador (incluindo subsistemas), sem a especificação da estrutura interna”. De maneira mais simples, interface é um conjunto de operações que descreve alguma parte do comportamento de uma classe e dá acesso a essas operações.

Por exemplo, a função de segurança do *CasaSegura* faz uso de um painel de controle que possibilita a um proprietário de imóvel controlar certos aspectos da função de segurança. Em uma versão mais avançada do sistema, as funções do painel de controle poderiam ser implementadas por meio de uma plataforma móvel (por exemplo, smartphones ou tablets).

A classe **PainelControle** (Figura 12.5) fornece o comportamento associado a um teclado e, conseqüentemente, deve implementar as operações *lerTeclaPressionada()* e *decodificarTecla()*. Se essas operações tiverem de ser fornecidas para outras classes (no caso, **Tablet** e **SmartPhone**), é útil definir uma interface conforme mostra a figura. A interface, chamada **Teclado**, é apresentada como um estereótipo <<interface>>, ou como um pequeno círculo identificado, ligado à classe por meio de uma linha. A interface é definida sem nenhum atributo e conjunto de operações necessárias para obter o comportamento de um teclado.

A linha tracejada com um triângulo com fundo branco em sua ponta (Figura 12.5) indica que a classe **PainelControle** fornece operações de **Teclado** como parte de seu comportamento. Na UML, isso é caracterizado como uma *realização*. Ou seja, parte do comportamento de **PainelControle** será implementada realizando as operações de **Teclado**. Essas operações serão fornecidas para outras classes que acessam a interface.

Informações extremamente valiosas sobre projeto de UI podem ser encontradas em www.useit.com.

“Um erro comum que as pessoas cometem ao tentar projetar algo completamente infalível é subestimar a criatividade de completos idiotas.”

Douglas Adams

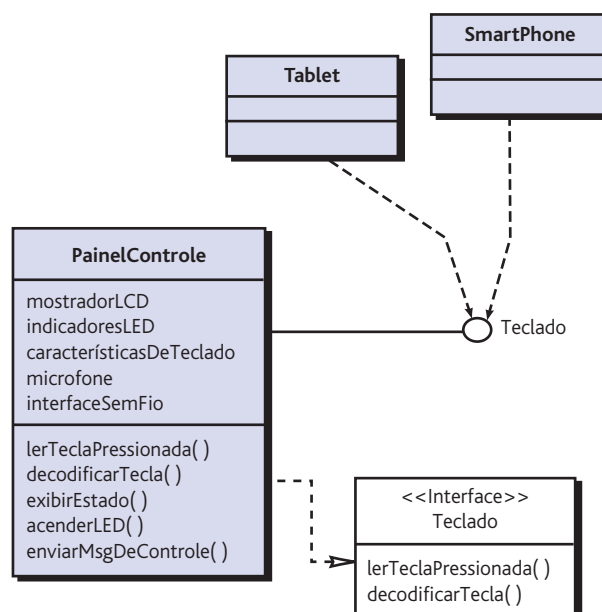


FIGURA 12.5 Representação da interface para PainelControle.

12.4.4 Elementos do projeto de componentes

O projeto de componentes para o software equivale a um conjunto de desenhos detalhados (e especificações) para cada cômodo de uma casa. Esses desenhos representam a fiação e o encanamento dentro de cada cômodo, a localização das tomadas e interruptores, torneiras, pias, chuveiros, banheiras, ralos, armários, roupeiros e todos os outros detalhes associados a um cômodo.

O projeto de componentes para software descreve completamente os detalhes internos de cada componente de software. Para tanto, o projeto no nível de componente define estruturas de dados para todos os objetos de dados locais e detalhes algorítmicos para todo o processamento que ocorre em um componente e uma interface que dá acesso a todas as operações de componentes (comportamentos).

No contexto da engenharia de software orientada a objetos, um componente é representado de forma esquemática em UML, conforme mostra a Figura 12.6. Nessa figura, é representado um componente chamado **GestãoDeSensor** (parte da função de segurança do *CasaSegura*). Uma seta pontilhada conecta o componente a uma classe chamada **Sensor** que é atribuída a ele. O componente **GestãoDeSensor** realiza todas as funções associadas aos sensores do *CasaSegura*, incluindo seu monitoramento e configuração. Uma discussão mais abrangente sobre diagramas de componentes é apresentada no Capítulo 14.

"Detalhes não são detalhes. Eles compõem o projeto."

Charles Eames

Os detalhes de projeto de um componente podem ser modelados em muitos níveis de abstração diferentes. Um diagrama de atividades UML pode ser utilizado para representar processamento lógico. O fluxo procedural detalhado para um componente pode ser representado usando pseudocódigo (uma representação semelhante a uma linguagem de programação descrita no Capítulo 14) ou alguma outra forma esquemática (por exemplo, fluxograma ou diagrama de blocos). A estrutura algorítmica segue as regras estabelecidas para a programação estruturada (um conjunto de construções procedurais restritas). As estruturas de dados escolhidas, tomando como base a natureza dos objetos de dados a ser processados, normalmente são modeladas usando pseudocódigo ou a linguagem de programação para implementação.

12.4.5 Elementos do projeto de implantação

Os elementos de projeto de implantação indicam como os subsistemas e a funcionalidade de software serão alocados no ambiente computacional físico que vai suportar o software. Por exemplo, os elementos do produto *CasaSe-*

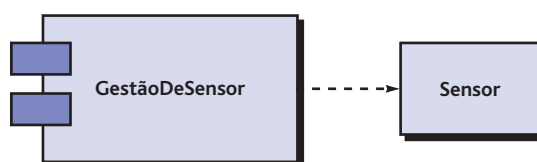


FIGURA 12.6 Um diagrama de componentes UML.

Os diagramas de disponibilização começam na forma de descritores, em que o ambiente de implantação é descrito em termos gerais. Posteriormente, é usada a forma de instância, e os elementos da configuração são descritos explicitamente.

gura são configurados para operar dentro de três ambientes computacionais principais – um PC localizado na casa, o painel de controle *CasaSegura* e um servidor localizado na CPI Corp. (fornecendo acesso ao sistema via Internet). Além disso, uma funcionalidade limitada pode ser fornecida com plataformas móveis.

Durante o projeto, um diagrama de implantação UML é desenvolvido e refinado, conforme mostra a Figura 12.7. Na figura são apresentados três ambientes computacionais (na verdade, existiriam outros, com a inclusão de sensores, câmeras e funcionalidade implementada por plataformas móveis). São indicados os subsistemas (funcionalidade) abrigados em cada elemento computacional. Por exemplo, o PC abriga subsistemas que implementam funções de segurança, vigilância, gestão residencial e de comunicação. Além disso, um subsistema de acesso externo foi projetado para gerenciar todas as tentativas de acessar o sistema *CasaSegura* a partir de uma fonte externa. Cada subsistema seria elaborado para indicar os componentes que implementa.

O diagrama apresentado na Figura 12.7 se encontra na *forma de descritores*. Isso significa que o diagrama de implantação mostra o ambiente computacional, mas não indica detalhes da configuração explicitamente. Por exemplo, o “computador pessoal” não tem uma identificação adicional. Poderia ser um Mac ou um PC com Windows, um computador com Linux ou uma plataforma móvel com seu sistema operacional associado. Esses detalhes são fornecidos quando o diagrama de implantação é revisitado na *forma de instância*, durante os últimos estágios do projeto ou quando começa a construção. Cada instância da implantação (uma configuração de hardware específica, com nome) é identificada.

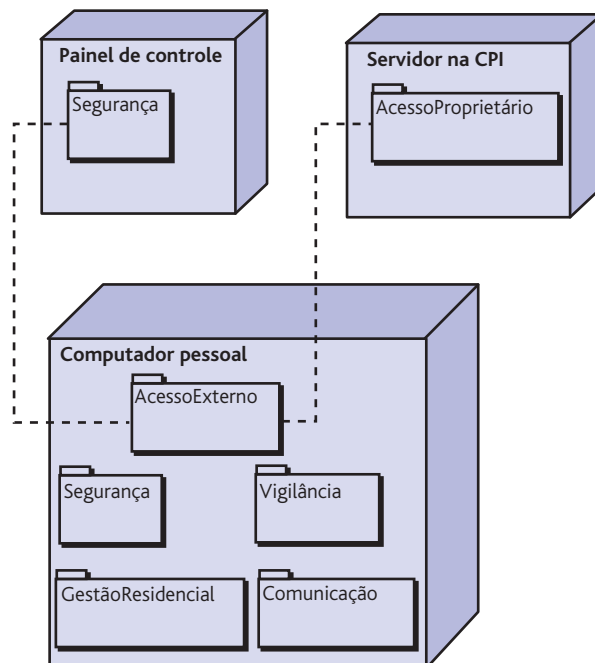


FIGURA 12.7 Um diagrama de implantação UML.

12.5 Resumo

O projeto de software começa quando termina a primeira iteração da engenharia de requisitos. O objetivo do projeto de software é aplicar um conjunto de princípios, conceitos e práticas que levem ao desenvolvimento de um sistema ou produto de alta qualidade. A meta do projeto é criar um modelo de software que implemente corretamente todos os requisitos do cliente e traga satisfação àqueles que o usarem. Os projetistas de software devem examinar completamente muitas alternativas de projeto e convergir para uma solução que melhor atenda às necessidades dos envolvidos no projeto.

O processo de projeto move-se de uma visão macro do software para uma visão mais estreita que define os detalhes necessários para implementar um sistema. O processo começa focando-se na arquitetura. São definidos subsistemas, estabelecidos mecanismos de comunicação entre os subsistemas, identificados componentes e desenvolvida uma descrição detalhada de cada componente. Além disso, são projetadas interfaces externas, internas e para o usuário.

Os conceitos de projeto evoluíram ao longo dos primeiros 60 anos do trabalho da engenharia de software. Eles descrevem atributos de software que devem estar presentes, independentemente do processo de engenharia de software escolhido, dos métodos de projeto aplicados ou das linguagens de programação usadas. Em essência, os conceitos de projeto enfatizam a necessidade da abstração como mecanismo para a criação de componentes de software reutilizáveis, a importância da arquitetura como forma para melhor entender a estrutura geral de um sistema, os benefícios da engenharia baseada em padrões como técnica para desenvolvimento de software com capacidades já comprovadas, o valor da separação de preocupações e da modularidade eficaz como forma de tornar o software mais compreensível – mais fácil de ser testado e mantido –, as consequências do encapsulamento de informações como um mecanismo para reduzir a propagação de efeitos colaterais quando da real ocorrência de erros, o impacto da independência funcional como critério para a construção de módulos eficazes, o uso do refinamento como mecanismo de projeto, a consideração de aspectos que interseccionem os requisitos do sistema, a aplicação da refatoração na otimização do projeto obtido, a importância das classes orientadas a objetos e das características a elas relacionadas, a necessidade de usar abstração para reduzir o acoplamento entre os componentes e a importância do projeto para teste.

O modelo de projeto abrange quatro elementos diferentes. À medida que cada um é desenvolvido, evolui uma visão mais completa do projeto. O elemento arquitetural usa informações extraídas do domínio de aplicação, do modelo de requisitos e de catálogos disponíveis para padrões e estilos para obter uma representação estrutural completa do software, seus subsistemas e componentes. Elementos de projeto de interfaces modelam interfaces internas e externas, bem como a interface do usuário. Elementos de componentes definem cada um dos módulos (componentes) que preenchem a arquitetura. Por fim, os elementos de implantação alocam a arquitetura, seus componentes e as interfaces para a configuração física que abrigará o software.

Problemas e pontos a ponderar

- 12.1. Você projeta software ao “escrever” um programa? O que torna o projeto de software diferente da codificação?
- 12.2. Se um projeto de software não é um programa (e não é mesmo), então o que ele é?
- 12.3. Como avaliar a qualidade de um projeto de software?
- 12.4. Examine o conjunto de tarefas apresentado para o projeto. Em que momento a qualidade é avaliada em um conjunto de tarefas? Como se consegue isso? Como os atributos de qualidade discutidos na Seção 12.2.1 são atingidos?
- 12.5. Dê exemplos de três abstrações de dados e as abstrações procedurais que podem ser usadas para manipulá-las.
- 12.6. Descreva arquitetura de software com suas próprias palavras.
- 12.7. Sugira um padrão de projeto que você encontra em uma categoria das coisas cotidianas (por exemplo, eletrônica de consumo, automóveis, aparelhos domésticos). Descreva o padrão sucintamente.
- 12.8. Descreva a separação de preocupações com suas próprias palavras. Existe um caso em que a estratégia “dividir para conquistar” poderia não ser apropriada? Como um caso desses poderia afetar o argumento da modularidade?
- 12.9. Quando um projeto modular deve ser implementado como um software monolítico? Como isso pode ser obtido? O desempenho é a única justificativa para a implementação de software monolítico?
- 12.10. Discuta a relação entre o conceito de encapsulamento de informações como um atributo da modularidade eficaz e o conceito da independência de módulos.
- 12.11. Como os conceitos de acoplamento e portabilidade de software estão relacionados? Dê exemplos para apoiar sua discussão.
- 12.12. Aplique uma “metodologia de refinamento gradual” para desenvolver três níveis diferentes de abstrações procedurais para um ou mais dos seguintes programas: (1) desenvolver um preenchedor de cheques que, dada uma quantia numérica, imprima a quantia por extenso como exigido no preenchimento de cheques; (2) encontrar iterativamente as raízes de uma equação transcendental; e (3) desenvolver um algoritmo de cronograma de tarefas simples para um sistema operacional.
- 12.13. Considere o software necessário para implementar um recurso de navegação completo (usando GPS) em um dispositivo de comunicação móvel portátil. Descreva duas ou três preocupações em comum que estariam presentes. Discuta como você representaria uma dessas preocupações na forma de um aspecto.
- 12.14. “Refatoração” significa que modificamos todo o projeto iterativamente? Em caso negativo, o que significa?
- 12.15. Descreva com suas próprias palavras o que é o princípio da inversão da dependência.
- 12.16. Por que o projeto para teste é tão importante?
- 12.17. Descreva brevemente cada um dos quatro elementos do modelo de projeto.

Leituras e fontes de informação complementares

Donald Norman escreveu três livros (*Emotional Design: We Love (or Hate) Everyday Things*, Basic Books, 2005), (*The Design of Everyday Things*, Doubleday, 1990) e (*The Psychology of Everyday Things*, HarperCollins, 1988) que se tornaram clássicos na lite-

ratura de projeto e uma leitura “obrigatória” para qualquer um que projete qualquer coisa utilizada pelos seres humanos. Adams (*Conceptual Blockbusting*, 4ª ed., Addison-Wesley, 2001) é o autor de um texto essencial para os projetistas que querem expandir sua maneira de pensar. Por fim, um clássico de Polya (*How to Solve It*, 2ª ed., Princeton University Press, 1988) fornece um processo genérico para solução de problemas que pode ajudar os projetistas de software ao se depararem com problemas complexos.

Livros de Hanington e Martin (*Universal Methods of Design: 100 Ways to Research Complex Problems, Develop Innovative Ideas, and Design Effective Solutions*, Rockport, 2012) e Hanington e Martin (*Universal Principles of Design: 125 Ways to Enhance Usability, Influence Perception, Increase Appeal, Make Better Design Decisions, and Teach through Design*, 2ª ed., Rockport, 2010) discutem os princípios de projeto em geral.

Seguindo a mesma tradição, Winograd *et al.* (*Bringing Design to Software*, Addison-Wesley, 1996) discutem projetos de software que funcionam, aqueles que não funcionam e por quê. Um livro fascinante, editado por Wixon e Ramsey (*Field Methods Casebook for Software Design*, Wiley, 1996), sugere métodos de pesquisa de campo (muito parecidos com aqueles usados por antropólogos) para entender como os usuários realizam o trabalho deles e depois projetam software que atende às suas necessidades. Holtzblatt (*Rapid Contextual Design: A How-to Guide to Key Techniques for User-Center Design*, Morgan Kaufman, 2004) e Beyer e Holtzblatt (*Contextual Design: A Customer-Centered Approach to Systems Designs*, Academic Press, 1997) oferecem outra visão do projeto de software que integra o cliente/usuário em todos os aspectos do processo de projeto de software. Bain (*Emergent Design*, Addison-Wesley, 2008) acopla padrões, refatoração e desenvolvimento dirigido por testes em uma abordagem de projeto eficaz.

Um tratamento abrangente do projeto no contexto da engenharia de software é apresentado por Otero (*Software Engineering Design: Theory and Practice*, Auerbach, 2012), Venit e Drake (*Prelude to Programming: Concepts and Design*, 5ª ed., Addison-Wesley, 2010), Fox (*Introduction to Software Engineering Design*, Addison-Wesley, 2006) e Zhu (*Software Design Methodology*, Butterworth-Heinemann, 2005). McConnell (*Code Complete*, 2ª ed., Microsoft Press, 2004) traz uma excelente discussão dos aspectos práticos para projetar software de alta qualidade. Robertson (*Simple Program Design*, 5ª ed., Course Technology, 2006) apresenta uma discussão introdutória sobre projeto de software que é útil para aqueles que estão iniciando seus estudos no assunto. Budgen (*Software Design*, 2ª ed., Addison-Wesley, 2004) introduz uma série de métodos de projeto populares, comparando e contrastando cada um deles. Fowler e seus colegas (*Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999) discutem técnicas para a otimização incremental de projetos de software. Rosenberg e Stevens (*Use Case Driven Object Modeling with UML*, Apress, 2007) abordam o desenvolvimento de projetos orientados a objetos usando casos de uso como base.

Uma excelente pesquisa histórica de projeto de software está contida em uma antologia editada por Freeman e Wasserman (*Software Design Techniques*, 4ª ed., IEEE, 1983). Esse tutorial reapresenta diversos artigos clássicos que formaram a base para as tendências atuais em projeto de software. Medidas da qualidade de projeto, apresentadas tanto sob as perspectivas técnica quanto de gerenciamento, são consideradas por Card e Glass (*Measuring Software Design Quality*, Prentice Hall, 1990).

Uma ampla gama de fontes de informação sobre projeto de software se encontra à disposição na Internet. Uma lista atualizada de referências relevantes (em inglês) para o projeto de software e para a engenharia de projeto pode ser encontrada no site: www.mhhe.com/pressman.

13

Projeto de arquitetura

Conceitos-chave

agilidade e arquitetura...	280
arquétipos	269
arquitetura	253
decisões de arquitetura	266
descrições de arquitetura	255
estilos de arquitetura	258
gêneros de arquitetura ..	257
linguagens de descrição da arquitetura	276
padrões de arquitetura ..	263

Projeto foi descrito como um processo em várias etapas no qual as representações de dados e da estrutura do programa, as características das interfaces e os detalhes procedurais são combinados com base nos requisitos de informação. Essa descrição é ampliada por Freeman [Fre80]:

Projeto é uma atividade preocupada com a tomada de decisões importantes, frequentemente de natureza estrutural. Ele divide com a programação a responsabilidade de abstrair a representação de informações e de sequências de processamento, porém o nível de detalhe é bastante diverso nos extremos. Um projeto constrói representações de programas coerentes e bem planejadas, que se concentram nas inter-relações das partes em um alto nível e nas operações lógicas envolvidas em níveis mais baixos.

PANORAMA

O projeto de arquitetura representa a estrutura de dados e os componentes de programa necessários para construir um sistema computacional. Ele considera o estilo de arquitetura que o sistema assumirá, a estrutura e as propriedades dos componentes que constituem o sistema, bem como as inter-relações que ocorrem entre todos os componentes da arquitetura de um sistema.

Quem realiza? Embora um engenheiro de software possa projetar tanto os dados quanto a arquitetura, essa tarefa frequentemente é atribuída a especialistas quando são construídos sistemas grandes e complexos. Um projetista de bancos de dados ou de depósito de dados cria a arquitetura de dados para um sistema. O “arquiteto de sistemas” escolhe um estilo de arquitetura apropriado com base nos requisitos obtidos durante a análise de requisitos.

Por que é importante? Você não tentaria construir uma casa sem uma planta, não é mesmo? Também não desenharia as plantas começando pela distribuição dos encanamentos da casa. Deve-se partir do contexto geral – a casa em si – antes de

O que é? O projeto de arquitetura representa a estrutura de dados e os componentes de programa necessários para construir um sistema computacional.

se preocupar com os detalhes. É exatamente isso o que faz o projeto de arquitetura – ele dá uma visão geral e garante que você a entendeu corretamente.

Quais são as etapas envolvidas? O projeto de arquitetura começa pelo projeto de dados e então prossegue para a derivação de uma ou mais representações da estrutura da arquitetura do sistema. São analisados estilos ou padrões de arquitetura alternativos para se obter uma estrutura mais adequada aos requisitos do cliente e atributos de qualidade. Uma vez que se tenha escolhido uma alternativa, a arquitetura é elaborada usando-se um método de projeto de arquitetura.

Qual é o artefato? Durante o projeto de arquitetura, é criado um modelo que engloba a arquitetura de dados e a estrutura dos programas. Além disso, são descritas as propriedades e as relações (interações) entre os componentes.

Como garantir que o trabalho foi realizado corretamente? A cada estágio, são revisados os produtos resultantes do projeto de software em termos de clareza, correção, completude e consistência com os requisitos e entre si.

Conforme observamos no Capítulo 12, o projeto é baseado em informações. Os métodos de projeto de software são obtidos considerando-se cada um dos três domínios do modelo de análise. Os domínios de dados, funcional e comportamental servem de orientação para a criação do projeto de software.

Neste capítulo, serão apresentados os métodos necessários para criar “representações coerentes e bem planejadas” das camadas de dados e da arquitetura do modelo de projeto. O objetivo é fornecer uma abordagem sistemática para a obtenção do projeto de arquitetura – o esquema com a visão geral preliminar por meio do qual o software é construído.

projeto de arquitetura . . .	267
refinamento da	
arquitetura.	270
verificação de	
conformidade da	
arquitetura.	279

13.1 Arquitetura de software

Em seu livro referência sobre o assunto, Shaw e Garlan [Sha96] abordam a arquitetura de software da seguinte maneira:

Desde que o primeiro programa foi dividido em módulos, os sistemas de software passaram a ter arquiteturas e os programadores passaram a ser responsáveis pelas interações entre os módulos e as propriedades globais do conjunto. Historicamente, as arquiteturas têm sido implícitas – acidentes de implementação ou sistemas legados do passado. Os bons desenvolvedores de software muitas vezes adotam um ou vários padrões de arquitetura como estratégia para a organização de sistemas, mas usam esses padrões informalmente e não têm um meio para torná-los explícitos no sistema resultante.

Hoje, arquitetura de software eficiente e sua representação explícita tornaram-se temas dominantes em engenharia de software.

13.1.1 O que é arquitetura?

Quando consideramos a arquitetura de um edifício, vários atributos diferentes vêm à mente. No nível mais simplista, pensamos na forma geral da estrutura física; mas, na realidade, arquitetura é muito mais do que isso. Ela é a maneira pela qual os vários componentes do edifício são integrados para formar um todo coeso. É o como o edifício se ajusta ao seu ambiente e se integra a outros edifícios da vizinhança. É o grau com que o edifício atende a seu propósito declarado e satisfaz às necessidades de seu proprietário. É o sentido estético da estrutura – o impacto visual do edifício – e a maneira como texturas, cores e materiais são combinados para criar a fachada e o “ambiente de moradia”. É também os pequenos detalhes – o projeto de iluminação, o tipo de piso, o posicionamento de painéis... a lista é interminável. E, por fim, ela é arte.

Arquitetura também é algo mais. Ela é “milhares de decisões, tanto grandes quanto pequenas” [Tyr05]. Algumas dessas decisões são tomadas logo no início do projeto e podem ter um impacto profundo sobre todas as ações subsequentes. Outras são postergadas ao máximo, eliminando, portanto, restrições que levariam a uma implementação inadequada do estilo arquitetônico.

“A arquitetura de um sistema constitui um framework abrangente que descreve sua forma e estrutura – seus componentes e como eles se integram.”

Jerrold Grochow

A arquitetura de software deve modelar a estrutura de um sistema, bem como a maneira por meio da qual os dados e os componentes procedurais colaboram entre si.

"Case-se depressa com sua arquitetura, arrependa-se quando quiser."

Barry Boehm

E o que dizer da arquitetura de software? Bass, Clements e Kazman [Bas03] definem esse termo difícil de descrever da seguinte maneira:

A arquitetura de software de um programa ou sistema computacional é a estrutura, ou estruturas, do sistema, o que abrange os componentes de software, as propriedades externamente visíveis desses componentes e as relações entre eles.

A arquitetura não é o software operacional. É uma representação que nos permite (1) analisar a efetividade do projeto no atendimento dos requisitos declarados, (2) considerar alternativas de arquitetura em um estágio em que fazer mudanças de projeto ainda é relativamente fácil e (3) reduzir os riscos associados à construção do software.

Essa definição enfatiza o papel dos “componentes de software” em qualquer representação de arquitetura. No contexto do projeto de arquitetura, um componente de software pode ser algo tão simples quanto um módulo de programa ou uma classe orientada a objetos, porém também pode ser ampliado para abranger bancos de dados e “middleware” que possibilitem a configuração de uma rede de clientes e servidores. As propriedades dos componentes são as características necessárias para o entendimento de como eles interagem com outros componentes. No nível da arquitetura, não são especificadas as propriedades internas (por exemplo, detalhes de um algoritmo). As relações entre componentes podem ser tão simples quanto a chamada procedural de um módulo a outro ou tão complexo quanto um protocolo de acesso a banco de dados.

Alguns membros da comunidade da engenharia de software (por exemplo, [Kaz03]) fazem distinção entre as ações associadas à obtenção de uma arquitetura de software (aquilo que denominamos “projeto de arquitetura”) e as ações aplicadas para obter o projeto de software. Conforme observado por um dos revisores de uma edição passada:

Há uma diferença marcante entre os termos *arquitetura* e *projeto*. *Projeto* é uma instância de uma *arquitetura*, da mesma forma que um objeto é uma instância de uma classe. Considere, por exemplo, a arquitetura cliente/servidor. Podemos projetar um sistema de software centralizado em redes de várias formas diferentes por meio dessa arquitetura, usando a plataforma Java (Java EE) ou a plataforma Microsoft (.NET framework). Existe uma arquitetura, porém vários projetos podem ser criados baseados nela. Consequentemente, não podemos confundir “arquitetura” com “projeto”.

Embora concordemos que um projeto de software seja uma instância de uma arquitetura de software específica, os elementos e estruturas definidos como parte de uma arquitetura são a raiz de todo projeto. Um projeto se inicia com uma consideração de arquitetura.

13.1.2 Por que a arquitetura é importante?

Em um livro dedicado à arquitetura de software, Bass e seus colegas [Bas03] identificaram três razões principais por que a arquitetura de software é importante:

- A arquitetura de software fornece uma representação que facilita a comunicação entre todos os envolvidos.

Links úteis para diversos sites sobre arquitetura de software podem ser encontrados em <http://www.ewita.com/links/softwareArchitectureLinks.htm>.

- A arquitetura destaca desde o início as decisões de projeto que terão um profundo impacto no trabalho de engenharia de software que se segue.
- A arquitetura “constitui um modelo relativamente pequeno e intelectualmente compreensível de como o sistema é estruturado e como seus componentes trabalham em conjunto” [Bas03].

O modelo de projeto de arquitetura e os padrões de arquitetura nele contidos são transferíveis. Gêneros, estilos e padrões de arquitetura (Seções 13.2 a 13.6) podem ser aplicados ao projeto de outros sistemas e representam um conjunto de abstrações que permitem aos engenheiros de software descrever a arquitetura de modo previsível.

O modelo de arquitetura fornece uma visão gestáltica do sistema, permitindo ao engenheiro de software examiná-lo como um todo.

13.1.3 Descrições de arquitetura

Todos nós temos uma imagem mental daquilo que a palavra *arquitetura* significa. Assim, os diferentes envolvidos verão uma arquitetura sob diferentes pontos de vista, orientados por diferentes conjuntos de interesses. Isso implica que uma descrição de arquitetura é, na verdade, um conjunto de artefatos que refletem diferentes visões do sistema.

Smolander, Rossi e Puraol [Smo08] identificaram várias metáforas, representando diferentes visões da mesma arquitetura, que os envolvidos utilizam para compreender o termo *arquitetura de software*. A *metáfora do esquema* com a visão geral parece ser a mais conhecida dos envolvidos que escrevem programas para implementar um sistema. Os desenvolvedores consideram as descrições da arquitetura como um modo de transferir informações explícitas dos arquitetos para os projetistas e engenheiros de software encarregados de produzir os componentes do sistema. A *metáfora da linguagem* vê a arquitetura como facilitadora da comunicação entre grupos de envolvidos. Essa visão é a preferida dos envolvidos que se concentram nos clientes (por exemplo, gerentes ou especialistas em marketing). A descrição arquitetural precisa ser concisa e fácil de entender, pois forma a base da negociação, particularmente na determinação dos limites do sistema.

A *metáfora da decisão* representa a arquitetura como o produto das decisões que envolve o balanceamento* entre propriedades como custo, usabilidade, facilidade de manutenção e desempenho. Cada uma dessas propriedades pode ter um impacto significativo sobre o projeto do sistema. Os envolvidos (por exemplo, gerentes de projeto) veem as decisões arquiteturais como a base para alocar recursos de projeto e tarefas para o trabalho. Essas decisões podem afetar a sequência das tarefas e a estrutura da equipe de software. A *metáfora da literatura* é usada para documentar as soluções arquiteturais construídas no passado. Essa visão suporta a construção de artefatos e a transferência de conhecimento entre projetistas e o pessoal de manutenção do software. Suporta também os envolvidos cuja preocupação é a reutilização de componentes e projetos.

* N. de R.T.: Balanceamento (*trade-off*) representa o equilíbrio entre um conjunto de fatores que não são totalmente atingíveis simultaneamente.

A descrição arquitetural de um sistema baseado em software deve exibir características que combinem essas metáforas. Tyree e Akerman [Tyr05] citam isso ao escreverem:

Os desenvolvedores desejam orientação clara e determinada sobre como prosseguir com um projeto. Os clientes querem um entendimento claro das mudanças que devem ocorrer no ambiente e garantias de que a arquitetura atenderá às suas necessidades de negócio. Outros arquitetos buscam um entendimento claro dos aspectos mais importantes da arquitetura.

Cada um desses “desejos” se reflete em uma metáfora diferente, representada sob um ponto de vista diferente.

A IEEE Computer Society propôs a norma IEEE-Std-1471-2000, *Recommended Practice for Architectural Description of Software-Intensive Systems*, [IEE00], com os seguintes objetivos: (1) estabelecer um framework conceitual e um vocabulário para uso durante o projeto de arquitetura de software, (2) fornecer diretrizes detalhadas para representar uma descrição da arquitetura e (3) encorajar práticas de projeto de arquitetura consistentes. Uma *descrição arquitetural* (DA) representa várias visões – cada visão é “a representação de um sistema como um todo, segundo a perspectiva de um conjunto de necessidades relacionadas aos envolvidos”.

13.1.4 Decisões de arquitetura

Cada visão desenvolvida como parte da descrição arquitetural trata de uma necessidade específica do envolvidos. Para desenvolver cada visão (e a descrição arquitetural como um todo), o arquiteto de sistemas considera uma variedade de alternativas e, por fim, decide sobre as características de uma arquitetura específica que melhor atendam à necessidade. Consequentemente, as próprias decisões de arquitetura podem ser consideradas uma visão de arquitetura. As razões pelas quais as decisões foram tomadas fornecem uma visão sobre a estrutura de um sistema e sua adequação às necessidades dos envolvidos.

Como arquiteto de sistemas, você pode usar o modelo (template) sugerido no quadro da página ao lado para documentar todas as decisões importantes. Desse modo, você fornece os fundamentos para o seu trabalho e estabelece um registro histórico que pode ser útil quando mudanças de projeto tiverem de ser feitas.

Grady Booch [Boo11a] escreve que, ao começar a construir um produto inovador, muitas vezes os engenheiros de software se sentem obrigados a lançar-se imediatamente ao trabalho, construir coisas, corrigir o que não funciona, melhorar o que funciona e, então, repetir o processo. Depois de fazer isso algumas vezes, eles começam a reconhecer que uma arquitetura deve ser definida e as decisões associadas às escolhas arquiteturais devem ser declaradas explicitamente. Talvez não seja possível prever as escolhas corretas antes de construir um novo produto. Contudo, se os inovadores acharem que vale a pena repetir as decisões arquiteturais após testarem seus protótipos no campo, então pode começar a surgir um *projeto dominante*¹ para esse tipo de

¹ O *projeto dominante* descreve uma arquitetura ou processo de software inovador que se torna um padrão do setor após um período de adaptação e uso bem-sucedidos no mercado.

INFORMAÇÕES



Template de descrição de decisões de arquitetura

Cada decisão de arquitetura importante pode ser documentada para posterior revisão pelos envolvidos que queiram entender a descrição da arquitetura proposta. O modelo apresentado neste quadro é uma versão resumida e adaptada de um gabarito proposto por Tyree e Ackerman [Tyr05].

Problema de projeto:	Descreva os problemas de projeto de arquitetura que devem ser tratados.	Alternativas:	Descreva brevemente as alternativas de projeto de arquitetura consideradas e por que foram rejeitadas.
Resolução:	Informe a abordagem escolhida para tratar o problema de projeto de arquitetura.	Argumento:	Explique por que escolheu a decisão em detrimento das alternativas.
Categoria:	Especifique a categoria de projeto que o problema e a resolução tratam (por exemplo, projeto de dados, estrutura de conteúdo, estrutura de componentes, integração, apresentação).	Implicações:	Indique as consequências de projeto ao tomar a decisão. Como a resolução afeta outras questões do projeto de arquitetura? A resolução vai restringir o projeto de alguma forma?
Hipóteses:	Indique quaisquer hipóteses que ajudem a dar forma à decisão.	Decisões relacionadas:	Que outras decisões documentadas estão relacionadas a essa decisão?
Restrições:	Especifique quaisquer restrições do ambiente que auxiliaram a dar forma à decisão (por exemplo, padrões de tecnologia, padrões disponíveis, questões relacionadas ao projeto).	Necessidades relacionadas:	Que outros requisitos estão relacionados a essa decisão?
		Artefatos:	Indique onde essa decisão vai se refletir na descrição da arquitetura.
		Notas:	Faça referência a quaisquer observações feitas pela equipe ou outra documentação utilizada para tomar a decisão.

produto. Sem a documentação do que funcionou e do que não funcionou, é difícil para os engenheiros de software decidirem quando devem inovar e quando devem usar uma arquitetura criada anteriormente.

13.2 Gêneros de arquitetura

Embora os princípios fundamentais do projeto de arquitetura se apliquem a todos os tipos de arquitetura, o *gênero* arquitetural normalmente ditará a abordagem arquitetural específica para a estrutura que deve ser construída. No contexto de projeto de arquitetura, *gênero* implica uma categoria específica no domínio de software geral. Em cada categoria, pode-se encontrar uma série de subcategorias. Por exemplo, dentro do gênero *edifícios*, poderíamos encontrar os seguintes estilos gerais: casas, condomínios, prédios de apartamentos, conjuntos comerciais, prédios industriais, armazéns e assim por diante. Em cada estilo geral poderiam ser aplicados estilos mais específicos (Seção 13.3). Cada estilo teria uma estrutura que pode ser descrita usando-se um conjunto de padrões previsíveis.

Em seu *Handbook of Software Architecture* [Boo08], Grady Booch sugere os seguintes gêneros arquiteturais para sistemas baseados em software, incluindo: inteligência artificial, comunicação, dispositivos, finanças, games, industrial, jurídico, médico, militar, sistemas operacionais, transportes e utilitários, dentre muitos outros.

Há uma série de estilos de arquitetura diferentes que poderiam ser aplicados a um gênero específico (também denominado domínio de aplicação).

“Por trás da mente de qualquer artista existe um padrão ou tipo de arquitetura.”

G. K. Chesterton

O que é estilo de arquitetura?

Estilos de arquitetura baseados em atributos (sigla ABAS) podem ser utilizados como blocos básicos para as arquiteturas de software. Informações sobre isso podem ser obtidas em www.sei.cmu.edu/architecture/abas.html.

13.3 Estilos de arquitetura

Quando um arquiteto usa a expressão “estilo colonial americano com hall central” para descrever uma casa, a maioria das pessoas familiarizadas com casas dos Estados Unidos será capaz de evocar uma imagem geral da aparência da casa e como provavelmente será a sua planta. O arquiteto usou um *estilo arquitetural* como um mecanismo descritivo para diferenciar a casa de outros estilos (por exemplo, casa pré-fabricada sobre uma estrutura de madeira em forma de A, rancho montado sobre uma base elevada, *Cape Cod*). Porém, mais importante ainda, o estilo arquitetural também é um modelo para construção. É preciso definir mais detalhes da casa, especificar suas dimensões finais, características personalizadas podem ser acrescentadas, também devem ser determinados os materiais de construção, porém o estilo – uma casa “colonial americano com hall central” – orienta o arquiteto em seu trabalho.

O software criado para sistemas computacionais também apresenta um estilo de arquitetura. Cada estilo descreve uma categoria de sistema que engloba (1) um conjunto de componentes (por exemplo, um banco de dados, módulos computacionais) que realiza uma função exigida por um sistema, (2) um conjunto de conectores que habilitam a “comunicação, coordenação e cooperação” entre os componentes, (3) restrições que definem como os componentes podem ser integrados para formar o sistema e (4) modelos semânticos que permitem a um projetista compreender as propriedades gerais de um sistema por meio da análise das propriedades conhecidas de suas partes constituintes [Bas03].

Um estilo arquitetural é uma transformação imposta ao projeto de um sistema inteiro. O objetivo é estabelecer uma estrutura para todos os componentes do sistema. No caso em que uma arquitetura existente deve sofrer um processo de reengenharia (Capítulo 36), a imposição de um estilo de arquitetura resultará em mudanças fundamentais na estrutura do software, incluindo uma nova atribuição da funcionalidade dos componentes [Bos00].

Um padrão de arquitetura, assim como um estilo arquitetural, impõe uma transformação no projeto de arquitetura. Entretanto, padrão difere de estilo em alguns modos fundamentais: (1) o escopo de um padrão é menos abrangente, concentrando-se em um aspecto da arquitetura e não na arquitetura em sua totalidade, (2) um padrão impõe uma regra sobre a arquitetura, descrevendo como o software vai tratar algum aspecto de sua funcionalidade em termos de infraestrutura (por exemplo, concomitância) [Bos00], (3) os padrões de arquitetura (Seção 13.3.2) tendem a tratar de questões comportamentais específicas no contexto da arquitetura (por exemplo, como as aplicações em tempo real tratam a sincronização ou as interrupções). Os padrões podem ser usados com um estilo de arquitetura para dar forma à estrutura global de um sistema.

13.3.1 Uma breve taxonomia dos estilos de arquitetura

Embora milhões de sistemas computacionais tenham sido criados nos últimos 60 anos, a vasta maioria pode ser classificada em um número relativamente pequeno de estilos de arquitetura:

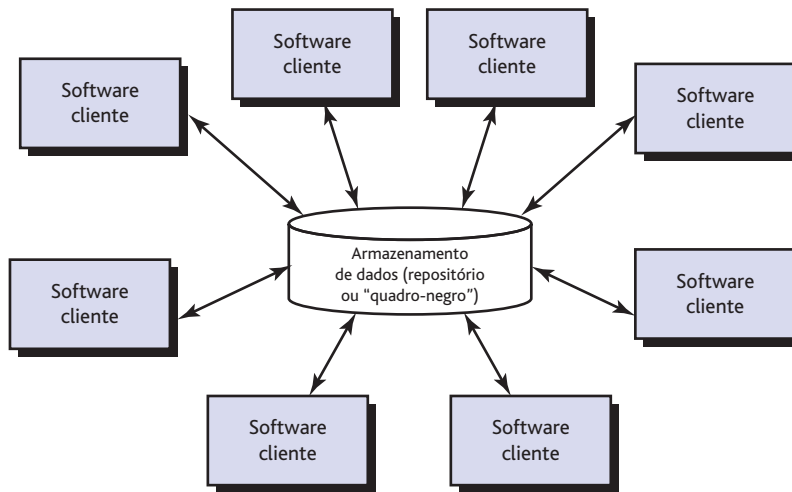


FIGURA 13.1 Arquitetura centralizada em dados.

Arquiteturas centralizadas em dados. Um repositório de dados (por exemplo, um arquivo ou banco de dados) reside no centro dessa arquitetura e é, em geral, acessado por outros componentes que atualizam, acrescentam, eliminam ou modificam de alguma outra maneira os dados contidos no repositório. A Figura 13.1 ilustra um estilo centralizado em dados típico. O software cliente acessa um repositório central. Em alguns casos o repositório de dados é passivo. Ou seja, o software cliente acessa os dados independentemente de quaisquer alterações nos dados ou das ações de outros softwares clientes. Uma variação dessa abordagem transforma o repositório em um “quadro-negro” que envia notificações ao software cliente quando os dados de seu interesse mudam.

As arquiteturas centralizadas em dados promovem a *integrabilidade* [Bas03]. Isto é, componentes existentes podem ser alterados e novos componentes clientes acrescentados à arquitetura sem se preocupar com outros clientes (pois os componentes clientes operam independentemente). Além disso, dados podem ser passados entre os clientes usando o mecanismo de quadro-negro (o componente quadro-negro serve para coordenar a transferência de informações entre os clientes). Os componentes clientes executam processos de maneira independente.

Arquiteturas de fluxo de dados. Essa arquitetura se aplica quando dados de entrada devem ser transformados por meio de uma série de componentes computacionais ou de manipulação em dados de saída. Um padrão tubos-e-filtro (Figura 13.2) tem um conjunto de componentes, denominado *filtros*, conectados por *tubos* que transmitem dados de um componente para o seguinte. Cada filtro trabalha de modo independente dos componentes que se encontram acima e abaixo deles, é projetado para esperar a entrada de dados de determinada forma e produz saída de dados (para o filtro seguinte) da forma especificada. Entretanto, o filtro não precisa conhecer o funcionamento interno de seus filtros vizinhos.

Se o fluxo de dados ocorre em uma única linha de transformações, ele é denominado *sequencial por lotes*. Essa estrutura aceita um lote de dados e aplica uma série de componentes sequenciais (filtros) para transformá-lo.

“O uso de padrões e estilos de projeto é universal nas disciplinas de engenharia.”

**Mary Shaw e
David Garlan**

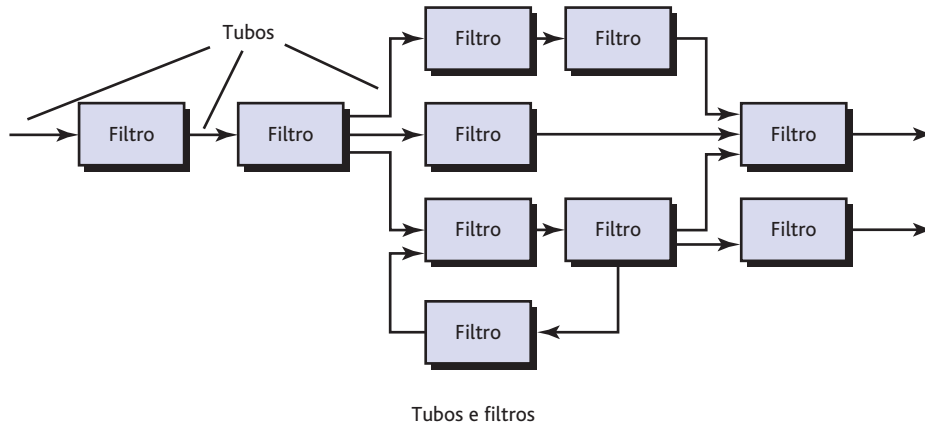


FIGURA 13.2 Arquitetura de fluxo de dados.

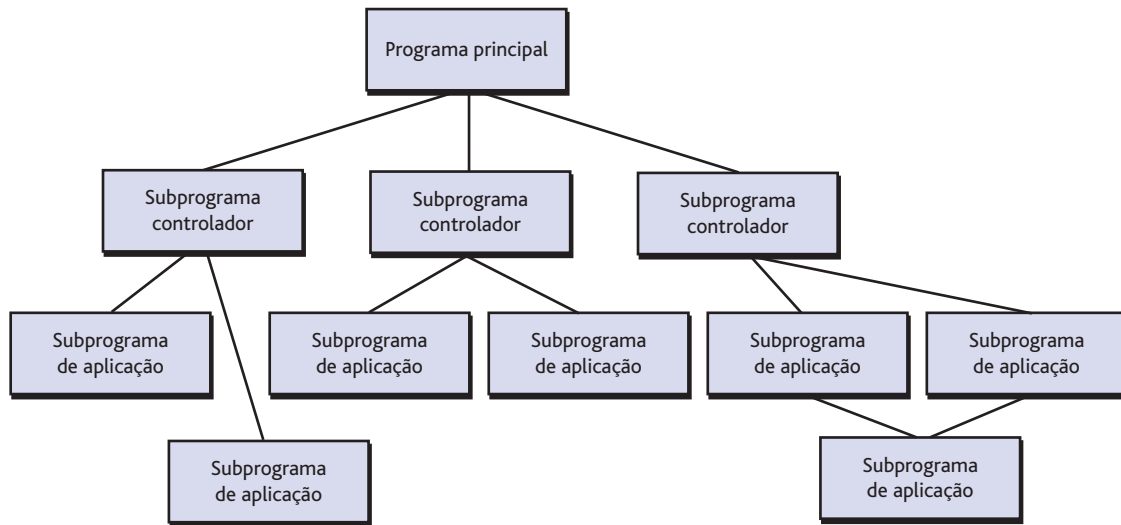


FIGURA 13.3 Arquitetura de programa principal/subprograma.

Arquiteturas de chamadas e retornos. Esse estilo de arquitetura permite-nos obter uma estrutura de programa relativamente fácil de modificar e aumentar. Existe uma série de subestilos [Bas03] dentro dessa categoria:

- *Arquiteturas de programa principal/subprograma.* Essa clássica estrutura de programas decompõe a função em uma hierarquia de controle na qual um programa “principal” invoca uma série de componentes de programa que, por sua vez, pode invocar outros. A Figura 13.3 ilustra uma arquitetura desse tipo.
- *Arquiteturas de chamadas a procedimentos remotos.* Os componentes de uma arquitetura de programa principal/subprograma são distribuídos ao longo de vários computadores em uma rede.

Arquiteturas orientadas a objetos. Os componentes de um sistema encapsulam dados e as operações que devem ser aplicadas para manipular os dados.

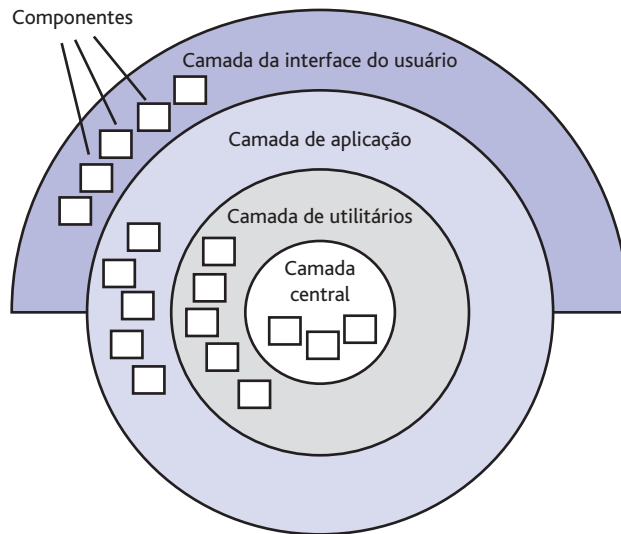


FIGURA 13.4 Arquitetura em camadas.

A comunicação e a coordenação entre componentes são realizadas por meio da passagem de mensagens.

Arquiteturas em camadas. A estrutura básica de uma arquitetura em camadas é ilustrada na Figura 13.4. São definidas várias camadas diferentes, cada uma realizando operações que progressivamente se tornam mais próximas do conjunto de instruções de máquina. Na camada mais externa, os componentes atendem às operações da interface do usuário. Na camada mais interna, fazem a interface com o sistema operacional. As camadas intermediárias fornecem serviços utilitários e funções de software de aplicação.

Os estilos de arquitetura apresentados são apenas um pequeno subconjunto dos estilos disponíveis.² Assim que a engenharia de requisitos revelar as características e restrições do sistema a ser construído, o estilo e/ou combinação de padrões de arquitetura que melhor se encaixar nessas características e restrições pode ser escolhido. Em muitos casos, mais de um padrão poderia ser apropriado, e estilos de arquitetura alternativos podem ser projetados e avaliados. Por exemplo, um estilo em camadas (apropriado para a maioria dos sistemas) pode ser combinado com uma arquitetura centralizada em dados em diversas aplicações de bancos de dados.

Pode ser complicado escolher o estilo arquitetural correto. Buschman [Bus10a] sugere dois conceitos complementares que podem oferecer alguma orientação. *Frames do problema (problem frames)* descrevem as características de problemas recorrentes, sem serem perturbadas por referências a detalhes do conhecimento do domínio ou pela programação de implementações da solução. O *projeto orientado a domínios* sugere que o projeto de software deve refletir o domínio e a lógica do problema de negócio que se deseja resolver com a aplicação (Capítulo 8).

² Consulte [Roz11], [Tay09], [Bus07], [Gor06] ou [Bas03] para uma discussão detalhada sobre padrões e estilos de arquitetura.



Escolha de um estilo de arquitetura

Cena: Sala do Jamie, quando é iniciada a modelagem de projeto.

Atores: Jamie e Ed – membros da equipe de engenharia de software do CasaSegura.

Conversa:

Ed (franzindo a testa): Modelamos a função de segurança usando UML... você sabe, classes, relações, esse tipo de coisas. Portanto, imagino que a arquitetura³ orientada a objetos seja o caminho a seguirmos.

Jamie: Mas...?

Ed: Mas... tenho dificuldade em visualizar o que é uma arquitetura orientada a objetos. Entendo a arquitetura de chamadas e retornos, uma espécie de hierarquia de processos convencional, mas orientada a objetos... eu não sei, ela parece um tanto amorfa.

Jamie (sorrindo): Amorfa, é?

Ed: Isso mesmo... o que eu quis dizer é que não consigo visualizar uma estrutura real, apenas classes de projeto fluindo no ar.

Jamie: Bem, isso não é verdade. Existem hierarquias de classes... pense na hierarquia (agregação) que fizemos para o objeto **Planta** [Figura 12.3]. Uma arquitetura orientada a objetos é uma combinação daquela estrutura e as interconexões – sabe, colaborações – entre as classes. Podemos mostrá-la descrevendo completamente os atributos e operações, a troca de mensagens que ocorre e a estrutura das classes.

Ed: Vou gastar uma hora mapeando uma arquitetura de chamadas e retornos; então voltarei e considerarei uma arquitetura orientada a objetos.

Jamie: Doug não terá nenhum problema com isso. Ele me disse que deveríamos considerar alternativas de arquitetura. Por sinal, não há absolutamente nenhuma razão para que essas duas arquiteturas não possam ser usadas de forma combinada.

Ed: Bom. Eu concordo.

Um *frame do problema* é a generalização de uma classe de problemas que pode ser usado para resolver o problema à mão. Existem cinco frames de problema fundamentais e, muitas vezes, estão associados aos estilos arquiteturais: peças de trabalho simples (ferramentas), comportamento exigido (centralizado nos dados), comportamento comandado (processador de comandos), exibição de informações (observador) e transformação (variações de tubo e filtro).

Frequentemente, os problemas do mundo real seguem mais de um frame de problema e, como consequência, um modelo arquitetural pode ser uma combinação de diferentes frames. Por exemplo, a arquitetura modelo-visão-controlador (MVC) utilizada no projeto de WebApp⁴ poderia ser visto como a combinação de dois frames de problema (comportamento comandado e exibição de informações). Na arquitetura MVC, o comando do usuário é enviado da janela do navegador para um processador de comandos (controlador), o qual gerencia o acesso ao conteúdo (modelo) e instrui o modelo de renderização* de informações (visão) a transformá-lo para exibição pelo software do navegador.

³ A arquitetura do CasaSegura deve ser considerada em um nível mais elevado do que a arquitetura citada. O CasaSegura possui uma série de subsistemas – funcionalidade de monitoramento da casa, o site de monitoramento da empresa e o subsistema executado no PC do proprietário do imóvel. Nos subsistemas, processos concorrentes (por exemplo, aqueles para monitorar sensores) e tratamento de eventos são frequentes. Algumas decisões em relação à arquitetura nesse nível são tomadas durante a engenharia de produto, porém o projeto de arquitetura na engenharia de software deve considerar muito bem essas questões.

⁴ A arquitetura MVC é vista em mais detalhes no Capítulo 17.

* N. de R.T.: Tradução de *rendering*.

A modelagem do domínio pode influenciar a escolha de estilo arquitetural, particularmente as propriedades básicas dos objetos do domínio. Os objetos do domínio que representam objetos físicos (por exemplo, sensores ou unidades de disco) devem ser tratados diferentemente daqueles que representam objetos lógicos (por exemplo, agendas ou fluxos de trabalho). Os objetos físicos devem obedecer a rigorosas restrições, como limitações da conexão ou o uso de recursos consumíveis. Os objetos lógicos podem ter comportamento em tempo real mais suave, que pode ser cancelado ou desfeito. Muitas vezes, o projeto orientado a domínios é mais bem suportado por um estilo de arquitetura em camadas [Eva04].

13.3.2 Padrões de arquitetura

Conforme o modelo de requisitos for sendo desenvolvido, você poderá perceber que o software deve tratar de problemas mais amplos que envolvem toda a aplicação. Por exemplo, o modelo de requisitos para praticamente qualquer aplicação de comércio eletrônico se depara com o seguinte problema: *como oferecer uma ampla variedade de produtos para muitos clientes diferentes e permitir que esses clientes comprem nossos artigos online?*

O modelo de requisitos também define um contexto no qual essa questão deve ser respondida. Por exemplo, uma aplicação de comércio eletrônico que vende equipamentos de golfe para clientes vai operar em um contexto diferente daquele de uma aplicação de comércio eletrônico que vende equipamentos industriais de preço elevado para empresas de médio e grande porte. Além disso, um conjunto de limitações e restrições pode afetar a forma de tratarmos o problema a ser resolvido.

Os padrões de arquitetura lidam com um problema específico de aplicação em um contexto específico e sob um conjunto de limitações e restrições. O padrão propõe uma solução de arquitetura capaz de servir como base para o projeto de arquitetura.

Falamos anteriormente neste capítulo que a maioria das aplicações enquadra-se em um domínio ou gênero específico e que um ou mais estilos de arquitetura poderiam ser apropriados para aquele gênero. Por exemplo, o estilo de arquitetura geral para uma aplicação poderia ser de chamadas e retornos ou orientado a objetos. Porém, nesse estilo, encontraremos um conjunto de problemas comuns que poderiam ser mais bem tratados com padrões de arquitetura específicos. Alguns desses problemas e uma discussão mais completa sobre padrões de arquitetura são apresentados no Capítulo 16.

13.3.3 Organização e refinamento

Como o processo de projeto muitas vezes permite várias alternativas de arquitetura, é importante estabelecer um conjunto de critérios de projeto que possam ser usados para avaliar o projeto de arquitetura obtido. As seguintes questões [Bas03] dão uma visão mais clara sobre um estilo de arquitetura:

Controle. Como o controle é gerenciado na arquitetura? Existe uma hierarquia de controle distinta e, em caso positivo, qual o papel dos componentes nessa hierarquia de controle? Como os componentes transferem controle no sistema? Como o controle é compartilhado entre os compo-

*"Talvez esteja no porão.
Vou subir e verificar."*

M. C. Escher

Como avaliar um
estilo de arquitetura
derivado?

nentes? Qual a topologia de controle (ou seja, a forma geométrica que o controle assume)? O controle é sincronizado ou os componentes operam de maneira assíncrona?

Dados. Como os dados são transmitidos entre os componentes? O fluxo de dados é contínuo ou os objetos de dados são passados esporadicamente para o sistema? Qual o modo de transferência de dados (ou seja, os dados são passados de um componente para outro ou os dados estão disponíveis globalmente para serem compartilhados entre os componentes do sistema)? Existem componentes de dados (por exemplo, um quadro-negro ou repositório) e, em caso positivo, qual o seu papel? Como os componentes funcionais interagem com os componentes de dados? Os componentes de dados são passivos ou ativos (isto é, o componente de dados interage ativamente com outros componentes do sistema)? Como os dados e controle interagem no sistema?

Essas questões permitem ao projetista fazer uma avaliação prévia da qualidade do projeto e formam a base para uma análise mais detalhada da arquitetura.

Modelos de processo evolutivos (Capítulo 4) se tornaram muito populares. Isso significa que as arquiteturas de software talvez precisem evoluir à medida que cada incremento do produto for planejado e implementado. No Capítulo 12, descrevemos esse processo como refatoração – melhorar a estrutura interna do sistema sem alterar seu comportamento externo.

13.4 Considerações sobre a arquitetura

Buschmann e Henny [Bus10b, Bus10c] sugerem várias considerações que podem oferecer orientações aos engenheiros de software quando são tomadas decisões sobre a arquitetura.

Quais questões devo considerar ao desenvolver uma arquitetura de software?

- **Economia** – Muitas arquiteturas de software padecem de complexidade desnecessária, motivada pela inclusão de recursos ou requisitos não funcionais desnecessários (por exemplo, capacidade de reutilização sem nenhum propósito). O melhor software é organizado e depende de abstração para reduzir os detalhes desnecessários.
- **Visibilidade** – Quando o modelo de projeto é criado, decisões sobre a arquitetura e as razões pelas quais foram tomadas devem ser óbvias para os engenheiros de software que examinarem o modelo posteriormente. Uma baixa visibilidade surge quando importantes conceitos de projeto e domínio são comunicados de forma deficiente àqueles que devem concluir o projeto e implementar o sistema.
- **Espaçamento** – A separação de preocupações em um projeto, sem a introdução de dependências ocultas, é um conceito de projeto desejável (Capítulo 12), às vezes referido como *espaçamento*. Espaçamento suficiente leva a projetos modulares, mas espaçamento demasiado leva à fragmentação e à perda de visibilidade. Métodos como o projeto orientado a domínios podem ajudar a identificar o que deve ser separado em um projeto e o que deve ser tratado como uma unidade coerente.

- **Simetria** – Simetria arquitetural significa que um sistema tem atributos consistentes e equilibrados. Projetos simétricos são mais fáceis de entender, compreender e comunicar. Como exemplo de simetria arquitetural, considere um objeto *conta de cliente* cujo ciclo de vida é modelado diretamente por uma arquitetura de software que exige os métodos *abrir()* e *fechar()*. A simetria arquitetural pode ser tanto estrutural quanto comportamental.
- **Emergência** – Comportamento e controle emergentes e auto-organizados frequentemente são o segredo da criação de arquiteturas de software expansíveis, eficientes e econômicas. Por exemplo, muitas aplicações de software de tempo real são orientadas a eventos. A sequência e a duração dos eventos que definem o comportamento do sistema é uma qualidade emergente. É muito difícil planejar toda sequência de eventos possível. Em vez disso, o arquiteto do sistema deve criar um sistema flexível que se adapte a esse comportamento emergente.

Essas considerações não são isoladas. Elas interagem umas com as outras e são moderadas por cada uma delas. Por exemplo, o espaçamento pode ser reforçado e reduzido pela economia. A visibilidade pode ser equilibrada pelo espaçamento.



Avaliação de decisões sobre a arquitetura

Cena: Sala do Jamie, à medida que a modelagem de projeto continua.

Atores: Jamie e Ed – membros da equipe de engenharia de software do *CasaSegura*.

Conversa:

Ed: Terminei meu modelo de arquitetura de chamadas e retornos da função de segurança.

Jamie: Ótimo! Acha que atende às nossas necessidades?

Ed: Ele não introduz características desnecessárias; portanto, parece econômico.

Jamie: E quanto à visibilidade?

Ed: Bem, eu entendo o modelo e não há problemas na implementação dos requisitos de segurança necessários para esse produto.

Jamie: Sei que você entende a arquitetura, mas talvez não seja você o programador dessa parte do projeto. Estou um pouco preocupado com o espaçamento. Talvez esse projeto não seja tão modular quanto um projeto orientado a objetos.

Ed: Talvez, mas isso pode limitar nossa capacidade de reutilizar parte de nosso código quando tivermos de criar a versão baseada na web desse *CasaSegura*.

Jamie: E quanto à simetria?

CASASEGURA

Ed: Bem, isso é mais difícil de avaliar. Parece-me que o único lugar para simetria na função de segurança é na adição e exclusão de informações de PIN.

Jamie: Isso vai ficar mais complicado quando adicionarmos recursos de segurança remotos ao produto baseado na web.

Ed: Suponho que seja verdade.

[Ambos fazem uma breve pausa, ponderando os problemas da arquitetura.]

Jamie: O *CasaSegura* é um sistema de tempo real; portanto, será difícil prever a transição de estados e a sequência de eventos.

Ed: É, mas o comportamento emergente desse sistema pode ser tratado com um modelo de estado finito.

Jamie: Como?

Ed: O modelo pode ser implementado com base na arquitetura de chamada e retorno. Interrupções podem ser facilmente tratadas em muitas linguagens de programação.

Jamie: Acha que precisamos fazer o mesmo tipo de análise para a arquitetura orientada a objetos que estivemos considerando inicialmente?

Ed: Suponho que possa ser uma boa ideia, porque é difícil mudar a arquitetura depois de iniciada a implementação.

Jamie: Nessas arquiteturas, também é importante mapearmos os requisitos não funcionais, além da segurança, para termos certeza de que foram completamente considerados.

Ed: Também é verdade.

A descrição da arquitetura de um produto de software não é explicitamente visível no código-fonte usado para implementá-la. Consequentemente, as modificações feitas no código com o passar do tempo (por exemplo, nas atividades de manutenção do software) podem causar uma lenta erosão na arquitetura do software. O desafio para o projetista é encontrar abstrações convenientes para as informações arquiteturais. Essas abstrações têm o potencial de adicionar uma estrutura que melhora a legibilidade e a facilidade de manutenção do código-fonte [Bro10b].

13.5 Decisões sobre a arquitetura

"Um médico pode enterrar seus erros, mas um arquiteto pode apenas aconselhar seu cliente a plantar videiras."

Frank Lloyd Wright

As decisões associadas à arquitetura do sistema capturam importantes problemas do projeto e o raciocínio por trás das soluções arquiteturais escolhidas. Algumas dessas decisões incluem a organização do sistema de software, a escolha dos elementos estruturais e suas interfaces, conforme definidas por suas colaborações, e a composição desses elementos em subsistemas cada vez maiores [Kru09]. Além disso, também podem ser feitas escolhas de padrões arquiteturais, tecnologias de aplicação, itens de middleware e linguagem de programação. O resultado das decisões sobre a arquitetura influencia as características não funcionais do sistema e muitos de seus atributos de qualidade [Zim11], podendo ser documentado com *anotações do desenvolvedor*. Essas anotações registram importantes decisões de projeto, junto com sua justificativa, fornecem uma referência para novos membros da equipe de projeto e servem como repositório de lições aprendidas.

Em geral, a prática da arquitetura de software se concentra nas visões arquiteturais que representam e documentam as necessidades de vários envolvidos. Contudo, é possível definir uma *visão da decisão* que encurte o caminho de várias visões de informações contidas nas representações arquiteturais tradicionais. A visão da decisão captura tanto as decisões de projeto de arquitetura quanto as razões que levaram à tomada da decisão.

A modelagem da *decisão de arquitetura orientada a serviços* (SOAD, *service-oriented architecture decision*)⁵ [Zim11] é um framework de gestão de conhecimento que fornece suporte para a captura de dependências da decisão de arquitetura, permitindo que elas guiem as futuras atividades de desenvolvimento.

Um *modelo de orientação* contém informações sobre as decisões de arquitetura exigidas ao se aplicar um estilo arquitetural em um gênero de aplicação em particular. Ele tem por base as informações arquiteturais obtidas a partir de projetos concluídos que empregaram o estilo de arquitetura nesse gênero. O modelo de orientação documenta lugares onde existem problemas de projeto e onde devem ser tomadas decisões sobre a arquitetura, junto com atributos de qualidade que devem ser considerados ao se fazer uma escolha dentre as possíveis alternativas. As soluções alternativas em potencial (com

⁵ A SOAD é parecida com o uso de padrões de arquitetura discutidos no Capítulo 16. Mais informações podem ser obtidas em: <http://soadecisions.org/soad.htm>.

seus prós e contras) de aplicações de software anteriores são incluídas para ajudar o arquiteto a tomar a melhor decisão possível.

O *modelo de decisão* documenta as decisões sobre arquitetura exigidas e registra as decisões realmente tomadas em projetos anteriores, com suas justificativas. O modelo de orientação alimenta o modelo de decisão da arquitetura em uma etapa de *personalização* que permite ao arquiteto excluir questões irrelevantes, aprimorar as importantes ou acrescentar novos problemas. Um modelo de decisão pode utilizar mais de um modelo de orientação e, depois que o projeto é concluído, fornece feedback para o modelo de orientação. Esse feedback pode ser obtido pela *coleta* de lições aprendidas a partir de análises feitas após o término do projeto.

13.6 Projeto de arquitetura

No início do projeto de arquitetura, deve-se estabelecer o contexto. Para isso, são descritas as entidades externas (por exemplo, outros sistemas, dispositivos, pessoas) que interagem com o software e a natureza de sua interação. De modo geral, essa informação pode ser obtida a partir do modelo de requisitos. Uma vez que o contexto é modelado e todas as interfaces de software externas foram descritas, podemos identificar um conjunto de arquétipos arquiteturais.

Arquétipo é uma abstração (similar a uma classe) que representa um elemento do comportamento do sistema. O conjunto de arquétipos fornece uma coleção de abstrações que deve ser modelada arquiteturalmente caso o sistema tenha de ser construído, porém os arquétipos em si não fornecem detalhes de implementação suficientes. Consequentemente, o projetista especifica uma estrutura de sistema por meio da definição e refinamento dos componentes de software que implementam cada arquétipo. Esse processo continua iterativamente até que uma estrutura de arquitetura completa tenha sido obtida.

Várias perguntas [Boo11b] devem ser formuladas e respondidas à medida que um engenheiro de software cria diagramas arquiteturais significativos. O diagrama mostra como o sistema responde às entradas ou aos eventos? Quais visualizações devem ser consideradas para ajudar a destacar áreas de risco? Como os padrões de projeto de sistema ocultos podem se tornar mais evidentes para outros desenvolvedores? Vários pontos de vista podem mostrar a melhor maneira de refatorar partes específicas do sistema? Os balanceamentos (*trade-offs*) do projeto podem ser representados de uma maneira significativa? Se uma representação esquemática da arquitetura do software responde a essas perguntas, ela terá valor para o engenheiro de software que a utilizar.

O que é um arquétipo?

13.6.1 Representação do sistema no contexto

No nível do projeto de arquitetura, um arquiteto de software usa um *diagrama de contexto arquitetural* (ACD, *architectural context diagram*) para modelar a maneira como o software interage com entidades externas às suas fronteiras. A estrutura genérica do diagrama de contexto arquitetural está ilustrada na Figura 13.5.

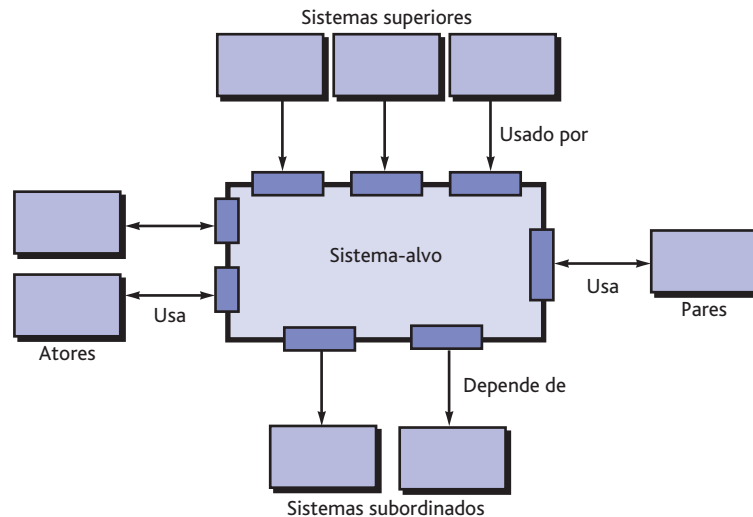


FIGURA 13.5 Diagrama de contexto arquitetural.
 Fonte: Adaptado de [Bos00].

Como os sistemas operam entre si?

Os sistemas que interoperam com o *sistema-alvo* (o sistema para o qual um projeto de arquitetura deve ser desenvolvido) são representados como:

- *Sistemas superiores* – sistemas que usam o sistema-alvo como parte de algum esquema de processamento de nível mais alto.
- *Sistemas subordinados* – sistemas que são utilizados pelo sistema-alvo e fornecem dados ou processamento necessários para completar a funcionalidade do sistema-alvo.
- *Sistemas de mesmo nível (pares)* – sistemas que interagem em uma base par-a-par (ou seja, as informações são produzidas ou consumidas pelos pares e pelo sistema-alvo).
- *Atores* – entidades (pessoas, dispositivos) que interagem com o sistema-alvo por meio da produção ou consumo de informações necessárias para o processamento.

Cada entidade externa se comunica com o sistema-alvo por meio de uma interface (os pequenos retângulos sombreados).

A título de ilustração do uso do ACD, considere a função de segurança residencial do produto *CasaSegura*. O controlador geral do produto *CasaSegura* e o sistema baseado na Internet são ambos superiores em relação à função de segurança, sendo mostrados acima da função na Figura 13.6. A função de vigilância é um *sistema de mesmo nível* e utiliza (é utilizado por) a função de segurança residencial em versões posteriores do produto. O proprietário do imóvel e os painéis de controle são os atores que produzem e consomem as informações usadas/produzidas pelo software de segurança. Por fim, são utilizados sensores pelo software de segurança e mostrados como subordinados a ele.

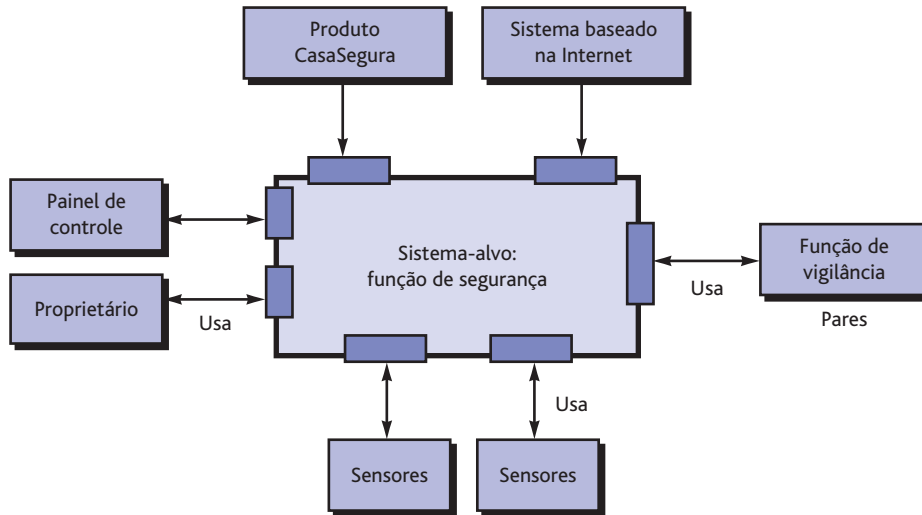


FIGURA 13.6 Diagrama de contexto arquitetural para a função de segurança do CasaSegura.

Como parte do projeto de arquitetura, os detalhes de cada interface da Figura 13.6 teriam de ser especificados. Todos os dados que fluem para dentro e para fora do sistema-alvo devem ser identificados nesse estágio.

13.6.2 Definição de arquétipos

Arquétipo é uma classe ou padrão que representa uma abstração que é crítica para o projeto de uma arquitetura para o sistema-alvo. Em geral, um conjunto relativamente pequeno de arquétipos é necessário para projetar até mesmo sistemas relativamente complexos. A arquitetura do sistema-alvo é composta por esses arquétipos, que representam elementos estáveis da arquitetura, porém podem ser instanciados de várias maneiras tomando como base o comportamento do sistema.

Em muitos casos, os arquétipos podem ser derivados examinando-se as classes de análise definidas como parte do modelo de requisitos. Prosseguindo com a discussão da função de segurança domiciliar do CasaSegura, poderíamos definir os seguintes arquétipos:

- **Nó.** Representa um conjunto coeso de elementos de entrada e saída da função de segurança domiciliar. Por exemplo, um nó poderia ser composto de (1) vários sensores e (2) uma variedade de indicadores de alarme (saída).
- **Detector.** Abstração que engloba todos os equipamentos de sensoriamento que alimentam o sistema-alvo com informações.
- **Indicador.** Abstração que representa todos os mecanismos (por exemplo, sirene de alarme, luzes intermitentes, campainha) para indicar a ocorrência de uma condição de alarme.
- **Controlador.** Abstração que representa o mecanismo que permite armar ou desarmar um nó. Se os controladores residem em uma rede, eles têm a capacidade de se comunicar entre si.

Arquétipos são os blocos básicos abstratos de um projeto de arquitetura.

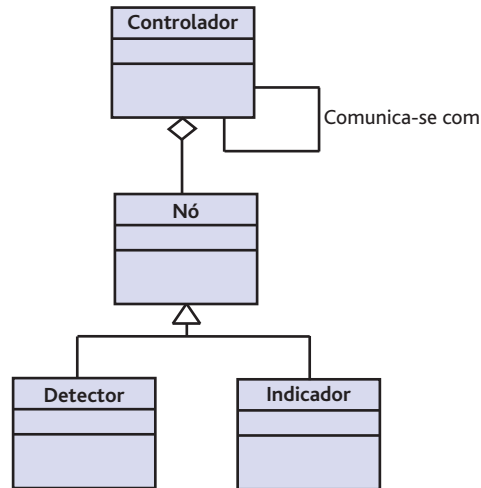


FIGURA 13.7 Relacionamentos em UML para os arquétipos da função de segurança do *CasaSegura*.

Fonte: Adaptado de [Bos00].

Cada um desses arquétipos é representado usando-se a notação UML, conforme indicado na Figura 13.7. Recorde-se que os arquétipos formam a base para a arquitetura, mas são as abstrações que devem ser refinadas à medida que o projeto de arquitetura prossegue. Por exemplo, **Detector** poderia ser refinado em uma hierarquia de classes de sensores.

13.6.3 Refinamento da arquitetura em componentes

Conforme a arquitetura de software é refinada em componentes, a estrutura do sistema começa a emergir. Mas como os componentes são escolhidos? Para responder a essa pergunta, começamos pelas classes descritas como parte do modelo de requisitos.⁶ Essas classes de análise representam entidades no domínio de aplicação que devem ser tratadas na arquitetura do software. Portanto, o domínio de aplicação é uma fonte para derivação e refinamento de componentes. Outra fonte é o domínio da infraestrutura. A arquitetura deve acomodar muitos componentes de infraestrutura que permitem componentes de aplicação, mas que não têm nenhuma relação de negócio com o domínio de aplicação. Por exemplo, componentes de gerenciamento de memória, componentes de comunicação, componentes de bancos de dados e componentes de gerenciamento de tarefas em geral são integrados à arquitetura do software.

As interfaces representadas no diagrama de contexto arquitetural (Seção 13.6.1) implicam um ou mais componentes especializados que processam os dados que fluem pela interface. Em alguns casos (por exemplo, uma interface gráfica do usuário), tem de ser projetada uma arquitetura de subsistemas completa, com vários componentes.

"A estrutura de um sistema de software fornece o meio no qual o código nasce, amadurece e morre. Um habitat bem projetado possibilita a evolução bem-sucedida de todos os componentes necessários em um sistema de software."

R. Pattis

⁶ Se for escolhida uma abordagem convencional (não orientada a objetos), os componentes poderão ser extraídos da hierarquia de chamada de subprogramas (consulte a Figura 13.3).

Continuando com o exemplo da função de segurança domiciliar do *CasaSegura*, poderíamos definir o conjunto de componentes de alto nível que trata da seguinte funcionalidade:

- *Gerenciamento da comunicação externa* – coordena a comunicação da função de segurança com entidades externas, como sistemas baseados na Internet e notificação externa de alarme.
- *Processamento de painel de controle* – gerencia toda a funcionalidade do painel de controle.
- *Gerenciamento de detectores* – coordena o acesso a todos os detectores conectados ao sistema.
- *Processamento de alarme* – verifica e atua sobre todas as condições de alarme.

Cada um dos componentes de alto nível teria de ser elaborado de forma iterativa e então posicionado na arquitetura global do *CasaSegura*. Para cada um deles, seriam definidas classes de projeto (com atributos e operações apropriados). É importante notar, no entanto, que os detalhes de projeto de todos os atributos e operações não seriam especificados até se chegar ao projeto de componentes (Capítulo 14).

A Figura 13.8 mostra a estrutura global da arquitetura (representada na forma de um diagrama de componentes UML). As transações são capturadas pelo *gerenciamento de comunicação externa*, à medida que se deslocam de componentes que processam a interface gráfica do usuário do *CasaSegura* e a interface com a Internet. Tais informações são gerenciadas por um componente executivo do *CasaSegura* que seleciona a função do produto apropriada (neste caso, segurança). O componente *processamento do painel de controle* interage com o proprietário do imóvel para armar/desarmar a função de segurança. O componente *gerenciamento de detectores* faz uma sondagem nos

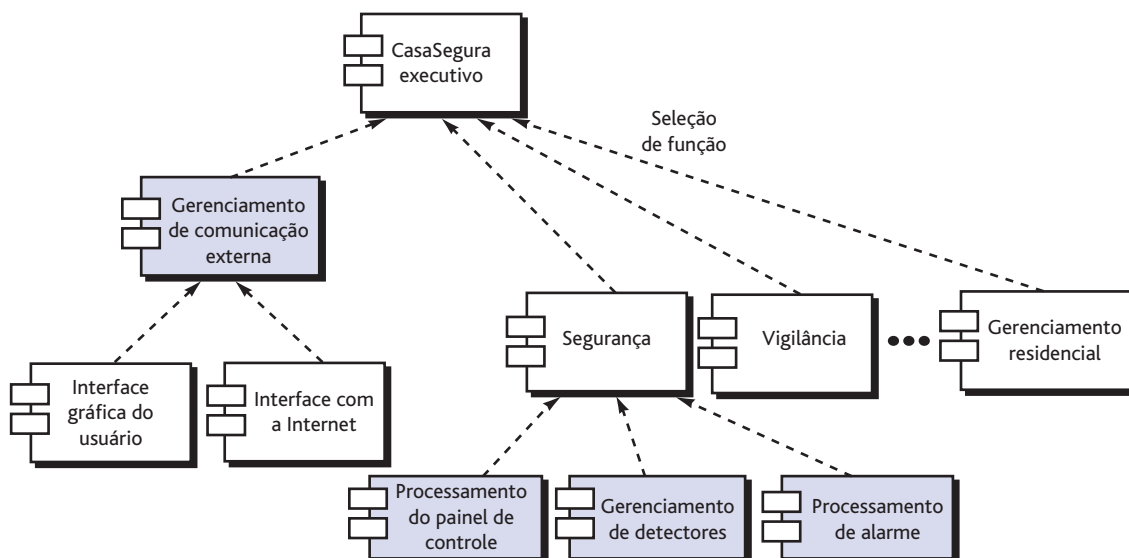


FIGURA 13.8 Estrutura arquitetural global para o sistema *CasaSegura* com os componentes de alto nível.

sensores para detectar uma condição de alarme, e o componente de *processamento de alarme* gera uma saída quando o alarme é detectado.

13.6.4 Descrição das instâncias do sistema

Até este ponto, o projeto de arquitetura modelado ainda é relativamente de alto nível. O contexto do sistema foi representado, arquétipos que indicam importantes abstrações contidas no domínio do problema foram definidos, a estrutura global do sistema está evidente e os principais componentes de software foram identificados. Entretanto, um maior refinamento (recorde-se que todo projeto é iterativo) ainda é necessário.

Para tanto, é desenvolvida uma instância real da arquitetura. Queremos dizer com isso que a arquitetura é aplicada a um problema específico com o intuito de demonstrar que a estrutura e os componentes são apropriados.

A Figura 13.9 ilustra uma instância da arquitetura do *CasaSegura* para o sistema de segurança. Os componentes da Figura 13.8 são elaborados para indicar mais detalhes. Por exemplo, o componente *gerenciamento de detectores* interage com o componente de infraestrutura *agendador* que implementa a sondagem (pooling) de cada objeto *sensor* usado pelo sistema de segurança. Uma elaboração semelhante é feita para cada um dos componentes representados na Figura 13.8.

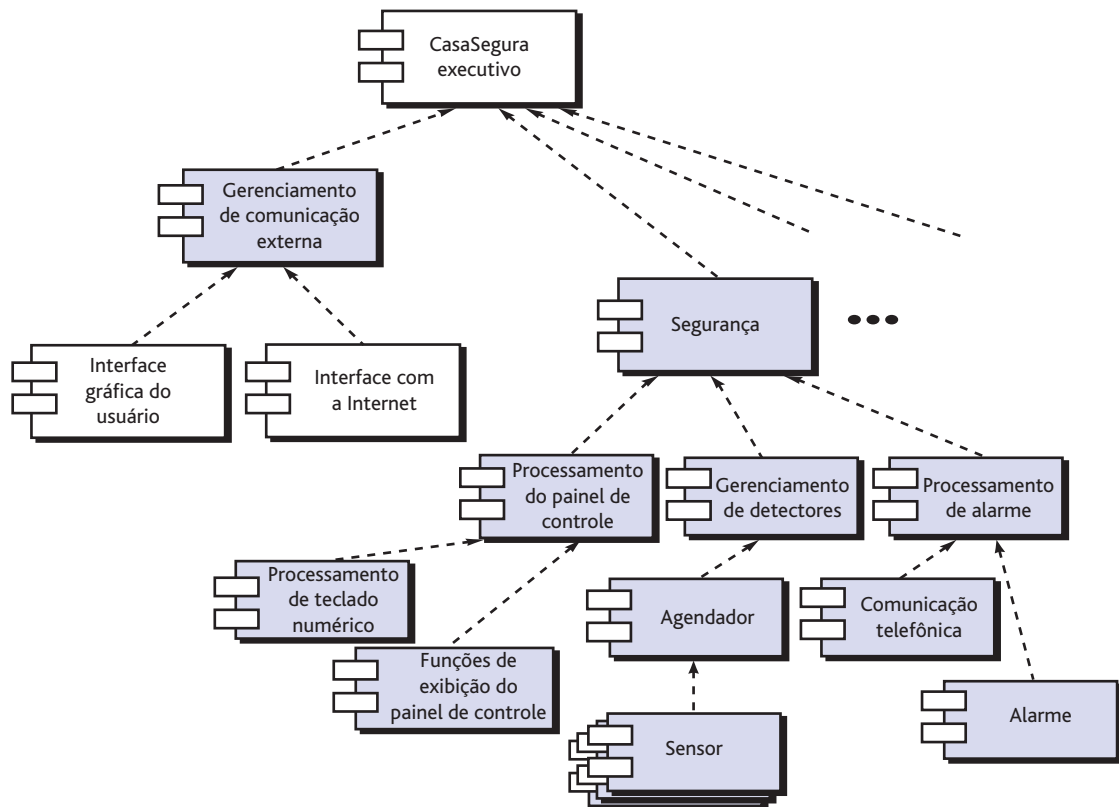


FIGURA 13.9 Uma instância da função de segurança com elaboração de componentes.

FERRAMENTAS DO SOFTWARE

**Projeto de arquitetura**

Objetivo: As ferramentas de projeto de arquitetura modelam a estrutura global do software, representando a interface com componentes, dependências e relações, bem como interações.

Mecanismos: O mecanismo das ferramentas é variado. Na maioria dos casos, a capacidade de projeto de arquitetura faz parte da funcionalidade fornecida por ferramentas automatizadas para análise e modelagem de projeto.

Ferramentas representativas:⁷

Adalon, desenvolvida pela Synthis Corp. (www.synthis.com), é uma ferramenta de projeto especializada para o

projeto e a construção de arquiteturas específicas de componentes baseados na Web.

ObjectiF, desenvolvida pela microTOOL GmbH (www.microtool.de/objectiF/en/), é uma ferramenta de projeto baseada em UML que conduz a arquiteturas (por exemplo, Coldfusion, J2EE, Fusebox) suscetíveis à engenharia de software baseada em componentes (Capítulo 14).

Rational Rose, desenvolvida pela Rational (<http://www-01.ibm.com/software/rational/>), é uma ferramenta de projeto baseada em UML que oferece suporte a todos os aspectos do projeto de arquitetura.

13.6.5 Projeto de arquitetura para aplicações web (WebApps)

WebApps⁸ são aplicações cliente-servidor normalmente estruturadas com arquiteturas de várias camadas, incluindo uma camada de interface ou visão do usuário, uma camada controladora, a qual direciona o fluxo de informações no navegador cliente de acordo com um conjunto de regras do negócio, e uma camada de conteúdo ou modelo, que também pode conter as regras do negócio da WebApp.

A interface do usuário de uma WebApp é projetada em torno das características do navegador web em execução na máquina cliente (normalmente, um computador pessoal ou dispositivo móvel). A camada de dados reside em um servidor. As regras do negócio podem ser implementadas com uma linguagem de script baseada no servidor, como PHP, ou baseada no cliente, como JavaScript. Um arquiteto examinará os requisitos de segurança e usabilidade para determinar quais recursos devem ser alocados para o cliente ou para o servidor.

O projeto arquitetural de uma WebApp também sofre influência da estrutura (linear ou não linear) do conteúdo que precisa ser acessado pelo cliente. Os componentes arquiteturais (páginas Web) de uma WebApp são projetados de forma a permitir que o controle seja passado para outros componentes do sistema, possibilitando estruturas de navegação muito flexíveis. O local físico da mídia e de outros recursos do conteúdo também influencia as escolhas arquiteturais feitas pelos engenheiros de software.

⁷ A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

⁸ O projeto de WebApps é discutido de forma mais detalhada no Capítulo 17.

13.6.6 Projeto de arquitetura para aplicativos móveis

Normalmente, os aplicativos móveis⁹ são estruturados com arquiteturas de várias camadas, incluindo uma camada de interface do usuário, uma camada do negócio e uma camada de dados. No caso de aplicativos móveis, pode-se escolher entre construir um cliente magro (*thin client*) baseado na Web ou um cliente rico (*rich client*). Em um cliente magro, apenas a interface do usuário reside no dispositivo móvel, enquanto as camadas do negócio e de dados residem em um servidor. Em um cliente rico, todas as três camadas podem residir no próprio dispositivo móvel.

Os dispositivos móveis diferem entre si em termos de suas características físicas (por exemplo, tamanhos de tela, dispositivos de entrada), de software (por exemplo, sistemas operacionais, suporte para linguagem) e de hardware (por exemplo, memória, conexões de rede). Cada um desses atributos molda o rumo das alternativas arquiteturais que podem ser escolhidas. Meier e seus colegas [Mei09] sugerem diversas considerações que podem influenciar o projeto de arquitetura de um aplicativo móvel: (1) o tipo de cliente web a ser construído (magro ou rico), (2) as categorias de dispositivos suportados (por exemplo, smartphones, tablets), (3) o grau de conectividade exigido (ocasional ou persistente), (4) a largura de banda exigida, (5) as restrições impostas pela plataforma móvel, (6) até que ponto a reutilização e a facilidade de manutenção são importantes e (7) as restrições de recurso do dispositivo (por exemplo, vida da bateria, tamanho da memória, velocidade do processador).

13.7 Avaliação das alternativas de projeto de arquitetura

Em seu livro sobre avaliação de arquiteturas de software, Clements e seus colegas [Cle03] afirmam:

Falando claramente, arquitetura é uma aposta, uma aposta no sucesso de um sistema. Não seria ótimo saber antecipadamente se apostamos em um vencedor, em vez de ter de esperar até que o sistema esteja quase concluído para só então saber se ele vai atender ou não aos requisitos? Se estivesse comprando um sistema ou pagando por seu desenvolvimento, você não gostaria de ter alguma certeza de que ele foi iniciado de maneira correta? Se fosse o próprio arquiteto, você não gostaria de ter um bom modo de validar suas intuições e experiência para poder dormir à noite sabendo que a confiança depositada em seu projeto é bem fundamentada?

Sem dúvidas as respostas a essas questões seriam valiosas. O projeto resulta em uma série de alternativas de arquitetura, cada uma das quais avaliada para determinar qual delas é a mais apropriada para o problema a ser resolvido. Nas seções a seguir, apresentaremos duas abordagens diferentes para a avaliação de projetos da arquitetura alternativos. A primeira usa um método iterativo para avaliar os prós e contras do projeto. A segunda aplica uma técnica pseudoquantitativa para avaliar a qualidade de um projeto.

⁹ O projeto de aplicativos móveis é discutido de forma mais detalhada no Capítulo 18.

O SEI (Software Engineering Institute) desenvolveu um *método de análise dos prós e contras de uma arquitetura* (ATAM, *architecture trade-off analysis method*) [Kaz98] que estabelece um processo de avaliação iterativa de arquiteturas de software. As atividades de análise de projeto a seguir são realizadas iterativamente:

1. *Coletar cenários.* É desenvolvido um conjunto de casos de uso (Capítulos 8 e 9) para representar o sistema sob o ponto de vista do usuário.
2. *Levantar requisitos, restrições e descrição do ambiente.* Essas informações são exigidas como parte da engenharia de requisitos e usadas para certificar que todas as necessidades dos envolvidos foram atendidas.
3. *Descrever os estilos/padrões de arquitetura escolhidos para lidar com os cenários e requisitos.* O(s) estilo(s) de arquitetura deve(m) ser descrito(s) usando uma das seguintes visões de arquitetura:
 - *Visão de módulos* para a análise de atribuições de trabalho com componentes e o grau que foi atingido pelo encapsulamento de informações.
 - *Visão de processos* para a análise do desempenho do sistema.
 - *Visão de fluxo de dados* para análise do grau em que a arquitetura atende às necessidades funcionais.
4. *Avaliar atributos de qualidade considerando cada atributo isoladamente.* O número de atributos de qualidade escolhidos para análise é uma função do tempo disponível para revisão e do grau em que os atributos de qualidade são relevantes para o sistema em questão. Entre os atributos de qualidade para avaliação de projetos da arquitetura temos: confiabilidade, desempenho, segurança, facilidade de manutenção, flexibilidade, testabilidade, portabilidade, reusabilidade e interoperabilidade.
5. *Identificar a sensibilidade dos atributos de qualidade em relação a vários atributos de arquitetura para um estilo de arquitetura específico.* Isso pode ser obtido fazendo-se pequenas alterações na arquitetura e determinando-se o quanto um atributo de qualidade é sensível (digamos, o desempenho) em relação a uma mudança. Quaisquer atributos afetados significativamente por uma variação na arquitetura são denominados *pontos de sensibilidade*.
6. *Criticar arquiteturas candidatas (desenvolvidas na etapa 3) usando a análise de sensibilidade realizada na etapa 5.* O SEI descreve esse método da seguinte maneira [Kaz98]:

Uma vez determinados os pontos de sensibilidade da arquitetura, encontrar o ponto de balanceamento é simplesmente identificar os elementos da arquitetura para os quais vários atributos são sensíveis. Por exemplo, o desempenho de uma arquitetura cliente/servidor poderia ser altamente sensível ao número de servidores (o desempenho aumenta, dentro de certo intervalo, aumentando-se o número de servidores)... O número de servidores é, então, um ponto de balanceamento com relação a essa arquitetura.

Essas seis etapas representam a primeira iteração ATAM. Com base nos resultados das etapas 5 e 6, algumas alternativas de arquitetura podem ser

eliminadas, uma ou mais arquiteturas remanescentes podem ser modificadas e representadas de forma mais detalhada e então as etapas do ATAM seriam reaplicadas.¹⁰

CASASEGURA



Avaliação de arquiteturas

Cena: Sala de Doug Miller à medida que a modelagem do projeto de arquitetura prossegue.

Atores: Vinod, Jamie e Ed, membros da equipe de engenharia de software do *CasaSegura*, e Doug Miller, gerente do grupo de engenharia de software.

Conversa:

Doug: Sei que vocês estão derivando algumas arquiteturas diferentes para o produto *CasaSegura*, e isso é bom. Mas aí pergunto, “como vamos escolher a melhor delas”?

Ed: Estou trabalhando em um estilo de chamadas e retornos e depois Jamie ou eu obteremos uma arquitetura orientada a objetos.

Doug: Certo, e como fazemos essa escolha?

Jamie: Fiz um curso de projeto no meu último ano de faculdade e recordo-me de que há uma série de maneiras para fazer isso.

Vinod: Realmente, porém elas são um tanto acadêmicas. Veja, acredito que possamos fazer nossa avaliação e escolher a arquitetura correta empregando casos de uso e cenários.

Doug: Isso não é a mesma coisa?

Vinod: Não quando se trata de avaliação de arquiteturas. Já temos um conjunto completo de casos de uso. Então aplicamos cada um deles às duas arquiteturas e verificamos como o sistema reage e como os componentes e conectores funcionam no contexto dos casos de uso.

Ed: É uma boa ideia. Garante que não nos esqueçamos de nada.

Vinod: É verdade, mas eles também nos informam se o projeto de arquitetura é complicado ou não e se o sistema tem ou não de se desdobrar para conseguir realizar sua tarefa.

Jamie: Cenários não é apenas outro nome para casos de uso?

Vinod: Não, neste caso um cenário significa algo diferente.

Doug: Você está se referindo a um cenário de qualidade ou um cenário de mudanças, certo?

Vinod: Sim. O que fazemos é procurar novamente os envolvidos e perguntar a eles como o *CasaSegura* provavelmente vai mudar ao longo dos próximos, digamos, três anos. Sabe, novas versões, recursos, esse tipo de coisa. Construímos um conjunto de cenários de mudanças. Também desenvolvemos um conjunto de cenários de qualidade que definem os atributos que gostaríamos de ver na arquitetura do software.

Jamie: E os aplicamos às alternativas.

Vinod: Exatamente. O estilo que se comportar melhor nos casos de uso e cenários será o escolhido.

13.7.1 Linguagens de descrição da arquitetura

A *linguagem de descrição da arquitetura* (ADL, *architectural description language*) fornece uma sintaxe e uma semântica para descrever uma arquitetura de software. Hofmann e seus colegas [Hof01] sugerem que uma ADL deve fornecer ao projetista a capacidade de decompor componentes da arquitetura, compor componentes individuais em blocos arquiteturais maiores e representar interfaces (mecanismos de conexão) entre componentes. Uma vez estabelecidas técnicas descritivas e baseadas em linguagens para o projeto da arqui-

¹⁰ O método de análise de arquitetura via software (SAAM, *software architecture analysis method*) é uma alternativa ao ATAM e vale a pena ser examinado por leitores interessados em análise de arquiteturas. Um artigo sobre o SAAM pode ser baixado em www.sei.cmu.edu/publications/articles/saam-metho-propert-sas.html.

FERRAMENTAS DO SOFTWARE

**Linguagens de descrição da arquitetura**

O resumo a seguir de uma série de importantes ADLs foi preparado por Rickard Land [Lan02] e é reimpresso com a permissão do autor. Deve-se salientar que as cinco ADLs listadas foram desenvolvidas para fins de pesquisa e não são produtos comerciais.

xArch (<http://www.isr.uci.edu/projects/xarchuci/>) é uma representação padrão, baseada em XML extensível, para arquiteturas de software.

UniCon (www.cs.cmu.edu/~UniCon) é “uma linguagem de descrição de arquitetura destinada a auxiliar os projetistas a definir arquiteturas de software em termos de abstrações que achem úteis”.

Wright (www.cs.cmu.edu/~able/wright/) é uma linguagem formal que inclui os seguintes elementos:

componentes com portas, conectores com papéis e conectores para ligar papéis às portas. Os estilos de arquitetura podem ser formalizados na linguagem por meio de predicados, permitindo, portanto, verificações estáticas para determinar a consistência e a completude de uma arquitetura.

Acme (www.cs.cmu.edu/~acme/) pode ser vista como uma ADL de segunda geração, já que seu intuito é identificar um mínimo denominador comum para ADLs.

UML (www.uml.org/) inclui muitos dos artefatos necessários para descrições de arquiteturas – processos, nós, visões etc. Para descrições informais, a UML é adequada apenas pelo fato de ser um padrão amplamente compreendido. Falta-lhe, entretanto, o poder necessário para uma descrição adequada de uma arquitetura.

tetura, é mais provável que métodos de avaliação de arquitetura mais efetivos sejam estabelecidos à medida que o projeto evoluir.

13.7.2 Revisões da arquitetura

Revisões da arquitetura representam um tipo de análise técnica especializada (Capítulo 20) que proporciona um modo de avaliar a capacidade de uma arquitetura de software atender aos requisitos de qualidade do sistema (por exemplo, escalabilidade ou desempenho) e identificar quaisquer riscos em potencial. Essas revisões têm o potencial de reduzir custos de projeto por detectar problemas no início.

Ao contrário das revisões de requisitos, que envolvem representantes de todos os envolvidos, as revisões de arquitetura frequentemente envolvem apenas os membros da equipe de engenharia de software, complementados por especialistas independentes. As técnicas de revisão de arquitetura mais usadas no setor são: raciocínio baseado na experiência,¹¹ avaliação de protótipo, revisão de cenário (Capítulo 9) e uso de checklists.¹² Muitas revisões de arquitetura ocorrem no início do ciclo de vida do projeto, mas também devem ocorrer depois que novos componentes ou pacotes são adquiridos no projeto baseado em componentes (Capítulo 14). Os engenheiros de software que realizam revisões de arquitetura observam que, às vezes, artefatos arquiteturais estão ausentes ou são inadequados, tornando, assim, difícil concluí-las [Bab09].

¹¹ O raciocínio *baseado na experiência* compara a nova arquitetura de software com uma arquitetura usada para criar um sistema semelhante a um do passado.

¹² Checklists representativas podem ser encontradas em <http://www.opengroup.org/architecture/togaf7-doc/arch/p4/comp/clists/syseng.htm>.

Exemplos de lições aprendidas no projeto de arquitetura de software podem ser encontrados em <http://www.sei.cmu.edu/library/abstracts/news-at-sei/01feature200707.cfm>.

Uma discussão sobre revisões de arquitetura baseadas em padrões aparece em <http://www.infoq.com/articles/ieee-pattern-based-architecture-reviews>.

13.8 Lições aprendidas

Os sistemas baseados em software são construídos por pessoas com necessidades e pontos de vista diferentes. Portanto, um arquiteto de software deve estabelecer um consenso entre os membros da equipe de software (e outros envolvidos) para obter a visão arquitetural para o produto de software final [Wri11].

Muitas vezes, os arquitetos se concentram no impacto de longo prazo dos requisitos não funcionais do sistema à medida que a arquitetura é criada. Os gerentes experientes avaliam a arquitetura dentro do contexto das metas e objetivos do negócio. Muitas vezes, os gerentes de projeto são induzidos por considerações de curto prazo de datas de entrega e orçamento. Os engenheiros de software frequentemente se concentram em seus próprios interesses tecnológicos e na entrega de funcionalidades. Cada um desses (e outros) participantes deve trabalhar no sentido de conseguir um consenso de que a arquitetura do software escolhida tem vantagens marcantes sobre quaisquer alternativas.

Wright [Wri11] sugere o uso de vários métodos de *análise de decisão e resolução* (DAR, *decision analysis and resolution*), que podem ajudar a neutralizar alguns obstáculos à colaboração. Esses métodos podem ajudar a aumentar a participação ativa dos membros da equipe e a probabilidade de adesão à decisão final. Os métodos DAR ajudam os membros da equipe a considerar, de maneira objetiva, várias alternativas de arquitetura viáveis. Três exemplos representativos de métodos DAR são:

- **Cadeia de causas.** Essa técnica é uma forma de análise da causa-raiz,¹³ na qual a equipe define uma meta ou efeito arquitetural e, então, enuncia as ações relacionadas que farão a meta ser atingida.
- **Espinha de peixe Ishikawa.**¹⁴ Técnica gráfica que identifica muitas ações ou causas possíveis, exigidas para se atingir uma meta arquitetural desejada.
- **Mapa mental ou diagramas de aranha.**¹⁵ Esse diagrama é usado para representar palavras, conceitos, tarefas ou artefatos de engenharia de software, organizados em torno de uma palavra-chave, restrição ou requisito central.

13.9 Revisão de arquitetura baseada em padrões

Revisões técnicas formais (Capítulo 20) podem ser aplicadas à arquitetura de software e oferecem um modo de gerenciar atributos de qualidade do sistema, descobrir erros e evitar retrabalho desnecessários. Contudo, nas situações em

¹³ Mais informações podem ser obtidas em: <http://www.thinkreliability.com/Root-Cause-Analysis-CM-Basics.aspx>.

¹⁴ Mais informações podem ser obtidas em: <http://asq.org/learn-about-quality/cause-analysis-tools/overview/fishbone.html>.

¹⁵ Mais informações podem ser obtidas em: <http://mindmappingsoftwareblog.com/5-best-mind-mapping-programs-for-brainstorming/>.

que as normas são ciclos de construção curtos, prazos finais apertados, requisitos voláteis e/ou equipes pequenas, a melhor opção pode ser um processo de revisão de arquitetura leve, conhecido como *revisão de arquitetura baseada em padrões* (PBAR, *pattern-based architecture review*).

PBAR é um método de avaliação que aproveita as relações entre padrões de arquitetura¹⁶ e atributos de qualidade de software. Uma PBAR é uma reunião de auditoria presencial envolvendo todos os desenvolvedores e outros participantes envolvidos. Também está presente um revisor externo com especialidade em arquitetura, padrões de arquitetura, atributos de qualidade e domínio de aplicação. O arquiteto do sistema é o principal apresentador.

Uma PBAR deve ser agendada após a conclusão do primeiro protótipo ou *esqueleto móvel*¹⁷ funcional. Ela abrange as seguintes etapas iterativas [Har11]:

1. Identificar e discutir os atributos de qualidade mais importantes para o sistema, acompanhando os casos de uso relevantes (Capítulo 9).
2. Discutir um diagrama da arquitetura do sistema em relação aos seus requisitos.
3. Ajudar os revisores a identificar padrões de arquitetura usados e a combinar a estrutura do sistema com a estrutura dos padrões.
4. Usar a documentação existente e casos de uso passados, examinar a arquitetura e os atributos de qualidade para determinar o efeito de cada padrão nos atributos de qualidade do sistema.
5. Identificar e discutir todas as questões relacionadas à qualidade levantadas pelos padrões de arquitetura utilizados no projeto.
6. Gerar um breve resumo dos problemas descobertos durante a reunião e fazer as revisões apropriadas no esqueleto móvel.

As PBARs são adequadas para pequenas equipes ágeis e exigem uma quantidade extra e relativamente pequena de tempo e esforço de projeto. Com seu curto tempo de preparação e revisão, a PBAR pode se ajustar a requisitos variáveis e ciclos de construção reduzidos, ao mesmo tempo que ajuda a equipe a entender melhor a arquitetura do sistema.

13.10 Verificação de conformidade da arquitetura

À medida que o processo de software passa do projeto para a construção, os engenheiros de software devem trabalhar no sentido de garantir que o sistema implementado e em evolução seja compatível com sua arquitetura planejada. Muitas coisas (por exemplo, requisitos conflitantes, dificuldades técnicas, pressões do prazo final) causam desvios em relação a uma arquitetura

¹⁶ Um *padrão de arquitetura* é uma solução generalizada para um problema de projeto de arquitetura, com um conjunto específico de condições ou restrições. Os padrões são discutidos de forma detalhada no Capítulo 16.

¹⁷ Um *esqueleto móvel* contém uma linha de base da arquitetura que suporta os requisitos funcionais, com as prioridades mais relevantes do processo de negócio e os atributos de qualidade mais desafiadores.

Um panorama da verificação de conformidade da arquitetura aparece em <http://www.cin.ufpe.br/~fcf3/Arquitetura%20de%20Software/arquitetura/getPDF3.pdf>.

definida. Se a conformidade da arquitetura não for verificada periodicamente, desvios não controlados podem causar a *erosão de arquitetura* e afetar a qualidade do sistema [Pas10].

A *análise estática da conformidade da arquitetura* (SACA, *static architecture-conformance analysis*) avalia se um sistema de software implementado é coerente com seu modelo arquitetural. O formalismo (por exemplo, UML) usado para modelar a arquitetura do sistema apresenta a organização estática dos seus componentes e como estes interagem. Frequentemente, o modelo arquitetural é usado por um gerente de projeto para planejar e alocar tarefas, assim como para avaliar o andamento da implementação.

FERRAMENTAS DO SOFTWARE



Ferramentas de compatibilidade arquitetural

Lattix Dependency Manager (<http://www.lattix.com/>). Essa ferramenta contém uma linguagem simples para declarar as regras de projeto que a implementação deve seguir, detecta violações nessas regras e representa-as visualmente como uma matriz de estrutura de dependências.

Source Code Query Languages (<http://www.semmle.com/>). Essa ferramenta pode ser usada para automatizar tarefas de desenvolvimento de software, como a definição e a verificação de restrições arquiteturais, e

utiliza uma linguagem do tipo Prolog para definir consultas recursivas na hierarquia de herança de sistemas orientados a objetos.

Reflexion Models (http://www.iese.fraunhofer.de/en/competencies/architecture/tools_architecture.html#contentPar_textblockwithpics). A ferramenta SAVE pode ser usada por engenheiros de software para construir um modelo de alto nível capturando a arquitetura de um sistema e, então, definir as relações entre esse modelo e o código-fonte. Então, a SAVE identifica relações ausentes ou errôneas entre o modelo e o código.

Uma discussão sobre o papel da arquitetura em processos de software ágeis aparece em <http://msdn.microsoft.com/enus/architecture/ff476940.aspx>.

13.11 Agilidade e arquitetura

Na visão de alguns proponentes do desenvolvimento ágil, o projeto arquitetural é equiparado a um “projeto grande inicial”. Segundo essa visão, isso leva à documentação desnecessária e à implementação de funcionalidades desnecessárias. Contudo, a maioria dos desenvolvedores ágeis concorda [Fal10] que é importante se concentrar na arquitetura do software quando um sistema é complexo (isto é, quando um produto tem um grande número de requisitos, muitos envolvidos ou ampla distribuição geográfica). Por isso, há a necessidade de integrar novas práticas de projeto arquitetural aos modelos de processo ágeis.

Para tomar decisões arquiteturais no início e evitar a reformulação exigida e/ou os problemas de qualidade encontrados quando a arquitetura errada é escolhida, os desenvolvedores ágeis devem antecipar os elementos arquiteturais¹⁸ e a estrutura baseada em uma coleção emergente de histórias de usuário (Capítulo 5). Criando um protótipo da arquitetura (por exemplo, um

¹⁸ Uma excelente discussão sobre agilidade arquitetural pode ser encontrada em [Bro10a].

esqueleto móvel) e desenvolvendo produtos de trabalho arquitetural explícitos para se comunicar com os envolvidos necessários, uma equipe ágil pode satisfazer a necessidade de um projeto arquitetural.

O desenvolvimento ágil oferece aos arquitetos de software repetidas oportunidades para trabalharem em conjunto com as equipes de negócio e técnicas a fim de orientar o rumo para um bom projeto de arquitetura. Madison [Mad10] sugere o uso de um framework híbrido contendo elementos de Scrum, XP e gerenciamento de projeto sequencial.¹⁹ Nesse framework, o planejamento antecipado define o rumo arquitetural, mas muda rapidamente para um storyboard [Bro10b].

Durante o storyboard, o arquiteto fornece histórias de usuário arquiteturais para o projeto e trabalha junto ao proprietário do produto para priorizá-las com as histórias de usuário de negócio, à medida que “sprints” (unidades de trabalho) são planejados. O arquiteto trabalha com a equipe durante o sprint para garantir que o software em evolução continue a exibir alta qualidade arquitetural. Se a qualidade é alta, a equipe pode continuar o desenvolvimento por conta própria. Caso contrário, o arquiteto se une à equipe durante o sprint. Uma vez concluído o sprint, o arquiteto examina a qualidade do protótipo funcional, antes que a equipe o apresente para os envolvidos em uma revisão de sprint formal. Projetos ágeis bem executados exigem a entrega iterativa de artefatos (incluindo a documentação da arquitetura) a cada sprint. Examinar os artefatos e o código, à medida que surgem de cada sprint, é uma forma útil de revisão da arquitetura.

Arquitetura orientada a responsabilidades (RDA, *responsibility-driven architecture*) é um processo que se concentra na tomada de decisão de arquitetura. Ela trata de quando e como as decisões de arquitetura devem ser tomadas e quem da equipe de projeto as toma. Essa abordagem também enfatiza o papel do arquiteto como líder servidor, em vez de tomador de decisões autocrático, e é coerente com a filosofia ágil. O arquiteto atua como facilitador e concentra-se em como a equipe de desenvolvimento trabalha com as preocupações dos envolvidos de fora (por exemplo, negócio, segurança, infraestrutura).

As equipes ágeis insistem na liberdade para fazer mudanças, à medida que surgem novos requisitos. Os arquitetos querem ter certeza de que as partes importantes da arquitetura tenham sido cuidadosamente consideradas e que os desenvolvedores tenham consultado os envolvidos adequados. As duas preocupações podem ser atendidas pelo uso de uma prática denominada *aprovação progressiva* (*progressive sign-off*), na qual o produto em evolução é documentado e aprovado à medida que cada protótipo sucessivo é concluído [Bla10].

O uso de um processo compatível com a filosofia ágil oferece uma aprovação que pode ser verificada por reguladores e auditores, sem impedir que equipes ágeis tomem as decisões necessárias. Ao final do projeto, a equipe tem um conjunto completo de artefatos, e a qualidade da arquitetura foi examinada à medida que evoluiu.

¹⁹ Scrum e XP são modelos de processo ágeis, discutidos no Capítulo 5.

13.12 Resumo

A arquitetura de software oferece uma visão holística do sistema a ser construído. Ela representa a estrutura e a organização dos componentes de software, suas propriedades e as conexões entre eles. Os componentes de software incluem módulos de programas e as várias representações de dados manipuladas pelo programa. Consequentemente, o projeto de dados é parte da obtenção da arquitetura de software. A arquitetura destaca decisões de projeto iniciais e fornece um mecanismo para considerar os benefícios de estruturas do sistema alternativas.

Diferentes estilos e padrões de arquitetura estão disponíveis para o engenheiro de software, e podem ser aplicados dentro de um dado gênero de arquitetura. Cada estilo descreve uma categoria de sistema que engloba um conjunto de componentes que realiza uma função exigida por um sistema; um conjunto de conectores que possibilita a comunicação, coordenação e cooperação entre os componentes; restrições que definem como os componentes podem ser integrados para formar o sistema; e modelos semânticos que permitem a um projetista compreender as propriedades gerais de um sistema.

De modo geral, o projeto de arquitetura é realizado em quatro etapas distintas. Primeiramente, o sistema deve ser representado no contexto. Ou seja, o projeto deve definir as entidades externas com as quais o software interage e a natureza da interação. Uma vez especificado o contexto, o projetista deve identificar um conjunto de abstrações de alto nível, denominado arquétipos, que representam elementos fundamentais do comportamento ou função do sistema. Depois de as abstrações serem definidas, o projeto começa a se aproximar do domínio da implementação. Os componentes são identificados e representados no contexto de uma arquitetura que os suporta. Por fim, são desenvolvidas instâncias específicas da arquitetura para “provar” o projeto em um contexto do mundo real.

O projeto de arquitetura pode coexistir com métodos ágeis, aplicando-se um framework de projeto arquitetural híbrido que faça uso de técnicas existentes derivadas de métodos ágeis conhecidos. Uma vez desenvolvida a arquitetura, ela pode ser avaliada para garantir a conformidade com metas do negócio, requisitos do software e atributos de qualidade.

Problemas e pontos a ponderar

13.1 Usando a arquitetura de uma casa ou edifício como metáfora, faça comparações com arquitetura de software. Em que sentido as disciplinas da arquitetura clássica e arquitetura de software são similares? Como diferem?

13.2 Apresente dois ou três exemplos de aplicações para cada um dos estilos de arquitetura citados na Seção 13.3.1.

13.3 Alguns dos estilos de arquitetura citados na Seção 13.3.1 são hierárquicos por natureza e outros não. Faça uma lista de cada um dos tipos. Como os estilos de arquitetura que não são hierárquicos seriam implementados?

13.4 Os termos *estilo de arquitetura*, *padrão de arquitetura* e *framework* (não discutido neste livro) são muitas vezes encontrados em discussões sobre arquitetura de software. Pesquise e descreva como cada um deles difere de seus equivalentes.

13.5 Escolha uma aplicação com a qual esteja familiarizado. Responda a cada uma das perguntas apresentadas para controle e dados na Seção 13.3.3.

13.6 Pesquise o ATAM (usando [Kaz98]) e apresente uma discussão detalhada das seis etapas indicadas na Seção 13.7.1.

13.7 Caso ainda não tenha feito, complete o Problema 9.5. Use a abordagem de projeto descrita no presente capítulo para desenvolver uma arquitetura de software para a PHTRS.

13.8 Use o modelo de decisão de arquitetura da Seção 13.1.4 para documentar uma das decisões para a arquitetura PHTRS desenvolvida no Problema 13.7.

13.9 Escolha um aplicativo móvel que você conheça e avalie-o usando as considerações arquiteturais (economia, visibilidade, espaçamento, simetria, emergência) da Seção 13.4.

13.10 Liste os pontos fortes e fracos da arquitetura PHTRS criada para o Problema 13.7.

13.11 Crie uma matriz de estrutura de dependência²⁰ para a arquitetura de software PHTRS criada para o Problema 13.7.

13.12 Escolha um modelo de processo ágil do Capítulo 5 e identifique as atividades de projeto de arquitetura incluídas.

Leituras e fontes de informação complementares

A literatura sobre arquitetura de software atingiu seu ápice ao longo da última década. Varma (*Software Architecture: A Case Based Approach*, Pearson, 2013) apresenta a arquitetura no contexto de uma série de estudos de caso. Livros de Bass e seus colegas (*Software Architecture in Practice*, 3ª ed., Addison-Wesley, 2012), Gorton (*Essential Software Architecture*, 2ª ed., Springer, 2011), Rozanski e Woods (*Software Systems Architecture*, 2ª ed., Addison-Wesley, 2011), Eeles e Cripps (*The Process of Software Architecting*, Addison-Wesley, 2009), Taylor e seus colegas (*Software Architecture*, Wiley, 2009), Reekie e McAdam (*A Software Architecture Primer*, 2ª ed., Angophora Press, 2006) e Albin (*The Art of Software Architecture*, Wiley, 2003) apresentam interessantes tratamentos de um tema intelectualmente desafiador.

Buschman e seus colegas (*Pattern-Oriented Software Architecture*, Wiley, 2007) e Kuchana (*Software Architecture Design Patterns in Java*, Auerbach, 2004) discutem aspectos orientados a padrões de projeto da arquitetura. Knoernschilf (*Java Application Architecture: Modularity Patterns with Examples Using OSGi*, Prentice Hall, 2012), Rozanski e Woods (*Software Systems Architecture*, 2ª ed., Addison-Wesley, 2011), Henderikson (*12 Essential Skills for Software Architects*, Addison-Wesley, 2011), Clements e seus colegas (*Documenting Software Architecture: View and Beyond*, 2ª ed., Addison-Wesley, 2010), Microsoft (*Microsoft Application Guide*, Microsoft Press, 2ª ed., 2009), Fowler (*Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003), Bosch [Bos00] e Hofmeister e seus colegas [Hof00] fornecem tratamentos aprofundados sobre arquitetura de software.

Hennesey e Patterson (*Computer Architecture*, 5ª ed., Morgan-Kaufmann, 2011) adotam uma visão quantitativa distinta sobre as questões de projeto da arquitetura de software. Clements e seus colegas (*Evaluating Software Architectures*, Addison-Wesley, 2002) consideram as questões associadas à avaliação de alternativas de arquitetura e à escolha da melhor arquitetura para determinado domínio de problema.

Livros abordando implementações específicas sobre arquitetura tratam do projeto da arquitetura em uma tecnologia ou ambiente de desenvolvimento específico. Erl

²⁰ Use a Wikipedia como ponto de partida para obter mais informações sobre DSM em: http://en.wikipedia.org/wiki/Design_structure_matrix.

(*SOA Design Patterns*, Prentice Hall, 2009) e Marks e Bell (*Service-Oriented Architecture*, Wiley, 2006) discutem uma abordagem de projeto que associa recursos comerciais e computacionais aos requisitos definidos por clientes. Brown *et al.* (*Model-Driven Software Engineering in Practice*, Morgan Claypool, 2012) e Stahl e seus colegas (*Model-Driven Software Development*, Wiley, 2006) discutem a arquitetura no contexto de abordagens de modelagem específicas do domínio. Radaideh e Al-ameed (*Architecture of Reliable Web Applications Software*, IGI Global, 2007) consideram arquiteturas apropriadas para WebApps. Esposito (*Architecting Mobile Solutions for the Enterprise*, Microsoft Press, 2012) discute a arquitetura de aplicativos móveis. Clements e Northrop (*Software Product Lines: Practices and Patterns*, Addison-Wesley, 2001) tratam do projeto de arquiteturas que suportam linhas de produtos de software. Shanley (*Protected Mode Software Architecture*, Addison-Wesley, 1996) fornece orientação para projeto da arquitetura para qualquer um que esteja desenvolvendo sistemas operacionais em tempo real baseados em PCs, sistemas operacionais multitarefa ou drivers de dispositivos.

A pesquisa atual em arquitetura de software é documentada anualmente nos *Proceedings of the International Workshop on Software Architecture*, patrocinados pela ACM e outras organizações de computação, e nos *Proceedings of the International Conference on Software Engineering*.

Uma ampla gama de fontes de informação sobre projeto da arquitetura se encontra à disposição na Internet. Uma lista atualizada de referências relevantes (em inglês) para o projeto de arquitetura pode ser encontrada no site: www.mhhe.com/pressman.

Projeto de componentes

14

O projeto de componentes ocorre depois que a primeira iteração do projeto da arquitetura tiver sido concluída. Nesse estágio, a estrutura geral dos dados e programas do software já foi estabelecida. O intuito é transformar o modelo de projeto em software operacional. Porém, o nível de abstração do modelo de projetos existente é relativamente alto, e o nível de abstração do programa operacional é baixo. A transformação pode ser desafiadora, pois é uma porta aberta para a introdução de erros sutis, difíceis de detectar e corrigir em estágios posteriores do processo de software. Em uma famosa palestra, Edsger Dijkstra, um importante colaborador para nosso entendimento de projeto de software, afirmou [Dij72]:

Software parece ser diferente de muitos outros produtos, em que, como regra, maior qualidade implica preços mais elevados. Aqueles que realmente querem software confiável descobrirão que devem primeiro encontrar um meio de evitar a maioria dos bugs, e, como resultado, o processo de programação se tornará mais barato... Programadores eficientes... Não devem perder tempo depurando – e, para início de conversa, não devem introduzir erros.

Embora essas palavras tenham sido proferidas muitos anos atrás, ainda são verdadeiras hoje. Ao transformarmos o modelo de projetos em código-fonte, devemos seguir um conjunto de princípios de projeto que não apenas realizam a transformação, como também não “introduzam bugs desde o início”.

Conceitos-chave

acoplamento	298
coesão	296
componente	286
adaptação	310
classificação	312
composição	310
qualificação	309
WebApp	305
componentes	
tradicionais	307
desenvolvimento	
baseado em	
componentes	308
diretrizes de projeto	295
engenharia de domínio	308
princípio da inversão	
da dependência	293
princípio da segregação	
de interfaces	293
princípio da substituição	
de Liskov	292
princípio do	
aberto-fechado	292
projeto de conteúdo	305
projeto para	
reutilização	312
visão orientada	
a objetos	286
visão relacionada a	
processos	291
visão tradicional	288

PANORAMA

O que é? Um conjunto completo de componentes de software é definido durante o projeto da arquitetura. Porém, os detalhes de processamento e estruturas de dados internas de cada componente não são representados em um nível de abstração próximo ao código. O projeto de componentes define as estruturas de dados, os algoritmos, as características das interfaces e os mecanismos de comunicação alocados a cada componente de software.

Quem realiza? Um engenheiro de software realiza o projeto de componentes.

Por que é importante? Você deve ser capaz de determinar se o software vai funcionar ou não antes de construí-lo. O projeto de componentes representa o software de maneira que lhe permita revisar os detalhes do projeto em termos de correção e consistência com outras representações de projeto (isto é, os projetos de interfaces, arquitetura e dados). Ele fornece um meio para avaliar se as estruturas de dados, interfaces e algoritmos vão funcionar.

Quais são as etapas envolvidas? As representações de projeto de dados, arquitetura e interfaces formam a base para o projeto de componentes. A definição de classes ou a narrativa de processamento para cada um dos componentes é traduzida em um projeto detalhado que faz uso de formas esquemáticas ou baseadas em texto que especificam estruturas de dados internas, detalhes de interfaces locais e lógica de processamento. A notação de projeto engloba diagramas UML e representações complementares. O projeto procedural é especificado usando-se um conjunto de construções da programação estruturada. Normalmente é possível adquirir componentes de software reutilizáveis, em vez de construir novos componentes.

Qual é o artefato? O projeto para cada componente, representado em notação gráfica, tabular ou baseada em texto, é o principal artefato durante o projeto de componentes.

Como garantir que o trabalho foi realizado corretamente?

É realizada uma revisão do projeto. O projeto é examinado para determinar se as estruturas de dados, interfaces, sequências de processamento e condições lógicas estão corretas e se vão produzir a transformação de controle ou de dados apropriada, atribuída ao componente durante etapas anteriores do projeto.

14.1 O que é componente?

“Detalhes não são detalhes. Eles compõem o projeto.”

Charles Eames

Componente é um bloco construtivo modular para software de computador. Mais formalmente, a *Especificação da Linguagem de Modelagem Unificada da OMG* (*OMG Unified Modeling Language Specification* [OMG03a]) define componente como “uma parte modular, possível de ser implantada e substituível de um sistema que encapsula implementação e expõe um conjunto de interfaces”.

Conforme discutido no Capítulo 13, os componentes preenchem a arquitetura de software e, como consequência, desempenham um papel para alcançar os objetivos e requisitos do sistema a ser construído. Pelo fato de os componentes residirem na arquitetura de software, devem se comunicar e colaborar com outros componentes e entidades (por exemplo, outros sistemas, dispositivos, pessoas) existentes fora dos limites do software.

O verdadeiro significado do termo componente diferirá dependendo do ponto de vista do engenheiro de software que o utiliza. Nas seções seguintes, examinaremos três importantes visões do que é e como é utilizado um componente à medida que a modelagem de projetos prossegue.

14.1.1 Uma visão orientada a objetos

Sob o ponto de vista orientado a objetos, um componente é um conjunto de classes colaborativas.

No contexto da engenharia de software orientada a objetos, um componente contém um conjunto de classes colaborativas.¹ Cada classe contida em um componente foi completamente elaborada para incluir todos os atributos e operações relevantes à sua implementação. Como parte da elaboração do projeto, também precisam ser definidas todas as interfaces que permitem que as classes se comuniquem e colaborem com outras classes de projeto. Para tanto, começamos com o modelo de análise e elaboramos as classes de análise (para componentes que se relacionam com o domínio do problema), bem como as classes de infraestrutura (para componentes que dão suporte a serviços para o domínio do problema).

¹ Em alguns casos, um componente pode conter uma única classe.

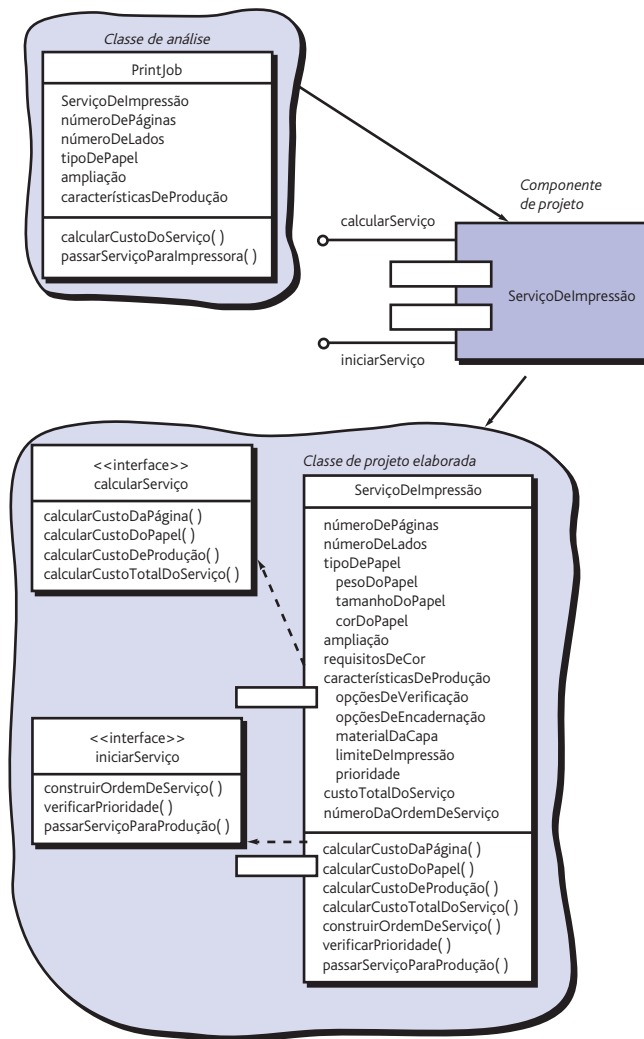


FIGURA 14.1 Elaboração de um componente de projeto.

Para ilustrarmos o processo de elaboração de projeto, consideremos o software a ser criado para uma sofisticada gráfica. O objetivo geral do software é coletar os requisitos do cliente na recepção da loja, orçar um trabalho e, em seguida, passar a tarefa para um centro de produção automatizado. Durante a engenharia de requisitos, foi obtida uma classe de análise denominada **ServiçoDeImpressão**. Os atributos e operações definidos durante a análise são indicados na parte superior da Figura 14.1. Durante o projeto da arquitetura, **ServiçoDeImpressão** é definida como um componente na arquitetura de software e é representada usando notação abreviada UML² no centro à direita da figura. Observe que **ServiçoDeImpressão** possui duas interfaces: **calcularServiço**, que fornece capacidade para orçar um trabalho, e **iniciarServiço**, que passa adiante a tarefa para o centro de produção. Estas são representadas usando-se os símbolos de “pirulito”, exibidos à esquerda do retângulo do componente.

² Os leitores que não estiverem familiarizados com a notação UML devem consultar o Apêndice 1.

Lembre-se de que tanto a modelagem da análise quanto a modelagem do projeto são ações iterativas. Elaborar a classe de análise original poderia exigir etapas de análise adicionais, seguidas, então, por etapas de modelagem de projetos para representar a classe de projeto elaborada (os detalhes do componente).

O projeto de componentes se inicia nesse ponto. Os detalhes do componente **ServiçoDeImpressão** devem ser elaborados a fim de fornecer informações suficientes para orientar a implementação. A classe de análise original é elaborada para dar corpo a todos os atributos e operações necessários para implementar a classe na forma do componente **ServiçoDeImpressão**. Com referência à parte inferior direita da Figura 14.1, a classe de projeto elaborada, **ServiçoDeImpressão**, contém informações de atributos mais detalhadas, bem como uma descrição ampliada das operações necessárias para implementar o componente. As interfaces *calcularServiço* e *iniciarServiço* implicam comunicação e colaboração com outros componentes (não indicados aqui). Por exemplo, a operação *calcularCustoDaPágina()* (parte da interface *cálculoDeServiço*) poderia colaborar com um componente **TabelaDePreço** contendo informações sobre os preços de serviços. A operação *verificarPrioridade()* (parte da interface *iniciarServiço*) poderia colaborar com um componente **FilaDeServiço** para determinar os tipos e prioridades dos serviços atualmente em espera para produção.

A atividade de elaboração é aplicada a todos os componentes definidos como parte do projeto da arquitetura. Uma vez concluída, aplica-se uma maior elaboração a cada atributo, operação e interface. Devem ser especificadas as estruturas de dados apropriadas para cada atributo. Além disso, é desenvolvido o detalhe algorítmico exigido para implementar a lógica de processamento associada a cada operação. Essa atividade de projeto procedural é discutida posteriormente, ainda neste capítulo. Por fim, desenvolvem-se os mecanismos necessários para implementar a interface. Para software orientado a objetos, isso poderia englobar a descrição de todas as mensagens necessárias para efetivar a comunicação entre objetos de um sistema.

14.1.2 A visão tradicional

No contexto da engenharia de software tradicional, componente é o elemento funcional de um programa que incorpora a lógica de processamento, as estruturas de dados internas necessárias para implementar a lógica de processamento e uma interface que permite chamar o componente e passar dados a ele. Um componente tradicional, também denominado *módulo*, reside na arquitetura de software e se presta a um de três importantes papéis: (1) um componente de controle que coordena a chamada de todos os demais componentes do domínio do problema, (2) um componente de domínio do problema que implementa uma função completa ou parcial solicitada pelo cliente ou (3) um componente de infraestrutura responsável por funções que dão suporte ao processamento necessário no domínio do problema.

Assim como os componentes orientados a objetos, os componentes de software tradicionais são obtidos a partir do modelo de análise. Nesse caso, entretanto, o elemento de elaboração de componentes do modelo de análise serve como base para essa obtenção. Cada componente representado na hierarquia de componentes é mapeado (Seção 13.6) em uma hierarquia de módulos.

Os componentes de controle (módulos) ficam próximos ao alto da hierarquia (arquitetura de programas) e os componentes de domínio do problema

"Invariavelmente, constata-se que um sistema complexo que funciona é a evolução de um sistema simples que funcionava."

John Gall

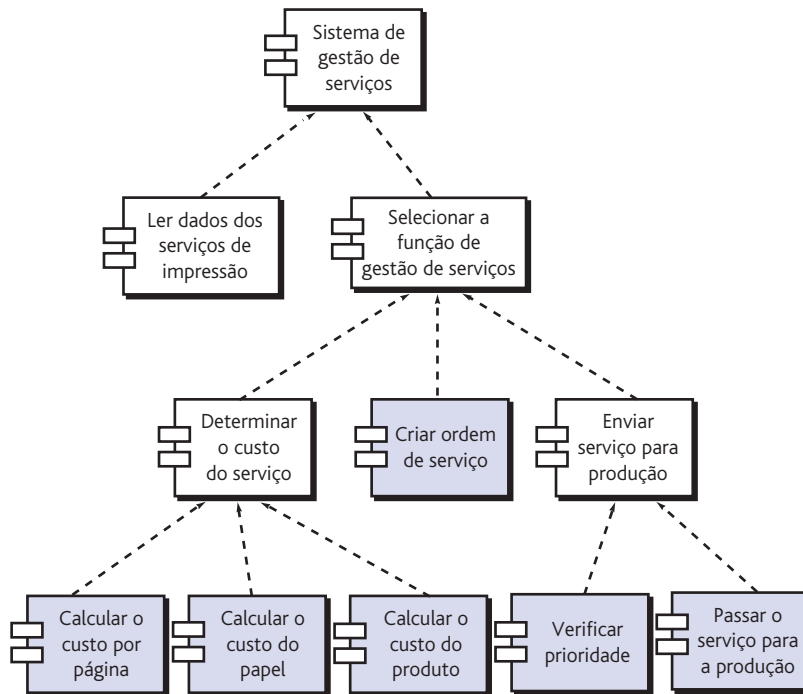


FIGURA 14.2 Diagrama de estruturas para um sistema tradicional.

tendem a ficar mais próximos da parte inferior da hierarquia. Para obter modularidade efetiva, conceitos de projeto, como independência funcional (Capítulo 12), são aplicados durante a elaboração dos componentes.

Para ilustrarmos o processo de elaboração de projetos para componentes tradicionais, consideremos novamente o software a ser criado para uma gráfica sofisticada. É produzida uma arquitetura hierárquica, mostrada na Figura 14.2. Cada retângulo representa um componente de software. Observe que os retângulos sombreados são equivalentes, em termos funcionais, às operações definidas para a classe **ServiçoDeImpressão** discutidas na Seção 14.1.1. Nesse caso, entretanto, cada operação é representada como um módulo separado, chamado conforme indicado na figura. São usados outros módulos para controlar o processamento, e estes são, portanto, componentes de controle.

Durante o projeto de componentes, é elaborado cada módulo da Figura 14.2. A interface de módulos é definida explicitamente. Ou seja, é representado cada objeto de dados ou de controle que flui através da interface. São definidas as estruturas de dados utilizadas internamente no módulo. O algoritmo que possibilita ao módulo cumprir sua função é desenhado usando-se o método de refinamento gradual discutido no Capítulo 12. Algumas vezes o comportamento do módulo é representado usando-se um diagrama de estado.

Para ilustrarmos o processo, consideremos o módulo **CalcularCustoPorPágina**. O intuito é calcular o custo de impressão por página tomando como base as especificações fornecidas pelo cliente. Os dados necessários para realizar essa função são: o número de páginas contidas no documento, o número

À medida que o projeto para cada componente de software é elaborado, o foco passa para o projeto de estruturas de dados específicas e para o projeto procedural para a manipulação de estruturas de dados. Entretanto, não se esqueça da arquitetura que deve abrigar os componentes ou as estruturas de dados globais que podem servir a vários componentes.

total de documentos a ser produzidos, impressão frente ou frente e verso, requisitos de cores, requisitos de tamanho. Esses dados são passados para *CalcularCustoPorPágina* via interface do módulo. *CalcularCustoPorPágina* os utiliza para determinar o custo de uma página com base no tamanho e na complexidade do trabalho – uma função de todos os dados passados para o módulo via interface. O custo por página é inversamente proporcional ao tamanho do serviço e diretamente proporcional à complexidade do serviço.

A Figura 14.3 representa o projeto de componentes usando uma notação UML modificada. O módulo *CalcularCustoPorPágina* acessa dados chamando o módulo *obterDadosDoServiço*, o qual possibilita passar todos os dados relevantes para o componente, e uma interface de banco de dados, *acessarBDCustos*, a qual possibilita que o módulo acesse um banco de dados contendo todos os custos de impressão. À medida que o projeto continua, o módulo *CalcularCustoPorPágina* é elaborado para fornecer detalhes algorítmicos e detalhes de interface (Figura 14.3). Os detalhes algorítmicos podem ser representados usando-se o texto em pseudocódigo mostrado na figura ou por meio de um diagrama de atividades UML. As interfaces são representadas como um conjunto de itens ou objetos de dados de entrada/saída. A elaboração do projeto continua até que detalhes suficientes tenham sido fornecidos para orientar a construção do componente.

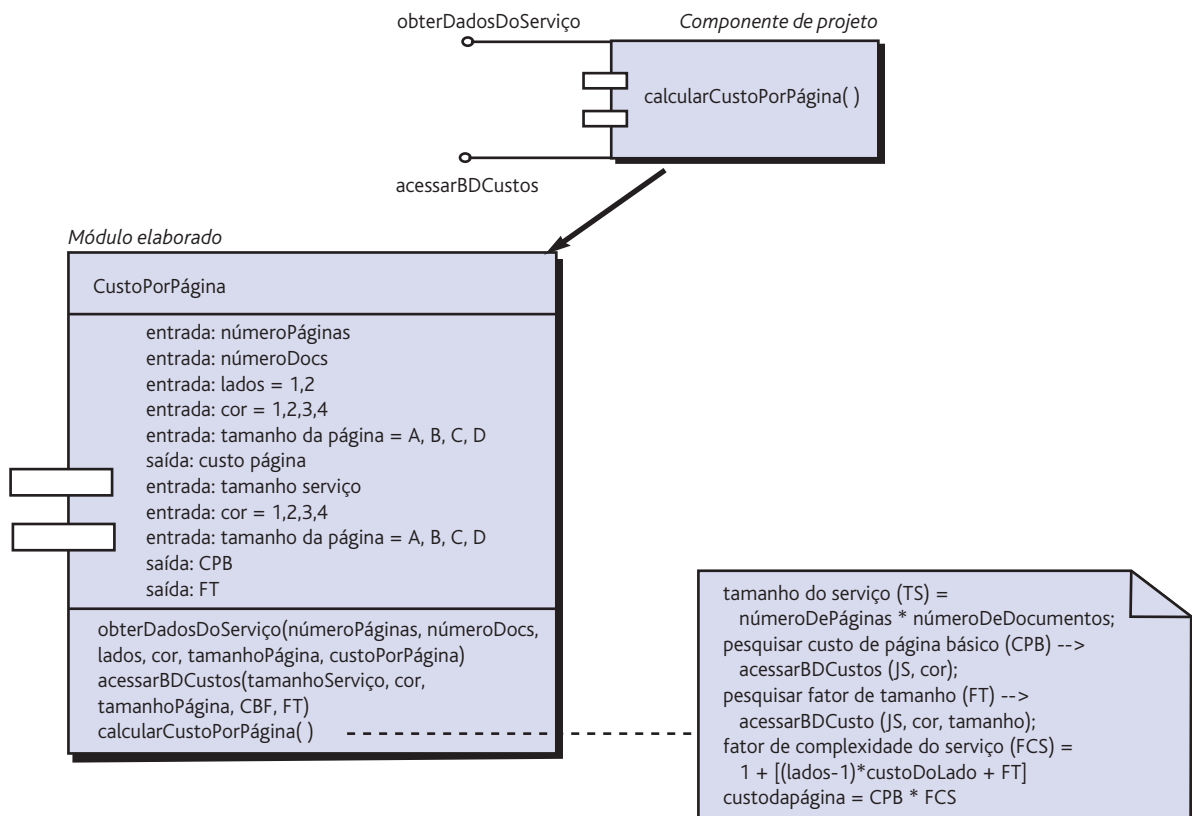


FIGURA 14.3 Projeto de componentes para *CalcularCustoPorPágina*.

14.1.3 Uma visão relacionada a processos

As visões tradicionais e orientadas a objetos de projeto de componentes apresentadas nas Seções 14.1.1 e 14.1.2 partem do pressuposto de que o componente está sendo projetado a partir do zero. Isto é, temos de criar um componente com base nas especificações obtidas do modelo de requisitos. Existe, obviamente, outra abordagem.

Ao longo das três últimas décadas, a comunidade de engenharia de software tem enfatizado a necessidade de se construir sistemas que façam uso de componentes de software ou de padrões de projeto existentes. Em essência, é colocado à disposição dos profissionais de software um catálogo de projetos ou código de componentes de qualidade comprovada à medida que o trabalho de projeto prossegue. Conforme a arquitetura de software é desenvolvida, escolhemos componentes ou padrões de projeto desse catálogo e os usamos para preencher a arquitetura. Pelo fato de esses componentes terem sido criados tendo a reusabilidade em mente, uma descrição completa de suas interfaces, a(s) função(ões) por eles realizada(s) e a comunicação e colaboração por eles exigidas estarão à nossa disposição. Discutiremos posteriormente, na Seção 14.6, alguns dos importantes aspectos da engenharia de software baseada em componentes (CBSE, component-based software engineering).

INFORMAÇÕES



Padrões e estruturas baseadas em componentes

Um dos elementos-chave que levam ao sucesso ou insucesso da CBSE é a disponibilidade de padrões baseados em componentes, algumas vezes denominados *middleware*. *Middleware* é um conjunto de componentes de infraestrutura que possibilita que os componentes do domínio do problema se comuniquem entre si por meio de uma rede ou em um sistema complexo. Os engenheiros de software que quiserem usar desenvolvimento baseado em componentes como processo de software poderão escolhê-los entre os seguintes padrões:

OMG CORBA – www.corba.org/

Microsoft COM – <http://www.microsoft.com/com/default.mspix>

Microsoft.NET – <http://msdn.microsoft.com/en-us/netframework/default.aspx>

Sun JavaBeans – <http://www.oracle.com/technetwork/java/javaee/ejb/index.html>

Os sites citados apresentam uma ampla variedade de tutoriais, artigos, ferramentas e recursos gerais sobre esses importantes padrões de *middleware*.

14.2 Projeto de componentes baseados em classes

Conforme já citado, o projeto de componentes apoia-se nas informações desenvolvidas como parte do modelo de requisitos (Capítulos 9 a 11) e representadas como parte do modelo da arquitetura (Capítulo 13). Quando é escolhida uma abordagem de engenharia de software orientada a objetos, o projeto de componentes se concentra na elaboração de classes específicas do domínio do problema e na definição e no refinamento de classes de infraestrutura contidas no modelo de requisitos. A descrição detalhada dos atributos, operações e interfaces utilizados por essas classes é o detalhe de projeto exigido como precursor da atividade de construção.

14.2.1 Princípios básicos de projeto

Quatro princípios básicos de projeto são aplicáveis ao projeto de componentes e têm sido amplamente adotados quando se aplica à engenharia de software orientada a objetos. A motivação por trás da aplicação desses princípios é criar projetos mais fáceis de modificar e reduzir a propagação de efeitos colaterais na ocorrência de modificações. Podemos usar tais princípios como guias, à medida que cada componente de software é desenvolvido.

Princípio do Aberto-Fechado (OCP, Open-Closed Principle). “Um módulo/componente deve ser aberto para a extensão, mas fechado para modificações” [Mar00]. Essa afirmação pode parecer uma contradição, mas representa uma das características mais importantes de um bom projeto de componentes. Em outras palavras, devemos especificar o componente para permitir que ele seja estendido (em seu domínio funcional) sem a necessidade de fazer modificações internas (em nível de código ou lógica) no próprio componente. Para tanto, criamos abstrações que servem como um divisor entre a funcionalidade que provavelmente será estendida e a classe de projeto em si.

Por exemplo, suponhamos que a função de segurança do *CasaSegura* faça uso de uma classe **Detector** que deve verificar o estado de cada tipo de sensor de segurança. É provável que, à medida que o tempo for passando, o número e os tipos de sensores de segurança cresçam. Se a lógica de processamento interna for implementada como uma sequência de construções se-então-senão, cada uma delas tratando de um tipo de sensor diferente, a adição de um novo tipo de sensor exigirá lógica de processamento interna adicional (ainda outro se-então-senão). Isso é uma violação do OCP.

Uma forma de concretizar o OCP para a classe **Detector** está ilustrada na Figura 14.4. A interface sensor apresenta uma visão consistente dos sensores para o componente detector. Se for adicionado um novo tipo de sensor, nenhuma mudança será necessária na classe **Detector** (componente). O OCP é preservado.

Princípio da Substituição de Liskov (LSP, Liskov Substitution Principle). “As subclasses devem ser substitutas de suas classes-base” [Mar00]. Esse princípio de projeto, originalmente proposto por Barbara Liskov [Lis88], sugere

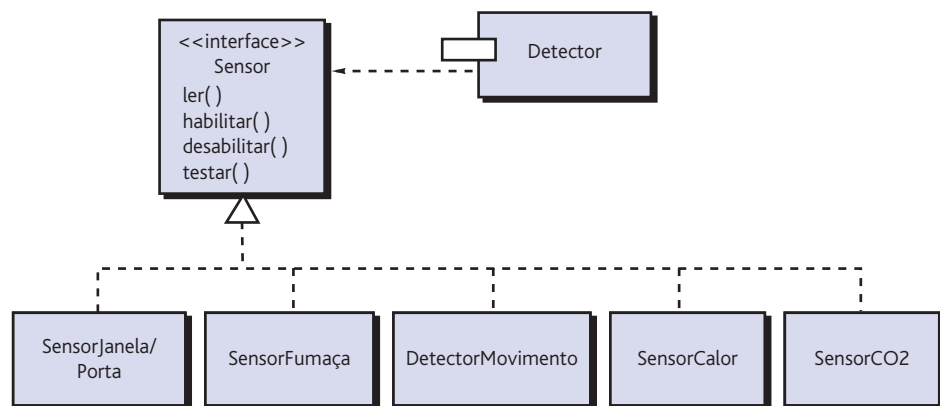


FIGURA 14.4 Seguindo o OCP.

CASASEGURA

**O OCP em ação****Cena:** Sala do Vinod.**Atores:** Vinod e Shakira – membros da equipe de engenharia de software do CasaSegura.**Conversa:****Vinod:** Acabei de receber uma ligação do Doug [o gerente da equipe]. Ele me disse que o pessoal de marketing quer acrescentar um novo sensor.**Shakira (com um sorriso de superioridade):** Outra vez não!**Vinod:** Isso mesmo... e você não vai acreditar no que esses caras inventaram.**Shakira:** Qual é a surpresa agora?**Vinod (rindo):** Algo que eles chamaram de sensor de mal-estar de cachorro.**Shakira:** O quê...?**Vinod:** Destina-se a pessoas que deixam seus bichinhos de estimação em apartamentos ou condomínios ou casas que são próximas umas das outras. O cachorro começa a latir. O vizinho se irrita e reclama. Com esse sensor, se o cachorro latir por, digamos, mais de um minuto, o sensor ativa um modo de alarme especial que chama o dono no seu celular.**Shakira:** Você tá brincando, não é?**Vinod:** Não. Doug quer saber quanto tempo levará para acrescentar essa característica à função de segurança.**Shakira (pensando um pouco):** Não muito... Veja. [Ela mostra ao Vinod a Figura 14.4.] Isolamos as verdadeiras classes de sensores atrás da interface **sensor**. Desde que tenhamos as especificações para o sensor de cachorros, acrescentá-lo deve ser moleza. A única coisa que terei de fazer é criar um componente apropriado... quer dizer, uma classe, para ele. Nenhuma mudança no componente **Detector** em si.**Vinod:** Então, vou dizer ao Doug que não é nada de outro mundo.**Shakira:** Conhecendo o Doug, ele nos manterá focados e não vai liberar essa coisa até a próxima versão.**Vinod:** Isso não é ruim, mas você consegue implementá-lo agora se ele quiser?**Shakira:** Sim, o jeito que projetamos a interface me permite fazê-lo sem grandes complicações.**Vinod (pensando um pouco):** Você já ouviu falar do princípio do aberto-fechado?**Shakira (encolhendo os ombros):** Nunca ouvi falar.**Vinod (sorrindo):** Sem problemas.

que um componente que usa uma classe-base deve continuar a funcionar apropriadamente caso uma classe derivada da classe-base seja passada para o componente em seu lugar. O LSP exige que qualquer classe derivada de uma classe-base deve honrar qualquer contrato implícito entre a classe-base e os componentes que a utilizam. No contexto desta discussão, um “contrato” é uma pré-condição que deve ser verdadeira antes de o componente usar uma classe-base e uma pós-condição que deve ser verdadeira após o componente usar uma classe-base. Ao criar classes derivadas, certifique-se de que atendem às pré-condições e às pós-condições.

Princípio da Inversão da Dependência (DIP, Dependency Inversion Principle). “Dependa de abstrações. Não dependa de concretizações” [Mar00]. Como vimos na discussão sobre OCP, é nas abstrações que um projeto pode ser estendido sem grandes complicações. Quanto mais um componente depender de outros componentes concretos (e não de abstrações, como uma interface), mais difícil será estendê-lo.

Princípio da Segregação de Interfaces (ISP, Interface Segregation Principle). “É melhor usar várias interfaces específicas do cliente do que uma única interface de propósito geral” [Mar00]. Há diversas ocasiões em que componentes para vários clientes usam uma operação fornecida por uma classe-servido-

Se você dispensa o projeto e vai direto ao código, lembre-se apenas de que código é a “concretização” final. Você estará violando o DIP.

Projetar componentes para reutilização requer mais do que um bom projeto técnico. Essa atividade também requer mecanismos de controle de configuração eficazes (Capítulo 29).

ra. O ISP sugere a criação de uma interface especializada para atender cada categoria principal de clientes. Apenas as operações que forem relevantes para determinada categoria de clientes devem ser especificadas na interface para esse cliente. Se vários clientes precisarem das mesmas operações, estas devem ser especificadas em cada uma das interfaces especializadas.

Como exemplo, considere a classe **Planta** usada para as funções de segurança e vigilância do *CasaSegura* (Capítulo 10). Para as funções de segurança, **Planta** é usada apenas durante atividades de configuração e usa as operações *colocarDispositivo()*, *mostrarDispositivo()*, *agruparDispositivo()* e *removerDispositivo()* para inserir, mostrar, agrupar e remover sensores da planta. A função de vigilância do *CasaSegura* usa as quatro operações citadas para segurança; porém, também exige operações especiais para gerenciar câmeras: *mostrarFOV()* e *mostrarIDDispositivo()*. Portanto, o ISP sugere que componentes clientes das duas funções do *CasaSegura* tenham interfaces especializadas definidas para eles. A interface para segurança englobaria apenas as operações *colocarDispositivo()*, *mostrarDispositivo()*, *agruparDispositivo()* e *removerDispositivo()*. A interface para vigilância incorporaria as operações *colocarDispositivo()*, *mostrarDispositivo()*, *agruparDispositivo()* e *removerDispositivo()*, juntamente com *mostrarFOV()* e *mostrarIDDispositivo()*.

Embora os princípios de projeto de componentes sejam úteis em termos de orientação, os componentes em si não vivem de forma isolada. Em muitos casos, componentes ou classes individuais são organizados em subsistemas ou pacotes. Faz sentido perguntar como deve ocorrer a atividade de empacotamento. Exatamente de que forma os componentes devem ser organizados à medida que o projeto prossegue? Martin [Mar00] sugere outros princípios de empacotamento aplicáveis ao projeto de componentes. Os princípios são os seguintes.

Princípio da Equivalência de Reutilização de Versões (REP, Release Reuse Equivalency Principle). “A granularidade da reutilização é a granularidade da versão” [Mar00]. Quando as classes ou os componentes são projetados tendo em vista a reutilização, é estabelecido um contrato implícito entre o desenvolvedor da entidade reutilizável e quem vai usá-la. O desenvolvedor se compromete a estabelecer um sistema de controle de versões que ofereça suporte e manutenção para as versões mais antigas da entidade, enquanto os usuários vão atualizando gradualmente para a versão mais recente. Em vez de tratar cada uma dessas classes individualmente, em geral é recomendável agrupar classes reutilizáveis em pacotes que possam ser gerenciados e controlados à medida que versões mais recentes evoluam.

Princípio do Fechamento Comum (CCP, Common Closure Principle). “Classes que mudam juntas, devem ficar juntas” [Mar00]. As classes devem ser empacotadas de forma coesa. Ao serem empacotadas como parte de um projeto, devem tratar da mesma área funcional ou comportamental. Quando alguma característica dessa área tiver de mudar, é provável que apenas as classes contidas no pacote precisem ser modificadas. Isso leva a um controle de mudanças e gerenciamento de versões mais eficiente.

Princípio Comum da Reutilização (CRP, Common Reuse Principle). “As classes que não são reutilizadas juntas não devem ser agrupadas juntas” [Mar00]. Quando uma ou mais classes com um pacote muda(m), o número da versão do pacote muda. Todas as demais classes ou pacotes que dependem do pacote alterado agora precisam ser atualizadas para a versão mais recente do pacote e testadas para garantir que a nova versão opere sem incidentes. Se as classes não forem agrupadas de forma coesa, é possível que uma classe sem nenhuma relação com as demais contidas em um pacote seja alterada. Isso precipitará integração e testes desnecessários. Por essa razão, apenas as classes reutilizadas juntas devem ser incluídas em um pacote.

14.2.2 Diretrizes para o projeto de componentes

Além dos princípios discutidos na Seção 14.2.1, pode-se aplicar um conjunto de diretrizes de projeto pragmáticas à medida que o projeto de componentes prossegue. Tais diretrizes se aplicam a componentes, suas interfaces e às características de dependência e herança que têm algum impacto sobre o projeto resultante. Ambler [Amb02b] sugere as seguintes diretrizes:

Componentes. Devem-se estabelecer convenções de nomes para componentes especificados como parte do modelo de arquitetura e, então, refiná-los e elaborá-los como parte do modelo de componentes. Os nomes de componentes de arquitetura devem ser extraídos do domínio do problema e ter significado para todos os envolvidos que veem o modelo de arquitetura. Por exemplo, o nome de classe **Planta** é significativo para qualquer um que o leia, independentemente de sua bagagem técnica. Por outro lado, componentes de infraestrutura ou classes elaboradas no nível de componentes devem receber nomes que reflitam significados específicos à implementação. Se uma lista ligada tiver de ser gerenciada como parte da implementação **Planta**, a operação *gerenciarLista()* é apropriada, mesmo que uma pessoa não técnica possa interpretá-la de forma incorreta.³

Podemos optar pelo uso de estereótipos para auxiliar na identificação da natureza dos componentes no nível de projeto mais detalhado. Por exemplo, poderíamos usar `<<infrastructure>>` para identificar um componente de infraestrutura; `<<database>>` para identificar um banco de dados que atenda a uma ou mais classes de projeto ou o sistema inteiro; e `<<table>>` poderia ser usado para identificar uma tabela em um banco de dados.

Interfaces. As interfaces nos fornecem importantes informações sobre a comunicação e a colaboração (bem como nos ajuda a alcançar o OPC). Entretanto, a representação totalmente livre de interfaces tende a complicar os diagramas de componentes. Ambler [Amb02c] recomenda o seguinte: (1) a representação “pirulito” de uma interface deve ser usada em conjunto com a abordagem mais formal da UML, que usa retângulos e setas pontilhadas, quando os diagramas se tornam mais complexos; (2) por consistência, as interfaces devem fluir da esquerda para a direita do retângulo do com-

O que devemos considerar ao atribuir nomes aos componentes?

³ É pouco provável que alguém da área de marketing ou da empresa do cliente (uma pessoa não técnica) examine informações detalhadas do projeto.

ponente; e (3) devem ser mostradas apenas as interfaces relevantes para o componente em consideração, mesmo que outras estejam disponíveis. Tais recomendações destinam-se a simplificar a natureza visual dos diagramas de componentes da UML.

Dependências e herança. Para melhorar a legibilidade, é aconselhável modelar as dependências da esquerda para a direita e as heranças de baixo (classes derivadas) para cima (classes-base). Além disso, as interdependências dos componentes devem ser representadas por meio de interfaces e não por meio da representação de uma dependência componente-para-componente. Seguindo a filosofia do OCP, isso facilitará a manutenção do sistema.

14.2.3 Coesão

No Capítulo 12, descrevemos coesão como o “foco único” de um componente. No contexto do projeto de componentes para sistemas orientados a objetos, *coesão* implica um componente ou classe encapsular apenas atributos e operações que estejam intimamente relacionados entre si e com a classe ou o componente em si. Lethbridge e Laganière [Let01] definem uma série de tipos diferentes de coesão (enumerados em ordem de nível de coesão):⁴

Funcional. Apresentado basicamente por operações, este nível de coesão ocorre quando um módulo efetua um e apenas um cálculo e então retorna um resultado.

De camadas. Apresentado por pacotes, componentes e classes, esse tipo de coesão ocorre quando uma camada mais alta acessa os serviços de uma camada mais baixa, porém as camadas mais baixas não acessam as mais elevadas. Consideremos, por exemplo, que a função de segurança do *CasaSegura* precise fazer uma ligação telefônica caso um alarme seja acionado. Seria possível definirmos um conjunto de pacotes em camadas conforme mostra a Figura 14.5. Os pacotes sombreados contêm componentes de infraestrutura. O acesso é do pacote de painel de controle para baixo.

De comunicação. Todas as operações que acessam os mesmos dados são definidas em uma classe. Em geral, tais classes se concentram exclusivamente nos dados em questão, acessando-os e armazenando-os.

As classes e componentes que apresentam coesão funcional, de camadas e comunicação são relativamente fáceis de ser implementados, testados e mantidos. Devemos nos esforçar ao máximo para atingir esses níveis de coesão, sempre que possível. É importante notar, entretanto, que questões pragmáticas de projeto e implementação algumas vezes nos forçam a optar por níveis de coesão mais baixos.

Embora a compreensão dos vários níveis de coesão seja instrutiva, no projeto de componentes é mais importante estar atento ao conceito geral. Mantenha o nível de coesão o mais elevado possível.

⁴ Em geral, quanto maior o nível de coesão, mais fácil é implementar, testar e manter um componente.

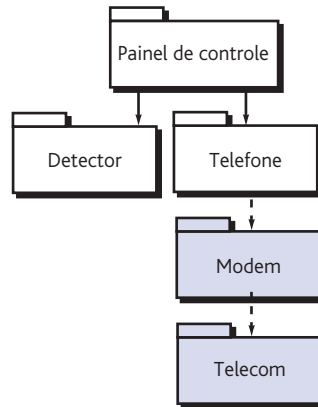


FIGURA 14.5 Coesão de camadas.

CASASEGURA



Coesão em ação

Cena: Sala do Jamie.

Atores: Jamie e Ed – membros da equipe de engenharia de software do *CasaSegura* que estão trabalhando na função de vigilância.

Conversa:

Ed: Tenho um projeto preliminar do componente **câmera**.

Jamie: Quer fazer uma rápida revisão?

Ed: Creio que sim... mas, na realidade, gostaria de sua opinião sobre algo.

(Jamie gesticula para que ele continue.)

Ed: Originalmente definimos cinco operações para **câmera**. Olha...

determinarTipo() informa o tipo de câmera;

traduzirLocalização() possibilita que a câmera seja movimentada pela planta;

exibirID() obtém o ID da câmera e o exibe próximo do ícone câmera;

exibirVisão() mostra graficamente o campo de visão da câmera;

exibirZoom() mostra a ampliação da câmera graficamente.

Ed: Projetei cada uma delas separadamente e são operações bastante simples. Portanto, imagino que seria uma boa ideia combinar todas as operações de exibição em apenas uma chamada *exibirCâmera()* – ela mostrará o ID, a vista e a ampliação. O que você acha?

Jamie (fazendo uma careta): Não tenho certeza de que seja uma ideia tão boa assim.

Ed (franzindo a testa): Por quê? Todas essas pequenas operações podem nos dar dor de cabeça.

Jamie: O problema de combiná-las é que perdermos coesão, sabe, a operação *exibirCâmera()* não será focada.

Ed (ligeiramente exasperado): E daí? O conjunto todo terá menos de 100 linhas de código-fonte, no máximo. Será mais fácil implementá-lo, eu acho.

Jamie: E se o Marketing decidir mudar a maneira como representamos o campo de visão?

Ed: Simplesmente mexo na operação *exibirCâmera()* e faço a modificação.

Jamie: E os efeitos colaterais?

Ed: O que você quer dizer com isso?

Jamie: Bem, digamos que você faça a modificação, mas, inadvertidamente, crie um problema na exibição do ID.

Ed: Eu não seria tão descuidado assim.

Jamie: Talvez não, mas e se daqui a dois anos alguém do suporte tiver de fazer a modificação? Pode ser que não entenda a operação tão bem quanto você e, vai saber, talvez ele seja descuidado.

Ed: Então você é contrário a isso?

Jamie: Você é o projetista... a decisão é sua... apenas certifique-se de ter compreendido as consequências da baixa coesão.

Ed (refletindo um pouco): Talvez seja melhor mesmo fazer duas operações de exibição separadas.

Jamie: Ótima decisão.

14.2.4 Acoplamento

Em discussões anteriores sobre análise e projeto, observamos que a comunicação e a colaboração são elementos essenciais de qualquer sistema orientado a objetos. Há, entretanto, o lado sinistro dessa importante (e necessária) característica. Como o volume de comunicação e colaboração aumenta (isto é, à medida que o grau de “conexão” entre as classes aumenta), a complexidade do sistema também cresce. E à medida que a complexidade aumenta, a dificuldade de implementação, testes e manutenção do software também aumenta.

O *acoplamento* é uma medida qualitativa do grau com que as classes estão ligadas entre si. Conforme as classes (e os componentes) se tornam mais interdependentes, o acoplamento aumenta. Um objetivo importante no projeto de componentes é manter o acoplamento o mais baixo possível.

O *acoplamento de classes* pode se manifestar de uma série de formas. Lethbridge e Laganière [Let01] definem um espectro de categorias de acoplamento: por exemplo, o *acoplamento de conteúdo* ocorre quando um componente “modifica de forma sub-reptícia os dados internos de outro componente” [Let01]. Isso viola o encapsulamento – um conceito de projeto básico. O *acoplamento de controle* ocorre quando a operação *AO* chama a operação *BO* e passa um flag de controle para *B*. O flag de controle “dirige”, então, a lógica de fluxo no interior de *B*. O problema dessa forma de acoplamento é que uma mudança não relacionada em *B* pode resultar na necessidade de alterar o significado do flag de controle passado por *A*. Se isso for menosprezado, acontecerá um erro. O *acoplamento externo* ocorre quando um componente se comunica ou colabora com componentes de infraestrutura (por exemplo, funções do sistema operacional, capacidade de bancos de dados, funções de telecomunicação). Embora esse tipo de acoplamento seja necessário, deve se limitar a um pequeno número de componentes ou classes em um sistema.

Um software deve se comunicar interna e externamente. Consequentemente, acoplamento é uma realidade a ser enfrentada. Entretanto, o projetista deve se esforçar para reduzir o acoplamento sempre que possível e compreender as ramificações do acoplamento elevado, quando não puder ser evitado.

CASASEGURA



Acoplamento em ação

Cena: Sala da Shakira.

Atores: Vinod e Shakira – membros da equipe de engenharia de software do CasaSegura que estão trabalhando na função de segurança.

Conversa:

Shakira: Pensei que tive uma grande ideia... depois refleti um pouco mais a respeito e me pareceu não ser uma ideia tão boa assim. Por fim, a rejeitei, mas pensei que deveria apresentá-la a você.

Vinod: Certamente. Qual é a ideia?

Shakira: Bem, cada um dos sensores reconhece uma condição de alarme de algum tipo, certo?

Vinod (sorrindo): É por isso que os chamamos de sensores, Shakira.

Shakira (exasperada): Sarcasmo, Vinod, você tem que trabalhar suas habilidades interpessoais.

Vinod: Voltando ao que você dizia...

Shakira: Certo, de qualquer modo, imaginei... por que não criar uma operação em cada objeto sensor, chamada

`fazerChamada()`, que colaboraria diretamente com o componente **ChamadaExterna**, enfim, com uma interface para o componente **ChamadaExterna**.

Vinod (pensativo): Você quer dizer, em vez de fazer com que a colaboração ocorra fora de um componente como **PainelControle** ou algo do gênero?

Shakira: Exato... mas depois pensei que cada objeto sensor será associado ao componente **ChamadaExterna** e que isso significa que ele está indiretamente acoplado ao mundo exterior e... bem, apenas imaginei que isso tornaria as coisas mais complicadas.

Vinod: Concordo. Nesse caso, é melhor deixar que a interface de sensores passe informações para o **PainelControle** e deixe que ele inicie a chamada telefônica. Além disso, diferentes sensores talvez resultem em diferentes números de telefone. Você não vai querer que o sensor armazene essas informações... se elas mudarem...

Shakira: Parece que isso não está certo.

Vinod: A heurística de projeto para acoplamento nos diz que não é correto.

Shakira: Que seja...

14.3 Condução de projetos de componentes

No início deste capítulo citamos que o projeto de componentes é de natureza elaborada. Temos de transformar informações de modelos de arquitetura e requisitos em uma representação de projeto que nos dê detalhes suficientes para orientar a atividade da construção (codificação e testes). As etapas a seguir representam um conjunto de tarefas típico para um projeto de componentes – quando ele é aplicado a um sistema orientado a objetos.

Etapla 1. Identificar todas as classes de projeto correspondentes ao domínio do problema. Usando o modelo de requisitos e de arquitetura, cada classe de análise e componente de arquitetura é elaborada conforme descrito na Seção 14.1.1.

Etapla 2. Identificar todas as classes de projeto correspondentes ao domínio de infraestrutura. Essas classes não são descritas no modelo de requisitos e normalmente não estão presentes no modelo de arquitetura; porém, têm de ser descritas neste ponto. Conforme já dito, entre as classes e componentes dessa categoria temos componentes de interfaces gráficas do usuário (muitas vezes disponíveis na forma de componentes reutilizáveis), componentes de sistemas operacionais, bem como componentes de administração de dados e objetos.

Etapla 3. Elaborar todas as classes de projeto que não são obtidas como componentes reutilizáveis. A elaboração exige que todas as interfaces, atributos e operações necessários para implementar a classe sejam descritos em detalhes. A heurística de projeto (por exemplo, coesão e acoplamento de componentes) deve ser considerada à medida que essa tarefa é conduzida.

Etapla 3a. Especificar detalhes de mensagens quando classes ou componentes colaboram entre si. O modelo de requisitos faz uso de um diagrama de colaboração para mostrar como as classes de análise colaboram entre si. À medida que o projeto de componentes prossegue, algumas vezes é útil mostrar os detalhes dessas colaborações especificando a estrutura das mensagens passadas entre objetos de um sistema. Embora essa atividade de projeto seja opcional, pode ser utilizada como precursora da especificação de

“Se eu tivesse mais tempo, teria escrito uma carta mais curta.”

Blaise Pascal

Se estiver trabalhando em um ambiente não orientado a objetos, as três primeiras etapas concentram-se no refinamento de objetos de dados e funções de processamento (transformações) identificadas como parte do modelo de análise.

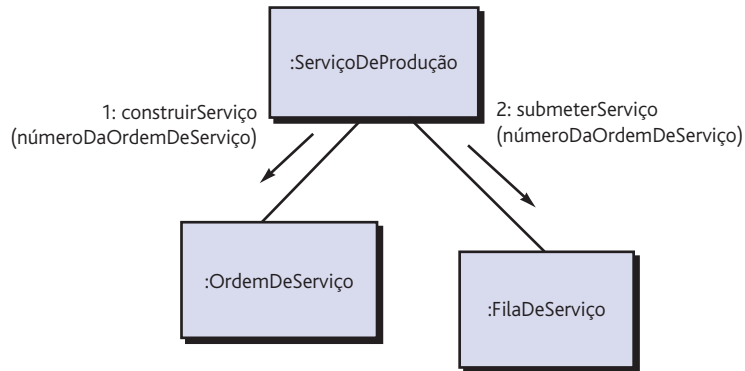


FIGURA 14.6 Diagrama de colaboração com as mensagens.

interfaces que mostra como os componentes em um sistema se comunicam e colaboram entre si.

A Figura 14.6 ilustra um diagrama de colaboração simples para o sistema de impressão discutido anteriormente. Três objetos, **ServiçoDeProdução**, **OrdemDeServiço** e **FilaDeServiço**, colaboram na preparação de um serviço de impressão a ser submetido ao fluxo de produção. São passadas mensagens entre os objetos, conforme ilustrado pelas setas da figura. Durante a modelagem de requisitos, as mensagens são especificadas conforme mostra a figura. Entretanto, à medida que o projeto prossegue, cada mensagem é elaborada por meio da expansão de sua sintaxe, da seguinte maneira [Ben02]:

[condição de guarda] sequência de expressões (valor de retorno):=

nome da mensagem (lista de argumentos)

onde uma [condição de guarda] é escrita em Linguagem de Restrição de Objetos (OCL, Object Constraint Language)⁵ e especifica qualquer conjunto de condições que devem ser atendidas antes de a mensagem poder ser enviada; sequência de expressões é um valor inteiro (ou outro indicador de ordem, por exemplo, 3.1.2) que indica a ordem sequencial em que uma mensagem é enviada; (valor de retorno) é o nome das informações retornadas por uma operação chamada pela mensagem; nome da mensagem identifica uma operação a ser chamada; e (lista de argumentos) é a lista de atributos passados para a operação.

Etapa 3b. Identificar interfaces adequadas para cada componente. No contexto do projeto de componentes, uma interface UML é “um grupo de operações externamente visíveis (públicas). A interface não contém nenhuma estrutura interna, nenhum atributo, nenhuma associação...” [Ben02]. Colocado mais formalmente, interface equivale a uma classe abstrata que fornece uma conexão controlada entre as classes de projeto. A elaboração de interfaces está ilustrada na Figura 14.1. Em essência, operações definidas para o diagrama de classes são classificadas em uma ou mais classes abstratas. Todas

⁵ A OCL é discutida brevemente no Apêndice 1.

as operações contidas em uma classe abstrata (a interface) devem ser coesas; elas devem apresentar processamento focado em uma função ou subfunção limitada.

Voltando à Figura 14.1, pode-se dizer que a interface *iniciarServiço* não apresenta coesão suficiente. Na realidade, ela realiza três subfunções distintas – criar uma ordem de serviço, verificar a prioridade do serviço e passar um serviço para produção. O projeto de interfaces deve ser refatorado. Uma abordagem poderia ser reexaminar as classes de projeto e definir uma nova classe **OrdemDeServiço** que cuidaria de todas as atividades associadas à montagem de uma ordem de serviço. A operação *construirOrdemDeServiço()* passa a fazer parte dessa classe. De modo similar, poderíamos definir uma classe **FilaDeServiço** que incorporaria a operação *verificarPrioridade()*. Uma classe **ServiçoDeProdução** englobaria todas as informações associadas a um serviço de produção a ser passado para o centro de produção. A interface *iniciarServiço* assumiria, então, a forma indicada na Figura 14.7. A interface *iniciarServiço* agora é coesa, focalizando uma função. As interfaces associadas a **ServiçoDeProdução**, **OrdemDeServiço** e **FilaDeServiço** são similarmente coesas.

Etapa 3c. Elaborar atributos e definir tipos de dados e estruturas de dados necessárias para implementá-los. Em geral, as estruturas e os tipos de dados utilizados para definir atributos são definidos no contexto da linguagem de programação que será usada para implementação. A UML define o tipo de dados de um atributo usando a seguinte sintaxe:

nome : tipo da expressão = valor inicial {string de propriedades}

onde nome é o nome do atributo, tipo da expressão é o tipo de dados, valor inicial é o valor que o atributo assume quando um objeto é criado e string de propriedades define uma propriedade ou característica do atributo.

Durante a primeira iteração do projeto de componentes, os atributos normalmente são descritos por nomes. Referindo-nos mais uma vez à Figura 14.1, a lista de atributos para **ServiçoDeImpressão** enumera apenas os nomes dos

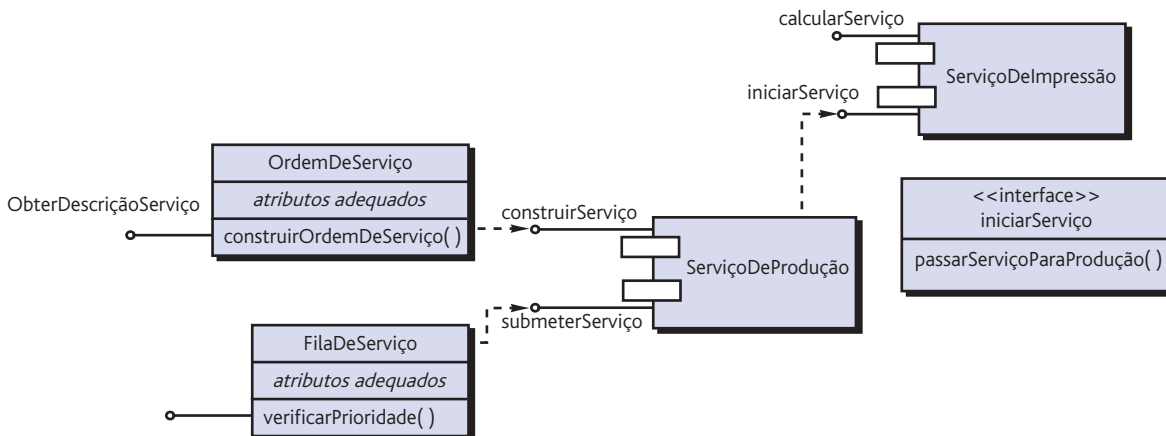


FIGURA 14.7 Interfaces de refatoração e definições de classes para ServiçoDeImpressão.

atributos. Entretanto, à medida que a elaboração do projeto prossegue, cada atributo é definido usando-se o formato de atributo UML citado. Por exemplo, `peso-TipoDoPapel` é definido da seguinte maneira:

```
peso-TipoDoPapel: string = "A" { contém 1 de 4 valores – A, B, C ou D }
```

que define `peso-TipoDoPapel` como uma variável de string inicializada com o valor A que pode assumir qualquer um dos quatro valores do conjunto {A, B, C, D}.

Se um atributo aparece repetidamente ao longo de uma série de classes de projeto e tem uma estrutura relativamente complexa, é melhor criar uma classe separada para acomodá-lo.

Etapa 3d. Descrever detalhadamente o fluxo de processamento contido em cada operação. Isso poderia ser concretizado usando-se um pseudocódigo baseado em linguagem de programação ou por meio de um diagrama de atividades UML. Cada componente de software é elaborado por meio de uma série de iterações que aplicam o conceito de refinamento gradual (Capítulo 12).

A primeira iteração define cada operação como parte da classe de projeto. Em cada caso, a operação deve ser caracterizada de modo a garantir alta coesão; a operação deve realizar uma única função ou subfunção determinada. A iteração seguinte faz pouco mais do que expandir o nome da operação. Por exemplo, a operação `calcularCustoPapel()` indicada na Figura 14.1 pode ser expandida da seguinte maneira:

```
calcularCustoPapel (peso, tamanho, cor): numérico
```

Isso indica que `calcularCustoPapel()` exige os atributos `peso`, `tamanho` e `cor` como entrada e retorna um valor numérico (na verdade um valor monetário) como saída.

Se o algoritmo necessário para implementar `calcularCustoPapel()` for simples e amplamente compreendido, talvez não seja necessário maior elaboração de projeto. O engenheiro de software que realiza a codificação fornece os detalhes necessários para implementar a operação. Entretanto, se o algoritmo for mais complexo ou enigmático, será necessária maior elaboração de projeto nesse estágio. A Figura 14.8 representa um diagrama de atividades UML para `calcularCustoPapel()`. Quando os diagramas de atividades são usados para a especificação de projeto de componentes, em geral são representados em um nível de abstração ligeiramente maior do que o do código-fonte. Uma abordagem alternativa – o uso de pseudocódigo para especificação de projeto – é discutida na Seção 14.5.3.

Etapa 4. Descrever fontes de dados persistentes (bancos de dados e arquivos) e identificar as classes necessárias para gerenciá-los. Os bancos de dados e arquivos normalmente transcendem a descrição de projeto de um componente individual. Na maioria dos casos, esses repositórios de dados persistentes são especificados inicialmente como parte do projeto de arquitetura. Entretanto, à medida que a elaboração do projeto prossegue, é útil fornecer detalhes adicionais sobre a estrutura e a organização das fontes de dados persistentes.

Use elaboração gradual à medida que for refinando o projeto de componentes. Pergunte sempre "Existe uma maneira de simplificar isso e ainda assim obter o mesmo resultado?"

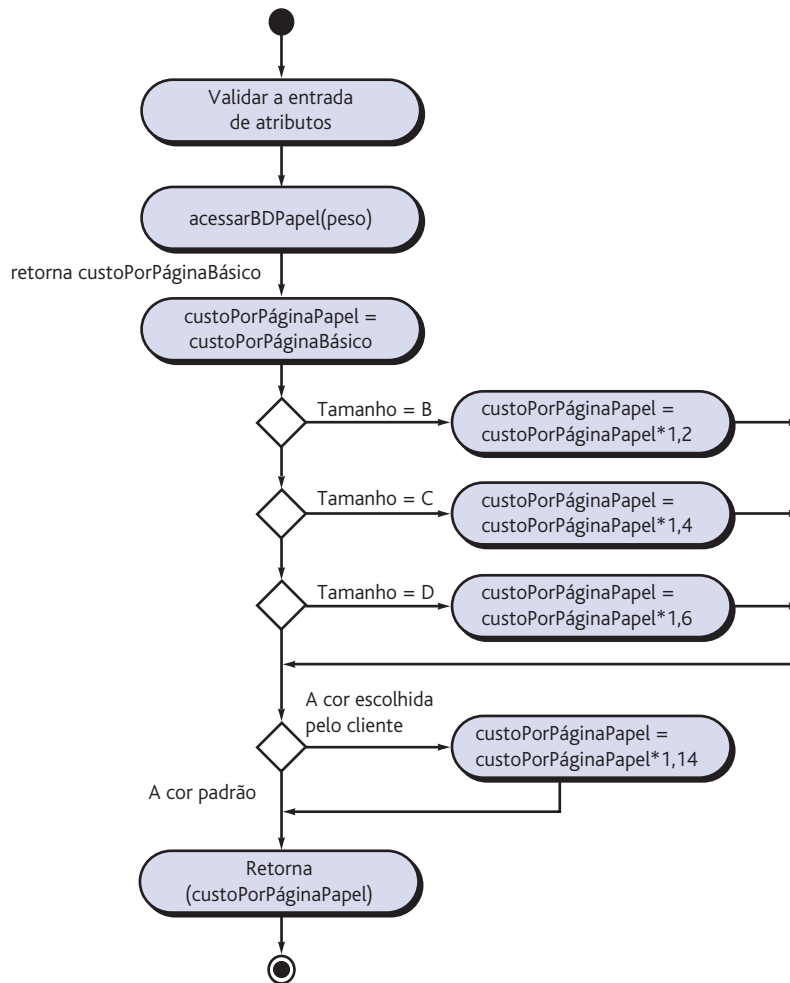


FIGURA 14.8 Diagrama de atividades UML para `calcularCustoPapel()`.

Etapa 5. Desenvolver e elaborar representações comportamentais para uma classe ou componente. Diagramas de estados UML foram usados como parte do modelo de requisitos para representar o comportamento externamente observável do sistema e o comportamento mais localizado de classes de análise individuais. Durante o projeto de componentes, algumas vezes é necessário modelar o comportamento de uma classe de projeto.

O comportamento dinâmico de um objeto (uma instanciación de uma classe de projeto à medida que o programa é executado) é afetado por eventos externos a ele e pelo estado atual (modo de comportamento) do objeto. Para compreender o comportamento dinâmico de um objeto, deve-se examinar todos os casos de uso relevantes para a classe de projeto ao longo de sua vida. Estes fornecem informações que ajudam a delinear os eventos que afetam o objeto e os estados em que o objeto reside à medida que o tempo passa e os eventos ocorrem. As transições entre estados (dirigidos por eventos) são representadas usando-se um diagrama de estado UML [Ben02], conforme ilustrado na Figura 14.9.

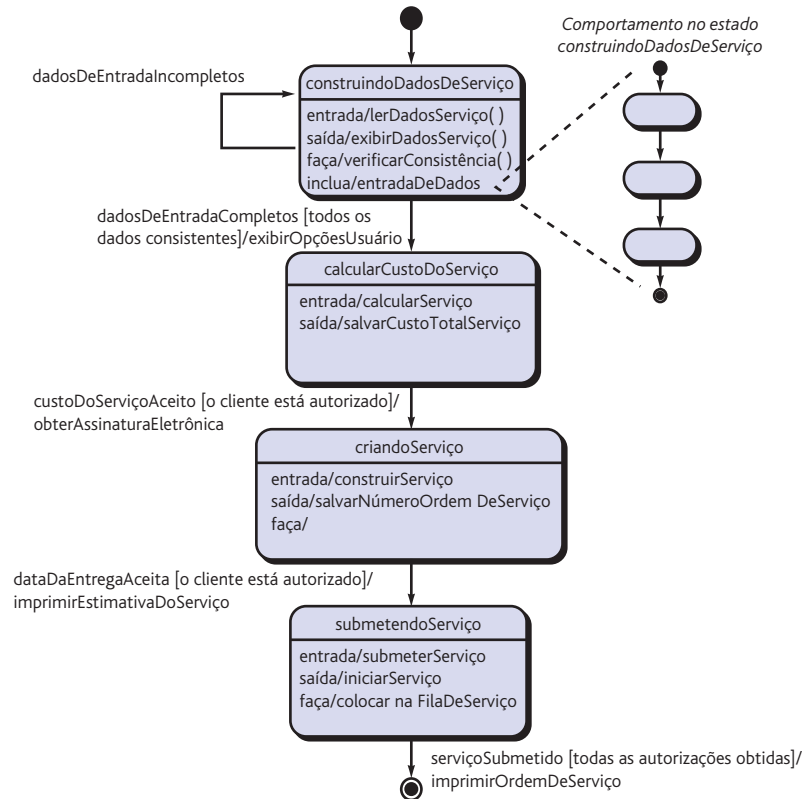


FIGURA 14.9 Fragmento de um diagrama de estados para a classe *ServiçoDeImpressão*.

A transição de um estado (representado por um retângulo com cantos arredondados) para outro ocorre como consequência de um evento que assume a seguinte forma:

nome-evento (lista-parâmetros) [condição-de-guarda] / expressão de ação

onde **nome-evento** identifica o evento, **lista-parâmetros** incorpora dados associados ao evento, **condição-de-guarda** é escrito em OCL e especifica uma condição que deve ser atendida antes de o evento poder ocorrer, e **expressão de ação** define uma ação que ocorre à medida que a transição ocorre.

Com referência à Figura 14.9, cada estado poderia definir ações *entrada/* e *saída/* que acontecem, respectivamente, à medida que ocorre a transição para dentro e para fora do estado. Na maioria dos casos, essas ações correspondem a operações relevantes à classe que está sendo modelada. O indicador *faça/* fornece um mecanismo para indicar atividades que ocorrem enquanto se encontram em determinado estado, ao passo que o indicador *inclua/* fornece um meio para elaborar o comportamento por meio da incorporação de mais detalhes de diagramas de estados na definição de um estado.

É importante notar que o modelo comportamental contém informações que não são óbvias em outros modelos de projeto. Por exemplo, o exame cuidadoso dos diagramas de estado da Figura 14.9 indica que o comportamento dinâmico da classe **ServiçoDeImpressão** é dependente de duas aprovações do cliente, à medida que dados de custo e cronograma para o ser-

viço de impressão são obtidos. Sem as aprovações (a condição de controle garante que o cliente é autorizado para aprovar), o serviço de impressão não pode ser submetido, pois não há nenhuma maneira de atingir o estado *submetendoServiço*.

Etapa 6. Elaborar diagramas de implantação para fornecer detalhes de implementação adicionais. Os diagramas de implantação (Capítulo 12) são usados como parte do projeto da arquitetura e representados na forma de descritores. Dessa forma, funções importantes do sistema (representadas como subsistemas) são representadas no contexto do ambiente computacional que vai abrigá-los.

Durante o projeto de componentes, os diagramas de implantação podem ser elaborados para representar a localização de pacotes de componentes fundamentais. Entretanto, em geral os componentes não são representados individualmente em um diagrama de componentes. A razão para tal é evitar a complexidade dos diagramas. Em alguns casos, os diagramas de implantação são elaborados na forma de instância naquele momento. Isso significa que o(s) ambiente(s) de sistema operacional e de hardware específicos utilizados é(são) especificado(s), e a localização de pacotes de componentes nesse ambiente é indicada.

Etapa 7. Refatorar toda representação de projetos de componentes e sempre considerar alternativas. Ao longo deste livro, enfatizamos que projeto é um processo iterativo. O primeiro modelo no nível de componentes que criamos não será tão completo, consistente ou preciso quanto o da n -ésima iteração aplicada ao modelo. É essencial refatorar à medida que o trabalho de projeto é conduzido.

Além disso, não se deve ter uma visão restrita. Sempre há soluções de projeto alternativas, e os melhores projetistas consideram todas (ou quase todas) elas antes de se decidirem pelo modelo de projeto final. Desenvolva alternativas e considere cuidadosamente cada uma delas, usando os princípios e conceitos apresentados no Capítulo 12 e neste capítulo.

14.4 Projeto de componentes para WebApps

A fronteira entre conteúdo e função normalmente fica indistinta quando se consideram sistemas e aplicações baseadas na Web (WebApps). Consequentemente, podemos perguntar: o que é um componente de WebApp?

No contexto do presente capítulo, um componente de WebApp é (1) uma função coesa bem definida que manipula conteúdo ou fornece processamento computacional ou de dados para um usuário ou (2) um pacote coeso de conteúdo e funcionalidade que fornece ao usuário alguma capacidade exigida. Consequentemente, o projeto de componentes para WebApps em geral incorpora elementos de projeto de conteúdo e de projeto funcional.

14.4.1 Projeto de conteúdo para componentes

O projeto de conteúdo no nível de componentes focaliza os objetos de conteúdo e a maneira como eles podem ser empacotados para apresentação ao

usuário de uma WebApp. A formalidade do projeto de conteúdo para componentes deve ser ajustada às características da WebApp a ser construída. Em muitos casos, os objetos de conteúdo não precisam ser organizados como componentes e podem ser manipulados individualmente. Entretanto, à medida que o tamanho e a complexidade (da WebApp, dos objetos de conteúdo e suas inter-relações) forem crescendo, talvez seja necessário organizar o conteúdo para permitir uma manipulação de projeto e referência mais fácil.⁶ Além disso, se o conteúdo for altamente dinâmico (por exemplo, o conteúdo para um site de leilões online), é importante estabelecer um claro modelo estrutural que incorpore os componentes de conteúdo.

14.4.2 Projeto funcional para componentes

A funcionalidade da WebApp é fornecida por meio de uma série de componentes desenvolvidos em paralelo com a arquitetura das informações para garantir a consistência. Em essência, partimos da consideração do modelo de requisitos, bem como da arquitetura das informações iniciais e, em seguida, examinamos como a funcionalidade afeta a interação do usuário com a aplicação, com as informações apresentadas e com as tarefas realizadas por ele.

Durante o projeto da arquitetura, o conteúdo e a funcionalidade da WebApp são combinados para criar uma arquitetura funcional. *Arquitetura funcional* é uma representação do domínio funcional da WebApp e descreve os componentes funcionais fundamentais da WebApp e como eles interagem entre si.

14.5 Projeto de componentes para aplicativos móveis

No Capítulo 13, mencionamos que os aplicativos móveis normalmente são estruturados com arquiteturas de várias camadas, incluindo uma camada de interface do usuário, uma camada de negócio e uma camada de dados. Caso esteja construindo um aplicativo móvel como um cliente fino baseado na Web, os únicos componentes residentes em um dispositivo móvel são aqueles exigidos para implementar a interface do usuário. Alguns aplicativos móveis podem incorporar os componentes necessários para implementar as camadas de negócio e/ou de dados no dispositivo móvel, sujeitando essas camadas às limitações das características físicas do dispositivo.

Considerando primeiro a camada de interface do usuário, é importante reconhecer que uma área de tela pequena exige que o projetista seja mais seletivo na escolha do conteúdo (texto e elementos gráficos) a ser exibido. Pode ser útil personalizar o conteúdo para um (ou mais) grupo de usuários específico e exibir somente o que cada grupo precisa. As camadas de negócio e de dados frequentemente são implementadas pela composição de componentes de serviços web ou da nuvem. Se os componentes que fornecem os serviços do negócio e de dados residirem totalmente no dispositivo móvel, os problemas de conectividade não serão uma preocupação significativa. Ao se projetar

⁶ Os componentes de conteúdo também podem ser reutilizados em outras WebApps.

componentes que exigem acesso aos dados do aplicativo atual que residem em um servidor da rede, deve-se considerar a conectividade intermitente (ou ausente) com a Internet.

Se um aplicativo de desktop está sendo portado para um dispositivo móvel, os componentes da camada de negócio talvez precisem ser revistos para saber se atendem aos requisitos não funcionais (por exemplo, segurança, desempenho, acessibilidade) exigidos pela nova plataforma. O dispositivo móvel de destino pode não ter a velocidade de processador, memória ou área útil na tela necessária. O projeto de aplicativos móveis é discutido com mais detalhes no Capítulo 18.

14.6 Projeto de componentes tradicionais

Os fundamentos do projeto de componentes para componentes de software⁷ tradicionais foram formados no início dos anos 1960 e solidificados com o trabalho de Edsger Dijkstra ([Dij65], [Dij76b]) e outros (por exemplo, [Boh66]). No final dos anos 1960, Dijkstra e outros propuseram o uso de um conjunto de construções lógicas restritas a partir das quais qualquer programa poderia ser formado. As construções enfatizavam “a manutenção do domínio funcional”. Ou seja, cada construção possuía uma estrutura lógica previsível e entrava-se nela pela parte superior e saía-se pela inferior, possibilitando a um leitor seguir mais facilmente o fluxo procedural.

As construções são sequência, condição e repetição. A *sequência* implementa etapas de processamento essenciais na especificação de qualquer algoritmo. A *condição* fornece a facilidade para processamento seletivo baseado em alguma ocorrência lógica, e a *repetição* possibilita o loop. Essas três construções são fundamentais para a *programação estruturada* – uma importante técnica para projetos de componentes.

As construções estruturadas foram propostas para limitar o projeto procedural de software a um pequeno número de estruturas lógicas previsíveis. As métricas de complexidade (Capítulo 30) indicam que o uso de construções estruturadas reduz a complexidade dos programas e, conseqüentemente, facilita a legibilidade, a realização de testes e a manutenção. O uso de um número limitado de construções lógicas também contribui para o processo de compreensão humana chamado pelos psicólogos de *agrupamento*. Para compreender esse processo, considere a maneira como você está lendo esta página. Não lemos as letras individualmente, mas reconhecemos padrões ou grupos de letras que formam as palavras ou frases. As construções estruturadas são grupos lógicos que permitem a um leitor reconhecer elementos procedurais de um módulo, em vez de ter de ler o projeto ou código linha por linha. A compreensão aumenta quando são encontrados padrões lógicos reconhecíveis.

Programação estruturada é uma técnica de projeto que restringe o fluxo lógico a três construções: sequência, condição e repetição.

⁷ Um componente de software tradicional implementa um elemento de processamento que trata uma função ou subfunção no domínio do problema ou alguma capacidade no domínio da infraestrutura. Em geral denominados módulos, procedimentos ou sub-rotinas, os componentes tradicionais não encapsulam dados da mesma forma que os componentes orientados a objetos.

Qualquer programa, independentemente da área de aplicação ou de sua complexidade técnica, pode ser projetado e implementado usando-se apenas as três construções estruturadas. Entretanto, deve-se frisar que o uso dogmático apenas dessas construções pode, algumas vezes, provocar dificuldades práticas.

14.7 Desenvolvimento baseado em componentes

No contexto da engenharia de software, a reutilização é uma ideia ao mesmo tempo antiga e nova. Os programadores têm reutilizado ideias, abstrações e processos desde os primórdios da computação, mas a abordagem inicial para a reutilização era improvisada. Hoje em dia, sistemas computacionais complexos e de alta qualidade devem ser construídos em prazos muito curtos e exigem uma abordagem mais organizada para a reutilização.

A engenharia de software baseada em componentes (CBSE) é um processo que enfatiza o projeto e a construção de sistemas baseados em computadores usando “componentes” de software reutilizáveis. Considerando essa descrição, surge uma série de questões. É possível construir sistemas complexos montando-os por meio de um catálogo de componentes de software reutilizáveis? Isso pode ser realizado de maneira eficaz em termos de custo e tempo? Podem ser estabelecidos incentivos apropriados para estimular os engenheiros de software à reutilização em vez de reinventar? A gerência está disposta a incorrer na despesa adicional associada à criação de componentes de software reutilizáveis? A biblioteca de componentes necessários para a reutilização pode ser criada para torná-la acessível aos que precisam dela? Os componentes existentes podem ser encontrados por aqueles que precisam deles? Cada vez mais, a resposta a cada uma dessas questões é “sim”.

14.7.1 Engenharia de domínio

O objetivo da engenharia de domínio é identificar, construir, catalogar e disseminar um conjunto de componentes de software que tenham aplicabilidade em software existente e futuro em determinado domínio de aplicação.⁸ O objetivo geral é estabelecer mecanismos que permitam aos engenheiros de software compartilhar esses componentes – para reutilizá-los – durante o trabalho em sistemas novos e existentes. A engenharia de domínio abrange três atividades principais – análise, construção e disseminação.

A abordagem geral para a *análise de domínio* é frequentemente caracterizada no contexto da engenharia de software orientada a objetos. As etapas do processo são: (1) definir o domínio a ser investigado, (2) classificar os itens extraídos do domínio, (3) coletar uma amostra representativa das aplicações do domínio, (4) analisar cada aplicação na amostra e definir classes de análise e (5) desenvolver um modelo de requisitos para as classes. É importante notar que a análise de domínio aplica-se a qualquer paradigma de engenharia de

“Engenharia de domínio significa encontrar características comuns entre os sistemas para identificar componentes que possam ser aplicados a vários sistemas e identificar famílias de programas que estejam em posição de tirar total proveito desses componentes.”

Paul Clements

⁸ No Capítulo 13, referimo-nos a gêneros de arquitetura que identificam domínios de aplicação específicos.

software e poderia ser aplicada tanto em desenvolvimento convencional quanto orientado a objetos.

14.7.2 Qualificação, adaptação e composição de componentes

A engenharia de domínio fornece a biblioteca de componentes reutilizáveis necessários para a CBSE. Alguns desses componentes reutilizáveis são desenvolvidos internamente, outros podem ser extraídos de aplicações existentes e outros ainda podem ser adquiridos de terceiros.

Infelizmente, a existência de componentes reutilizáveis não garante que possam ser integrados de forma fácil e eficaz à arquitetura escolhida para uma nova aplicação. É por essa razão que uma sequência de atividades de desenvolvimento baseado em componentes é aplicada quando um componente é proposto para uso.

Qualificação de componentes. A qualificação de componentes garante que um componente candidato vai realizar a função necessária, que ele “se encaixará” adequadamente no estilo de arquitetura (Capítulo 13) especificada para o sistema e apresentará as características de qualidade (por exemplo, desempenho, confiabilidade, usabilidade) exigidas para a aplicação.

Projeto por contrato é uma técnica que se concentra na definição de especificações de interface de componentes claras e verificáveis, permitindo, assim, que usuários em potencial do componente entendam sua finalidade rapidamente. Afirmações, conhecidas como *precondições*, *pós-condições* e *invariantes*, são adicionadas à especificação do componente.⁹ As afirmações permitem aos desenvolvedores saber o que esperar do componente e como ele se comporta sob certas condições. As afirmações tornam fácil para os desenvolvedores identificar componentes qualificados e, como consequência, estarem mais propensos a confiar no uso do componente em seus projetos. O projeto por contrato é aprimorado quando os componentes têm uma “interface econômica”; ou seja, a interface de componentes tem um pequeno conjunto de operações necessárias para permitir que ela cumpra suas responsabilidades (contrato).

Uma especificação da interface fornece informações úteis sobre a operação e o uso de um componente de software; porém, não fornece todas as informações necessárias para determinar se um componente proposto pode, de fato, ser reutilizado efetivamente em uma nova aplicação. Entre os muitos fatores considerados durante a qualificação de componentes, temos [Bro96]:

- Interface de programas aplicativos (*application programming interface*, API).
- Ferramentas de desenvolvimento e integração exigidas pelo componente.
- Requisitos de tempo de execução, incluindo o emprego de recursos (por exemplo, memória ou armazenamento), *timing* ou velocidade e protocolo de redes.

O processo de análise discutido nesta seção concentra-se em componentes reutilizáveis. Entretanto, a análise de sistemas comerciais completos (por exemplo, aplicações de comércio eletrônico, aplicações para automação da equipe de vendas) também pode fazer parte da análise de domínio.

Quais fatores são considerados durante a qualificação de componentes?

⁹ *Precondições* são declarações sobre suposições que devem ser verificadas antes de se usar um componente, *pós-condições* são declarações sobre serviços garantidos a serem fornecidos por um componente e *invariantes* são declarações sobre atributos do sistema que não serão alterados pelos componentes. Esses conceitos serão discutidos no Capítulo 28.

- Requisitos de serviços, incluindo interfaces para sistemas operacionais e suporte de outros componentes.
- Características de segurança, incluindo controles de acesso e protocolo de autenticação.
- Pressupostos implícitos de projetos, incluindo o uso de algoritmos numéricos e não numéricos.
- Tratamento de exceções.

Cada um dos fatores é relativamente fácil de avaliar quando são propostos componentes reutilizáveis desenvolvidos internamente. Se forem aplicadas práticas adequadas de engenharia de software durante o desenvolvimento de um componente, poderão ser dadas respostas àquelas questões implícitas na lista. Entretanto, é muito mais difícil determinar o funcionamento interno de componentes comerciais de prateleira (*commercial off-the-shelf*, COTS) ou de terceiros, pois a única informação disponível pode ser a especificação da própria interface.

Adaptação de componentes. Em condições ideais, a engenharia de domínio cria uma biblioteca de componentes que pode ser facilmente integrada à arquitetura de uma aplicação. Para termos a “facilidade de integração”, subentende-se que: métodos consistentes de gerenciamento de recursos tenham sido implementados para todos os componentes da biblioteca, atividades comuns, como gerenciamento de dados, existam para todos os componentes, e as interfaces internas da arquitetura e com ambiente externo tenham sido implementadas de maneira consistente.

Na realidade, mesmo após qualificar um componente para uso na arquitetura de uma aplicação, podem ocorrer conflitos em uma ou mais das áreas citadas. Para evitá-los, algumas vezes é usada uma técnica de adaptação chamada *empacotamento de componentes* [Bro96]. Quando uma equipe de software tem acesso completo ao projeto e ao código internos de um componente (o que normalmente não ocorre, a menos que sejam usados componentes comerciais de prateleira com código-fonte aberto), é aplicado um *empacotamento caixa branca*. Da mesma forma que seu equivalente em testes de software (Capítulo 23), o empacotamento caixa branca examina os detalhes de processamento interno do componente e realiza modificações no código para eliminar qualquer conflito. O *empacotamento caixa cinza* é aplicado quando a biblioteca de componentes oferece uma linguagem de extensão de componentes ou uma API que possibilite a eliminação ou o mascaramento de conflitos. O *empacotamento caixa preta* exige a introdução de pré e pós-processamento na interface de componentes para eliminar ou mascarar conflitos. Temos de determinar se o esforço exigido para empacotar um componente adequadamente se justifica ou se vale mais a pena criar um componente personalizado (projetado para eliminar os conflitos encontrados).

Composição de componentes. A tarefa de composição de componentes monta componentes qualificados, adaptados e construídos para compor a arquitetura estabelecida para uma aplicação. Para tal, deve ser estabelecida uma infraestrutura para vincular os componentes a um sistema operacional. A infraestrutura (em geral uma biblioteca de componentes especializados) for-

Além de avaliarmos se o custo da adaptação para a reutilização se justifica, também devemos avaliar se a obtenção da funcionalidade e desempenho exigidos pode ser feita de modo eficaz em termos de custo.

nece um modelo para a coordenação de componentes e serviços específicos que habilitam os componentes a se coordenarem entre si e realizarem tarefas comuns.

Devido ao potencial de ser enorme o impacto da reutilização e de CBSE no setor de software, várias companhias importantes e consórcios da indústria propuseram padrões para software de componentes.¹⁰ Esses padrões incluem: CCM (Corba Component Model),¹¹ Microsoft COM e .NET,¹² JavaBeans¹³ e OSGI (Open Services Gateway Initiative IOSG13).¹⁴ Nenhum desses padrões domina o mercado. Embora muitos desenvolvedores tenham adotado algum deles como padrão, é provável que grandes empresas de software optem por adotar um padrão baseado nas plataformas e categorias de aplicação escolhidas.

14.7.3 Divergência arquitetural

Um dos desafios enfrentados pela reutilização ampla é a *divergência arquitetural* [Gar09a]. Os projetistas de componentes reutilizáveis frequentemente fazem suposições implícitas sobre o ambiente em que o componente está acoplado. Muitas vezes, essas suposições se concentram no modelo de controle do componente, na natureza das conexões do componente (interfaces), na própria infraestrutura arquitetural e na natureza do processo de construção. Se essas suposições estiverem incorretas, ocorrerá uma divergência arquitetural.

Conceitos de projeto, como abstração, encapsulamento, independência funcional, refinamento e programação estruturada, juntamente com métodos, testes e garantia da qualidade de software (*software quality assurance*, SQA) orientados a objetos, bem como métodos para verificação da correção (Capítulo 28), contribuem para a criação de componentes de software reutilizáveis e impedem a divergência arquitetural.

Se as suposições dos envolvidos forem documentadas explicitamente, poderá ocorrer a detecção antecipada da divergência arquitetural. Além disso, o uso de um modelo de processo orientado a riscos enfatiza a definição de protótipos arquiteturais antecipados e aponta para áreas de divergência. Frequentemente, é muito difícil reparar uma divergência arquitetural sem o uso de mecanismos como empacotadores ou adaptadores.¹⁵ Às vezes é necessário reformar completamente uma interface de componentes ou o próprio componente para eliminar problemas de acoplamento.

¹⁰ Greg Olsen [Ols06] apresenta uma excelente discussão sobre empreendimentos do setor no passado e no presente para tornar a CBSE uma realidade. Ivica Crnkovic [Crb11] apresenta uma discussão sobre os modelos de componentes industriais mais recentes.

¹¹ Mais informações sobre CCM podem ser encontradas em: www.omg.org.

¹² Informações sobre COM e .NET podem ser encontradas em: [www.microsoft.com/COM e msdn2.microsoft.com/en-us/netframework/default.aspx](http://www.microsoft.com/COM/msdn2.microsoft.com/en-us/netframework/default.aspx).

¹³ As informações mais recentes sobre Javabeans podem ser encontradas em: java.sun.com/products/javabeans/docs/.

¹⁴ Informações sobre OSGI podem ser encontradas em: <http://www.osgi.org/Main/HomePage>.

¹⁵ Um *adaptador* é um dispositivo de software que permite a um cliente com uma interface incompatível acessar um componente, transformando uma solicitação de serviço em uma forma que possa acessar a interface original.

14.7.4 Análise e projeto para reutilização

Elementos do modelo de requisitos (Capítulos 9 a 11) são comparados com as descrições de componentes reutilizáveis em um processo algumas vezes conhecido como “correspondência da especificação” [Bel95]. Se a correspondência da especificação apontar para um componente existente que atenda às necessidades da aplicação atual, poderemos extrair o componente de uma biblioteca (repositório) de reutilização e usá-lo no projeto de um novo sistema. Se não puderem ser encontrados componentes (se não existir nenhuma correspondência), é criado um novo componente. É nesse ponto – quando se inicia a criação de um novo componente – que o projeto visando a reutilização (*design for reuse*, DFR) deve ser considerado.

O DFR pode ser bem difícil quando componentes tiverem de fazer interface ou ser integrados a sistemas legados ou com vários sistemas cuja arquitetura e protocolos de interface são inconsistentes.

Conforme já dito, o DFR exige a aplicação de conceitos e princípios de projeto de software sólidos (Capítulo 12). Porém, as características do domínio de aplicação também devem ser consideradas. Binder [Bin93] sugere uma série de questões-chave¹⁶ que formam a base para o projeto visando a reutilização. Se o domínio de aplicação tem estruturas de dados globais padrão, o componente deve ser projetado utilizando essas estruturas. Devem ser adotados protocolos de interface padrão dentro de um domínio de aplicação, e um estilo de arquitetura (Capítulo 13) adequado para o domínio pode servir como modelo para o projeto arquitetural do novo software. Uma vez estabelecidos os modelos de programas, de interfaces e de dados, temos uma estrutura na qual criar o projeto. Componentes novos que se adaptem a essa estrutura têm maior probabilidade de reutilização posterior.

14.7.5 Classificação e recuperação de componentes

Consideremos um grande repositório de componentes. Dezenas de milhares de componentes de software reutilizáveis residem nele. Mas como encontrar aquele de que precisamos? Para responder a essa pergunta, surge uma segunda: Como descrever componentes de software em termos inequívocos e classificáveis? Essas são perguntas difíceis, e ainda não se chegou a uma resposta definitiva.

Um componente de software reutilizável pode ser descrito de várias maneiras; porém, uma descrição ideal engloba aquilo que Tracz [Tra95] denominou *modelo 3C* – conceito, conteúdo e contexto –, uma descrição daquilo que o componente faz, como isso é obtido com conteúdo que pode ficar oculto para usuários eventuais e que precisa ser conhecido apenas por aqueles que pretendem modificar ou testar o componente e onde o componente reside em seu domínio de aplicabilidade.

Para serem úteis em um ambiente prático, conceito, conteúdo e contexto devem ser traduzidos em um esquema de especificação concreto. Dezenas de trabalhos e artigos foram escritos tratando dos esquemas de classificação para componentes de software reutilizáveis (por exemplo, [Nir10], [Cec06]), e todos devem ser implementados em um ambiente de reutilização que apresente as seguintes características:

¹⁶ Em geral, as preparações DFR devem ser realizadas como parte da engenharia de domínios.

- Um banco de dados de componentes capaz de armazenar componentes de software e as informações de classificação necessárias para recuperá-los.
- Um sistema de gerenciamento de bibliotecas que dê acesso ao banco de dados.
- Um sistema de recuperação de componentes de software (por exemplo, um agente de solicitação de objetos) que permita a uma aplicação-cliente recuperar componentes e serviços do servidor de bibliotecas.
- As ferramentas CBSE que dão suporte à integração de componentes reutilizados em um novo projeto ou implementação.

Quais são as principais características de um ambiente de reutilização de componentes?

Cada uma dessas funções interage ou é incorporada a uma biblioteca de reutilização, um elemento de um repositório de software maior (Capítulo 29) que oferece recursos para o armazenamento de componentes de software e uma ampla variedade de artefatos reutilizáveis (por exemplo, especificações, projetos, padrões, frameworks, trechos de código, casos de teste, guias de usuário).

FERRAMENTAS DO SOFTWARE



CBSE

Objetivo: Apoiar a modelagem, projeto, revisão e integração de componentes de software como parte de um sistema mais amplo.

Mecanismos: a mecânica das ferramentas varia. Em geral, as ferramentas CBSE auxiliam em uma ou mais das seguintes capacidades: especificação e modelagem da arquitetura de software; navegação e seleção de componentes de software disponíveis; e integração de componentes.

Ferramentas representativas¹⁷

Component Source (www.componentsource.com) oferece uma ampla variedade de componentes (e ferramentas) de software comerciais, suportada em diversos padrões de componentes.

Component Manager, desenvolvida pela Flashline (<http://www.softlookup.com/download.asp?id=8204>), "é uma aplicação que permite, promove e mede a reutilização de componentes de software".

Select Component Factory, desenvolvida pela Select Business Solutions (www.selectbs.com), "é um conjunto integrado de produtos para projeto de software, revisão de software, gerenciamento de serviços/componentes, gerenciamento de requisitos e geração de código".

Software Through Pictures-ACD, distribuída pela Aonix (www.aonix.com), permite modelagem completa, empregando UML para a arquitetura dirigida a modelos OMG – uma abordagem aberta e desvinculada de qualquer fornecedor específico para CBSE.

14.8 Resumo

O processo para projeto de componentes abrange uma sequência de atividades que reduz lentamente o nível de abstração com o qual um software é representado. Em última análise, o projeto de componentes representa o software em um nível de abstração próximo do código.

Podem ser adotadas três visões diferentes de projeto de componentes, dependendo da natureza do software a ser desenvolvido. A visão orientada a

¹⁷ A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

objetos se concentra na elaboração de classes de projeto provenientes tanto do domínio do problema quanto de infraestrutura. A visão tradicional refina três tipos diferentes de componentes ou módulos: módulos de controle, módulos do domínio do problema e os módulos de infraestrutura. Em ambos os casos, são aplicados conceitos e princípios básicos de projeto que levam a um software de alta qualidade. Quando considerado do ponto de vista de processos, o projeto de componentes faz uso de componentes de software reutilizáveis e padrões de projeto que são elementos fundamentais da engenharia de software baseada em componentes.

Uma série de importantes princípios e conceitos orienta o projetista à medida que as classes são elaboradas. Ideias englobadas pelo princípio do aberto-fechado e da inversão da dependência, além de conceitos como acoplamento e coesão, orientam o engenheiro de software na construção de componentes de software que podem ser testados, implementados e mantidos. Para conduzir projetos de componentes nesse contexto, são elaboradas classes por meio da especificação de detalhes de mensagens, identificação de interfaces apropriadas, elaboração de atributos e definição de estruturas de dados para implementá-las, descrevendo o fluxo de processamento em cada operação e representando o comportamento em termos de componentes ou classes. Em todos os casos, a iteração (refatoração) do projeto é uma atividade essencial.

O projeto de componentes tradicional exige a representação de estruturas de dados, interfaces e algoritmos para um módulo de programa em detalhes suficientes para nos orientar na geração de código-fonte em uma linguagem de programação. Para tanto, o projetista usa uma série de notações de projeto que representam detalhes no nível de componentes, em formato gráfico, tabular ou com base em texto.

O projeto de componentes para WebApps considera tanto o conteúdo quanto a funcionalidade, já que é fornecido por um sistema baseado na Web. O projeto de conteúdo no nível de componentes se concentra nos objetos de conteúdo e a maneira pela qual podem ser empacotados para apresentação ao usuário de uma WebApp. O projeto funcional para WebApps concentra-se nas funções de processamento que manipulam conteúdo, efetuam cálculos, consultas e acesso a um banco de dados, bem como estabelecem interfaces com outros sistemas. Todos os princípios e diretrizes de projeto de componentes se aplicam.

O projeto de componentes para aplicativos móveis utiliza uma arquitetura de várias camadas que inclui uma camada de interface do usuário, uma camada de negócio e uma camada de dados. Se o aplicativo móvel exige o projeto de componentes que implementam as camadas de negócio e/ou de dados no dispositivo móvel, as limitações das características físicas do dispositivo se tornam importantes restrições de projeto.

Programação estruturada é uma filosofia de projeto procedural que restringe o número e o tipo de construções lógicas usadas para representar detalhes algorítmicos. O intuito da programação estruturada é auxiliar o projetista na definição de algoritmos que sejam menos complexos e, conseqüentemente, mais fáceis de ser lidos, testados e mantidos.

A engenharia de software baseada em componentes identifica, constrói, cataloga e dissemina um conjunto de componentes de software em determinado domínio de aplicação. Esses componentes são então qualificados, adaptados e integrados para uso em um novo sistema. Os componentes reutilizáveis devem ser projetados em um ambiente que estabeleça estruturas de dados, protocolos de interface e arquiteturas de programa-padrão para cada domínio de aplicação.

Problemas e pontos a ponderar

14.1. O termo componente é, algumas vezes, difícil de definir. Forneça inicialmente uma definição genérica e, a seguir, definições mais explícitas para software tradicional e orientado a objetos. Por fim, escolha três linguagens de programação que você conheça e ilustre como cada uma define um componente.

14.2. Por que componentes de controle são necessários em software tradicional e em geral não são em software orientado a objetos?

14.3. Descreva o OCP com suas próprias palavras. Por que é importante criar abstrações que sirvam como interface entre componentes?

14.4. Descreva o DIP com suas próprias palavras. O que poderia acontecer se um projetista criasse uma dependência muito grande nas concretizações?

14.5. Selecione três componentes que você tenha desenvolvido recentemente e avalie os tipos de coesão que cada um apresenta. Caso tivesse de definir o principal benefício da coesão elevada, qual seria?

14.6. Selecione três componentes que você tenha desenvolvido recentemente e avalie os tipos de acoplamento que cada um apresenta. Caso tivesse de definir o principal benefício de um baixo acoplamento, qual seria?

14.7. Pode-se dizer que componentes de domínio do problema jamais devem apresentar acoplamento externo? Caso concorde, que tipos de componentes apresentariam acoplamento externo?

14.8. Desenvolva: (1) uma classe de projeto elaborada, (2) descrições de interface, (3) um diagrama de atividades para uma das operações contidas na classe e (4) um diagrama de estados detalhado para uma das classes do *CasaSegura* já discutidas em capítulos anteriores.

14.9. Refinamento gradual e refatoração são a mesma coisa? Em caso negativo, em que diferem?

14.10. O que é um componente de WebApp?

14.11. Selecione um pequeno trecho de um programa existente (aproximadamente 50 a 75 linhas de código-fonte). Isole as construções de programação estruturadas, desenhando retângulos em torno delas no código-fonte. O trecho de programa contém construções que violam a filosofia da programação estruturada? Em caso positivo, reformule o código para que fique de acordo com as construções de programação estruturadas. Em caso negativo, o que você notou em relação aos retângulos que desenhou?

14.12. Todas as linguagens de programação modernas implementam as construções de programação estruturada. Dê exemplos de três linguagens de programação.

14.13. Selecione um pequeno componente codificado e represente-o usando um diagrama de atividades.

14.14. Por que a ideia de “agrupamento” é importante durante o processo de revisão em projetos de componentes?

Leituras e fontes de informação complementares

Foram publicados muitos livros sobre desenvolvimento baseado em componentes e reutilização de componentes nos últimos anos. Szyperski (*Component Software*, 2ª ed., Addison-Wesley, 2011) enfatiza a importância dos componentes de software como blocos construtivos para sistemas eficientes. Hamlet (*Composing Software Components*, Springer, 2010), Curtis (*Modular Web Design*, New Riders, 2009), Apperly e seus colegas (*Service- and Component-Based Development*, Addison-Wesley, 2004), Heineman e Councill (*Component Based Software Engineering*, Addison-Wesley, 2001), Brown (*Large-Scale Component-Based Development*, Prentice Hall, 2000), Allen (*Realizing e-Business with Components*, Addison-Wesley, 2000) e Leavens e Sitaraman (*Foundations of Component-Based Systems*, Cambridge University Press, 2000) abordam muitos aspectos importantes do processo CBSE. Stevens (*UML Components*, Addison-Wesley, 2006), Apperly e seus colegas (*Service- and Component-Based Development*, 2ª ed., Addison-Wesley, 2003), Cheesman e Daniels (*UML Components*, Addison-Wesley, 2000) discutem CBSE com ênfase em UML.

Malik (*Component-Based Software Development*, Lap Lambert Publishing, 2013) apresenta métodos para construir repositórios de componentes eficientes. Gross (*Component-Based Software Testing with UML*, Springer, 2010) e Gao e seus colegas (*Testing and Quality Assurance for Component-Based Software*, Artech House, 2006) discutem testes e questões de SQA para sistemas baseados em componentes.

Nos últimos anos foram publicados dezenas de livros que descrevem os padrões de mercado baseados em componentes. Eles tratam dos detalhes de funcionamento dos padrões em si, mas também consideram diversos tópicos importantes da CBSE.

O trabalho de Linger, Mills e Witt (*Structured Programming – Theory and Practice*, Addison-Wesley, 1979) se mantém como um verdadeiro tratado sobre o assunto. O texto contém uma ótima PDL, bem como discussões detalhadas sobre as ramificações da programação estruturada. Outros livros que focalizam as questões procedurais para sistemas tradicionais seriam os de Farrell (*A Guide to Programming Logic and Design*, Course Technology, 2010), Robertson (*Simple Program Design*, 5ª ed., Course Technology, 2006), Bentley (*Programming Pearls*, 2ª ed., Addison-Wesley, 1999) e Dahl (*Structured Programming*, Academic Press, 1997).

Relativamente poucos livros recentes têm se dedicado ao projeto de componentes. Em geral, livros de linguagens de programação tratam do projeto procedural com certo nível de detalhe, mas sempre no contexto da linguagem apresentada pelo livro. Centenas de títulos desse tipo estão disponíveis.

Uma ampla gama de fontes de informação sobre projeto de componentes se encontra disponível na Internet. Uma lista atualizada de referências relevantes (em inglês) para o projeto de componentes pode ser encontrada no site: www.mhhe.com/pressman.

Projeto de interfaces do usuário

15

Vivemos em um mundo de produtos de alta tecnologia e praticamente todos eles – produtos eletrônicos de consumo, equipamentos industriais, automóveis, sistemas corporativos, sistemas militares, software para PC, aplicativos móveis e WebApps – exigem interação humana. Para que um produto de software seja bem-sucedido, deve apresentar boa *usabilidade* – uma medida qualitativa da facilidade e eficiência com a qual um ser humano consegue empregar as funções e os recursos oferecidos pelo produto de alta tecnologia.

Nas três primeiras décadas da era computacional, a usabilidade não era uma preocupação dominante entre aqueles que construíam software. Em seu clássico livro sobre projeto, Donald Norman [Nor88] argumentou que já era tempo de uma mudança de atitude:

Para criar tecnologia que se adapte ao ser humano, é necessário estudá-lo. Mas hoje temos a tendência de estudar apenas a tecnologia. Como consequência, exige-se que as pessoas se adaptem à tecnologia. É chegada a hora de inverter a tendência, de fazer a tecnologia se adaptar às pessoas.

À medida que os tecnólogos passaram a estudar a interação humana, surgiram duas questões preponderantes. Primeiro, identificou-se um conjunto de *regras de ouro* (discutidas na Seção 15.1). Essas se aplicavam a toda interação humana com produtos tecnológicos. Em segundo lugar, definiu-se um conjunto de *mecanismos de interação* para permitir aos projetistas de software construir sistemas que implementassem as regras de ouro de forma apropriada. Os mecanismos de interação, coletivamente denominados interface gráfica do usuário, eliminaram parte da maioria dos atrozes problemas associados às interfaces humanas. Mas mesmo hoje, todos nós encontramos interfaces do usuário difíceis de assimilar, difíceis de usar, confusas, contrárias à intuição, pouco flexíveis e, em muitos casos, totalmente frustrantes. Mesmo assim, alguns gastam tempo e energia construindo cada uma dessas interfaces e é pouco provável que seus construtores tenham criado os problemas propositalmente.

Conceitos-chave

acessibilidade	336
análise de interfaces	325
análise de tarefas	326
análise de usuários	325
atribuição de nomes	
a comandos	335
avaliação de projeto	342
carga de memória	319
controle	318
elaboração de tarefas	327
interface consistente	321
internacionalização	336
modelos de projeto	
de interface	322
princípios e diretrizes	337
processo	323
projeto de interfaces	332
projeto de interfaces	
para WebApps e	
aplicativos móveis	337
recursos de ajuda	335
regras de ouro	318
tempo de resposta	335
tratamento de erros	335
usabilidade	322

PANORAMA

O que é? O projeto de interfaces do usuário cria um meio de comunicação efetivo entre o ser humano e o computador. Seguindo-se um conjunto de princípios de projeto de interfaces, o projeto identifica objetos e

ações de interface e então cria um layout de tela que forma a base para um protótipo de interface do usuário.

Quem realiza? Um engenheiro de software projeta a interface do usuário por meio da aplicação de um processo iterativo que faz uso de princípios de projeto predefinidos.

Por que é importante? Se um software for difícil de ser utilizado, se ele o compele a incorrer em erros ou frustra seus esforços de atingir suas metas, você não gostará dele, independentemente do poder computacional apresentado, do conteúdo fornecido ou da funcionalidade oferecida. A interface deve ser correta, pois molda a percepção do software por parte do usuário.

Quais são as etapas envolvidas? O projeto de interfaces do usuário se inicia pela identificação do usuário, das tarefas e dos requisitos do ambiente. Uma vez identificadas as tarefas de usuário, são criados e analisados cenários de usuário para definir um conjunto de ações e objetos de interface. Os cenários formam a base para a criação do layout da tela que repre-

senta o projeto gráfico e o posicionamento de ícones, a definição de texto descritivo na tela, a especificação e os títulos de janelas, bem como a especificação de itens de menu principais e secundários. São usadas ferramentas para criar protótipos e, por fim, implementar o modelo de projeto, e o resultado é avaliado em termos de qualidade.

Qual é o artefato? São criados cenários de usuário e gerados layouts de tela. É desenvolvido um protótipo de interface, modificado de forma iterativa.

Como garantir que o trabalho foi realizado corretamente? Os usuários fazem um "test-drive" da interface, e o feedback desse teste é usado para a próxima modificação iterativa do protótipo.

15.1 As regras de ouro

Em seu livro sobre projeto de interfaces, Theo Mandel [Man97] cunha três *regras de ouro*:

1. Deixar o usuário no comando.
2. Reduzir a carga de memória do usuário.
3. Tornar a interface consistente.

Essas regras formam, na verdade, a base para um conjunto de princípios para o projeto de interfaces do usuário que orienta esse importante aspecto do projeto de software.

15.1.1 Deixar o usuário no comando

Durante uma sessão para levantamento de requisitos para um importante e novo sistema de informação, perguntou-se a um usuário-chave sobre os atributos da interface gráfica orientada a janelas.

"O que realmente gostaria", disse o usuário solenemente, "é de um sistema que leia minha mente. Ele saberia o que quero fazer antes mesmo de eu ter de fazê-lo, e isso me facilitaria tremendamente a vida. Isso é tudo, apenas isso".

Sua primeira reação poderia ser sacudir a cabeça e sorrir; porém, reflita por um momento. Não havia absolutamente nada de errado com a solicitação do usuário. Ele queria um sistema que reagisse às suas necessidades e o ajudasse a concretizar suas tarefas. Ele queria controlar o computador, e não que o computador o controlasse.

A maioria das limitações e restrições de interface impostas por um projetista destina-se a simplificar o modo de interação. Mas para quem?

Como projetista, talvez sejamos tentados a introduzir restrições e limitações para simplificar a implementação da interface. O resultado pode ser uma interface fácil de ser construída, mas frustrante sob o ponto de vista do usuário. Mandel [Man97] define uma série de princípios de projeto que permitem a um usuário manter o controle:

"É melhor projetar a experiência do usuário do que retificá-la."

Jon Meads

Defina modos de interação para não forçar o usuário a realizar ações desnecessárias ou indesejadas. Modo de interação é o estado atual da interface. Por exemplo, se for escolhido o comando de *correção ortográfica* no menu de um processador de texto, o software entra no modo de correção ortográfica. Não há nenhuma razão para forçar o usuário a permanecer no modo de revisão ortográfica se ele quer apenas fazer uma pequena edição de texto no meio do caminho. O usuário deve ser capaz de entrar e sair desse modo com pouco ou nenhum esforço.

Proporcione interação flexível. Pelo fato de diferentes usuários terem preferências de interação diversas, deve-se fornecer opções. Por exemplo, um software poderia permitir a um usuário interagir por meio de comandos de teclado, movimentação do mouse, caneta digitalizadora, tela multitoque ou por comandos de reconhecimento de voz. Mas nem toda ação é suscetível a todo mecanismo de interação. Considere, por exemplo, a dificuldade de usar comandos via teclado (ou entrada de voz) para desenhar uma forma complexa.

Possibilite que a interação de usuário possa ser interrompida e desfeita. Mesmo quando envolvido em uma sequência de ações, o usuário deve ser capaz de interromper a sequência para fazer alguma outra coisa (sem perder o trabalho que já havia feito). O usuário também deve ser capaz de “desfazer” qualquer ação.

Simplifique a interação à medida que os níveis de competência avançam e permita que a interação possa ser personalizada. Geralmente os usuários constatarem que realizam a mesma sequência de interações repetidamente. Vale a pena criar um mecanismo de “macros” que permita a um usuário avançado personalizar a interface para facilitar sua interação.

Oculte os detalhes técnicos de funcionamento interno do usuário casual. A interface do usuário deve levá-lo ao mundo virtual da aplicação. O usuário não deve se preocupar com o sistema operacional, as funções de arquivos ou alguma outra tecnologia computacional enigmática.

Projete para interação direta com objetos que aparecem na tela. O usuário tem uma sensação de controle quando lhe é permitido manipular os objetos necessários para realizar uma tarefa de maneira similar ao que ocorreria caso o objeto fosse algo físico. Por exemplo, a interface de uma aplicação que permita a um usuário arrastar um documento para o “lixo” é uma implementação de manipulação direta.

“Sempre desejei que meu computador fosse tão fácil de ser usado quanto meu telefone. Meu desejo tornou-se realidade. Já não sei mais como usar meu telefone.”

Bjarne Stroustrup
(criador do C++)

15.1.2 Reduzir a carga de memória do usuário

Uma interface do usuário bem projetada não sobrecarrega sua memória, pois, quanto mais ele precisar lembrar, mais propensa a erros será a interação. Sempre que possível, o sistema deve “se lembrar” de informações pertinentes e auxiliar o usuário em um cenário de interação que o ajude a recordar-se. Mandel [Man97] define princípios de projeto que possibilitam a uma interface reduzir a carga de memória do usuário:

Reduza a demanda de memória recente. Quando os usuários estão envolvidos em tarefas complexas, a demanda de memória recente pode ser significativa. A interface deve ser projetada para reduzir a exigência de recordar ações, entradas e resultados passados. Isso pode ser obtido pelo fornecimento de pistas visuais que permitam a um usuário reconhecer ações passadas, em vez de ter de se recordar delas.

Estabeleça defaults significativos. O conjunto de parâmetros iniciais (defaults) deve fazer sentido para o usuário comum, porém um usuário também deve ser capaz de especificar suas preferências individuais. Entretanto, deve-se fornecer uma opção “reset” que permita o reestabelecimento dos valores-padrão originais.

Defina atalhos intuitivos. Quando forem usados mnemônicos para realizar alguma função de sistema (por exemplo, alt-P para chamar a função de *impressão*), esse mnemônico deve estar ligado à ação de uma forma que seja fácil de ser memorizada (por exemplo, a primeira letra da tarefa a ser solicitada).

O layout visual da interface deve se basear na metáfora do mundo real. Por exemplo, um sistema de pagamento de contas deve usar uma metáfora de talão de cheques e registro de cheques para orientar o usuário pelo processo de pagamento de uma conta. Isso permite ao usuário se apoiar em indicações visuais bem compreensíveis, em vez de ter de memorizar uma sequência de interações misteriosa.

Revele as informações de maneira progressiva. A interface deve ser organizada hierarquicamente. As informações sobre uma tarefa, um objeto ou algum comportamento devem ser apresentadas inicialmente em um alto nível de abstração. Mais detalhes devem ser apresentados após o usuário demonstrar interesse.

CASASEGURA



Violação de uma regra de ouro de uma interface do usuário

Cena: Sala do Vinod, quando é iniciado o projeto da interface do usuário.

Atores: Vinod e Jamie, membros da equipe de engenharia de software do CasaSegura.

Conversa:

Jamie: Estive pensando sobre a interface da função de vigilância.

Vinod (sorrindo): Pensar faz bem.

Jamie: Creio que talvez possamos simplificar um pouco as coisas.

Vinod: O que você quer dizer?

Jamie: Bem, que tal se eliminássemos totalmente a planta. Ela é atraente, porém vai nos dar muito trabalho de projeto. Em vez disso, poderíamos apenas solicitar ao usuário para que especifique a câmera que deseja ver, e, então, exibir o vídeo em uma janela de vídeo.

Vinod: Como o proprietário vai se lembrar de quantas câmeras estão configuradas e onde se encontram?

Jamie (levemente irritado): Ele é o proprietário do imóvel; ele deve saber essas coisas.

Vinod: Mas e se não souber?

Jamie: Ele deve.

Vinod: Essa não é a questão... e se ele se esquecer?

Jamie: Poderíamos fornecer-lhe uma lista de câmeras em operação e suas posições.

Vinod: Isso é possível, mas por que ele deveria ter de solicitar uma lista?

Jamie: Certo, nós fornecemos a lista se ele pedir ou não.

Vinod: Melhor. Pelo menos não terá de se lembrar de coisas que podemos dar a ele.

Jamie (pensando por um instante): Mas você gosta da planta, não é mesmo?

Vinod: Sim.

Jamie: Qual delas você acha que o pessoal de marketing vai gostar?

Vinod: Tá brincando, não é mesmo?

Jamie: Não.

Vinod: Sei lá... aquela com efeito piscante... Eles adoram características chamativas para o produto... Eles não estão interessados em qual é a mais fácil de ser construída.

Jamie (suspirando): Certo, talvez possamos fazer um protótipo para ambas.

Vinod: Boa ideia... depois deixamos o cliente decidir.

15.1.3 Tornar a interface consistente

A interface deve apresentar e obter informações de forma consistente. Isso implica: (1) todas as informações visuais são organizadas de acordo com regras de projeto mantidas ao longo de todas as exibições de telas, (2) mecanismos de entrada são restritos a um conjunto limitado que é usado de forma consistente por toda a aplicação e (3) mecanismos de navegação para passar de uma tarefa a outra são definidos e implementados de maneira consistente. Mandel [Man97] define um conjunto de princípios de projeto que ajudam a tornar a interface consistente:

Permita ao usuário inserir a tarefa atual em um contexto significativo. Muitas interfaces implementam camadas de interações complexas com dezenas de imagens de tela. É importante fornecer indicadores (por exemplo, títulos para as janelas, ícones gráficos, sistema de cores padronizado) que possibilitem ao usuário saber o contexto do trabalho em mãos. Além disso, o usuário deve ser capaz de determinar de onde ele veio e quais alternativas existem para transição para uma nova tarefa.

Mantenha a consistência em uma linha de produtos completa. Uma família de aplicações (ou seja, linha de produtos) deve implementar as mesmas regras de projeto de modo que a consistência seja mantida para toda a interação.

Se modelos interativos anteriores tiverem criado expectativa nos usuários, não faça alterações a menos que haja uma forte razão para isso. Uma vez que determinada sequência interativa tenha se tornado um padrão de fato (por exemplo, o uso de alt-S para salvar um arquivo), o usuário pressupõe que isso vai ocorrer em qualquer aplicação que vá utilizar. Uma mudança (por exemplo, usar alt-S para chamar uma função de escala) vai causar confusão.

Os princípios de projeto para interfaces discutidos nesta e em seções anteriores dão uma orientação básica. Nas seções seguintes, veremos o processo de projeto de interfaces em si.

“Coisas que parecem diferentes devem agir distintamente. Coisas que parecem iguais devem agir da mesma forma.”

Larry Marine

INFORMAÇÕES

**Usabilidade**

Em um artigo esclarecedor sobre usabilidade, Larry Constantine [Con95] faz uma pergunta que tem relevância significativa sobre o tema: “O que os usuários querem, afinal de contas?”. Ele responde da seguinte maneira:

“O que os usuários realmente querem são boas ferramentas. Todos os sistemas de software, de sistemas operacionais e linguagens a aplicações de entrada de dados e de apoio à decisão, são apenas ferramentas. Os usuários querem das ferramentas que criamos para eles praticamente o mesmo que esperamos das ferramentas que utilizamos. Eles querem sistemas fáceis de aprender e que os ajude a realizar seu trabalho. Querem software que não os retarde, não os engane nem os confunda, que não facilite a prática de erros ou dificulte a finalização de seus trabalhos.”

Constantine argumenta que a usabilidade não é derivada da estética, mecanismos de interação de última geração ou de inteligência incorporada às interfaces. Ao contrário, ela ocorre quando a arquitetura da interface atende às necessidades das pessoas que a usarão.

Uma definição formal de usabilidade é um tanto ilusória. Donahue e seus colegas [Don99] a definem da seguinte maneira: “Usabilidade é uma medida de quanto um sistema computacional... facilita o aprendizado; ajuda os aprendizes a se lembrarem daquilo que aprenderam; reduz a probabilidade de erros; permite que se tornem eficientes; e os deixa satisfeitos com o sistema”.

A única maneira de determinar se existe ou não “usabilidade” em um sistema que estamos construindo é realizar a

avaliação ou teste de usabilidade. Observe os usuários interagirem com o sistema e faça-lhes as seguintes perguntas [Con95]:

- O sistema é utilizável sem necessidade de ajuda ou aprendizado contínuo?
- As regras de interação ajudam um usuário experiente a trabalhar eficientemente?
- Os mecanismos de interação se tornam mais flexíveis à medida que os usuários se tornam mais capacitados?
- O sistema foi ajustado para o ambiente físico e social em que será usado?
- O usuário está ciente do estado do sistema? O usuário sempre sabe onde se encontra?
- A interface é estruturada de maneira lógica e consistente?
- Os mecanismos de interação, ícones e procedimentos são consistentes por toda a interface?
- A interação antecipa erros e ajuda o usuário a corrigi-los?
- A interface é tolerante com erros cometidos?
- A interação é simples?

Se cada uma dessas questões for respondida positivamente, é provável que a usabilidade tenha sido atingida.

Entre os muitos benefícios mensuráveis obtidos de um sistema utilizável, temos [Don99]: aumento nas vendas e na satisfação do cliente, vantagem competitiva, melhores avaliações por parte da mídia, melhor recomendação boca a boca, menores custos de suporte, aumento da produtividade do usuário, redução nos custos de treinamento e de documentação e menor probabilidade de litígio com clientes descontentes.

15.2 Análise e projeto de interfaces

Uma excelente fonte de informações sobre projeto de interfaces do usuário pode ser encontrada em www.nngroup.com.

O processo geral para análise e projeto de uma interface do usuário se inicia com a criação de diferentes modelos de funções do sistema (segundo uma percepção do mundo externo). Começamos pelo delineamento das tarefas orientadas à interação homem-máquina necessárias para alcançar a função do sistema e, em seguida, consideramos as questões de projeto que se aplicam a todos os projetos de interface. São usadas ferramentas para prototipação e, por fim, a implementação do modelo de projeto para o resultado ser avaliado pelos usuários em termos de qualidade.

15.2.1 Modelos de análise e projeto de interfaces

Quatro modelos distintos entram em cena ao se analisar e projetar uma interface do usuário. Um engenheiro humano (ou o engenheiro de software) estabelece um *modelo de usuário*, o engenheiro de software cria um *modelo de projeto*, o usuário final desenvolve uma imagem mental em geral chamada

modelo mental do usuário ou *percepção do sistema*, e os implementadores do sistema criam um *modelo de implementação*. Infelizmente, cada um dos modelos pode diferir significativamente. O papel de um projetista de interfaces é harmonizar as diferenças e obter uma representação consistente da interface.

O modelo de usuário estabelece o perfil dos usuários do sistema. Em sua coluna introdutória sobre “projeto centralizado no usuário”, Jeff Patton [Pat07] observa o seguinte:

A verdade é que projetistas e desenvolvedores – até eu mesmo – em geral pensam nos usuários. Entretanto, na ausência de um modelo mental consistente de usuários específicos, colocamo-nos no lugar desses usuários. A autossubstituição não é centralizada no usuário – é egocêntrica.

Para construir uma interface do usuário efetiva, “todo projeto deve começar com um entendimento dos usuários pretendidos, incluindo seus perfis de idade, gênero, habilidades físicas, educação, formação cultural ou origem étnica, motivação, metas e personalidade” [Shn04]. Além disso, os usuários podem ser classificados como novatos, entendidos, usuários intermitentes e com conhecimento ou usuários frequentes e com conhecimento.

O *modelo mental* (percepção do sistema) do usuário é a imagem do sistema que os usuários trazem em suas mentes. Por exemplo, se fosse solicitado ao usuário de um aplicativo móvel que classifica restaurantes para que descrevesse sua operação, a percepção do sistema orientaria a resposta. A precisão da descrição vai depender do perfil do usuário (por exemplo, novatos dariam no máximo uma resposta muito superficial) e da familiaridade geral com o software no domínio de aplicação. Um usuário que entenda completamente de aplicativos de classificação de restaurantes, mas que trabalhou com um aplicativo específico apenas algumas vezes, talvez seja capaz de dar uma descrição mais completa de sua função do que o novato que investiu dias tentando usar o aplicativo eficientemente.

O *modelo de implementação* combina a manifestação externa do sistema computacional (a aparência e a percepção da interface) com todas as informações de apoio (livros, manuais, fitas de vídeo, arquivos de ajuda) que descrevem a sintaxe e a semântica da interface. Quando o modelo de implementação e o modelo mental do usuário são coincidentes, em geral os usuários sentem-se à vontade com o software e o utilizam de maneira eficaz. Para conseguir tal “amálgama” dos modelos, o modelo de projeto deve ter sido desenvolvido levando em conta as informações contidas no modelo de usuário, e o modelo de implementação deve refletir precisamente as informações sintáticas e semânticas sobre a interface.

“Se há algum ‘truque’, a interface do usuário é fraca.”

Douglas Anderson

Até mesmo um usuário novato quer atalhos; mesmo usuários frequentes e com conhecimento algumas vezes precisam de orientação. Dê a eles aquilo de que precisam.

“Preste atenção naquilo que os usuários fazem e não no que dizem.”

Jakob Nielsen

15.2.2 O processo

O processo de análise e projeto para interfaces do usuário é iterativo e pode ser representado por meio de um modelo espiral semelhante ao discutido no Capítulo 4. De acordo com a Figura 15.1, o processo de análise e projeto de interfaces do usuário começa no interior da espiral e engloba quatro atividades estruturais distintas [Man97]: (1) análise e modelagem de interfaces, (2) projeto de interfaces, (3) construção de interfaces e (4) validação de interfaces. A espiral da Figura 15.1 indica que cada uma dessas tarefas ocorrerá mais de uma vez; cada volta em torno da espiral representa a elaboração adicional dos requisitos

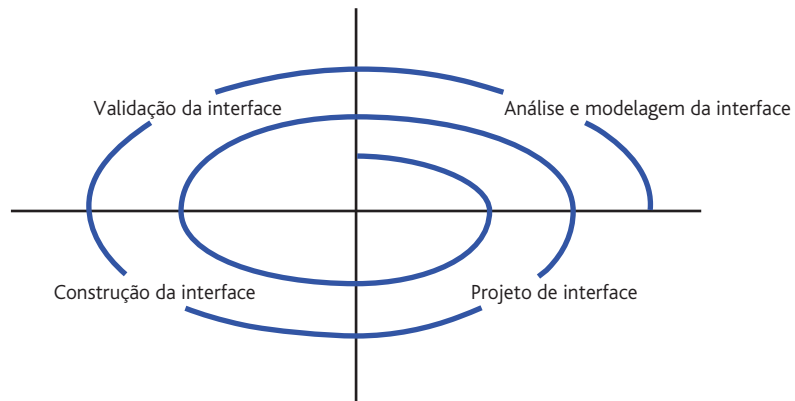


FIGURA 15.1 O processo de projeto de interface do usuário.

e do projeto resultante. Na maioria dos casos, a atividade de construção envolve prototipação – a única maneira prática de validar o que foi projetado.

A *análise de interfaces* se concentra no perfil dos usuários que vão interagir com o sistema. O nível de habilidades, o conhecimento da área e a receptividade geral em relação ao novo sistema são registrados, e categorias de usuários são definidas. Para cada categoria de usuário, é feito o levantamento de requisitos. Em essência, trabalhamos para compreender a percepção do sistema (Seção 15.2.1) para cada classe de usuário.

Uma vez definidos os requisitos gerais, é realizada uma *análise de tarefas* mais detalhada. As tarefas que o usuário realiza para alcançar os objetivos do sistema são identificadas, descritas e elaboradas (ao longo de uma série de passagens iterativas pela espiral). A análise de tarefas é discutida de forma mais detalhada na Seção 15.3. Por fim, a análise do ambiente do usuário concentra-se nas características do ambiente de trabalho físico (por exemplo, local, iluminação, restrições posicionais).

As informações coletadas como parte da ação de análise são usadas para criar um modelo de análise para a interface. Usando esse modelo como base, a atividade de projeto se inicia.

A meta do *projeto da interface* é definir um conjunto de objetos e ações de interface (e suas representações na tela) que permitam a um usuário realizar todas as tarefas definidas para atender a todas as metas de usabilidade estabelecidas para o sistema. O projeto de interfaces é discutido de forma mais detalhada na Seção 15.4.

A *construção da interface* em geral se inicia com a criação de um protótipo que permite a avaliação de cenários de uso. À medida que o processo de projeto iterativo prossegue, um kit de ferramentas de interfaces do usuário (Seção 15.5) pode ser usado para completar a construção da interface.

A *validação da interface* se concentra: (1) na capacidade de a interface implementar corretamente todas as tarefas de usuário, levar em conta todas as variações de tarefas, bem como atender a todos os requisitos gerais dos usuários; (2) no grau de facilidade de uso e aprendizado da interface; e (3) na aceitação do usuário da interface como uma ferramenta útil no seu trabalho.

Conforme já citado, as atividades descritas nesta seção ocorrem iterativamente. Consequentemente, não há necessidade de tentar especificar to-

dos os detalhes (para modelo de análise ou de projeto) na primeira passagem. Passagens subsequentes ao longo do processo elaboram detalhes de tarefas, informações de projeto e características operacionais da interface.

15.3 Análise de interfaces¹

Um princípio fundamental de todos os modelos de processos de engenharia de software é o seguinte: *entender o problema antes de tentar desenvolver uma solução*. No caso do projeto de interfaces do usuário, entender o problema significa entender: (1) as pessoas (usuários) que vão interagir com o sistema por meio da interface, (2) as tarefas que os usuários devem realizar para completar seus trabalhos, (3) o conteúdo que é apresentado como parte da interface e (4) o ambiente onde essas tarefas serão conduzidas. Nas seções a seguir, examinaremos cada um dos elementos da análise de interfaces com o objetivo de estabelecer uma sólida base para as tarefas de projeto que se seguem.

15.3.1 Análise de usuários

A frase *interface do usuário* provavelmente é a única justificativa necessária para despendermos algum tempo para entender o usuário antes de nos preocuparmos com as questões técnicas. Citamos anteriormente que cada usuário tem uma imagem mental do software, a qual pode ser diversa da imagem mental desenvolvida por outros usuários. Além do mais, a imagem mental do usuário pode ser muito diferente do modelo de projeto do engenheiro de software. A única maneira para se fazer com que a imagem mental e o modelo de projeto convirjam é tentar entender os próprios usuários, bem como as pessoas que usam o sistema. Informações de uma ampla variedade de fontes (entrevistas com o usuário, dados de vendas, dados de marketing, dados do suporte) podem ser usadas para concretizar isso.

O conjunto de perguntas a seguir (adaptado de [Hac98]) vai ajudá-lo a entender melhor os usuários de um sistema:

- Os usuários são profissionais treinados, técnicos, do setor administrativo ou pessoal de fábrica?
- Que nível de educação formal o usuário médio possui?
- Os usuários são capazes de aprender por meio de material escrito ou expressaram seu desejo por um treinamento em sala de aula?
- Os usuários são digitadores experientes ou têm fobia a teclados?
- Qual a faixa etária da comunidade de usuários?
- Os usuários serão representados predominantemente por um gênero?
- Como os usuários são recompensados pelo trabalho realizado?
- Os usuários trabalham em expediente normal ou ficam até que o trabalho seja concluído?

Como colocar-se a par da demografia e características dos usuários?

¹ Esta seção poderia estar no Capítulo 8, 9, 10 ou 11, já que questões de análise de requisitos são discutidas lá. Ela foi inserida aqui porque a análise e o projeto de interfaces estão intimamente ligados entre si, e a fronteira entre os dois em geral é imprecisa.

- O software deverá ser parte do trabalho dos usuários ou será usado apenas esporadicamente?
- Qual o principal idioma falado pelos usuários?
- Quais as consequências se um usuário cometer um erro ao usar o sistema?
- Os usuários são especialistas no assunto tratado pelo sistema?
- Os usuários querem saber sobre a tecnologia que se encontra por trás da interface?

Assim que essas perguntas forem respondidas, saberemos quem são os usuários – o que provavelmente vai motivá-los, já que poderão ser agrupados em diferentes classes ou perfis –, quais são seus modelos mentais do sistema e como uma interface deve ser caracterizada para atender a suas necessidades.

15.3.2 Análise e modelagem de tarefas

O objetivo da análise de tarefas é responder às seguintes questões:

- Que trabalho o usuário vai realizar em circunstâncias específicas?
- Quais tarefas e subtarefas serão realizadas à medida que o usuário desenvolve seu trabalho?
- Quais os objetos do domínio de problema específicos que o usuário vai manipular à medida que o trabalho é desenvolvido?
- Qual a sequência de tarefas – o fluxo de trabalho?
- Qual é a hierarquia das tarefas?

Para responder a essas questões, deve-se fazer uso das técnicas discutidas anteriormente neste livro, mas, nesse caso, elas são aplicadas à interface do usuário.

Casos de uso. Em capítulos anteriores vimos que um caso de uso descreve a maneira como um ator (no contexto do projeto de interfaces com o usuário, um ator é sempre uma pessoa) interage com um sistema. Quando usado como parte da análise de tarefas, o caso de uso é desenvolvido para mostrar como um usuário realiza alguma tarefa relacionada a algum trabalho específico. Na maioria das vezes, o caso de uso é redigido em estilo informal (um parágrafo simples) em primeira pessoa. Suponhamos, por exemplo, que uma pequena empresa de software queira construir um sistema de projeto apoiado por computador explicitamente para arquitetos de interiores. Para se ter uma ideia de como eles realizam seus trabalhos, solicita-se a arquitetos de interiores que descrevam uma função de projeto específica. Ao ser indagado: “Como você decide onde colocar o mobiliário em uma sala?”, um arquiteto de interiores escreveu o seguinte caso de uso informal:

Começo esboçando a planta baixa da sala, as dimensões e a localização das janelas e portas. Preocupo-me muito com a iluminação do ambiente, com a vista das janelas (caso seja bonita, quero que sobressaia), com o comprimento total livre de uma parede e com o fluxo de movimento pelo ambiente. Então, examino a lista de móveis escolhidos por meu cliente e por mim... Em seguida, crio um *rendering* (uma figura 3-D) do ambiente para que meu cliente tenha uma noção de como ele ficará.

A meta do usuário é realizar uma ou mais tarefas via interface do usuário. Para tanto, a interface deve fornecer mecanismos que permitam ao usuário atingir sua meta.

Uma excelente fonte de informações sobre modelagem de usuários pode ser encontrada em <http://web.eecs.umich.edu/~kieras/docs/GOMS/>.

CASASEGURA



Casos de uso para o projeto de interfaces do usuário

Cena: Sala do Vinod, enquanto prossegue o projeto de interface do usuário.

Atores: Vinod e Jamie, membros da equipe de engenharia de software do *CasaSegura*.

Conversa:

Jamie: Fiz com que nosso contato de marketing redigisse um caso de uso para a interface de vigilância.

Vinod: Do ponto de vista de quem?

Jamie: Do proprietário do imóvel. Quem mais poderia ser?

Vinod: Há também o papel do administrador do sistema, mesmo que o próprio proprietário esteja desempenhando a função; é um ponto de vista diferente. O administrador ativa o sistema, configura as coisas, faz o layout da planta, posiciona as câmeras...

Jamie: Em suma, desempenha o papel do proprietário quando ele quiser assistir ao vídeo.

Vinod: Tudo bem. Esse é um dos principais comportamentos da interface da função de vigilância. Mas também teremos de examinar o comportamento da administração do sistema.

Jamie (irritado): Você está certo.

[Jamie sai em busca da pessoa de marketing. Ele retorna algumas horas mais tarde.]

Jamie: Tive sorte, encontrei-a e trabalhamos juntos no caso de uso do administrador. Basicamente, vamos definir "administração"

como uma função que se aplica a todas as demais funções do *CasaSegura*. Eis a conclusão a que chegamos.

[Jamie mostra o caso de uso informal a Vinod.]

Caso de uso informal: Quero ser capaz de estabelecer ou editar o layout do sistema a qualquer momento. Ao configurar o sistema, seleciono uma função administrativa. Ela me pergunta se quero fazer uma nova configuração ou editar uma já existente. Caso opte por uma nova configuração, o sistema exibe uma tela de desenho que me permitirá desenhar a planta em uma grade de pontos. Existirão ícones para as paredes, janelas e portas para facilitar o desenho. Tenho simplesmente que "esticar" os ícones até atingirem os comprimentos apropriados. O sistema vai mostrar os comprimentos em pés ou metros (posso escolher o sistema de medidas). Posso escolher de uma biblioteca de sensores e câmeras e colocá-los na planta. Tenho de dar nome a cada um deles ou deixar que o sistema faça isso automaticamente. Posso estabelecer ajustes para os sensores e câmeras por meio de menus apropriados. Caso opte por editar, poderei movimentar os sensores ou as câmeras, acrescentar novas(os) ou eliminar algum(a) existente, editar a planta, bem como fazer os ajustes de configuração para as câmeras e sensores. Em cada um dos casos, espero que o sistema realize testes de consistência e me ajude a não cometer erros.

Vinod (após ler o cenário): Certo, provavelmente existem alguns padrões de projeto úteis [Capítulo 12] ou componentes reutilizáveis para GUIs para programas de desenho. Após 50 pratos que conseguimos implementar parte ou a maior parte da interface de administrador utilizando-os.

Jamie: De acordo. Verificarei isso.

Esse caso de uso fornece uma descrição básica de uma importante tarefa para o sistema de projeto com auxílio de computador. Por meio dele, podemos extrair tarefas, objetos e o fluxo geral da interação. Além disso, também poderiam ser concebidas outras características do sistema que poderiam agradar o arquiteto de interiores. Por exemplo, poderíamos tirar uma foto digital da vista de cada janela do ambiente. Quando o ambiente passar pelo processo de *rendering*, a vista externa real poderia ser representada através de cada janela.

Elaboração de tarefas. No Capítulo 12, discutimos a elaboração gradual (também denominada decomposição funcional ou refinamento gradual) como um mecanismo para refinar as tarefas de processamento necessárias para o software realizar alguma função desejada. A análise de tarefas para projeto de interfaces usa uma abordagem de refinamento que ajuda a entender as atividades humanas a que uma interface do usuário deve atender.

Primeiro, devemos definir e classificar as tarefas humanas exigidas para atingir o objetivo do sistema ou aplicação. Reconsideremos, por exemplo, o sistema de projeto auxiliado por computador para arquitetos de interiores,

discutido anteriormente. Observando o trabalho de um arquiteto de interiores, percebemos que o projeto do interior compreende uma série de atividades principais: layout do mobiliário (note o caso de uso discutido anteriormente), escolha de tecidos e materiais, escolha de papéis de parede e acabamentos para janelas, apresentação (para o cliente), custo e compras. Cada uma das tarefas principais pode ser elaborada em subtarefas. Por exemplo, usando informações contidas no caso de uso, o layout do mobiliário pode ser refinado nas seguintes tarefas: (1) desenhar uma planta nas dimensões do ambiente, (2) colocar as janelas e as portas nos locais apropriados, (3a) usar gabaritos de mobiliário para desenhar contornos de mobiliário em escala na planta, (3b) usar gabaritos de acessórios para desenhar acessórios em escala na planta, (4) movimentar contornos de mobiliário e de acessórios para obter o melhor posicionamento, (5) identificar todos os contornos de mobiliário e acessórios, (6) indicar as dimensões para mostrar as posições e (7) desenhar uma vista em perspectiva para o cliente. Poderia ser usada uma abordagem semelhante para cada uma das principais tarefas.

As subtarefas 1 a 7 podem ser refinadas ainda mais. As subtarefas 1 a 6 serão realizadas por meio da manipulação das informações e da realização de ações em uma interface do usuário. Por outro lado, a subtarefa 7 pode ser realizada automaticamente no software e resultará em pouca interação direta com o usuário.² O modelo de projeto da interface deve considerar cada uma das tarefas de maneira consistente com o modelo de usuário (o perfil de um “típico” arquiteto de interiores) e a percepção do sistema (o que o arquiteto de interiores espera de um sistema automatizado).

Embora a elaboração de objetos seja útil, ela não deve ser usada como um método isolado. A opinião do usuário tem de ser considerada durante a análise de tarefas.

Elaboração de objetos. Em vez de nos concentrarmos nas tarefas que um usuário tem de realizar, podemos examinar o caso de uso e outras informações obtidas do usuário e extrair os objetos físicos utilizados pelo arquiteto de interiores. Tais objetos podem ser classificados em classes. São definidos os atributos de cada classe, e uma avaliação das ações aplicadas a cada objeto fornece uma lista de operações. Por exemplo, o gabarito de mobiliário talvez pudesse ser traduzido em uma classe chamada **Mobiliário**, com atributos que poderiam incluir tamanho, forma, posição e outros. O arquiteto de interiores *selecionaria* o objeto da classe **Mobiliário**, *moveria-o* para uma posição na planta (outro objeto nesse contexto), *desenharia* o contorno da mobília e assim por diante. As tarefas *selecionar*, *mover* e *desenhar* são operações. O modelo de análise da interface do usuário não forneceria uma implementação literal para cada uma dessas operações. Entretanto, à medida que o projeto é elaborado, os detalhes das operações são definidos.

Análise do fluxo de trabalho. Quando uma série de usuários, desempenhando diferentes papéis, faz uso de uma interface do usuário, algumas vezes se faz necessário ir além da análise de tarefas e da elaboração dos objetos e aplicar a *análise do fluxo de trabalho*. Essa técnica permite que entendamos como

² Entretanto, talvez esse não seja o caso. O arquiteto de interiores talvez queira especificar a perspectiva a ser desenhada, a escala, o uso de cores e outras informações. O caso de uso relacionado ao desenho de vistas em perspectiva com efeito de *rendering* forneceria as informações necessárias para lidar com essa tarefa.

um processo de trabalho é completado quando várias pessoas (e papéis) estão envolvidas. Consideremos uma empresa que pretende automatizar completamente o processo de prescrição e entrega de medicamentos. Todo o processo³ vai girar em torno de uma aplicação baseada na Web que pode ser acessada por médicos (ou seus assistentes), farmacêuticos e pacientes. O fluxo de trabalho pode ser representado efetivamente por um diagrama de raias UML (uma variação do diagrama de atividades).

Consideramos apenas uma pequena parte do processo de trabalho: a situação que ocorre quando um paciente solicita uma revalidação da receita. A Figura 15.2 apresenta um diagrama de raias que indica as tarefas e decisões para cada um dos três papéis citados anteriormente. As informações talvez sejam levantadas via entrevista ou por meio de casos de uso escritos por cada ator. Independentemente disso, o fluxo de eventos (mostrado na figura) nos permite reconhecer uma série de características-chave da interface:

1. Cada usuário implementa tarefas distintas via interface; consequentemente, o aspecto da interface projetada para o paciente será diferente do definido para os farmacêuticos ou médicos.
2. O projeto da interface para os farmacêuticos e médicos deve contemplar acesso e exibição de informações de fontes secundárias (por exemplo, acesso ao estoque para o farmacêutico e acesso a informações sobre medicamentos alternativos para o médico).
3. Muitas das atividades citadas no diagrama de raias podem ser elaboradas ainda mais com o uso de análise de tarefas e/ou elaboração de objetos (por exemplo, *Preenche receita* poderia implicar uma entrega via correio, uma ida à farmácia ou a um centro de distribuição de medicamentos especial).

"É muito melhor adaptar a tecnologia ao usuário do que forçá-lo a se adaptar à tecnologia."

Larry Marine

Representação hierárquica. Um processo de elaboração ocorre quando se começa a analisar a interface. Uma vez estabelecido o fluxo de trabalho, pode ser definida uma hierarquia de tarefas para cada tipo de usuário. A hierarquia é obtida por meio de uma elaboração gradual de cada tarefa identificada para o usuário. Consideremos, por exemplo, a tarefa de usuário *Solicita que uma receita seja revalidada*. É desenvolvida a seguinte hierarquia de tarefas:

Solicita que uma receita seja revalidada

- *Fornecer informações de identificação.*
 - *Especificar nome.*
 - *Especificar identificação do usuário.*
 - *Especificar PIN e senha.*
- *Especificar número da receita.*
- *Especificar se é exigida a data de revalidação.*

Para completar a tarefa, são definidas três subtarefas. Uma das subtarefas, *Fornecer informações de identificação*, é elaborada ainda mais em três subsubtarefas adicionais.

³ Esse exemplo foi adaptado de [Hac98].

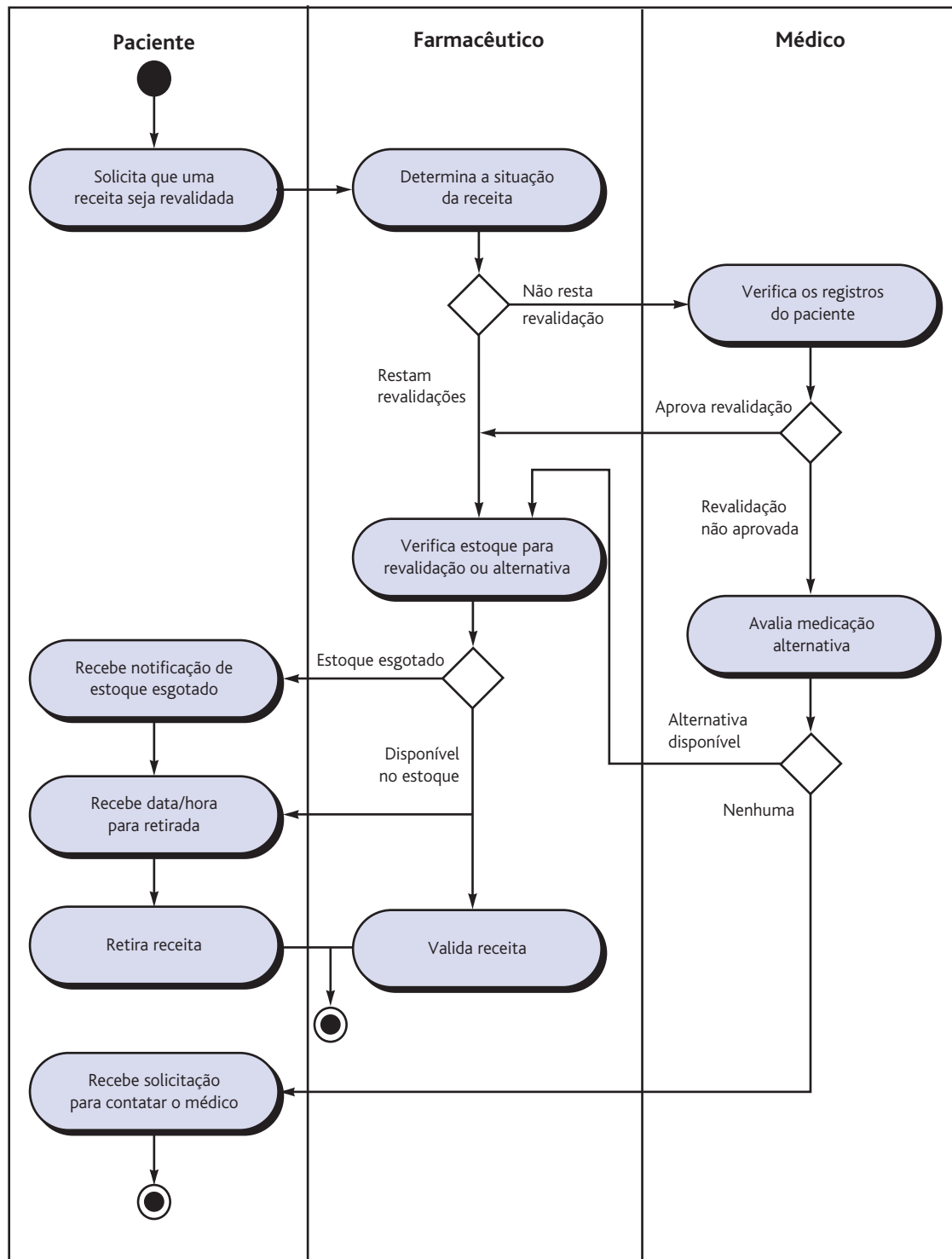


FIGURA 15.2 Diagrama de raias para a função de revalidação de receita.

15.3.3 Análise do conteúdo exibido

As tarefas de usuário identificadas na Seção 15.3.2 levam à apresentação de uma série de tipos diferentes de conteúdo. As técnicas de modelagem de análise discutidas nos Capítulos 9 a 11 identificam os objetos de dados de saída produzidos por uma aplicação. Tais objetos de dados poderiam ser: (1) gerados por componentes (não relacionados à interface) em outras partes de uma aplicação, (2) obtidos de dados armazenados em um banco de dados acessível para a aplicação ou (3) transmitidos de sistemas externos à aplicação em questão.

Durante essa etapa de análise de interface, são considerados o formato e a estética do conteúdo (como ele é exibido pela interface). Entre as perguntas feitas e respondidas, temos:

- Os diferentes tipos de dados são alocados em posições geográficas padronizadas na tela (por exemplo, fotos sempre apareceriam no canto superior direito)?
- O usuário pode personalizar a localização do conteúdo na tela?
- Foi atribuída identificação apropriada na tela para todos os conteúdos?
- Se for preciso apresentar um relatório grande, como seria subdividido para facilitar sua compreensão?
- Haverá mecanismos disponíveis para ir diretamente a informações resumidas em conjuntos de dados volumosos?
- A saída gráfica será apresentada em escala para caber nos limites do dispositivo de exibição utilizado?
- Como serão usadas cores para melhorar o entendimento?
- Como serão apresentadas ao usuário mensagens de erro e alertas?

Como determinar o formato e a estética de conteúdo exibido como parte da interface do usuário?

As respostas a essas (e outras) questões nos ajudarão a estabelecer os requisitos para a apresentação de conteúdo.

15.3.4 Análise do ambiente de trabalho

Hackos e Redish [Hac98] declaram o seguinte sobre a análise do ambiente de trabalho, ao afirmarem: “As pessoas não realizam seus trabalhos de forma isolada. São influenciadas pela atividade em torno delas, pelas características físicas do local de trabalho, pelo tipo de equipamento utilizado e pelas relações de trabalho que têm com outras pessoas”. Em algumas aplicações, a interface do usuário para um sistema baseado em computador é colocada em uma “posição que facilita o usuário” (por exemplo, iluminação apropriada, altura adequada da tela, fácil acesso ao teclado), porém em outras (por exemplo, no chão de fábrica ou no *cockpit* de um avião) talvez a iluminação não seja tão adequada, o ruído pode ser um fator importante, um teclado, mouse ou tela de toque talvez não seja uma opção, o posicionamento da tela talvez seja abaixo do ideal. O projetista de interfaces talvez esteja restrito por fatores que reduzem a facilidade de uso.

Além dos fatores ambientais físicos, a cultura do local de trabalho também entra em cena. A interação do sistema será medida de alguma maneira (por exemplo, tempo por transação ou precisão de uma transação)? Duas ou

mais pessoas terão de compartilhar informações antes de uma opinião poder ser fornecida? Como será oferecido suporte aos usuários do sistema? Essas e muitas outras questões relacionadas devem ser respondidas antes de o projeto de interface iniciar.

15.4 Etapas no projeto de interfaces

"O projeto interativo [é] uma mescla perfeita de arte gráfica, tecnologia e psicologia."

Brad Wieners

Assim que a análise de interface tiver sido concluída, todas as tarefas (ou objetos e ações) exigidas pelo usuário foram identificadas de forma detalhada, e a atividade de projeto de interface começa. O projeto de interfaces, assim como todo projeto de engenharia de software, é um processo iterativo. Cada etapa de um projeto de interface do usuário ocorre uma série de vezes, elaborando e refinando informações desenvolvidas na etapa anterior.

Embora tenham sido propostos diversos modelos diferentes para projeto de interfaces do usuário (por exemplo, [Nor86], [Nie00]), todos sugerem alguma combinação das seguintes etapas: (1) definir objetos e ações de interface (operações), (2) identificar eventos (ações de usuário) que farão o estado da interface do usuário mudar, (3) descrever a representação de cada estado e (4) indicar como o usuário interpreta cada estado a partir da informação fornecida por meio da interface.

15.4.1 Aplicação das etapas para projeto de interfaces

A definição dos objetos da interface e as ações aplicadas a eles é uma importante etapa no projeto. Para concretizar isso, cenários de usuário são analisados sintaticamente, quase da mesma forma descrita no Capítulo 9. Ou seja, é escrito um caso de uso. Os substantivos (objetos) e os verbos (ações) são isolados para criar uma lista de objetos e ações.

Assim que os objetos e ações tiverem sido definidos e elaborados iterativamente, eles são classificados por tipo. Identificam-se objetos de destino, origem e aplicação. Um *objeto de origem* (por exemplo, um ícone de relatório) é arrastado e solto sobre um *objeto de destino* (por exemplo, um ícone de impressora). A implicação dessa ação é criar um relatório impresso. Um *objeto de aplicação* representa dados específicos à aplicação que não são diretamente manipulados como parte da interação de tela. Por exemplo, uma lista de endereços é usada para armazenar nomes para uma postagem. A própria lista poderia ser classificada, combinada ou filtrada (ações baseadas em menus), mas ela não é arrastada e solta por meio de interação com o usuário.

Quando estiver satisfeito com todas as ações e objetos importantes definidos (para uma iteração de projeto), realize o layout da tela. Assim como outras atividades do projeto de interfaces, o layout da tela é um processo iterativo no qual são realizados o projeto gráfico e o posicionamento dos ícones, a definição de texto de tela descritivo, a especificação e a colocação de títulos para as janelas, bem como a definição de itens de menu principais e secundários. Se uma metáfora do mundo real for apropriada para a aplicação, ela é especificada nesse momento e o layout é organizado para complementar tal metáfora.

Para darmos um breve exemplo das etapas de projeto citadas anteriormente, consideremos um cenário de usuário para o sistema *CasaSegura* (dis-

cutido em capítulos anteriores). Segue um caso de uso preliminar (redigido pelo proprietário do imóvel) para a interface:

Caso de uso preliminar: Quero ter acesso ao meu sistema *CasaSegura* de qualquer ponto remoto via Internet. Por meio de um navegador instalado em meu notebook (enquanto estou no trabalho ou viajando), posso determinar o estado do sistema de alarme, armar ou desarmá-lo, reconfigurar zonas de segurança e ver diferentes ambientes da casa via câmeras de vídeo pré-instaladas.

Para acessar o *CasaSegura* de um ponto remoto, forneço um identificador de usuário e uma senha. Esses definem níveis de acesso (por exemplo, nem todo usuário poderá reconfigurar o sistema) e dão segurança. Uma vez validados, posso verificar o estado do sistema e alterá-lo, armando ou desarmando o *CasaSegura*. Posso reconfigurar o sistema exibindo uma planta da casa, vendo cada um dos sensores de segurança, exibindo cada zona configurada atualmente e modificando as zonas conforme necessário. Posso ver o interior da casa via câmeras de vídeo estrategicamente posicionadas. Posso deslocar e ampliar a imagem de cada câmera para ter visões diferentes do seu interior.

Tomando como base esse caso de uso, são identificados as seguintes tarefas, objetos e dados do proprietário do imóvel:

- *Acessa* o sistema *CasaSegura*.
- *Introduz* um **ID** e **senha** para permitir acesso remoto.
- *Verifica estado do sistema*.
- *Arma* ou *desarma* o sistema *CasaSegura*.
- *Exibe planta* e **localização dos sensores**.
- *Exibe zonas* na planta.
- *Altera zonas* na planta.
- *Exibe localização das câmeras de vídeo* na planta.
- *Seleciona câmera de vídeo* para visualização.
- *Visualiza imagens de vídeo* (quatro quadros por segundo).
- *Desloca* ou *amplia* o **foco da câmera de vídeo**.

Os objetos (em negrito) e as ações (em itálico) são extraídos da lista de tarefas do proprietário do imóvel. A maioria dos objetos citados é composta por objetos de aplicação. Entretanto, **localização das câmeras de vídeo** (um objeto de origem) é arrastado e solto sobre **câmera de vídeo** (um objeto de destino) para criar uma **imagem de vídeo** (uma janela com exibição de vídeo).

É criado um esboço preliminar do layout da tela para monitoramento de vídeo (Figura 15.3).⁴ Para chamar a imagem de vídeo, é selecionado um ícone de localização de câmera de vídeo, C, localizado na planta exibida na janela de monitoramento. Nesse caso, a localização da câmera na sala de estar (SE) é então arrastada e solta sobre o ícone de câmera de vídeo no canto superior esquerdo da tela. Surge a janela de imagem de vídeo, mostrando vídeo *streaming* da câmera localizada em SE. Os controles deslizantes para comandar a

Embora as ferramentas automatizadas possam ser úteis no desenvolvimento de protótipos de layout, algumas vezes basta um lápis e papel.

⁴ Observe que ele difere ligeiramente da implementação desses recursos em capítulos anteriores. Esse poderia ser considerado um projeto preliminar e representa uma alternativa que possa vir a ser considerada.

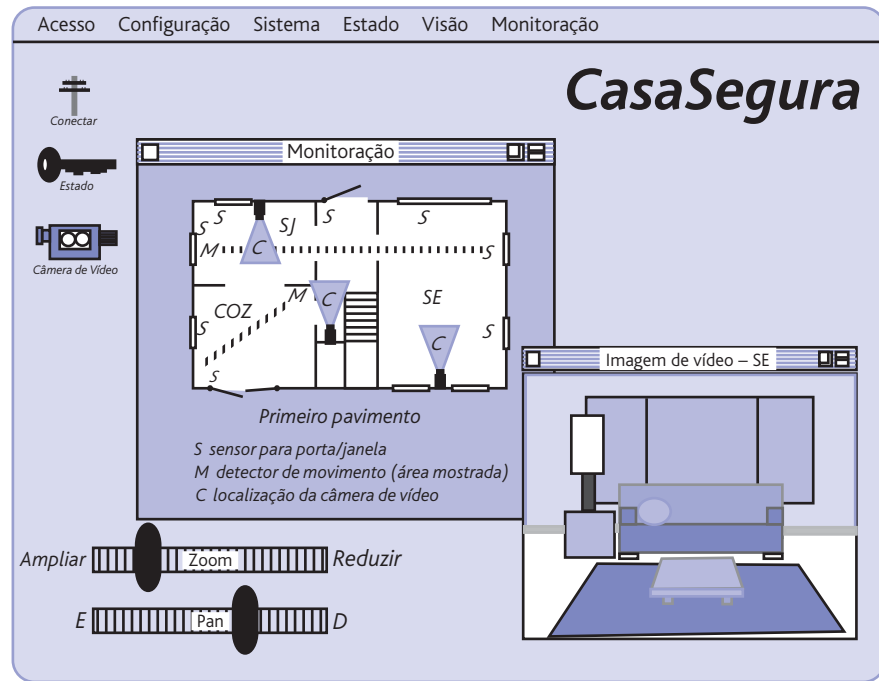


FIGURA 15.3 Layout preliminar da tela.

ampliação e o deslocamento são usados para controlar a ampliação e a direção da imagem de vídeo. Para selecionar uma vista de outra câmera, o usuário apenas arrasta e solta um ícone de localização de câmera diferente sobre o ícone de câmera no canto superior esquerdo da tela.

O esboço do layout teria de ser complementado com uma expansão de cada item de menu contido na barra de menus, indicando as ações disponíveis para o modo (estado) de monitoramento de vídeo. Um conjunto completo de esboços para cada tarefa do proprietário do imóvel citada no cenário de usuário seria criado durante o projeto de interfaces.

15.4.2 Padrões de projeto de interfaces do usuário

As interfaces gráficas do usuário tornaram-se tão comuns que surgiu uma ampla variedade de padrões de projeto de interfaces. Um *padrão de projeto* é uma abstração que prescreve uma solução de projeto para um problema de projeto específico e bem delimitado.

Como exemplo de problema comumente encontrado no projeto de interfaces, consideremos a situação em que um usuário tem de introduzir uma ou mais datas, às vezes com meses de antecedência. Existem várias soluções possíveis para esse problema simples e uma série de padrões diferentes que poderiam ser propostos. Laakso [Laa00] sugere um padrão denominado **FaixaDeCalendário**, que produz um calendário – contínuo e que rola –, onde a data atual é destacada e datas futuras podem ser selecionadas indicando-as no calendário. A metáfora de calendário é bem conhecida de todo usuário e oferece um mecanismo eficiente para colocar uma data futura no contexto.

Ao longo da última década foram propostos vários padrões de projeto de interfaces. Uma discussão mais detalhada sobre padrões de projeto de interfaces do usuário é apresentada no Capítulo 16. Além disso, Erickson [Eri08] indica links para vários conjuntos baseados na Web.

15.4.3 Questões de projeto

À medida que o projeto de uma interface do usuário evolui, quatro questões de projeto comuns quase sempre vêm à tona: tempo de resposta do sistema, recursos de ajuda ao usuário, informações de tratamento de erros e atribuição de nomes a comandos. Infelizmente, muitos projetistas não tratam desses problemas até um ponto relativamente avançado do processo de projeto (algumas vezes a primeira pista de um problema não ocorre até que um protótipo operacional esteja disponível). Em geral, decorrem iterações desnecessárias, atrasos de projeto e frustração por parte do usuário. É muito melhor estabelecer cada um desses como um problema de projeto a ser considerado no início do projeto de software, quando as mudanças são fáceis e custam pouco.

Tempo de resposta. O tempo de resposta do sistema apresenta duas importantes características: duração e variabilidade. Se a resposta do sistema for muito longa, frustração e estresse por parte do usuário serão inevitáveis. A *variabilidade* refere-se ao desvio do tempo de resposta médio e, em vários aspectos, é a característica de tempo de resposta mais importante. Baixa variabilidade permite ao usuário estabelecer um ritmo de interação, mesmo que o tempo de resposta seja relativamente longo. Por exemplo, uma resposta de 1 segundo a um comando normalmente é preferível a uma resposta que varia de 0,1 a 2,5 segundos. Quando a variabilidade é significativa, o usuário sempre se desequilibra, sempre conjecturando se algo “diferente” ocorreu ou não nos bastidores.

Recursos de ajuda. Quase todo usuário de um sistema computacional interativo exige ajuda de vez em quando. O software moderno deve fornecer recursos de ajuda online que permitem ao usuário obter a resposta para determinada questão ou resolver um problema sem ter de abandonar a interface.

Tratamento de erros. Em geral, toda mensagem de erro ou alerta produzida por um sistema interativo deve apresentar as seguintes características: (1) descrever o problema em um jargão que o usuário consiga entender; (2) fornecer conselhos construtivos para recuperação do erro; (3) indicar quaisquer consequências negativas do erro (por exemplo, arquivos de dados provavelmente corrompidos), de modo que o usuário possa fazer uma verificação para garantir que não tenham ocorrido (ou corrigi-las, caso tenham ocorrido); (4) ser acompanhada por algum sinal audível ou visual; e (5) jamais deve colocar a culpa no usuário.

Atribuição de nomes a comandos e menus. O comando digitado já foi o modo mais comum de interação entre o usuário e um software de sistema e era usado comumente para aplicações de todo tipo. Hoje em dia, o uso de interfaces orientadas a janelas e apontar e clicar reduziu a dependência de comandos digitados, mas alguns usuários com maior conhecimento ainda preferem um

Foi proposta uma ampla gama de padrões de projeto para interfaces do usuário. Para encontrar uma grande variedade de sites com esse tipo de padrões, visite <http://www.hcipatterns.org/patterns/borchers/patternIndex.html>.

“Um erro comum que as pessoas cometem ao tentar projetar algo completamente infalível é subestimar a criatividade de completos idiotas.”

Douglas Adams

“A interface do inferno: ‘Para corrigir esse erro e prosseguir, introduza qualquer número primo de 11 dígitos...”

Autor desconhecido

modo de interação orientado a comandos. Surge uma série de problemas de projeto quando são fornecidos comandos digitados ou identificadores de menus como modo de interação:

- Toda opção de menu terá um comando correspondente?
- Qual a forma a ser assumida pelos comandos? As opções incluem uma sequência de controle (por exemplo, alt-P), teclas de função ou uma palavra digitada.
- Qual será o grau de dificuldade para aprender e lembrar-se dos comandos? O que pode ser feito se um comando for esquecido?
- Os comandos podem ser personalizados ou abreviados pelo usuário?
- Os identificadores de menus são autoexplicativos no contexto da interface?
- Os submenus são consistentes com a função sugerida por um item de menu principal?
- Foram estabelecidas convenções apropriadas para a utilização de comandos em toda uma família de aplicações?

Diretrizes para o desenvolvimento de software acessível podem ser encontradas em <http://www-03.ibm.com/able/guidelines/software/access-software.html>.

Acessibilidade da aplicação. À medida que as aplicações de computador tornam-se comuns, os engenheiros de software devem garantir que o projeto de interfaces englobe mecanismos que permitam fácil acesso para aqueles com necessidades especiais. A *acessibilidade* para usuários (e engenheiros de software) que podem vir a ser desafiados fisicamente é um imperativo por razões éticas, jurídicas e comerciais. Uma grande variedade de diretrizes de acessibilidade (por exemplo, [W3C03]) – muitas projetadas para aplicações Web, mas em geral aplicáveis a todos os tipos de software – dão sugestões detalhadas para projeto de interfaces que atingem níveis de acessibilidade variados. Outros (por exemplo, [App13], [Mic13]) apresentam orientações específicas para “tecnologia assistente” que lida com as necessidades daqueles com problemas visuais, auditivos, motores, de fala e de aprendizado.

Internacionalização. Os engenheiros de software e seus gerentes invariavelmente subestimam o esforço e as capacidades necessárias para criar interfaces do usuário que levem em conta as necessidades de diferentes localidades e idiomas. Muitas vezes, as interfaces são projetadas para um país e idioma e então improvisadas para funcionar em outros. O desafio para os projetistas de interfaces é criar um software “globalizado”. Isto é, as interfaces do usuário devem ser projetadas para atender um núcleo genérico de funcionalidade que possa ser entregue a todos que usam o software. Recursos de *localização* permitem que a interface seja personalizada para um mercado específico.

Uma grande variedade de diretrizes de internacionalização (por exemplo, [IBM13]) se encontra disponível para os engenheiros de software. Tais diretrizes tratam de uma ampla variedade de questões de projeto (por exemplo, os layouts de tela talvez difiram em vários mercados) e problemas de implementação diferenciados (por exemplo, diferentes alfabetos talvez criem exigências especiais de atribuição de nomes e espaçamento). O padrão *Unicode* [Uni03] foi desenvolvido para tratar do intimidante desafio de gerenciar dezenas de linguagens naturais com centenas de caracteres e símbolos.

FERRAMENTAS DO SOFTWARE

**Desenvolvimento de interfaces do usuário**

Objetivo: Essas ferramentas permitem a um engenheiro de software criar uma sofisticada interface gráfica do usuário com relativamente pouco desenvolvimento de software personalizado. As ferramentas dão acesso a componentes reutilizáveis e tornam a criação da interface uma simples questão de selecionar capacidades predefinidas, agrupadas usando-se a ferramenta.

Mecanismos: As interfaces do usuário modernas são construídas usando-se um conjunto de componentes reutilizáveis, associados a alguns componentes personalizados desenvolvidos para oferecer recursos especializados. A maioria das ferramentas para desenvolvimento de interfaces do usuário permite que um engenheiro de software crie uma interface usando recursos de “arrastar e soltar”. Isto é, o desenvolvedor escolhe alguns de vários recursos predefinidos (por exemplo, recursos de construção de formulários, mecanismos de interação, processamento de comandos) e insere tais recursos no conteúdo da interface a ser criada.

Ferramentas representativas:⁵

LegaSuite GUI, desenvolvida pela Seagull Software (<http://www-304.ibm.com/partnerworld/gsd/solution-details.do?solution=1020&expand=true&lc=en>), permite a criação de GUIs baseadas em navegadores e oferece recursos para a reengenharia de interfaces antiquadas.

Motif Common Desktop Environment, desenvolvida pelo The Open Group (www.osf.org/tech/desktop/cde/), é uma interface gráfica do usuário integrada para sistemas desktop abertos. Ela oferece uma única interface gráfica padrão para o gerenciamento de dados e arquivos (a área de trabalho gráfica) e aplicações.

Altia Design 8.0, desenvolvida pela Altia (www.altia.com), é uma ferramenta para criação de GUIs em uma variedade de plataformas diferentes (por exemplo, automotiva, dispositivos portáteis, industrial).

15.5 Projeto de interfaces para WebApps e aplicativos móveis

Toda interface do usuário – seja ela projetada para uma WebApp, um aplicativo móvel, uma aplicação de software tradicional, um produto de consumo ou para um dispositivo industrial – deve apresentar as características de usabilidade discutidas neste capítulo. Dix [Dix99] diz que as interfaces de WebApps e aplicativos móveis respondem a três perguntas básicas: *Onde me encontro? O que posso fazer agora? Onde estive e aonde posso ir?* As respostas dessas perguntas permitem que o usuário entenda o contexto e navegue com mais eficiência pela aplicação.

“Se um site é perfeitamente utilizável, mas falta um estilo de projeto elegante e adequado, ele falhará.”

Curt Cloninger

15.5.1 Princípios e diretrizes para projeto de interfaces

A interface do usuário de uma WebApp ou de um aplicativo móvel é sua “primeira impressão”. Independentemente do valor de seu conteúdo, da sofisticação de seus recursos e serviços de processamento, bem como do benefício geral da própria aplicação, uma interface malfeita desapontará o usuário em potencial e talvez faça com que ele, de fato, procure outra opção. Devido ao grande número de WebApps e aplicativos móveis concorrentes em praticamente qualquer área de aplicação, a interface tem de “atrair” imediatamente um possível usuário.

Evidentemente, existem importantes diferenças entre WebApps e aplicativos móveis. Devido às restrições físicas impostas pelos dispositivos móveis

⁵ A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nessa categoria.

pequenos (por exemplo, smartphones), o projetista de interface para aplicativos móveis deve compactar a interação de forma orientada. Contudo, os princípios básicos discutidos nesta seção continuam válidos.

Bruce Tognozzi [Tog01] define um conjunto de princípios de projeto fundamentais que conduzem a uma melhor usabilidade:⁶

Existe um conjunto de princípios básicos que possam ser aplicados à medida que desenhamos uma GUI?

Antecipação. *Uma aplicação deve ser projetada para prever o próximo passo do usuário.* Por exemplo, um usuário solicitou um objeto de conteúdo que apresenta informações sobre um driver de impressora para uma nova versão de um sistema operacional. O projetista da WebApp deve prever que o usuário pode vir a solicitar um download do driver e oferecer recursos de navegação que permitam que isso aconteça diretamente.

Comunicação. *A interface deve comunicar o estado de qualquer atividade iniciada pelo usuário.* A comunicação deve ser óbvia (por exemplo, uma mensagem de texto) ou sutil (por exemplo, a imagem de uma folha de papel deslocando-se pela impressora para indicar que a impressão está em andamento).

Consistência. *O uso de controles de navegação, menus, ícones e estética (por exemplo, cor, forma, layout) deve ser consistente.* Por exemplo, se um aplicativo móvel utiliza um conjunto de quatro ícones (para representar funções importantes) na parte inferior da tela, esses ícones devem aparecer em todas as telas e não ser movido para sua parte superior. O significado dos ícones deve ser evidente no contexto do aplicativo.

Autonomia controlada. *A interface deve facilitar a movimentação do usuário pela aplicação, mas deve fazê-lo de forma que faça valer as convenções de navegação estabelecidas para a aplicação.* Por exemplo, a navegação para conteúdo que exige acesso controlado deve acontecer por meio de identificação e senhas de usuário, e não deve existir nenhum mecanismo de navegação que possibilite a um usuário burlar tais controles.

Eficiência. *O projeto de uma aplicação e sua interface deve otimizar a eficiência de trabalho do usuário, e não a eficiência do desenvolvedor que a projeta e constrói ou o ambiente cliente/servidor que a executa.* Tognozzi [Tog01] discute isso ao escrever: “Esta simples verdade é a razão de ser tão importante para todos... Perceberem a importância de fazer com que a produtividade do usuário seja a primeira meta e compreender a diferença vital entre construir uma aplicação eficiente e dar poderes a um usuário eficiente”.

Flexibilidade. *A interface deve ser flexível o bastante para permitir que alguns usuários cumpram tarefas diretamente, ao passo que outros devem explorar a aplicação de maneira um tanto aleatória.* Em todos os casos, ela deve permitir ao usuário compreender onde ele se encontra e também lhe dar um recurso para desfazer erros e refazer caminhos de navegação mal escolhidos.

Foco. *A interface (e o conteúdo apresentado) deve permanecer focado na(s) tarefa(s) do usuário em questão.* Esse conceito é particularmente importante

⁶ Os princípios de Tognozzi originais foram adaptados e estendidos para uso neste livro. Veja [Tog01] para uma discussão mais ampla sobre esses princípios.

para aplicativos móveis, que podem se tornar muito congestionados pelas tentativas do projetista de fazer coisas demais.

Objetos de interface humana. *Desenvolveu-se uma vasta biblioteca de objetos de interface humana reutilizáveis para WebApps e aplicativos móveis. Utilize-as.* Qualquer objeto de interface que possa ser “visto, ouvido, tocado ou de alguma outra forma percebido” [Tog01] por um usuário pode ser obtido de qualquer uma das várias bibliotecas de objetos.

Redução da latência. *Em vez de fazer com que o usuário espere por alguma operação interna completar (por exemplo, baixar uma imagem complexa), a aplicação deve usar multitarefas de uma forma que deixe o usuário prosseguir com seu trabalho como se a operação tivesse sido completada.* Além de reduzir a latência, os atrasos devem ser reconhecidos para que o usuário entenda o que está ocorrendo. Isso inclui: (1) fornecer feedback de áudio quando uma seleção não resulte em uma ação imediata pela aplicação, (2) exibir uma animação de relógio ou barra de progresso para indicar que o processamento está em andamento e (3) fornecer algum entretenimento (por exemplo, uma apresentação de texto ou animação) enquanto ocorre processamento moroso.

Facilidade de aprendizagem. *A interface de uma aplicação deve ser projetada para minimizar o tempo de aprendizagem e, uma vez aprendida, minimizar a reaprendizagem necessária quando a aplicação for reutilizada.* Em geral, a interface deve enfatizar um projeto simples e intuitivo que organiza conteúdo e funcionalidade em categorias óbvias para o usuário.

Metáforas. *Uma interface que usa uma metáfora de interação é mais fácil de aprender e usar, desde que a metáfora seja apropriada para a aplicação e para o usuário.* A metáfora deve invocar imagens e conceitos da experiência do usuário, mas não precisa ser uma reprodução exata de uma experiência do mundo real.

Legibilidade. *Todas as informações apresentadas em uma interface devem ser legíveis por jovens e idosos.* O projetista de interfaces deve dar prioridade a estilos de tipos e tamanhos de fontes controláveis pelo usuário e fundos coloridos que aumentem o contraste.

Acompanhar o estado da interação. *Quando apropriado, o estado da interação de usuário deve ser acompanhado e armazenado de modo que um usuário possa sair do sistema e retornar mais tarde, prosseguindo do ponto onde parou.* Em geral, podem ser projetados cookies para armazenar informações de estado. Entretanto, os cookies são uma tecnologia controversa, e outras soluções de projeto talvez sejam mais aceitáveis para alguns usuários.

Navegação visível. *Uma interface bem projetada fornece “a ilusão de que os usuários se encontram no mesmo lugar, com o trabalho sendo levado até eles”* [Tog01]. Quando é usada essa abordagem, a navegação não é uma preocupação para o usuário. Ao contrário, o usuário recupera objetos de conteúdo e seleciona funções que são exibidas e executadas através da interface.

Nielsen e Wagner [Nie96] sugerem algumas “proibições” pragmáticas para projeto de interfaces (baseadas na experiência dos autores na reformu-

“O melhor caminho é aquele com o menor número de passos. Diminua a distância entre o usuário e sua meta.”

Autor desconhecido

Metáforas são uma excelente ideia, pois espelham a experiência do mundo real. Apenas certifique-se de que a metáfora escolhida seja bem conhecida pelos usuários.



Revisão de projeto de interfaces

Cena: Escritório de Doug Miller.

Atores: Doug Miller (gerente do grupo de engenharia de software do CasaSegura) e Vinod Raman, membro da equipe de engenharia de software do produto CasaSegura.

Conversa:

Doug: Vinod, você e a equipe tiveram a chance de revisar o protótipo de interface para comércio eletrônico CasaSeguraGarantida.com?

Vinod: Sim... todos nós navegamos nele do ponto de vista técnico, e tenho um bocado de comentários. Ontem, enviei-os por e-mail a Sharon [gerente da equipe de WebApps do fornecedor terceirizado para o site de comércio eletrônico CasaSegura].

Doug: Você e a Sharon podiam se reunir e discutir os pequenos detalhes... Entregue-me um resumo das questões importantes.

Vinod: Em termos gerais, eles fizeram um bom trabalho. Nada de revolucionário, porém é uma interface típica para comércio eletrônico. Estética aceitável, layout razoável, cobriram todas as funções importantes...

Doug (sorrindo tristemente): Mas?

Vinod: Bem, há algumas coisinhas...

Doug: Como...?

Vinod (mostrando a Doug a sequência de storyboards para o protótipo de interface): Aqui está o menu das funções principais que é apresentado na homepage:

**Conheça o CasaSegura.
Descreva sua casa.**

Obtenha recomendações de componentes para o CasaSegura.

Compre um sistema CasaSegura.

Obtenha suporte técnico.

O problema não é essas funções. Todas estão corretas, mas o nível de abstração não.

Doug: Todas são funções importantes, não é mesmo?

Vinod: São, mas aqui está o principal... é possível comprar um sistema apenas entrando com uma lista de componentes... Não há necessidade de descrever a casa se não quiser fazê-lo. Sugeriria apenas quatro opções de menu na homepage:

Conheça o CasaSegura.

Especifique o sistema CasaSegura de que você precisa.

Compre um sistema CasaSegura.

Obtenha suporte técnico.

Ao selecionar **Especifique o sistema CasaSegura de que você precisa**, você terá então as seguintes opções:

Selecionar Componentes do CasaSegura.

Obtenha recomendações de componentes para o CasaSegura.

Se você for um usuário experiente, escolherá componentes de um conjunto de menus *pull-down* classificados por sensores, câmeras, painéis de controle etc. Se precisar de ajuda, você solicitará uma recomendação que exigirá que descreva sua casa. Imagino que seja um pouco mais lógico.

Doug: Concordo. Você conversou com a Sharon sobre isso?

Vinod: Não, queria discutir isso primeiro com o Marketing; depois telefonaria para ela.

lação de uma importante WebApp). Elas oferecem um excelente complemento aos princípios sugeridos anteriormente nesta seção.

"As pessoas têm muito pouca paciência com sites WWW mal projetados."

**Jakob Nielsen e
Annette Wagner**

- Não force o usuário a ler toneladas de texto, particularmente quando o texto explica a operação da WebApp ou auxilia na navegação.
- Não faça os usuários descenderem até o fim da página, a não ser que seja absolutamente inevitável.
- Não dependa de funções do navegador para ajudar na navegação.
- Não permita que a estética suplante a funcionalidade.
- Não obrigue o usuário a ficar procurando na tela para determinar como fazer o link para outros conteúdos ou serviços.

Uma interface bem projetada aumenta a percepção do usuário do conteúdo ou serviços fornecidos pelo site. Ela não precisa, necessariamente, ser chamativa, mas sempre deve ser bem estruturada e ergonomicamente sólida.

15.5.2 Fluxo de trabalho de projeto de interfaces para WebApps e aplicativos móveis⁷

No início do capítulo, citamos que o projeto de interfaces do usuário começa com a identificação dos requisitos do usuário, de tarefas e dos ambientes. Uma vez que as tarefas de usuário tenham sido identificadas, cenários de usuário (casos de uso) são criados e analisados para definir um conjunto de ações e objetos de interface.

Informações contidas no modelo de requisitos formam a base para a criação de um layout de tela que representa o design gráfico e o posicionamento de ícones, a definição de texto de tela descritivo, a especificação e a colocação de títulos para as janelas, bem como a especificação de itens de menu principais e secundários. Então, são usadas ferramentas para prototipação e, por fim, a implementação do modelo de projeto para interface. As tarefas a seguir representam um fluxo de trabalho rudimentar:

1. **Revise as informações contidas no modelo de requisitos e refine conforme necessário.**
2. **Desenvolva um esboço do layout para interfaces WebApp.** Se o layout da interface (um protótipo desenvolvido durante a modelagem de requisitos) já existe, ele deve ser revisado e refinado conforme necessário.
3. **Mapeie os objetivos do usuário em ações de interface específicas.** Para a grande maioria das WebApps e aplicativos móveis, o usuário terá um conjunto relativamente pequeno de objetivos primários. Eles devem ser mapeados na interface específica. Basicamente, temos de responder à seguinte pergunta: “Como a interface habilita o usuário a cumprir cada objetivo?”.
4. **Defina um conjunto de tarefas de usuário que estão associadas a cada ação.** Cada ação de interface (por exemplo, “comprar um produto”) está associada a um conjunto de tarefas de usuário. Tais tarefas foram identificadas durante a modelagem de requisitos. Durante o projeto, elas devem ser mapeadas em interações específicas que englobem questões de navegação, objetos de conteúdo e funções da aplicação.
5. **Imagens de tela storyboard para cada ação de interface.** À medida que cada ação é considerada, uma sequência de imagens de storyboard (imagens de tela) deve ser criada para representar como a interface responde à interação com o usuário. Devem-se identificar objetos de conteúdo (mesmo que ainda não tenham sido projetados e desenvolvidos), e links de navegação devem ser indicados.
6. **Refine o layout de interface e storyboards usando entrada do projeto estético.** Na maioria dos casos, você será responsável pelo layout e pelo storyboarding preliminar, mas o aspecto estético para um importante site comercial em geral é desenvolvido por designers gráficos e não por técnicos.
7. **Identifique objetos de interface do usuário necessários para implementar a interface.** Essa tarefa poderia exigir uma busca em uma biblioteca de objetos para encontrar os objetos (classes) reutilizáveis apropriados para

⁷ Discussões mais detalhadas sobre o projeto para WebApps e aplicativos móveis são apresentadas nos Capítulos 17 e 18, respectivamente.

a interface. Além disso, quaisquer classes personalizadas são especificadas neste momento.

8. **Desenvolva uma representação procedural da interação do usuário com a interface.** Essa tarefa opcional usa diagramas de sequências UML e/ou diagramas de atividades (Apêndice 1) para representar o fluxo de atividades (e decisões) que ocorrem à medida que o usuário interage com a WebApp.
9. **Desenvolva uma representação comportamental da interface.** Essa tarefa opcional faz uso de diagramas de estados UML (Apêndice 1) para representar transições de estados e os eventos que os provocam. São definidos mecanismos de controle (isto é, os objetos e as ações disponíveis para o usuário alterar o estado de uma aplicação).
10. **Descreva o layout da interface para cada estado.** Usando as informações de projeto desenvolvidas nas Tarefas 2 e 5, associe um layout ou imagens de tela específicas a cada estado da WebApp descrito na Tarefa 8.
11. **Refine e revise o modelo de projeto de interface.** A revisão da interface deve se concentrar na usabilidade.

É importante notar que o conjunto final de tarefas escolhido deve ser adaptado às exigências especiais da aplicação a ser construída.

15.6 Avaliação de projeto

Assim que um protótipo operacional de interface do usuário for criado, ele deve ser avaliado para determinar se atende aos requisitos do usuário. A avaliação pode abranger um espectro de formalidade que vai de um *test-drive* informal em que um usuário fornece feedback imediato até um estudo formalmente projetado que usa os métodos estatísticos para a avaliação de questionários preenchidos por uma população de usuários.

O ciclo de avaliação de interfaces do usuário assume a forma indicada na Figura 15.4. Após a finalização do modelo de projeto, cria-se um protótipo de primeiro nível. O protótipo é avaliado pelo usuário,⁸ o qual nos fornece comentários diretos sobre a eficácia da interface. Além disso, se forem usadas técnicas de avaliação formais (por exemplo, questionários, formulários de avaliação), podemos extrair informações desses dados (por exemplo, 80% de todos os usuários não gostam do mecanismo para salvar arquivos de dados). São feitas modificações de projeto com base nas informações fornecidas pelos usuários, e é criado o protótipo do nível seguinte. O ciclo de avaliação continua até que nenhuma modificação adicional seja necessária para o projeto de interfaces.

A abordagem de prototipação é eficaz, mas é possível avaliar a qualidade de uma interface do usuário antes de um protótipo ser construído?⁹ Se forem

⁸ É importante notar que os especialistas em ergonomia e projeto de interfaces também podem realizar revisões da interface. Essas revisões são denominadas *avaliações heurísticas* ou *revisões cognitivas*.

⁹ Alguns engenheiros de software preferem desenvolver uma maquete de baixa fidelidade da interface do usuário (IU), denominada protótipo em papel, para permitir que os envolvidos testem o conceito da IU antes de entregar quaisquer recursos de programação. O processo está descrito aqui: http://www.paperprototyping.com/what_examples.html.

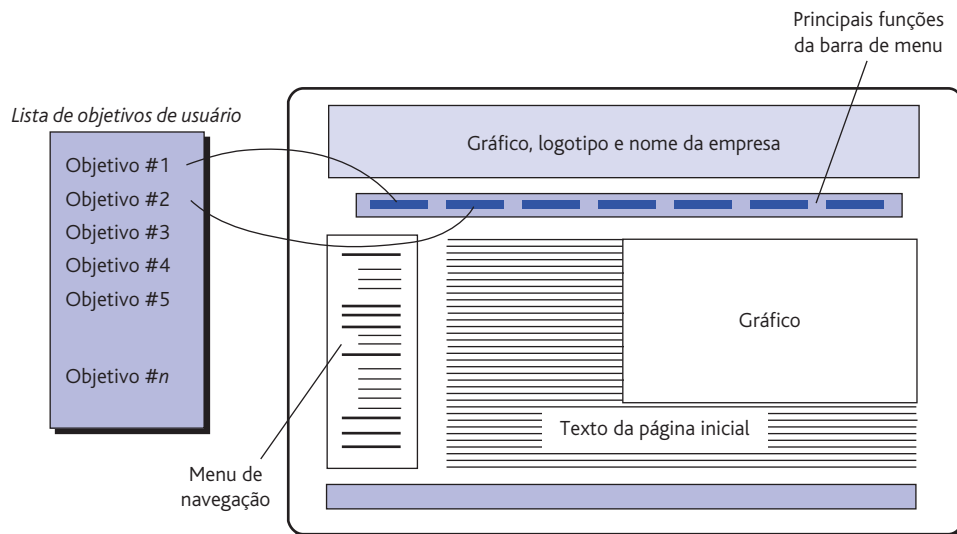


FIGURA 15.4 O ciclo de avaliação de projeto de interfaces.

identificados e corrigidos possíveis problemas precocemente, o número de laços realizados no ciclo de avaliação será reduzido, e o tempo de desenvolvimento será abreviado. Se um modelo de projeto de interface tiver sido criado, uma série de critérios de avaliação [Mor81] poderá ser aplicada durante revisões de projeto precoces:

1. A duração e a complexidade do modelo de requisitos ou a especificação por escrito do sistema e sua interface dão uma indicação do volume de aprendizado exigido dos usuários do sistema.
2. O número de tarefas de usuário especificado e o número médio de ações por tarefa indicam o tempo de interação e a eficiência geral do sistema.
3. O número de ações, tarefas e estados do sistema indicados pelo modelo de projeto sugere a carga de memória necessária por parte dos usuários do sistema.
4. O estilo da interface, recursos de ajuda e o protocolo de tratamento de erros dão uma indicação geral da complexidade da interface e do grau de aceitação por parte dos usuários.

Assim que o primeiro protótipo tiver sido construído, podemos coletar uma variedade de dados qualitativos e quantitativos que vão auxiliar na avaliação da interface. Para a coleta de dados qualitativos, podemos distribuir questionários para permitir que os usuários avaliem o protótipo da interface. Se forem desejados dados quantitativos, pode ser realizada uma espécie de análise de estudo de tempos. Os usuários são observados durante a interação, e dados – como uma série de tarefas corretamente completadas durante um período de tempo padrão, frequência de ações, sequência de ações, tempo gasto “observando” a tela, número e tipos de erros, tempo de recuperação de erros, tempo gasto usando o sistema de ajuda e o número de referências de ajuda por período de tempo padrão – são coletados e usados como um guia para modificação da interface.

Uma discussão completa dos métodos de avaliação de interfaces do usuário está fora dos objetivos deste livro. Para maiores informações, consulte [Hac98] e [Sto05].

15.7 Resumo

A interface do usuário é discutivelmente o elemento mais importante de um produto ou sistema computacional. Se a interface for mal projetada, a capacidade de o usuário aproveitar todo o poder computacional e conteúdo de informações de uma aplicação pode ser seriamente afetada. Na realidade, uma interface fraca pode fazer com que uma aplicação, em outros aspectos bem projetada e solidamente implementada, falhe.

Três importantes princípios orientam o projeto de interfaces do usuário eficazes: (1) deixar o usuário no comando, (2) reduzir a carga de memória do usuário e (3) tornar a interface consistente. Para obter uma interface que observe esses princípios, deve ser realizado um processo de projeto organizado.

O desenvolvimento de uma interface do usuário começa com uma série de tarefas de análise. A análise dos usuários define os perfis de vários usuários e é reunida com base em uma série de fontes técnicas e comerciais. A análise de tarefas define as tarefas e ações de usuários usando uma abordagem de refinamento ou orientada a objetos, aplicando casos de uso, elaboração de tarefa e objetos, análise de fluxos de trabalho e representações hierárquicas de tarefas para entender completamente a interação homem-computador. A análise do ambiente identifica as estruturas físicas e sociais em que a interface deve operar.

Uma vez identificadas as tarefas, são criados e analisados cenários de usuário para definir um conjunto de ações e objetos de interface. Isso fornece uma base para a criação de um layout de tela que represente o design gráfico e o posicionamento de ícones, a definição de texto de tela descritivo, a especificação e a colocação de títulos para as janelas, bem como a especificação de itens de menu principais e secundários. Questões de projeto, como tempo de resposta, estrutura de comandos e ações, tratamento de erros e recursos de ajuda, são consideradas à medida que o modelo de projeto é refinado. Uma grande variedade de ferramentas de implementação é usada para construir um protótipo para avaliação por parte do usuário.

Assim como ocorre no projeto de interfaces para software convencional, o projeto de interfaces para WebApps e aplicativos móveis descreve a estrutura e a organização de uma interface do usuário e abrange a representação do layout de tela, a definição dos modos de interação e a descrição de mecanismos de navegação. Um conjunto de princípios para o projeto de interfaces e um fluxo de trabalho para projeto de interfaces orientam o projetista de WebApps ou aplicativos móveis quando o layout e mecanismos de controle da interface são projetados.

A interface do usuário é a janela para o software. Em muitos casos, ela molda a percepção do usuário quanto à qualidade de um sistema. Se a “janela” estiver embaçada, ondulada ou quebrada, o usuário poderá rejeitar um sistema computacional que, de outra forma, seria considerado poderoso.

Problemas e pontos a ponderar

15.1. Descreva a pior interface com a qual você já tenha trabalhado até hoje e critique-a em relação aos conceitos introduzidos neste capítulo. Descreva a melhor interface com a qual já tenha trabalhado até hoje e critique-a em relação aos conceitos introduzidos neste capítulo.

15.2. Desenvolva mais dois princípios de projeto que “deixem o usuário no comando”.

15.3. Desenvolva mais dois princípios de projeto que “reduzam a carga de memória do usuário”.

15.4. Desenvolva mais dois princípios de projeto que “tornem a interface consistente”.

15.5. Considere uma das seguintes aplicações interativas (ou uma aplicação designada por seu professor):

- a. Um sistema de editoração eletrônica
- b. Um sistema de projeto auxiliado por computador
- c. Um sistema de projeto de interiores (conforme descrito na Seção 15.3.2)
- d. Um sistema automatizado de matrículas para uma universidade
- e. Um sistema de gerenciamento de bibliotecas
- f. Uma urna eletrônica baseada na Internet para eleições públicas
- g. Um sistema de *home banking*
- h. Uma aplicação interativa designada por seu professor

Desenvolva um modelo de usuário, modelo de projeto, modelo mental e um modelo de implementação para qualquer um desses sistemas.

15.6. Realize uma análise detalhada das tarefas para qualquer um dos sistemas enumerados no Problema 15.5. Use uma abordagem de refinamento ou orientada a objetos.

15.7. Acrescente pelo menos cinco outras questões à lista desenvolvida para análise de conteúdo da Seção 15.3.3.

15.8. Continuando o Problema 15.5, defina objetos e ações de interface para a aplicação que você escolheu. Identifique cada tipo de objeto.

15.9. Desenvolva um conjunto de layouts de tela com uma definição de itens de menu primários e secundários para o sistema escolhido no Problema 15.5.

15.10. Desenvolva um conjunto de layouts de tela com uma definição de itens de menu primários e secundários para o sistema *CasaSegura*. Você pode optar por uma abordagem diferente daquela indicada para o layout de tela da Figura 15.3.

15.11. Descreva sua abordagem para recursos de ajuda ao usuário para o modelo de projeto de análise de tarefas e para a análise de tarefas realizada como parte dos Problemas 15.5, 15.7 e 15.8.

15.12. Dê alguns exemplos que ilustrem por que a variabilidade no tempo de resposta pode ser um problema.

15.13. Desenvolva uma abordagem que integraria automaticamente mensagens de erro e um recurso de ajuda ao usuário. Isto é, o sistema reconheceria automaticamente o tipo de erro e forneceria uma janela de ajuda com sugestões para corrigi-lo. Realize um projeto de software razoavelmente completo que considere estruturas de dados e algoritmos apropriados.

15.14. Desenvolva um questionário para avaliação de interfaces contendo 20 perguntas genéricas que se aplicariam à maioria das interfaces. Faça com que 10 colegas de classe completem o questionário para um sistema interativo que todos usarão. Sintetize os resultados e relate-os à sua turma.

Leituras e fontes de informação complementares

Embora este livro não seja especificamente sobre interfaces homem-computador, muito daquilo que Donald Norman (*The Design of Everyday Things*, reedição, Basic Books, 2002) tem a dizer sobre a psicologia de projeto efetivo se aplica a uma interface do usuário. É uma leitura recomendada a todos que estejam empenhados em desenvolver um projeto de interface de alta qualidade. Weinschenk (*100 Things Every Designer Should Know About People*, New Riders, 2011) não se concentra especificamente em software, mas apresenta uma discussão informativa sobre o projeto centrado no usuário. Johnson (*Designing with the Mind in Mind*, Morgan Kaufman, 2010) utiliza psicologia cognitiva para criar regras para o projeto de interfaces eficientes.

As interfaces gráficas do usuário são onipresentes no mundo moderno da computação. Seja ela um caixa eletrônico, um telefone celular, um painel eletrônico de comandos em um automóvel, um site ou uma aplicação comercial, a interface do usuário oferece uma janela para o software. É por essa razão que há inúmeros livros que tratam de projeto de interfaces. Dentre os muitos que valem a pena considerar, estão:

Ballard (*Designing the Mobile User Experience*, Wiley, 2007).

Butow (*User Interface Design for Mere Mortals*, Addison-Wesley, 2007).

Cooper e seus colegas (*About Face 3: The Essentials of Interaction Design*, 3ª ed., Wiley, 2007).

Galitz (*The Essential Guide to User Interface Design*, 3ª ed., Wiley, 2007).

Goodwin e Cooper (*Designing for the Digital Age: How to Create Human-Centered Products and Services*, Wiley, 2009).

Hartson e Pyla (*The UX Book: Process and Guidelines for Ensuring a Quality User Experience*, Morgan Kaufman, 2012).

Lehikonen e colegas (*Personal Content Experience: Managing Digital Life in the Mobile Age*, Wiley-Interscience, 2007).

Nielsen (*Coordinating User Interfaces for Consistency*, Morgan-Kaufmann, 2006).

Pratt e Nunes (*Interactive Design*, Rockport, 2013).

Rogers e colegas (*Interaction Design: Beyond Human-Computer Interaction*, 3ª ed., Wiley, 2011).

Shneiderman e colegas (*Designing the User Interface: Strategies for Effective Human-Computer Interaction*, 5ª ed., Addison-Wesley, 2009).

Tidwell (*Designing Interfaces*, O'Reilly Media, 2ª ed., 2011).

Johnson (*GUI Bloopers: Common User Interface Design Don'ts and Do's*, 2ª ed., Morgan-Kaufmann, 2007; e *GUI Bloopers: Don'ts and Do's for Software Developers and Web Designers*, Morgan-Kaufmann, 2000) fornece uma útil orientação para aqueles que aprendem mais efetivamente por meio do exame de contraexemplos. Um agradável livro de Cooper (*The Inmates Are Running the Asylum*, Sams Publishing, 2004) discute por que os produtos de alta tecnologia nos deixam loucos e como projetar algum deles que não tenha esse efeito.

Uma ampla gama de fontes de informação sobre projeto de interfaces se encontra à disposição na Internet. Uma lista atualizada de referências relevantes (em inglês) para o projeto de interfaces do usuário pode ser encontrada no site: www.mhhe.com/pressman.

Projeto baseado em padrões

16

Todos nós já nos deparamos com um problema de projeto e, silenciosamente, pensamos: *será que alguém já desenvolveu uma solução para esse problema?* A resposta é quase sempre *sim!* O problema é encontrar a solução; garantir que, de fato, adapte-se ao problema em questão; entender as restrições que talvez limitem a maneira pela qual a solução é aplicada e, por fim, traduzir a solução proposta para seu ambiente de projeto.

Mas e se a solução fosse codificada de alguma forma? E se existisse uma maneira padronizada de descrever um problema (de tal forma que pudéssemos pesquisá-lo) e um método organizado para representar a solução para o problema? Os problemas de software seriam codificados e descritos usando-se um modelo padronizado e seriam propostas soluções (com restrições) para eles. Denominado *padrões de projeto*, esse método codificado para descrição de problemas e suas soluções permite que os profissionais da engenharia de software adquiram conhecimento de projeto para que ele seja reutilizado.

Conceitos-chave

erros de projeto	401
frameworks	351
granularidade	365
linguagens de padrões	353
padrões criacionais	350
padrões de arquitetura	359
padrões	
comportamentais	350
padrões de projeto	348
padrões de projeto de	
componentes	360
padrões de projeto para	
interfaces do usuário	362
padrões de projeto para	
WebApps	364
padrões estruturais	350

PANORAMA

conjunto de soluções comprovadas para um conjunto de problemas claramente delineados. Cada problema (e sua solução) é descrito por um padrão de projeto que foi catalogado e investigado por outros engenheiros de software, que se depararam com o problema e implementaram a solução ao projetarem outras aplicações. Cada padrão de projeto nos oferece uma abordagem comprovada para parte do problema a ser resolvido.

Quem realiza? Um engenheiro de software examina cada problema que surge para uma nova aplicação e tenta encontrar uma solução relevante por meio de pesquisa em um ou mais repositórios de padrões.

Por que é importante? Você já ouviu a expressão "reinventar a roda"? Ela ocorre toda hora em desenvolvimento de software e é uma perda de tempo e energia. Ao usarmos padrões de projeto, podemos encontrar uma solução comprovada para um problema específico. À medida que cada padrão é aplicado, são integradas soluções, e a aplicação a ser construída se aproxima cada vez mais de um projeto completo.

O que é? O projeto baseado em padrões cria uma nova aplicação por meio da busca de um

Quais são as etapas envolvidas? O modelo de requisitos é examinado para isolar o conjunto hierárquico de problemas a serem resolvidos. O espaço de problemas é subdividido de modo que subconjuntos de problemas associados a funções e características de software específicas possam ser identificados. Os problemas também podem ser organizados por tipo: de arquitetura, de componentes, algorítmicos, de interfaces do usuário etc. Uma vez definido um subconjunto de problemas, pesquisam-se um ou mais repositórios de padrões para determinar se existe um padrão de projeto, representado em um nível de abstração apropriado. Padrões aplicáveis são adaptados às necessidades específicas do software a ser construído. A solução de problemas personalizados é aplicada em situações em que não foi encontrado nenhum padrão.

Qual é o artefato? É desenvolvido um modelo de projeto que representa a estrutura da arquitetura, a interface do usuário e detalhes em nível de componentes.

Como garantir que o trabalho foi realizado corretamente? À medida que cada padrão de projeto é traduzido em algum elemento do modelo de projeto, os artefatos são revistos em termos de clareza, correção, completude e consistência em relação aos requisitos e entre si.

sistema de forças	348
tabela para organização de padrões	358
tipos de padrões	349

O início da história dos padrões de software não começa com um cientista da computação, mas com um arquiteto, Christopher Alexander, que reconheceu o fato de ser encontrado um conjunto de problemas recorrentes toda vez que um edifício era projetado. Ele caracterizou esses problemas recorrentes e suas soluções como *padrões*, descrevendo-os da seguinte maneira [Ale77]:

Cada padrão descreve um problema que ocorre repetidamente em nosso ambiente e então descreve o cerne de uma solução para aquele problema para podermos usá-la repetidamente um milhão de vezes sem jamais ter de fazer a mesma coisa duas vezes.

As ideias de Alexander foram traduzidas inicialmente para o mundo do software em livros como os de Gamma [Gam95], Buschmann [Bus96] e seus colegas.¹ Hoje, existem dezenas de repositórios de padrões, e projetos baseados em padrões podem ser aplicados em diversos domínios de aplicação.

16.1 Padrões de projeto

Forças são as características do problema e os atributos de uma solução que restringem a maneira como o projeto pode ser desenvolvido.

"Nossa responsabilidade é fazer o que podemos, aprender o que podemos, aperfeiçoar as soluções e passá-las adiante."

Richard P. Feynman

Um *padrão de projeto* pode ser caracterizado como "uma regra de três partes que expressa uma relação entre um contexto, um problema e uma solução" [Ale79]. Para projeto de software, o *contexto* permite ao leitor compreender o ambiente em que o problema reside e qual solução poderia ser apropriada nesse ambiente. Um conjunto de requisitos, incluindo limitações e restrições, atua como um *sistema de forças* que influencia a maneira pela qual o problema pode ser interpretado em seu contexto e como a solução pode ser aplicada eficientemente.

A maioria dos problemas possui várias soluções, porém uma solução é eficaz somente se for apropriada no contexto do problema existente. É o sistema de forças que faz com que um projetista escolha uma solução específica. O intuito é fornecer uma solução que melhor atenda ao sistema de forças, mesmo quando essas forças são contraditórias. Por fim, toda solução tem consequências que poderiam ter um impacto sobre outros aspectos do software, e ela própria poderia fazer parte do sistema de forças para outros problemas a serem resolvidos no sistema mais amplo.

Coplien [Cop05] caracteriza um padrão de projeto eficaz da seguinte maneira:

- *Ele soluciona um problema*: os padrões apreendem soluções, não apenas estratégias ou princípios abstratos.
- *Ele é um conceito comprovado*: os padrões apreendem soluções com um histórico, não teorias ou especulação.
- *Uma solução não é óbvia*: muitas técnicas para resolução de problemas (como paradigmas ou métodos de projeto de software) tentam obter soluções com base nos primeiros princípios. Os melhores padrões *geram* uma solução para um problema indiretamente – uma abordagem necessária para os problemas de projeto mais difíceis.
- *Ele descreve uma relação*: os padrões não apenas descrevem módulos, como também estruturas e mecanismos de sistema mais profundos.

¹ Existem discussões mais antigas sobre padrões de software, porém esses dois clássicos foram os primeiros tratados coesos sobre o assunto.

- O padrão possui um componente humano significativo (*minimizar a intervenção humana*). Todo software visa a atender o conforto humano ou a qualidade de vida; os melhores padrões apelam explicitamente à estética e à utilidade.

Um padrão de projeto evita que tenhamos de “reinventar a roda” ou, pior ainda, inventar uma “nova roda” que não será perfeitamente redonda; será muito pequena para o uso pretendido e muito estreita para o terreno onde irá rodar. Os padrões de projeto, se usados de maneira eficiente, invariavelmente o tornarão um melhor projetista de software.

16.1.1 Tipos de padrões

Uma das razões para os engenheiros de software se interessarem (e ficarem intrigados) por padrões de projeto é o fato de os seres humanos serem inerentemente bons no reconhecimento de padrões. Se não fossemos, teríamos parado no tempo e no espaço – incapazes de aprender com experiências passadas, desinteressados em nos aventurarmos devido à nossa incapacidade de reconhecer situações que talvez nos levassem a correr altos riscos, transtornados por um mundo que parece não ter regularidade ou consistência lógica. Felizmente, nada disso acontece porque efetivamente reconhecemos padrões em praticamente todos os aspectos de nossas vidas.

No mundo real, os padrões que reconhecemos são aprendidos ao longo de toda uma vida de experiências. Reconhecemos instantaneamente e compreendemos inerentemente seus significados e como eles poderiam ser usados. Alguns desses padrões nos dão uma melhor visão do fenômeno da recorrência. Por exemplo, você está voltando do trabalho para casa na Marginal, quando seu sistema de navegação (ou o rádio do carro) informa que um grave acidente ocorreu na Marginal no sentido oposto. Você se encontra a 6 quilômetros do acidente, porém já começa a perceber tráfego lento, reconhecendo um padrão que chamaremos **RubberNecking (olhar com curiosidade)**. Os motoristas deslocando-se na pista expressa em sua direção estão diminuindo de velocidade, para ter uma melhor visão do que aconteceu no sentido contrário. O padrão **RubberNecking** produz resultados notavelmente previsíveis (um congestionamento), mas nada mais faz do que descrever um fenômeno. No jargão dos padrões, ele poderia ser denominado padrão *não generativo*, pois descreve um contexto e um problema, mas não fornece nenhuma solução explícita.

Quando são considerados padrões de projeto de software, faz-se um esforço para identificar e documentar padrões *generativos*. Ou seja, identificamos um padrão que descreve um aspecto importante e repetível de um sistema e que nos dá uma maneira de construir esse aspecto em um sistema de forças que são únicas em determinado contexto. Em um ambiente ideal, um conjunto de padrões de projeto generativos poderia ser usado para “gerar” uma aplicação ou sistema computacional cuja arquitetura permitisse que se adaptasse à mudança. Algumas vezes chamada *generatividade*, “a aplicação sucessiva de vários padrões, cada um deles encapsulando seu próprio problema e forças, desdobra-se em uma solução mais ampla que emerge indiretamente como resultado das soluções menores” [App00].

Um padrão “generativo” descreve o problema, um contexto e forças, mas também descreve uma solução pragmática para o problema.

Não imponha um padrão, mesmo que ele atenda ao problema em questão. Se o contexto e as forças estiverem errados, procure outro padrão.

Existe uma maneira de classificar tipos de padrão?

Os padrões de projeto abrangem um amplo espectro de abstração e aplicação. Os *padrões de arquitetura* descrevem problemas de projeto de caráter amplo e diverso, resolvidos usando-se uma abordagem estrutural. Os *padrões de dados* descrevem problemas orientados a dados recorrentes e as soluções de modelagem de dados que podem ser usadas para resolvê-los. Os *padrões de componentes* (também conhecidos como *padrões de projeto*) tratam de problemas associados ao desenvolvimento de subsistemas e componentes, da maneira pela qual eles se comunicam entre si e de seu posicionamento em uma arquitetura maior. Os *padrões de projeto para interfaces* descrevem problemas comuns de interfaces do usuário e suas soluções, com um sistema de forças que inclui as características específicas dos usuários. Os *padrões para WebApp* tratam de um conjunto de problemas encontrados ao se construir WebApps e, em geral, incorporam muitas das demais categorias de padrões que acabamos de mencionar. Os *padrões móveis* descrevem os problemas comumente encontrados ao se desenvolver soluções para plataformas móveis. Em um nível de abstração mais baixo, os *idiomas* descrevem como implementar todos ou parte de um algoritmo específico ou a estrutura de dados para um componente de software no contexto de uma linguagem de programação específica.

Em seu livro seminal sobre padrões de projeto, Gamma e seus colegas² [Gam95] focalizam três tipos de padrões particularmente relevantes para projetos orientados a objetos: padrões criacionais, padrões estruturais e padrões comportamentais.

Os *padrões criacionais* se concentram na “criação, composição e representação” de objetos e dispõem de mecanismos que facilitam a instanciação de objetos em um sistema e que impõem “restrições sobre o tipo e número de objetos que podem ser criados em um sistema” [Maa07]. Os *padrões estruturais* focalizam problemas e soluções associadas a como classes e objetos são organizados e integrados para construir uma estrutura maior. Os *padrões comportamentais* tratam de problemas associados à atribuição de responsabilidade entre objetos e a maneira como a comunicação é realizada entre objetos.

INFORMAÇÕES



Padrões criacionais, estruturais e comportamentais

Foi proposta uma ampla variedade de padrões de projeto que se encaixam nas categorias criacional, estrutural e comportamental, e eles podem ser encontrados na Web. A Wikipedia (http://en.wikipedia.org/wiki/Software_design_pattern) menciona a seguinte amostra:

Padrões criacionais

- **Padrão de fábrica abstrata (*abstract factory pattern*):** centraliza a decisão de que fábrica instanciar.

- **Padrão de métodos de fábrica (*factory method pattern*):** centraliza a criação de um objeto de um tipo específico escolhendo uma de várias implementações.
- **Padrão construtor (*builder pattern*):** separa a construção de um objeto complexo de sua representação de modo que o mesmo processo de construção possa criar diferentes representações.

Padrões estruturais

- **Padrão adaptador (*adapter pattern*):** “adapta” uma interface de uma classe para uma que um cliente espera.

² Gamma e seus colegas [Gam95] são normalmente conhecidos como a “Gang of Four” (GoF) na literatura sobre padrões.

- **Padrão de agregação (*aggregate pattern*)**: uma versão do padrão de composição com métodos para agregação de objetos filhos.
- **Padrão de composição (*composite pattern*)**: uma estrutura de objetos em forma de árvore em que cada objeto possui a mesma interface.
- **Padrão container (*container pattern*)**: cria objetos com o propósito exclusivo de conter outros objetos e gerenciá-los.
- **Padrão proxy (*proxy pattern*)**: uma classe atuando como uma interface para outra.
- **Tubos e filtros (*pipes and filters*)**: uma cadeia de processos em que a saída de cada processo é a entrada do seguinte.
- **Padrão de comandos (*command pattern*)**: objetos de comando encapsulam uma ação e seus parâmetros.
- **Padrão iterador (*iterator pattern*)**: os iteradores são usados para acessar sequencialmente os elementos de um objeto agregado sem expor sua representação subjacente.
- **Padrão mediador (*mediator pattern*)**: fornece uma interface unificada para um conjunto de interfaces em um subsistema.
- **Padrão visitante (*visitor pattern*)**: uma forma de separar um algoritmo de um objeto.
- **Padrão visitante hierárquico (*hierarchical visitor pattern*)**: fornece uma forma de visitar todos os nós em uma estrutura de dados hierárquica como, por exemplo, uma árvore.

Padrões comportamentais

- **Padrão de cadeia de responsabilidades (*chain of responsibility pattern*)**: objetos e comandos são manipulados ou passados para outros objetos por meio de objetos de processamento com lógica.

Descrições detalhadas de cada um desses padrões podem ser obtidas via links em www.wikipedia.org.

16.1.2 Frameworks

Os próprios padrões talvez não sejam suficientes para desenvolver um projeto completo. Em alguns casos pode ser necessário fornecer uma infraestrutura mínima específica para implementação, chamada *framework*. Podemos selecionar uma “*mini-arquitetura reutilizável* que fornece a estrutura genérica e o comportamento para uma família de abstrações de software, juntamente com um contexto... que especifica sua colaboração e uso em um domínio de dados” [Amb98].

Esse framework não é um padrão de arquitetura, mas sim um esqueleto com um conjunto de “pontos de conexão” (também chamados *ganchos* e *encaixes*) que permitem que esse esqueleto seja adaptado a um domínio de problemas específico.* Os pontos de conexão possibilitam que integremos ao esqueleto classes ou funcionalidades específicas de um problema. Em um contexto orientado a objetos, um framework é um conjunto de classes que cooperam entre si.

Gamma e seus colegas [Gam95] descrevem as diferenças entre padrões de projeto e frameworks de uso de padrões da seguinte maneira:

1. *Os padrões de projeto são mais abstratos do que os frameworks.* Os frameworks podem ser incorporados ao código, mas apenas *exemplos* de padrões podem ser incorporados ao código. Um ponto forte dos frameworks de uso de padrões é que podem ser escritos em linguagens de programação e não só estudados, mas executados e reutilizados diretamente...
2. *Os padrões de projeto são elementos arquiteturais menores do que os frameworks.* Esses frameworks podem conter vários padrões de projeto, mas a recíproca jamais é verdadeira.

Framework é uma “mini-arquitetura” reutilizável que serve como base para e a partir da qual outros padrões de projeto podem ser aplicados.

* N. de R.T.: Um framework inclui aspectos estáticos/estruturais e dinâmicos/comportamentais que dão suporte ao uso dos padrões de projeto.

3. *Os padrões de projeto são menos especializados do que os frameworks.* Os frameworks apresentam um domínio de aplicação particular. Os padrões de projeto podem ser usados em praticamente qualquer tipo de aplicação. Embora sejam possíveis padrões de projeto mais especializados, até mesmo esses não ditiariam uma arquitetura de aplicação.

Basicamente, o projetista de um framework de uso de padrão argumentará que uma miniarquitetura reutilizável se aplica a todo software a ser desenvolvido em um domínio de aplicação limitado. Para serem mais eficientes, esses frameworks são aplicados sem nenhuma alteração. Podem-se acrescentar outros elementos de projeto, mas apenas por meio de pontos de conexão que permitem ao projetista dar corpo ao esqueleto desses frameworks.

16.1.3 Descrição de padrões

O projeto baseado em padrões começa com o reconhecimento de padrões na aplicação que se pretende construir, continua com a pesquisa para determinar se outros trataram do padrão e termina com a aplicação de um padrão apropriado para o problema em questão. Em geral, a segunda dessas três tarefas é a mais difícil. Como encontrar padrões que atendam às nossas necessidades?

A resposta deve depender da comunicação efetiva do problema de que o padrão trata, do contexto em que o padrão reside, do sistema de forças que molda o contexto e da solução proposta. Para transmitir essas informações de forma inequívoca, é necessário um formulário ou modelo padronizado para descrições de padrão. Embora tenham sido propostos vários modelos de padrão distintos, quase todos contêm um subconjunto principal do conteúdo sugerido por Gamma e seus colegas [Gam95]. No quadro a seguir é mostrado um modelo de padrão simplificado.

"Os padrões são semiprontos – isso significa que sempre temos de finalizá-los e adaptá-los ao nosso ambiente."

Martin Fowler

INFORMAÇÕES



Modelo de padrões de projeto

Nome do padrão – descreve a essência do padrão em um nome curto, mas expressivo.

Problema – descreve o problema de que o padrão trata.

Motivação – dá um exemplo do problema.

Contexto – descreve o ambiente onde o problema reside, incluindo o domínio de aplicação.

Forças – enumera o sistema de forças que afetam a maneira como o problema deve ser resolvido; inclui uma discussão das limitações e restrições que devem ser consideradas.

Solução – fornece uma descrição detalhada de uma solução proposta para o problema.

Objetivo – descreve o padrão e o que ele faz.

Colaborações – descrevem como outros padrões contribuem para uma solução.

Consequências – descrevem os possíveis prós e contras que devem ser considerados quando o padrão é implementado e as consequências do uso do padrão.

Implementação – identifica questões especiais que devem ser consideradas ao se implementar o padrão.

Usos conhecidos – dá exemplos de usos práticos do padrão de projeto em aplicações reais.

Padrões relacionados – remetem a padrões de projeto relacionados.

Os nomes de padrões de projeto devem ser escolhidos com cuidado. Um dos principais problemas técnicos em projeto baseado em padrões é a incapacidade de encontrar um nome adequado para um padrão.

cidade de encontrar padrões existentes quando há centenas ou milhares de candidatos. A busca pelo padrão “correto” é tremendamente facilitada por um nome de padrão significativo.

Um modelo de padrões fornece um meio padronizado para descrição de padrões de projeto. Cada uma das entradas do modelo representa características do padrão de projeto que podem ser pesquisadas (por exemplo, por um banco de dados) e a maneira como o padrão apropriado pode ser encontrado.

16.1.4 Linguagens e repositórios de padrões

Quando usamos o termo *linguagem*, a primeira coisa que nos vêm à mente é uma linguagem natural (por exemplo, inglês, espanhol, chinês) ou uma linguagem de programação (por exemplo, C++, Java). Em ambos os casos, a linguagem possui uma sintaxe e semântica usadas para transmitir ideias ou instruções procedurais de forma eficiente.

Quando se usa o termo *linguagem* no contexto de padrões de projeto, ele assume um significado ligeiramente diferente. Uma *linguagem de padrões* engloba um conjunto de padrões, cada qual descrito por meio do uso de um modelo de padrões (Seção 16.1.3) e inter-relacionados para mostrar como esses padrões colaboram para solucionar problemas em um domínio de aplicação.³

Em uma linguagem natural, as palavras são organizadas em sentenças que transmitem significado. A estrutura das sentenças é descrita pela sintaxe da linguagem. Em uma linguagem de padrões, os padrões de projeto são organizados para fornecer um “método estruturado para descrição de práticas de projeto adequadas em um domínio”.⁴

De certa maneira, uma linguagem de padrões é análoga a um manual de instruções em hipertexto para resolução de problemas em áreas de aplicação específicas. O domínio do problema é descrito primeiro em termos hierárquicos, começando com problemas de projeto abrangentes associados ao domínio e então refinando-se cada um dos problemas abrangentes em níveis de abstração menores. No contexto de software, problemas de projeto abrangentes tendem a ser problemas de arquitetura por natureza e tratam da estrutura geral da aplicação e dos dados ou conteúdo que o atendem. Os problemas de arquitetura são refinados para níveis de abstração menores, levando a padrões de projeto que resolvem subproblemas e colaboram entre si no nível de componentes (ou classes). Em vez de uma lista sequencial de padrões, a linguagem de padrões representa um conjunto interconectado em que o usuário pode partir de um problema de projeto abrangente e “ir fundo” para revelar problemas específicos e suas soluções.

Uma extensa lista de linguagens de padrões foi proposta para projeto de software [Hil13], contendo indicações para padrões de projeto que fazem parte de linguagens de padrões em repositórios de padrões acessíveis pela Web. O re-

Caso não consiga encontrar uma linguagem de padrões que trate do domínio de seu problema, procure analogias em outro conjunto de padrões.

Para uma lista de linguagens de padrões úteis, consulte c2.com/ppr/titles.html. Informações adicionais podem ser obtidas em hillside.net/patterns/.

³ Christopher Alexander propôs originalmente linguagens de padrões para arquitetura e planejamento urbano. Hoje, linguagens de padrões foram desenvolvidas para diversos setores, das ciências sociais ao processo de engenharia de software.

⁴ Essa descrição da Wikipedia pode ser encontrada em http://en.wikipedia.org/wiki/Pattern_language.

positório fornece um índice de todos os padrões de projeto e contém links de hiper-mídia que permitem ao usuário compreender as colaborações entre padrões.

16.2 Projeto de software baseado em padrões

Os melhores projetistas de qualquer área têm uma habilidade extraordinária de vislumbrar padrões que caracterizam um problema e padrões correspondentes que podem ser combinados para criar a solução. Ao longo do processo de projeto, devemos buscar toda oportunidade de aplicar padrões de projeto existentes (quando atenderem às necessidades do projeto), em vez de criar novos.

16.2.1 Contexto do projeto baseado em padrões

O projeto baseado em padrões não é utilizado isoladamente. Os conceitos e as técnicas discutidas para projeto da arquitetura, de componentes e para interfaces do usuário (Capítulos 13 a 15) são usados em conjunto com uma abordagem baseada em padrões.

No Capítulo 12, citamos que um conjunto de diretrizes e atributos de qualidade serve como base para todas as decisões de projeto de software. As próprias decisões são influenciadas por um conjunto de conceitos de projeto fundamentais (por exemplo, separação de preocupações, refinamento gradual, independência funcional) que são obtidos usando-se heurísticas que evoluíram ao longo de várias décadas e práticas melhores (por exemplo, técnicas, notação de modelagem) propostas para fazer com que o projeto seja mais fácil de ser realizado e mais efetivo como base para a construção.

O papel do projeto baseado em padrões está ilustrado na Figura 16.1. Um projetista de software inicia com um modelo de requisitos (seja ele explícito, seja implícito) que apresenta uma representação abstrata do sistema. O modelo de requisitos descreve o conjunto de problemas, estabelece o contexto e identifica o sistema de forças que exerce domínio. Talvez sugira o projeto de maneira abstrata, mas o modelo de requisitos faz pouco para representar o projeto explicitamente.

Ao iniciar seu trabalho como projetista, é sempre importante manter os atributos de qualidade (Capítulo 12) em mente. Esses atributos estabelecem uma maneira de avaliar a qualidade do software, mas pouco faz para ajudar a obtê-lo. Consequentemente, devemos aplicar técnicas comprovadas para traduzir as abstrações contidas no modelo de requisitos dessa maneira mais concreta que é o projeto de software. Para tanto, usaremos os métodos e as ferramentas de modelagem disponíveis para projeto da arquitetura, de componentes e para interfaces. Mas apenas quando nos depararmos com um problema, um contexto e um sistema de forças que ainda não foram resolvidos anteriormente. Se já existir uma solução, utilize-a! E isso significa aplicar uma abordagem de projeto baseado em padrões.

16.2.2 Pense em termos de padrões

O projeto baseado em padrões implica uma “nova maneira de pensar” [Sha05] que começa considerando o contexto – o quadro geral. Quando o

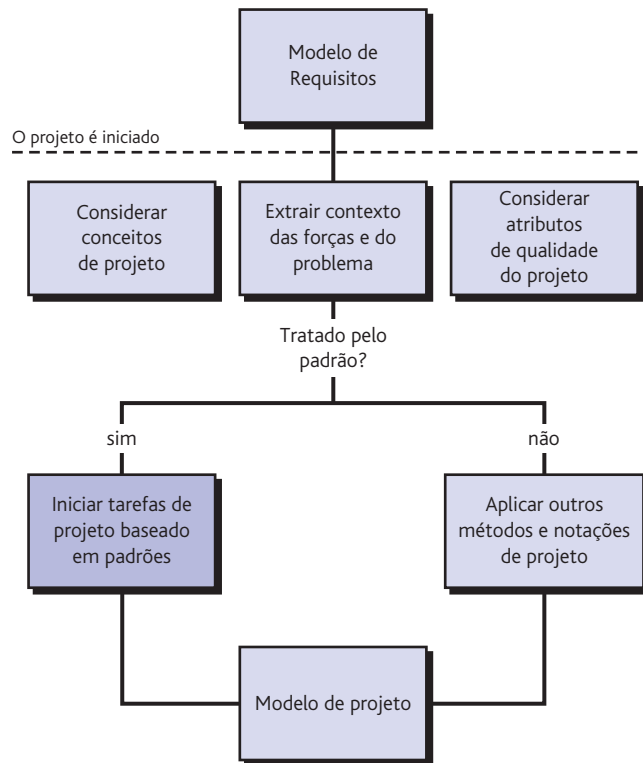


FIGURA 16.1 Contexto do projeto baseado em padrões.

contexto é avaliado, extraímos uma hierarquia de problemas que devem ser resolvidos. Alguns desses problemas serão de natureza global, enquanto outros tratarão de características e funções específicas do software. Todos serão afetados por um sistema de forças que vai influenciar a natureza de uma solução proposta.

Shalloway e Trott [Sha05] sugerem a seguinte⁵ abordagem que permite a um projetista pensar em termos de padrões:

1. Certifique-se de ter entendido o quadro geral – o contexto em que o software a ser construído reside. O modelo de requisitos deve comunicar isso a você.
2. Examine o quadro geral, extraia os padrões presentes nesse nível de abstração.
3. Inicie seu projeto com os padrões do “quadro geral” que estabeleçam um contexto ou esqueleto para trabalho de projeto posterior.
4. “Trabalhe em direção à essência, partindo do contexto” [Sha05], buscando padrões em níveis de abstração mais baixos que contribuam para a solução do projeto.
5. Repita os passos 1 a 4 até que o projeto completo ganhe corpo.
6. Refine o projeto, adaptando cada padrão às especificidades do software que está tentando construir.

O projeto baseado em padrões parece interessante para o problema que tenho de resolver. Por onde devo começar?

⁵ Baseada no trabalho de Christopher Alexander [Ale79].

É importante notar que os padrões não são entidades independentes. Os padrões de projeto que estão presentes em nível de abstração elevado influenciarão invariavelmente a maneira pela qual outros padrões serão aplicados em níveis de abstração mais baixos. Além disso, os padrões em geral colaboram entre si. A implicação: ao se selecionar um padrão de arquitetura, ele poderá influenciar os padrões de projeto escolhidos no nível de componentes. Da mesma forma, ao selecionar um padrão de projeto específico para interfaces, muitas vezes você se vê forçado a usar outros padrões que colaboram com ele.

Para ilustrar, consideremos a WebApp **CasaSeguraGarantida.com**. Se considerarmos o quadro geral, a WebApp deve tratar de como fornecer informações sobre os produtos e serviços do *CasaSegura*, como vender aos clientes os produtos e serviços *CasaSegura* e como estabelecer o monitoramento e o controle baseados na Internet de um sistema de segurança instalado. Cada um desses problemas fundamentais pode ser refinado ainda mais em um conjunto de subproblemas.

Por exemplo, *Como vender via Internet* implica um padrão **E-commerce (de comércio eletrônico)** que por si só implica um grande número de padrões em níveis de abstração mais baixos. O padrão **E-commerce** (provavelmente um padrão de arquitetura) implica mecanismos para configurar uma conta de cliente, exibir os produtos a serem vendidos, selecionar produtos para compra e assim por diante. Portanto, se pensarmos em padrões, é importante determinar se um padrão para configurar uma conta existe ou não. Se **Configurar-Conta** estiver disponível como um padrão viável para o contexto do problema, talvez ele colabore com outros padrões como **CriarFormulárioDeEntrada**, **GerenciarPreenchimentoDeFormulários** e **ValidarPreenchimentoFormulários**. Cada um desses padrões delinea problemas a ser resolvidos e soluções que possam ser aplicadas.

16.2.3 Tarefas de projeto

As tarefas de projeto a seguir são aplicadas quando se adota uma filosofia de projeto baseado em padrões:

1. **Examine o modelo de requisitos e desenvolva uma hierarquia de problemas.** Descreva cada problema e subproblema isolando o problema, o contexto e o sistema de forças que se aplicam. Trabalhe inicialmente em problemas mais abrangentes (nível de abstração elevado), indo depois para subproblemas menores (em níveis de abstração mais baixos).
2. **Determine se uma linguagem de padrões confiável foi desenvolvida para o domínio do problema.** Conforme observado na Seção 16.1.4, uma linguagem de padrões trata de problemas associados a um domínio de aplicação específico. A equipe de software do *CasaSegura* procuraria uma linguagem de padrões desenvolvida especificamente para produtos de segurança domiciliar. Se esse nível de especificidade de linguagem de padrões não pudesse ser encontrado, a equipe subdividiria o problema de software *CasaSegura* em uma série de domínios de problemas genéricos (por exemplo, problemas de monitoramento de dispositivos digitais, problemas de interfaces do usuário, problemas

de gerenciamento de vídeo digital) e buscaria linguagens de padrões apropriadas.

3. **Iniciando com um problema abrangente, determine se um ou mais padrões de arquitetura se encontram disponíveis para ele.** Se existir um padrão de arquitetura, certifique-se de examinar todos os padrões colaboradores. Se o padrão for apropriado, adapte a solução de projeto proposta e construa um elemento de modelo de projeto que o represente adequadamente. Por exemplo, um problema abrangente para a WebApp **Casa-SeguraGarantida.com** é tratado com um padrão **E-commerce** (de comércio eletrônico) (Seção 16.2.2). Esse sugerirá uma arquitetura específica para lidar com os requisitos de comércio eletrônico.
4. **Use as colaborações fornecidas para o padrão de arquitetura, examine problemas de subsistemas ou de componentes e procure padrões apropriados para tratar deles.** Talvez seja necessário pesquisar outros repositórios de padrões, bem como a lista de padrões que corresponde à solução de arquitetura. Se for encontrado um padrão apropriado, adapte a solução de projeto proposta e construa um elemento de modelo de projeto que o represente adequadamente. Certifique-se de aplicar o passo 7.
5. **Repita os passos 2 a 4 até que todos os problemas mais amplos tenham sido tratados.** A implicação é começar com o quadro geral e elaborar para resolver problemas em níveis cada vez mais detalhados.
6. **Se os problemas de projeto para interfaces do usuário tiverem sido isolados (quase sempre esse é o caso), pesquise os vários repositórios de padrões de projeto para interfaces do usuário em busca de padrões apropriados.** Prossiga de maneira similar para os passos 3, 4 e 5.
7. **Independentemente de seu nível de abstração, se uma linguagem de padrões e/ou repositório de padrões ou padrão individual se mostrar promissor, compare o problema a ser resolvido em relação ao(s) padrão(ões) existente(s) apresentado(s).** Certifique-se de examinar o contexto e as forças para garantir que o padrão forneça, de fato, uma solução suscetível para o problema.
8. **Certifique-se de refinar o projeto à medida que é obtido de padrões usando critérios de qualidade de projeto como guia.**

Embora essa abordagem de projeto seja *top-down*, as soluções de projeto na vida real são, algumas vezes, mais complexas. Gillis [Gil06] tece comentários sobre isso ao escrever:

Os padrões de projeto na engenharia de software destinam-se ao uso racional e dedutivo. Portanto, temos o problema ou requisito geral, X, o padrão de projeto Y resolve X, consequentemente usamos Y. Agora, ao refletir sobre meu próprio processo – e tenho razões para acreditar que não sou o único a pensar assim –, descubro que é mais orgânico que isso, mais indutivo do que dedutivo, mais *bottom-up* do que *top-down*.

Além disso, a abordagem baseada em padrões deve ser usada em conjunto com outros conceitos e técnicas de projeto de software.

	Banco de dados	Aplicação	Implementação	Infraestrutura
Dados/Conteúdo				
Enunciado do problema...	Nome(s)do(s)Padrão(ões)		Nome(s)do(s)Padrão(ões)	
Enunciado do problema...		Nome(s)do(s)Padrão(ões)		Nome(s)do(s)Padrão(ões)
Enunciado do problema...	Nome(s)do(s)Padrão(ões)			Nome(s)do(s)Padrão(ões)
Interface do usuário				
Enunciado do problema...		Nome(s)do(s)Padrão(ões)		
Enunciado do problema...		Nome(s)do(s)Padrão(ões)		Nome(s)do(s)Padrão(ões)
Enunciado do problema...				
Componentes				
Enunciado do problema...		Nome(s)do(s)Padrão(ões)	Nome(s)do(s)Padrão(ões)	
Enunciado do problema...				Nome(s)do(s)Padrão(ões)
Enunciado do problema...		Nome(s)do(s)Padrão(ões)	Nome(s)do(s)Padrão(ões)	
Arquitetura				
Enunciado do problema...		Nome(s)do(s)Padrão(ões)	Nome(s)do(s)Padrão(ões)	
Enunciado do problema...		Nome(s)do(s)Padrão(ões)	Nome(s)do(s)Padrão(ões)	
Enunciado do problema...		Nome(s)do(s)Padrão(ões)	Nome(s)do(s)Padrão(ões)	

FIGURA 16.2 Uma tabela para organização de padrões.

Fonte: Adaptado de [Mic04].

16.2.4 Construção de uma tabela para organização de padrões

À medida que o projeto baseado em padrões prossegue, talvez encontremos problemas para organizar e classificar prováveis padrões contidos em vários repositórios e linguagens de padrões. Para auxiliar a organizar nossa avaliação de prováveis padrões, a Microsoft [Mic13] sugere a criação de uma *tabela para organização de padrões* que assume a forma geral indicada na Figura 16.2.

Uma tabela para organização de padrões pode ser implementada como um modelo de planilha, usando o formato mostrado na figura. Uma lista resumida dos enunciados dos problemas, organizados por tópicos como dados/conteúdo, arquitetura, componentes e interfaces do usuário, é apresentada na coluna mais à esquerda (sombra mais escura). Quatro tipos padrão – bancos de dados, aplicação, implementação e infraestrutura – são listados ao longo da linha superior. Os nomes de prováveis padrões são indicados nas células da tabela.

Para fornecermos entradas para a tabela organizadora, pesquisaremos várias linguagens e repositórios de padrões em busca de padrões que atendam a determinado enunciado de problema. Quando são encontrados um ou mais prováveis padrões, esses são introduzidos na linha correspondente ao enunciado do problema e na coluna correspondente ao tipo de padrão. O nome do padrão é introduzido como um hiperlink para o URL do endereço Web que contém uma descrição completa do padrão.

16.2.5 Erros comuns de projeto

Uma série de erros comuns ocorre quando se usa projeto baseado em padrões. Em alguns casos, não se dedicou tempo suficiente para entender o problema subjacente e seu contexto e forças e, como consequência, escolhe-se um padrão que parece correto, mas que, na verdade, é inadequado para a solução exigida. Assim que o padrão incorreto é escolhido, recusamo-nos a admitir o erro e forçamos a adequação do padrão. Em outros casos, o problema possui forças não consideradas pelo padrão escolhido, resultando em uma adequação deficiente ou errônea. Algumas vezes um padrão é aplicado de forma demasiadamente literal, e as adaptações necessárias para o espaço do problema não são implementadas.

Esses erros poderiam ser evitados? Na maioria dos casos a resposta é “sim”. Todo bom projetista pede uma segunda opinião e aceita revisões em seu trabalho. As técnicas de revisão discutidas no Capítulo 20 podem ajudar a garantir que o projeto baseado em padrões que desenvolvemos resulte em uma solução de alta qualidade para o problema de software a ser resolvido.

Não imponha um padrão, mesmo que ele atenda ao problema em questão. Se o contexto e as forças estiverem errados, procure outro padrão.

16.3 Padrões de arquitetura

Se um engenheiro civil decide construir uma casa colonial com hall central, existe um único estilo arquitetônico a ser aplicado. Os detalhes do estilo (por exemplo, número de lareiras, fachada da casa, posição das portas e janelas) podem variar consideravelmente, mas, uma vez tomada a decisão sobre a arquitetura geral da casa, o estilo é imposto sobre o projeto.⁶

Os padrões de arquitetura são um pouco diferentes. Por exemplo, toda casa (e todo estilo arquitetônico para casas) usa um padrão **Kitchen** (cozinha). O padrão **Kitchen** e padrões com os quais ele colabora atendem a problemas associados ao armazenamento e à preparação de alimentos, aos utensílios necessários para cumprir essas tarefas e às regras para a colocação dos utensílios em relação ao fluxo no ambiente. Além disso, o padrão poderia atender a problemas associados a balcões, iluminação, interruptores, uma ilha central, pisos e assim por diante. Obviamente, há mais de um projeto para uma cozinha, em geral ditado pelo contexto e pelo sistema de forças. Mas todo projeto pode ser concebido no contexto da “solução” sugerida pelo padrão **Kitchen**.

Antes que um padrão de arquitetura possa ser escolhido em um domínio específico, ele deve ser avaliado em termos de sua adequação para a aplicação e estilo de arquitetura geral, bem como o contexto e o sistema de forças que ele especifica.

Uma arquitetura de software pode ter uma série de padrões de arquitetura que tratam de questões como concorrência, persistência e distribuição.

⁶ Isso significa que haverá um hall central e um corredor, que as salas serão colocadas à esquerda e à direita do hall, que a casa terá dois (ou mais) pisos, que os quartos da casa estarão no piso de cima e assim por diante. Essas “regras” são impostas, uma vez tomada a decisão de adotar o estilo colonial com hall central.

INFORMAÇÕES

**Repositórios de padrões de projeto**

Existem várias fontes de padrões de projeto disponíveis na Web. Alguns padrões podem ser obtidos de linguagens de padrões publicadas individualmente, enquanto outros se encontram disponíveis como parte de um portal ou repositório de padrões. Vale a pena dar uma olhada nas seguintes fontes na Web:

Hillside.net

<http://hillside.net/patterns/> Um dos conjuntos mais completos da Web em termos de padrões e linguagens de padrões.

Portland Pattern Repository

<http://c2.com/ppr/index.html> Contém links para uma ampla variedade de recursos e coleções de padrões.

Pattern Index

<http://c2.com/cgi/wiki?PatternIndex> Uma "coleção eclética de padrões".

Handbook of Software Architecture

http://researcher.watson.ibm.com/researcher/view_project.php?id=3206/ Referência bibliográfica com centenas de padrões de projeto de arquitetura e componentes.

Coleções de padrões para interfaces do usuário

Padrões para interfaces do usuário/interfaces

homem-máquina

<http://www.hcipatterns.org/patterns/borchers/patternIndex.html>

Padrões para interfaces do usuário de Jennifer Tidwell
<http://designinginterfaces.com/patterns/>

Padrões de projeto para interfaces do usuário móveis
<http://profs.info.uaic.ro/~evalica/patterns/>

Linguagem de padrões para projeto de interfaces do usuário
www.maplefish.com/todd/papers/Experiences.html

Biblioteca de projeto interativo para jogos
<http://www.eelke.com/files/pubs/usabilitypatternsingames.pdf>

Padrões de projeto para interfaces do usuário
www.cs.helsinki.fi/u/salaakso/patterns/

Padrões de projeto especializados:

Aviônica

<http://g.oswego.edu/dl/acs/acs/acs.html>

Sistemas de informações de negócios
www.objectarchitects.de/arcus/cookbook/

Processamento distribuído

www.cs.wustl.edu/~schmidt/

Padrões IBM para e-Business

www-128.ibm.com/developerworks/patterns/

Yahoo! Biblioteca de padrões de projeto

<http://developer.yahoo.com/ypatterns/>

WebPatterns.org

<http://www.welie.com/index.php>

16.4 Padrões de projeto de componentes

Os padrões de projeto de componentes nos dão soluções comprovadas que tratam de um ou mais subproblemas extraídos do modelo de requisitos. Em muitos casos, os padrões de projeto desse tipo se concentram em algum elemento funcional de um sistema. Por exemplo, a aplicação **CasaSeguraGarantida.com** deve tratar do seguinte subproblema de projeto: *Como podemos obter especificações de produtos e informações relacionadas para qualquer dispositivo do CasaSegura?*

Tendo enunciado o subproblema que deve ser resolvido, devemos considerar agora o contexto e o sistema de forças que afetam a solução. Examinando-se o caso de uso do modelo de requisitos apropriado, constataremos que o consumidor usa a especificação para um dispositivo do *CasaSegura* (por exemplo, um sensor ou câmera de segurança) para fins de informação. Entretanto, outras informações relacionadas à especificação (por exemplo, determinação de preços) poderiam ser empregadas quando a funcionalidade de comércio eletrônico fosse selecionada.

A solução para o subproblema envolve uma pesquisa. Como pesquisar é um problema muito comum, não deve ser nenhuma surpresa a existência

de muitos padrões relacionados à pesquisa. Pesquisando uma série de repositórios de padrões, descobriremos os seguintes padrões, juntamente com o problema que cada um resolve:

AdvancedSearch. Os usuários precisam encontrar um item específico em um grande conjunto de itens.

HelpWizard. Os usuários precisam de ajuda sobre certo tópico relativo ao site ou quando precisam encontrar uma página específica dentro desse site.

SearchArea. Os usuários precisam encontrar uma página Web.

SearchTips. Os usuários precisam saber como controlar o mecanismo de busca.

SearchResults. Os usuários têm de processar uma lista de resultados de uma busca.

SearchBox. Os usuários têm de encontrar um item ou informações específicas.

Para **CasaSeguraGarantida.com**, o número de produtos não é particularmente grande, e cada um deles possui uma classificação relativamente simples, de modo que **AdvancedSearch** e **HelpWizard** provavelmente não sejam necessários. Similarmente, a busca é relativamente simples para não precisar de **SearchTips**. A descrição de **SearchBox**, entretanto, é dada (em parte) como:

Search Box

(Adaptado de www.welie.com/patterns/showPattern.php?patternID=search)

Problema:	Os usuários precisam encontrar um item ou informações específicos.
Motivação:	Qualquer situação em que uma busca com palavra-chave é aplicada em um conjunto de objetos de conteúdo organizados na forma de páginas Web.
Contexto:	Em vez de usar navegação para obter informações ou conteúdo, o usuário quer fazer uma busca direta em conteúdo contido em várias páginas Web. Qualquer site que já possui recursos primários de navegação. O usuário talvez queira procurar um item numa categoria. Ele talvez queira especificar uma consulta de forma mais detalhada.
Forças:	O site já possui recursos primários de navegação. Os usuários talvez queiram procurar um item de determinada categoria. Eles talvez queiram especificar uma consulta de forma mais detalhada, por meio de operadores booleanos simples.
Solução:	Oferecer funcionalidade de busca formada por um identificador de busca, um campo para palavra-chave, um filtro (se aplicável) e um botão "avançar". Pressionar a tecla Enter tem a mesma função que selecionar o botão Avançar. Também fornece Dicas de Busca e exemplos em uma página separada. Um link para essa página é colocado próximo à funcionalidade de busca. A caixa de edição para o termo de pesquisa é suficientemente grande para acomodar três consultas de usuário típicas (normalmente, por volta de 20 caracteres). Se o número de filtros for maior que 2, use uma caixa de combinação para seleção de filtros ou um botão de seleção. Os resultados da pesquisa são apresentados em uma nova página, com um identificador claro contendo pelo menos "Resultados da busca" ou algo similar. A função de busca é repetida na parte superior da página, com as palavras-chave anteriormente introduzidas, de modo que os usuários saberão quais eram elas.

A descrição de padrões continua com outras entradas conforme descrito na Seção 16.1.3.

O padrão prossegue para descrever como os resultados da busca são acessados, apresentados, correspondidos e assim por diante. Com base nisso, a equipe do **CasaSeguraGarantida.com** pode projetar os componentes necessários para implementar a busca ou (mais provavelmente) obter componentes reutilizáveis existentes.

CASASEGURA



Aplicação de padrões

Cena: Discussão informal durante o projeto de um incremento do software que implementa controle de sensores via Internet para o **CasaSeguraGarantida.com**.

Atores: Jamie (responsável pelo projeto) e Vinod (arquiteto-chefe do sistema **CasaSeguraGarantida.com**).

Conversa:

Vinod: Então, como está indo o projeto da interface para controle de câmeras?

Jamie: Nada mal. Já projetei a maior parte da capacidade de conexão com os verdadeiros sensores sem grandes problemas. Já comecei também a pensar sobre a interface para os usuários poderem efetivamente movimentar, deslocar e ampliar as câmeras de uma página Web remota, porém ainda não estou certo de tê-la feito corretamente.

Vinod: Qual sua ideia?

Jamie: Bem, entre os requisitos, o controle de câmeras precisa ser altamente interativo – à medida que o usuário movimenta o controle, a câmera deve se movimentar o quanto antes possível. Portanto, estava imaginando um conjunto de botões dispostos como uma câmera comum; porém, ao clicá-los, o usuário controlaria a câmera.

Vinod: Huumm. Sim, isso funcionaria, mas não estou certo de que esteja correto – para cada clique de um controle precisaríamos esperar que ocorresse toda a comunicação cliente/servidor e, portanto, não teríamos a sensação de uma resposta imediata.

Jamie: Foi isso que imaginei – é por isso que não estou muito satisfeito com a abordagem, mas não sei exatamente como poderia fazê-lo de outra forma.

Vinod: Bem, por que não usa simplesmente o padrão **InteractiveDeviceControl**?

Jamie: Hummm – o que é isso? Creio nunca ter ouvido falar dele.

Vinod: Basicamente, trata-se de um padrão destinado exatamente para o problema que você está me descrevendo. A solução que ele propõe é criar uma conexão de controle do servidor com o dispositivo, por meio da qual comandos de controle poderiam ser enviados. Dessa maneira, não seria preciso enviar solicitações HTTP normais. E o padrão ainda indicaria como implementar isso usando algumas técnicas AJAX simples. Temos alguns JavaScripts simples no cliente que se comunicam diretamente com o servidor e transmitem os comandos assim que o usuário fizer alguma coisa.

Jamie: Legal! Era exatamente isso que eu precisava para resolver essa questão. Onde posso encontrá-lo?

Vinod: Ele se encontra à disposição em um repositório online. Eis a URL.

Jamie: Vou verificar isso.

Vinod: Sim – mas lembre-se de verificar as consequências do padrão. Lembro de haver algo lá sobre tomar cuidado com questões de segurança. Acho que tem a ver com o fato de você criar um canal de controle separado e, portanto, contornar os mecanismos de segurança comuns da Web.

Jamie: Excelente observação. Provavelmente não teria atentado a esse fato! Obrigado.

16.5 Padrões de projeto para interfaces do usuário

Foram propostas centenas de padrões para interfaces do usuário (UI, em inglês) nos últimos anos. A maior parte deles cai em uma das dez categorias de padrões, conforme descrito por Tidwell [Tid02] e vanWelie [Wel01]. A seguir estão algumas categorias representativas (discutidas com um exemplo simples⁷):

⁷ Um modelo padrão sintetizado é usado aqui. Descrições de padrões completas (juntamente com dezenas de outros padrões) podem ser encontradas em [Tid02] e [Wel01].

Toda a Interface com o Usuário. Fornece orientação para estrutura de alto nível e navegação por toda a interface.

Padrão: *Top-level navigation*

Descrição: Usada quando um site ou uma aplicação implementa uma série de funções principais. Oferece um menu de alto nível, geralmente acoplado a um logotipo ou imagem identificadora, que possibilita a navegação direta para qualquer uma das principais funções do sistema.

Detalhes: Funções principais (em geral limitadas a quatro a sete nomes de função) são listadas na parte superior da tela (formatos de colunas verticais também são possíveis) em uma linha horizontal de texto. Cada nome fornece um link para uma fonte de informações ou função apropriada. Geralmente, usada com o padrão *bread crumbs* discutido mais à frente.

Elementos de navegação: Cada nome de função/conteúdo representa um link para a função ou conteúdo apropriado.

Layout da página. Trata da organização geral das páginas (para sites) ou exibições em tela distintas (para aplicações interativas).

Padrão: *Card stack*

Descrição: Usado quando uma série de subfunções ou categorias de conteúdo específicas relacionadas a um recurso ou função deve ser selecionada em ordem aleatória. Dá a aparência de uma pilha de fichas indexadoras, cada uma delas selecionável com um clique de mouse e cada qual representando subfunções ou categorias de conteúdo específicas.

Detalhes: As fichas indexadoras são uma metáfora bem compreendida e fácil para o usuário manipular. Cada ficha indexadora (separador) pode ter um formato ligeiramente diverso. Algumas poderão exigir entrada de dados e possuir botões ou outros mecanismos de navegação; outras podem ser informativos. Poderiam ser combinadas com outros padrões como *drop-down list*, *fill-in-the-blanks* e outros.

Elementos de navegação: Um clique de mouse em uma guia faz com que a ficha apropriada apareça. Os recursos de navegação contidos na ficha também poderiam estar presentes; porém, em geral, essas deveriam iniciar uma função relacionada aos dados da ficha e não provocar um link real para alguma outra tela.

Formulários e introdução de dados. Considera uma grande variedade de técnicas de projeto para realizar a introdução de dados em formulários.

Padrão: *Fill-in-the-blanks*

Descrição: Possibilita que dados alfanuméricos sejam introduzidos em uma “caixa de texto”.

Detalhes: Os dados poderiam ser introduzidos em uma caixa de texto. Em geral, são validados e processados após a seleção de algum indicador de texto ou gráfico (por exemplo, um botão contendo “avançar”, “subme-

ter”, “próximo”). Em muitos casos, esse padrão pode ser combinado com uma lista suspensa ou outros padrões (por exemplo, SEARCH <drop down list> FOR <caixa de texto fill-in-the-blanks>).

Elementos de navegação: Um indicador de texto ou gráfico que inicia a validação e o processamento.

Navegação. Ajuda o usuário na navegação por meio de menus hierárquicos, páginas Web e telas interativas.

Padrão: *Edit-in-place*

Descrição: Fornece um recurso de edição simples para certos tipos de conteúdo, no local em que é exibido. Nenhuma necessidade de o usuário introduzir explicitamente uma função ou modo de edição de texto.

Detalhes: O usuário vê na tela o conteúdo que deve ser alterado. Um clique duplo com o mouse sobre o conteúdo indica ao sistema que se deseja editar. O conteúdo é realçado para significar que o modo de edição está disponível e o usuário faz as mudanças apropriadas.

Elementos de navegação: Nenhum.

E-commerce. Específicos para sites, esses padrões implementam elementos recorrentes de aplicações para comércio eletrônico.

Padrão: *Shopping cart*

Descrição: Fornece uma lista de itens selecionados para compra.

Detalhes: Lista informações de itens, quantidade, código de produto, disponibilidade (disponível, esgotado), preço, informações para entrega, custos de remessa e outras informações de compra relevantes. Também oferece a capacidade de editar (por exemplo, remover, modificar quantidade).

Elementos de navegação: Contém a capacidade de prosseguir com a compra ou sair para encerrar as compras.

Cada um dos exemplos de padrões anteriores (e todos os padrões em cada categoria) também teria um projeto completo de componentes, incluindo classes de projeto, atributos, operações e interfaces.

Uma discussão completa sobre interfaces do usuário vai além dos objetivos deste livro. Caso tenha maior interesse, consulte [Yah13], [UXM10], [Gub09], [Duy02], [Tid02] e [Bor01] para mais informações.

16.6 Padrões de projeto para WebApps

Ao longo deste capítulo vimos que há diferentes tipos de padrões e várias maneiras distintas de classificá-los. Ao considerarmos os problemas de projeto a ser solucionados quando uma WebApp é construída, vale a pena examinar as categorias de padrão concentrando-nos em duas dimensões: o foco de projeto do padrão e seu nível de granularidade. O *foco do projeto* identifica qual aspecto do modelo de projeto é relevante (por exemplo, arquitetura das in-

formações, navegação, interação). A *granularidade* identifica o nível de abstração que está sendo considerado (por exemplo, o padrão se aplica a toda a WebApp ou a uma única página Web, a um subsistema ou a um componente WebApp individual?).

16.6.1 Foco do projeto

Em capítulos anteriores, enfatizamos uma progressão de projeto que inicia considerando-se questões de arquitetura e de componentes e representações de interfaces do usuário. O foco do projeto se torna “mais limitado” à medida que avançamos no projeto. Os problemas (e soluções) que encontraremos ao projetarmos uma arquitetura de informações para uma WebApp são diferentes dos problemas (e soluções) encontrados ao realizarmos o projeto de interfaces. Consequentemente, não será nenhuma surpresa o fato de que os padrões para o projeto de WebApps possam ser desenvolvidos para diferentes níveis de foco de projeto, para podermos tratar dos problemas únicos (e soluções relativas) encontrados em cada nível. Os padrões para WebApps podem ser classificados usando-se os seguintes níveis de foco do projeto:

- **Padrões de arquitetura de informações** relacionam-se à estrutura geral do espaço de informações e as maneiras pelas quais os usuários vão interagir com as informações.
- **Padrões de navegação** definem estruturas de links de navegação como hierarquias, anéis, tours e assim por diante.
- **Padrões de interação** tratam da maneira pela qual a interface informa o usuário das consequências de uma ação específica.
- **Padrões de apresentação** tratam de como organizar as funções de controle de interfaces do usuário para melhor usabilidade, como mostrar a relação entre uma ação de interface e os objetos de conteúdo que ela afeta e como estabelecer hierarquias de conteúdo eficientes.
- **Padrões funcionais** definem os fluxos de trabalho, comportamentos, processamento, comunicação e outros elementos algorítmicos contidos em uma WebApp.

Na maioria dos casos, seria infrutífero explorar o conjunto dos **padrões de arquitetura de informações** quando se encontra um problema no projeto de interação. Teríamos de examinar **padrões de interação**, pois esse é o foco do projeto relevante ao trabalho que está sendo realizado.

16.6.2 Granularidade do projeto

Quando um problema envolve questões de “quadro geral”, devemos tentar desenvolver soluções (e usar padrões relevantes) que se concentram no quadro geral. Quando o foco é muito limitado (por exemplo, selecionar unicamente um item de um pequeno conjunto de cinco ou menos itens), a solução (e o padrão correspondente) é obtida com pouca margem de erro. Em termos do nível de granularidade, os padrões para WebApps seguem os mesmos níveis de abstração discutidos anteriormente neste capítulo.

Os padrões de arquitetura definem a estrutura geral da WebApp, indicam as relações entre os diferentes componentes ou incrementos e definem as regras para especificar as relações entre os elementos (páginas, pacotes, componentes, subsistemas) da arquitetura. Os padrões de projeto tratam de um elemento específico do projeto de WebApps, como uma agregação de componentes para resolver algum problema de projeto, relações entre os elementos em uma página ou os mecanismos para realizar a comunicação componente-para-componente. Os padrões de componentes relacionam-se com elementos de pequena escala de uma WebApp. Entre os exemplos, temos elementos de interação individual (por exemplo, botões de seleção, livros-texto), itens de navegação (por exemplo, como formataríamos links?) ou elementos funcionais (por exemplo, algoritmos específicos).

INFORMAÇÕES



Repositórios de padrões de projeto para hipermídia

Na Internet existem vários catálogos e repositórios de padrões de hipermídia úteis. Centenas de padrões de projeto são representadas nos sites a seguir:

InteractionPatterns, de Tom Erickson

www.pliant.org/personal/Tom_Erickson/InteractionPatterns.html

Padrões de projeto para a Web, de Martijn vanWelie

www.welie.com/patterns/

Padrões de projeto de interface do usuário para a Web

<http://www.onextrapixel.com/2010/11/03/15-ui-design-patterns-web-designers-should-keep-handly/>

Aperfeiçoamento de sistemas de informações na Web com padrões de navegação

<http://www8.org/w8-papers/5b-hypertext-media/improving/improving.html>

Padrões para interface do usuário e relacionados, compilados em Uzilla.net

http://uzilla.net/uzilla/blog/hci_directory/searchresultde45.html

Common Ground – A Pattern Language for HCI Design

www.mit.edu/~jtidwell/interaction_patterns.html

Padrões para sites pessoais

www.rdrop.com/~half/Creations/Writings/Web.patterns/index.html

16.7 Padrões para aplicativos móveis

Por natureza, os aplicativos móveis se resumem à interface. Em muitos casos, os padrões para interface do usuário móvel [Mob12] são representados como uma coleção do “que há de melhor” em termos de imagens de tela para aplicativos, em diversas categorias diferentes. Exemplos típicos poderiam incluir:

Telas de registro. Como faço o registro a partir de um local específico, como faço um comentário e compartilho comentários com amigos e seguidores em uma rede social?

Mapas. Como exibo um mapa no contexto de um aplicativo que trata de algum outro assunto? Por exemplo, analisar um restaurante e representar sua localização em uma cidade.

Popovers. Como represento uma mensagem ou informação (do aplicativo ou de outro usuário) que surge em tempo real ou como consequência de uma ação do usuário?

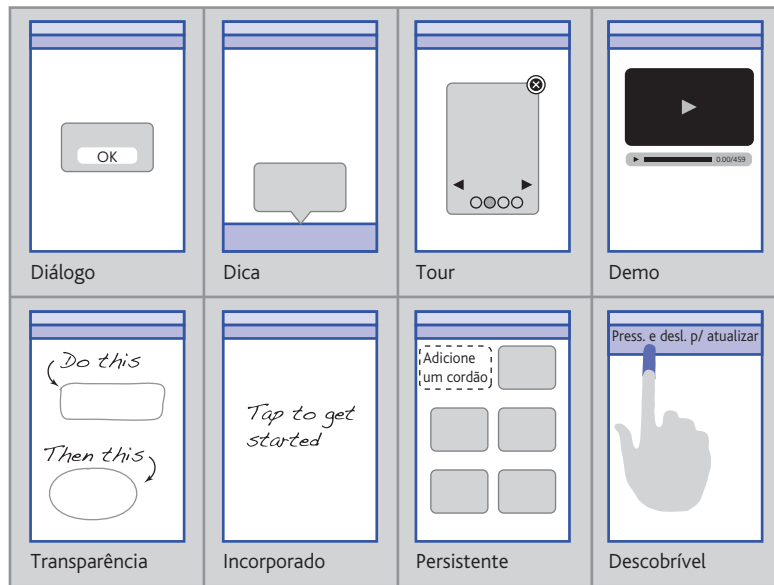


FIGURA 16.3 Exemplos de padrões de convites para aplicativos móveis [Nei11].

Fluxos de assinatura. Como forneço um modo simples de assinar ou se registrar para obter informações ou funcionalidade?

Navegação personalizada em guias. Como represento uma variedade de objetos de conteúdo diferentes de um modo que permita ao usuário escolher o que ele deseja?

Convites. Como informo ao usuário que ele deve participar de alguma ação ou diálogo? Exemplos típicos estão ilustrados na Figura 16.3.

Mais informações sobre padrões para interfaces do usuário de aplicativos móveis podem ser encontradas em [Nei12] e [Hoo12]. Além dos padrões para interfaces do usuário, Meier e seus colegas [Mei12] propõem uma variedade de descrições de padrão mais gerais para aplicativos móveis. Mais informações sobre padrões para aplicativos móveis, incluindo uma ampla biblioteca de padrões⁸, foram desenvolvidas pela Nokia [Nok13].

16.8 Resumo

Os padrões de projeto fornecem um mecanismo codificado para descrever problemas e suas soluções de maneira que a comunidade da engenharia de software possa adquirir conhecimento de projeto que possa ser reutilizado. Um padrão descreve um problema, indica o contexto que permite ao usuário compreender o ambiente em que o problema reside e lista um sistema de forças que indicam como o problema pode ser interpretado no seu contexto e como uma solução pode ser aplicada. No trabalho de engenharia de software,

⁸ Links para diversas bibliotecas de padrões para aplicativos móveis também podem ser encontrados em: <http://fourth.com/wiki/Other%20Mobile%20Pattern%20Libraries>.

identificamos e documentamos padrões generativos. Esses padrões descrevem um aspecto importante e repetível de um sistema, dando-nos então uma forma de construir esse aspecto em um sistema de forças que seja único em determinado contexto.

Os padrões de arquitetura descrevem problemas de projeto abrangentes, resolvidos usando-se uma abordagem estrutural. Os padrões de dados descrevem problemas recorrentes orientados a dados e as soluções de modelagem de dados que podem ser usadas para resolvê-los. Os padrões de componentes (também conhecidos como padrões de projeto) tratam de problemas associados ao desenvolvimento de subsistemas e componentes, a maneira pela qual eles se comunicam entre si e seu posicionamento em uma arquitetura mais ampla. Os padrões de projeto para interfaces descrevem problemas comuns de interfaces do usuário e suas soluções com um sistema de forças que inclui as características específicas dos usuários. Os padrões para WebApps lidam com um conjunto de problemas encontrado ao se construir WebApps e muitas vezes incorporam muitas das demais categorias de padrões mencionados. Os padrões para aplicativos móveis tratam da natureza única da interface e da funcionalidade móvel e controlam elementos específicos para as plataformas móveis.

Um framework fornece a infraestrutura em que padrões podem residir, e idiomas descrevem detalhes de implementação específicos de uma linguagem de programação para toda ou parte de uma estrutura de dados ou algoritmo específico. Um formulário ou modelo padrão é usado para descrições de padrões. Uma linguagem de padrões engloba um conjunto de padrões, cada qual descrito por meio do uso de um modelo de padrões e inter-relacionado para mostrar como esses padrões colaboram para solucionar problemas em um domínio de aplicação.

O projeto baseado em padrões é usado com métodos de projeto de arquitetura, de componentes e de interfaces do usuário. A abordagem de projeto começa com um exame do modelo de requisitos para isolar problemas, definir o contexto e descrever o sistema de forças. Em seguida, buscam-se linguagens de padrões para o domínio do problema, a fim de determinar se há padrões para os problemas que foram isolados. Uma vez encontrados padrões apropriados, eles são usados como um guia de projeto.

Problemas e pontos a ponderar

- 16.1. Discuta as três “partes” de um padrão de projeto e dê um exemplo concreto de cada um deles, de algum outro campo que não seja o de software.
- 16.2. Qual a diferença entre um padrão não generativo e generativo?
- 16.3. Como os padrões de arquitetura diferem dos padrões de componentes?
- 16.4. O que é um framework de uso de padrões e como ele difere de um padrão? O que é um idioma e como ele difere de um padrão?
- 16.5. Usando o modelo de padrões de projeto apresentado na Seção 16.1.3, desenvolva uma descrição de padrão completa para um padrão sugerido por seu professor.
- 16.6. Desenvolva uma linguagem de padrões estrutural para um esporte com o qual você esteja familiarizado. Você pode começar lidando com o contexto, o sistema de for-

ças e os problemas abrangentes que um técnico e a equipe devem solucionar. É preciso não apenas especificar os nomes de padrão, mas também elaborar uma descrição em uma sentença para cada padrão.

16.7. Encontre cinco repositórios de padrões e apresente uma descrição abreviada dos tipos de padrões contidos em cada um deles.

16.8. Quando Christopher Alexander diz “um bom projeto de software não pode ser alcançado simplesmente juntando-se peças operantes”, o que você acha que ele quer dizer com isso?

16.9. Usando as tarefas para projeto baseado em padrões citadas na Seção 16.2.3, desenvolva um esboço de projeto para o “sistema de design de interiores” descrito na Seção 15.3.2.

16.10. Construa uma tabela para organizar os padrões utilizados no Problema 16.9.

16.11. Usando o modelo de padrões de projeto apresentado na Seção 16.1.3, desenvolva uma descrição completa para o padrão **Kitchen** mencionado na Seção 16.3.

16.12. A “gangue dos quatro” [Gam95] propôs uma variedade de padrões de componentes aplicáveis a sistemas orientados a objetos. Escolha um (eles se encontram à disposição na Web) e discuta-o.

16.13. Encontre três repositórios de padrões para interfaces do usuário. Escolha um padrão de cada e apresente uma descrição abreviada dele.

16.14. Encontre três repositórios de padrões para padrões de WebApps. Escolha um padrão de cada e apresente uma descrição abreviada dele.

16.15. Encontre três repositórios de padrões para aplicativos móveis. Escolha um padrão de cada e apresente uma descrição abreviada dele.

Leituras e fontes de informação complementares

Lançaram-se vários livros sobre projeto baseado em padrões destinados a engenheiros de software. Gamma e seus colegas [Gam95] escreveram o livro seminal sobre o tema. Entre algumas contribuições mais recentes, temos obras como as de Burris (*Programming in the Large with Design Patterns*, Pretty Print Press, 2012), Smith (*Elemental Design Patterns*, Addison-Wesley, 2012), Lasater (*Design Patterns*, Wordware Publishing, 2007), Holzner (*Design Patterns for Dummies*, For Dummies, 2006), Freeman e seus colegas (*Head First Design Patterns*, O'Reilly Media, 2005) e Shalloway e Trott (*Design Patterns Explained*, 2ª ed., Addison-Wesley, 2004). Kent Beck (*Implementation Patterns*, Addison-Wesley, 2008) trata de padrões para problemas de codificação e implementação encontrados durante a atividade de construção.

Outros livros focalizam padrões de projeto à medida que são fornecidos em ambientes específicos de linguagem e desenvolvimento de aplicações. Entre as contribuições nessa área, temos: Bowers e seus colegas (*Pro HTML5 and CSS3 Design Patterns*, Apress, 2011), Scott e Neil (*Designing Web Interfaces: Principles and Patterns for Rich Interactions*, O'Reilly, 2009), Tropashko e Burleson (*SQL Design Patterns: Expert Guide to SQL Programming*, Rampant Tech-press, 2007), Mahemoff (*Ajax Design Patterns*, O'Reilly Media, 2006), Bevis (*Java Design Pattern Essentials*, Ability First Limited, 2010), Metsker e Wake (*Design Patterns in Java*, Addison-Wesley, 2006), Millett (*Professional ASP.NET Design Patterns*, Wrox, 2010), Nilsson (*Applying Domain-Driven Design and Patterns: With Examples in C# and .NET*, Addison-Wesley, 2006), Stefanov (*JavaScript Patterns*, O'Reilly, 2010), Sweat (*PHP | Architect's Guide to PHP Design Patterns*, Marco Tabini & Associates, 2005), Paul (*Design Patterns in C#*), Metsker (*Design Patterns C#*, Addison-Wesley, 2004), Grand e Merrill (*Visual Basic .NET Design Patterns*, Wiley, 2003), Crawford e Kaplan (*J2EE*

Design Patterns, O'Reilly Media, 2003), Juric *et al.* (*J2EE Design Patterns Applied*, Wrox Press, 2002) e Marinescu e Roman (*EJB Design Patterns*, Wiley, 2002).

Outras obras ainda tratam de campos de aplicação específicos. Entre elas estão a de Kuchana (*Software Architecture Design Patterns in Java*, Auerbach, 2004), Joshi (*C++ Design Patterns and Derivatives Pricing*, Cambridge University Press, 2004), Douglass (*Real-Time Design Patterns*, Addison-Wesley, 2002) e Schmidt e Rising (*Design Patterns in Communication Software*, Cambridge University Press, 2001).

Clássicos do arquiteto Christopher Alexander (*Notes on the Synthesis of Form*, Harvard University Press, 1964 e *A Pattern Language: Towns, Buildings, Construction*, Oxford University Press, 1977) são leituras que valem a pena para um projetista de software que pretenda entender completamente os padrões de projeto.

Uma ampla gama de fontes de informação sobre projeto baseado em padrões se encontra à disposição na Internet. Uma lista atualizada de referências relevantes (em inglês) para o processo para o projeto baseado em padrões pode ser encontrada no site: www.mhhe.com/pressman.

Em seu respeitado livro sobre projeto para Web, Jakob Nielsen [Nie00] afirma: “Existem, essencialmente, duas abordagens básicas para projeto: o ideal artístico de se expressar e o ideal de engenharia de resolver um problema para um cliente”. Durante a primeira década de desenvolvimento da Web, o ideal artístico foi a abordagem que muitos desenvolvedores escolheram. O projeto ocorria de maneira assistemática e normalmente era feito à medida que o código HTML era gerado. O projeto se desenvolveu a partir de uma visão artística que evoluiu junto com a construção de WebApps.

Mesmo hoje, muitos desenvolvedores Web usam WebApps como a representação perfeita de um “projeto limitado”. Eles argumentam que o imediatismo e a volatilidade das WebApps aliviam o processo de projeto formal; que o projeto evolui à medida que uma aplicação é construída (implementada); e que relativamente pouco tempo deve ser gasto na criação de um modelo de projeto mais detalhado. Esse argumento tem seus méritos, mas apenas para WebApps relativamente simples. Quando o conteúdo e a funcionalidade são complexos; quando o tamanho da WebApp engloba centenas ou milhares de

Conceitos-chave

arquitetura da WebApp..	384
arquitetura de conteúdo	381
design gráfico	377
lista de controle (checklist) de qualidade	374
modelo-visão-controlador	384
objetivos de projeto	374
objetos de conteúdo	379
pirâmide de projeto	375
projeto de arquitetura	381
projeto de conteúdo	379
projeto de navegação	385
projeto em nível de componentes	387
projeto estético	377
qualidade de projeto	372

PANORAMA

O que é? O projeto de WebApps abrange atividades técnicas e não técnicas: estabelecer a percepção e a aparência da WebApp, criar o layout estético da interface do usuário, definir a estrutura geral da arquitetura, desenvolver o conteúdo e a funcionalidade que residem na arquitetura e planejar a navegação que ocorre na WebApp.

Quem realiza? Engenheiros de aplicações para a Web, designers gráficos, desenvolvedores de conteúdo e outros envolvidos participam na criação de um modelo de projeto de uma WebApp.

Por que é importante? O projeto permite criar um modelo que pode ser avaliado em termos de qualidade e aperfeiçoado antes de os códigos e conteúdos serem gerados; os testes serem realizados; e muitos usuários se envolverem. É no projeto que se estabelece a qualidade de uma WebApp.

Quais são as etapas envolvidas? O projeto de WebApps abrange seis etapas principais, orientadas por informações obtidas durante a modelagem de requisitos. O projeto de conteúdo usa o modelo de conteúdo (desenvolvido durante

a análise) como base para estabelecer o projeto de objetos de conteúdo. O projeto estético (também chamado design gráfico) estabelece o layout que o usuário verá. O projeto da arquitetura se concentra na estrutura geral de hipermídia de todos os objetos de conteúdo e funções. O projeto da interface estabelece mecanismos de layout e interação que definem a interface do usuário. O projeto de navegação define como o usuário navega pela estrutura de hipermídia e o projeto de componentes representa a estrutura interna detalhada dos elementos funcionais da WebApp.

Qual é o artefato? Um modelo de projeto abrangendo questões de conteúdo, estética, arquitetura, interface, navegação e de projeto de componentes é o principal artefato gerado durante o projeto de uma WebApp.

Como garantir que o trabalho foi realizado corretamente? Cada elemento do modelo de projeto é revisado na tentativa de se descobrir erros, inconsistências ou omissões. Além disso, são consideradas soluções alternativas e também é avaliado o grau com que o modelo de projeto atual vai levar a uma implementação efetiva.

objetos de conteúdo, funcionalidades e classes de análise; e quando o sucesso da WebApp terá um impacto direto no sucesso do negócio, o projeto não pode e não deve ser tratado de maneira superficial.

Essa realidade nos leva à segunda abordagem de Nielsen: “o ideal da engenharia de resolver um problema para um cliente”. A engenharia para Web¹ adota essa filosofia, e uma abordagem mais rigorosa para projeto de WebApps permite aos desenvolvedores alcançarem tal objetivo.

17.1 Qualidade de projeto em WebApps

Todo mundo que já navegou na Web tem opinião formada sobre o que faz uma WebApp ser “boa”. Os pontos de vista individuais variam muito. Alguns usuários adoram imagens chamativas, outros querem apenas texto. Alguns exigem informações detalhadas, outros desejam uma apresentação resumida. Alguns preferem ferramentas analíticas sofisticadas ou acesso a bancos de dados, outros preferem a simplicidade. Na realidade, a percepção do usuário de “excelência” (e a resultante aceitação ou rejeição da WebApp como consequência) talvez seja mais importante do que qualquer discussão técnica sobre a qualidade das WebApps.

Mas como a qualidade de uma WebApp é percebida? Quais atributos devem ser apresentados para atingir a excelência segundo a visão dos usuários se, ao mesmo tempo, apresentar as características técnicas de qualidade que tornará possível corrigir, adaptar, melhorar e dar suporte à WebApp no longo prazo?

Na realidade, todas as características técnicas da qualidade de projetos discutidas no Capítulo 12 e os atributos de qualidade gerais apresentados no Capítulo 19 se aplicam às WebApps. Entretanto, os atributos de qualidade gerais mais relevantes – usabilidade, funcionalidade, confiabilidade, eficiência e facilidade de manutenção – fornecem uma base útil para se avaliar a qualidade de sistemas baseados na Web.

Olsina e seus colegas [Ols99] prepararam uma “árvore de requisitos de qualidade” identificando um conjunto de atributos técnicos – usabilidade, funcionalidade, confiabilidade, eficiência e facilidade de manutenção – que levam a WebApps de alta qualidade.² A Figura 17.1 sintetiza o trabalho desses pesquisadores. Os critérios citados na figura são de particular interesse, caso você tenha de projetar, construir e manter WebApps no longo prazo.

Offutt [Off02] amplia os cinco principais atributos de qualidade citados na Figura 17.1, acrescentando os seguintes atributos:

Segurança. As WebApps se tornaram altamente integradas a bancos de dados críticos como os corporativos e os governamentais. Aplicações de comércio eletrônico extraem e depois armazenam informações confidenciais de clientes. Por essas e muitas outras razões, a segurança da WebApp é primordial em várias situações. A principal medida de segu-

Quais os principais atributos de qualidade para as WebApps?

¹ Engenharia para Web (*Web engineering*) [Pre08] é uma versão adaptada da abordagem de engenharia de software apresentada ao longo deste livro. Ela propõe uma estrutura ágil, embora disciplinada, para construção de sistemas e aplicações baseados na Web com alta qualidade.

² Esses atributos de qualidade são muito parecidos com os apresentados nos Capítulos 12 e 19. A implicação: características de qualidade são universais para todo software.

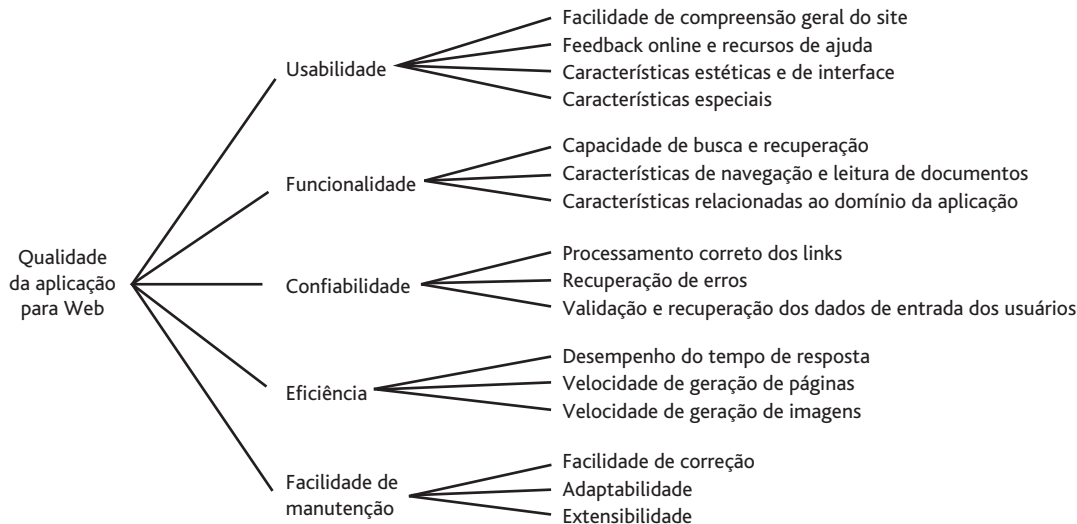


FIGURA 17.1 Árvore de requisitos de qualidade.

Fonte: [Ols99].

rança é a capacidade de a WebApp e seu ambiente de servidor rejeitarem acesso não autorizado e/ou frustrarem um ataque mal-intencionado. A engenharia de segurança é discutida no Capítulo 27. Para mais informações sobre WebApp, consulte [Web13], [Pri10], [Vac06] e [Kiz05].

Disponibilidade. Até mesmo a melhor WebApp não atenderá às necessidades dos usuários caso esteja indisponível. Em um sentido técnico, disponibilidade é a medida da porcentagem de tempo que uma WebApp está disponível para uso. Mas Offutt [Off02] sugere que “as características de uso disponíveis em apenas um navegador ou uma plataforma” torna a WebApp indisponível para aqueles que usam um navegador/plataforma diferente. O usuário invariavelmente vai procurar uma alternativa.

Escalabilidade. A WebApp e seus servidores podem ser dimensionados para atender 100, 1.000, 10.000 ou 100.000 usuários? A WebApp e os sistemas integrados conseguem lidar com variação significativa de volume ou sua capacidade de resposta cairá significativamente (ou cessará de vez)? É importante projetar uma WebApp capaz de acomodar as responsabilidades inerentes ao sucesso (isto é, um número significativamente maior de usuários) e se tornar cada vez mais bem-sucedida.

Tempo para colocação no mercado (*time-to-market*). Embora o tempo para colocação de um produto no mercado não seja um verdadeiro atributo de qualidade no sentido técnico, é uma medida de qualidade do ponto de vista comercial. A primeira WebApp a atender determinado segmento de mercado em geral captura um número grande de usuários.

Bilhões de páginas Web se encontram disponíveis para quem busca informações. Mesmo as buscas na Web bem direcionadas resultam em uma avalanche de conteúdo. Com tantas fontes de informação à escolha, como o usuário pode avaliar a qualidade (por exemplo, veracidade, precisão, completude, oportunidade) do conteúdo apresentado em uma WebApp?

INFORMAÇÕES



Projeto de WebApps – checklist de qualidade

A checklist a seguir, adaptada das informações apresentadas em **Webreference.com**, fornece um conjunto de perguntas que ajuda tanto os engenheiros de aplicações para Web quanto os usuários a avaliar a qualidade geral de uma WebApp:

- Opções de conteúdo e/ou funcionalidade e/ou navegação podem ser ajustadas às preferências dos usuários?
- O conteúdo e/ou funcionalidade podem ser personalizados para a largura de banda em que o usuário se comunica?
- Imagens e outras mídias não textuais foram usadas adequadamente? Os tamanhos de arquivos gráficos são otimizados para eficiência de exibição?
- As tabelas são organizadas e dimensionadas para torná-las compreensíveis e exibidas de forma eficiente?
- O código HTML é otimizado para eliminar ineficiências?
- O projeto geral de páginas é fácil de ler e navegar?
- Todos os links fornecem informações de interesse dos usuários?
- É provável que a maioria dos links tenha persistência na Web?
- A WebApp é equipada com recursos de administração de sites que incluem ferramentas para acompanhamento de uso, testes de links, busca de locais e segurança?

O que devemos considerar ao avaliarmos a qualidade do conteúdo?

Tillman [Til00] sugere um conjunto útil de critérios para avaliar a qualidade dos conteúdos: O escopo e a profundidade do conteúdo podem ser facilmente determinados para garantir que atenda às necessidades dos usuários? A experiência e a confiabilidade dos autores do conteúdo podem ser facilmente identificadas? É possível determinar a atualidade do conteúdo, a última atualização e o que foi atualizado? O conteúdo e sua localização são estáveis (isto é, permanecerão na URL referida)? O conteúdo é confiável? O conteúdo é exclusivo? A WebApp fornece algum benefício para aqueles que o usam? O conteúdo tem valor desejado pela comunidade de usuários? O conteúdo é bem organizado? É indexado? É facilmente acessível? Essas perguntas representam apenas uma pequena amostra das questões que devem ser tratadas à medida que o projeto de uma WebApp evolui.

17.2 Objetivos de projeto

Em sua coluna regular sobre projeto para a Web, Jean Kaiser [Kai02] sugere um detalhado conjunto de objetivos de projeto que podem ser aplicados a praticamente qualquer WebApp, independentemente do domínio de aplicação, do tamanho ou da complexidade:

“Só porque você pode não significa que deve.”

Jean Kaiser

Simplicidade. Embora possa parecer ultrapassado, o aforismo “tudo com moderação” se aplica às WebApps. Há uma tendência entre alguns projetistas de fornecer “em excesso” ao usuário— conteúdo exaustivo, aspectos visuais excessivos, animação invasiva, páginas Web enormes, navegação complexa... a lista é longa. É melhor se esforçar pela moderação e simplicidade.

Consistência. Esse objetivo de projeto se aplica a praticamente qualquer elemento do modelo de projeto. O conteúdo deve ser construído com consistência (por exemplo, a formatação de texto e os estilos de fonte devem ser os mesmos ao longo de todos os documentos de texto; a arte gráfica deve ter aspecto, combinação de cores e estilo coerentes). O design gráfi-

co (estética) deve apresentar um aspecto semelhante em todas as partes da WebApp. Os mecanismos de navegação devem ser usados coerentemente por todos os elementos da WebApp. Como Kaiser [Kai02] observa: “Lembre-se de que, para um visitante, um site é um lugar físico. Fica muito confuso se as páginas tiverem um design padrão”.

Identidade. A estética, a interface e o projeto de navegação de uma WebApp devem ser coerentes com o domínio de aplicação para o qual ela será construída. Um site de uma empresa musical sem dúvida terá percepção e aparência diferentes de uma WebApp projetada para uma companhia financeira.

Robustez. Com base na identidade estabelecida, uma WebApp frequentemente faz uma “promessa” implícita ao usuário. O usuário espera conteúdo e funções robustos, relevantes às suas necessidades. Se esses elementos estiverem faltando ou forem insuficientes, é provável que a WebApp fracasse.

Navegabilidade. A navegação deve ser projetada de maneira intuitiva e previsível. Ou seja, o usuário deve saber navegar pela WebApp sem ter de buscar links ou instruções de navegação. Por exemplo, se um campo de imagens ou ícones contiver ícones ou imagens selecionados que serão usados como mecanismos de navegação, eles devem ser identificados visualmente. Nada é mais frustrante do que tentar encontrar o link ativo apropriado entre muitas imagens.

Apelo visual. De todas as categorias de software, as aplicações para Web são, inquestionavelmente, as mais visuais, as mais dinâmicas e as mais estéticas. A beleza (apelo visual) é um conceito que varia segundo a ótica de quem a vê; porém, muitas características de projeto (por exemplo, a percepção e aspecto do conteúdo, o layout da interface, a coordenação das cores, o equilíbrio entre texto, imagens e outras mídias, os mecanismos de navegação) contribuem efetivamente para o apelo visual.

Compatibilidade. Uma WebApp será usada em uma variedade de ambientes (por exemplo, hardware, tipos de conexão de Internet, sistemas operacionais e navegadores diferentes) e deve ser projetada para ser compatível com cada um deles.

“Para alguns, o projeto para Web enfoca o aspecto visual... Para outros, projeto para Web significa estruturar informações e navegação pelo espaço do documento. Outros, ainda, podem até considerar que o projeto para a Web trata da tecnologia usada para construir aplicações Web interativas. Na realidade, o projeto inclui todas essas coisas e talvez ainda mais.”

Thomas Powell

17.3 Uma pirâmide de projeto para WebApps

No contexto da engenharia para Web, o que é projeto? Essa simples pergunta é mais difícil de responder do que se possa imaginar. Pressman e Lowe [Pre08] discutem isso ao escreverem:

Tipicamente, a criação de um projeto eficaz exigirá um conjunto de habilidades diversas. Às vezes, para pequenos projetos, um único desenvolvedor precisaria ter vários talentos. Para projetos maiores, talvez seja prudente e/ou viável recorrer à experiência de especialistas: engenheiros Web, designers gráficos, desenvolvedores de conteúdo, programadores, especialistas em banco de dados, arquitetos de informação, engenheiros de rede, especialistas em segurança e testadores. Fazer uso dessas diversas capacidades permite a criação de um modelo que pode ser

“Se um site é perfeitamente usável, mas falta um estilo de projeto elegante e adequado, ele fracassará.”

Curt Cloninger

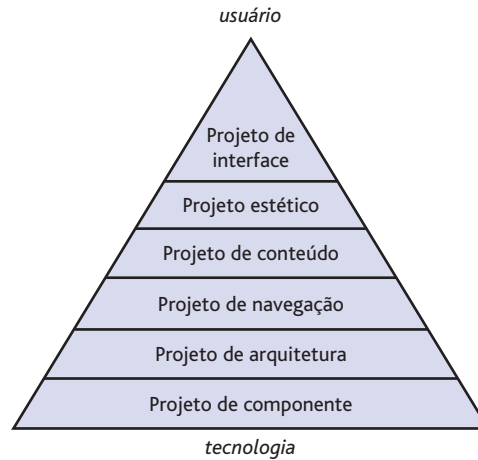


FIGURA 17.2 Uma pirâmide de projeto para WebApps.

avaliado em termos de qualidade e aperfeiçoado *antes* de o conteúdo e o código serem gerados, os testes serem realizados e os usuários envolverem-se em grande número. Se análise é o momento em que *se estabelece a qualidade de uma WebApp*, então projeto é o momento em que a *qualidade é realmente incorporada*.

O mix apropriado de habilidades de projeto vai variar dependendo da natureza da WebApp. A Figura 17.2 apresenta uma pirâmide de projeto para WebApps. Cada nível da pirâmide representa uma ação de projeto descrita nas seções a seguir.

17.4 Projeto de interfaces para WebApp

Quando um usuário interage com um sistema computacional, aplica-se um conjunto de princípios fundamentais e diretrizes de projeto primordiais. Estes foram discutidos no Capítulo 15.³ Embora as WebApps apresentem alguns desafios especiais no projeto da interface do usuário, as diretrizes e princípios básicos se aplicam.

Um dos desafios no projeto da interface para WebApps é a natureza indeterminada do ponto de entrada do usuário. Ou seja, o usuário pode entrar na WebApp em um local “home” (por exemplo, a homepage) ou, por meio de um link, pode entrar em algum nível mais específico da arquitetura da WebApp. Em alguns casos, a WebApp pode ser projetada para redirecionar o usuário a um local home, mas, se isso for indesejável, o projeto da WebApp deve fornecer recursos de navegação de interface que acompanhem todos os objetos de conteúdo e estejam disponíveis independentemente de como o usuário entre no sistema.

Os objetivos de uma interface para WebApp são: (1) estabelecer uma janela para o conteúdo e a funcionalidade fornecidos pela interface, (2) guiar o usuário com uma série de interações com a WebApp e (3) organizar as opções de navegação e conteúdo disponíveis para o usuário. Para obtermos uma in-

³ A Seção 15.5 é dedicada ao projeto da interface de WebApps. Caso ainda não tenha feito, leia-a agora.

terface sólida, devemos primeiro usar a estética do projeto (Seção 17.5) para estabelecer um “aspecto” coerente. Isso abrange várias características, mas deve enfatizar o layout e a forma dos mecanismos de navegação. Para orientarmos a interação com o usuário, podemos fazer uso de uma metáfora⁴ apropriada que permita ao usuário ter um entendimento intuitivo da interface. Para implementar opções de navegação, podemos selecionar *menus de navegação* posicionados de modo padrão em páginas Web, *ícones gráficos* representados de uma maneira que permita ao usuário reconhecer que o ícone é um elemento de navegação e/ou *imagens gráficas* que forneçam um link para um objeto de conteúdo ou para uma funcionalidade da WebApp. É importante notar que um ou mais desses mecanismos de navegação devem ser fornecidos em todos os níveis da hierarquia de conteúdo.

Nem todo engenheiro de aplicações Web (ou engenheiro de software) tem talento artístico (estético). Caso se enquadre nessa categoria, contrate um designer gráfico experiente para realizar a tarefa de projeto estético.

17.5 Projeto estético

O projeto estético, também chamado *design gráfico*, é o esforço artístico que complementa os aspectos técnicos do projeto de WebApps. Sem ele, uma WebApp poderia ser funcional, mas não atraente. Com ele, uma WebApp atrai seus usuários para um mundo que os envolve em um nível físico e intelectual.

Mas o que é estética? Há um velho ditado que diz: “A beleza está nos olhos de quem vê”. Isso é particularmente apropriado quando se considera o projeto estético de WebApps. Para realizar um projeto estético eficaz, retorne à hierarquia de usuários desenvolvida como parte do modelo de requisitos (Capítulo 8) e pergunte: Quem são os usuários da WebApp e que “visual” eles desejam?

Quais mecanismos de interação estão disponíveis para os projetistas de WebApps?



Design gráfico

Cena: Escritório de Doug Miller, após a primeira revisão do protótipo de interface web.

Atores: Doug Miller, gerente de projeto de engenharia de software do CasaSegura, e Vinod Raman, membro da equipe de engenharia de software do CasaSegura.

Conversa:

Doug: Que impressão você teve do novo design da página Web?

Vinod: Gosto dele, mas o mais importante é nossos clientes gostarem.

Doug: A designer gráfica que emprestamos do marketing ajudou?

Vinod: Na verdade, muito. Ela é ótima em layout de páginas e sugeriu um tema gráfico incrível para elas. Muito melhor do que o nosso.

Doug: Isso é bom. Algum problema?

Vinod: Ainda precisamos criar páginas alternativas para levar em conta as questões de acessibilidade para alguns de nossos usuários deficientes visuais. Mas precisaríamos fazer isso para todo projeto de página Web de que necessitamos.

Doug: Precisamos de ajuda com design gráfico nas páginas alternativas também?

Vinod: Certamente. A designer tem um bom entendimento dos problemas de usabilidade e acessibilidade.

Doug: Certo, vou perguntar ao marketing se podemos empréstá-la por mais algum tempo.

CASASEGURA

⁴ Nesse contexto, *metáfora* é uma representação (extraída da experiência real do usuário) que pode ser modelada no contexto da interface. Um exemplo simples poderia ser um controle deslizante usado para controlar o volume do áudio de um arquivo .mpg.

“Constatamos que as pessoas avaliam rapidamente um site apenas pelo projeto visual.”

Diretrizes para Credibilidade na Web da Stanford

17.5.1 Questões de layout

Toda página Web tem uma quantidade limitada de “terreno” que pode ser usado para dar suporte à estética não funcional, recursos de navegação, conteúdo de informação e funcionalidade dirigida ao usuário. O desenvolvimento desse terreno é planejado durante o projeto estético.

Assim como todas as questões estéticas, não há regras absolutas quando se desenvolve o layout da tela. Entretanto, vale a pena considerarmos uma série de diretrizes gerais para layout:

Não tenha medo de espaços abertos. É desaconselhável preencher com informação cada centímetro de uma página Web. O congestionamento visual resultante torna difícil para o usuário identificar as informações ou os recursos necessários e cria um caos visual desagradável.

Enfatize o conteúdo. Afinal de contas, essa é razão para o usuário estar lá. Nielsen [Nie00] sugere que o uso de página Web típico deve ter 80% de conteúdo e o espaço restante dedicado à navegação e outros recursos.

Organize os elementos do layout da parte superior esquerda para a inferior direita. A grande maioria dos usuários percorrerá uma página Web de uma forma muito parecida ao que faz ao ler a página de um livro – da parte superior esquerda para a inferior direita.⁵ Se os elementos do layout tiverem prioridades específicas, os de alta prioridade devem ser colocados na parte superior esquerda do espaço da página.

Agrupe a navegação, o conteúdo e as funções geograficamente dentro da página. Os seres humanos buscam padrões em quase tudo. Se não existirem padrões discerníveis em uma página Web, a frustração do usuário provavelmente aumentará (devido a buscas infrutíferas por informação necessária).

Não estenda seu espaço com a barra de rolagem. Embora muitas vezes a rolagem seja necessária, a maioria dos estudos indica que os usuários preferem não ficar rolando a página. Frequentemente, é melhor reduzir o conteúdo da página Web ou apresentar o conteúdo necessário em várias páginas.

Considere a resolução e o tamanho da janela do navegador ao elaborar seu layout. Em vez de definir tamanhos fixos em um layout, o projeto deve especificar todos os itens de layout como uma porcentagem do espaço disponível [Nie00]. Com o crescente uso de dispositivos móveis com diferentes tamanhos de tela, esse conceito se torna cada vez mais importante.

17.5.2 Questões de design gráfico

O design gráfico considera todos os aspectos visuais de uma WebApp. O processo de design gráfico começa com o layout (Seção 17.5.1) e prossegue com a consideração de combinações de cores gerais, tipografias, tamanhos e estilos

⁵ Existem exceções que se baseiam em questões culturais e do idioma usado, mas tal regra se aplica à maioria dos usuários.

de texto, o uso de mídia complementar (por exemplo, áudio, vídeo, animação) e todos os demais elementos estéticos de uma aplicação.

Uma discussão completa sobre questões relativas ao design gráfico em WebApps está fora dos objetivos deste livro. Você pode obter dicas e diretrizes em muitos sites dedicados ao tema (por exemplo, www.graphic-design.com, www.webdesignfromscratch.com, www.wpdfd.com) ou em um ou mais recursos impressos (por exemplo, [Bea11], [McN10], [Lup08] e [Roc06]).

INFORMAÇÕES



Sites bem projetados

Algumas vezes, a melhor maneira de entender um bom projeto de uma WebApp é ver alguns exemplos. Em seu artigo “The Top Twenty Web Design Tips”, Marcelle Toor (www.graphic-design.com/Web/feature/tips.html) sugere os seguintes sites como exemplos de design gráfico adequado:

<http://www.marcelletoordesigns.com/> – ILR Website
Design é um recurso de ponta para clientes que estejam desenvolvendo seus sites e outras aventuras na Internet.

www.workbook.com – este site mostra o trabalho de ilustradores e designers.

www.pbs.org/riverofsong – série para TV pública e rádio sobre música americana.

www.RKDINC.com – empresa de design com portfólio online e excelentes dicas sobre design.

<http://www.creativehotlist.com/> – excelente fonte de sites bem desenhados por agências de propaganda, empresas de artes gráficas e outros especialistas da comunicação.

www.btdnyc.com – empresa de design dirigida por Beth Toudreau.

17.6 Projeto de conteúdo

O projeto de conteúdo aborda duas tarefas de projeto diferentes, cada uma delas tratada por indivíduos com conjuntos de habilidades diferentes. Primeiro, são desenvolvidos uma representação de projeto para objetos de conteúdo e os mecanismos necessários para estabelecer suas relações. Segundo, são criadas as informações em um objeto de conteúdo específico. Esta última tarefa pode ser realizada por redatores publicitários, designers gráficos e outros que geram o conteúdo a ser utilizado em uma WebApp.

“Bons designers são capazes de criar normalidade no caos; eles conseguem transmitir ideias claramente por meio da organização e manipulação de palavras e figuras.”

Jeffery Veen

17.6.1 Objetos de conteúdo

No contexto de projeto para WebApps, um objeto de conteúdo está mais em consonância com um objeto de dados para software tradicional. Um *objeto de conteúdo* possui atributos que incluem informações específicas de conteúdo (normalmente definidas durante a modelagem de requisitos da WebApp) e atributos de implementação exclusivos, especificados como parte do projeto.

Consideremos, por exemplo, uma classe de análise, **ComponenteDoProduto**, desenvolvida para o sistema de comércio eletrônico do *CasaSegura*. O atributo da classe de análise, **descrição**, é representado como uma classe de projeto chamada **DescriçãoDeComponente**, composta de cinco objetos de conteúdo: **DescriçãoMarketing**, **Fotografia**, **DescriçãoTécnica**, **Esquema** e **Vídeo**,

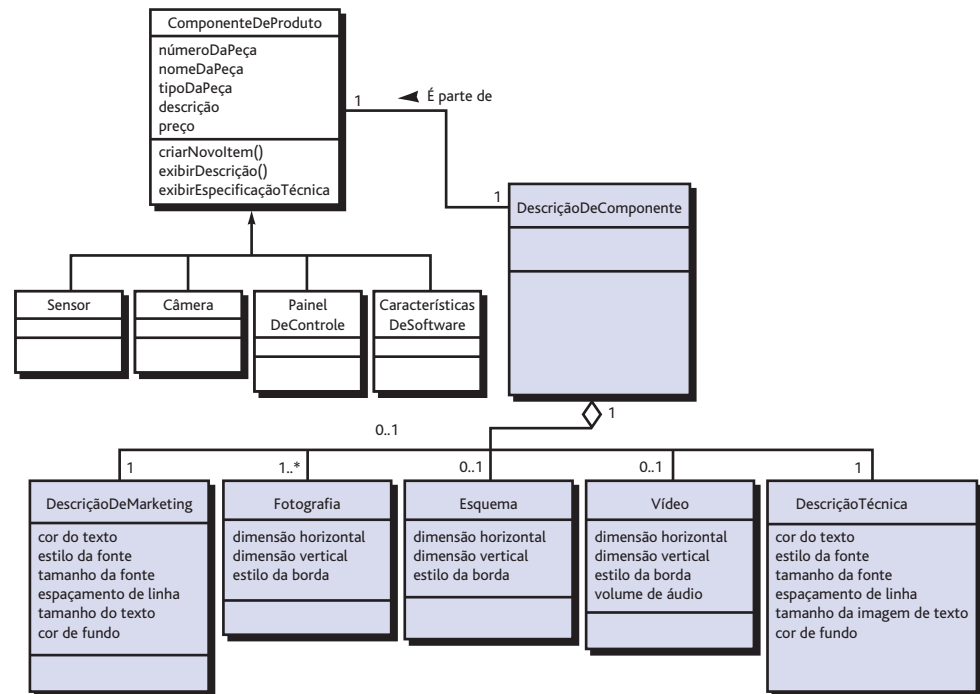


FIGURA 17.3 Representação de projeto dos objetos de conteúdo.

mostrados como objetos sombreados na Figura 17.3. As informações contidas no objeto de conteúdo são indicadas na forma de atributos. Por exemplo, **Fotografia** (uma imagem .jpg) possui os atributos **dimensão horizontal**, **dimensão vertical** e **estilo da borda**.

Associação e agregação⁶ UML podem ser usadas para representar relações entre os objetos de conteúdo. Por exemplo, a associação UML da Figura 17.3 indica que é usada uma classe **DescriçãoDeComponente** para cada instância da classe **ComponenteDoProduto**. **DescriçãoDeComponente** é composta pelos cinco objetos de conteúdo mostrados. Entretanto, a notação de multiplicidade indica que **Esquema** e **Vídeo** são opcionais (é possível que não haja nenhuma ocorrência), uma **DescriçãoDeMarketing** e uma **DescriçãoTécnica** são necessárias e que são usadas uma ou mais instâncias de **Fotografia**.

17.6.2 Questões de projeto de conteúdo

Assim que todos os objetos de conteúdo forem modelados, as informações que cada objeto deve fornecer precisam passar por um processo de autoria e formatação para melhor atender às necessidades do cliente. A autoria de conteúdo é tarefa de especialistas que projetam o objeto de conteúdo fornecendo um resumo das informações a serem entregues e uma indicação dos tipos de objetos de conteúdo genéricos (por exemplo, texto descritivo, imagens, fotografias) usados para transmiti-las. O projeto estético (Seção 17.5) também pode ser aplicado para representar o aspecto visual do conteúdo.

⁶ Essas duas representações são discutidas no Apêndice 1.

À medida que os objetos de conteúdo são projetados, eles são “agrupados” [Pow02] para formar páginas Web. O número de objetos de conteúdo incorporados em uma única página é função das necessidades do usuário, das restrições impostas pela velocidade de download da conexão de Internet disponível e das restrições impostas pelo nível de rolagem que o usuário vai tolerar.

Os usuários tendem a tolerar rolagem vertical mais facilmente do que a horizontal. Evite formatos de página largos.

17.7 Projeto de arquitetura

O projeto de arquitetura está ligado aos objetivos estabelecidos para uma WebApp, ao conteúdo a ser apresentado, aos usuários que visitarão a página e à filosofia de navegação estabelecida. Como projetistas da arquitetura, temos de identificar a arquitetura do conteúdo e a arquitetura da WebApp. A *arquitetura do conteúdo*⁷ focaliza a maneira pela qual objetos de conteúdo (ou objetos compostos, como páginas Web) são estruturados para apresentação e navegação. A *arquitetura das WebApps* lida com a maneira pela qual a aplicação é estruturada para administrar a interação com o usuário, tratar tarefas de processamento interno, navegação efetiva e apresentação de conteúdo.

Na maioria dos casos, o projeto de arquitetura é conduzido em paralelo com os projetos da interface, estético e de conteúdo. Como a arquitetura da WebApp pode ter uma forte influência sobre a navegação, as decisões tomadas durante as etapas de projeto influenciarão o trabalho conduzido durante o projeto de navegação.

“A estrutura arquitetural de um site bem projetado nem sempre é aparente para o usuário – e nem deve ser.”

Thomas Powell

17.7.1 Arquitetura de conteúdo

O projeto da arquitetura de conteúdo concentra-se na definição da estrutura geral de hipermídia da WebApp. Embora algumas vezes sejam criadas arquiteturas personalizadas, sempre temos a opção de escolher uma de quatro estruturas de conteúdo diferentes [Pow02]:

Estruturas lineares (Figura 17.4) são encontradas quando uma sequência de interações previsível (com certa variação ou desvios) é comum. Um exemplo clássico poderia ser uma apresentação de um tutorial em que as páginas de informação, junto de imagens, vídeos de curta duração ou áudio relacionados, são apresentados apenas após as informações de pré-requisito terem sido apresentadas. A sequência da apresentação do conteúdo é predefinida e, em geral, linear. Outro exemplo poderia ser a sequência de preenchimento do pedido de um produto em que certas informações devem ser especificadas em determinada ordem. Em tais casos, as estruturas da Figura 17.4 são apropriadas. À medida que o conteúdo e o processamento forem se tornando mais complexos, o fluxo puramente linear na parte esquerda da figura dá lugar a estruturas lineares mais sofisticadas, nas quais pode ser solicitado conteúdo alternativo ou uma mudança de direção para obter conteúdo complementar (a estrutura do lado direito da Figura 17.4).

Quais são os tipos de arquitetura de conteúdo mais comuns?

⁷ O termo *arquitetura da informação* também é usado para conotar estruturas que levam a uma melhor organização, atribuição de nomes, navegação e busca de objetos de conteúdo.

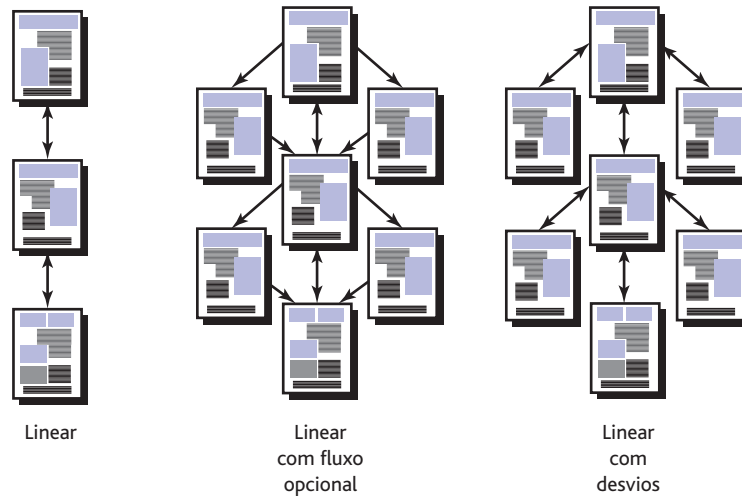


FIGURA 17.4 Estruturas lineares.

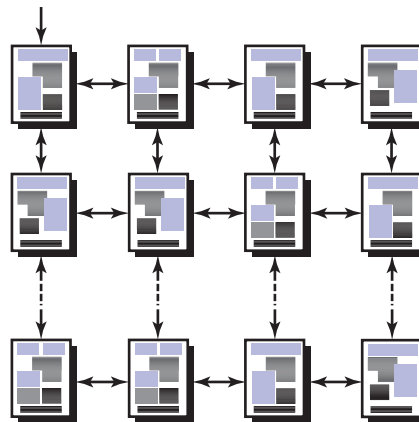
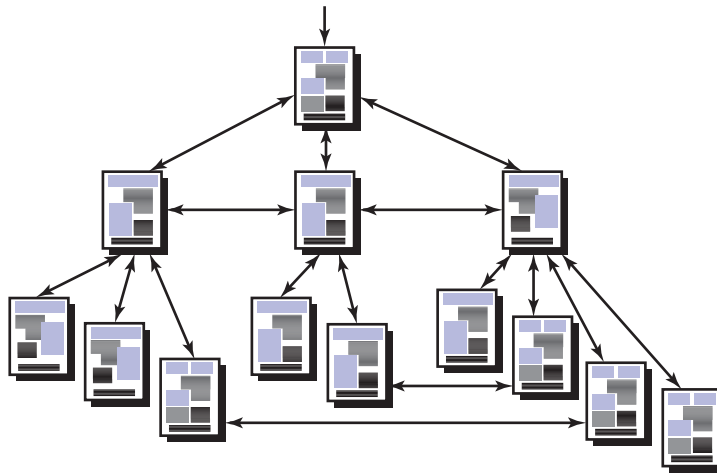
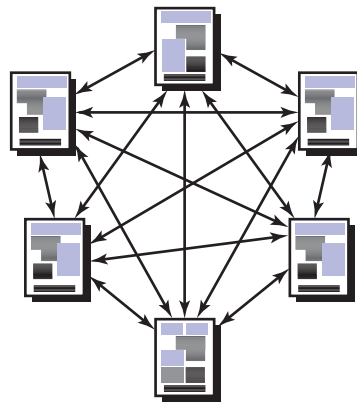


FIGURA 17.5 Estrutura em grade.

O uso de *estruturas em grade* (Figura 17.5) é uma opção arquitetural que podemos aplicar quando o conteúdo da WebApp pode ser organizado em categorias de duas (ou mais) dimensões. Consideremos, por exemplo, uma situação em que um site de comércio eletrônico vende tacos de golfe. A dimensão horizontal da grade representaria o tipo de taco a ser vendido (por exemplo, de madeira, ferro, cunha, curto). A dimensão vertical representa os produtos fornecidos pelos diversos fabricantes de tacos de golfe. Portanto, um usuário poderia navegar horizontalmente pela grade para encontrar a coluna de tacos curtos e, então, verticalmente para examinar os produtos fornecidos pelos fabricantes que vendem tacos curtos. Essa arquitetura de WebApp é útil apenas quando é encontrado conteúdo altamente regular [Pow02].

As *estruturas hierárquicas* (Figura 17.6) são, sem dúvida nenhuma, a arquitetura para WebApp mais comum. Ao contrário das hierarquias de software particionadas, discutidas no Capítulo 13, que estimulam o fluxo de controle

**FIGURA 17.6** Estrutura hierárquica.**FIGURA 17.7** Estrutura em rede.

apenas ao longo das ramificações verticais da hierarquia, uma estrutura hierárquica para WebApp pode ser projetada para possibilitar (via ramificação de hipertexto) o fluxo de controle horizontal ao longo de ramificações verticais da estrutura. Portanto, o conteúdo apresentado no ramo mais à esquerda da hierarquia pode ter links de hipertexto que levem diretamente a conteúdo existente no meio ou no ramo mais à direita da estrutura. Entretanto, deve-se notar que, embora tal ramificação possibilite rápida navegação pelo conteúdo de uma WebApp, ela pode confundir o usuário.

Uma *estrutura em rede* ou "*pura teia*" (Figura 17.7) é similar em muitos aspectos à arquitetura que evoluiu para sistemas orientados a objetos. Os componentes da arquitetura (nesse caso, páginas Web) são projetados de modo que possam passar o controle (via links de hipertexto) para praticamente qualquer outro componente do sistema. Essa estratégia possibilita uma flexibilidade de navegação considerável, mas, ao mesmo tempo, pode ser confusa para o usuário.

As estruturas de arquitetura discutidas nos parágrafos anteriores podem ser combinadas para formar *estruturas compostas*. A arquitetura geral de uma WebApp pode ser hierárquica; porém, parte de sua estrutura poderia apresentar características lineares, enquanto outra poderia ser em rede. Nosso objetivo como projetistas de arquitetura é combinar a estrutura da WebApp com o conteúdo a ser apresentado e o processamento a ser realizado.

17.7.2 Arquitetura de uma WebApp

A arquitetura da WebApp descreve uma infraestrutura que permite a um sistema ou aplicação baseados na Web atingir os objetivos de seu domínio de aplicação. Jacyntho e seus colegas [Jac02b] descrevem as características básicas dessa infraestrutura da seguinte maneira:

As aplicações devem ser construídas usando-se camadas em que diferentes preocupações são levadas em conta; em particular, os dados da aplicação devem ser separados do conteúdo da página (nós de navegação) e, por sua vez, os conteúdos devem estar claramente separados dos aspectos da interface (páginas).

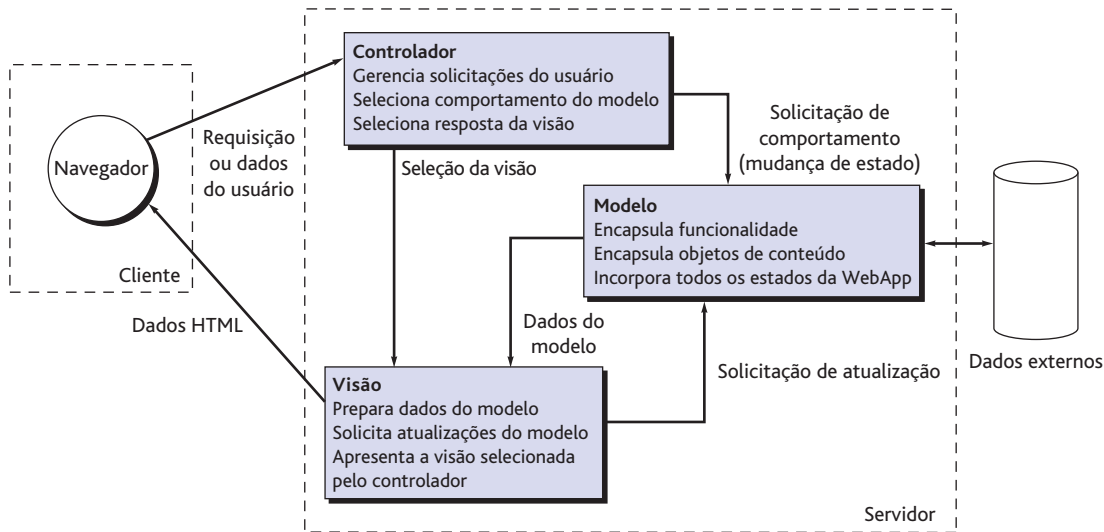
Os autores sugerem uma arquitetura de projeto em três camadas que não associe a interface da navegação e o comportamento da aplicação. Eles argumentam que manter a interface, a aplicação e a navegação separadas simplifica a implementação e aumenta a reutilização.

A arquitetura *Modelo-Visão-Controlador* (MVC, *Model-View-Controller*) [Kra88]⁸ é uma de vários modelos de infraestrutura sugeridos para WebApps que separam a interface do usuário da funcionalidade e do conteúdo de informações de uma WebApp. O *modelo* (algumas vezes conhecido como “objeto-modelo”) contém todo o conteúdo e a lógica de processamento específicos à aplicação, inclusive todos os objetos de conteúdo, acesso a fontes de dados/informações externas e toda a funcionalidade de processamento específica para a aplicação. A *visão* contém todas as funções específicas à interface e possibilita a apresentação do conteúdo e lógica de processamento, inclusive todos os objetos de conteúdo, acesso a fontes de dados/informações externas e toda a funcionalidade de processamento exigida pelo usuário. O *controlador* gerencia o acesso ao modelo e à visão e coordena o fluxo de dados entre eles. Em uma WebApp, “a visão é atualizada pelo controlador com dados do modelo baseados nas informações fornecidas pelos usuários” [WMT02]. Uma representação esquemática da arquitetura MVC aparece na Figura 17.8.

Com referência à figura, as solicitações ou os dados do usuário são manipulados pelo controlador. O controlador também seleciona o objeto visão aplicável, de acordo com a solicitação do usuário. Uma vez determinado o tipo de solicitação, é transmitida uma solicitação de comportamento ao modelo, que implementa a funcionalidade ou recupera o conteúdo necessário para atender à solicitação. O objeto-modelo pode acessar dados armazenados em um banco de dados corporativo, como parte de um repositório de dados local ou de um conjunto de arquivos independentes. Os dados desenvolvidos pelo

A arquitetura MVC separa a interface do usuário da funcionalidade e do conteúdo de informação da WebApp.

⁸ Deve-se observar que a MVC é, na verdade, um padrão de projeto de arquitetura desenvolvido para o ambiente Smalltalk (consulte www.smalltalk.org) e pode ser usado para qualquer aplicação interativa.

**FIGURA 17.8** A arquitetura MVC.

Fonte: Adaptado de [Jac02b].

modelo devem ser formatados e organizados pelo objeto de visão apropriado e transmitidos do servidor de aplicações de volta para o navegador instalado no cliente para exibição na máquina do usuário.

Em muitos casos, a arquitetura da WebApp é definida no contexto do ambiente de desenvolvimento em que a aplicação será implementada. Caso tenha maior interesse, consulte [Fow03] para uma discussão sobre os ambientes de desenvolvimento e seus papéis no projeto de arquiteturas para aplicações para Web.

17.8 Projeto de navegação

Assim que a arquitetura da WebApp tiver sido estabelecida – e os componentes (páginas, scripts, applets e outras funções de processamento) da arquitetura, identificados –, temos de definir os percursos de navegação que permitirão aos usuários acessarem o conteúdo e as funções da WebApp. Para tanto, identificamos a semântica de navegação para diferentes usuários do site e definimos a mecânica (sintaxe) para obter a navegação.

“Apenas espere, Maria, até que a lua surja, e então iremos ver as migalhas de pão que espalhei pelo chão; elas nos indicarão o caminho de volta para casa.”

João e Maria

17.8.1 Semântica de navegação

Assim como muitas atividades de projeto para WebApps, o projeto de navegação começa considerando-se a hierarquia dos usuários e casos de uso relativos (Capítulo 9) desenvolvidos para cada categoria de usuário (ator). Cada ator pode usar a WebApp de forma ligeiramente distinta e, portanto, apresentar diferentes necessidades de navegação. Além disso, os casos de uso desenvolvidos para cada ator vão definir um conjunto de classes englobando um ou mais objetos de conteúdo ou funções de WebApp. À medida que o usuário interage com a WebApp, encontra uma série de *unidades semânticas de navegação* (NSUs, *navigation semantic units*) – “um conjunto de informações e estruturas

Uma NSU descreve os requisitos de navegação para cada caso de uso. Basicamente, a NSU mostra como um ator se movimenta pelos objetos de conteúdo ou funções da WebApp.

de navegação relacionadas que colaboram no cumprimento de um subconjunto de requisitos de usuário relacionados” [Cac02].

Uma NSU é composta por um conjunto de elementos de navegação denominado *modos de navegação* (WoN, *ways of navigating*) [Gna99]. Um WoN representa o melhor percurso de navegação para atingir uma meta de navegação para um tipo de usuário específico. Cada WoN é organizado como um conjunto de *nós de navegação* (NN, *navigational nodes*) interligados por links de navegação. Em alguns casos, um link de navegação pode ser outra NSU. Consequentemente, a estrutura geral de navegação para uma WebApp pode ser organizada como uma hierarquia de NSUs.

Para ilustrarmos o desenvolvimento de uma NSU, consideremos o caso de uso **Selecionar Componentes do CasaSegura**:

Caso de uso: Selecionar Componentes do CasaSegura

A WebApp vai recomendar componentes de produto (por exemplo, painéis de controle, sensores, câmeras) e outras características (por exemplo, funcionalidade baseada em PC implementada por software) para cada cômodo e para a entrada externa. Se eu solicitar alternativas, a WebApp vai fornecê-las, caso existam. Serei capaz de obter informações descritivas e de preços para cada componente do produto. A WebApp criará e exibirá uma lista de materiais à medida que for selecionando vários componentes. Serei capaz de dar um nome à lista de materiais e salvá-la para referência futura (veja o caso de uso **Salvar Configuração**).

Os itens sublinhados na descrição do caso de uso representam classes e objetos de conteúdo que serão incorporados em uma ou mais NSUs, as quais possibilitarão a um novo cliente representar o cenário descrito no caso de uso **Selecionar Componentes do CasaSegura**.

A Figura 17.9 representa uma análise semântica parcial da navegação implícita no caso de uso **Selecionar Componentes do CasaSegura**. Usando a terminologia introduzida anteriormente, a figura também representa uma forma de navegação (WoN) para a WebApp **CasaSeguraGarantida.com**. São mostradas importantes classes de domínio do problema com objetos de conteúdo selecionados (nesse caso, o pacote de objetos de conteúdo chamado DescriçãoDeComponente, um atributo da classe **ComponenteDeProduto**). Esses itens são nós de navegação. Cada uma das setas representa um link de navegação⁹ e é rotulada com a ação iniciada pelo usuário que faz com que o link ocorra.

Podemos criar uma NSU para cada caso de uso associado ao papel de cada usuário. Por exemplo, um **novo cliente** do **CasaSeguraGarantida.com** poderia ter casos de uso diferentes, todos resultando no acesso a diferentes informações e funções de WebApp. É criada uma NSU para cada objetivo.

Durante os estágios iniciais do projeto de navegação, a arquitetura do conteúdo da WebApp é avaliada para determinar um ou mais WoNs para cada

“O problema da navegação em um site é conceitual, técnico, espacial, filosófico e logístico. Consequentemente, as soluções tendem a exigir combinações improvisadas e complexas de arte, ciências e psicologia organizacional.”

Tim Horgan

⁹ Esses são, algumas vezes, conhecidos como *links de semântica de navegação* (NSL, *navigational semantic links*) [Cac02].

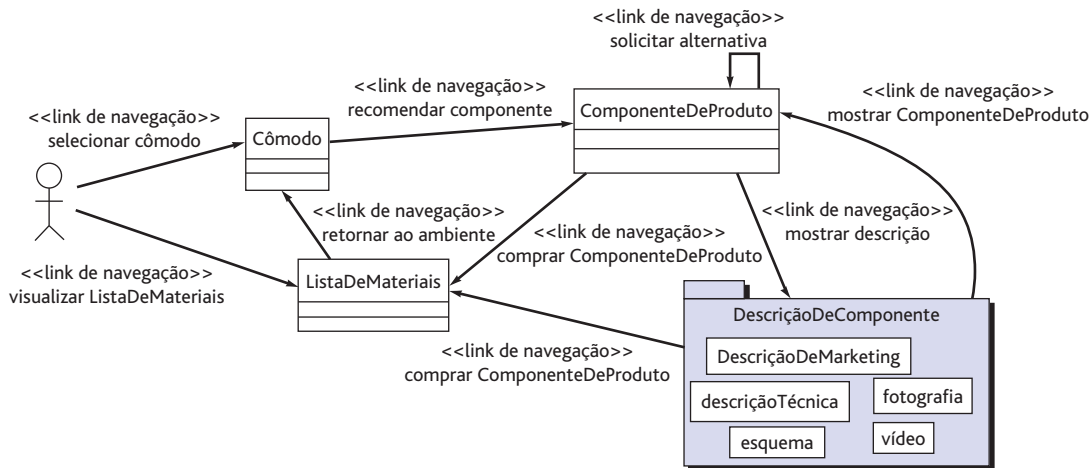


FIGURA 17.9 Criação de uma NSU.

caso de uso. Conforme citado, um WoN identifica nós de navegação (por exemplo, conteúdo) e links que possibilitam a navegação entre eles. Os WoNs são organizados em NSUs.

17.8.2 Sintaxe de navegação

À medida que o projeto prossegue, sua tarefa é definir a mecânica de navegação. A maioria dos sites utiliza uma ou mais das opções de navegação a seguir para implementar cada NSU: links de navegação individuais, barras de navegação horizontais ou verticais (listas), guias ou acesso a um mapa completo do site.

Além de escolhermos a mecânica de navegação, também podemos estabelecer convenções e ferramentas de ajuda de navegação adequadas. Por exemplo, ícones e links gráficos devem ter um aspecto “clicável” por meio de elevação das arestas para conferir à imagem um aspecto tridimensional. Deve-se implementar feedback sonoro ou visual para dar ao usuário uma indicação de que a opção de navegação foi escolhida. Para navegação baseada em elementos textuais, devem-se usar cores para indicar links de navegação e fornecer uma indicação dos links já navegados. Essas são apenas algumas das dezenas de convenções de projeto que tornam a navegação mais amigável.

Na maioria das situações, opte por mecanismos de navegação horizontais ou verticais, mas não ambos.

O mapa de um site deve ser acessível a partir de qualquer página. O próprio mapa deve ser organizado de modo que a estrutura de informações da WebApp estejam prontamente visíveis.

17.9 Projeto em nível de componentes

As WebApps modernas oferecem funções de processamento cada vez mais sofisticadas que: (1) executam processamento localizado para gerar recursos de navegação e conteúdo de forma dinâmica, (2) fornecem recursos de cálculo ou processamento de dados apropriados para o campo de aplicação da WebApp, (3) fornecem sofisticadas consultas e acesso a bancos de dados e (4) estabelecem interfaces de dados com sistemas corporativos externos. Para alcançarmos essas (e muitas outras) capacidades, temos de projetar e construir

componentes de programa que sejam idênticos em sua forma aos componentes para software tradicional.

Os métodos de projeto discutidos no Capítulo 14 se aplicam aos componentes para WebApp com poucas modificações, se houver. O ambiente de implementação, as linguagens de programação e os padrões de projeto, estruturas e software podem variar um pouco, mas a estratégia de projeto geral permanece a mesma.

17.10 Resumo

A qualidade de uma WebApp – definida em termos de usabilidade, funcionalidade, confiabilidade, eficiência, facilidade de manutenção, segurança, escalabilidade e tempo para colocação no mercado – é introduzida durante o projeto. Para alcançar tais atributos de qualidade, um bom projeto de WebApp deve apresentar as seguintes características: simplicidade, consistência, identidade, robustez, navegabilidade e apelo visual. A atividade de projeto da WebApp se concentra em seis elementos distintos do projeto.

O projeto da interface descreve a estrutura e a organização da interface do usuário e abrange uma representação do layout da tela, uma definição dos modos de interação e uma descrição dos mecanismos de navegação. Um conjunto de princípios de projeto da interface e um fluxo de trabalho do projeto de uma interface nos orientam quando o layout e os mecanismos de controle da interface são projetados.

O projeto estético, também denominado design gráfico, descreve “o aspecto” da WebApp e inclui combinação de cores; layout geométrico; tamanho, fonte e posicionamento de textos; o uso de elementos gráficos; e decisões estéticas relacionadas. Um conjunto de diretrizes para design gráfico fornece a base para uma abordagem ao projeto.

O projeto de conteúdo define o layout, a estrutura e um resumo para todo o conteúdo apresentado como parte da WebApp e estabelece as relações entre os objetos de conteúdo. O projeto de conteúdo começa com a representação dos objetos de conteúdo, suas associações e relações. Um conjunto de primitivas de navegação estabelece a base para o projeto de navegação.

O projeto da arquitetura identifica a estrutura de hipermídia geral para a WebApp e engloba tanto a arquitetura do conteúdo quanto a arquitetura da WebApp. Os estilos de arquitetura para conteúdo incluem estruturas lineares, em grade, hierárquicas e em rede. A arquitetura da WebApp descreve uma infraestrutura que permite a um sistema ou aplicação baseados na Web atingir os objetivos de seu domínio de aplicação.

Projeto de navegação cria um fluxo de navegação entre objetos de conteúdo e para todas as funcionalidades da WebApp. A semântica da navegação é definida descrevendo-se um conjunto de unidades semânticas de navegação. Cada unidade é composta por formas de navegação e links e nós de navegação. A sintaxe de navegação representa os mecanismos usados para realizar a navegação descrita como parte da semântica.

O projeto de componentes desenvolve a lógica de processamento detalhada para implementar os componentes funcionais que implementam as fun-

cionalidades da WebApp completa. As técnicas de projeto descritas no Capítulo 14 se aplicam à criação de componentes para WebApps.

Problemas e pontos a ponderar

17.1 Por que o “ideal artístico” é uma filosofia de projeto insuficiente ao se construir WebApps modernas? Existe algum caso em que o ideal artístico é a filosofia a ser seguida?

17.2 Neste capítulo, escolhemos uma ampla variedade de atributos de qualidade para WebApps. Escolha os três que você acredita serem os mais importantes e defenda um argumento que explique por que cada um deve ser enfatizado no trabalho de projeto para WebApps.

17.3 Acrescente pelo menos cinco outras perguntas à checklist de projeto de WebApps de qualidade apresentada na Seção 17.1.

17.4 Você é projetista de WebApps da FutureLearning Corporation, uma empresa de ensino à distância. Você pretende implementar um “mecanismo de ensino” baseado na Internet que deixará à disposição conteúdo de cursos para os alunos. O mecanismo de ensino fornece a infraestrutura básica para transmissão de conteúdo didático sobre qualquer tema (os projetistas de conteúdo prepararão o conteúdo apropriado). Desenvolva um protótipo de projeto da interface para o mecanismo de ensino.

17.5 Qual o site mais esteticamente atraente que você já visitou até hoje e por quê?

17.6 Considere o objeto de conteúdo **Pedido**, gerado assim que o usuário do **CasaSeguraGarantida.com** tenha completado a escolha de todos os componentes e esteja pronto para finalizar sua compra. Desenvolva uma descrição em UML para **Pedido** com todas as representações de projeto apropriadas.

17.7 Qual a diferença entre arquitetura de conteúdo e arquitetura de uma WebApp?

17.8 Reconsiderando o “mecanismo de ensino” da FutureLearning descrito no Problema 17.4, escolha uma arquitetura de conteúdo que seja apropriada para a WebApp. Discuta a razão para ter feito tal escolha.

17.9 Use UML para desenvolver três ou quatro representações de projeto para objetos de conteúdo que seriam encontrados enquanto o “mecanismo de ensino” descrito no Problema 17.4 é projetado.

17.10 Pesquise mais sobre a arquitetura MVC e decida se seria ou não a arquitetura de WebApp apropriada para o “mecanismo de ensino” discutido no Problema 17.4.

17.11 Qual a diferença entre sintaxe de navegação e navegação semântica?

17.12 Defina duas ou três NSUs para a WebApp **CasaSeguraGarantida.com**. Descreva cada uma delas com certo nível de detalhe.

Leituras e fontes de informação complementares

Van Duyne e seus colegas (*The Design of Sites*, 2ª ed., Prentice Hall, 2007) escreveram um livro completo que cobre os mais importantes aspectos do processo de projeto de WebApps. Modelos de processo de projeto e padrões de projeto são vistos em detalhe. Johnson (*Designing with the Mind in Mind: Simple Guide to Understanding User Interface Design Rules*, Morgan Kaufman, 2010) escreveu um livro sobre projeto de interação do usuário com muitos exemplos aplicáveis ao projeto para a Web. Wodtke e Gavella (*Information Architecture: Blueprints for the Web*, 2ª ed., New Riders Publishing, 2009),

Rosenfeld e Morville (*Information Architecture for the World Wide Web*, 3ª ed., O'Reilly & Associates, 2006) e Reiss (*Practical Information Architecture*, Addison-Wesley, 2000) tratam da arquitetura de conteúdo e outros tópicos.

Embora tenham sido escritos centenas de livros sobre “Web design”, poucos discutem métodos técnicos significativos para a realização do trabalho de projeto. Na melhor das hipóteses, são apresentadas várias diretrizes úteis para projeto de WebApps, são mostrados exemplos de páginas Web e programação Java que valem a pena e são discutidos detalhes técnicos importantes para a implementação de WebApps modernas. Dentre as muitas ofertas nessa categoria estão os livros de Butler (*The Strategic Web Designer*, How Books, 2013), Campos (*Web Design Source Book*, PromoPress, 2013), DeFederici (*The Web Designer's Roadmap*, Sitepoint, 2012), Robbins (*Learning Web Design*, O'Reilly Media, 2012), Sklar (*Principles of Web Design*, 5ª ed., Course Technology, 2011), Cederholm (*Bulletproof Web Design*, 3ª ed., New Riders Press, 2011) e Shelly e seus colegas (*Web Design*, 4ª ed., Course Technology, 2011). Livros de Zeldman e Marcotte (*Designing with Web Standards*, 3ª ed., New Riders Publishing, 2009), McIntire (*Visual Design for the Modern Web*, New Riders Press, 2007), Watrall e Siarto (*Head First Web Design*, O'Reilly, 2008), Niederst (*Web Design in a Nutshell*, 3ª ed., O'Reilly, 2006) e Eccher (*Professional Web Design*, Course Technology, 2010; e *Advanced Professional Web Design*, Charles River Media, 2006) também merecem consideração.

Livros como os de Beaird (*The Principles of Beautiful Web Design*, 2ª ed., SitePoint, 2010), Clarke e Holzschlag (*Transcending CSS: The Fine Art of Web Design*, New Riders Press, 2006) e Golbeck (*Art Theory for Web Design*, Addison-Wesley, 2005) enfatizam o projeto estético e devem ser lidos por profissionais com pouca experiência no assunto.

A visão ágil de projeto (e outros tópicos) para WebApps é apresentada por Wallace e seus colegas (*Extreme Programming for Web Projects*, Addison-Wesley, 2003). Conallen (*Building Web Applications with UML*, 2ª ed., Addison-Wesley, 2002) e Rosenberg e Scott (*Applying Use-Case Driven Object Modeling with UML*, Addison-Wesley, 2001) apresentam exemplos detalhados de WebApps modeladas com o emprego da UML.

Uma ampla gama de fontes de informação sobre projeto de WebApps se encontra à disposição na Internet. Uma lista atualizada das referências da Web (em inglês) pode ser encontrada no site: www.mhhe.com/pressman.

Projeto de aplicativos móveis

18

Os dispositivos móveis – smartphones, tablets, dispositivos vestíveis e outros produtos especializados – se tornaram a nova onda da computação. Em agosto de 2012, o *Los Angeles Times* [Rod12] noticiou:

Um novo relatório informa que, pela primeira vez na história, mais americanos possuem smartphones do que telefones normais ou sem fio.

O relatório, publicado pela Chetan Sharma Consulting¹, mostra que a penetração de smartphones ultrapassou a marca dos 50% pela primeira vez nos EUA.

E a tendência não é exclusividade das quatro principais empresas de telefonia dos EUA. Como mostra a GigaOM², operadoras regionais e outras pequenas empresas de telefonia também estão “explorando o tesouro dos smartphones mais baratos do mercado”.

Conceitos-chave

ambientes de mobilidade.	403
aplicativos sensíveis ao contexto.	399
metas.	396
projeto melhores práticas.	401
checklist da qualidade ..	397
computação em nuvem.	405

PANORAMA

O que é? O projeto de aplicativos móveis (MobileApps) abrange atividades técnicas e não técnicas:

estabelecer a percepção e a aparência do aplicativo móvel, criar o layout estético da interface do usuário, estabelecer o ritmo da interação do usuário, definir a estrutura arquitetural geral, desenvolver o conteúdo e a funcionalidade residentes na arquitetura e planejar a navegação que ocorre no aplicativo. É preciso dar atenção especial aos elementos que adicionam reconhecimento de contexto ao aplicativo móvel.

Quem realiza? Engenheiros de software, designers gráficos, desenvolvedores de conteúdo e outros envolvidos participam na criação de um modelo de projeto de um aplicativo móvel.

Por que é importante? O projeto permite criar um modelo que pode ser avaliado em termos de qualidade e aperfeiçoado antes de os códigos e conteúdos serem gerados; os testes serão realizados; e muitos usuários se envolverem. É no projeto que se estabelece a qualidade de um aplicativo móvel.

Quais são as etapas envolvidas? O projeto de um aplicativo móvel é semelhante ao de uma WebApp e abrange seis etapas principais, orientadas por informações obtidas durante a modelagem dos requisitos. O projeto do conteúdo trata dos mesmos problemas, tanto para WebApp quanto para Mobile-

App. Durante o projeto arquitetural, os desenvolvedores de aplicativos móveis determinam quais funções vão ser implementadas no aplicativo nativo que será executado no dispositivo móvel e quais serão implementadas como web services ou serviços da nuvem. O projeto da interface estabelece mecanismos de layout e interação que definem a experiência do usuário. Garantir que o aplicativo móvel utilize o contexto de forma adequada afeta tanto o projeto da interface quanto o projeto do conteúdo. O projeto de navegação define como o usuário navega pela estrutura do conteúdo, e o projeto de componentes representa a estrutura interna detalhada dos elementos funcionais do aplicativo móvel.

Qual é o artefato? Um modelo de projeto abrangendo questões de conteúdo, estética, arquitetura, interface, navegação e de projeto de componentes é o artefato primário gerado durante o projeto de um aplicativo móvel.

Como garantir que o trabalho foi realizado corretamente? Cada elemento do modelo de projeto é revisado na tentativa de descobrir erros, inconsistências ou omissões. Além disso, são consideradas soluções alternativas e também é avaliado até que ponto o modelo de projeto atual irá levar a uma implementação efetiva.

¹ Consulte <http://www.chetanisharma.com/USmarketupdateQ22012.htm>

² Consulte <http://gigaom.com/mobile/carrier-data-confirms-it-half-of-us-now-owns-a-smartphone/>

considerações técnicas . . .	393
desafios	392
desenvolvimento de	
aplicativos móveis	395
erros	401
projeto da interface	
de usuário	398

O GartnerGroup [Gar12] informa que, naquele mesmo trimestre fiscal, 419 milhões de smartphones foram vendidos no mundo e que havia uma comercialização anual projetada de 119 milhões de tablets – um aumento de quase 100% em relação ao ano anterior. A computação móvel se tornou uma força dominante.

18.1 Os desafios

Embora os dispositivos móveis tenham muitas características em comum, frequentemente seus usuários têm percepções muito diferentes dos recursos que esperam encontrar. Alguns esperam os mesmos recursos fornecidos em seus computadores pessoais. Outros dirigem a atenção à liberdade proporcionada pelos dispositivos portáteis e aceitam de bom grado a funcionalidade reduzida da versão móvel de um produto de software conhecido. Outros, ainda, esperam experiências únicas, impossíveis na computação tradicional ou em aparelhos de entretenimento. Para o usuário, a percepção de “eficácia” pode ser mais importante do que qualquer uma das dimensões técnicas da qualidade do aplicativo móvel em si.

18.1.1 Considerações sobre o desenvolvimento

Como todos os dispositivos de computação, as plataformas móveis se diferenciam pelo software que apresentam – uma combinação do sistema operacional (por exemplo, Android ou iOS) e um pequeno subconjunto das centenas de milhares de aplicativos móveis que fornecem uma ampla variedade de funcionalidades. Novas ferramentas permitem que pessoas com pouco treinamento formal criem e vendam aplicativos ao lado de outros, desenvolvidos por grandes equipes de desenvolvedores de software.

Mesmo sendo desenvolvidos por amadores, muitos engenheiros de software acham que os aplicativos móveis estão entre os sistemas de software mais desafiadores em construção na atualidade [Voa12]. As plataformas móveis são muito complexas. Os sistemas operacionais Android e iOS contêm mais de 12 milhões de linhas de código cada um. Muitas vezes, os dispositivos móveis têm mini navegadores que não exibem todo o conjunto de conteúdo disponível em uma página Web. Diferentes dispositivos móveis utilizam diferentes sistemas operacionais e ambientes de desenvolvimento dependentes da plataforma. Os dispositivos móveis tendem a ter tamanhos de tela menores e mais variados do que os computadores pessoais. Isso pode exigir maior atenção aos problemas de projeto de interface do usuário, incluindo decisões de limitar a exibição de algum conteúdo. Além disso, os aplicativos móveis devem ser projetados levando em conta a interrupção intermitente da conectividade, limitações da vida da bateria e outras restrições dos dispositivos³ [Whi08].

Nos ambientes de computação móvel, é provável que os componentes do sistema mudem de lugar, enquanto os aplicativos móveis estão executando. Para manter a conectividade em redes nômades⁴, devem ser desenvolvidos mecanis-

O que torna o desenvolvimento de aplicativos móveis diferente e desafiador?

³ Disponível em <http://www.devx.com/SpecialReports/Article/37693>.

⁴ As *redes nômades* têm conexões variáveis para dispositivos ou servidores móveis.

mos de coordenação para descoberta de dispositivos, troca de informações, manutenção da segurança e da integridade da comunicação e sincronismo de ações.

Além disso, os engenheiros de software devem identificar o balanceamento de projeto correto entre o expressivo poder do aplicativo móvel e as preocupações com a segurança por parte dos envolvidos. Os desenvolvedores devem tentar descobrir algoritmos (ou adaptar os já existentes) que sejam eficientes com relação à energia para preservar a carga da bateria, quando possível. Talvez seja necessário criar middleware para permitir que diferentes tipos de dispositivos móveis se comuniquem nas mesmas redes móveis [Gru00].

Os engenheiros de software devem produzir uma experiência de usuário que tire proveito das características do dispositivo e de aplicativos sensíveis ao contexto. Os requisitos não funcionais (por exemplo, segurança, desempenho, usabilidade) são um pouco diferentes dos que se aplicam a WebApps ou aplicativos de software de desktop. O teste de aplicativos móveis (Capítulo 26) oferece desafios adicionais, pois o usuário espera trabalhar em um grande número de ambientes fisicamente diferentes. Como frequentemente os aplicativos móveis são executados em diversas plataformas de dispositivo, a portabilidade é uma consideração importante. Além disso, o tempo e o esforço para acomodar múltiplas plataformas muitas vezes aumenta o custo global do projeto [Was10].

Sempre existe um balanceamento entre a segurança e outros elementos do projeto de aplicativos móveis.

18.1.2 Considerações técnicas

O baixo custo envolvido na adição de recursos Web em dispositivos comuns, como telefones, câmeras e TVs, está transformando a maneira de acessar informações e usar serviços de rede [Sch11]. Dentre as muitas considerações técnicas a serem tratadas pelos aplicativos móveis, estão as seguintes:

Diversas plataformas de hardware e software. Não é difícil um aplicativo móvel ser executado em muitas plataformas diferentes (tanto móveis quanto fixas) com uma variedade de níveis de funcionalidade. Em parte, os motivos para essas diferenças são o hardware e o software disponíveis, que diferem muito de um dispositivo para outro. Isso aumenta o custo e o tempo de desenvolvimento. Além disso, pode dificultar o gerenciamento da configuração (Capítulo 29).

Quais são as principais considerações técnicas ao se construir um aplicativo móvel?

Muitas estruturas de desenvolvimento e linguagens de programação. Atualmente, os aplicativos móveis estão sendo escritos em pelo menos três linguagens de programação diferentes (Java, Objective C e C#), para pelo menos cinco frameworks de desenvolvimento populares (Android, iOS, BlackBerry, Windows, Symbian) [Was10]. Poucos dispositivos móveis permitem desenvolvimento direto no próprio aparelho. Em vez disso, os desenvolvedores de aplicativos móveis utilizam emuladores, executando em sistemas de desenvolvimento no desktop. Esses emuladores podem ou não refletir com precisão as limitações do dispositivo em si. Frequentemente, é mais fácil portar aplicativos clientes “magros” para vários dispositivos do que aplicativos projetados para executar exclusivamente no dispositivo móvel.

Muitas lojas de aplicativos com diferentes regras e ferramentas. Cada plataforma móvel tem sua própria loja de aplicativos e seus próprios padrões

para aceitar aplicativos (por exemplo, Apple⁵, Google⁶, RIM⁷, Microsoft⁸ e Nokia⁹ publicam seus próprios padrões). O desenvolvimento de um aplicativo móvel para várias plataformas deve ocorrer separadamente, e cada versão do aplicativo precisa de seu próprio especialista em padrões.

Ciclos de desenvolvimento muito curtos. Muitas vezes, os engenheiros de software utilizam processos de desenvolvimento ágeis ao construir aplicativos móveis, na tentativa de reduzir o tempo de desenvolvimento [Was10].

Limitações da interface do usuário e complexidades da interação com sensores e câmeras. Os dispositivos móveis têm telas menores do que os computadores pessoais e um conjunto de possibilidades de interação mais rico (por exemplo, voz, toque, gesto, monitoramento dos olhos), além de utilizar cenários baseados no reconhecimento do contexto. O estilo e a aparência da interface do usuário muitas vezes são impostos pela natureza de ferramentas de desenvolvimento específicas da plataforma [Rot02]. A possibilidade de dispositivos inteligentes interagirem com espaços inteligentes oferece o potencial para se criar plataformas de aplicativo personalizadas, ligadas em rede e de alta fidelidade, como as que se surgem por meio da fusão de smartphones e sistemas de entretenimento informativo de automóveis.¹⁰

Uso eficiente do contexto. Os usuários esperam que os aplicativos móveis apresentem experiências personalizadas, de acordo com o local físico de um dispositivo em relação aos recursos de rede disponíveis. O projeto de interfaces do usuário e os aplicativos sensíveis ao contexto são discutidos com mais detalhes na Seção 18.2.

Gerenciamento de energia. Muitas vezes, a vida da bateria é uma das restrições mais limitantes nos aplicativos móveis. Iluminação traseira, leitura e escrita na memória, uso de conexões sem fio, uso de hardware especializado e velocidade do processador... Tudo isso tem impacto sobre a utilização de energia e precisa ser levado em conta pelos desenvolvedores de software [Mei09].

Modelos e políticas de segurança e privacidade. É difícil proteger a comunicação sem fio contra escuta clandestina. Aliás, impedir *ataques de homem do meio* (MITM, *man-in-the-middle*)¹¹ em aplicativos automotivos pode ser crítico para a segurança dos usuários [Bos11]. Dados armazenados em um dispositivo móvel estão sujeitos a roubo, caso o dispositivo seja perdido ou seja baixado um aplicativo mal-intencionado. Políticas

⁵ <https://developer.apple.com/appstore/guidelines.html>.

⁶ <http://developer.android.com/distribute/googleplay/publish/preparing.html>.

⁷ <https://appworld.blackberry.com/isvportal/guidelines.do>.

⁸ <http://msdn.microsoft.com/en-us/library/ff941089%28v=vs.92%29.aspx>.

⁹ <http://support.publish.nokia.com/?p=64>.

¹⁰ Quando usados no cenário automotivo, os dispositivos inteligentes devem ser capazes de restringir o acesso a serviços que possam distrair o motorista e permitir operação com viva-voz quando o veículo estiver em movimento [Bos11].

¹¹ Esses ataques envolvem um terceiro interceptando a comunicação entre duas fontes confiáveis e a personificação de uma ou de ambas as partes.

de software que aumentam o nível de confiança na segurança e na privacidade de um aplicativo móvel, frequentemente reduzem a utilização do aplicativo e a espontaneidade da comunicação entre os usuários [Rot02].

Limitações computacionais e de armazenamento. Há muito interesse no uso de dispositivos móveis para controlar ambientes domésticos e serviços de segurança. Quando aplicativos móveis podem interagir com dispositivos e serviços em seus ambientes, é fácil sobrecarregar o dispositivo móvel (armazenamento, velocidade de processamento, energia consumida) com o enorme volume de informações [Spa11]. Talvez os desenvolvedores precisem procurar atalhos de programação e meios de reduzir as demandas impostas ao processador e aos recursos de memória.

Aplicativos dependentes de serviços externos. A construção de clientes móveis “magros” sugere a necessidade de contar com provedores de Web service e recursos de armazenamento na nuvem. Isso aumenta a preocupação com acessibilidade e segurança tanto de dados quanto de serviços [Rot02].

Complexidade do teste. É particularmente desafiador testar aplicativos clientes móveis “magros”.¹² Eles apresentam muitos dos mesmos desafios encontrados em WebApps (Capítulo 25), mas trazem preocupações adicionais associadas à transmissão de dados por meio de gateways da Internet e de redes telefônicas [Was10]. O teste de aplicativos móveis será discutido no Capítulo 26.

18.2 Desenvolvimento de aplicativos móveis

Andreou [And05] descreve um modelo de processo de engenharia em espiral para o projeto de aplicativos móveis, contendo seis atividades:

Formulação. Envolve projeto arquitetural, projeto de navegação; as metas, os recursos e as funções do aplicativo móvel são identificados para determinar a abrangência e o tamanho do primeiro incremento. Os desenvolvedores devem estar conscientes das atividades humanas, sociais, culturais e organizacionais que podem revelar aspectos ocultos das necessidades dos usuários e afetar as metas de negócio e a funcionalidade do aplicativo móvel proposto.

Planejamento. São determinados os custos totais e os riscos do projeto. É estabelecida a agenda detalhada, e o processo dos próximos incrementos é documentado.

Análise. São especificados todos os requisitos do usuário móvel e identificados os itens do conteúdo necessários. As ações incluem análise de conteúdo, análise da interação, análise funcional e análise da configuração. É nesse estágio que os desenvolvedores identificam se vão construir um cliente “magro” ou “gordo”. A identificação da natureza das metas do

O desenvolvimento de WebApps utiliza um modelo de processo de engenharia ágil, em espiral.

¹² Os aplicativos móveis executados inteiramente no dispositivo podem ser testados com métodos de teste de software tradicionais (Capítulo 23) ou com emuladores executando em computadores pessoais.

usuário (informativas ou transacionais) ajudará a determinar o tipo de aplicativo móvel a ser desenvolvido.

Engenharia. Envolve projeto arquitetural, projeto de navegação, projeto da interface, projeto do conteúdo e produção de conteúdo. Os engenheiros de software examinam as restrições impostas pelos dispositivos móveis pretendidos, incluindo as considerações impostas pelas tecnologias de rede sem fio escolhidas e a natureza dos Web services necessários para implementar o aplicativo móvel.

Implementação e teste. Durante essa atividade, o aplicativo móvel é codificado e testado. Dentre os problemas que podem tornar os testes um desafio estão: (1) altas taxas de perda, decorrentes da interferência de rádio, e a frequente desconexão, decorrente de problemas de cobertura de rede, (2) frequentes atrasos na transmissão de dados, decorrentes da largura de banda relativamente baixa e (3) preocupações com a segurança, pois os dispositivos móveis são menos seguros e relativamente fáceis de atacar.

Avaliação do usuário. O aplicativo móvel é avaliado quanto à sua usabilidade e acessibilidade; então, começa o processo de formulação do próximo incremento.

O que devemos considerar ao avaliar a qualidade do aplicativo móvel?

Andreou [And05] sugere que a ubiquidade, a personalização, a flexibilidade e a localização devem ser metas de projeto predominantes de todo aplicativo móvel. Os usuários móveis esperam ter a capacidade de receber informações e realizar transações em tempo real, independentemente de seus locais físicos ou do número de usuários concorrentes. Os aplicativos móveis devem apresentar serviços e aplicações que sejam personalizados de acordo com as preferências do usuário. Os usuários de dispositivos móveis devem ser capazes de executar atividades como receber informações ou realizar transações com facilidade. Os usuários móveis devem ter acesso a informações e a serviços locais. Isso significa reconhecer a importância ou o contexto ao se projetar a experiência do usuário do aplicativo móvel.

CASASEGURA



Formulando requisitos de aplicativos móveis

Cena: Uma sala de reunião. A primeira reunião para identificar os requisitos de uma versão móvel da WebApp *CasaSegura*.

Atores: Jamie Lazar, membro da equipe de software; Vinod Raman, membro da equipe de software; Ed Robbins, membro da equipe de software; Doug Miller, gerente da engenharia de software; três membros do Depto. de Marketing; um representante da Engenharia de Produto; e um facilitador.

Conversa:

Facilitador (apontando para um quadro branco): Portanto, esta é a lista atual de objetos e serviços para a função segurança domiciliar presente na WebApp.

Vinod (interrompendo): Então as pessoas querem que a funcionalidade do *CasaSegura* também esteja acessível a partir de dispositivos móveis... inclusive a função de segurança domiciliar?

Representante do Depto. de Marketing: Sim, é isso mesmo... teremos de adicionar essa funcionalidade e tentar torná-la sensível ao contexto para ajudar a personalizar a experiência do usuário.

Facilitador: Sensível ao contexto em que sentido?

Representante do Depto. de Marketing: Talvez as pessoas queiram usar um smartphone, em vez do painel de controle, e não ter de fazer login em um site quando estiverem na garagem de casa. Ou, então, talvez não queiram que todos os membros da família tenham acesso ao painel de controle mestre do sistema a partir de seus telefones.

Facilitador: Você tem dispositivos móveis específicos em mente?

Representante do Depto. de Marketing: Bem, todos os smartphones seria ótimo. Teremos uma versão Web pronta, então o aplicativo móvel não vai funcionar em todos eles?

Jamie: Não exatamente. Se adotarmos a abordagem do navegador do celular, poderemos reutilizar grande parte de nossas WebApps. Mas lembre-se: o tamanho da tela dos smartphones varia e podem ou não ter todos os mesmos recursos de toque. Portanto, no mínimo, teríamos de criar um site para dispositivo móvel que levasse em conta os recursos do dispositivo.

Ed: Talvez devêssemos construir a versão móvel do site primeiro.

Representante do Depto. de Marketing: Certo, mas uma solução de site móvel não era o que tínhamos em mente.

Vinod: Cada plataforma móvel também parece ter seu próprio ambiente de desenvolvimento exclusivo.

Representante da Engenharia de Produto: Podemos restringir o desenvolvimento do aplicativo móvel a apenas um ou dois tipos de smartphones?

Representante do Depto. de Marketing: Acho que pode funcionar. A menos que eu esteja errado, o mercado de smartphones é dominado por duas plataformas atualmente.

Jamie: Há também a questão da segurança. É melhor termos certeza de que um intruso não conseguirá entrar no sistema, desarmá-lo e roubar o local ou coisa pior. Além disso, um telefone pode ser perdido ou roubado com mais facilidade do que um laptop.

Doug: Uma grande verdade.

Representante do Depto. de Marketing: Mas ainda precisamos do mesmo nível de segurança... também deve ser garantido que um estranho não possa ter acesso com um telefone roubado.

Ed: É fácil dizer, o duro é fazer...

Facilitador (interrompendo): Não vamos nos preocupar com esses detalhes ainda.

(Doug, atuando como o secretário da reunião, faz um apontamento apropriado.)

Facilitador: Como ponto de partida, podemos identificar quais elementos da função de segurança da WebApp são necessários no aplicativo móvel e quais precisarão ser criados? Então, poderemos decidir quantas plataformas móveis vamos suportar e quando poderemos ir adiante nesse projeto.

(O grupo gasta os 20 minutos seguintes refinando e expandindo os detalhes da função segurança domiciliar.)

18.2.1 Qualidade do aplicativo móvel

Na realidade, quase toda dimensão e fator da qualidade discutidos no Capítulo 19 se aplicam aos aplicativos móveis. Contudo, Andreou [And05] sugere que a satisfação do usuário com um aplicativo móvel é ditada por seis importantes fatores da qualidade: funcionalidade, confiabilidade, usabilidade, eficiência, facilidade de manutenção e portabilidade.

INFORMAÇÕES



Aplicativos móveis – checklist da qualidade

A checklist a seguir fornece um conjunto de perguntas que ajudarão os engenheiros de software e os usuários a avaliar a qualidade global de um aplicativo móvel:

- Opções de conteúdo e/ou função e/ou navegação podem ser ajustadas às preferências dos usuários?
- O conteúdo e/ou funcionalidade podem ser personalizados para a largura de banda em que o usuário se comunica? O aplicativo leva em conta sinais fracos ou perda de sinal de maneira aceitável?
- Opções de conteúdo e/ou função e/ou navegação podem se tornar sensíveis ao contexto de acordo com as preferências dos usuários?
- Foi dada consideração adequada à disponibilidade de energia no(s) dispositivo(s)-alvo?
- Elementos gráficos, mídia (áudio, vídeo) e outros Web services ou da nuvem foram usados apropriadamente?
- O projeto geral de páginas é fácil de ler e navegar? O aplicativo leva em conta as diferenças de tamanho de tela?
- A interface do usuário é adequada aos padrões de exibição e interação adotados para o dispositivo (ou dispositivos) móvel pretendido?
- O aplicativo atende às expectativas de confiabilidade, segurança e privacidade de seus usuários?
- Quais provisões foram feitas para garantir que um aplicativo permaneça atualizado?
- O aplicativo móvel foi testado em todos os ambientes alvos do usuário e para todos os dispositivos pretendidos?

18.2.2 Projeto de interface de usuário

Os usuários de dispositivos móveis esperam que o mínimo de tempo de aprendizado seja necessário para dominar um aplicativo móvel. Para isso, os projetistas utilizam representações e posicionamento padrão de ícones entre várias plataformas. Além disso, eles devem ser sensíveis à expectativa de privacidade do usuário com relação à exibição de informações pessoais na tela do dispositivo móvel. Interfaces de toque e gesto, junto com entrada de voz sofisticada, estão amadurecendo rapidamente [Shu12] e já fazem parte da caixa de ferramentas do projetista de interface de usuário.

A acessibilidade é uma questão de projeto importante e deve ser considerada no projeto centrado no usuário.

A pressão jurídica e ética para fornecer acesso a todas as pessoas sugere que as interfaces de dispositivos móveis precisam levar em conta diferenças de marca, diferenças culturais, diferenças na experiência de computação, usuários idosos e usuários com necessidades especiais (por exemplo, visuais, auditivas, motoras). Os efeitos de uma capacidade de utilização insatisfatória podem ser os usuários não conseguirem concluir suas tarefas ou não ficarem satisfeitos com os resultados. Isso sugere a importância das atividades de projeto centrado no usuário em cada uma das áreas de utilização (interface do usuário, interface acessória externa e interface de serviço). Para atender às expectativas de utilização dos envolvidos, os desenvolvedores de aplicativos móveis devem tentar responder às perguntas a seguir para avaliar a imediata prontidão do dispositivo:

- A interface do usuário é consistente entre aplicativos?
- O dispositivo pode operar em conjunto com diferentes serviços de rede?
- O dispositivo é aceitável, em termos dos valores dos envolvidos¹³ no mercado-alvo?

Eisenstein [Eis01] afirma que o uso de modelos abstratos, neutros quanto a plataforma, para descrever uma interface de usuário facilita muito o desenvolvimento de interfaces consistentes, multiplataforma, para dispositivos móveis. Denominada *projeto baseado em modelo*, essa abordagem utiliza três modelos diferentes. Um *modelo de plataforma* descreve as restrições impostas por cada plataforma a ser suportada. Um *modelo de apresentação* descreve a aparência da interface do usuário. O *modelo de tarefa* é uma representação estruturada das tarefas que o usuário precisa executar para atingir suas metas. No melhor caso, o projeto baseado em modelo (Capítulo 12) envolve a criação de bancos de dados que contêm os modelos e tem o suporte de ferramentas para gerar automaticamente interfaces de usuário para vários dispositivos. A utilização de técnicas de projeto baseado em modelo também pode ajudar os projetistas a reconhecer e adaptar os contextos exclusivos e as mudanças de contexto presentes na computação móvel. Sem uma descrição abstrata de uma interface de usuário, o desenvolvimento de interfaces móveis é propenso a erros e demorado.

"Os aplicativos móveis obrigam os projetistas a entender profundamente as necessidades do usuário para fornecer as funções corretas em uma interface que pode ser aprendida e útil."

Ben Schneiderman

18.2.3 Aplicativos sensíveis ao contexto

O contexto permite a criação de novos aplicativos baseados no local do dispositivo móvel e da funcionalidade a ser apresentada por ele. O contexto tam-

¹³ Marca, preferências éticas, preferências morais, crenças cognitivas.

INFORMAÇÕES



Considerações sobre o projeto de interface de usuário de aplicativos móveis

As escolhas de projeto afetam o desempenho e devem ser examinadas no início do processo do projeto da interface do usuário. Ivo Weevers [Wee11] divulgou várias práticas de projeto de interface de usuário móvel que se mostraram úteis no projeto de aplicativos móveis:

- **Defina assinaturas de marca da interface do usuário.** Diferencie o aplicativo de seus concorrentes. Torne os elementos básicos da assinatura da marca os mais responsivos possível, pois os usuários os utilizarão repetidamente.
- **Concentre-se no portfólio de produtos.** Tenha como alvo as plataformas mais importantes para o sucesso do aplicativo e da empresa. Nem todas as plataformas têm o mesmo número de usuários.
- **Identifique histórias de usuário básicas.** Utilize técnicas como *Quality Function Deployment (QFD)** (Capítulo 8) para reduzir uma longa lista de requisitos a implementar usando os recursos restritos dos dispositivos móveis.
- **Otimize fluxos e elementos da interface do usuário.** Os usuários não gostam de esperar. Identifique gargalos em potencial no fluxo de trabalho do usuário e certifique-se de que o usuário receba uma indicação do andamento, quando ocorrerem atrasos. Verifique se o tempo de exibição de elementos de tela é justificado, em termos de benefícios para o usuário.
- **Defina regras de escala.** Determine as opções que serão utilizadas quando a informação a ser exibida for grande demais para a tela. O gerenciamento da funcionalidade, da estética, da utilização e do desempenho é um ato contínuo de equilíbrio.
- **Use um painel de desempenho.** Utilizado para comunicar o estado atual de conclusão do produto (por exemplo, o número de histórias de usuário implementadas), seu desempenho relativo às suas metas e, talvez, comparações com seus concorrentes.
- **Promova habilidades especiais de engenharia de interface de usuário.** É importante entender que a implementação de layout, elementos gráficos e animação têm implicações no desempenho. Técnicas para intercalar a representação de itens na tela e a execução do programa podem ser úteis.

bém pode ajudar a personalizar aplicativos de computador pessoal para dispositivos móveis (por exemplo, baixar informações sobre um paciente em um dispositivo transportado por um profissional de saúde domiciliar quando ele chega à casa da pessoa).

Usar interfaces contextuais altamente adaptáveis é uma boa maneira de lidar com as limitações do dispositivo (por exemplo, tamanho da tela e memória). Facilitar o desenvolvimento de interação de usuário sensível ao contexto exige o suporte de arquiteturas de software correspondentes.

Em uma discussão sobre aplicativos sensíveis ao contexto, Rodden [Rod98] assinala que a computação móvel mescla os mundos real e virtual, fornecendo funcionalidade que permite a um dispositivo ser sensível a sua localização, tempo e outros objetos próximos. O dispositivo poderia estar em um local fixo, como um sensor de alarme, incorporado a um dispositivo autônomo ou ser transportado por uma pessoa. Como pode ser projetado para ser usado por indivíduos, grupos ou pelo público, o dispositivo deve detectar a presença e a identidade do usuário, assim como os atributos do contexto relevantes ou permitidos para esse usuário (mesmo que o usuário seja outro dispositivo).

Para reconhecerem o contexto, os sistemas móveis precisam produzir informações confiáveis na presença de dados incertos e rapidamente mutantes de uma variedade de fontes heterogêneas. É um desafio extrair informações de contexto relevantes pela combinação de dados de vários sensores,

Como um projeto consegue reconhecer o contexto?

* N. de R. T.: *Quality Function Deployment (QFD)* é uma técnica para priorização das necessidades dos clientes.

devido a problemas de ruído, calibração errada, desgaste, danos e clima. É preferível usar comunicação baseada em eventos no gerenciamento de fluxos contínuos de dados de alto nível de abstração em aplicativos sensíveis ao contexto [Kor03].

Em ambientes de computação ubíquos, vários usuários trabalham com uma ampla diversidade de dispositivos. A configuração dos dispositivos deve ser flexível o suficiente para mudar com frequência, devido às práticas da maneira móvel de trabalhar. É importante que a infraestrutura de software suporte diferentes estilos de interação (por exemplo, gestos, voz e caneta eletrônica) e os armazene em abstrações que possam ser facilmente compartilhadas.

Há ocasiões em que um usuário quer trabalhar com mais de um dispositivo simultaneamente no mesmo produto (por exemplo, usar um dispositivo touch screen para editar uma imagem em um documento e um teclado pessoal para editar o texto do documento). É desafiador integrar dispositivos móveis que nem sempre estão conectados na rede e têm restrições diversas [Tan01]. Os jogos multijogadores têm de lidar com esses problemas, armazenando o estado do jogo em cada dispositivo e compartilhando as informações sobre alterações entre os dispositivos de outros jogadores em tempo real.

18.2.4 Lições aprendidas

Anteriormente neste capítulo, mencionamos diversas diferenças importantes entre os aplicativos móveis e software convencional. Como consequência dessas diferenças, os engenheiros de software precisam modificar e ampliar técnicas convencionais para analisar, projetar, construir e testar aplicativos móveis. De Sá e Carrico [Des08] sugerem diversas lições aprendidas.

Os cenários de utilização (Capítulo 15) de aplicativos móveis devem considerar as variáveis contextuais (localização, usuário e dispositivo) e as transições entre os cenários contextuais (por exemplo, o usuário sai do banheiro e entra na cozinha ou troca a caneta eletrônica pelo dedo). De Sá e Carriço identificaram um conjunto de tipos de variável que devem ser considerados no desenvolvimento de cenários de usuário – localizações e ambientes, movimento e postura, dispositivos e utilizações, volumes de trabalho e distrações, preferências do usuário.

*Observação etnográfica*¹⁴ é um método amplamente usado para reunir informações sobre usuários representativos de um produto de software quando ele está sendo projetado. Muitas vezes é difícil observar os usuários quando eles mudam os contextos, pois o observador precisa segui-los por longos períodos de tempo – algo que pode trazer preocupações com a privacidade.¹⁵ Um fator complicador é que, às vezes, os usuários executam tarefas em ambientes privados e sociais de formas diferentes em cada caso. Os mesmos usuários tal-

Casos de uso podem funcionar bem no desenvolvimento de aplicativos móveis, mas as variáveis de contexto devem ser consideradas ao desenvolvê-los.

¹⁴ *Observação etnográfica* é uma maneira de determinar a natureza de tarefas de usuário, observando-os em seus ambientes de trabalho.

¹⁵ Talvez seja suficiente os usuários preencherem questionários anônimos, quando não for possível uma observação direta.

vez precisem ser observados executando tarefas em vários contextos, enquanto se monitora transições e se registra suas reações às mudanças.

Protótipos de interface de usuário iniciais para aplicativos móveis podem ser criados no papel, usando-se esboços ou uma combinação de fichas e/ou anotações em adesivos para simular os mecanismos de interação importantes. O segredo é permitir que todos os mecanismos de interação sejam avaliados, todos os contextos de utilização sejam examinados e todos os mecanismos de interação do usuário sejam especificados. Além disso, também podem ser simulados o uso de widgets de interação e o posicionamento e a localização global na tela. Esses protótipos iniciais aproximados em papel podem ajudar na descoberta de erros, inconsistências e omissões antes que os dispositivos móveis pretendidos entrem em ação. Então, protótipos posteriores podem ser criados para serem executados nos dispositivos móveis pretendidos quando os problemas de layout e posicionamento estiverem resolvidos.

INFORMAÇÕES



Erros de projeto de aplicativos móveis

Joh Koester [Koe12] publica vários exemplos de práticas de projeto de aplicativo móvel que devem ser evitadas:

- **Miscelânea.** Não adicione recursos demais ao aplicativo e widgets demais à tela. Simples é inteligível. Simples é comercializável.
- **Inconsistência.** Para evitar isso, defina padrões para navegação de páginas, uso de menus, botões, guias e outros elementos de interface de usuário. Prefira uma aparência e comportamento uniformes.
- **Projeto demasiado.** Seja implacável ao projetar um aplicativo móvel. Remova elementos desnecessários e elementos gráficos que causam muito consumo. Não caia na tentação de adicionar elementos apenas porque você acha que deve.
- **Falta de velocidade.** Os usuários não se importam com restrições de dispositivo – eles querem ver as coisas rapidamente. Carregue previamente o que for possível. Elimine o que não for necessário.
- **Verbosidade.** Menus e exibições de tela desnecessariamente longos e prolixos são indicações de um aplicativo móvel que não foi testado com usuários e de desenvolvedores que não passaram tempo suficiente entendendo a tarefa do usuário.
- **Interação não padronizada.** Um motivo para ter uma plataforma como alvo é aproveitar a experiência do usuário com o modo de fazer as coisas nela. Onde existirem padrões, utilize-os. Isso precisa ser equilibrado com a necessidade de fazer com que um aplicativo se pareça e se comporte da mesma maneira em vários dispositivos, quando possível.
- **"Ajudites e perguntites".** Adicionar ajuda online não é a maneira de reparar uma interface de usuário mal projetada. Certifique-se de ter testado seu aplicativo com os usuários da plataforma-alvo e de ter reparado os defeitos identificados.

18.3 Projeto de aplicativos móveis – boas práticas

Existem várias diretrizes para o desenvolvimento de aplicativos móveis¹⁶ e de aplicativos para plataformas específicas, como iOS¹⁷ da Apple ou Android do Google¹⁸. Schumacher [Sch09] reuniu muitas ideias de melhores práticas e divulgou várias, especialmente adaptadas ao projeto de um aplicativo móvel,

¹⁶ <http://www.w3.org/TR/mwabp/>.

¹⁷ <https://developer.apple.com/library/iOS/navigation>.

¹⁸ <http://developer.android.com/guide/components/index.html>.

Quais são as coisas em que devo pensar ao projetar aplicativos móveis touch screen?

como páginas Web.¹⁹ Algumas considerações importantes ao se projetar aplicativos móveis touch screen, listadas por Schumacher, incluem:

- **Identificar seu público.** O aplicativo deve ser escrito tendo-se em mente as expectativas e as experiências de seus usuários. Usuários experientes querem fazer as coisas rapidamente. Os menos experientes gostarão de uma abordagem guiada ao usarem o aplicativo pela primeira vez.
- **Projetar tendo em vista o contexto de uso.** É importante considerar como o usuário vai interagir com o mundo real enquanto utiliza o aplicativo. O ato de assistir a um filme em um avião exige uma interface de usuário diferente de uma utilizada para verificar o clima antes de sair do escritório.
- **Há uma linha tênue entre simplicidade e preguiça.** Criar uma interface de usuário intuitiva em um dispositivo móvel é muito mais difícil do que simplesmente remover recursos encontrados na interface de usuário do aplicativo em execução em um dispositivo maior. A interface do usuário deve fornecer toda a informação que permita a ele tomar sua próxima decisão.
- **Usar a plataforma como uma vantagem.** Navegação touch screen não é intuitiva e deve ser aprendida por todos os novos usuários. Esse aprendizado será mais fácil se os projetistas da interface do usuário obedecerem aos padrões definidos para a plataforma.
- **Tornar as barras de rolagem e o realce de seleção mais salientes.** Barras de rolagem frequentemente são difíceis de localizar em dispositivos de toque, pois são pequenas demais. Certifique-se de que as bordas de menus ou ícones sejam largas o suficiente para mudanças de cor, para chamar a atenção dos usuários. Ao usar codificação em cores, certifique-se de haver contraste suficiente entre as cores de primeiro e segundo plano para permitir que sejam distinguidas por usuários daltônicos.
- **Aumentar a capacidade de descoberta de funcionalidade avançada.** Às vezes, teclas de atalho e outros atalhos são incluídos em aplicativos móveis para permitir que usuários experientes concluam suas tarefas com maior rapidez. Você pode aumentar a capacidade de descoberta de recursos como esses incluindo dicas de design visual na interface do usuário.
- **Usar rótulos limpos e coerentes.** Os rótulos de widget devem ser reconhecidos por todos os usuários do aplicativo, independentemente dos padrões utilizados por plataformas específicas. Use abreviações de forma cautelosa e, se possível, evite-as.
- **Nunca se deve desenvolver ícones inteligentes à custa do entendimento do usuário.** Às vezes, os ícones só fazem sentido para seus projetistas. Os usuários devem ser capazes de saber seus significados rapidamente. É difícil garantir que os ícones sejam significativos em todos os idiomas e grupos de usuários. Uma boa estratégia para melhorar o reconhecimento é adicionar um rótulo textual embaixo de um ícone novo.

¹⁹ <http://www.usercentric.com/news/2011/06/15/best-practices-designing-mobile-touch-screen-applications>.

- **Atender às expectativas do usuário quanto a personalização.** Os usuários de dispositivos móveis esperam ser capazes de personalizar tudo. No mínimo, os desenvolvedores devem permitir que os usuários definam seus locais (ou o detectem automaticamente) e selecionem opções de conteúdo que possam estar disponíveis nesse local. É importante indicar para os usuários quais recursos podem ser personalizados e como podem personalizá-los.
- **Formulários longos abrangem várias telas em dispositivos móveis.** Os usuários de dispositivo móvel experientes querem todas as informações em uma única tela de entrada, mesmo que isso exija rolagem. Os usuários iniciantes frequentemente se tornam experientes rapidamente e se aborrecem com várias telas de entrada.

O desenvolvimento de aplicativos nativos para várias plataformas de dispositivo pode ser dispendioso e demorado. Os custos de desenvolvimento podem ser reduzidos pelo uso de tecnologias conhecidas dos desenvolvedores Web (por exemplo, JavaScript, CSS e HTML) para criar aplicativos móveis que serão acessados com um navegador Web no dispositivo móvel. Open webOS²⁰ é uma plataforma independente de dispositivo feita para permitir esse tipo de desenvolvimento.

18.4 Ambientes de mobilidade

O quadro contém indicações sobre várias ferramentas que podem ser usadas no desenvolvimento de aplicativos móveis para plataformas populares. Cada uma tem suas vantagens e desvantagens.²¹ Algumas usam tecnologias restritas a dispositivos de um único fabricante (por exemplo, iOS e Objective C). Algumas plataformas são licenciadas para vários fabricantes (por exemplo, Android e Java ou Windows 8 e C#). Algumas são de código-fonte aberto e projetadas para funcionar em muitos dispositivos (por exemplo, webOS e Enyo). Cada plataforma tem suas regras próprias de comercialização e distribuição, e cada uma varia no grau com que suporta tecnologias de aplicativo específicas, como jogos.

A escolha de uma (ou mais) plataforma exige uma concepção cuidadosa por parte dos desenvolvedores de aplicativos móveis. Às vezes, a plataforma (ou plataformas) escolhida será imposta por objetivos de negócio do cliente. Em outras situações, as escolhas de plataforma serão determinadas pelos recursos do dispositivo que suportam ou pelas limitações de hardware existentes. Yuan [Yua02] utiliza os seguintes critérios para avaliar vários *ambientes de desenvolvimento interativo móvel* (MIDEs, *mobile interactive development environments*):

- **Recursos de produtividade gerais.** O MIDE deve conter ferramentas para suportar edição, gerenciamento de projeto, depuração, design arquitetural, documentação e testes de unidade.

Como devo escolher ambientes de mobilidade e plataformas?

²⁰ Informações sobre webOS podem ser encontradas em <https://developer.palm.com>.

²¹ Discussão adicional pode ser encontrada em http://www.cs.colorado.edu/~kena/classes/5828/s10/presentations/software_engineering_mobile.pdf.

INFORMAÇÕES



Ferramentas de desenvolvimento de aplicativos móveis

Muitas das ferramentas de desenvolvimento de aplicativos móveis para dispositivos populares estão disponíveis como downloads gratuitos na Web.²²

<https://developer.apple.com/resources> – O iOS Dev Center da Apple contém ferramentas que podem ser usadas no desenvolvimento de aplicativos para iPod, iPad e iPhone.

<http://developer.android.com/index.html> – Esse site fornece um plug-in para permitir desenvolvimento com Android usando o ambiente de programação Eclipse.

<https://developer.blackberry.com/java/download/eclipse/> – Esse site fornece um plug-in para permitir desenvolvimento com BlackBerry usando o ambiente de programação Eclipse.

<http://eclipse.org/> – Site de download para o Eclipse Programming Environment.

<http://create.msdn.com/en-US/> – Ferramentas da Microsoft para desenvolvimento de aplicativos para Windows Phone e jogos para Xbox.

http://www.developer.nokia.com/Develop/Java/Tools/Series_40_platform_SDKs/ – Site de download para várias ferramentas de desenvolvimento com Nokia baseadas em Java.

<https://developer.palm.com/content/resources/> – Site de download para o ambiente de desenvolvimento com webOS da HP.

<http://enyojs.com/> – Site de download para o ambiente de desenvolvimento multiplataforma Enyo.

<http://www.scirra.com/construct2> – Site de download para o ambiente de desenvolvimento de jogos multiplataforma Construct2.

- **Integração com SDK de terceiros.** É provável que cada serviço de rede e da nuvem exija o uso de uma API ou de um SDK específico. É mais fácil trabalhar em apenas um IDE do que em vários.
- **Ferramentas pós-compilação.** Um MIDE eficiente contém ferramentas que permitem otimizar o código-fonte de um aplicativo concluído para um dispositivo ou serviço móvel específico.
- **Suporte para distribuição sem fio.** Um bom MIDE deve permitir o teste do aplicativo distribuído dentro do ambiente de desenvolvimento. Isso pode ser complicado quando o aplicativo móvel precisa acessar Web services ou outros aplicativos.
- **Desenvolvimento de aplicativos móveis de fim-a-fim.** Frequentemente, os dispositivos móveis não são poderosos o bastante para processar ou armazenar grandes volumes de informação de forma local. É importante permitir que os desenvolvedores criem, testem e distribuam projetos móveis inteiros usando um MIDE de desktop.
- **Documentação e tutoriais.** Mesmo as ferramentas de desenvolvimento gratuitas precisam ser fáceis de aprender e usar. É fundamental ter materiais de apoio adequados.
- **Construtores de interfaces de usuário gráficas.** Se o MIDE suporta construção visual de telas de usuário, protótipos podem ser feitos e testados rapidamente.

Conforme já mencionamos, é difícil projetar e implementar um aplicativo móvel que execute perfeitamente em várias plataformas. Isso é particu-

²² A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

larmente verdade para aplicativos no desenvolvimento de jogos móveis (um setor que gera bilhões de dólares). Os games mais populares são desenvolvidos em paralelo para vários dispositivos móveis. Esse tipo de fragmentação do desenvolvimento aumenta os custos e enfatiza a necessidade de uma melhor padronização de ferramentas e APIs de desenvolvimento. Galavas [Gal11] menciona que portabilidade, funcionalidade, velocidade de desenvolvimento e desempenho são critérios de seleção importantes ao se considerar as plataformas de desenvolvimento móvel a usar.

Middleware de computação móvel pode ser usado para facilitar a comunicação e a coordenação de componentes de sistema distribuídos. Isso permite que os desenvolvedores de aplicativos móveis contem com abstrações que ocultem os detalhes de parte das complexidades dos ambientes móveis. Para que middleware seja útil no desenvolvimento de aplicativos móveis, tanto o cliente móvel quanto o provedor de serviço devem permitir conexões intermitentes assíncronas. Os componentes do middleware em execução no cliente móvel não devem consumir recursos computacionais significativos no dispositivo móvel. O middleware também deve ajudar o aplicativo móvel a atingir o nível de reconhecimento do contexto exigido por seus usuários [Mas02].

INFORMAÇÕES



Middleware de aplicativo móvel

Os produtos de middleware a seguir representam aqueles desenvolvidos especificamente para aplicativos móveis:

<http://www.infrae.com/products/mobi> – O middleware móvel Mobi é um conjunto de bibliotecas e componentes WSGI que interagem entre um servidor Web e aplicativos que tornam dados móveis disponíveis.

<http://smartsoftmobile.com/> – A SmartSoft Mobile Solutions fornece soluções empresariais e baseadas na nuvem (por exemplo, SAP) para plataformas de dispositivos móveis.

<http://www.sybase.com/> – A Sybase fornece uma Mobile Enterprise Application Platform (MEAP) que oferece ferramentas e middleware cliente-servidor para desenvolvimento de aplicativos móveis e empresariais. Consulte também: <http://scn.sap.com/community/mobile>.

<http://code.google.com/p/skeenzzone/> – SkeenZone é um middleware Java leve e extensível que permite o desenvolvimento de aplicativos móveis distribuídos.

<http://modolabs.com/platform> – Kurogo é uma plataforma de código-fonte aberto projetada para acionar sites móveis multifacetados, ricos em conteúdo e aplicativos para iPhone e Android.

18.5 A nuvem

A computação de serviços²³ e a computação em nuvem²⁴ permitem o desenvolvimento rápido de aplicativos distribuídos em larga escala [Yau11]. Esses paradigmas da computação tornaram mais fácil e econômico criar aplicativos em muitos dispositivos diferentes (computadores pessoais, smartphones e tablets). Os dois paradigmas permitem a terceirização de recursos e a transferência de informações do gerenciamento da tecnologia da informação para

²³ A *computação de serviços* se concentra no projeto arquitetural e permite o desenvolvimento de aplicativos por meio de descoberta e composição de serviços.

²⁴ A *computação em nuvem* se concentra na distribuição efetiva de serviços para os usuários, por meio de virtualização flexível e escalável de recursos e do balanceamento de carga.

"A engenharia de software voltada a serviços incorpora as melhores características dos paradigmas da computação de serviços e na nuvem."

Stephan Yau

"Como os negócios dos usuários contam fortemente com provedores de serviço, existem sérias preocupações quanto a como as ameaças à confiabilidade do serviço poderiam afetá-los e, consequentemente, afetar o negócio do usuário da nuvem."

Stephan Yau

provedores de serviço, ao mesmo tempo que reduz o impacto das limitações de recursos em alguns dispositivos móveis. Uma arquitetura voltada para serviços fornece o estilo arquitetural (por exemplo, REST²⁵), os protocolos padronizados (por exemplo, XML²⁶, SOAP²⁷) e as interfaces (por exemplo, WSDL²⁸) necessários para o desenvolvimento de aplicativos móveis. A computação em nuvem permite o acesso de rede conveniente e sob demanda a um pool compartilhado de recursos de computação que podem ser configurados (servidores, armazenamento, aplicativos e serviços).

A *computação de serviço* desobriga os desenvolvedores de aplicativos móveis a integrar código-fonte de serviço no cliente que está sendo executado em um dispositivo móvel. Em vez disso, o serviço é executado fora do servidor do provedor e é pouco acoplado aos aplicativos que o utilizam por meio de protocolos de troca de mensagens. Um serviço típico fornece uma interface de programas aplicativos (API, application programming interface) para permitir que seja tratado como uma caixa preta abstrata.

A *computação em nuvem* permite que o cliente (um usuário ou um programa) solicite recursos de computação de acordo com a necessidade, além dos limites da rede, em qualquer lugar ou a qualquer momento. A arquitetura da nuvem tem três camadas, cada uma das quais podendo ser chamada como um serviço. A camada *software como serviço* consiste em componentes de software e aplicativos hospedados por outros provedores de serviço. A camada *plataforma como serviço* fornece uma plataforma de desenvolvimento colaborativo para dar suporte ao projeto, à implementação e aos testes feitos por membros da equipe, distribuídos geograficamente. A *infraestrutura como serviço* fornece recursos de computação virtuais (armazenamento, poder de processamento, conectividade de rede) na nuvem.

Os dispositivos móveis podem acessar serviços da nuvem a partir de qualquer lugar, a qualquer momento. Os riscos de roubo de identidade e sequestro de serviços exigem que os provedores de serviços móveis e de computação em nuvem empreguem técnicas de engenharia de segurança rigorosas para proteger seus usuários. As preocupações com segurança e privacidade associadas à computação em nuvem são discutidas no Capítulo 27. O uso de um serviço de nuvem neutro com relação ao fornecedor pode tornar mais fácil criar aplicativos multiplataforma [Rat12].

Taivalsaari [Tai12] mostra que fazer uso de armazenamento na nuvem pode tornar possível que qualquer dispositivo móvel ou recursos de software sejam atualizados facilmente em milhões de dispositivos em todo o mundo. Na verdade, é possível virtualizar toda a experiência do usuário móvel, de modo que todos os aplicativos sejam baixados da nuvem.

²⁵ *Representation State Transfer* descreve um estilo arquitetural web em rede onde a representação do recurso (por exemplo, uma página Web) coloca o cliente em um novo estado. O cliente muda ou transfere o estado com cada representação de recurso.

²⁶ A *Extensible Markup Language*, XML, foi projetada para armazenar e transportar dados, enquanto a HTML foi projetada para exibir dados.

²⁷ *Simple Object Access Protocol* é uma especificação de protocolo para troca de informações estruturadas na implementação de Web Services em redes de computador.

²⁸ *Web Services Description Language* é uma linguagem baseada na XML projetada para descrever Web services e como acessá-los.

18.6 A aplicabilidade da engenharia de software convencional

Não há nenhuma garantia de que um aplicativo de desktop ou uma WebApp possa ser facilmente adaptado para implementação como um aplicativo móvel. Contudo, muitas das práticas da engenharia de software ágil (Capítulo 5), utilizadas para criar aplicativos de computador desktop, podem ser usadas para criar aplicativos móveis independentes ou software cliente móvel, e muitas das práticas utilizadas para criar WebApps de qualidade se aplicam à criação de Web services usados pelos aplicativos móveis.

Durante a formulação, é montado um conjunto de metas e histórias de usuário, seguindo práticas utilizadas em muitos modelos de processos ágeis. Isso definirá a experiência de usuário exigida e determinará as necessidades dos envolvidos a serem satisfeitas e as variáveis contextuais a serem levadas em conta pelo aplicativo móvel. Provavelmente, o papel do contexto e a sensibilidade à localização não será considerada no estabelecimento de metas para um aplicativo de desktop ou Web.

Durante a atividade de planejamento, as dificuldades do desenvolvimento para mais de um dispositivo ou plataforma devem ser consideradas no orçamento e no cronograma do projeto para que os recursos necessários para satisfazer todas as preocupações dos envolvidos sejam alocados adequadamente. As dificuldades da realização de testes de utilização significativos e de testes de campo adequados aumentam os custos de desenvolvimento de aplicativos móveis. A análise de risco deve considerar o impacto de perdas, caso ocorram incidentes de segurança ou violações de privacidade. Também pode ser desejável traçar um plano de análise de segurança dos requisitos (Capítulo 27).

No desenvolvimento de aplicativos móveis, o tempo de colocação no mercado é fundamental. Além disso, são frequentemente introduzidos novos elementos tecnológicos e mudanças nos requisitos do usuário à medida que o desenvolvimento avança. Conforme mencionado anteriormente, um modelo de processo ágil e iterativo (Capítulos 4 e 5) oferece oportunidades para os desenvolvedores fazerem ajustes nos requisitos, com base em avaliações do protótipo do produto em evolução.

Durante a engenharia do produto, a análise e o projeto do conteúdo são similares às ações adotadas na construção de uma WebApp (Capítulo 17). O conteúdo a ser incluído no aplicativo móvel precisa ser selecionado e agrupado de acordo com as limitações dos dispositivos e plataformas de destino.

O projeto do aplicativo móvel pode ser acelerado pelo uso do conjunto de padrões de projeto que está em franca expansão (Capítulo 13) e de projeto baseado em componentes (Capítulo 14) voltado para aplicativos móveis [Mes08]. Quando serviços já existentes são incorporados a um aplicativo móvel, é aplicada uma estratégia de composição usando-se projeto baseado em componentes e orientado para objetos. Reutilizar sem comprometer a qualidade dos serviços é um objetivo fundamental no desenvolvimento de aplicativos móveis [Zha05].

O projeto de interface de usuário aproveita-se decisivamente das lições aprendidas no design gráfico e estético de páginas Web (Capítulo 17) para dar suporte aos objetivos diferenciadores dos aplicativos móveis. O projeto centrado no usuário, com sua ênfase na utilização e na acessibilidade, é impor-

tante para se criar interfaces de usuário de qualidade para aplicativos móveis (Capítulo 15).

Muitas vezes, a decisão de projeto arquitetural mais importante é se vai ser construído um cliente “magro” ou “gordo”. A arquitetura modelo-visão-controlador (MVC, model-view-controller), estudada no Capítulo 17, é comumente usada em aplicativos móveis. Como a arquitetura do aplicativo móvel pode ter uma forte influência sobre a navegação, as decisões tomadas durante as etapas de projeto influenciarão o trabalho conduzido durante o projeto de navegação. O projeto arquitetural deve levar em conta os recursos do dispositivo (armazenamento, velocidade do processador e conectividade de rede). O projeto deve incluir provisões para serviços e dispositivos móveis que podem ser descobertos.

Testes de utilização e distribuição ocorrem durante cada ciclo do desenvolvimento do protótipo. Revisões de código centradas em problemas de segurança devem constar como parte das atividades de implementação. Essas revisões devem ter como base os objetivos de segurança apropriados e as ameaças identificadas nas atividades de projeto do sistema. O teste de segurança é uma parte rotineira do teste do sistema (Capítulo 22).

18.7 Resumo

A qualidade de um aplicativo móvel – definida em termos de funcionalidade, confiabilidade, usabilidade, eficiência, segurança, facilidade de manutenção, escalabilidade e portabilidade – é introduzida durante o projeto. Um bom aplicativo móvel deve ter como base as seguintes metas de projeto: simplicidade, ubiquidade, personalização, flexibilidade e localização.

O projeto da interface descreve a estrutura e a organização da interface do usuário e abrange uma representação do layout da tela, uma definição dos modos de interação e uma descrição dos mecanismos de navegação. Além disso, a interface de um bom aplicativo móvel promoverá a assinatura da marca e se concentrará em sua plataforma (ou plataformas) de dispositivo pretendida. Para eliminar funcionalidades desnecessárias do aplicativo, a fim de gerenciar seus requisitos de recursos, utiliza-se um conjunto de histórias de usuário básicas. Os dispositivos sensíveis ao contexto utilizam serviços que podem ser descobertos para ajudar a personalizar a experiência do usuário.

O projeto do conteúdo é criticamente importante e leva em conta a tela e outras limitações dos dispositivos móveis. O projeto estético, também denominado design gráfico, descreve “o aspecto” do aplicativo móvel e inclui combinação de cores, layout gráfico, o uso de elementos gráficos e decisões estéticas relacionadas. O projeto estético também deve levar em consideração as limitações do dispositivo.

O projeto de arquitetura identifica a estrutura de hipermídia geral do aplicativo móvel e engloba tanto a arquitetura de conteúdo quanto a arquitetura do aplicativo. É fundamental determinar quanto da funcionalidade do aplicativo móvel vai residir no dispositivo móvel e quanto será fornecido por Web services ou serviços da nuvem.

O projeto de navegação representa um fluxo de navegação entre objetos de conteúdo e para todas as funções do aplicativo móvel. A sintaxe de

navegação é definida pelos widgets disponíveis no dispositivo (ou dispositivos) móvel pretendido, e a semântica frequentemente é determinada pela plataforma móvel. O agrupamento do conteúdo deve levar em conta interrupções intermitentes do serviço e as demandas do usuário por um desempenho rápido.

O projeto de componentes desenvolve a lógica de processamento detalhada para implementar os componentes utilizados para construir uma função de aplicativo móvel completa. As técnicas de projeto descritas no Capítulo 14 podem ser aplicadas à criação de componentes para aplicativos móveis.

Problemas e pontos a ponderar

18.1 Explique por que optar por desenvolver um aplicativo móvel para vários dispositivos pode ser uma decisão de projeto dispendiosa. Há uma maneira de reduzir os riscos de dar suporte para a plataforma errada?

18.2 Neste capítulo, listamos muitos atributos de qualidade para aplicativos móveis. Escolha os três que você acredita serem os mais importantes e defenda um argumento que explique por que cada um deve ser enfatizado no trabalho de projeto de aplicativos móveis.

18.3 Acrescente pelo menos cinco outras questões à checklist de qualidade para o projeto de aplicativos móveis apresentada na Seção 18.2.

18.4 Você é projetista de aplicativos móveis da *Project Planning Corporation*, uma empresa que constrói software de produtividade. Você quer implementar o equivalente a um fichário digital que permita aos usuários de tablet organizar e classificar documentos eletrônicos de vários tipos sob guias definidas por eles. Por exemplo, um projeto de remodelagem da cozinha poderia exigir um catálogo em .pdf, um desenho de layout em .jpg ou .dxf, uma proposta no MS Word e uma planilha do Excel armazenada sob uma guia Carpintaria. Uma vez definido, o fichário e seu conteúdo em guias podem ser armazenados no tablet ou em algum armazenamento na nuvem. O aplicativo precisa fornecer cinco funções importantes: definição do fichário e das guias, aquisição de documento digital de um local na Web ou do dispositivo, funções de gerenciamento de fichário, funções de exibição de página e uma função de anotações para permitir a adição de uma anotação em adesivo em qualquer página. Desenvolva um projeto de interface para o aplicativo de fichário e implemente-o como um protótipo em papel.

18.5 Qual foi o aplicativo móvel mais esteticamente atraente que você usou até hoje e por quê?

18.6 Crie histórias de usuário para o aplicativo de fichário descrito no Problema 18.4.

18.7 O que poderia ser considerado para se transformar o aplicativo de fichário em um aplicativo móvel sensível ao contexto?

18.8 Reconsiderando o aplicativo de fichário da *Project Planning* descrito no Problema 18.4, selecione uma plataforma de desenvolvimento para o primeiro protótipo funcional. Discuta a razão para ter feito tal escolha.

18.9 Use UML para desenvolver representações de projeto para os objetos da interface que seriam encontrados no projeto do aplicativo de fichário descrito no Problema 18.4.

18.10 Pesquise mais sobre a arquitetura MVC e decida se seria ou não a arquitetura de aplicativo móvel apropriada para o fichário discutido no Problema 18.4.

18.11 Descreva três recursos sensíveis ao contexto que poderiam ser adicionados a um aplicativo móvel *CasaSegura*.

18.12 Faça uma pesquisa na Internet para identificar um produto de middleware projetado para suportar aplicativos móveis. Descreva os recursos do middleware e a plataforma (ou plataformas) suportada.

Leituras e fontes de informação complementares

Kumar e Xie (*Handbook of Mobile Systems Applications and Services*, Auerbach Publications, 2012) editaram um livro que aborda serviços móveis e a função das arquiteturas voltadas para serviço na computação móvel. Livros sobre computação ubíqua de Adelstein (*Fundamentals of Mobile and Pervasive Computing*, McGraw-Hill, 2004), Hansmann (*Pervasive Computing: The Mobile World*, 2ª ed., Springer, 2003, reimpresso em 2013), definem os princípios do contexto na computação móvel. Livros de Kuniavsky (*Smart Things: Ubiquitous Computing User Experience Design*, Morgan Kaufman, 2010) e Polstad (*Ubiquitous Computing: Smart Devices, Environments and Interactions*, Wiley, 2009) descrevem a computação sensível ao contexto em termos das interações entre dispositivos inteligentes, ambientes inteligentes e interações inteligentes. Neil (*Mobile Design Pattern Gallery*, O'Reilly, 2012) documenta padrões de projeto móvel, ilustrados por meio de exemplos de seis plataformas móveis.

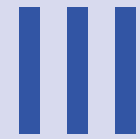
Existem muitos livros sobre projeto de interface. Uma boa referência geral é o livro editado por Schumacher (*Handbook of Global User Research*, Morgan-Kaufmann, 2009). Hooper (*Designing Mobile Interfaces*, O'Reilly, 2011) descreve melhores práticas independentes de dispositivo para composição de páginas, exibição de informações e utilização de sensores. Nielsen (*Mobile Usability*, New Riders, 2012) oferece conselhos sobre como projetar interfaces úteis que levam em conta o tamanho da tela de dispositivos móveis. Colborne (*Simple and Usable Web, Mobile, and Interaction Design*, New Riders, 2010) descreve o processo de simplificação da interação do usuário. Ginsburg (*Designing the iPhone User Experience: A User-Centered Approach to Sketching and Prototyping iPhone Apps*, Addison-Wesley, 2010) discute a importância de diferenciar seu aplicativo da concorrência, adotando uma abordagem centrada no usuário para planejar uma experiência de usuário de alta qualidade.

O *Microsoft Application Architecture Guide* (Microsoft Press, 2009) contém informações úteis sobre projeto e arquitetura de aplicativos móveis. O projeto arquitetural de tablets é discutido em um livro de Lee (*Mobile Applications: Architecture, Design, and Development*, Prentice Hall, 2004). Esposito (*Architecting Mobile Solutions for the Enterprise*, Microsoft Press, 2012) descreve o processo de desenvolvimento de um aplicativo multiplataforma, construindo um site móvel eficiente. O middleware móvel é discutido em um livro editado por Garbinato (*Middleware for Network Eccentric and Mobile Applications*, Springer, 2009).

Existem muitos livros sobre programação de aplicativos móveis que enfocam uma plataforma específica. Os livros de Firtman (*Programming the Mobile Web*, O'Reilly, 2010), Mednieke (*Programming Android*, O'Reilly, 2011) ou Lee (*Test-Driven iOS Development*, Addison-Wesley, 2012) são representativos.

Uma ampla gama de fontes de informação sobre projeto de aplicativos móveis se encontra à disposição na Internet. Uma lista atualizada das referências da Web (em inglês) pode ser encontrada no site: www.mhhe.com/pressman.

PARTE



Gestão da qualidade

Nesta parte do livro, serão vistos os princípios, as técnicas e os conceitos aplicados ao gerenciamento e ao controle da qualidade de software. Estas questões são tratadas nos capítulos que seguem:

- Quais são as características genéricas de um software de alta qualidade?
- Como revisar a qualidade e como realizar revisões eficazes?
- O que é garantia da qualidade de software?
- Quais são as estratégias aplicáveis aos testes de software?
- Quais métodos são usados para projetar casos de teste eficazes?
- Existem métodos práticos para garantir que um software está correto?
- Como gerenciar e controlar mudanças que sempre ocorrem quando um software é criado?
- Que medidas e métricas podem ser usadas para avaliar a qualidade dos modelos de requisitos e de projeto, código-fonte e casos de teste?

Respondidas essas questões, você estará mais bem preparado para a produção de software de alta qualidade.

19

Conceitos de qualidade

Conceitos-chave

ações administrativas...	426
custo da qualidade.....	422
dimensões de	
qualidade.....	415
fatores de qualidade.....	416
o dilema da qualidade ..	420
qualidade.....	413
responsabilidade civil...	425
riscos.....	424
segurança.....	425
software "bom o	
suficiente".....	421
visão quantitativa.....	420

A necessidade de maior qualidade de software surgiu realmente a partir do momento que o software ficou cada vez mais integrado a todas as atividades de nossas vidas. Na década de 1990, as principais empresas reconheciam que bilhões de dólares por ano eram desperdiçados em software que não apresentava as características e as funcionalidades prometidas. Pior ainda, tanto o governo quanto as empresas estavam cada vez mais preocupados com o fato de que uma falha grave de software poderia inutilizar importantes infraestruturas, aumentando o custo em dezenas de bilhões. Na virada do século, a *CIO Magazine* anunciou em manchete: “Chega de desperdiçar US\$ 78 bilhões por ano”, lamentando o fato de que “as empresas americanas gastavam bilhões em software que não fazia o que supostamente deveria fazer” [Lev01]. A *InformationWeek* [Ric01] manifestou a mesma preocupação:

Apesar das boas intenções, código maldito continua a ser o “fantasma” do mercado de software, sendo responsável por até 45% do tempo de inatividade dos sistemas computacionais e custando às empresas americanas cerca de US\$ 100 bilhões no último ano, em termos de manutenção e redução da produtividade, afirma o Standish Group, empresa de pesquisa de mercado. Isso não inclui o custo da perda de clientes insatisfeitos. Como as empresas de TI escrevem aplicações que dependem de pacotes de software de infraestrutura, código de má qualidade também pode causar estragos em aplicações personalizadas...

Qual o prejuízo causado por software de má qualidade? As definições variam, mas especialistas dizem que bastam três ou quatro defeitos a cada 1.000 linhas de código para que um programa execute de forma inadequada. Acrescente

PANORAMA

mesmo assim, pode ser algo difícil de definir. Porém, para software de computador, qualidade é algo que tem de ser definido, e é isso o que é feito neste capítulo.

Quem realiza? Todos os participantes – engenheiros de software, gerentes, envolvidos – da produção de software são responsáveis pela qualidade.

Por que é importante? Ou você faz certo da primeira vez ou faz tudo de novo. Se uma equipe de software buscar a qualidade em todas as atividades de engenharia de software, a quantidade de retrabalho será reduzida. Isso resulta em custos menores e, mais importante, menor tempo para disponibilização do produto no mercado.

O que é? A resposta não é tão simples quanto se imagina. Saiba-se o que é qualidade ao vê-la e,

Quais são as etapas envolvidas? Para obter software de alta qualidade, devem ocorrer quatro atividades: processo e prática comprovados de engenharia de software, gerenciamento consistente de projetos, controle global de qualidade e a presença de uma infraestrutura para garantir a qualidade.

Qual é o artefato? Software que atenda às necessidades do cliente, tenha um desempenho preciso e confiável e gere valor para todos que o utilizam.

Como garantir que o trabalho foi realizado corretamente? Acompanhando a qualidade com a verificação dos resultados de todas as atividades de controle de qualidade e medindo a qualidade com a verificação de erros antes da entrega e de defeitos que acabaram escapando e indo para a produção.

a isso que a maioria dos programadores insere cerca de um erro a cada 10 linhas de código escrito, multiplicados por milhões de linhas de código em vários produtos comerciais. Assim, deduz-se que o custo dos fornecedores de software será de pelo menos a metade dos seus orçamentos para a realização dos testes e correção dos erros. Percebe o tamanho do problema?

Em 2005, a *ComputerWorld* [Hil05] lamentou que “software de má qualidade está em praticamente todas as organizações que usam computadores, provocando horas de trabalho perdidas durante o tempo em que a máquina fica parada, dados perdidos ou corrompidos, oportunidades de vendas perdidas, custos de suporte e manutenção de TI elevados e baixa satisfação do cliente”. Um ano mais tarde, a *InfoWorld* [Fos06] escreveu sobre “o estado de penúria da qualidade de software”, relatando que o problema da qualidade não havia melhorado. À medida que a ênfase na qualidade do software aumentou, um levantamento com 100.000 profissionais administrativos [Rog12] indicou que os engenheiros de qualidade de software eram “os trabalhadores mais felizes na América”!

Hoje, a qualidade de software continua a ser um problema, mas quem é o culpado? Os clientes culpam os desenvolvedores, argumentando que práticas descuidadas levam a um software de baixa qualidade. Os desenvolvedores de software culpam os clientes (e outros envolvidos), argumentando que datas de entrega absurdas e um fluxo contínuo de mudanças os obrigam a entregar o software antes de ele estar completamente validado. Quem está com a razão? *Ambos* – e esse é o problema. Neste capítulo, a qualidade de software é vista como um conceito; também se examina por que vale a pena considerá-la seriamente toda vez que as práticas de engenharia de software forem aplicadas.

19.1 O que é qualidade?

Em seu livro místico *Zen e a Arte da Manutenção de Motocicletas*, Robert Pirsig [Pir74] comentou sobre aquilo que denominamos *qualidade*:

Qualidade... sabemos o que ela é, embora não saibamos o que ela é. Mas essa afirmação é contraditória. Mas algumas coisas são melhores do que outras; ou seja, elas têm mais qualidade. Mas quando tentamos dizer o que é qualidade, separando-a das coisas que a têm, tudo desaparece como num passe de mágica! Não há nada a dizer a respeito. Mas se não conseguimos dizer o que é qualidade, como saber o que ela é ou como saber até se ela existe mesmo? Se ninguém sabe o que ela é, então, para fins práticos, ela não existe. Porém, para fins práticos, ela realmente existe. Em que mais se baseia a qualidade? Por que outro motivo as pessoas pagariam fortunas por algumas coisas e jogariam outras na lata de lixo? Obviamente, algumas coisas são melhores do que outras... mas o que quer dizer melhor?... E por aí vai (andando em círculos), girando rodas mentais e em nenhum lugar encontrando um ponto de tração. Mas o que é mesmo qualidade? O que é?

De fato – o que é isso?

Em um nível mais pragmático, David Garvin [Gar84], da Harvard Business School, sugere que “qualidade é um conceito complexo e multifacetado” que pode ser descrito segundo cinco pontos de vista diferentes. A *visão transcendental* sustenta (assim como Pirsig) que qualidade é algo que se reconhece imediatamente, mas não se consegue definir explicitamente. A *visão do usuário*

Quais são as diferentes maneiras pelas quais a qualidade pode ser visualizada?

"As pessoas esquecem quão rápido um trabalho foi realizado – mas elas sempre lembram quão bem ele foi realizado."

Howard Newton

enxerga a qualidade em termos das metas específicas de um usuário. Se um produto atende a essas metas, ele apresenta qualidade. A *visão do fabricante* define qualidade em termos da especificação original do produto. Se o produto atende às especificações, ele apresenta qualidade. A *visão do produto* sugere que a qualidade pode ser ligada às características inerentes (por exemplo, funções e recursos) de um produto. Finalmente, a *visão baseada em valor* mede a qualidade tomando como base o quanto um cliente estaria disposto a pagar por um produto. Na realidade, qualidade engloba todas essas visões e outras mais.

Qualidade de projeto refere-se às características que os projetistas especificam para um produto. A qualidade dos materiais, as tolerâncias e as especificações de desempenho, todos são fatores que contribuem para a qualidade de um projeto. Quanto mais materiais de alta qualidade forem usados, tolerâncias mais rígidas e níveis de desempenho maiores forem especificados, mais aumentará a qualidade de projeto de um produto se ele for fabricado de acordo com essas especificações.

No desenvolvimento de software, a qualidade de um projeto engloba o grau de atendimento às funções e características especificadas no modelo de requisitos. A *qualidade de conformidade* focaliza o grau em que a implementação segue o projeto e que o sistema resultante atende às suas necessidades e às metas de desempenho.

Mas qualidade de projeto e qualidade de conformidade são as únicas questões que os engenheiros de software devem considerar? Robert Glass [Gla98] sustenta que o indicado é uma relação mais "intuitiva":

satisfação do usuário = produto compatível + boa qualidade + entrega dentro do orçamento e do prazo previsto

Ou seja, Glass argumenta que a qualidade é importante, mas se o usuário não estiver satisfeito, nada mais importa. DeMarco [DeM98] reforça esse ponto de vista ao afirmar: "A qualidade de um produto é função do quanto ele transforma o mundo para melhor". Essa visão de qualidade sustenta que se um produto de software fornece benefício substancial a seus usuários, é possível que eles estejam dispostos a tolerar problemas ocasionais de confiabilidade ou desempenho.

19.2 Qualidade de software

Até mesmo os desenvolvedores de software mais experientes concordarão que software de alta qualidade é um objetivo importante. Mas como definir a *qualidade de software*? No sentido mais geral, a qualidade de software pode ser definida como: *uma gestão de qualidade efetiva aplicada de modo a criar um produto útil que forneça valor mensurável para aqueles que o produzem e para aqueles que o utilizam.*¹

Não há dúvida nenhuma de que essa definição pode ser modificada ou estendida e debatida interminavelmente. Para os propósitos deste livro, a definição serve para enfatizar três pontos importantes:

Como posso definir qualidade de software da melhor maneira possível?

¹ Essa definição foi adaptada de [Bes04] e substitui uma visão mais voltada para a manufatura apresentada em edições anteriores deste livro.

1. Uma *gestão de qualidade efetiva* estabelece a infraestrutura que dá suporte a qualquer tentativa de construir um produto de software de alta qualidade. Os aspectos administrativos do processo criam mecanismos de controle e equilíbrio de poderes que ajudam a evitar o caos no projeto – um fator-chave para uma qualidade inadequada. As práticas de engenharia de software permitem ao desenvolvedor analisar o problema e elaborar uma solução consistente – aspectos críticos na construção de software de alta qualidade. Finalmente, as atividades de apoio, como o gerenciamento de mudanças e as revisões técnicas, têm muito a ver com a qualidade, assim como qualquer outra parte da prática de engenharia de software.
2. Um *produto útil* fornece o conteúdo, as funções e os recursos que o usuário deseja; além disso, e não menos importante, deve fornecer confiabilidade e isenção de erros. Um produto útil sempre satisfaz às exigências definidas explicitamente pelos envolvidos. Além disso, ele satisfaz a um conjunto de requisitos implícitos (por exemplo, facilidade de uso) que se espera de todo software de alta qualidade.
3. Ao *agregar valor tanto para o fabricante quanto para o usuário* de um produto de software, um software de alta qualidade gera benefícios para a empresa de software e para a comunidade de usuários. A empresa fabricante do software ganha valor agregado pelo fato de um software de alta qualidade exigir menos manutenção, menos correções de erros e menos suporte ao cliente. Isso permite que os engenheiros de software despendam mais tempo criando aplicações novas e menos tempo em manutenções. A comunidade de usuários ganha um valor agregado, pois a aplicação fornece a capacidade de agilizar algum processo de negócio. O resultado final é: (1) maior receita gerada pelo produto de software, (2) maior rentabilidade quando uma aplicação suporta um processo de negócio e/ou (3) maior disponibilidade de informações cruciais para o negócio.

"Seja um parâmetro de qualidade. Algumas pessoas não estão acostumadas com um ambiente onde se espera excelência."

Steve Jobs

19.2.1 Dimensões de qualidade de Garvin

David Garvin [Gar87] sugere que a qualidade deve ser considerada adotando-se um ponto de vista multidimensional que começa com uma avaliação da conformidade e termina com uma visão transcendental (estética). Embora as oito dimensões de qualidade de Garvin não tenham sido desenvolvidas especificamente para software, elas podem ser aplicadas quando se considera qualidade de software:

Qualidade do desempenho. O software fornece todo o conteúdo, funções e recursos especificados como parte do modelo de requisitos, de forma a gerar valor ao usuário?

Qualidade dos recursos. O software fornece recursos que surpreendem e encantam usuários que os utilizam pela primeira vez?

Confiabilidade. O software fornece todos os recursos e capacidades sem falhas? Está disponível quando necessário? Fornece funcionalidade sem a ocorrência de erros?

Você pode usar um diagrama de radar para fornecer uma representação visual de cada uma das dimensões de qualidade de Garvin, conforme são empregadas a um aplicativo.

Conformidade. O software está de acordo com os padrões de software locais e externos relacionados com a aplicação? Segue as convenções de projeto e codificação de fato? Por exemplo, a interface do usuário está de acordo com as regras de projeto aceitas para seleção de menus ou entrada de dados?

Durabilidade. O software pode ser mantido (modificado) ou corrigido (depurado) sem a geração involuntária de efeitos colaterais indesejados? As mudanças farão com que a taxa de erros ou a confiabilidade degradem com o passar do tempo?

Facilidade de manutenção. O software pode ser mantido (modificado) ou corrigido (depurado) em um período de tempo aceitável e curto? O pessoal de suporte pode obter todas as informações necessárias para realizar alterações ou corrigir defeitos? Douglas Adams [Ada93] comenta ironicamente: “A diferença entre algo que pode dar errado e algo que possivelmente não pode dar errado é que, quando algo que possivelmente não pode dar errado dá errado, normalmente acaba sendo impossível acessá-lo ou repará-lo”.

Estética. Não há dúvida nenhuma de que cada um de nós tem uma visão diferente e muito subjetiva do que é estética. Mesmo assim, a maioria de nós concordaria que uma entidade estética tem certa elegância, um fluir único e uma “presença” que são difíceis de quantificar, mas que, não obstante, são evidentes. Um software estético possui essas características.

Percepção. Em algumas situações, temos alguns preconceitos que influenciarão nossa percepção de qualidade. Por exemplo, se for apresentado um produto de software construído por um fornecedor que, no passado, havia produzido software de má qualidade, ficaremos com a nossa percepção de qualidade do produto de software influenciada negativamente. Similarmente, se um fornecedor tem uma excelente reputação, talvez percebamos qualidade, mesmo quando ela realmente não existe.

As dimensões de qualidade de Garvin nos dão uma visão “indulgente” da qualidade de software. Muitas (mas não todas) dessas dimensões podem ser consideradas apenas subjetivamente. Por tal razão, também precisamos de um conjunto de fatores de qualidade que podem ser classificados em duas grandes categorias: (1) fatores que podem ser medidos diretamente (por exemplo, defeitos revelados durante testes) e (2) fatores que podem ser medidos apenas indiretamente (por exemplo, usabilidade ou facilidade de manutenção). Em cada um dos casos deve ocorrer medição. Devemos comparar o software com algum dado e chegarmos a uma indicação da qualidade.

19.2.2 Fatores de qualidade de McCall

McCall, Richards e Walters [McC77] criaram uma proposta de categorização dos fatores que afetam a qualidade de software. Esses fatores de qualidade de software, apresentados na Figura 19.1, se concentram em três importantes aspectos de um produto de software: as características operacionais, a capacidade de suportar mudanças e a adaptabilidade a novos ambientes.

Referindo-se aos fatores citados na Figura 19.1, McCall e seus colegas fazem as seguintes descrições:



FIGURA 19.1 Fatores de qualidade de software de McCall.

Correção. O quanto um programa satisfaz a sua especificação e atende aos objetivos da missão do cliente.

Confiabilidade. O quanto se pode esperar que um programa realize a função pretendida com a precisão exigida. [Observe que foram propostas outras definições mais completas de confiabilidade (veja o Capítulo 21).]

Eficiência. A quantidade de recursos computacionais e código exigidos por um programa para desempenhar sua função.

Integridade. O quanto o acesso ao software ou dados por pessoas não autorizadas pode ser controlado.

Usabilidade. Esforço necessário para aprender, operar, preparar a entrada de dados e interpretar a saída de um programa.

Facilidade de manutenção. Esforço necessário para localizar e corrigir um erro em um programa. [Trata-se de uma definição muito limitada.]

Flexibilidade. Esforço necessário para modificar um programa em operação.

Testabilidade. Esforço necessário para testar um programa de modo a garantir que ele desempenhe a função pretendida.

Portabilidade. Esforço necessário para transferir o programa de um ambiente de hardware e/ou software para outro.

Reusabilidade. O quanto um programa ou partes de um programa podem ser reutilizados em outras aplicações – relacionado com o empacotamento e o escopo das funções que o programa executa.

Interoperabilidade. Esforço necessário para integrar um sistema a outro.

É difícil – e, em alguns casos, impossível – desenvolver medidas² diretas desses fatores de qualidade. Na realidade, muitas das métricas definidas por McCall e colegas podem ser medidas apenas indiretamente. Entretanto, avaliar a qualidade de uma aplicação usando esses fatores possibilitará uma sólida indicação da qualidade de um software.

“O amargo da má qualidade permanece muito tempo depois da doçura do cumprimento do prazo ter sido esquecida.”

Karl Weigers
(citação não atribuída)

² Uma *medida direta* implica existir um único valor contável que dê uma indicação direta do atributo que está sendo examinado. Por exemplo, o “tamanho” de um programa pode ser medido diretamente contando-se o número de linhas de código.

19.2.3 Fatores de qualidade ISO 9126

O padrão ISO 9126 foi desenvolvido como uma tentativa de identificar os atributos fundamentais de qualidade para software de computador. O padrão identifica seis atributos fundamentais de qualidade:

Embora seja tentador criar medidas quantitativas para os fatores de qualidade aqui citados, também podemos criar uma lista de verificação simples dos atributos que dão uma sólida indicação de que o fator está presente.

Funcionalidade. O grau com que o software satisfaz as necessidades declaradas, conforme indicado pelos seguintes subatributos: adequabilidade, exatidão, interoperabilidade, conformidade e segurança.

Confiabilidade. A quantidade de tempo por que o software fica disponível para uso, conforme indicado pelos seguintes subatributos: maturidade, tolerância a falhas, facilidade de recuperação.

Usabilidade. O grau de facilidade de utilização do software, conforme indicado pelos seguintes subatributos: facilidade de compreensão, facilidade de aprendizagem, operabilidade.

Eficiência. O grau de otimização do uso, pelo software, dos recursos do sistema, conforme indicado pelos seguintes subatributos: comportamento em relação ao tempo, comportamento em relação aos recursos.

Facilidade de manutenção. A facilidade com a qual uma correção pode ser realizada no software, conforme indicado pelos seguintes subatributos: facilidade de análise, facilidade de realização de mudanças, estabilidade, testabilidade.

Portabilidade. A facilidade com a qual um software pode ser transposto de um ambiente para outro, conforme indicado pelos seguintes subatributos: adaptabilidade, facilidade de instalação, conformidade, facilidade de substituição.

Assim como outros fatores de qualidade de software discutidos nas subseções anteriores, os fatores da ISO 9126 não levam, necessariamente, à medição direta. Entretanto, eles fornecem uma base razoável para medidas indiretas e uma excelente lista de verificação para avaliar a qualidade de um sistema.

19.2.4 Fatores de qualidade desejados

"Qualquer atividade se torna criativa quando o executor se preocupa em fazê-la de maneira correta, ou melhor."

John Updike

As dimensões e os fatores de qualidade apresentados nas Seções 19.2.1 e 19.2.2 concentram-se no software como um todo e podem ser usados como uma indicação genérica da qualidade de uma aplicação. A equipe de software pode desenvolver um conjunto de características de qualidade e questões associadas que investigaria o grau em que cada fator foi satisfeito.³ Por exemplo, McCall identifica a *usabilidade* como um importante fator de qualidade. Se lhe fosse solicitado para revisar uma interface com o usuário e avaliar sua usabilidade, como você procederia? Você poderia começar com os subatributos sugeridos por McCall – facilidade de compreensão, facilidade de aprendizagem, operabilidade –, mas qual seu significado em um sentido prático?

³ Essas características e questões podem ser abordadas como parte de uma revisão de software (Capítulo 20).

Para fazer sua avaliação, você precisaria lidar com atributos específicos, mensuráveis (ou pelo menos, reconhecíveis) da interface. Por exemplo [Bro03]:

Intuição. O grau em que a interface segue padrões de uso esperados de modo que até mesmo um novato possa usá-la sem treinamento significativo.

- O layout da interface favorece a fácil compreensão?
- As operações da interface são fáceis de ser localizadas e iniciadas?
- A interface utiliza uma metáfora reconhecível?
- É especificada entrada para economizar toques de teclado ou cliques de mouse?
- A interface segue as três regras de ouro? (Ver o Capítulo 15.)
- A estética ajuda no entendimento e uso?

Eficiência. A facilidade com a qual as operações e informações podem ser localizadas ou iniciadas.

- O layout e o estilo da interface permitem a um usuário localizar eficientemente as operações e informações?
- Uma sequência de operações (ou entrada de dados) pode ser realizada reduzindo-se o número de movimentos?
- Os dados de saída ou o conteúdo são apresentados de modo a ser imediatamente compreendidos?
- As operações hierárquicas foram organizadas de modo a minimizar o nível em que um usuário deve navegar para realizar algo?

Robustez. O grau com o qual o software trata dados de entrada incorretos ou interação inapropriada com o usuário.

- O software reconhecerá erros caso sejam introduzidos valores de dados dentro ou fora dos limites prescritos? Mais importante ainda, o software continuará a operar sem falha ou degradação?
- A interface reconhece erros cognitivos ou manipuladores comuns e orienta explicitamente o usuário para retomar o caminho certo?
- A interface oferece diagnósticos e orientação úteis quando é descoberta uma condição de erro (associada à funcionalidade do software)?

Riqueza. O grau em que a interface oferece um conjunto rico de recursos importantes.

- A interface pode ser personalizada de acordo com as necessidades específicas de um usuário?
- A interface dispõe de recursos de macros que permitem ao usuário identificar uma sequência de operações comuns por meio de uma única ação ou comando?

Enquanto o projeto de interface é desenvolvido, a equipe de software revisaria o protótipo de projeto e faria as perguntas citadas. Se a resposta a essas perguntas for “sim”, é provável que a interface com o usuário apresente alta

qualidade. Um conjunto de perguntas similares seria desenvolvido para cada fator de qualidade a ser avaliado.

19.2.5 A transição para uma visão quantitativa

Nas subseções anteriores, apresentamos um conjunto de fatores qualitativos para a “medição” da qualidade de um software. A comunidade da engenharia de software se esforça ao máximo para desenvolver medidas precisas para a qualidade de software e algumas vezes é frustrada pela natureza subjetiva da atividade. Cavano e McCall [Cav78] discutem essa situação:

A determinação da qualidade é um fator-chave nos eventos do dia a dia – concursos de degustação de vinhos, eventos esportivos (por exemplo, ginástica), concursos de talentos etc. Nessas situações, a qualidade é julgada da forma mais fundamental e direta: comparação entre dois objetos sob condições idênticas e com conceitos predeterminados. O vinho pode ser julgado segundo sua limpidez, cor, buquê, sabor etc. Entretanto, esse tipo de julgamento é muito subjetivo; para ter algum valor, ele precisa ser feito por um especialista.

A subjetividade e a especialização também se aplicam na determinação da qualidade do software. Para ajudar a solucionar esse problema, é necessária uma definição mais precisa da qualidade de software, bem como uma maneira de obter medidas quantitativas da qualidade de software para uma análise objetiva... Como não existe conhecimento absoluto sobre isso, não se deve ter a expectativa de medir a qualidade de software com exatidão, já que cada medida é parcialmente imperfeita. Jacob Bronkowsky descreve esse paradoxo do conhecimento da seguinte maneira: “Ano após ano, inventamos instrumentos mais precisos com os quais observamos a natureza com maior perfeição. E, ao analisarmos essas observações, ficamos desconcertados em ver que elas ainda são distorcidas e incertas”.

No Capítulo 30, apresentaremos um conjunto de métricas de software que podem ser aplicadas para a avaliação quantitativa da qualidade de software. Em todos os casos, as métricas representam medidas indiretas; isto é, jamais medimos realmente a *qualidade*, mas sim alguma manifestação dessa qualidade. O fator complicador é a relação precisa entre a variável medida e a qualidade de software.

19.3 O dilema da qualidade do software

Em uma entrevista [Ven03] publicada na Internet, Bertrand Meyer discute o que chamamos de *dilema da qualidade*:

Se produzimos um sistema de software de péssima qualidade, perdemos porque ninguém vai querer comprá-lo. Se, por outro lado, gastamos um tempo infinito, um esforço extremamente grande e grandes somas de dinheiro para construir um software absolutamente perfeito, então ele levará muito tempo para ser concluído, e o custo de produção será tão alto que iremos à falência. Ou perdemos a oportunidade de mercado ou então simplesmente esgotamos todos os nossos recursos. Assim, os profissionais desta área tentam encontrar aquele meio-termo mágico em que o produto é suficientemente bom para não ser rejeitado logo de cara, como, por exemplo, durante uma avaliação, mas também não é objeto de tamanho perfeccionismo e trabalho que levaria muito tempo ou que seria demasiadamente caro para ser finalizado.

Ao se deparar com o dilema da qualidade (e todo mundo se depara com ele uma hora ou outra), tente alcançar o equilíbrio – esforço suficiente para produzir qualidade aceitável sem “enterrar” o projeto.

Embora seja tentador criar medidas quantitativas para os fatores de qualidade aqui citados, também podemos criar uma lista de verificação simples dos atributos que dão uma sólida indicação de que o fator está presente.

É válido afirmar que os engenheiros de software devem se esforçar para produzir sistemas de alta qualidade. Muito melhor seria aplicar boas práticas na tentativa de obter essa alta qualidade. Porém, a situação discutida por Meyer é a realidade e representa um dilema até mesmo para as melhores organizações de engenharia de software.

19.3.1 Software “bom o suficiente”

Falando claramente, se concordarmos com o argumento de Meyer, é aceitável produzir software “bom o suficiente”? A resposta a essa pergunta deve ser “sim”, pois atualmente as principais empresas de software agem dessa forma. Elas criam software com erros (bugs) conhecidos e ele é entregue a vários usuários. Elas reconhecem que algumas funções e características disponibilizadas na Versão 1.0 podem não ser da melhor qualidade e planejam melhoramentos para a Versão 2.0. Elas fazem isso, mesmo sabendo que alguns clientes vão reclamar; entretanto, reconhecem que o tempo de colocação do produto no mercado pode superar uma qualidade melhor, desde que o produto fornecido seja “bom o suficiente”.

O que é exatamente “bom o suficiente”? Software bom o suficiente fornece funções e características de alta qualidade que os usuários desejam, mas, ao mesmo tempo, fornece outras funções e características mais obscuras ou especializadas e ainda contendo erros conhecidos. O fornecedor de software espera que a grande maioria dos usuários ignore os erros pelo fato de estarem muito satisfeitos com as outras funcionalidades oferecidas pela aplicação.

Essa ideia pode ter um significado especial para muitos leitores. Se você for um deles, pedimos para considerar alguns dos argumentos contra software “bom o suficiente”.

É verdade que software “bom o suficiente” pode funcionar em alguns domínios de aplicação e para algumas grandes empresas de software. Afinal de contas, se uma empresa tiver um grande orçamento para o marketing e conseguir convencer um número suficiente de pessoas a comprar a versão 1.0, ela será bem-sucedida. Conforme citado anteriormente, pode-se argumentar que a qualidade será melhorada nas próximas versões. Ao entregar a versão 1.0 boa o suficiente, a empresa já monopolizou o mercado.

Caso trabalhe em uma pequena empresa, não confie nessa filosofia. Ao entregar um produto bom o suficiente (com erros), você corre o risco de arruinar permanentemente a reputação da empresa. Talvez jamais tenha a chance de entregar a versão 2.0, pois, devido à má propaganda resultante dessa filosofia, as vendas podem despencar e, conseqüentemente, a empresa falir.

Caso trabalhe em certos domínios de aplicação (por exemplo, software embarcado para aplicações em tempo real) ou crie software de aplicação integrado com o hardware (por exemplo, software para a indústria automotiva, software para telecomunicações), entregar software com erros conhecidos pode ser negligente e expõe sua empresa a litígios dispendiosos. Em alguns casos, pode constituir crime. Ninguém quer software bom o suficiente e inútil!

Portanto, proceda com cautela caso acredite que “bom o suficiente” seja um atalho capaz de resolver seus problemas de qualidade de software. Pode

ser que funcione, mas apenas para poucos casos e em um conjunto limitado de domínios de aplicação.⁴

19.3.2 Custo da qualidade

A discussão prossegue mais ou menos assim: *sabemos que a qualidade é importante, mas ela nos custa tempo e dinheiro – tempo e dinheiro demais para obter o nível de qualidade de software que realmente desejamos*. Dessa forma, o argumento parece sensato (veja os comentários de Meyer no início desta seção). Não há dúvida nenhuma de que a qualidade tem um preço, mas a falta de qualidade também tem um preço – não apenas para os usuários, que terão de conviver com um software com erros, mas também para a organização de software que o criou e, além de tudo, terá de fazer a manutenção. A verdadeira questão é a seguinte: *com qual custo devemos nos preocupar?* Para responder a essa pergunta, devemos entender tanto o custo para obter alta qualidade quanto o custo de software de baixa qualidade.

O *custo da qualidade* inclui todos os custos necessários para a busca de qualidade ou para a execução de atividades relacionadas à qualidade, assim como os custos causados pela falta de qualidade. Para compreender esses custos, uma organização deve reunir métricas para prover uma base para o custo corrente da qualidade, identificar oportunidades para reduzir esses custos e fornecer uma base normalizada de comparação. O custo da qualidade pode ser dividido em custos associados à prevenção, avaliação e falhas.

Os *custos de prevenção* incluem: (1) o custo de atividades de gerenciamento necessárias para planejar e coordenar todas as atividades de controle e garantia da qualidade, (2) o custo de atividades técnicas adicionais para desenvolver modelos completos de requisitos e de projeto, (3) os custos de planejamento de testes e (4) o custo de todo o treinamento associado a essas atividades.

Os *custos de avaliação* incluem atividades para a compreensão aprofundada da condição do produto “pela primeira vez através de” cada processo. Entre os exemplos de custos de avaliação, temos: (1) o custo da realização de revisões técnicas (Capítulo 20), (2) o custo dos artefatos de engenharia de software, (3) o custo da coleta de dados e avaliação de métricas (Capítulo 30) e (3) o custo dos testes e depuração (Capítulos 22 a 26).

Os *custos de falhas* são aqueles que desapareceriam caso não surgisse nenhum erro antes ou depois da entrega de um produto aos clientes. Esses custos podem ser subdivididos em custos de falhas internas e custos de falhas externas. Os *custos de falhas internas* ocorrem quando se detecta um erro em um produto antes de ele ser entregue. Eles abrangem: (1) o custo necessário para realizar reformulações (reparos) para corrigir um erro, (2) o custo que ocorre quando reformulações geram, inadvertidamente, efeitos colaterais que devem ser reduzidos e (3) os custos associados à reunião de métricas de qualidade que permitam a uma organização avaliar os modos de falha. Os *custos de falhas externas* estão associados a defeitos encontrados após o produto ter sido entregue ao cliente. Exemplos de custos de falhas externas são a solu-

Não tenha medo de incorrer em custos significativos de prevenção. Esteja seguro de que esse investimento possibilitará um excelente retorno.

“Leva menos tempo para fazer algo certo do que explicar por que o fez errado.”

H. W. Longfellow

⁴ Uma interessante discussão dos prós e contras do software “bom o suficiente” pode ser encontrada em [Bre02].

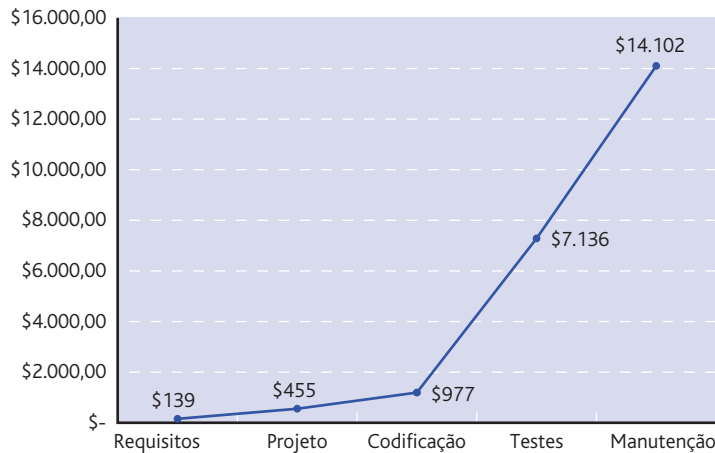


FIGURA 19.2 Custo relativo para correção de erros e defeitos.

Fonte: Adaptado de [Boe01b].

ção de reclamações, a devolução e substituição de produtos, o suporte por telefone/e-mail e custos de mão de obra associados à garantia do produto. Uma má reputação e a consequente perda de negócios é outro custo de falhas externas difícil de ser quantificado, mas bastante real. Coisas negativas acontecem quando se produz software de baixa qualidade.

Em uma censura aos desenvolvedores de software que se recusam a considerar os custos de falhas externas, Cem Kaner [Kan95] afirma:

Muitos dos custos de falhas externas, assim como sua reputação no mercado, são difíceis de ser quantificados, e, conseqüentemente, muitas empresas os ignoram no cálculo das relações custo-benefício. Outros custos de falhas externas podem ser reduzidos (por exemplo, o fornecimento de suporte pós-venda mais barato e de menor qualidade ou cobrando o suporte dos clientes) sem aumentar a satisfação do cliente. Ao ignorar os custos de produtos ruins para nossos clientes, os engenheiros de qualidade estimulam tomadas de decisão relacionadas à qualidade que penalizam nossos clientes, em vez de satisfazê-los.

Como era de esperar, os custos relativos para descobrir e reparar um erro ou defeito aumentam drasticamente à medida que avançamos dos custos de prevenção para custos de detecção de falhas internas e para custos de falhas externas. A Figura 19.2, fundamentada em dados coletados por Boehm e Basili [Boe01b] e ilustrada pela Cigital Inc. [Cig07], exemplifica esse fenômeno.

O custo médio da indústria de software para corrigir um defeito durante a geração de código é de aproximadamente US\$ 977 por erro. O custo médio para corrigir o mesmo erro caso ele tenha sido descoberto durante os testes do sistema passa a ser de US\$ 7.136 por erro. A Cigital Inc. [Cig07] considera uma grande aplicação em que foram introduzidos 200 erros durante a codificação:

De acordo com os dados médios do setor, o custo para descobrir e corrigir defeitos durante a fase de codificação é de US\$ 977 por defeito. Portanto, o custo total para corrigir os 200 defeitos “críticos” durante essa fase ($200 \times \text{US\$ } 977$) é de aproximadamente US\$ 195.400.

Os dados médios do setor mostram que o custo para descobrir e corrigir defeitos durante a fase de testes é de US\$ 7.136 por defeito. Nesse caso, supondo que a

fase de testes do sistema tenha revelado aproximadamente 50 defeitos críticos (ou apenas 25% daqueles encontrados pela Cigital na fase de codificação), o custo para descobrir e corrigir esses defeitos ($50 \times \text{US\$ } 7.136$) teria sido de aproximadamente US\$ 356.800. Isso teria resultado em 150 erros críticos sem ser detectados e corrigidos. O custo para descobrir e corrigir esses 150 defeitos remanescentes na fase de manutenção ($150 \times \text{US\$ } 14.102$) teria sido de US\$ 2.115.300. Portanto, o custo total para descobrir e corrigir os 200 defeitos após a fase de codificação teria sido de US\$ 2.472.100 ($\text{US\$ } 2.115.300 + \text{US\$ } 356.800$).

Mesmo que sua organização de software possua custos equivalentes à metade da média do setor (a maioria não tem a mínima ideia de quanto são seus custos!), a economia de custos associados a atividades iniciais de controle com garantia da qualidade (conduzidas durante a análise de requisitos e projeto) é considerável.

CASASEGURA



Questões de qualidade

Cena: Escritório de Doug Miller no início do projeto de software *CasaSegura*.

Atores: Doug Miller (gerente da equipe de engenharia de software do *CasaSegura* e outros membros da equipe de engenharia de software do produto.

Conversa:

Doug: Eu estava vendo um relatório da indústria sobre os custos para reparar defeitos de software. Eles dão o que pensar.

Jamie: Já estamos trabalhando no desenvolvimento de casos de teste para cada requisito funcional.

Doug: Isso é bom, mas notei que o custo para reparar um defeito descoberto nos testes é oito vezes maior do que se o defeito for detectado e reparado durante a codificação.

Vinod: Estamos usando programação em pares; portanto, deveremos detectar a maioria dos defeitos durante a codificação.

Doug: Acho que você não está entendendo. Qualidade é mais do que simplesmente remover erros de codificação. Precisamos examinar os objetivos de qualidade do projeto e garantir que os produtos de software em evolução os atinjam.

Jamie: Você quer dizer coisas como usabilidade, segurança e confiabilidade?

Doug: Sim, isso mesmo. Precisamos realizar verificações no processo de software para monitorar nosso progresso no sentido de atingir nossas metas de qualidade.

Vinod: Não podemos terminar o protótipo primeiro e então verificar a qualidade?

Doug: Temo que não. Devemos estabelecer uma cultura de qualidade no início do projeto.

Vinod: O que você quer que façamos, Doug?

Doug: Acho que precisamos encontrar uma técnica que nos permita monitorar a qualidade dos produtos do *CasaSegura*. Vamos pensar a respeito e rever isso novamente amanhã.

19.3.3 Riscos

No Capítulo 1 dissemos que “as pessoas apostam seus empregos, seu conforto, sua segurança, seu entretenimento, suas decisões e suas próprias vidas em software. Tomara que estejam certas”. A implicação disso é que software de baixa qualidade aumenta os riscos tanto para o desenvolvedor quanto para o usuário. Na subseção anterior, discutimos um desses riscos (custo). Mas o lado negativo de aplicações mal projetadas e implementadas nem sempre resulta apenas em altos custos e mais tempo. Um exemplo [Gag04] extremo pode servir como ilustração.

Ao longo do mês de novembro de 2000, em um hospital no Panamá, 28 pacientes receberam doses maciças de raio gama durante tratamento para uma série de tipos de câncer. Nos meses seguintes, 5 desses pacientes morreram

por contaminação radioativa, e outros 15 desenvolveram sérias complicações. O que provocou essa tragédia?

Um pacote de software, desenvolvido por uma companhia americana, foi modificado por técnicos do hospital para calcular doses alteradas de radiação para cada paciente.

Os três médicos panamenhos com especialização em física, que ajustaram o software para aumentar a capacidade funcional, foram processados por homicídio doloso, mas sem premeditação. A empresa americana enfrentou litígios graves em dois países. Gage e McCormick comentam:

Não se trata de um conto com fundo moral para técnicos em medicina, muito embora eles possam lutar para manter-se fora da prisão, caso tenham interpretado ou usado uma tecnologia de forma errada. Não se trata também de um conto sobre como seres humanos podem ser feridos ou sofrer algo ainda mais grave por software mal projetado ou documentado, embora existam muitos exemplos que o comprovem. Trata-se de um alerta para qualquer criador de programas de computador: a qualidade de software é importante, as aplicações têm de ser infalíveis e – independentemente de estarem embutidas no motor de um carro, em um braço mecânico em uma fábrica ou em um aparelho médico em um hospital – código mal-empregado pode matar.

A baixa qualidade induz a riscos, alguns muito sérios.

19.3.4 Negligência e responsabilidade civil

A história é bastante comum. Um órgão do governo ou empresa contrata uma grande empresa de consultoria ou desenvolvedora de software para analisar os requisitos e então projetar e construir um “sistema” baseado em software para apoiar alguma atividade importante. O sistema poderia oferecer suporte a uma importante função corporativa (por exemplo, administração de aposentadorias) ou alguma função governamental (por exemplo, administração do sistema de saúde ou de segurança do território nacional).

O trabalho inicia com as melhores das intenções de ambas as partes, mas, na época em que o sistema é entregue, as coisas não andam bem. O sistema é lento, não fornece os recursos e funções desejados, é suscetível a erros e não tem a aprovação do usuário. Segue-se um litígio.

Na maioria dos casos, o cliente alega que o desenvolvedor foi negligente (na maneira de aplicar as práticas de software) e, portanto, não tem direito a receber seu pagamento. Normalmente, o desenvolvedor alega que o cliente mudou repetidamente os requisitos e subverteu a parceria de desenvolvimento de outras formas. Em todos os casos, a questão é a qualidade do sistema entregue.

19.3.5 Qualidade e segurança

À medida que o caráter crítico das aplicações e dos sistemas móveis baseados na Internet aumenta, a segurança da aplicação tem se tornado cada vez mais importante. Ou seja, software que não apresente alta qualidade é mais fácil de ser copiado (“pirateado”), e, como consequência, o software de baixa qualidade pode aumentar indiretamente o risco de segurança, assim como todos os problemas e custos associados.

Em uma entrevista à revista *ComputerWorld*, o autor e especialista em segurança Gary McGraw comenta [Wil05]:

A segurança de software se relaciona inteira e completamente à qualidade. Devemos nos preocupar com a segurança, a confiabilidade, a disponibilidade e a fidelidade – nas fases iniciais, de projeto, de arquitetura, de testes e de codificação, ao longo de todo o ciclo de vida [qualidade] de um software. Até mesmo pessoas cientes do problema da segurança têm se concentrado em coisas relacionadas ao ciclo de vida do software. O quanto antes descobriremos um problema de software, melhor. E existem dois tipos de problemas de software. O primeiro são os bugs, que são problemas de implementação. Os demais são falhas de software – problemas de arquitetura do projeto. As pessoas prestam muita atenção aos bugs, mas não o suficiente às falhas.

Para construir um sistema seguro, temos de focar na qualidade, e esse foco deve iniciar durante o projeto. Os conceitos e os métodos discutidos na Parte II deste livro nos conduzem a uma arquitetura de software que reduz “falhas”. Uma discussão mais detalhada sobre engenharia de segurança é apresentada no Capítulo 27.

19.3.6 O impacto das ações administrativas

A qualidade de software normalmente é influenciada tanto pelas decisões administrativas quanto pelas decisões técnicas. Até mesmo as melhores práticas de engenharia de software podem ser subvertidas por decisões de negócios inadequadas e ações questionáveis de gerenciamento de projeto.

Na Parte IV deste livro discutiremos o gerenciamento de projetos no contexto da gestão da qualidade. Quando cada tarefa de projeto é iniciada, um líder de projeto toma decisões que podem ter um impacto significativo sobre a qualidade do produto.

Decisões de estimativas. Raramente uma equipe de software pode se dar ao luxo de fornecer uma estimativa para um projeto *antes* das datas de entrega serem estabelecidas e um orçamento geral ser especificado. Em vez disso, a equipe realiza um “exame de sanidade” para garantir que as datas de entrega são racionais. Em muitos casos, existe uma enorme pressão de colocação do produto no mercado que obriga uma equipe a aceitar datas de entrega impraticáveis. Como resultado, são tomados atalhos, as atividades que produzem um software de maior qualidade talvez sejam deixadas de lado, e, assim, a qualidade do produto sofre as consequências. Se uma data de entrega for absurda, é importante manter-se firme. Explique por que você precisa de mais tempo ou, como alternativa, sugira um subconjunto de funcionalidades que possa ser entregue (com alta qualidade) no tempo alocado.

Decisões de cronograma. Quando um cronograma de projeto de software é estabelecido (Capítulo 34), as tarefas são sequenciadas tomando-se como base as dependências. Por exemplo, como o componente **A** depende do processamento que ocorre nos componentes **B**, **C** e **D**, o componente **A** não pode ser agendado para testes até que os componentes **B**, **C** e **D** sejam completamente testados. O cronograma de um projeto deve refletir essa situação. Mas se o tempo for muito curto e **A** tem de estar disponível para outros testes críticos, talvez se opte por testar **A** sem seus componentes subordinados (que estão li-

geiramente atrasados em relação ao cronograma), de modo a torná-lo disponível para outros testes que devem ser feitos antes da entrega. Afinal de contas, o prazo final se aproxima. Dessa forma, A pode ter defeitos que estão ocultos e que seriam descobertos muito mais tarde. A qualidade sofre as consequências.

Decisões orientadas a riscos. A administração de riscos (Capítulo 35) é um dos atributos fundamentais de um projeto de software bem-sucedido. Precisamos realmente saber o que poderia dar errado e estabelecer um plano de contingência caso isso aconteça. Um número muito grande de equipes de software prefere um otimismo cego, estabelecendo um cronograma de desenvolvimento sob a hipótese de que nada vai dar errado. Pior ainda, eles não têm um método para lidar com as coisas que dão errado. Consequentemente, quando um risco se torna realidade, reina o caos, e, à medida que o grau de loucura aumenta, o nível de qualidade invariavelmente cai.

O dilema da qualidade de software pode ser mais bem sintetizado enunciando-se a Lei de Meskimen – *Nunca há tempo para fazer a coisa certa, mas sempre há tempo para fazê-la de novo*. Nosso conselho: tomar o tempo necessário para fazer certo da primeira vez quase nunca é uma decisão errada.

19.4 Alcançando a qualidade de software

A qualidade de software não aparece simplesmente do nada. Ela é o resultado de um bom gerenciamento de projeto e uma prática consistente de engenharia de software. O gerenciamento e a prática são aplicados no contexto de quatro atividades amplas que ajudam uma equipe de software a atingir alto padrão de qualidade de software: métodos de engenharia de software, técnicas de gerenciamento de projeto, ações de controle de qualidade e garantia da qualidade de software.

19.4.1 Métodos de engenharia de software

Se nossa expectativa é construir software de alta qualidade, temos de entender o problema a ser resolvido. Temos também de ser capazes de criar um projeto que seja adequado ao problema e, ao mesmo tempo, apresente características que levem a um software com as dimensões e fatores de qualidade discutidos na Seção 19.2.

Na Parte II deste livro, apresentamos uma ampla gama de conceitos e métodos capazes de levar a um entendimento relativamente completo do problema e a um projeto abrangente que estabeleça uma base sólida para a atividade de construção. Se aplicarmos esses conceitos e adotarmos os métodos de análise e projeto apropriados, a probabilidade de criarmos software de alta qualidade aumentará substancialmente.

19.4.2 Técnicas de gerenciamento de software

O impacto de decisões de gerenciamento inadequadas sobre a qualidade de software foi discutido na Seção 19.3.6. As implicações são claras: se (1) um gerente de projeto usar estimativas para verificar que as datas de entrega são plausíveis, (2) as dependências de cronograma forem entendidas e a equipe resistir à tentação

O que é preciso fazer para afetar a qualidade de forma positiva?

de usar atalhos e (3) o planejamento de riscos for conduzido de modo que problemas não gerem caos, a qualidade do software será afetada de forma positiva.

Além disso, o plano de projeto deve incluir técnicas explícitas para gerenciamento de mudanças e qualidade. Técnicas que levam a práticas ótimas de gerenciamento de projeto são discutidas na Parte IV do livro.

19.4.3 Controle de qualidade

O que é controle de qualidade de software?

O controle de qualidade engloba um conjunto de ações de engenharia de software que ajudam a garantir que cada produto resultante atinja suas metas de qualidade. Os modelos são revistos de modo a garantir que sejam completos e consistentes. O código poderia ser inspecionado de modo a revelar e corrigir erros antes de os testes começarem. Aplica-se uma série de etapas de teste para descobrir erros na lógica de processamento, na manipulação de dados e na comunicação da interface. Uma combinação de medições e feedback permite a uma equipe de software ajustar o processo quando qualquer um desses artefatos deixe de atender às metas estabelecidas para a qualidade. As atividades de controle de qualidade são discutidas em detalhe ao longo do restante da Parte III deste livro.

19.4.4 Garantia da qualidade

Links úteis sobre a garantia da qualidade de software podem ser encontrados em www.niwotridge.com/Resources/PM-SWEResources/SoftwareQualityAssurance.htm.

A garantia da qualidade estabelece a infraestrutura que suporta métodos sólidos de engenharia de software, gerenciamento racional de projeto e ações de controle de qualidade – todos fundamentais para a construção de software de alta qualidade. Além disso, a garantia da qualidade consiste em um conjunto de funções de auditoria e de relatórios que possibilita uma avaliação da efetividade e da completude das ações de controle de qualidade. O objetivo da garantia da qualidade é fornecer ao pessoal técnico e administrativo os dados necessários para serem informados sobre a qualidade do produto, ganhando, portanto, entendimento e confiança de que as ações para atingir a qualidade desejada do produto estão funcionando. Obviamente, se os dados fornecidos pela garantia da qualidade identificarem problemas, é responsabilidade do gerenciamento tratar desses problemas e aplicar os recursos necessários para resolver os problemas de qualidade. A garantia da qualidade de software é discutida em detalhe no Capítulo 21.

19.5 Resumo

A preocupação com a qualidade de sistemas de software cresceu à medida que o software ficou cada vez mais integrado a cada aspecto da vida cotidiana. No entanto, é difícil desenvolver uma descrição completa sobre qualidade de software. Neste capítulo, a qualidade foi definida como uma gestão de qualidade efetiva aplicada de modo a criar um produto útil que forneça valor mensurável para aqueles que o produzem e para aqueles que o utilizam.

Uma grande variedade de dimensões e fatores para qualidade de software foi proposta ao longo dos anos. Todos tentam definir um conjunto de características que, se atingidas, levarão a um software de alta qualidade. Os fa-

tores de qualidade de McCall e da ISO 9126 estabelecem características como confiabilidade, usabilidade, facilidade de manutenção, funcionalidade e portabilidade como indicadores de que a qualidade existe.

Todas as organizações envolvidas com software se deparam com o dilema da qualidade de software. Basicamente, todos querem construir sistemas de alta qualidade, mas o tempo e o esforço necessários para produzir um software “perfeito” não existem em um mundo orientado ao mercado. A pergunta passa a ser: devemos construir software “bom o suficiente”? Embora muitas empresas façam exatamente isso, há um grande porém que deve ser considerado.

Independentemente da estratégia escolhida, a qualidade tem, efetivamente, um custo que pode ser discutido em termos de prevenção, avaliação e falha. Os custos de prevenção incluem todas as ações de engenharia de software desenvolvidas para, em primeiro lugar, evitar defeitos. Os custos de avaliação estão associados às ações que avaliam os artefatos resultantes para determinar sua qualidade. Os custos de falhas englobam o preço de falhas internas e os efeitos externos que a má qualidade gera.

A qualidade de software é atingida por meio da aplicação de métodos de engenharia de software, práticas administrativas consistentes e controle de qualidade completo – todos suportados por uma infraestrutura de garantia da qualidade de software. Nos capítulos seguintes, o controle e a garantia da qualidade são discutidos com maior nível de detalhamento.

Problemas e pontos a ponderar

19.1 Descreva como você avaliaria a qualidade de uma universidade antes de se candidatar a ela. Quais fatores seriam importantes? Quais seriam críticos?

19.2 Garvin [Gar84] descreve cinco visões diferentes da qualidade. Dê um exemplo de cada uma delas usando um ou mais produtos eletrônicos conhecidos com os quais você esteja familiarizado.

19.3 Usando a definição de qualidade de software proposta na Seção 19.2, você acredita que seja possível criar um produto útil que gere valor mensurável sem usar um processo eficaz? Justifique sua resposta.

19.4 Acrescente duas outras perguntas a cada uma das dimensões de qualidade de Garvin apresentadas na Seção 19.2.1.

19.5 Os fatores de qualidade de McCall foram desenvolvidos durante a década de 1970. Praticamente todos os aspectos da computação mudaram drasticamente desde essa época e, mesmo assim, os fatores de McCall ainda se aplicam a software moderno. Você seria capaz de tirar alguma conclusão com base nesse fato?

19.6 Usando os subatributos citados na Seção 19.2.3 para o fator de qualidade “facilidade de manutenção” da ISO 9126, desenvolva um conjunto de perguntas que explore se esses atributos estão presentes ou não. Siga o exemplo mostrado na Seção 19.2.4.

19.7 Descreva o dilema da qualidade de software com suas próprias palavras.

19.8 O que é software “bom o suficiente”? Cite uma empresa específica e produtos específicos que você acredita terem sido desenvolvidos usando essa filosofia.

19.9 Considerando cada um dos quatro aspectos do custo da qualidade, qual você acredita ser o mais caro? Por quê?

19.10 Faça uma busca na Internet e encontre três outros exemplos de “riscos” para o público que podem ser diretamente atribuídos à baixa qualidade de software. Pense em iniciar sua pesquisa em <http://catless.ncl.ac.uk/risks>.

19.11 *Qualidade e segurança* são a mesma coisa? Explique.

19.12 Explique por que muitos de nós continuamos a viver da lei de Meskimen. Como isso se aplica a um negócio de software?

Leituras e fontes de informação complementares

Conceitos básicos sobre qualidade de software são considerados em livros de Chemutri (*Master Software Quality Assurance: Best Practices, Tools and Techniques for Software Developers*, Ross Publishing, 2010), Henry e Hanlon (*Software Quality Assurance*, Prentice Hall, 2008), Khan e seus colegas (*Software Quality: Concepts and Practice*, Alpha Science International, Ltd., 2006), O'Regan (*A Practical Approach to Software Quality*, Springer, 2002) e Daughtrey (*Fundamental Concepts for the Software Quality Engineer*, ASQ Quality Press, 2001).

Duvall e seus colegas (*Continuous Integration: Improving Software Quality and Reducing Risk*, Addison-Wesley, 2007), Tian (*Software Quality Engineering*, Wiley-IEEE Computer Society Press, 2005), Kandt (*Software Engineering Quality Practices*, Auerbach, 2005), Godbole (*Software Quality Assurance: Principles and Practice*, Alpha Science International, Ltd., 2004) e Galin (*Software Quality Assurance: From Theory to Implementation*, Addison-Wesley, 2003) apresentam tratados detalhados sobre SQA. A garantia da qualidade no contexto do processo ágil é considerada por Sterling (*Managing Software Debt*, Addison-Wesley, 2010) e Stamelos e Sfetos (*Agile Software Development Quality Assurance*, IGI Global, 2007).

Projeto consistente leva a qualidade de software elevada. Jayasawal e Patton (*Design for Trustworthy Software*, Prentice Hall, 2006) e Ploesch (*Contracts, Scenarios and Prototypes*, Springer, 2004) discutem ferramentas e técnicas para desenvolver software “robusto”.

A medição é um importante componente da engenharia de qualidade de software. Jones e Bonsignour (*The Economic of Software Quality*, Addison-Wesley, 2011), Ejiogu (*Software Metrics: The Discipline of Software Quality*, BookSurge Publishing, 2005), Kan (*Metrics and Models in Software Quality Engineering*, Addison-Wesley, 2002), e Nance e Arthur (*Managing Software Quality*, Springer, 2002) discutem importantes métricas e modelos relacionados com a qualidade. Os aspectos da qualidade de software orientados a equipes são considerados por Evans (*Achieving Software Quality through Teamwork*, Artech House Publishers, 2004).

Uma ampla gama de fontes de informação sobre qualidade de software se encontra à disposição na Internet. Uma lista atualizada das referências da Web (em inglês) pode ser encontrada no site: www.mhhe.com/pressman.

Técnicas de revisão 20

As revisões de software são como um “filtro” para a gestão de qualidade. Isso significa que elas são aplicadas em várias etapas durante o processo de engenharia de software e servem para revelar erros e defeitos que podem ser eliminados. As revisões de software “purificam” os artefatos da engenharia de software, até mesmo os modelos de requisitos e de projeto, dados de teste e código. Freedman e Weinberg [Fre90] discutem a necessidade de fazer revisões da seguinte maneira:

O trabalho técnico precisa de revisão pela mesma razão que o lápis precisa de borracha: *errar é humano*. A segunda razão para precisarmos de revisões técnicas é que, embora as pessoas sejam boas para “descobrir” alguns de seus próprios erros, vários tipos de erros escapam mais facilmente daquele que os cometeu do que de outras pessoas. O processo de revisão é, portanto, a resposta para a oração de Robert Burns:

*Oh! Que uma força conceda poder
para nos vermos como os outros nos veem*

Uma revisão – qualquer revisão – é uma forma de usar a diversidade de um grupo de pessoas para:

1. Apontar aperfeiçoamentos necessários no produto de uma única pessoa ou de uma equipe.

Conceitos-chave

amplificação de	
defeitos	433
bugs	432
defeitos	432
densidade de erros	435
eficácia dos custos	436
erros	432
manutenção	
de registros	442
relatórios de revisão	442
revisões informais	439
revisões por	
amostragem	444
revisões técnicas	441

PANORAMA

O que é? À medida que desenvolvemos o trabalho de engenharia de software, cometemos erros.

Não há motivo para se envergonhar disso – desde que tentemos, com muita dedicação, encontrar e corrigir os erros antes que sejam passados para os usuários. As revisões técnicas são o mecanismo mais eficaz para descobrir erros logo no início da gestão de qualidade.

Quem realiza? Os engenheiros de software realizam as revisões técnicas, também chamadas revisões paritárias, com seus colegas.

Por que é importante? Ao se descobrir um erro logo no início do processo, fica menos caro corrigi-lo. Além disso, os erros podem aumentar à medida que o processo prossegue. Portanto, um erro relativamente insignificante sem tratamento no início do processo pode ser amplificado e se transformar em um conjunto de erros graves para a sequência do projeto. Por fim, as revisões minimizam o tempo, devido à redução do

número de reformulações que serão necessárias ao longo do projeto.

Quais são as etapas envolvidas? A abordagem em relação às revisões vai variar dependendo do grau de formalidade escolhido. Em geral, são empregadas seis etapas, embora nem todas sejam usadas para todo tipo de revisão: planejamento, preparação, estruturação da reunião, anotação de erros, realização das correções (feita fora da revisão) e verificação de se as correções foram feitas apropriadamente.

Qual é o artefato? O artefato de uma revisão é uma lista de problemas e/ou erros que foram descobertos. Além disso, também é indicado o estado técnico do produto resultante.

Como garantir que o trabalho foi realizado corretamente? Primeiramente, escolha o tipo de revisão apropriado para o seu ambiente de desenvolvimento. Siga as diretrizes que levam a revisões bem-sucedidas. Se as revisões realizadas conduzirem a software de maior qualidade, elas foram feitas corretamente.

Revisões são como um filtro no fluxo de trabalho da gestão de qualidade. Um número muito pequeno de revisões e o fluxo será "sujo". Um número muito grande de revisões e o fluxo diminui muito e vira um gotejamento. Use métricas para determinar quais revisões funcionam e as enfatize. Elimine do fluxo as revisões ineficazes para acelerar o processo.

2. Confirmar as partes de um produto em que aperfeiçoamentos são indesejáveis ou desnecessários.
3. Obter trabalho técnico de qualidade mais uniforme, ou pelo menos mais previsível; qualidade que possa ser alcançada sem revisões, de modo a tornar o trabalho técnico mais gerenciável.

Diversos tipos de revisão podem ser realizados como parte do processo de engenharia de software. Cada um deles tem sua função. Um encontro informal na máquina de café é uma forma de revisão, caso sejam discutidos problemas técnicos. Uma apresentação formal da arquitetura do software para um público formado por clientes, pessoal técnico e gerencial também é uma forma de revisão. Neste livro, entretanto, focaremos as *revisões técnicas ou paritárias*, exemplificadas por *revisões informais*, *walkthroughs* e *inspeções*. Uma revisão técnica (RT) é o filtro mais eficiente do ponto de vista de controle da qualidade. Conduzida por engenheiros de software (e outros) para engenheiros de software, a RT é um meio eficaz para revelar erros e aumentar a qualidade do software.

20.1 Impacto de defeitos de software nos custos

No contexto da gestão da qualidade, os termos *defeito* e *falha* são sinônimos. Ambos indicam um problema de qualidade que é descoberto *depois* que o software é liberado para os usuários (ou para outra atividade estrutural dentro da

INFORMAÇÕES



Bugs, erros e defeitos

O objetivo do controle da qualidade de software e da gestão da qualidade em geral é, em sentido mais amplo, eliminar problemas de qualidade no software. Tais problemas são conhecidos por diversos nomes – *bugs*, *falhas*, *erros* ou *defeitos*, apenas para citar alguns. Esses termos são sinônimos ou existem diferenças sutis entre eles?

Neste livro é feita uma distinção clara entre *erro* (um problema de qualidade encontrado *antes* de o software ser liberado para os usuários finais) e *defeito* (um problema de qualidade encontrado apenas *depois* de o software ter sido liberado para os usuários finais).¹ Essa distinção é feita porque os erros e os defeitos podem acarretar impactos econômicos, comerciais, psicológicos e humanos muito diferentes. Os engenheiros de software têm a missão de encontrar e corrigir o maior número possível de erros antes dos clientes e/ou usuários finais. Devem-se evitar defeitos – pois (de modo justificável) criam uma imagem negativa do pessoal de software.

É importante notar, entretanto, que a distinção temporal entre erros e defeitos feita neste livro *não* é um pensamento dominante. O consenso geral na comunidade de engenharia de software é que defeitos e erros, falhas e bugs são sinônimos. Ou seja, o momento em que o problema foi encontrado não tem nenhuma influência no termo usado para descrevê-lo. Parte do argumento a favor desta visão é que, muitas vezes, fica difícil fazer uma distinção clara entre pré e pós-entrega (consideremos, por exemplo, um processo incremental usado no desenvolvimento ágil).

Independentemente da maneira escolhida para interpretar esses termos ou do momento em que um problema é descoberto, o que importa efetivamente é que os engenheiros de software devem se esforçar – *muito* – para encontrar problemas antes que seus clientes e usuários finais o façam. Caso tenha maior interesse nessa questão, uma discussão razoavelmente completa sobre a terminologia envolvendo bugs pode ser encontrada em www.software-development.ca/bugs.shtml.

¹ Se considerarmos o aperfeiçoamento na gestão de qualidade, um problema de qualidade que se propaga a partir de uma atividade estrutural do processo (por exemplo, **modelagem**) para outra (por exemplo, **construção**) também pode ser chamado de “defeito”, pois o problema deveria ter sido descoberto antes de um artefato (por exemplo, um modelo de projeto) ter sido “liberado” para a atividade seguinte.

gestão de qualidade). Em capítulos anteriores, usamos o termo *erro* para indicar um problema de qualidade que é descoberto por engenheiros de software (ou outros) *antes* de o software ser liberado para usuário final (ou para outra atividade estrutural dentro da gestão de qualidade).

O principal objetivo das revisões técnicas é encontrar erros durante o processo, para que não se tornem defeitos depois da liberação do software. A vantagem evidente das revisões técnicas é a descoberta precoce de erros, a fim de que não sejam propagados para a próxima etapa na gestão de qualidade.

Diversos estudos e análises sobre o tema indicam que as atividades de projeto introduzem de 50 a 65% de todos os erros (e, em última instância, todos os defeitos) durante o processo de software. Entretanto, técnicas de revisão demonstraram ser até 75% eficazes [Jon86] na descoberta de falhas de projeto. Detectando e eliminando um grande percentual desses erros, o processo de revisão reduz substancialmente o custo das atividades subsequentes na gestão de qualidade.

O principal objetivo de uma RTF é encontrar erros antes que eles passem para outra atividade de engenharia de software ou sejam liberados para o usuário.

20.2 Amplificação e eliminação de defeitos

Um *modelo de amplificação de defeitos* [IBM81] pode ser usado para ilustrar a geração e a detecção de erros durante o projeto e nas ações para geração de código de uma gestão de qualidade. O modelo é ilustrado esquematicamente na Figura 20.1. Um retângulo representa uma ação de engenharia de software. Durante a ação, os erros podem ser gerados inadvertidamente. Pode ser que a revisão falhe na descoberta de erros recentes e erros de etapas anteriores, resultando em uma série de erros que são passados para a etapa seguinte. Em alguns casos, esses erros passados e provenientes de etapas anteriores são amplificados (fator de amplificação x) pelo trabalho atual. As subdivisões dos retângulos representam cada uma dessas características e o percentual de eficiência para a detecção de erros, uma função do nível de detalhe da revisão.

"Algumas enfermidades, como dizem os médicos, são fáceis de curar logo no início, mas difíceis de diagnosticar... Com o tempo, quando elas não foram diagnosticadas e tratadas no início, tornam-se de fácil diagnóstico, mas de difícil cura."

Nicolau Maquiavel

A Figura 20.2 ilustra um exemplo hipotético da amplificação de defeitos para uma gestão de qualidade em que não foi realizada nenhuma revisão. De acordo com a figura, supõe-se que cada etapa de teste revele e corrija 50% de todos os erros de entrada sem introduzir nenhum erro novo (uma hipótese otimista). Dez defeitos preliminares de projeto são amplificados para 94 erros antes do início do teste. São liberados para o campo 20 erros latentes. A Figura 20.3 considera as mesmas condições, exceto que as revisões de projeto e código são realizadas como parte das ações de engenharia de software. Nesse caso, dez

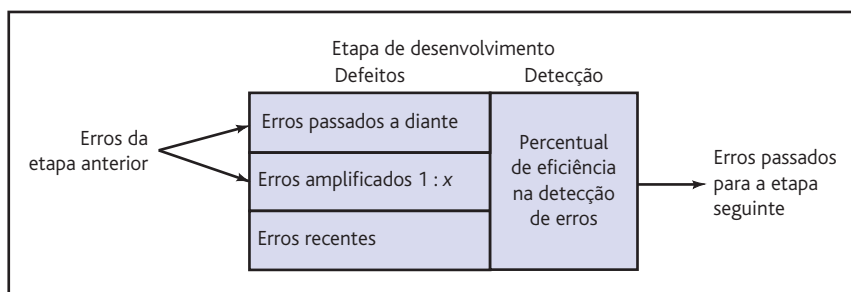


FIGURA 20.1 Modelo de amplificação de defeitos.

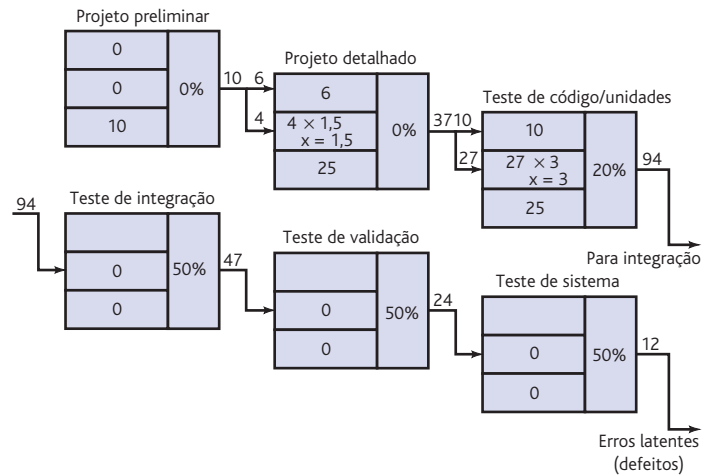


FIGURA 20.2 Amplificação de defeitos – sem revisão.

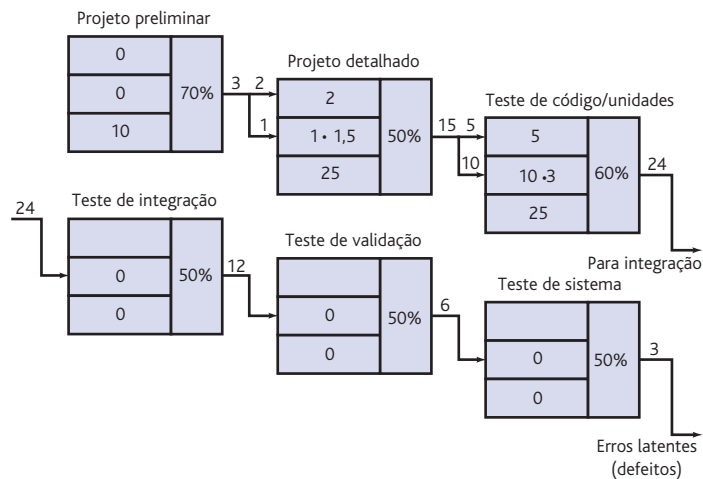


FIGURA 20.3 Amplificação de defeitos – com revisões realizadas.

erros de projeto preliminares (de arquitetura) iniciais são amplificados para 24 erros antes do início dos testes. Existem apenas três erros latentes. Podem ser estabelecidos os custos relativos associados à descoberta e à correção dos erros, bem como o custo total (com e sem revisão, para nosso exemplo hipotético). O número de erros revelados durante cada uma das etapas indicadas nas Figuras 20.2 e 20.3 é multiplicado pelo custo para eliminar um erro (1,5 unidades de custo para projeto, 6,5 unidades de custo antes do teste, 15 unidades de custo durante o teste e 67 unidades de custo depois da entrega).² Usando esses dados, o custo total para desenvolvimento e manutenção quando são realizadas revisões é de 783 unidades de custo. Quando não é feita nenhuma revisão, o custo total é de 2.177 unidades – aproximadamente três vezes mais caro.

Para fazer as revisões, é preciso investir tempo e esforço, e a empresa de desenvolvimento tem de disponibilizar recursos financeiros; os resultados do

² Esses multiplicadores são ligeiramente diferentes dos dados apresentados na Figura 19.2, que são mais comuns. Entretanto, servem para ilustrar bem a amplificação dos custos de defeitos.

exemplo anterior, no entanto, não deixam nenhuma dúvida de que podemos pagar agora ou então pagar muito mais no futuro.

20.3 Métricas de revisão e seu emprego

As revisões técnicas representam uma das muitas ações exigidas como parte de uma prática de engenharia de software adequada. Cada ação exige dedicação de nossa parte. Já que o esforço disponível para o projeto é finito, é importante para uma organização de engenharia de software compreender a eficácia de cada ação definindo um conjunto de métricas (Capítulo 30) que podem ser usadas para avaliar sua eficácia.

Apesar de poderem ser definidas muitas métricas para as revisões técnicas, um subconjunto relativamente pequeno pode nos dar uma boa ideia da situação. As métricas a seguir podem ser reunidas para cada revisão a ser realizada:

- *Esforço de preparação*, E_p – o esforço (em pessoas/hora) exigido para revisar um artefato antes da reunião de revisão.
- *Esforço de avaliação*, E_a – o esforço (em pessoas/hora) que é despendido durante a revisão em si.
- *Esforço de reformulação*, E_r – o esforço (em pessoas/hora) dedicado à correção dos erros revelados durante a revisão.
- *Tamanho do artefato de software*, TAS – uma medida do tamanho do artefato de software que foi revisto (por exemplo, o número de modelos UML ou o número de páginas de documento ou então o número de linhas de código).
- *Erros secundários encontrados*, Err_{sec} – o número de erros encontrados que podem ser classificados como secundários (exigindo menos para ser corrigidos do que algum esforço pré-especificado).
- *Erros graves encontrados*, Err_{graves} – o número de erros encontrados que podem ser classificados como graves (exigindo mais para ser corrigidos do que algum esforço pré-especificado).

Essas métricas podem ser ainda mais refinadas com a associação do tipo de artefato de software que foi revisto para as métricas.

20.3.1 Análise de métricas

Antes que a análise possa iniciar, devem ser realizados alguns cálculos simples. O esforço total de revisão e o número total de erros descobertos são definidos como:

$$E_{revisão} = E_p + E_a + E_r$$

$$Err_{tot} = Err_{sec} + Err_{graves}$$

A *densidade de erros* representa os erros encontrados por unidade do artefato de software revisto.

$$\text{Densidade de erros} = \frac{Err_{tot}}{TAS}$$

Por exemplo, se um modelo de requisitos for revisado para revelar erros, inconsistências e omissões, seria possível calcular a densidade de erros de diferentes maneiras. O modelo de requisitos contém 18 diagramas UML como parte de um total de 32 páginas de material descritivo. A revisão revela 18 erros secundários e 4 erros graves. Consequentemente, $Err_{tot} = 5,22$. A densidade de erros é de 1,2 erros por diagrama UML ou 0,68 erros por página de modelo de requisitos.

Se as revisões forem realizadas para diferentes tipos de artefatos (por exemplo, modelo de necessidades, modelo de projeto, código e casos de teste), a porcentagem de erros revelados para cada revisão pode ser calculada em relação ao número total de erros encontrados para todas as revisões. Além disso, podemos calcular também a densidade de erros para cada artefato.

Uma vez coletados os dados de várias revisões realizadas durante vários projetos, os valores médios da densidade de erros permitem estimar o número de erros a serem encontrados em um novo documento, mesmo que não seja revisado. Se, por exemplo, a densidade média de erros para um modelo de requisitos for de 0,6 erros por página, e um novo modelo de requisitos tiver 32 páginas, uma estimativa grosseira sugeriria que a equipe de software vai encontrar aproximadamente 19 ou 20 erros durante a revisão do documento. Se encontrados apenas 6 erros, fizemos um trabalho extremamente bom no desenvolvimento do modelo de requisitos, ou então a estratégia de revisão não foi suficientemente completa.

Logo que os testes forem realizados (Capítulos 22 a 26), é possível reunir dados de erros adicionais, incluindo o esforço necessário para encontrar e corrigir erros revelados durante os testes e a densidade de erros do software. Os custos associados à descoberta e à correção de um erro durante um teste podem ser comparados com aqueles obtidos nas revisões. Isso é discutido na Seção 20.3.2.

20.3.2 Eficácia dos custos de revisões

É difícil medir, em tempo real, a eficácia dos custos de qualquer revisão técnica. Uma empresa de engenharia de software pode avaliar a eficácia das revisões e seu custo-benefício apenas após as revisões terem sido completadas, as métricas de revisão terem sido reunidas e os dados médios terem sido calculados. E, por fim, a partir desse ponto, medir a qualidade do software (por meio de testes).

Voltando ao exemplo apresentado na Seção 20.3.1, determinou-se que a densidade média de erros para os modelos de requisitos foi de 0,6 por página. Verificou-se que o esforço necessário para corrigir um erro secundário do modelo (imediatamente após a revisão) exige 4 pessoas/hora. Verificou-se que o esforço necessário para corrigir um erro grave era de 18 pessoas/hora. Examinando-se os dados coletados na revisão, determina-se que os erros secundários ocorrem com uma frequência aproximada 6 vezes maior do que a de erros graves. Consequentemente, podemos estimar que o esforço médio para encontrar e corrigir um erro de requisitos durante uma revisão é de cerca de 6 pessoas/hora.

Os erros relacionados aos requisitos, revelados durante os testes, exigem uma média de 45 pessoas/hora para ser encontrados e corrigidos (não há dados disponíveis sobre a gravidade relativa do erro). Usando as médias observadas, obtemos:

$$\begin{aligned}\text{Esforço poupado por erro} &= \text{Err}_{\text{testes}} - \text{Err}_{\text{revisões}} \\ 45 - 6 &= 30 \text{ pessoas/hora/erro}\end{aligned}$$

Como foram encontrados 22 erros durante a revisão do modelo de requisitos, obteríamos uma economia aproximada de 660 pessoas/hora de esforço de testes. E essa economia refere-se apenas aos erros relacionados ao modelo de requisitos. Os erros associados a projeto e codificação aumentariam ainda mais o benefício geral. Em suma – o esforço poupado nos leva a ciclos de entrega mais curtos e a menor tempo de colocação do produto no mercado.

Em seu livro sobre revisões paritárias, Karl Wieggers [Wie02] discute os dados de grandes empresas que usam *inspeções* (um tipo de revisão técnica relativamente formal) como parte de suas atividades de controle da qualidade de software. A Hewlett-Packard (HP) relatou um retorno sobre o investimento de 10:1 para inspeções e citou que a entrega real do produto foi acelerada a uma média de 1,8 mês. A AT&T indicou que as inspeções reduziram o custo total de erros de software por um fator de 10, a qualidade foi incrementada em um grau de magnitude, e a produtividade aumentou 14%. Outros relatos informam benefícios semelhantes. As revisões técnicas (para projeto e outras atividades técnicas) fornecem uma relação custo-benefício demonstrável e efetivamente economizam tempo.

Entretanto, para muitas pessoas da área de software, essa afirmação é absurda. “As revisões tomam tempo”, argumentam os profissionais de software “e nós não temos tempo a perder!”. Eles defendem que tempo é um bem precioso em qualquer projeto de software e que a habilidade de revisar “detalhadamente todos os artefatos resultantes” absorve muito tempo.

Os exemplos apresentados anteriormente nesta seção indicam o contrário. Mais importante ainda, foram coletados dados do setor para revisões de software por mais de duas décadas e estão sintetizados de forma qualitativa nos gráficos da Figura 20.4.

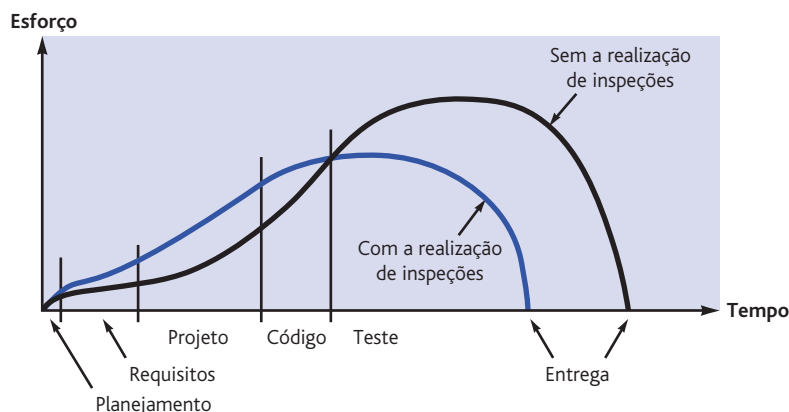


FIGURA 20.4 Esforço despendido com e sem o emprego de revisões.

Fonte: Adaptado de [Fag86].

Em relação à figura, o esforço despendido quando revisões são empregadas aumenta efetivamente no início do desenvolvimento de um módulo de software, mas esse investimento inicial para revisões rende dividendos, pois o esforço de testes e correções é reduzido. Igualmente importante, a data de entrega/distribuição para o desenvolvimento com revisões é anterior àquela sem o uso de revisões. As revisões não gastam tempo, elas poupam!

20.4 Revisões: um espectro de formalidade

As revisões técnicas devem ser aplicadas com um nível de formalidade adequado para o produto a ser construído, a cronologia do projeto e as pessoas que estão realizando o trabalho. A Figura 20.5 mostra um modelo de referência para revisões técnicas [Lai02] que identifica quatro características que contribuem para a formalidade com a qual uma revisão é conduzida.

As características do modelo de referência ajudam a definir o nível da formalidade da revisão. A formalidade de uma revisão aumenta quando: (1) são definidos explicitamente os papéis distintos para os revisores, (2) há um nível suficiente de planejamento e preparação para a revisão, (3) é definida uma estrutura distinta para a revisão (incluindo tarefas e artefatos internos) e (4) ocorre o follow-up pelos revisores para qualquer correção realizada.

Para entender o modelo de referência, vamos supor que decidimos revisar o projeto da interface para o **CasaSeguraGarantida.com**. Isso pode ser feito de várias formas, que vão desde relativamente informal a extremamente rigorosa. Caso ache que a abordagem informal é a mais apropriada, solicite a alguns colegas (pares) para que examinem o protótipo da interface em uma tentativa de descobrir problemas em potencial. Todos os participantes decidem que não haverá nenhuma preparação prévia, mas que você vai avaliar o protótipo de forma razoavelmente estruturada – verificando primeiramente o layout, em seguida a estética, depois as opções de navegação e assim por diante. Na qualidade de projetista, você decide fazer algumas anotações, mas nada formal.

O que acontece se a interface for fundamental para o sucesso do projeto inteiro? E se vidas humanas dependerem de uma interface ergonomicamen-



FIGURA 20.5 Modelo de referência para revisões técnicas.

te consistente? Será necessária uma abordagem mais rigorosa. Deverá ser formada uma equipe de revisão. Cada membro da equipe deve ter um papel específico a ser desempenhado – liderar a equipe, registrar as descobertas, apresentar o material e assim por diante. Cada revisor deve ter acesso ao artefato de software (neste caso, o protótipo da interface) antes da revisão e gastar tempo procurando erros, inconsistências e omissões. Um conjunto de tarefas específicas necessita ser conduzido tomando como base uma agenda desenvolvida antes de a revisão ocorrer. Os resultados da revisão precisam ser registrados formalmente, e a equipe deve decidir sobre a situação do artefato de software com base na revisão. Os membros da equipe de revisão também podem verificar se as correções foram feitas de forma apropriada.

Neste livro, consideramos duas grandes categorias de revisão técnica: revisões informais e revisões técnicas mais formais. Dentro de cada categoria, podem ser escolhidas várias abordagens distintas. Elas são apresentadas nas próximas seções.

20.5 Revisões informais

As revisões informais incluem um teste de mesa simples de um artefato de engenharia de software (com um colega), uma reunião informal (envolvendo mais de duas pessoas) com a finalidade de revisar um artefato, ou os aspectos orientados a revisões da programação em pares (Capítulo 5).

Um *teste de mesa* simples ou uma *reunião informal* realizada com um colega é uma revisão. Entretanto, por não haver planejamento ou preparação antecipados, cronograma ou estrutura de reuniões e *follow-up* sobre os erros encontrados, a eficiência dessas revisões é consideravelmente menor do que as abordagens mais formais. Mas um simples teste de mesa pode realmente revelar erros que, de outra forma, poderiam se propagar ainda mais na gestão de qualidade.

Uma forma de aumentar a eficácia de uma revisão do tipo teste de mesa é desenvolver um conjunto de listas de verificação simples para cada artefato produzido pela equipe de software. As questões levantadas na lista de verificação são genéricas, mas servirão como guia para os revisores verificarem o artefato. Por exemplo, vamos reexaminar um teste de mesa do protótipo da interface para **CasaSeguraGarantida.com**. Em vez de simplesmente ficar testando o protótipo na estação de trabalho do projetista, o projetista e um colega examinam o protótipo usando uma lista de verificação para interfaces:

- O layout é projetado usando convenções padronizadas? Da esquerda para a direita? De cima para baixo?
- A apresentação precisa de barra de rolagem?
- A cor e o posicionamento, o tipo e o tamanho dos elementos são usados eficientemente?
- Todas as opções de navegação ou funções representadas estão no mesmo nível de abstração?
- Todas as opções de navegação são claramente identificadas?

e assim por diante. Qualquer erro ou problema verificado pelos revisores é registrado pelo projetista para ser solucionado mais adiante. Poderiam ser programados testes de mesa de forma improvisada, ou eles seriam compulsórios, como parte da boa prática de engenharia de software. Em geral, a quantidade de material a ser revisada é relativamente pequena, e o tempo total gasto em um teste de mesa vai um pouco além de uma ou duas horas.

No Capítulo 5, descrevemos a *programação em pares* da seguinte maneira: A XP recomenda que duas pessoas trabalhem juntas em uma mesma estação de trabalho para criar código para uma história. Isso disponibiliza um mecanismo para a solução de problemas em tempo real (duas cabeças normalmente funcionam melhor do que uma) e a garantia da qualidade em tempo real.

A programação em pares pode ser caracterizada como um teste de mesa contínuo. Em vez de agendar em momento para a revisão ser realizada, a programação em pares estimula a revisão contínua enquanto um artefato de software (projeto ou código) é criado. A vantagem é a descoberta imediata de erros e, conseqüentemente, maior qualidade do artefato final.

Em sua discussão sobre a eficácia da programação em pares, Williams e Kessler [Wil00] afirmam:

Tanto evidências incidentais quanto evidências estatísticas iniciais indicam que a programação em pares é uma técnica poderosa para gerar, de forma produtiva, produtos de software de alta qualidade. A dupla trabalha e compartilha ideias para lidar com as complexidades do desenvolvimento de software. Eles realizam inspeções continuamente nos artefatos um do outro, levando à forma mais precoce e eficiente possível de eliminação de defeitos. Além disso, cada um faz o outro se manter atentamente focado na tarefa em questão.

Alguns engenheiros de software sustentam que a redundância inerente da programação em pares é um desperdício de recursos. Afinal de contas, por que alocar duas pessoas para um trabalho que apenas uma é capaz de realizar? A resposta a essa pergunta pode ser encontrada na Seção 20.3.2. Se a qualidade do artefato da programação em pares for significativamente melhor do que o trabalho individual, as economias relacionadas à qualidade são plenamente capazes de justificar a “redundância” implícita na programação em pares.

INFORMAÇÕES



Listas de verificação para revisões

Mesmo quando as revisões são bem organizadas e apropriadamente conduzidas, não é uma má ideia munir os revisores de um recurso a mais. Vale a pena ter uma lista de verificação que forneça a cada revisor as perguntas que devem ser feitas sobre o artefato específico que está passando por revisão.

Um dos conjuntos mais completos de listas de verificação para revisões foi desenvolvido pela NASA no Goddard Space Flight Center e se encontra disponível em

<http://www.hq.nasa.gov/office/codeq/software/ComplexElectronics/checklists.htm>.

Outras listas de verificação para revisões técnicas muito úteis também foram propostas por:

Process Impact

www.processimpact.com/pr_goodies.shtml

Macadamian

www.macadamian.com

20.6 Revisões técnicas formais

Revisão técnica formal (RTF) é uma atividade de controle da qualidade de software realizada por engenheiros de software (e outros profissionais). Os objetivos de uma RTF são: (1) descobrir erros na função, lógica ou implementação para qualquer representação do software; (2) verificar se o software que está sendo revisado atende aos requisitos; (3) garantir que o software foi representado de acordo com padrões predefinidos; (4) obter software que seja desenvolvido de maneira uniforme; e (5) tornar os projetos mais gerenciáveis. Além disso, a RTF serve como base de treinamento, possibilitando que engenheiros mais novos observem diferentes estratégias para análise, projeto e implementação de software. A RTF também serve para promover respaldo e continuidade, pois muitas pessoas conhecem partes do software que, de outra maneira, jamais teriam visto.

Na verdade, RTF é uma classe de revisões que inclui *walkthroughs* e *inspeções*. Cada RTF é realizada como uma reunião e será bem-sucedida somente se for apropriadamente planejada, controlada e tiver a participação de todos os envolvidos. Nas seções a seguir são apresentadas orientações similares àquelas para um *walkthrough* na forma de uma revisão técnica formal representativa. Caso tenha interesse em inspeções de software, bem como queira informações adicionais sobre *walkthrough*, consulte [Rad02], [Wie02] ou [Fre90].

“Não há impulso maior do que aquele de um homem revisar o trabalho de outro homem.”

Mark Twain

20.6.1 A reunião de revisão

Seja qual for o formato de RTF escolhido, cada reunião de revisão deve observar as seguintes restrições:

- Devem estar envolvidas de três a cinco pessoas em uma revisão (tipicamente).
- Deve ocorrer uma preparação antecipada, porém não deve tomar mais do que duas horas de trabalho de cada pessoa.
- A duração da reunião de revisão deve ser de menos de duas horas.

Dadas essas restrições, deve ser óbvio que uma RTF se concentre em uma parte específica (e pequena) do software. Por exemplo, em vez de tentar revisar um projeto inteiro, os *walkthroughs* são realizados para cada componente ou para um pequeno grupo de componentes. Afunilando-se o foco, a RTF terá maior probabilidade de revelar erros.

O foco da RTF é um artefato (por exemplo, parte de um modelo de requisitos, o projeto detalhado de um componente, o código-fonte de um componente). O indivíduo que desenvolveu o artefato – o *produtor* – informa ao líder de projeto que o artefato está concluído e que é necessário fazer uma revisão. O líder de projeto contata um *líder de revisão*, que avalia o artefato em termos de completude, gera cópias dos materiais resultantes e as distribui para dois ou três *revisores* para preparação prévia. Espera-se que cada revisor passe de uma a duas horas revisando o artefato, tomando notas e familiarizando-se com o trabalho realizado. Ao mesmo tempo, o líder da

O *Formal Inspection Guidebook* do SATC da NASA pode ser baixado de http://www.everyspec.com/NASA/NASA-General/NASA-GB-A302_2418/.

Uma RTF se concentra em uma parte relativamente pequena de um artefato.

Em algumas situações, é uma boa ideia que outra pessoa, que não seja o produtor, analise o produto que está sendo submetido a uma revisão. Isso leva a uma interpretação literal do artefato e a um melhor diagnóstico dos erros.

reunião de revisão também revisa o artefato e estabelece uma agenda para a reunião de revisão, que normalmente é marcada para o dia seguinte.

Uma reunião de revisão tem a participação de um líder de revisão, todos os revisores e o produtor. Um dos revisores assume o papel de *registrador*, isto é, o indivíduo que registra (por escrito) todas as questões importantes surgidas durante a revisão. A RTF começa com uma introdução da agenda e uma breve introdução por parte do produtor. Então, o produtor “repassa” (*walkthrough*) o artefato, explicando o material, enquanto os revisores levantam questões com base em sua preparação prévia. Quando são descobertos problemas ou erros válidos, o registrador toma nota de cada um deles.

Ao final da revisão, todos os participantes da RTF devem decidir se: (1) aceitam o artefato sem as modificações adicionais, (2) rejeitam o artefato devido a erros graves (uma vez corrigidos, outra revisão deve ser realizada) ou (3) aceitam o artefato provisoriamente (foram encontrados erros secundários que devem ser corrigidos, mas não haverá nenhuma outra revisão). Após uma tomada de decisão, todos os participantes da RTF assinam um documento de aprovação, indicando sua participação na revisão e sua concordância com as descobertas da equipe de revisão.

20.6.2 Relatório de revisão e manutenção de registros

Durante a RTF, um revisor (o registrador) registra ativamente todos os problemas levantados. Eles são sintetizados ao final da reunião de revisão e é produzida uma *lista de problemas de revisão*. Além disso, um *relatório sintetizado da revisão técnica formal* é completado. O relatório sintetizado de revisão deve responder a três questões:

1. O que foi revisado?
2. Quem revisou?
3. Quais foram as descobertas e as conclusões?

O relatório sintetizado da revisão é um formulário de uma página (com possíveis anexos). Ele torna-se um registro histórico do projeto e pode ser distribuído ao líder do projeto e a outras partes envolvidas.

A lista de problemas de revisão atende a dois propósitos: (1) identificar áreas problemáticas no artefato e (2) servir como uma lista de verificação de itens de ação que orienta o produtor à medida que são feitas as correções. Normalmente é anexada uma lista de problemas ao relatório sintetizado.

Devemos estabelecer um procedimento de acompanhamento para garantir que os itens contidos na lista de problemas tenham sido corrigidos apropriadamente. Caso isso não seja feito, é possível que problemas levantados possam “ficar para trás”. Uma das estratégias é atribuir a responsabilidade pelo acompanhamento (follow-up) ao líder da revisão.

20.6.3 Diretrizes de revisão

Devem-se estabelecer previamente diretrizes para a realização de revisões técnicas formais, distribuídas a todos os revisores, ter a anuência de todos e, então, segui-las à risca. Uma revisão não controlada muitas vezes pode ser

pior do que não fazer nenhuma revisão. A seguir, apresentamos um conjunto mínimo de diretrizes para revisões técnicas formais:

1. *Revisar o produto, não o produtor.* Uma RTF envolve pessoas e egos. Conduzida de forma apropriada, a RTF deve deixar todos os seus participantes com uma agradável sensação de dever cumprido. Conduzido de forma imprópria, a RTF pode assumir a aura de uma inquisição. Os erros devem ser apontados gentilmente; o clima da reunião deve ser descontraído e construtivo; o intuito não deve ser o de causar embaraços ou menosprezo. O líder da revisão deve conduzir a reunião de revisão de forma a garantir que o clima seja mantido e as atitudes sejam apropriadas; além disso, deve interromper imediatamente uma revisão que começou a sair do controle.
2. *Estabelecer uma agenda e mantê-la.* Um dos principais males de reuniões de todos os tipos é desviar do foco. Uma RTF deve ser mantida em seu rumo e prazo estabelecidos. O líder da revisão tem a responsabilidade de manter o cronograma da reunião e não deverá ficar receoso em alertar quando alguém estiver saindo do foco.
3. *Limitar debates e refutação.* Quando uma questão é levantada por um revisor, talvez não haja um acordo universal sobre seu impacto. Em vez de perder tempo debatendo a questão, o problema deve ser registrado para posterior discussão, fora da reunião.
4. *Enunciar as áreas problemáticas, mas não tentar resolver todo problema registrado.* Uma revisão não é uma sessão para solução de problemas. A solução de um problema pode, muitas vezes, ser realizada pelo próprio produtor ou com a ajuda de apenas outro colega. A solução de problemas deve ser deixada para depois da reunião de revisão.
5. *Tomar notas.* Algumas vezes é uma boa ideia o registrador fazer apontamentos em um quadro, de modo que os termos e as prioridades possam ser avaliados por outros revisores à medida que as informações são registradas. Como alternativa, as anotações podem ser digitadas diretamente em um notebook.
6. *Limitar o número de participantes e insistir na preparação antecipada.* Duas cabeças funcionam melhor do que uma, mas catorze cabeças não funcionam, necessariamente, melhor do que quatro. Mantenha o número de pessoas envolvidas no mínimo necessário. Entretanto, todos os membros da equipe de revisão devem se preparar com antecedência. O líder da revisão deve solicitar comentários escritos (fornecendo uma indicação de que o revisor examinou o material).
7. *Desenvolver uma lista de verificação para cada artefato que provavelmente será revisado.* A lista de verificação ajuda o líder da revisão a estruturar a RTF e auxilia cada revisor a se concentrar nas questões importantes. As listas de verificação devem ser desenvolvidas para análise, projeto, código e até mesmo para o teste dos artefatos.
8. *Alocar os recursos e programar o tempo para as RTFs.* Para as revisões serem eficazes, elas devem ser programadas como tarefas durante a gestão de qualidade. Além disso, deve-se programar o tempo para as inevitáveis modificações que ocorrerão como resultado de uma RTF.

Não aponte erros de forma áspera. Uma maneira gentil é fazer uma pergunta que possibilite ao produtor descobrir o próprio erro.

"Uma reunião é, muitas vezes, um evento em que minutos são perdidos e horas são desperdiçadas."

Autor desconhecido

"É uma das mais belas recompensas da vida, que nenhum homem pode, sinceramente, tentar ajudar outro sem ajudar a si mesmo."

**Ralph Waldo
Emerson**

9. *Realizar treinamento significativo para todos os revisores.* Para serem eficazes, todos os participantes de uma revisão deveriam receber algum treinamento formal. O treinamento deve enfatizar tanto questões relacionadas a processos quanto o lado psicológico das revisões. Freedman e Weinberg [Fre90] estimam uma curva de aprendizado para cada vinte pessoas que participarão efetivamente de revisões.

10. *Revisar revisões iniciais.* Um interrogatório pode ser benéfico na descoberta de problemas no próprio processo de revisão. Os primeiros artefatos a serem revisados devem ser as próprias diretrizes de revisão.

Como muitas variáveis (por exemplo, o número de participantes, o tipo de artefatos resultantes, o timing, ou tempo adequado e a duração, a abordagem de revisão específica) causam impacto sobre uma revisão bem-sucedida, uma organização de software deve fazer experimentos para determinar qual abordagem funcionará melhor em um contexto local.

20.6.4 Revisões por amostragem

Em um ambiente ideal, todo artefato de engenharia de software passaria por uma revisão técnica formal. No mundo real dos projetos de software, os recursos são limitados, e o tempo é escasso. Como consequência, as revisões são muitas vezes esquecidas, muito embora seu valor como mecanismo de controle de qualidade seja reconhecido.

Thelin e seus colegas [The01] sugerem um processo de revisão por amostragem em que amostras de todos os artefatos da engenharia de software sejam inspecionadas para determinar quais são mais suscetíveis a erro. Recursos completos de RTF são, então, direcionados apenas para os artefatos com maior probabilidade de ser suscetíveis a erro (com base em dados coletados durante a amostragem).

Para ser eficaz, o processo de revisão por amostragem deve tentar quantificar os produtos de trabalho que são alvos primários para as RTFs completas. Para conseguir isso, são sugeridas as seguintes etapas [The01]:

1. Inspeccionar uma fração α_i de cada artefato de software resultante i . Registrar o número de falhas f_i encontradas em α_i .
2. Desenvolver uma estimativa total do número de falhas contido no artefato i , multiplicando f_i por $1/\alpha_i$.
3. Classificar os artefatos em ordem decrescente, de acordo com a estimativa total do número de falhas contidas em cada um deles.
4. Concentrar recursos de revisão disponíveis naqueles artefatos que possuem o maior número estimado de falhas.

A fração do trabalho amostrada deve ser representativa do artefato como um todo e suficientemente grande para ser significativa para os revisores que fazem a amostragem. À medida que α_i aumenta, a probabilidade de que a amostra seja uma representação válida do artefato também aumenta. Entretanto, os recursos necessários para realizar a amostragem também aumen-

As revisões tomam tempo, mas é um tempo bem empregado. Entretanto, se o tempo for curto e você não tiver outra opção, não descarte as revisões. Em vez disso, use revisões por amostragem.

tam. Uma equipe de engenharia de software deve estabelecer o melhor valor para a_i para os tipos de artefatos produzidos.³

CASASEGURA**Questões de qualidade**

Cena: Escritório de Doug Miller no início do projeto de software *CasaSegura*.

Atores: Doug Miller (gerente da equipe de engenharia de software do *CasaSegura*) e outros membros da equipe de engenharia de software do produto.

Conversa:

Doug: Sei que não investimos tempo para desenvolver um plano de qualidade para este projeto, mas nós já estamos nele e temos de considerar a qualidade... certo?

Jamie: Com certeza. Já decidimos que, enquanto desenvolvermos o modelo de requisitos [Capítulos 9 a 11], Ed se comprometeu a desenvolver um procedimento de testes para cada requisito.

Doug: Isso é realmente interessante, mas não vamos esperar até que os testes avaliem a qualidade, não é mesmo?

Vinod: Não! Obviamente, não. Temos revisões programadas no plano de projeto para esse incremento de software. Começaremos o controle da qualidade com as revisões.

Jamie: Estou bastante preocupado, pois acredito que não teremos tempo suficiente para realizar todas as revisões. Na realidade, eu sei que não teremos.

Doug: Huumm. Então o que você sugere?

Jamie: Acho que devemos escolher os elementos mais críticos dos modelos de requisitos e de projeto e os revisarmos.

Vinod: Mas e se deixarmos alguma coisa de lado em uma parte do modelo em que não fizemos uma revisão?

Shakira: Li alguma coisa sobre uma técnica de amostragem [Seção 20.6.4] que poderia nos ajudar a determinar candidatas a uma revisão. (Shakira explica a estratégia.)

Jamie: Talvez... mas não estou certo se teremos tempo até mesmo para amostrar cada elemento dos modelos.

Vinod: O que você quer que façamos, Doug?

Doug: Usemos algo da Extreme Programming (Programação Extrema) [Capítulo 5]. Desenvolveremos os elementos de cada modelo em pares – duas pessoas – e faremos uma revisão informal de cada um deles à medida que prosseguimos. Em seguida, separaremos elementos “críticos” para uma revisão mais formal em equipe, mas manteremos essas revisões em um número mínimo. Dessa forma, tudo será examinado por mais de uma pessoa, mas ainda manteremos nossas datas de entrega.

Jamie: Isso significa que teremos de revisar o cronograma.

Doug: Que assim seja. A qualidade prevalece sobre o cronograma nesse projeto.

20.7 Avaliações post-mortem

Muitas lições podem ser aprendidas se uma equipe de software dedicar tempo para avaliar os resultados de um projeto de software depois que o software tiver sido entregue aos usuários finais. Baaz e seus colegas [Baa10] sugerem o uso de uma *avaliação post-mortem* (PME, *post-mortem evaluation*) como mecanismo para determinar o que deu certo e o que deu errado, quando o processo e a prática de engenharia de software são aplicados em um projeto específico.

Ao contrário de uma RTF, que se concentra em um artefato específico, uma PME examina o projeto de software inteiro, focalizando tanto as “*exce-lências*” (isto é, realizações e experiências positivas) quanto os *desafios* (problemas e experiências negativas)” [Baa10]. Frequentemente realizada em for-

³ Thelin e seus colegas realizaram uma simulação detalhada que pode ajudar a fazer essa determinação. Consulte [The01] para mais detalhes.

ma de workshop, os participantes de uma PME são os membros da equipe de software e os envolvidos. A intenção é identificar as excelências e os desafios, e extrair as lições aprendidas de ambos. O objetivo é sugerir aprimoramentos tanto no processo quanto na prática futuros.

20.8 Resumo

O objetivo de qualquer revisão técnica é encontrar erros e revelar problemas que teriam um impacto negativo sobre o software a ser entregue. Quanto antes um erro for descoberto e corrigido, menor a probabilidade de que ele se propague a outros artefatos de engenharia de software, amplificando o problema e gerando um esforço maior para corrigi-lo.

Para determinar se as atividades de controle da qualidade estão funcionando, é preciso reunir um conjunto de métricas. As métricas de revisão focam-se no esforço necessário para conduzir uma revisão e nos tipos e gravidade dos erros descobertos durante a revisão. Assim que os dados sobre métricas tiverem sido coletados, eles poderão ser usados para avaliar a eficácia das revisões realizadas. Os dados do setor indicam que as revisões geram um retorno significativo sobre o investimento.

Um modelo de referência da formalidade de uma revisão identifica os papéis desempenhados pelas pessoas, o planejamento e a preparação, a estrutura das reuniões, a abordagem e a verificação da correção como indicadores do nível de formalidade com que uma revisão é conduzida. As revisões informais são superficiais por natureza, mas ainda assim podem ser eficientes na descoberta de erros. As revisões formais são mais estruturadas e têm maior probabilidade de resultar em software de alta qualidade.

As revisões informais caracterizam-se por planejamento e preparação mínimos e poucos registros. Os testes de mesa e a programação em pares se enquadram na categoria de revisão informal.

Uma revisão técnica formal é uma reunião estilizada que se mostrou extremamente eficaz na descoberta de erros. Os *walkthroughs* e as inspeções estabelecem papéis definidos para cada revisor, estimulam a antecipação do planejamento e da preparação, exigem a aplicação de diretrizes de revisão já definidas e tornam compulsórios a manutenção de registros e o relatório de estado das revisões. As revisões por amostragem podem ser usadas quando não é possível realizar revisões técnicas formais para todos os artefatos.

Problemas e pontos a ponderar

20.1 Explique a diferença entre *erro* e *defeito*.

20.2 Por que não podemos simplesmente aguardar até que os testes terminem para descobrir e corrigir todos os erros de software?

20.3 Suponha que tenham sido introduzidos 10 erros no modelo de requisitos e que cada erro será amplificado no projeto detalhado por um fator de 2:1 e que outros 20 erros de projeto sejam introduzidos e então amplificados na razão de 1,5:1 no código em que mais 30 erros são introduzidos. Suponha ainda que o teste de todas as unidades vai

encontrar 30% de todos os erros, a integração descobrirá 30% dos erros remanescentes, e os testes de validação encontrarão 50% dos erros restantes. Não é realizada nenhuma revisão. Quantos erros serão liberados para o campo?

20.4 Reconsidere a situação descrita no Problema 20.3, mas suponha agora que sejam efetuadas as revisões de requisitos, de projeto e de código e que elas terão uma eficiência de 60% na descoberta de todos os erros nessa etapa. Quantos erros chegarão ao campo?

20.5 Reconsidere a situação descrita nos Problemas 20.3 e 20.4. Se cada um dos erros que chegam aos usuários custar US\$ 4.800 para ser encontrado e corrigido, e cada erro encontrado na revisão custar US\$ 240 para ser encontrado e corrigido, quanto é poupado em termos monetários com a realização das revisões?

20.6 Descreva com suas próprias palavras o significado da Figura 20.4.

20.7 Qual das características do modelo de referência você imagina que possua a maior influência sobre a formalidade da revisão? Justifique.

20.8 Você seria capaz de imaginar alguns casos em que um teste de mesa poderia criar problemas em vez de gerar benefícios?

20.9 A revisão técnica formal é eficaz apenas se todos estiverem preparados com antecedência. Como se reconhece um participante da revisão que não se preparou? O que você faria caso fosse o líder da revisão?

20.10 Considerando-se todas as diretrizes para a revisão apresentada na Seção 20.6.3, o que você acha que é mais importante e por quê?

Leituras e fontes de informação complementares

Embora na última década tenham sido escritos relativamente poucos livros novos sobre revisões de software, adições recentes à literatura incluem livros de McCann (*Cost-Benefit Analysis of Quality Practices*, IEEE Press, 2012), Wong (*Modern Software Review*, IRM Press, 2006) e Young (*Project Requirements: A Guide to Best Practices*, Management Concepts, 2006). Contribuições mais antigas que oferecem orientações úteis incluem: Radice (*High Quality, Low Cost Software Inspections*, Paradoxicon Publishers, 2002), Wieggers (*Peer Reviews in Software: A Practical Guide*, Addison-Wesley, 2001) e Gilb e Graham (*Software Inspection*, Addison-Wesley, 1993). Freedman e Weinberg (*Handbook of Walkthroughs, Inspections and Technical Reviews*, Dorset House, 1990) permanece um clássico e continua a fornecer informações úteis sobre esse importante assunto.

Livros de Rubin (*Essential Scrum: A Practical Guide to the Most Popular Agile Process*, Addison-Wesley, 2012) e Adkins (*Coaching Agile Teams: A Companions for Scrum-Masters, Agile Coaches, and Project Managers in Transition*, Addison-Wesley, 2010) descrevem os papéis das revisões nos processos ágeis de software.

Uma ampla gama de fontes de informação sobre revisões de software se encontra à disposição na Internet. Uma lista atualizada das referências da Web (em inglês) pode ser encontrada no site: www.mhhe.com/pressman.

21 Garantia da qualidade de software

Conceitos-chave

abordagens formais	455
confiabilidade do software	459
elementos da garantia da qualidade de software . .	450
estatística da garantia da qualidade de software . .	456
metas	454
padrão ISO 9001:2008	462
plano de SQA	463
recursos para gestão da qualidade	452
segurança do software . .	460
Seis Sigma	458
tarefas da SQA	453

A abordagem de engenharia de software descrita neste livro tem um único objetivo: produzir software de alta qualidade no prazo condizente com as necessidades dos envolvidos. Mesmo assim, muitos leitores vão se sentir desafiados pela pergunta: “O que é qualidade de software?” Philip Crosby [Cro79], em seu livro histórico sobre qualidade, fornece uma resposta sarcástica a essa pergunta:

O problema da gestão da qualidade não é o que as pessoas não sabem a seu respeito, mas sim o que elas pensam que sabem...

Nesse aspecto, a qualidade tem muita semelhança com o sexo. Todo mundo é a favor. (Sob certas condições, é claro.) Todo mundo acha que entende. (Mesmo que não queira explicá-lo.) Todo mundo pensa que a consumação é apenas uma questão de seguir as inclinações naturais. (Afinal, nos entendemos de algum modo.) E, é claro, a maioria das pessoas acha que os problemas nessas áreas são causados pelas outras pessoas. (Se ao menos tiverem paciência para fazer as coisas direito.)

PANORAMA

(1) definir explicitamente o seu significado, o que realmente se quer dizer com “qualidade de software”, (2) criar um conjunto de atividades que ajude a garantir que todo artefato resultante da engenharia de software apresente alta qualidade, (3) realizar atividades de garantia e controle da qualidade do software em todos os projetos de software, (4) usar métricas para desenvolver estratégias para aperfeiçoar o processo de software e, conseqüentemente, a qualidade do produto final.

Quem realiza? Todos os envolvidos no processo de engenharia de software são responsáveis pela qualidade.

Por que é importante? Ou você faz certo da primeira vez ou faz tudo de novo. Se uma equipe de software buscar a qualidade em todas as atividades de engenharia de software, a quantidade de retrabalho será reduzida. Isso resulta em custos menores e, mais importante, menor tempo para disponibilização do produto no mercado.

O que é? Não basta dizer simplesmente que a qualidade do software é importante. É preciso:

Quais são as etapas envolvidas? Antes das atividades de garantia da qualidade de software (SQA, software quality assurance) iniciarem, é importante definir *qualidade de software* em diferentes níveis de abstração. A partir do momento em que se entende o que é qualidade, a equipe de software deve identificar um conjunto de atividades de SQA para filtrar erros do artefato antes que passem adiante.

Qual é o artefato? É criado um Plano de Garantia da Qualidade de Software para definir a estratégia de SQA de uma equipe de software. Durante a modelagem e a codificação, o artefato principal da SQA é o resultado das revisões técnicas (Capítulo 20). Durante os testes (Capítulos 22 a 26), são produzidos procedimentos e planos de testes. Também podem ser gerados outros produtos associados ao aperfeiçoamento do processo.

Como garantir que o trabalho foi realizado corretamente? Encontrando erros antes de se tornarem defeitos! Ou seja, trabalhando para melhorar a eficiência da remoção dos defeitos (Capítulo 30), reduzindo, portanto, a quantidade de retrabalho que a equipe de software terá de realizar.

De fato, qualidade é um conceito desafiador – que foi apresentado em detalhes no Capítulo 19.¹

Alguns desenvolvedores de software continuam a acreditar que a qualidade do software é algo com que começamos a nos preocupar depois que o código é gerado. Nada poderia estar mais distante da verdade! A *garantia da qualidade de software* (SQA, *software quality assurance*, muitas vezes denominada *gestão da qualidade*), é uma atividade universal (Capítulo 3) aplicada em todo processo do software.

A garantia da qualidade de software (SQA) abrange: (1) um processo de SQA, (2) tarefas específicas de garantia e controle da qualidade (inclusive revisões técnicas e uma estratégia de testes multicamadas), (3) prática efetiva de engenharia de software (métodos e ferramentas), (4) controle de todos os artefatos de software e as mudanças feitas nesses produtos (Capítulo 29), (5) um procedimento para garantir a conformidade com os padrões de desenvolvimento de software (quando aplicáveis) e (6) mecanismos de medição e de geração de relatórios.

Este capítulo concentra-se em problemas de gerenciamento e em atividades específicas de processos que permitem garantir a uma organização de software fazer “as coisas certas, no momento certo e da maneira certa”.

21.1 Plano de fundo

A garantia e o controle da qualidade são atividades essenciais para qualquer empresa de produtos a serem usados por terceiros. Antes do século 20, o controle de qualidade era responsabilidade exclusiva do artesão que construía um produto. À medida que o tempo foi passando e técnicas de produção em massa tornaram-se comuns, o controle de qualidade tornou-se uma atividade realizada por outras pessoas e não por aquelas que constroem o produto.

A primeira função formal da garantia e do controle da qualidade foi introduzida no Bell Labs em 1916 e difundiu-se rapidamente no mundo da manufatura. Durante os anos 1940, foram sugeridas abordagens de controle de qualidade mais formais. Elas contavam com medições e aprimoramento contínuo do processo [Dem86] como elementos-chave da gestão de qualidade.

A história da garantia da qualidade no desenvolvimento de software é análoga à história da qualidade na fabricação de hardware. Durante os primórdios da computação (décadas de 1950 e 1960), a qualidade era responsabilidade exclusiva do programador. Padrões para a garantia da qualidade foram introduzidos no desenvolvimento de software por parte de empresas terceirizadas pela indústria militar durante a década de 1970 e difundiram-se rapidamente para o desenvolvimento de software no mundo comercial [IEE93a]. Estendendo a definição apresentada anteriormente, a garantia da qualidade de software é um “padrão de ações planejado e sistematizado” [Sch98c], ações essas exigidas para garantir alta qualidade no software. O escopo da responsabilidade da garantia da qualidade poderia ser mais bem caracterizado parafraseando-se um famoso comercial de uma indústria automobilística:

“Você cometeu muitos erros errados.”

Yogi Berra

¹ Caso não tenha lido o Capítulo 19, você deve fazê-lo agora.

“A Qualidade é a Tarefa nº 1”. A implicação para a área de software é que os elementos distintos têm as suas responsabilidades sobre a garantia da qualidade de software – engenheiros de software, gerentes de projeto, clientes, vendedores e os indivíduos que trabalham em um grupo de SQA.

O grupo de SQA funciona como um serviço de defesa do cliente. Ou seja, as pessoas que realizam a SQA devem examinar o software sob o ponto de vista do cliente. O software atende adequadamente aos fatores de qualidade citados no Capítulo 19? As práticas de engenharia de software foram conduzidas de acordo com padrões preestabelecidos? As disciplinas técnicas desempenharam apropriadamente seus papéis como parte da atividade de SQA? O grupo de SQA tenta responder a essas e outras perguntas para garantir que a qualidade do software seja mantida.

21.2 Elementos de garantia da qualidade de software

A garantia da qualidade de software engloba um amplo espectro de preocupações e atividades que se concentram na gestão da qualidade de software. Elas podem ser sintetizadas da seguinte maneira [Hor03]:

Padrões. O IEEE, a ISO e outras organizações de padronizações produziram uma ampla gama de padrões para engenharia de software e documentos relacionados. Os padrões podem ser adotados voluntariamente por uma organização de engenharia de software ou impostos pelo cliente ou outros envolvidos. O papel da SQA é garantir que os padrões que tenham sido adotados sejam seguidos e que todos os produtos resultantes estejam em conformidade com eles.

Revisões e auditorias. As revisões técnicas são uma atividade de controle de qualidade realizada por engenheiros de software para engenheiros de software (Capítulo 20). Seu intuito é o de revelar erros. Auditorias são um tipo de revisão realizada pelo pessoal de SQA com o intuito de assegurar que as diretrizes de qualidade estejam sendo seguidas no trabalho de engenharia de software. Por exemplo, pode ser realizada uma auditoria do processo de revisão para garantir que as revisões estejam sendo feitas de maneira que conduza à maior probabilidade de descoberta de erros.

Testes. Os testes de software (Capítulos 22 a 26) são uma função de controle de qualidade com um objetivo principal: encontrar erros. O papel da SQA é garantir que os testes sejam planejados apropriadamente e conduzidos eficientemente de modo que se tenha a maior probabilidade possível de alcançar seu objetivo primário.

Coleta e análise de erros/defeitos. A única forma de melhorar é medir o nosso desempenho. A SQA reúne e analisa dados de erros e defeitos para melhor compreender como os erros são introduzidos e quais atividades de engenharia de software são as mais adequadas para sua eliminação.

Gerenciamento de mudanças. As mudanças são um dos aspectos mais disruptivos de qualquer projeto de software. Se não forem administra-

Uma discussão aprofundada sobre SQA, incluindo uma ampla variedade de definições, pode ser obtida em http://www.swqual.com/images/FoodforThought_Jan2011.pdf.

das apropriadamente, podem gerar confusão, e confusão quase sempre leva a uma qualidade inadequada. A SQA garante que práticas adequadas de gerenciamento de mudanças (Capítulo 29) tenham sido instituídas.

Educação. Toda organização de software quer melhorar suas práticas de engenharia de software. Um fator fundamental para o aperfeiçoamento é a educação dos engenheiros de software, seus gerentes e outros envolvidos. A organização de SQA assume a liderança no processo de aperfeiçoamento do software (Capítulo 37) e é um proponente fundamental e patrocinador de programas educacionais.

Gerência dos fornecedores. Adquirem-se três categorias de software de fornecedores externos de software – *pacotes comerciais prontos* (por exemplo, Microsoft Office), um *shell personalizado* [Hor03] que fornece um esqueleto estrutural básico, personalizado de acordo com as necessidades do comprador, e *software sob encomenda*, que é projetado e construído de forma personalizada a partir de especificações fornecidas pela empresa-cliente. O papel do grupo de SQA é garantir software de alta qualidade por meio da sugestão de práticas específicas de garantia da qualidade, que o fornecedor deve (sempre que possível) seguir, e incorporar exigências de qualidade como parte de qualquer contrato com um fornecedor externo.

Administração da segurança. Com o aumento dos crimes cibernéticos e novas regulamentações governamentais referentes à privacidade, toda organização de software deve instituir políticas que protejam os dados em todos os níveis, estabelecer proteção por meio de firewalls para as aplicações da Internet (WebApps) e garantir que o software não tenha sido alterado internamente sem autorização. A SQA garante o emprego de processos e tecnologias apropriados para se ter a segurança de software desejada (Capítulo 27).

Proteção. O fato de o software ser quase sempre um componente fundamental de sistemas que envolvem vidas humanas (por exemplo, aplicações na indústria automotiva ou aeronáutica), o impacto de defeitos ocultos pode ser catastrófico. A SQA pode ser responsável por avaliar o impacto de falhas de software e por iniciar as etapas necessárias para redução de riscos.

Gestão de riscos. Embora a análise e a redução de riscos (Capítulo 35) sejam preocupações dos engenheiros de software, o grupo de SQA garante que as atividades de gestão de riscos sejam conduzidas apropriadamente e que planos de contingência relacionados a riscos tenham sido estabelecidos.

Além de cada uma dessas preocupações e atividades, a SQA trabalha para garantir que atividades de suporte ao software (por exemplo, manutenção, suporte online, documentação e manuais) sejam realizadas ou produzidas tendo a qualidade como preocupação dominante.

"Excelência é a capacidade ilimitada de melhorar a qualidade daquilo que se tem a oferecer."

Rick Petin

INFORMAÇÕES

**Recursos para gestão da qualidade**

Dezenas de recursos para gestão da qualidade estão disponíveis na Internet, incluindo associações de profissionais, organizações de padrões e fontes de informação genéricas. Veja aqui os sites que são um bom ponto de partida:

American Society for Quality (ASQ) Software Division
www.asq.org/software

Association for Computer Machinery
www.acm.org

Cyber Security and Information Systems Information Analysis Center (CSIIAC)
<https://sw.thecsiac.com/>

International Organization for Standardization (ISO)
www.iso.ch

ISO SPICE **<http://www.spiceusergroup.org/>**

Malcolm Baldrige National Quality Award
<http://www.nist.gov/baldrige/>

Software Engineering Institute **www.sei.cmu.edu/**

Software Testing and Quality Engineering
www.stickyminds.com

Six Sigma Resources
www.isixsigma.com/ **www.asq.org/sixsigma/**

TickIT International: Quality certification topics
www.tickit.org/international.htm

Total Quality Management (TQM)
<http://www.isixsigma.com/methodology/total-quality-management-tqm/>
<http://asq.org/learn-about-quality/total-quality-management/overview/overview.html>

21.3 Processos da SQA e características do produto

Ao iniciarmos uma discussão sobre garantia da qualidade de software, é importante observar que os procedimentos e abordagens da SQA que funcionam em um ambiente de software podem não funcionar tão bem em outro. Mesmo dentro de uma empresa que adota uma abordagem consistente² de engenharia de software, diferentes produtos de software podem exibir diferentes níveis de qualidade [Par11].

A solução para esse dilema é entender os requisitos de qualidade específicos de um produto de software e então selecionar o processo e as ações e tarefas de SQA específicas que vão ser usadas para atender a esses requisitos. Os padrões CMMI do Software Engineering Institute e ISO 9000 são as metodologias de processo mais comumente usadas. Cada um propõe “uma sintaxe e uma semântica” [Par11] que levará à implementação de práticas de engenharia de software que melhoram a qualidade do produto. Em vez de utilizar uma ou outra metodologia em sua totalidade, uma empresa de software pode “harmonizar” os dois modelos, selecionando elementos das duas metodologias e fazendo-os corresponder aos requisitos de qualidade de um produto.

21.4 Tarefas, metas e métricas da SQA

A garantia da qualidade de software é composta por uma série de tarefas associadas a dois elementos distintos – os engenheiros de software que realizam o trabalho técnico e um grupo de SQA que é responsável pelo planejamento, supervisão, manutenção de registros, análise e relatórios referentes à garantia da qualidade.

² Por exemplo, processos e práticas definidos pelo CMMI (Capítulo 37).

Os engenheiros de software tratam da qualidade (e realizam atividades de controle de qualidade) por meio da aplicação de medidas e métodos técnicos consistentes, conduzindo as revisões técnicas e realizando testes de software bem planejados.

21.4.1 Tarefas da SQA

A prerrogativa do grupo de SQA é ajudar a equipe de software a obter um produto final de alta qualidade. O SEI (Software Engineering Institute) recomenda um conjunto de atividades de SQA que tratam do planejamento, da supervisão, da manutenção de registros, da análise e de relatórios relativos à garantia da qualidade. Essas atividades são realizadas (ou facilitadas) por um grupo de SQA independente que:

Prepara um plano de SQA para um projeto. O plano é desenvolvido como parte do planejamento do projeto e é revisado por todos os envolvidos. As atividades de garantia da qualidade executadas pela equipe de engenharia de software e pelo grupo de SQA são governadas pelo plano. O plano identifica as avaliações, auditorias e revisões a ser realizadas, padrões aplicáveis ao projeto, procedimentos para relatório e acompanhamento de erros, produtos resultantes produzidos pelo grupo de SQA e o feedback que será fornecido à equipe de software.

Participa no desenvolvimento da descrição da gestão de qualidade do projeto. A equipe de software seleciona um processo para o trabalho a ser realizado. O grupo de SQA revisa a descrição de processos para conformidade com a política organizacional, padrões internos de software, padrões impostos externamente (por exemplo, ISO-9001) e outras partes do plano de projeto de software.

Revisa as atividades de engenharia de software para verificar sua conformidade com a gestão de qualidade definida. O grupo de SQA identifica, documenta e acompanha desvios do processo e verifica se as correções foram feitas.

Audita os artefatos de software designados para verificar sua conformidade com aqueles definidos como parte da gestão de qualidade. O grupo de SQA revisa os artefatos selecionados; identifica, documenta e acompanha os desvios, verifica se as correções foram feitas; e, periodicamente, relata os resultados de seu trabalho para o gerente de projeto.

Garante que os desvios no trabalho de software e artefatos sejam documentados e tratados de acordo com um procedimento documentado. Podem ser encontrados desvios no plano de projeto, na descrição do processo, nos padrões aplicáveis ou nos artefatos da engenharia de software.

Registra qualquer não conformidade e relata à alta direção. Itens com problemas (que não atendem às especificações) são acompanhados até que tais problemas sejam resolvidos.

Além dessas atividades, o grupo de SQA coordena o controle e o gerenciamento de mudanças (Capítulo 29) e ajuda a coletar e analisar métricas de software.

Qual é o papel de um grupo de SQA?

"A qualidade jamais é acidental; ela é sempre o resultado de intenção extraordinária, esforço sincero, orientação inteligente e uma hábil execução; representa a escolha inteligente entre muitas alternativas."

William A. Foster



Garantia da Qualidade de Software

Cena: Escritório de Doug Miller no início do projeto de software do CasaSegura.

Atores: Doug Miller (gerente da equipe de engenharia de software do CasaSegura) e outros membros da equipe de engenharia de software do produto.

Conversa:

Doug: Como estão as coisas nas revisões informais?

Jamie: Estamos realizando revisões informais dos elementos críticos do projeto em pares, à medida que codificamos, mas antes dos testes. Está indo mais rápido do que imaginava.

Doug: Isso é bom, mas quero que o grupo de SQA de Bridget Thorton faça auditorias de nossos artefatos para termos certeza de que estamos seguindo nossos processos e atingindo nossas metas de qualidade.

Vinod: Eles já não estão fazendo a maior parte dos testes?

Doug: Sim, estão. Mas QA é mais do que testar. Precisamos ter certeza de que nossos documentos estão evoluindo junto com nosso código e de que não estamos introduzindo erros ao integrarmos novos componentes.

Jamie: Não quero ser avaliado com base no que eles encontrarem.

Doug: Não se preocupe. As auditorias se concentram na adaptação de nossos artefatos aos requisitos e no processo de nossas atividades. Vamos usar os resultados da auditoria apenas para tentar aprimorar nossos processos e nossos produtos de software.

Vinod: Sou obrigado a pensar que isso vai ocupar mais de nosso tempo.

Doug: No final das contas, vamos economizar tempo quando encontrarmos defeitos antecipadamente. Também custa menos corrigir defeitos detectados no início.

Jamie: Então isso parece muito bom.

Doug: Também é importante identificar as atividades onde foram introduzidos defeitos e acrescentar tarefas de revisão para capturá-los no futuro.

Vinod: Isso nos ajudará a determinar se estamos experimentando cuidadosamente com nossas atividades de revisão.

Doug: Acho que as atividades de SQA nos tornarão uma equipe melhor no longo prazo.

21.4.2 Metas, atributos e métricas

As atividades de SQA descritas na seção anterior são realizadas para atingir um conjunto de metas pragmáticas:

- **Qualidade dos requisitos.** A correção, a completude e a consistência do modelo de requisitos terão forte influência sobre a qualidade de todos os produtos seguintes. A SQA deve assegurar-se de que a equipe de software tenha revisto apropriadamente o modelo de requisitos para a obtenção de um alto nível de qualidade.
- **Qualidade do projeto.** Todo elemento do modelo de projeto deve ser avaliado pela equipe de software para garantir que apresente alta qualidade e que o próprio projeto esteja de acordo com os requisitos. A SQA busca atributos do projeto que sejam indicadores de qualidade.
- **Qualidade do código.** O código-fonte e os artefatos relacionados (por exemplo, outras informações descritivas) devem estar em conformidade com os padrões locais de codificação e apresentar características que facilitem a manutenção. A SQA deve isolar os atributos que permitem uma análise razoável da qualidade do código.
- **Eficácia do controle de qualidade.** A equipe de software deve aplicar os recursos limitados de forma a obter a maior probabilidade possível de atingir um resultado de alta qualidade. A SQA analisa a alocação de recursos para revisões e testes para verificar se eles estão ou não sendo alocados da maneira mais efetiva.

Meta	Atributo	Métrica
Qualidade dos requisitos	Ambiguidade	Número de modificadores ambíguos (por exemplo, muitos, grande, amigável)
	Compleitude	Número de TBA, TBD
	Compreensibilidade	Número de seções/subseções
	Volatilidade	Número de mudanças por requisito
		Tempo (por atividade) quando é solicitada a mudança
	Rastreabilidade	Número de requisitos não rastreáveis ao projeto/código
	Clareza do modelo	Número de modelos UML Número de páginas descritivas por modelo
Qualidade do projeto	Integridade da arquitetura	Existência do modelo de arquitetura
	Compleitude dos componentes	Número de componentes mapeados no modelo de arquitetura
		Complexidade do projeto procedural
	Complexidade da interface	Número médio de cliques para chegar a uma função ou conteúdo típico
		Adequação do layout
Qualidade do código	Padrões	Número de padrões usados
	Complexidade	Complexidade ciclomática
	Facilidade de manutenção	Fatores de projeto (Capítulo 8)
	Compreensibilidade	Porcentagem de comentários internos
		Convenções de atribuição de variáveis
	Reutilização	Porcentagem de componentes reutilizados
Eficiência do controle de qualidade	Documentação	Índice de legibilidade
	Alocação de recursos	Porcentagem de horas de pessoal por atividade
	Taxa de completude	Tempo de conclusão real <i>versus</i> previsto
	Eficácia da revisão	Ver métricas de revisão (Capítulo 14)
	Eficácia dos testes	Número de erros encontrados e gravidade
		Esforço exigido para corrigir um erro Origem do erro

FIGURA 21.1 Metas, atributos e métricas para qualidade de software.

Fonte: Adaptado de [Hya96].

A Figura 21.1 (adaptada de [Hya96]) identifica os atributos indicadores da existência de qualidade para cada uma das metas discutidas. Também são mostradas as métricas que podem ser utilizadas para indicar a força relativa de um atributo.

21.5 Abordagens formais da SQA

Nas seções anteriores, dissemos que a qualidade de software é tarefa de todos e que ela pode ser atingida por meio da prática competente de engenharia de

software, bem como da aplicação de revisões técnicas, uma estratégia de testes multicamadas, melhor controle dos artefatos de software e das mudanças nele feitas, e a aplicação dos padrões aceitos de engenharia de software e de metodologias de processo. Além disso, a qualidade pode ser definida em termos de uma ampla gama de atributos de qualidade e medida (indiretamente) usando-se uma variedade de índices e métricas.

Ao longo das últimas três décadas, um segmento pequeno, mas eloquente, da comunidade de engenharia de software sustentava que era necessária uma abordagem mais formal para garantir a qualidade do software. Pode-se dizer que um programa de computador é um objeto matemático. Pode-se definir uma sintaxe e uma semântica rigorosas para todas as linguagens de programação e está disponível uma rigorosa abordagem para a especificação dos requisitos do software (Capítulo 28). Se o modelo de requisitos (especificação) e a linguagem de programação podem ser representados de maneira rigorosa, deve ser possível aplicar uma prova matemática da correção para demonstrar a adequação exata de um programa às suas especificações.

Tentativas de se provar que os programas são corretos não são novas. Dijkstra [Dij76a] e Linger, Mills e Witt [Lin79], entre outros, defenderam provas da correção de programas e associaram-nas ao uso dos conceitos de programação estruturada (Capítulo 14).

21.6 Estatística da garantia da qualidade de software

"20% do código contém 80% dos erros. Encontre-os e corrija-os!"

Lowell Arthur

Quais são as etapas necessárias para realizar estatística de SQA?

"Uma análise estatística, conduzida adequadamente, é uma dissecação delicada de incertezas, uma cirurgia de suposições."

M. J. Moroney

A estatística da garantia da qualidade reflete uma tendência crescente em toda a indústria de software de tornar mais quantitativa a análise da qualidade. Para software, a estatística da garantia da qualidade implica as seguintes etapas:

- 1 Informações sobre erros e defeitos de software são coletadas e classificadas.
- 2 É feita uma tentativa de associar cada erro e defeito à sua causa subjacente (por exemplo, a não adequação às especificações, erros de projeto, violação de padrões, comunicação inadequada com o cliente).
- 3 Usando o princípio de Pareto (80% dos defeitos podem ser associados a 20% de todas as possíveis causas), são isolados os 20% (as *poucas causas vitais*).
- 4 Assim que as poucas causas vitais tiverem sido identificadas, prossegue-se para a correção dos problemas que provocaram os erros e defeitos.

Esse conceito relativamente simples representa um importante passo para a criação de um processo de software adaptável em que mudanças são feitas para melhorar os elementos do processo que introduzem erros.

21.6.1 Um exemplo genérico

Para ilustrar o uso de métodos estatísticos para a engenharia de software, suponhamos que uma organização de engenharia de software reúna informações sobre erros e defeitos por um período de um ano. Alguns desses erros são revelados à medida que o software é desenvolvido. Outros defeitos são

encontrados após o software ter sido liberado para os usuários. Embora sejam encontrados centenas de problemas diferentes, todos podem ser associados a uma (ou mais) das seguintes causas:

- Especificações incompletas ou errôneas (IES, incomplete or erroneous specifications)
- Má interpretação da comunicação do cliente (MCC, misinterpretation of customer communication)
- Desvio intencional das especificações (IDS, intentional deviation from specifications)
- Violação dos padrões de programação (VPS, violation of programming standards)
- Erro na representação de dados (EDR, error in data representation)
- Interface inconsistente de componentes (ICI, inconsistent component interface)
- Erro na lógica do projeto (EDL, error in design logic)
- Testes incompletos ou errôneos (IET, incomplete or erroneous testing)
- Documentação imprecisa ou incompleta (IID, inaccurate or incomplete documentation)
- Erro na tradução do projeto para linguagem de programação (PLT, error in programming language translation of design)
- Interface homem-máquina ambígua ou inconsistente (HCI, ambiguous or inconsistent human/computer interface)
- Outros (MIS, miscellaneous)

Para aplicar a estatística da SQA, é construída a tabela da Figura 21.2. A tabela indica que IES, MCC e EDR são as poucas causas vitais responsáveis por 53% de todos os erros. Deve-se notar, entretanto, que IES, EDR, PLT e EDL seriam escolhidas como as poucas causas vitais se fossem considerados apenas os erros graves. Uma vez determinadas as poucas causas vitais, a organização de engenharia de software pode começar a ação corretiva. Por exemplo, para corrigir a MCC, talvez fosse preciso implementar as técnicas de reunião de

Erro	Total		Grave		Moderado		Secundário	
	No.	%	No.	%	No.	%	No.	%
IES	205	22%	34	27%	68	18%	103	24%
MCC	156	17%	12	9%	68	18%	76	17%
IDS	48	5%	1	1%	24	6%	23	5%
VPS	25	3%	0	0%	15	4%	10	2%
EDR	130	14%	26	20%	68	18%	36	8%
ICI	58	6%	9	7%	18	5%	31	7%
EDL	45	5%	14	11%	12	3%	19	4%
IET	95	10%	12	9%	35	9%	48	11%
IID	36	4%	2	2%	20	5%	14	3%
PLT	60	6%	15	12%	19	5%	26	6%
HCI	28	3%	3	2%	17	4%	8	2%
MIS	56	6%	0	0%	15	4%	41	9%
Totais	942	100%	128	100%	379	100%	435	100%

FIGURA 21.2 Coleta de dados para estatística da SQA.

requisitos (Capítulo 8) para melhorar a qualidade da comunicação do cliente e das especificações. Para melhorar o EDR, pode-se adquirir ferramentas para modelagem de dados e realizar revisões mais rigorosas do projeto de dados.

É importante notar que a ação corretiva se concentra basicamente nas poucas causas vitais. À medida que as poucas causas vitais são corrigidas, novos candidatos vão para o topo da lista.

As técnicas de estatística da garantia da qualidade para software proporcionaram um aperfeiçoamento significativo da qualidade (por exemplo, [Rya11], [Art97]). Em alguns casos, as organizações de software atingiram uma redução de 50% por ano nos defeitos após a aplicação dessas técnicas.

A aplicação de estatística de SQA e o princípio de Pareto podem ser sintetizados em uma única frase: *Invista seu tempo concentrando-se em coisas que realmente importam, mas, primeiramente, certifique-se de ter entendido aquilo que realmente importa!*

21.6.2 Seis Sigma para engenharia de software

Seis Sigma é a estratégia para a estatística da garantia da qualidade mais utilizada na indústria atual. Originalmente popularizada pela Motorola, na década de 1980, a estratégia Seis Sigma “é uma metodologia rigorosa e disciplinada que usa análise estatística e de dados para medir e melhorar o desempenho operacional de uma empresa por meio da identificação e da eliminação de defeitos em processos de fabricação e relacionados a serviços” [ISI08]. O termo *Seis Sigma* é derivado de seis desvios-padrão – 3,4 ocorrências (defeitos) por milhão – implicando um padrão de qualidade extremamente elevado. A metodologia Seis Sigma define três etapas essenciais:

Quais são as etapas essenciais da metodologia Seis Sigma?

- *Definir* os requisitos do cliente e os artefatos a serem entregues, bem como as metas de projeto, através de métodos bem definidos da comunicação com o cliente.
- *Medir* o processo existente e seu resultado para determinar o desempenho da qualidade atual (coletar métricas de defeitos).
- *Analisar* as métricas de defeitos e determinar as poucas causas vitais.

Se já existir uma gestão de qualidade e for necessário um aperfeiçoamento, a estratégia Seis Sigma sugere duas etapas adicionais:

- *Melhorar* o processo por meio da eliminação das causas básicas dos defeitos.
- *Controlar* o processo para garantir que trabalhos futuros não reintroduzam as causas dos defeitos.

Essas etapas essenciais e adicionais são, algumas vezes, conhecidas como método DMAIC (definir, medir, analisar, aperfeiçoar/improvel e controlar).

Se uma organização estiver desenvolvendo uma gestão de qualidade (e não aperfeiçoando uma já existente), nas etapas essenciais são incluídas:

- *Projetar* o processo para: (1) evitar as causas básicas dos defeitos e (2) atender aos requisitos do cliente.
- *Verificar* se o modelo de processos vai, de fato, evitar defeitos e atender aos requisitos do cliente.

Essa variação é algumas vezes denominada método DMADV (definir, medir, analisar, projetar [design] e verificar).

É melhor deixarmos uma discussão completa sobre Seis Sigma para fontes dedicadas ao assunto. Caso tenha maior interesse, consulte [ISI08], [Pyz03] e [Sne03].

21.7 Confiabilidade de software

Não há nenhuma dúvida de que a confiabilidade de um programa de computador é um elemento importante de sua qualidade global. Se um programa falhar frequentemente e repetidas vezes, pouco importa se outros fatores da qualidade de software sejam aceitáveis.

A confiabilidade do software, diferentemente de outros fatores de qualidade, pode ser medida diretamente e estimada usando-se dados históricos e de desenvolvimento. A *confiabilidade de software* é definida em termos estatísticos como “a probabilidade de operação sem falhas de um programa de computador em dado ambiente por determinado tempo” [Mus87]. Para ilustrarmos esse conceito, estima-se que o programa X tenha uma confiabilidade de 0,999 depois de decorridas oito horas de processamento. Em outras palavras, se o programa X tiver de ser executado 1.000 vezes e precisar de um total de oito horas de tempo de processamento (tempo de execução), é provável que ele opere corretamente (sem falhas) 999 vezes.

Toda vez que discutimos confiabilidade de software, surge uma questão fundamental: Qual o significado do termo *falha*? No contexto de qualquer discussão sobre qualidade de software e confiabilidade, falha é a falta de conformidade com os requisitos de software. Mesmo dentro dessa definição existem variações. As falhas podem ser apenas problemáticas ou catastróficas. Determinada falha pode ser corrigida em segundos, enquanto outras necessitarão de semanas ou até mesmo meses para serem corrigidas. Para complicar ainda mais o problema, a correção de uma falha pode, na realidade, resultar na introdução de outros erros que resultarão em outras falhas.

“O preço inevitável da confiabilidade é a simplicidade.”

C. A. R. Hoare

21.7.1 Medidas de confiabilidade e disponibilidade

Os primeiros trabalhos sobre confiabilidade de software tentaram extrapolar a matemática da teoria da confiabilidade de hardware para a previsão da confiabilidade de software. A maioria dos modelos de confiabilidade relacionada com hardware tem como base falhas devido a desgaste e não as falhas devido a defeitos de projeto. Em hardware, as falhas devido a desgaste físico (por exemplo, os efeitos decorrentes de temperatura, corrosão, choque) são mais prováveis do que uma falha relacionada ao projeto. Infelizmente, a recíproca é verdadeira para software. Na realidade, todas as falhas de software podem ser associadas a problemas de projeto ou de implementação; o desgaste (ver Capítulo 1) não entra em questão.

Tem havido um debate contínuo sobre a relação entre conceitos-chave na confiabilidade de hardware e sua aplicabilidade ao software. Embora ainda seja preciso estabelecer uma associação irrefutável, consideram-se alguns conceitos simples que se aplicam aos elementos de ambos os sistemas.

Os problemas de confiabilidade de software quase sempre podem ser associados a defeitos de projeto ou de implementação.

É importante notar que o MTBF e medidas relacionadas se baseiam em tempo de máquina e não em tempo de relógio tradicional.

Se considerarmos um sistema computacional, uma medida de confiabilidade simples é o *tempo médio entre falhas* (MTBF, *mean-time-between-failure*):

$$\text{MTBF} = \text{MTTF} + \text{MTTR}$$

em que os acrônimos MTTF e MTTR são, respectivamente, *tempo médio para falhar* (*mean-time-to-failure*) e *tempo médio para reparar*³ (*mean-time-to-repair*).

Muitos pesquisadores defendem que o MTBF é uma medida mais útil do que quaisquer outras métricas de software relacionadas com a garantia da qualidade discutidas no Capítulo 30. De maneira simples, um usuário se preocupa com falhas e não com o número total de defeitos. Como cada defeito contido em um programa não tem a mesma taxa de falhas, o número total de defeitos fornece pouca indicação da confiabilidade de um sistema. Por exemplo, considere um programa que esteve em operação por 3.000 horas de processamento sem nenhuma falha. Vários defeitos neste programa podem não ser detectados por dezenas de milhares de horas antes de serem descobertos. Com esses erros obscuros, o MTBF poderia ser de 30.000 ou até mesmo 60.000 horas de processador. Outros defeitos, embora ainda não descobertos, poderiam ter uma taxa de falhas de 4.000 ou 5.000 horas. Mesmo se cada um dos erros da primeira categoria (aqueles com MTBF longo) fosse eliminado, o impacto sobre a confiabilidade do software seria desprezível.

Entretanto, o MTBF pode ser problemático por duas razões: (1) ele projeta um período de tempo entre falhas, mas não fornece uma projeção da taxa de falhas e (2) o MTBF pode ser mal interpretado como sendo tempo de vida médio, muito embora *não* seja esse o significado.

Uma medida de confiabilidade alternativa é *falha ao longo do tempo* (FIT, *failures-in-time*) – uma medida estatística de quantas falhas um componente terá ao longo de um bilhão de horas de operação. Consequentemente, 1 FIT equivale a uma falha a cada bilhão de horas de operação.

Além de uma medida da confiabilidade, deve-se também desenvolver uma medida de disponibilidade. *Disponibilidade de software* é a probabilidade de que um programa esteja operando de acordo com os requisitos em dado instante e é definida da seguinte forma:

$$\text{Disponibilidade} = \frac{\text{MTTF}}{(\text{MTTF} + \text{MTTR})} \times 100\%$$

A medida de confiabilidade MTBF é igualmente sensível ao MTTF e ao MTTR. A medida de disponibilidade é ligeiramente mais sensível ao MTTR, uma medida indireta da facilidade de manutenção do software. Para uma ampla discussão sobre medidas de confiabilidade de software, consulte [Laz11].

Alguns aspectos da disponibilidade (não discutidos aqui) não têm nada a ver com falha. Por exemplo, programar a parada do sistema (para funções de suporte) deixa o software indisponível.

“A segurança das pessoas deve ser a maior das leis.”

Cícero

21.7.2 Segurança do software

Segurança do software é uma atividade de garantia da qualidade de software que se concentra na identificação e na avaliação de possíveis problemas que podem afetar negativamente um software e provocar falha em todo o sistema. Se os problemas podem ser identificados precocemente na gestão de qualida-

³ Embora a depuração (e correções relacionadas) possa ser necessária como consequência de uma falha, em muitos casos o software funcionará adequadamente depois de um reinício, sem nenhuma outra mudança.

de, as características para eliminar ou controlar esses problemas podem ser especificadas no projeto do software.

Um processo de modelagem e análise é efetuado como parte da segurança do software. Inicialmente, os problemas são identificados e classificados pelo caráter crítico e pelo risco. Por exemplo, problemas associados a um controle computadorizado de um automóvel podem: (1) provocar uma aceleração descontrolada que não pode ser interrompida, (2) não responder ao acionamento do pedal do freio (por meio de uma desativação), (3) não operar quando a chave é ativada e (4) perder ou ganhar velocidade lentamente. Uma vez identificados esses perigos em nível de sistema, técnicas de análise são utilizadas para atribuir gravidade e probabilidade de ocorrência.⁴ Para ser eficaz, o software deve ser analisado no contexto de todo o sistema. Por exemplo, um erro sutil cometido pelo usuário na entrada de dados (as pessoas são componentes do sistema) talvez seja ampliado por uma falha de software e produza dados de controle que posicione um dispositivo mecânico de forma inadequada. Se e somente se um conjunto de condições ambientais externas for atendido, a posição imprópria do dispositivo mecânico provocará uma falha desastrosa. Análises técnicas [Eri05], como a análise da árvore de falhas, lógica em tempo real e modelos em redes de Petri, podem ser usadas para prever a cadeia de eventos que podem causar problemas e a probabilidade de cada um dos eventos ocorrer para criar a cadeia.

Uma vez identificados e analisados os problemas, os requisitos relacionados à segurança podem ser especificados para o software. Ou seja, a especificação pode conter uma lista de eventos indesejáveis e as respostas desejadas do sistema para esses eventos. O papel do software em administrar eventos indesejáveis é então indicado.

Embora a confiabilidade e a segurança do software estejam intimamente relacionadas, é importante entender a diferença sutil entre elas. A confiabilidade do software usa análise estatística para determinar a probabilidade de ocorrência de uma falha de software. Entretanto, a ocorrência de uma falha não resulta, necessariamente, em um problema ou contratempo. A segurança do software examina as maneiras pelas quais as falhas resultam em condições que podem levar a um contratempo. Ou seja, as falhas não são consideradas isoladamente, mas sim avaliadas no contexto de todo o sistema computacional e seu ambiente.

Uma discussão completa sobre segurança de software vai além dos objetivos deste livro. Caso tenha maior interesse no tema segurança de software e em questões relacionadas, consulte [Fir12], [Har12], [Smi05] e [Lev95].

"Não consigo imaginar nenhuma condição que faria este navio afundar. A construção naval moderna ultrapassou essa condição."

E. I. Smith, capitão do Titanic

Uma proveitosa coleção de artigos sobre segurança de software pode ser encontrada em www.safewareeng.com/.

21.8 Os padrões de qualidade ISO 9000⁵

Um *sistema de garantia da qualidade* pode ser definido como a estrutura organizacional com responsabilidades, procedimentos, processos e recursos para implementação da gestão da qualidade [ANS87]. Os sistemas de garan-

⁴ Essa abordagem é similar aos métodos de análise de riscos descritos no Capítulo 35. A diferença fundamental é a ênfase em problemas de tecnologia, em vez de tópicos relacionados ao projeto.

⁵ Esta seção, escrita por Michael Stovsky, foi adaptada de *Fundamentals of ISO 9000*, um livro desenvolvido para *Essential Software Engineering*, um programa de estudos em vídeo desenvolvido pela R. S. Pressman & Associates, Inc. Reimpresso com permissão.

tia da qualidade são criados para ajudar as organizações a garantir que seus produtos e serviços satisfaçam às expectativas do cliente por meio do atendimento às suas especificações. Tais sistemas cobrem uma grande variedade de atividades, englobando todo o ciclo de vida de um produto, incluindo planejamento, controle, medições, testes e geração de relatórios e melhorando os níveis de qualidade ao longo de todo o processo de desenvolvimento e fabricação. A ISO 9000 descreve elementos de garantia da qualidade em termos gerais que podem ser aplicados a qualquer empresa, independentemente do tipo de produtos ou serviços oferecidos.

Para obter a certificação em um dos programas de garantia da qualidade contidos na ISO 9000, as operações e o sistema de qualidade de uma empresa são examinados por auditores independentes para verificação de sua conformidade ao padrão e operação efetiva. Após aprovação, um organismo representado pelos auditores emite um certificado para a empresa. Auditorias de inspeção semestrais garantem conformidade contínua ao padrão.

Os requisitos delineados pelos tópicos da ISO 9001:2008 são: responsabilidade administrativa, um sistema de qualidade, revisão do contrato, controle de projeto, controle de dados e documentos, identificação e rastreabilidade de produtos, controle de processos, inspeções e testes, ações preventivas e corretivas, registros de controle de qualidade, auditorias de qualidade internas, treinamento, manutenção e técnicas estatísticas. Para que uma organização de software seja certificada com a ISO 9001:2008, ela tem de estabelecer políticas e procedimentos para atender a cada um dos requisitos que acabamos de citar (e outros) e, depois, ser capaz de demonstrar que tais políticas e procedimentos estão sendo seguidos. Caso deseje maiores informações sobre a ISO 9001:2008, consulte [Coc11], [Hoy09] ou [Cia09].

Uma longa lista de links para recursos relacionados à ISO 9000/9001 pode ser encontrada em www.tantara.ab.ca/info.htm.

INFORMAÇÕES



O padrão ISO 9001:2008

A descrição a seguir define os elementos básicos do padrão ISO 9001:2000. Informações completas sobre o padrão podem ser obtidas da International Organization for Standardization (www.iso.ch) e de outras fontes na Internet (por exemplo, www.praxiom.com).

Estabelecer os elementos de um sistema de gestão da qualidade.

- Desenvolver, implementar e aperfeiçoar o sistema.
- Definir uma política que enfatize a importância do sistema.

Documentar o sistema de qualidade.

- Descrever o processo.
- Produzir um manual operacional.
- Desenvolver métodos para controlar (atualizar) documentos.
- Estabelecer métodos para manutenção de registros.

Dar suporte ao controle e à garantia da qualidade.

- Promover a importância da qualidade entre todos os envolvidos.
- Concentrar-se na satisfação do cliente.

Definir um plano de qualidade que atenda aos objetivos, às responsabilidades e à autoridade.

Definir mecanismos de comunicação entre os envolvidos.

Estabelecer mecanismos de revisão para um sistema de gestão da qualidade.

Identificar métodos de revisão e mecanismos de feedback.

Definir procedimentos de acompanhamento.

Identificar recursos de qualidade, incluindo elementos de pessoal, treinamento e infraestrutura.

Estabelecer mecanismos de controle.

Para planejamento.

Para requisitos do cliente.

Para atividades técnicas (por exemplo, análise, projeto, testes).

Para monitoramento e gerenciamento de projeto.

Definir métodos de reparo.

Avaliar dados e métricas de qualidade.

Definir a abordagem para processos e aperfeiçoamento contínuos da qualidade.

21.9 O plano de SQA

O *plano de SQA* fornece um roteiro para instituir a garantia da qualidade de software. Desenvolvido pelo grupo de SQA (ou pela equipe de software, caso não exista um grupo de SQA), o plano serve como um modelo para atividades de SQA instituídas para cada projeto de software.

Foi publicado pela IEEE [IEE93] um padrão para planos de SQA. O padrão recomenda uma estrutura que identifique: (1) o propósito e o escopo do plano, (2) uma descrição de todos os artefatos de engenharia de software (por exemplo, modelos, documentos, código-fonte) que caem na alçada da SQA, (3) todos os padrões e práticas aplicados durante a gestão de qualidade, (4) as ações e tarefas da SQA (incluindo revisões e auditorias) e sua aplicação na gestão de qualidade, (5) as ferramentas e os métodos que dão suporte às ações e tarefas da SQA, (6) procedimentos para gestão de configuração do software (Capítulo 29), (7) métodos para montagem, salvaguarda e manutenção de todos os registros relativos à SQA e (8) papéis e responsabilidades dentro da organização, relacionados à qualidade do produto.

FERRAMENTAS DO SOFTWARE



Gestão da qualidade de software

Objetivo: o objetivo das ferramentas de SQA é ajudar uma equipe de projeto a avaliar e aperfeiçoar a qualidade do artefato de software.

Mecanismos: a mecânica das ferramentas varia. Em geral, o intuito é avaliar a qualidade de um artefato específico. Observação: normalmente são incluídas várias ferramentas para teste de software (consulte os Capítulos 22 a 26) dentro da categoria de ferramentas para SQA.

Ferramentas representativas:⁶

QA Complete, desenvolvida pela SmartBear (<http://smartbear.com/products/qa-tools/test-management>), o gerenciamento de QA garante completa cobertura de

teste em cada estágio do processo de desenvolvimento de software.

QPR Suite, desenvolvida pela QPR Software (<http://www.qpr.com>), oferece suporte para Seis Sigma e outras abordagens de gestão de qualidade.

Quality Tools and Templates, desenvolvida pela iSixSigma (<http://www.isixsigma.com/tools-templates/>), descreve uma ampla variedade de ferramentas e métodos úteis para gestão da qualidade.

NASA Quality Resources, desenvolvida pelo Goddard Space Flight Center (<http://www.hq.nasa.gov/office/codesq/software/ComplexElectronics/checklists.htm>), fornece formulários proveitosos, modelos, checklists e ferramentas para SQA.

21.10 Resumo

A garantia da qualidade de software é uma atividade universal da engenharia de software que é aplicada a cada etapa da gestão de qualidade. A SQA abrange procedimentos para a aplicação efetiva de métodos e ferramentas, a supervisão de atividades de controle de qualidade, como revisões técnicas e testes de software, procedimentos para o gerenciamento de mudanças e pro-

⁶ A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

cedimentos para garantir a conformidade a padrões, bem como mecanismos para medição e geração de relatórios.

Para realizar a garantia da qualidade de software de forma apropriada, devem ser reunidos, avaliados e disseminados os dados sobre o processo de engenharia de software. A estatística da SQA ajuda a melhorar a qualidade do produto e da própria gestão de qualidade. Os modelos de confiabilidade de software estendem as medidas obtidas, possibilitando que dados de defeitos coletados sejam extrapolados para projeção de taxas de falhas e previsões de confiabilidade.

Em suma, devemos observar as palavras de Dunn e Ullman [Dun82]: “A garantia da qualidade de software é o mapeamento dos preceitos da gestão e das disciplinas de projeto da garantia da qualidade para a área gerencial e tecnológica da engenharia de software”. A capacidade de garantir a qualidade é a medida de uma engenharia disciplinada e madura. Quando esse mapeamento é realizado com sucesso, o resultado é uma engenharia de software com maturidade.

Problemas e pontos a ponderar

21.1 Algumas pessoas dizem que “o controle das variações é o cerne do controle de qualidade”. Como todo programa criado é diferente de qualquer outro programa, quais são as variações que buscamos e como controlá-las?

21.2 É possível avaliar a qualidade do software se o cliente ficar alterando continuamente aquilo que o software supostamente deveria fazer?

21.3 Qualidade e confiabilidade são conceitos relacionados, mas, fundamentalmente, são diferentes em uma série de aspectos. Discuta as diferenças.

21.4 Um programa pode ser correto e ainda assim não ser confiável? Explique.

21.5 Um programa pode ser correto e ainda assim não apresentar boa qualidade? Explique.

21.6 Por que normalmente existe tensão entre um grupo de engenharia de software e um grupo de garantia da qualidade de software independente? Isso é salutar?

21.7 Você foi incumbido da responsabilidade de melhorar a qualidade do software na organização. Qual é a primeira coisa a ser feita? E a seguinte?

21.8 Além da contagem de erros e defeitos, existem outras características contáveis de software que impliquem a qualidade? Quais são? Elas podem ser medidas diretamente?

21.9 O conceito de MTBF para software está sujeito a críticas. Justifique.

21.10 Considere dois sistemas críticos de proteção que são controlados por computador. Enumere pelo menos três perigos de cada um deles que podem ser associados diretamente a falhas de software.

21.11 Adquira uma cópia da ISO 9001:2000 e da ISO 9000-3. Prepare uma apresentação que discuta três requisitos da ISO 9001 e como eles se aplicam no contexto de software.

Leituras e fontes de informação complementares

Livros como os de Chemuturi (*Mastering Software Quality Assurance*, J. Ross Publishing, 2010), Hoyle (*Quality Management Essentials*, Butterworth-Heinemann, 2007),

Tian (*Software Quality Engineering*, Wiley-IEEE Computer Society Press, 2005), El Emam (*The ROI from Software Quality*, Auerbach, 2005), Horch (*Practical Guide to Software Quality Management*, Artech House, 2003) e Nance e Arthur (*Managing Software Quality*, Springer, 2002) são excelentes apresentações em termos gerenciais dos benefícios dos programas formais para garantia da qualidade de software. Livros como os de Deming [Dem86], Defoe e Juran (*Juran's Quality Handbook*, 6ª ed., McGraw-Hill, 2010), Juran (*Juran on Quality by Design*, Free Press, 1992) e Crosby ([Cro79] e *Quality Is Still Free*, McGraw-Hill, 1995) não se concentram em software, mas são leituras obrigatórias para gerentes seniores responsáveis pelo desenvolvimento de software. Gluckman e Roome (*Everyday Heroes of Quality Movement*, Dorset House, 1993) humaniza as questões da qualidade, contando a história dos participantes de um processo de qualidade. Kan (*Metrics and Models in Software Quality Engineering*, Addison-Wesley, 1995) apresenta uma visão quantitativa da qualidade de software.

As obras de Evans (*Quality & Performance Excellence*, South-Western College Publishing, 2007; e *Total Quality: Management, Organization and Strategy*, 4ª ed., South-Western College Publishing, 2004), Bru (*Six Sigma for Managers*, McGraw-Hill, 2005) e Dobb (*ISO 9001:2000 Quality Registration Step-by-Step*, 3ª ed., Butterworth-Heinemann, 2004) são representativos dos muitos livros escritos sobre TQM, Six Sigma e ISO 9001:2000, respectivamente.

O'Connor e Kleyner (*Practical Reliability Engineering*, Wiley, 2012), Naik e Tripathy (*Software Testing and Quality Assurance: Theory and Practice*, Wiley-Spektrum, 2008), Pham (*System Software Reliability*, Springer, 2006), Musa (*Software Reliability Engineering: More Reliable Software, Faster Development and Testing*, 2ª ed., McGraw-Hill, 2004) e Peled (*Software Reliability Methods*, Springer, 2001) escreveram guias práticos que descrevem métodos para medir e analisar a confiabilidade do software.

Vincoli (*Basic Guide to System Safety*, Wiley, 2006), Dhillon (*Computer System Reliability, Safety and Usability*, CRC Press, 2013; e *Engineering Safety*, World Scientific Publishing Co., 2003), Hermann (*Software Safety and Reliability*, Wiley-IEEE Computer Society Press, 2010), Verma, Ajit e Karanki (*Reliability and Safety Engineering*, Springer, 2010), Storey (*Safety-Critical Computer Systems*, Addison-Wesley, 1996) e Leveson [Lev95] são as discussões mais abrangentes publicadas até hoje sobre proteção de software e de sistemas. Além desses, van der Meulen (*Definitions for Hardware and Software Safety Engineers*, Springer-Verlag, 2000) oferece um compêndio completo de importantes conceitos e termos de confiabilidade e proteção; Gardiner (*Testing Safety-Related Software*, Springer-Verlag, 1999) é um guia especializado para testar sistemas de proteção críticos; Friedman e Voas (*Software Assessment: Reliability Safety and Testability*, Wiley, 1995) fornecem modelos úteis para avaliar a confiabilidade e a proteção. Ericson (*Hazard Analysis Primer*, CreateSpace Independent Publishing Platform, 2012; e *Hazard Analysis Techniques for System Safety*, Wiley, 2005) tratam da cada vez mais importante área da análise de riscos.

Uma ampla gama de fontes de informação sobre garantia da qualidade de software encontra-se à disposição na Internet. Uma lista atualizada das referências da Web (em inglês) pode ser encontrada no site: www.mhhe.com/pressman.

22 Estratégias e teste de software

Conceitos-chave

conclusão.....	472
depuração.....	488
estratégias de teste para aplicativos móveis.....	483
estratégias de teste para WebApps.....	482
grupo independente de teste.....	469
integração ascendente.....	477
integração descendente.....	476
pseudocontroladores.....	475
pseudocontrolados.....	475
revisão de configuração.....	484
software orientado a objetos.....	481
teste alfa.....	485
teste baseado em sequências de execução.....	481
teste beta.....	485
teste de classe.....	481
teste de conjunto.....	482
teste de desempenho.....	487
teste de disponibilização.....	487
teste de esforço (stress).....	487
teste de integração.....	475
teste de recuperação.....	486
teste de regressão.....	478
teste de segurança.....	486
teste de sistema.....	486
teste de unidade.....	473
teste de validação.....	483
teste fumaça.....	479
validação.....	468
verificação.....	467

A estratégia de teste de software fornece um roteiro que descreve os passos a serem executados como parte do teste, define quando esses passos são planejados e então executados e quanto trabalho, tempo e recursos serão necessários. Portanto, qualquer estratégia de teste deve incorporar planejamento dos testes, projeto de casos de teste, execução dos testes e coleta e avaliação dos dados resultantes.

Uma estratégia de teste de software deve ser flexível o bastante para promover uma estratégia de teste personalizada. Ao mesmo tempo, deve ser rígida o bastante para estimular um planejamento razoável e o acompanhamento à medida que o projeto progride. Shooman [Sho83] discute essas questões:

De muitas formas, o teste é um processo individualista, e o número de tipos diferentes de testes varia tanto quanto as diferentes estratégias de desenvolvimento. Por muitos anos, nossas únicas defesas contra os erros de programação eram um projeto cuidadoso e a inteligência do programador. Estamos agora em uma era na qual as modernas técnicas de projeto [e revisões técnicas] nos ajudam a reduzir a quantidade de erros iniciais inerentes ao código. Da mesma forma, diferentes métodos de teste estão começando a se agrupar em várias abordagens e filosofias distintas.

Essas “abordagens e filosofias” são chamadas de *estratégia* – o assunto que será apresentado neste capítulo. Nos Capítulos 23 a 26, apresentamos os métodos e técnicas de teste que implementam a estratégia.

22.1 Uma abordagem estratégica do teste de software

Teste é um conjunto de atividades que podem ser planejadas com antecedência e executadas sistematicamente. Por essa razão, deverá ser definido, para o processo de software, um modelo para o teste – um conjunto de etapas no

PANORAMA

O que é? O software é testado para revelar erros cometidos inadvertidamente quando foi projetado e construído. Mas como realizar os testes? Devemos estabelecer um plano formal para nossos testes? Devemos testar o programa como um todo ou executar testes somente em uma parte dele? Devemos refazer os testes quando acres-

centamos novos componentes ao sistema? Quando devemos envolver o cliente? Essas e muitas outras questões são respondidas quando desenvolvemos uma estratégia de teste de software.

Quem realiza? Uma estratégia para teste de software é desenvolvida pelo gerente de projeto, pelos engenheiros de software e pelos especialistas em testes.

Por que é importante? O teste muitas vezes exige mais trabalho de projeto do que qualquer outra ação da engenharia de software. Se for feito casualmente, perde-se tempo, fazem-se esforços desnecessários, e, ainda pior, erros passam sem ser detectados. Portanto, é sensato estabelecer uma estratégia sistemática para teste de software.

Quais são as etapas envolvidas? O teste começa pelo “pequeno” e passa para o “grande”. Ou seja, os testes iniciais focam um único componente ou um pequeno grupo de componentes relacionados e descobrem erros nos dados e na lógica de processamento que foram encapsulados pelo(s) componente(s). Depois de testados, os componentes devem ser integrados até que o sistema completo esteja pronto. Nesse ponto, são executados muitos testes de ordem superior para descobrir erros ao atender aos requisitos do cliente.

À medida que os erros forem descobertos, devem ser diagnosticados e corrigidos usando-se um processo chamado depuração.

Qual é o artefato? A *especificação do teste* documenta a abordagem da equipe de software para o teste, definindo um plano que descreve uma estratégia global e um procedimento e designando etapas específicas de teste e os tipos de testes que serão feitos.

Como garantir que o trabalho foi realizado corretamente? Revisando a *especificação do teste* antes do teste, pode-se avaliar a integralidade dos casos de teste e das tarefas de teste. Um plano e um procedimento de teste eficazes levarão a uma construção ordenada do software e à descoberta de erros em cada estágio do processo de construção.

qual podem ser empregadas técnicas específicas de projeto de caso de teste e métodos de teste.

Muitas estratégias de teste de software já foram propostas na literatura. Todas elas fornecem um modelo para o teste e todas têm as seguintes características genéricas:

- Para executar um teste eficaz, faça revisões técnicas eficazes (Capítulo 20). Fazendo isso, muitos erros serão eliminados antes do começo do teste.
- O teste começa no nível de componente e progride em direção à integração do sistema computacional como um todo.
- Diferentes técnicas de teste são apropriadas para diferentes abordagens de engenharia de software e em diferentes pontos no tempo.
- O teste é realizado pelo desenvolvedor do software e (para grandes projetos) por um grupo de teste independente.
- O teste e a depuração são atividades diferentes, mas a depuração deve ser associada a alguma estratégia de teste.

Uma estratégia de teste de software deve acomodar testes de baixo nível, necessários para verificar se um pequeno segmento de código fonte foi implementado corretamente, bem como testes de alto nível, que validam as funções principais do sistema de acordo com os requisitos do cliente. Uma estratégia deve fornecer diretrizes para o profissional e uma série de metas para o gerente. Como os passos da estratégia de teste ocorrem no instante em que as pressões pelo prazo começam a aumentar, deve ser possível medir o progresso no desenvolvimento, e os problemas devem ser revelados o mais cedo possível.

Recursos úteis para o teste de software podem ser encontrados em www.mtsu.edu/~storm/.

22.1.1 Verificação e validação

O teste de software é um elemento de um tema mais amplo, muitas vezes conhecido como verificação e validação (V&V). *Verificação* refere-se ao conjunto de tarefas que garantem que o software implementa corretamente uma fun-

“O teste é uma parte inevitável de qualquer trabalho responsável para o desenvolvimento de um sistema de software.”

William Howden

Não seja tolo e não considere o teste como uma “rede de segurança” que detectará todos os erros ocorridos devido a práticas deficientes de engenharia de software. Isso não acontecerá. Enfatize a qualidade e a detecção de erro em todo o processo de software.

“Otimismo é a doença ocupacional da programação; o teste é seu tratamento.”

Kent Beck

ção específica. *Validação* refere-se a um conjunto de tarefas que asseguram que o software foi criado e pode ser rastreado segundo os requisitos do cliente. Boehm [Boe81] define de outra maneira:

Verificação: “Estamos criando o produto corretamente?”

Validação: “Estamos criando o produto certo?”

A definição de V&V abrange muitas atividades de garantia da qualidade do software (Capítulo 21).¹

A verificação e a validação incluem uma ampla gama de atividades de SQA (*software quality assurance* – garantia da qualidade de software): revisões técnicas, auditorias de qualidade e configuração, monitoramento de desempenho, simulação, estudo de viabilidade, revisão de documentação, revisão de base de dados, análise de algoritmo, teste de desenvolvimento, teste de usabilidade, teste de qualificação, teste de aceitação e teste de instalação. Embora a aplicação de teste tenha um papel extremamente importante em V&V, muitas outras atividades também são necessárias.

O teste proporciona o último elemento a partir do qual a qualidade pode ser estimada e, mais pragmaticamente, os erros podem ser descobertos. Mas o teste não deve ser visto como uma rede de segurança. Como se costuma dizer, “Você não pode testar qualidade. Se a qualidade não está lá antes de um teste, ela não estará lá quando o teste terminar”. A qualidade é incorporada ao software por meio do processo de engenharia de software. A aplicação correta de métodos e ferramentas, de revisões técnicas eficazes e de um sólido gerenciamento e avaliação conduzem todos à qualidade que é confirmada durante o teste.

Miller [Mil77] relaciona teste de software com garantia da qualidade dizendo que “a motivação que está por trás do teste de programas é a confirmação da qualidade do software com métodos que podem ser econômica e efetivamente aplicados a todos os sistemas, de grande e pequena escala”.

22.1.2 Organizando o teste de software

Para todo projeto de software, há um conflito de interesses inerente que ocorre logo que o teste começa. As pessoas que criaram o software são agora convocadas para testá-lo. Isso parece essencialmente inofensivo; afinal, quem conhece melhor o programa do que os seus próprios desenvolvedores? Infelizmente, esses mesmos desenvolvedores têm interesse em demonstrar que o programa é isento de erros e funciona de acordo com os requisitos do cliente – e também que será concluído dentro do prazo e do orçamento previstos. Cada um desses interesses vai contra o teste completo.

Do ponto de vista psicológico, a análise e o projeto de software (juntamente com a sua codificação) são tarefas construtivas. O engenheiro de software

¹ Deve-se observar que há uma forte divergência de opinião sobre quais tipos de testes constituem “validação”. Algumas pessoas acreditam que *todo* teste é verificação e que a validação é realizada quando os requisitos são examinados e aprovados – e, mais tarde, pelo usuário, com o sistema já em operação. Outras pessoas consideram o teste de unidade e de integração (Seções 22.3.1 e 22.3.2) como verificação e o teste de ordem superior (Seções 22.6 e 22.7) como validação.

analisa, modela e então cria um programa de computador e sua documentação. Como qualquer outro construtor, o engenheiro de software tem orgulho do edifício que construiu e encara com desconfiança qualquer um que tente estragar sua obra. Quando o teste começa, há uma tentativa sutil, embora definida, de “quebrar” aquela coisa que o engenheiro de software construiu. Do ponto de vista do construtor, o teste pode ser considerado como (psicologicamente) destrutivo. Assim, o construtor vai, calmamente, projetando e executando testes que demonstram que o programa funciona, em vez de descobrir os erros. Infelizmente, no entanto, os erros estão presentes. E, se o engenheiro de software não os encontrar, o cliente encontrará!

Frequentemente, há muitas noções incorretas que podem ser inferidas a partir da discussão apresentada: (1) que o desenvolvedor de software não deve fazer nenhum teste, (2) que o software deve ser “atirado aos leões”, ou seja, entregue a estranhos que realizarão testes implacáveis, (3) que os testadores se envolvem no projeto somente no início das etapas do teste. Todas essas afirmativas são incorretas.

O desenvolvedor do software é sempre responsável pelo teste das unidades individuais (componentes) do programa, garantindo que cada uma execute a função ou apresente o comportamento para o qual foi projetada. Em muitos casos, o desenvolvedor também faz o teste de integração – uma etapa de teste que leva à construção (e teste) da arquitetura completa do software. Somente depois que a arquitetura do software está concluída é que o grupo de teste independente se envolve.

O papel de um *grupo independente de teste* (ITG, *independent test group*) é remover problemas inerentes associados ao fato de deixar o criador testar aquilo que ele mesmo criou. O teste independente remove o conflito de interesses que, de outra forma, poderia estar presente. Afinal, o pessoal de ITG é pago para encontrar erros.

No entanto, você não entrega simplesmente o programa para o pessoal do ITG e vai embora. O desenvolvedor e o pessoal do ITG trabalham juntos durante todo o projeto de software para garantir que testes completos sejam realizados. Enquanto o teste está sendo realizado, o desenvolvedor deve estar disponível para corrigir os erros encontrados.

O ITG faz parte da equipe de desenvolvimento de software, pois se envolve durante a análise e o projeto e permanece envolvido (planejando e especificando procedimentos de teste) durante o projeto inteiro. No entanto, em muitos casos o ITG se reporta à organização de garantia da qualidade do software, adquirindo, assim, um grau de independência que poderia não ser possível se fizesse parte da equipe de engenharia de software.

Um grupo de teste independente não tem o “conflito de interesses” que os construtores do software podem experimentar.

“O primeiro erro que as pessoas cometem é pensar que a equipe de teste é responsável por garantir a qualidade.”

Brian Marick

22.1.3 Estratégia de teste de software – visão global

O processo de software pode ser visto como a espiral ilustrada na Figura 22.1. Inicialmente, a engenharia de sistemas define o papel do software e passa à análise dos requisitos de software, na qual são estabelecidos o domínio da informação, função, comportamento, desempenho, restrições e critérios de validação para o software. Deslocando-se para o interior da espiral, chega-se ao projeto e, por fim, à codificação. Para desenvolver software de computador,

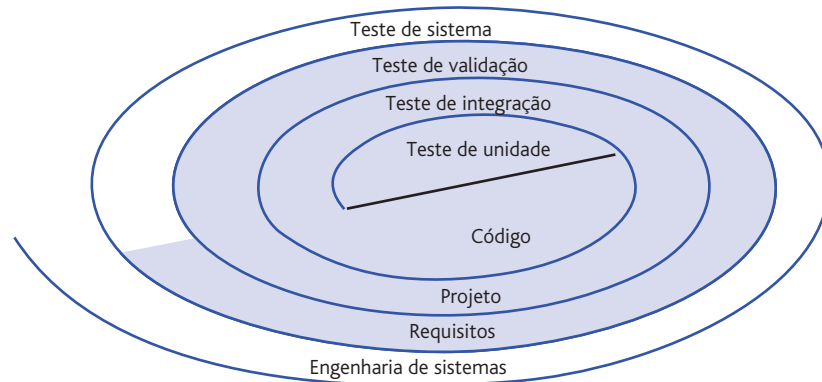


FIGURA 22.1 Estratégia de teste.

Qual é a estratégia global para teste de software?

percorre-se a espiral para o interior ao longo de linhas que indicam a diminuição do nível de abstração a cada volta.

Uma estratégia para teste de software pode também ser vista no conceito da espiral (Figura 22.1). O *teste de unidade* começa no centro da espiral e se concentra em cada unidade (por exemplo: componente, classe ou objeto de conteúdo de WebApp) do software, conforme implementado no código-fonte. O teste prossegue movendo-se em direção ao exterior da espiral, passando pelo *teste de integração*, em que o foco está no projeto e construção da arquitetura de software. Continuando na mesma direção da espiral, encontramos o *teste de validação*, em que requisitos estabelecidos como parte dos requisitos de modelagem são validados em relação ao software criado. Por fim, chegamos ao *teste do sistema*, no qual o software e outros elementos são testados como um todo. Para testar um software de computador, percorre-se a espiral em direção ao seu exterior, ao longo de linhas que indicam o escopo do teste a cada volta.

Recursos úteis para testadores de software podem ser encontrados em www.SQAtester.com.

Considerando o processo de um ponto de vista procedimental, o teste dentro do contexto de engenharia de software é, na realidade, uma série de quatro etapas implementadas sequencialmente. As etapas estão ilustradas na Figura 22.2. Inicialmente, os testes focalizam cada componente individualmente, garantindo que ele funcione adequadamente como uma unidade. Daí o nome *teste de unidade*. O teste de unidade usa intensamente técnicas de teste, com caminhos específicos na estrutura de controle de um componente para garantir a cobertura completa e a máxima detecção de erros. Em seguida, o componente deve ser montado ou integrado para formar o pacote de software completo. O *teste de integração* cuida de problemas associados a aspectos duais de verificação e construção de programa. Técnicas de projeto de casos de teste que focalizam entradas e saídas são mais predominantes durante a integração, embora técnicas que usam caminhos específicos de programa possam ser utilizadas para segurança dos principais caminhos de controle. Depois que o software foi integrado (construído), é executada uma série de *testes de ordem superior*. Os critérios de validação (estabelecidos durante a análise de requisitos) devem ser avaliados. O *teste de validação* proporciona a garantia final de que o software satisfaz a todos os requisitos funcionais, comportamentais e de desempenho.

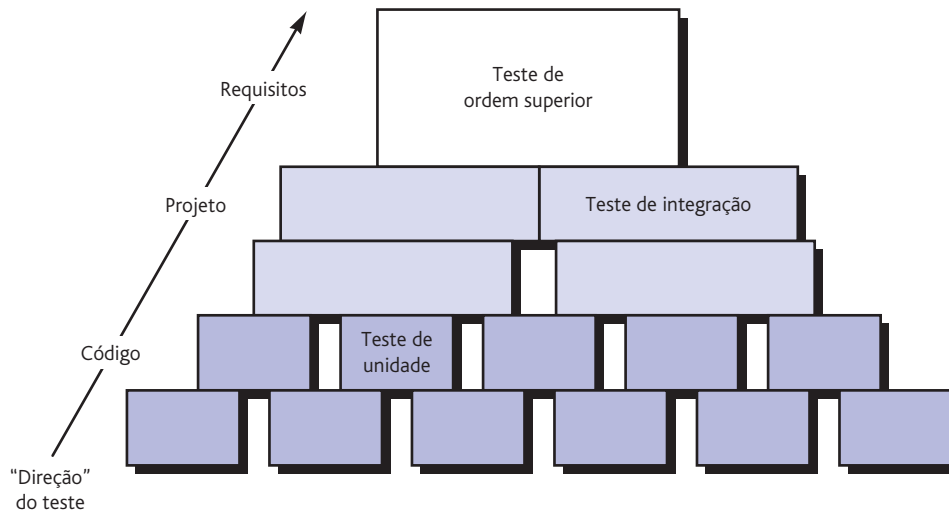


FIGURA 22.2 Etapas de teste de software.

A última etapa de teste de ordem superior extrapola os limites da engenharia de software, entrando em um contexto mais amplo de engenharia de sistemas de computadores. O software, uma vez validado, deve ser combinado com outros elementos do sistema (por exemplo, hardware, pessoas, base de dados). O *teste de sistema* verifica se todos os elementos se combinam corretamente e se a função/desempenho global do sistema é obtida.

CASASEGURA



Preparando-se para o teste

Cena: No escritório de Doug Miller, quando o projeto no nível de componente está em andamento e começa a construção de certos componentes.

Atores: Doug Miller, gerente de engenharia de software, Vinod, Jamie, Ed e Shakira – membros da equipe de engenharia de software do CasaSegura.

Conversa:

Doug: Parece-me que não dedicamos tempo suficiente para falar sobre o teste.

Vinod: É verdade, mas nós estávamos todos um tanto ocupados. Além disso, estivemos pensando sobre isso... Na verdade, mais do que pensando.

Doug (sorrindo): Eu sei... todos nós estamos sobrecarregados, mas ainda temos de pensar adiante.

Shakira: Eu gosto da ideia de projetar testes de unidades antes de começar a codificar qualquer um de meus componentes – e é o que estou tentando fazer. Tenho um arquivo grande de testes a ser executados logo que o código dos meus componentes estiver completo.

Doug: Esse é o conceito de Extreme Programming [um processo ágil de desenvolvimento de software; veja o Capítulo 5], não é mesmo?

Ed: É. Apesar de não estarmos usando Extreme Programming diretamente, decidimos que seria uma boa ideia projetar testes unitários antes de criar o componente – o projeto nos dá todas as informações de que precisamos.

Jamie: Eu já fiz a mesma coisa.

Vinod: E eu assumi o papel de integrador, de forma que, todas as vezes que um dos rapazes passar um componente para mim, vou integrá-lo e executar uma série de testes de regressão no programa parcialmente integrado. Estive trabalhando para projetar uma série de testes apropriados para cada função do sistema.

Doug (para Vinod): Com que frequência você fará os testes?

Vinod: Todos os dias... até que o sistema esteja integrado... Bem, quero dizer, até que o incremento de software que pretendemos fornecer esteja integrado.

Doug: Vocês estão mais adiantados do que eu!

Vinod (rindo): Antecipação é tudo no negócio de software, chefe.

22.1.4 Critérios para conclusão do teste

Quando finalizamos o teste?

“Teste que leva em conta somente os requisitos do usuário é como a inspeção de um edifício baseado apenas no trabalho executado pelo projetista de interiores, não considerando as fundações, vigas e tubulações.”

Boris Beizer

Uma questão clássica surge todas as vezes que se discute teste de software: “Quando podemos dizer que terminamos os testes – como podemos saber que já testamos o suficiente?”. Infelizmente, não há uma resposta definitiva para essa pergunta, mas há algumas respostas pragmáticas e algumas tentativas iniciais e empíricas.

Uma resposta é: “O teste nunca termina; o encargo simplesmente passa do engenheiro de software para o usuário”. Todas as vezes que o usuário executa o programa no computador, o programa está sendo testado. Esse fato destaca a importância de outras atividades de garantia da qualidade do software. Outra resposta (um tanto cínica, mas, ainda assim, exata) é: “O teste acaba quando o tempo ou o dinheiro acabam”.

Embora alguns profissionais possam argumentar a respeito dessas respostas, o fato é que é necessário um critério mais rigoroso para determinar quando já foram executados testes em número suficiente. A abordagem *engenharia de software sala limpa* (Capítulo 28) sugere técnicas de uso estatísticas [Kel00] que executam uma série de testes derivados de uma amostragem estatística de todas as execuções possíveis do programa por todos os usuários em uma população escolhida. Coletando métricas durante o teste do software e utilizando modelos estatísticos existentes, é possível desenvolver diretrizes significativas para responder à questão: “Quando terminamos o teste?”.

22.2 Problemas estratégicos

Quais as diretrizes que levam a uma estratégia de teste de software bem-sucedida?

Uma excelente lista de recursos de teste pode ser encontrada em www.SQAtester.com.

Mais adiante neste capítulo, apresentamos uma estratégia sistemática para teste de software. Mas até mesmo a melhor estratégia fracassará se não for resolvida uma série de problemas e obstáculos. Tom Gilb [Gil95] argumenta que uma estratégia de teste de software terá sucesso somente quando os testadores de software: (1) especificarem os requisitos do produto de uma maneira quantificável muito antes de começar o teste, (2) definirem explicitamente os objetivos do teste, (3) entenderem os usuários do software e desenvolverem um perfil para cada categoria de usuário, (4) desenvolverem um plano de teste que enfatize o “teste do ciclo rápido”,² (5) criarem software “robusto” que seja projetado para testar-se a si próprio (o conceito de antidefeito lantibugging¹ é discutido na Seção 22.3.1), (6) usarem revisões técnicas eficazes como filtro antes do teste, (7) realizarem revisões técnicas para avaliar a estratégia de teste e os próprios casos de teste e (8) desenvolverem abordagem de melhoria contínua (Capítulo 37) para o processo de teste.

² Gilb [Gil95] recomenda que uma equipe de software “aprenda a testar em ciclos rápidos (2% do trabalho de projeto) de incrementos de funcionalidade e/ou melhora da qualidade úteis ao cliente, ou pelo menos passíveis de experimentação no campo”. O retorno gerado por esses testes de ciclo rápido pode ser usado para controlar os níveis de qualidade e as correspondentes estratégias de teste.

22.3 Estratégias de teste para software convencional³

Muitas estratégias podem ser utilizadas para testar um software. Em um dos extremos, pode-se aguardar até que o sistema esteja totalmente construído e então realizar os testes no sistema completo na esperança de encontrar erros. Essa abordagem, embora atraente, simplesmente não funciona. Ela resultará em um software cheio de erros que desapontará a todos os envolvidos. No outro extremo, você pode executar testes diariamente, sempre que uma parte do sistema for construída.

Uma estratégia de teste escolhida por muitas equipes de software está entre os dois extremos. Ela assume uma visão incremental do teste, começando com o teste das unidades individuais de programa, passando para os testes destinados a facilitar a integração de unidades (às vezes diariamente) e culminando com testes que usam o sistema concluído. Cada uma dessas classes de testes é descrita nas próximas seções.

22.3.1 Teste de unidade

O *teste de unidade* focaliza o esforço de verificação na menor unidade de projeto do software – o componente ou módulo de software. Usando como guia a descrição de projeto no nível de componente, caminhos de controle importantes são testados para descobrir erros dentro dos limites do módulo. A complexidade relativa dos testes e os erros que revelam são limitados pelo escopo restrito estabelecido para o teste de unidade. Esse teste enfoca a lógica interna de processamento e as estruturas de dados dentro dos limites de um componente. Esse tipo de teste pode ser conduzido em paralelo para diversos componentes.

Considerações sobre o teste de unidade. Os testes de unidade estão ilustrados esquematicamente na Figura 22.3. A interface do módulo é testada para assegurar que as informações fluam corretamente para dentro e para fora da unidade de programa que está sendo testada. A estrutura de dados local é examinada para garantir que os dados armazenados temporariamente mantenham sua integridade durante todos os passos na execução de um algoritmo. Todos os caminhos independentes da estrutura de controle são usados para assegurar que todas as instruções em um módulo tenham sido executadas pelo menos uma vez. As condições limite são testadas para garantir que o módulo opere adequadamente nas fronteiras estabelecidas para limitar ou restringir o processamento. Por fim, são testados todos os caminhos de manipulação de erro.

O fluxo de dados por meio da interface de um componente é testado antes de iniciar qualquer outro teste. Se os dados não entram e saem corretamente, todos os outros testes são discutíveis. Além disso, estruturas de dados locais deverão ser ensaiadas, e o impacto local sobre dados globais deve ser apurado (se possível) durante o teste de unidade.

Não é má ideia projetar casos de teste de unidade antes de desenvolver o código para um componente. É bom assegurar-se de que você desenvolverá código que passará nos testes.

³ Neste livro, utilizamos os termos *software convencional* ou *software tradicional* para nos referir às arquiteturas de software do tipo hierárquico comum ou chamada-e-retorno, frequentemente encontradas em uma variedade de domínios de aplicações. Arquiteturas de software tradicionais não são orientadas a objeto e não abrangem WebApps, nem aplicativos móveis.

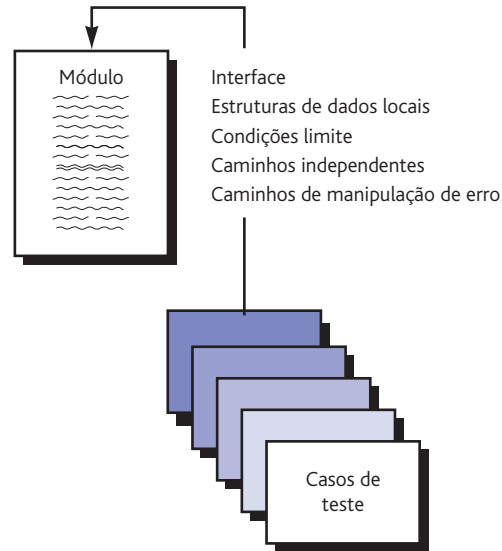


FIGURA 22.3 Teste de unidade.

Quais são os erros mais comuns encontrados durante o teste de unidade?

O teste seletivo de caminhos de execução é uma tarefa essencial durante o teste de unidade. Casos de teste devem ser projetados para descobrir erros devido a computações errôneas, comparações incorretas ou fluxo de controle inadequado.

Projete testes para executar todos os caminhos de manipulação de erro. Se não fizer isso, o caminho pode falhar quando for solicitado, piorando uma situação já ruim.

O teste de fronteira é uma das tarefas mais importantes do teste de unidade. O software frequentemente falha nas suas fronteiras. Isto é, os erros frequentemente ocorrem quando o n -ésimo elemento de um conjunto n -dimensional é processado, quando a i -ésima repetição de um laço com i passadas é chamada ou quando o valor máximo ou mínimo permitido é encontrado. Casos de teste que utilizam estrutura de dados, fluxo de controle e valores de dados logo abaixo, iguais ou logo acima dos máximos e mínimos têm grande possibilidade de descobrir erros.

Um bom projeto prevê condições de erro e estabelece caminhos de manipulação de erro para redirecionar ou encerrar ordenadamente o processamento quando ocorre um erro. Yourdon [You75] chama essa abordagem de *antidefeitos*. Infelizmente, há uma tendência de incorporar manipulação de erro no software e nunca testá-la. Se são implementados caminhos de manipulação de erros, eles devem ser testados.

Entre os erros em potencial que devem ser testados quando a manipulação de erro é avaliada, estão: (1) descrição confusa do erro, (2) o erro apontado não corresponde ao erro encontrado, (3) a condição do erro causa intervenção do sistema antes da manipulação do erro, (4) o processamento exceção-condição é incorreto ou (5) a descrição do erro não fornece informações suficientes para ajudar na localização da causa do erro.

Procedimentos de teste de unidade. O teste de unidade normalmente é considerado um auxiliar para a etapa de codificação. O projeto dos testes de unidade pode ocorrer antes de a codificação começar ou depois que o código-fonte tiver sido gerado. Um exame das informações de projeto fornece instruções para estabelecer casos de teste que provavelmente mostrarão os erros em

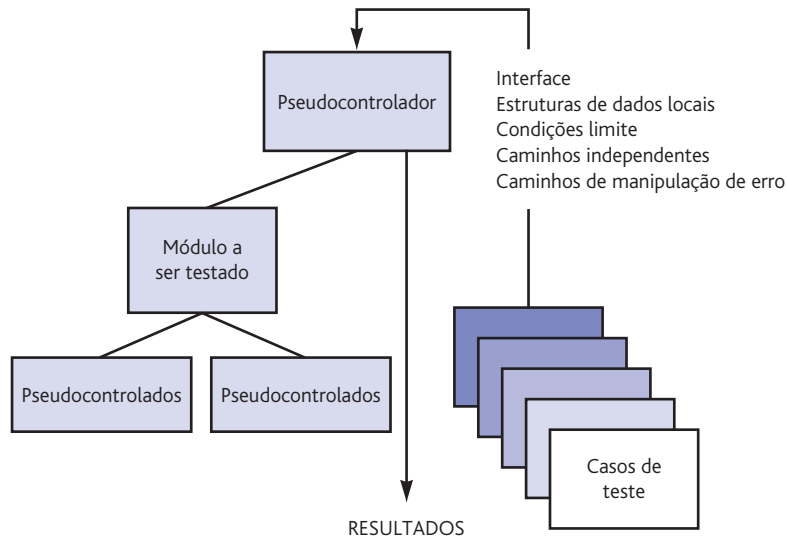


FIGURA 22.4 Ambiente de teste de unidade.

cada uma das categorias descritas anteriormente. Cada caso de teste deverá ser acoplado a um conjunto de resultados esperados.

Como um componente não é um programa independente, deve ser desenvolvido um pseudocontrolador (*driver*) e/ou um pseudocontrolado (*stub*) para cada teste de unidade. O ambiente de teste de unidade está ilustrado na Figura 22.4. Em muitas aplicações, um *pseudocontrolador* nada mais é do que um “programa principal” que aceita dados do caso de teste, passa esses dados para o componente (a ser testado) e imprime resultados relevantes. Os *pseudocontroladores* servem para substituir módulos subordinados (chamados pelo) ao componente a ser testado. Um pseudocontrolado, ou “pseudosubprograma”, usa a interface dos módulos subordinados, pode fazer uma manipulação de dados mínima, fornece uma verificação de entrada e retorna o controle para o módulo que está sendo testado.

Pseudocontroladores e pseudocontrolados representam despesas indiretas. Isto é, ambos são software que devem ser codificados (projeto formal normalmente não é aplicado), mas que não são fornecidos com o produto de software final. Se os pseudocontroladores e pseudocontrolados são mantidos simples, as despesas reais indiretas são relativamente baixas. Infelizmente, muitos componentes não podem ser adequadamente testados no nível de unidade de modo adequado com software adicional simples. Em tais casos, o teste completo pode ser adiado até a etapa de integração (em que os pseudocontroladores e pseudocontrolados também são usados).

22.3.2 Teste de integração

Um novato no mundo do software pode levantar uma questão aparentemente legítima quando todos os módulos tiverem passado pelo teste de unidade: “Se todos funcionam individualmente, por que você duvida que funcionem quando estiverem juntos?”. O problema, naturalmente, é “colocá-los todos juntos” – fazer interfaces. Dados podem ser perdidos através de uma interface; um componente

Há algumas situações nas quais você não terá recursos para fazer um teste de unidade simples. Selecione os módulos críticos ou complexos e faça o teste de unidade apenas neles.

Adotar o método “big bang” é uma abordagem ineficaz, destinada a fracassar. Integre de forma incremental, testando enquanto trabalha.

pode ter um efeito inesperado ou adverso sobre outro; subfunções, quando combinadas, podem não produzir a função principal desejada; imprecisão aceitável individualmente pode ser amplificada em níveis não aceitáveis; estruturas de dados globais podem apresentar problemas. Infelizmente, a lista não acaba.

O teste de integração é uma técnica sistemática para construir a arquitetura de software, ao mesmo tempo em que se realizam testes para descobrir erros associados às interfaces. O objetivo é construir uma estrutura de programa determinada pelo projeto a partir de componentes testados em unidade.

Muitas vezes há uma tendência de tentar a integração não incremental; isto é, construir o programa usando uma abordagem “*big bang*”. Todos os componentes são combinados antecipadamente, e o programa inteiro é testado como um todo. E, normalmente, o resultado é o caos! Os erros são encontrados, mas a correção é difícil porque o isolamento das causas é complicado pela vasta extensão do programa inteiro.

A integração incremental é o oposto da abordagem *big bang*. O programa é construído e testado em pequenos incrementos, em que os erros são mais fáceis de isolar e corrigir; as interfaces têm maior probabilidade de ser testadas completamente; e uma abordagem sistemática de teste pode ser aplicada. Nos próximos parágrafos serão discutidas algumas estratégias de integração incremental diferentes.

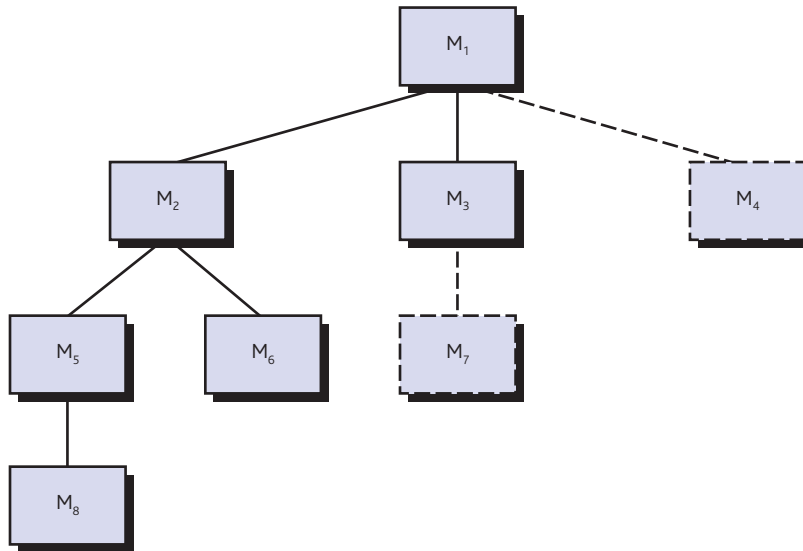
Ao desenvolver um cronograma de projeto, você terá de considerar a maneira pela qual a integração ocorrerá para que os componentes estejam disponíveis quando forem necessários.

Integração descendente. *Teste de integração descendente (top-down)* é uma abordagem incremental para a construção da arquitetura de software. Os módulos são integrados deslocando-se para baixo por meio da hierarquia de controle, começando com o módulo de controle principal (programa principal). Módulos subordinados ao módulo de controle principal são incorporados à estrutura de uma maneira primeiro-em-profundidade ou primeiro-em-largura (*depth-first* ou *breadth-first*).

Na Figura 22.5, a *integração primeiro-em-profundidade* integra todos os componentes em um caminho de controle principal da estrutura do programa. A seleção de um caminho principal é, de certa forma, arbitrária e depende das características específicas da aplicação. Por exemplo, selecionando o caminho da esquerda, os componentes M_1 , M_2 e M_5 seriam integrados primeiro. Em seguida, seria integrado M_8 – ou, se necessário para o funcionamento apropriado de M_2 , M_6 . Depois, são criados os caminhos de controle central e da direita. A *integração primeiro-em-largura* incorpora todos os componentes diretamente subordinados a cada nível, movendo-se através da estrutura horizontalmente. Pela figura, os componentes M_2 , M_3 e M_4 seriam integrados primeiro. Em seguida vem o próximo nível de controle, M_5 , M_6 e assim por diante. O processo de integração é executado em uma série de cinco passos:

1. O módulo de controle principal é utilizado como um testador (*test driver*), e todos os componentes diretamente subordinados ao módulo de controle principal são substituídos por pseudocontroladores.
2. Dependendo da abordagem de integração selecionada (isto é, primeiro-em-profundidade ou primeiro-em-largura), pseudocontroladores subordinados são substituídos, um de cada vez, pelos componentes reais.
3. Os testes são feitos à medida que cada componente é integrado.

Quais são os passos para a integração descendente?

**FIGURA 22.5** Integração descendente.

4. Ao fim de cada conjunto de testes, outro pseudocontrolador substitui o componente real.
5. O teste de regressão (discutido mais adiante nesta seção) pode ser executado para garantir que não tenham sido introduzidos novos erros.

O processo continua a partir do passo 2 até que toda a estrutura do programa esteja concluída. A estratégia de integração descendente verifica os principais pontos de controle ou decisão antecipada no processo de teste. Em uma estrutura de programa bem construída, a tomada de decisão ocorre nos níveis superiores da hierarquia e, portanto, é encontrada primeiro. Se existirem problemas de controle principal, um reconhecimento prévio é essencial. Se for selecionada a integração em profundidade, uma função completa do software pode ser implementada e demonstrada. A demonstração antecipada da capacidade funcional é um gerador de confiança para todos os envolvidos no programa.

Integração ascendente. O teste de integração ascendente (*bottom-up*), como o nome diz, começa a construção e o teste com *módulos atômicos* (isto é, componentes nos níveis mais baixos na estrutura do programa). Como os componentes são integrados de baixo para cima, a funcionalidade proporcionada por componentes subordinados a determinado nível está sempre disponível, e a necessidade de pseudocontroladores é eliminada. Uma estratégia de integração ascendente pode ser implementada com os seguintes passos:

1. Componentes de baixo nível são combinados em agregados (*clusters*, também chamados de *builds*, construções) que executam uma subfunção específica de software.
2. Um *pseudocontrolador* (um programa de controle para teste) é escrito para coordenar entrada e saída do caso de teste.
3. O agregado é testado.

Quais problemas podem ser encontrados quando é escolhida a integração descendente?

Quais são os passos para a integração ascendente?

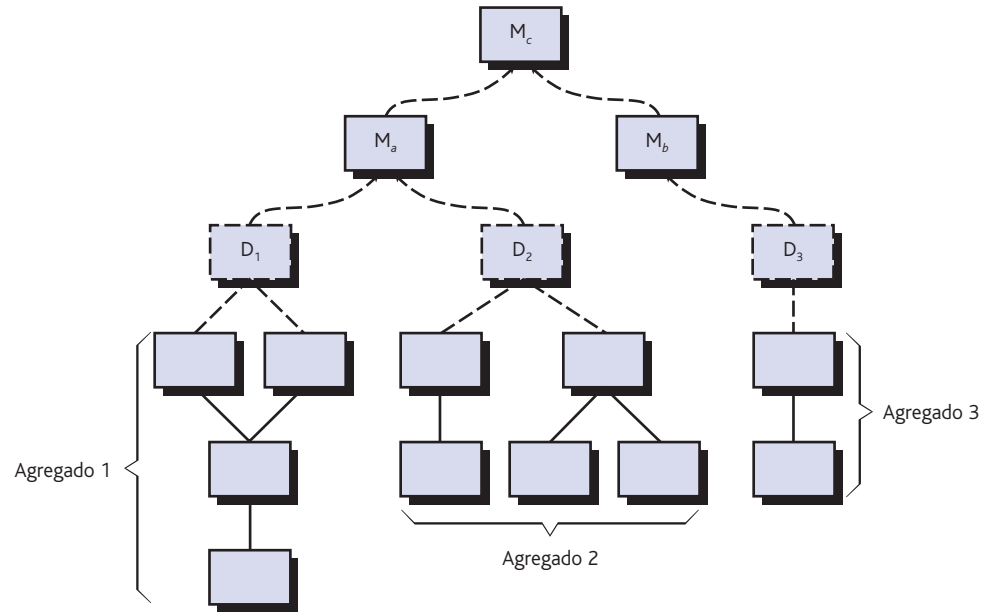


FIGURA 22.6 Integração ascendente.

4. Os pseudocontroladores (*drivers*) são removidos, e os agregados são combinados movendo-se para cima na estrutura do programa.

A integração ascendente elimina a necessidade de pseudocontroladores complexos.

A integração segue o padrão ilustrado na Figura 22.6. Os componentes são combinados para formar os agregados (*clusters*) 1, 2 e 3. Cada um dos agregados é testado usando um pseudocontrolador (mostrado como um bloco tracejado). Componentes nos agregados 1 e 2 são subordinados a M_a . Os pseudocontroladores D_1 e D_2 são removidos, e os agregados fazem interface diretamente com M_a . De forma semelhante, o pseudocontrolador D_3 para o agregado 3 é removido antes da integração com o módulo M_b . M_a e M_b serão ambos finalmente integrados ao componente M_c e assim por diante.

À medida que a integração se move para cima, a necessidade de pseudocontroladores de testes separados diminui. Na verdade, se os níveis descendentes da estrutura do programa forem integrados de cima para baixo, o número de pseudocontroladores pode ser bastante reduzido, e a integração de agregados fica bastante simplificada.

O teste de regressão é uma estratégia importante para reduzir "efeitos colaterais". Execute testes de regressão toda vez que for feita uma alteração grande no software (incluindo a integração de novos componentes).

Teste de regressão. Cada vez que um novo módulo é acrescentado como parte do teste de integração, o software muda. Novos caminhos de fluxo de dados são estabelecidos, podem ocorrer novas entradas e saídas, e nova lógica de controle é chamada. Os efeitos colaterais associados a essas alterações podem causar problemas em funções que antes funcionavam corretamente. No contexto de uma estratégia de teste de integração, o *teste de regressão* é a reexecução do mesmo subconjunto de testes que já foram executados, para assegurar que as alterações não tenham propagado efeitos colaterais indesejados. O teste de regressão ajuda a garantir que as alterações (devido ao teste ou por outras razões) não introduzam comportamento indesejado ou erros adicionais.

O teste de regressão pode ser executado manualmente, reexecutando um subconjunto de todos os casos de teste ou usando ferramentas automáti-

cas de captura/reexecução. *Ferramentas de captura/reexecução* permitem que o engenheiro de software capture casos de teste e resultados para reexecução e comparação subsequente. O *conjunto de teste de regressão* (o subconjunto de testes a serem executados) contém três classes diferentes de casos de teste:

- Uma amostra representativa dos testes que usam todas as funções do software.
- Testes adicionais que focalizam as funções de software que podem ser afetadas pela alteração.
- Testes que focalizam os componentes do software que foram alterados.

À medida que o teste de integração progride, o número de testes de regressão pode crescer muito. Portanto, o conjunto de testes de regressão deve ser projetado de forma a incluir somente aqueles testes que tratam de uma ou mais classes de erros em cada uma das funções principais do programa.

Teste fumaça. *Teste fumaça* é uma abordagem de teste de integração usada frequentemente quando produtos de software são desenvolvidos. É projetado como um mecanismo de marca-passo para projetos com prazo crítico, permitindo que a equipe de software avalie o projeto frequentemente. Em essência, a abordagem teste fumaça abrange as seguintes atividades:

1. Componentes de software que foram traduzidos para um código são integrados em uma “*construção*” (*build*). Uma construção inclui todos os arquivos de dados, bibliotecas, módulos reutilizáveis e componentes necessários para implementar uma ou mais funções do produto.
2. Uma série de testes é criada para expor erros que impedem a construção de executar corretamente sua função. A finalidade deve ser descobrir erros “bloqueadores” (*showstopper*) que apresentam a mais alta probabilidade de atrasar o cronograma do software.
3. A construção é integrada a outras construções, e o produto inteiro (em sua forma atual) passa diariamente pelo teste fumaça. A abordagem de integração pode ser descendente ou ascendente.

A frequência diária dos testes dá, tanto aos gerentes quanto aos profissionais, uma ideia realística do progresso do teste de integração. McConnell [McC96] descreve o teste fumaça da seguinte maneira:

O teste fumaça deve usar o sistema inteiro de fim-a-fim. Ele não precisa ser exaustivo, mas deve ser capaz de expor os principais problemas. O teste fumaça deve ser bastante rigoroso, de forma que, se a construção passar, você pode supor que ele é estável o suficiente para ser testado mais rigorosamente.

O teste fumaça proporciona muitos benefícios quando aplicado a projetos de engenharia de software complexos e de prazo crítico:

- *O risco da integração é minimizado.* Como os testes fumaça são feitos diariamente, as incompatibilidades e outros erros bloqueadores são descobertos logo, reduzindo, assim, a probabilidade de impacto sério no cronograma quando os erros são descobertos.

O teste fumaça pode ser caracterizado como uma estratégia de integração rolante. O software é recriado (com novos componentes acrescentados), e o teste fumaça é realizado todos os dias.

“Trate a construção diária (daily build) como o marca-passo do projeto. Se não houver marca-passo, o projeto está morto.”

Jim McCarthy

Quais benefícios podem ser obtidos do teste fumaça?

- *A qualidade do produto final é melhorada.* Como a abordagem é orientada para a construção (integração), o teste fumaça pode descobrir erros funcionais, bem como erros de arquitetura e de projeto no nível de componente. Se esses erros forem corrigidos logo, isso resultará em melhor qualidade do produto.
- *O diagnóstico e a correção dos erros são simplificados.* Como todas as abordagens de teste de integração, os erros descobertos durante o teste fumaça provavelmente estarão associados aos “novos incrementos do software” – ou seja, o software que acaba de ser acrescentado à(s) construção(ões) é uma causa provável de um erro que acaba de ser descoberto.
- *É mais fácil avaliar o progresso.* A cada dia que passa, uma parte maior do software já está integrada e é demonstrado que funciona. Isso melhora a moral da equipe e dá aos gerentes uma boa indicação de que houve progressos.

Artefatos do teste de integração. Um plano global para integração do software e uma descrição dos testes específicos são documentados em uma *Especificação de Teste*. Esse artefato incorpora um plano de teste e um documento de teste e torna-se parte da configuração do software. O teste é dividido em fases e construções que tratam de características funcionais e comportamentais específicas do software. Por exemplo, o teste de integração para o sistema de segurança *CasaSegura* pode ser dividido nas seguintes fases de teste: interação com o usuário, processamento do sensor, funções de comunicação e processamento do alarme.

Cada uma dessas fases de teste de integração representa uma categoria funcional ampla dentro do software e geralmente pode ser relacionada a um domínio específico dentro da arquitetura do software. Portanto, são criadas construções de programa (grupos de módulos) para corresponder a cada fase.

Um cronograma para a integração, o desenvolvimento de software de uso geral e tópicos relacionados também são discutidos como parte do plano de teste. São estabelecidas as datas de início e fim para cada fase e são definidas “janelas de disponibilidade” para módulos submetidos a teste de unidade. Uma breve descrição do software de uso geral (pseudocontroladores e pseudocontrolados) concentra-se nas características que poderiam exigir dedicação especial. Por fim, são descritos o ambiente e os recursos de teste. Configurações de hardware não usuais, simuladores exóticos e ferramentas ou técnicas especiais de teste são alguns dos muitos tópicos que também podem ser discutidos.

Em seguida, é descrito o procedimento de teste detalhado necessário para realizar o plano de teste. São descritas a ordem de integração e os testes correspondentes em cada etapa de integração. É incluída também uma lista de todos os casos de teste (anotados para referência subsequente) e dos resultados esperados.

Um histórico dos resultados reais do teste, problemas ou peculiaridades é registrado em um *Relatório de Teste* que pode ser anexado à *Especificação de Teste*, se desejado. Essas informações podem ser vitais durante a manutenção do software. São apresentados também referências e apêndices apropriados.

22.4 Estratégias de teste para software orientado a objetos⁴

De forma simplificada, o objetivo do teste é encontrar o maior número possível de erros com um esforço gerenciável durante um intervalo de tempo realístico. Embora esse objetivo fundamental permaneça inalterado para software orientado a objetos, a natureza do software orientado a objetos muda tanto a estratégia quanto a tática de teste (Capítulo 24).

22.4.1 Teste de unidade em contexto orientado a objetos

Quando consideramos o software orientado a objetos, o conceito de unidades se modifica. O encapsulamento controla a definição de classes e objetos. Isso significa que cada classe e cada instância de uma classe (objeto) empacotam atributos (dados) e as operações que manipulam esses dados. Uma classe encapsulada é usualmente o foco do teste de unidade. No entanto, operações (métodos) dentro da classe são as menores unidades testáveis. Como uma classe pode conter um conjunto de diferentes operações, e uma operação em particular pode existir como parte de um conjunto de diferentes classes, a tática aplicada ao teste de unidade precisa ser modificada.

Não podemos mais testar uma única operação isoladamente (a visão convencional do teste de unidade), mas sim como parte de uma classe. Para ilustrar, considere uma hierarquia de classes na qual uma operação *X* é definida para a superclasse e é herdada por várias subclasses. Cada subclasse usa uma operação *X*, mas é aplicada dentro do contexto dos atributos e operações privadas definidas para a subclasse. O contexto no qual a operação *X* é usada varia de maneira sutil; desse modo, é necessário testar a operação *X* no contexto de cada uma das subclasses. Isso significa que testar a operação *X* isoladamente (a abordagem de teste de unidade convencional) é usualmente ineficaz no contexto orientado a objetos.

O teste de classe para software orientado a objetos é equivalente ao teste de unidade para software convencional. Ao contrário do teste de unidade para software convencional, que tende a focalizar o detalhe algorítmico de um módulo e os dados que fluem por meio da interface do módulo, o teste de classe para software orientado a objetos é controlado pelas operações encapsuladas pela classe e o comportamento de estado da classe.

O teste de classe para OO é análogo ao teste de módulo para software convencional. Não é aconselhável testar operações isoladamente.

22.4.2 Teste de integração em contexto orientado a objetos

Devido ao software orientado a objetos não ter uma estrutura óbvia de controle hierárquico, as estratégias tradicionais de integração descendente e ascendente (Seção 22.3.2) têm pouco significado. Além disso, integrar operações uma de cada vez em uma classe (a abordagem convencional de integração incremental) frequentemente é impossível devido “às interações diretas e indiretas dos componentes que formam a classe” [Ber93].

Há duas estratégias diferentes para teste de integração de sistemas OO [Bin94b]. A primeira, *teste baseado em sequência de execução* (*thread-based testing*), integra o conjunto de classes necessárias para responder a uma en-

⁴ Os conceitos básicos orientados a objetos são apresentados no Apêndice 2.

Uma estratégia importante para integração de software OO é o teste com base em sequência de execução (*thread*). Sequências de execução são conjuntos de classes que respondem a uma entrada ou evento. Testes baseados em uso focalizam classes que não colaboram intensamente com outras classes.

trada ou um evento do sistema. Cada sequência de execução é integrada e testada individualmente. O teste de regressão é aplicado para garantir que não ocorram efeitos colaterais. A segunda abordagem de integração, *teste baseado em uso* (*use-based testing*), inicia a construção do sistema testando as classes (chamadas de *classes independentes*) que usam poucas (ou nenhuma) classes *servidoras*. Depois que as classes independentes são testadas, testa-se a próxima camada de classes, chamadas *classes dependentes*, que usam as classes independentes. Essa sequência de camadas de teste de classes dependentes continua até que todo o sistema seja construído.

O uso de pseudocontroladores e pseudocontrolados também muda quando é executado o teste de integração de sistemas OO. Pseudocontroladores podem ser utilizados para testar operações no nível mais baixo e para o teste de grupos inteiros de classes. Um pseudocontrolador também pode ser usado para substituir a interface de usuário, de maneira que os testes da funcionalidade do sistema podem ser conduzidos antes da implementação da interface. Pseudocontrolados podem ser usados em situações nas quais é necessária a colaboração entre classes, mas uma (ou mais) das classes colaboradoras ainda não foi totalmente implementada.

O *teste de conjunto* (*cluster testing*) é uma etapa no teste de integração de software OO. Nesse caso, um agregado de classes colaboradoras (determinado examinando-se os modelos CRC e o modelo objeto-relacionamento) é exercitado projetando-se casos de teste que tentam descobrir erros nas colaborações.

22.5 Estratégias de teste para WebApps

A estratégia para teste de WebApp adota os princípios básicos para todo o teste de software e aplica táticas usadas em sistemas OO. Os passos a seguir resumem a abordagem:

A estratégia geral para teste de WebApp pode ser resumida nos dez passos descritos aqui.

1. O modelo de conteúdo para a WebApp é revisto para descobrir erros.
2. O modelo de interface é revisto para garantir que todos os casos de uso possam ser acomodados.
3. O modelo de projeto da WebApp é revisto para descobrir erros de navegação.
4. A interface com o usuário é testada para descobrir erros nos mecanismos de apresentação e/ou navegação.
5. Para cada componente funcional é feito o teste de unidade.
6. É testada a navegação por toda a arquitetura.
7. A WebApp é implementada em uma variedade de configurações ambientais diferentes e testada quanto à compatibilidade com cada configuração.
8. São executados testes de segurança na tentativa de explorar vulnerabilidades na WebApp ou em seu ambiente.
9. São realizados testes de desempenho.

Excelentes artigos sobre teste de WebApp podem ser encontrados em www.stickyminds.com/testing.asp.

10. A WebApp é testada por uma população de usuários finais controlados e monitorados. Os resultados da interação desses usuários com o sistema são avaliados quanto a erros.

Como muitas WebApps evoluem continuamente, o processo de teste é uma atividade contínua, conduzido pelo pessoal de suporte que usa testes de regressão derivados dos testes de desenvolvimento quando a WebApp foi desenvolvida inicialmente. Os métodos de teste para WebApp são considerados no Capítulo 25.

22.6 Estratégias de teste para aplicativos móveis

A estratégia de teste de aplicativos móveis adota os princípios básicos de todo teste de software. Contudo, a natureza única dos aplicativos móveis exige considerar várias abordagens de teste especializadas:

- *Teste da experiência do usuário.* Os usuários são incluídos no início do processo de desenvolvimento para garantir que o aplicativo móvel cumpra as expectativas de usabilidade e acessibilidade dos envolvidos em todos os dispositivos suportados.
- *Teste de compatibilidade de dispositivo.* Os testadores verificam se o aplicativo móvel funciona corretamente em todas as combinações de hardware e software exigidas.
- *Teste de desempenho.* Os testadores verificam os requisitos não funcionais exclusivos dos dispositivos móveis (por exemplo, tempos de download, velocidade do processador, capacidade de armazenamento, disponibilidade de energia).
- *Teste de conectividade.* Os testadores verificam se o aplicativo móvel consegue acessar quaisquer redes ou Web services necessários e se pode tolerar acesso à rede fraco ou interrompido.
- *Teste de segurança.* Os testadores verificam se o aplicativo móvel não compromete os requisitos de privacidade ou segurança de seus usuários.
- *Teste em condições naturais.* O aplicativo é testado sob condições realistas em dispositivos reais, em uma variedade de ambientes de rede em todo o mundo.
- *Teste de certificação.* Os testadores verificam se o aplicativo móvel atende aos padrões estabelecidos pelas lojas de aplicativo que vão distribuí-lo.

Os métodos de teste para aplicativos móveis são considerados no Capítulo 26.

22.7 Teste de validação

O teste de validação começa quando termina o teste de integração, quando os componentes individuais já foram exercitados, o software está completamente montado como um pacote e os erros de interface já foram descobertos

Como todas as outras etapas de teste, a validação tenta descobrir erros, mas o foco está no nível de requisitos – em coisas que ficarão imediatamente aparentes para o usuário.

e corrigidos. No nível de validação ou de sistema, a distinção entre diferentes categorias de software desaparece. O teste focaliza ações visíveis ao usuário e saídas do sistema reconhecíveis pelo usuário.

A validação pode ser definida de várias maneiras, mas uma definição simples (embora rigorosa) é que a validação tem sucesso quando o software funciona de uma maneira que pode ser razoavelmente esperada pelo cliente. Nesse ponto, um desenvolvedor de software veterano pode protestar: “Quem ou o que é o árbitro para decidir o que são expectativas razoáveis?”. Se uma *Especificação de Requisitos de Software* foi desenvolvida, ela descreve todos os atributos do software visíveis ao usuário e contém uma seção denominada *Critérios de Validação* que forma a base para uma abordagem de teste de validação.

22.7.1 Critérios de teste de validação

A validação de software é conseguida por meio de uma série de testes que demonstram conformidade com os requisitos. Um plano de teste descreve as classes de testes a serem realizados e um procedimento de teste define casos de teste específicos destinados a garantir que todos os requisitos funcionais sejam satisfeitos, todas as características comportamentais sejam obtidas, todo o conteúdo seja preciso e adequadamente apresentado, todos os requisitos de desempenho sejam atendidos, a documentação esteja correta e outros requisitos sejam cumpridos (por exemplo, transportabilidade, compatibilidade, recuperação de erro, facilidade de manutenção). Se for descoberto um desvio em relação à especificação, é criada uma *lista de deficiências*. Deve ser estabelecido um método para solucionar deficiências (aceitável para os envolvidos).

22.7.2 Revisão da configuração

Um elemento importante do processo de validação é a *revisão da configuração*. A finalidade da revisão é garantir que todos os elementos da configuração do software tenham sido adequadamente desenvolvidos, estejam catalogados e tenham os detalhes necessários para amparar as atividades de suporte. A revisão de configuração, também chamada de auditoria, é discutida em mais detalhes no Capítulo 29.

22.7.3 Testes alfa e beta

É praticamente impossível para um desenvolvedor de software prever como o cliente realmente usará um programa. As instruções de uso podem ser mal interpretadas, combinações estranhas de dados podem ser usadas, resultados que pareçam claros para o testador podem ser confusos para um usuário no campo.

Quando é construído um software personalizado para um cliente, são feitos testes de aceitação para permitir ao cliente validar todos os requisitos. Conduzido pelo usuário e não por engenheiros de software, um teste de aceitação pode variar desde um *test drive* informal até uma série de testes planejados e sistematicamente executados. Na verdade, um teste de aceitação pode ser executado por um período de semanas ou meses, descobrindo, assim, erros cumulativos que poderiam degradar o sistema ao longo do tempo.

Se um software é desenvolvido como um produto para ser usado por muitos clientes, é impraticável executar testes formais de aceitação para cada

“Com uma boa verificação, todos os erros ficam expostos (por exemplo, com uma base suficientemente ampla de testadores beta e colegas desenvolvedores, quase todos os problemas serão caracterizados rapidamente, e a solução será óbvia para alguém).”

E. Raymond

cliente. Muitos construtores de software usam um processo chamado de teste alfa e beta para descobrir erros que somente o usuário parece ser capaz de encontrar.

O *teste alfa* é realizado na instalação do desenvolvedor por um grupo representativo de usuários finais. O software é usado em um cenário natural, com o desenvolvedor “espiando por cima dos ombros” dos usuários, registrando os erros e os problemas de uso. Os testes alfa são conduzidos em um ambiente controlado.

O *teste beta* é realizado nas instalações de um ou mais usuários finais. Diferentemente do teste alfa, o desenvolvedor geralmente não está presente. Portanto, o teste beta é uma aplicação “ao vivo” do software em um ambiente que não pode ser controlado pelo desenvolvedor. O cliente registra todos os problemas (reais ou imaginários) encontrados durante o teste beta e relata esses problemas para o desenvolvedor em intervalos regulares. Como resultado dos problemas relatados durante o teste beta, os engenheiros de software fazem modificações e então preparam a liberação do software para todos os clientes.

Uma variação do teste beta, chamada *teste de aceitação do cliente*, às vezes é executada quando é fornecido software personalizado a um cliente sob contrato. O cliente executa uma série de testes específicos na tentativa de descobrir erros antes de aceitar o software do desenvolvedor. Em alguns casos (por exemplo, um grande sistema corporativo ou governamental) o teste de aceitação pode ser muito formal e levar vários dias ou até mesmo semanas.

Qual é a diferença entre um teste alfa e um teste beta?



Preparando-se para a validação

Cena: Escritório de Doug Miller, onde continua o projeto no nível de componente e a criação de certos componentes.

Atores: Doug Miller, gerente de engenharia de software, Vinod, Jamie, Ed e Shakira – membros da equipe de engenharia de software do *CasaSegura*.

Conversa:

Doug: O primeiro incremento estará pronto para validação em... cerca de três semanas?

Vinod: Certo. A integração está indo bem. Estamos fazendo o teste fumaça diariamente, encontrando alguns erros, mas nada que não se possa resolver. Até agora, tudo bem.

Doug: Fale sobre a validação.

Shakira: Bem, usaremos como base para nosso projeto de teste todos os casos de uso. Ainda não comecei, mas desenvolveremos testes para todos os casos de uso pelos quais sou responsável.

Ed: A mesma coisa aqui.

Jamie: Eu também, mas temos de juntar as nossas ações para teste de aceitação e também para teste alfa e beta, não?

Doug: Sim. Na verdade, estive pensando que poderíamos contratar alguém para nos ajudar na validação. Tenho verba disponível no orçamento... e teríamos um novo ponto de vista.

Vinod: Eu acho que temos tudo sob controle.

Doug: Estou certo de que sim, mas um ITG nos dará uma visão independente sobre o software.

Jamie: Estamos com o tempo apertado aqui, Doug. Não temos tempo suficiente para pajear qualquer um que você trouxe aqui.

Doug: Eu sei, eu sei. Mas se um ITG trabalhar a partir dos requisitos e casos de uso, não será necessário muito acompanhamento.

Vinod: Eu ainda acho que temos tudo sob controle.

Doug: Eu sei, Vinod, mas vou insistir nisso. Vamos marcar uma reunião com o representante do ITG ainda esta semana. Vamos dar o pontapé inicial e ver até onde eles chegam.

Vinod: Ok, talvez eles possam aliviar a carga.

CASASEGURA

"Assim como a morte e os impostos, o teste é tanto desagradável quanto inevitável."

Ed Yourdon

22.8 Teste de sistema

No início deste livro, chamamos a atenção para o fato de que o software é apenas um elemento de um grande sistema de computador. No final, o software é incorporado a outros elementos do sistema (por exemplo, hardware, pessoas, informações), e é executada uma série de testes de integração de sistema e validação. Esses testes estão fora do escopo do processo de software e não são executados somente por engenheiros de software. No entanto, as etapas executadas durante o projeto de software e o teste podem aumentar muito a probabilidade de uma integração de software bem-sucedida em um sistema maior.

Um problema clássico do teste de sistema é a "procura do culpado". Isso ocorre quando é descoberto um erro e os desenvolvedores de diversos elementos do sistema começam a acusar um ao outro pelo problema. Em lugar de adotar essa postura sem sentido, você deve se antecipar aos problemas de interface em potencial e (1) criar caminhos de manipulação de erro que testem todas as informações vindas de outros elementos do sistema, (2) executar uma série de testes que simulem dados incorretos ou outros erros em potencial na interface de software, (3) registrar os resultados dos testes para usar como "evidência" se ocorrer a caça ao culpado e (4) participar do planejamento e projeto de testes do sistema para assegurar que o software seja testado adequadamente.

22.8.1 Teste de recuperação

Muitos sistemas de computador devem se recuperar de falhas e retomar o processamento em pouco ou nenhum tempo de parada. Em alguns casos, um sistema tem de ser tolerante a falhas; ou seja, falhas no processamento não devem causar a paralisação total do sistema. Em outros casos, uma falha no sistema deve ser corrigida dentro de determinado período de tempo; caso contrário, poderão ocorrer sérios prejuízos financeiros.

Teste de recuperação é um teste do sistema que obriga o software a falhar de várias formas e verifica se a recuperação é executada corretamente. Se a recuperação for automática (executada pelo próprio sistema), a reinicialização, os mecanismos de verificação, recuperação de dados e reinício são avaliados quanto à correção. Se a recuperação exige intervenção humana, o tempo médio de reparo (MTTR, mean-time-to-repair) é avaliado para determinar se está dentro dos limites aceitáveis.

22.8.2 Teste de segurança

Qualquer sistema de computador que trabalhe com informações sigilosas ou que cause ações que podem inadequadamente prejudicar (ou beneficiar) indivíduos é um alvo para acesso impróprio ou ilegal. As invasões abrangem uma ampla gama de atividades: hackers que tentam invadir sistemas por diversão, funcionários desgostosos que tentam invadir por vingança, indivíduos desonestos que tentam invadir para obter ganhos pessoais ilícitos.

O *teste de segurança* tenta verificar se os mecanismos de proteção incorporados ao sistema vão de fato protegê-lo contra acesso indevido. Citando Beizer [Bei84]: "A segurança de um sistema deve, naturalmente, ser testada

quanto à invulnerabilidade por um ataque frontal – mas deve também ser testada quanto à invulnerabilidade por ataques laterais ou pela retaguarda”.

Com tempo e recursos suficientes, um bom teste de segurança finalmente conseguirá invadir o sistema. O papel do criador do sistema é tornar o custo da invasão maior do que o valor das informações que poderiam ser obtidas. O teste de segurança e a engenharia de segurança são discutidos com mais detalhes no Capítulo 27.

22.8.3 Teste por esforço

As etapas anteriores de teste resultam em uma avaliação completa das funções normais do programa e do desempenho. Os testes por esforço (stress) servem para colocar os programas em situações anormais. Basicamente, o testador que executa teste por esforço pergunta: “Até onde podemos forçar o sistema até que ele falhe?”.

O *teste por esforço* usa um sistema de maneira que demande recursos em quantidade, frequência ou volumes anormais. Por exemplo, (1) testes especiais podem gerar dez interrupções por segundo, quando uma ou duas é a média normal, (2) a taxa de entrada de dados pode ser aumentada por certa magnitude para determinar como as funções de entrada responderão, (3) são executados casos de teste que exigem o máximo de memória ou outros recursos, (4) são executados casos de teste que podem causar problemas de memória em um sistema operacional virtual, (5) são criados casos de teste que podem causar excessiva procura por dados residentes em disco. Basicamente, o testador tenta quebrar o programa.

Uma variação do teste por esforço é uma técnica chamada *teste de sensibilidade*. Em algumas situações (as mais comuns ocorrem em algoritmos matemáticos), um intervalo muito pequeno de dados, contido dentro dos limites de dados válidos para um programa, pode causar um processamento extremo e até mesmo errôneo ou uma profunda degradação do desempenho. O teste de sensibilidade tenta descobrir combinações de dados dentro de classes de entrada válidas que possam causar instabilidade ou processamento inadequado.

22.8.4 Teste de desempenho

Para sistemas em tempo real e embutidos, um software que execute a função necessária, mas não esteja em conformidade com os requisitos de desempenho, é inaceitável. O teste de desempenho é projetado para testar o desempenho em tempo de execução do software dentro do contexto de um sistema integrado. O teste de desempenho é feito em todas as etapas no processo de teste. Até mesmo em nível de unidade, o desempenho de um módulo individual pode ser avaliado durante o teste. No entanto, o verdadeiro desempenho de um sistema só pode ser avaliado depois que todos os elementos do sistema estiverem totalmente integrados.

Os testes de desempenho muitas vezes são acoplados ao teste de esforço e usualmente exigem instrumentação de hardware e software. Isto é, frequentemente é necessário medir a utilização dos recursos (por exemplo, ciclos de processador) de forma precisa. Instrumentação externa pode monitorar in-

“Se você está tentando encontrar verdadeiros defeitos no sistema e ainda não submeteu o seu software a um verdadeiro teste de esforço, então é hora de começar a fazê-lo.”

Boris Beizer

tervalos de execução, log de eventos (por exemplo, interrupções) à medida que ocorrem e verificar os estados da máquina regularmente. Monitorando o sistema com instrumentos, o testador pode descobrir situações que levam à degradação e possível falha do sistema.

22.8.5 Teste de disponibilização

Em muitos casos, o software deve operar em uma variedade de plataformas e sob mais de um ambiente de sistema operacional. O *teste de disponibilização*, também chamado de *teste de configuração*, exercita o software em cada ambiente no qual ele deve operar. Além disso, o teste de disponibilização examina todos os procedimentos de instalação e software de instalação especializado (por exemplo, os “instaladores”) que serão usados pelos clientes e toda a documentação que será usada para fornecer o software aos usuários finais.

FERRAMENTAS DO SOFTWARE



Planejamento e gerenciamento do teste

Objetivo: essas ferramentas ajudam a equipe de software no planejamento da estratégia de teste escolhida e no gerenciamento do processo de teste enquanto ele é executado.

Mecanismos: as ferramentas nessa categoria cuidam do planejamento do teste, armazenamento do teste, gerenciamento e controle, rastreabilidade dos requisitos, integração, rastreamento de erros e geração de relatórios. Os gerentes de projeto usam essas ferramentas para suplementar as ferramentas de cronograma de projeto. Testadores usam essas ferramentas para planejar atividades de teste e para controlar o fluxo de informações à medida que o processo de teste avança.

Ferramentas representativas:⁵

QaTraQ Test Case Management Tool, desenvolvida pela TraQ Software (www.testmanagement.com), “sugere

uma abordagem estruturada para o gerenciamento do teste”.

QAComplete, desenvolvida pela SmartBear (<http://smartbear.com/products/qa-tools/test-management>), oferece um ponto de controle único para gerenciar todas as fases do processo de teste ágil.

TestWorks, desenvolvida pela Software Research (<http://www.testworks.com/>), contém um conjunto totalmente integrado de ferramentas de teste, incluindo ferramentas de gerenciamento de testes e relatórios.

OpensourceTesting.org (www.opensourcetesting.org/testmgt.php) lista uma variedade de ferramentas de código-fonte aberto para gerenciamento e planejamento de teste.

OpensourceTestManagement.com (<http://www.opensourcetestmanagement.com/>) lista uma variedade de ferramentas de código-fonte aberto para gerenciamento e planejamento de teste.

22.9 A arte da depuração

O teste de software é um processo que pode ser sistematicamente planejado e especificado. Casos de teste podem ser projetados, uma estratégia pode ser definida, e os resultados podem ser avaliados de acordo com as expectativas prescritas.

A *depuração* ocorre como consequência de um teste bem-sucedido. Isto é, quando um caso de teste descobre um erro, a depuração é o processo que

⁵ A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

resulta na remoção do erro. Embora a depuração possa e deva ser um processo ordenado, ela ainda é, em grande parte, uma arte. Um engenheiro de software, avaliando os resultados de um teste, é frequentemente confrontado com uma indicação sintomática de um problema de software. Isso significa que a manifestação externa do erro e sua causa interna podem não ter nenhuma relação óbvia uma com a outra. O processo mental mal compreendido que conecta um sintoma a uma causa é a depuração.

22.9.1 O processo de depuração

A depuração não é teste, mas frequentemente ocorre em consequência do teste.⁶ De acordo com a Figura 22.7, o processo de depuração começa com a execução de um caso de teste.

Os resultados são avaliados, e uma falta de correspondência entre o desempenho esperado e o desempenho real é encontrada. Em muitos casos, os dados não correspondentes são um sintoma de uma causa subjacente, embora oculta. O processo de depuração tenta combinar o sintoma com a causa, levando, assim, à correção do erro.

O processo de depuração usualmente apresentará um dentre dois resultados: (1) a causa será encontrada e corrigida ou (2) a causa não será encontrada. Neste último caso, quem está executando a depuração pode suspeitar de uma causa, criar um caso de teste para ajudar a confirmar aquela suspeita e trabalhar na correção do erro de forma iterativa.

"Assim que começamos a programar, descobrimos, para nossa surpresa, que não era tão fácil obter os programas certos como pensávamos. A depuração tinha de ser descoberta. Posso me lembrar do momento exato em que percebi que uma grande parte da minha vida desde então seria dedicada a encontrar os erros dos meus próprios programas."

**Maurice Wilkes
descobre a
depuração, 1949**

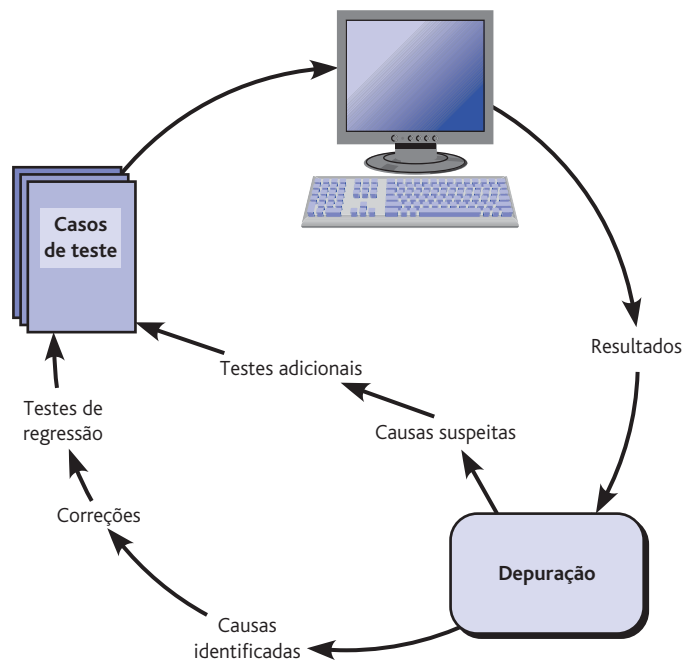


FIGURA 22.7 O processo de depuração.

⁶ Ao fazer essa afirmativa, assumimos a visão mais ampla possível do teste. Não só o desenvolvedor testa o software antes da entrega, mas também o cliente/usuário testa o software todas as vezes que ele é usado!

Por que a depuração é tão difícil?

"Todos sabem que a depuração é duas vezes mais difícil do que escrever um programa a partir do zero. Assim, se você foi o mais esperto possível ao escrever o programa, como poderá depurá-lo?"

Brian Kernighan

Por que a depuração é tão difícil? Com toda certeza, a psicologia humana (veja a Seção 22.9.2) tem muito mais a ver com a resposta do que a tecnologia de software. No entanto, algumas características dos erros fornecem alguns indícios:

1. O sintoma e a causa podem ser geograficamente remotos. Isto é, o sintoma pode aparecer em uma parte de um programa, enquanto a causa pode estar localizada em um ponto muito afastado. Componentes altamente acoplados (Capítulo 12) pioram essa situação.
2. O sintoma pode desaparecer (temporariamente) quando outro erro for corrigido.
3. O sintoma pode ser, na realidade, causado por coisas que não são erros (por exemplo, imprecisões de arredondamento).
4. O sintoma pode ser causado por erro humano que não é facilmente rastreado.
5. O sintoma pode ser resultado de problemas de temporização e não de problemas de processamento.
6. Pode ser difícil reproduzir as condições de entrada com precisão (por exemplo, uma aplicação em tempo real na qual a ordem das entradas é indeterminada).
7. O sintoma pode ser intermitente. Isso é particularmente comum em sistemas embarcados que acoplam hardware e software inextricavelmente.
8. O sintoma pode ocorrer devido a causas distribuídas por várias tarefas, executando em diferentes processadores.

Durante a depuração, encontramos erros que variam desde levemente desagradáveis (por exemplo, um formato de saída incorreto) até catastróficos (por exemplo, o sistema falha, causando sérios danos econômicos ou físicos). À medida que as consequências de um erro aumentam, a pressão para encontrá-lo também aumenta. Às vezes, a pressão obriga um desenvolvedor de software a corrigir um erro e ao mesmo tempo introduzir outros dois.

Defina um limite – digamos, duas horas – para o tempo que você passa tentando depurar um problema por conta própria. Após esse tempo, procure ajuda!

22.9.2 Considerações psicológicas

Infelizmente, parece haver certa evidência de que a destreza na depuração é uma peculiaridade humana inata. Algumas pessoas são boas nisso e outras não. Embora a evidência experimental em depuração seja aberta a muitas interpretações, é possível observar uma grande variação na habilidade de depuração entre programadores com o mesmo nível de formação e experiência. Embora possa ser difícil “aprender” a depurar, várias abordagens do problema podem ser propostas. Vamos examiná-las na Seção 22.9.3.

22.9.3 Estratégias de depuração

Independentemente da abordagem adotada, a depuração tem um objetivo primordial: encontrar e corrigir a causa de um erro ou defeito de software. O objetivo é alcançado por uma combinação de avaliação sistemática, intuição e sorte.

Em geral, foram propostas três estratégias de depuração [Mye79]: força bruta, rastreamento e eliminação da causa. Cada uma dessas estratégias pode

CASASEGURA

**Depuração**

Cena: Sala de Ed enquanto é executado o teste de código e unidade.

Atores: Ed e Shakira – membros da equipe de engenharia de software do CasaSegura.

Conversa:

Shakira (espiando pela entrada do cubículo): Olá... onde você estava na hora do almoço?

Ed: Aqui mesmo... trabalhando.

Shakira: Você parece tão mal... o que houve?

Ed (suspirando alto): Estou trabalhando neste erro desde que o descobri às 9:30 desta manhã, e já são 14:45... até agora nada.

Shakira: Pensei que havíamos combinado que não devemos gastar mais de uma hora com problemas de depuração sozinho; depois disso, devemos procurar ajuda, certo?

Ed: Sim, mas...

Shakira (entrando no cubículo): Qual é o problema?

Ed: É complicado e, além disso, estou pesquisando há 5 horas. Você não vai poder resolvê-lo em 5 minutos.

Shakira: Desculpe-me... qual é o problema?

[Ed explica o problema para Shakira, que o examina por 30 segundos sem dizer nada, então...]

Shakira (com um sorriso): Já sei! Bem aqui, a variável denominada *setAlarmCondition*. Ela não deveria ser definida como "false" antes de iniciar o laço?

[Ed olha a tela incrédulo, inclina-se para frente e começa a bater com a cabeça no monitor. Shakira, rindo muito, se levanta e sai.]

ser conduzida manualmente, mas as modernas ferramentas de depuração podem tornar o processo muito mais eficaz.

Táticas de depuração. A categoria *força bruta* para depuração é provavelmente o método mais comum e menos eficiente para isolar a causa de um erro de software. Usamos métodos de depuração do tipo força bruta quando todo o resto falha. Usando uma filosofia do tipo "deixe o computador encontrar o erro", ocorrem despejos de memória, rastreamentos em tempo de execução, e o programa é sobrecarregado com instruções de saída. Espera-se que, no meio de toda aquela confusão de informações produzidas, consigamos encontrar um indício que possa levar à causa de um erro. Embora a grande quantidade de informações produzidas possa, no fim, levar ao sucesso, mais frequentemente é uma perda de tempo e trabalho. É preciso pensar primeiro!

Rastreamento (backtracking) é uma abordagem comum de depuração que pode ser usada com sucesso em programas pequenos. Começando no ponto onde o sintoma foi descoberto, o código-fonte é investigado retroativamente (manualmente) até que a causa seja encontrada. Infelizmente, à medida que o número de linhas de código-fonte aumenta, o número de caminhos retroativos em potencial pode se tornar demasiadamente grande.

A terceira abordagem de depuração – *eliminação da causa* – é manifestada por indução ou dedução e introduz o conceito de particionamento binário. Os dados relacionados à ocorrência do erro são organizados para isolar as possíveis causas. É imaginada uma "hipótese de causa", e os dados mencionados anteriormente são usados para provar ou negar a hipótese. Como alternativa, é preparada uma lista de todas as causas possíveis, e são realizados testes para eliminar cada uma delas. Se os testes iniciais indicam que determinada causa hipotética promete resultado, os dados são refinados tentando isolar o defeito.

"O primeiro passo para reparar um programa com problema é fazê-lo falhar repetidamente (no exemplo mais simples possível)."

T. Duff

Depuração automática. Cada um desses métodos de depuração pode ser complementado com ferramentas de depuração que podem fornecer suporte semiautomático para o engenheiro de software à medida que são tentadas as estratégias. Hailpern e Santhanam [Hai02] resumem o estado dessas ferramentas ao mencionarem: “... foram propostas muitas abordagens novas e estão disponíveis muitos ambientes de depuração comerciais. Ambientes Integrados de Desenvolvimento (IDE) proporcionam uma maneira de capturar alguns dos erros predeterminados, específicos da linguagem (por exemplo, falta de caracteres de fim de instrução, variáveis indefinidas, e assim por diante), sem exigir compilação”. Há disponível uma ampla variedade de compiladores de depuração, auxílios dinâmicos de depuração (“rastreadores”), geradores de casos de teste automáticos e ferramentas de mapeamento de referências cruzadas. No entanto, as ferramentas não substituem uma avaliação cuidadosa, fundamentada em um modelo completo de projeto e um código-fonte claro.

O fator humano. Qualquer discussão sobre abordagens e ferramentas de depuração é incompleta se não mencionar um poderoso aliado: outras pessoas! Um ponto de vista novo, não ofuscado por horas de frustração, pode fazer maravilhas.⁷ Uma máxima final para a depuração seria: “Quando tudo mais falhar, procure ajuda!”.

FERRAMENTAS DO SOFTWARE



Depuração

Objetivo: essas ferramentas proporcionam auxílio automático para aqueles que devem depurar problemas de software. A intenção é fornecer informações que podem ser difíceis de obter se o processo de depuração for abordado manualmente.

Mecanismos: muitas ferramentas de depuração são específicas quanto à linguagem de programação e quanto ao ambiente.

Ferramentas representativas:⁸

Borland Silk, distribuída pela Borland (<http://www.borland.com/products/>), ajuda tanto no teste quanto na depuração.

Coverty Development Testing Platform, desenvolvida pela Coverty (<http://www.coverity.com/products/>), oferece

um modo de introduzir testes de qualidade e segurança no início do processo de desenvolvimento.

C++Test, desenvolvida pela Parasoft (www.parasoft.com), é uma ferramenta de teste de unidade que suporta uma ampla variedade de testes em código C e C++. Recursos de depuração ajudam no diagnóstico de erros encontrados.

CodeMedic, desenvolvida pela NewPlanet Software (www.newplanetsoftware.com/medic/), proporciona uma interface gráfica para o depurador UNIX padrão, *gdb*, e implementa suas características mais importantes. Atualmente, o *gdb* suporta C/C++, Java, PalmOS, vários sistemas embarcados, linguagem Assembly, FORTRAN e Modula-2.

GNATS, um aplicativo *freeware* (www.gnu.org/software/gnats/), é um conjunto de ferramentas para rastrear relatórios de erro.

⁷ O conceito de programação em pares (recomendado como parte do modelo Extreme Programming discutido no Capítulo 5) proporciona um mecanismo para “depuração”, à medida que o software é projetado e codificado.

⁸ A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

22.9.4 Correção do erro

Uma vez encontrado um erro, ele precisa ser corrigido. Entretanto, conforme já observamos, a correção de um erro pode introduzir outros erros e, portanto, causar mais danos do que trazer benefícios. Van Vleck [Van89] sugere três perguntas simples que você deve fazer antes de fazer a “correção” que remove a causa de um erro:

1. *A causa do defeito é reproduzida em alguma outra parte do programa?*
Em muitas situações, o defeito de um programa é causado por um padrão incorreto de lógica que pode ser reproduzido em qualquer outro ponto. Uma consideração explícita do padrão lógico pode resultar na descoberta de outros erros.
2. *Qual “próximo erro” poderia ser introduzido pela correção que estou prestes a fazer?* Antes de fazer a correção, o código-fonte (ou, melhor, o projeto) deve ser avaliado para verificar acoplamento de lógica e estruturas de dados. Se a correção tem de ser feita em uma parte do programa altamente acoplada, deve ser tomado cuidado especial ao fazer qualquer alteração.
3. *O que poderíamos ter feito para evitar esse defeito já no início?* Essa pergunta é o primeiro passo para o estabelecimento de uma abordagem estatística de garantia da qualidade de software (Capítulo 21). Se você corrigir o processo juntamente com o produto, o erro será removido do programa em questão e poderá ser eliminado em todos os programas futuros.

“O melhor testador não é aquele que encontra o maior número de erros... O melhor testador é aquele que consegue corrigir o maior número de erros.”

Cem Kaner et al.

22.10 Resumo

O teste de software absorve a maior parte do esforço técnico em um processo de software. Independentemente do tipo de software criado, uma estratégia para planejamento sistemático de teste, execução e controle começa considerando pequenos elementos do software e se encaminha para fora no sentido de abranger o programa como um todo.

O objetivo do teste de software é descobrir erros. Para o software convencional, esse objetivo é atingido com uma série de etapas de teste. Testes de unidade e de integração concentram-se na verificação funcional de um componente e na incorporação dos componentes na arquitetura de software. O teste de validação demonstra os requisitos de rastreabilidade do software e o teste de sistema aprova o software quando ele é incorporado em um sistema maior. Cada etapa de teste é realizada com uma série de técnicas sistemáticas que auxiliam no projeto dos casos de teste. Em cada etapa, o nível de abstração com o qual o software é considerado é ampliado.

A estratégia para teste de software OO começa com testes que exercitam as operações dentro de uma classe e depois passa para o teste baseado em sequências de execução para integração. Sequências de execução são conjuntos de classes que respondem a uma entrada ou evento. Testes baseados em uso focam classes que não colaboram intensamente com outras classes.

WebApps e aplicativos móveis são testados de maneira muito semelhante aos sistemas OO. No entanto, os testes são projetados para exercitar conteúdo, funcionalidade, interface, navegação e outros aspectos de desempenho

e segurança da aplicação. Os aplicativos móveis exigem abordagens de teste especializadas, as quais se concentram no teste do aplicativo em vários dispositivos e em ambientes de rede reais.

Diferentemente do teste (uma atividade sistemática, planejada), a depuração pode ser vista como uma arte. Começando com a indicação sintomática de um problema, a atividade de depuração deve rastrear a causa de um erro. Dentre os vários recursos disponíveis durante a depuração, o mais valioso é o conselho de outros membros da equipe de engenharia de software.

Problemas e pontos a ponderar

22.1 Usando as suas próprias palavras, descreva a diferença entre verificação e validação. Ambas usam métodos de projeto de caso de teste e estratégias de teste?

22.2 Liste alguns dos problemas que podem ser associados à criação de um grupo de teste independente. Um grupo ITG e um grupo SQA são formados pelas mesmas pessoas?

22.3 É sempre possível desenvolver uma estratégia para teste de software que use a sequência de etapas de teste descrita na Seção 22.1.3? Que possíveis complicações podem surgir para sistemas embarcados?

22.4 Por que um módulo altamente acoplado é difícil de testar em unidade?

22.5 O conceito de antidefeito (antibugging) (Seção 22.2.1) é uma maneira extremamente eficaz de proporcionar auxílio de depuração interno quando um erro é descoberto:

- a. Desenvolva uma série de diretrizes para antidepuração.
- b. Discuta as vantagens do uso da técnica.
- c. Discuta as desvantagens.

22.6 Como o cronograma de projeto pode afetar o teste de integração?

22.7 O teste de unidade é possível ou até mesmo desejável em todas as circunstâncias? Dê exemplos para justificar a sua resposta.

22.8 Quem deve executar o teste de validação – o desenvolvedor do software ou o usuário do software? Justifique a sua resposta.

22.9 Desenvolva uma estratégia de teste completa para o sistema *CasaSegura* discutido anteriormente neste livro. Documente-a em uma *Especificação de Teste*.

22.10 Como projeto de classe, desenvolva um *Guia de Depuração* para a sua instalação. O guia deve fornecer informações orientadas à linguagem e ao sistema que você tenha aprendido na escola da vida! Comece com um esboço dos tópicos que serão revisados pela classe e pelo seu professor. Distribua o guia para os outros no seu ambiente local.

Leituras e fontes de informação complementares

Praticamente todos os livros sobre teste de software discutem estratégias juntamente com métodos para projeto de casos de teste. Whittaker (*How Google Tests Software*, Addison-Wesley, 2012; e *How to Break Software*, Addison-Wesley, 2002), Spiller e seus colegas (*Software Testing Foundations*, Rocky Nook, 2011), Black (*Managing the Testing*, 3ª ed., Wiley, 2009) e (*Pragmatic Software Testing*, Wiley, 2007), Page e seus colegas (*How We Test Software at Microsoft*, Microsoft Press, 2008), Lewis (*Software Testing and Continuous Quality Improvement*, 3ª ed., Auerbach, 2008), Everett e Raymond (*Software Testing*, Wiley-IEEE Computer Society Press, 2007), Perry (*Effective Methods for Soft-*

ware Testing, 3ª ed., Wiley, 2005), Loveland e seus colegas (*Software Testing Techniques*, Charles River Media, 2004), Burnstein (*Practical Software Testing*, Springer, 2003), Dustin (*Effective Software Testing*, Addison-Wesley, 2002), Craig e Kaskiel (*Systematic Software Testing*, Artech House, 2002) e Tamres (*Introducing Software Testing*, Addison-Wesley, 2002) são apenas uma pequena amostra dos muitos livros que discutem princípios, conceitos, estratégias e métodos de teste.

Para os leitores interessados nos métodos ágeis de desenvolvimento de software, Gartner e Gartner (*ATDD by Example: A Practical Guide to Acceptance Test-Driven Development*, Addison-Wesley, 2012), Crispin e Gregory (*Agile Testing: A Practical Guide for Testers and Teams*, Addison-Wesley, 2009), Crispin e House (*Testing Extreme Programming*, Addison-Wesley, 2002) e Beck (*Test Driven Development: By Example*, Addison-Wesley, 2002) apresentam estratégias e táticas de teste para Extreme Programming. Kamer e seus colegas (*Lessons Learned in Software Testing*, Wiley, 2001) apresentam uma coleção de mais de 300 “lições” (diretrizes) pragmáticas que todo testador de software deve conhecer. Watkins (*Testing IT: An Off-the-Shelf Testing Process*, 2ª ed., Cambridge University Press, 2010) estabelece um framework de teste eficaz para todos os tipos de software desenvolvido e adquirido. Manges e O'Brien (*Agile Testing with Ruby and Rails*, Apress, 2008) tratam das estratégias e técnicas de teste para a linguagem de programação Ruby e framework para a Internet.

Bashir e Goel (*Testing Object-Oriented Software*, Springer-Verlag, 2012), Sykes e McGregor (*Practical Guide to Testing Object-Oriented Software*, Addison-Wesley, 2001), Binder (*Testing Object-Oriented Systems*, Addison-Wesley, 1999), Kung e seus colegas (*Testing Object-Oriented Software*, IEEE Computer Society Press, 1998) e Marick (*The Craft of Software Testing*, Prentice Hall, 1997) apresentam estratégias e métodos para teste de sistemas orientados a objetos.

Diretrizes para depuração podem ser encontradas nos livros de Grötter e seus colegas (*The Developer's Guide to Debugging*, 2ª ed., CreateSpace Independent Publishing, 2012), Whittaker (*Exploratory Testing*, Addison-Wesley, 2009), Zeller (*Why Programs Fail: A Guide to Systematic Debugging*, 2ª ed., Morgan Kaufmann, 2009), Butcher (*Debug It*, Pragmatic Bookshelf, 2009), Agans (*Debugging*, Amacon, 2006) e Tells e Hsieh (*The Science of Debugging*, The Coreolis Group, 2001). Kaspersky (*Hacker Debugging Uncovered*, A-List Publishing, 2005) trata da tecnologia de ferramentas de depuração. Younessi (*Object-Oriented Defect Management of Software*, Prentice Hall, 2002) apresenta técnicas para cuidar de defeitos encontrados em sistemas orientados a objetos. Beizer [Bei84] apresenta uma interessante “sistemática de defeitos” que pode levar a métodos eficazes para planejamento de teste.

Livros de Graham e Fewster (*Experiences of Test Automation*, Addison-Wesley, 2012) e Dustin e seus colegas (*Implementing Automated Software Testing*, Addison-Wesley, 2009) discutem os testes automatizados. Livros de Hunt e John (*Java Performance*, Addison-Wesley, 2011), Hewardt e seus colegas (*Advanced.NET Debugging*, Addison-Wesley, 2009), Matloff e seus colegas (*The Art of Debugging with GDB, DDD, and Eclipse*, No Starch Press, 2008), Madisetti e Akgul (*Debugging Embedded Systems*, Springer, 2007), Robbins (*Debugging Microsoft.NET 2.0 Applications*, Microsoft Press, 2005), Best (*Linux Debugging and Performance Tuning*, Prentice Hall, 2005), Ford e Teorey (*Practical Debugging in C++*, Prentice Hall, 2002), Brown (*Debugging Perl*, McGraw-Hill, 2000) e Mitchell (*Debugging Java*, McGraw-Hill, 2000) tratam da natureza especial da depuração para os ambientes indicados em seus títulos.

Uma ampla gama de fontes de informação sobre métodos de teste orientado a objetos está disponível na Internet. Uma lista atualizada de referências relevantes para as estratégias de teste de software (em inglês) pode ser encontrada no site: www.mhhe.com/pressman.

23 Teste de aplicativos convencionais

Conceitos-chave

análise de valor limite ..	512
complexidade	
ciclomática	503
grafos de fluxo.....	500
matrizes de grafos.....	506
métodos de teste	
baseados em grafos	509
padrões.....	519
particionamento	
de equivalência.....	511
teste baseado	
em modelo	516
teste caixa-branca	500
teste caixa-preta.....	509

O teste apresenta um dilema interessante para os engenheiros de software, que são, por natureza, pessoas construtivas. Ele requer que o desenvolvedor descarte noções preconcebidas da “corretividade” do software recém-desenvolvido e passe a trabalhar arduamente projetando casos de teste para “quebrar” o software. Beizer [Bei90] descreve muito bem essa situação quando declara:

Há um mito de que, se fôssemos realmente bons em programação, não precisaríamos caçar erros. Se pudéssemos realmente nos concentrar, se todos usassem programação estruturada, projeto com detalhamento progressivo... Então não haveria erros. E assim continua o mito. Existem erros, diz o mito, porque somos ruins no que fazemos; e se somos ruins no que fazemos, devemos nos sentir culpados por isso. Portanto, o teste e o planejamento de casos de teste é um

PANORAMA

tantos erros quanto possível antes de ser fornecido ao cliente. Sua meta é projetar um conjunto de casos de teste que tenha a mais alta probabilidade de encontrar erros – mas como? É aqui que entram em cena as técnicas de teste de software. Essas técnicas fornecem diretrizes sistemáticas para projetar testes que (1) exercitam a lógica interna e as interfaces de todos os componentes do software e (2) exercitam os domínios de entrada e saída do programa para descobrir erros no funcionamento, comportamento e desempenho do programa.

Quem realiza? Durante os primeiros estágios do teste, um engenheiro de software executa todos os testes. Porém, à medida que o processo de teste avança, especialistas podem ser envolvidos.

Por que é importante? Revisões e outras atividades de SQA podem descobrir (e realmente descobrem) erros, mas não são suficientes. Toda vez que o programa é executado, o cliente o testa! Portanto, você tem de executar o programa com o objetivo específico de encontrar e remover todos os erros antes que ele chegue ao cliente. Para encontrar o maior número possível de erros, devem ser executados testes siste-

maticamente, e os casos de teste devem ser projetados usando técnicas disciplinadas.

Quais são as etapas envolvidas? Para aplicações convencionais, o software é testado a partir de duas perspectivas diferentes: (1) a lógica interna do programa é exercitada usando técnicas de projeto de caso de teste “caixa-branca”, e (2) os requisitos de software são exercitados usando técnicas de projeto de casos de teste “caixa-preta”. Casos de teste de uso ajudam no projeto de testes para descobrir erros no nível de validação do software. Em todos os casos, a intenção é encontrar o número máximo de erros com o mínimo de esforço e tempo.

Qual é o artefato? Um conjunto de casos de teste projetados para exercitar a lógica interna, interfaces, colaborações entre componentes e os requisitos externos é projetado e documentado, os resultados esperados são definidos, e os resultados obtidos são registrados.

Como garantir que o trabalho foi realizado corretamente? Quando você começar o teste, mude o seu ponto de vista. Tente “quebrar” o software! Projete casos de teste de forma disciplinada e reveja os casos de teste que você criou, quanto à perfeição. Além disso, você pode avaliar a abrangência do teste e monitorar as atividades de detecção de erros.

reconhecimento de falha que sugere uma boa dose de culpa. E o tédio do teste é exatamente a punição pelos nossos erros. Punição por quê? Por sermos humanos? Culpados de quê? De não conseguirmos atingir a perfeição desumana? De não distinguirmos entre o que outro programador pensa e o que ele diz? Por não conseguirmos ser telepáticos? Por não resolvermos problemas de comunicação humana que já existem... há quarenta séculos?

teste de estrutura de controle.....	507
teste de matriz ortogonal.....	513
teste do caminho básico.....	500

O teste deve realmente insinuar culpa? O teste é realmente destrutivo? A resposta a essas questões é “não!”.

Neste capítulo, discutimos técnicas para projetar casos de teste de software para aplicações convencionais. O projeto de casos de teste foca um conjunto de técnicas para a criação de casos de teste, as quais satisfazem os objetivos globais e as estratégias de teste discutidas no Capítulo 22.

23.1 Fundamentos do teste de software

O objetivo do teste é encontrar erros, e um bom teste é aquele que tem alta probabilidade de encontrar um erro. Portanto, um engenheiro de software deve projetar e implementar um sistema ou produto baseado em computador tendo em mente a “testabilidade”. Ao mesmo tempo, os próprios testes devem ter uma série de características que permitam atingir o objetivo de encontrar o maior número de erros com o mínimo de esforço.

“Todo programa faz alguma coisa certa, só que pode não ser aquilo que queremos que ele faça.”

Autor desconhecido

Testabilidade. James Bach¹ dá a seguinte definição para testabilidade: “*Testabilidade de software* é simplesmente a facilidade com que um programa de computador pode ser testado”. As seguintes características levam a um software testável:

Operabilidade. “Quanto melhor funcionar, mais eficientemente pode ser testado”. Se um sistema for projetado e implementado tendo em mente a qualidade, haverá poucos defeitos bloqueando a execução dos testes, permitindo que o teste ocorra sem sobressaltos.

Observabilidade. “O que você vê é o que você testa”. Entradas fornecidas como parte do teste produzem saídas distintas. Estados e variáveis do sistema são visíveis ou podem ser consultados durante a execução. Saída incorreta é facilmente identificada. Erros internos são automaticamente detectados e relatados. O código-fonte é acessível.

Quais são as características da testabilidade?

Controlabilidade. “Quanto melhor pudermos controlar o software, mais o teste pode ser automatizado e otimizado”. Todas as possíveis saídas podem ser geradas por meio de alguma combinação de entrada, e os formatos de entrada e saída são consistentes e estruturados. Todo código é executável por meio de alguma combinação de entrada. Estados e variáveis de software e hardware podem ser controlados diretamente pelo engenheiro

¹ Os parágrafos seguintes são usados com permissão de James Bach (©1994) e foram adaptados a partir de material que originalmente apareceu em uma postagem no newsgroup comp.software-eng.

de teste. Os testes podem ser convenientemente especificados, automatizados e reproduzidos.

Decomponibilidade. “Controlando o escopo do teste, podemos isolar problemas mais rapidamente e testar de forma mais racional”. O sistema de software é construído a partir de módulos independentes que podem ser testados de forma independente.

Simplicidade. “Quanto menos tivermos que testar, mais rapidamente podemos testá-lo”. O programa deve ter *simplicidade funcional* (por exemplo, o conjunto de características é o mínimo necessário para satisfazer os requisitos), *simplicidade estrutural* (por exemplo, a arquitetura é modularizada para limitar a propagação de falhas) e *simplicidade de código* (por exemplo, é adotado um padrão de codificação para facilitar a inspeção e a manutenção).

“Erros são mais comuns, mais disseminados e mais problemáticos no software do que em outras tecnologias.”

David Parnas

Estabilidade. “Quanto menos alterações, menos interrupções no teste”. As alterações no software são pouco frequentes, controladas quando ocorrem e não invalidam os testes existentes. O software recupera-se bem das falhas.

Compreensibilidade. “Quanto mais informações tivermos, mais inteligente será o teste”. O projeto arquitetural e as dependências entre componentes internos, externos e compartilhados são bem compreendidas. A documentação técnica é instantaneamente acessível, bem organizada, específica, detalhada e precisa. Alterações no projeto são comunicadas aos testadores.

Os atributos sugeridos por Bach podem ser utilizados para se desenvolver um artefato de software sensível ao teste.

Características do teste. E quanto aos próprios testes? Kaner, Falk e Nguyen [Kan93] sugerem os seguintes atributos para um “bom” teste:

O que é um teste “bom”?

Um bom teste tem alta probabilidade de encontrar um erro. Para atingir esse objetivo, o testador deve entender o software e tentar desenvolver uma imagem mental de como ele pode falhar.

Um bom teste não é redundante. O tempo e os recursos de teste são limitados. Não faz sentido realizar um teste que tenha a mesma finalidade de outro teste. Cada teste deve ter uma finalidade diferente (mesmo que seja sutilmente diferente).

Um bom teste deve ser “o melhor da raça” [Kan93]. Em um grupo de testes com finalidades similares, as limitações de tempo e recursos podem induzir à execução de apenas um subconjunto dos testes que tenha a maior probabilidade de revelar uma classe inteira de erros.

Um bom teste não deve ser nem muito simples nem muito complexo. Embora algumas vezes seja possível combinar uma série de testes em um caso de teste, os possíveis efeitos colaterais associados a essa abordagem podem mascarar erros. Em geral, cada teste deve ser executado separadamente.

CASASEGURA

**Projetando testes únicos****Cena:** Sala de Vinod.**Atores:** Vinod e Ed – membros da equipe de engenharia de software do CasaSegura.**Conversa:****Vinod:** Então, esses são os casos de teste que você pretende usar para a operação *validaçãoDeSenha*.**Ed:** Sim, eles devem abranger muito bem todas as possibilidades para os tipos de senhas que um usuário possa digitar.**Vinod:** Então, vamos ver... você notou que a senha correta será 8080, certo?**Ed:** Certo.**Vinod:** E você especifica as senhas 1234 e 6789 para testar o erro no reconhecimento de senhas inválidas?**Ed:** Certo, e também testo senhas semelhantes à senha correta, veja... 8081 e 8180.**Vinod:** Essas parecem OK, mas eu não vejo muito sentido em testar 1234 e 6789. Elas são redundantes... testam a mesma coisa, não é isso?**Ed:** Bem, são valores diferentes.**Vinod:** Verdade, mas se 1234 não revela um erro... em outras palavras... a operação *validacaoDeSenha* nota que essa é uma senha inválida, é improvável que 6789 nos mostre algo novo.**Ed:** Entendo o que você quer dizer.**Vinod:** Não estou tentando ser exigente... É que nós temos um tempo limitado para testar, portanto é uma boa ideia executar testes que tenham alta possibilidade de encontrar erros novos.**Ed:** Sem problemas... Vou pensar um pouco mais nisso.

23.2 Visões interna e externa do teste

Qualquer produto de engenharia (e muitas outras coisas) pode ser testado por uma de duas maneiras: (1) conhecendo-se a função especificada para o qual um produto foi projetado para realizar, podem ser feitos testes que demonstrem que cada uma das funções é totalmente operacional, embora ao mesmo tempo procurem erros em cada função; (2) conhecendo-se o funcionamento interno de um produto, podem ser realizados testes para garantir que “tudo se encaixa” – isto é, que as operações internas foram realizadas de acordo com as especificações e que todos os componentes internos foram adequadamente exercitados. A primeira abordagem de teste usa uma visão externa e é chamada de teste caixa-preta. A segunda estratégia exige uma visão interna e é chamada de teste caixa-branca.²

O *teste caixa-preta* faz referência a testes realizados na interface do software. Um teste caixa-preta examina alguns aspectos fundamentais de um sistema, com pouca preocupação em relação à estrutura lógica interna do software. O *teste caixa-branca* fundamenta-se em um exame rigoroso do detalhe procedural. Os caminhos lógicos do software e as colaborações entre componentes são testados exercitando conjuntos específicos de condições e/ou ciclos.

À primeira vista poderia parecer que um teste caixa-branca realizado de forma rigorosa resultaria em “programas 100% corretos”. Tudo o que seria preciso fazer seria definir todos os caminhos lógicos, desenvolver casos de teste para exercitá-los e avaliar os resultados, ou seja, gerar casos de teste para

“Há apenas uma regra no projeto de casos de teste: abranger todas as características; porém, não faça muitos casos de teste.”

Tsuneo Yamaura

Testes caixa-branca só podem ser projetados depois que o projeto no nível de componente (ou código-fonte) existir. Os detalhes lógicos do programa devem estar disponíveis.

² Os termos *teste funcional* e *teste estrutural* às vezes são usados em lugar de teste caixa-preta e teste caixa-branca, respectivamente.

exercitar a lógica do programa de forma exaustiva. Infelizmente, o teste exaustivo apresenta certos problemas logísticos. Mesmo para programas pequenos, o número de caminhos lógicos possíveis pode ser muito grande. No entanto, o teste caixa-branca não deve ser descartado como impraticável. Um número limitado de caminhos lógicos importantes pode ser selecionado e exercitado. A validade de estruturas de dados importantes pode ser investigada.

INFORMAÇÕES



Teste exaustivo

Considere um programa de 100 linhas em linguagem C. Após algumas declarações básicas de dados, o programa contém dois laços aninhados que executam de 1 a 20 vezes cada um, dependendo das condições especificadas na entrada. Dentro do ciclo, são necessárias 4 construções se-então-senão (*if-then-else*). Há aproximadamente 10^{14} caminhos possíveis que podem ser executados nesse programa!

Para colocar esse número sob perspectiva, vamos supor que um processador de teste mágico ("mágico" porque não existe tal processador) tenha sido desenvolvido para teste exaustivo. O processador pode desenvolver um caso de teste, executá-lo e avaliar os resultados em um milissegundo. Trabalhando 24 horas por dia, 365 dias por ano, o processador gastaria 3.170 anos para testar o programa. Isso, sem dúvida, tumultuaria qualquer cronograma de desenvolvimento.

Portanto, pode-se afirmar que o teste exaustivo é impossível para grandes sistemas de software.

23.3 Teste caixa-branca

"Os erros ficam à espreita nas esquinas e se reúnem nas fronteiras."

Boris Beizer

O teste caixa-branca, também chamado de *teste da caixa-de-vidro* ou *teste estrutural*, é uma filosofia de projeto de casos de teste que usa a estrutura de controle descrita como parte do projeto no nível de componentes para derivar casos de teste. Usando métodos de teste caixa-branca, o engenheiro de software pode criar casos de teste que (1) garantam que todos os caminhos independentes de um módulo foram exercitados pelo menos uma vez, (2) exercitem todas as decisões lógicas nos seus estados verdadeiro e falso, (3) executem todos os ciclos em seus limites e dentro de suas fronteiras operacionais e (4) exercitem estruturas de dados internas para assegurar a sua validade.

23.4 Teste do caminho básico

O teste de caminho básico é uma técnica de teste caixa-branca proposta por Tom McCabe [McC76]. O teste de caminho básico permite ao projetista de casos de teste derivar uma medida da complexidade lógica de um projeto procedural e usar essa medida como guia para definir um conjunto base de caminhos de execução. Casos de teste criados para exercitar o conjunto base executam com certeza todas as instruções de um programa pelo menos uma vez durante o teste.

23.4.1 Notação de grafo de fluxo

Antes de apresentarmos o método do caminho básico, deve ser introduzida uma notação simples para a representação do fluxo de controle, chamada

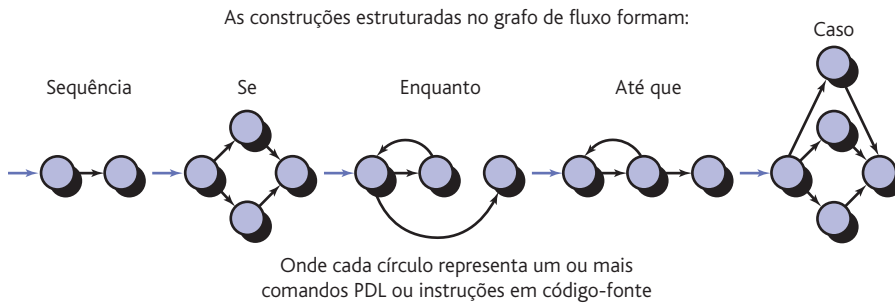


FIGURA 23.1 Notação de grafo de fluxo.

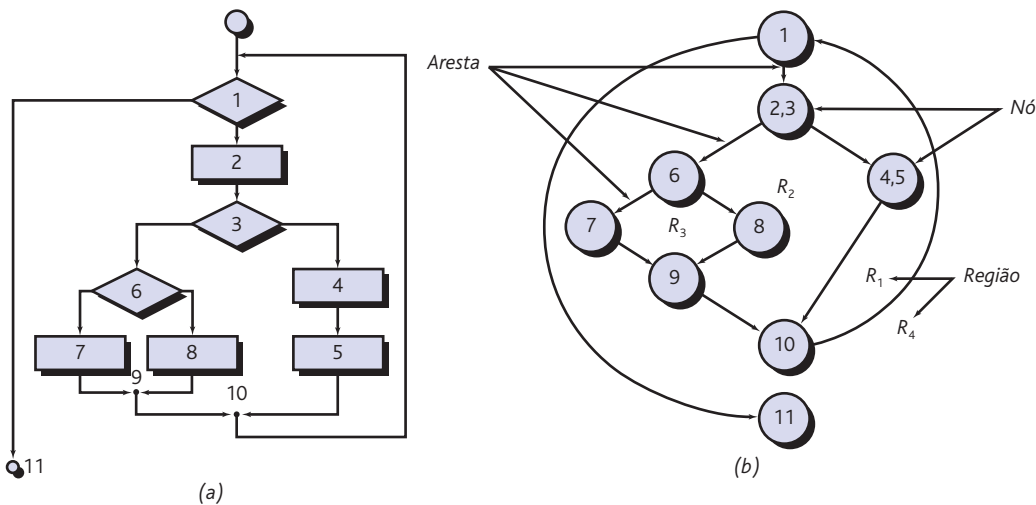


FIGURA 23.2 (a) Fluxograma e (b) grafo de fluxo.

de *grafo de fluxo* (ou *grafo de programa*).³ O grafo de fluxo representa o fluxo de controle lógico usando a notação ilustrada na Figura 23.1. Cada construção estruturada (Capítulo 14) tem um símbolo correspondente no grafo de fluxo.

Para ilustrar o uso de um grafo de fluxo, considere a representação do projeto procedimental da Figura 23.2a. É usado um fluxograma para mostrar a estrutura de controle do programa. A Figura 23.2b mapeia o fluxograma em um grafo de fluxo correspondente (considerando que os losangos de decisão do fluxograma não contêm nenhuma condição composta). Na Figura 23.2b, cada círculo, chamado de *nó do grafo de fluxo*, representa um ou mais comandos procedurais. Uma sequência de retângulos de processamento e um losango de decisão podem ser mapeados em um único nó. As setas no grafo de fluxo, chamadas de *arestas* ou *ligações*, representam fluxo de controle e são análogas às setas do fluxograma. Uma aresta deve terminar em um nó, mesmo que esse nó não represente qualquer comando procedural (por exemplo, veja o símbolo do diagrama de fluxo para a construção se-então-senão – *if-then-*

Um grafo de fluxo somente deve ser desenhado quando a estrutura lógica de um componente for complexa. O grafo de fluxo permite seguir mais facilmente os caminhos de um programa.

³ Na realidade, o método do caminho básico pode ser executado sem o uso de grafos de fluxo. No entanto, eles servem como uma notação útil para entender o fluxo de controle e ilustrar a abordagem.

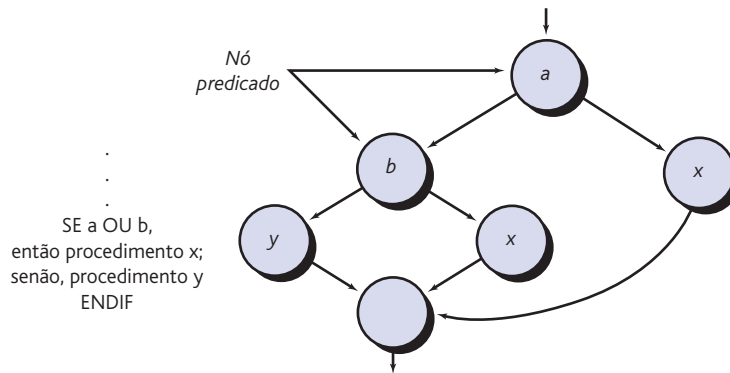


FIGURA 23.3 Lógica composta.

-else). As áreas limitadas por arestas e nós são chamadas de *regiões*. Ao contarmos as regiões, incluímos a área fora do grafo como uma região.⁴

Quando condições compostas são encontradas em um projeto procedimental, a geração de um grafo de fluxo torna-se ligeiramente mais complicada. Uma condição composta ocorre quando um ou mais operadores booleanos (OR, AND, NAND, NOR lógicos) estão presentes em um comando condicional. De acordo com a Figura 23.3, o trecho em PDL (*program design language*) é traduzido no grafo de fluxo mostrado. Note que é criado um nó separado para cada uma das condições *a* e *b* no comando SE *a* OU *b*. Cada nó contendo uma condição é chamado de *nó predicado* (*predicate node*) e é caracterizado por duas ou mais arestas saindo dele.

23.4.2 Caminhos de programa independentes

Um *caminho independente* é qualquer caminho através do programa que introduz pelo menos um novo conjunto de comandos de processamento ou uma nova condição. Quando definido em termos de um grafo de fluxo, um caminho independente deve incluir pelo menos uma aresta que não tenha sido atravessada antes de o caminho ser definido. Por exemplo, um conjunto de caminhos independentes para o grafo de fluxo ilustrado na Figura 23.2b é

Caminho 1: 1-11

Caminho 2: 1-2-3-4-5-10-1-11

Caminho 3: 1-2-3-6-8-9-10-1-11

Caminho 4: 1-2-3-6-7-9-10-1-11

Note que cada novo caminho introduz uma nova aresta. O caminho

1-2-3-4-5-10-1-2-3-6-8-9-10-1-11

não é considerado um caminho independente porque é simplesmente uma combinação dos caminhos já especificados e não atravessa nenhuma nova aresta.

⁴ Uma discussão mais detalhada sobre grafos e seus usos é apresentada na Seção 23.6.1.

Os caminhos de 1 a 4 constituem um *conjunto base* para o grafo de fluxo da Figura 23.2b. Isto é, se testes podem ser projetados para forçar a execução desses caminhos (conjunto base), cada comando do programa terá sido executado com certeza pelo menos uma vez, e cada condição terá sido executada em seus lados verdadeiro e falso. Deve-se notar que o conjunto base não é único. De fato, vários conjuntos base diferentes podem ser derivados para um dado projeto procedimental.

Como sabemos quantos caminhos procurar? O cálculo da complexidade ciclomática fornece a resposta. *Complexidade ciclomática* é uma métrica de software que fornece uma medida quantitativa da complexidade lógica de um programa. Quando usada no contexto do método de teste de caminho básico, o valor calculado para a complexidade ciclomática define o número de caminhos independentes no conjunto base de um programa, fornecendo um limite superior para a quantidade de testes que devem ser realizados para garantir que todos os comandos tenham sido executados pelo menos uma vez.

A complexidade ciclomática tem um fundamento na teoria dos grafos e fornece uma métrica de software extremamente útil. A complexidade é calculada por uma de três maneiras:

1. O número de regiões do grafo de fluxo corresponde à complexidade ciclomática.
2. A complexidade ciclomática $V(G)$ para um grafo de fluxo G é definida como

$$V(G) = E - N + 2$$

onde E é o número de arestas do grafo de fluxo, e N é o número de nós do grafo de fluxo.

3. A complexidade ciclomática $V(G)$ para um grafo de fluxo G é definida como

$$V(G) = P + 1$$

onde P é o número de nós predicados contidos no grafo de fluxo G .

Examinando mais uma vez o diagrama de fluxo da Figura 23.2b, a complexidade ciclomática pode ser calculada usando cada um dos algoritmos citados anteriormente:

1. O grafo de fluxo tem quatro regiões.
2. $V(G) = 11$ arestas $- 9$ nós $+ 2 = 4$.
3. $V(G) = 3$ nós predicados $+ 1 = 4$.

Portanto, a complexidade ciclomática para o grafo de fluxo da Figura 23.2b é 4.

E o mais importante: o valor para $V(G)$ fornece um limite superior para o número de caminhos independentes que podem formar o conjunto base e, como consequência, um limite superior sobre o número de testes que devem ser projetados e executados para garantir a abrangência de todos os comandos do programa.

A complexidade ciclomática é uma métrica útil para prever módulos que têm a tendência de apresentar erros. Ela pode ser usada tanto para o planejamento de teste quanto para o projeto de casos de teste.

Como se calcula a complexidade ciclomática?

A complexidade ciclomática fornece o limite superior no número de casos de teste que precisam ser executados para garantir que cada comando do programa tenha sido executado pelo menos uma vez.



Usando a complexidade ciclomática

Cena: Sala da Shakira.

Atores: Vinod e Shakira – membros da equipe de engenharia de software do CasaSegura que estão trabalhando no planejamento de teste para as funções de segurança.

Conversa:

Shakira: Olha... sei que deveríamos fazer o teste de unidade em todos os componentes da função de segurança, mas eles são muitos, e, se você considerar o número de operações que precisam ser exercitadas, eu não sei... talvez devamos nos esquecer o teste caixa-branca, integrar tudo e começar a fazer os testes caixa-preta.

Vinod: Você acha que não temos tempo suficiente para fazer o teste dos componentes, realizar as operações e então integrar?

Shakira: O prazo final para o primeiro incremento está se esgotando, e eu gostaria de...

Vinod: Por que você não aplica testes caixa-branca pelo menos nas operações que têm maior probabilidade de apresentar erros?

Shakira (desesperada): E como posso saber exatamente quais são as que têm maior possibilidade de erro?

Vinod: V de G .

Shakira: Hein?

Vinod: Complexidade ciclomática – V de G . Basta calcular $V(G)$ para cada uma das operações dentro de cada um dos componentes e ver quais têm os maiores valores para $V(G)$. São essas que têm maior tendência a apresentar erro.

Shakira: E como calculo V de G ?

Vinod: É muito fácil. Aqui está um livro que descreve como fazer.

Shakira (folheando o livro): OK, não parece difícil. Vou tentar. As operações que tiverem os maiores $V(G)$ serão as candidatas aos testes caixa-branca.

Vinod: Mas lembre-se de que não há garantia. Um componente com baixo valor $V(G)$ pode ainda estar sujeito a erro.

Shakira: Tudo bem. Isso pelo menos me ajuda a limitar o número de componentes que precisam passar pelo teste caixa-branca.

23.4.3 Derivação de casos de teste

"Errar é humano;
encontrar um defeito é
divino."

Robert Dunn

O método do teste de caminho base pode ser aplicado a um projeto procedural ou ao código-fonte. Nesta seção, apresentamos o teste de caminho básico como uma série de passos. O procedimento *média* (*average*), mostrado em PDL na Figura 23.4, será usado como exemplo para ilustrar cada passo no método do projeto de casos de teste. Note que *média*, embora sendo um algoritmo extremamente simples, contém condições compostas e ciclos. Os passos a seguir podem ser aplicados para derivar o conjunto base:

1. **Usando o projeto ou o código como base, desenhe o grafo de fluxo correspondente.** Um grafo de fluxo é criado usando os símbolos e regras de construção apresentados na Seção 23.4.1. De acordo com a PDL para *média* na Figura 23.4, é criado um grafo de fluxo enumerando-se os comandos PDL que serão mapeados por nós correspondentes do grafo de fluxo. O grafo de fluxo correspondente está na Figura 23.5.
2. **Determine a complexidade ciclomática do diagrama de fluxo resultante.** A complexidade ciclomática $V(G)$ é determinada aplicando-se os algoritmos descritos na Seção 23.4.2. Deve-se notar que $V(G)$ pode ser determinada sem desenvolver um grafo de fluxo, contando-se todos os comandos condicionais em PDL (para o procedimento *média*, o total

PROCEDURE average;

* Este procedimento calcula a média de 100 ou menos números situados entre valores limites; também calcula a soma e o total de números válidos.

INTERFACE RETURNS average, totalInput, totalValid;
INTERFACE ACCEPTS value, minimum, maximum;

TYPE value[1:100] IS SCALAR ARRAY;
TYPE average, totalInput, totalValid;
minimum, maximum, sum IS SCALAR;
TYPE i IS INTEGER

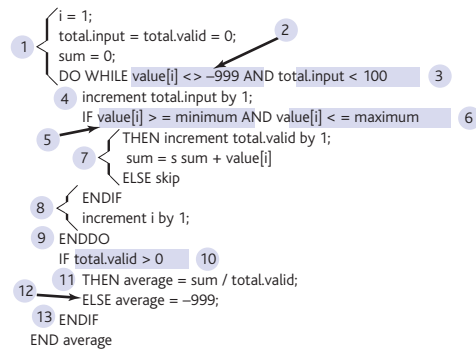


FIGURA 23.4 PDL com nós identificados.

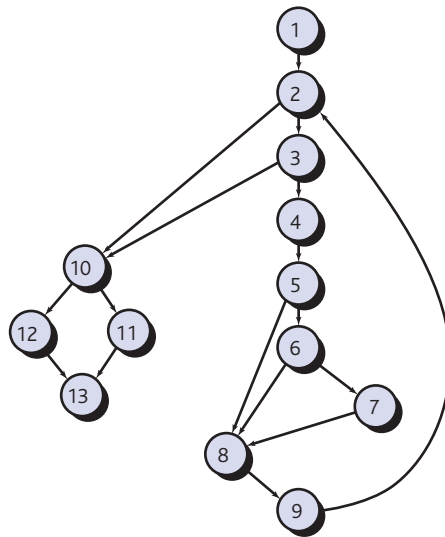


FIGURA 23.5 Grafo de fluxo para o procedimento médio.

de condições compostas é igual a dois) e somando 1. De acordo com a Figura 23.5,

$V(G) = 6$ regiões

$V(G) = 17$ arestas $- 13$ nós $+ 2 = 6$.

$V(G) = 5$ nós predicados $+ 1 = 6$.

"O foguete Ariane 5 explodiu no lançamento por causa de um defeito de software (uma falha) envolvendo a conversão de um valor em ponto flutuante de 64 bits em um inteiro de 16 bits. O foguete e seus quatro satélites não estavam seguros e valiam \$500 milhões. [Testes de caminho que exercitam o caminho de conversão] teriam descoberto o defeito, mas foram vetados por razões de orçamento."

Notícia de um jornal

- 3. Determine um conjunto base de caminhos linearmente independentes.** O valor de $V(G)$ fornece o número de caminhos linearmente independentes por meio da estrutura de controle do programa. No caso do procedimento *média*, esperamos especificar seis caminhos:

Caminho 1: 1-2-10-11-13
 Caminho 2: 1-2-10-12-13
 Caminho 3: 1-2-3-10-11-13
 Caminho 4: 1-2-3-4-5-8-9-2-...
 Caminho 5: 1-2-3-4-5-6-8-9-2-...
 Caminho 6: 1-2-3-4-5-6-7-8-9-2-...

A reticência (...) após os caminhos 4, 5 e 6 indica que qualquer caminho através do restante da estrutura de controle é aceitável. Muitas vezes compensa identificar nós predicados como um auxílio na dedução de casos de teste. Nesse caso, os nós 2, 3, 5, 6 e 10 são nós predicados.

- 4. Prepare casos de teste que obriguem a execução de cada caminho do conjunto base.** Os dados devem ser escolhidos de modo que as condições nos nós predicados sejam definidas de forma apropriada à medida que cada caminho é testado. Cada caso de teste é executado e comparado com os resultados esperados. Depois que todos os casos de teste tiverem sido completados, o testador pode ter a certeza de que todos os comandos do programa foram executados pelo menos uma vez.

É importante notar que alguns caminhos independentes (por exemplo, o caminho 1 em nosso exemplo) não podem ser testados de forma individual. Isso significa que a combinação de dados necessária para percorrer o caminho não pode ser obtida no fluxo normal do programa. Nesses casos, esses caminhos são testados como parte de outro teste de caminho.

23.4.4 Matrizes de grafos

O procedimento para derivar o grafo de fluxo e até determinar um conjunto de caminhos base é passível de mecanização. Uma estrutura de dados, chamada de *matriz de grafos*, pode ser muito útil para o desenvolvimento de uma ferramenta de software que ajuda no teste do caminho base.

Uma matriz de grafo é uma matriz quadrada cujo tamanho (número de linhas e colunas) é igual ao número de nós no grafo de fluxo. Cada linha e coluna corresponde a um nó identificado, e as entradas da matriz correspondem a conexões (arestas) entre nós. Um exemplo simples de um grafo de fluxo e sua matriz de grafo [Bei90] correspondente é mostrado na Figura 23.6.

Observando a figura, cada nó do grafo de fluxo é identificado por números, enquanto cada aresta é identificada por letras. Uma letra na matriz corresponde à conexão entre dois nós. Por exemplo, o nó 3 está conectado ao nó 4 pela aresta *b*.

Até aqui, a matriz de grafo nada mais é do que uma representação tabular de um grafo de fluxo. No entanto, acrescentando um *peso de ligação* a cada

O que é uma matriz de grafo e como podemos ampliá-la para uso em teste?

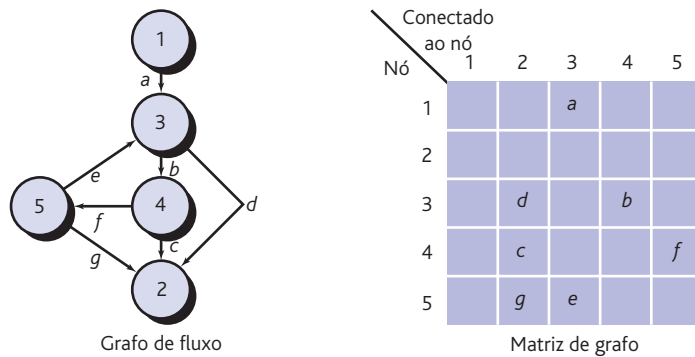


FIGURA 23.6 Matriz de grafo.

entrada da matriz, a matriz de grafo pode se tornar uma poderosa ferramenta para avaliar a estrutura de controle do programa durante o teste. O peso da ligação fornece informações adicionais sobre o fluxo de controle. Em sua forma mais simples, o peso da ligação é 1 (existe uma conexão) ou 0 (não existe uma conexão). Mas pesos de ligação podem ser atribuídos de acordo com outras propriedades mais interessantes:

- A probabilidade de que uma ligação (aresta) será executada.
- O tempo de processamento gasto para percorrer uma ligação.
- A quantidade de memória necessária para percorrer uma ligação.
- Os recursos necessários para percorrer uma ligação.

Beizer [Bei90] apresenta um tratamento completo de outros algoritmos matemáticos que podem ser aplicados às matrizes de grafos. Usando essas técnicas, a análise necessária para projeto de casos de teste pode ser parcial ou totalmente automatizada.

23.5 Teste de estrutura de controle

A técnica de teste de caminho base descrita na Seção 23.4 é uma dentre várias técnicas para teste de estrutura de controle. Embora o teste de caminho base seja simples e altamente eficaz, ele sozinho não é suficiente. Nesta seção discutimos outras variações do teste de estrutura de controle. Elas ampliam a abrangência do teste e melhoram a qualidade do teste caixa-branca.

Teste de condição [Tai89] é um método de projeto de caso de teste que exercita as condições lógicas contidas em um módulo de programa. O *teste de fluxo de dados* [Fra93] seleciona caminhos de teste de um programa de acordo com a localização de definições e usos de variáveis no programa.

O *teste de ciclo* é uma técnica de teste caixa-branca que se concentra exclusivamente na validade das construções de ciclo. Podem ser definidas quatro diferentes classes de ciclos [Bei90]: ciclos simples, ciclos concatenados, ciclos aninhados e ciclos não estruturados (Figura 23.7).

"Dar mais atenção à execução dos testes do que ao seu projeto é um erro clássico."

Brian Marick

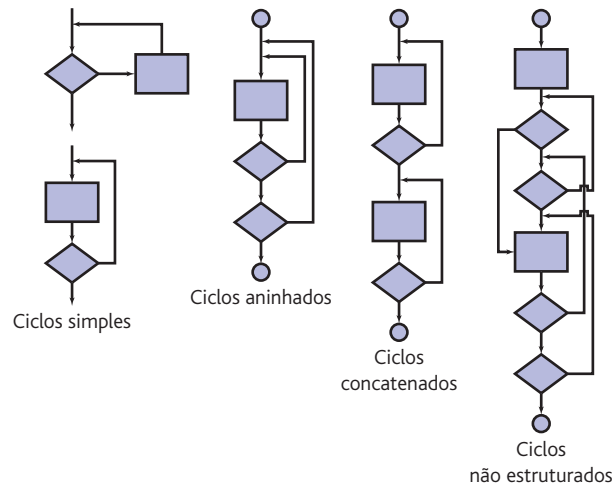


FIGURA 23.7 Classes de ciclos.

"Bons testadores são mestres na arte de encontrar 'alguma coisa esquisita' e agir sobre ela."

Brian Marick

Ciclos simples. O seguinte conjunto de testes pode ser aplicado a ciclos simples, onde n é o número máximo de passadas permitidas através do ciclo.

1. Pular o ciclo inteiramente.
2. Somente uma passagem pelo ciclo.
3. Duas passagens pelo ciclo.
4. m passagens através do ciclo onde $m < n$.
5. $n - 1$, n , $n + 1$ passagens através do ciclo.

Ciclos aninhados. Se fôssemos estender a abordagem de teste de ciclos simples para ciclos aninhados, o número de testes possíveis cresceria geometricamente à medida que o nível de aninhamento aumentasse. O resultado seria um número impossível de testes. Beizer [Bei90] sugere uma abordagem que ajudará a reduzir o número de testes:

1. Comece pelo ciclo mais interno. Coloque todos os outros ciclos nos seus valores mínimos.
2. Faça os testes de ciclo simples para o ciclo mais interno mantendo, ao mesmo tempo, os ciclos externos em seus parâmetros mínimos de iteração (por exemplo, contador do ciclo). Acrescente outros testes para valores fora do intervalo ou excluídos.
3. Trabalhe para fora, fazendo testes para o próximo ciclo, mas mantendo todos os outros ciclos externos nos seus valores mínimos e outros ciclos aninhados com valores "típicos".
4. Continue até que todos os ciclos tenham sido testados.

Você não pode testar ciclos não estruturados eficientemente. Refatore-os.

Ciclos concatenados. Ciclos concatenados podem ser testados usando a abordagem definida para ciclos simples, se cada um for independente do outro. No entanto, se dois ciclos forem concatenados e a contagem para o ciclo 1 for usada como valor individual para o ciclo 2, então os ciclos não são independentes. Quando os ciclos não são independentes, é recomendada a abordagem aplicada a ciclos aninhados.

Ciclos não estruturados. Sempre que possível, essa classe de ciclos deve ser redesenhada para refletir o uso das construções de programação estruturada (Capítulo 14).

23.6 Teste caixa-preta

Teste caixa-preta, também chamado de *teste comportamental* ou *teste funcional*, focaliza os requisitos funcionais do software. As técnicas de teste caixa-preta permitem derivar séries de condições de entrada que utilizarão completamente todos os requisitos funcionais para um programa. O teste caixa-preta não é uma alternativa às técnicas caixa-branca. Em vez disso, é uma abordagem complementar, com possibilidade de descobrir uma classe de erros diferente daquela obtida com métodos caixa-branca.

O teste caixa-preta tenta encontrar erros nas seguintes categorias: (1) funções incorretas ou ausentes, (2) erros de interface, (3) erros em estruturas de dados ou acesso a bases de dados externas, (4) erros de comportamento ou de desempenho e (5) erros de inicialização e término.

Diferentemente do teste caixa-branca, que é executado antecipadamente no processo de teste, o teste caixa-preta tende a ser aplicado durante estágios posteriores do teste (veja o Capítulo 22). Devido ao teste caixa-preta propositalmente desconsiderar a estrutura de controle, a atenção é focalizada no domínio das informações. Os testes são feitos para responder às seguintes questões:

- Como a validade funcional é testada?
- Como o comportamento e o desempenho do sistema são testados?
- Que classes de entrada farão bons casos de teste?
- O sistema é particularmente sensível a certos valores de entrada?
- Como as fronteiras de uma classe de dados são isoladas?
- Que taxas e volumes de dados o sistema pode tolerar?
- Que efeito combinações específicas de dados terão sobre a operação do sistema?

Que perguntas são respondidas pelos testes caixa-preta?

Com a aplicação de técnicas de caixa-preta, é extraído um conjunto de casos de teste que satisfazem aos seguintes critérios [Mye79]: casos de teste que reduzem, de um valor maior que um, o número de casos de teste adicionais que devem ser projetados para se obter um teste razoável, e casos de teste que dizem alguma coisa sobre a presença ou ausência de classes de erros, em vez de um erro associado somente ao teste específico que se está fazendo.

23.6.1 Métodos de teste baseados em grafos

O primeiro passo no teste caixa-preta é entender os objetos modelados⁵ no software e as relações que unem esses objetos. Uma vez conseguido isso, o próximo passo é definir uma série de testes que verificam se “todos os objetos

⁵ Nesse contexto, considera-se o termo *objeto* no sentido mais amplo possível. Abrange os objetos de dados, componentes tradicionais (módulos) e elementos de software orientados a objeto.

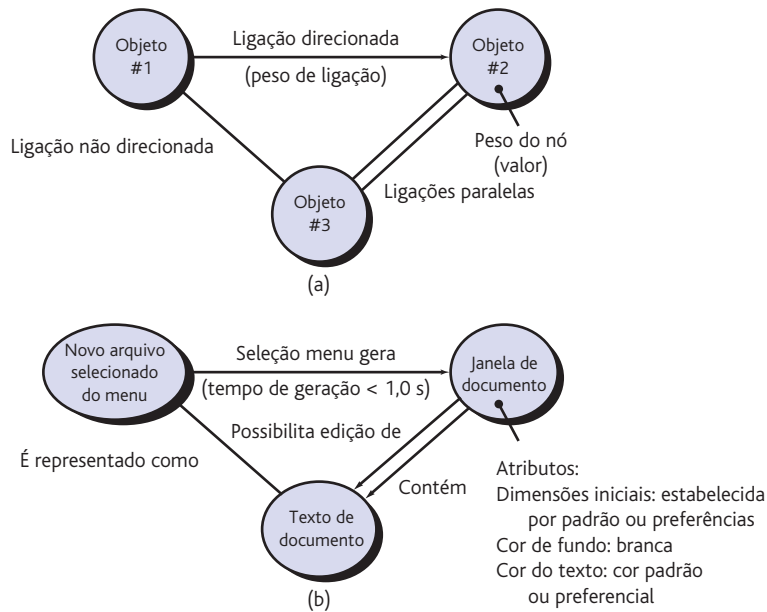


FIGURA 23.8 (a) Notação de grafo; (b) exemplo simples.

têm a relação esperada uns com os outros“ [Bei95]. Colocado de outra forma, o teste do software começa criando um grafo de objetos importantes e suas relações e, então, imaginando uma série de testes que abrangerá o grafo de forma que cada objeto e relação sejam exercitados, e os erros sejam descobertos.

Para executar esses passos, você começa criando um *grafo* – uma coleção de *nós* que representam objetos, *ligações* que representam as relações entre objetos, *pesos de nó* que descrevem as propriedades de um nó (por exemplo, o valor específico de um dado ou comportamento de estado), e *pesos de ligação* (*link weights*) que descrevem alguma característica de uma ligação.

A representação simbólica de um grafo é mostrada na Figura 23.8a. Os nós são representados como círculos unidos por ligações que assumem várias formas diferentes. Uma *ligação direta* (representada por uma seta) indica que uma relação se move apenas em uma direção. Uma *ligação bidirecional*, também chamada de *ligação simétrica*, significa que a relação se aplica em ambas as direções. *Ligações paralelas* são usadas quando várias relações diferentes são estabelecidas entre os nós do grafo.

Como um exemplo simples, considere uma parte de um grafo para uma aplicação de processador de texto (Figura 23.8b) na qual

Objeto #1 = **arquivoNovo** (seleção de menu)

Objeto #2 = **janelaDeDocumento**

Objeto #3 = **textoDeDocumento**

De acordo com a figura, uma seleção de menu em **arquivoNovo** gera uma janela de documento. O peso do nó de **janelaDeDocumento** fornece uma lista dos atributos de janela que devem ser esperados quando a janela é gerada. O peso da ligação indica que a janela deve ser gerada em menos de 1,0 segundo. Uma ligação indireta estabelece uma relação simétrica entre a seleção de menu

Um grafo representa a relação entre objetos dados e objetos programa, permitindo criar casos de teste que procuram erros associados a essas relações.

arquivoNovo e **textoDeDocumento**, e ligações paralelas indicam relações entre **janelaDeDocumento** e **textoDeDocumento**. Na realidade, teria de ser gerado um grafo muito mais detalhado como um precursor para o projeto do caso de teste. Você pode, então, criar casos de teste atravessando o grafo e percorrendo cada uma das relações mostradas. Esses casos de teste são projetados numa tentativa de encontrar erros em qualquer uma das relações. Beizer [Bei95] descreve uma série de métodos de teste comportamental que utilizam grafos:

Modelagem de fluxo de transação. Os nós representam passos em alguma transação (por exemplo, os passos necessários para fazer uma reserva em uma empresa aérea usando um serviço online), e as ligações representam a conexão lógica entre os passos. Por exemplo, **flightInformationInput** (entrada de informação sobre o voo) é seguido por **validationAvailabilityProcessing()** (processamento de validação da disponibilidade).

Modelagem de estado finito. Os nós representam diferentes estados observáveis pelo usuário do software (por exemplo, cada uma das “telas” que aparecem quando um atendente recebe um pedido por telefone), e as ligações representam as transições que ocorrem na mudança de um estado para outro (por exemplo, **orderInformation** (informação sobre o pedido) é verificada durante **inventoryAvailabilityLook-up** (consulta da disponibilidade no inventário) e é seguida pela entrada **customerBillingInformation** (informações para faturamento)). O grafo de estado (Capítulo 11) pode ser usado para ajudar a criar grafos desse tipo.

Modelagem de fluxo de dados. Os nós são objetos dados, e as ligações são as transformações que ocorrem para traduzir um objeto de dados em outro. Por exemplo, o nó **FICATaxWithheld** (FTW, imposto retido na fonte) é calculado a partir do salário bruto (**GW**, *gross wages*) usando a relação $FTW = 0,62 \times GW$.

Modelagem no tempo. Os nós são objetos de programa, e as ligações são conexões sequenciais entre aqueles objetos. Pesos de ligação são usados para especificar os tempos de execução necessários enquanto o programa é executado.

Uma discussão detalhada sobre cada um desses métodos de teste com base em diagrama vai além dos objetivos deste livro. Se estiver interessado, consulte [Bei95] para uma descrição detalhada.

23.6.2 Particionamento de equivalência

Particionamento de equivalência é um método de teste caixa-preta que divide o domínio de entrada de um programa em classes de dados a partir das quais podem ser criados casos de teste. Um caso de teste ideal descobre, sozinho, uma classe de erros (por exemplo, processamento incorreto de todos os dados de caracteres) que poderia, de outro modo, exigir a execução de muitos casos de teste até que o erro geral aparecesse.

O projeto de casos de teste para particionamento de equivalência tem como base a avaliação das *classes de equivalência* para uma condição de entrada. Usando conceitos introduzidos na seção anterior, se um conjunto de

objetos pode ser vinculado por relações simétricas, transitivas e reflexivas, uma classe de equivalência estará presente [Bei95]. Uma classe de equivalência representa um conjunto de estados válidos ou inválidos para condições de entrada. Tipicamente, uma condição de entrada é um valor numérico específico, um intervalo de valores, um conjunto de valores relacionados ou uma condição booleana. Classes de equivalência podem ser definidas de acordo com as seguintes regras:

Como definir classes de equivalência para teste?

1. Se uma condição de entrada especifica um intervalo, são definidas uma classe de equivalência válida e duas classes de equivalência inválidas.
2. Se uma condição de entrada exige um valor específico, são definidas uma classe de equivalência válida e duas classes de equivalência inválidas.
3. Se uma condição de entrada especifica um membro de um conjunto, são definidas uma classe de equivalência válida e uma classe de equivalência inválida.
4. Se uma condição de entrada for booleana, são definidas uma classe válida e uma inválida.

Aplicando as diretrizes para a derivação de classes de equivalência, podem ser desenvolvidos e executados casos de teste para o domínio de entrada de cada item de dado. Os casos de teste são selecionados de maneira que o máximo de atributos de uma classe de equivalência seja exercitado ao mesmo tempo.

“Uma maneira eficaz de testar código é exercitá-lo em suas fronteiras naturais.”

Brian Kernighan

23.6.3 Análise de valor limite

Um número maior de erros ocorre nas fronteiras do domínio de entrada e não no “centro”. É por essa razão que foi desenvolvida a *análise do valor limite* (BVA, *boundary value analysis*) como uma técnica de teste. A análise de valor limite leva a uma seleção de casos de teste que utilizam valores limites.

A análise de valor limite é uma técnica de projeto de casos de teste que complementa o particionamento de equivalência. Em vez de selecionar qualquer elemento de uma classe de equivalência, a BVA conduz à seleção de casos de teste nas “bordas” da classe. Em vez de focalizar somente condições de entrada, a BVA obtém casos de teste também a partir do domínio de saída [Mye79].

As diretrizes para a BVA são similares, em muitos aspectos, àquelas fornecidas para o particionamento de equivalência:

1. Se uma condição de entrada especifica um intervalo limitado por valores a e b , deverão ser projetados casos de teste com valores a e b imediatamente acima e abaixo de a e b .
2. Se uma condição de entrada especifica um conjunto de valores, deverão ser desenvolvidos casos de teste que usam os números mínimo e máximo. São testados também valores imediatamente acima e abaixo dos valores mínimo e máximo.
3. Aplique as diretrizes 1 e 2 às condições de saída. Por exemplo, suponha que um programa de análise de engenharia precisa ter como saída uma tabela de temperatura *versus* pressão. Deverão ser projetados casos de teste para criar um relatório de saída que produza o número máximo (e mínimo) permitido de entradas da tabela.

A BVA estende o particionamento de equivalência focalizando os dados nas “bordas” de uma classe de equivalência.

4. Se as estruturas de dados internas do programa prescreveram fronteiras (por exemplo, uma tabela tem um limite definido de 100 entradas), não se esqueça de criar um caso de teste para exercitar a estrutura de dados na fronteira.

Até certo ponto, a maioria dos engenheiros de software executa intuitivamente a BVA. Aplicando essas diretrizes, o teste de fronteira será mais completo, tendo, assim, uma possibilidade maior de detecção de erro.

23.6.4 Teste de matriz ortogonal

Há muitas aplicações nas quais o domínio de entrada é relativamente limitado. Isso significa que o número de parâmetros de entrada é pequeno e que os valores que cada um dos parâmetros pode assumir estão claramente limitados. Quando esses números são muito pequenos (por exemplo, três parâmetros de entrada assumindo três valores discretos cada um), é possível considerar cada permutação de entrada e testar exaustivamente o domínio de entrada. No entanto, à medida que cresce o número de valores de entrada e o número de valores discretos para cada item de dado aumenta, o teste exaustivo torna-se impraticável ou impossível.

O *teste de matriz ortogonal* pode ser aplicado a problemas nos quais o domínio de entrada é relativamente pequeno, mas muito grande para acomodar o teste exaustivo. O método de teste de matriz ortogonal é particularmente útil para encontrar erros associados a falhas de regiões – uma categoria de erro associada à lógica defeituosa em um componente de software.

Para ilustrar a diferença entre teste de matriz ortogonal e abordagens mais convencionais do tipo “uma entrada de cada vez”, considere um sistema que tenha três itens de entrada, X, Y e Z. Cada um desses itens de entrada tem três valores discretos associados. Existem $3^3 = 27$ casos de teste possíveis. Phadke [Pha97] sugere uma visualização geométrica dos casos de teste possíveis, associados a X, Y e Z, ilustrada na Figura 23.9. Olhando a figura, um item de entrada de cada vez pode ser variado em sequência ao longo de cada eixo de entrada. Isso resulta em uma abrangência relativamente limitada do domínio de entrada (representado pelo cubo da esquerda na figura).

Quando ocorre o teste de matriz ortogonal, é criada uma matriz ortogonal L9 de casos de teste. O conjunto ortogonal L9 tem uma “propriedade de

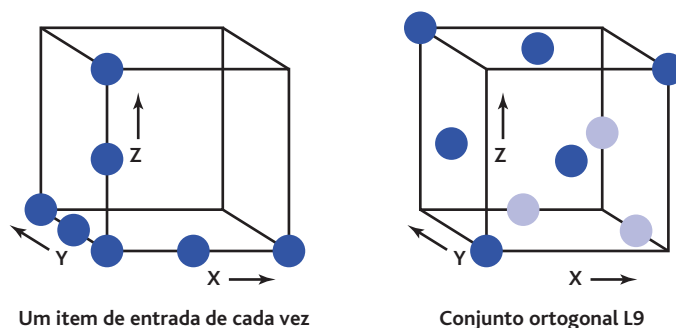


FIGURA 23.9 Uma visualização geométrica dos casos de teste.

Fonte: [Pha97].

balanceamento” [Pha97]. Ou seja, casos de teste (representados por pontos escuros na figura) são “dispersos uniformemente pelo domínio do teste”, conforme ilustra o cubo da direita na Figura 23.9. A cobertura de teste ao longo do domínio de entrada é mais completa.

Para ilustrar o uso da matriz ortogonal L9, considere a função *envie* para uma aplicação de fax. São passados quatro parâmetros, P1, P2, P3 e P4, para a função *envie*. Cada parâmetro assume três valores discretos. Por exemplo, P1 assume os valores:

P1 = 1, enviar agora

P1 = 2, enviar após 1 hora

P1 = 3, enviar depois da meia-noite

P2, P3 e P4 também assumiriam valores 1, 2 e 3, significando outras funções de envio.

Se fosse escolhida a estratégia de teste “um item de entrada de cada vez”, seria especificada a seguinte sequência (P1, P2, P3, P4) de testes: (1, 1, 1, 1), (2, 1, 1, 1), (3, 1, 1, 1), (1, 2, 1, 1), (1, 3, 1, 1), (1, 1, 2, 1), (1, 1, 3, 1), (1, 1, 1, 2) e (1, 1, 1, 3). Mas isso revelaria apenas *falhas de modo simples* (*single mode faults*) [Pha97]; ou seja, falhas acarretadas por um único parâmetro.

Dado o número relativamente pequeno de parâmetros de entrada e valores discretos, o teste exaustivo é possível. O número de testes necessários é $3^4 = 81$ – grande, porém controlável. Todas as falhas associadas à permutação de itens de dados seriam encontradas, mas o trabalho necessário é relativamente alto.

A abordagem do teste de matriz ortogonal permite obter boa abrangência de teste com bem menos casos de teste do que a estratégia exaustiva. Uma matriz ortogonal L9 para a função *envie* da aplicação de fax está ilustrada na Figura 23.10.

Phadke [Pha97] avalia o resultado dos testes usando a matriz ortogonal L9 da seguinte maneira:

Casos de teste	Parâmetros de teste			
	P1	P2	P3	P4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

FIGURA 23.10 Uma matriz ortogonal L9.

Detecte e isole todas as falhas de modo simples. Uma falha de modo simples é um problema consistente com qualquer nível de qualquer parâmetro isolado. Por exemplo, se todos os casos de teste de fator $P1 = 1$ causam uma condição de erro, trata-se de uma falha de modo simples. Nesse exemplo, os testes 1, 2 e 3 [Figura 23.10] mostrarão erros. Analisando as informações sobre quais testes mostram erros, pode-se identificar quais valores de parâmetros causam a falha. Nesse exemplo, observando que os testes 1, 2 e 3 causam um erro, podemos isolar o processamento lógico associado a “enviar agora” ($P1 = 1$) como origem do erro. Esse isolamento da falha é importante para poder corrigi-la.

Detecte todas as falhas de modo duplo. Se existe um problema consistente quando níveis específicos de dois parâmetros ocorrem juntos, chamamos isso de *falha de modo duplo* (*double mode fault*). Sem dúvida, uma falha de modo duplo é uma indicação de incompatibilidade do par ou interações danosas entre dois parâmetros de teste.

Falhas multimodo. Matrizes ortogonais ldo tipo mostradol somente podem garantir a detecção de falhas de modo simples e duplo. No entanto, muitas falhas multimodo também são detectadas por esses testes.

Uma discussão detalhada do teste de matriz ortogonal pode ser encontrada em [Pha89].

FERRAMENTAS DO SOFTWARE



Projeto de caso de teste

Objetivo: auxiliar a equipe de software no desenvolvimento de uma série completa de casos de teste, tanto para teste caixa-preta quanto para caixa-branca.

Mecanismos: essas ferramentas se classificam em duas grandes categorias: ferramentas de teste estático e ferramentas de teste dinâmico. Na indústria, são usados três tipos diferentes de ferramentas de teste estático: ferramentas de teste baseadas em código, linguagens especializadas de teste e ferramentas de teste baseadas em requisitos. As ferramentas de teste baseadas em código aceitam código-fonte como entrada e executam uma série de análises que resultam na geração de casos de teste. As linguagens de teste especializadas (por exemplo, ATLAS) permitem a um engenheiro de software escrever especificações detalhadas de teste que descrevem cada caso de teste e as logísticas para sua execução. As ferramentas de teste baseadas em requisitos isolam requisitos específicos de usuário e sugerem casos de teste (ou classes de testes) que os exercitarão. As ferramentas de teste dinâmico interagem com um programa em execução, verificando a amplitude do caminho, testando asserções sobre o valor de variáveis específicas e instrumentando o fluxo de execução do programa de qualquer modo.

Ferramentas representativas:⁶

McCabeTest, desenvolvida pela McCabe & Associates (www.mccabe.com), implementa uma variedade de técnicas de teste de caminho derivadas de uma avaliação da complexidade ciclomática e de outras métricas de software.

TestWorks, desenvolvida pela Software Research (<http://www.testworks.com/stwhome.html>), é uma série completa de ferramentas de teste automáticas que ajudam no projeto de casos de teste para software desenvolvido em C/C++ e Java e proporciona suporte para teste de regressão.

T-VEC Test Generation System, desenvolvida pela T-VEC Technologies (www.t-vec.com), é uma ferramenta que suporta teste de unidade, integração e validação, ajudando no projeto de casos de teste usando informações contidas em uma especificação de requisitos Orientados a Objeto (OO).

e-TEST Suite, desenvolvida pela Empirix (www.empirix.com), abrange uma série completa de ferramentas para testar WebApps, incluindo ferramentas que ajudam no projeto de casos de teste e no planejamento de testes.

⁶ A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

"É bastante difícil encontrar um erro no seu código quando você o está procurando; é ainda mais difícil quando você presume que o seu software é isento de erros."

Steve McConnell

23.7 Teste baseado em modelos

Teste baseado em modelo (MBT, *model-based testing*) é uma técnica de teste caixa-preta que usa informações contidas no modelo de requisitos como base para a geração de casos de teste [DAC03]. Em muitos casos, as técnicas baseadas em modelo usam diagramas de estado UML, um elemento do modelo comportamental (Capítulo 11), como base para o projeto de casos de teste.⁷ A técnica MBT requer cinco passos:

1. **Analise um modelo comportamental existente para o software ou crie um.** Lembre-se de que o *modelo comportamental* indica como o software responderá a eventos ou estímulos externos. Para criar o modelo, você deverá executar os passos discutidos no Capítulo 11: (1) avaliar todos os casos de uso para entender completamente a sequência de interação dentro do sistema, (2) identificar eventos que controlam a sequência de interação e entender como esses eventos se relacionam com objetos específicos, (3) criar uma sequência para cada caso de uso, (4) criar um diagrama de estado UML para o sistema (por exemplo, veja a Figura 11.1) e (5) rever o modelo comportamental para verificar exatidão e consistência.
2. **Percorra o modelo comportamental e especifique as entradas que forçarão o software a fazer a transição de um estado para outro.** As entradas vão disparar eventos que farão a transição ocorrer.
3. **Reveja o modelo comportamental e observe as saídas esperadas à medida que o software faz a transição de um estado para outro.** Lembre-se de que cada transição de estado é disparada por um evento e que, em consequência da transição, alguma função é chamada, e saídas são criadas. Para cada conjunto de entradas (casos de teste) especificadas no passo 2, especifique as saídas esperadas, conforme caracterizadas no modelo comportamental.
4. **Execute os casos de teste.** Os testes podem ser executados manualmente, ou pode ser criado um script de teste e executado usando uma ferramenta de teste.
5. **Compare os resultados real e esperado e tome a ação corretiva necessária.**

O MBT ajuda a descobrir erros no comportamento do software e, conseqüentemente, é extremamente útil ao testar aplicações acionadas por eventos.

23.8 Teste da documentação e dos recursos de ajuda

O termo *teste de software* passa a imagem de grande quantidade de casos de teste preparados para exercitar programas de computador e os dados que manipulam. Mas erros nos recursos de ajuda ou na documentação do programa podem ser tão devastadores para a aceitação dos programas quanto os erros nos dados ou no código-fonte. Nada é mais frustrante do que seguir exa-

⁷ O teste baseado em modelo também pode ser usado quando os requisitos de software são representados por tabelas de decisões, gramáticas ou cadeias de Markov [DAC03].

tamente um guia do usuário ou um recurso de ajuda online e obter resultados ou comportamentos que não coincidem com aqueles previstos pela documentação. É por isso que o teste da documentação deve ser parte significativa de todo plano de teste de software.

O teste da documentação pode ser feito em duas fases. A primeira fase, a revisão técnica (Capítulo 20), examina o documento quanto à clareza de edição. A segunda fase, o teste ao vivo, usa a documentação em conjunto com o programa real.

Surpreendentemente, um teste ao vivo para documentação pode ser feito usando técnicas análogas a muitos métodos de teste caixa-preta discutidos anteriormente. Teste baseado em diagrama pode ser usado para descrever o uso do programa; particionamento de equivalência e análise de valor limite podem ser usados para definir várias classes de entradas e interações associadas. MBT pode ser usado para garantir que o comportamento documentado e o comportamento real coincidam. O uso do programa é, então, acompanhado com base na utilização da documentação.

INFORMAÇÕES



Teste de documentação

As seguintes questões devem ser respondidas durante o teste de documentação e/ou recurso de ajuda:

- A documentação descreve com precisão como proceder em cada modo de utilização?
- A descrição de cada sequência de interação é precisa?
- Os exemplos são precisos?
- A terminologia, as descrições dos menus e as respostas do sistema estão de acordo com o programa real?
- É relativamente fácil localizar diretrizes dentro da documentação?
- A solução de problemas pode ser obtida facilmente com a documentação?
- O sumário e o índice do documento são bons, exatos e completos?

- O estilo do documento (layout, fontes, recuos, gráficos) conduz ao entendimento e à rápida assimilação das informações?
- Todas as mensagens de erro do software que aparecem para o usuário estão descritas em mais detalhes no documento? As ações a ser tomadas em consequência de uma mensagem de erro estão claramente delineadas?
- Se forem usadas ligações de hipertexto, elas são precisas e completas?
- Se for usado hipertexto, o estilo de navegação é apropriado para as informações exigidas?

A única maneira viável de responder a essas questões é por meio de uma consultoria independente (por exemplo, usuários escolhidos) para testar a documentação no contexto do uso do programa. Todas as discrepâncias são anotadas, e as áreas do documento que apresentam ambiguidade ou deficiências são marcadas para ser reescritas.

23.9 Teste para sistemas em tempo real

A natureza assíncrona, dependente do tempo, de muitas aplicações em tempo real acrescenta um elemento novo e potencialmente difícil ao conjunto de testes – o tempo. O projetista do caso de teste tem de considerar não apenas os casos de teste convencionais, mas também a manipulação de eventos (o processamento de interrupção), a temporização dos dados e o paralelismo das tarefas (processos) que manipulam os dados. Em muitas situações, dados de teste fornecidos quando o sistema em tempo real está em um estado resul-

tarão em um processamento correto, enquanto os mesmos dados fornecidos quando o sistema está em um estado diferente podem resultar em erro.

Por exemplo, o software em tempo real que controla uma nova fotocopadora aceita interrupções do operador (o operador da máquina pressiona teclas de controle como RESET ou ESCURECER) sem resultar em erro quando a máquina está fazendo cópias (no estado *copiando*). Essas mesmas interrupções do operador, se forem acionadas quando a máquina está no estado *atolada* (*jammed*), causam a perda do código de diagnóstico que indicava a localização do enrosco (um erro).

Além disso, a relação íntima existente entre um software em tempo real e seu ambiente de hardware também pode causar problemas de teste. Testes de software devem levar em consideração o impacto das falhas do hardware sobre o processamento do software. Essas falhas podem ser extremamente difíceis de simular realisticamente.

Pode ser proposta uma estratégia global de quatro etapas para teste de software em tempo real:

Qual é uma estratégia eficaz para testar um sistema em tempo real?

- **Teste de tarefa.** Teste cada tarefa independentemente. Testes convencionais são projetados para cada tarefa e executados independentemente durante esses testes. Testes de tarefa descobrem erros em lógica e funções, mas não em temporização ou comportamento.
- **Teste comportamental.** Usando modelos de sistema criados com ferramentas automáticas, é possível simular o comportamento de um sistema em tempo real e examinar seu comportamento em consequência de eventos externos. Essas atividades de análise podem servir como base para o projeto de casos de teste executados quando o software de tempo real é criado. Usando uma técnica que é similar ao particionamento de equivalência (Seção 23.6.2), eventos (por exemplo, interrupções, sinais de controle) são classificados para teste. Por exemplo, eventos para a fotocopadora podem ser interrupções de usuário (por exemplo, zerar o contador), interrupções mecânicas (por exemplo, papel enroscado), interrupções de sistema (por exemplo, pouco toner) e modos de falha (por exemplo, rolo superaquecido). Cada um desses eventos é testado individualmente, e o comportamento do sistema executável é examinado para detectar erros que ocorrem em consequência do processamento associado a esses eventos.
- **Teste intertarefas.** Uma vez isolados os erros em tarefas individuais e no comportamento do sistema, o teste passa para os erros relacionados ao tempo. Tarefas assíncronas que devem comunicar-se umas com as outras são testadas com diferentes taxas de dados e carga de processamento para determinar se ocorrerão erros de sincronização intertarefas. Além disso, tarefas que se comunicam via fila de mensagens ou armazenamento de dados são testadas para descobrir erros no dimensionamento dessas áreas de armazenamento.
- **Teste de sistema.** O software e o hardware são integrados, e é feita uma gama completa de testes do sistema na tentativa de descobrir erros na interface software-hardware. A maioria dos sistemas em tempo real processa interrupções. Portanto, é essencial testar a manipulação desses

eventos booleanos. Usando o diagrama de estado (Capítulo 11), o testador desenvolve uma lista de todas as interrupções possíveis e o processamento que ocorre em consequência das interrupções. São planejados, então, testes para avaliar as seguintes características do sistema:

- São atribuídas prioridades corretas às interrupções e elas são manipuladas corretamente?
- O processamento de cada interrupção é manipulado corretamente?
- O desempenho (por exemplo, tempo de processamento) de cada procedimento de manipulação de interrupção está conforme os requisitos?
- A chegada de um alto volume de interrupções em instantes críticos cria problemas em função ou desempenho?

Além disso, devem ser testadas as áreas globais de dados usadas para transferir informações como parte do processamento de interrupção para avaliar o potencial de geração de efeitos colaterais.

23.10 Padrões para teste de software

O uso de padrões como um mecanismo para descrever soluções para problemas de projeto específicos foi discutido no Capítulo 16. Mas os padrões também podem ser usados para propor soluções para outras situações de engenharia de software – nesse caso, o teste de software. Os *padrões de teste* descrevem problemas comuns de teste e soluções que podem ajudar a lidar com eles.

Grande parte do teste de software, inclusive durante a década passada, tem sido uma atividade improvisada. Se os padrões de teste podem ajudar uma equipe de software a se comunicar sobre testes de forma mais eficaz, a entender as forças motivadoras que conduzem a uma abordagem específica para o teste e a encarar o projeto de testes como uma atividade evolucionária, na qual cada iteração resulta em um conjunto mais completo de casos de teste, então os padrões realmente dão uma grande contribuição.

Os padrões de teste são descritos de maneira muito semelhante aos padrões de projeto (Capítulo 16). Já foram propostos dezenas de padrões de teste na literatura (por exemplo, [Mar02]). Os três padrões a seguir (apresentados apenas de uma forma superficial) fornecem exemplos representativos:

Nome do padrão: **Teste em dupla**

Resumo: Um padrão orientado a processo, o **teste em dupla** descreve uma técnica que é análoga à programação em pares (Capítulo 5), na qual dois testadores trabalham em conjunto para projetar e executar uma série de testes que podem ser aplicados a atividades de teste de unidade, integração ou validação.

Nome do padrão: **Interface de teste separada**

Resumo: Há necessidade de testar todas as classes em um sistema orientado a objetos, incluindo “classes internas” (isto é, classes que não expõem qualquer interface fora do componente que as utilizou). O padrão **Interface de teste separada** descreve como criar “uma interface de teste que pode ser usada para descrever testes específicos em classes que são visíveis somente internamente a um componente” [Lan01].

Um catálogo de padrões de teste de software pode ser encontrado em <http://c2.com/cgi-bin/wiki?TestingPatterns>.

Padrões de teste podem ajudar uma equipe de software a comunicar sobre o teste mais eficientemente e a entender melhor as forças que levam a uma abordagem de teste específica.

Nome do padrão: Teste de cenário

Resumo: Uma vez feitos os testes de unidade e de integração, é necessário determinar se o software se comportará ou não de maneira que satisfaça aos usuários. O padrão **Teste de cenário** descreve uma técnica para exercitar o software do ponto de vista do usuário. Uma falha nesse nível indica que o software deixou de atender aos requisitos visíveis para o usuário [Kan01].

Uma discussão abrangente sobre os padrões de teste está fora dos objetivos deste livro. Se você estiver interessado em mais detalhes, consulte [Bin99], [Mar02] e [Tho04] para informações adicionais sobre esse importante assunto.

23.11 Resumo

O principal objetivo do projeto de caso de teste é criar uma série de testes que tenha a mais alta probabilidade de descobrir erros no software. Para conseguir esse objetivo, são usadas duas categorias de técnicas de projeto de caso de teste: teste caixa-branca e teste caixa-preta.

Os testes caixa-branca focam a estrutura de controle do programa. São criados casos de teste para assegurar que todas as instruções do programa foram executadas pelo menos uma vez durante o teste e que todas as condições lógicas foram exercitadas. O teste de caminho base, uma técnica caixa-branca, usa diagramas de programa (ou matrizes de grafo) para derivar o conjunto de testes linearmente independentes que garantirão a cobertura. O teste de condições e de fluxo de dados exercita mais a lógica do programa, e o teste de ciclos complementa outras técnicas caixa-branca, fornecendo um procedimento para exercitar ciclos com vários graus de complexidade.

Hetzel [Het84] descreve o teste caixa-branca como “teste no pequeno” (para o que é particular). Sua implicação é que os testes caixa-branca que foram considerados neste capítulo são aplicados tipicamente a pequenos componentes de programas (por exemplo, módulos ou pequenos grupos de módulos). O teste caixa-preta, por outro lado, amplia o seu foco e pode ser chamado de “teste no grande” (para o que é amplo).

Os testes caixa-preta são projetados para validar requisitos funcionais sem levar em conta o funcionamento interno de um programa. As técnicas caixa-preta focam o domínio de informações do software, derivando casos de teste e particionando o domínio de entrada e saída de um programa de forma a proporcionar uma ampla cobertura do teste. O particionamento de equivalência divide o domínio de entrada em classes de dados que tendem a usar uma função específica do software. A análise de valor limite investiga a habilidade do programa para manipular dados nos limites do aceitável. O teste de matriz ortogonal proporciona um método eficiente e sistemático para testar sistemas com poucos parâmetros de entrada. O teste baseado em modelo usa elementos do modelo de requisitos para testar o comportamento de um aplicativo.

Desenvolvedores de software experientes muitas vezes afirmam: “O teste nunca termina, ele apenas se transfere de você [o engenheiro de software] para o seu cliente. Toda vez que um cliente usa o programa, um teste está sendo realizado”. Aplicando o projeto de caso de teste, você pode obter um teste mais completo e, assim, descobrir e corrigir o maior número de erros antes que comecem os “testes do cliente”.

Problemas e pontos a ponderar

23.1 Myers [Mye79] usa o seguinte programa como uma autoavaliação para a sua habilidade em especificar teste adequado: Um programa lê três valores inteiros. Os três valores são interpretados como representando os comprimentos dos lados de um triângulo. O programa imprime uma mensagem que diz se o triângulo é escaleno, isósceles ou equilátero. Desenvolva um conjunto de casos de teste que você acha que testará adequadamente esse programa.

23.2 Projete e implemente o programa (com manipulação de erro onde for apropriado) especificado no Problema 23.1. Crie um grafo de fluxo para o programa e aplique teste de caminho base para desenvolver casos de teste que garantirão que todos os comandos no programa foram testados. Execute os casos e mostre seus resultados.

23.3 Você consegue pensar em objetivos de teste adicionais que não foram discutidos na Seção 23.1.1?

23.4 Selecione um componente de software que você tenha projetado e implementado recentemente. Projete um conjunto de casos de teste que garantirão que todos os comandos foram executados usando teste de caminho base.

23.5 Especifique, projete e implemente uma ferramenta de software que calcule a complexidade ciclomática para a linguagem de programação de sua escolha. Use uma matriz de grafo como estrutura de dados operativos em seu projeto.

23.6 Leia Beizer [Bei95] ou uma fonte de informação relacionada à Internet (por exemplo, www.laynetworks.com/Discrete%20Mathematics_1g.htm) e determine como o programa que você desenvolveu no Problema 23.5 pode ser ampliado para acomodar vários pesos de ligação. Amplie a sua ferramenta para probabilidades de execução de processo ou tempos de processamento de ligação.

23.7 Projete uma ferramenta automatizada que reconheça ciclos e classifique-os conforme indicado na Seção 23.5.3.

23.8 Amplie a ferramenta descrita no Problema 23.7 para gerar casos de teste para cada categoria de ciclo, quando encontrado. Será necessário executar essa função iterativamente com o testador.

23.9 Dê pelo menos três exemplos nos quais o teste caixa-preta pode dar a impressão de que “está tudo OK”, enquanto os testes caixa-branca podem descobrir um erro. Dê pelo menos três exemplos nos quais o teste caixa-branca pode dar a impressão de que “está tudo OK”, enquanto os testes caixa-preta podem descobrir um erro.

23.10 Poderá o teste exaustivo (mesmo que seja possível para programas muito pequenos) garantir que o programa está 100% correto?

23.11 Teste um manual de usuário (ou um recurso de ajuda) de uma aplicação que você usa frequentemente. Encontre pelo menos um erro na documentação.

Leituras e fontes de informação complementares

Praticamente todos os livros dedicados a teste de software consideram tanto estratégia quanto táticas. Portanto, as leituras complementares citadas no Capítulo 22 são igualmente aplicáveis a este capítulo. Burnstein (*Practical Software Testing*, Springer, 2010), Crispin e Gregory (*Agile Testing: A Practical Guide for Testers and Agile Teams*, Addison-Wesley, 2009), Lewis (*Software Testing and Continuous Quality Improvement*, 3ª ed., Auerbach, 2008), Ammann e Offutt (*Introduction to Software Testing*, Cambridge University Press, 2008), Everett e McCleod (*Software Testing*, Wiley-IEEE Computer Society

Press, 2007), Black (*Pragmatic Software Testing*, Wiley, 2007), Spiller e seus colegas (*Software Testing Process: Test Management*, Rocky Nook, 2007), Perry (*Effective Methods for Software Testing*, 3ª ed., Wiley, 2006), Loveland e seus colegas (*Software Testing Techniques*, Charles River Media, 2004), Dustin (*Effective Software Testing*, Addison-Wesley, 2002), Craig e Kaskiel (*Systematic Software Testing*, Artech House, 2002), Tamres (*Introducing Software Testing*, Addison-Wesley, 2002) e Whittaker (*Exploratory Software Testing: Tips, Tricks, and Techniques to Guide Test Design*, Addison-Wesley, 2009; e *How to Break Software*, Addison-Wesley, 2002) são apenas uma pequena amostra dos muitos livros que discutem princípios, conceitos, estratégias e métodos de teste.

Uma terceira edição do trabalho clássico de Myers [Mye79] foi produzida por Myers e seus colegas (*The Art of Software Testing*, 3ª ed., Wiley, 2011) e abrange técnicas de projeto de caso de teste com detalhe considerável. Black (*Managing the Testing Process*, 3ª ed., Wiley, 2009), Jorgensen (*Software Testing: A Craftsman's Approach*, 3ª ed., CRC Press, 2008), Pezze e Young (*Software Testing and Analysis*, Wiley, 2007), Perry (*Effective Methods for Software Testing*, 3ª ed., Wiley, 2006), Copeland (*A Practitioner's Guide to Software Test Design*, Artech, 2003) e Hutcheson (*Software Testing Fundamentals*, Wiley, 2003) oferecem apresentações úteis de métodos e técnicas de projeto de caso de teste. A obra clássica de Beizer [Bei90] contém uma cobertura ampla das técnicas caixa-branca, introduzindo um nível de rigor matemático que frequentemente falta em outros tratados sobre teste. Seu mais recente livro [Bei95] apresenta um tratamento conciso de importantes métodos.

Teste de software é uma atividade intensiva em termos de recursos. É por isso que muitas organizações automatizam partes do processo de teste. Livros de Graham e seus colegas (*Experiences of Test Automation: Case Studies of Software Test Automation*, Addison-Wesley, 2012; e *Software Test Automation*, Addison-Wesley, 1999), Li e Wu (*Effective Software Test Automation*, Sybex, 2004), Mosely e Posey (*Just Enough Software Test Automation*, Prentice Hall, 2002), Dustin, Rashka e Poston (*Automated Software Testing: Introduction, Management, and Performance*, Addison-Wesley, 1999), e Poston (*Automating Specification-Based Software Testing*, IEEE Computer Society, 1996) discutem ferramentas, estratégias e métodos para teste automático. Nguyen e seus colegas (*Happy About Global Software Test Automation*, Happy About Press, 2006) apresentam uma visão geral executiva da automação de teste.

Meszaros (*Unit Test Patterns: Refactoring Test Code*, Addison-Wesley, 2007), Thomas e seus colegas (*Java Testing Patterns*, Wiley, 2004) e Binder [Bin99] descrevem padrões de teste que abrangem métodos de teste, classes/clusters, subsistemas, componentes reutilizáveis, frameworks e sistemas, bem como automação de teste e testes especializados de bases de dados.

Há disponível na Internet uma ampla variedade de recursos de informação sobre métodos de projeto de casos de teste. Uma lista atualizada de referências relevantes para técnicas de teste (em inglês) pode ser encontrada no site: www.mhhe.com/pressman.

Teste de aplicações orientadas a objeto

24

No Capítulo 23, vimos que, basicamente, o objetivo de conduzir testes é encontrar o maior número possível de erros investindo-se uma quantidade de esforço gerenciável dentro de um intervalo de tempo realista. Embora esse objetivo fundamental permaneça inalterado para software orientado a objetos, a natureza dos programas orientados a objetos muda tanto a estratégia de teste quanto suas táticas.

Poderíamos argumentar que, à medida que o tamanho das bibliotecas de classe reutilizáveis aumenta, uma maior reutilização diminui a necessidade de testes mais pesados dos sistemas orientados a objetos. É exatamente o oposto. Binder [Bin94b] discute isso:

Cada reutilização é um novo contexto de uso, e é prudente fazer testes novamente. Parece provável que mais testes sejam necessários, e não menos, para se atingir uma alta confiabilidade em sistemas orientados a objetos.

Conceitos-chave

consistência.....	526
estratégias de teste	528
métodos de teste	529
modelos orientados a objetos	526
teste aleatório.....	532
teste baseado em cenários.....	532
teste baseado em falhas.....	531
teste baseado em sequências de execução	529
teste de conjunto.....	529

PANORAMA

O que é? A arquitetura de software orientado a objetos (OO) tem uma série de subsistemas

em camadas que encapsulam classes colaboradoras. Cada um desses elementos de sistema (subsistemas e classes) executa funções que ajudam a satisfazer os requisitos do sistema. É necessário testar um sistema orientado a objetos em vários níveis diferentes para descobrir erros que podem ocorrer à medida que as classes colaboram umas com as outras e os subsistemas se comunicam por meio de camadas da arquitetura.

Quem realiza? O teste orientado a objetos é executado por engenheiros de software e especialistas em teste.

Por que é importante? É necessário executar o programa antes que seja entregue ao cliente para remover todos os erros, de maneira que o usuário não passe por nenhuma frustração com um produto de má qualidade. Para encontrar o maior número possível de erros, devem ser executados testes sistematicamente, e os casos de teste devem ser projetados usando técnicas disciplinadas.

Quais são as etapas envolvidas? O teste orientado a objetos é estrategicamente análogo ao de sistemas convencionais, mas é taticamente diferente. Como os modelos de análise

e projeto orientados a objeto são similares em estrutura e conteúdo ao programa orientado a objetos resultante, o "teste" é iniciado com a revisão dos modelos. Uma vez gerado o código, o teste orientado a objetos começa "inicialmente" com o teste de classes. É projetada uma série de testes que exercitam as operações de classe e examinam se existem erros enquanto uma classe colabora com outras. À medida que as classes são integradas para formar um subsistema, aplicam-se testes baseados em sequências de execução (*thread*), baseados no uso e o teste de conjunto (*cluster testing*), junto com estratégias baseadas em falhas para usar completamente as classes colaboradoras. Por fim, são usados casos de uso (desenvolvidos como parte do modelo de análise) para descobrir erros em nível de validação de software.

Qual é o artefato? É projetado e documentado um conjunto de casos de teste, desenvolvidos para usar as classes, suas colaborações e comportamentos; são definidos os resultados esperados e registrados os obtidos.

Como garantir que o trabalho foi realizado corretamente?

Quando você começar o teste, mude o seu ponto de vista. Tente "quebrar" o software! Projete casos de teste de maneira disciplinada e reveja os casos que você criou para que sejam completos.

teste baseado em uso ...	529
teste de classe.....	528
teste de conjunto.....	529
teste de múltiplas	
classes.....	534
teste de partição.....	533

Para testar adequadamente os sistemas orientados a objetos, é preciso fazer três coisas: (1) a definição do teste deve ser ampliada para incluir as técnicas de descoberta de erro aplicadas à análise orientada a objetos e modelos de projeto, (2) a estratégia para teste de unidade e de integração deve mudar significativamente e (3) o projeto de casos de teste deve levar em conta as características especiais do software orientado a objetos.

24.1 Ampliando a visão do teste

A construção de software orientado a objetos começa com a criação dos modelos de análise e de projeto.¹ Devido à natureza evolucionária do paradigma da engenharia de software orientado a objetos, esses modelos começam como representações relativamente informais de requisitos de sistema e evoluem para modelos detalhados de classes, relações entre classes, projeto e alocação de sistema e projeto de objeto (incorporando um modelo da conectividade de objeto via mensagem). Em cada estágio, os modelos podem ser “testados” para descobrir erros antes de sua propagação para a próxima iteração.

A revisão da análise orientada a objetos e dos modelos de projeto é especialmente útil porque a mesma construção semântica (por exemplo, classes, atributos, operações, mensagens) aparece nos níveis de análise, projeto e código. Assim, um problema na definição de atributos de classe detectado durante a análise evitará efeitos colaterais que poderiam ocorrer se o problema só fosse descoberto na fase de projeto ou codificação (ou mesmo na próxima iteração da análise).

Por exemplo, considere uma classe na qual um conjunto de atributos é definido durante a primeira iteração da análise. Um atributo estranho é acrescentado à classe (decorrente da falta de entendimento do domínio do problema). Duas operações são, então, especificadas para manipular o atributo. É feita uma revisão, e um especialista em domínios aponta o problema. Eliminando-se o atributo estranho nesse estágio, problemas e trabalho desnecessário podem ser evitados durante a análise:

1. Subclasses especiais poderiam ter sido geradas para acomodar o atributo desnecessário ou suas exceções. O trabalho envolvido na criação de subclasses desnecessárias foi evitado.
2. Uma interpretação incorreta da definição de classe pode levar a relações de classe incorretas ou estranhas.
3. O comportamento do sistema ou suas classes pode ser caracterizado inadequadamente para acomodar o atributo estranho.

Se o problema não fosse descoberto (por meio de revisão antecipada) durante a análise e se propagasse, poderiam ocorrer estas situações durante o projeto:

Embora a revisão da análise orientada a objetos e dos modelos de projeto seja parte do “teste” de uma aplicação orientada a objetos, lembre-se de que isso não é suficiente. Devem-se realizar testes executáveis também.

¹ As técnicas de análise e modelagem de projeto são apresentadas na Parte II deste livro. Os conceitos de orientação a objetos básicos são apresentados no Apêndice 2.

1. Alocação imprópria da classe para o subsistema e/ou tarefas poderia ocorrer durante o projeto do sistema.
2. Trabalho de projeto desnecessário poderia ser despendido para criar o projeto procedural para as operações que lidam com o atributo estranho.
3. O modelo de mensagens estaria incorreto (porque devem ser designadas mensagens para as operações estranhas).

Se o problema não for detectado durante o projeto e passar para a codificação, será despendido um considerável esforço para gerar código que implemente um atributo desnecessário, duas operações desnecessárias, mensagens que controlam comunicação entre objetos e muitos outros problemas relacionados. Além disso, o teste da classe absorverá mais tempo do que o necessário. Uma vez descoberto o problema, deve ser feita a modificação do sistema, levando-se sempre em conta a alta possibilidade de efeitos colaterais causados pela alteração.

Durante estágios posteriores de seu desenvolvimento, os modelos de análise orientada a objetos (OOA, object-oriented analysis) e projeto orientado a objetos (OOD, object-oriented design) proporcionam informações substanciais sobre a estrutura e comportamento do sistema. Por essa razão, esses modelos deverão ser submetidos a rigorosa revisão antes da geração do código.

Todos os modelos orientados a objetos deverão ser testados (nesse contexto, o termo *teste* incorpora revisões técnicas) quanto a sua exatidão, integralidade e consistência com o contexto de sintaxe do modelo, semânticas e pragmatismo.

"As ferramentas que utilizamos têm uma profunda (e definitiva!) influência sobre nossos hábitos de pensar e, portanto, sobre nossas habilidades de pensar".

Edsger Dijkstra

24.2 Teste de modelos de análise e de projeto orientados a objetos

Análises e modelos de projeto não podem ser testados no sentido convencional, pois não podem ser executados. No entanto, podem ser usadas revisões técnicas (Capítulo 20) para examinar sua exatidão e consistência.

24.2.1 Exatidão dos modelos de OOA e OOD

A notação e a sintaxe usadas para representar modelos de análise e projeto estarão vinculadas a métodos específicos de análise e projeto escolhidos para o projeto. Logo, a exatidão sintática é julgada com base no uso adequado da simbologia; cada modelo é revisado para garantir que sejam mantidas as convenções apropriadas de modelagem.

Durante a análise e o projeto, é possível verificar a exatidão semântica de acordo com o modelo no domínio do problema do mundo real. Se o modelo reflete o mundo real com precisão (com um nível de detalhes apropriado para o estágio de desenvolvimento no qual o modelo é revisado), ele é semanticamente correto. Para determinar se, de fato, o modelo reflete os requisitos do mundo real, deve ser apresentado aos especialistas em domínio de problema que examinarão as definições de classe e a hierarquia quanto

a omissões e ambiguidades. Relações de classe (conexões de instância) são avaliadas para determinar se refletem precisamente as conexões do objeto no mundo real.²

24.2.2 Consistência dos modelos orientados a objetos

A consistência dos modelos orientados a objetos pode ser avaliada “considerando-se a relação entre entidades do modelo. Um modelo de análise ou projeto inconsistente tem representações em uma parte que não são refletidas corretamente em outras partes do modelo” [McG94].

Para avaliar a consistência, é preciso examinar todas as classes e suas conexões com as outras classes. Para facilitar essa atividade, podem ser usados o modelo classe-responsabilidade-colaboração (CRC) e um diagrama de relações entre objetos. Conforme vimos no Capítulo 10, o modelo CRC é composto de cartões de índice CRC. Cada cartão lista o nome da classe, suas responsabilidades (operações) e colaboradores (outras classes às quais envia mensagens e das quais depende para a execução de suas responsabilidades). As colaborações implicam uma série de relações (conexões) entre classes do sistema orientado a objetos. O modelo de relacionamento de objeto fornece uma representação gráfica das conexões entre as classes. Todas essas informações podem ser obtidas do modelo de análise (Capítulo 10).

Para avaliar o modelo de classe, os seguintes passos são recomendados [McG94]:

1. **Reveja o modelo CRC e o modelo de relação de objetos.** Faça uma verificação comparativa para garantir que todas as colaborações relacionadas ao modelo de requisitos estão corretamente refletidas em ambos.
2. **Inspecione a descrição de cada cartão de índice CRC para determinar se uma responsabilidade delegada faz parte da definição do colaborador.** Por exemplo, considere uma classe definida para um sistema de controle de ponto de venda que se chama **CréditoDeVenda**. Essa classe tem um cartão de índice CRC conforme mostra a Figura 24.1.
Para essa coleção de classes e colaborações, pergunte se uma responsabilidade (por exemplo, *lerCartãoCrédito*) é atendida se for delegada ao colaborador citado (**CartãoDeCrédito**). Isto é, a classe **CartãoDeCrédito** tem uma operação que a habilita a ser lida? Nesse caso, a resposta é “sim”. A relação-objeto é percorrida para garantir que todas essas conexões são válidas.
3. **Inverta a conexão para garantir que cada colaborador ao qual é solicitado serviço esteja recebendo solicitações de uma origem razoável.** Por exemplo, se a classe **CartãoDeCrédito** receber uma solicitação para **valor da compra** da classe **CréditoDeVenda**, haverá um problema. **CartãoDeCrédito** não conhece o valor da compra.

² Casos de uso podem ser valiosos para rastrear modelos de análise e projeto em um cenário de uso no mundo real para o sistema orientado a objetos.

nome da classe: créditoDeVenda	
tipo da classe: eventoDeTransação	
característica da classe: nãoTangível, atômica, sequencial, permanente, protegida	
responsabilidades:	colaboradores:
lerCartãoCrédito	cartãoDeCrédito
pegarAutorização	autoridadeDeCrédito
solicitarValorDaCompra	tiqueteDoProduto
	registroDoVendedor
	arquivoDeBalanço
gerarConta	conta

FIGURA 24.1 Um exemplo de um cartão de índice CRC usado para revisão.

4. Usando as conexões invertidas examinadas na etapa 3, determine se podem ser necessárias outras classes ou se as responsabilidades estão corretamente agrupadas entre as classes.
5. Determine se responsabilidades solicitadas mais amplas podem ser combinadas em uma única responsabilidade. Por exemplo, *lerCartãoCrédito* e *pegarAutorização* ocorrem em todas as situações. Elas podem ser combinadas em uma responsabilidade de *validação de solicitação de crédito*, que incorpora a obtenção do número do cartão de crédito e obtenção da autorização.

Você deve aplicar os passos 1 a 5 iterativamente a cada classe e através de cada evolução do modelo de requisitos.

Uma vez criado o modelo de projeto (Capítulos 12 a 18), é preciso também realizar revisões do projeto do sistema e do projeto do objeto. O projeto do sistema representa a arquitetura geral do produto, os subsistemas que compõem o produto, a maneira como os subsistemas são alocados a processadores, a alocação de classes para subsistemas e o projeto da interface do usuário. O modelo de objeto apresenta os detalhes de cada classe e as atividades de mensagens necessárias para implementar colaborações entre classes.

O projeto do sistema é revisado examinando-se o modelo objeto-comportamento desenvolvido durante a análise orientada a objetos e mapeando-se o comportamento exigido do sistema em relação aos subsistemas projetados para atender a esse comportamento. A concorrência e a alocação de tarefas também é revista segundo o contexto de comportamento do sistema. Os estados comportamentais do sistema são avaliados para determinar quais existem concorrentemente. São usados casos de uso para exercitar o projeto da interface do usuário.

O modelo orientado a objetos deve ser testado em relação à rede objeto-relacionamento para assegurar que todos os objetos de projeto contenham os atributos e operações necessários para implementar as colabora-

ções definidas para cada cartão de índice CRC. Além disso, é revisada a especificação dos detalhes de operação (isto é, os algoritmos que implementam as operações).

24.3 Estratégias de teste orientado a objetos

Conforme observado no Capítulo 23, a estratégia de teste de software clássica começa com o “teste no pequeno” (para o que é particular) e se amplia para o “teste no grande” (para o que é amplo). De acordo com o jargão do teste de software (Capítulo 23), você começa com *teste de unidade*, depois passa para o *teste de integração* e termina com a *validação e teste do sistema*. Em aplicações convencionais, o teste de unidade focaliza a menor unidade de programa compilável – o subprograma (por exemplo, componente, módulo, sub-rotina, procedimento). Depois que cada uma dessas unidades é testada individualmente, elas são integradas em uma estrutura de programa enquanto é executada uma série de testes de regressão para descobrir erros decorrentes das interfaces entre módulos e dos efeitos colaterais causados pela adição de novas unidades. Por fim, o sistema como um todo é testado para assegurar que sejam descobertos os erros em requisitos.

24.3.1 Teste de unidade em contexto orientado a objetos

Quando consideramos o software orientado a objetos, o conceito de unidades se modifica. O encapsulamento controla a definição de classes e objetos. Cada classe e cada instância de uma classe (objeto) empacotam atributos (dados) e as operações (também conhecidas como métodos ou serviços) que manipulam esses dados. Em vez de testar um módulo individual, a menor unidade testável é a classe encapsulada. Como uma classe pode conter muitas operações diferentes, e uma operação em particular pode existir como parte de um conjunto de classes diferentes, o significado do teste de unidade muda significativamente.

Não podemos mais testar uma única operação isoladamente (a visão convencional do teste de unidade), mas como parte de uma classe. Para ilustrar, considere uma hierarquia de classes na qual uma operação *X()* é definida para a superclasse e é herdada por várias subclasses. Cada subclasse usa a operação *X()*, mas ela é aplicada de acordo com o contexto de atributos privados e operações definidas para cada subclasse. O contexto no qual a operação *X()* é usada varia de maneira sutil; desse modo, é necessário testar a operação *X()* no contexto de cada uma das subclasses. Isso significa que testar a operação *X()* isoladamente (a abordagem tradicional do teste de unidade) é ineficaz no contexto orientado a objetos.

O *teste de classe* para software orientado a objetos equivale ao teste de unidade para software convencional.³ Ao contrário do teste de unidade para software convencional, que tende a focar o detalhe algorítmico de um módulo e os dados que fluem pela interface do módulo, o teste de classe para software orientado a objetos é controlado pelas operações encapsuladas pela classe e pelo comportamento de estado da classe.

A menor “unidade” que pode ser testada em software orientado a objetos é a classe. O teste de classe é controlado pelas operações encapsuladas pela classe e pelo comportamento de estado da classe.

³ Os métodos de projeto de casos de teste para classes orientadas a objetos são discutidos nas Seções 24.4 a 24.6.

24.3.2 Teste de integração em contexto orientado a objetos

Como o software orientado a objetos não tem uma estrutura de controle hierárquica, as estratégias de integração convencionais descendente (*top-down*) e ascendente (*bottom-up*) têm pouco significado. Além disso, integrar operações uma de cada vez em uma classe (a abordagem convencional da integração incremental) frequentemente é impossível devido “às interações diretas e indiretas dos componentes que formam a classe” [Ber93].

Há duas estratégias para teste de integração de sistemas orientados a objetos [Bin94a]. A primeira, *teste baseado em sequência de execução* (*thread-based testing*), integra o conjunto de classes necessárias para responder a uma entrada ou um evento do sistema. Cada sequência de execução é integrada e testada individualmente. O teste de regressão é aplicado para garantir que não ocorram efeitos colaterais. A segunda abordagem de integração, *teste baseado em uso*, inicia a construção do sistema, testando as classes (chamadas *classes independentes*) que usam bem poucas (se usar alguma) classes servidoras. Depois que as classes independentes são testadas, testa-se a próxima camada de classes, chamadas *classes dependentes*, que usam as classes independentes. Essa sequência de camadas de teste de classes dependentes continua até que todo o sistema seja construído. Diferentemente da integração convencional, o uso de pseudocontrolador e pseudocontrolados (Capítulo 23), como operações substitutas, deve ser evitado quando possível.

Teste de conjunto (*cluster testing*) [McG94] é uma etapa do teste de integração de software orientado a objetos. Nesse caso, um agregado de classes colaboradoras (determinado examinando-se os modelos CRC e o modelo objeto-relação) é exercitado projetando-se casos de teste que tentam descobrir erros nas colaborações.

O teste de integração para software orientado a objetos testa um conjunto de classes necessárias para responder a determinado evento.

24.3.3 Teste de validação em contexto orientado a objetos

No nível de validação ou de sistema, os detalhes das conexões de classes desaparecem. Assim como a validação convencional, a validação de software orientado a objetos enfoca as ações visíveis pelo usuário e as saídas do sistema reconhecíveis por ele. Para ajudar na criação de testes de validação, o testador deve fundamentar-se nos casos de uso (Capítulos 9 e 10) que fazem parte do modelo de requisitos. O caso de uso proporciona um cenário com grande possibilidade de detectar erros em requisitos de interação de usuário.

Os métodos convencionais de teste caixa-preta (Capítulo 23) podem ser usados para controlar testes de validação. Além disso, pode-se optar por criar casos de teste com base no modelo de comportamento de objeto e em um diagrama de fluxo de evento criado como parte da análise orientada a objetos.

24.4 Métodos de teste orientados a objetos

A arquitetura de software orientado a objetos resulta em uma série de subsistemas em camada que encapsulam as classes de colaboração. Cada elemento

"Encaro os testadores como os guarda-costas do projeto. Nós defendemos o flanco de nossos desenvolvedores contra as falhas, enquanto eles concentram-se na efetivação do sucesso."

James Bach

de sistema (subsistemas e classes) executa funções que ajudam a satisfazer os requisitos do sistema. É necessário testar um sistema orientado a objetos em vários níveis diferentes para descobrir erros que podem ocorrer à medida que as classes colaboram umas com as outras e os subsistemas se comunicam por meio de camadas da arquitetura.

Os métodos de projeto de casos de teste para software orientado a objetos continuam a evoluir. Uma abordagem geral para projeto de casos de teste orientado a objetos foi sugerida por Berard [Ber93]:

1. Cada caso de teste deve ser identificado de forma única e associado explicitamente à classe a ser testada.
2. A finalidade do teste deve ser definida.
3. Uma lista das etapas para cada teste deve ser feita, e ela deve conter: uma lista dos estados especificados para a classe a ser testada, uma lista das mensagens e operações que serão executadas em consequência do teste, uma lista das exceções que podem ocorrer enquanto a classe é testada, uma lista de condições externas (isto é, alterações no ambiente externo ao software que deve existir para fazer corretamente o teste) e informações complementares que ajudarão a entender ou implementar o teste.

Ao contrário do projeto convencional de casos de teste, controlado por uma visão entrada-processo-saída do software ou do detalhe algorítmico dos módulos individuais, o teste orientado a objetos concentra-se no planejamento de sequências de operação apropriadas para executar os estados de uma classe.

24.4.1 As implicações dos conceitos de orientação a objetos no projeto de casos de teste

Conforme uma classe evolui pelos modelos de análise e de projeto, ela se torna alvo para o projeto de casos de teste. Como os atributos e operações são encapsulados, as operações de teste fora da classe em geral são improdutivas. Embora o encapsulamento seja um conceito de projeto essencial para software orientado a objetos, ele pode criar um pequeno obstáculo na realização do teste. Binder [Bin94a] observa que "O teste requer um relato sobre o estado concreto e abstrato de um objeto". Ainda assim, o encapsulamento pode dificultar a obtenção dessa informação. A menos que sejam inseridas operações internas para relatar os valores dos atributos da classe, pode ser difícil obter um resumo do estado de um objeto.

A herança também pode apresentar desafios adicionais durante o projeto de casos de teste. Já vimos que a cada novo contexto de uso é exigido um novo teste, mesmo que a reutilização tenha sido possível. Além disso, a herança múltipla⁴ complica ainda mais o teste, aumentando o número de contextos para os quais o teste é necessário [Bin94a]. Se subclasses instanciadas a partir de uma superclasse forem usadas dentro do mesmo domínio do problema, é provável que o conjunto de casos de teste derivado para a

Uma excelente coleção de publicações e recursos sobre teste orientado a objetos pode ser encontrada pesquisando-se o site <https://www.thecsiac.com/>.

⁴ Um conceito de orientação a objetos que deve ser usado com extremo cuidado.

superclasse possa ser usado para testar a subclasse. No entanto, se a subclasse for usada em um contexto inteiramente diferente, os casos de teste da superclasse terão pouca aplicabilidade, e um novo conjunto de testes precisará ser projetado.

24.4.2 Aplicabilidade dos métodos convencionais de projeto de casos de teste

Os métodos de teste caixa-branca descritos no Capítulo 23 podem ser aplicados às operações definidas para uma classe. Técnicas de caminho-base, teste de ciclos ou fluxo de dados podem ajudar a garantir que cada instrução em uma operação seja testada. Entretanto, a estrutura concisa de muitas operações de classe sugere argumentos de que o esforço aplicado no teste caixa-branca poderia ser mais bem redirecionado para testes no nível de classe.

Os métodos de teste caixa-preta são adequados para sistemas orientados a objetos, assim como para sistemas desenvolvidos com métodos convencionais de engenharia de software. Conforme mencionado no Capítulo 23, casos de uso podem proporcionar informações úteis no projeto de testes caixa-preta e baseados em estado.

24.4.3 Teste baseado em falhas⁵

O objetivo do *teste baseado em falhas* dentro de um sistema orientado a objetos é projetar testes que tenham grande probabilidade de descobrir falhas plausíveis. Como o produto ou sistema deve satisfazer os requisitos do cliente, o planejamento preliminar necessário para realizar o teste baseado em falhas começa com o modelo de análise. O testador procura por falhas plausíveis (aspectos da implementação do sistema que podem resultar em defeitos). Para determinar se essas falhas existem, projetam-se casos de teste para exercitar o projeto ou código.

Sem dúvida, a eficiência dessas técnicas depende de como os testadores consideram uma falha plausível. Se falhas reais em um sistema orientado a objetos são vistas como não plausíveis, essa abordagem não é melhor do que qualquer técnica de teste aleatório. Por outro lado, se os modelos de análise e projeto puderem proporcionar conhecimento aprofundado sobre o que provavelmente pode sair errado, o teste baseado em falhas pode encontrar uma quantidade significativa de erros com esforço relativamente pequeno.

O teste de integração procura por falhas plausíveis em chamadas de operação ou conexões de mensagem. Três tipos de falhas encontram-se nesse contexto: resultado inesperado, uso de operação/mensagem errada e invocação incorreta. Para determinar as falhas plausíveis quando funções (operações) são invocadas, o comportamento da operação deve ser examinado.

A estratégia para o teste baseado em falhas é formular uma hipótese de uma série de falhas possíveis e criar testes para provar cada uma das hipóteses.

Que tipos de falhas são encontrados em chamadas de operação e nas conexões de mensagens?

⁵ As Seções 24.4.3 e 24.4.4 foram adaptadas de um artigo de Brian Marick originalmente postado na Internet no grupo comp.testing. Essa adaptação foi incluída com permissão do autor. Para mais informações sobre esses tópicos, veja [Mar94]. Deve-se notar que as técnicas discutidas nas Seções 24.4.3 e 24.4.4 também são aplicáveis para software convencional.

O teste de integração se aplica tanto a atributos quanto a operações. Os “comportamentos” de um objeto são definidos pelos valores atribuídos a seus atributos. O teste deve exercitar os atributos para determinar se ocorrem os valores apropriados para tipos distintos de comportamento de objeto.

É importante observar que o teste de integração tenta encontrar erros no objeto-cliente, não no servidor. Em termos convencionais, o foco do teste de integração é determinar se existem erros no código chamador, não no código chamado. A chamada da operação é usada como um indício, uma maneira de encontrar os requisitos de teste que usam o código chamador.

24.4.4 Projeto de teste baseado em cenários

O teste baseado em falhas omite dois tipos principais de erro: (1) especificações incorretas e (2) interações entre subsistemas. Quando ocorrem erros associados a uma especificação incorreta, o produto não realiza o que o cliente deseja. Ele pode fazer alguma coisa errada ou omitir uma funcionalidade importante. Mas, em qualquer circunstância, a qualidade (conformidade com os requisitos) é prejudicada. Erros associados à interação de subsistemas ocorrem quando o comportamento de um subsistema cria circunstâncias (por exemplo, eventos, fluxo de dados) que faz outro subsistema falhar.

O teste baseado em cenários concentra-se naquilo que o usuário faz, não no que o produto faz. Isso significa detectar as tarefas (por meio de casos de uso) que o usuário tem de executar e aplicá-las, bem como suas variantes, como testes.

Cenários descobrem erros de interação. Mas, para tanto, os casos de teste devem ser mais complexos e realistas do que os testes baseados em falhas. Os testes baseados em cenários tendem a usar vários subsistemas em um único teste (os usuários não se limitam ao uso de um subsistema de cada vez).

Testes baseados em cenários indicarão erros que ocorrem quando qualquer ator interage com o software.

24.5 Métodos de teste aplicáveis no nível de classe

O teste “no pequeno” foca uma única classe e os métodos encapsulados por ela. Teste aleatório e particionamento são métodos que podem ser usados para simular uma classe durante o teste orientado a objetos.

24.5.1 Teste aleatório para classes orientadas a objetos

Para uma breve ilustração desses métodos, considere uma aplicação bancária na qual uma classe **Conta** tem estas operações: *abrir()*, *estabelecer()*, *depositar()*, *retirar()*, *obterSaldo()*, *resumir()*, *limiteDeCrédito()* e *fechar()* [Kir94]. Cada operação pode ser aplicada para **Conta**, mas certas restrições (por exemplo, primeiro a conta precisa ser aberta, para que as outras operações possam ser aplicadas, e fechada depois que todas as operações são concluídas) são implícitas à natureza do problema. Mesmo com essas restrições, há muitas permutações das operações. O histórico de comportamento mínimo de uma instância de **Conta** inclui as seguintes operações:

abrir•estabelecer•depositar•retirar•fechar

O número de permutações possíveis para teste aleatório pode crescer muito. Uma estratégia similar ao teste de matriz ortogonal pode ser usada para melhorar a eficiência do teste.

Isso representa a sequência mínima de teste para Conta. No entanto, pode ocorrer uma ampla variedade de outros comportamentos nessa sequência:

`abrir•estabelecer•depositar•[depositar|retirar|obterSaldo|resumir|limiteDeCrédito]n•retirar•fechar`

Uma variedade de diferentes sequências de operações pode ser gerada aleatoriamente. Por exemplo:

Caso de teste r_1 : `abrir•estabelecer•depositar•obterSaldo•resumir•retirar•fechar`

Caso de teste r_2 : `abrir•estabelecer•depositar•retirar•depositar•obterSaldo•limiteDeCrédito•retirar•fechar`

Esses e outros testes de ordem aleatória podem ser usados para exercitar diferentes históricos de duração de instância de classe.

CASASEGURA



Teste de classe

Cena: Sala da Shakira.

Atores: Jamie e Shakira – membros da equipe de engenharia de software que estão trabalhando no projeto de um caso de teste para função de segurança do CasaSegura.

Conversa:

Shakira: Desenvolvi alguns testes para a classe **Detector** [Figura 14.4] – você sabe, aquela que permite acesso a todos os objetos **Sensor** para a função de segurança. Conhece?

Jamie (rindo): Claro, é aquela que você usou para acrescentar o sensor “mal-estar de cachorro”.

Shakira: Essa mesma. Bem, ela tem uma interface com quatro operações: `ler()`, `habilitar()`, `desabilitar()` e `testar()`. Para que um sensor possa ser lido, ele primeiro tem de ser habilitado. Uma vez habilitado, pode ser lido e testado. Ele pode ser desabilitado a qualquer instante, exceto se uma condição de alarme estiver sendo processada. Assim, defini uma sequência simples de teste que vai simular sua história comportamental. [Mostra a Jamie a seguinte sequência.]

#1: `habilitar•testar•ler•desabilitar`

Jamie: Vai funcionar, mas você terá de fazer mais testes do que isso.

Shakira: Eu sei, eu sei. Aqui estão algumas outras sequências que eu descobri. [Mostra a Jamie as seguintes sequências.]

#2: `habilitar•testar*[ler]n•testar•desabilitar`

#3: `[ler]n`

#4: `habilitar*desabilitar•[testar | ler]`

Jamie: Bem, deixe-me ver se entendi o objetivo dessas sequências. #1 acontece de maneira trivial, assim como um uso convencional. #2 repete a operação de leitura n vezes, e esse é um cenário provável. #3 tenta ler o sensor antes de ele ser habilitado... Isso deve produzir uma mensagem de erro de algum tipo, certo? #4 habilita e desabilita o sensor e então tenta lê-lo. Isso não é o mesmo que o teste #2?

Shakira: Na verdade, não. Em #4, o sensor foi habilitado. O que #4 realmente testa é se a operação de desabilitar funciona como deveria. Um `ler()` ou `testar()` após `desabilitar()` deve gerar uma mensagem de erro. Se isso não acontecer, temos um erro na operação desabilitar.

Jamie: Certo. Lembre-se de que os quatro testes têm de ser aplicados a cada tipo de sensor, já que todas as operações podem ser sutilmente diferentes dependendo do tipo de sensor.

Shakira: Não se preocupe. Está planejado.

24.5.2 Teste de partição em nível de classe

O teste de partição reduz o número de casos de teste necessários para simular a classe de maneira muito semelhante ao particionamento de equivalência (Capítulo 23) para software tradicional. As entradas e saídas são classificadas, e os casos de teste são projetados para exercitar cada categoria. Mas como são criadas as categorias de particionamento?

Que opções de teste estão disponíveis em nível de classe?

"A fronteira que define o escopo do teste de unidade e do teste de integração é diferente para o desenvolvimento orientado a objetos. Podem ser projetados e usados testes em muitos pontos no processo. Assim, 'projetar pequeno, codificar pequeno' torna-se 'projetar pequeno, codificar pequeno, testar pequeno'."

Robert Binder

O *particionamento baseado em estados* classifica as operações de classe baseado na sua capacidade de mudar o estado da classe. Como exemplo, considere a classe **Conta**. As operações de estado incluem *depósito()* e *retirada()*, enquanto as operações que não são de estado incluem *saldo()*, *resumo()*, e *limiteDeCrédito()*. Os testes são projetados de forma a simular operações que mudam e que não mudam o estado, separadamente. Portanto,

Caso de teste p_1 : abrir•estabelecer•depositar•depositar•retirar•retirar•fechar

Caso de teste p_2 : abrir•estabelecer•depositar•resumir•limiteDeCrédito•retirar•fechar

O caso de teste p_1 muda o estado, enquanto o caso de teste p_2 simula operações que não mudam o estado (exceto aquelas na sequência de teste mínima).

Outros tipos de teste de partição também podem ser aplicados. O *particionamento baseado em atributos* classifica operações de classe com base nos atributos que elas usam. O *particionamento baseado em categorias* classifica operações de classe com base na função genérica que cada uma executa.

24.6 Projeto de caso de teste entre classes

O projeto de caso de teste torna-se mais complicado quando começa a integração do sistema orientado a objetos. É nesse estágio que o teste de colaborações entre classes deve começar. Para ilustrar a "geração de caso de teste entre classes" [Kir94], vamos expandir o exemplo bancário apresentado na Seção 24.5 para incluir as classes e colaborações da Figura 24.2. As direções das setas na figura indicam a direção das mensagens, e os rótulos indicam as operações chamadas como consequência das colaborações sugeridas pelas mensagens.

Como o teste de classes individuais, o teste de colaboração entre classes pode ser feito com a aplicação de métodos aleatórios e de particionamento, bem como teste baseado em cenários e teste comportamental.

24.6.1 Teste de múltiplas classes

Kirani e Tsai [Kir94] sugerem a seguinte sequência de passos para gerar casos de teste aleatórios de múltiplas classes:

1. Para cada classe cliente, use a lista de operações de classe para gerar uma série de sequências aleatórias de teste. As operações enviarão mensagens para outras classes servidoras.
2. Para cada mensagem gerada, determine a classe colaboradora e a operação correspondente no objeto servidor.
3. Para cada operação no objeto servidor (que foi chamado por mensagens enviadas pelo objeto cliente), determine as mensagens que ela transmite.
4. Para cada uma das mensagens, determine o próximo nível de operações chamadas e incorpore-as na sequência de teste.

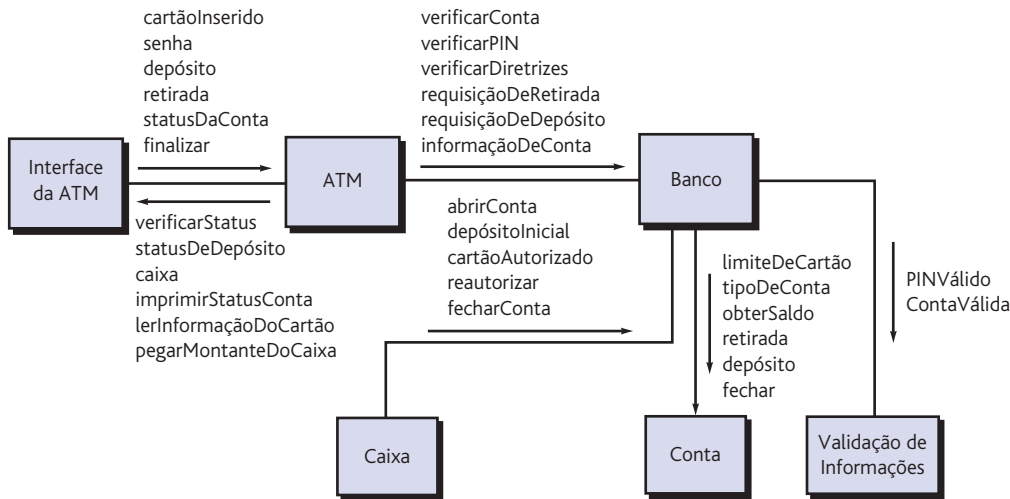


FIGURA 24.2 Diagrama de colaboração de classe para a aplicação bancária.

Fonte: Adaptado de [Kir94].

Para ilustrar [Kir94], considere uma sequência de operações para a classe **Banco** relativas a uma classe **ATM** (caixa eletrônico) (Figura 24.2):

```

verificarConta•verificarPIN•[[verificarDiretrizes•requisiçãoDeRetirada]
|requisiçãoDeDepósito|requisiçãoDeInformaçãoDeConta]n

```

Um caso de teste aleatório para a classe **Banco** poderia ser

Caso de teste r_3 = verificarConta•verificarPIN•requisiçãoDeDepósito

Para as colaborações envolvidas nesse teste, são consideradas as mensagens associadas a cada uma das operações citadas no caso de teste r_3 . **Banco** deve colaborar com **informaçãoDeValidação** para executar a *verificarConta()* e *verificarPIN()*. **Banco** deve colaborar com **Conta** para executar *requisiçãoDeDepósito()*. Daí, um novo caso de teste que exercita essas colaborações é

Caso de teste r_4 = verificarConta [Banco:contaVálidaInformaçãoDeValidação]•verificarPIN

[Banco: PINVálidoInformaçãoDeValidação]•requisiçãoDeDepósito [Banco: depósitoConta]

A abordagem para teste de partição de múltiplas classes é similar à usada para teste de partição de classes individuais. Uma classe simples é particionada conforme discutido na Seção 24.5.2. A equivalência de teste é expandida para incluir operações chamadas via mensagens para classes colaboradoras. Uma abordagem alternativa particiona os testes baseada nas interfaces para uma classe em particular. Conforme a Figura 24.2, a classe **Banco** recebe mensagens das classes **ATM** e **Caixa**. Os métodos da classe **Banco** podem, portanto, ser testados particionando-os naqueles que servem **ATM** e naqueles que servem **Caixa**. Pode ser usada a partição baseada em estado (Seção 24.5.2) para refinar ainda mais as partições.

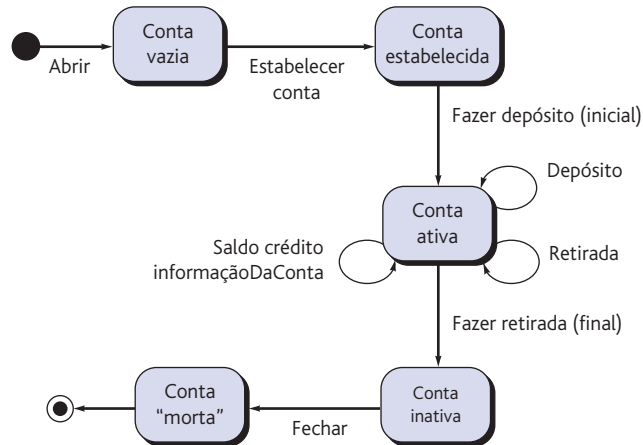


FIGURA 24.3 Diagrama de estados para a classe Conta.

Fonte: Adaptado de [Kir94].

24.6.2 Testes derivados de modelos comportamentais

O uso dos diagramas de estado como um modelo que representa o comportamento dinâmico de uma classe é discutido no Capítulo 11. O diagrama de estado de uma classe pode ser usado para ajudar a derivar uma sequência de testes que vão simular o comportamento dinâmico da classe (e as classes que colaboram com ela). A Figura 24.3 [Kir94] ilustra um diagrama de estado para a classe **Conta** discutida anteriormente. Na figura, as transações iniciais movem-se para os estados *conta vazia* e *conta estabelecida*. A maior parte do comportamento para instâncias da classe ocorre ainda no estado *conta ativa*. Uma retirada final e fechamento da conta fazem a classe **Conta** transitar para os estados *conta inativa* e *conta "morta"*, respectivamente.

Os testes a serem projetados deverão cobrir todos os estados. Isto é, as sequências de operação deverão fazer a classe **Conta** fazer a transição por todos os estados permitidos:

Caso de teste s_1 : abrir•estabelecerConta•fazerDepósito (inicial)•fazerRetirada (final)•fechar

É importante notar que essa sequência é idêntica à sequência de teste mínimo discutida na Seção 24.5.2. Acrescentando as sequências de teste adicionais à sequência mínima,

Caso de teste s_2 : abrir•estabelecerConta•fazerDepósito (inicial)•fazerDepósito•obterSaldo•obterLimiteDeCrédito•fazerRetirada (final)•fechar

Caso de teste s_3 : abrir•estabelecerConta•fazerDepósito(inicial)•fazerDepósito•fazerRetirada•informaçãoDaConta•fazerRetirada (final)•fechar

Mais casos de teste poderiam ser criados para garantir que todos os comportamentos da classe fossem adequadamente simulados. Em situações nas quais o comportamento da classe resulta em uma colaboração com uma ou mais classes, são usados diagramas de estados múltiplos para acompanhar o fluxo comportamental do sistema.

O modelo de estado pode ser percorrido de uma maneira “primeiro-em-largura” [McG94]. Nesse contexto, primeiro-em-largura implica que um caso de teste simula uma única transição, e que, quando uma nova transição deve ser testada, somente as transições testadas anteriormente são usadas.

Considere um objeto **CartãoDeCrédito** que faz parte do sistema bancário. O estado inicial de **CartãoDeCrédito** é *indefinido* (não foi fornecido nenhum número de cartão de crédito). Após ler o cartão de crédito durante uma venda, o objeto assume um estado *definido*; isto é, os atributos **número do cartão** e **data de validade**, juntamente com identificadores específicos do banco, são definidos. O cartão de crédito é *submetido* (enviado) ao ser enviado para autorização e é *aprovado* quando a autorização é recebida. A transição de um estado para outro de **CartãoDeCrédito** pode ser testada derivando casos de teste que fazem a transição ocorrer. Uma abordagem primeiro-em-largura para esse tipo de teste não simularia *submetido* antes de simular *indefinido* e *definido*. Se o fizesse, faria uso de transições que não foram testadas previamente e, portanto, violaria o critério primeiro-em-largura.

24.7 Resumo

O objetivo geral do teste orientado a objetos – encontrar o número máximo de erros com um mínimo de esforço – é idêntico ao objetivo do teste de software convencional. A estratégia e tática para o teste orientado a objetos, porém, diferem bastante. A visão do teste se amplia para incluir a revisão dos requisitos e do modelo de projeto. Além disso, o foco do teste afasta-se do componente procedural (o módulo) e aproxima-se da classe.

Como os requisitos e modelos de projeto orientados a objetos e o código-fonte resultante são semanticamente acoplados, o teste (na forma de revisões técnicas) começa durante a atividade de modelagem. Por essa razão, a revisão do CRC, da relação de objeto e dos modelos de comportamento pode ser vista como um teste de primeiro estágio.

Uma vez disponível o código, o teste de unidade é aplicado para cada classe. O projeto de testes para uma classe usa vários métodos: teste baseado em falhas, teste aleatório e teste de partição. Cada um desses métodos simula as operações encapsuladas pela classe. Sequências de teste são projetadas para garantir que sejam exercitadas as operações relevantes. O estado da classe, representado pelos valores de seus atributos, é examinado para determinar se existem erros.

O teste de integração pode ser feito com uma estratégia baseada em sequências de execução (*threads*) ou baseada em uso. O teste baseado em sequências de execução integra o conjunto de classes que colaboram para responder a uma entrada ou evento. O teste baseado em uso constrói o sistema em camadas, começando com as classes que não fazem uso de classes servidas. Métodos de projeto de caso de teste de integração também podem fazer uso de testes aleatórios e de partição. Além disso, testes baseados em cenário e derivados de modelos comportamentais podem ser usados para testar uma classe e suas colaboradoras. Uma sequência de teste acompanha o fluxo de operações por meio das colaborações de classe.

Teste de validação orientado a objetos é teste orientado a caixa-preta e pode ser realizado aplicando-se os mesmos métodos caixa-preta discutidos para software convencional. Entretanto, o teste baseado em cenários domina a validação de sistemas orientados a objeto, tornando o caso de uso um pseudo-controlador primário para teste de validação.

Problemas e pontos a ponderar

24.1 Com suas palavras, descreva por que a classe é a menor unidade adequada para teste em um sistema orientado a objetos.

24.2 Por que temos de testar novamente as subclasses instanciadas de uma classe existente se a classe existente já foi completamente testada? Podemos usar o projeto de caso de teste para a classe existente?

24.3 Por que o “teste” deve começar com análise e projeto orientado a objetos?

24.4 Crie uma série de cartões de índice CRC para o *CasaSegura* e execute os passos descritos na Seção 24.2.2 para determinar se existem inconsistências.

24.5 Qual a diferença entre estratégias baseadas em sequências de execução (*threads*) e estratégias baseadas em uso para teste de integração? Como o teste de conjunto (*cluster testing*) se encaixa?

24.6 Aplique teste aleatório e teste de particionamento a três classes definidas no projeto do sistema *CasaSegura*. Produza casos de teste que indiquem as sequências de operação que serão chamadas.

24.7 Aplique teste de múltiplas classes e testes derivados de modelo comportamental para o projeto *CasaSegura*.

24.8 Crie quatro testes adicionais usando teste aleatório e métodos de particionamento, bem como teste de múltiplas classes e testes derivados do modelo comportamental para a aplicação bancária apresentada nas Seções 24.5 e 24.6.

Leituras e fontes de informação complementares

Muitos livros sobre testes citados nas seções *Leituras e Fontes de Informação Complementares* dos Capítulos 22 e 23 discutem o teste de sistemas orientados a objetos até certo ponto. Bashir e Goel (*Testing Object-Oriented Software*, Springer, 2012), Schach (*Object-Oriented and Classical Software Engineering*, McGraw-Hill, 8ª ed., 2010) e Bruege e Dutoit (*Object-Oriented Software Engineering Using UML, Patterns, and Java*, Prentice Hall, 3ª ed., 2009) consideram o teste orientado a objetos no contexto da prática mais ampla da engenharia de software. Jorgensen (*Software Testing: A Crafts-man's Approach*, Auerbach, 3ª ed., 2008) discute técnicas formais e técnicas orientadas a objetos. Yurga (*Testing and Testability of Object-Oriented Software Systems via Metrics: A Metrics-Based Approach to the Testing Process and Testability of Object-Oriented Software Systems*, LAP Lambert, 2011), Sykes e McGregor (*Practical Guide to Testing Object-Oriented Software*, Addison-Wesley, 2001), Binder (*Testing Object-Oriented Systems*, Addison-Wesley, 1999) e Kung e seus colegas (*Testing Object-Oriented Software*, Wiley-IEEE Computer Society Press, 1998) tratam do teste orientado a objetos com bastante detalhes. Freeman e Pryce (*Growing Object-Oriented Software, Guided by Tests*, Addison-Wesley, 2009) discutem o projeto voltado a testes de software orientado a objetos. Denney (*Use Case Levels of Test: A Four-Step Strategy for Building Time and Innovation in Software Test Design*, CreateSpace Inde-

pendent Publishing, 2012) discute técnicas que podem ser aplicadas ao teste de sistemas orientados a objetos.

Uma ampla gama de fontes de informação sobre métodos de teste orientado a objetos está disponível na Internet. Uma lista atualizada de referências relevantes (em inglês) para técnicas de teste pode ser encontrada no site: **www.mhhe.com/presman**.

25

Teste de aplicações para Web

Conceitos-chave

estratégia	543
planejamento	543
qualidade, dimensões da	541
teste da interface do usuário	549
teste de banco de dados	546
teste de carga	562
teste de configuração	558
teste de conteúdo	545
teste de desempenho	560
teste de esforço (stress)	562
teste de navegação	556
teste de segurança	559
teste no nível de componente	554
testes de compatibilidade	553
testes de usabilidade	552

Há certa urgência que sempre permeia um projeto de WebApp (aplicações para Web). Os grupos de envolvidos – preocupados com a competição de outras WebApps, coagidos pelas exigências do cliente e temerosos com a perda de um nicho de mercado – fazem pressão para colocá-la online. Como resultado, as atividades técnicas que muitas vezes ocorrem mais tarde no processo, como teste, dispõem algumas vezes de um prazo muito curto. Isso pode ser um erro catastrófico. Para evitar isso, os membros da equipe precisam assegurar que cada artefato tenha alta qualidade. Wallace e seus colegas [Wal03] apontam isso quando afirmam:

O teste não deve esperar até que o projeto esteja terminado. Comece o teste antes de escrever uma linha de código. Teste constante e efetivamente, e você desenvolverá um site muito mais duradouro.

Como os requisitos e modelos de projeto não podem ser testados no sentido clássico, toda a equipe deve realizar revisões técnicas (Capítulo 20) e também testes executáveis. A intenção é descobrir e corrigir erros antes que a WebApp seja disponibilizada aos usuários.

Como garantir que o trabalho foi realizado corretamente? Embora nunca se possa ter certeza de ter executado todos os testes necessários, é possível verificar se erros foram apontados (e corrigidos). Além disso, se foi estabelecido um plano de teste, é aconselhável certificar-se de que todos os testes planejados foram realizados.

PANORAMA

O que é? O teste de WebApp é um conjunto de atividades relacionadas com um único objetivo:

descobrir erros no conteúdo, na função, na usabilidade, na navegabilidade, no desempenho, na capacidade e na segurança da WebApp. Para tanto, deve ser utilizada uma estratégia de teste que abranja as revisões e o teste executável.

Quem realiza? Os engenheiros de aplicações para Web e outros envolvidos no projeto (gerentes, clientes e usuários), todos participam do teste da WebApp.

Por que é importante? Se os usuários encontrarem erros que abalem sua credibilidade na WebApp, buscarão em outro lugar o conteúdo e a função de que precisam, e a WebApp

será um fracasso. Por essa razão, deve-se trabalhar para eliminar tantos erros quantos forem possíveis antes que a WebApp entre no ar.

Quais são as etapas envolvidas? O processo de teste começa focando os aspectos visíveis da WebApp ao usuário e passa para os testes de tecnologia e infraestrutura. Executam-se sete etapas de teste: de conteúdo, interface, navegação, componente, configuração, desempenho e segurança.

Qual é o artefato? Em alguns casos é produzido um plano de teste de uma WebApp. Em todos os casos, é desenvolvida uma série de casos de teste para todas as etapas, e é mantido um arquivo dos resultados para uso futuro.

25.1 Conceitos de teste para WebApps

O teste é um processo pelo qual se experimenta o software com a intenção de encontrar (e, por fim, corrigir) erros. Essa filosofia fundamental, apresentada inicialmente no Capítulo 22, não muda para as WebApps. Na verdade, pelo fato de os sistemas e aplicações baseados na Web residirem em uma rede e interoperarem com muitos sistemas operacionais, navegadores (ou outros dispositivos de comunicação), plataformas de hardware, protocolos de comunicação e aplicações “de retaguarda” diferentes, a procura dos erros representam um desafio significativo para os engenheiros.

Para entender os objetivos do teste no contexto da engenharia de Web, deve-se considerar as muitas dimensões da qualidade da WebApp.¹ Aqui, consideramos as dimensões de qualidade particularmente relevantes em qualquer discussão de teste de WebApp. Consideramos também a natureza dos erros encontrados como consequência do teste e a estratégia de teste aplicada para descobrir esses erros.

25.1.1 Dimensões da qualidade

A qualidade é incorporada a uma aplicação Web como consequência de um bom projeto. Ela é avaliada aplicando-se uma série de revisões técnicas que investigam vários elementos do modelo de projeto e utilizando-se um processo de teste que é discutido no decorrer deste capítulo. Revisões e teste examinam uma ou mais das seguintes dimensões da qualidade [Mil00a]:

- O *conteúdo* é avaliado em nível sintático e semântico. No nível sintático, examina-se a ortografia, pontuação e gramática em documentos baseados em texto. No nível semântico, são analisadas: exatidão (das informações apresentadas), consistência (por todo o objeto de conteúdo e objetos relacionados) e ausência de ambiguidade.
- A *função* é testada para descobrir erros que indicam falta de conformidade com os requisitos do cliente. Cada função da WebApp é avaliada quanto à exatidão, instabilidade e conformidade geral com os padrões apropriados de implementação (por exemplo, padrões Java ou AJAX).
- A *estrutura* é avaliada para assegurar o fornecimento apropriado de conteúdo e função da WebApp, que seja extensível e que possa ser mantido na medida em que novo conteúdo ou nova funcionalidade são acrescentados.
- A *usabilidade* é testada para garantir que cada categoria de usuário seja suportada pela interface e que possa aprender e aplicar toda a sintaxe e semântica de navegação necessárias.
- A *navegabilidade* é testada para assegurar que toda a sintaxe e semântica de navegação sejam experimentadas para descobrir quaisquer erros de navegação (por exemplo, links inativos, impróprios, errados).

Como investigamos a qualidade no contexto de uma WebApp e de seu ambiente?

¹ As dimensões de qualidade de software genérico, igualmente aplicável a WebApps, são discutidas no Capítulo 19.

"Inovação é um negócio agriçoce para os testadores de software. Quando parece que sabemos como testar uma tecnologia em particular, uma nova tecnologia [WebApps] aparece e todas as apostas perdem o sentido."

James Bach

- O *desempenho* é testado sob uma variedade de condições de operação, configurações e carga para assegurar que o sistema responda à interação com o usuário e suporte cargas extremas sem degradação inaceitável da operação.
- A *compatibilidade* é testada executando-se a WebApp em uma variedade de diferentes configurações hospedeiras, tanto no lado do cliente quanto no lado do servidor. A finalidade é encontrar erros específicos de determinada configuração de hospedeira.
- A *interoperabilidade* é testada para garantir que a WebApp tenha uma interface adequada com outras aplicações e/ou bancos de dados.
- A *segurança* é testada para investigar vulnerabilidades em potencial e tentar explorar cada uma delas. Qualquer tentativa de invasão bem-sucedida é considerada uma falha de segurança.

Desenvolveram-se estratégias e táticas para teste de WebApp para experimentar cada uma dessas dimensões da qualidade – e elas serão discutidas no restante deste capítulo.

25.1.2 Erros em um ambiente WebApp

Os erros encontrados em consequência de um teste de WebApp bem-sucedido têm uma série de características especiais [Ngu00]:

O que torna os erros encontrados durante a execução de WebApp de certa forma diferentes daqueles identificados em software convencional?

1. Devido a muitos tipos de testes de WebApp detectarem problemas evidenciados primeiro no lado do cliente (isto é, via interface implementada em um navegador específico ou em um dispositivo de comunicação pessoal), muitas vezes nos deparamos com o indício de erro e não com o erro em si.
2. Devido a uma WebApp ser implementada em diferentes configurações e ambientes, pode ser difícil ou impossível reproduzir um erro fora do ambiente no qual foi encontrado originalmente.
3. Embora alguns erros sejam resultado de projeto incorreto ou codificação HTML (ou outra linguagem de programação) incorreta, muitos erros podem ser atribuídos à configuração da WebApp.
4. Devido às WebApps residirem em uma arquitetura cliente-servidor, os erros podem ser difíceis de localizar por meio das três camadas de arquitetura: o cliente, o servidor ou a própria rede.
5. Alguns erros são decorrentes do *ambiente operacional estático* (a configuração específica na qual o teste é executado), enquanto outros são atribuíveis ao ambiente operacional dinâmico (a carga instantânea de recursos ou erros relacionados ao tempo).

Esses cinco atributos de erros sugerem que o ambiente tem um papel importante no diagnóstico de todos os erros encontrados durante o teste da WebApp. Em algumas situações (por exemplo, teste de conteúdo), o local dos erros é óbvio, mas, em muitos outros tipos de teste de WebApp (por exemplo, teste de navegação, teste de desempenho, teste de segurança), a causa subjacente do erro pode ser consideravelmente mais difícil de determinar.

25.1.3 Estratégia de teste

A estratégia para teste de WebApp adota os princípios básicos para todo teste de software (Capítulo 22) e aplica estratégia e táticas recomendadas para sistemas orientados a objetos (Capítulo 24). Os passos a seguir resumem a abordagem:

1. O modelo de conteúdo para a WebApp é revisto para descobrir erros.
2. O modelo de interface é revisto para garantir que todos os casos de uso possam ser acomodados.
3. O modelo de projeto da WebApp é revisto para descobrir erros de navegação.
4. A interface do usuário é testada para descobrir erros nos mecanismos de apresentação e/ou navegação.
5. Os componentes funcionais são submetidos a testes de unidade.
6. É testada a navegação por toda a arquitetura.
7. A WebApp é implementada em uma variedade de configurações ambientais diferentes e testada quanto à compatibilidade com cada configuração.
8. São executados testes de segurança na tentativa de explorar vulnerabilidades na WebApp ou em seu ambiente.
9. São realizados testes de desempenho.
10. A WebApp é testada por uma população de usuários controlados e monitorados; os resultados de suas interações com o sistema são avaliados quanto a erros de conteúdo e navegação, preocupações de usabilidade, preocupações de compatibilidade, segurança, confiabilidade e desempenho da WebApp.

A estratégia geral para teste de WebApp pode ser resumida nos dez passos descritos aqui.

Excelentes artigos sobre teste de WebApp podem ser encontrados em www.stickyminds.com/testing.asp.

Como muitas WebApps evoluem continuamente, o processo de teste é uma atividade contínua, executada por pessoal de suporte que usa testes de regressão derivados dos desenvolvidos quando a WebApp foi inicialmente criada.

25.1.4 Planejamento de teste

O uso da palavra *planejamento* (em qualquer contexto) é anátema para alguns desenvolvedores Web. Estes não planejam; simplesmente iniciam – na esperança de que surja uma WebApp poderosa. Uma abordagem mais disciplinada reconhece que o planejamento estabelece um roteiro para todo o trabalho a ser feito. Isso compensa o esforço. Em seu livro sobre teste de WebApp, Splaine e Jaskiel [Spl01] afirmam:

Exceto para o mais simples dos sites, torna-se imediatamente aparente que algum tipo de planejamento de teste é necessário. Com muita frequência, o número inicial de erros encontrados por meio de um teste eventual é tão grande que nem todos são corrigidos quando detectados. Isso apresenta uma dificuldade adicional para os testadores de sites e aplicações. Eles devem não apenas evocar novos testes criativos, mas também lembrar como os anteriores foram executados para testar novamente o site/aplicação de forma confiável e assegurar que os erros conhecidos tenham sido removidos e que novos erros não tenham sido introduzidos.

O plano de teste identifica um conjunto de tarefas de teste, os artefatos a serem desenvolvidos e a maneira pela qual os resultados devem ser avaliados, registrados e reutilizados.

As perguntas a serem feitas são: como “podemos evocar novos testes criativos” e o que esses testes devem focalizar? As respostas a essas perguntas estão contidas em um plano de teste.

Um plano de teste de WebApp identifica (1) o conjunto de tarefas² a ser aplicado quando o teste começa, (2) os artefatos que serão produzidos na medida em que cada tarefa de teste for executada e (3) a maneira pela qual os resultados do teste serão avaliados, registrados e reutilizados quando for executado o teste de regressão. Em alguns casos, o plano de teste é integrado ao plano de projeto. Em outros, o plano de teste é um documento separado.

25.2 O processo de teste – uma visão geral

“Em geral, as técnicas de teste de software usadas em outras aplicações são as mesmas que as utilizadas em aplicações baseadas na Web... A diferença entre os dois tipos de teste é que a tecnologia varia na multiplicidade do ambiente Web.”

Hung Nguyen

O processo de teste da WebApp começa com testes que experimentam o conteúdo e a funcionalidade da interface, pontos imediatamente visíveis para os usuários. Na medida em que o teste é realizado, são experimentados aspectos da arquitetura de projeto e da navegação da WebApp. Por fim, o foco muda para testes que examinam os recursos tecnológicos que nem sempre são aparentes para os usuários – tópicos de infraestrutura e de instalação/implementação da WebApp.

A Figura 25.1 justapõe o processo de teste da WebApp com a pirâmide de projeto para WebApps (Capítulo 17). Observe que, enquanto ocorre o fluxo do

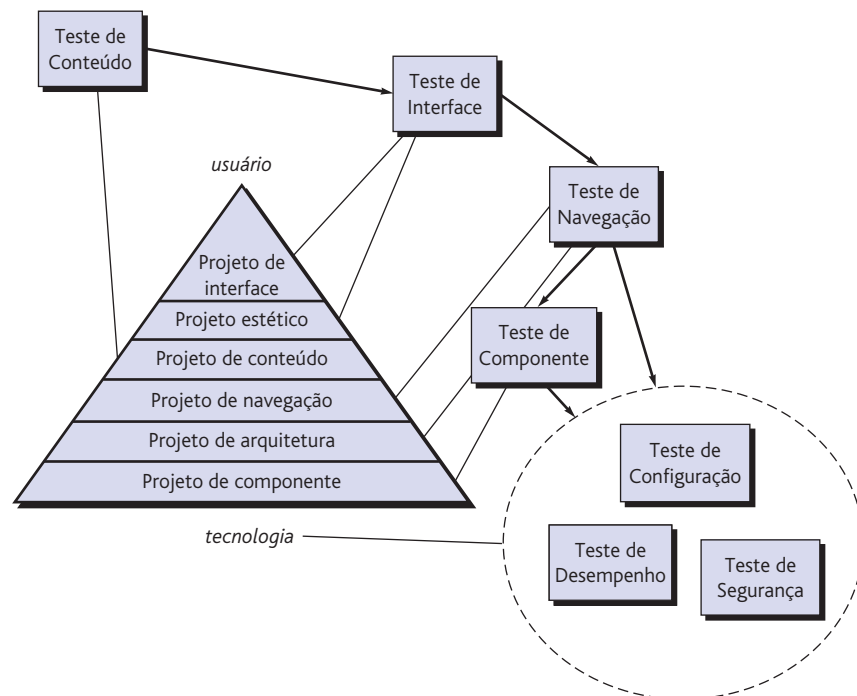


FIGURA 25.1 O processo de teste.

² Conjuntos de tarefas são discutidos no Capítulo 3. Um termo relacionado – *fluxo de trabalho* – também é usado para descrever uma série de tarefas necessárias para exercer uma atividade de engenharia de software.

teste da esquerda para a direita e de cima para baixo, os elementos do projeto da WebApp visíveis para o usuário (elementos do topo da pirâmide) são testados primeiro, seguidos pelos elementos de projeto da infraestrutura.

25.3 Teste de conteúdo

Erros no conteúdo da WebApp podem ser tão triviais quanto pequenos erros tipográficos ou muito significativos, como informações incorretas, organização inadequada ou violação de leis de propriedade intelectual. O teste de conteúdo tenta descobrir esses e muitos outros problemas antes que sejam encontrados pelos usuários.

Ele combina tanto revisões quanto geração de casos de testes executáveis. A revisão é aplicada para descobrir erros semânticos no conteúdo (discutido na Seção 25.3.1). O teste executável é usado para descobrir erros de conteúdo que podem ser atribuídos a conteúdo extraído dinamicamente, controlado por dados adquiridos de um ou mais banco de dados.

25.3.1 Objetivos do teste de conteúdo

O teste de conteúdo tem três importantes objetivos: (1) descobrir erros de sintaxe (por exemplo, erros ortográficos, erros gramaticais) em documentos de texto, representações gráficas e outros meios; (2) descobrir erros de semântica (isto é, erros na exatidão ou integralidade das informações) em qualquer objeto de conteúdo apresentado quando ocorre a navegação; e (3) encontrar erros na organização ou estrutura do conteúdo apresentado ao usuário.

Para atingir o primeiro objetivo, usam-se verificadores automáticos de ortografia e gramática. Porém, muitos erros de sintaxe fogem à detecção dessas ferramentas e devem ser descobertos por um revisor humano (testador). De fato, um site grande pode contratar os serviços de um revisor profissional para descobrir erros de ortografia, erros de gramática, erros na coesão do conteúdo, erros em representações gráficas e erros de referência cruzada.

O teste de semântica concentra-se nas informações apresentadas em cada objeto de conteúdo. O revisor (testador) deve responder às seguintes questões:

- As informações são efetivamente precisas?
- As informações são concisas e direcionadas ao assunto?
- É fácil para o usuário entender o layout do objeto de conteúdo?
- As informações contidas em um objeto de conteúdo podem ser encontradas facilmente?
- Foram fornecidas referências apropriadas para todas as informações derivadas de outras fontes?
- As informações apresentadas são coesas internamente e coerentes com as informações apresentadas em outros objetos de conteúdo?
- O conteúdo é ofensivo, confuso ou dá margem a litígio?

Embora as revisões técnicas não façam parte do teste, a de conteúdo deve ser executada para garantir que o conteúdo tenha qualidade.

Os objetivos do teste de conteúdo são: (1) descobrir erros de sintaxe no conteúdo, (2) descobrir erros de semântica e (3) encontrar erros estruturais.

Quais questões devem ser formuladas e respondidas para descobrir erros de semântica no conteúdo?

- O conteúdo desrespeita os direitos autorais ou de marcas registradas existentes?
- O conteúdo contém links que complementam o conteúdo existente? Os links estão corretos?
- O estilo estético do conteúdo está em conflito com o estilo estético da interface?

Obter respostas para cada uma dessas perguntas para uma WebApp grande (contendo centenas de objetos de conteúdo) pode ser uma tarefa assustadora. No entanto, se não forem descobertos os erros de semântica, será abalada a confiança do usuário na WebApp, e isso pode levar ao fracasso da aplicação.

Objetos de conteúdo existem em uma arquitetura que tem um estilo específico (Capítulo 17). Durante o teste de conteúdo, a estrutura e a organização da arquitetura de conteúdo são verificadas para assegurar que o conteúdo necessário seja apresentado ao usuário na ordem e nas relações apropriadas. Por exemplo, a WebApp **CasaSeguraGarantida.com** apresenta uma variedade de informações sobre sensores usados como parte dos produtos de segurança e vigilância. Objetos de conteúdo fornecem informações descritivas, especificações técnicas, uma representação fotográfica e informações relacionadas. Os testes da arquitetura de conteúdo do **CasaSeguraGarantida.com** procuram descobrir erros na apresentação dessas informações (por exemplo, uma descrição do Sensor X é apresentada com uma foto do Sensor Y).

25.3.2 Teste de banco de dados

As WebApps modernas fazem muito mais do que apresentar objetos de conteúdo estáticos. Em muitos domínios de aplicação, fazem interface com sofisticados sistemas de gerenciamento de banco de dados e criam objetos de conteúdo dinâmico em tempo real usando os dados adquiridos de um banco de dados.

Por exemplo, uma WebApp de serviços financeiros pode produzir informações complexas baseadas em texto, na forma tabular e gráfica sobre um fundo específico (por exemplo, um fundo de ações ou fundo mútuo). O objeto de conteúdo composto que apresenta essas informações é criado dinamicamente quando o usuário solicita informações sobre um fundo específico. Para tanto, são necessários os seguintes passos: (1) é consultado um banco de dados de fundos, (2) são extraídos os dados relevantes do banco de dados, (3) os dados extraídos devem ser organizados como um objeto de conteúdo, e (4) esse objeto de conteúdo (representando informações personalizadas requisitadas por um usuário) é transmitido para o ambiente do cliente para ser apresentado. Erros podem ocorrer, e efetivamente ocorrem, em consequência de cada uma dessas etapas. O objetivo do teste de banco de dados é descobrir erros, mas esse tipo de teste é complicado por uma variedade de fatores:

Quais são os tópicos que complicam o teste de banco de dados para WebApps?

1. *A solicitação original de informações do lado do cliente raramente é apresentada de maneira (por exemplo, linguagem de consulta estruturada, SQL, structured query language) que possa ser colocada em um sistema*

INFORMAÇÕES



Ferramentas de teste de conteúdo Web

Objetivo: O objetivo das ferramentas de teste de conteúdo Web é identificar erros que impedem uma página Web de exibir conteúdo de forma legível e organizada.

Mecanismos: Normalmente, essas ferramentas solicitam a URL de um recurso Web a ser testado. Então, cada ferramenta fornece uma lista de erros (por exemplo, não seguir o padrão da linguagem de marcação), com sugestões sobre como corrigi-los.

Ferramentas representativas:³

<http://validator.w3.org/> – A ferramenta online WC3 verifica a validade da linguagem de marcação (HTML, XHTML, SMIL, MathML) de páginas Web.

<http://jigsaw.w3.org/css-validator/> – Ferramenta online WC3 que verifica folhas de estilo CSS e documentos usando folhas de estilo CSS.

<http://validator.w3.org/feed/> – Ferramenta online WC3 que verifica a sintaxe de feeds Atom ou RSS.

de gerenciamento de banco de dados (DBMS, *database management system*). Portanto, devem ser projetados testes para descobrir erros de tradução da solicitação do usuário em uma forma que possa ser processada por esse DBMS.

2. O banco de dados pode ser remoto ao servidor que abriga a WebApp. Portanto, devem ser desenvolvidos testes que descubram erros na comunicação entre a WebApp e o banco de dados remoto.⁴

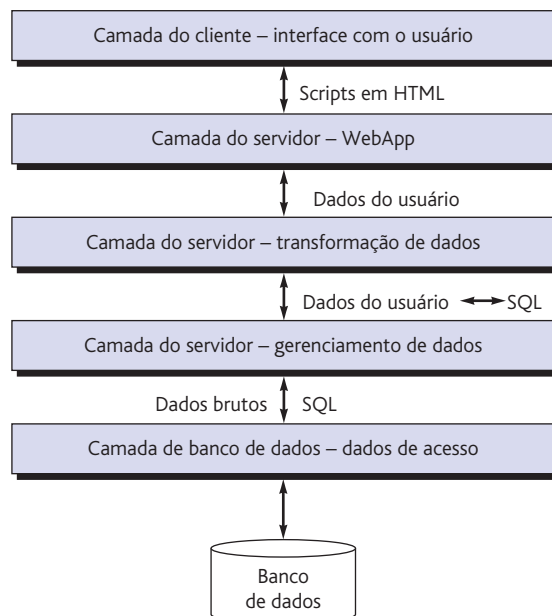


FIGURA 25.2 Camadas de interação.

³ A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

⁴ Esses testes podem se tornar complexos quando são encontrados bancos de dados distribuídos ou quando é necessário acesso a um armazém de dados (Capítulo 1).

3. *Dados brutos adquiridos do banco de dados devem ser transmitidos para o servidor da WebApp e formatados corretamente para subsequente transmissão ao cliente.* Portanto, devem ser criados testes que demonstrem a validade dos dados brutos recebidos pelo servidor da WebApp e também devem ser criados testes adicionais que demonstrem a validade das transformações aplicadas a esses dados para criar objetos de conteúdo válidos.
4. *O(s) objeto(s) de conteúdo dinâmico deve(m) ser transmitido(s) ao cliente de maneira que possa(m) ser apresentado(s) ao usuário final.* Portanto, uma série de testes deverá ser projetada para (1) descobrir erros no formato do objeto de conteúdo e (2) testar a compatibilidade com diferentes configurações de ambiente.

Considerando esses quatro fatores, os métodos de projeto de casos de teste devem ser aplicados a cada uma das “camadas de interação” [Ngu01] observadas na Figura 25.2. O teste deve assegurar que (1) sejam passadas informações válidas entre o cliente e o servidor por meio da camada de interface, (2) a WebApp processa scripts corretamente e extrai ou formata adequadamente os dados do usuário, (3) os dados do usuário são passados corretamente para uma função de transformação de dados no lado servidor que formata apropriadamente as consultas (por exemplo, SQL), (4) as consultas são passadas para uma camada de gerenciamento de dados⁵ que se comunica com as rotinas de acesso a banco de dados (possivelmente localizadas em outras máquinas).

As camadas de transformação de dados, gerenciamento de dados e acesso a banco de dados da Figura 25.2 muitas vezes são construídas com componentes reutilizáveis que foram validados separadamente e como um pacote. Se for esse o caso, o teste da WebApp focaliza o projeto de casos de testes para experimentar as interações entre a camada do cliente e as duas primeiras camadas de servidor (WebApp e transformação de dados) mostradas na figura.

A camada da interface de usuário é testada para garantir que os scripts sejam construídos corretamente para cada consulta de usuário e adequadamente transmitidos para o lado servidor. A camada da WebApp no lado do servidor é testada para assegurar que os dados do usuário sejam adequadamente extraídos dos scripts e transmitidos corretamente para a camada de transformação de dados no lado do servidor. As funções de transformação de dados são testadas para assegurar que o código SQL correto seja criado e passado para os componentes de gerenciamento de dados apropriados.

Uma discussão detalhada sobre a tecnologia subjacente que deve ser entendida para se projetar adequadamente esses testes de banco de dados está além dos objetivos deste livro. Para mais detalhes, consulte [Sce02], [Ngu01] e [Bro01].

“Não confiamos em um site que sofre de frequentes paradas, trava no meio de uma transação ou tem pouco senso de utilidade. O teste, portanto, tem um papel crucial no processo geral de desenvolvimento.”

Wing Lam

⁵ A camada de gerenciamento de dados tipicamente incorpora uma interface SQL em nível de chamada (SQL-CLI), como, por exemplo, o Microsoft OLE/ADO ou Java Database Connectivity (JDBC).

25.4 Teste da interface do usuário

A verificação e a validação de uma interface de usuário de uma WebApp ocorre em três pontos distintos. Durante a análise, o modelo de interface é revisado para assegurar que esteja em conformidade com os requisitos dos envolvidos e com outros elementos do modelo de requisitos. Durante o projeto, o modelo de projeto de interface é revisado para assegurar que critérios genéricos de qualidade estabelecidos para todas as interfaces de usuário (Capítulo 15) tenham sido satisfeitos e que os problemas de projeto de interface específicos da aplicação tenham sido corretamente resolvidos. Durante o teste, o foco passa para a execução de aspectos da interação com o usuário específicos da aplicação à medida que são manifestados pela sintaxe e semântica da interface. Além disso, o teste fornece uma avaliação final de usabilidade.

Com exceção de aspectos específicos orientados para a WebApp, a estratégia de interface observada aqui é aplicável a todos os tipos de software cliente-servidor.

25.4.1 Estratégia de teste de interface

O teste de interface experimenta mecanismos de interação e valida aspectos estéticos da interface de usuário. A estratégia geral é (1) descobrir erros relacionados a mecanismos específicos da interface (por exemplo, erros na execução apropriada de um link de menu ou na maneira como os dados são colocados em um formulário) e (2) descobrir erros na maneira como a interface implementa a semântica de navegação, funcionalidade da WebApp ou exibição de conteúdo. Para essa estratégia, há uma série de objetivos a ser atingidos:

- *Características da interface são testadas para garantir que as regras de projeto, estética e conteúdo visual relacionado estejam disponíveis para o usuário sem erro.*
- *Mecanismos individuais da interface são testados de maneira análoga ao teste de unidade.* Por exemplo, são projetados testes para experimentar todos os formulários, scripts no lado do cliente, HTML dinâmico, scripts, conteúdo concatenado e mecanismos de interface específicos da aplicação (por exemplo, um carrinho de compras para uma aplicação de E-commerce).
- *Cada mecanismo da interface é testado de acordo com o contexto de um caso de uso ou de uma unidade semântica de navegação, ou NSU (Capítulo 17), para uma categoria de usuário específica.*
- *A interface completa é testada em relação aos casos de uso selecionados e NSUs para descobrir erros na semântica da interface.* É nesse estágio que uma série de testes de usabilidade é executada.
- *A interface é testada segundo uma variedade de ambientes (por exemplo, navegadores) para assegurar que sejam compatíveis.*

25.4.2 Teste de mecanismos de interface

Quando um usuário interage com uma WebApp, a interação ocorre por meio de um ou mais mecanismos de interface. Uma rápida revisão das considerações para cada mecanismo de interface é apresentada nos próximos parágrafos [Spl01].

O teste de links externos deve ocorrer durante toda a vida da WebApp. Como parte de estratégia de manutenção, testes de links devem ser agendados regularmente.

Links. Cada link de navegação é testado para assegurar que o objeto de conteúdo ou a função apropriados sejam acessados.⁶ O teste inclui links associados ao layout da interface (por exemplo, barras de menu, itens de índice, links dentro de cada objeto de conteúdo e links para WebApps externas).

Formulários. Em nível macroscópico, são executados testes para assegurar que (1) rótulos identifiquem corretamente os campos do formulário e que os campos obrigatórios sejam identificados visualmente para o usuário, (2) o servidor receba todas as informações contidas no formulário e que não seja perdido nenhum dado na transmissão entre o cliente e o servidor, (3) sejam usadas escolhas padrão (*defaults*) apropriadas quando o usuário não selecionar de um menu pull down ou por meio de uma série de botões, (4) funções do navegador (por exemplo, a seta de retorno) não corrompam os dados introduzidos em um formulário e (5) scripts que realizam verificação de erros sobre dados introduzidos funcionem adequadamente e proporcionem mensagens de erro coerentes.

Em um nível mais específico, os testes devem assegurar que (1) os campos do formulário tenham tamanhos e tipos de dados corretos, (2) o formulário estabeleça proteções apropriadas que impeçam que o usuário digite cadeias de texto mais longas do que um comprimento máximo predefinido, (3) todas as opções apropriadas para menus pull-down sejam especificadas e ordenadas de maneira que tenha significado para o usuário, (4) os recursos de “preenchimento automático” do navegador não causem erros nas entradas de dados e (5) a tecla Tab (ou alguma outra tecla) faça a movimentação apropriada entre os campos do formulário.

Script no lado do cliente. Os testes caixa-preta são realizados para descobrir quaisquer erros no processamento quando o script é executado. Eles são muitas vezes acoplados ao teste de formulário, porque, em geral, entrada de script é derivada de dados fornecidos como parte de processamento de formulários.

HTML dinâmico. Cada página Web que contenha HTML dinâmico é executada para assegurar que a exibição dinâmica esteja correta. Além disso, deve ser feito um teste de compatibilidade para garantir que o HTML funcione corretamente nas configurações ambientais que suportam a WebApp.

Janelas pop-up. Uma série de testes garante que (1) o pop-up esteja dimensionado e posicionado corretamente, (2) o pop-up não cubra a janela original da WebApp, (3) o projeto estético do pop-up esteja de acordo com o da interface e (4) barras de rolagem e outros mecanismos de controle acrescentados ao pop-up estejam corretamente localizados e funcionem conforme desejado.

Scripts CGI. São feitos testes caixa-preta com ênfase na integridade dos dados (quando os dados são passados para o script CGI) e processamento de script (uma vez que dados validados tenham sido recebidos). Além disso, pode ser feito teste de desempenho para assegurar que a configuração do lado ser-

Os testes de script no lado do cliente e os testes associados a HTML dinâmico devem ser repetidos sempre que uma nova versão de navegador popular for lançada.

⁶ Esses testes podem ser realizados tanto como parte do teste de interface quanto do de navegação.

vidor possa acomodar as demandas de processamento de múltiplas chamadas de scripts CGI [Spl01].

Conteúdo encadeado (*streaming*). Os testes devem demonstrar que os dados encadeados são atualizados, exibidos corretamente e que podem ser suspensos sem erro e reiniciados sem dificuldade.

Cookies. São necessários testes do lado do servidor e do lado do cliente. No lado do servidor, devem assegurar que um cookie seja construído (contenha dados corretos) e transmitido corretamente para o lado do cliente, quando solicitado um conteúdo ou uma funcionalidade específica. Além disso, a persistência apropriada do cookie é testada para assegurar que sua data de expiração esteja correta. No lado do cliente, testes determinam se a WebApp anexa corretamente os cookies existentes de acordo com a solicitação específica (enviada ao servidor).

Mecanismos de interface específicos de aplicativo. Os testes seguem uma checklist de funcionalidade e características definidas pelo mecanismo de interface. Por exemplo, Splaine e Jaskiel [Spl01] sugerem a seguinte checklist para funcionalidade do carrinho de compras definido para uma aplicação de e-commerce:

- Faça o teste de fronteiras (Capítulo 23) do número mínimo e máximo de itens que podem ser colocados no carrinho de compras.
- Teste uma solicitação de “saída” para um carrinho de compras vazio.
- Teste a remoção apropriada de um item do carrinho de compras.
- Teste para determinar se uma compra esvazia todo o conteúdo de um carrinho de compras.
- Teste para determinar a persistência do conteúdo do carrinho de compras (isso deve ser especificado como parte dos requisitos do cliente).
- Teste para determinar se a WebApp pode restaurar o conteúdo do carrinho de compras em alguma data futura (supondo que não foi feita nenhuma compra).

25.4.3 Teste da semântica da interface

Uma vez que cada mecanismo de interface tenha sido testado em termos de “unidade”, o foco muda para a semântica da interface. O teste de semântica da interface “avalia quão bem o projeto se preocupa com os usuários, oferece diretrizes claras, fornece realimentação e mantém o padrão de linguagem e abordagem” [Ngu00].

Uma revisão rigorosa do modelo de projeto de interface pode proporcionar respostas parciais às questões do parágrafo anterior. No entanto, uma vez implementada a WebApp, cada cenário de caso de uso (para cada categoria de usuário) deve ser testado. Basicamente, um caso de uso torna-se a entrada para o projeto de uma sequência de testes. A finalidade da sequência de testes é descobrir erros que impedirão o usuário de atingir o objetivo associado ao caso de uso.

Uma boa orientação para teste de usabilidade pode ser encontrada em <http://www.keyrelevance.com/articles/usability-tips.htm>.

Quais características da usabilidade tornam-se o foco do teste e quais objetivos específicos são considerados?

25.4.4 Testes de usabilidade

O testes de usabilidade são similares aos de semântica da interface (Seção 25.4.3), pois também avaliam o grau com o qual os usuários podem interagir efetivamente com a WebApp e o grau com que a WebApp dirige as ações do usuário, proporciona uma boa realimentação e reforça uma abordagem de interação coerente. Em vez de concentrar-se intencionalmente na semântica de algum objetivo interativo, as revisões e testes de usabilidade são projetados para determinar o grau com o qual a interface da WebApp facilita a vida do usuário.⁷

Invariavelmente o projetista contribuirá para o projeto dos testes de usabilidade, mas eles serão feitos pelos usuários. O teste de usabilidade pode ocorrer em diferentes níveis de abstração: (1) pode ser investigada a usabilidade de um mecanismo específico da interface (por exemplo, um formulário), (2) pode ser investigada a usabilidade de uma página Web completa (abrangendo mecanismos de interface, objetos de dados e funções relacionadas) ou (3) pode ser considerada a usabilidade da WebApp completa.

O primeiro passo no teste de utilidade é identificar uma série de categorias de usabilidade e estabelecer objetivos para cada uma das categorias. As seguintes categorias de teste e objetivos (escritos na forma de perguntas) ilustram tal abordagem:⁸

Interatividade – Os mecanismos de interação (por exemplo, menus pull down, botões, ponteiros) são fáceis de entender e usar?

Layout – Os mecanismos de navegação, conteúdo e funções são colocados de maneira que permita ao usuário encontrá-los rapidamente?

Clareza – O texto é bem escrito e fácil de ser entendido?⁹ As representações gráficas são fáceis de entender?

Estética – O layout, a cor, o tipo de letra e características relacionadas facilitam o uso? Os usuários se sentem “confortáveis” com a aparência e comportamento da WebApp?

Características da tela – A WebApp otimiza o uso do tamanho e da resolução da tela?

Sensibilidade ao tempo – Características importantes, funções e conteúdo podem ser usados ou acessados no tempo oportuno?

Personalização – A WebApp se adapta a necessidades específicas de diferentes categorias de usuário ou de usuários individuais?

Acessibilidade – A WebApp é acessível a pessoas com necessidades especiais?

⁷ O termo *amigável ao usuário* tem sido usado nesse contexto. O problema, naturalmente, é que a percepção de um usuário sobre uma interface “amigável” pode ser radicalmente diferente da de outro usuário.

⁸ Para mais informações sobre usabilidade, consulte o Capítulo 15.

⁹ O FOG Readability Index e outros podem ser usados para dar uma visão quantitativa da clareza. Veja mais detalhes em <http://developer.gnome.org/gdp-style-guide/stable/usability-readability.html.en>.

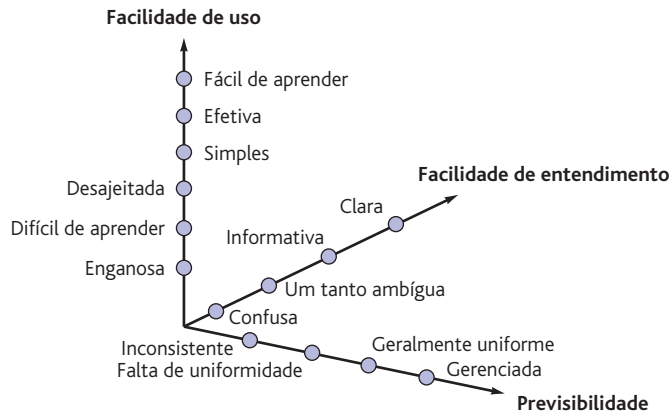


FIGURA 25.3 Avaliação qualitativa da usabilidade.

Em cada uma dessas categorias é projetada uma série de testes. Em alguns casos, o “teste” pode ser uma revisão visual de uma página Web. Em outros, testes de semântica da interface podem ser executados novamente, mas nesse caso os problemas de usabilidade são essenciais.

Como exemplo, consideramos a avaliação da usabilidade para mecanismos de interação e interface. Constantine e Lockwood [Con99] sugerem que a seguinte lista de características de interface deve ser revista e testada quanto à usabilidade: animação, botões, cores, controle, diálogo, campos, formulários, molduras, gráficos, rótulos, links, menus, mensagens, páginas de navegação, seletores, texto e barras de ferramentas. À medida que cada característica é avaliada, ela é classificada em uma escala qualitativa pelos usuários que estão fazendo o teste. A Figura 25.3 mostra um conjunto possível de “graus” de avaliação que podem ser selecionados pelos usuários. Esses graus são aplicados a cada característica individualmente, para uma página Web completa ou para a WebApp como um todo.

25.4.5 Testes de compatibilidade

Diferentes computadores, dispositivos de imagem, sistemas operacionais, navegadores e velocidades de conexão de rede podem ter influência significativa na operação da WebApp. Cada configuração de computador pode resultar em diferenças nas velocidades de processamento no lado do cliente, resoluções de tela e velocidades de conexão. Excentricidades de sistemas operacionais podem causar problemas no processamento da WebApp. Diferentes navegadores às vezes produzem resultados ligeiramente diferentes, independentemente do grau de padronização HTML na WebApp. Os plug-ins necessários podem ou não estar disponíveis para determinada configuração. O *teste de compatibilidade* procura descobrir esses problemas antes que a WebApp entre no ar (fique online).

O primeiro passo no teste de compatibilidade é definir uma série de configurações de computadores “comumente encontradas” no lado do cliente e suas variantes. Basicamente, é criada uma estrutura em árvore, identificando cada plataforma de computador, dispositivos típicos de imagem, sistemas operacionais suportados na plataforma, navegadores

INFORMAÇÕES



Ferramentas de teste de interface do usuário Web

Objetivo: O objetivo das ferramentas de teste de interface do usuário Web é determinar problemas de usabilidade ou acessibilidade presentes em uma página Web ou em um site.

Mecanismos: Aqui, são listados dois tipos de ferramentas. Algumas solicitam a URL de uma página Web e imprimem uma lista de sugestões sobre como corrigir quaisquer falhas de compatibilidade com um conjunto de heurísticas. Algumas capturam as ações do usuário e solicitam retorno dele, enquanto trabalham nas páginas de um site.

Ferramentas representativas:¹⁰

<http://www.usabilla.com/> – *Usabilla* é uma ferramenta online que permite aos desenvolvedores monitorar as ações do usuário e coletar opiniões durante o uso ativo de uma página Web.

<http://www.google.com/analytics/> – *Google Analytics* é uma ferramenta online que fornece um conjunto amplo de ferramentas de monitoramento e análise de dados de site, que pode ser usada para avaliar a usabilidade do site.

<http://valet.webthing.com/access/url.html> – *Web Valet* fornece um serviço online para verificar problemas de acessibilidade em páginas Web.

<http://wave.webaim.org/> – *Wave* fornece um serviço online que marca páginas Web para mostrar problemas de acessibilidade.

<http://www.sidar.org/hera/index.php.en> – *Hera* fornece um serviço online que usa o Web Content Accessibility Guidelines para verificar problemas de acessibilidade de páginas Web.

disponíveis, velocidades prováveis de conexão à Internet e informações similares. Em seguida, é derivada uma série de testes de validação de compatibilidade, muitas vezes adaptados de testes de interfaces, de navegação, de desempenho e de segurança existentes. A finalidade desses testes é descobrir erros ou problemas de execução que podem ser atribuídos a diferenças na configuração.

25.5 Teste no nível de componente

O teste no nível de componente, também chamado de *teste de função*, concentra-se em um conjunto de testes que tentam descobrir erros nas funções da WebApp. Cada função da WebApp é um módulo de software (implementado em uma dentre uma variedade de linguagens de programação ou scripts) e pode ser testado por meio de técnicas de caixa-preta (e, em alguns casos, caixa-branca) discutidas no Capítulo 23.

Casos de testes em nível de componente muitas vezes são controlados por entrada no nível de formulários. Uma vez definidos os dados dos formulários, o usuário seleciona um botão ou outro mecanismo de controle para iniciar a execução. O particionamento de equivalência, a análise de valor limite e o teste de caminho (Capítulo 23) podem ser adaptados para uso em teste de entrada baseada em formulários e da funcionalidade aplicada a ela.

Além desses métodos de projeto de casos de teste, uma técnica chamada *teste de erro forçado* [Ngu01] é usada para gerar casos de testes que propo-

¹⁰ A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

CASASEGURA

**Teste de WebApp**

Cena: Escritório de Doug Miller.

Atores: Doug Miller, gerente do grupo de engenharia de software do *CasaSegura*, e Vinod Raman, membro da equipe de engenharia de software do produto.

Conversa:

Doug: O que você acha da WebApp de e-commerce *CasaSeguraGarantida.com* V.0.0?

Vinod: O prestador de serviço do fornecedor fez um bom trabalho. Sharon (gerente de desenvolvimento do fornecedor) disse-me que estão testando conforme conversamos.

Doug: Gostaria que você e os demais membros da equipe fizessem um pequeno teste informal no site de e-commerce.

Vinod (resmungando): Pensei que iríamos contratar uma empresa de teste terceirizada para validar a WebApp. Ainda estamos tentando liberar esse artefato.

Doug: Vamos contratar um fornecedor para testar o desempenho e a segurança, e nosso fornecedor já está testando. Apenas pensei que outro ponto de vista seria útil, e, além disso, precisamos manter os custos dentro dos limites, portanto...

Vinod (suspirando): O que você está procurando?

Doug: Quero ter certeza de que a interface e toda a navegação estão sólidas.

Vinod: Suponho que podemos começar com os casos de uso para cada uma das principais funções de interface:

Conheça o *CasaSegura*.

Especifique o sistema *CasaSegura* de que você precisa.

Compre um sistema *CasaSegura*.

Obtenha suporte técnico.

Doug: Bom. Mas percorra os caminhos de navegação até o fim.

Vinod (examinando um caderno de casos de uso): Sim, quando você seleciona **Especifique o sistema *CasaSegura* de que você precisa**, isso vai levá-lo a:

Selecionar Componentes do *CasaSegura*.

Obtenha recomendações de componentes para o *CasaSegura*.

Podemos exercitar a semântica de cada caminho.

Doug: Aproveite para verificar o conteúdo que aparece em cada nó de navegação.

Vinod: Claro... E os elementos funcionais também. Quem está testando a usabilidade?

Doug: Humm... O fornecedor de teste coordenará o teste de usabilidade. Nós contratamos uma empresa de pesquisa de mercado para selecionar 20 usuários típicos para o estudo de usabilidade, mas se vocês descobrirem quaisquer problemas de usabilidade...

Vinod: Eu sei, passamos para eles.

Doug: Obrigado, Vinod.

talmente conduzem o componente da WebApp para uma condição de erro. A finalidade é descobrir erros que ocorrem durante a manipulação de erro (por exemplo, mensagens de erro incorretas ou inexistentes, falha da WebApp como consequência do erro, saída errônea causada por entrada errônea, efeitos colaterais relacionados ao processamento do componente).

Cada caso de teste no nível de componente especifica todos os valores de entrada e a saída esperada, fornecida pelo componente. A saída real produzida decorrente do teste é registrada para referência futura durante o suporte e a manutenção.

25.6 Testes de navegação

Um usuário navega por uma WebApp de maneira muito semelhante a um visitante que caminha por uma loja ou museu. Podem ser trilhados muitos caminhos, podem ser feitas muitas paradas, muitas coisas a aprender e observar, atividades a iniciar e decisões a tomar. Esse processo de navegação é

previsível no sentido de que cada visitante tem uma série de objetivos quando chega. Ao mesmo tempo, o processo de navegação pode ser imprevisível porque o visitante, influenciado por alguma coisa que vê ou aprende, pode escolher um caminho ou iniciar uma ação que não é típica para o objetivo original. A tarefa do teste de navegação é (1) garantir que os mecanismos que permitem ao usuário navegar pela WebApp estejam todos em funcionamento e (2) confirmar que cada unidade semântica de navegação (NSU, navigation semantic unit) possa ser alcançada pela categoria apropriada de usuário.

25.6.1 Teste da sintaxe de navegação

"Não estamos perdidos. Apenas mudamos a localização."

John M. Ford

A primeira fase do teste de navegação começa realmente durante o teste da interface. Os mecanismos de navegação são verificados para garantir que cada um execute sua função planejada. Splaine e Jaskiel [Spl01] sugerem os seguintes mecanismos de navegação a ser testados: links e âncoras de todos os tipos, redirecionamentos (quando um usuário solicita um URL inexistente), marcadores de página (*bookmarks*), molduras e conjunto de molduras (frames e framesets), mapas de site e a precisão de recursos de busca interna.

Alguns dos testes citados podem ser executados por ferramentas automáticas (por exemplo, verificação de link), enquanto outros são projetados e executados manualmente. O objetivo geral é garantir que os erros em mecanismos de navegação sejam encontrados antes que a WebApp entre no ar.

25.6.2 Teste da semântica de navegação

No Capítulo 17, a unidade semântica de navegação (NSU) é definida como "uma série de informações e estruturas de navegação relacionadas que colaboram no atendimento a um subconjunto de requisitos de usuário relacionados" [Cac02]. Cada NSU é definida por um conjunto de caminhos de navegação (chamados "modos de navegação") que conectam nós de navegação (por exemplo, páginas Web, objetos de conteúdo ou funcionalidade). Considerada como um todo, cada NSU permite ao usuário satisfazer requisitos específicos definidos por um ou mais casos de uso para uma categoria de usuário. O teste de navegação exercita cada NSU para assegurar que esses requisitos possam ser atendidos. Você deve responder às seguintes perguntas à medida que cada NSU é testada:

Se não foram criadas NSUs como parte da análise ou projeto de WebApp, você pode aplicar casos de uso para o projeto de casos de teste de navegação. É proposto e respondido o mesmo conjunto de perguntas.

- A NSU é obtida em sua totalidade sem erro?
- Cada nó de navegação (definido por uma NSU) é acessível no contexto dos caminhos de navegação definidos para a NSU?
- Se a NSU pode ser atendida usando mais de um caminho de navegação, todos os caminhos relevantes foram testados?
- Se forem fornecidas instruções pela interface de usuário para ajudar na navegação, as instruções são corretas e inteligíveis à medida que a navegação ocorre?

- Existe algum mecanismo (que não seja a seta “de retorno” do navegador) para voltar a um nó de navegação anterior e ao início do caminho de navegação?
- Os mecanismos de navegação em um nó grande de navegação (isto é, uma longa página Web) funcionam corretamente?
- Se uma função deve ser executada em um nó e o usuário opta por não fornecer entrada, o restante da NSU pode ser completado?
- Se uma função é executada em um nó e ocorre um erro no processamento da função, a NSU pode ser completada?
- Há uma maneira de interromper a navegação antes que todos os nós tenham sido alcançados, mas depois retornar ao ponto onde a navegação foi interrompida e continuar a partir dali?
- Todos os nós podem ser acessados do mapa do site? Os nomes dos nós têm significado para os usuários?
- Se um nó em uma NSU é alcançado a partir de uma origem externa, é possível processar o próximo nó no caminho de navegação? É possível retornar ao nó anterior no caminho de navegação?
- O usuário entende sua localização dentro da arquitetura de conteúdo à medida que a NSU é executada?

Quais perguntas devem ser feitas e respondidas à medida que cada NSU é testada?

O teste de navegação, bem como os testes de interface e de usabilidade, deverá ser feito por diferentes clientes, quando possível. Os engenheiros da Web têm a responsabilidade pelos primeiros estágios do teste de navegação, mas os estágios posteriores devem ser testados por outros envolvidos no projeto, por uma equipe de teste independente e, por último, por usuários não técnicos. O objetivo é exercitar a navegação da WebApp completamente.

INFORMAÇÕES



Ferramentas de teste de navegação Web

Objetivo: O objetivo das ferramentas de teste de navegação do usuário Web é identificar quaisquer links ou páginas Web interrompidos que não podem ser acessados em um site.

Mecanismos: As ferramentas solicitam a URL de uma fonte Web e percorrem sua linguagem de marcação em busca de links que não retornam o tipo correto de recurso Web. Algumas tentam avançar lentamente pelo site inteiro para procurar erros em links mais profundos.

Ferramentas representativas:¹¹

<http://validator.w3.org/checklink> – Verificador de links WC3 online que analisa documentos HTML e XHTML em busca de links interrompidos.

<http://www.relsoftware.com/> – Site de download da *Rel Link Checker Lite*, uma ferramenta gratuita para identificar links interrompidos e arquivos órfãos.

¹¹ A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

25.7 Teste de configuração

A variabilidade e a instabilidade da configuração são fatores importantes que tornam o teste da WebApp um desafio. Hardware, sistema operacional, navegadores, capacidade de armazenamento, velocidades de comunicação de rede e uma variedade de outros fatores do lado do cliente são difíceis de prever para cada usuário. Além disso, a configuração para um usuário pode mudar regularmente (por exemplo, atualização do sistema operacional, novo provedor e novas velocidades de conexão). O resultado pode ser um ambiente do lado do cliente sujeito a erros sutis e significativos. A impressão de um usuário sobre a WebApp e a maneira como ele interage com ela pode ser significativamente diferente da experiência de outro usuário, se ambos não estiverem trabalhando na mesma configuração do cliente.

O objetivo do *teste de configuração* não é exercitar todas as configurações possíveis no lado do cliente. Em vez disso, é testar um conjunto de prováveis configurações do cliente e do servidor para assegurar que a experiência do usuário seja a mesma em todos os casos e isolar erros que podem ser específicos de determinada configuração.

25.7.1 Tópicos no lado do servidor

No lado do servidor, os casos de teste de configuração são projetados para verificar se a configuração do servidor (isto é, servidor WebApp, servidor de banco de dados, sistema operacional, software de firewall, aplicações concorrentes) pode suportar a WebApp sem erro.

Quando os testes de configuração no lado do servidor são projetados, deve-se considerar cada componente da configuração. Entre as perguntas a ser feitas e respondidas durante o teste de configuração no lado do servidor, destacam-se as seguintes:

Quais perguntas devem ser feitas e respondidas à medida que é feito o teste de configuração no lado do servidor?

- A WebApp é totalmente compatível com o sistema operacional do servidor?
- Os arquivos de sistema, diretórios e dados de sistema relacionados são criados corretamente quando a WebApp está em operação?
- As medidas de segurança do sistema (por exemplo, firewalls ou criptografia) permitem que a WebApp seja executada e sirva os usuários sem interferência ou degradação do desempenho?
- A WebApp foi testada com a configuração de servidor distribuído¹² (se existir alguma) escolhida?
- A WebApp está adequadamente integrada ao software de banco de dados? A WebApp é sensível a diferentes versões do software de banco de dados?
- Os scripts WebApp do lado do servidor executam corretamente?
- Os erros do administrador do sistema foram examinados quanto ao seu efeito sobre as operações da WebApp?
- Se forem usados servidores substitutos (proxy), as diferenças em sua configuração foram resolvidas com o teste *in loco*?

¹² Por exemplo, pode ser usado um servidor de aplicação e um servidor de banco de dados separados. A comunicação entre as duas máquinas ocorre por meio de uma conexão em rede.

25.7.2 Tópicos no lado do cliente

No lado do cliente, os testes de configuração focalizam mais intensamente a compatibilidade da WebApp com configurações que contenham uma ou mais permutações dos seguintes componentes: hardware, sistemas operacionais, software de navegador, componentes de interface do usuário, plug-ins e serviços de conectividade (por exemplo, cabo, DSL, WiFi). Além desses componentes, outras variáveis incluem software de rede, as excentricidades do provedor de serviços (ISP) e aplicações executando concorrentemente.

Para projetar testes de configuração do cliente, deve-se reduzir o número de variáveis de configuração a um número controlável.¹³ Desse modo, cada categoria de usuário é avaliada para determinar as prováveis configurações que serão encontradas. Além disso, dados de mercado podem ser usados para prever as combinações mais prováveis de componentes. A WebApp é, então, testada nesses ambientes.

INFORMAÇÕES



Ferramentas de teste de configuração Web

Objetivo: O objetivo das ferramentas de teste de configuração Web é determinar os problemas que podem ocorrer quando uma página é exibida por diferentes combinações de navegador Web e sistema operacional.

Mecanismos: Essas ferramentas solicitam a URL de uma página Web e permitem escolher dezenas de combinações de navegadores e sistemas operacionais. Elas mostram miniaturas da página Web conforme aparecem em cada versão de navegador selecionada.

Ferramentas representativas:¹⁴

<http://browsershots.org/> – *Browsershots* fornece um serviço online que permite testar seu site em muitos navegadores e sistemas operacionais diferentes.

<http://testingbot.com/> – *TestingBot* oferece uma experiência gratuita e limitada de um serviço online que permite testar seu site usando muitos navegadores e sistemas operacionais diferentes.

25.8 Teste de segurança¹⁵

Segurança de WebApp é um assunto complexo que deve ser muito bem entendido antes de se realizar um teste de segurança efetivo.¹⁶ WebApps e os ambientes cliente e servidor nos quais estão alojadas representam um alvo atraente para invasores (hackers) externos, funcionários insatisfeitos, concorrentes desonestos e qualquer outro que queira roubar informações sigilosas, modificar conteúdo de forma mal intencionada, degradar o desempenho, desabilitar funcionalidade ou atrapalhar uma pessoa, organização ou negócio.

“A Internet é um lugar arriscado para fazer negócios ou armazenar valores. Hackers, invasores, bisbilhoteiros, falsificadores,... vândalos, criadores de vírus e criadores de programas mal-intencionados estão à solta.”

**Dorothy e
Peter Denning**

¹³ Realizar testes em todas as combinações possíveis de componentes de configuração é um processo extremamente demorado.

¹⁴ A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

¹⁵ O teste de segurança também é discutido como parte da engenharia de segurança no Capítulo 27.

¹⁶ Livros de Cross e Fisher [Cro07], Andrews e Whittaker [And06] e Trivedi [Tri03] fornecem informações úteis sobre esse assunto.

Se a WebApp é muito importante para os negócios, contém dados sigilosos ou é um alvo em potencial para invasores, é aconselhável terceirizar o teste de segurança com um prestador de serviço especializado.

Devem ser projetados testes de segurança para testar firewalls, autenticação, criptografia e autorização.

Os testes de segurança são projetados para investigar vulnerabilidades no ambiente do lado do cliente, comunicações de rede que ocorrem quando os dados são passados do cliente para o servidor e vice-versa, e no ambiente do lado do servidor. Cada um desses domínios pode ser atacado, e é tarefa do testador de segurança descobrir os pontos fracos que podem ser explorados por aqueles que têm a intenção de fazer isso.

No lado do cliente, as vulnerabilidades podem ser atribuídas muitas vezes a erros preexistentes em navegadores, programas de e-mail ou software de comunicação. No lado do servidor, as vulnerabilidades incluem ataques que causam recusa de serviço e scripts mal-intencionados que podem ser passados para o lado do cliente ou usados para desabilitar operações do servidor. Além disso, bancos de dados do servidor podem ser acessados sem autorização (roubo de dados).

Para a proteção contra essas vulnerabilidades (e muitas outras), podem ser usados firewalls, autenticação, criptografia e técnicas de autorização. Os testes de segurança devem ser projetados para investigar cada uma dessas tecnologias de segurança em um esforço para descobrir brechas na segurança.

O projeto real de testes de segurança exige profundo conhecimento do funcionamento interno de cada elemento de segurança e de uma ampla gama de tecnologias de rede. Em muitos casos, o teste de segurança é terceirizado, atribuindo-se a empresas que se especializaram nessas tecnologias.

INFORMAÇÕES



Ferramentas de teste de segurança Web

Objetivo: O objetivo das ferramentas de teste de segurança Web é ajudar a identificar possíveis problemas de segurança presentes em um site.

Mecanismos: Normalmente, essas ferramentas são baixadas e executadas no ambiente de desenvolvimento. Elas verificam a aplicação Web quanto a scripts que podem injetar dados prejudiciais que possam alterar a funcionalidade do site. Algumas delas permitem programar seu uso como ferramentas de sondagem ou monitoramento.

Ferramentas representativas:¹⁷

<http://www.mavitunasecurity.com/communityedition/> – Site de download para uma ferramenta (*Netsparker*) que verifica vulnerabilidades quanto à injeção de SQL em WebApps.

<http://enyojs.com/> – Site de download para a ferramenta gratuita *N-Stalker*, a qual realiza várias verificações de segurança em sites usando o banco de dados de assinaturas de ataque web *N-Stealth*.

<http://code.google.com/p/skipfish/> – Site de download da *skipfish*, que prepara um relatório sobre vulnerabilidades de segurança encontradas pelo exame das páginas de um site.

25.9 Teste de desempenho

Nada é mais frustrante do que uma WebApp que leva muitos minutos para carregar o conteúdo, quando sites concorrentes fazem download de con-

¹⁷ A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

teúdo similar em segundos. Nada é mais desgastante do que tentar entrar em uma WebApp e receber uma mensagem do tipo “servidor ocupado”, sugerindo que você tente mais tarde. Nada é mais desconcertante do que uma WebApp que responde instantaneamente em algumas situações e depois parece entrar em um estado de espera infinita em outras. Todas essas ocorrências acontecem diariamente na Web e todas estão relacionadas ao desempenho.

O *teste de desempenho* é usado para descobrir problemas de desempenho que podem resultar da falta de recursos no lado do servidor, largura de banda na rede inadequada, recursos de banco de dados inadequados, recursos deficientes do sistema operacional, funcionalidade da WebApp mal projetada e outros problemas de hardware e software que podem causar degradação de desempenho cliente-servidor. A intenção é dupla: (1) entender como o sistema responde quando a *carga* (isto é, número de usuários, número de transações ou volume geral de dados) aumenta e (2) reunir métricas que conduzirão a modificações de projeto para melhorar o desempenho.

25.9.1 Objetivos do teste de desempenho

Os testes de desempenho são projetados para simular situações de carga do mundo real. À medida que cresce o número de usuários simultâneos da WebApp, o número de transações online ou a quantidade de dados (download ou upload), o teste de desempenho ajudará a responder as seguintes questões:

- O tempo de resposta do servidor degrada a um ponto em que se torna notável e inaceitável?
- Em que ponto (em termos de usuários, transações ou carga de dados) o desempenho se torna inaceitável?
- Que componentes do sistema são responsáveis pela degradação de desempenho?
- Qual o tempo médio de resposta para usuários sob uma variedade de condições de carga?
- A degradação do desempenho tem um impacto sobre a segurança do sistema?
- A confiabilidade ou precisão da WebApp é afetada quando a carga no sistema aumenta?
- O que acontece quando são aplicadas cargas maiores do que a capacidade máxima do servidor?
- A degradação de desempenho tem impacto sobre os lucros da empresa?

Para obter respostas a essas perguntas, são feitos dois diferentes testes de desempenho: (1) o *teste de carga* examina cargas reais em uma variedade de níveis e em uma variedade de combinações e (2) o *teste de esforço* (stress) força o aumento de carga até o ponto de ruptura para determinar com que capacidade o ambiente WebApp pode lidar. Cada uma dessas estratégias é considerada a seguir.

Alguns aspectos do desempenho da WebApp, pelo menos como são observados pelo usuário, são difíceis de testar. A carga da rede, as excentricidades do hardware de interface de rede e problemas similares não são facilmente testados no nível da WebApp.

Se uma WebApp usa múltiplos servidores para proporcionar uma capacidade significativa, o teste de carga deve ser feito em um ambiente multisservidor.

25.9.2 Teste de carga

A finalidade do teste de carga é determinar como a WebApp e seu ambiente do lado do servidor responderá a várias condições de carga. À medida que é feito o teste, permutações das variáveis a seguir definem uma série de condições de teste:

N , número de usuários concorrentes

T , número de transações online por usuários por unidade de tempo

D , carga de dados processados pelo servidor por transação

Em cada caso, as variáveis são definidas de acordo com os limites de operação normal do sistema. Enquanto ocorre cada uma das condições de teste, coletam-se uma ou mais das seguintes medidas: resposta média do usuário, tempo médio para o download de uma unidade padronizada de dados ou tempo médio para processar uma transação. Devem-se examinar essas medidas para determinar se uma diminuição repentina no desempenho pode ser atribuída a uma combinação específica de N , T e D .

O teste de carga também pode ser usado para avaliar a velocidade de conexão recomendada para usuários da WebApp. O resultado geral, P , é calculado da seguinte maneira:

$$P = N \times T \times D$$

Como exemplo, considere um site popular de notícias esportivas. Em dado momento, 20 mil usuários concomitantes enviam uma solicitação (uma transação, T) a cada 2 minutos, em média. Cada transação exige que a WebApp faça o download de um novo artigo que, em média, tem um tamanho de 3K bytes. Portanto, o resultado pode ser calculado como:

$$\begin{aligned} P &= [20.000 \times 0,5 \times 3\text{Kb}] / 60 = 500 \text{ Kbytes/segundo} \\ &= 4 \text{ megabits por segundo} \end{aligned}$$

A conexão de rede do servidor teria, portanto, de suportar essa taxa de transferência de dados e deveria ser testada para assegurar que fosse capaz disso.

25.9.3 Teste de esforço

A finalidade do teste de esforço é entender melhor como o sistema falha quando é forçado além de seus limites operacionais.

O teste de esforço é uma continuação do teste de carga, mas nesse caso as variáveis N , T e D são forçadas a alcançar e exceder os limites operacionais. A finalidade desses testes é responder às seguintes questões:

- O sistema se degrada “suavemente” ou o servidor desliga quando é excedida a capacidade?
- O software servidor gera mensagens “servidor não disponível”? De maneira geral, os usuários ficam cientes de que não podem acessar o servidor?
- O servidor coloca as requisições por recursos em fila e esvazia a fila quando a demanda de capacidade diminui?
- São perdidas transações quando a capacidade é excedida?
- A integridade dos dados é afetada quando a capacidade é excedida?

- Quais valores de N , T e D forçam o ambiente servidor a falhar? Como a falha se manifesta? São enviadas notificações automáticas para o pessoal de suporte técnico no local do servidor?
- Se o sistema falha, quanto tempo demora até que volte a ficar online?
- Certas funções da WebApp (por exemplo, funcionalidade de computação intensiva, recursos de encadeamento de dados) são interrompidas quando a capacidade atinge um nível de 80% ou 90%?

Uma variação do teste de esforço às vezes é chamada de *teste de alternância de pico* (*spike/bounce testing*) [Spl01]. Nesse regime de teste, a carga é elevada até a capacidade, depois diminuída rapidamente para as condições normais de operação e, em seguida, elevada novamente. Ao devolver a carga do sistema, você determina quão bem o servidor pode reunir recursos para atender à demanda muito alta e então liberá-los quando as condições normais se restabelecerem (de forma que fiquem prontos para o próximo pico).

INFORMAÇÕES



Ferramentas de teste de desempenho Web

Objetivo: O objetivo das ferramentas de teste de desempenho Web é procurar gargalos que possam causar desempenho deficiente ou simular condições que possam fazer um site falhar completamente.

Mecanismos: Ferramentas online solicitam a URL de um recurso Web. Algumas realizam automaticamente uma série de testes de carga simulados. Algumas coletam estatísticas sobre carga da página e tempos de resposta do servidor, à medida que os desenvolvedores navegam pelo site.

Ferramentas representativas:¹⁸

<http://loadimpact.com/> – *LoadImpact* é uma ferramenta online que realiza teste de impacto de carga usando cargas de usuário simuladas em servidores web.

<http://www.websitepulse.com/help/testtools.website-test.html/> – *WebSitePulse* é uma ferramenta online que mede disponibilidade de servidor e o tempo de resposta de um site.

<http://www.websiteoptimization.com/services/analyze/> – *Web Page Analyzer* é uma ferramenta online que mede o desempenho do site e fornece uma lista de sugestão de alterações para melhorar os tempos de carga.

<http://developer.yahoo.com/yslow/> – *Yslow* é uma ferramenta online que analisa páginas Web e sugere aprimoramentos com base em regras de desenvolvimento de sites de alto desempenho.

<http://tools.pingdom.com/fpt/> – *Pingdom* é uma ferramenta online que mede gargalos no tempo de carga de página Web analisando os elementos componentes individualmente.

25.10 Resumo

O objetivo do teste de WebApp é testar cada uma das muitas dimensões da qualidade da WebApp com a finalidade de encontrar erros ou descobrir problemas que podem levar a falhas de qualidade. O teste foca conteúdo, função, estrutura, utilização, navegabilidade, desempenho, compatibilidade, interoperabilidade, capacidade e segurança. Ele incorpora revisões que

¹⁸ A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

ocorrem quando a WebApp é projetada e testes feitos depois que a WebApp é implantada.

A estratégia de teste para WebApp testa cada dimensão da qualidade examinando inicialmente “unidades” de conteúdo, funcionalidade ou navegação. Uma vez validadas as unidades, o foco passa para os testes que avaliam a WebApp como um todo. Para tanto, muitos testes são derivados da perspectiva do usuário e orientados por informações contidas em casos de uso. Um plano de teste para WebApp é desenvolvido e identifica as etapas do teste, os artefatos finais (por exemplo, casos de teste) e os mecanismos para a avaliação dos resultados. O processo de teste abrange sete tipos de teste.

O teste de conteúdo (e revisões) concentra-se nas várias categorias de conteúdo. O objetivo é descobrir erros de sintaxe e semântica que afetam a exatidão do conteúdo ou a maneira como ele é apresentado ao usuário. O teste de interface testa os mecanismos de interação que possibilitam a um usuário se comunicar com a WebApp e validar aspectos estéticos da interface. A finalidade é descobrir erros que resultam de mecanismos de interação mal implementados ou de omissões, inconsistências ou ambiguidades na semântica da interface.

O teste de navegação aplica casos de uso, criados como parte da atividade de modelagem, no projeto de casos de teste que testam cada cenário de uso em relação ao projeto de navegação. Os mecanismos de navegação são verificados para assegurar que quaisquer erros que impeçam a realização de um caso de uso sejam identificados e corrigidos. O teste de componente testa as unidades de conteúdo e funcional da WebApp.

O teste de configuração tenta descobrir erros e/ou problemas de compatibilidade específicos de um ambiente de determinado cliente ou servidor. São então aplicados testes para descobrir erros associados a cada configuração possível. O teste de segurança incorpora uma série de testes projetados para explorar vulnerabilidades na WebApp e seu ambiente. A finalidade é encontrar brechas de segurança. O teste de desempenho abrange uma série de testes projetados para avaliar o tempo de resposta e a confiabilidade da WebApp quando aumenta a demanda de recursos do servidor.

Problemas e pontos a ponderar

25.1 Existem quaisquer situações nas quais o teste de WebApp deve ser totalmente desconsiderado?

25.2 Discuta os objetivos do teste em um contexto de WebApp.

25.3 Compatibilidade é uma dimensão de qualidade importante. O que deve ser testado para garantir que exista compatibilidade em uma WebApp?

25.4 Quais erros tendem a ser mais graves – erros no cliente ou erros no servidor? Por quê?

25.5 Quais elementos da WebApp podem ser “testados em unidade”? Que tipos de testes devem ser executados apenas depois que os elementos da WebApp estiverem integrados?

25.6 É sempre necessário desenvolver um plano formal de teste escrito? Explique.

25.7 É correto dizer que a estratégia geral de teste para WebApp começa com elementos visíveis para o usuário e passa para os elementos de tecnologia? Há exceções a essa estratégia?

25.8 O teste de conteúdo está *realmente* testando em um sentido convencional? Explique.

25.9 Descreva os passos associados ao teste de banco de dados para uma WebApp. O teste de banco de dados é predominantemente uma atividade do lado do cliente ou do lado do servidor?

25.10 Qual é a diferença entre teste associado aos mecanismos de interface e teste que cuida da semântica da interface?

25.11 Suponha que você esteja desenvolvendo uma farmácia online (**YourCornerPharmacy.com**) dedicada aos aposentados e idosos. A farmácia tem funções típicas, mas também mantém um banco de dados de cada cliente para que possa fornecer informações sobre medicamentos e alertar sobre interações de certos medicamentos. Discuta quaisquer testes especiais de utilização para essa WebApp.

25.12 Suponha que você tenha implementado uma função de verificação de interação de medicamentos para **YourCornerPharmacy.com** (Problema 20.11). Discuta os tipos de testes no nível de componente que teriam de ser feitos para garantir que essa função funcione corretamente. (Observação: teria de ser usado um banco de dados para implementar essa função.)

25.13 Qual é a diferença entre teste de sintaxe de navegação e teste de semântica de navegação?

25.14 É possível testar todas as configurações que uma WebApp pode encontrar no lado do servidor? E no lado do cliente? Se não, como você seleciona um conjunto significativo de testes de configuração?

25.15 Qual o objetivo do teste de segurança? Quem executa essa atividade de teste?

25.16 A **YourCornerPharmacy.com** (Problema 25.11) tornou-se extraordinariamente bem-sucedida, e o número de usuários aumentou significativamente nos primeiros dois meses de operação. Trace um gráfico que mostre o tempo de resposta provável em função do número de usuários para um conjunto fixo de recursos no servidor. Inclua legendas no gráfico para indicar pontos de interesse na “curva de resposta”.

25.17 Em resposta ao seu sucesso, a **YourCornerPharmacy.com** (Problema 25.11) implementou um servidor especial para cuidar de renovações de receitas. Em média, 1.000 usuários concorrentes enviam uma solicitação de renovação a cada dois minutos. A WebApp faz o download de um bloco de 500 bytes de dados em resposta. Qual a vazão/taxa de saída (*throughput*) aproximada necessária para esse servidor em megabits por segundo?

25.18 Qual a diferença entre teste de carga e teste de esforço?

Leituras e fontes de informação complementares

A literatura para teste de WebApp continua a evoluir. Livros de Andrews e Whittaker (*How to Break Web Software*, Addison-Wesley, 2006), Ash (*The Web Testing Companion*, Wiley, 2003), Nguyen e seus colegas (*Testing Applications for the Web*, 2ª ed., Wiley, 2003), Dustin e seus colegas (*Quality Web Systems*, Addison-Wesley, 2002) e Splaine e Jaskiel [Spl01] estão entre os tratados mais completos sobre o assunto publicados até hoje. Whittaker e seus colegas descrevem mais práticas de teste Web (*How Google Tests Software*, Addison-Wesley, 2012). Mosley (*Client-Server Software Testing on the Desktop and the Web*, Prentice Hall, 1999) examina os problemas de teste tanto no lado do cliente quanto no do servidor.

Informações úteis sobre estratégias e métodos de teste de WebApp, bem como uma discussão conveniente sobre ferramentas de teste automáticas, são apresentadas por David (*Selenium 2 Testing Tools: A Beginner's Guide*, Packit Publishing, 2012) e Stottlemeyer (*Automated Web Testing Toolkit*, Wiley, 2001). Graham e seus colegas (*Experiences of Test Automation*, Addison-Wesley, 2012) e (*Software Test Automation*, Addison-Wesley, 1999) apresentam material adicional sobre ferramentas automatizadas.

A Microsoft (*Performance Testing Guidance for Web Applications*, Microsoft Press, 2008) e Subraya (*Integrated Approach to Web Performance Testing*, IRM Press, 2006) apresentam tratamentos detalhados sobre teste de desempenho para WebApps. Hope e Walther (*Web Security Testing Cookbook*, O'Reilly, 2008), Skoudis (*Counter Hack Reloaded*, Prentice Hall, 2ª ed, 2006), Andreu (*Profession Pen Testing for Web Applications*, Wrox, 2006), Chirillo (*Hack Attacks Revealed*, 2ª ed., Wiley, 2003), Splaine (*Testing Web Security*, Wiley, 2002) e Klevinsky e seus colegas (*Hack I.T.: Security through Penetration Testing*, Addison-Wesley, 2002) fornecem muitas informações úteis para aqueles que devem projetar testes de segurança. Além disso, livros que tratam de teste de segurança para software em geral podem oferecer orientações importantes para aqueles que devem testar WebApps. Entre os títulos representativos estão: Engebretson (*Basics of Hacking and Penetration Testing*, Syngress, 2011), Basta e Halton (*Computer Security and Penetration Testing*, Thomson Delmar Learning, 2007), Wysopal e seus colegas (*The Art of Software Security Testing*, Addison-Wesley, 2006) e Gallagher e seus colegas (*Hunting Security Bugs*, Microsoft Press, 2006).

Há uma grande variedade de recursos de informação sobre teste de WebApp disponível na Internet. Uma lista atualizada das referências da Web (em inglês) pode ser encontrada no site: www.mhhe.com/pressman.

Teste de aplicativos móveis

26

O mesmo senso de urgência que orienta os projetos de WebApp também se aplica aos projetos de aplicativos móveis. Os envolvidos estão preocupados em não perder uma oportunidade do mercado e fazem pressão para ter o aplicativo móvel disponível nas lojas. As atividades técnicas que muitas vezes ocorrem mais tarde no processo, como testes de desempenho e segurança, dispõem algumas vezes de um prazo muito curto. O teste de usabilidade, que deve ocorrer durante a fase de projeto, pode acabar sendo adiado para pouco antes da distribuição. Esses erros podem ser catastróficos. Para evitar essa situação, os membros da equipe precisam assegurar que cada artefato tenha alta qualidade. Um relatório técnico divulgado por Soasta [Soa11] resume isso:

A tecnologia móvel está crescendo mais rapidamente do que outras tecnologias o fizeram no passado – talvez seja a curva de adoção mais rápida da história. E isso tem implicações importantes para seu modelo de negócio. Você chega mais rápido ao mercado, mas também precisa estar preparado para a rápida adoção. Se seu aplicativo tiver desempenho deficiente ou falhar sob carga, muitos concorrentes estarão prontos para tomar seu lugar – as barreiras de entrada são baixas.

Os requisitos e os modelos de projeto do aplicativo móvel não podem ser testados unicamente com casos de teste executáveis. A equipe deve realizar revisões técnicas (Capítulo 20) e testar a usabilidade (Capítulo 15) e o desempenho. O teste de configuração é particularmente importante como mecanis-

Conceitos-chave

automação.	571
checklist.	570
diretrizes.	568
estratégias.	570
ferramentas e ambientes.	579
internacionalização.	578
matriz de teste.	572
teste de esforço (stress).	573
teste de usabilidade.	575
teste em condições naturais.	573
teste em tempo real.	578

PANORAMA

O que é? O teste de aplicativos móveis é um conjunto de atividades relacionadas com um único

objetivo: descobrir erros no conteúdo, na função, na usabilidade, na navegabilidade, no desempenho, na capacidade e na segurança do aplicativo móvel. Para tanto, deve ser utilizada uma estratégia de teste que abrange as revisões e o teste executável.

Quem realiza? Os engenheiros de software e outros envolvidos no projeto (gerentes, clientes e usuários), todos participam do teste do aplicativo móvel.

Por que é importante? Se os usuários encontrarem erros ou dificuldades dentro do aplicativo móvel, vão procurar o conteúdo e a função personalizados de que precisam em outro lugar. Por isso, você deve procurar e corrigir o máximo de erros possível antes que o aplicativo móvel seja colocado em uma loja ou repositório de aplicativos.

Quais são as etapas envolvidas? O processo de teste de um aplicativo móvel começa focalizando os aspectos visíveis do aplicativo para o usuário e passa para os testes de tecnologia e infraestrutura. Executam-se várias etapas de teste: de conteúdo, interface, navegação, componente, configuração, desempenho e segurança.

Qual é o artefato? Frequentemente é produzido um plano de teste para o aplicativo móvel. É desenvolvida uma série de casos de teste para cada etapa e é mantido um arquivo dos resultados para uso futuro.

Como garantir que o trabalho foi realizado corretamente? Embora nunca se possa ter certeza de ter executado todos os testes necessários, é possível verificar se erros foram apontados (e corrigidos). Além disso, se foi estabelecido um plano de teste, é aconselhável certificar-se de que todos os testes planejados foram realizados.

mo de verificação da capacidade do aplicativo móvel de levar o contexto em conta. A intenção é descobrir e corrigir erros antes que o aplicativo móvel seja lançado para a comunidade de usuários.

Existem várias perguntas importantes a fazer ao se criar uma estratégia de teste de aplicativos móveis [Sch09]:

- É preciso construir um protótipo totalmente funcional antes de testar com os usuários?
- Você deve testar com o dispositivo do usuário ou fornecer um dispositivo para teste?
- Quais dispositivos e grupos de usuários devem ser incluídos no teste?
- Quais são as vantagens/desvantagens dos testes de laboratório em relação aos testes remotos?

Tratamos de cada uma dessas questões ao longo deste capítulo.

26.1 Diretrizes de teste

Os aplicativos móveis executados inteiramente em um dispositivo móvel podem ser testados com métodos de teste de software tradicionais (Capítulo 23) ou com emuladores executando em computadores pessoais. Por outro lado, o teste de aplicativos móveis do tipo cliente magro (*thin-client*), que utilizam recursos baseados no servidor, é particularmente desafiador. Além dos muitos desafios de teste apresentados pelas WebApps (Capítulo 25), o teste de aplicativos móveis de cliente magro também devem considerar a transmissão de dados por gateways da Internet e rede telefônicas [Was10]. Os usuários esperam que os aplicativos móveis sejam sensíveis ao contexto e apresentem experiências personalizadas, de acordo com o local físico de um dispositivo em relação aos recursos de rede disponíveis. Mas é difícil, se não impossível, testar aplicativos móveis em um ambiente dinâmico de rede *ad hoc* usando todas as configurações de rede e dispositivo disponíveis.

Para entender os objetivos dos testes de aplicativos móveis, você deve considerar os muitos desafios únicos enfrentados pelos projetistas. Espera-se que os aplicativos móveis apresentem grande parte da complexa funcionalidade e confiabilidade encontradas nos aplicativos de PC, mas eles residem em plataformas móveis, com recursos relativamente limitados. As diretrizes a seguir oferecem uma base para o teste de aplicativos móveis [Kea07]:

- *Entenda o cenário do dispositivo e da rede antes de testar, a fim de identificar gargalos.* O teste além de fronteiras é discutido na Seção 26.4.
- *Realize testes em condições reais, sem controle* (testes de campo), especialmente para um aplicativo móvel de várias camadas lógicas. O teste no ambiente de produção é discutido na Seção 26.2.5.
- *Escolha a ferramenta de teste de automação correta.* Idealmente, a ferramenta deve suportar todas as plataformas desejadas, permitir teste de vários tipos de tela, resoluções e mecanismos de entrada (como tela de toque e teclado numérico) e implementar conectividade com o sistema externo para a realização de teste de um extremo ao outro. As ferramen-

Envolva os usuários o mais cedo possível no ciclo de teste – um retorno antecipado ajuda na correção do projeto.

tas de teste de aplicativos móveis são discutidas de forma mais detalhada na Seção 26.6.

- *Use o método da matriz de plataformas de dispositivo com valores ponderados (weighted device platform matrix) para identificar a combinação de hardware/plataforma mais crítica a testar.* Esse método é muito útil, especialmente quando existem muitas combinações de hardware/plataforma e o tempo para testar é reduzido. Os detalhes da aplicação desse método estão descritos na Seção 26.2.3.
- *Pelo menos uma vez, verifique o fluxo funcional de um extremo ao outro em todas as plataformas possíveis.* Quando há envolvimento de Web services, sem ferramentas de desempenho é difícil traçar o caminho de rede real exigido para distribuir uma função de aplicativo móvel. O uso de ferramentas é discutido na Seção 26.6.
- *Realize testes de desempenho, de interface gráfica do usuário e de compatibilidade utilizando dispositivos reais.* Mesmo que esses testes possam ser feitos com emuladores, recomenda-se testar com dispositivos reais. O teste da interação do usuário é discutido na Seção 26.3, e os problemas de desempenho são discutidos na Seção 26.2.
- *Avalie o desempenho somente em condições realistas de tráfego de rede sem fio e carga de usuários.* Os problemas do teste em tempo real para aplicativos móveis são discutidos na Seção 26.5.

26.2 As estratégias de teste

Apenas tecnologia não é suficiente para garantir o sucesso comercial de um aplicativo móvel. Os usuários abandonam aplicativos móveis rapidamente se não funcionam bem ou não atingem as expectativas. É importante lembrar que o teste tem dois objetivos fundamentais: (1) criar casos de teste que revelem defeitos no início do ciclo de desenvolvimento e (2) verificar a presença de atributos de qualidade importantes. Os atributos de qualidade de aplicativos móveis têm por base aqueles estabelecidos no ISO 9126* [Spr04] e abrangem funcionalidade, confiabilidade, usabilidade, eficiência, facilidade de manutenção e portabilidade (Capítulo 19).

O desenvolvimento de uma estratégia de teste de aplicativo móvel exige entender os testes de software e os desafios que tornam os dispositivos móveis e sua infraestrutura de rede únicos [Kho12a]. Além de um conhecimento completo das abordagens de teste de software convencionais (Capítulos 22 e 23), um testador de aplicativos móveis deve conhecer bem os princípios das telecomunicações e reconhecer as diferenças e capacidades das plataformas de sistemas operacionais móveis. Esse conhecimento básico deve ser complementado com um entendimento completo dos diferentes tipos de testes móveis (por exemplo, teste de aplicativo móvel, de telefone celular, de site móvel), com o uso de simuladores, ferramentas de automação de teste e serviços de acesso remoto a dados (RDA, remote data access). Cada um desses assuntos será discutido mais adiante neste capítulo.

"Quero ser enterrada com um telefone celular, para o caso de eu não estar morta."

Amanda Holden

* N. de R.T.: Ver também a norma ISO 25010:2011, evolução da norma ISO 9126.



Teste de aplicativo móvel – checklist

Nari Kannan, diretor executivo da empresa de consultoria de aplicativos móveis appsparq, Inc., divulgou as seguintes recomendações para teste de aplicativos móveis [Kan11]:

- **Teste conceitual** – Obter a opinião de possíveis usuários do aplicativo móvel antes de iniciar o desenvolvimento ajuda a garantir que recursos desnecessários não sejam incluídos e que todos os recursos essenciais estejam presentes. (A reunião de requisitos e as técnicas de validação são discutidas no Capítulo 9.)
- **Teste de unidade e de sistema** – A implementação de testes de unidade e de sistema regulares adapta qualquer software aplicativo que contenha vários componentes e interaja com redes (Capítulos 22 e 26).
- **Teste da experiência do usuário** – Envolver usuários reais no início do processo de desenvolvimento garante que a experiência cumpra as expectativas de usabilidade e acessibilidade dos envolvidos. O teste da usabilidade de aplicativos móveis é discutido na Seção 26.3.
- **Teste de estabilidade** – Garantir que o aplicativo móvel não falhe devido a incompatibilidades com serviços de rede ou Web services. O teste de aplicativos móveis em ambientes de produção é discutido na Seção 23.2.5.
- **Teste de conectividade** – É fundamental testar os acessos a todos os componentes e recursos Web essenciais para verificar se o aplicativo móvel está utilizando o contexto de forma apropriada. O teste além de fronteiras é discutido na Seção 26.6.
- **Teste de desempenho** – Testar a capacidade do aplicativo móvel de atender seus requisitos não funcionais (tempos de download, velocidade do processador, capacidade de armazenamento etc.). O teste de desempenho é discutido nas Seções 26.2.4 e 26.5.
- **Teste de compatibilidade do dispositivo** – Verificar se o aplicativo móvel funciona corretamente em todos os dispositivos pretendidos. A tarefa é discutida na Seção 26.2.3.
- **Teste de segurança** – Garantir que o aplicativo móvel atenda aos requisitos de segurança estabelecidos pelos envolvidos. O teste de segurança é discutido no Capítulo 27.
- **Teste de certificação** – Verificar se o aplicativo móvel atende aos padrões estabelecidos pelas lojas de aplicativo que vão distribuí-lo.

Quem precisa testar aplicativos móveis estará bem servido se adotar a estratégia geral para teste de WebApps.

26.2.1 As estratégias convencionais são adequadas?

Um programa de teste de aplicativo móvel abrangente inclui a estratégia em espiral genérica, discutida no Capítulo 22, mas também terá as adaptações discutidas para arquiteturas cliente-servidor, computação em tempo real, interfaces gráficas de usuário, WebApps e sistemas orientados a objetos (Capítulos 23 a 25). O teste de aplicativos móveis também tem desafios únicos que devem ser encarados para garantir que um aplicativo satisfaça seus requisitos funcionais e não funcionais.¹

Vinson [Vin11] sugere que os testadores de aplicativos móveis adaptem a estratégia utilizada para testar WebApps (Capítulo 25). O conteúdo deve ser testado para verificar se foi escolhido tendo-se em mente as limitações dos dispositivos móveis e das redes *ad hoc*. Os testes de compatibilidade e distribuição são mais desafiadores no mundo móvel, devido à grande variedade nas características dos dispositivos e nos ambientes dos usuários. O teste de desempenho precisa determinar se o armazenamento, processamento, conectividade e poder limitados, disponíveis em um dispositivo móvel, podem afetar negativamente os recursos ou a funcionalidade. Muitas vezes, o teste de desempenho de aplicativo móvel é realizado com um nível de detalhe visto apenas no desenvolvimento de sistemas em tempo real. O teste de seguran-

¹ Um panorama dos problemas específicos do teste de aplicativos móveis pode ser encontrado em: <http://www.utest.com/landing-interior/crowdsourcing-your-mobile-app-testing>.

ça precisa levar em consideração a perda de um dispositivo físico e o fato de que, frequentemente, os aplicativos móveis são executados diretamente no hardware do dispositivo, expondo dados pessoais a roubo. Muitas vezes, os aplicativos móveis são projetados para serem usados por pessoas com menos conhecimento técnico do que o usuário típico da Web, acarretando a necessidade de testes mais amplos da experiência do usuário (Seção 26.3). Podem ser usados modelos de processo de desenvolvimento ágil (Capítulo 5) e/ou modelos de desenvolvimento orientado a testes (Capítulo 24).

26.2.2 A necessidade de automação

Um testador de aplicativo móvel frequentemente encontra muitas variantes de configuração (dispositivos, sistemas operacionais e redes móveis) que talvez precisem ser incluídas para garantir que o aplicativo funcione e seja sensível ao contexto. Como é importante testar um aplicativo móvel de forma eficiente e completa, a automação pode ser útil para testes de configuração (Seção 25.7) ou de regressão (Seção 22.3). Contudo, deve-se mencionar que talvez não seja possível automatizar todas as partes do teste do aplicativo móvel (por exemplo, as interações de usuário encontradas em um videogame portátil).

Ferramentas de teste automatizadas podem melhorar o moral da equipe, quando os testadores, de outro modo, seriam obrigados a processar mecanicamente um grande número de casos de teste repetitivos. A disponibilidade dessas ferramentas pode estimular a realização antecipada e mais frequente de testes, permitindo a descoberta antecipada de defeitos no aplicativo móvel. Um processo de desenvolvimento ágil (Capítulo 5) impõe o uso de ciclos de construção diários que exigem testes de regressão para garantir que alterações não tenham produzido efeitos colaterais involuntários.

A Mobile Labs, Inc. [Mob11] propôs uma estratégia para automatizar testes de aplicativos móveis, abrangendo os seguintes elementos:

Análise da viabilidade. Determina quais testes e casos de teste terão o maior retorno sobre o investimento (ROI, return on investment) se automatizados. O enfoque deve estar em automatizar casos de teste que possam ser repetidos e frequentemente utilizados. O objetivo é tentar automatizar de 50 a 60% dos casos de teste manuais.

Prova de conceito. Valida o valor da automação do teste. Para isso, um número limitado de scripts de teste manuais é automatizado para determinar o ROI do esforço. A equipe de teste deve determinar a escalabilidade para os outros scripts e o grau de reutilização em ciclos de teste subsequentes.

Estrutura de teste de melhor prática. Fornece uma metodologia específica para aplicativos móveis que serve como base para o processo de testes. As estruturas definem as regras para implementação e teste do aplicativo móvel e são desenvolvidas para cada plataforma móvel e personalizadas de acordo com o conjunto de aplicativos da organização.

Ferramentas de teste personalizadas. Personaliza ferramentas de teste para cada plataforma móvel (e para o aplicativo que está sendo testado), alavancando técnicas de script avançadas.

"Hoje, existem centenas de milhões de dispositivos móveis, mas, para encará-los como desenvolvedor, você precisa saber um pouco sobre o que cada um é capaz de fazer."

John Fowler

Quais são os elementos mais importantes dos testes móveis automatizados?

Testar sob condições reais. Verifica como o aplicativo vai funcionar em um dispositivo real fora do laboratório de teste. Testar em dispositivos reais, em vez de usar emuladores, reduz a incidência de falsos relatos de defeito e garante que os erros em nível de usuário tenham maior probabilidade de serem descobertos (Seção 26.2.5).

Rápida solução de defeitos. Acelera a implementação por meio do envio automático de informações de defeito e da geração de relatórios de discrepância, permitindo aos desenvolvedores reduzir o tempo exigido para solucionar falhas.

Reutilização de scripts de teste. Proporciona redução de gastos, eliminando a necessidade de começar a criação de casos de teste a partir do zero quando são feitos aprimoramentos. É importante que a arquitetura da ferramenta de teste permita a separação de interfaces de função e lógica de teste. Isso possibilita que as interfaces sejam empacotadas em funções reutilizáveis à medida que as ferramentas são adaptadas para novas plataformas e dispositivos.

26.2.3 Construção de uma matriz de teste

Muitas vezes, os aplicativos móveis são desenvolvidos para vários dispositivos e projetados para serem usados em muitos contextos e locais diferentes. Uma *matriz de plataformas de dispositivo com valores ponderados* (WDPM, *weighted device platform matrix*) ajuda a garantir que a cobertura do teste inclua cada combinação de dispositivo móvel e variáveis de contexto.² A WDPM também pode ser usada para ajudar a priorizar as combinações de dispositivo/contexto para que os mais importantes sejam testados primeiro.

As etapas de construção da WDPM (Figura 26.1) para vários dispositivos e sistemas operacionais são: (1) listar as variantes importantes do sistema operacional como os rótulos de coluna da matriz, (2) listar os dispositivos-alvo como os rótulos de linha da matriz, (3) atribuir uma classificação (por exemplo, de 0 a 10) para indicar a importância relativa de cada sistema operacional e de cada dispositivo e (4) calcular o produto de cada par de classificações e inserir cada produto como entrada de célula na matriz (use ND para combinações que não estejam disponíveis).

O trabalho de teste deve ser ajustado de modo que as combinações de dispositivo/plataforma com as classificações mais altas recebam a máxima atenção para cada variável de contexto sob consideração. Na Figura 26.1, **Dispositivo 4** e **SO3** têm a classificação mais alta; portanto, receberiam atenção de alta prioridade durante o teste.

		SO1	SO2	SO3
	Classificação	3	4	7
Dispositivo 1	7	ND	28	49
Dispositivo 2	3	9	ND	ND
Dispositivo 3	4	14	ND	ND
Dispositivo 4	9	ND	36	63

FIGURA 26.1 Matriz de plataformas de dispositivo ponderada.

² *Variáveis de contexto* são aquelas associadas à conexão atual ou à transação atual que o aplicativo móvel vai usar para orientar seu comportamento visível para o usuário.

26.2.4 Teste de esforço (stress)

O teste de esforço (*stress*) de aplicativos móveis tenta encontrar erros que vão ocorrer sob condições operacionais extremas. Além disso, oferece um mecanismo para determinar se o aplicativo móvel vai degradar discretamente, sem comprometer a segurança. Dentre as muitas ações que poderiam gerar condições extremas estão: (1) executar vários aplicativos móveis no mesmo dispositivo, (2) infectar o software de sistema com vírus ou malware, (3) tentar assumir o controle de um dispositivo e utilizá-lo para espalhar spam, (4) obrigar o aplicativo móvel a processar desordenadamente grandes quantidades de transações e (5) armazenar volumes excessivamente grandes de dados no dispositivo. Quando essas condições são encontradas, o aplicativo móvel é confeccionado para verificar se os serviços que utilizam muitos recursos (por exemplo, streaming de mídia) são tratados corretamente.

Um teste de esforço eficiente [Soa11] deve ser realizado “em condições naturais”, em que diferentes dispositivos e diferentes ambientes operacionais são comuns. Devem ser testadas condições que utilizem de duas a três vezes a capacidade nominal. Os testes devem refletir o comportamento do usuário em contextos da vida real e refletir usuários que mudam de lugar em determinado local, assim como usuários de outros países, com diferentes configurações e padrões de rede.

Nas próximas seções, consideraremos algumas dessas diretrizes em mais detalhes.

26.2.5 Testes em um ambiente de produção

Muitos desenvolvedores de aplicativos móveis defendem o teste em condições naturais, ou teste nos ambientes nativos dos usuários, com as versões de lançamento de produção dos recursos do aplicativo. O teste em condições naturais é projetado para ser ágil e responder às mudanças à medida que o aplicativo móvel evolui [Ute12].

Algumas das características do teste em condições naturais incluem ambientes adversos e imprevisíveis, navegadores e plug-ins obsoletos, hardware exclusivo e conectividade imperfeita (tanto Wi-Fi quanto operadora de celular). Para espelhar as condições do mundo real, as características demográficas dos testadores devem corresponder às dos usuários-alvo, assim como às de seus dispositivos. Além disso, você deve incluir casos de uso envolvendo poucos usuários, navegadores menos conhecidos e um conjunto diversificado de dispositivos móveis. O teste em condições naturais é sempre um pouco imprevisível, e os planos de teste devem ser adaptados à medida que o teste progride. Para mais informações, Rooksby e seus colegas identificaram temas que estão presentes em estratégias bem-sucedidas para teste em condições naturais [Roo09].

Criar ambientes de teste internos é um processo dispendioso e propenso a erros. Um teste baseado na nuvem pode oferecer uma infraestrutura padronizada e imagens de software previamente configuradas, isentando a equipe do aplicativo móvel da necessidade de se preocupar com a descoberta de servidores ou adquirir suas próprias licenças para software e ferramentas de teste. Os provedores de serviços de nuvem fornecem aos testadores acesso a laboratórios virtuais escalonáveis e prontos para o usuário, com uma bibliote-

Degradação discreta é uma propriedade importante de sistemas tolerantes a falhas. Um sistema que degrada discretamente tentará chegar a um estado seguro em caso de erro antes de desligar se não puder reparar o dano e permitir que a execução continue.

“O importante em relação à tecnologia móvel é que todo mundo tem um computador no bolso.”

Ben Horowitz

ca de sistemas operacionais, ferramentas de gerenciamento de teste e execução e armazenamento necessários para a criação de um ambiente de teste que espelha bem de perto o mundo real [My111].

O teste baseado na nuvem tem alguns problemas: ausência de padrões, questões de segurança em potencial, questões de localização de dados e integridade, suporte de infraestrutura incompleto, utilização incorreta de serviços e problemas de desempenho são apenas alguns dos desafios comuns enfrentados pelas equipes de desenvolvimento que utilizam a estratégia da nuvem.

CASASEGURA



Teste de aplicativos móveis no ambiente de produção

Cena: Escritório de Doug Miller.

Atores: Doug Miller (gerente do grupo de engenharia de software do CasaSegura) e Vinod Raman (membro da equipe de engenharia de software do produto).

Conversa:

Doug: O que você acha da parte e-commerce de nosso aplicativo móvel CasaSeguraGarantida V0.0?

Vinod: O fornecedor terceirizado fez um bom trabalho de adaptação da WebApp CasaSeguraGarantida.com para o ambiente móvel. Sharon [gerente de desenvolvimento do fornecedor] disse-me que estão testando o protótipo conforme conversamos.

Doug: Ouvi dizer que estavam testando para o site de e-commerce usando emuladores de dispositivo. Acho que devemos fazer alguns testes em dispositivos reais.

Vinod (fazendo careta): Pensei que iríamos contratar uma empresa de teste para validar o aplicativo móvel. Ainda estamos tentando liberar esse artefato.

Doug: Vamos contratar um fornecedor para testar o desempenho, a segurança e a configuração. Nosso fornecedor terceirizado já está fazendo testes. Apenas pensei que outro

ponto de vista seria útil e, além disso, precisamos manter os custos dentro dos limites, portanto...

Vinod (suspirando): O que você está procurando?

Doug: Quero ter certeza de que a experiência do usuário está sólida.

Vinod: Suponho que podemos começar com os casos de uso para cada uma das principais funções de interface.

Doug: Bom. Mas percorra os caminhos lógicos do início ao fim. Dê uma olhada na matriz de plataformas de dispositivo com valores ponderados. Gostaria que você verificasse seu desempenho nos seis dispositivos mais importantes e, enquanto estiver fazendo isso, examine o conteúdo que aparece em cada nó de navegação. Certifique-se de levar em conta as características do dispositivo à medida que cada tela for renderizada.

Vinod: Claro... e os elementos funcionais também. Quem está testando a usabilidade?

Doug: Humm... O fornecedor de teste coordenará o teste de usabilidade. Nós contratamos uma empresa de pesquisa de mercado para selecionar 20 usuários típicos para o estudo de usabilidade, mas se vocês descobrirem quaisquer problemas de usabilidade...

Vinod: Eu sei, passamos para eles.

Doug (sorrindo): Obrigado, Vinod.

26.3 Considerações sobre o espectro da interação do usuário

Quais características da usabilidade dos aplicativos móveis se tornam o foco dos testes e quais objetivos específicos são tratados?

Em um mercado abarrotado, no qual os produtos oferecem a mesma funcionalidade, os usuários escolherão o aplicativo móvel mais fácil de usar. A interface do usuário e seus mecanismos de interação são visíveis para os usuários de aplicativos móveis. É importante testar a qualidade da experiência do usuário fornecida pelo aplicativo móvel para garantir que ele atenda às expectativas de seus usuários.

Muitos dos procedimentos usados para avaliar a usabilidade de interfaces de usuário de software, discutidos no Capítulo 15, podem ser utilizados para avaliar aplicativos móveis. Do mesmo modo, muitas das estratégias usa-

das para avaliar a qualidade de WebApps (Capítulo 25) podem ser usadas para testar a parte relativa à interface de usuário do aplicativo móvel. Construir uma boa interface de usuário para aplicativos móveis envolve mais do que apenas reduzir o tamanho da interface de usuário de um aplicativo de computador pessoal já existente.

INFORMAÇÕES



Componentes de teste de usabilidade de aplicativos móveis

Várias recomendações sobre componentes importantes para o teste de usabilidade de aplicativos móveis foram apresentadas no MobileApp Testing Blog.³

- **Funcionalidade** – Garantir que a funcionalidade básica seja suportada pelas histórias de usuário e levar em conta as metas e expectativas dos envolvidos.
- **Arquitetura da informação** – Verificar se conteúdo e links foram estruturados e apresentados de maneira lógica, levando agrupamento⁴ e perspectivas em consideração.
- **Conteúdo** – Usar texto, vídeo, imagens e multimídia somente quando der suporte para a tarefa do usuário em um contexto móvel, deixar o usuário decidir se vai iniciar mídia ou não, garantir que o conteúdo seja apresentado no formato do dispositivo móvel.
- **Design** – Design feito para uma varredura visual rápida da tela, considerar as orientações de exibição nos modos retrato e paisagem, repensar o layout da tela, não apenas reduzi-la.
- **Entrada do usuário** – Tornar fácil para os usuários inserir dados, oferecer função AutoCompletar e correção ortográfica, exibir valores padrão, oferecer mecanismos de entrada alternativos baseados nos recursos do dispositivo individual.
- **Contexto móvel** – Considerar alterações no contexto (hora do dia, localização, redes) e usar características e recursos do dispositivo para antecipar e dar suporte ao contexto de uso do usuário.
- **Usabilidade** – Garantir que dispositivos de interação (touch screens, teclados, áudio) e widgets (botões, links, barra de rolagem) funcionem bem juntos em todos os dispositivos pretendidos, seguir convenções e reduzir a curva de aprendizado.
- **Confiabilidade** – Ser sensível à privacidade e à segurança, não coletar informações pessoais sem permissão explícita do usuário, permitir que o usuário controle o modo como informações pessoais são compartilhadas e declarar suas práticas comerciais.
- **Feedback** – Obter informações importantes para seus usuários, minimizar o número de alertas, tornar os alertas breves e informativos, fornecer confirmação de ações do usuário sem interromper seu fluxo de trabalho.
- **Ajuda** – Tornar fácil para os usuários acessar opções de ajuda e suporte, oferecer ajuda contextual.

26.3.1 Teste de gestos

Touch screens são muito populares em dispositivos móveis e, como consequência, os desenvolvedores têm adicionado gestos multitoque (por exemplo, deslizar, aproximar e afastar, rolar, selecionar) como uma maneira de aumentar as possibilidades de interação do usuário sem perder espaço útil na tela. Infelizmente, interfaces que utilizam muito os gestos apresentam vários desafios para testes.

É difícil usar ferramentas automatizadas para testar ações de toque ou gesto na interface. É complicado registrar gestos precisamente para reprodução. A localização de objetos na tela é afetada pelo tamanho e pela resolução, assim como por ações anteriores do usuário. Não é possível usar protótipos

³ O blog pode ser encontrado em <http://www.mobileapptesting.com/10-key-components-of-successful-mobile-app-usability/2012/09/>.

⁴ *Agrupamento* é a prática de decompor documentos de hipermídia em grupos menores de informações relacionadas para permitir assimilação mais rápida por parte do leitor.

em papel, às vezes desenvolvidos como parte do projeto, para testar gestos adequadamente. Em vez disso, os testadores precisam criar programas (“frameworks”) de teste que chamem funções para simular eventos de gesto. Tudo isso é caro e demorado.

O teste de acessibilidade para usuários deficientes visuais é desafiador, pois as interfaces de gesto normalmente não fornecem retorno tátil ou auditivo. O teste de usabilidade e acessibilidade para gestos se torna muito importante para dispositivos onipresentes, como os smartphones. Pode ser importante testar o funcionamento do dispositivo quando não houver operações de gesto.

De maneira ideal, as jornadas de usuário ou casos de uso são escritos com detalhes suficientes para permitir sua utilização como base para scripts de teste. É importante recrutar usuários representativos e incluir todos os dispositivos pretendidos, a fim de levar em conta as diferenças de tela ao testar gestos com um aplicativo móvel. Por fim, os testadores devem garantir que os gestos obedeam aos padrões e contextos definidos para o dispositivo ou plataforma móvel.

26.3.2 Entrada e reconhecimento de voz

Agora, os dispositivos móveis inteligentes utilizam entrada de voz para permitir sua operação quando se está com as mãos e os olhos ocupados. A entrada de voz pode assumir várias formas, com diferentes níveis de complexidade de programação exigida para processar cada uma delas. A entrada de caixa postal ocorre quando uma mensagem é simplesmente gravada para reprodução posterior. O reconhecimento de palavras separadas pode ser usado para permitir que os usuários selecionem verbalmente itens de um menu com um pequeno número de escolhas. O reconhecimento de voz contínuo reflete tentativas de transformar fala ditada em sequências de textos com significados. Cada tipo de entrada de voz tem seus próprios desafios de teste.

De acordo com Shneiderman [Shn09], todas as formas de entrada e processamento de fala são atrapalhadas pela interferência de ambientes barulhentos. O uso de comandos de voz para controlar um dispositivo impõe uma maior carga cognitiva sobre o usuário, em comparação com apontar para um objeto na tela ou pressionar uma tecla. O usuário precisa pensar na palavra (ou palavras) correta para fazer o aplicativo móvel executar a ação desejada. No entanto, ao apontar para um objeto, o usuário precisa simplesmente reconhecer o objeto de tela apropriado e selecioná-lo. Contudo, a quantidade e a precisão dos sistemas de reconhecimento de fala estão evoluindo rapidamente, e é provável que o reconhecimento de voz se torne a forma dominante de comunicação em muitos aplicativos móveis.

Testar a qualidade e a confiabilidade da entrada e do reconhecimento de voz apresenta desafios técnicos até para as melhores empresas de teste. Entrada de voz errada (devido a erro do usuário, palavras ou frases pronunciadas incorretamente ou interferência ambiental) deve ser testada para garantir que uma entrada inválida não cause a falha do aplicativo móvel ou do dispositivo. Como cada combinação de usuário/dispositivo é diferente, uma ampla população de usuários deve ser envolvida para se garantir uma taxa de erros aceitável. Por último, é importante registrar os erros para que os desenvolvedores possam aprimorar a capacidade do aplicativo móvel de processar entrada de voz.

O teste de entrada de voz deve levar em conta as condições ambientais e a variação individual de voz.

26.3.3 Entrada por teclado virtual

Como um teclado virtual pode ocultar parte da tela de exibição quando ativado, é importante testar o aplicativo móvel para garantir que informações essenciais da tela não sejam ocultadas enquanto o usuário digita. Se as informações da tela precisam ser ocultadas, é importante testar a capacidade do aplicativo móvel de permitir que o usuário troque de página sem perder dados digitados [Sch09].

Normalmente, os teclados virtuais são menores do que os teclados de computador pessoal e, portanto, é difícil digitar com os 10 dedos. Como as próprias teclas são menores e mais difíceis de pressionar, além de não fornecerem retorno tátil, o aplicativo móvel deve ser testado para verificar se permite fácil correção de erros e se podem gerenciar palavras digitadas de forma errada sem falhar.

Tecnologias preditivas (isto é, função AutoCompletar para palavras digitadas parcialmente) são frequentemente utilizadas com teclados virtuais para ajudar a acelerar a entrada do usuário. É importante testar a exatidão das palavras completadas para a linguagem natural escolhida pelo usuário, caso o aplicativo móvel seja projetado para um mercado global. Também é importante testar a usabilidade de qualquer mecanismo que permita ao usuário ignorar um “completar” automático sugerido.

Os testes de teclados virtuais frequentemente são realizados no laboratório de usabilidade, mas alguns devem ser realizados em condições naturais. Se os testes do teclado virtual descobrirem problemas significativos, a única alternativa pode ser garantir que o aplicativo móvel possa aceitar entrada de outros dispositivos, que não um teclado virtual (por exemplo, um teclado físico ou entrada de voz).

26.3.4 Alertas e condições extraordinárias

Quando um aplicativo móvel executa em um ambiente de tempo real, existem fatores que podem ter impacto em seu comportamento. Por exemplo, um sinal de Wi-Fi pode ser perdido ou uma mensagem de texto, ligação telefônica ou alerta de calendário podem ser recebidos enquanto o usuário está trabalhando com o aplicativo móvel.

Esses fatores podem interromper o fluxo de trabalho do usuário do aplicativo, apesar de muitos usuários optarem por permitir alertas e outras interrupções enquanto trabalham. Seu ambiente de teste de aplicativo móvel deve ser capaz de simular esses alertas e condições. Além disso, você deve testar a capacidade do aplicativo de tratar os alertas e condições no ambiente de produção em dispositivos reais (Seção 26.5).

Parte do teste do aplicativo móvel deve se concentrar nas questões de usabilidade relacionadas a alertas e mensagens pop-up. O teste deve examinar a clareza e o contexto dos alertas, a conveniência de sua localização na tela de exibição do dispositivo e, quando estiverem envolvidos outros idiomas, verificar se a tradução de um idioma para outro está correta.

Muitos alertas e condições podem ser disparados de formas diferentes em vários dispositivos móveis ou por mudanças de rede ou de contexto. Embora muitos dos processos de tratamento de exceção possam ser simulados com um equipamento de teste de software, você não pode depender unicamente do

teste no ambiente de desenvolvimento. Novamente, isso enfatiza a importância de testar o aplicativo móvel em condições naturais, em dispositivos reais.

26.4 Teste além de fronteiras

“O mundo está sendo remodelado pela convergência das redes sociais, da computação móvel, da nuvem, de grandes quantidades de dados, da comunidade e outras forças poderosas. A combinação dessas tecnologias abre uma oportunidade incrível para conectar tudo de uma nova maneira e está transformando radicalmente nosso modo de viver e trabalhar.”

Marc Benioff

Internacionalização é o processo de criar um produto de software de modo que possa ser usado em vários países e com vários idiomas, sem exigir alterações de engenharia. *Localização* é o processo de adaptar um software aplicativo para uso em determinadas regiões do globo, adicionando-se requisitos específicos para a localidade e traduzindo-se elementos textuais para os idiomas apropriados. Além das diferenças nos idiomas, o trabalho de localização pode exigir levar em consideração a moeda, cultura, impostos e padrões (tanto técnicos como jurídicos) de cada país [Sla12]. Seria insensato lançar um aplicativo móvel em muitas partes do mundo sem testá-lo.

Como pode ser muito dispendioso construir instalações de teste internas em cada país para o qual está planejada a localização, muitas vezes é mais econômico terceirizar o teste para fornecedores locais em cada país [Reu12]. Contudo, o uso de uma estratégia de terceirização corre o risco de degradar a comunicação entre a equipe de desenvolvimento do aplicativo móvel e aqueles que estão realizando os testes de localização.

A *colaboração em massa* (*crowdsourcing*) tem se tornado popular em muitas comunidades online.⁵ Revenui [Reu12] sugere que ela poderia ser usada para empregar testadores de localização dispersos pelo mundo, fora do ambiente de desenvolvimento. Para isso, é importante encontrar uma comunidade que se orgulhe de sua reputação e tenha antecedentes de sucessos. Uma plataforma em tempo real e fácil de utilizar permite que os membros da comunidade se comuniquem com os tomadores de decisão do projeto. Para proteger a propriedade intelectual, só podem participar membros confiáveis da comunidade que estejam dispostos a assinar contratos de confidencialidade.

26.5 Problemas do teste em tempo real

Os dispositivos móveis reais têm limitações inerentes, acarretadas pela combinação de hardware e firmware neles existentes. Se a variedade de plataformas de dispositivo em potencial é grande, fica caro e demorado realizar testes de aplicativos móveis.

Os dispositivos móveis não são feitos tendo-se testes em mente. O poder de processamento e a capacidade de armazenamento limitados talvez não permitam carregar o software de diagnóstico necessário para registrar o desempenho do caso de teste. Muitas vezes, dispositivos emulados são mais fáceis de gerenciar e permitem aquisição de dados de teste mais fácil.

Cada rede móvel (existem centenas mundo afora) utiliza sua própria infraestrutura única para suportar a Web móvel. Os proxys Web implementados pelas

⁵ *Colaboração em massa* (*crowdsourcing*) é um modelo de solução de problemas distribuído, em que os membros da comunidade trabalham em soluções para problemas postados no grupo.

operadoras de Web móvel determinam como, quando e se você pode se conectar a recursos Web específicos usando suas redes. Isso pode restringir o fluxo de informações entre seu servidor e o cliente de teste. Alguns proxys podem retirar informações vitais de cabeçalhos http de que seu aplicativo necessita para fornecer funcionalidade ou adaptação a dispositivos. A intensidade do sinal da rede pode ser um problema. Os emuladores frequentemente não conseguem emular os efeitos e o timing de serviços de rede, e você pode não ver os problemas que os usuários terão ao executar o aplicativo móvel em um dispositivo real.

Um dispositivo móvel remoto é um equipamento de teste útil que pode ser usado para superar algumas das limitações dos emuladores. Um *dispositivo móvel remoto* é um telefone celular montado em uma caixa com uma unidade de controle remoto e uma antena remota. A unidade de controle remoto é conectada aos circuitos de controle da tela e do teclado do dispositivo. Quando conectada à Internet, essa solução permite que um usuário em um PC ou cliente Web local pressione botões e colete dados sobre o que está acontecendo no dispositivo remoto. Alguns dispositivos remotos têm a capacidade de gravar casos de teste para reprodução subsequente, a fim de ajudar no processo de automação do teste de regressão.

Por fim, é importante monitorar o consumo de energia especificamente associado ao uso do aplicativo móvel em um dispositivo móvel. A transmissão de informações de dispositivos móveis consome mais energia do que o monitoramento do sinal de uma rede. Processar streaming de mídia consome mais energia do que carregar uma página Web ou enviar uma mensagem de texto. A avaliação precisa do consumo de energia deve ser feita em tempo real, no dispositivo real e em condições naturais.

26.6 Ferramentas e ambientes de teste

Na Seção 26.3.2, discutimos os motivos para automatizar alguns aspectos do teste de aplicativos móveis, a fim de reduzir o tempo necessário e melhorar a qualidade e a cobertura do processo. Do mesmo modo, na Seção 26.2.5, discutimos a importância de testar no ambiente de produção. Contudo, existem ocasiões em que é necessário fazer testes manuais. Mas, mesmo nesses casos, ferramentas podem ser usadas para monitorar o comportamento dos aplicativos móveis e dos usuários em dispositivos entre redes.

Khode [Kho12b] sugere vários critérios a serem usados ao se escolher ferramentas de automação de teste móvel. Esses critérios também podem ser aplicados às ferramentas de teste móvel em geral.

- *Identificação do objeto* – A ferramenta pode reconhecer objetos de dispositivo usando uma variedade de métodos (por exemplo, identificação do objeto, processamento de imagens, reconhecimento de texto, objetos HTML5/DOM).
- *Segurança* – A ferramenta não deve exigir o uso de um dispositivo desprotegido conectado à Internet pública.
- *Dispositivos* – A ferramenta utiliza dispositivos de usuário reais, sem exigir o uso de modos de desenvolvedor especiais.

Quais critérios devo usar para escolher ferramentas de automação para testes móveis?

- **Funcionalidade** – Toda a funcionalidade do dispositivo é suportada, inclusive gestos multitoque, entrada com teclado virtual, chamadas e mensagens de texto recebidas, processamento de alertas e outros.
- **Emuladores e plug-ins** – O mesmo teste pode ser executado em diferentes dispositivos e diferentes sistemas operacionais usando-se o ambiente de teste existente.
- **Conectividade** – Conexão simultânea de vários dispositivos usando USB, Wi-Fi, nuvem privada e operadora de celular para testar a estabilidade e a cobertura da conexão.⁶

FERRAMENTAS DO SOFTWARE



Ferramentas selecionadas para teste de aplicativo móvel

Aqui está uma lista com várias ferramentas que podem ser úteis no teste de aplicativos móveis. Esse é um campo muito volátil. Esta lista de ferramentas representativas foi recentemente recomendada por Brown [Bro11] e Vinson [Vin11].⁶

Ferramentas de página Web móvel tentam determinar até que ponto uma página Web é amigável para um dispositivo móvel. O usuário digita uma URL da Web e a ferramenta fornece uma lista de defeitos.

Ferramenta representativa

WC3mobileOKChecker (<http://validator.w3.org/mobile/>)

Emuladores de navegador móvel mostram a aparência de uma página Web em navegadores móveis. O usuário digita uma URL da Web e a ferramenta mostra como ela apareceria em um navegador móvel.

Ferramentas representativas

Mobile Phone Emulator (<http://www.mobilephoneemulator.com/>)

iPhoneY (<http://www.marketcircle.com/iphoney/>)

Emuladores de dispositivo são dispositivos virtuais normalmente executados em um computador pessoal e permitem desenvolver e testar aplicativos móveis sem acessar dispositivos físicos.

Ferramentas representativas

Android Emulators (<http://developer.android.com/sdk/index.html>)

iPad Peek (<http://ipadpeek.com/>)

Adobe Edge Inspect (<http://html.adobe.com/edge/inspect/>)

Blackberry Simulators (<http://us.blackberry.com/sites/developers/resources/simulators.html>)

Ferramentas automatizadas registram interações no iOS ou no Android e permitem sua reprodução como scripts de teste. Normalmente, são executadas em um computador pessoal com um emulador de dispositivo.

Ferramentas representativas

MonkeyTalk (<http://www.gorillalogic.com/testing-tools/monkeytalk>)

Eggplant Mobile (<http://www.testplant.com/>)

Device Anywhere (<http://www.keynotedevicewhere.com/>)

Ferramentas de monitoramento de rede adicionam, modificam e filtram cabeçalhos de pedido HTTP enviados para servidores Web. São instaladas como um plug-in de navegador.

Ferramenta representativa

Modify headers (<https://addons.mozilla.org/enus/firefox/addon/modify-headers/>)

Teste analítico móvel coleta dados para analisar como os usuários interagem com o aplicativo móvel, o que é importante para avaliar o ROI (retorno sobre o investimento). Normalmente, exige um Web service ou serviço de nuvem para ajudar na coleta e no armazenamento de dados.

Ferramentas representativas

Flurry (<http://www.flurry.com/flurry-analytics.html>)

Google Mobile Analytics (<http://www.google.com/analytics/mobile/>)

Distimo Monitor (<http://monitor.distimo.com/>)

⁶ A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos revendedores.

26.7 Resumo

O objetivo do teste de aplicativos móveis é experimentar cada uma das muitas dimensões da qualidade do aplicativo com a finalidade de encontrar erros ou descobrir problemas que podem levar a falhas de qualidade. O teste se concentra em elementos da qualidade, como conteúdo, função, estrutura, usabilidade, uso do contexto, navegabilidade, desempenho, gerenciamento de energia, compatibilidade, operação conjunta, capacidade e segurança. Incorpora revisões e avaliações de usabilidade que ocorrem quando o aplicativo móvel é projetado e testes feitos depois que o aplicativo é implantado e distribuído em um dispositivo real.

A estratégia de teste para aplicativos móveis verifica cada dimensão da qualidade examinando inicialmente “unidades” de conteúdo, funcionalidade ou navegação. Uma vez validadas as unidades individuais, o foco passa para os testes que experimentam o aplicativo como um todo. Para tanto, são criados muitos testes sob a perspectiva do usuário e controlados por informações contidas em casos de uso. Um plano de teste para aplicativos móveis é desenvolvido e identifica as etapas do teste, os artefatos finais (por exemplo, casos de teste) e os mecanismos para a avaliação dos resultados. O processo de teste abrange vários tipos diferentes de teste.

Teste de conteúdo (e revisões) concentra-se nas várias categorias de conteúdo. O objetivo é examinar erros que afetam a apresentação do conteúdo para o usuário. O conteúdo precisa ser examinado quanto a problemas de desempenho impostos pelas restrições dos dispositivos móveis. O teste da interface examina os mecanismos de interação que definem a experiência do usuário fornecida pelo aplicativo móvel. O objetivo é descobrir erros resultantes quando o aplicativo móvel não leva em conta o contexto do dispositivo, do usuário ou da localização.

O teste de navegação é baseado em casos de uso, produzidos como parte da atividade de modelagem. Os casos de teste são projetados para examinar cada cenário de utilização em relação ao projeto de navegação dentro da estrutura arquitetural utilizada para implantar o aplicativo móvel. O teste de componente experimenta as unidades funcionais e de conteúdo do aplicativo móvel.

O teste de configuração tenta descobrir erros e/ou problemas de compatibilidade específicos de um dispositivo ou ambiente de rede em particular. São então aplicados testes para descobrir erros associados a cada configuração possível. Isso é complicado pelo grande número de dispositivos móveis e provedores de serviços de rede. O teste de segurança incorpora uma série de testes projetados para explorar vulnerabilidades no aplicativo móvel ou em seu ambiente. O objetivo é encontrar brechas na segurança, tanto no ambiente operacional do dispositivo quanto nos Web services acessados. O teste de desempenho abrange uma série de exames projetados para avaliar o tempo de resposta e a confiabilidade do aplicativo móvel quando aumentam as demandas de recursos do servidor. Por fim, o teste de aplicativos móveis deve tratar de problemas de desempenho, como utilização de energia, velocidade de processamento, limitações da memória, capacidade de recuperação em caso de falhas e problemas de conectividade.

Problemas e pontos a ponderar

- 26.1 Existem situações nas quais o teste do aplicativo móvel em dispositivos reais pode ser desconsiderado?
- 26.2 Com suas próprias palavras, discuta os objetivos do teste de um aplicativo móvel.
- 26.3 Compatibilidade é uma dimensão de qualidade importante. O que deve ser testado para garantir a compatibilidade de um aplicativo móvel?
- 26.4 Localize uma ferramenta de teste de aplicativo móvel gratuita. Critique o desempenho da ferramenta em relação ao aplicativo móvel que você conhece.
- 26.5 Quais elementos do aplicativo móvel podem passar por “teste de unidade”? Que tipos de testes devem ser executados somente depois que os elementos do aplicativo móvel estiverem integrados?
- 26.6 É sempre necessário desenvolver um plano de teste escrito formal? Explique.
- 26.7 É correto dizer que a estratégia de teste global de aplicativos móveis começa com os elementos visíveis para o usuário e prossegue para os elementos tecnológicos? Há exceções a essa estratégia?
- 26.8 O teste de certificação é *realmente* um teste no sentido convencional? Explique.
- 26.9 Descreva os passos associados ao teste da experiência do usuário para um aplicativo móvel.
- 26.10 Suponha que você está desenvolvendo um aplicativo móvel para acessar uma farmácia online que atende idosos. A farmácia tem funções típicas, mas também mantém um banco de dados de cada cliente para que possa fornecer informações sobre medicamentos e alertar sobre interações de certos medicamentos. Discuta quaisquer testes especiais de usabilidade para esse aplicativo móvel.
- 26.11 Suponha que você implementou um Web service que fornece uma função de verificação–interação medicamentosa para **YourCornerPharmacy.com** (consulte o Problema 26.10). Discuta os tipos de testes em nível de componente que teriam de ser feitos no dispositivo móvel para garantir que o aplicativo móvel acesse essa função corretamente.
- 26.12 Como pode ser testada a capacidade de um aplicativo móvel de levar em conta o contexto?
- 26.13 É possível testar cada configuração que um aplicativo móvel provavelmente vai encontrar no ambiente de produção? Se não, como você seleciona um conjunto significativo de testes de configuração?
- 26.14 Descreva um teste de segurança que talvez precise ser realizado para o aplicativo móvel **YourCornerPharmacy** (Problema 26.10). Quem deve realizar esse teste?
- 26.15 Qual a diferença entre o teste de esforço de uma WebApp e de um aplicativo móvel?

Leituras e fontes de informação complementares

Muitos livros descrevem a computação móvel e frequentemente contêm discussões sobre testes de aplicativos móveis. Kumar e Xie (*Handbook of Mobile Systems Applications and Services*, Auerbach Publications, 2012) editaram um livro que aborda serviços móveis e a função das arquiteturas voltadas para serviço na computação móvel. Livros sobre computação onipresente de Chalmers (*Sensing and Systems in Pervasive Computing: Engineering Context Aware Systems* Springer, 2011), Adelstein (*Fundamen-*

tals of Mobile and Pervasive Computing, McGraw-Hill, 2004) e Hansmann (*Pervasive Computing: The Mobile World*, 2ª ed., Springer, 2003) definem os princípios do contexto na computação móvel. O livro de Nguyen et al. (*Testing Applications on the Web: Test Planning for Mobile and Internet-Based Systems*, 2ª ed., Wiley, 2003) discute o teste de aplicativos móveis com ênfase no teste de acessibilidade, confiabilidade e segurança.

Existem muitos livros sobre projeto de interface. Muitos deles contêm informações sobre teste de usabilidade de aplicativo móvel. O livro de Ben Shneiderman e colegas (*Designing the User Experience*, 5ª ed., Addison-Wesley, 2009) é um trabalho clássico sobre a usabilidade. Outras boas referências gerais são os livros de Quesenberry e Szuc (*Global UX: Design and Research in a Connected World*, Morgan Kaufmann, 2011) e Schumacher (*Handbook of Global User Research*, Morgan Kaufmann, 2009). Nielsen (*Mobile Usability*, New Riders, 2012) oferece conselhos sobre como projetar interfaces úteis que levam em conta o tamanho da tela de dispositivos móveis. Colborne (*Simple and Usable Web, Mobile, and Interaction Design*, New Riders, 2010) descreve o processo de simplificação da interação do usuário. Ginsburg (*Designing the iPhone User Experience: A User-Centered Approach to Sketching and Prototyping iPhone Apps*, Addison-Wesley, 2010) discute a importância de adotar uma estratégia centrada no usuário para avaliar a experiência do usuário.

Meier (*The Microsoft Application Architecture Guide*, 2ª ed. Microsoft Press, 2009) editou um livro que contém informações úteis sobre testes de aplicativos móveis. Um livro de Graham (*Experiences of Test Automation: Case Studies of Software Test Automation*, Addison-Wesley, 2012) fornece uma boa base sobre automação de testes. Lee (*Test-Driven iOS Development*, Addison-Wesley, 2012) discute o processo de teste de aplicativos móveis no contexto do projeto orientado a testes.

Há uma grande variedade de recursos de informação sobre teste de WebApp disponível na Internet. Uma lista atualizada das referências da Web (em inglês) pode ser encontrada no site: www.mhhe.com/pressman.

27

Engenharia de segurança

Conceitos-chave

ameaça	
análise	585
modelagem	594
ataque	585
padrão	589
superfície	596
ativo	585
caso de garantia	592
exposição	585
segurança	
caso de	592
engenharia da	588
garantia de	592
modelo de	590
requisitos de	585
verificação de	
credibilidade	591

Devanbu e Stubblebine [Dev00] tecem o seguinte comentário, como introdução às suas diretrizes para engenharia de segurança:

Existe algum sistema de software que não precise ser seguro? Quase todo sistema controlado por software enfrenta ameaças de adversários em potencial, desde aplicativos clientes para Internet executando em PCs e complexos sistemas de telecomunicações e energia acessíveis pela Internet até produtos de software com mecanismos de proteção contra cópia. Os engenheiros de software precisam ter conhecimento dessas ameaças e de sistemas de engenharia com defesas confiáveis enquanto ainda entregam valor aos consumidores.

As ameaças sobre as quais os autores discutiam há mais de uma década se multiplicaram com o explosivo crescimento da Web, a ubiquidade de aplicativos móveis e o amplo uso da nuvem. Cada uma dessas tecnologias gerou novas preocupações sobre privacidade do usuário e a possível perda ou roubo de informações pessoais. A segurança não preocupa apenas as pessoas que desenvolvem software para instituições militares, governamentais ou órgãos

PANORAMA

O que é? Engenheiros de segurança constroem sistemas que têm a capacidade de proteger seus ativos contra ataques. Usando análise de ameaça, você pode determinar os controles exigidos para reduzir a exposição resultante quando ataques exploram vulnerabilidades do sistema. A segurança é um pré-requisito fundamental de fatores da qualidade do software, como integridade, disponibilidade, confiabilidade e proteção.

Quem realiza? Engenheiros de software trabalhando junto aos consumidores e quaisquer outros envolvidos que dependam de resultados ou serviços do sistema.

Por que é importante? Cada tecnologia emergente traz consigo novas preocupações sobre privacidade do usuário e a possível perda ou roubo de informações valiosas. A segurança não se preocupa apenas com as pessoas que desenvolvem software para instituições militares, governamentais ou órgãos da saúde. Atualmente, a segurança deve ser uma preocupação de qualquer engenheiro de software que tenha recursos dos clientes para proteger.

Quais são as etapas envolvidas? Primeiramente são identificados os ativos do sistema e é determinada a exposição à perda devido a brechas na segurança. Então, a arquitetura do sistema é modelada no nível de componente. Em seguida, são criados uma especificação dos requisitos de segurança e um plano de mitigação de risco. À medida que o sistema é construído, a garantia da segurança é pré-formada e continua ao longo do processo do software.

Qual é o artefato? Os principais artefatos são uma especificação de segurança (que pode fazer parte de um modelo de requisitos) e um caso de segurança documentado, que faz parte dos documentos da garantia da qualidade do sistema. Para desenvolver esses artefatos, também são criados um modelo de ameaça e um plano de avaliação de risco à segurança e de mitigação de risco.

Como garantir que o trabalho foi realizado corretamente? Use a evidência resultante das análises, inspeções e resultados de teste da segurança e apresente como um caso de segurança que permita aos envolvidos no sistema avaliar seu grau de credibilidade de que ele protege seus ativos e preserva sua privacidade.

da saúde. Atualmente, a segurança deve ser uma preocupação de qualquer engenheiro de software que tenha recursos dos clientes para proteger.

Na acepção mais simples, a segurança de software fornece os mecanismos que permitem a um sistema de software proteger seus ativos contra ataques. *Ativos* são recursos de sistema que têm valor para um ou mais envolvidos. Os ativos incluem informações de bancos de dados, arquivos, programas, espaço de armazenamento no disco rígido, memória de sistema ou até mesmo capacidade do processador. Os *ataques* frequentemente tiram proveito de pontos fracos ou de vulnerabilidades do software que permitem acesso não autorizado a um sistema. Por exemplo, uma vulnerabilidade comum poderia ser proveniente da falha na autenticação de usuários antes de permitir o acesso a recursos valiosos do sistema.

Segurança é um aspecto da garantia da qualidade do software (Capítulo 21). Anteriormente no livro, você aprendeu que não é possível aumentar a qualidade de um sistema respondendo a relatos de erros. De modo similar, é muito difícil aumentar a segurança de um sistema existente respondendo a relatos de vulnerabilidades exploradas [Gho01]. As preocupações com a segurança devem ser consideradas no início do processo do software, incorporadas ao projeto do software, implementadas como parte da codificação e verificadas durante os testes e a distribuição.

Neste capítulo, apresentamos um levantamento de importantes questões na engenharia de segurança de software. Uma discussão completa desse assunto está fora dos objetivos deste livro. Mais informações podem ser encontradas em [All08], [Lip10] e [Sin08].

"A segurança é sempre excessiva, até que não seja suficiente."

Robbie Sinclair

27.1 Análise dos requisitos de segurança

Os requisitos de segurança de software são determinados pelo trabalho junto ao cliente com o fim de identificar os ativos que precisam ser protegidos e o custo associado à perda desses ativos, caso ocorra uma brecha na segurança. O valor da perda de um ativo é conhecido como sua *exposição*. As perdas podem ser medidas em termos de tempo ou custo para recuperar ou recriar um ativo. Ativos com valores insignificantes não precisam ter segurança.

Uma parte importante da construção de sistemas seguros é antecipar condições ou ameaças que possam ser utilizadas para danificar recursos do sistema ou para torná-los inacessíveis a usuários autorizados. Esse processo é chamado *análise de ameaça*. Uma vez identificados os ativos, as vulnerabilidades e as ameaças do sistema, podem ser criados controles para evitar ataques ou mitigar seus danos.

A segurança do software é um pré-requisito fundamental de sua integridade, disponibilidade, confiabilidade e proteção (Capítulo 19). Talvez não seja possível criar um sistema que possa defender seus ativos contra todas as ameaças imagináveis; por isso, pode ser necessário estimular os usuários a manter cópia de backup de dados críticos, componentes de sistema redundantes e garantir o emprego de controles de privacidade.

Concentre-se nos bens de maior valor e maior exposição.

CASASEGURA



Preocupações dos envolvidos com relação à segurança

Cena: Área de trabalho da equipe de engenharia de software.

Atores: Jamie Lazar, membro da equipe de software; Vinod Raman, membro da equipe de software; Ed Robbins, membro da equipe de software; Doug Miller, gerente da engenharia de software; Lisa Perez, membra da equipe de Marketing; e um representante da Engenharia de Produto.

Conversa:

Vinod: Se não houver problemas, vou atuar como facilitador nesta reunião.
(Todos acenam com a cabeça, concordando.)

Precisamos começar a determinar o que nos preocupa com relação à segurança do projeto *CasaSegura*.

Doug: Podemos começar listando o que estamos preocupados em proteger?

Jamie: Bem, e se um intruso invadir o sistema *CasaSegura* e conseguir roubar ou danificar a casa de alguém?

Lisa: A reputação da empresa seria prejudicada, caso fosse divulgado que algum hacker desabilitou nossos sistemas.

Jamie: Sem falar na responsabilidade civil, se fosse determinado que o sistema foi mal projetado.

Doug: A interface web para o produto possibilita que alguém intercepte senhas ao serem transmitidas.

Ed: Temos que ressaltar que a interface web vai exigir um banco de dados contendo informações dos clientes; portanto, temos preocupações com a privacidade.

Vinod: Talvez este seja um bom momento para que alguém perca uns 10 minutos listando cada bem que pode ser perdido ou comprometido por um ataque.
[10 minutos se passam.]

Vinod: Certo, vamos afixá-los no quadro de avisos e ver se existem preocupações semelhantes.
[15 minutos depois, a lista está criada.]

Lisa: Parece que são muitas as preocupações. Como podemos tratar de todas elas?

Doug: Precisamos priorizar nossa lista com base no custo para reparar o dano causado pela perda do ativo.

Lisa: Como podemos fazer isso?

Vinod: Precisamos obter os custos reais da substituição de ativos perdidos usando dados históricos do projeto. E, Lisa, você precisa falar com o jurídico para sabermos qual poderia ser nossa responsabilidade civil.

27.2 Segurança e privacidade em um mundo online

A atividade na Internet está mudando da navegação tradicional com desktops e passando a cenários nos quais os navegadores fornecem conteúdo dinâmico e personalizado. Os usuários podem inserir seus próprios conteúdos em fóruns de comunidades que incorporam “mashups” de dados de sites de terceiros. Muitos aplicativos de desktop utilizam interfaces de navegador web para acessar dados locais e oferecem uma experiência de usuário uniforme em várias plataformas de computação. Como consequência, os desenvolvedores web precisam de controle e mecanismos de segurança melhores. Dados e código de fontes não confiáveis não devem ter os mesmos privilégios que código de um programador confiável. Para que o navegador web seja uma interface de usuário eficiente, a privacidade, a credibilidade e a segurança precisam estar entre seus atributos de qualidade mais importantes [Sei11].

Em muitos casos, informações confidenciais de um usuário fluem além dos limites organizacionais na Internet. Por exemplo, informações eletrônicas de um paciente talvez precisem ser compartilhadas entre hospitais, companhias de seguro e médicos. Da mesma forma, informações sobre viagens talvez precisem ser compartilhadas entre agências de viagem, hotéis e companhias aéreas. Nessas situações, os usuários precisam revelar informações pessoais para receber um serviço. Muitas vezes, o usuário não tem controle sobre o que uma organização faz com essas informações, quando recebidas em forma digi-

“Depender do governo para proteger nossa privacidade é como pedir a um voyeur para que instale suas persianas.”

John Perry Barlow

tal. Em circunstâncias ideais, os usuários podem controlar diretamente como seus dados devem ser manipulados e compartilhados, mas isso exige a criação de preferências de compartilhamento de dados prescritas pelo usuário para os dados em si, ao serem capturados de forma eletrônica [Pea11].

27.2.1 Mídia social

O explosivo crescimento e a popularidade das redes de mídia social online as transformaram em alvos atraentes para programadores mal-intencionados. Devido à absoluta confiança que a maioria dos usuários tem em um ambiente de rede social, é fácil para um hacker utilizar uma conta comprometida para enviar mensagens infectadas com malware para amigos do titular da conta. Redes sociais podem ser usadas para atrair usuários para sites de phishing¹ com a intenção de persuadi-los a enviar dados pessoais ou mandar dinheiro para um falso amigo necessitado. Outra artimanha é o uso de e-mails que incluem detalhes roubados dos dados pessoais de um usuário [Sae11].

Algumas redes de mídia social permitem que os usuários desenvolvam seus próprios aplicativos. Esses aplicativos pedem inocentemente para que o usuário conceda acesso a dados pessoais e, então, utilizam esses dados de formas jamais pretendidas por ele. Às vezes, um aplicativo ou jogo é tão atraente, que mesmo um usuário bem-informado permitirá esse tipo de acesso para poder usá-lo. Muitas vezes, as redes sociais têm recursos de inscrição que permitem a criminosos atingir pessoas por meio de suas atividades na vida real. Apesar das preocupações em relação ao roubo de identidade, spam e spyware, muitos usuários de computador não são motivados a dar os passos necessários para se protegerem [Sta10].

“Se você revelar seus segredos ao vento, não deve culpá-lo por ele revelá-los às árvores.”

Kahlil Gibran

27.2.2 Aplicativos móveis

Os usuários de aplicativos móveis têm acesso a praticamente os mesmos web services que os usuários da Internet fixa. Os usuários da Internet sem fio herdaram todos os riscos de segurança associados ao comércio de desktops, além de muitos riscos novos das redes móveis. A natureza das redes sem fio exige confiança e cooperação entre os nós, os quais podem ser explorados por programas mal-intencionados para negar serviços ou coletar informações confidenciais. As plataformas e linguagens desenvolvidas para dispositivos móveis podem ser hackeadas, e código mal-intencionado pode ser inserido no software de sistema do dispositivo com todos os privilégios de seu proprietário. Isso significa que tecnologias de segurança, como assinaturas, autenticação e criptografia, podem ser minadas com relativa facilidade.

Quais ameaças são encontradas nos aplicativos móveis?

27.2.3 Computação em nuvem

A computação em nuvem apresenta preocupações adicionais com a confidencialidade e a privacidade, pois dados são entregues aos cuidados de servidores remotos gerenciados por provedores de serviço. Esses provedores de serviços de nuvem têm acesso e controle completos sobre nossos dados. Eles são

¹ Um *site de phishing* disfarça-se de site conhecido e confiável, atraindo o usuário a fornecer informações pessoais que podem levar à perda de ativos de segurança.

confiáveis e não os compartilham com outros (nem acidentalmente nem deliberadamente) e adotam medidas responsáveis para impedir sua perda. Repositórios de dados online são fontes muito atraentes para “data mining” (por exemplo, para coletar informações demográficas ou de marketing). O problema é que o autor dos dados não deu seu consentimento explícito [Rya11a]. Os legisladores devem criar políticas e regulamentos para garantir que os provedores de serviços não abusem da confiança de seus usuários.

O limite entre o “interior confiável” e o “exterior não confiável” fica indistinto quando uma empresa adota a computação em nuvem. Com os dados e aplicativos fora da organização, é possível um novo tipo de intruso mal-intencionado. Dados confidenciais estão a apenas alguns comandos distantes do acesso por um administrador de sistema mal-intencionado ou incompetente. A maioria dos provedores de serviços de nuvem tem procedimentos rígidos em vigor para monitorar o acesso de funcionários aos dados dos clientes. Políticas para evitar o acesso físico aos dados não são eficazes contra ataques remotos, e o monitoramento frequentemente detecta um ataque somente depois que ele acontece. Para estimular a confiança dos usuários em um sistema na nuvem, pode ser importante dar a eles a capacidade de avaliar se os mecanismos necessários para proteger a confidencialidade e a privacidade estão em vigor [Roc11].

A ubiquidade do acesso à Web e o advento da computação em nuvem permitiram novas formas de colaborações de negócio. Compartilhar informações e proteger a confidencialidade é uma tarefa difícil. Tornar segura a computação de várias partes aumenta os riscos de comportamento egoísta, a não ser que todas as partes tenham confiança de que ninguém pode tirar proveito dos sistemas. Essa situação enfatiza uma dimensão psicológica da credibilidade e da segurança dos sistemas que não pode ser resolvida apenas pela engenharia de software [Ker11].

27.2.4 A Internet das coisas

Alguns visionários descrevem uma “Internet das coisas” [Rom11], na qual tudo que é real tem um equivalente virtual na Internet. Essas entidades virtuais podem produzir e consumir serviços e colaborar para um objetivo comum. Por exemplo, o telefone de um usuário conhece o estado físico e mental do usuário por meio de uma rede de dispositivos circundantes que podem atuar em nome desse usuário. Engenheiros automotivos imaginam carros que se comunicam de forma autônoma com outros veículos, fontes de dados e dispositivos e que não exigem controle direto do motorista.

Contudo, a segurança é um dos maiores obstáculos no caminho dessa visão. Sem fortes bases de segurança, ataques e defeitos superarão qualquer um dos benefícios alcançados por uma Internet das coisas. Os legisladores devem considerar o equilíbrio entre governança e inovação. Governança excessiva pode facilmente obstruir a inovação, e, por sua vez, a inovação pode involuntariamente ignorar direitos humanos [Rom11].

27.3 Análise da engenharia de segurança

As tarefas de análise da segurança incluem levantamento de requisitos, modelagem de ameaças, análise de risco, projeto de medidas e verificações de

exatidão. Essas tarefas consideram os detalhes funcionais e não funcionais do sistema, juntamente com seu caso de negócio [Bre03].

27.3.1 Levantamento de requisitos de segurança

As técnicas gerais de levantamento de requisitos discutidas no Capítulo 8 são igualmente aplicáveis ao levantamento de requisitos de segurança. Com relação à segurança, os requisitos são não funcionais² e influenciam fortemente o *projeto arquitetural* dos sistemas de software. Uma vez refinados e priorizados os requisitos de sistema, utilizando-se modelagem de ameaças e análise de risco, podem ser definidas as políticas de segurança do sistema. Essas políticas serão refinadas com modelagem de segurança e decomposição, junto com considerações sobre a utilização, para produzir a arquitetura de segurança exigida. Antes de serem implementados, os aspectos relacionados à segurança da arquitetura são validados [Bod09].

Em alguns casos, os requisitos de segurança e outros requisitos de software podem entrar em conflito. Por exemplo, segurança e usabilidade podem ter divergências, e deve ser encontrado um equilíbrio entre as duas. Sistemas altamente seguros frequentemente são mais difíceis de usar por parte de usuários inexperientes. Na *engenharia de segurança centrada no usuário*, o levantamento de requisitos de segurança encontra respostas para três perguntas importantes [Mar02]: (1) Quais são as necessidades dos usuários com relação ao software de segurança?; (2) Como pode ser projetada uma arquitetura segura que favoreça um bom projeto de interface de usuário?; (3) Como pode ser projetada uma boa interface de usuário que seja segura, mas, ao mesmo tempo, permita utilização eficaz, eficiente e satisfatória? As respostas dessas perguntas devem ser incorporadas nos cenários de caso de uso (Capítulo 8) que descrevem como os envolvidos interagem com os recursos do sistema.

À medida que o levantamento de requisitos prossegue, o analista deve identificar padrões de ataque. Um *padrão de ataque* é um tipo de padrão de projeto (Capítulo 16) que identifica as falhas de segurança de um sistema. Os padrões de ataque podem acelerar a análise da segurança, fornecendo pares problema/solução para vulnerabilidades de segurança comuns. A reutilização de padrões de ataque pode ajudar os engenheiros a identificar vulnerabilidades do sistema. Não há necessidade de recriar diferentes maneiras de atacar um sistema. Os padrões de ataque permitem aos desenvolvedores usar nomes bem conhecidos (por exemplo, phishing, injeção de SQL e scripting entre sites) para problemas de segurança de software. Com o passar do tempo, os padrões de ataque comuns podem ser aprimorados [Sin08]. A dificuldade para usar padrões de ataque é saber quando um padrão específico se aplicará.

Alguns engenheiros de software acreditam que os rigores da engenharia de segurança são incompatíveis com a natureza informal do levantamento de requisitos em processos ágeis (Capítulo 5). No entanto, uma técnica que poderia ser usada para reconciliar a “falta de formalidade” é a criação de histórias de violação na área de competência dos requisitos. *Histórias de violação* são baseadas em informações do cliente que descrevem ameaças aos ativos do sistema. Elas ampliam o conceito ágil já bem estabelecido das histórias de

“Os usuários vão sempre preferir os porcos dançantes, em vez da segurança.”

Bruce Schneier

Quais perguntas sobre levantamento de requisitos de segurança devo fazer?

“Tornar um sistema de computador seguro tradicionalmente tem sido uma batalha de inteligência: o invasor tenta encontrar as brechas, e o projetista tenta fechá-las.”

Gosser

² Às vezes, eles são chamados de *preocupações transversais* – e foram discutidos no Capítulo 4.

usuário e podem ajudar a conseguir a rastreabilidade dos requisitos de segurança necessária para permitir que a garantia da segurança prossiga [Pee11].

27.3.2 Modelagem de segurança

Modelagem é um processo importante para especificar e analisar requisitos. Um *modelo de segurança* é uma descrição formal da *política de segurança* do sistema de software. Uma política de segurança fornece uma definição da segurança do sistema representando seus principais requisitos e também contém regras descrevendo como a segurança será imposta durante a operação do sistema.

O modelo de segurança pode fornecer orientação precisa durante os processos de projeto, codificação e revisão. Uma vez construído o sistema, o modelo de segurança fornece uma base para auxiliar na verificação da exatidão da implementação da segurança [Dan09]. O modelo de segurança também é uma referência valiosa à medida que o sistema evolui ou exige reparo durante atividades de manutenção.

Quais informações
estão contidas em um
modelo de segurança?

Um modelo de segurança pode ser representado usando texto ou formalismo gráfico. Independentemente de sua representação, um modelo de segurança precisa contemplar os seguintes itens: (1) objetivos da política de segurança, (2) requisitos da interface externa, (3) requisitos de segurança do software, (4) regras de operação e (5) especificações descrevendo a correspondência modelo-sistema.

Alguns modelos de segurança são representados com máquinas de estado.³ Cada estado deve conter informações sobre aspectos relevantes da segurança do sistema. Como engenheiro de software preocupado com a segurança, você deve garantir que quaisquer transições de estado permitidas no sistema comecem e terminem em um estado seguro. Deve também confirmar se o estado inicial do sistema é seguro. Para ser completo, o modelo precisa ter uma interpretação que mostre como ele se relaciona com o sistema real.

Formalidades de modelagem executáveis permitem aos desenvolvedores verificar um modelo de segurança e seu comportamento antes de aceitá-lo. Uma vez aceito, o modelo forma uma boa base para o projeto. Duas linguagens usadas para modelar requisitos de segurança são *UML.sec* (uma extensão da UML que utiliza estereótipos e restrições) e *GRL* (linguagem de requisitos baseada em metas para capturar requisitos não funcionais). O uso de linguagens de modelagem formais pode ajudar a aumentar a credibilidade do sistema à medida que ele é desenvolvido [Sal11].

Métodos formais (Capítulo 28) foram propostos como um modo de ampliar a análise e a verificação da segurança de sistemas. O uso de especificações formais para os requisitos de segurança tem o potencial de ajudar na criação de casos de teste para examinar a segurança baseada em modelos. O uso de provas de exatidão formais para componentes críticos do sistema pode aumentar a confiança do desenvolvedor de que o sistema realmente obedece à sua especificação. É claro que se deve tomar cuidado para que as suposições básicas das provas sejam satisfeitas.

³ Uma *máquina de estado finito* é definida por uma lista dos estados de transição possíveis a partir de cada estado atual e pela condição de disparo de cada transição.

27.3.3 Projeto de medidas

Para ser seguro, o software deve exibir três propriedades: *confiabilidade* (opere sob condições hostis), *credibilidade* (o sistema não se comportará de forma mal-intencionada) e *capacidade de sobrevivência* (continua a funcionar quando comprometido).⁴ As métricas⁵ e medidas de segurança precisam se concentrar na avaliação dessas propriedades.

Métricas de segurança úteis devem se basear em medidas que permitam aos desenvolvedores avaliar até que ponto a confidencialidade dos dados ou a integridade do sistema podem estar em risco. Três medidas necessárias para criar essas métricas são o *valor do ativo*, a *probabilidade de ameaça* e a *vulnerabilidade do sistema*. Não é fácil medir essas propriedades diretamente. O custo da perda de um ativo pode ser maior do que o custo de recriá-lo.

As melhores medidas são aquelas que estão prontamente disponíveis durante o desenvolvimento ou operação do software. O número de queixas relacionadas à segurança ou o número de casos de teste de segurança malsucedidos pode oferecer algumas medidas (por exemplo, o número de incidentes de roubo de identidade informados a cada mês). Vulnerabilidades podem não ser conhecidas até que ocorram ataques, mas o número de ataques bem-sucedidos pode ser contado.

27.3.4 Verificações de exatidão

As verificações de exatidão da segurança precisam ocorrer ao longo do ciclo de desenvolvimento do software. A exposição de ativos dos envolvidos a ataques contra vulnerabilidades do sistema deve ser determinada no início do processo de desenvolvimento.

Então, a equipe de software garante que o modelo de ameaça extraído dos casos de uso do sistema foi considerado como parte da mitigação de riscos, monitoramento e plano de gerenciamento da segurança. As atividades de garantia da qualidade incluem a identificação de padrões de segurança e o desenvolvimento de diretrizes de segurança para usar durante as atividades de modelagem e construção. As atividades de verificação do software garantem que os casos de teste de segurança sejam completos e rastreáveis de acordo com os requisitos de segurança do sistema.

Muitas dessas verificações de segurança devem ser incluídas nas auditorias, inspeções e atividades de teste incorporadas às tarefas de engenharia de software convencionais (Seção 27.6). Os dados coletados durante essas verificações são analisados e resumidos como parte do *caso de segurança* do sistema, conforme descrito na Seção 27.4. O processo de *verificação da credibilidade* é discutido na Seção 27.7.

Um software seguro deve exibir três propriedades: *confiabilidade*, *credibilidade* e *capacidade de sobrevivência*.

"Teoricamente, alguém pode construir sistemas comprovadamente seguros. Teoricamente, a teoria pode ser aplicada à prática, mas, na prática, não."

M. Dacie

⁴ <https://buildsecurityin.us-cert.gov>.

⁵ *Métricas* são indicadores quantitativos que mostram até que ponto um componente ou processos de um sistema possuem determinado atributo. Boas métricas satisfazem os critérios SMART (isto é, eSpecificas, Mensuráveis, Atingíveis, Repetíveis e dependentes do Tempo). Normalmente, são derivadas de medidas que utilizam técnicas estatísticas para descobrir as relações. Uma discussão adicional sobre métricas e medidas está no Capítulo 30.

27.4 Garantia da segurança

Como software está integrado em nossas vidas, falhas na segurança e as perdas associadas a ele se tornam mais dispendiosas e fatídicas. A boa prática de engenharia de software envolve averiguar requisitos, desenvolver um projeto adequado e demonstrar que você criou o produto certo. A garantia da segurança tem a intenção de demonstrar que você construiu um produto seguro e, como consequência, inspira confiança entre os usuários e outros envolvidos.

27.4.1 O processo da garantia da segurança

A verificação faz parte da tarefa que fornece evidência de que os envolvidos podem estar confiantes de que o software está de acordo com os requisitos. Quando considerados no contexto da engenharia de segurança, você escolhe um subconjunto básico dos requisitos ou reclamações de segurança para o software e cria um caso de garantia demonstrando que o software está adequado a esses requisitos ou reclamações.

Um caso de segurança apoia a alegação de que o software é seguro.

Um *caso de garantia* é um artefato fundamentado e verificável que apoia a alegação de que o software resolve a reclamação que está sendo feita. Casos de garantia são usados há muito tempo para proteção de software e, agora, estão sendo utilizados para segurança de software; frequentemente, são chamados de *casos de segurança*.

Cada caso de segurança consiste em três elementos: (1) as próprias reclamações, (2) os argumentos que vinculam as reclamações por meio de evidência e suposições e (3) o conjunto de provas e suposições explícitas que apoiam os argumentos.

Para um caso de segurança ser válido, ele deve satisfazer três objetivos. Deve especificar reclamações convenientes e adequadas para o sistema em questão, documentar que foram aplicadas práticas de engenharia convenientes para que as reclamações possam ser resolvidas e mostrar a realização dessas reclamações dentro do nível de risco exigido [Red10].

Vários tipos de evidência podem ser usados para provar o caso de segurança. Provas formais da exatidão do software (Capítulo 28) podem ser úteis caso o código tenha sido projetado com a intenção de provar sua exatidão. Existem ferramentas que suportam verificação automática de software [DSi08] e outras que realizam varreduras estáticas em busca de fraquezas na segurança do software (por exemplo, RATS, ITS4, SLAM).⁶ Mas apenas ferramentas não podem construir um caso de segurança.

Parte da evidência será proveniente da análise dos artefatos do sistema, usando variações de análises técnicas formais e inspeções dos artefatos (Capítulo 20). Contudo, essas análises se concentram exclusivamente nas reclamações relacionadas à segurança. Examinadores especialistas em segurança podem analisar o sistema ou seu caso de segurança. Uma lista de avaliações também pode ser usada para verificar se as diretrizes e passos do processo de segurança foram seguidos.

⁶ Para uma lista de ferramentas de segurança, consulte: <http://www.tech-faq.com/how-to-find-security-vulnerabilities-in-source-code.html>.

CASASEGURA

**Construção do caso de segurança**

Cena: Área de trabalho da equipe de engenharia de software.

Atores: Jamie Lazar, membro da equipe de software; Vinod Raman, membro da equipe de software; Ed Robbins, membro da equipe de software; e Bridget Thornton, líder do grupo de qualidade do software.

Conversa:

Ed: Obrigado por se juntar a nós, Bridget. Precisamos construir o caso de segurança para o projeto *CasaSegura*.

Vinod: Como devemos começar?

Bridget: Poderíamos começar escolhendo uma preocupação com a segurança e ver qual evidência podemos encontrar para apoiar seu caso.

Ed: Que tipo de evidência?

Bridget: Vamos escolher uma das preocupações com a segurança primeiro.

Vinod: Vamos nos concentrar nas preocupações relacionadas à segurança do banco de dados do cliente.

Bridget: Certo, vamos começar listando as reclamações com a segurança feitas para o acesso ao banco de dados.

Jamie: Você quer dizer os elementos do modelo de segurança referentes ao banco de dados?

Bridget: Sim. Em seguida, damos uma olhada nas checklists de inspeção e nos resumos das análises técnicas formais que acontecem à medida que cada ponto de controle do projeto é concluído.

Ed: E quanto às auditorias de processo e documentos de pedido de alteração produzidos por seu grupo?

Bridget: É importante incluí-los também.

Vinod: Usamos um ITG para criar e executar a maioria dos casos de teste do sistema.

Bridget: Um resumo do comportamento dos casos de teste de segurança comparando a saída esperada e a saída real é uma parte muito importante do caso de segurança.

Jamie: Parece muita informação para assimilar.

Bridget: E é mesmo. Por isso, o próximo passo é pegar cada reclamação feita à segurança do banco de dados e resumir a evidência que apoia ou refuta a reclamação de proteção de bens adequada.

Ed: Você pode nos ajudar a examinar nosso caso de segurança quanto estiver montado?

Bridget: Naturalmente. Meu grupo precisa dialogar continuamente com sua equipe à medida que esse projeto avançar, tanto antes quanto depois do lançamento.

27.4.2 Organização e gerenciamento

A pressa de lançar software rapidamente muitas vezes obriga os gerentes de projeto a se preocuparem mais com recursos e funções do que com a segurança. Os engenheiros de software devem se concentrar na robustez do projeto e da arquitetura do software, mas, além disso, práticas seguras devem ser adotadas quando sistemas baseados em software são construídos [Sob10].

As atividades de garantia da segurança e identificação de riscos são planejadas, gerenciadas e controladas da mesma maneira que outras atividades da engenharia de software. A equipe de software coleta dados (por exemplo, o número de violações de acesso, de falhas de sistema e de registros de dados perdidos) para determinar o que funciona e o que não funciona. Isso exige que os desenvolvedores analisem cada falha relatada (para determinar se sua causa está relacionada a uma vulnerabilidade do sistema) e, então, avaliem a exposição de ativos resultante.

27.5 Análise de risco de segurança⁷

Modelagem de ameaças é um método de análise de segurança usado para identificar ameaças com o potencial mais alto de causar danos.

Quais são os passos exigidos para se construir um modelo de ameaça?

Identificar e gerenciar riscos à segurança são tarefas importantes do planejamento do projeto (Capítulo 31). A engenharia de segurança é guiada pelos riscos identificados pela equipe de software e por outros envolvidos. Os riscos impactam as atividades de gerenciamento e garantia da segurança do projeto.

Modelagem de ameaças é um método de análise de segurança usado para identificar ameaças com o potencial mais alto de causar danos a um sistema baseado em software. É realizada nas fases iniciais de um projeto, usando os requisitos e modelos de análise. A criação de um modelo de ameaça envolve identificar componentes importantes de um aplicativo, decompor o aplicativo, identificar e classificar as ameaças aos componentes, avaliar e classificar as ameaças a cada componente, avaliar os componentes com base em sua classificação de risco e desenvolver estratégias de mitigação de risco. A Microsoft utiliza os seguintes passos para criar um modelo de ameaça [Sin08]:

- 1 **Identificar ativos.** Listar toda informação sigilosa e propriedade intelectual: onde está armazenada, como está armazenada e quem tem acesso.
- 2 **Criar uma visão geral da arquitetura.** Escrever casos de uso do sistema e construir um modelo dos componentes do sistema.
- 3 **Decompor o aplicativo.** O objetivo é garantir que todos os dados enviados entre componentes do aplicativo sejam validados.
- 4 **Identificar ameaças.** Anotar todas as ameaças que podem comprometer ativos do sistema, usando métodos como árvores de ataque ou padrões de ataque; frequentemente, o processo envolve examinar ameaças à rede, à configuração do sistema host e a aplicativos.
- 5 **Documentar as ameaças.** Criar um formulário de informações de riscos, detalhando como cada ameaça deve ser monitorada e mitigada.
- 6 **Classificar as ameaças.** A maioria dos projetos não têm recursos suficientes para tratar de todas as ameaças concebíveis; portanto, elas precisam ser classificadas usando-se seu impacto e probabilidade.

Ativos valiosos devem ser protegidos dos riscos de alta probabilidade. Para classificar as ameaças, pode-se usar um processo quantitativo de avaliação de risco (Capítulo 35). Primeiramente, são identificados todos os bens a serem avaliados e é determinado o valor financeiro da perda ou recriação de um bem. Para cada ativo, é criada uma lista das principais ameaças, e dados históricos são usados para determinar a provável ocorrência de cada ameaça em um ano típico. É calculado o valor financeiro por ano da perda em potencial por ameaça importante de cada bem, junto com a *expectativa de perda anual* (ALE, *annual loss expectancy*) (determinada multiplicando-se a ocorrência pela perda em potencial). Por fim, a ameaça combinada da perda de um bem é calculada somando-se os valores de ALE associados a cada ameaça individual.

⁷ Uma discussão geral sobre análise de risco de projetos de software, abrangendo todos os tipos de riscos que ameaçam o projeto e seu sucesso, é apresentada no Capítulo 35.

CASASEGURA

**Etapas da segurança**

Cena: Área de trabalho do grupo de garantia da qualidade de software.

Atores: Jamie Lazar, membro da equipe de software; Vinod Raman, membro da equipe de software; e Bridget Thornton, líder do grupo de qualidade do software.

Conversa:

Vinod: Oi, Bridget. Doug quer que trabalhem na análise de riscos à segurança.

Bridget: É para ajudar a definir as prioridades da segurança para o desenvolvimento?

Jamie: Acho que sim.

Vinod: Podemos ver as preocupações com a segurança do banco de dados?

Bridget: Certamente. Sabemos quais são os custos para fazer backup e reparar os registros de dados utilizando dados históricos. Podemos não saber qual é a responsabilidade pelos danos que podem resultar se dados dos clientes forem roubados, mas temos dados do setor sobre esses custos.

Jamie: Isso é tudo de que precisamos?

Bridget: Bem, você já tem os diagramas arquiteturais do sistema. É mais fácil verificar se todos os dados trocados en-

tre os componentes foram validados. Também precisaremos determinar as ameaças a cada bem.

Vinod: Como fazemos isso?

Bridget: Podemos criar uma árvore de ataque. Começaríamos definindo uma meta de ataque na raiz. Por exemplo, o objetivo de um invasor poderia ser roubar informações dos clientes.

Vinod: E...

Bridget: Então, você examina seu catálogo de padrões de ataque ao banco de dados para ver qual se aplica e lista cada um deles como submetas na árvore.

Jamie: E daí?

Bridget: Você precisa refinar as ameaças e criar formulários de informações de risco para cada uma, descrevendo o impacto da ameaça e quaisquer passos de monitoramento ou mitigação que devem vigorar para tratar dela.

Vinod: Como isso ajuda a definir prioridades de desenvolvimento?

Bridget: Você determina o custo de cada ameaça calculando sua expectativa de perda anual (ALE), utilizando dados históricos. Podemos ajudá-lo nessa parte do processo.

Jamie: Obrigado, Bridget. Voltaremos para obter suas informações sobre o cálculo de ALE quando tivermos identificado e refinado as ameaças.

27.6 A função das atividades da engenharia de software convencional

Construir um sistema e torná-lo seguro *a posteriori* é altamente ineficiente e propenso a falhas. Apesar disso, alguns desenvolvedores de software argumentam que as ameaças a um sistema não podem ser previstas até que ele seja construído. Como consequência, eles ignoram problemas de segurança até a fase de teste e, então, "tapam buracos" para eliminar erros de segurança cometidos anteriormente no processo do software. Acrescentar emendas de segurança a um sistema já existente de maneira *ad hoc* talvez não seja possível sem grandes alterações em seu projeto ou em sua arquitetura. Portanto, a abordagem de tapar buracos é ineficiente e dispendiosa.

A natureza de um processo iterativo e incremental (Capítulo 4) torna difícil tratar de todas as preocupações com a segurança antes da realização de qualquer trabalho de desenvolvimento. Muitas vezes, os requisitos do software mudam durante o processo de desenvolvimento. Além disso, decisões sobre o projeto arquitetural podem ter impacto direto nas preocupações com a segurança. Por esses motivos, é difícil tratar de todos os problemas de segurança no início de um projeto. Mesmo quando a maioria das preocupações é tratada antecipadamente, decisões de projeto mais adiante no processo de software podem afetar vulnerabilidades da segurança no sistema final [Mei06].

Um processo de software eficiente inclui um conjunto razoável de oportunidades de revisão e ajuste. Muitas atividades relacionadas à segurança com-

"Em um mundo no qual o conhecimento humano total é duplicado a cada 10 anos, nossa segurança só depende de nossa capacidade de aprender."

Nathaniel Branden

plementam umas às outras e têm um efeito sinérgico na qualidade do software. Por exemplo, é sabido que as revisões de código reduzem o número de defeitos no produto antes dos testes, os quais, por sua vez, eliminam brechas de segurança em potencial e melhoram a qualidade do software.

Durante o *planejamento*, o orçamento e o calendário do projeto devem levar em conta a segurança para que os recursos necessários para satisfazer os objetivos da segurança⁸ do sistema sejam alocados adequadamente. Como parte da avaliação do risco da segurança e da privacidade, cada requisito funcional precisa ser examinado para ver se pode afetar um ativo associado a um objetivo da segurança do sistema. Durante a análise de riscos, o custo associado a cada perda deve ser determinado ou estimado.

A identificação de mecanismos para lidar com ameaças específicas ao sistema é frequentemente adiada até que os requisitos de um incremento do software sejam transformados em seus requisitos de *projeto*. É aí que a superfície de ataque deve ser identificada. A *superfície de ataque* é definida como o conjunto de vulnerabilidades acessíveis e exploráveis presentes em um produto de software. Muitas vulnerabilidades da segurança serão encontradas nas interseções das camadas do sistema. Por exemplo, informações inseridas por meio de um formulário na interface do usuário podem ser interceptadas ao viajar por uma rede até um servidor de banco de dados. Podem ser desenvolvidas diretrizes de projeto que incluam provisões de segurança tratando diretamente da superfície de ataque.

Pode ser interessante separar as análises de segurança das análises de projeto gerais. Revisões de código centradas em problemas de segurança devem constar como parte das atividades de *implementação*. Essas revisões devem ter como base os objetivos de segurança apropriados e as ameaças identificadas nas atividades de projeto do sistema.

O teste de segurança é uma parte rotineira do teste do sistema (Capítulo 22). A avaliação de riscos à segurança pode ser uma fonte de casos de teste que permitam aos testes de segurança ser mais focalizados. Um plano de resposta a incidentes (IRP, incident response plan) explicita as ações a serem executadas por cada envolvido no sistema, em resposta a ataques específicos [Pra07]. Uma análise completa do IRP também deve fazer parte do processo de verificação da segurança.

Além disso, a verificação deve incluir revisões separadas das operações de segurança e procedimentos de arquivamento de bens. O plano de gestão de riscos à segurança deve ser revisto periodicamente, como parte do processo de manutenção.

Quando são relatados incidentes com a segurança depois que um aplicativo foi distribuído, os desenvolvedores devem avaliar a eficácia dos procedimentos de gestão de riscos à segurança como parte da manutenção do sistema (Capítulo 36). Se os procedimentos de mudança do sistema (Capítulo 29) incluem analisar a causa raiz, isso pode ajudar a descobrir vulnerabilidades no projeto global do sistema.

Uma *superfície de ataque* é definida como o conjunto de vulnerabilidades acessíveis e exploráveis presentes em um produto de software.

O que é um plano de resposta a incidentes?

"Quando você sabe que é capaz de lidar com o que quer que seja, você tem a única segurança que o mundo tem a oferecer."

Harry Browne

⁸ Por exemplo, proteção de dados de clientes ou reconhecimento de requisitos de conformidade jurídica ou reguladora associados à confidencialidade, integridade ou disponibilidade das informações do sistema ou de outra propriedade intelectual.

27.7 Verificação de sistemas confiáveis

Quando considerada no contexto da segurança de software, *credibilidade* indica o nível de confiança com que uma entidade do sistema (ou organização) pode contar com outra. As *entidades do sistema* abrangem sistemas inteiros, subsistemas e componentes de software. A credibilidade tem uma dimensão psicológica e uma dimensão técnica. Em geral, diz-se que uma entidade confia em outra quando presume que a segunda se comportará exatamente como a primeira espera. Demonstrar que essa suposição é correta significa verificar a credibilidade de um sistema. Embora tenham sido propostos diversos modelos de credibilidade [Sin08], nosso enfoque será na garantia de que o sistema obedece às práticas de mitigação criadas dentro de seu modelo de ameaça.

A tarefa de verificação garante que os requisitos de sistemas confiáveis sejam avaliados pela utilização de métricas específicas e quantificáveis, baseadas em técnicas de teste, inspeção e análise [She10]. As métricas de teste podem incluir a proporção do número de falhas detectadas em relação ao número de falhas previstas ou a relação entre casos de teste de segurança bem-sucedidos e o número total dos realizados. Outras métricas poderiam incluir a eficiência da remoção de defeitos (Capítulo 32) das atividades de inspeção formais. Também é interessante garantir a rastreabilidade dos casos de teste de segurança em relação aos casos de uso de segurança desenvolvidos durante as atividades de análise.

A evidência usada para provar o caso de segurança deve ser aceitável e convincente para todos os colaboradores da entidade confiável. Os usuários de sistemas confiáveis devem estar convencidos de que o sistema não tem vulnerabilidades exploráveis ou lógica mal-intencionada. Como consequência da tarefa de verificação, os usuários devem ter certeza da confiabilidade do sistema e de sua capacidade de sobrevivência quando comprometido. Isso significa que um dano no software será minimizado e que o sistema pode se recuperar rapidamente, com uma capacidade de operação aceitável. Casos e procedimentos de teste de segurança específicos também representam uma parte importante do processo de verificação [Mea10].

Credibilidade indica o nível de confiança com que uma entidade do sistema (ou organização) pode contar com outra.



Criação de caso de teste de segurança

Cena: Sala de Vinod.

Atores: Vinod Raman, membro da equipe de software, e Ed Robbins, membro da equipe de software.

Conversa:

Vinod: Precisamos criar um caso de teste de segurança para acessar o vídeo do CasaSegura remotamente.

Ed: Devemos começar examinando o caso de uso de segurança que Doug e Bridget [líder do grupo de qualidade de software] desenvolveram.

Vinod: Acho que poderíamos deixar o pessoal do ITG fazer isso, mas esse parece ser um caso de teste muito simples.

Ele também deve ser acrescentado ao conjunto de casos de teste que utilizamos para teste de regressão.

Ed: Certo, o caso de uso de senha exige que o usuário faça login em um site usando uma conexão segura com uma identificação de usuário válida, dois níveis de senhas, além de precisar inserir um número de identificação pessoal de quatro dígitos, após solicitar o sinal de vídeo.

Vinod: Isso nos dá vários caminhos lógicos para testar. O usuário precisa inserir quatro dados. Cada entrada precisa ser testada com um valor válido, um valor incorreto, um valor nulo e um valor de dado formatado incorretamente.

Ed: Para cobrir todos os caminhos lógicos são necessários 256 casos de teste distintos.

Vinod: Sim, é isso mesmo. Também precisamos definir a resposta de cada um.

CASASEGURA

Ed: De acordo com a política de segurança, o usuário dispõe de três tentativas para cada informação.

Vinod: Certo, e o usuário deve digitar os dados após cada tentativa malsucedida.

Ed: E, se qualquer uma delas falhar na terceira tentativa, deve enviar, por e-mail, um alerta para a empresa e para o usuário.

Vinod: Provavelmente seria interessante tornar aleatória a ordem dos casos de teste apresentados para o examinador

de senhas. Poderíamos executar nossos casos de teste mais de uma vez para ter certeza de que o examinador de senhas não é afetado pelo histórico.

Ed: Devemos escrever um pequeno programa para examinar esses casos de teste e registrar os resultados.

Vinod: Isso! Vai ser bastante trabalho. Talvez devamos deixar o ITG trabalhar com a equipe SQA da Bridget para desenvolver os testes de segurança.

Atualmente, a medida da qualidade do software não trata adequadamente da garantia da credibilidade e da segurança. As medidas existentes (por exemplo, medidas de confiabilidade, como o tempo médio entre falhas, ou medidas de confiabilidade, como a densidade de defeitos) frequentemente ignoram numerosos fatores que podem comprometer o software e deixá-lo vulnerável a ataques. Em parte, isso se dá porque muitas dessas métricas não levam em conta o fato de que existem agentes ativos continuamente sondando as vulnerabilidades do software.

Métricas de segurança eficazes mantêm dados históricos baseados no comportamento passado de uma entidade em situações que envolvem credibilidade. Como exemplo, considere a credibilidade estabelecida quando um site de e-commerce permite avaliar seus vendedores e compradores. Evidentemente, esse tipo de sistema de avaliação deve garantir que as entidades que estão sendo avaliadas sejam identificadas corretamente e não tenham dados errados registrados sobre elas. Problemas como esses às vezes infestam sistemas de informe crédito.

O Departamento de Segurança Interna dos Estados Unidos defende a adoção de práticas de projeto de software seguras que empreguem ferramentas de medida confiáveis e padronizadas. Em condições ideais, essas ferramentas (quando existem) podem ajudar os desenvolvedores a reduzir o número de vulnerabilidades introduzidas em um sistema durante o desenvolvimento [Mea10]. Isso permitiria aos usuários de sistemas confiáveis tomar decisões fundamentadas sobre a credibilidade de um sistema. Mas, como a confiabilidade de sistemas, o usuário pode basear esse julgamento no grau das perdas experimentadas ao usar o sistema.

FERRAMENTAS DO SOFTWARE



Engenharia de segurança

Objetivo: Ferramentas de engenharia de segurança ajudam a identificar vulnerabilidades no código-fonte.

Mecanismos: Em geral, código-fonte de software é processado por permitir que a ferramenta o leia e sinalize constru-

ções de programação para os desenvolvedores examinarem atentamente.

Ferramentas representativas:⁹

RATS (*Rough Auditing Tool for Security*), desenvolvida pela Secure Software (<http://code.google.com/p/rough-auditing-tool-for-security/>), é uma ferramenta de

⁹ A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

varredura que fornece ao analista de segurança uma lista de possíveis pontos problemáticos a enfocar, junto com a descrição do problema e possivelmente sugerindo soluções.

ITS4, desenvolvida pela Cigital (<http://freecode.com/projects/its4/>), é uma ferramenta para procurar, estaticamente, vulnerabilidades de segurança críticas em código-fonte C e C++.

SLAM, desenvolvida pela Microsoft (<http://research.microsoft.com/en-us/projects/slam/>), é uma ferramenta

para verificar se o software satisfaz propriedades comportamentais fundamentais das interfaces que utiliza e para ajudar os engenheiros de software no projeto de interfaces e software que garanta um funcionamento confiável e correto.

Muitas outras ferramentas de varredura de código-fonte de segurança podem ser encontradas em <http://www.tech-faq.com/how-to-find-security-vulnerabilities-in-source-code.html>

27.8 Resumo

A engenharia de segurança de software se preocupa com o desenvolvimento de software que proteja contra ameaças os ativos que gerencia. As ameaças podem envolver ataques que exploram vulnerabilidades do sistema para comprometer a confidencialidade, integridade ou disponibilidade de seus serviços ou dados.

A gestão de riscos à segurança se preocupa com a avaliação do impacto de possíveis ameaças e com a produção de requisitos de segurança para minimizar perdas críticas. O projeto voltado à segurança envolve a criação de uma arquitetura de sistema que minimize a introdução de vulnerabilidades conhecidas. Os engenheiros de software devem utilizar técnicas para evitar, repelir e recuperar de ataques, como uma maneira de mitigar os efeitos de perdas.

Para inspirar credibilidade entre os envolvidos, é necessário que os desenvolvedores considerem a garantia da segurança como uma atividade abrangente, que esteja presente no início de todo processo de software. O desenvolvimento de métricas de segurança ainda está em sua infância. A construção de um caso de segurança para um sistema envolve coletar evidências usando testes de segurança, realizar análises técnicas formais voltadas à segurança e inspeção para garantir que as diretrizes de segurança e as práticas de mitigação estejam sendo seguidas.

Problemas e pontos a ponderar

27.1 Considere um aplicativo para celular que você possua. Primeiramente, descreva-o brevemente e, então, liste pelo menos de três a cinco riscos à segurança.

27.2 Descreva uma estratégia de migração de segurança para um dos riscos mencionados no Problema 27.1.

27.3 Identifique cinco padrões de ataque que podem ser comumente usados para atacar aplicativos web.

27.4 Descreva o modelo de credibilidade usado em um site de leilões como o eBay.

27.5 Descreva os requisitos de segurança para um repositório de fotos baseado na nuvem.

27.6 O que a mesma política de origem tem a ver com sistemas confiáveis?

27.7 Use a Internet para determinar o custo médio anual da incidência única de roubo de identidade para um cliente individual.

27.8 Explique alguns dos problemas que podem ser encontrados se você tentar tratar do risco à segurança depois de concluído um sistema.

27.9 Use a Internet para encontrar os detalhes necessários para criar um padrão de ataque por phishing.

27.10 Calcule a expectativa de perda anual (ALE) para um servidor de dados cujo valor de substituição é de US\$30.000, cuja ocorrência de perda de dados devida à invasão seja de 5% anualmente e cujo potencial de perda seja de US\$20.000.

Leituras e fontes de informação complementares

Livros de Vacca (*Computer and Information Security Handbook*, 2ª ed., Morgan Kaufman, 2013), Goodrich e Tamassia (*Introduction to Computer Security*, Addison-Wesley, 2010), Anderson (*Security Engineering*, 2ª ed., 2008), Kern et al. (*Foundations of Security*, Apress, 2007), McGraw (*Software Security*, Addison-Wesley, 2006) e Dowd e seus colegas (*The Art of Software Assessment: Identifying and Preventing Software Vulnerabilities*, Addison-Wesley, 2006) tratam de importantes problemas de segurança. Zalewski (*The Tangled Web: A Guide to Securing Modern Web Applications*, No Starch Press, 2011) e Howard e LeBlanc (*Writing Secure Code*, 2ª ed., Microsoft Press, 2003) discutem a construção de sistemas seguros. A visão de um gerente de projeto é apresentada por Allen et al. (*Software Security Engineering*, Addison-Wesley, 2008). Howard e Lipner (*The Security Development Lifecycle*, Microsoft Press, 2006) discutem o processo da engenharia de segurança. Brown et al. (*Security Patterns*, Wiley, 2006) ajudam a compreender melhor o uso de padrões como um elemento da engenharia de segurança eficaz.

Outros se concentram no aspecto da “invasão”. Livros de Barnett e Grossan (*Web Application Defender's Cookbook: Battling Hackers and Protecting Users*, Wiley, 2012), Ottenheimer e Wallace (*Securing the Virtual Environment: How to Defend Against Enterprise Attack*, Wiley, 2012), Studdard e Pinto (*The Web Application Hacker's Handbook*, Wiley, 2011), Howard e seus colegas (*24 Deadly Sins of Software Security*, McGraw-Hill, 2009), Erickson (*Hacking: The Art of Exploration*, No Starch Press, 2008), Andrews e Whittaker (*How to Break Web Software*, Addison-Wesley, 2006) e Whittaker (*How to Break Software Security*, Addison-Wesley, 2003) dão ideias úteis para engenheiros de software que precisam saber como são os ataques a sistemas e aplicativos.

Sikorski e Honig (*Practical Malware Analysis*, No Starch Press, 2012) e Ligh et al. (*Malware Analyst's Cookbook*, Wiley, 2010) ajudam a compreender o funcionamento interno do malware. Swiderski (*Threat Modeling*, Microsoft Press, 2004) apresenta uma discussão detalhada sobre a modelagem de ameaças.

Diretrizes para quem precisa realizar testes de segurança são fornecidas em livros de Allen (*Advanced Penetration Testing for Highly Secured Environments*, Packt Publishing, 2012), O’Gorman (*Metasploit: The Penetration Tester's Guide*, No Starch Press, 2011), Faircloth (*Penetration Tester's Open Source Tool Kit*, Syngress, 2011), Engebretson (*The Basics of Hacking and Penetration Testing*, Syngress, 2011), Faircloth (*Penetration Tester's Open Source Tool Kit*, Syngress, 2011), Hope e Walther (*Web Security Testing Cookbook*, O’Reilly Media, 2008) e Wysopal et al. (*The Art of Software Security Testing*, Addison-Wesley, 2006).

Uma ampla gama de fontes de informação sobre engenharia de software se encontra à disposição na Internet. Uma lista atualizada de referências relevantes (em inglês) para o processo de projeto baseado em padrões pode ser encontrada no site: www.mhhe.com/pressman.

Modelagem formal e verificação

28

Ao contrário das revisões e dos testes, que começam logo que os modelos e os códigos de software são desenvolvidos, a modelagem formal e a verificação incorporam métodos de modelagem especializados que são integrados a abordagens de verificação formalmente prescritas. Sem a abordagem adequada de modelagem, a verificação não pode ser feita.

Neste capítulo e no Apêndice 3, discutimos dois métodos de modelagem formal e de verificação: *engenharia de software sala limpa* e *métodos formais*. Ambos exigem uma abordagem de verificação especializada e a cada um se aplica um método único de verificação. Ambos são muito rigorosos e nenhum é usado amplamente pela comunidade de engenharia de software, mas para criar um software “à prova de balas”, eles podem ser muito úteis. Vale a pena conhecê-los.

Conceitos-chave

certificação	612
especificação <i>box structure</i>	604
especificação funcional ..	604
métodos formais	615
modelo de processo sala limpa	603
projeto sala limpa	607
refinamento do projeto ..	608
teste de uso estatístico ..	610
verificação de correção ..	603
verificação de projeto ..	608

PANORAMA

O que é? Quantas vezes você já ouviu alguém dizer “Faça direito já na primeira vez”? Se aplicás-

semos isso ao software, seriam investidos bem menos esforços com retrabalho desnecessário. Dois métodos avançados de engenharia de software – engenharia de software sala limpa e métodos formais – ajudam a equipe de software a “fazer direito já na primeira vez”, proporcionando uma abordagem matemática baseada na modelagem de programas e na capacidade de verificar se o modelo está correto. A engenharia de software sala limpa enfatiza a verificação matemática da correção antes de começar a construção do programa e a certificação da confiabilidade do software como parte da atividade de teste. Os métodos formais usam a teoria dos conjuntos e a notação lógica para criar uma definição clara dos fatos (requisitos) que podem ser analisados para melhorar (ou até mesmo provar) a correção e a consistência. O ponto principal de ambos os métodos é a criação de software com taxas de falhas extremamente baixas.

Quem realiza? Um engenheiro de software especialmente treinado.

Por que é importante? Erros geram retrabalho. O retrabalho consome tempo e aumenta os custos. Não seria maravilhoso se pudéssemos reduzir significativamente a quantidade

de erros (defeitos, bugs) introduzidos à medida que o software é projetado e construído? Essa é a premissa da modelagem formal e verificação.

Quais são as etapas envolvidas? Modelos de requisitos e de projeto são criados usando notação especializada que é favorável à verificação matemática. A engenharia de software sala limpa utiliza a representação *box structure* (estrutura de caixas) que encapsula o sistema (ou algum aspecto do sistema) em um nível de abstração específico. Uma vez concluído o projeto de representação *box structure*, é aplicada a verificação de correção. Depois de verificada a correção para cada representação *box structure*, inicia-se o teste de uso estatístico. Métodos formais transformam os requisitos do software em uma representação mais formal, aplicando a notação e heurística de conjuntos para definir a invariante de dados, estados e operações para uma função do sistema.

Qual é o artefato? É desenvolvido um modelo específico e formal de requisitos. São registrados os resultados das provas de correção e dos testes estatísticos de uso.

Como garantir que o trabalho foi realizado corretamente? É aplicada uma prova formal de correção ao modelo de requisitos. Os testes estatísticos de uso experimentam cenários de uso para assegurar que os erros na funcionalidade de usuário sejam descobertos e corrigidos.

Engenharia de software sala limpa é uma abordagem que enfatiza a necessidade de agregar precisão ao software enquanto ele está sendo desenvolvido. No lugar do ciclo clássico de análise, codificação, teste e depuração, a abordagem sala limpa sugere um ponto de vista diferente [Lin94]:

A filosofia por trás da engenharia de software sala limpa é evitar a dependência de processos onerosos de remoção de defeitos, escrevendo incrementos de código corretos já na primeira vez e verificando sua precisão antes do teste. Seu modelo de processo incorpora a certificação estatística de qualidade dos incrementos de código, à medida que vão se acumulando no sistema.

De muitas formas, sala limpa eleva a engenharia de software a outro nível, enfatizando a necessidade de *provar* a precisão.

Modelos desenvolvidos com *métodos formais* são descritos usando sintaxe e semântica formais que especificam a função e o comportamento do sistema. A especificação é na forma matemática (por exemplo, o cálculo de predicado pode ser usado como base para uma linguagem de especificação formal). Em sua introdução aos métodos formais, Anthony Hall [Hal90] faz um comentário que se aplica igualmente aos métodos sala limpa:

Métodos formais e a engenharia de software sala limpa são controversos. Seus defensores dizem que podem revolucionar o desenvolvimento de software. Seus críticos acham que são impossíveis e difíceis. Enquanto isso, para a maioria das pessoas, métodos formais e engenharia de software sala limpa são tão estranhos, que é difícil julgar os argumentos a favor ou contra.

Apresentamos aqui um panorama dos conceitos da modelagem formal e verificação. No Apêndice 3, exploramos alguns dos detalhes técnicos da modelagem formal e verificação.

28.1 Abordagem sala limpa

A engenharia de software sala limpa usa uma versão especializada do modelo incremental de software introduzido no Capítulo 4. Uma “sequência de incrementos de software” [Lin94b] é desenvolvida por pequenas equipes de software independentes. À medida que cada incremento é certificado, ele é integrado no todo, por isso o sistema cresce com o tempo.

A sequência das tarefas sala limpa para cada incremento é ilustrada na Figura 28.1. De acordo com a sequência para incrementos sala limpa, ocorrem as seguintes tarefas:

Qual é o modelo de processo usado para projetos sala limpa?

Planejamento de incremento. Um plano de projeto que adota a estratégia incremental é desenvolvido. São criados a funcionalidade de cada incremento, seu tamanho projetado e um cronograma de desenvolvimento sala limpa. Deve-se ter cuidado especial para garantir que os incrementos certificados sejam integrados no seu devido tempo.

Coleta dos requisitos. Usando técnicas semelhantes às introduzidas no Capítulo 8, é desenvolvida uma descrição mais detalhada dos requisitos no nível do cliente (para cada incremento).

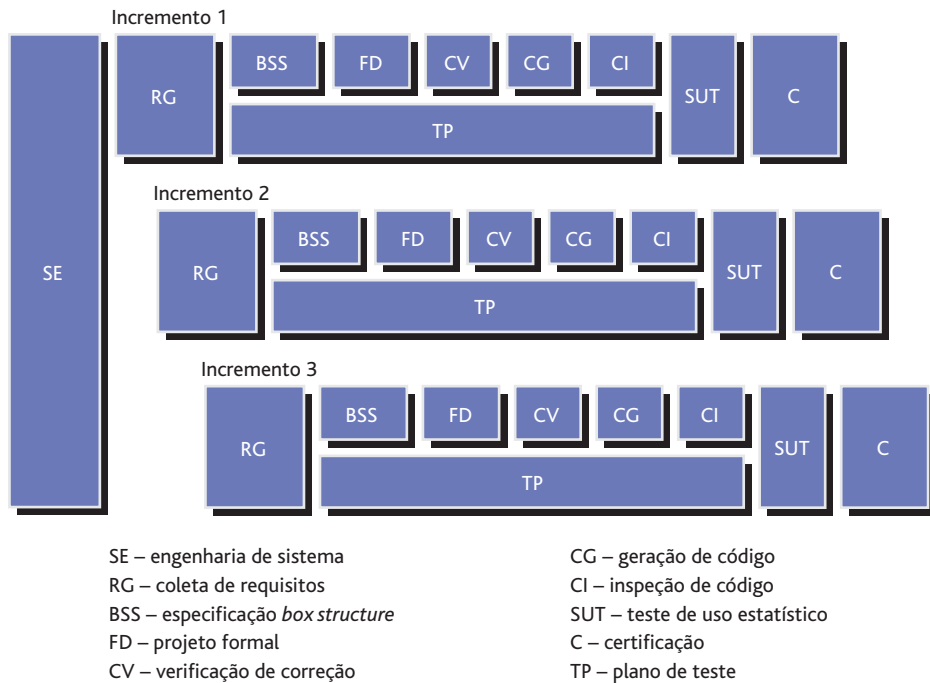


FIGURA 28.1 O modelo de processo sala limpa.

Especificação *box structure*. Um método de especificação que emprega as *box structures* (estruturas de caixas) é usado para descrever a especificação funcional. As *box structures* “isolam e separam a definição criativa de comportamento, dados e procedimentos em cada nível de refinamento” [Hev93].

Projeto formal. Usando a abordagem *box structure*, o projeto sala limpa torna-se uma extensão natural e contínua da especificação. Embora seja possível fazer uma distinção clara entre as duas atividades, as especificações (chamadas de *caixas-pretas*) são refinadas iterativamente (dentro de um incremento) para se tornarem análogas aos projetos arquiteturais e no nível de componentes (chamados de *caixas de estado* e *caixas-claras*, respectivamente).

Verificação de correção. A equipe sala limpa conduz uma série de atividades de verificação rigorosa da correção no projeto e depois no código. A verificação (Seção 28.3.2) começa com uma *box structure* de nível mais alto (especificação) e move-se em direção ao detalhe de projeto e código. O primeiro nível de verificação de correção ocorre aplicando-se uma série de “perguntas de correção” [Lin88]. Se essas perguntas não demonstrarem que a especificação está correta, utilizam-se métodos de verificação mais formais (matemáticos).

Geração de código, inspeção e verificação. As especificações *box structure*, representadas em uma linguagem especializada, são traduzidas na linguagem de programação adequada. São usadas revisões técnicas (Capítulo 20) para garantir a conformidade da semântica do código e das *box*

“A única maneira de erros ocorrerem em um programa é serem colocados lá pelo autor. Não há outro mecanismo conhecido... A prática correta procura apresentar a inserção de erros e, se isso falhar, deve-se removê-los antes de testar ou de utilizar qualquer outro programa.”

Harlan Mills

structure e a correção sintática do código. Então é executada a verificação de correção para o código-fonte.

Planejamento do teste estatístico. O uso projetado do software é analisado, e uma série de casos de teste que testam uma “distribuição de probabilidades” de uso (Seção 28.4) é planejada e projetada. De acordo com a Figura 28.1, essa atividade sala limpa é executada em paralelo à especificação, verificação e geração de código.

Teste de uso estatístico. Relembrando que testar exaustivamente o software é impossível (Capítulo 23), é sempre necessário projetar um número finito de casos de teste. As técnicas estatísticas de uso [Poo88] executam uma série de testes derivados de uma amostragem estatística (a distribuição de probabilidades que mencionamos antes) de todas as execuções de programa possíveis por todos os usuários de uma população-alvo (Seção 28.4).

Certificação. Uma vez concluídos a verificação, a inspeção e os testes de uso (e todos os erros corrigidos), o incremento é certificado como pronto para a integração.

As quatro primeiras atividades do processo sala limpa preparam o cenário para as atividades de verificação formal que vêm em seguida. Por essa razão, começamos a discussão da abordagem sala limpa com as atividades de modelagem essenciais para a verificação formal a ser aplicada.

28.2 Especificação funcional

Uma “caixa” encapsula o sistema em algum nível de abstração e é usada na especificação funcional.

A abordagem de modelagem na engenharia de software sala limpa usa um método chamado *especificação box structure*. Uma “caixa” encapsula o sistema (ou algum aspecto do sistema) em algum nível de detalhe. Por meio de um processo de elaboração ou refinamento passo a passo, as caixas são refinadas em uma hierarquia em que cada caixa tem transparência referencial. Ou seja, “o conteúdo de informações da especificação de cada caixa é suficiente para definir seu refinamento, sem depender da implementação de qualquer outra caixa” [Lin94b]. Isso habilita o analista a particionar um sistema hierarquicamente, mudando da representação essencial no topo para o detalhe específico de implementação na base. São usados três tipos de caixas:

Caixa-preta. A caixa-preta especifica o comportamento de um sistema ou de uma parte dele. O sistema (ou parte) responde a estímulos específicos (eventos) aplicando um conjunto de regras de transição que mapeiam os estímulos em uma resposta.

Caixa de estado. A caixa de estado encapsula dados de estado e serviços (operações) de maneira análoga a objetos. Nessa visão de especificação, são representadas entradas para a caixa de estado (estímulos) e saídas (respostas). A caixa de estado representa o “histórico de estímulo” da caixa-preta; isto é, os dados encapsulados na caixa de estado que devem ser retidos entre as transições inferidas.

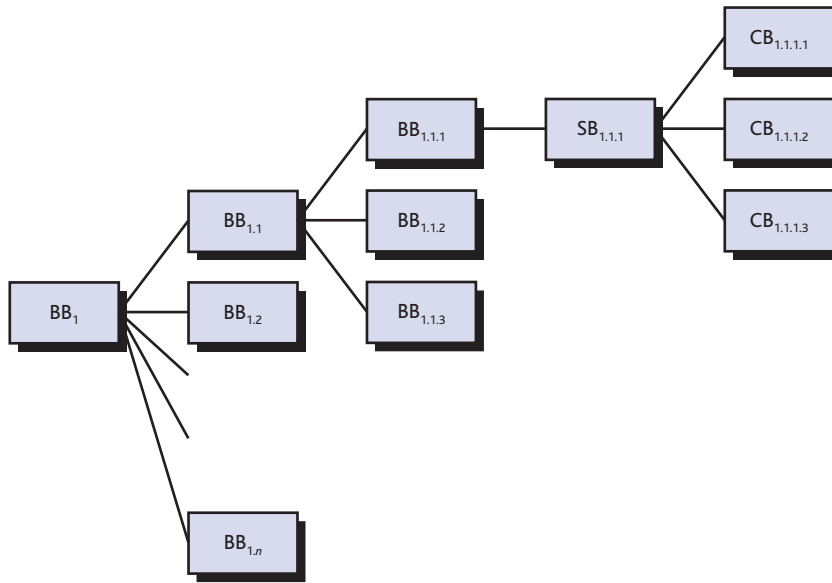


FIGURA 28.2 Refinamento da *box structure*.

Caixa-clara. As funções de transição inferidas pela caixa de estado são definidas na caixa-clara. Em outras palavras, uma caixa-clara contém o projeto procedimental para a caixa de estado.

A Figura 28.2 mostra a abordagem de refinamento usando a especificação *box structure*. Uma caixa-preta, *black box*, (BB_1) define as respostas para um conjunto completo de estímulos. BB_1 pode ser refinada em um conjunto de caixas-pretas, $BB_{1,1}$ até $BB_{1,n}$, cada uma das quais lida com uma classe de comportamento. O refinamento continua até que seja identificada uma classe coerente de comportamento (por exemplo, $BB_{1,1}$). Então, é definida uma caixa de estado, *state box*, ($SB_{1,1,1}$) para a caixa-preta ($BB_{1,1}$). Nesse caso, $SB_{1,1,1}$ contém todos os dados e serviços necessários para implementar o comportamento definido por $BB_{1,1,1}$. Por último, $SB_{1,1,1}$ é refinada em caixas-claras, *clear boxes*, ($CB_{1,1,1,n}$) e os detalhes do projeto procedimental são especificados.

À medida que ocorre cada um desses passos de refinamento, ocorre também a verificação de correção. As especificações de caixa de estado são verificadas para garantir que cada uma esteja conforme o comportamento definido pela especificação da caixa-preta pai. De maneira semelhante, as especificações de caixas-claras são verificadas em relação à caixa de estado pai.

“Toda solução para qualquer problema é simples. É na distância entre os dois que reside o mistério.”

Derek Landy

28.2.1 Especificação caixa-preta

A especificação *caixa-preta* descreve uma abstração, estímulo e resposta usando a notação mostrada na Figura 28.3 [Mil88]. A função f é aplicada à sequência S^* de entradas (estímulos) S e transforma-as em uma saída (resposta) R . Para componentes simples de software, f pode ser uma função matemática, mas, em geral, é descrita por meio de linguagem natural (ou linguagem de especificação formal).

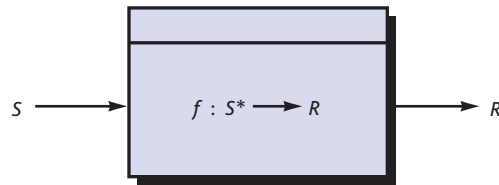


FIGURA 28.3 Uma especificação caixa-preta.

Muitos dos conceitos introduzidos para sistemas orientados a objetos também são aplicáveis à caixa-preta. As abstrações de dados e as operações que manipulam as abstrações são encapsuladas pela caixa-preta. Como em uma hierarquia de classes, a especificação caixa-preta pode exibir hierarquias de uso nas quais as caixas de baixo nível herdam as propriedades das caixas de mais alto nível na estrutura.

28.2.2 Especificação caixa de estado

A *caixa de estado* é uma generalização de uma máquina de estado [Mil88]. Lembrando a discussão da modelagem comportamental e diagramas de estado no Capítulo 11, um estado é algum modo observável do comportamento de um sistema. À medida que ocorre o processamento, um sistema responde a eventos (estímulos) fazendo uma transição do estado atual para algum novo estado. Enquanto é feita a transição, pode ocorrer uma ação. A caixa de estado utiliza a abstração de dados para determinar a transição para o próximo estado e a ação (resposta) que ocorrerá em consequência da transição.

Observando a Figura 28.4, a caixa de estado incorpora uma caixa-preta g . O estímulo S , colocado na caixa-preta, chega de alguma fonte externa e de um conjunto de estados internos de sistema T . Mills [Mil88] fornece uma descrição matemática da função f da caixa-preta contida na caixa de estado:

$$g : S^* \times T^* \rightarrow R \times T$$

onde g é uma subfunção vinculada a um estado específico t . Quando considerados coletivamente, os pares de subfunção de estado (t, g) definem a função caixa-preta f .

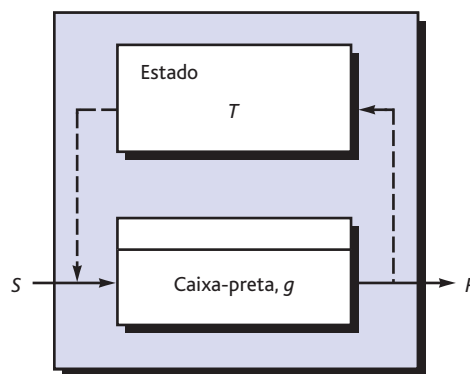


FIGURA 28.4 Uma especificação caixa de estado.

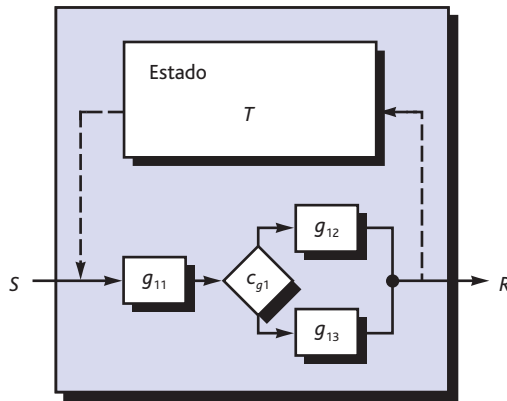


FIGURA 28.5 Uma especificação caixa-clara.

28.2.3 Especificação caixa-clara

A especificação *caixa-clara* está relacionada ao projeto procedimental e à programação estruturada. Basicamente, a subfunção g dentro da caixa de estado é substituída pelas construções de programação estruturada que implementam g .

Como exemplo, considere a caixa-clara da Figura 28.5. A caixa-preta g , da Figura 28.3, é substituída por uma sequência de construções que incorpora uma condicional. Estas, por sua vez, podem ser refinadas em caixas-claras em um nível mais baixo, à medida que o refinamento passo a passo prossegue.

É importante notar que a especificação procedimental descrita na hierarquia caixa-clara pode ter sua correção provada. Esse tópico é considerado na Seção 28.3.

28.3 Projeto sala limpa

A engenharia de software sala limpa usa intensamente a filosofia de programação estruturada (Capítulo 14). Neste caso, porém, a programação estruturada é aplicada com muito mais rigor.

Funções básicas de processamento (descritas durante os primeiros refinamentos da especificação) são refinadas usando uma “expansão passo a passo de funções matemáticas em estruturas de conectivos lógicos [por exemplo, if-then-else/se-então-senão] e subfunções, em que a expansão [é] executada até que todas as subfunções identificadas possam ser diretamente declaradas na linguagem de programação empregada para implementação” [Dye92].

A abordagem de programação estruturada pode ser usada de maneira eficaz para refinar uma função, mas e quanto ao projeto de dados? Aqui entram em cena muitos conceitos fundamentais de projeto (Capítulo 12). Dados de programação são encapsulados como um conjunto de abstrações atendidas por subfunções. Utilizam-se os conceitos de encapsulamento de dados, ocultamento de informações e tipos de dados para criar o projeto de dados.

Caso tenha usado construções de programação estruturada, um conjunto de perguntas simples permite provar que seu código está correto.

28.3.1 Refinamento do projeto

Cada especificação caixa-clara representa o projeto de um procedimento (subfunção) necessário para obter uma transição caixa de estado. Na caixa-clara, construções de programação estruturada e refinamento passo a passo são usados para representar o detalhe procedimental. Por exemplo, uma função de programa f é refinada em uma sequência de subfunções g e h . Estas, por sua vez, são refinadas em construções condicionais (por exemplo, if-then-else e do-while). O refinamento continua até que haja detalhe procedimental suficiente para criar o componente em questão.

Em cada nível de refinamento, a equipe sala limpa¹ executa uma *verificação formal de correção*. Para tanto, uma série de condições genéricas de correção é anexada às construções de programação estruturada. Se uma função f é expandida em uma sequência g e h , a condição de correção para toda a entrada em f é

- g seguida por h produz f ?

Quando uma função p é refinada em uma condicional da forma if $\langle c \rangle$ then q , else r , a condição de correção para toda entrada em p é

- Sempre que a condição $\langle c \rangle$ for verdadeira, q produz p ; e, sempre que $\langle c \rangle$ for falsa, r produz p ?

Quando a função m é refinada como um laço, as condições de correção para toda a entrada a m são

- O término está garantido?
- Sempre que $\langle c \rangle$ for verdadeira, n seguido de m produz m ; sempre que $\langle c \rangle$ for falsa, a saída do laço ainda produz m ?

A cada vez que uma caixa-clara for refinada para o próximo nível de detalhe, essas condições de correção são aplicadas.

28.3.2 Verificação de projeto

O uso de construções de programação estruturada limita o número de testes de correção que devem ser feitos. Apenas uma condição é verificada para as sequências; duas condições são testadas para if-then-else e três condições são verificadas para laços.

A fim de ilustrar a verificação de correção para um projeto procedimental, usamos um exemplo simples introduzido por Linger, Mills e Witt [Lin79]. O objetivo é projetar e verificar um pequeno programa que encontra a parte inteira y da raiz quadrada de um número inteiro x . O projeto procedimental é representado pelo fluxograma da Figura 28.6.²

Para verificar a correção desse projeto, são acrescentadas condições de entrada e saída, conforme mostra a Figura 28.6. A condição de entrada men-

¹ Como a equipe inteira está envolvida no processo de verificação, é pouco provável que seja produzido um erro na condição da própria verificação.

² A Figura 28.6 foi adaptada de [Lin94]. Usada com permissão.

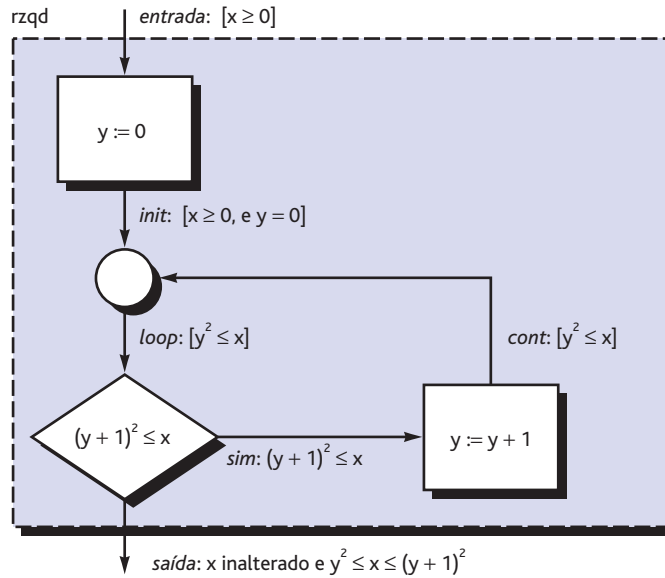


FIGURA 28.6 Cálculo da parte inteira de uma raiz quadrada.

Fonte: [Lin79].

ciona que x deve ser maior ou igual a 0. A condição de saída exige que x permaneça inalterado e que y satisfaça a expressão da figura. Para provar que o projeto está correto, é necessário provar que as condições *init*, *loop*, *cont*, *sim* e *saída* mostradas na Figura 28.6 são verdadeiras em todos os casos. Às vezes chama-se isso de *subprovas*.

1. A condição *init* exige que $|x| \geq 0$ e $y = 0$. Com base nos requisitos do problema, assume-se que a condição de entrada é correta.³ Portanto, a primeira parte da condição *init*, $x \geq 0$, é satisfeita. Observando-se o fluxograma, a instrução que precede imediatamente a condição *init* produz $y = 0$. Portanto, a segunda parte da condição *init* também é satisfeita. Então, *init* é verdadeira.
2. A condição *loop* pode ser encontrada de duas formas: (1) diretamente de *init* (neste caso, a condição *loop* é satisfeita diretamente) ou via fluxo de controle que passa pela condição *cont*. Como a condição *cont* é idêntica à condição *loop*, *loop* é verdadeira independentemente do caminho que leva até ela.
3. A condição *cont* é encontrada apenas depois que o valor de y é incrementado por 1. Além disso, o caminho do fluxo de controle que leva a *cont* pode ser invocado somente se a condição *sim* também for verdadeira. Assim, se $(y + 1)^2 \leq x$, segue-se que $y^2 \leq x$. A condição *cont* é satisfeita.
4. A condição *sim* é testada na lógica condicional mostrada. Logo, a condição *sim* deve ser verdadeira quando o fluxo de controle se move ao longo do caminho mostrado.

Para provar que um projeto está correto, você precisa primeiro identificar todas as condições e então provar que cada uma delas assume um valor booleano apropriado. Elas são chamadas de *subprovas*.

³ Um valor negativo para a raiz quadrada não tem significado nesse contexto.

5. A condição *saída* exige primeiramente que x permaneça inalterado. Um exame do projeto indica que x não aparece em nenhum lugar à esquerda de um operador de atribuição. Não há chamadas de função que usem x . Portanto, ele fica inalterado. Como o teste condicional $(y + 1)^2 \leq x$ deve falhar para se chegar à condição *saída*, segue-se que $(y + 1)^2 \leq x$. Além disso, a condição *loop* ainda deve ser verdadeira (isto é, $y^2 \leq x$). Portanto, $(y + 1)^2 > x$ e $y^2 \leq x$ podem ser combinados para satisfazer a condição de saída.

Você também deve garantir que o laço termine. Um exame da condição *loop* indica que, como y é incrementado e $x \geq 0$, o laço deve terminar em algum momento.

Os cinco passos que acabamos de descrever são uma prova da correção do projeto do algoritmo mostrado na Figura 28.6. Você tem certeza agora de que o projeto vai, de fato, calcular a parte inteira de uma raiz quadrada.

É possível usar uma abordagem matemática mais rigorosa de verificação de projeto. No entanto, uma discussão sobre esse assunto está fora dos objetivos deste livro. Se tiver interesse, consulte [Lin79].

28.4 Teste sala limpa

"A qualidade não é um ato, é um hábito."

Aristóteles

A estratégia e as táticas de teste sala limpa são fundamentalmente diferentes das abordagens de teste convencionais (Capítulos 22 até 26). Métodos convencionais derivam uma série de casos de teste para descobrir erros de projeto e codificação. O objetivo do teste sala limpa é validar requisitos de software demonstrando que uma amostragem estatística de casos de uso (Capítulo 8) foi executada de forma bem-sucedida.

28.4.1 Teste de uso estatístico

Mesmo que você decida não usar a abordagem sala limpa, vale a pena considerar o teste de uso estatístico como parte de sua estratégia de teste.

O usuário de um programa de computador raramente precisa entender os detalhes técnicos do projeto. O comportamento do programa visível ao usuário é controlado por entradas e eventos que muitas vezes são produzidos pelo usuário. Mas, em sistemas complexos, o espectro possível de entradas e eventos (os casos de uso) pode ser extremamente amplo. Qual subconjunto de casos de uso verificará adequadamente o comportamento do programa? Essa é a primeira questão tratada no teste de uso estatístico.

O teste de uso estatístico "resume-se no teste do software da maneira que os usuários pretendem usá-lo" [Lin94]. Para tanto, as equipes de teste sala limpa (também chamadas de *equipes de certificação*) devem determinar a distribuição de probabilidade de uso para o software. A especificação (caixa-preta) para cada incremento do software é analisada para determinar uma série de estímulos (entradas ou eventos) que fazem o software mudar seu comportamento. Com base em entrevistas com usuários em potencial, na criação de cenários de uso e em uma compreensão geral do domínio de aplicação, é atribuída a cada estímulo uma probabilidade de uso.

São gerados casos de testes para cada conjunto de estímulos⁴, de acordo com a distribuição de probabilidade de uso. Para ilustrar, considere o sistema *CasaSegura* já discutido neste livro. A engenharia de software sala limpa está sendo usada para desenvolver um incremento de software que controle a interação do usuário com o teclado do sistema de segurança. Para esse incremento, cinco estímulos foram identificados. A análise indica a distribuição percentual de probabilidade de cada estímulo. Para facilitar a seleção de casos de teste, essas probabilidades são mapeadas em intervalos numerados de 1 a 99 [Lin94] e ilustradas na tabela a seguir:

Ao contrário dos testes convencionais, as abordagens sala limpa são baseadas em estatísticas.

Estímulo ao programa	Probabilidade (%)	Intervalo
Armar/Desarmar (AD)	50	1–49
Definir zona (ZS)	15	50–63
Consultar (Q)	15	64–78
Testar (T)	15	79–94
Alarme de pânico	5	95–99

Para gerar uma sequência de casos de testes de uso que esteja de acordo com a distribuição de probabilidade de uso, geram-se números aleatórios entre 1 e 99. Cada número aleatório corresponde a um intervalo na distribuição de probabilidades mencionada. Portanto, a sequência de casos de testes de uso é definida aleatoriamente, mas corresponde a uma probabilidade apropriada de ocorrência do estímulo. Por exemplo, suponha que sejam geradas as seguintes sequências de números aleatórios:

13-94-22-24-45-56

81-19-31-69-45-9

38-21-52-84-86-4

Selecionando-se os estímulos apropriados com base no intervalo de distribuição mostrado na tabela, são gerados os seguintes casos de uso:

AD-T-AD-AD-AD-ZS

T-AD-AD-AD-Q-AD-AD

AD-AD-ZS-T-T-AD

A equipe de teste executa esses casos de uso e verifica o comportamento do software em relação à especificação do sistema. São registrados os tempos para os testes de maneira que os intervalos de tempo possam ser determinados. Usando os intervalos de tempo, a equipe de certificação pode calcular o tempo médio para falhas (MTTF, mean-time-to-failure). Se for executada uma longa sequência de testes sem falha, o MTTF é baixo e a confiabilidade do software pode ser considerada alta.

⁴ Podem ser usadas ferramentas automáticas para conseguir isso. Para mais informações, consulte [Dye92].

28.4.2 Certificação

As técnicas de verificação e teste discutidas anteriormente neste capítulo levam a componentes de software (e incrementos completos) que podem ser certificados. No contexto da abordagem de engenharia de software sala limpa, a *certificação* implica que a confiabilidade (medida pelo MTTF) pode ser especificada para cada componente.

O impacto provável de componentes de software certificáveis vai muito além de um simples projeto sala limpa. Componentes de software reutilizáveis podem ser armazenados com seus cenários de uso, estímulos de programa e distribuições de probabilidade. Cada componente teria uma confiabilidade certificada sob o cenário de uso e regime de teste descrito. Essas informações são valiosas para aqueles que quiserem usar os componentes.

Como certificamos um componente de software?

A abordagem de certificação envolve cinco passos [Woh94]: (1) devem ser criados cenários de uso, (2) é especificado um perfil de uso, (3) são gerados casos de teste com base no perfil, (4) são executados os testes e os dados de falhas são registrados e analisados e (5) a confiabilidade é calculada e certificada. Os passos 1 a 4 foram discutidos em uma seção anterior. A certificação para engenharia de software sala limpa exige a criação de três modelos [Poo93]:

Modelo de amostragem. O teste do software executa m casos de teste aleatórios e é certificado se não ocorrerem falhas ou se ocorrer um número especificado de falhas. O valor de m é derivado matematicamente para garantir que a confiabilidade desejada seja obtida.

Modelo de componente. Um sistema formado por n componentes deve ser certificado. O modelo de componente permite que o analista determine a probabilidade de o componente i falhar antes do término.

Modelo de certificação. A confiabilidade global do sistema é projetada e certificada.

Ao final do teste de uso estatístico, a equipe de certificação tem as informações necessárias para entregar um software com um MTTF certificado e calculado com cada um desses modelos. Se estiver interessado em detalhes adicionais, consulte [Cur86], [Mus87] ou [Poo93].

28.5 Reconsideração dos métodos formais

A maioria dos engenheiros de software concorda que é difícil, se não impossível, criar sistemas de software sem falhas, seguindo um paradigma de modelagem, projeto, codificação e teste. Existem alguns sistemas que não podem ser testados adequadamente antes de serem implantados (por exemplo, um robô operando em um planeta distante sob condições ambientais hostis).

Os métodos formais oferecem um modo de verificar se um sistema importante vai funcionar de acordo com sua especificação. Para isso, o software é tratado como uma entidade matemática cuja exatidão pode ser provada com operações lógicas. Constata-se que é mais fácil fazer isso para programas im-

plementados em linguagens de programação imperativas⁵ do que em aplicações baseadas em eventos e implementadas em linguagens orientadas a objetos.

São diversas as vantagens em potencial [Abr09] com o uso de métodos formais na tentativa de desenvolver sistemas sem falhas. Os requisitos redigidos em linguagens naturais muitas vezes são ambíguos ou incompletos. Os métodos formais modelam um sistema como uma série de transições de estado para representar o que os desenvolvedores do sistema vão observar à medida que o programa executar. O ato de modelar o sistema pode revelar vários defeitos nele presentes. A modelagem de uma aplicação de software grande exige várias iterações.

O *refinamento horizontal* elabora estados do software do abstrato para o concreto por meio da adição de detalhamentos. Esse refinamento horizontal é a base para permitir o rastreamento de requisitos do software. As afirmações utilizadas para impedir que o software atinja um estado inválido podem ser definidas; e sua localização, verificada. Uma vez concluído o refinamento horizontal desse modelo separado, o *refinamento vertical* é usado para transformar os estados e as transições de modo que possam ser implementados na linguagem de programação pretendida. Refinamento vertical é o processo que tenta “colar” o abstrato no concreto, sem permitir que falhas de comunicação em uma linguagem de destino mal escolhida afetem as especificações dos requisitos.

A integração de código legado complica o uso de métodos formais, pois os requisitos legados podem não se ajustar bem ao novo sistema. Em muitos casos, é melhor capturar o comportamento do código legado na especificação e, então, implementar o comportamento no novo código.

Alguns oponentes dos métodos formais dizem que muitas de suas práticas são contrárias aos princípios dos modelos de processo ágeis. Contudo, Black e seus colegas [Bla09] sugerem que os elementos dos métodos formais e dos processos ágeis podem ser combinados para se criar produtos de software melhores. Ambos têm o mesmo objetivo básico de tentar criar software confiável. Os métodos formais podem valorizar o desenvolvimento ágil, obrigando os desenvolvedores a assegurar que os axiomas de propriedade de proteção de sistemas⁶ sejam válidos.

Podem ser usadas técnicas dos métodos formais (por exemplo, ferramentas de análise estática e de prova de teorema) para gerar casos de teste a partir de modelos do sistema automaticamente e indicar onde devem ser colocadas as afirmações no código de programa em evolução. Requisitos informais podem ser traduzidos para uma notação formal incorporada ao código-fonte. As inconsistências das afirmativas redigidas em notação formal podem ser verificadas automaticamente pelo computador. Muitas vezes, o código precisa ser refatorado⁷ à medida que evolui. Os métodos formais podem fornecer a base para a definição de transformações que preservam a exatidão para garantir que o código refatorado ainda satisfaça os requisitos.

“Os métodos formais nunca terão um impacto significativo até que possam ser usados pelas pessoas que não os compreendem.”

Tom Melham

⁵ Uma *linguagem de programação imperativa* obtém seu principal efeito atribuindo os valores de expressões algébricas.

⁶ Axiomas de propriedade de proteção são declarações sobre o que o software pode não permitir. A proteção foi discutida como parte da segurança no Capítulo 27.

⁷ A *refatoração* (Capítulo 5) melhora o código sem alterar seu significado.

Embora o uso de métodos formais possa retardar a entrega do primeiro incremento do software, isso pode reduzir o volume de reformulações feitas durante o tempo de vida do projeto e, portanto, oferecer um retorno significativo sobre o investimento de tempo aplicado. Se os requisitos do cliente são extremamente voláteis, não há garantia de que um projeto possa ser concluído mais rapidamente usando-se técnicas ágeis. O propósito é criar um produto que atenda aos requisitos do cliente e complementar uma abordagem ágil com métodos formais que ajudem a garantir que esses requisitos foram atendidos.

Meyer e seus colegas [Mey09] descrevem três tipos de ferramentas que podem ser úteis na automação dos testes, facilitando, com isso, o uso estatístico das abordagens de teste. Ferramentas de *geração de teste* criam e executam casos de teste sem entrada humana. Ferramentas de *extração de teste* produzem casos de teste para posterior reprodução a partir de falhas de execução do programa. Ferramentas de *integração de teste manual* ajudam a desenvolver e gerenciar casos de teste produzidos manualmente.

A parte que provavelmente mais vai ser automatizada é a execução de casos de teste, a qual é muito importante para testes de regressão eficientes. Meyer [Mey09] sugere que a geração de casos de teste seja relativamente simples para linguagens como a Eiffel, que suporta projeto por contrato.⁸ Os casos de teste são gerados para pôr em prática as precondições e pós-condições representadas como parte do código-fonte (discutidas na Seção 28.6). Quando os desenvolvedores encontram erros nos programas, normalmente os registram e os corrigem, perdendo a oportunidade de criar casos de teste para futuros testes de regressão. Quando ocorre um erro durante a execução do programa, as ferramentas de teste capturam as informações sobre violações de precondição ou falhas de pós-condição e as convertem automaticamente em casos de teste.

FERRAMENTAS DO SOFTWARE



Métodos formais

Objetivo: o objetivo das ferramentas de métodos formais é ajudar uma equipe de software na especificação e verificação da correção.

Mecanismos: o processo utilizado pelas ferramentas varia. Em geral, as ferramentas ajudam na especificação de uma prova automática da correção, usualmente definindo uma linguagem especializada para prova de teorema. Muitas ferramentas não são comercializadas e foram desenvolvidas para fins de pesquisa.

Ferramentas representativas:⁹

ACL2, desenvolvida na Universidade do Texas (www.cs.utexas.edu/users/moore/acl2/), é “tanto uma linguagem de programação na qual você pode modelar sistemas de computadores quanto uma ferramenta para ajudá-lo a provar propriedades daqueles modelos”.

Indicações de vários repositórios de ferramentas de métodos formais podem ser encontradas em um site hospedado pelo Cyber Security and Information Systems Information Analysis Center (CSIIAC) <https://sw.thecsiac.com/databases/url/key/53/57#.UHRvKYbwpuh/>.

⁸ O *projeto por contrato* utiliza precondições, pós-condições e invariantes para responder às perguntas: O que o programa espera? O que ele garante? O que ele mantém?

⁹ A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

28.6 Conceitos de métodos formais

The Encyclopedia of Software Engineering [Mar01] define métodos formais da seguinte maneira:

Métodos formais usados no desenvolvimento de sistemas de computadores são técnicas de base matemática para descrever propriedades de sistemas. Esses métodos formais proporcionam uma estrutura dentro da qual as pessoas podem especificar, desenvolver e verificar sistemas de maneira sistemática, em vez de casual.

As propriedades desejadas de uma especificação formal são os objetivos de todos os métodos de especificação. No entanto, a linguagem de especificação com base matemática, usada para métodos formais, resulta em uma possibilidade muito maior de obter essas propriedades. A sintaxe formal de uma linguagem de especificação (Apêndice 3) possibilita que os requisitos ou o projeto sejam interpretados de uma única maneira, eliminando a ambiguidade que frequentemente ocorre quando uma linguagem natural (por exemplo, inglês) ou uma notação gráfica (por exemplo, UML) precisa ser interpretada por um leitor. Os recursos descritivos da teoria dos conjuntos e a notação lógica possibilitam uma definição clara dos requisitos. Para serem consistentes, os requisitos especificados em um ponto de uma especificação não podem ser contrariados em outro ponto. A consistência é obtida¹⁰ provando-se matematicamente que os fatos iniciais podem ser mapeados de maneira formal (usando regras de inferência) em declarações posteriores dentro da especificação.

Para introduzirmos os conceitos básicos de métodos formais, vamos considerar alguns exemplos simples para ilustrar o uso da especificação matemática, sem se aprofundar em detalhes matemáticos.

Exemplo 1: Uma tabela de símbolos. Um programa é usado para manter uma tabela de símbolos. Emprega-se esse tipo de tabela frequentemente em diferentes tipos de aplicações. Ela consiste em uma coleção de itens sem qualquer duplicação. Um exemplo de uma tabela de símbolos típica está na Figura 28.7. Ela representa a tabela usada por um sistema operacional para manter os nomes dos usuários do sistema. Outros exemplos de tabelas incluem a coleção de nomes dos funcionários em um sistema de folha de pagamento, a coleção de nomes dos computadores em um sistema de comunicações em rede e a coleção de locais de destino em um sistema que gera as tabelas de horários de um sistema de transportes.

Suponha que a tabela apresentada neste exemplo seja formada apenas por nomes *MaxIds*. Essa declaração, que impõe uma restrição na tabela, é um componente de uma condição conhecida como *invariante de dados*. Invarian-

Uma especificação formal deve ter consistência, completude e nenhuma ambiguidade.

¹⁰ Na realidade, a totalidade é difícil de garantir, mesmo quando se utilizam métodos formais. Alguns aspectos de um sistema podem ser deixados indefinidos quando as especificações estão sendo criadas; outras características podem ser omitidas propositalmente para permitir que os projetistas tenham certa liberdade na escolha da abordagem de implementação e, por último, é impossível considerar todos os cenários operacionais em um sistema grande e complexo. Coisas podem ser omitidas por engano.

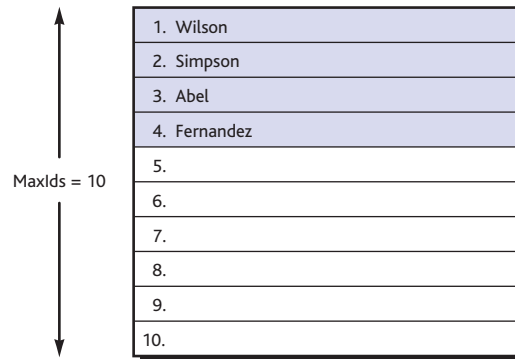


FIGURA 28.7 Uma tabela de símbolos.

te de dados é uma condição que é verdadeira ao longo de toda a execução do sistema que contém uma coleção de dados. A invariante de dados que se aplica à tabela de símbolos que acabamos de discutir tem dois componentes: (1) que a tabela terá apenas *MaxIds* e nada mais além disso e (2) que não haverá nomes duplicados na tabela. No caso do programa da tabela de símbolos, isso significa que não importa quando a tabela de símbolos será examinada durante a execução do sistema, ela sempre terá apenas *MaxIds* e nada mais, e não conterá duplicatas.

Outro conceito importante é o de *estado*. Muitas linguagens formais usam a noção de estado como discutido no Capítulo 11; isto é, um sistema pode estar em um dentre vários estados, cada um representando um modo de comportamento observável externamente. No entanto, uma definição diferente para o termo *estado* é usada em algumas linguagens de especificação. Nessas linguagens, o estado de um sistema é representado pelos dados nele armazenados. Usando a última definição no exemplo do programa da tabela de símbolos, o estado é a tabela de símbolos.

O conceito final é o de *operação*. Essa é uma ação que tem lugar em um sistema e lê ou escreve dados. Se o programa da tabela de símbolos está cuidando de adicionar ou remover nomes da tabela de símbolos, ele estará associado a duas operações: a operação *acrescentar()* um nome especificado à tabela de símbolos e a operação *remover()* da tabela um nome existente.¹¹ Se o programa proporciona a facilidade de verificar se um nome específico está ou não contido na tabela, haverá uma operação que retornará alguma indicação sobre se o nome está ou não nela.

Três tipos de condições podem ser associados às operações: invariantes, precondições e pós-condições. *Invariante* define o que com certeza não mudará. Por exemplo, a tabela de símbolos tem uma invariante dizendo que o número de elementos é sempre menor ou igual a *MaxIds*. Uma *precondição* define as circunstâncias nas quais uma operação em particular é válida. Por exemplo, a precondição para uma operação que acrescenta um nome a uma tabela de símbolos identificadores de funcionários é válida somente se o nome

¹¹ Deve-se observar que a adição de um nome não pode ocorrer no estado *full* (cheio) e que a exclusão de um nome é impossível no estado *empty* (vazio).

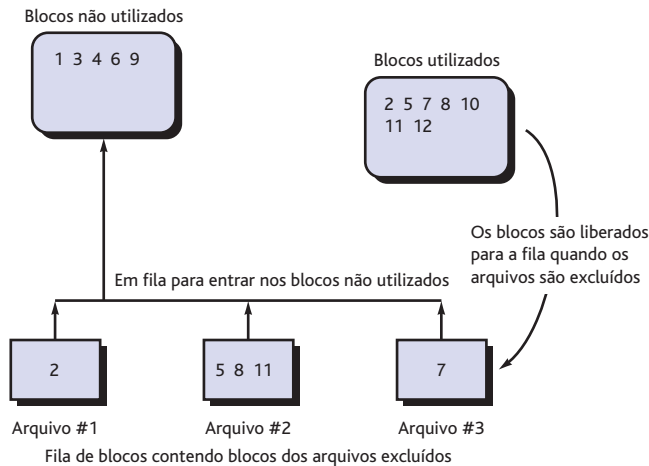


FIGURA 28.8 Um tratador de blocos.

a ser adicionado não está contido na tabela e também se houver menos do que *MaxIds* identificadores de funcionários na tabela. A *pós-condição* de uma operação define o que se garante ser verdadeiro após completar uma operação. Isso é definido por seu efeito sobre os dados. Para a operação *acrescentar()*, a pós-condição especificaria matematicamente que a tabela foi aumentada com o novo identificador.

Exemplo 2: Um tratador de blocos. Uma das partes mais importantes de um sistema operacional simples é o subsistema que mantém os arquivos criados pelos usuários. Parte do subsistema de arquivos é o *tratador de blocos*. Os arquivos no sistema de armazenamento de arquivos são formados por blocos de armazenamento mantidos em um dispositivo de armazenamento. Durante a operação do computador, arquivos serão criados e excluídos, e isso exige a aquisição e liberação de blocos de armazenamento. Para enfrentar a tarefa, o subsistema de arquivamento deve manter um reservatório de blocos não utilizados (livres) e manter controle dos blocos que estão em uso no momento. Quando os blocos são liberados porque um arquivo foi excluído, eles normalmente são acrescentados a uma fila de blocos à espera de serem acrescentados ao reservatório de blocos não utilizados. Isso está apresentado na Figura 28.8. Nessa figura, há vários componentes: o reservatório de blocos não utilizados, os blocos que no momento formam os arquivos administrados pelo sistema operacional e os blocos que estão esperando para serem acrescentados ao reservatório. Os blocos em espera são mantidos em fila, em que cada elemento da fila contém um conjunto de blocos de um arquivo excluído.

Para esse subsistema, o estado é a coleção de blocos livres, a coleção de blocos utilizados e a fila de blocos devolvidos. A invariante de dados, expressa em linguagem natural, é

- Nenhum bloco será marcado como não usado e usado.
- Todos os conjuntos de blocos mantidos na fila serão subconjuntos da coleção de blocos usados correntemente.

- Nenhum elemento da fila terá o mesmo número de bloco.
- A coleção de blocos usados e blocos não usados será a coleção total de blocos que compõem os arquivos.
- A coleção de blocos não usados não terá números de bloco duplicados.
- A coleção de blocos usados não terá números de bloco duplicados.

Algumas das operações associadas a esses dados são: *acrescentar()* uma coleção de blocos ao fim da fila, *remover()* uma coleção de blocos usados da frente da fila e colocá-los em uma coleção de blocos não usados e *verificar()* se uma fila de blocos está vazia.

A pré-condição de *acrescentar()* é que os blocos a ser adicionados devem estar na coleção de blocos usados. A pós-condição é que a coleção de blocos agora se encontra no fim da fila. A pré-condição de *remover()* é que a fila deve conter pelo menos um item. A pós-condição é que os blocos devem ser adicionados à coleção de blocos não usados. A operação *verificar()* não tem pré-condição. Isso significa que a operação é sempre definida, independentemente do valor do estado. A pós-condição fornece o valor *true* (verdadeiro) se a fila estiver vazia e *false* (falso) caso contrário.

Nos exemplos apresentados nesta seção, introduzimos conceitos fundamentais de especificação formal, mas sem ênfase à matemática necessária para tornar a especificação formal. No Apêndice 3, consideraremos como a notação matemática pode ser usada para especificar formalmente algum elemento de um sistema.

28.7 Argumentos alternativos

Parnas [Par10] critica de várias formas a abordagem normalmente usada para provar a correção de um programa. Ele diz que o uso de valores de variáveis de expressão para definir o estado do cálculo ignora o papel dos manipuladores de interrupção em sistema de tempo real. Os métodos formais normalmente desestimulam o uso de efeitos colaterais de expressões, que podem ser comuns em alguns domínios de aplicação. A ocultação de informações facilitada pelo uso de tipos de dados abstratos dá margem ao uso de estados ocultos, os quais devem ser levados em conta ao se escrever invariantes de dados. Frequentemente, isso torna as afirmações mais complexas do que o código em si. Os métodos formais foram inventados para programas deterministas.¹² Isso não leva em conta os programas projetados para executar indefinidamente.

Os métodos formais dependem do uso de modelos destinados a fornecer uma visão simplificada do programa. Não há nenhum método confiável para transformar esses modelos em código de programa. É possível provar que um programa correto pode ter defeitos, porque o modelo ou o código era falho.

No trabalho com métodos formais, o processo de provar a correção do software se transforma na tarefa de mostrar que um programa pode ser re-

¹² Os *programas deterministas* sempre começam em um estado inicial definido e terminam em um estado final único.

duzido a uma sequência de transições de estados¹³ ou de dados válida. Isso pode não ser suficiente para todos os programas. Talvez sejam necessários predicados que possam verificar o comportamento correto de um programa em tempo de execução para verificar a exatidão de algoritmos paralelos ou de tempo real.

As afirmações ajudam os desenvolvedores a documentar e a entender o software de maneira mais efetiva, mas são práticas apenas para programas pequenos. O uso de afirmações não elimina a necessidade de documentação externa, que frequentemente é imprecisa devido às ambiguidades da linguagem natural utilizada para descrever o sistema. Além disso, os métodos formais não ajudam os engenheiros de software a fazer uma escolha dentre vários projetos de software corretos.

28.8 Resumo

A engenharia de software sala limpa é uma abordagem formal para o desenvolvimento de software que pode conduzir a uma qualidade notavelmente alta. Ela usa a especificação *box structure* para análise e modelagem de projeto e dá ênfase à verificação da correção, em vez do teste, como principal mecanismo para localizar e remover erros. O teste de uso estatístico é aplicado para desenvolver as informações necessárias de taxa de falhas para certificar a confiabilidade do software fornecido.

A abordagem sala limpa começa com a análise e modelos de projeto que usam uma representação *box structure*. Uma “caixa” encapsula o sistema (ou algum aspecto do sistema) em um nível específico de abstração. Caixas-pretas são usadas para representar o comportamento de um sistema observável externamente. Caixas de estado encapsulam dados de estado e operações. Caixas-claras são usadas para modelar o projeto procedimental inferido pelos dados e operações de uma caixa de estado.

Uma vez concluído o projeto *box structure*, é aplicada a verificação de correção. O projeto procedimental para um componente de software é particionado em uma série de subfunções. Para provar a correção das subfunções, definem-se condições de saída para cada subfunção e aplica-se uma série de subprovas. Se cada condição de saída é satisfeita, o projeto deve estar correto.

Uma vez concluída a verificação da correção, inicia-se o teste de uso estatístico. Diferentemente do teste convencional, a engenharia de software sala limpa não enfatiza o teste de unidade ou de integração. Em vez disso, o software é testado definindo-se um conjunto de cenários de uso, determinando-se a probabilidade de uso para cada cenário e, por fim, definindo-se testes aleatórios que se sujeitam às probabilidades. Os registros de erros resultantes são combinados com os modelos de amostragem, componente e certificação para possibilitar o cálculo matemático da confiabilidade projetada para o componente de software.

¹³ Um estado válido é um conjunto de variáveis do programa que receberam valores permitidos pelo processamento das entradas do programa.

Os métodos formais usam recursos descritivos da teoria dos conjuntos e notação lógica para possibilitar ao engenheiro de software criar uma definição clara dos fatos (requisitos). Os conceitos subjacentes que governam os métodos formais são: (1) a invariante de dados, uma condição verdadeira por toda a execução do sistema que contém uma coleção de dados; (2) o estado, uma representação do modo de comportamento do sistema observável externamente ou (em Z e linguagens relacionadas) os dados armazenados que um sistema acessa e altera; e (3) a operação, uma ação que tem lugar em um sistema e lê ou escreve dados para um estado. A operação está associada a duas condições: uma pré-condição e uma pós-condição.

Será que a engenharia de software *sala limpa* ou os métodos formais serão algum dia usados amplamente? A resposta é “provavelmente não”. Eles são mais difíceis de aprender do que os métodos de engenharia de software convencionais e representam um “choque cultural” significativo para alguns profissionais. Mas na próxima vez em que você ouvir alguém se lamentando “Por que não podemos escrever esse software certo já na primeira vez?”, saberá que há técnicas que podem ajudá-lo a conseguir exatamente isso.

Problemas e pontos a ponderar

28.1 Se você tivesse de escolher um aspecto da engenharia de software *sala limpa* que a torna radicalmente diferente das abordagens de engenharia de software convencionais orientadas a objeto, qual seria?

28.2 Como um modelo de processo incremental e certificação funcionam em conjunto para produzir software de alta qualidade?

28.3 Por meio da especificação *box structure*, desenvolva modelos de análise e projeto “primeira-passada” para o sistema *CasaSegura*.

28.4 Um algoritmo de ordenação da bolha (bubble sort) é definido da seguinte maneira:

```

procedure bubblesort;
var i, t, integer;
begin
  repeat until t=a[1]
    t:=a[1];
    for j:= 2 to n do
      if a[j-1] > a[j] then begin
        t:=a[j-1];
        a[j-1]:=a[j];
        a[j]:=t;
      end
    endrep
  end

```

Particione o projeto em subfunções e defina um conjunto de condições que possibilitariam provar que o algoritmo está correto.

28.5 Documente uma prova de verificação de correção para o bubble sort discutido no Problema 28.4.

28.6 Escolha um programa que você use regularmente (por exemplo, um programa de e-mail, um processador de texto, um programa de planilha). Crie um conjunto de ce-

nários de uso para o programa. Defina a probabilidade do uso de cada cenário e depois desenvolva uma tabela de distribuição de probabilidades e estímulos de programa similar à apresentada na Seção 28.4.1.

28.7 Para a tabela de distribuição de estímulos e probabilidades de programa desenvolvida no Problema 28.6, use um gerador de números aleatórios para desenvolver um conjunto de casos de teste para um teste de uso estatístico.

28.8 Descreva o objetivo da certificação no contexto de engenharia de software sala limpa.

28.9 Você foi designado para uma equipe que está desenvolvendo software para um fax modem. Sua tarefa é desenvolver a parte da “lista telefônica” da aplicação. A função lista telefônica permite armazenar até *MaxNomes* de pessoas associadas ao nome da empresa, o número do fax e outras informações relacionadas. Usando linguagem natural, defina

- a. a invariante de dados.
- b. o estado.
- c. as operações possíveis.

28.10 Você foi designado para uma equipe de software que está desenvolvendo um programa chamado MemoryDoubler, que fornece ao PC uma memória aparentemente maior do que a memória física. Isso é conseguido pela identificação, coleta e nova atribuição de blocos de memória que foram atribuídos a uma aplicação existente, mas que não estão sendo usados. Os blocos não usados são novamente atribuídos a aplicações que exigem memória adicional. Adotando as hipóteses apropriadas e usando linguagem natural, defina

- a. a invariante de dados.
- b. o estado.
- c. as operações possíveis.

Leituras e fontes de informação complementares

Nos anos recentes foram publicados relativamente poucos livros sobre técnicas avançadas de especificação e verificação. No entanto, vale considerar algumas novas adições à literatura. Livros de Gabbar (*Modern Formal Methods and Applications*, Springer, 2010) e Boca e seus colegas (*Formal Methods: State of the Art and New Directions*, Springer, 2010) apresentam os fundamentos, novos desenvolvimentos e aplicações avançadas. Jackson (*Software Abstractions*, 2ª ed., MIT Press, 2012) traz todos os fundamentos básicos e uma abordagem que ele chama de “métodos formais leves”. Monin e Hinchey (*Understanding Formal Methods*, Springer, 2003) proporcionam uma excelente introdução ao assunto. Butler e outros editores (*Integrated Formal Methods*, Springer, 2002) apresentam uma variedade de trabalhos sobre tópicos de métodos formais.

Além dos livros citados neste capítulo, Prowell e seus colegas (*Cleanroom Software Engineering: Technology and Process*, Addison-Wesley, 1999) fornecem um tratamento profundo de todos os aspectos importantes da abordagem sala limpa. Discussões interessantes sobre tópicos sala limpa foram editadas por Liu (*Formal Engineering for Industrial Software Development: Using the SOFL Method*, Springer, 2010) e Poore e Trammell (*Cleanroom Software Engineering: A Reader*, Blackwell Publishing, 1996). Becker e Whittaker (*Cleanroom Software Engineering Practices*, Idea Group Publishing, 1997) apresentam uma excelente visão geral para aqueles que não estão familiarizados com as práticas sala limpa.

O *Cleanroom Pamphlet* (Software Technology Support Center, Hill AF Base, April 1995) contém reimpressões de vários artigos importantes. O Cyber Security and Infor-

mation Systems Information Analysis Center (CSIAC) (www.thecsiac.com) fornece muitos artigos, guias e outras fontes de informações úteis sobre engenharia de software sala limpa.

A verificação de projeto por meio da prova de correção está no centro da abordagem sala limpa. Livros de Cupillari (*The Nuts and Bolts of Proofs*, 4ª ed., Academic Press, 2012), Solow (*How to Read and Do Proofs*, 5ª ed., Wiley, 2009), Eccles (*An Introduction to Mathematical Reasoning*, Cambridge University Press, 1998) fornecem excelentes introduções aos fundamentos básicos da matemática. Stavely (*Toward Zero-Defect Software*, Addison-Wesley, 1998), Baber (*Error-Free Software*, Wiley, 1991) e Schulmeyer (*Zero Defect Software*, McGraw-Hill, 1990) discutem a prova de correção em detalhe considerável.

Na área dos métodos formais, livros de Hinchey e Bowan (*Industrial Strength Formal Methods*, Springer-Verlag, 1999) e Hussmann (*Formal Foundations for Software Engineering Methods*, Springer-Verlag, 1997). O Apêndice 3 desta obra contém mais discussões sobre os fundamentos matemáticos dos métodos formais e o papel das linguagens de especificação na engenharia de software.

Uma ampla variedade de fontes de informações sobre engenharia de software sala limpa e métodos formais está disponível na Internet. Uma lista atualizada das referências da Web pode ser encontrada em “recursos de engenharia de software” no site da SEPA: www.mhhe.com/pressman.

Gestão de configuração de software

29

Mudanças são inevitáveis quando o software de computador é construído e podem causar confusão quando os membros de uma equipe de software estão trabalhando em um projeto. A confusão surge quando as mudanças não são analisadas antes de serem feitas, não são registradas antes de serem implementadas, não são relatadas àqueles que precisam saber ou não são controladas de maneira que melhorem a qualidade e reduzam os erros. Babich [Bab86] discute isso quando afirma:

A arte de coordenar o desenvolvimento de software para minimizar a... confusão é chamada de gestão de configuração. A gestão de configuração é a arte de identificar, organizar e controlar modificações no software que está em construção por uma equipe de programação. O objetivo é maximizar a produtividade minimizando os erros.

A gestão de configuração de software (SCM, software configuration management) é uma atividade de apoio, aplicada a toda a gestão da qualidade. Como as mudanças podem ocorrer a qualquer instante, as atividades de SCM são desenvolvidas para (1) identificar a alteração, (2) controlar a alteração, (3)

Conceitos-chave

auditoria de	
configuração	639
controle de alterações ..	635
controle de versão	634
gestão de configuração,	
elementos da.	626
gestão de conteúdo	643
gestão de impacto	638
identificação	633
itens de configuração	
de software	628
objetos de	
configuração	642
processo SCM	632
referenciais	626
relatório de status.	639
repositório	630
WebApps e	
aplicativos móveis	640

PANORAMA

O que é? Durante a criação de software, mudanças acontecem. E, por isso, precisamos gerenciá-

las eficazmente. A gestão de configuração de software (SCM), também chamada de gestão de alterações, é um conjunto de atividades destinadas a gerenciar as alterações, identificando os artefatos que precisam ser alterados, estabelecendo relações entre eles, definindo mecanismos para gerenciar diferentes versões desses artefatos, controlando as alterações impostas e auditando e relatando as alterações feitas.

Quem realiza? Todas as pessoas envolvidas na gestão da qualidade estão envolvidas com a gestão de alterações até certo ponto, mas muitas vezes são criadas funções de suporte especializadas para gerenciar o processo SCM.

Por que é importante? Se você não controlar as alterações, elas controlarão você – e isso nunca é bom. É muito fácil uma sequência de alterações não controladas transformar um bom software em um caos. Como consequência, a qualidade do software é prejudicada, e a entrega atrasa. Por essa razão, a gestão de alterações é parte essencial da gestão da qualidade.

Quais são as etapas envolvidas? Como muitos artefatos são produzidos quando o software é criado, cada um deles deve ser identificado de forma única. Feito isso, podem ser estabelecidos mecanismos para controle de versão e alteração. Para assegurar que a qualidade seja mantida quando são feitas alterações, o processo é auditado; e, para assegurar que aqueles que precisam saber sobre as alterações sejam informados, são gerados relatórios.

Qual é o artefato? Um plano de gestão de configuração de software define a estratégia de projeto para a gestão das alterações. Além disso, quando é invocada a SCM formal, o processo de controle de alterações produz requisições de alteração de software, relatórios e ordens de alteração de engenharia.

Como garantir que o trabalho foi realizado corretamente? Quando cada artefato pode ser levado em conta, rastreado e controlado, todas as alterações podem ser rastreadas e analisadas e todos aqueles que precisam saber sobre as alterações já foram informados, você fez tudo certo.

assegurar que a alteração esteja sendo implementada corretamente e (4) relatar as alterações a outros envolvidos.

É importante fazer uma distinção clara entre suporte de software e gestão de configuração de software. Suporte é um conjunto de atividades de engenharia que ocorrem depois que o software é fornecido ao cliente e posto em operação. Gestão de configuração é um conjunto de atividades de rastreamento e controle iniciadas quando um projeto de engenharia de software começa e terminadas apenas quando o software sai de operação.

Um dos principais objetivos da engenharia de software é incrementar a facilidade com que as alterações podem ser acomodadas e reduzir o esforço necessário quando alterações tiverem de ser feitas. Neste capítulo, discutiremos as atividades específicas que permitem gerenciar a mudança.

29.1 Gestão de configuração de software

O processo de software resulta em informações que podem ser divididas em três categorias principais: (1) programas de computador (tanto na forma de código-fonte quanto na forma executável), (2) produtos que descrevem os programas de computador (focado em vários envolvidos) e (3) dados ou conteúdo (contidos nos programas ou externos a ele). Os itens que compõem todas as informações produzidas como parte do processo de software são chamados coletivamente de *configuração de software*.

À medida que o trabalho de engenharia de software avança, cria-se uma hierarquia de *itens de configuração de software* (SCIs, *software configuration items*) – um elemento de informação com nome, que pode ser tão pequeno quanto um simples diagrama UML ou tão grande quanto um documento de projeto completo. Se cada SCI simplesmente conduzir a outros SCIs, resultará em pouca confusão. Infelizmente, outra variável entra no processo – *alteração*. A alteração pode ocorrer a qualquer momento e por qualquer razão. De fato, a *Primeira Lei da Engenharia de Sistemas* [Ber80] diz: “Não importa onde você esteja no ciclo de vida do sistema, o sistema mudará, e o desejo de alterá-lo persistirá por todo o ciclo de vida”.

Qual é a origem dessas alterações? A resposta a essa pergunta é variada, assim como as próprias alterações. No entanto, há quatro fontes fundamentais de alterações:

- Novos negócios ou condições de mercado ditam mudanças nos requisitos do produto ou nas regras comerciais.
- Novas necessidades dos envolvidos demandam modificação dos dados produzidos pelos sistemas de informação, na funcionalidade fornecida pelos produtos ou nos serviços fornecidos por um sistema baseado em computador.
- Reorganização ou crescimento/enxugamento causam alterações em prioridades de projeto ou na estrutura da equipe de engenharia de software.
- Restrições orçamentárias ou de cronograma causam a redefinição do sistema ou produto.

“Não há nada permanente, exceto as mudanças.”

Heráclito, 500 a.C.

Qual é a origem das alterações solicitadas para o software?

A gestão de configuração de software é um conjunto de atividades desenvolvidas para gerenciar alterações ao longo de todo o ciclo de vida de um software. A SCM pode ser vista como uma atividade de garantia de qualidade do software aplicada em todo o processo do software. Nas seções a seguir, descrevemos as principais tarefas da SCM e conceitos importantes que podem nos ajudar a gerenciar as alterações.

29.1.1 Um cenário SCM¹

Um cenário operacional de gestão de configuração (CM) típico inclui um gerente de projeto encarregado de um grupo de software, um gerente de configuração encarregado dos procedimentos e políticas CM, os engenheiros de software responsáveis pelo desenvolvimento e manutenção do artefato e o cliente que usa o produto. No cenário, suponha que o produto seja um item pequeno, envolvendo aproximadamente 15 mil linhas de código, que está sendo desenvolvido por uma equipe de quatro pessoas. (Note que são possíveis outros cenários com equipes menores ou maiores, mas, essencialmente, há problemas genéricos que cada um desses projetos enfrenta em relação à CM.)

No nível operacional, o cenário envolve vários papéis e tarefas. Para o gerente de projeto, o objetivo é garantir que o produto seja desenvolvido em certo prazo. O gerente monitora o progresso do desenvolvimento e reconhece e reage aos problemas. Isso é feito gerando e analisando relatórios sobre o estado do sistema de software e fazendo revisões no sistema.

As metas do gerente de configuração são garantir que sejam seguidos os procedimentos e políticas para criar, alterar e testar o código, bem como tornar acessíveis as informações sobre o projeto. Para implementar técnicas para manter controle sobre as mudanças de código, esse gerente introduz mecanismos para fazer solicitações oficiais de alterações, avaliá-las (por meio de um Grupo de Controle de Alterações responsável pela aprovação das alterações do sistema) e autorizar as alterações. O gerente cria e distribui listas de tarefas para os engenheiros e basicamente cria o contexto do projeto. Além disso, o gerente coleta dados estatísticos sobre os componentes do sistema de software, como, por exemplo, informações determinando quais componentes do sistema são problemáticos.

Para os engenheiros de software, o objetivo é trabalhar eficazmente. Isso significa que os engenheiros não interferem uns com os outros de forma desnecessária na criação e teste do código e na produção de artefatos de suporte. Mas, ao mesmo tempo, eles tentam se comunicar e coordenar eficientemente. Os engenheiros usam ferramentas que ajudam a criar artefatos consistentes. Eles se comunicam e se coordenam notificando uns aos outros sobre as tarefas necessárias e as tarefas concluídas. As alterações são propagadas por meio do trabalho dos outros mesclando arquivos. Existem mecanismos que asseguram que, para componentes submetidos a alterações simultâneas, há uma maneira de resolver conflitos e mesclar alterações. É mantido um histórico da evolu-

Quais são os objetivos e as atividades executadas pelas divisões envolvidas na gestão de alterações?

Deve haver um mecanismo para assegurar que alterações simultâneas ao mesmo componente sejam adequadamente rastreadas, gerenciadas e executadas.

¹ Esta seção foi extraída de [Dar01]. A permissão especial para reproduzir “Gama de Funcionalidade no Sistema CM” por Susan Dart [Dar01], ©2001 pelo Carnegie Mellon University, foi concedida pelo Software Engineering Institute.

ção de todos os componentes do sistema, juntamente com um registro (log) com os motivos das alterações e um registro do que realmente foi alterado. Os engenheiros têm seu próprio espaço de trabalho para criar, alterar, testar e integrar o código. Em certo ponto, o código é transformado em referencial, com base no qual o desenvolvimento continua e por meio do qual são criadas variações para as máquinas de destino.

O cliente usa o produto. Como o produto está sob o controle da Gestão de Configuração (CM), o cliente segue procedimentos formais para solicitar alterações e para indicar erros no produto.

No caso ideal, um sistema de CM usado nesse cenário deveria suportar todos esses papéis e tarefas; isto é, os papéis determinam a funcionalidade exigida para um sistema de CM. O gerente de projeto vê a CM como um mecanismo de auditoria; o gerente de configuração a vê como um mecanismo de controle, rastreamento e criador de políticas; o engenheiro de software a vê como um mecanismo de alteração, criação e controle de acesso; e o cliente a vê como um mecanismo de garantia de qualidade.

29.1.2 Elementos de um sistema de gestão de configuração

Em sua publicação sobre gestão de configuração de software, Susan Dart [Dar01] identifica quatro importantes elementos que devem existir quando um sistema de gestão de configuração é desenvolvido:

- *Elementos de componente* – conjunto de ferramentas acopladas em um sistema de gestão de arquivos (por exemplo, um banco de dados) que possibilita acesso à gestão de cada item de configuração de software.
- *Elementos de processo* – coleção de procedimentos e tarefas que definem uma abordagem eficaz de gestão de alterações (e atividades relacionadas) para todas as partes envolvidas na gestão, engenharia e uso do software.
- *Elementos de construção* – conjunto de ferramentas que automatizam a construção do software, assegurando que tenha sido montado o conjunto apropriado de componentes validados (isto é, a versão correta).
- *Elementos humanos* – conjunto de ferramentas e características de processo (abrangendo outros elementos de CM) usados pela equipe de software para implementar uma SCM eficaz.

Esses elementos (discutidos com mais detalhes em seções posteriores) não são mutuamente exclusivos. Por exemplo, elementos de componente funcionam em conjunto com elementos de construção à medida que o processo do software evolui. Elementos de processo guiam muitas atividades humanas relacionadas à SCM e, portanto, podem também ser consideradas como elementos humanos.

29.1.3 Referenciais

A mudança ou alteração é um fato normal no desenvolvimento de software. Clientes querem modificar requisitos. Desenvolvedores querem modificar a abordagem técnica. Gerentes querem modificar a abordagem do projeto. Por que todas essas modificações? A resposta é realmente muito simples. À medida que o tempo passa, todas as partes envolvidas sabem mais (sobre o que pre-

cisam, qual será a melhor abordagem e como conseguir que seja feito e ainda ganhar dinheiro). Esse conhecimento adicional é a força motriz que está por trás da maioria das alterações e leva à constatação de um fato que para muitos profissionais de engenharia de software é difícil de aceitar: *muitas alterações são justificadas!*

Uma *referência* é um conceito de gestão de configuração de software que ajuda a controlar alterações sem obstruir seriamente as alterações justificáveis. O IEEE (IEEE Std. No. 610.12-1990) define uma referência como:

Uma especificação ou produto que tenha sido formalmente revisado e acordado, que depois serve de base para mais desenvolvimento e pode ser alterado somente por meio de procedimentos formais de controle de alteração.

Para que um item de configuração de software se torne uma referência para o desenvolvimento, as alterações devem ser feitas rápida e informalmente. No entanto, uma vez estabelecida uma referência, podem ser feitas alterações, mas deve ser aplicado um processo específico e formal para avaliar e verificar cada alteração.

No contexto da engenharia de software, uma referência é um marco no desenvolvimento de software. Uma referência é marcada pelo fornecimento de um ou mais itens de configuração de software que foram aprovados em consequência de uma revisão técnica (Capítulo 20). Por exemplo, os elementos de um modelo de projeto foram documentados e revisados. Erros foram encontrados e corrigidos. Uma vez que todas as partes do modelo foram revisadas, corrigidas e, então, aprovadas, o modelo do projeto torna-se uma referência. Outras alterações na arquitetura do programa (documentadas no modelo de projeto) podem ser feitas apenas depois que cada uma tenha sido avaliada e aprovada. Embora as referências possam ser definidas em qualquer nível de detalhe, as referências de software mais comuns estão na Figura 29.1.

Muitas alterações de software são justificadas; portanto, não faz sentido reclamar delas. Em vez disso, esteja certo de ter os mecanismos prontos para cuidar delas.

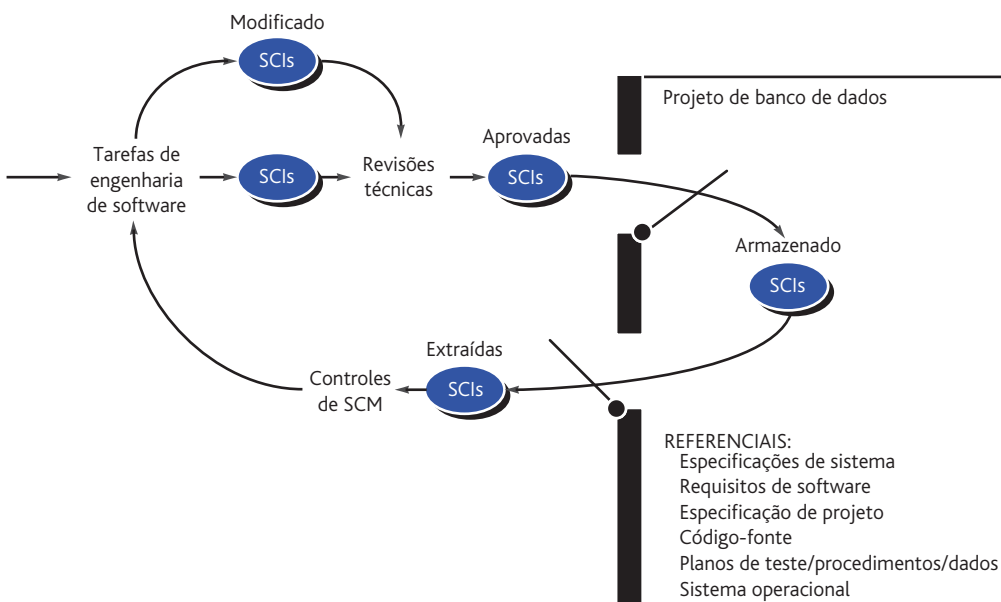


FIGURA 29.1 SCLs que se tornaram referenciais e o banco de dados de projeto.

Certifique-se de que o banco de dados de projeto seja mantido em um local centralizado e controlado.

A sequência de eventos que levam a uma referência também está ilustrada na Figura 29.1. Tarefas de engenharia de software produzem um ou mais SCIs. Depois que os SCIs são revisados e aprovados, eles são colocados em um *banco de dados de projeto* (também chamado de *biblioteca de projeto* ou *repositório de software* e discutido na Seção 29.3). Quando um membro de uma equipe de engenharia de software quer fazer uma modificação em um SCI que se tornou referencial, ele é copiado do banco de dados de projeto para o espaço de trabalho privado do engenheiro. Porém, esse SCI extraído só pode ser modificado se os controles de SCM (discutidos mais adiante neste capítulo) forem seguidos. As setas na Figura 29.1 ilustram o caminho de modificação para um SCI referencial.

29.1.4 Itens de configuração de software

Já definimos que um item de configuração de software é uma informação criada como parte do processo de engenharia de software. Em um caso extremo, um SCI poderia ser considerado uma única seção de uma grande especificação ou um caso de teste em um grande conjunto de testes. Em uma visão mais realista, um SCI é todo ou parte de um artefato (por exemplo, um documento, um conjunto inteiro de casos de teste ou um programa ou componente com nome).

Além dos SCIs derivados dos artefatos de software, muitas organizações de engenharia de software também colocam ferramentas de software sob o controle de configuração. Isto é, versões específicas de editores, compiladores, navegadores e outras ferramentas automáticas são “congeladas” como parte da configuração do software. Como essas ferramentas foram usadas para produzir documentação, código-fonte e dados, devem estar disponíveis quando alterações forem feitas na configuração do software. Embora os problemas sejam raros, é possível que uma nova versão de uma ferramenta (por exemplo, um compilador) possa produzir resultados diferentes daqueles da versão original. Por essa razão, as ferramentas, assim como o software que elas ajudam a produzir, podem ser referenciadas como parte de um processo bem especificado de gestão de configuração.

Na realidade, os SCIs são organizados para formar objetos de configuração que podem ser catalogados no banco de dados do projeto com um nome único. Um *objeto de configuração* tem um nome, atributos e é “conectado” a outros objetos por relações. De acordo com a Figura 29.2, os objetos de configuração, **DesignSpecification**, **DataModel**, **ComponentN**, **SourceCode** e **TestSpecification** são definidos separadamente. No entanto, cada um dos objetos está relacionado aos outros, como mostram as setas. Uma seta curva indica uma relação de composição. Isto é, **DataModel** e **ComponentN** fazem parte do objeto **DesignSpecification**. Uma seta reta bidirecional indica uma inter-relação. Se for feita uma alteração no objeto **SourceCode**, as inter-relações permitem determinar quais outros objetos (e SCIs) podem ser afetados.²

² Essas relações são definidas no banco de dados. A estrutura do banco de dados (repositório) é discutida em mais detalhes na Seção 29.3.

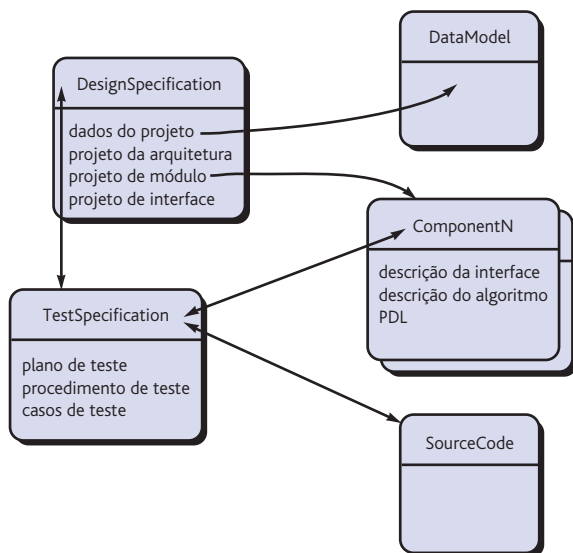


FIGURA 29.2 Objetos de configuração.

29.1.5 Gestão de dependências e alterações

Apresentamos a noção de rastreabilidade e o uso de matriz de rastreabilidade na Seção 8.2.6. A matriz de rastreabilidade é um modo de documentar dependências entre requisitos, decisões de arquitetura (Seção 13.5) e causas de defeito (Seção 21.6). Essas dependências precisam ser levadas em conta ao se determinar o impacto de uma alteração proposta e orientar a escolha dos casos de teste que devem ser usados para teste de regressão (Seção 22.3.2). A gestão de dependências pode ser vista como gestão de impacto.³ Isso ajuda os desenvolvedores a se concentrar em como as alterações feitas afetam seu trabalho [Sou08].

A *análise de impacto* se concentra no comportamento organizacional e nas ações individuais. A gestão de impacto envolve dois aspectos complementares: (1) garantir que os desenvolvedores de software empreguem estratégias para minimizar o impacto das ações de seus colegas em seu próprio trabalho e (2) estimular os desenvolvedores de software a usar práticas que minimizem o impacto de seu trabalho no de seus colegas. É importante observar que, quando um desenvolvedor tenta minimizar o impacto de seu trabalho sobre o de outros, ele também está diminuindo o trabalho necessário para minimizar o impacto de seu próprio trabalho no deles [Sou08].

É importante manter os artefatos de software para garantir que os desenvolvedores estejam cientes das dependências entre os SCIs. Os desenvolvedores precisam estabelecer uma disciplina ao verificar itens dentro e fora do repositório SCM e ao fazer alterações aprovadas, conforme discutido na Seção 29.2. Porém, software de monitoramento de erros também é útil para ajudar a descobrir dependências de SCI. A comunicação eletrônica (e-mail, wikis, redes sociais) oferece maneiras convenientes para os desenvolvedores compartilharem dependências e problemas não documentados, à medida que surgem.

³ A gestão de impacto é discutida com mais detalhes na Seção 29.3.4

29.2 O repositório de SCM

O repositório de SCM é um conjunto de mecanismos e estruturas de dados que permitem a uma equipe de software gerenciar alterações de maneira eficaz. Ele fornece as funções óbvias de um sistema moderno de gestão de banco de dados, garantindo a integridade dos dados, compartilhamento e integração. Além disso, o repositório de SCM proporciona um centralizador (hub) para a integração das ferramentas de software, está no centro do fluxo do processo de software e pode impor estrutura e formato uniformes para os artefatos.

Para tanto, o repositório é definido em termos de um metamodelo. O *metamodelo* determina como as informações são armazenadas no repositório, como os dados podem ser acessados pelas ferramentas e visualizados pelos engenheiros de software, quão bem pode ser mantida a segurança e a integridade dos dados e com que facilidade o modelo existente pode ser estendido para satisfazer a novas necessidades.

29.2.1 Características gerais e conteúdo

As características e o conteúdo do repositório são mais bem compreendidos quando observados a partir de duas perspectivas: o que tem de ser armazenado no repositório e quais são os serviços específicos fornecidos pelo repositório. A Figura 29.3 mostra uma divisão detalhada dos tipos de representações, documentos e outros produtos que são armazenados no repositório.

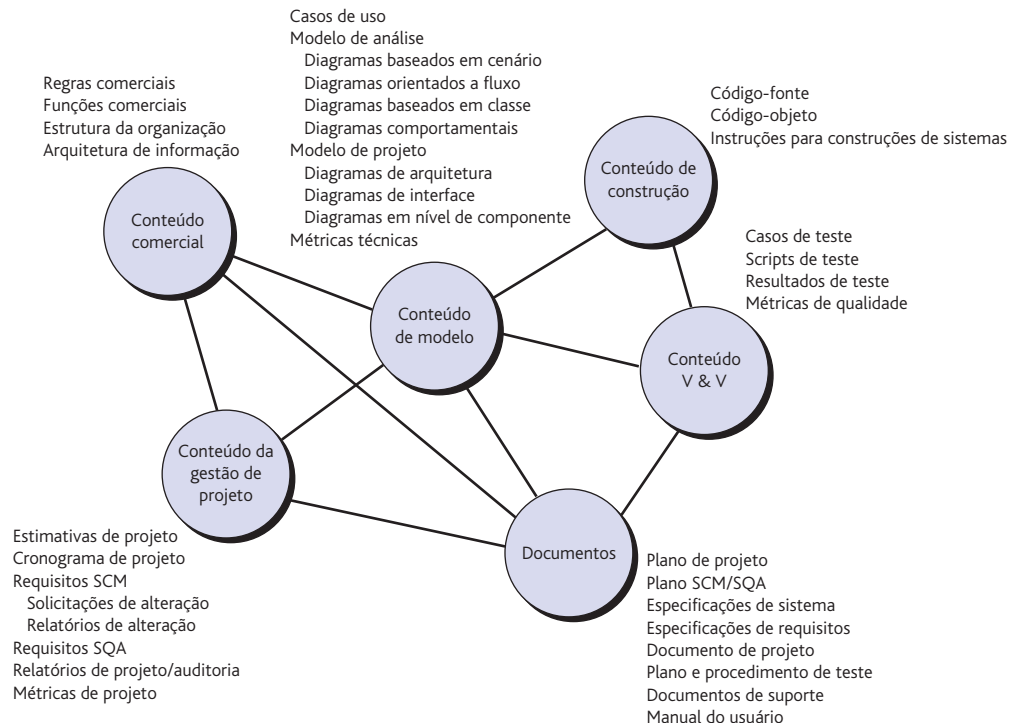


FIGURA 29.3 Conteúdo do repositório.

Um repositório robusto fornece duas classes diferentes de serviços: (1) os mesmos tipos de serviços que podem ser esperados de qualquer sistema sofisticado de gerenciamento de banco de dados e (2) serviços específicos ao ambiente de engenharia de software.

Um repositório que sirva a uma equipe de engenharia de software deve também (1) integrar ou suportar diretamente as funções de gestão de processo, (2) suportar regras específicas que governam a função SCM e os dados mantidos no repositório, (3) fornecer uma interface para outras ferramentas de engenharia de software e (4) acomodar o armazenamento de objetos de dados sofisticados (por exemplo, texto, gráficos, vídeo, áudio).

Um exemplo de repositório disponível comercialmente pode ser obtido em www.oracle.com/technology/products/repository/index.html.

29.2.2 Características da SCM

Para suportar a SCM, o repositório deve ter um conjunto de ferramentas que dê suporte para estas características:

Versões. À medida que um projeto avança, serão criadas muitas versões (Seção 29.3.2) dos artefatos individuais. O repositório deve ser capaz de salvar todas essas versões para possibilitar uma gestão eficaz das versões do produto e permitir aos desenvolvedores retroceder a versões anteriores durante o teste e depuração.

O repositório deve poder controlar uma grande variedade de tipos de objetos, incluindo texto, gráficos, bitmaps, documentos complexos e objetos especiais, como definições de tela e relatório, arquivos de objeto, dados de testes e resultados. Um repositório maduro rastreia versões de objetos com níveis arbitrários de granularidade; por exemplo, podem ser rastreados uma definição de dados especial ou um conjunto de módulos.

O repositório deve ser capaz de manter SCIs relacionados a muitas versões diferentes do software. Mais importante ainda, deve fornecer os mecanismos para montagem desses SCIs em uma configuração específica da versão.

Acompanhamento de dependências e gestão de alterações. O repositório gerencia uma grande variedade de relações entre os elementos de dados nele armazenados. Isso inclui relações entre entidades e processos corporativos, entre as partes do projeto de uma aplicação, entre componentes de projeto e arquitetura de informações corporativas, entre elementos de projeto e outros artefatos e assim por diante. Algumas dessas relações são meramente associações, e outras são relações de dependência ou de obrigatoriedade.

A capacidade de manter o controle de todas essas relações é crucial para a integridade das informações armazenadas no repositório e para a geração de outros produtos nele baseados e é uma das contribuições mais importantes do conceito de repositório para o aperfeiçoamento do processo de desenvolvimento de software. Por exemplo, se um diagrama de classes UML é modificado, o repositório pode detectar se as classes relacionadas, as descrições de interface e os componentes de código também exigem modificações e podem chamar a atenção do desenvolvedor para os CSIs afetados.

Controle de requisitos. Essa função especial depende da gestão de links e proporciona a capacidade de controlar todos os componentes de projeto e construção e outros produtos que resultam de uma especificação especial de requisitos (acompanhamento adiante). Além disso, proporciona a capacidade de identificar quais requisitos geraram determinado produto (acompanhamento retroativo).

Gestão de configuração. O recurso de gestão de configuração mantém controle de uma série de configurações representando marcos de projeto específicos ou versões de produção.

Pistas de auditoria. Uma pista de auditoria estabelece informações adicionais sobre quando, por que e por quem foram feitas as alterações. As informações sobre a origem das alterações podem ser colocadas como atributos de objetos específicos no repositório. Um mecanismo de disparo de repositório é útil para avisar o desenvolvedor ou a ferramenta que está sendo usada para iniciar a aquisição de informações de auditoria (como, por exemplo, a razão de uma alteração) sempre que um elemento de projeto for modificado.

29.3 O processo SCM

"Qualquer alteração, mesmo uma alteração para melhor, é acompanhada de contratempos e desconfortos."

Arnold Bennett

O processo de gestão de configuração de software define uma série de tarefas que têm quatro objetivos principais: (1) identificar todos os itens que coletivamente definem a configuração do software, (2) gerenciar alterações de um ou mais desses itens, (3) facilitar a construção de diferentes versões de uma aplicação e (4) assegurar que a qualidade do software seja mantida à medida que a configuração evolui com o tempo.

Um processo que atinja esses objetivos não precisa ser burocrático e pesado, mas deve ser caracterizado de maneira que permita à equipe de software desenvolver respostas a várias questões complexas:

- Como uma equipe de software identifica os elementos discretos de uma configuração de software?
- Como uma organização lida com as várias versões de um programa (e sua documentação) de maneira que venha a permitir que a alteração seja acomodada eficientemente?
- Como uma organização controla alterações antes e depois que o software é entregue ao cliente?
- Como uma organização avalia o impacto das alterações e gerencia o impacto efetivamente?
- Quem tem a responsabilidade de aprovar e classificar as alterações solicitadas?
- Como podemos assegurar que as alterações foram feitas corretamente?
- Que mecanismo é usado para alertar outras pessoas sobre as alterações feitas?

A partir dessas questões são definidas cinco tarefas SCM – identificação, controle de versão, controle de alteração, auditoria de configuração e relatos –, ilustradas na Figura 29.4.

De acordo com a figura, as tarefas SCM podem ser vistas como camadas concêntricas. Os SCIs fluem para fora através dessas camadas por toda a sua vida útil, tornando-se, por fim, parte da configuração do software de uma ou mais versões da aplicação ou sistema. À medida que um SCI se move através de uma camada, as ações deduzidas por cada tarefa SCM podem ser ou não

Que questões o processo SCM está designado a responder?

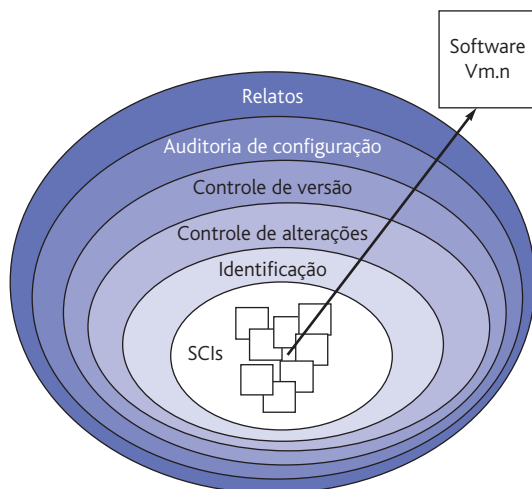


FIGURA 29.4 Camadas do processo SCM.

aplicáveis. Por exemplo, quando um novo SCI é criado, ele deve ser identificado. No entanto, se não forem solicitadas alterações para o SCI, a camada de controle de alteração não se aplica. O SCI é atribuído a uma versão específica do software (aqui entram em ação os mecanismos de controle de versão). É mantido um registro do SCI (seu nome, data de criação, designação da versão etc.) para fins de auditoria da configuração e relato para aqueles que precisam ter conhecimento. Nas próximas seções, examinaremos cada uma dessas camadas de processo SCM em mais detalhes.

29.3.1 Identificação de objetos na configuração de software

Para controlar e gerenciar itens de configuração de software, cada um deles deve ser nomeado separadamente e depois organizado usando uma abordagem orientada a objetos. Podem ser identificados dois tipos de objeto [Cho89]: objetos básicos e objetos agregados.⁴ O *objeto básico* é uma unidade de informação que se cria durante a análise, projeto, codificação ou teste. Por exemplo, o objeto básico pode ser uma seção de especificação de requisitos, parte de um modelo de projeto, código-fonte para um componente ou um conjunto de casos de teste usados para exercitar o código. O *objeto agregado* é uma coleção de objetos básicos e outros objetos agregados. Por exemplo, uma **DesignSpecification** é um objeto agregado. Conceitualmente, ela pode ser vista como uma lista nomeada (identificada) de ponteiros que especificam objetos agregados, como, por exemplo, **ArchitecturalModel** e **DataModel**, e *objetos básicos*, como **ComponentN** e **UMLDiagramN**.

Cada objeto tem um conjunto de características distintas que o identificam de forma única: um nome, uma descrição, uma lista de recursos e uma “realização”. O nome do objeto é uma sequência de caracteres que o identifica de forma não ambígua. A descrição do objeto é uma lista de itens de dados que identifica o tipo de SCI (por exemplo, elemento de modelo, programa, dados)

As inter-relações estabelecidas para objetos de configuração permitem avaliar o impacto da alteração.

⁴ O conceito de objeto agregado [Gus89] foi proposto como mecanismo para representar uma versão completa de uma configuração de software.

representado pelo objeto, um identificador de projeto e informações sobre alteração e/ou versão. Recursos são “entidades fornecidas, processadas, referenciadas ou de qualquer outra forma exigidas pelo objeto” [Cho89]. Por exemplo, tipos de dados, funções específicas ou até mesmo nomes de variáveis podem ser considerados recursos de objeto. A realização pode ser um ponteiro para a “unidade de texto” para um objeto básico e null para um objeto agregado.

A identificação do objeto de configuração também pode considerar as relações entre objetos nomeados. Por exemplo, usando a notação simples

Class diagram <part-of> requirements model;

Requirements model <part-of> requirements specification;

criamos uma hierarquia de SCIs.

Em muitos casos, objetos são inter-relacionados por ramificações da hierarquia do objeto. Essas relações que cruzam a estrutura podem ser representadas da seguinte maneira:

data model <interrelated> data flow model;

data model <interrelated> test case class m;

No primeiro caso, a inter-relação é entre um objeto composto, enquanto a segunda relação é entre um objeto agregado (**DataModel**) e um objeto básico (**TestCaseClassM**).

O esquema de identificação para objetos de software deve reconhecer que objetos evoluem por meio do processo de software. Antes que um objeto seja um referencial, ele pode mudar muitas vezes, e, até mesmo após um referencial ter sido estabelecido, as mudanças podem ser muito frequentes.

29.3.2 Controle de versão

O controle de versão combina procedimentos e ferramentas para gerenciar diferentes versões dos objetos de configuração criados durante o processo de software. Um sistema de controle de versão implementa ou está diretamente integrado a quatro recursos principais: (1) um banco de dados de projeto (repositório) que armazena todos os objetos de configuração relevantes, (2) um recurso de *gestão de versão* que armazena todas as versões de um objeto de configuração (ou permite que qualquer versão seja construída usando diferenças das versões anteriores), (3) uma *facilidade de construir* que permite coletar todos os objetos de configuração relevantes e construir uma versão específica do software. Além disso, os sistemas de controle de versão e controle de alteração muitas vezes implementam um recurso chamado *acompanhamento de tópicos* (também conhecido como *acompanhamento de bug*), que permite à equipe de software registrar e acompanhar o status de todos os problemas pendentes associados a cada objeto de configuração.

Alguns sistemas de controle de versão criam um *conjunto de modificações* – uma coleção de todas as alterações (em relação a alguma configuração referencial) necessárias para criar uma versão específica do software. Dart [Dar91] observa que um conjunto de modificações “captura todas as alterações

Mesmo que o banco de dados do projeto tenha a capacidade de estabelecer essas relações, elas demoram a ser estabelecidas e são difíceis de manter atualizadas. Embora muito úteis para análise de impacto, não são essenciais para a gestão geral das alterações.

em todos os arquivos da configuração, juntamente com a razão para aquelas alterações e os detalhes de quem as fez e quando”.

Alguns conjuntos de modificações que receberam denominação podem ser identificados para uma aplicação ou sistema. Isso permite construir uma versão do software especificando os conjuntos de modificações (pelo nome) que devem ser aplicados à configuração referencial. Para isso, aplica-se uma abordagem de *modelagem de sistema*. O modelo de sistema contém: (1) um *gabarito* que inclui hierarquia de componentes e uma “ordem de construção” para os componentes, descrevendo como o sistema deve ser construído, (2) regras de construção e (3) regras de verificação.⁵

Com o passar dos anos foram propostas muitas abordagens automáticas diferentes para o controle de versão. A principal diferença entre as abordagens é a sofisticação dos atributos usados para construir versões específicas e variantes de um sistema e os mecanismos do processo de construção.

FERRAMENTAS DO SOFTWARE



O sistema de versões concorrentes (CVS, concurrent versions system)

O uso de ferramentas para obter o controle de versão é essencial para uma gestão eficaz das alterações. O sistema de versões concorrentes (CVS) é uma ferramenta amplamente utilizada para controle de versão. Projetada originalmente para código-fonte, mas útil para qualquer arquivo baseado em texto, o sistema CVS (1) estabelece um repositório simples, (2) mantém todas as versões de um arquivo sob um único nome de arquivo, armazenando apenas as diferenças entre versões progressivas do arquivo original e (3) protege contra alterações simultâneas de um arquivo, estabelecendo diferentes diretórios para cada desenvolvedor e isolando, assim, uns dos outros. O CVS mescla as alterações quando cada desenvolvedor completa seu trabalho.

É importante notar que o CVS não é um sistema de construção; ele não constrói uma versão específica do software. Outras ferramentas (por exemplo, *Makefile*) devem ser integradas ao CVS para conseguir isso. O CVS não implementa um processo de controle de alteração (por exemplo, solicitações de alterações, relatos de alterações, acompanhamento de bugs).

Mesmo com essas limitações, o CVS “é um sistema de controle de versão predominante de código aberto transparente à rede [que] é útil para qualquer um, desde desenvolvedores individuais até grandes equipes distribuídas” [CVS07]. Sua estrutura cliente-servidor permite aos usuários acessar arquivos via conexões de Internet e sua filosofia de código aberto o torna disponível às plataformas mais populares.

O CVS está disponível sem custo para ambientes Windows, Mac OS, Linux e UNIX, e existe uma versão de código-fonte aberto da aplicação [CVS12].⁶

29.3.3 Controle de alterações

A realidade do controle de alterações em um contexto de engenharia de software foi resumida elegantemente por James Bach [Bac98]:

O controle de alterações é vital, mas as forças que o tornam necessário também o tornam inconveniente. Temos medo das alterações porque uma pequena perturbação no código pode criar uma enorme falha no produto, mas elas também podem reparar uma grande falha ou habilitar novos e maravilhosos recursos. Temos medo das alterações porque um único desenvolvedor irresponsável pode afundar o projeto todo; embora ideias brilhantes possam surgir nas mentes desses brincalhões, um controle de processo de alterações pesado poderia desestimulá-los em seu trabalho criativo.

“A arte do progresso é preservar a ordem nas alterações e preservar as alterações na ordem.”

Alfred North Whitehead

⁵ É possível também consultar o modelo do sistema para avaliar como uma alteração em um componente afeta outros componentes.

⁶ Link de download para [CVS12]: <http://olex.openlogic.com/packages/cvs>.

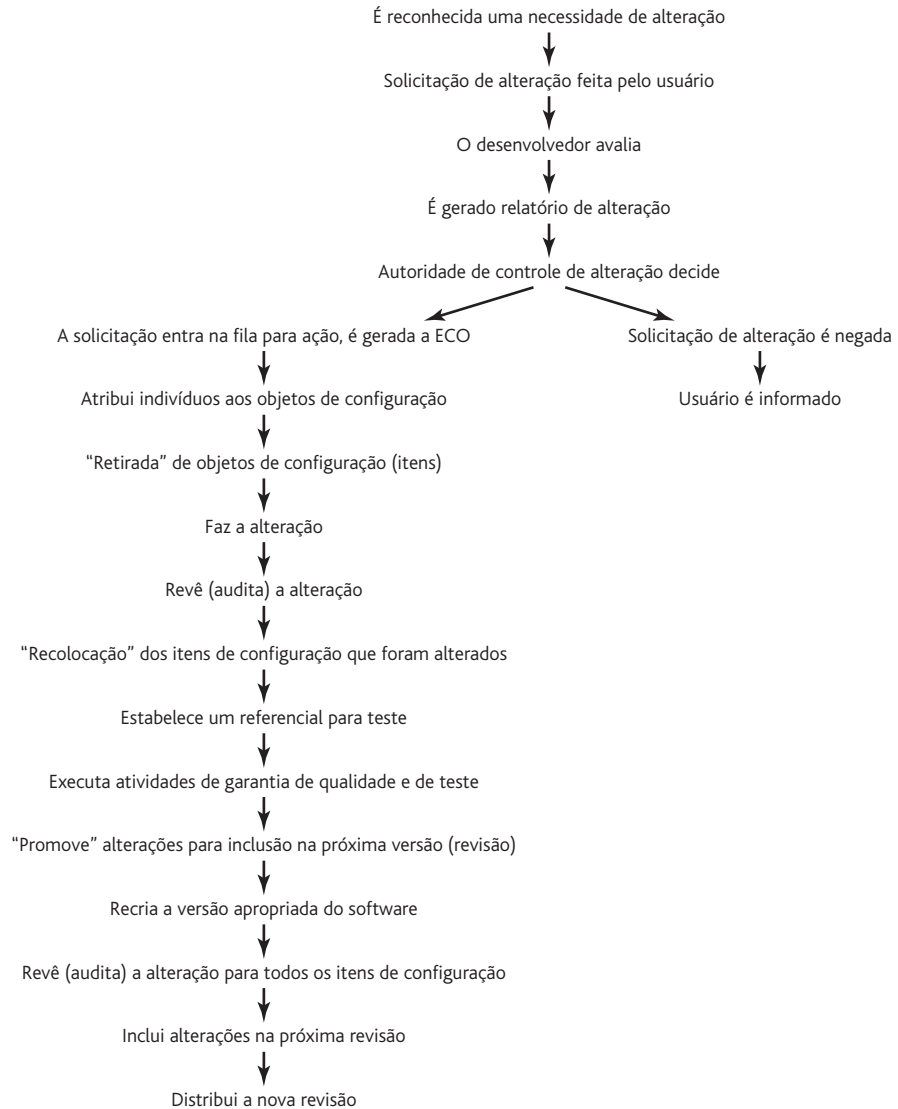


FIGURA 29.5 O processo de controle de alterações.

Deve-se observar que muitas solicitações de alterações podem ser combinadas para resultar em uma única ECO e que ECOs resultam tipicamente em alterações em vários objetos de configuração.

Bach reconhece que temos aqui uma lei de equilíbrio. Se tivermos muito controle das alterações, criaremos problemas. Se tivermos pouco controle, criaremos outros problemas.

Em um projeto de software grande, alterações não controladas levam rapidamente ao caos. Para projetos assim, o controle de alterações combina procedimentos humanos e ferramentas automatizadas, proporcionando um mecanismo para o controle de alterações. O processo de controle de alterações está ilustrado esquematicamente na Figura 29.5. Uma *solicitação de alteração* é apresentada e avaliada para determinar o mérito técnico, efeitos colaterais em potencial, o impacto global sobre outros objetos de configuração e funções do sistema e o custo projetado da alteração. Os resultados da avaliação são apresentados como um *relatório de alterações*, usado por uma *autoridade de controle de alterações* (CCA, *change control authority*) – uma pessoa ou grupo de pessoas que toma a decisão final sobre o status e a prioridade da alteração.

Uma *ordem de alteração de engenharia* (ECO, *engineering change order*) é gerada para cada alteração aprovada. A ECO descreve a alteração a ser feita, as restrições que devem ser respeitadas e o critério para revisar e auditar.

Os objetos a serem alterados podem ser colocados em um diretório que é controlado apenas pelo engenheiro de software que está fazendo a alteração. Um sistema de controle de versão (consulte o quadro “Ferramentas de software” sobre CVS) atualiza o arquivo original logo que a alteração foi feita. Como alternativa, os objetos a serem alterados podem ser “retirados” do banco de dados do projeto (repositório), a alteração é feita e são aplicadas as atividades SQA apropriadas. Os objetos são então “colocados” no banco de dados e são usados mecanismos de controle de versão apropriados (Seção 29.3.2) para criar a próxima versão do software.

Esses mecanismos de controle de versão, integrados ao processo de controle de alterações, implementam dois elementos importantes da gestão de alterações – controle de acesso e controle de sincronização. O *controle de acesso* determina quais engenheiros de software têm autoridade para acessar e modificar um objeto de configuração específico. O *controle de sincronização* ajuda a assegurar que alterações paralelas, executadas por duas pessoas diferentes, não sobrescrevam uma à outra.

O nível de burocracia gerado pela descrição do processo de controle de alteração mostrado na Figura 29.5 pode incomodar. Essa sensação é comum. Sem as condições de segurança apropriadas, o controle de alterações pode retardar o progresso e criar barreiras desnecessárias. Grande parte dos desenvolvedores de software que usam mecanismos de controle de alterações (infelizmente, muitos não usam nenhum) já criou uma série de camadas de controle para ajudar a evitar os problemas mencionados aqui.

Antes de um SCI se tornar um referencial, só é necessário usar o *controle informal de alteração*. O desenvolvedor do objeto de configuração (SCI) em questão pode fazer todas as alterações justificáveis pelo projeto e pelos requisitos técnicos (desde que as alterações não afetem requisitos mais amplos do sistema que estejam fora do escopo de trabalho do desenvolvedor). Uma vez que o objeto tenha passado pela revisão técnica e tenha sido aprovado, pode ser criado um referencial.⁷ Uma vez que um SCI se torna um referencial, é implementado o *controle de alterações em nível de projeto*. Agora, para fazer uma alteração, o desenvolvedor precisa ter novamente a aprovação do gerente de projeto (se a alteração for “local”) ou da CCA, se a alteração afetar outros SCIs. Em alguns casos, o desenvolvedor prescinde da geração formal de pedidos de alteração, relatórios de alterações e ECOs. No entanto, é feita a avaliação de cada alteração e todas as alterações são acompanhadas e revisadas.

Quando o artefato de software é liberado para os clientes, institui-se o *controle formal de alterações*. O procedimento formal de controle de alterações foi resumido na Figura 29.5.

A autoridade de controle de alterações desempenha um papel ativo no segundo e terceiro níveis de controle. Dependendo do tamanho e do tipo

Opte por um pouco mais de controle de alterações do que você acha que precisará. É provável que a dose certa seja bem maior.

“O troco é inevitável, exceto para as máquinas de refrigerantes automáticas.”

Adesivo em para-choque

⁷ Um referencial também pode ser criado por outras razões. Por exemplo, quando são criadas “construções diárias”, todos os componentes verificados por determinado tempo se tornam o referencial para o trabalho do dia seguinte.

de projeto de software, a CCA pode ser composta por uma pessoa – o gerente de projeto – ou um grupo de pessoas (por exemplo, representantes do software, hardware, engenharia do banco de dados, suporte, marketing). O papel da CCA é assumir uma visão global, ou seja, avaliar o impacto das alterações além da SCI em questão. Como a alteração afetará o hardware? Como a alteração afetará o desempenho? Como a alteração modificará a percepção do cliente com relação ao produto? Como a alteração afetará a qualidade e a confiabilidade do produto? Essas e muitas outras questões são resolvidas pela CCA.

CASASEGURA

**Problemas de SCM**

Cena: Escritório de Doug Miller no início do projeto do software *CasaSegura*.

Atores: Doug Miller (gerente da equipe de engenharia de software do *CasaSegura*), Vinod Raman, Jamie Lazar e outros membros da equipe.

Conversa:

Doug: Sei que ainda é cedo para isso, mas precisamos falar sobre gestão de alterações.

Vinod (rindo): Difícil! O pessoal do marketing ligou hoje de manhã e eles tinham pensado melhor e estavam com algumas “dúvidas”. Nada importante, mas é só o começo.

Jamie: Fomos muito informais sobre a gestão de alterações em projetos anteriores.

Doug: Eu sei, mas este é maior e mais importante, e pelo que me lembro...

Vinod (acenando afirmativamente): Nós nos matamos por causa de alterações descontroladas no projeto de controle de luz ambiental... lembro dos atrasos...

Doug (franzindo a testa): Um pesadelo que prefiro não lembrar.

Jamie: Então o que faremos?

Doug: Acho que devemos fazer três coisas. Primeiro temos que desenvolver – ou tomar emprestado – um processo de controle de alterações.

Jamie: Você quer dizer “o modo como as pessoas solicitam alterações”?

Vinod: Sim, mas também como avaliamos a alteração, como decidimos quem faz (se é que nós decidimos isso) e como mantemos os registros do que é afetado pela alteração.

Doug: Segundo, precisamos realmente arranjar uma boa ferramenta de SCM para controle de versão e alteração.

Jamie: Podemos criar um banco de dados para todos os nossos artefatos.

Vinod: Nesse contexto, elas são chamadas de SCIs, e há muitas ferramentas boas que proporcionam suporte para isso.

Doug: É um bom começo, agora temos que...

Jamie: Ei, Doug, você disse que eram três coisas...

Doug (sorrindo): Terceiro: todos nós temos que seguir os processos de gestão de alterações e usar as ferramentas. Custe o que custar, ok?

29.3.4 Gestão de impacto

Uma teia de interdependências de artefatos de software precisa ser considerada sempre que uma alteração é feita. A *gestão de impacto* abrange o trabalho exigido para se entender corretamente essas interdependências e controlar seus efeitos sobre outros SCIs (e as pessoas responsáveis por eles).

A gestão de impacto é realizada com três ações [Sou08]. Primeiro, uma *rede de impacto* identifica os membros de uma equipe de software (e outros envolvidos) que podem afetar ou ser afetados pelas alterações feitas no software. Uma definição clara da arquitetura do software (Capítulo 13) ajuda muito na criação de uma rede de impacto. Em seguida, a *gestão de impacto adian-*

te (*forward impact management*) avalia o impacto das alterações feitas por você sobre os membros da rede de impacto e, então, informa-os sobre o impacto dessas alterações. Por último, a *gestão de impacto retroativo* (*backward impact management*) examina as alterações feitas por outros membros da equipe e o impacto sobre o seu trabalho e incorpora mecanismos para reduzir o impacto.

29.3.5 Auditoria de configuração

Identificação, controle de versão e controle de alterações ajudam a manter a ordem naquilo que, de outra forma, seria uma situação caótica. No entanto, até mesmo os melhores mecanismos de controle rastreiam uma alteração somente até que seja gerada uma ECO. Como a equipe de software pode assegurar que a alteração foi implementada corretamente? A resposta é dupla: (1) revisões técnicas e (2) a auditoria de configuração de software.

A revisão técnica (Capítulo 20) focaliza a exatidão técnica do objeto de configuração modificado. Os revisores avaliam o SCI para determinar a consistência com outros SCIs, omissões ou efeitos colaterais em potencial. Deve ser feita uma revisão técnica para todas as alterações, exceto as mais triviais.

Uma *auditoria de configuração de software* complementa a revisão técnica avaliando o objeto de configuração quanto a características que em geral não são consideradas durante a revisão. A auditoria propõe e responde às seguintes questões:

1. Foi feita a alteração especificada na ECO? Alguma modificação adicional foi incorporada?
2. Foi feita uma revisão técnica para avaliar a exatidão técnica?
3. Seguiu-se o processo do software e os padrões de engenharia de software foram aplicados adequadamente?
4. A alteração foi “destacada” no SCI? A data e o autor da alteração foram especificados? Os atributos do objeto de configuração refletem a alteração?
5. Seguiram-se os procedimentos da SCM para anotar, registrar e relatar a alteração?
6. Todos os SCIs relacionados foram adequadamente atualizados?

Quais são as perguntas primárias que fazemos durante uma auditoria de configuração?

Em alguns casos, as perguntas de auditoria são formuladas como parte da revisão técnica. No entanto, quando a SCM é uma atividade formal, a auditoria de configuração é conduzida separadamente pelo grupo de garantia de qualidade. Essas auditorias de configuração formais também asseguram que os SCIs corretos (por versão) tenham sido incorporados em uma construção específica e que toda a documentação esteja atualizada e consistente com a versão construída.

29.3.6 Relatório de status

O *relatório de status de configuração* (às vezes chamado de *contabilidade de status*) é uma tarefa da SCM que responde às seguintes questões: (1) O que aconteceu? (2) Quem fez? (3) Quando aconteceu? (4) O que mais será afetado?

Desenvolva uma lista do tipo "precisa saber" para todo objeto de configuração e mantenha-a atualizada. Quando for feita uma alteração, certifique-se de que todos os que estão na lista sejam notificados.

O fluxo de informações para o relatório de status de configuração (CSR) está ilustrado na Figura 29.5. A cada vez que é atribuída uma identificação nova ou atualizada a um SCI, faz-se uma entrada no CSR. Cada vez que uma alteração é aprovada pela CCA (isto é, é gerada uma ECO), é feita uma entrada no CSR. Cada vez que se executa uma auditoria de configuração, os resultados são relatados como parte da tarefa do CSR. A saída do CSR pode ser colocada em um banco de dados online ou em um site, de forma que os desenvolvedores de software ou pessoal de suporte possam acessar as informações de alterações por categoria de palavra-chave. Além disso, é gerado um relatório do CSR regularmente; ele se destina a manter a gerência e os profissionais informados sobre alterações importantes.

FERRAMENTAS DO SOFTWARE



Suporte de SCM

Objetivo: as ferramentas de SCM proporcionam suporte para uma ou mais das atividades de processo discutidas na Seção 29.3.

Mecanismos: muitas ferramentas de SCM modernas funcionam em conjunto com um repositório (um sistema de banco de dados) e proporcionam mecanismos para identificação, versão e controle de alterações, auditoria e relatórios.

Ferramentas representativas:⁸

Software Change Manager, distribuída pela Computer Associates (<http://www.ca.com/us/products/detail/ca-change-manager-enterprise-workbench.aspx>), é um sistema SCM multiplataforma.

ClearCase, desenvolvida pela Rational, fornece uma família de funções SCM (<http://www-03.ibm.com/software/products/us/en/clearcase>).

Serena Dimensions CMF, distribuída pela Serena (<http://www.serena.com/index.php/en/products/dimensions-cmf/>), fornece um conjunto completo de ferramentas SCM aplicáveis a software convencional e WebApps.

Allura, distribuída pela SourceForge Inc. (<http://sourceforge.net/p/allura/wiki/Allura%20Wiki/>), fornece gerenciamento de versão, recursos de construção, rastreamento de problemas/bugs e muitos outros recursos de gestão.

SurroundSCM, desenvolvida pela Seapine Software, fornece recursos completos de gestão de alterações (www.seapine.com).

Vesta, distribuída pela Compac, é um sistema de SCM de domínio público que pode suportar projetos pequenos (< 10 KLOC) e grandes (10.000 KLOC) (www.vestasys.org).

Uma lista abrangente de ferramentas e ambientes de SCM comerciais pode ser encontrada em www.cmtoday.com.

29.4 Gestão de configuração para WebApps e aplicativos móveis

Anteriormente neste livro, discutimos a natureza especial das WebApps e dos aplicativos móveis e os métodos especializados⁹ necessários para criá-los. Dentre as muitas características que diferenciam essas aplicações do software tradicional está a natureza onipresente da alteração.

Os desenvolvedores de WebApps e aplicativos móveis muitas vezes usam um modelo de processo iterativo, incremental, que aplica muitos princípios derivados do desenvolvimento ágil de software (Capítulo 5). Por meio dessa

⁸ A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

⁹ Consulte [Pre08] para ver uma discussão simples dos métodos de engenharia para Web.

abordagem, uma equipe de engenharia muitas vezes desenvolve um incremento em um período de tempo muito curto usando uma abordagem focada no cliente. Incrementos subsequentes adicionam conteúdo e funcionalidade, e cada um deles tende a implementar alterações que levam a um conteúdo aperfeiçoado, melhor utilização, melhor estética, melhor navegação, melhor desempenho e maior segurança. Portanto, no mundo ágil das WebApps e aplicativos móveis, a alteração é vista de forma um tanto diferente.

Se você é membro de uma equipe de software que constrói WebApps ou aplicativos móveis, tem de adotar as alterações. E ainda mais, uma equipe ágil típica evita todas as coisas que parecem ser intensivas e que tornam o processo pesado, burocrático e formal. A gestão de configuração de software é com frequência vista (embora incorretamente) como detentora dessas características. Essa contradição é remediada não pela rejeição dos princípios, práticas e ferramentas de SCM, mas sim moldando-as para satisfazerem às necessidades especiais dos projetos de WebApps e aplicativos móveis.

29.4.1 Problemas predominantes

À medida que as WebApps e os aplicativos móveis se tornam mais importantes para a sobrevivência e o crescimento dos negócios, crescem as necessidades da gestão de configuração. Por quê? Porque sem controles eficazes, alterações impróprias nessas aplicações (lembre-se de que o imediatismo e a evolução contínua são os atributos dominantes) podem levar a uma colocação não autorizada de informações sobre novos produtos, funcionalidade errônea ou mal testada que causa frustração nos usuários, brechas na segurança que põem em risco os sistemas internos da empresa e outras consequências economicamente desagradáveis ou mesmo desastrosas.

As estratégias gerais para a gestão de configuração de software (SCM) descritas neste capítulo são aplicáveis, mas as táticas e as ferramentas devem ser adaptadas para se conformarem à natureza especial das WebApps e dos aplicativos móveis.

Quatro pontos [Dar99] devem ser considerados no desenvolvimento de táticas para gestão de configuração de WebApp.

Conteúdo. Uma WebApp típica contém um vasto conjunto de conteúdo – texto, gráficos, applets, scripts, arquivos de áudio/vídeo, formulários, elementos de página ativos, tabelas, dados encadeados e muitos outros. O desafio é organizar esse mar de conteúdo em um conjunto racional de objetos de configuração (Seção 29.1.4) e, então, estabelecer mecanismos de controle de configuração apropriados para esses objetos. Uma abordagem é modelar o conteúdo da WebApp usando técnicas convencionais de modelagem de dados [Wik12], anexando um conjunto de propriedades especializadas a cada objeto. A natureza estática/dinâmica de cada objeto e sua longevidade projetada (por exemplo, objeto temporário, de existência fixa ou permanente) são exemplos de propriedades necessárias para estabelecer uma abordagem de SCM eficaz. Por exemplo, se um item de conteúdo é alterado a cada hora, ele tem longevidade temporária. Os mecanismos de controle para esse item seriam diferentes (menos formais) daqueles aplicados a um componente de formulários, que é um objeto permanente.

Que impacto uma alteração não controlada tem sobre uma WebApp?

Como determinar quem tem a responsabilidade pela CM da WebApp?

Pessoas. Como uma porcentagem significativa do desenvolvimento de WebApp continua a ser executada de maneira improvisada, qualquer pessoa envolvida na WebApp pode criar conteúdo (e frequentemente o faz). Muitos criadores de conteúdo não possuem conhecimentos de engenharia de software e ignoram completamente a necessidade de gestão de configuração. Consequentemente, a aplicação cresce e é alterada de maneira não controlada.

Escalabilidade. As técnicas e controles aplicados a uma pequena WebApp não são bem escaláveis. Não é raro uma WebApp simples crescer significativamente, enquanto são implementadas interconexões com sistemas de informação, bancos de dados, armazém de dados e gateways de portais existentes. À medida que o tamanho e a complexidade crescem, pequenas mudanças podem ter efeitos amplos e inesperados que podem se tornar problemáticos. Portanto, o rigor dos mecanismos de configuração deve ser diretamente proporcional à escala de aplicação.

Políticas. Quem é o “dono” de uma WebApp? Essa é uma pergunta feita em empresas grandes e pequenas, e sua resposta tem um impacto significativo sobre as atividades de gerenciamento e controle. As perguntas a seguir [Dar99] ajudam uma equipe de software a entender as políticas associadas à engenharia para Web: Quem assume a responsabilidade pela exatidão das informações no site? Quem garante que os processos de controle de qualidade foram obedecidos antes que as informações fossem publicadas no site? Quem é responsável por fazer alterações? Quem assume o custo da alteração? As respostas a essas perguntas ajudam a determinar as pessoas na organização que devem adotar um processo de gestão de configuração para WebApps.

As técnicas de SCM para aplicativos móveis adotam muitos dos princípios aplicados ao desenvolvimento de software ágil. Além das tarefas SCM convencionais, discutidas anteriormente neste capítulo, a gestão de alterações para aplicativos móveis deve considerar também as implicações na segurança de cada alteração e seu impacto em uma base de usuários ampla, operando em um ambiente de plataforma diversificado.

O crescimento impressionante das lojas de aplicativos mudou o modo de distribuir software móvel. Alterações feitas em um aplicativo específico podem ser promulgadas amplamente em questão de dias, exigindo, assim, uma análise muito cuidadosa do impacto entre as plataformas, antes que uma nova versão de um aplicativo móvel seja colocada em uma loja de aplicativos para distribuição.

Em muitos casos, um processo de SCM convencional pode ser muito complicado para WebApps e alguns aplicativos móveis, mas na década passada surgiu uma nova geração de *ferramentas de gestão de conteúdo* especificamente projetadas para essas áreas de aplicação. Essas ferramentas estabelecem um processo que adquire as informações existentes (objetos de conteúdo), gerencia as alterações nos objetos, estrutura essas alterações para que possam ser apresentadas a um usuário e as apresenta no ambiente do lado do cliente para ser exibidas.

29.4.2 Objetos de configuração

As WebApps e os aplicativos móveis abrangem uma grande variedade de objetos de configuração – objetos de conteúdo (por exemplo, texto, gráficos, ima-

gens, vídeo e áudio), componentes funcionais (por exemplo, scripts, applets) e objetos de interface (por exemplo, COM ou CORBA para WebApps). Os objetos podem ser identificados (receber nomes de arquivo) de qualquer forma que seja apropriada para a organização.

Todo conteúdo tem formato e estrutura. Os formatos de arquivos internos são ditados pelo ambiente de computação no qual o conteúdo está armazenado. No entanto, o *formato de renderização* (muitas vezes chamado de *formato de exibição*) é definido pelo estilo estético e regras de design estabelecidas para WebApps ou aplicativos móveis. A *estrutura de conteúdo* define uma arquitetura de conteúdo; ela define a maneira pela qual são montados os objetos de conteúdo para apresentar informações claras ao usuário. Boiko [Boi04] define estrutura como “mapas que você coloca sobre um conjunto de conteúdo lobjetos para organizá-los e torná-los acessíveis às pessoas que precisam deles”.

29.4.3 Gestão de conteúdo

A *gestão de conteúdo* está relacionada à gestão de configuração no sentido de que um sistema de gestão de conteúdo (CMS) estabelece um processo (suportado por ferramentas apropriadas) que adquire o conteúdo existente (de uma ampla variedade de objetos de configuração de WebApps e/ou aplicativos móveis), estrutura esse conteúdo de maneira que ele possa ser apresentado a um usuário e, então, fornece-o ao ambiente no lado do cliente para ser exibido.

O uso mais comum de um sistema de gestão de conteúdo ocorre quando é criada uma aplicação dinâmica. WebApps ou aplicativos móveis dinâmicos criam páginas “dinamicamente”. Isto é, o usuário tipicamente consulta a aplicação solicitando informações específicas. A aplicação consulta um banco de dados no lado do servidor, formata as informações corretamente e as apresenta ao usuário. Por exemplo, uma loja de músicas (por exemplo, Apple iTunes) tem à venda centenas de milhares de faixas. Quando o usuário solicita uma faixa musical, um banco de dados é consultado, e uma variedade de informações sobre o artista – o CD (por exemplo, sua imagem ou elementos gráficos), o conteúdo musical e uma amostra de áudio – é baixado e configurado em um modelo de conteúdo padrão. A página resultante é criada no lado do servidor e passada para o lado do cliente para ser examinada pelo usuário. Uma representação genérica para WebApps aparece na Figura 29.6.

No sentido mais geral, um CMS “configura” conteúdo para o usuário por meio da invocação de três subsistemas integrados: um subsistema de coleta, um subsistema de gestão e um subsistema de publicação [Boi04].

O subsistema de coleta. O conteúdo é extraído dos dados e de informações que devem ser criados ou adquiridos por um gerador de conteúdo. O *subsistema de coleta* abrange todas as ações necessárias para criar e/ou adquirir conteúdo e as funções técnicas necessárias para (1) converter conteúdo de maneira que possa ser representado por uma linguagem de marcação (por exemplo, HTML, XML) e (2) organiza o conteúdo em pacotes que podem ser efetivamente mostrados no lado do cliente.

Criação e aquisição de conteúdo (também chamada de *autoria*) ocorrem, muitas vezes, em paralelo com outras atividades de desenvolvimento e em ge-

“Gestão de conteúdo é um antídoto para o emaranhado de informações de hoje.”

Bob Boiko

O subsistema de coleta abrange todas as ações necessárias para adquirir e/ou converter conteúdo de maneira que possa ser apresentado no lado do cliente.

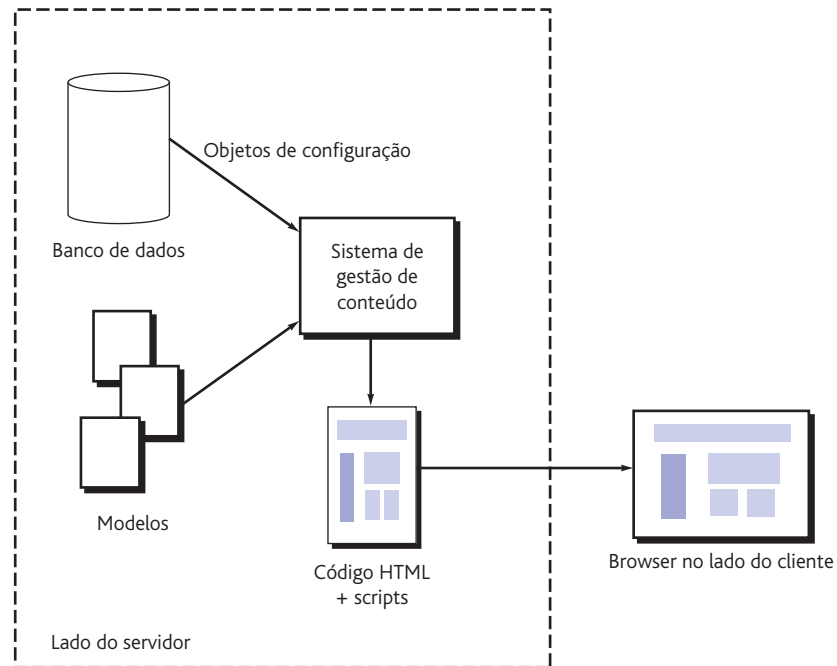


FIGURA 29.6 Sistema de gestão de conteúdo.

ral é conduzido por criadores de conteúdo não técnicos. Essa atividade combina elementos de criatividade e pesquisa e é suportada por ferramentas que permitem ao autor do conteúdo caracterizá-lo de modo que possa ser padronizado para uso dentro da WebApp ou do aplicativo móvel.

Uma vez que o conteúdo exista, ele deve ser convertido para se adaptar aos requisitos de um CMS. Isso implica retirar do conteúdo bruto quaisquer informações desnecessárias (por exemplo, representações gráficas redundantes), formatar o conteúdo para se adaptar aos requisitos do CMS e mapear os resultados em uma estrutura de informações que permita que seja gerenciado e publicado.

O subsistema de gestão implementa um repositório para todo o conteúdo. A gestão de configuração é executada nesse subsistema.

O subsistema de gestão. Uma vez que o conteúdo exista, ele deve ser armazenado em um repositório, catalogado para aquisição e uso subsequente e rotulado para definir (1) o status atual (por exemplo, o objeto de conteúdo está completo ou em desenvolvimento?), (2) a versão apropriada do objeto de conteúdo e (3) os objetos de conteúdo relacionados. Portanto, o *subsistema de gestão* implementa um repositório que abrange os seguintes elementos:

- *Banco de dados de conteúdo* – a estrutura de informações estabelecida para armazenar todos os objetos de conteúdo.
- *Recursos de banco de dados* – funções que permitem ao CMS pesquisar objetos de conteúdo específicos (ou categorias de objetos), armazenar e recuperar objetos e gerenciar a estrutura de arquivos estabelecida para o conteúdo.
- *Funções de gestão de configuração* – os elementos funcionais e o fluxo de trabalho associado que suporta identificação do objeto de conteúdo, controle de versão, gestão de alterações, gestão de auditoria e relatos.

Além desses elementos, o subsistema de gestão implementa uma função de administração que abrange os metadados e as regras que controlam a estrutura global do conteúdo, e a maneira pela qual ele é suportado.

O subsistema de publicação. O conteúdo deve ser extraído de um repositório, convertido para uma forma conveniente para a publicação e formatado de maneira que possa ser transmitido aos navegadores do lado do cliente. O subsistema de publicação executa essas tarefas usando uma série de modelos. Cada *modelo* é uma função que cria uma publicação por meio de um dentre três componentes diferentes [Boi04]:

- *Elementos estáticos* – texto, gráficos, mídia e scripts que não exigem outros processamentos são transmitidos diretamente para o lado do cliente.
- *Serviços de publicação* – chamadas de função para serviços específicos de acesso e formatação que personalizam o conteúdo (usando regras predefinidas), executam a conversão dos dados e criam links de navegação apropriados.
- *Serviços externos* – fornecem acesso à infraestrutura de informação corporativa externa como, por exemplo, aplicações de dados ou de retaguarda da empresa.

Um subsistema de gestão de conteúdo que abrange cada um desses subsistemas é aplicável à maior parte dos projetos para Web ou móveis. No entanto, a filosofia e funcionalidade básicas associadas a um CMS são aplicáveis a todas as aplicações dinâmicas.

O subsistema de publicação extrai o conteúdo do repositório e fornece-o aos navegadores do lado do cliente.

FERRAMENTAS DO SOFTWARE



Gestão de conteúdo

Objetivo: ajudar os engenheiros de software e criadores de conteúdo na gestão de conteúdo incorporado nas WebApps.

Mecanismos: ferramentas nesta categoria permitem aos engenheiros da Web e criadores de conteúdo atualizar o conteúdo da WebApp de forma controlada. Muitos estabelecem um sistema simples de gestão de arquivo que atribui permissões de atualizações, página por página, para vários tipos de conteúdo de WebApp. Outros mantêm um sistema de controle de versões para que uma versão anterior de um conteúdo possa ser arquivada para fins históricos.

Ferramentas representativas:¹⁰

Open Text Web Experience Management, desenvolvida pela Vignette ([http://www.opentext.com/global/products/web-content-management/web-experience-management/opentext-web-experience-manage-](http://www.opentext.com/global/products/web-content-management/web-experience-management/opentext-web-experience-manage-ment.htm)

[ment.htm](http://www.opentext.com/global/products/web-content-management/web-experience-management/opentext-web-experience-manage-ment.htm)), é um conjunto de ferramentas de gestão de conteúdo empresarial.

ektron-CMS300, desenvolvida pela ektron (www.ektron.com), é um conjunto de ferramentas que fornece recursos de gestão de conteúdo e ferramentas de desenvolvimento para Web.

OmniUpdate, desenvolvida pela WebsiteASP, Inc. (www.omniupdate.com), é uma ferramenta que permite aos provedores de conteúdo autorizados desenvolverem atualizações controladas para conteúdo específico de WebApp.

Informações adicionais sobre SCM e ferramentas de gestão de conteúdo para engenharia para Web podem ser encontradas em um ou mais dos seguintes sites: *WebDeveloper* (www.webdeveloper.com), *Developer Shed* (www.devshed.com), *webknowhow.net* (www.webknowhow.net) ou *WebReference* (www.webreference.com).

¹⁰ A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

29.4.4 Gestão de alterações

O fluxo de trabalho associado ao controle de alterações para software convencional (Seção 29.3.3) em geral é muito pesado para desenvolvimento de software para WebApps e aplicativos móveis. É pouco provável que a solicitação de alteração, o relato da alteração e a sequência de ordem de mudança de engenharia possam ser conseguidos de forma ágil que seja aceitável para muitos dos projetos de desenvolvimento de WebApps e aplicativos móveis. Como podemos, então, controlar um fluxo contínuo de alterações de conteúdo e funcionalidade solicitadas?

Para implementar um gerenciamento efetivo de alterações segundo a filosofia “codifique e vá em frente” que continua a dominar grande parte do desenvolvimento para a Web e móvel, o processo convencional de controle de alterações deve ser modificado. Cada alteração deve ser classificada em uma dentre quatro classes:

Classe 1 – alteração de conteúdo ou função que corrige um erro ou melhora o conteúdo ou a funcionalidade local.

Classe 2 – alteração de conteúdo ou função que tenha impacto sobre outros objetos de conteúdo ou sobre os componentes funcionais.

Classe 3 – alteração de conteúdo ou função que tenha um amplo impacto em uma aplicação (por exemplo, extensão ou funcionalidade principal, melhora significativa ou redução em conteúdo, alterações importantes necessárias na navegação).

Classe 4 – alteração importante de projeto (por exemplo, alteração na abordagem de projeto da interface ou na estratégia de navegação) que será notada imediatamente por uma ou mais categorias de usuário.

Estando a solicitação de alteração classificada, ela pode ser processada de acordo com o algoritmo mostrado na Figura 29.7 para WebApps, mas serve igualmente para aplicativos móveis.

De acordo com a figura, as alterações classe 1 e classe 2 são tratadas informalmente e manipuladas de modo ágil. Para uma alteração classe 1, você avaliaria o impacto da mudança, mas não é necessária nenhuma revisão externa ou documentação. À medida que a alteração é feita, os procedimentos-padrão de entrada (*check-in*) e saída (*check-out*) são apoiados por ferramentas de repositório de configuração. Para alterações classe 2, você deve revisar o impacto da alteração sobre objetos relacionados (ou pedir a outros desenvolvedores responsáveis por aqueles objetos que o façam). Se a alteração pode ser feita sem necessidade de alterações significativas em outros objetos, a modificação ocorre sem revisão ou documentação adicional. Se forem necessárias alterações substanciais, mais avaliação e planejamento serão exigidos.

Alterações classe 3 e 4 também são tratadas de forma ágil, mas é necessária alguma documentação descritiva e procedimentos de revisão mais formais. Para as alterações classe 3, é desenvolvida uma *descrição de alteração* – descrevendo a alteração e fornecendo uma breve avaliação do seu impacto. A descrição é distribuída a todos os membros da equipe que a examinam para melhor avaliar seu impacto. É desenvolvida também uma descrição de alteração para as alterações classe 4, mas, nesse caso, a revisão é conduzida por todos os envolvidos.

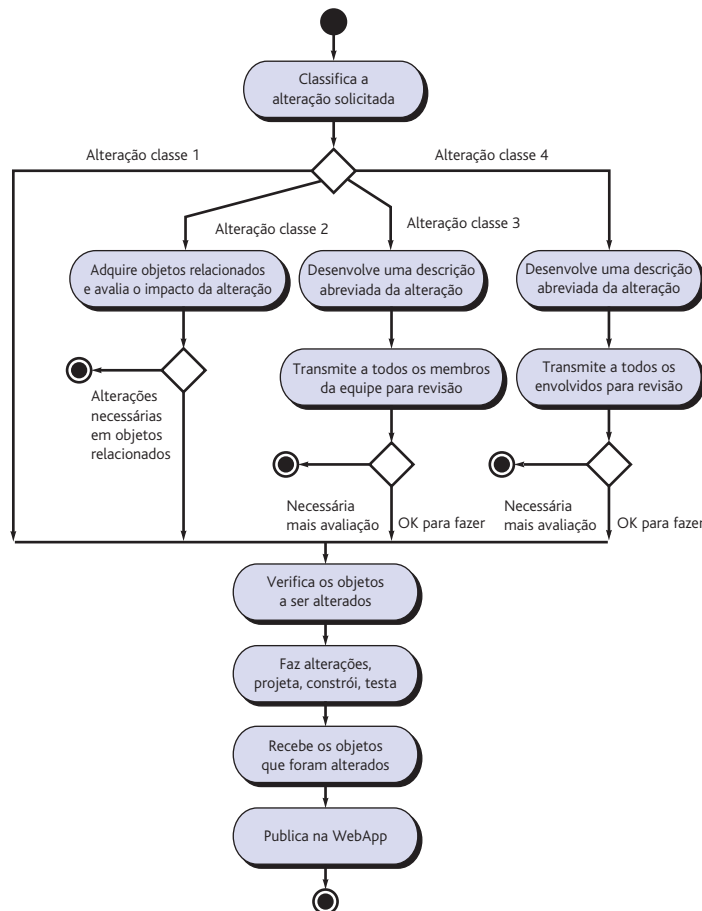


FIGURA 29.7 Gestão de alterações para WebApps.

FERRAMENTAS DO SOFTWARE



Gestão de alterações

Objetivo: ajudar os projetistas da Web e criadores de conteúdo na gestão de alterações à medida que elas são feitas nos objetos de configuração para WebApp.

Mecanismos: as ferramentas desta categoria foram desenvolvidas originalmente para software convencional, mas podem ser adaptadas para ser usadas pelos engenheiros da Web e criadores de conteúdo para fazer alterações controladas nas WebApps. Elas suportam entrar e sair automaticamente, controlar versão e desfazer alterações (*rollback*), relatos e outras funções da SCM.

Ferramentas representativas:¹¹

Dimension CM, desenvolvida pela Serena (<http://www.sere-na.com/index.php/en/products/dimensions-cm/>), é um conjunto de ferramentas de gestão de alterações que fornece recursos de SCM completos.

ClearCase, desenvolvida pela Rational (<http://www-03.ibm.com/software/products/us/en/clearcase>), é um conjunto de ferramentas que proporcionam recursos completos de gestão de configuração para WebApps.

PTC Integrity, desenvolvida pela PTC (<http://www.mks.com/platform/our-product>), é uma ferramenta de SCM que pode ser integrada a ambientes de desenvolvimento selecionados.

¹¹ A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

29.4.5 Controle de versão

À medida que uma WebApp ou um aplicativo móvel evoluem com uma série de incrementos, pode acontecer de várias versões diferentes existirem ao mesmo tempo. Uma versão (a aplicação operacional atual) está disponível na Internet para os usuários; outra versão (o próximo incremento da aplicação) pode estar nos estágios finais de teste antes da distribuição/instalação; uma terceira versão está em desenvolvimento e representa uma grande atualização em conteúdo, estética e funcionalidade. Objetos de configuração devem ser claramente definidos para que cada um possa ser associado à versão apropriada. Além disso, devem ser estabelecidos mecanismos de controle. Dreilinger [Dre99] discute a importância do controle de versão (e alteração) quando escreve:

Em um site *não controlado*, no qual vários autores têm acesso para editar e contribuir, surge o potencial para conflitos e problemas – mais ainda quando esses autores trabalham em locais diferentes, em diferentes horários do dia e da noite. Você pode passar o dia melhorando o arquivo *index.html* para um cliente. Depois que você fez as suas alterações, outro desenvolvedor que trabalha em casa após o horário comercial ou em outro escritório pode gastar a noite fazendo o upload de sua própria nova versão revisada do arquivo *index.html*, sobrescrevendo completamente o seu trabalho sem maneira de recuperá-lo!

É provável que você já tenha passado por uma situação semelhante. Para evitar isso, é necessário um processo de controle de versão.

1. *Um repositório central para o projeto de WebApp ou aplicativo móvel deve ser estabelecido.* O repositório terá as versões atuais de todos os objetos de configuração (conteúdo, componentes funcionais e outros).
2. *Cada projetista Web cria sua própria pasta de trabalho.* A pasta contém os objetos que estão sendo criados ou alterados em determinado instante.
3. *Os relógios das estações de trabalho de todos os desenvolvedores devem estar sincronizados.* Isso é feito para evitar conflitos de sobrescrita, quando dois desenvolvedores fazem alterações em horários muito próximos.
4. *À medida que novos objetos de configuração são desenvolvidos ou objetos existentes são alterados, são importados para o repositório central.* A ferramenta de controle de versão (veja discussão sobre CVS no quadro) vai gerenciar todas as funções de *check-in* (entrada) e *check-out* (saída) das pastas de trabalho de cada desenvolvedor. A ferramenta também fornecerá atualizações automáticas de e-mail a todas as partes envolvidas quando forem feitas alterações no repositório.
5. *À medida que objetos são importados ou exportados do repositório, é gerada uma mensagem automática com data e hora.* Isso proporciona informações úteis para auditoria e pode se tornar parte de um esquema eficaz de relatórios.

A ferramenta de controle de versão mantém diferentes versões da aplicação e pode reverter para uma versão mais antiga, se necessário.

29.4.6 Auditoria e relatório

Para fins de agilidade, as funções de auditoria e de relatório não são enfatizadas durante o desenvolvimento de WebApps ou aplicativos móveis.¹² Contudo, não são completamente eliminadas. Todos os objetos que entram (*check-in*) ou saem (*check-out*) do repositório são registrados em um log (registro) que pode ser revisto quando desejado. Pode-se criar um relatório completo, de forma que todos os membros da equipe tenham uma cronologia das alterações durante um período definido. Além disso, uma notificação automática enviada por email (e endereçada a todos os desenvolvedores e envolvidos que a desejem) pode ser enviada sempre que um objeto entra ou sai do repositório.

INFORMAÇÕES



Normas de SCM

A seguir é apresentada uma lista de normas de SCM (extraída em parte do site www.12207.com) razoavelmente abrangente:

Padrões do IEEE

IEEE 828 Software Configuration Management Plans

IEEE 1042 Software Configuration Management

Padrões ISO

ISO 10007-1995 Quality Management, Guidance for CM

ISO/IEC 12207 Information Technology-Software Life Cycle Processes

ISO/IEC TR 15271 Guide for ISO/IEC 12207

ISO/IEC TR 15846 Software Engineering-Software Life Cycle Process-Configuration Management for Software Order

Padrões EIA

EIA 649 National Consensus Standard for Configuration Management

EIA CMB4-1A Configuration Management Definitions for Digital Computer Programs

EIA CMB4-2 Configuration Identification for Digital Computer Programs

EIA CMB4-3 Computer Software Libraries

EIA CMB4-4 Configuration Change Control for Digital Computer Programs

EIA CMB6-1C Configuration and Data Management References Order

EIA CMB6-3 Configuration Identification

EIA CMB6-4 Configuration Control

EIA CMB6-5 Textbook for Configuration Status Accounting

EIA CMB7-1 Electronic Interchange of Configuration Management Data

Padrões militares dos EUA

DoD MIL STD-973 Configuration Management

MIL-HDBK-61 Configuration Management Guidance

Outros padrões

DO-178B Guidelines for the Development of Aviation Software

ESA PSS-05-09 Guide to Software Configuration Management

AECL CE-1001-STD Standard for Software Engineering rev.1 of Safety Critical Software

Checklist DOE SCM: <http://energy.gov/cio/downloads/software-quality-systems-engineering-program-software-configuration-management>

BS-6488 British Std., Configuration Management of Computer-Based Systems

¹² Isso está começando a mudar. Há uma ênfase cada vez maior no SCM como elemento de segurança da aplicação [Sar06]. Fornecendo um mecanismo para rastrear e relatar todas as alterações feitas em cada objeto da aplicação, uma ferramenta de gestão de alterações pode proporcionar uma valiosa proteção contra alterações mal-intencionadas.

Best Practice—UK

Office of Government Commerce:
<http://www.cabinetoffice.gov.uk/content/office-government-commerce-ogc>

CMII

Institute of CM Best Practices: www.icmhq.com

Um guia de recursos de gestão de configuração (*Configuration Management Resource Guide*) fornece informações complementares para os envolvidos nos processos e prática de gestão de alterações (CM). Ele se encontra disponível em <http://cmpic.com/cmresourceguide.htm>.

29.5 Resumo

A gestão de configuração de software (SCM) é uma atividade de apoio aplicada ao processo de software inteiro. Ela identifica, controla, faz auditoria e relata modificações que invariavelmente ocorrem enquanto o software está em desenvolvimento e depois que foi entregue ao cliente. Todos os artefatos criados como parte da engenharia de software tornam-se parte de uma configuração de software. A configuração é organizada de modo a permitir o controle ordenado das alterações.

A configuração de software é composta por objetos inter-relacionados, também chamados de itens de configuração de software (SCIs), produzidos como resultado de alguma atividade de engenharia de software. Além dos artefatos de engenharia de software, o ambiente de desenvolvimento usado para criar software também pode ser colocado sob controle de configuração. Todas os SCIs são armazenados em um repositório que implementa vários mecanismos e estruturas de dados para garantir a integridade dos dados, proporcionar suporte de integração para outras ferramentas de software, suportar compartilhamento de informações entre todos os membros da equipe de software e implementar funções no suporte do controle de versão e alteração.

Uma vez desenvolvido e revisado um objeto de configuração, ele se torna uma referência. Alterações em um objeto referencial resultam na criação de uma nova versão daquele objeto. A evolução de um programa pode ser acompanhada examinando-se o histórico de revisão de todos os objetos de configuração. O controle de versão é uma série de procedimentos e ferramentas para gerenciar o uso desses objetos.

O controle de alteração é uma atividade procedimental que garante qualidade e consistência quando são feitas alterações em um objeto de configuração. O processo de controle de alterações começa com uma solicitação de alteração, leva a uma decisão sobre fazer ou rejeitar a solicitação de alteração e culmina com uma atualização controlada do SCI que deve ser alterado.

A auditoria de configuração é uma atividade de SQA que ajuda a garantir que a qualidade seja mantida quando são feitas alterações. Os relatórios de status fornecem informações sobre cada alteração para aqueles que precisam ter conhecimento do assunto.

A gestão de configuração para WebApps e aplicativos móveis é semelhante em muitos aspectos à de SCM para software convencional. No entanto, todas as tarefas centrais de SCM devem ser agilizadas para torná-la o mais leve possível, e provisões especiais para gestão de conteúdo devem ser implementadas.

Problemas e pontos a ponderar

- 29.1** Por que a Primeira Lei da Engenharia de Software é verdadeira? Dê exemplos específicos para cada uma das quatro razões fundamentais para alterações.
- 29.2** Quais são os quatro elementos que existem quando é implementado um sistema de SCM eficaz? Discuta cada um sucintamente.
- 29.3** Discuta as razões para referenciais com suas próprias palavras.
- 29.4** Suponha que você seja o gerente de um pequeno projeto. Que referenciais definiria para o projeto e como os controlaria?
- 29.5** Desenvolva um sistema de banco de dados de projeto (repositório) que permitiria a um engenheiro de software armazenar, estabelecer referências cruzadas, acompanhar, atualizar, alterar etc. todos os itens importantes da configuração de software. Como o banco de dados trataria as diferentes versões do mesmo programa? O código-fonte seria tratado de forma diferente da documentação? Como dois desenvolvedores seriam impedidos de fazer alterações diferentes no mesmo SCI ao mesmo tempo?
- 29.6** Pesquise uma ferramenta de SCM existente e descreva como ela implementa o controle para versões, variantes e objetos de configuração em geral.
- 29.7** As relações <part-of> e <interrelated> representam relações simples entre objetos de configuração. Descreva cinco relações adicionais que podem ser úteis no contexto de um repositório de SCM.
- 29.8** Pesquise uma ferramenta de SCM existente e descreva como ela implementa o mecanismo do controle de versão. Como alternativa, leia duas ou três publicações sobre SCM e descreva as diferentes estruturas de dados e mecanismos de referência usados para o controle de versão.
- 29.9** Desenvolva uma *checklist* para usar durante as auditorias de configuração.
- 29.10** Qual é a diferença entre uma auditoria de SCM e uma revisão técnica? Suas funções podem ser incluídas em uma revisão? Quais são os prós e os contras?
- 29.11** Descreva sucintamente as diferenças entre a SCM para software convencional e a SCM para WebApps e aplicativos móveis.
- 29.12** O que é gestão de conteúdo? Use a Web para pesquisar as características de uma ferramenta de gestão de conteúdo e forneça um breve resumo.

Leituras e fontes de informação complementares

Dentre as ofertas mais recentes relacionadas a SCM estão Aiello e seus colegas (*Configuration Management Best Practices: Practical Methods That Work in the Real World*, Addison-Wesley, 2010), Moreira (*Adapting Configuration Management for Agile Teams: Balancing Sustainability and Speed*, Wiley, 2009), Duvall e seus colegas (*Continuous Integration: Improving Software Quality and Reducing Risk*, Addison-Wesley, 2007), Leon (*Software Configuration Management Handbook*, 2ª ed., Artech House Publishers, 2005), Maraia (*The Build Master: Microsoft's Software Configuration Management Best Practices*, Addison-Wesley, 2005), Keyes (*Software Configuration Management*, Auerbach, 2004) e Hass (*Configuration Management Principles and Practice*, Addison-Wesley, 2002). Cada um desses livros apresenta todo o processo de SCM com detalhes substanciais. Moreira (*Software Configuration Management Implementation Roadmap*, Wiley, 2004) apresenta um guia prático para quem precisa implementar SCM em uma organização. Lyon (*Practical CM III: Best Practices for the 21st Century*, Raven Publishing, 2013, disponível em www.configuration.org) escreveu um guia abrangente para o profissional de

CM, incluindo diretrizes pragmáticas para implementar todos os aspectos de um sistema de gestão de configuração (atualizado anualmente). Girod e Shpichko (*IBM Rational ClearCase 7.0: Master the Tools That Monitor, Analyze, and Manage Software Configurations*, Packt, 2011) e Bellagio e Mulligan apresentam a SCM no contexto de uma das ferramentas de SCM mais populares.

Berczuk e Appleton (*Software Configuration Management Patterns*, Addison-Wesley, 2003) mostram uma variedade de padrões úteis que ajudam a entender a SCM e a implementar sistemas de SCM eficazes. Brown *et al.* (*Anti-Patterns and Patterns in Software Configuration Management*, Wiley, 1999) discutem o que não se deve fazer (antipadrões) ao implementar um processo de SCM e, em seguida, tratam das soluções. Humble e Fowler (*Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, Addison-Wesley, 2010) e Bays (*Software Release Methodology*, Prentice Hall, 1999) se concentram na mecânica do “lançamento de produto bem-sucedido”, um importante complemento da SCM eficaz.

Conforme as WebApps se tornam mais dinâmicas, a gestão de conteúdo tem se tornado um tópico essencial para engenheiros da Web. Livros de Rockley e Cooper (*Managing Enterprise Content: A Unified Content Strategy*, 2ª ed., New Riders, 2012), Jenkins e seus colegas (*Managing Content in the Cloud-Enterprise Content Management 2.0*, Open Text Corporation, 2010) e (*Enterprise Content Management Methods*, Open Text Corporation, 2005), White (*The Content Management Handbook*, Curtin University Books, 2005), Boiko [Boi04], Mauthe e Thomas (*Professional Content Management Systems*, Wiley, 2004), Addey e seus colegas (*Content Management Systems*, Glasshaus, 2003), Hackos (*Content Management for Dynamic Web Delivery*, Wiley, 2002) e Nakano (*Web Content Management*, Addison-Wesley, 2001) apresentam tratados úteis sobre o tema.

Além das discussões genéricas sobre o assunto, Halvorson e Bach (*Content Strategy for the Web*, 2ª ed., New Riders, 2012), Hauschildt (*CMS Made Simple 1.6: Beginners' Guide*, Packt, 2010), Lim e seus colegas (*Enhancing Microsoft Content Management Server with ASP.NET 2.0*, Packt Publishing, 2006), Ferguson (*Creating Content Management Systems in Java*, Charles River Media, 2006), IBM Redbooks (*IBM Workplace Web Content Management for Portal 5.1 and IBM Workplace Web Content Management 2.5*, Vivante, 2006), Fritz e seus colegas (*Typo3: Enterprise Content Management*, Packt Publishing, 2005) e Forta (*Reality ColdFusion: Intranets and Content Management*, Pearson Education, 2002) abordam a gestão de conteúdo no contexto de ferramentas e linguagens específicas.

Uma ampla variedade de fontes de informação sobre gestão de configuração de software e gestão de conteúdo está disponível na Internet. Uma lista atualizada das referências da Web pode ser encontrada em “recursos de engenharia de software” no site da SEPA: www.mhhe.com/pressman.

Métricas de produto 30

Um elemento-chave de qualquer processo de engenharia é a medição. Você pode usar medidas para compreender melhor os atributos dos modelos criados e para avaliar a qualidade dos produtos ou sistemas construídos. Entretanto, ao contrário de outras disciplinas de engenharia, a engenharia de software não é fundamentada nas leis quantitativas da física. Medidas diretas, como tensão elétrica, massa, velocidade ou temperatura, não são comuns no mundo do software. Como medidas e métricas de software são, muitas vezes, indiretas, elas estão abertas ao debate. Fenton [Fen91] aborda esse assunto:

Medição é o processo pelo qual números ou símbolos são anexados aos atributos de entidades no mundo real para defini-los de acordo com regras claramente estabelecidas... Nas ciências físicas, medicina, economia e, mais recentemente, nas ciências sociais, podemos medir os atributos antes considerados incomensuráveis... É claro que essas medidas não são tão refinadas como muitas feitas nas ciências físicas..., mas elas existem e são tomadas decisões importantes baseadas

Conceitos-chave

indicador	655
medição	654
medida	654
meta/questão/métrica (GQM)	656
métricas	
aplicativos móveis	672
atributos de	657
código-fonte	675
de componentes	671
modelo de requisitos	659
modelo de projeto	663
projeto de interface de usuário	672
teste	676
WebApps	672

PANORAMA

O que é? Por sua natureza, a engenharia é uma disciplina quantitativa. A métrica de produto ajuda os engenheiros de software a visualizar o projeto e a construção do software, focando nos atributos específicos e mensuráveis dos artefatos da engenharia de software.

Quem realiza? Os engenheiros de software usam métricas de produto como apoio para criar software de mais alta qualidade.

Por que é importante? Sempre haverá um elemento qualitativo na criação de software. O problema é que a avaliação qualitativa pode não ser suficiente. É preciso ter critérios objetivos para ajudar a direcionar o projeto de dados, arquitetura, interfaces e componentes. Ao testar, precisamos de orientação quantitativa, que nos auxiliará na seleção de casos de teste e seus objetivos. A métrica de produto proporciona uma base por meio da qual análise, projeto, codificação e teste podem ser conduzidos mais objetivamente e avaliados de maneira mais quantitativa.

Quais são as etapas envolvidas? O primeiro passo no processo de medição é derivar as medições de software e as métricas adequadas para a representação do software. Depois, os dados necessários são coletados para derivar as métricas formuladas. Uma vez calculadas, as métricas apropriadas são analisadas com base em diretrizes preestabelecidas e dados do passado. Os resultados das análises são interpretados para obter informações sobre a qualidade do software e os dados da interpretação levam à modificação dos requisitos e modelos de projeto, código-fonte ou casos de teste. Em alguns casos, pode também levar à modificação do próprio processo de software.

Qual é o artefato? As métricas de produto calculadas por meio de dados coletados dos requisitos e modelos de projeto, código-fonte e casos de teste.

Como garantir que o trabalho foi realizado corretamente? Devemos estabelecer os objetivos da medição antes de iniciarmos a coleta de dados, definindo cada métrica de produto de maneira não ambígua. Defina apenas algumas métricas e então as use para obter informações sobre a qualidade de um artefato de software.

métricas de projeto de arquitetura	663
métricas orientadas a classes	667
métricas para projeto orientado a objetos	666
ponto de função (FP) ...	659
princípios da medição ..	656

nelas! Percebemos que a obrigação de tentar “medir o incomensurável” para melhorar nossa compreensão de entidades particulares é tão poderosa na engenharia de software quanto em qualquer outra disciplina.

No entanto, alguns membros da comunidade de software continuam a argumentar que o software é “incomensurável” ou que tentativas de medição devem ser adiadas até entendermos melhor o software e os atributos a serem usados para descrevê-lo. Isso é um erro.

Embora as métricas de produto para programas de computadores sejam imperfeitas, elas podem ser uma maneira sistemática de avaliar a qualidade com base em um conjunto de regras claramente definidas. Elas também proporcionam uma visão objetiva, que “vai direto ao ponto”, e não “após o fato”. Isso permite descobrir e corrigir problemas em potencial antes que se tornem defeitos catastróficos.

Neste capítulo, apresentamos medidas que podem ser usadas para avaliar a qualidade do produto enquanto ele está sendo projetado. Essas medidas de atributos internos do produto fornecem uma indicação em tempo real da eficácia dos modelos de requisitos, do projeto e do código; da eficiência dos casos de teste; e da qualidade geral do software que será criado.

30.1 Framework para métricas de produto

“Uma ciência é tão desenvolvida quanto suas ferramentas de medição.”

Louis Pasteur

A medição especifica números ou símbolos a atributos de entidades do mundo real. Para tanto, é necessário um modelo de medição abrangendo um conjunto coerente de regras. Embora a teoria da medição (por exemplo, [Kyb84]) e sua aplicação a programas de computadores (por exemplo, [Zus97]) sejam tópicos que estão além dos objetivos deste livro, vale estabelecer uma estrutura fundamental e um conjunto de princípios básicos que orientem a definição de métricas de produto para software.

30.1.1 Medidas, métricas e indicadores

Qual é a diferença entre medida e métrica?

Embora os termos *medida*, *medição* e *métricas* sejam, com frequência, usados indistintamente, é importante notar as diferenças sutis entre eles. Pelo fato de que *medida* pode ser usada como substantivo ou como verbo, as definições do termo podem se tornar confusas. No contexto da engenharia de software, uma *medida* fornece uma indicação quantitativa da extensão, quantidade, capacidade ou tamanho de algum atributo de um produto ou processo. *Medição* é o ato de determinar uma medida. O *IEEE Standard Glossary of Software Engineering Terminology* [IEE93b] define *métrica* como “uma medida quantitativa do grau com o qual um sistema, componente ou processo possui determinado atributo”.

Quando um único ponto de dado é coletado (por exemplo, o número de erros descobertos em um componente de software), foi estabelecida uma medida. A medição ocorre como resultado da coleta de um ou mais pontos de dados (por exemplo, um conjunto de revisões de componente e testes de unidade são investigados para coletar medidas do número de erros para cada um).

Uma métrica de software relaciona as medidas individuais de alguma maneira (por exemplo, o número médio de erros encontrados por revisão ou o número médio de erros encontrados por teste de unidade).

Um engenheiro de software coleta medidas e desenvolve métricas para obter indicadores. Um *indicador* é uma métrica ou combinação de métricas que fornecem informações sobre o processo de software, em um projeto de software ou no próprio produto. Um indicador fornece informações que permitem ao gerente de projeto ou aos engenheiros de software ajustar o processo, o projeto ou o produto para incluir melhorias.

Um indicador é uma métrica ou métricas que proporcionam uma visão do processo, do produto ou do projeto.

30.1.2 O desafio das métricas de produto

Durante as últimas quatro décadas, muitos pesquisadores tentaram desenvolver uma métrica única que fornecesse uma medida abrangente da complexidade do software. Fenton [Fen94] caracteriza essa pesquisa como uma busca pelo “Santo Graal impossível”. Embora tenham sido propostas dezenas de medidas de complexidade [Zus90], cada uma delas tem uma visão diferente do que é complexidade e quais são os atributos de um sistema que levam à complexidade. Por analogia, considere uma métrica para avaliar um carro de luxo. Alguns podem destacar o projeto do chassi; outros podem destacar as características mecânicas; outros ainda podem destacar o custo ou desempenho ou o uso de combustíveis alternativos ou a facilidade de reciclagem quando o carro já estiver inutilizável. Como cada uma dessas características pode “brigar” com as outras, é difícil derivar um valor único para “atraente”. O mesmo problema ocorre com o software.

Ainda assim, a necessidade de medir e controlar a complexidade do software existe. E, se é difícil obter um valor único dessa métrica de qualidade, deve ser possível desenvolver medidas de diferentes atributos internos de programa (por exemplo, modularidade efetiva, independência funcional e outros discutidos no Capítulo 12). Essas medidas e métricas derivadas dos atributos podem ser usadas como indicadores, independentemente da qualidade dos modelos de requisitos e projeto. Mas aqui também temos problemas. Fenton [Fen94] observa isso quando diz: “O perigo de tentar encontrar medidas que caracterizem tantos atributos diferentes é que inevitavelmente as medidas têm de satisfazer interesses em conflito. Isso é contrário à teoria representacional da medição”. Embora a afirmação de Fenton esteja correta, muitos argumentam que a medição de produto executada durante os primeiros estágios do processo de software fornece aos engenheiros um mecanismo consistente e objetivo para avaliar a qualidade.¹

Volumosas informações sobre métricas de produto foram compiladas pelo Departamento de Segurança Interna dos Estados Unidos em <https://buildsecurityin.us-cert.gov/bsi/articles/best-practices/measurement.html>.

¹ Embora seja comum na literatura a crítica a métricas específicas, muitas críticas focam-se em aspectos herméticos e perdem o objetivo principal das métricas no mundo real: ajudar o engenheiro de software a estabelecer uma maneira sistemática e objetiva de visualizar seu trabalho e melhorar a qualidade do produto como resultado.

"Assim como a medição da temperatura começou com um dedo indicador... e chegou até as escalas, instrumentos e técnicas sofisticadas, a medição de software também está evoluindo."

Shari Pfleeger

30.1.3 Princípios da medição

Antes de apresentarmos uma série de métricas de produto que (1) ajudam na avaliação dos modelos de análise e projeto, (2) proporcionam uma indicação da complexidade dos projetos procedimentais e código-fonte e (3) facilitam o projeto de testes mais eficazes, é importante entendermos os princípios básicos das medições. Roche [Roc94] sugere um processo de medição que pode ser caracterizado por cinco atividades: formulação, coleta, análise, interpretação e *feedback*. As métricas de software serão úteis somente se forem efetivamente caracterizadas e validadas de forma que demonstrem valer a pena. Os princípios a seguir [Let03b] são representativos de muitos que podem ser propostos para caracterização e validação da métrica:

- *Uma métrica deve ter as propriedades matemáticas desejadas.* O valor da métrica deve estar em um intervalo significativo (por exemplo, 0 a 1, onde 0 significa ausência, 1 indica o valor máximo e 0,5 representa o "ponto médio"). Além disso, uma métrica que deve estar em uma escala racional não deve ser composta de componentes medidos apenas em escala ordinal.
- *Quando uma métrica representa uma característica do software que aumenta na ocorrência de peculiaridades positivas ou diminui na ocorrência de peculiaridades indesejadas, o valor da métrica deve aumentar ou diminuir da mesma maneira.*
- *Cada métrica deve ser validada empiricamente em uma ampla variedade de contextos antes de ser publicada ou usada para tomada de decisões.* Uma métrica deve medir o fator de interesse, independentemente de outros fatores. Ela deve ser "ampliada" para sistemas grandes e funcionar em uma variedade de linguagens de programação e domínios de sistemas.

Embora a formulação, caracterização e validação sejam de extrema importância, a coleta e a análise são as atividades que regem o processo de medição. Roche [Roc94] sugere as seguintes diretrizes para essas atividades: (1) sempre que possível, a coleta e análise de dados deve ser automática; (2) devem ser aplicadas técnicas estatísticas válidas para estabelecer relações entre atributos internos de produto e características externas de qualidade (por exemplo, se o nível de complexidade da arquitetura está correlacionado ao número de defeitos relatados em uso); e (3) diretrizes e recomendações interpretativas devem ser estabelecidas para cada métrica.

30.1.4 Medição de software orientada a metas

O paradigma *Meta/Questão/Métrica* (GQM, *Goal/Question/Metric*) foi desenvolvido por Basili e Weiss [Bas84] como uma técnica para identificar métricas significativas para qualquer parte do processo de software. O GQM enfatiza a necessidade de (1) estabelecer uma *meta* de medição explícita específica para a atividade do processo ou característica de produto que deve ser avaliada, (2) definir um conjunto de *questões* que devem ser respondidas para atingir o objetivo e (3) identificar *métricas* bem formuladas que ajudem a responder a essas questões.

Uma discussão útil sobre GQM pode ser encontrada em https://www.thecsiac.com/resources/ref_documents/software-acquisition-gold-practice-goal-question-metric-gqm-approach.

Pode ser usado um *modelo de definição de meta* (*goal definition template*) [Bas94] para definir o objetivo de cada medição. O modelo toma a seguinte forma:

Analisar {o nome da atividade ou atributo a ser medido} **com a finalidade de** {o objetivo geral da análise}² **com relação a** {o aspecto da atividade ou atributo considerado} **do ponto de vista de** {a pessoa que tem o interesse na medição} **no contexto de** {o ambiente no qual a medição ocorre}.

Como exemplo, considere um modelo de definição de meta para o *CasaSegura*:

Analisar a arquitetura de software do *CasaSegura* **com a finalidade de** avaliar componentes da arquitetura **com relação a** habilidade em tornar o *CasaSegura* mais ampliável **do ponto de vista dos** engenheiros de software que estão executando o trabalho **no contexto de** aperfeiçoamento de produto durante os próximos três anos.

Com a meta da medição explicitamente definida, desenvolve-se uma série de questões. As respostas a essas questões ajudam a equipe de software (ou outros envolvidos) a determinar se o objetivo da medição foi atingido. Entre as questões que podem ser formuladas estão:

Q₁: Os componentes de arquitetura são caracterizados para distinguir função de dados relacionados?

Q₂: A complexidade de cada componente está dentro dos limites que facilitarão a modificação e extensão?

Cada uma dessas questões deve ser respondida quantitativamente, usando uma ou mais medidas e métricas. Por exemplo, uma métrica que proporciona uma indicação da coesão (Capítulo 12) de um componente de arquitetura pode ser útil na resposta de Q₁. Métricas discutidas mais adiante neste capítulo podem proporcionar uma visão para Q₂. Em todos os casos, as métricas escolhidas (ou derivadas) devem estar de acordo com os princípios de medição discutidos na Seção 30.1.3 e os atributos de medição discutidos na Seção 30.1.5.

30.1.5 Atributos de métricas de software eficazes

Centenas de métricas já foram propostas para programas de computadores, mas nem todas são práticas para o engenheiro de software. Algumas demandam medições muito complexas, outras são tão esotéricas que poucos profissionais do mundo real têm qualquer esperança de entendê-las, e outras ainda violam as noções intuitivas básicas do que é realmente um software de alta qualidade.

Ejiogu [Eji91] define um conjunto de atributos que devem ser abrangidos por métricas de software efetivas. Deve ser relativamente fácil aprender a derivar a métrica, e seu cálculo não deve exigir esforço ou tempo fora do normal. A métrica deve satisfazer as noções intuitivas do engenheiro sobre o atributo do produto considerado (por exemplo, uma métrica que mede coesão de módulos deve crescer em valor à medida que o nível de coesão aumenta). A métrica deve sempre produzir resultados que não sejam ambíguos. O cálculo

A experiência indica que a métrica de um produto será usada somente se ela for clara e fácil de calcular. Se forem necessárias dezenas de "contagens" e cálculos complexos, é pouco provável que seja amplamente adotada.

² Van Solingen e Berghout [Sol99] sugerem que o objetivo é quase sempre "entender, controlar ou improvisar" a atividade de processo ou atributo de produto.

matemático da métrica deve usar medidas que não resultem em combinações bizarras de unidades. Por exemplo, multiplicar o número de pessoas nas equipes de projeto pelas variáveis da linguagem de programação no programa resulta em uma mistura duvidosa de unidades que não é claramente convincente. As métricas devem ser baseadas no modelo de requisitos, no modelo de projeto ou na própria estrutura do programa. Elas não devem ser dependentes dos caprichos da sintaxe ou semântica das linguagens de programação. Por último, a métrica deve fornecer informações que podem levar a um produto de melhor qualidade.

Embora muitas métricas de software satisfaçam a todos esses atributos, algumas usadas comumente podem não satisfazer a um ou outro. Um exemplo é o ponto de função (FP, function point) (discutido na Seção 30.2.1) – uma medida da “funcionalidade” fornecida pelo software. Pode-se argumentar que um terceiro, independente, talvez não consiga derivar o mesmo valor ponto de função que um colega usando as mesmas informações sobre o software.³ Devemos, então, rejeitar a medida de ponto de função? A resposta, evidentemente, é “não!” A métrica FP proporciona informações úteis e valor distinto, mesmo que não satisfaça a um atributo perfeitamente.

CASASEGURA



Debate sobre métricas de produto

Cena: Sala do Vinod.

Atores: Vinod, Jamie e Ed – membros da equipe de engenharia de software do CasaSegura estão continuando o trabalho de projeto em nível de componente e projeto de casos de teste.

Conversa:

Vinod: Doug [Doug Miller, gerente de engenharia de software] me disse que todos nós devemos usar métricas de produto, mas ele foi um pouco vago. Ele falou também que não vai “forçar a barra”... usar ou não, depende de nós.

Jamie: Ótimo, porque não tenho como arranjar tempo para começar a medir coisas. Estamos sofrendo para manter o cronograma.

Ed: Concordo com o Jamie. Somos contra, aqui... não temos tempo.

Vinod: Sim, eu sei, mas provavelmente deve haver algum mérito em usar essas métricas.

Jamie: Não estou discutindo isso, Vinod, é questão de tempo... e nenhum de nós tem tempo livre.

Vinod: Mas e se a medição ajudar a poupar tempo?

Ed: Errado, demanda horas livres e, como disse o Jamie...

Vinod: Não, espere... e se isso nos ajudar a poupar tempo?

Jamie: Como?

Vinod: Com retrabalho. Se uma medição nos ajudar a evitar um problema maior ou mesmo moderado, e se isso evitar o retrabalho em uma parte do sistema, poupará tempo. Não?

Ed: É possível, acho, mas você pode garantir que alguma métrica de produto nos ajudará a encontrar um problema?

Vinod: Você pode garantir que não?

Jamie: Então o que você propõe?

Vinod: Acho que poderíamos selecionar algumas métricas de projeto, provavelmente orientadas a classes, e usá-las como parte de nosso processo de revisão para qualquer componente que desenvolvermos.

Ed: Não conheço métricas orientadas a classes.

Vinod: Vou verificar tudo isso e fazer uma recomendação... ok por vocês, pessoal?

[Ed e Jamie balançam a cabeça sem muito entusiasmo.]

³ Pode-se apresentar um contra-argumento igualmente forte: essa é a natureza da métrica de software.

30.2 Métricas para o modelo de requisitos

Na engenharia de software, o trabalho técnico começa com a criação do modelo de requisitos. É nesse estágio que os requisitos são formulados e uma base para o projeto é estabelecida. Portanto, métricas de produto que proporcionem informações sobre a qualidade do modelo de análise são desejáveis.

Embora relativamente poucas métricas de análise e especificação tenham aparecido na literatura, é possível adaptar as usadas frequentemente para estimativa de projeto e aplicá-las nesse contexto. Essas métricas examinam o modelo de requisitos com a intenção de prever o “tamanho” do sistema resultante. O tamanho é, às vezes (mas nem sempre), um indicador da complexidade do projeto e quase sempre é um indicador do trabalho cada vez maior de codificação, integração e teste.

30.2.1 Métricas baseadas em função

A *métrica ponto de função* (FP) pode ser usada como um meio para medir a funcionalidade fornecida por um sistema.⁴ Por meio de dados históricos, a métrica FP pode ser empregada para (1) estimar o custo ou trabalho necessário para projetar, codificar e testar o software; (2) prever o número de erros que serão encontrados durante o teste; e (3) prever o número de componentes e/ou o número de linhas projetadas de código-fonte no sistema implementado.

Pontos de função são derivados por meio de uma relação empírica baseada em medidas calculáveis (diretas) do domínio de informações do software e avaliações qualitativas da complexidade do software. Valores do domínio de informações são definidos da seguinte maneira:⁵

Número de entradas externas (EIs, external inputs). Cada *entrada externa* é originada de um usuário ou transmitida de outra aplicação e fornece dados distintos orientados a aplicação ou informações de controle. Entradas são muitas vezes usadas para atualizar *arquivos lógicos internos* (*internal logical files* – ILFs). As entradas devem ser diferenciadas das consultas, que são contadas separadamente.

Número de saídas externas (EOs, external outputs). Cada *saída externa* é formada por dados derivados da aplicação e fornece informações para o usuário. Nesse contexto, saída externa se refere a relatórios, telas, mensagens de erro etc. Itens individuais de dados em um relatório não são contados separadamente.

Número de consultas externas (EQs, external inquiries). Uma *consulta externa* é definida como uma entrada online que resulta na geração de al-

Muitas informações úteis sobre pontos de função podem ser obtidas em www.ifpug.org e <http://www.functionpoint.com/>.

⁴ Centenas de livros, dissertações e artigos já foram escritos sobre métricas FP. Uma boa bibliografia pode ser encontrada em [IFP05].

⁵ Atualmente, a definição dos valores do domínio de informações e a maneira pela qual elas são contadas é algo um pouco mais complexo. O leitor interessado deve consultar [IFP01] para mais detalhes.

Valor do Domínio de Informação	Contagem		Fator de peso				
			Simples	Média	Complexo		
Entradas Externas (EIs)		x	3	4	6	=	
Saídas Externas (EOs)		x	4	5	7	=	
Consultas Externas (EQs)		x	3	4	6	=	
Arquivos Lógicos Internos (ILFs)		x	7	10	15	=	
Arquivos de Interface Externos (EIFs)		x	5	7	10	=	
Contagem total							

FIGURA 30.1 Calculando pontos de função.

guma resposta imediata do software na forma de uma saída online (muitas vezes obtida de um ILF).

Número de arquivos lógicos internos (ILFs, internal logical files). Cada *arquivo lógico interno* é um agrupamento lógico de dados que reside dentro das fronteiras do aplicativo e é mantido por meio de entradas externas.

Número de arquivos de interface externos (EIFs, external interface files). Cada *arquivo de interface externo* é um agrupamento lógico de dados que reside fora da aplicação, mas fornece dados que podem ser usados pela aplicação.

Coletados os dados, a tabela da Figura 30.1 é completada e um valor de complexidade é associado a cada contagem. Organizações que usam métodos ponto de função desenvolvem critérios para determinar se determinada entrada é simples, média ou complexa. No entanto, a determinação da complexidade é, de certo modo, subjetiva.

Para calcular pontos de função (FP), usa-se a seguinte relação:

$$FP = \text{contagem total} \times [0,65 + 0,01 \times \sum(F_i)] \quad (30.1)$$

onde a contagem total é a soma de todas as entradas FP obtidas da Figura 30.1.

Os F_i ($i = 1$ a 14) são *fatores de ajuste de valor* (VAF, *value adjustment factors*) baseados em respostas às questões a seguir [Lon02]:

1. O sistema exige salvamento (*backup*) e recuperação (*recovery*) confiável?
2. São necessárias comunicações de dados especializadas para transferir informações para a aplicação ou da aplicação?
3. Há funções de processamento distribuído?
4. O desempenho é crítico?
5. O sistema será executado em um ambiente operacional existente e intensamente utilizado?
6. O sistema exige entrada de dados online?
7. A entrada online de dados exige que a transação de entrada seja composta de várias telas ou operações?

Fatores de ajuste de valor são usados para fornecer uma indicação da complexidade do problema.

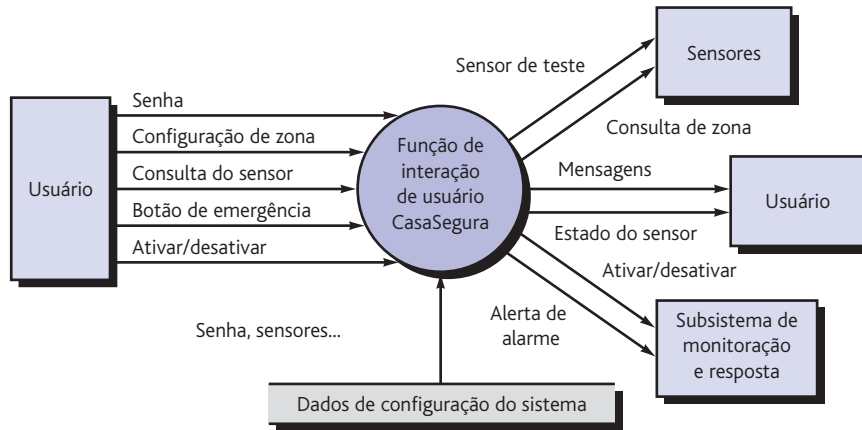


FIGURA 30.2 Um modelo de fluxo para a função de interação do usuário do *CasaSegura*.

8. Os ILFs são atualizados online?
9. As entradas, saídas, arquivos ou consultas são complexos?
10. O processamento interno é complexo?
11. O código é projetado para ser reutilizável?
12. A conversão e instalação estão incluídas no projeto?
13. O sistema é projetado para múltiplas instalações em diferentes organizações?
14. A aplicação é projetada para facilitar a troca e o uso pelo usuário?

Cada uma dessas perguntas é respondida por meio de uma escala ordinal que varia de 0 (não importante ou não aplicável) a 5 (absolutamente essencial). Os valores das constantes na Equação 30.1 e os fatores de peso aplicados aos valores do domínio de informações são determinados empiricamente.

Para ilustrarmos o uso da métrica FP nesse contexto, consideramos um diagrama de fluxo para uma função de interação do usuário no software *CasaSegura*, representado na Figura 30.2. A função gerencia a interação do usuário, aceitando uma senha para ativar ou desativar o sistema, e possibilita consultas sobre o estado das zonas de segurança e vários sensores de segurança. A função mostra uma série de mensagens imediatas e envia sinais de controle apropriados para os vários componentes do sistema de segurança.

O diagrama de fluxo é avaliado para determinar um conjunto-chave de medidas de domínio de informação necessárias para o cálculo da métrica ponto de função. Três entradas externas – **senha**, **botão de emergência** e **ativar/desativar** – são mostradas na figura com duas consultas externas – **consulta de zona** e **consulta de sensor**. É mostrado um ILF (**arquivo de configuração de sistema**). São apresentadas também duas saídas externas (**mensagens** e **estado do sensor**) e quatro EIFs (**sensor de teste**, **configuração de zona**, **ativar/desativar** e **alerta de alarme**). Esses dados, com a complexidade apropriada, estão na Figura 30.3.

Uma calculadora de FP online pode ser encontrada em <http://groups.engin.umd.umich.edu/CIS/course.des/cis375/projects/fp99/main.html>.

Valor do Domínio de Informação	Fator de peso				
	Contagem		Simples	Média	Complexo
Entradas Externas (EIs)	3	×	3	4	6 = 9
Saídas Externas (EOs)	2	×	4	5	7 = 8
Consultas Externas (EQs)	2	×	3	4	6 = 6
Arquivos de Interface Externos (EIFs)	1	×	7	10	15 = 7
Arquivos Lógicos Internos (ILFs)	4	×	5	7	10 = 20
Contagem total					50

FIGURA 30.3 Calculando pontos de função.

O total da contagem mostrado na Figura 30.3 deve ser ajustado usando-se a Equação 30.1. Para as finalidades deste exemplo, supomos que $\Sigma(F_i)$ é 46 (um produto moderadamente complexo). Portanto,

$$FP = 50 \times [0,65 + (0,01 \times 46)] = 56$$

"Em vez de ficar apenas meditando sobre 'para que serve a nova métrica'... devemos fazer também a pergunta mais básica, 'o que vamos fazer com as métricas?'"

Michael Mah e Larry Putnam

Com base no valor FP projetado, derivado do modelo de requisitos, a equipe de projeto pode estimar o tamanho total implementado da função de interação de usuário do *CasaSegura*. Vamos supor que dados já conhecidos indicam que um FP se traduz em 60 linhas de código (será usada uma linguagem orientada a objetos) e que 12 FPs são produzidos por cada pessoa-mês de trabalho. Esses dados históricos fornecem ao gerente do projeto informações importantes de planejamento, baseadas no modelo de requisitos e não nas estimativas preliminares. Suponha ainda que projetos anteriores tenham apresentado uma média de três erros por ponto de função durante as revisões de requisitos e projeto e quatro erros por ponto de função durante o teste de unidade e integração. Esses dados podem finalmente ajudá-lo a avaliar a totalidade das suas atividades de revisão e teste. Uemura e seus colegas [Uem99] sugerem que pontos de função também podem ser calculados por meio dos diagramas de classe UML e sequência.

30.2.2 Métricas para qualidade de especificação

Medindo-se características da especificação, é possível obter informações quantitativas sobre peculiaridade e totalidade.

Davis e seus colegas [Dav93] propõem uma lista de características que podem ser usadas para avaliar a qualidade do modelo de requisitos e as especificações de requisitos correspondentes: *peculiaridade* (ausência de ambiguidade), *totalidade*, *exatidão*, *entendimento*, *repetibilidade*, *consistência interna e externa*, *disponibilidade*, *brevidade*, *rastreabilidade*, *modificação*, *precisão* e *reutilização*. Além disso, os autores observam que há especificações de alta qualidade armazenadas eletronicamente; executáveis ou pelo menos interpretáveis; comentadas de acordo com sua importância relativa; e versões atualizadas estáveis, referências cruzadas organizadas e especificadas com o nível correto de detalhe.

Embora muitas dessas características pareçam ser qualitativas em sua natureza, cada uma delas pode ser representada usando uma ou mais métricas. [Dav93] Por exemplo, assumimos que há n_r requisitos em uma especificação, tal que

$$n_r = n_f + n_{nf}$$

onde n_f é o número de requisitos funcionais e n_{nf} é o número de requisitos não funcionais (por exemplo, desempenho).

Para determinar a *peculiaridade* (ausência de ambiguidade) dos requisitos, Davis e colegas sugerem uma métrica baseada na consistência da interpretação de cada requisito por parte dos revisores:

$$Q_1 = \frac{n_{ui}}{n_r}$$

onde n_{ui} é o número de requisitos para os quais todos os revisores tiveram interpretações idênticas. Quanto mais próximo o valor Q estiver de 1, mais baixa será a ambiguidade da especificação.

A *totalidade* dos requisitos funcionais pode ser determinada calculando-se a relação

$$Q_2 = \frac{n_u}{[n_i \times n_s]}$$

onde n_u é o número de requisitos funcionais únicos, n_i é o número de entradas (estímulos) definidas ou implícitas pela especificação e n_s é o número de estados especificados. A relação Q_2 mede a porcentagem de funções necessárias especificadas para um sistema. No entanto, ela não trata de requisitos não funcionais. Para incorporar esses requisitos em uma métrica global para a totalidade, deve-se considerar o grau com o qual os requisitos foram validados:

$$Q_3 = \frac{n_c}{[n_c + n_{nv}]}$$

onde n_c é o número de requisitos validados como corretos e n_{nv} é o número de requisitos que ainda não foram validados.

"Meça aquilo que é mensurável e torne mensurável o que não for."

Galileu

30.3 Métricas para o modelo de projeto

É inconcebível que o projeto de um novo avião, de um novo chip de computador ou de um novo edifício comercial seja conduzido sem definir medidas, sem determinar métricas para os vários aspectos de qualidade e sem usá-las como indicadores para orientar a maneira pela qual o projeto evoluirá. E, apesar disso, o projeto de sistemas complexos baseados em software muitas vezes é realizado praticamente sem nenhuma medição. A ironia disso tudo é que estão disponíveis métricas de projeto para software, mas a grande maioria dos engenheiros de software continua ignorando sua existência.

As métricas de projeto para software de computador, como todas as outras métricas de software, não são perfeitas. Continua o debate sobre sua eficiência e a maneira pela qual devem ser aplicadas. Muitos especialistas argumentam que é necessária mais experimentação para que as medições de projeto possam ser usadas. Ainda assim, projeto sem medição é uma alternativa inaceitável.

As métricas podem fornecer informações sobre dados estruturais e complexidade do sistema associadas ao projeto da arquitetura.

30.3.1 Métricas de projeto de arquitetura

As métricas de projeto de arquitetura focalizam as características da arquitetura do programa (Capítulo 13) com ênfase na estrutura arquitetural e na

eficácia dos módulos ou componentes dentro da arquitetura. Essas métricas são uma “caixa-preta”, no sentido de que não exigem qualquer conhecimento do funcionamento interno de determinado componente de software.

Card e Glass [Car90] definem três medidas de complexidade de projeto de software: complexidade estrutural, complexidade de dados e complexidade de sistema.

Para arquiteturas hierárquicas (por exemplo, arquiteturas de chamada e retorno), a *complexidade estrutural* de um módulo i é definida da seguinte maneira:

$$S(i) = f_{\text{out}}^2(i) \quad (30.2)$$

onde $f_{\text{out}}(i)$ é o fan-out⁶ do módulo i .

A *complexidade dos dados* (*data complexity*) proporciona uma indicação da complexidade na interface interna para um módulo i e é definida como

$$D(i) = \frac{\nu(i)}{|f_{\text{out}}(i) + 1|} \quad (30.3)$$

onde $\nu(i)$ é o número de variáveis de entrada e saída passadas para e do módulo i .

Por fim, a *complexidade do sistema* (*system complexity*) é definida como a soma da complexidade estrutural e de dados, especificada como

$$C(i) = S(i) + D(i) \quad (30.4)$$

À medida que esses valores de complexidade aumentam, a complexidade global da arquitetura do sistema também aumenta. Isso leva a uma maior probabilidade de que o trabalho de integração e teste também aumente.

Fenton [Fen91] sugere um conjunto de métricas de morfologia (isto é, forma) simples que permite comparar diferentes arquiteturas de programa usando uma série de dimensões diretas. Referindo-se à arquitetura de chamada e retorno na Figura 30.4, podem ser definidas as seguintes métricas:

$$\text{Tamanho} = n + a$$

onde n é o número de nós e a é o número de arcos. Para a arquitetura mostrada na Figura 30.4,

$$\text{Tamanho} = 17 + 18 = 35$$

Profundidade = caminho mais longo desde o nó raiz (topo) até o nó da folha. Para a arquitetura mostrada na Figura 30.4, profundidade = 4.

Largura = número máximo de nós em qualquer nível da arquitetura. Para a arquitetura mostrada na Figura 30.4, largura = 6.

A relação arco para nó, $r = a/n$, mede a densidade de conectividade da arquitetura e pode fornecer uma indicação simples do acoplamento da arquitetura. Para a arquitetura mostrada na Figura 30.4, $r = 18/17 = 1,06$.

⁶ *Fan-out* é definido como o número de módulos imediatamente subordinados ao módulo i ; ou seja, o número de módulos chamados diretamente pelo módulo i .

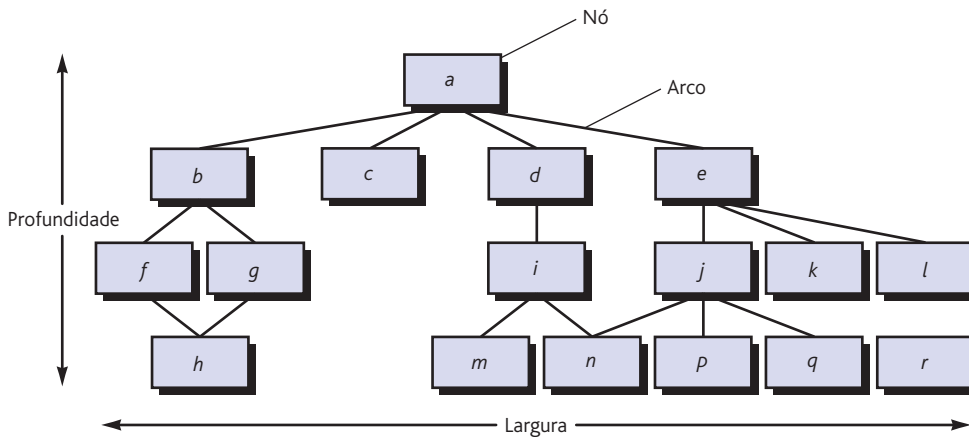


FIGURA 30.4 Métricas de morfologia.

A Força Aérea Americana (U.S. Air Force Systems Command) [USA87] desenvolveu um conjunto de indicadores de qualidade de software baseados nas características de projeto mensuráveis de um programa de computador. Usando conceitos similares àqueles propostos na norma IEEE Std. 982.1-2005 [IEE05], a Força Aérea usa informações obtidas do projeto de dados e de arquitetura para criar um *índice de qualidade de projeto de estrutura* (DSQI, *design structure quality index*) que varia de 0 a 1. Os seguintes valores devem ser apurados para calcular o DSQI [Cha89]:

S_1 = número total de módulos definidos na arquitetura do programa

S_2 = número de módulos cuja função correta depende da origem dos dados de entrada ou que produzem dados para ser usados em outro lugar (em geral, módulos de controle, entre outros, não seriam contados como parte de S_2)

S_3 = número de módulos cuja função correta depende de processamento anterior

S_4 = número de itens de banco de dados (inclui objetos de dados e todos os atributos que definem objetos)

S_5 = número total de itens únicos de banco de dados

S_6 = número de segmentos de banco de dados (diferentes registros ou objetos individuais)

S_7 = número de módulos com uma única entrada e saída (processamento de exceção não é considerado saída múltipla)

Uma vez determinados os valores S_1 a S_7 para um programa de computador, podem ser calculados os seguintes valores intermediários:

Estrutura de programa: D_1 , definido da seguinte maneira: se o projeto da arquitetura foi desenvolvido por meio de um método distinto (por exemplo, projeto orientado a fluxo de dados ou projeto orientado a objetos), então $D_1 = 1$; caso contrário, $D_1 = 0$.

"A medição pode ser vista como um desvio. Esse desvio é necessário porque os humanos, na sua maioria, não são capazes de tomar decisões claras e objetivas [sem um suporte quantitativo]."

Horst Zuse

$$\text{Independência de módulo: } D_2 = 1 - \left(\frac{S_2}{S_1} \right)$$

$$\text{Módulos não dependentes de processamento anterior: } D_3 = 1 - \left(\frac{S_3}{S_1} \right)$$

$$\text{Tamanho da base de dados: } D_4 = 1 - \left(\frac{S_5}{S_4} \right)$$

$$\text{Divisão da base de dados: } D_5 = 1 - \left(\frac{S_6}{S_4} \right)$$

$$\text{Característica de entrada/saída do módulo: } D_6 = 1 - \left(\frac{S_7}{S_1} \right)$$

Com os valores intermediários determinados, o DSQI é calculado da seguinte maneira:

$$\text{DSQI} = \sum w_i D_i \quad (30.5)$$

onde $i = 1$ a 6, w_i é o peso relativo da importância de cada um dos valores intermediários e $\sum w_i = 1$ (se todos os D_i tiverem o mesmo peso, então $w_i = 0,167$).

O valor de DSQI para projetos passados pode ser determinado e comparado com um projeto que esteja atualmente em desenvolvimento. Se o DSQI for significativamente mais baixo do que a média, é aconselhável executar mais trabalho de projeto e revisão. De modo semelhante, se grandes alterações devem ser feitas em um projeto, o efeito dessas alterações sobre o DSQI pode ser calculado.

30.3.2 Métricas para projeto orientado a objetos

Muita coisa é subjetiva no projeto orientado a objetos – um projetista experiente “sabe” como caracterizar um sistema orientado a objetos de modo que implemente os requisitos do cliente. Entretanto, à medida que um modelo de projeto orientado a objetos cresce em tamanho e complexidade, uma visão mais objetiva das características do projeto pode favorecer tanto o projetista experiente (que obtém uma visão adicional) quanto o novato (que obtém uma indicação da qualidade que, de outra forma, não estaria disponível).

Em um tratamento detalhado das métricas de software para sistemas orientados a objetos, Whitmire [Whi97] descreve nove características distintas e mensuráveis de um projeto orientado a objetos. O *tamanho* é definido tomando-se uma contagem estática das entidades orientadas a objetos, como classes ou operações, acopladas à profundidade de uma árvore de herança. A *complexidade* é definida pelas características estruturais, examinando-se como as classes de um projeto orientado a objetos se inter-relacionam. O *acoplamento* é medido pela contagem de conexões físicas entre elementos do projeto orientado a objetos (por exemplo, o número de colaborações entre classes ou o número de mensagens passadas entre objetos). A *suficiência* é “o grau com o qual uma abstração [class] possui as características exigidas dela...” [Whi97]. A *totalidade* determina se uma classe entrega o conjunto de propriedades que reflete totalmente as necessidades do domínio do problema. A *coesão* é determinada examinando-se se todas as operações trabalham em conjunto para atingir uma finalidade única e bem definida. A *originalidade* é o grau segundo o qual uma operação é atômica – isto é, a operação não pode ser construída por meio de uma sequência de outras operações contidas dentro de uma clas-

se. A similaridade determina o grau segundo o qual duas ou mais classes são similares em termos de sua estrutura, função, comportamento ou finalidade. A volatilidade mede a probabilidade da ocorrência de uma alteração.

Na realidade, as métricas de produto para sistemas orientados a objetos podem ser aplicadas não só ao modelo de projeto, mas também ao modelo de requisitos. Nas próximas seções, discutiremos as métricas que indicam a qualidade em nível de classe orientada a objetos e em nível de operação. Além disso, serão exploradas também métricas aplicáveis a gerenciamento de projeto e teste.

30.3.3 Métricas orientadas a classes – o conjunto de métricas CK

A classe é a unidade fundamental de um sistema orientado a objetos. Portanto, medidas e métricas para uma classe individual, para a hierarquia da classe e para as colaborações entre classes serão valiosas quando tivermos de avaliar a qualidade de um projeto orientado a objetos. Uma classe encapsula dados, e a função manipula os dados. Com frequência é o “pai” das subclasses (às vezes chamadas de filhas) que herda seus atributos e operações. Muitas vezes elas colaboram com outras classes. Cada uma dessas características pode ser usada como base para medição.⁷

Chidamber e Kemerer (CK) propuseram um dos conjuntos mais conhecidos de métricas de software orientado a objetos [Chi94]. Comumente chamado de *conjunto de métricas CK* (*CK metrics suite*), nele os autores propõem seis métricas de projeto baseado em classe para sistemas orientados a objetos.⁸

Métodos ponderados por classe (WMC, weighted methods per class). Suponha que n métodos de complexidade $c_1, c_2, \dots, c_n$ são definidos para uma classe C . A métrica específica de complexidade escolhida (por exemplo, complexidade ciclomática) deve ser normalizada de maneira que a complexidade nominal para um método assuma o valor 1.0.

$$\text{WMC} = \sum c_i$$

para $i = 1$ a n . O número de métodos e sua complexidade são indicadores adequados do trabalho necessário para implementar e testar uma classe. Além disso, quanto maior for o número de métodos, mais complexa será a árvore de herança (todas as subclasses herdam os métodos de seus pais). Por fim, conforme o número de métodos cresce para uma dada classe, ela tende a se tornar cada vez mais específica da aplicação, limitando, assim, seu potencial de reutilização. Por todas essas razões, o WMC deverá ser mantido o mais baixo possível.

Extensão da árvore de herança (DIT, depth of the inheritance tree). Essa métrica é “o comprimento máximo do nó até a raiz da árvore” [Chi94]. De acordo

⁷ A validade de algumas das métricas abordadas neste capítulo está em discussão na literatura técnica. Aqueles que defendem a teoria da medição exigem um grau de formalismo que algumas métricas orientadas a objetos não fornecem. No entanto, é possível afirmar que as métricas citadas proporcionam informações úteis para o engenheiro de software.

⁸ Chidamber, Darcy e Kemerer usam o termo *métodos* em vez de *operações*. O uso desse termo é citado nesta seção.

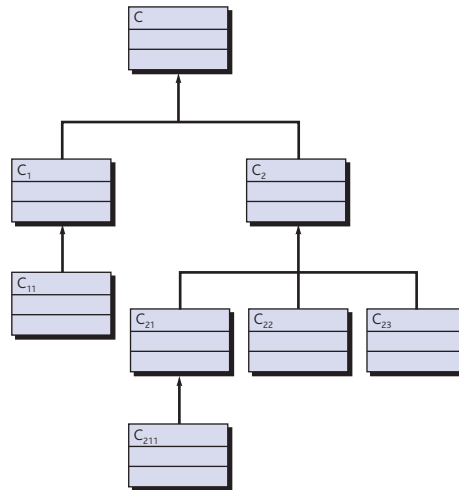


FIGURA 30.5 Uma hierarquia de classes.

com a Figura 30.5, o valor da DIT para a hierarquia de classes mostrada é 4. À medida que a DIT cresce, é possível que classes de nível inferior herdem muitos métodos. Isso causa dificuldades em potencial quando se tenta prever o comportamento de uma classe. Uma hierarquia de classes profunda (com DIT grande) também leva a uma complexidade maior no projeto. Pelo lado positivo, grandes valores DIT implicam que muitos métodos podem ser reutilizados.

Os conceitos de acoplamento e coesão se aplicam ao software convencional e ao software orientado a objetos. Mantenha o acoplamento de classes baixo e a coesão de operação alta.

Número de filhas (NOC, number of children). As subclasses imediatamente subordinadas a uma classe na hierarquia de classes são chamadas de suas filhas. De acordo com a Figura 30.5, a classe C_2 tem três filhas – as subclasses C_{21} , C_{22} e C_{23} . Conforme cresce o número de filhas, a reutilização aumenta, mas também a abstração representada pela classe pai pode ser diluída se algumas das filhas não forem membros apropriados da classe pai. À medida que o NOC aumenta, a quantidade de teste (necessário para exercitar cada filha em seu contexto operacional) também aumentará.

Acoplamento entre objetos de classes (CBO, coupling between object classes). O modelo CRC (Capítulo 10) pode ser usado para determinar o valor do CBO. Basicamente, CBO é o número de colaborações listadas para uma classe em seu cartão de indexação CRC.⁹ À medida que o CBO aumenta, é possível que a reutilização de uma classe diminua. Altos valores de CBO também complicam modificações e o teste resultante dessas modificações. Em geral, os valores de CBO para cada classe devem ser mantidos o mais baixos possível. Isso está em consonância com a diretriz geral de reduzir o acoplamento em software convencional.

Resposta para uma classe (RFC, response for a class). O conjunto de respostas de uma classe é “um conjunto de métodos com potencial de serem executados em resposta a uma mensagem recebida por um objeto daquela classe” [Chi94]. RFC é o número de métodos no conjunto de respostas. Conforme a

⁹ Se os cartões de indexação CRC são desenvolvidos manualmente, a totalidade e a consistência devem ser avaliadas para que o CBO possa ser determinado de forma confiável.

RFC aumenta, o trabalho necessário para o teste também aumenta, porque a sequência de testes (Capítulo 24) cresce. Além disso, à medida que a RFC aumenta, a complexidade geral de projeto da classe aumenta.

Falta de coesão em métodos (LCOM, lack of cohesion in methods). Cada método dentro de uma classe **C** acessa um ou mais atributos (também chamados de variáveis de instância). LCOM é o número de métodos que acessam um ou mais dos mesmos atributos.¹⁰ Se nenhum método acessa os mesmos atributos, LCOM = 0. Para ilustrar o caso em que LCOM ≠ 0, considere uma classe com seis métodos. Quatro dos métodos têm um ou mais atributos em comum (eles acessam atributos comuns). Portanto, LCOM = 4. Se LCOM for alto, métodos podem ser acoplados uns aos outros via atributos. Isso aumenta a complexidade do projeto de classe. Embora haja casos em que um valor alto de LCOM seja justificável, é desejável manter a coesão alta; isto é, manter LCOM baixo.¹¹

CASASEGURA



Aplicando métricas CK

Cena: Sala do Vinod.

Atores: Vinod, Jamie, Shakira e Ed – membros da equipe de engenharia de software do CasaSegura que continuam a trabalhar no projeto em nível de componente e projeto de caso de teste.

Conversa:

Vinod: Vocês conseguiram ler a descrição sobre o conjunto de métricas CK que eu enviei na quarta-feira e fazer aquelas medições?

Shakira: Não foi muito complicado. Utilizei minha classe UML e diagramas de sequência, como você sugeriu, e fiz contagens aproximadas de DIT, RFC e LCOM. Não consegui encontrar o modelo CRC, então não fiz contagem de CBO.

Jamie (sorrindo): Você não conseguiu encontrar o modelo CRC porque ele estava comigo.

Shakira: É isso que eu adoro nesta equipe: uma excelente comunicação.

Vinod: Fiz minhas contagens... vocês desenvolveram valores para as métricas CK?

[Jamie e Ed acenam afirmativamente.]

Jamie: Como eu tinha os cartões CRC, dei uma olhada no CBO e ele parecia bastante uniforme na maioria das classes. Havia apenas uma exceção.

Ed: Há algumas classes em que o RFC é muito alto, comparado com as médias... talvez devêssemos dar um jeito de simplificá-las.

Jamie: Talvez sim, talvez não. Ainda estou preocupado com o prazo, e não quero corrigir coisas que não tenham realmente erros.

Vinod: Concordo. Talvez vocês deveriam dar uma olhada nas classes que apresentam números ruins em pelo menos duas ou mais das métricas CK. Umas duas investidas e está feito.

Shakira (observando a lista de classes de Ed com alto RFC): Olhe, veja esta classe, com LCOM e RFC ambos altos. Duas investidas?

Vinod: Penso que sim... será difícil de implementar devido à complexidade e difícil de testar pela mesma razão. Provavelmente vale a pena projetar duas classes separadas para conseguir o mesmo comportamento.

Jamie: Você acha que modificando economizaremos tempo?

Vinod: No longo prazo, sim.

¹⁰ A definição formal é um pouco mais complexa. Consulte os detalhes em [Chi94].

¹¹ A métrica LCOM fornece informações úteis em algumas situações, mas pode ser confusa em outras. Por exemplo, mantendo-se o acoplamento encapsulado (dentro de uma classe), aumenta-se a coesão do sistema como um todo. Portanto, pelo menos em um sentido importante, LCOM mais alto realmente sugere que uma classe possa ter coesão mais alta, não mais baixa.

"Analisar o software orientado a objetos para avaliar sua qualidade está se tornando cada vez mais importante, à medida que o paradigma orientado a objetos continua a crescer em popularidade."

**Rachel Harrison
et al.**

30.3.4 Métricas orientadas a classes – o conjunto de métricas MOOD

Harrison, Counsell e Nithi [Har98b] propõem um conjunto de métricas para projeto orientado a objetos que oferece indicadores quantitativos para características do projeto. Veja a seguir exemplos das métricas MOOD.

Fator de herança de método (MIF, method inheritance factor). O grau segundo o qual a arquitetura de classe de um sistema orientado a objetos faz uso de herança tanto para métodos (operações) quanto para atributos. É definido como

$$\text{MIF} = \frac{\sum M_i(C_i)}{\sum M_a(C_i)}$$

onde a somatória ocorre com $i = 1$ a TC. TC é definido como o número total de classes na arquitetura, C_i é a classe dentro da arquitetura e

$$M_a(C_i) = M_d(C_i) + M_i(C_i)$$

onde

$M_a(C_i)$ = número de métodos que podem ser invocados em associação com C_i

$M_d(C_i)$ = número de métodos declarados na classe C_i

$M_i(C_i)$ = número de métodos herdados (e não cancelados) em C_i

O valor de MIF [o fator de herança de atributo (AIF)] é definido de maneira análoga fornece uma indicação do impacto da herança sobre o software orientado a objetos.

Fator de acoplamento (CF, coupling factor). Anteriormente neste capítulo, observou-se que o acoplamento é uma indicação das conexões entre elementos do projeto orientado a objetos. O conjunto de métricas MOOD define o acoplamento da seguinte maneira:

$$\text{CF} = \sum_i \sum_j \text{is\_client} \frac{(C_i, C_j)}{\text{TC}^2 - \text{TC}}$$

onde os somatórios ocorrem com $i = 1$ a TC e $j = 1$ a TC. A função

$\text{is\_client} = 1$, se, e somente se, existir uma relação entre a classe-cliente C_c e a classe-servidora C_s , e $C_c \neq C_s$.

= 0, caso contrário

Embora muitos fatores afetem a complexidade, o entendimento e a sustentabilidade do software, faz sentido concluir que, conforme o fator CF aumenta, a complexidade do software orientado a objetos também aumenta, e o entendimento, a sustentabilidade e o potencial de reutilização podem ficar prejudicados.

Harrison e seus colegas [Har98b] apresentam uma análise detalhada do MIF e CF com suas métricas e examinam sua validade para uso na avaliação da qualidade do projeto.

30.3.5 Métricas orientadas a objeto propostas por Lorenz e Kidd

Em seu livro sobre métricas orientadas a objetos, Lorenz e Kidd [Lor94] dividem as métricas baseadas em classe em quatro categorias principais, cada uma ocupando uma posição no projeto em nível de componente: tamanho, herança, aspectos internos e externos. As métricas orientadas a tamanho para uma classe de projeto orientado a objetos focam-se nas contagens de atributos e operações para uma classe individual e valores médios para o sistema orientado a objetos como um todo. Métricas baseadas em herança focam-se na maneira como as operações são reutilizadas por meio da hierarquia de classe. Métricas para aspectos internos de classe procuram a coesão (Seção 30.3.3) e questões orientadas a código, e as métricas de aspectos externos examinam acoplamento e reutilização. Um exemplo das métricas propostas por Lorenz e Kidd é o seguinte:

Tamanho de classe (CS, class size). O tamanho global de uma classe pode ser determinado por meio das seguintes medidas:

- O número total de operações (operações de instâncias herdadas e privadas) encapsuladas dentro da classe
- O número de atributos (atributos de instâncias herdadas e privadas) encapsulados pela classe

A métrica WMC proposta por Chidamber e Kemerer (Seção 30.3.3) também é uma medida ponderada do tamanho da classe. Conforme observamos antes, valores altos de CS indicam que uma classe pode ter muita responsabilidade. Isso reduzirá a reutilização da classe e complicará a implementação e o teste. Em geral, operações e atributos, herdados ou públicos, devem ser ponderados mais intensamente na determinação do tamanho da classe [Lor94]. Operações privadas e atributos privados permitem especialização e são mais localizados no projeto. Podem ser calculados também valores médios para o número de atributos e operações de classe. Quanto mais baixos forem os valores médios para o tamanho, maior será a probabilidade de as classes dentro do sistema de serem reutilizadas amplamente.

Durante a revisão do modelo de análise, os cartões de indexação CRC fornecerão uma indicação razoável dos valores esperados para CS. Se você encontrar uma classe com um alto número de responsabilidades, pense em dividi-la.

30.3.6 Métricas de projeto em nível de componente

As métricas de projeto em nível de componente para componentes convencionais de software focalizam características internas de um componente de software e incluem medidas dos “três Cs” – coesão de módulo (*module cohesion*) [Bie94], acoplamento (*coupling*) [Dha95] e complexidade (*complexity*) [Zus97]. Essas medidas podem ajudá-lo a avaliar a qualidade de um projeto em nível de componente.

Métricas de projeto em nível de componente podem ser aplicadas, uma vez desenvolvido o projeto procedural; elas são uma “caixa de vidro” pelo fato de exigirem conhecimento do funcionamento interno do módulo que está sendo considerado. Como alternativa, elas podem ser adiadas até que o código-fonte esteja disponível.

É possível calcular medidas da independência funcional – acoplamento e coesão – de um componente e usá-las para avaliar a qualidade de um projeto.

30.3.7 Métricas orientadas a operação

Devido à classe ser a unidade dominante nos sistemas orientados a objetos, poucas métricas têm sido propostas para operações que residem dentro de

uma classe. Churcher e Shepperd [Chu95] discutem isso quando afirmam: “Resultados de estudos recentes indicam que métodos tendem a ser pequenos, tanto em termos do número de instruções quanto em complexidade lógica [Wil93], sugerindo que a estrutura de conectividade de um sistema pode ser mais importante do que o conteúdo de módulos individuais”. No entanto, podem ser obtidas algumas informações examinando-se as características médias dos métodos (operações). Três métricas simples, propostas por Lorenz e Kidd [Lor94], são apropriadas:

Tamanho médio de operação (OS_{avg} , average operation size). O tamanho pode ser determinado contando-se o número de linhas de código ou o número de mensagens enviadas pela operação. À medida que aumenta o número de mensagens enviadas por uma única operação, é possível que as responsabilidades não tenham sido bem alocadas dentro da classe.

Complexidade de operação (OC, operation complexity). A complexidade de uma operação pode ser calculada usando-se quaisquer métricas de complexidade propostas para software convencional [Zus90]. Como as operações devem ser limitadas a uma responsabilidade específica, o projetista deve se esforçar para manter a OC o mais baixo possível.

Número médio de parâmetros por operação (NP_{avg} , average number of parameters per operation). Quanto maior é o número de parâmetros de operação, mais complexa é a colaboração entre objetos. Em geral, NP_{avg} deve ser mantida o mais baixa possível.

30.3.8 Métricas de projeto de interface de usuário

Embora haja uma literatura significativa sobre o projeto de interface humanos/computadores (Capítulo 15), têm sido publicadas poucas informações sobre métricas que poderiam dar uma ideia da qualidade e da utilização da interface.

Um estudo das métricas para páginas Web [Ivo01] indica que características simples dos elementos do layout também podem ter um impacto significativo sobre a qualidade aparente do projeto da GUI. O número de palavras, links, gráficos, cores e fontes (entre outras características) contidos em uma página Web afetam a complexidade aparente e a qualidade dela.

Embora as métricas de interface de usuário possam ser úteis em alguns casos, o árbitro final deve ser a entrada do usuário baseada em protótipos de GUI. Nielsen e Levy [Nie94] relatam que “há uma chance razoavelmente grande de sucesso se alguém escolher entre interface [projetos] baseada unicamente nas opiniões dos usuários. O desempenho médio da tarefa do usuário e sua satisfação subjetiva com uma GUI estão altamente correlacionados”.

“Você pode aprender pelo menos um princípio de projeto de interface de usuário quando utiliza uma lavadora de louças. Se você colocar muitos objetos lá dentro, nada ficará muito limpo.”

Autor desconhecido

30.4 Métricas de projeto para WebApps e aplicativos móveis

Um bom conjunto de medidas e métricas para WebApps fornece respostas quantitativas para as seguintes questões:

- A interface de usuário promove a utilização?

- O aspecto estético da WebApp é apropriado para o domínio de aplicação e é agradável ao usuário?
- O conteúdo é projetado de forma a reunir o máximo de informações com o mínimo esforço?
- A navegação é eficiente e direta?
- A arquitetura da WebApp foi projetada para acomodar as metas e objetivos especiais de seus usuários, a estrutura de conteúdo e funcionalidade e o fluxo de navegação exigido para usar o sistema eficientemente?
- Os componentes são projetados de maneira a reduzir a complexidade de procedimento e melhorar a exatidão, confiabilidade e desempenho?

Atualmente, essas questões podem ser resolvidas apenas qualitativamente, pois não existe um conjunto validado de métricas que forneçam respostas quantitativas.

A seguir, apresentamos uma amostra representativa das métricas de projeto para WebApps e aplicativos móveis propostas na literatura. É importante observar que muitas dessas métricas ainda não foram validadas e devem ser usadas com muito critério.

Métricas de interface. Para WebApps, podem ser consideradas as seguintes medidas de interface:

<i>Métrica sugerida</i>	<i>Descrição</i>
Adequação do layout	A posição relativa das entidades dentro da interface
Complexidade de layout	Número de regiões distintas ¹² definidas para uma interface
Complexidade da região de layout	Número médio de links distintos por região
Complexidade de reconhecimento	Número médio de itens distintos que o usuário deve examinar antes de tomar uma decisão de navegação ou de entrada de dados
Tempo de reconhecimento	Tempo médio (em segundos) que o usuário gasta para selecionar a ação apropriada para uma tarefa
Trabalho de digitação	Número médio de toques necessários para uma função específica
Esforço de clique do mouse	Número médio de cliques do mouse por função
Complexidade de seleção	Número médio de links que podem ser selecionados por página
Tempo de aquisição de conteúdo	Número médio de palavras de texto por página da Web
Carga de memória	Número médio de itens de dados distintos que o usuário deve lembrar para atingir um objetivo específico

Muitas dessas métricas são aplicáveis a todas as interfaces de usuário e devem ser consideradas junto com as apresentadas na Seção 30.3.8.

Métricas de estética (design gráfico). Por sua natureza, o projeto estético depende da avaliação qualitativa e geralmente não é favorável à medição e às métricas. No entanto, Ivory e seus colegas [Ivo01] propõem um conjunto de medidas que podem ser úteis para avaliar o impacto do projeto estético:

¹² Uma região distinta é uma área do layout de tela que executa um conjunto específico de funções relacionadas (por exemplo, uma barra de menu, uma tela gráfica estática, uma área de conteúdo, uma tela animada).

<i>Métrica sugerida</i>	<i>Descrição</i>
Número de palavras	Número total de palavras que aparecem em uma página
Porcentagem de texto de corpo	Porcentagem de palavras que são texto de corpo <i>versus</i> texto de título (por exemplo, cabeçalhos)
Porcentagem de texto de corpo destacado	Parte do texto de corpo que é destacado (por exemplo, negrito, caixa-alta)
Contagem de posicionamento de texto	Mudanças na posição do texto a partir do alinhamento à esquerda
Contagem de grupos de texto	Áreas de texto destacadas com cores, regiões com bordas, réguas ou listas
Contagem de links	Total de links em uma página
Tamanho de página	Total de bytes na página, bem como elementos, gráficos e folhas de estilo
Porcentagem gráfica	Porcentagem de bytes da página que são usados para gráficos (figuras)
Contagem gráfica	Total de figuras na página (não incluindo figuras especificadas em scripts, applets e objetos)
Contagem de cores	Total de cores usadas
Contagem de fonte	Total de fontes empregadas (por exemplo, tipo + tamanho + negrito + itálico)

Métricas de conteúdo. Métricas nessa categoria focalizam a complexidade de conteúdo e agrupamentos de objetos de conteúdo organizado em páginas [Men01].

<i>Métrica sugerida</i>	<i>Descrição</i>
Espera de página	Tempo médio necessário para que uma página seja baixada em diferentes velocidades de conexão
Complexidade da página	Número médio de diferentes tipos de mídia usada na página, não incluindo texto
Complexidade gráfica	Número médio de mídia gráfica por página
Complexidade de áudio	Número médio de mídia de áudio por página
Complexidade de vídeo	Número médio de mídia de vídeo por página
Complexidade de animação	Número médio de animações por página
Complexidade de imagem escaneada	Número médio de imagens escaneadas por página

Métricas de navegação. Métricas nessa categoria tratam da complexidade do fluxo de navegação [Men01]. Em geral, são direcionadas apenas para aplicações Web estáticas, que não incluem links e páginas gerados dinamicamente.

<i>Métrica sugerida</i>	<i>Descrição</i>
Complexidade de link de página	Número de links por página
Conectividade	Número total de links internos, não incluindo links gerados dinamicamente
Densidade de conectividade	Conectividade dividida por número de página

Com um subconjunto das métricas sugeridas, pode ser possível derivar relações empíricas que permitam a uma equipe de desenvolvimento de WebApp avaliar a qualidade técnica e prever o trabalho necessário com base nas estimativas projetadas da complexidade. Há ainda muito a ser feito nessa área.

FERRAMENTAS DO SOFTWARE



Métricas técnicas para WebApps

Objetivo: ajudar os engenheiros da Web a desenvolver métricas significativas para WebApps que forneçam informações sobre a qualidade global de um aplicativo.

Mecanismos: o mecanismo das ferramentas é variado.

Ferramentas representativas:¹³

Netmechanic Tools, desenvolvida pela Netmechanic (www.netmechanic.com), é uma coleção de ferramentas que ajudam a melhorar o desempenho do site, focalizando assuntos específicos de implementação.

NIST Web Metrics Testbed, desenvolvida pelo National Institute of Standards and Technology (zing.ncsl.nist.gov/WebTools/), abrange a seguinte coleção de ferramentas úteis disponíveis para download:

Web Static Analyzer Tool (WebSAT) – verifica o HTML da página da Web segundo diretrizes de utilização típica.

Web Category Analysis Tool (WebCAT) – permite ao engenheiro de utilização criar e executar uma análise de categoria Web.

Web Variable Instrumenter Program (WebVIP) – capacita um site a capturar um log de interação de usuário.

Framework for Logging Usability Data (FLUD) – implementa um formatador de arquivo e um analisador sintático (parser) para representação dos logs de interação do usuário.

VisVIP Tool – produz uma visualização 3D dos caminhos de navegação do usuário por meio de um site.

TreeDec – acrescenta auxílios de navegação às páginas de um site.

30.5 Métricas para código-fonte

A teoria de Halstead da “ciência do software” [Hal77] propôs as primeiras “leis” analíticas para programas de computador.¹⁴ Halstead atribuiu leis quantitativas ao desenvolvimento de software usando um conjunto de medidas primitivas que podem ser obtidas depois que o código é gerado – ou estimadas quando o projeto estiver completo. As medidas são:

n_1 = número de operadores distintos que aparecem em um programa

n_2 = número de operandos distintos que aparecem em um programa

N_1 = número total de ocorrências de operador

N_2 = número total de ocorrências de operando

Halstead usa essas medidas primitivas para desenvolver expressões para o tamanho global do programa, volume mínimo potencial para um algoritmo, o volume atual (número de bits necessários para especificar um programa), nível do programa (medida da complexidade do software), nível de linguagem (constante para uma dada linguagem) e outras características, como esforço de desenvolvimento, tempo de desenvolvimento e até um número projetado de falhas no software.

“O cérebro humano segue uma série de regras mais rígidas [para desenvolver algoritmos] do que ele próprio poderia imaginar.”

Maurice Halstead

¹³ A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nessa categoria.

¹⁴ Deve-se observar que as “leis” de Halstead geraram controvérsias substanciais, e muitos acreditam que a teoria na qual elas se baseiam apresenta falhas. No entanto, executaram-se verificações experimentais para algumas linguagens de programação (por exemplo, [Fel89]).

Operadores incluem todas as construções (constructs) de controle de fluxo, operações condicionais e matemáticas. Operandos abrangem todas as variáveis e constantes de programas.

Halstead mostra que o tamanho N pode ser estimado da seguinte maneira:

$$N = n_1 \log_2 n_1 + n_2 \log_2 n_2$$

e o volume do programa pode ser definido como:

$$V = N \log_2 (n_1 + n_2)$$

Devemos notar que V varia com a linguagem de programação e representa o volume de informações (em bits) necessário para especificar um programa.

Teoricamente, deve existir um volume mínimo para determinado algoritmo. Halstead define a razão de volume L como a razão entre o volume da forma mais compacta de um programa e o volume do programa real. Na realidade, L deve ser sempre menor do que 1. Em termos de medidas primitivas, a relação de volume pode ser expressa como

$$L = \frac{2}{n_1} \times \frac{n_2}{N_2}$$

O trabalho de Halstead é favorável à verificação experimental, e muitas pesquisas já foram feitas para investigar a ciência do software. Uma discussão desse trabalho está fora dos objetivos deste livro. Para mais informações, consulte [Zus90], [Fen91] e [Zus97].

30.6 Métricas para teste

A maioria das métricas propostas para teste se concentra no processo de teste e não nas características técnicas dos testes em si. Em geral, os testadores precisam se basear nas métricas de análise, projeto e código para guiá-los no projeto e execução dos casos de teste.

As métricas de projeto da arquitetura fornecem informações sobre a facilidade ou dificuldade associadas ao teste de integração (Seção 30.3) e sobre a necessidade de software de teste especializado (por exemplo, pseudocontroladores (*drivers*) e pseudocontrolados (*stubs*)). A complexidade ciclomática (uma métrica de projeto em nível de componente) depende do teste de caminho básico, um método de projeto de caso de teste apresentado no Capítulo 23. Além disso, a complexidade ciclomática pode ser usada para escolher módulos como candidatos para testes de unidade extensivos. Módulos com alta complexidade ciclomática são mais propensos a apresentar erro do que os de complexidade menor. Por essa razão, você deve despender esforços acima da média para descobrir erros nesses módulos antes de eles serem integrados em um sistema.

30.6.1 Métricas de Halstead aplicadas ao teste

O trabalho de teste pode ser estimado por meio de métricas obtidas das medidas de Halstead (Seção 30.5). Usando as definições para volume V de programa e nível PL , o trabalho Halstead e pode ser calculado como

$$PL = \frac{1}{[(n_1/2) \times (N_2/n_2)]} \quad (30.6a)$$

$$e = \frac{V}{PL} \quad (30.6b)$$

As métricas de teste se classificam em duas grandes categorias: (1) métricas que tentam prever o número provável de testes necessários em vários níveis de teste e (2) métricas que focalizam a abrangência do teste para determinado componente.

A porcentagem de trabalho de teste global a ser alocado a um módulo k pode ser estimada com a seguinte relação:

$$\text{Porcentagem de trabalho de teste } (k) = \frac{e(k)}{\sum e(i)} \quad (30.7)$$

onde $e(k)$ é calculado para o módulo k por meio das Equações (30.6), e a soma no denominador da Equação (30.7) é a soma do trabalho Halstead por todos os módulos do sistema.

30.6.2 Métricas para teste orientado a objetos

As métricas de projeto orientado a objetos apresentadas na Seção 30.3 fornecem uma indicação da qualidade do projeto. Elas também dão uma indicação geral do trabalho de teste necessário para testar um sistema orientado a objetos. Binder [Bin94b] sugere um conjunto amplo de métricas de projeto que têm influência direta sobre a “testabilidade” de um sistema orientado a objetos. As métricas consideram aspectos do encapsulamento e herança.

Falta de coesão em métodos (LCOM, lack of cohesion in methods).¹⁵ Quanto mais alto é o valor de LCOM, mais estados precisam ser testados para garantir que os métodos não gerem efeitos colaterais.

Porcentagem pública e protegida (PAP, percent public and protected). Atributos públicos são herdados de outras classes e, portanto, são visíveis àquelas classes. Atributos protegidos são acessíveis a métodos em subclasses. Essa métrica indica a porcentagem de atributos de classe públicos ou protegidos. Altos valores para PAP aumentam a probabilidade de efeitos colaterais entre classes porque atributos públicos e protegidos levam a um alto potencial de acoplamento.¹⁶ Os testes devem ser projetados para garantir que esses efeitos colaterais sejam descobertos.

Acesso público a membros de dados (PAD, public access to data members). Essa métrica indica o número de classes (ou métodos) que podem acessar atributos de outra classe, uma violação do encapsulamento. Altos valores de PAD levam a efeitos colaterais em potencial entre classes. Os testes devem ser projetados para garantir que esses efeitos colaterais sejam descobertos.

Número de classes-raiz (NOR, number of root classes). Essa métrica é uma contagem das hierarquias distintas de classe descritas no modelo de projeto. Devem ser desenvolvidos conjuntos de testes para cada classe-raiz e hierarquia de classe correspondente. À medida que o NOR aumenta, o trabalho de teste também aumenta.

Fan-in (FIN). Quando usado no contexto orientado a objeto, fan-in na hierarquia de herança é uma indicação de herança múltipla. $FIN > 1$ indica que uma classe herda seus atributos e operações de mais de uma classe-raiz. $FIN > 1$ deve ser evitado sempre que possível.

O teste orientado a objetos pode ser muito complexo. As métricas podem ajudá-lo a direcionar os recursos de teste para threads, cenários e pacotes de classes que são “suspeitos” com base nas características medidas. Utilize-as.

¹⁵ Veja uma descrição de LCOM na Seção 30.3.3.

¹⁶ Algumas pessoas fazem projetos nos quais nenhum dos atributos é público ou privado, ou seja, PAP = 0. Isso implica que todos os atributos devem ser acessados em outras classes via métodos.

Número de filhas (NOC, number of children) e altura da árvore de herança (DIT, depth of the inheritance tree).¹⁷ Conforme mencionamos no Capítulo 24, métodos de superclasse terão de ser novamente testados para cada subclasse.

30.7 Métricas para manutenção

Todas as métricas de software introduzidas neste capítulo podem ser usadas para o desenvolvimento de novo software e para manutenção do software existente. No entanto, existem também métricas projetadas explicitamente para atividades de manutenção.

A norma IEEE Std. 982.1-2005 [IEE05] sugere um *índice de maturidade de software* (SMI, *software maturity index*) que fornece uma indicação da estabilidade de um produto (com base nas alterações que ocorrem para cada versão do produto). São determinadas as seguintes informações:

M_T = número de módulos na versão atual

F_c = número de módulos na versão atual que foram alterados

F_a = número de módulos na versão atual que foram acrescentados

F_d = número de módulos da versão anterior que foram excluídos na versão atual

O índice de maturidade de software é calculado da seguinte maneira:

$$SMI = \frac{|M_T - (F_a + F_c + F_d)|}{M_T}$$

FERRAMENTAS DO SOFTWARE



Métricas de produto

Objetivo: ajudar os engenheiros de software no desenvolvimento de métricas com significado que avaliam os produtos gerados durante o projeto de análise e modelagem, geração de código-fonte e teste.

Mecanismos: as ferramentas dessa categoria abrangem um grande conjunto de métricas e são implementadas como uma aplicação independente ou (mais comumente) como funcionalidade que existe dentro das ferramentas para análise e projeto, codificação ou teste. Em muitos casos, a ferramenta métrica analisa uma representação do software (por exemplo, um modelo UML ou código-fonte) e desenvolve uma ou mais métricas como resultado.

Ferramentas representativas:¹⁸

Krakatau Metrics, desenvolvida pela Power Software (www.powersoftware.com/products), calcula complexidade, Halstead e métricas relacionadas a C/C++ e Java.

Rational Rose, distribuída pela IBM (<http://www-01.ibm.com/software/awdtools/developer/rose/>), é um conjunto de ferramentas para modelagem UML que incorpora uma série de características de análise de métricas.

RSM, desenvolvida pela M-Squared Technologies (<http://msquaredtechnologies.com/Resource-Standard-Metrics.html>), calcula uma grande variedade de métricas orientadas a código para C, C++ e Java.

Understand, desenvolvida pela Scientific Toolworks, Inc. (www.scitools.com), calcula métricas orientadas a código para uma variedade de linguagens de programação.

¹⁷ Veja uma descrição de NOC e DIT na Seção 30.3.3.

¹⁸ A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nessa categoria.

À medida que o SMI se aproxima de 1,0, o produto começa a estabilizar. O SMI pode ser usado também como uma métrica para planejamento de atividades de manutenção de software. O tempo médio para produzir uma versão de um software pode ser correlacionado com o SMI, e podem ser desenvolvidos modelos empíricos para o trabalho de manutenção.

30.8 Resumo

As métricas de software permitem avaliar quantitativamente a qualidade dos atributos internos do produto, possibilitando que a qualidade seja avaliada antes que ele seja criado. As métricas fornecem as informações necessárias para criar requisitos e modelos de projeto eficazes, código sólido e testes completos.

Para ser útil na prática, uma métrica de software deve ser simples e calculável, persuasiva, consistente e objetiva. Ela deve ser independente da linguagem de programação e fornecer um retorno eficaz.

Métricas para o modelo de requisitos focam-se em função, dados e comportamento – os três componentes do modelo. Métricas para projeto consideram aspectos de arquitetura, projeto em nível de componente e projeto de interface. Métricas de projeto de arquitetura consideram os aspectos estruturais do modelo de projeto. Métricas de projeto em nível de componente fornecem indicação da qualidade do módulo, estabelecendo medidas indiretas para coesão, acoplamento e complexidade. Métricas de projeto de interface de usuário fornecem indicação da facilidade com que uma GUI pode ser usada. Métricas para WebApp consideram aspectos da interface de usuário, bem como a estética da WebApp, conteúdo e navegação.

Métricas para sistemas orientados a objetos concentram-se em medições que podem ser aplicadas às características da classe e do projeto – localização, encapsulamento, ocultamento de informações, herança e técnicas de abstração de objeto – que tornam a classe única. O conjunto de métricas CK define seis métricas de software orientado à classe que focam-se na classe e na hierarquia de classes. O conjunto de métricas também desenvolve métricas para avaliar as colaborações entre classes e a coesão de métodos que residem em uma classe. Em um nível orientado a classe, o conjunto de métricas CK pode ser ampliado com métricas propostas por Lorenz e Kidd e o conjunto de métricas MOOD.

Halstead apresenta um conjunto fascinante de métricas em nível de código-fonte. Usando uma série de operadores e operandos presentes no código, a ciência do software fornece uma variedade de métricas que podem ser usadas para avaliar a qualidade do programa.

Poucas métricas de produto foram propostas para uso direto em teste e manutenção de software. No entanto, muitas outras podem ser empregadas para orientar o processo de teste e como mecanismo para avaliar a sustentabilidade de um programa de computador. Uma grande variedade de métricas orientadas a objetos foi proposta para avaliar a testabilidade de um sistema orientado a objetos.

Problemas e pontos a ponderar

30.1 A teoria de medidas é um tópico avançado que tem grande influência sobre as métricas de software. Usando [Zus97], [Fen91], [Zus90] ou fontes baseadas na Web, escreva um pequeno artigo que destaque os principais princípios da teoria das medidas. Projeto individual: desenvolva uma apresentação sobre o assunto e apresente para a sua classe.

30.2 Por que não é possível desenvolver uma métrica única e totalmente abrangente para complexidade de programa ou qualidade de programa? Tente criar uma medida ou métrica utilizada na prática que contrarie os atributos de métricas eficazes de software definidas na Seção 30.1.5.

30.3 Um sistema tem 12 entradas externas e 24 saídas externas, responde a 30 diferentes consultas externas, gerencia 4 arquivos de lógica interna e estabelece interface com 6 diferentes sistemas legados (6 EIFs). Todos esses dados são de complexidade média, e o sistema global é relativamente simples. Calcule FP para o sistema.

30.4 O software para o Sistema X tem 24 requisitos funcionais e 14 requisitos não funcionais. Qual a peculiaridade dos requisitos? E a totalidade?

30.5 Um grande sistema de informações tem 1.140 módulos. Há 96 módulos que executam funções de controle e coordenação e 490 módulos cuja função depende de processamento anterior. O sistema processa aproximadamente 220 objetos de dados, tendo cada um deles em média três atributos. Há 140 itens distintos de banco de dados e 90 segmentos distintos de banco de dados. Por fim, 600 módulos têm pontos distintos de entrada e saída. Calcule o DSQI para esse sistema.

30.6 Uma classe X tem 12 operações. A complexidade ciclomática foi calculada para todas as operações no sistema orientado a objetos, e o valor médio da complexidade de módulo é 4. Para a classe X, a complexidade para as operações 1 a 12 é 5, 4, 3, 3, 6, 8, 2, 2, 5, 5, 4, 4, respectivamente. Calcule os métodos ponderados por classe.

30.7 Desenvolva uma ferramenta de software que calcule a complexidade ciclomática para um módulo de linguagem de programação. Você pode escolher a linguagem.

30.8 Desenvolva uma pequena ferramenta de software que execute uma análise Halstead sobre código-fonte de linguagem de programação de sua escolha.

30.9 Um sistema legado tem 940 módulos. A última versão exigia que 90 desses módulos fossem alterados. Além disso, 40 novos módulos foram acrescentados e 12 módulos antigos foram removidos. Calcule o índice de maturidade do software para o sistema.

Leituras e fontes de informação complementares

Há uma quantidade surpreendentemente grande de livros dedicados às métricas de software, embora a maioria deles focalize métricas de processo e projeto, não incluindo métricas de produto. Jones e Bonsignour (*The Economics of Software Quality*, Addison-Wesley, 2011), Lanza e seus colegas (*Object-Oriented Metrics in Practice*, Springer, 2010) e Jones (*Applied Software Measurement: Global Analysis of Productivity and Quality*, McGraw-Hill, 2008) discutem as métricas orientadas a objetos e seu uso para analisar a qualidade de um projeto. Genero (*Metrics for Software Conceptual Models*, Imperial College Press, 2005) e Ejiogu (*Software Metrics*, BookSurge Publishing, 2005) apresentam uma grande variedade de métricas técnicas para casos de uso, modelos UML e outras representações de modelagem. Hutcheson (*Software Testing Fundamentals: Methods and Metrics*, Wiley, 2003) apresenta um conjunto de métricas para teste. Abran (*Software Metrics and Software Metrology*, Wiley-IEEE Computer Society, 2010), Kan (*Metrics and Models in Software Quality Engineering*, Addison-Wesley, 2ª ed., 2002), Fenton e

Pfleegeer (*Software Metrics: A Rigorous and Practical Approach*, Brooks-Cole Publishing, 1998) e Zuse [Zus97] escreveram tratados completos sobre métricas de produto.

Os livros de Card e Glass [Car90], Zuse [Zus90], Fenton [Fen91], Ejiogu [Eji91], Moeller e Paulish (*Software Metrics*, Chapman and Hall, 1993) e Hetzel [Het93] tratam de métricas de produto com certo detalhe. Oman e Pfleegeer (*Applying Software Metrics*, IEEE Computer Society Press, 1997) editaram uma antologia de importantes artigos sobre métricas de software.

O grupo International Function Point Users publicou um livro sobre o uso de métricas de função (*The IFPUG Guide to OT and Software Measurement*, Auerbach Publications, 2012). Métodos para estabelecer um programa de métricas e seus princípios subjacentes para medidas de software são considerados por Ebert e seus colegas (*Best Practices in Software Measurement*, Springer, 2004). Shepperd (*Foundations of Software Measurement*, Prentice-Hall, 1996) também trata da teoria de medidas com certo detalhe. As pesquisas atuais são apresentadas no *Proceedings of the Symposium on Software Metrics* (IEEE, publicado anualmente).

Um resumo abrangente de dezenas de métricas úteis de software é apresentado na [IEE93]. Em geral, foi resumida uma discussão de cada uma das métricas com as “primitivas” (medidas) essenciais necessárias para calcular a métrica e as relações apropriadas para efetuar o cálculo. Um apêndice fornece discussão e muitas referências.

Whitmire [Whi97] apresenta um tratamento abrangente e matematicamente sofisticado das métricas orientadas a objetos. Lorenz e Kidd [Lor94] e Hendersen-Sellers (*Object-Oriented Metrics: Measures of Complexity*, Prentice Hall, 1996) fornecem tratamentos dedicados às métricas orientadas a objetos.

Uma ampla gama de fontes de informação sobre métricas de software se encontra à disposição na Internet. Uma lista atualizada de referências relevantes para métricas de software pode ser encontrada no site: www.mhhe.com/pressman.

Esta página foi deixada em branco intencionalmente.

PARTE IV

Gerenciamento de projetos de software

Nesta parte do livro, você vai aprender as técnicas de gerenciamento necessárias para planejar, organizar, monitorar e controlar projetos de software. Estas questões são tratadas nos capítulos que seguem:

- Como as pessoas, os processos e os problemas devem ser gerenciados durante um projeto de software?
- Como as métricas de software podem ser usadas para gerenciar um projeto de software e o processo de software?
- Como uma equipe de software pode gerar estimativas confiáveis de trabalho, custo e duração do projeto?
- Que técnicas podem ser usadas para avaliar formalmente os riscos que podem ter um impacto sobre o sucesso do projeto?
- Como um gerente de projeto de software seleciona o conjunto de tarefas de engenharia de software?
- Como é criado um cronograma de projeto?
- Por que a manutenção e a reengenharia são tão importantes para os gerentes de engenharia de software e para os profissionais?

Uma vez respondidas essas questões, você estará mais bem preparado para gerenciar projetos de software, cumprindo seus prazos e entregando um produto de alta qualidade.

31

Conceitos de gerenciamento de projeto

Conceitos-chave

coordenação e	
comunicação.....	693
decomposição do	
problema.....	694
envolvidos.....	687
equipe de software.....	689
equipes ágeis.....	691
escopo do software.....	693
líderes de equipe.....	688
o princípio W ³ HH.....	698
peçoas.....	687
práticas vitais.....	699
produtos.....	693
projeto.....	697

No prefácio de seu livro sobre gerenciamento de projetos de software, Meller Page-Jones faz uma afirmação que pode ser repetida por muitos consultores da engenharia de software:

Visitei dezenas de lojas, boas e ruins, e observei muitos gerentes [de TI], também bons e ruins. Com muita frequência, testemunhei, horrorizado, esses gerentes dedicarem esforços em projetos que eram verdadeiros pesadelos, contorcendo-se em prazos impossíveis ou entregando sistemas que ultrajavam seus usuários e continuavam a devorar um bocadinho de tempo de manutenção.

O que Page-Jones descreve são sintomas que resultam em um leque de problemas técnicos e de gerenciamento. Entretanto, se um exame *post-mortem* fosse feito para todo projeto, chegaria-se a um tema constante: o gerenciamento do projeto foi fraco.

PANORAMA

ainda resta uma atividade bastante útil quando sistemas e projetos computacionais são desenvolvidos. Gerenciamento de projeto envolve planejamento, monitoração e controle de pessoas, processos e eventos que ocorrem à medida que o software evolui desde os conceitos preliminares até sua disponibilização operacional e completa.

Quem realiza? Todo mundo gerencia, até certo ponto, mas o escopo das atividades de gerenciamento varia entre os envolvidos em um projeto de software. Um engenheiro de software gerencia suas atividades diárias, planejando, monitorando e controlando as tarefas técnicas. Os gerenciadores de projeto planejam, monitoram e controlam o trabalho de uma equipe de engenheiros de software. Já um gerente sênior coordena a interface entre o “lado comercial” e os profissionais de software.

Por que é importante? Desenvolvimento de software é uma tarefa complexa, principalmente se envolver muitas pessoas trabalhando por um tempo longo. Por isso, os projetos de software precisam ser gerenciados.

O que é? Embora muitos de nós (em momentos mais críticos) adotemos a visão de Dilbert,

Quais são as etapas envolvidas? Entenda os 4 Ps: pessoas, produto, processo e projeto. As pessoas devem ser organizadas para o trabalho de desenvolvimento de forma efetiva. A comunicação com o cliente e com outros envolvidos deve ocorrer para que o escopo e os requisitos do produto sejam compreendidos. Deve ser selecionado um projeto adequado para as pessoas e para o produto. O projeto deve ser planejado com base na estimativa do esforço e do prazo para a realização das tarefas: definindo artefatos, estabelecendo pontos de verificação de qualidade e identificando mecanismos para monitorar e controlar o trabalho no plano de projeto.

Qual é o artefato? Assim que as atividades de gerenciamento iniciam, faz-se um plano de projeto. Define-se o processo e as tarefas a serem conduzidas, as pessoas que realizarão o trabalho e os mecanismos de avaliação de riscos, de controle de alterações e de avaliação da qualidade.

Como garantir que o trabalho foi realizado corretamente? Nunca se está completamente seguro de que o plano de projeto está correto até que se entregue um produto de alta qualidade, no prazo e dentro do orçamento. Entretanto, um gerente de projeto age corretamente quando estimula o pessoal de desenvolvimento a trabalhar como uma verdadeira equipe, concentrando-se nas necessidades do cliente e na qualidade do produto.

Neste capítulo e nos Capítulos 32 a 37, serão apresentados conceitos que conduzem a um gerenciamento de projetos eficaz. Aqui, trataremos dos princípios e dos conceitos básicos do gerenciamento de projetos. No Capítulo 32, abordaremos a medição de projeto e de processo, base para a tomada de decisões de um gerenciamento eficiente. Técnicas utilizadas para estimar custos estão no Capítulo 33. O Capítulo 34 auxiliará na definição de um cronograma realista do projeto. As atividades de gerenciamento que levam a uma efetiva monitoração e mitigação de riscos estão no Capítulo 35. O Capítulo 36 considera a manutenção e a reengenharia e discute questões de gerenciamento encontradas quando se lida com sistemas legados. Por último, o Capítulo 37 discute técnicas para estudar e melhorar os processos de engenharia de software de sua equipe.

31.1 O espectro de gerenciamento

O gerenciamento eficiente do desenvolvimento de software se concentra nos 4 Ps: pessoas, produto, processo e projeto. Essa ordem não é arbitrária. O gerente que se esquecer de que o trabalho do engenheiro de software consiste em esforço humano nunca terá sucesso no gerenciamento de projeto. Da mesma forma, o que não estimula a ampla comunicação entre os envolvidos no início da evolução de um produto corre o risco de desenvolver uma solução elegante para o problema errado. Um gerente que preste pouca atenção ao processo se arrisca a inserir métodos e ferramentas técnicas competentes em um vácuo. Aquele que embarca sem um plano de projeto sólido compromete o sucesso do projeto.

31.1.1 Pessoas

Desde os anos 1960, debate-se a valorização da cultura de ter pessoal de desenvolvimento motivado e de alto nível. Realmente, recursos humanos é um fator de tal importância que o Software Engineering Institute (SEI) desenvolveu um *Modelo de Maturidade de Capacitação de Pessoas* – o People-CMM (*People Capability Maturity Model*) – em reconhecimento ao fato de que: “Toda organização precisa aprimorar continuamente sua habilidade de atrair, desenvolver, motivar, organizar e reter a força de trabalho necessária para atingir os objetivos estratégicos de seus negócios” [Cur 01].

O People-CMM define as seguintes práticas-chave para o pessoal de software: formação de equipe, comunicação, ambiente de trabalho, gerenciamento do desempenho, treinamento, compensação, análise de competência e de desenvolvimento, desenvolvimento de carreira, do grupo de trabalho, da cultura e da equipe e outros. Em organizações que conseguem altos níveis de maturidade e capacidade, o People-CMM tem maior probabilidade de implementar práticas de gerenciamento de software eficazes.

O People-CMM é um parceiro do modelo de integração para maturidade e capacidade em software: o SW – CMMi (Capítulo 37) conduz as organizações para a criação de um processo de software maduro. As questões relacionadas ao gerenciamento de recursos humanos e à estruturação dos projetos de software serão discutidas posteriormente neste capítulo.

Aqueles que aderem à filosofia do processo ágil (Capítulo 5) argumentam que ele é mais enxuto do que os outros. Isso pode ser verdade, mas ainda assim eles têm um processo, e um software de engenharia ágil ainda exige disciplina.

31.1.2 O produto

Antes de traçarmos um plano de projeto, devemos estabelecer os objetivos do produto e seu escopo, considerar as soluções alternativas e identificar as restrições técnicas e de gerenciamento. Sem essas informações, é impossível definir de modo razoável (e preciso) a estimativa de custo, a avaliação efetiva dos riscos, a análise realista das tarefas do projeto ou um cronograma gerenciável do projeto que forneça a indicação significativa de progresso das atividades.

Como desenvolvedores, devemos nos reunir com os envolvidos no software para definir os objetivos e o escopo do produto. Em muitos casos, tal atividade se inicia como parte da engenharia do sistema ou da engenharia do processo de negócio e continua como a primeira etapa da engenharia de requisitos do software (Capítulo 8). Os objetivos identificam as metas gerais do produto (do ponto de vista dos envolvidos) sem considerar como tais metas serão alcançadas. O escopo identifica os principais dados, funções e comportamentos que caracterizam o produto e, mais importante, tenta mostrar as fronteiras e limitações dessas características de maneira quantitativa.

Entendidos os objetivos e o escopo, consideram-se soluções alternativas. Apesar de se discutir muito pouco os detalhes, as alternativas capacitam os gerentes e desenvolvedores a selecionar a melhor estratégia, dadas as restrições impostas pelos prazos de entrega, restrições orçamentárias, disponibilidade de pessoal, interfaces técnicas e uma infinidade de outros fatores.

31.1.3 O processo

Um processo de software (Capítulos 3 a 5) fornece a metodologia por meio da qual pode ser estabelecido um plano de projeto abrangente para o desenvolvimento de software. Poucas atividades metodológicas são aplicáveis a todos os projetos de software, independentemente do seu tamanho e complexidade. Uma quantidade de diferentes conjuntos de atividades – tarefas, pontos de controle, artefatos de software e pontos de garantia de qualidade possibilitam que as atividades metodológicas sejam adaptadas às características do projeto de software e aos requisitos de equipe. Por fim, as atividades de apoio, como Garantia de Qualidade de Software, Gerenciamento de Configuração e de Medições, sobrepõem-se ao modelo do processo. Essas são independentes de quaisquer das atividades metodológicas e ocorrem ao longo do processo.

31.1.4 O projeto

Conduzimos projetos com planejamento e com controle por uma única e principal razão: é a única maneira de administrar a complexidade. E, mesmo assim, as equipes de software têm de se esforçar. Em um estudo de 250 grandes projetos de software entre 1998 e 2004, Caper Jones [Jon 04] constatou que “perto de 25 obtiveram sucesso em cumprir cronograma, custos e objetivos quanto à qualidade. Em torno de 50 apresentaram atrasos ou retardamentos abaixo de 35%, enquanto 175 projetos tiveram atrasos e retardamentos sérios ou não foram concluídos”. Apesar de que, atualmente, a taxa de sucesso nos

“Um projeto é como uma rodovia. Alguns são simples e rotineiros, como dirigir até uma loja em um dia ensolarado. Entretanto, a maioria dos que valem ser realizados parece mais com dirigir um caminhão à noite em uma serra cheia de curvas.”

**Cem Kaner,
James Bach e
Bret Pettichord**

projetos de software possa ter melhorado de algum modo, a taxa de falhas em projeto permanece mais alta do que deveria.¹

Para evitar falha de projeto, o gerente e os engenheiros que desenvolvem o produto devem evitar uma série de sinais de alerta comuns, devem entender os fatores críticos de sucesso que conduzem ao bom gerenciamento e desenvolver uma estratégia de senso comum no que se refere a planejamento, monitoramento e controle de projeto. Cada um desses itens é discutido na Seção 31.5 e no capítulo seguinte.

31.2 As pessoas

Pessoas constroem software de computador, e os projetos são bem-sucedidos porque pessoas bem treinadas e motivadas fazem as coisas. Todos nós, de vice-presidentes de engenharia a desenvolvedores mais simples, com frequência não damos o devido valor às pessoas. Os gerentes afirmam que pessoal é prioridade; entretanto, suas ações às vezes desmentem suas palavras. Na próxima seção, examinamos os envolvidos que participam do processo de software e a maneira pela qual são organizados para desempenhar ações de engenharia de software.

31.2.1 Os envolvidos

O processo de software (e todo o projeto de software) é formado por envolvidos (*stakeholders*) que podem ser categorizados em um de cinco grupos:

1. *Gerentes seniores*, que definem os problemas do negócio que, com frequência, têm influência significativa no projeto.
2. *Gerentes de projeto (técnicos)*, que devem planejar, motivar, organizar e controlar os programadores que executam o trabalho de software.
3. *Profissionais*, que têm as habilidades técnicas necessárias para desenvolver a engenharia de um produto ou aplicação.
4. *Clientes*, que especificam os requisitos para o software a serem submetidos ao processo de engenharia, e outros envolvidos que têm interesses periféricos no produto final.
5. *Usuários*, que interagem com o software depois que é disponibilizado para uso operacional.

Todo projeto de software é composto por pessoas que se enquadram nessa taxonomia.² Para ser eficiente, a equipe do projeto deve estar organizada para maximizar cada capacidade e habilidade dos profissionais. E essa é a tarefa do líder da equipe.

¹ Dadas essas estatísticas, pode-se questionar como os impactos computacionais evoluem exponencialmente. Parte da resposta, achamos, é que um significativo número dos projetos que falham nas primeiras tentativas é preconcebido com falhas. Clientes perdem o interesse muito rapidamente (porque o que eles pediram não é tão importante quanto acharam que era em princípio), e os projetos são cancelados.

² Quando se desenvolvem WebApps ou aplicativos móveis, outras pessoas não especialistas em software são envolvidas na criação de conteúdos.

31.2.2 Líderes de equipe

Gerenciamento de projeto é uma atividade que envolve muitas pessoas e, por essa razão, programadores competentes em geral resultam em maus líderes de equipe. Eles simplesmente não possuem a mistura certa de habilidade com pessoas. Ainda assim, como Edgemon afirma: “infelizmente e, com demasiada frequência, parece que os indivíduos só assumem o papel de gerente de projeto e se tornam gerentes de projetos por acidente” [Ed 95].

Em um excelente livro sobre liderança técnica, Jerry Weinberg [Wei 86] sugere um modelo MOI de liderança:

Motivação. A capacidade de estimular o pessoal técnico a produzir com o melhor da sua habilidade.

Organização. A habilidade para moldar os processos já existentes (ou inventar novos) que vão capacitar o conceito inicial a ser transformado em um produto final.

Ideias ou inovação. A habilidade de estimular pessoas a criar e serem criativas, mesmo quando estiverem trabalhando de acordo com padrões estabelecidos para um produto ou aplicativo de software específico.

Weinberg sugere que líderes de projeto bem-sucedidos aplicam um estilo de gerenciamento de solução de problemas. Isto é, um gerente de projeto de software deve se concentrar em entender o problema a ser resolvido, administrando o fluxo de ideias, e, ao mesmo tempo, deixar claro para todos da equipe (por meio de palavras e, muito mais importante, de ações) que a qualidade conta muito e que não deve ser comprometida.

Outra visão [Edg 95] das características que definem um gerenciamento de projeto eficaz concentra-se em quatro traços essenciais:

Solução de problemas. Um gerente de projeto eficaz sabe diagnosticar os problemas técnicos e organizacionais mais relevantes, sistematicamente estrutura uma solução ou motiva apropriadamente outros desenvolvedores a buscar a solução, põe em prática as lições aprendidas de projetos anteriores para novas situações e permanece suficientemente flexível para mudar de direção, caso as tentativas iniciais para a solução de problemas tenham sido infrutíferas.

Identidade gerencial. Um bom gerente de projeto deve assumir a responsabilidade pelo projeto. Deve ter a confiança para assumir o controle quando necessário e deve assegurar que permitirá ao pessoal técnico seguir os seus instintos.

Realizações. Um gerente competente deve recompensar iniciativas e realizações para otimizar a produtividade de uma equipe. Deve demonstrar, por meio de seus próprios atos, que não haverá punição por se assumir riscos controlados.

Influência e formação da equipe. Um gerente eficiente deve ser capaz de “ler” pessoas, de compreender sinais verbais e não verbais e reagir às necessidades das pessoas que estão enviando esses sinais. O gerente deve permanecer controlado em situações de alto estresse.

O que buscamos ao selecionar alguém para liderar um projeto de software?

“Resumindo, um líder é aquele que sabe aonde quer ir, que se levanta e vai.”

John Erskine

31.2.3 A equipe de software

Há quase tantas estruturas organizacionais humanas para desenvolvimento de software quanto há organizações que desenvolvem software. Para melhor ou pior, a estrutura organizacional não pode ser facilmente modificada. Preocupações com os efeitos práticos e políticos da mudança organizacional não fazem parte do escopo da responsabilidade do gerente de projeto de software. Entretanto, a organização do pessoal diretamente envolvido em um novo projeto está ao alcance do gerente do projeto.

A melhor estrutura de equipe depende do estilo de gerenciamento das organizações, da quantidade de pessoas na equipe, de seus níveis de habilidade e do grau de dificuldade geral do problema. Mantei [Man81] descreve sete fatores que devem ser considerados ao planejarmos a estrutura da equipe de engenharia de software: (1) dificuldade do problema a ser resolvido; (2) “tamanho” do programa (ou programas) resultante, em linhas de código ou pontos de função; (3) tempo durante o qual a equipe vai permanecer reunida (vida da equipe); (4) até que ponto o problema pode ser modularizado; (5) qualidade e confiabilidade exigidas do sistema a ser construído; (6) flexibilidade da data de entrega; e (7) grau de sociabilidade (comunicação) exigida para o projeto.

Constantine [Con93] sugere quatro “paradigmas organizacionais” para equipes de engenharia de software:

1. O *paradigma fechado* estrutura uma equipe em termos de uma hierarquia de autoridade tradicional. Tais equipes podem trabalhar bem em produção de software bastante similar a esforços de épocas passadas, mas se mostram menos propícias a ser inovadoras trabalhando sob o paradigma fechado.
2. O *paradigma randômico* estrutura uma equipe vagamente e depende da iniciativa individual de seus membros. Quando for necessária uma inovação ou um avanço tecnológico, as equipes que seguem o paradigma randômico se destacarão. Mas essas equipes podem brigar, quando for exigido um “desempenho ordenado”.
3. O *paradigma aberto* atém-se a estruturar a equipe de maneira que consiga alguns dos controles associados ao paradigma fechado, mas também muito da inovação que ocorre ao se usar o paradigma randômico. O trabalho é feito de forma colaborativa, com forte comunicação e tomada de decisão baseada no consenso, executando com as características marcantes das equipes de paradigma aberto. As estruturas das equipes de paradigma aberto são bem adequadas à solução de problemas complexos, mas não conseguem desempenhar tão bem quanto outras equipes.
4. O *paradigma sincronizado* baseia-se na compartimentalização natural de um problema e organiza os membros da equipe para trabalhar nas partes do problema com pouca comunicação entre si.

Como um comentário histórico, uma das mais antigas organizações de equipe de software foi paradigma de uma estrutura fechada, denominada originalmente *equipe com um programador-chefe* (principal). Essa estrutura foi primeiramente proposta por Harlan Mills e descrita por Baker [Bak 72]. Como um contraponto para a estrutura da equipe de programadores-chefe, o paradigma randômico de Constantine [Con 93] sugere a primeira equipe

“Nem todo grupo é uma equipe, nem toda equipe é eficiente.”

Glenn Parker

Quais fatores devem ser considerados quando a estrutura de uma equipe de software já está estabelecida?

Quais são as opções quando se tem de definir a estrutura de uma equipe de software?

“Se deseja ser incrementalmente melhor, seja competitivo. Se deseja ser exponencialmente melhor, seja cooperativo.”

Autor desconhecido

criativa, cuja estratégia de trabalho pode ser mais bem denominada *anarquia inovadora*. Embora a abordagem de espírito livre para o trabalho de software tenha apelo, a energia da criatividade direcionada para uma equipe de alto desempenho deve ser o objetivo central de uma organização de engenharia de software. Para se obter uma equipe de alto desempenho: os membros da equipe devem confiar uns nos outros, a distribuição de habilidades deve ser adequada ao problema e estrelismos devem ser excluídos da equipe para manter a coesão do grupo.

Seja qual for a organização da equipe, o objetivo de todo gerente de projeto é ajudar a montar uma equipe coesa. Em seu livro *Peopleware*, De Marco e Listes [DeM 98] discorrem sobre esse tema:

O que é equipe
"consistente"?

Tendemos a utilizar a palavra *equipe* de forma constante e vaga na área de negócios, denominando qualquer grupo de profissionais designados a trabalhar juntos de "equipe". Muitos desses grupos, porém, não se assemelham a equipes. Eles não têm uma definição comum de sucesso nem um espírito de equipe identificável. O que falta é um fenômeno que se denomina *consistência*.

Uma equipe consistente é um grupo de pessoas tão fortemente unidas que o todo é maior do que a soma das partes.

Uma vez que uma equipe começa a ser consistente, a probabilidade de sucesso aumenta muito. A equipe pode se tornar imbatível, um rolo compressor de sucesso... Não é preciso gerenciá-la do modo tradicional e, com certeza, ela não precisará ser motivada. Ela adquire ímpeto.

De Marco e Lister afirmam que os membros de equipes consistentes são mais produtivos e mais motivados do que a média. Compartilham um objetivo comum, uma cultura comum e, em muitos casos, um "senso de elitização" que os torna únicos.

Por que motivo
equipes falham em
ser consistentes?

Entretanto, nem todas as equipes tornam-se consistentes. De fato, muitas sofrem do que Jackman [Jac 98] chama de "toxicidade de equipe". Ela define cinco fatores que "promovem um ambiente em equipe potencialmente tóxico": (1) uma atmosfera de trabalho frenética, (2) alto grau de frustração que causa atrito entre os membros da equipe, (3) um processo de software "fragmentado ou coordenado de forma deficiente", (4) uma definição nebulosa das funções dentro da equipe de software e (5) a contínua e repetida exposição a falhas.

Para evitar um ambiente de trabalho frenético, o gerente de projeto deve estar certo de que a equipe tem acesso a todas as informações necessárias para realizar o trabalho e de que as metas e objetivos prioritários (papéis) não devem ser alterados uma vez estabelecidos, a não ser que absolutamente necessário. Uma equipe pode evitar frustrações se lhe for oferecida, tanto quanto possível, responsabilidade para tomada de decisão. Um processo inapropriado (por exemplo, tarefas pesadas ou desnecessárias ou artefatos mal selecionados) pode ser evitado por meio da compreensão do produto a ser desenvolvido, das pessoas que realizam o trabalho e pela permissão para que a equipe selecione o modelo de processo. A própria equipe deve estabelecer seus mecanismos de responsabilidades (revisões técnicas³ são excelentes meios para

³ Revisões técnicas são tratadas em detalhes no Capítulo 20.

conseguir isso) e deve definir uma série de estratégias para correções quando um membro falhar em suas atribuições. E, por fim, a chave para evitar uma atmosfera de derrota consiste em estabelecer técnicas baseadas na equipe voltadas para realimentação (*feedback*) e solução de problemas.

Além das cinco toxinas descritas por Jackman, uma equipe de software frequentemente despende esforços com as diferentes características de seus membros. Uns são extrovertidos, outros introvertidos. Uns coletam informações intuitivamente, destilando conceitos amplos de fatos disparatados. Outros processam informações linearmente, coletando e organizando detalhes mínimos dos dados fornecidos. Alguns se sentem confortáveis tomando decisões apenas quando um argumento lógico e ordenado for apresentado. Outros são intuitivos, acostumados a tomar decisões baseadas em percepções. Certos desenvolvedores querem um cronograma detalhado, preenchido por tarefas organizadas que os tornem aptos a chegar ao encerramento de algum elemento do projeto. Outros ainda preferem um ambiente mais espontâneo, no qual resultados e questões abertas já estarão bons. Alguns trabalham arduamente para conseguir que as etapas sejam concluídas bem antes da data estabelecida, evitando, portanto, estresse à medida que a data-limite se aproxima, enquanto outros são estimulados pela correria em fazer até a data e minutos-limite. Uma discussão detalhada sobre a psicologia envolvida nessas características e as formas pelas quais um líder de equipe hábil pode auxiliar pessoas com traços opostos a trabalhar juntas está além dos objetivos deste livro.⁴ Entretanto, é importante notar que o reconhecimento das forças humanas é o primeiro passo em direção à criação de equipes consistentes.

"Faça ou não faça. Não existe o tentar."

**Yoda, personagem
de Guerra nas
Estrelas**

31.2.4 Equipes ágeis

Muitas empresas defendem o desenvolvimento de software ágil (Capítulo 5) como um antídoto para muitos problemas que se alastraram nas atividades de projeto de software. Relembrando, a filosofia ágil enfatiza a satisfação do cliente e a entrega prévia incremental de software, pequenas equipes de projetos altamente motivadas, métodos informais, mínimos artefatos de engenharia de software e total simplicidade de desenvolvimento.

A pequena e altamente motivada equipe de projeto, também denominada *equipe ágil*, adota muitas das características das equipes de software bem-sucedidas discutidas na seção anterior e evita muitas das toxinas geradoras de problemas. Entretanto, a filosofia ágil enfatiza competência individual (membro da equipe), casada com colaboração em grupo, como fatores críticos de sucesso da equipe. Cockburn e Highsmith [Coc01a] observam isso ao escreverem:

Se as pessoas do projeto forem boas o suficiente, podem usar quase qualquer processo e realizar a sua missão. Se elas não forem boas o suficiente, nenhum processo vai reparar a sua inadequação. "Pessoas são o trunfo do processo" é uma forma de dizer isso. Entretanto, falta de suporte ao desenvolvedor e ao usuário pode ma-

⁴ Uma excelente introdução a essas questões, no que se refere a equipes de projeto de software, pode ser encontrada em [Fer 98].

tar um projeto – “política é o trunfo das pessoas”. Suporte inadequado pode fazer com que mesmo os bons fracassem na realização de seus trabalhos...

Para fazer uso eficiente das competências de cada membro da equipe e fomentar a colaboração efetiva ao longo do projeto, as equipes ágeis se *auto-organizam*. A equipe que se auto-organiza não mantém, necessariamente, uma estrutura de equipe única, mas usa elementos dos paradigmas randômico, aberto e sincronizado de Constantine, discutidos na Seção 31.2.3.

Muitos modelos ágeis de processo (por exemplo, Scrum) dão à equipe ágil autonomia para fazer o gerenciamento do projeto e tomar decisões técnicas necessárias à conclusão do trabalho. O planejamento é mantido em um nível mínimo, e a equipe tem permissão para escolher sua própria estratégia (por exemplo, processo, método, ferramentas), limitada somente pelos requisitos do negócio e pelos padrões organizacionais. Conforme o projeto prossegue, a equipe se auto-organiza, concentrando-se em competências individuais para maior benefício do projeto em determinado ponto do cronograma. Para tanto, uma equipe ágil pode realizar reuniões de pessoal diariamente para coordenar e sincronizar as atividades que devem ser realizadas naquele dia.

Com base nas informações obtidas durante essas reuniões, a equipe adapta sua estratégia para incrementar o trabalho. A cada dia, auto-organizações e colaborações contínuas conduzem a equipe em direção a um incremento de software concluído.

Uma equipe ágil é aquela que se auto-organiza, tem autonomia para planejar e toma decisões técnicas.

CASASEGURA



Estrutura da equipe

Cena: Escritório de Doug Miller antes do início do projeto do software *CasaSegura*.

Atores: Doug Miller (gerente da equipe de engenharia de software do *CasaSegura*) e Vinod Raman, Jamie Lazar e outros membros da equipe.

Conversa:

Doug: Vocês deram uma olhada no informativo preliminar do *CasaSegura* que o departamento de marketing preparou?

Vinod (balançando afirmativamente a cabeça e olhando para seus companheiros de equipe): Sim. Mas temos muitas dúvidas.

Doug: Vamos deixar isso de lado por um momento. Gostaria de conversar sobre como vamos estruturar a equipe, quem será responsável pelo quê...

Jamie: Eu realmente comprei a ideia da filosofia ágil, Doug. Acho que devemos ser uma equipe que se auto-organiza.

Vinod: Concordo. Devido ao cronograma apertado, ao grau de incerteza e ao fato de todos sermos realmente competentes (risos), parece ser o caminho certo a seguir.

Doug: Tudo bem por mim, mas vocês conhecem o procedimento.

Jamie (sorrindo e falando como se estivesse recitando): “Tomamos decisões táticas sobre quem faz o que e quando, mas é nossa responsabilidade ter o produto pronto sem atraso”.

Vinod: E com qualidade.

Doug: Exatamente. Mas lembrem-se de que há restrições. O marketing define os incrementos de software a serem desenvolvidos – consultando-nos, é claro.

Jamie: E?

Doug: E usaremos UML como estratégia de modelagem.

Vinod: Mas mantenham a documentação adicional em um mínimo absoluto.

Doug: Quem é meu contato?

Jamie: Decidimos que Vinod será o coordenador técnico – ele tem mais experiência; portanto, será o intermediário, mas sinta-se à vontade para conversar com qualquer um de nós.

Doug (rindo): Não se preocupe, farei isso.

31.2.5 Questões de comunicação e coordenação

Há muitas razões para que o projeto de software tenha problemas. A escala de muitos esforços de desenvolvimento é ampla, conduzindo a complexidade, confusão e dificuldades significativas na coordenação dos membros da equipe. Incertezas são comuns, resultando em uma contínua cadeia de alterações que racham a equipe de projeto. Interoperabilidade torna-se um aspecto-chave para muitos sistemas. Novo software deve se comunicar com os softwares existentes e se ajustar às restrições predefinidas impostas pelo sistema ou pelo produto.

Essas características do software moderno – escala, incerteza e interoperabilidade – são fatos da vida. Para lidar efetivamente com eles, devem-se estabelecer métodos eficazes para coordenar as pessoas que realizam o trabalho. Para tanto, devem-se estabelecer mecanismos para comunicação formal e informal entre os membros da equipe. A comunicação formal é realizada por meio de “comunicação escrita, reuniões estruturadas e de outros canais de comunicação relativamente não interativos e impessoais” [Kra 95]. A comunicação informal é mais pessoal. Os membros de uma equipe de software compartilham ideias de forma assistemática, solicitam ajuda conforme surgem os problemas e interagem uns com os outros diariamente.

“Propriedades coletivas nada mais são do que uma ilustração da ideia de que os produtos devem ser atribuídos à equipe (ágil) e não aos indivíduos que compõem a equipe.”

Jim Highsmith

31.3 O produto

Um gerente de projeto de software confronta-se com um dilema sempre que inicia um projeto. É preciso ter estimativas quantitativas e um plano organizado, mas informações sólidas ainda não estão disponíveis. Uma análise detalhada dos requisitos de software fornece as informações necessárias para as estimativas, mas as análises frequentemente levam semanas ou até mesmo meses para estarem concluídas. Pior ainda, os requisitos podem ser fluidos, mudando regularmente conforme o projeto prossegue. Ainda assim, um planejamento é necessário agora!

Gostando-se ou não, deve-se examinar o produto e o problema que se pretende solucionar logo no início do projeto. No mínimo, o escopo do produto deve ser estabelecido e delimitado.

31.3.1 Escopo do software

A primeira atividade do gerenciamento do projeto de software consiste em determinar o *escopo do software*, que é definido respondendo-se às seguintes questões:

Contexto. Como o software a ser desenvolvido se ajusta a um sistema, produto ou contexto de negócio maior e quais são as restrições impostas como resultado do contexto?

Objetivos da informação. Quais objetos de dados visíveis ao cliente são produzidos como saída do software? Quais objetos de dados são necessários como entrada?

Função e desempenho. Que função que o software executa para transformar os dados de entrada em dados de saída? Há quaisquer características especiais de desempenho a serem tratadas?

Se não puder estabelecer uma característica do software que pretende construir, liste-a como um risco de projeto (Capítulo 35).

O escopo do projeto de software não deve ter ambiguidades e deve ser compreensível tanto no nível gerencial quanto no nível técnico. Deve-se estabelecer o escopo do software. Isto é, dados quantitativos (por exemplo, número de usuários simultâneos, tamanho da mala direta, tempo máximo de resposta) são estabelecidos explicitamente, restrições e/ou limitações (por exemplo, o custo do produto restringe o tamanho da memória) são determinadas, e fatores mitigadores (por exemplo, algoritmos desejados são bem compreendidos e avaliados em Java) são descritos.

31.3.2 Decomposição do problema

Para elaborar um planejamento de projeto razoável, deve-se decompor o problema. Para tanto, utiliza-se uma lista de funções ou casos de uso, ou então, para trabalho ágil, histórias de usuário.

A decomposição do problema, também chamada de *elaboração do problema* ou *particionamento*, consiste em uma atividade que ocupa o centro da análise de requisitos de software (Capítulos 8 a 11). Durante a atividade de escopo, não se busca decompor completamente o problema. Em vez disso, aplica-se a decomposição em duas áreas vitais: (1) na funcionalidade e no conteúdo (informação) que deve ser entregue e (2) no processo que será utilizado para entregar o software.

As pessoas tendem a aplicar a estratégia de dividir para conquistar quando confrontadas por um problema complexo. Ao simplificarmos um problema complexo, ele é particionado em pequenas questões mais gerenciáveis. Essa é a estratégia a aplicar no início do planejamento do projeto. Funções de software, descritas no estabelecimento do escopo, são avaliadas e refinadas para proporcionar mais prioridades de detalhes logo no início das estimativas (Capítulo 33). Como as estimativas de custo e cronograma são ambas funcionalmente orientadas, muitas vezes é aconselhável certo grau de decomposição. Do mesmo modo, o conteúdo principal ou objetos de dados são decompostos em suas partes constituintes, propiciando compreensão razoável da informação a ser gerada pelo software.

31.4 O processo

As atividades metodológicas (Capítulo 2) que caracterizam o processo de software são aplicáveis a todos os projetos de software. A dificuldade está em selecionar o modelo de processo adequado para o software a ser desenvolvido (pelo processo de engenharia) por sua equipe.

A equipe deve decidir qual modelo de processo será mais apropriado para (1) os clientes que solicitaram o produto e os profissionais que realizarão o trabalho; (2) as próprias características do produto; e (3) o ambiente de projeto no qual a equipe trabalhará. Quando um modelo de processo é selecionado, a equipe define o planejamento preliminar do projeto com base no conjunto de atividades metodológicas de processo. Uma vez definido o planejamento preliminar, inicia-se o particionamento (decomposição) do projeto. Ou seja, deve ser criado um planejamento completo que reflita as tarefas necessárias para preencher as atividades metodológicas. Essas atividades serão abordadas brevemente nas seções seguintes, e uma visão mais detalhada será apresentada no Capítulo 33.

31.4.1 Combinando o produto e o processo

O projeto começa com a junção do produto com o processo. Cada função a ser desenvolvida por engenharia deve passar pelas atividades metodológicas definidas pela organização responsável pelo software.

Suponha que a organização tenha adotado as atividades metodológicas genéricas – **comunicação, planejamento, modelagem, construção e entrega** – discutidas no Capítulo 2. Os membros da equipe que trabalham em uma funcionalidade do produto aplicarão cada uma das atividades de framework à função. Em essência, será criada uma matriz similar à Figura 31.1. Cada função principal do produto (a figura mostra a função do software processador de texto discutido anteriormente) é listada na coluna da esquerda. As atividades metodológicas são relacionadas (indicadas) na parte superior das colunas. As tarefas de trabalho de engenharia (para cada atividade metodológica) são incluídas nas linhas seguintes.⁵ O trabalho do gerente de projeto e de outros membros da equipe é estimar as necessidades de recursos para cada célula da matriz, datas de início e de fim para as tarefas associadas a cada célula e artefatos a serem produzidos como consequência de cada ação. Tais atividades são examinadas no Capítulo 26.

31.4.2 Decomposição do processo

A equipe de software deve ter um grau de flexibilidade ao escolher o modelo de processo de software mais adequado ao projeto e às tarefas de engenharia de software que fazem parte do modelo selecionado. Um projeto relativamente pequeno poderia ser mais bem realizado por meio de abordagem sequencial linear. Se o prazo final estiver bem apertado a ponto de a funcionalidade completa não poder ser razoavelmente entregue, uma estratégia incremental poderá ser a melhor indicação. Similarmente, projetos com outras características (por exemplo, requisitos indefinidos, tecnologias avançadas recentes,

A metodologia de processo estabelece um esquema para o planejamento do projeto. Ela é adaptada pela alocação de um conjunto adequado de tarefas para aquele projeto.

ATIVIDADES METODOLÓGICAS COMUNS DO PROCESSO	comunicação	planejamento	modelagem	construção	entrega
Tarefas de Engenharia de Software					
Funções de produto					
Entrada de texto					
Edição e formatação					
Edição de cópia automática					
Capacidade de formatação do layout da página					
Indexação e sumário automáticos					
Gerenciamento de arquivo					
Elaboração dos documentos impressos					

FIGURA 31.1 Integração do problema com o processo.

⁵ Observe que as tarefas devem ser adaptadas às necessidades específicas do projeto, com base em diversos critérios de adaptação.

clientes difíceis, potencial de reutilização significativo) levarão à escolha de outros modelos de processo.⁶

Uma vez escolhido o processo, a metodologia é adaptada ao projeto. Em todo caso, a metodologia de processo genérica discutida anteriormente pode ser usada. Ela vai funcionar para modelos lineares, iterativos e incrementais e até mesmo para os modelos de montagem por componentes ou por paralelismo. A metodologia do processo é invariável e serve como base para todo o trabalho realizado por uma organização de software.

Entretanto, as tarefas concretas variam. A decomposição do processo se inicia quando o coordenador do projeto pergunta: “Como realizamos essa atividade metodológica?”. Por exemplo, um projeto pequeno e relativamente simples pode exigir as seguintes tarefas para a atividade de comunicação:

1. Desenvolver uma lista de questões para esclarecimentos.
2. Reunir-se com envolvidos para esclarecer as questões pendentes.
3. Desenvolver conjuntamente uma base documentada do escopo.
4. Revisar a base considerando todos os envolvidos.
5. Alterar a base conforme necessário.

Esses eventos podem ocorrer em um período menor do que 48 horas. Eles representam uma decomposição do processo apropriada para projetos pequenos e relativamente simples.

Agora, considere um projeto mais complexo, com um escopo mais amplo e um impacto comercial mais significativo. Tal projeto pode exigir as seguintes tarefas para a **comunicação**:

1. Revisão da solicitação do cliente.
2. Planejamento e agendamento de reuniões facilitadas e formais com todos os envolvidos.
3. Realização de uma pesquisa para especificar a solução proposta e as abordagens existentes.
4. Preparação de um “documento de trabalho” e de um cronograma para a reunião formal.
5. Realização de reunião.
6. Desenvolvimento em conjunto de miniespecificações que reflitam os dados, a função e os fatores comportamentais do software. Outra opção, desenvolvimento de casos de uso que descrevam o software sob o ponto de vista do usuário.
7. Revisão de cada miniespecificação ou caso de uso quanto à exatidão, consistência e ausência de ambiguidades.
8. Reunião das miniespecificações em um documento de escopo.
9. Revisão do documento de escopo ou coleta de casos de uso com todos os envolvidos.
10. Alteração do documento de escopo ou de casos de uso, conforme necessário.

⁶ Vale lembrar que características de projeto têm forte influência sobre a estrutura da equipe de software (Seção 31.2.3).

Ambos os projetos realizam a atividade metodológica que denominamos **comunicação**, mas a primeira equipe executa metade das tarefas de trabalho de engenharia.

31.5 O projeto

Para gerenciar um projeto de software bem-sucedido, é preciso saber o que pode sair errado, de modo que os problemas possam ser evitados. Em um excelente artigo sobre projetos de software, John Reel [Ree 99] define sinais indicadores de que um projeto de sistemas de informações está em perigo. Em alguns casos, o pessoal de software não entende as necessidades de seus clientes, o que leva a um projeto com escopo mal definido. Em outros projetos, as alterações são mal gerenciadas. Às vezes, a tecnologia escolhida ou as necessidades do negócio mudam e perde-se o apoio da direção. A gerência pode definir prazos finais não realistas ou os usuários podem se opor ao novo sistema. Há casos em que a equipe de projeto simplesmente não tem as habilidades necessárias. E, por último, existem desenvolvedores que parecem nunca aprender com seus erros.

Com frequência, profissionais da indústria esgotados referem-se à “regra 90-90” ao debaterem sobre projetos particularmente difíceis. Os primeiros 90% de um sistema absorvem 90% dos esforços e tempos alocados. Os 10% restantes consomem outros 90% de esforço e tempo alocados [Zah 94]. As situações que conduzem à regra 90-90 foram incluídas nos indicadores da lista anterior.

Chega de negatividade! Como um gerente age para evitar os problemas mencionados? Reel [Ree 99] sugere uma estratégia de cinco partes para os projetos de software:

1. *Comece com o pé-direito.* Trabalhando arduamente (muito arduamente), é possível compreender o problema a ser solucionado e, então, estabelecer expectativas e objetivos realistas para todos os envolvidos no projeto. Isso é reforçado formando-se a equipe correta (Seção 31.2.3) e concedendo-lhe autonomia, autoridade e tecnologia necessárias para realizar o trabalho.
2. *Mantenha o ímpeto.* Muitos projetos começam bem e depois desintegram lentamente. Para manter o ímpeto, o coordenador deve fornecer incentivos para que a rotatividade de pessoal fique em um nível absolutamente mínimo. A equipe deve dar ênfase à qualidade em todas as tarefas que realiza, e o coordenador sênior deve fazer todo o possível para posicionar-se fora do caminho da equipe.⁷
3. *Verificado o andamento.* Em um projeto de software, o andamento é verificado e mapeado conforme os artefatos (modelos, código-fonte, conjuntos de casos de teste) são produzidos e aprovados (usando-se as revisões técnicas) como parte de uma atividade de garantia de qualidade. Como acréscimo, o processo de software e as medições do projeto (Capítulo 32) podem ser coletados e utilizados para avaliar o progresso (andamento) em

Quais são os sinais de que um projeto de software está em perigo?

⁷ A implicação dessa orientação é que se reduz a burocracia a um mínimo, reuniões extras são eliminadas e radicalismos ao processo e às regras do projeto são atenuados. A equipe deve ser auto-organizada e autônoma.

relação às médias desenvolvidas para a organização de desenvolvimento de software.

4. *Tome decisões inteligentes.* Em essência, a decisão do gerente de projeto e da equipe de software deve ser a de “manter a simplicidade”. Sempre que possível, decida-se por utilizar software comercial ou por componentes e modelos de software existentes. Evite interfaces personalizadas quando houver disponibilidade de estratégias padrão. Fique atento aos riscos óbvios a fim de evitá-los e decida-se por alocar maior prazo do que o necessário para tarefas arriscadas ou complexas (serão necessários todos os minutos). Somente as práticas vitais associadas à “integridade do projeto” são registradas aqui.
5. *Faça uma análise post-mortem.* Estabeleça um mecanismo coerente para extrair aprendizados de cada projeto. Avalie os cronogramas planejados e os realizados, as métricas de projetos de software coletadas e analisadas, obtenha *feedback* dos membros da equipe e dos clientes e registre por escrito.

31.6 O princípio W⁵HH

Em um excelente artigo sobre projeto e processo de software, Barry Boehm [Boe 96] afirma: “É preciso haver um princípio organizacional que facilite a obtenção de planejamentos simples para projetos simples”. Boehm propõe uma abordagem voltada aos objetivos do projeto, marcos (pontos de referência) e cronogramas (agendas), responsabilidades, gerenciamento, abordagens técnicas e recursos necessários. Ele a chamou de *Princípio W⁵HH*, por causa de uma série de perguntas (em inglês) que conduzem a uma definição das características-chave do projeto e do planejamento do projeto resultante:

Como definir as características-chave do projeto?

(Why) Por que o sistema está sendo desenvolvido? Todos os envolvidos devem avaliar a validade das razões comerciais para o trabalho de software. O propósito justifica os gastos referentes a pessoal, tempo e dinheiro?

(What) O que será feito? Define-se o conjunto de tarefas necessárias para o projeto.

(When) Quando será feito? A equipe definirá o cronograma de projeto, identificando quando serão realizadas as tarefas e quando os pontos de referências (marcos) serão atingidos.

(Who) Quem será o responsável por uma função? Os papéis e responsabilidades de cada membro serão definidos.

(Where) Onde se posicionam organizacionalmente? Nem todas as situações e responsabilidades estão a cargo dos desenvolvedores de software. O cliente, os usuários e outros envolvidos também têm as suas responsabilidades.

(How) Como será realizado o trabalho técnica e gerencialmente? Uma vez estabelecido o escopo, deve-se definir uma estratégia técnica e gerencial.

(How much) Quanto de cada recurso será necessário? A resposta a essa pergunta vai ser derivada de estimativas desenvolvidas (Capítulo 33), baseando-se nas respostas das perguntas anteriores.

O princípio W⁵HH de Boehm é aplicável independentemente do tamanho ou da complexidade do projeto. As questões apontadas fornecem um excelente esquema de planejamento.

31.7 Práticas vitais

O Conselho Airlie⁸ desenvolveu uma lista de “práticas de software vitais para o gerenciamento baseado no desempenho”. Esses procedimentos são “usados de forma consistente e considerados críticos para projetos de software altamente bem-sucedidos e por organizações cujo desempenho mínimo é consistentemente melhor do que as médias da indústria” [Air 99].

Práticas vitais⁹ incluem: gerenciamento de projeto baseado em métricas (Capítulo 32), custos empíricos e estimativas de cronogramas (Capítulos 33 e 34), acompanhamento de valorização (Capítulo 34), acompanhamento de defeitos em contrapartida com os objetivos de qualidade (Capítulos 19 a 21) e gerenciamento consciente de pessoal (Seção 31.2). Cada uma dessas práticas é mencionada ao longo da Parte IV deste livro.

FERRAMENTAS DO SOFTWARE



Ferramentas de software para gerentes de projeto

As ferramentas listadas aqui são genéricas e se aplicam a uma larga escala de atividades realizadas por gerentes de projetos. Ferramentas específicas para o gerenciamento de projeto (por exemplo, ferramentas para elaboração de cronogramas, para estimativas e para análise de riscos) serão consideradas em capítulos posteriores.

Ferramentas representativas:¹⁰

Projectmanagement.com (<http://www.projectmanagement.com>) desenvolveu um conjunto de *checklists* úteis para gerentes de projeto.

Ittoolkit.com (www.ittoolkit.com) fornece “uma coleção de guias de planejamento, modelos de processo e planilhas inteligentes”, disponível em CD-ROM.

31.8 Resumo

O gerenciamento de projeto de software é uma atividade de apoio da engenharia de software. Ele inicia antes de qualquer atividade técnica e prossegue ao longo da modelagem, construção e utilização do software.

Os quatro Ps (4 Ps – pessoas, produto, processo e projeto) têm grande influência sobre o gerenciamento do projeto de software. As pessoas devem ser organizadas em equipes eficientes, devem ser motivadas a realizar um trabalho de software de alta qualidade e devem ser coordenadas para uma co-

⁸ O Conselho Airlie foi formado por uma equipe de especialistas em engenharia de software e registrado pelo Departamento de Defesa dos Estados Unidos para ajudar a desenvolver um guia de melhores práticas para o gerenciamento de projeto de software e para engenharia de software. Para mais informações sobre melhores práticas, consulte http://www.swqual.com/e_newsletter.html.

⁹ As práticas vitais adotadas aqui se referem apenas à “integridade do projeto”.

¹⁰ A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

municação eficaz. Requisitos de produto devem ser comunicados do cliente ao desenvolvedor, decompostos em partes e posicionados para a execução pela equipe de software. O processo deve ser adaptado às pessoas e ao problema. Uma metodologia de processo comum é selecionada, um paradigma de engenharia de software apropriado é aplicado e um conjunto de tarefas é escolhido para que o trabalho se realize. Por fim, o projeto deve ser organizado de modo a capacitar a equipe de software a ser bem-sucedida.

O elemento-chave de todos os projetos de software são os profissionais. Engenheiros de software podem ser organizados em diferentes estruturas de equipes que incluem desde hierarquias de controle tradicional até equipes de “paradigma aberto”. Uma variedade de técnicas de coordenação e comunicação pode ser aplicada para dar suporte ao trabalho da equipe. Em geral, revisões técnicas e comunicação informal pessoa a pessoa têm o maior valor para os desenvolvedores.

As atividades de gerenciamento de projeto englobam medições e métricas, estimativas e agendamento, análise de riscos, acompanhamento e controle. Cada um desses tópicos será considerado nos próximos capítulos.

Problemas e pontos a ponderar

31.1 Baseando-se nas informações deste capítulo e na sua experiência, elabore “dez mandamentos” para dar poder o engenheiro de software. Ou seja, faça uma lista de dez princípios que guiarão os desenvolvedores a trabalhar com o máximo de potencial.

31.2 O People-CMM do Software Engineering Institute (SEI) adota uma visão organizada para as “áreas de processo-chave” (KPAs), cultivando o bom pessoal de software. O seu instrutor vai indicar uma KPA para análise e resumo.

31.3 Descreva três situações cotidianas nas quais o cliente e o usuário final são os mesmos. Descreva três situações nas quais eles são distintos.

31.4 As decisões tomadas pelo gerenciamento sênior podem ter impacto significativo na eficiência da equipe de engenharia de software. Forneça cinco exemplos ilustrativos em que isso seja verdadeiro.

31.5 Faça uma análise do livro de Weinberg [Wei 86] e um resumo de duas ou três páginas sobre os itens a considerar ao aplicarmos o modelo MOI.

31.6 Você foi nomeado gerente de projeto em uma organização de sistemas de informações. Sua tarefa é construir uma aplicação bastante similar a outras que sua equipe desenvolveu, embora essa seja maior e mais complexa. Os requisitos foram completamente documentados pelo cliente. Qual estrutura de equipe você escolheria e por quê? Qual modelo de processo de software escolheria e por quê?

31.7 Você foi nomeado gerente de projeto em uma companhia de software. Sua tarefa é desenvolver algo inovador que combine hardware de realidade virtual com software moderno. Pelo fato de a competitividade pelo mercado de entretenimento doméstico ser intensa, há pressão significativa quanto à conclusão do trabalho. Qual estrutura de equipe você escolheria e por quê? Qual modelo de processo de software escolheria e por quê?

31.8 Você foi nomeado gerente de projeto em uma grande empresa de software. Seu trabalho é gerenciar o desenvolvimento da versão da próxima geração do seu amplamente utilizado processador de texto. Por haver intensa concorrência, prazos de entrega curtos foram estabelecidos e anunciados. Qual estrutura de equipe você escolheria e por quê? Qual modelo de processo de software escolheria e por quê?

31.9 Você foi nomeado gerente de projeto de software em uma empresa que presta serviços para o setor de engenharia genética. Seu trabalho é administrar o desenvolvimento de um novo software que acelerará o ritmo de classificação de tipos de genes. O trabalho é de pesquisa e desenvolvimento, mas o objetivo é gerar um produto para o próximo ano. Qual estrutura de equipe você escolheria e por quê? Qual modelo de processo de software escolheria e por quê?

31.10 Você foi convidado a desenvolver uma pequena aplicação para analisar cada curso oferecido por uma universidade e emitir relatórios sobre a média obtida no curso (para determinada turma). Faça uma declaração de escopo que englobe esse problema.

31.11 Faça uma decomposição funcional de nível 1 da função de formatação de página discutida rapidamente na Seção 31.3.2.

Leituras e fontes de informação complementares

O Project Management Institute (*Guide to the Project Management Body of Knowledge*, 4ª ed., PMI, 2009) aborda todos os aspectos importantes do gerenciamento de projeto. Wysocki (*Effective Software Project Management: Traditional, Agile, Extreme*, 6ª ed., Wiley, 2011), Slinger e Broderick (*The Software Project Manager's Bridge to Agility*, Addison-Wesley, 2008), Bech-told (*Essentials of Software Project Management*, 2ª ed., Management Concepts, 2007), Stellman e Greene (*Applied Software Project Management*, O'Reilly, 2005) e Berkun (*Making Things Happen: Mastering Project Management—Theory in Practice*, O'Reilly, 2008) ensinam habilidades básicas e fornecem orientação detalhada para todas as tarefas de gerenciamento de projetos de software. McConnell (*Professional Software Development*, Addison-Wesley, 2004) oferece informações pragmáticas para conseguir “cronogramas mais breves, produtos de melhor qualidade e projetos de melhor êxito”. Henry (*Software Project Management*, Addison-Wesley, 2003) oferece conselhos práticos que serão úteis para todos os gerentes de projeto.

Tom DeMarco e seus colegas (*Adrenaline Junkies and Template Zombies*, Dorset House, 2008) escreveram um tratado bastante esclarecedor sobre os padrões humanos encontrados em todos os projetos de software. Uma excelente série de quatro volumes escrita por Weinberg (*Quality Software Management*, Dorset House, 1992, 1993, 1994, 1996) apresenta conceitos de pensamento e gerenciamento para sistemas básicos, explica como usar as medições de modo eficaz e trata da “ação congruente”, a habilidade de estabelecer “a adaptação” entre as necessidades do gerente, as necessidades do pessoal técnico e as necessidades do negócio. Ele apresenta informações úteis para gerentes iniciantes ou experientes. Futrell e seus colegas (*Quality Software Project Management*, Prentice Hall, 2002) apresentam um volumoso tratado sobre gerenciamento de projeto. Livros de Neill e seus colegas (*Antipatterns: Managing Software Organizations*, 2ª ed., Auerbach Publications, 2011) e Brown e seus colegas (*Antipatterns in Project Management*, Wiley, 2000) discutem o que não fazer durante o gerenciamento de um projeto de software.

Brooks (*The Mythical Man-Month*, Anniversary Edition, Addison-Wesley, 1995) atualizou seu livro clássico para proporcionar uma nova visão sobre os aspectos de projeto e gerenciamento de software. McConnell (*Software Project Survival Guide*, Microsoft Press, 1997) apresenta excelente guia pragmático para aqueles que devem gerenciar projetos de software. Purba e Shah (*How to Manage a Successful Software Project*, 2ª ed., Wiley, 2000) fornecem vários estudos de caso que indicam por que alguns projetos têm sucesso e outros não. Livros de Kerzner (*Project Management: A Systems Approach to Planning Scheduling and Controlling*, 10ª ed., Wiley, 2009) e Bennatan (*On Time Within Budget*, 3ª ed., Wiley, 2000) apresentam dicas e diretrizes úteis para gerentes de projeto de software.

Weigers (*Practical Project Initiation*, Microsoft Press, 2007) fornece orientações práticas para realizar de forma bem-sucedida um projeto de software desde o início.

Pode-se argumentar que o aspecto mais importante do gerenciamento de projeto de software seja o gerenciamento das pessoas. Cockburn (*Agile Software Development*, Addison-Wesley, 2002) apresenta uma das melhores discussões sobre software desenvolvido atualmente. DeMarco e Lister [DeM98] produziram o livro definitivo sobre profissionais e projetos de software. Além disso, nos últimos anos foram publicadas também as seguintes obras que devem ser examinadas:

- Cantor, M., *Software Leadership: A Guide to Successful Software Development*, Addison-Wesley, 2001.
- Carmel, E., *Global Software Teams: Collaborating Across Borders and Time Zones*, Prentice Hall, 1999.
- Chandler, H. M., *Game Production Handbook*, 2ª ed., Charles River Media, 2008.
- Constantine, L., *Peopeware Papers: Notes on the Human Side of Software*, Prentice Hall, 2001.
- Ebert, C., *Global Software and IT: A Guide to Distributed Development, Projects, and Outsourcing*, Wiley-IEEE Computer Society, 2011.
- Fairley, R. E., *Managing and Leading Software Projects*, Wiley-IEEE Computer Society, 2009.
- Garton, C. e Wegryn, K., *Managing Without Walls*, McPress, 2006.
- Humphrey, W. S. e Over, J. W., *Leadership, Teamwork, and Trust: Building a Competitive Software Capability*, Addison-Wesley, 2011.
- Humphrey, W. S., *Managing Technical People: Innovation, Teamwork, and the Software Process*, Addison-Wesley, 1997.
- Humphrey, W. S., *TSP-Coaching Development Teams*, Addison-Wesley, 2006.
- Jones, P. H., *Handbook of Team Design: A Practitioner's Guide to Team Systems Development*, McGraw-Hill, 1997.
- Karolak, D. S., *Global Software Development: Managing Virtual Teams and Environments*, IEEE Computer Society, 1998.
- Misrik, I., et al., *Collaborative Software Engineering*, Springer, 2010.
- Peters, L., *Getting Results from Software Development Teams*, Microsoft Press, 2008.
- Whitehead, R., *Leading a Software Development Team*, Addison-Wesley, 2001.

Apesar de não se relacionarem especificamente com o mundo do software e muitas vezes apresentarem uma simplificação demasiada e uma ampla generalização, os *best-sellers* sobre “gerenciamento” de Kanter (*Confidence*, Three Rivers Press, 2006), Covey (*The 8th Habit*, Free Press, 2004), Bossidy (*Execution: The Discipline of Getting Things Done*, Crown Publishing, 2002), Drucker (*Management Challenges for the 21st Century*, Harper Business, 1999), Buckingham e Coffman (*First, Break All the Rules: What the World's Greatest Managers Do Differently*, Simon e Schuster, 1999) e Christensen (*The Innovator's Dilemma*, Harvard Business School Press, 1997) enfatizam “novas regras” definidas por uma economia em rápida evolução. Títulos mais antigos como *Who Moved My Cheese?*, *The One-Minute Manager* e *In Search of Excellence* continuam a proporcionar visões valiosas que podem ajudá-lo a gerenciar pessoas e projetos de modo mais eficaz.

Uma ampla variedade de fontes de informação sobre gerenciamento de projeto de software pode ser encontrada na Internet. Uma lista atualizada das referências da Web pode ser encontrada em “recursos de engenharia de software” no site da SEPA: www.mhhe.com/pressman.

Métricas de processo e de projeto 32

A medição permite entender melhor o processo e o projeto, pois é um mecanismo para a avaliação objetiva. Lord Kelvin afirmou certa vez:

Quando você pode medir aquilo sobre o qual está falando e expressar em números, sabe alguma coisa daquilo; mas quando você não pode medir, quando não pode expressar em números, o seu conhecimento é do tipo escasso e insatisfatório: pode ser o começo do conhecimento, mas você avançou muito pouco em seu raciocínio para o estágio de uma ciência.

A comunidade de engenharia de software levou as palavras de Lord Kelvin a sério – mas não sem frustração e mais do que um pouco de controvérsia!

Medições podem ser aplicadas ao processo de software com a intenção de melhorá-lo continuamente. Elas podem ser usadas durante um projeto de software para ajudar nas estimativas, no controle de qualidade, na produtividade e no controle de projeto. Elas também podem ser usadas por engenheiros de software para avaliar a qualidade dos artefatos e auxiliar na tomada de decisões táticas à medida que o projeto progride (Capítulo 30).

No contexto do processo de software e dos projetos feitos usando o processo, uma equipe de software se preocupa principalmente com as métricas

Conceitos-chave

eficiência na remoção de defeitos (DRE)	718
medição	708
métricas	
argumentos para ...	720
baseadas em LOC ..	712
estabelecimento de um programa	722
orientadas a casos de uso	714
orientadas a função	710
orientadas a objetos	713
orientadas a tamanho	709
privadas e públicas ..	706
processo	704
produtividade	712

PANORAMA

O que é? As métricas de projeto e de processo de software são medidas quantitativas que per-

mitem verificar a eficácia do processo de software e dos projetos que utilizam o processo como framework. São coletados dados básicos de qualidade e produtividade, que são, então, analisados, comparados com médias passadas e avaliados para determinar se ocorreram melhorias de qualidade e produtividade. As métricas também são usadas para apontar áreas com problemas, de modo que correções possam ser desenvolvidas, e o processo de software, melhorado.

Quem realiza? Métricas de software são analisadas e avaliadas por gerentes de software. As medidas muitas vezes são coletadas por engenheiros de software.

Por que é importante? Se você não medir, sua avaliação será apenas subjetiva. Com a medição, tendências (tanto boas quanto ruins) podem ser detectadas, estimativas podem ser

mais bem feitas e melhorias significativas podem ser obtidas ao longo do tempo.

Quais são as etapas envolvidas? Comece definindo um conjunto limitado de medidas de processo, projeto e produto que sejam fáceis de coletar. Essas medidas são, muitas vezes, normalizadas por métricas orientadas a tamanho ou função. O resultado é analisado e comparado com médias anteriores de projetos semelhantes feitos pela empresa. São avaliadas as tendências e são obtidas as conclusões.

Qual é o artefato? Um conjunto de métricas de software que proporcionam *insight* sobre o processo e possibilitam entender o projeto.

Como garantir que o trabalho foi realizado corretamente? Aplicando um esquema de medições sólido e simples, que nunca deve ser usado para avaliar, recompensar ou punir indivíduos por seu desempenho pessoal.

projeto.....	707
qualidade de software.....	716
referencial.....	720
WebApps.....	714
ponto de função (FP)...	710

de produtividade e qualidade – medidas da “saída” do desenvolvimento de software em função do esforço e do tempo aplicados e medidas da “adequação para uso” dos artefatos produzidos. Para fins de planejamento e estimativa, nosso interesse é histórico. Qual foi a produtividade do desenvolvimento de software em projetos passados? Qual foi a qualidade do software produzido? Como os dados de produtividade e qualidade do passado podem ser extrapolados para o presente? Como isso pode nos ajudar a fazer planos e estimativas mais precisos?

Em seu guia sobre medições de software, Park, Goethert e Florac [Par96b] observam as razões por que medimos: (1) *caracterizar*, para obter um entendimento “de processos, produtos, recursos e ambientes, e estabelecer referenciais para comparações com futuras avaliações”; (2) *avaliar* “para determinar o status com relação aos planos”; (3) *prever*, “entendendo as relações entre os processos e produtos e criando modelos dessas relações”; e (4) *melhorar*, “identificando etapas, causas-raiz de problemas, ineficiências e outras oportunidades de melhoria da qualidade do produto e do desempenho do processo”.

A medição é uma ferramenta de gerenciamento. Se for usada adequadamente, ela aumenta o conhecimento do gerente de projeto. E, conseqüentemente, ajuda o gerente de projeto e a equipe de software a tomarem decisões que levarão a um projeto bem-sucedido.

32.1 Métricas no domínio do processo e do projeto

Métricas de processo são coletadas em todos os projetos e no decorrer de longos períodos de tempo. Sua finalidade é proporcionar um conjunto de indicadores de processo que levam à melhoria do processo de software no longo prazo (Capítulo 37). *Métricas de projeto* permitem ao gerente de projeto de software (1) avaliar o estado de um projeto em andamento, (2) rastrear os riscos em potencial, (3) descobrir áreas problemáticas antes que se tornem “críticas”, (4) ajustar o fluxo de trabalho ou tarefas e (5) avaliar a habilidade da equipe de projeto para controlar a qualidade dos artefatos de software.

Medidas coletadas por uma equipe de projeto e convertidas em métricas para uso durante um projeto também podem ser transmitidas para os responsáveis pelo aperfeiçoamento do processo de software. Por essa razão, muitas das mesmas métricas são usadas tanto no domínio do processo quanto no do projeto.

32.1.1 Métricas de processo e aperfeiçoamento do processo de software

A única maneira lógica de melhorar qualquer processo é medir atributos específicos do processo, desenvolver um conjunto de métricas significativas com base nesses atributos e então usar as métricas para fornecer indicadores que levem a uma estratégia de aperfeiçoamento (Capítulo 37). Mas antes de dis-

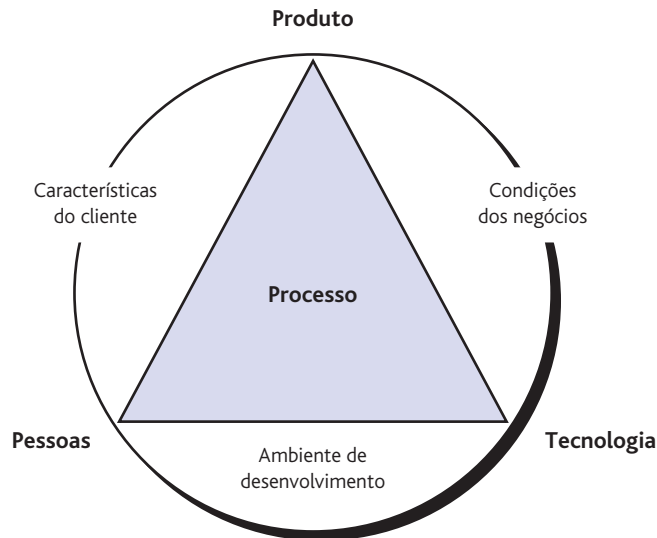


FIGURA 32.1 Determinantes para a qualidade do software e a eficácia organizacional.

Fonte: Adaptado de [Pau94]

cutirmos as métricas de software e seu impacto sobre a melhoria do processo de software, é importante notar que o processo é apenas um item dentre uma série de “fatores controláveis na melhoria da qualidade do software e do desempenho organizacional” [Pau94].

De acordo com a Figura 32.1, o processo está no centro do triângulo que conecta três fatores de profunda influência sobre a qualidade do software e do desempenho organizacional. Já foi demonstrado [Boe81] que a habilidade e a motivação das pessoas representam os fatores mais influentes na qualidade e no desempenho. A complexidade do produto pode ter um impacto significativo sobre a qualidade e o desempenho da equipe. A tecnologia (isto é, os métodos e ferramentas de engenharia de software) que preenche o processo também tem um impacto.

Além disso, o triângulo do processo encontra-se dentro de um círculo de condições ambientais que inclui o ambiente de desenvolvimento (por exemplo, ferramentas de software integradas), condições de negócios (por exemplo, prazos de entrega, regras do negócio) e características do cliente (por exemplo, facilidade de comunicação e colaboração).

Só é possível medir a eficácia de um processo de software indiretamente. Isto é, você cria um conjunto de métricas baseadas nos resultados que podem ser obtidos do processo. Os resultados incluem medidas de erros descobertos antes da entrega do software, defeitos relatados pelos usuários finais, artefatos fornecidos (produtividade), esforço humano gasto, tempo usado, conformidade com o cronograma e outras medidas. Também é possível obter métricas de processo medindo-se as características de tarefas específicas de engenharia de software. Por exemplo, você pode medir o esforço e o tempo despendidos executando as atividades genéricas de engenharia de software descritas no Capítulo 2.

A habilidade e motivação dos profissionais que fazem o trabalho são os fatores mais importantes que influenciam na qualidade do software.

“As métricas de software permitem que você saiba quando deve rir e quando deve chorar.”

Tom Gilb

Qual é a diferença entre usos públicos e privados de métricas de software?

Grady [Gra92] afirma que há usos “privados e públicos” para diferentes tipos de dados de processo. Como é natural os engenheiros de software podem ser sensíveis ao uso de métricas coletadas individualmente, esses dados devem ser privados e servir apenas como um indicador individual. Exemplos de *métricas privadas* incluem taxas de defeito (por indivíduo), taxas de defeito (por componente) e erros encontrados durante o desenvolvimento.

A filosofia de “dados privados do processo” combina bem com a abordagem Personal Software Process (Capítulo 4) proposta por Humphrey [Hum05]. Ele reconheceu que a melhoria do processo de software pode e deve começar em nível individual. Dados privados do processo podem servir como um motivador importante quando você trabalha para melhorar a sua abordagem de engenharia de software.

Algumas métricas de processo são privadas para a equipe de projeto de software, mas públicas para todos os membros da equipe. Exemplos incluem defeitos relatados para funções principais do software (que foram desenvolvidas por vários profissionais), erros encontrados durante revisões técnicas e linhas de código ou pontos de função por componente ou função.¹ A equipe examina esses dados para descobrir indicadores que possam melhorar o desempenho da equipe.

Métricas públicas geralmente assimilam informações que originalmente eram privadas para os indivíduos e equipes. Taxas de defeito em nível de projeto (absolutamente não atribuídas a um indivíduo), esforço, prazos agendados e dados relacionados são coletados e avaliados na tentativa de descobrir indicadores que possam melhorar o desempenho do processo organizacional.

Métricas de processo de software podem produzir benefícios significativos quando uma organização trabalha para melhorar seu nível geral de maturidade de processo. No entanto, assim como todas as métricas, essas podem ser mal utilizadas, criando mais problemas do que podem resolver. Grady [Gra92] sugere uma “etiqueta de métricas de software” apropriada para os gerentes e para os profissionais quando instituem um programa de métricas de processo:

Que diretrizes devem ser aplicadas quando coletamos métricas de software?

- Use bom senso e sensibilidade organizacional ao interpretar dados de métricas.
- Forneça *feedback* regularmente para os indivíduos e equipes que coletam medidas e métricas.
- Não use métricas para avaliar indivíduos.
- Trabalhe com profissionais e equipes para definir objetivos claros e as métricas que serão usadas para alcançá-los.
- Nunca use métricas para ameaçar indivíduos ou equipes.
- Dados de métricas que indicam uma área com problema não devem ser considerados “negativos”. Esses dados são simplesmente um indicador para melhoria do processo.

¹ Métricas de linhas de código e pontos de função são discutidas nas Seções 32.2.1 e 32.2.2.

- Não seja obsessivo sobre uma única métrica, excluindo outras métricas importantes.

À medida que uma organização se sente mais à vontade com a coleta e uso de métricas de processo, a derivação de indicadores simples dá margem a uma abordagem mais rigorosa, chamada *melhoria estatística de processo de software* (SSPI, *statistical software process improvement*). Basicamente, a SSPI usa a análise de falhas de software para coletar informações sobre todos os erros e defeitos² encontrados quando uma aplicação, sistema ou produto é desenvolvido e usado.

32.1.2 Métricas de projeto

Ao contrário das métricas de processo de software, que são usadas para fins estratégicos, as medidas de projeto de software são táticas. Isto é, as métricas de projeto e os indicadores derivados delas são usados por um gerente de projeto e por uma equipe de software para adaptar o fluxo de trabalho do projeto e as atividades técnicas.

Na maioria dos projetos de software, a primeira aplicação das métricas de projeto ocorre durante as estimativas. Métricas coletadas de projetos passados são usadas como uma base a partir da qual são feitas as estimativas de esforços e tempo para o trabalho de software atual. À medida que um projeto progride, medidas de esforço e tempo despendidos são comparadas com as estimativas originais (e com o cronograma do projeto). O gerente de projeto usa esses dados para monitorar e controlar o progresso.

Quando o trabalho técnico inicia, outras métricas de projeto começam a ter importância. São medidas as taxas de produção, representadas em modelos criados, revisões, pontos de função e linhas de código-fonte fornecidas. Além disso, são rastreados os erros descobertos durante cada tarefa de engenharia de software. Com a evolução do software dos requisitos para o projeto, métricas técnicas (Capítulo 30) são coletadas para avaliar a qualidade do projeto e fornecer indicadores que terão influência na estratégia adotada para a geração de código e teste.

O objetivo das métricas de projeto é duplo. Primeiro, as métricas são usadas para minimizar o cronograma de desenvolvimento, fazendo os ajustes necessários para evitar atrasos e mitigar problemas e riscos em potencial. Segundo, as métricas de projeto são usadas para avaliar a qualidade do produto de forma contínua e, quando necessário, modificar a abordagem técnica para melhorar a qualidade.

À medida que a qualidade melhora, os defeitos são minimizados, e, à medida que a contagem de defeitos diminui, a quantidade de retrabalho necessário durante o projeto também é reduzida. Isso leva a uma redução no custo total do projeto.

Como devemos usar as métricas durante o próprio projeto?

² Neste livro, um *erro* é definido como alguma falha em um artefato descoberta *antes* que o software seja fornecido ao usuário final. Um *defeito* é uma falha descoberta *depois* que o software é fornecido ao usuário final. Devemos destacar que muitas pessoas não fazem essa distinção.



Estabelecendo uma abordagem de métricas

Cena: Escritório de Doug Miller, quando o projeto de software *CasaSegura* está para começar.

Atores: Doug Miller (gerente da equipe de engenharia de software do *CasaSegura*), Vinod Raman e Jamie Lazar, membros da equipe de engenharia de artefatos de software.

Conversa:

Doug: Antes de começarmos a trabalhar neste projeto, eu gostaria que vocês definissem e coletassem um conjunto de métricas simples. Para começar, vocês terão de definir nossas metas.

Vinod (com cara de contrariado): Nós nunca fizemos isso antes e...

Jamie (interrompendo): E com base no cronograma que a gerência tem falado, nós nunca teremos tempo. Para que servem essas métricas, afinal?

Doug (erguendo a mão para interromper a discussão): Calma, respirem fundo. O fato de nunca termos feito isso antes é mais um motivo para começar a fazer agora, e o trabalho com as métricas não deve tomar muito tempo... na verdade, elas podem até nos poupar tempo.

Vinod: Como?

Doug: Olha, faremos muito mais trabalho interno de engenharia de software à medida que nossos produtos se tornarem mais inteligentes, habilitados para a Web, tecnologia móvel, tudo isso... e precisaremos entender o processo que usamos para criar o software... e melhorá-lo para criar um

software melhor. A única maneira de fazer isso é por meio de medições.

Jamie: Mas estamos sendo pressionados no prazo, Doug. Não sou a favor de gerar mais papel... precisamos de tempo para fazer nosso trabalho, não para coletar dados.

Doug (calmamente): Jamie, o trabalho de um engenheiro envolve coleta de dados, avaliação desses dados e o uso do resultado para melhorar o produto e o processo. Estou errado?

Jamie: Não, mas...

Doug: E que tal se mantivermos o número de medidas que coletarmos em não mais de cinco ou seis e focarmos na qualidade?

Vinod: Ninguém pode argumentar contra a alta qualidade...

Jamie: Certo... mas, não sei. Ainda acho que não é necessário.

Doug: Me deem uma chance neste ponto. O que vocês sabem sobre métricas?

Jamie (olhando para Vinod): Não muito.

Doug: Aqui estão algumas referências da Web... dediquem algumas horas para se informar.

Jamie (sorrindo): Eu achei que você tinha falado que isso não tomaria muito tempo.

Doug: O tempo que você gasta aprendendo nunca é perdido... façam isso e então estabeleceremos nossas metas, façam algumas perguntas e definam as métricas que precisamos coletar.

32.2 Medição de software

No Capítulo 30, vimos que as medições no mundo físico podem ser classificadas de duas maneiras: medidas diretas (por exemplo, o comprimento de um parafuso) e medidas indiretas (por exemplo, a “qualidade” dos parafusos produzidos, medida contando os rejeitos). As métricas de software podem ser classificadas de maneira similar.

As *medidas diretas* do processo de software incluem custos e trabalho aplicado. As medidas diretas do produto incluem linhas de código (LOC, lines of code) produzidas, velocidade de execução, tamanho da memória e defeitos relatados durante determinado período de tempo. Medidas indiretas do produto incluem funcionalidade, qualidade, complexidade, eficiência, confiabilidade, manutenibilidade e muitas outras “ades” discutidas no Capítulo 19.

O custo e o trabalho exigidos para criar o software, o número de linhas de código produzidas e outras medidas diretas são relativamente fáceis de coletar, desde que antecipadamente sejam estabelecidas convenções para

“Nem tudo o que pode ser contado importa e nem tudo o que importa pode ser contado.”

Albert Einstein

as medições. No entanto, a qualidade e a funcionalidade do software ou sua eficiência ou manutenibilidade são mais difíceis de avaliar e só podem ser medidas de forma indireta.

Dividimos o domínio das métricas de software em processo, projeto e métricas de produto e observamos que as métricas de produto privadas para um indivíduo muitas vezes são combinadas para desenvolver métricas de projeto públicas para a equipe de software. Métricas de projeto são, então, consolidadas para criar métricas de processo que são públicas à organização de software como um todo. Mas como uma organização combina métricas que vieram de diferentes indivíduos ou projetos?

A título de ilustração, considere um exemplo simples. Indivíduos em duas equipes de projeto diferentes registram e classificam todos os erros que encontram durante o processo de software. As medidas individuais são, então, combinadas para criar medidas de equipe. A Equipe A encontrou 342 erros durante o processo de software antes da entrega. A Equipe B encontrou 184 erros. Sendo iguais todas as outras coisas, qual equipe é mais eficaz na descoberta de erros em todo o processo? Como você não sabe qual é o tamanho ou complexidade dos projetos, não pode responder a essa pergunta. No entanto, se as medidas forem normalizadas, é possível criar métricas de software que possibilitem a comparação com médias organizacionais mais amplas.

Como muitos fatores afetam o trabalho de software, não use métricas para comparar indivíduos ou equipes.

32.2.1 Métricas orientadas a tamanho

Métricas de software orientadas a tamanho são criadas pela normalização das medidas de qualidade e/ou produtividade, levando-se em consideração o *tamanho* do software produzido. Se uma empresa de software mantém registros simples, pode ser criada uma tabela de medidas orientadas a tamanho, como a da Figura 32.2. A tabela lista todos os projetos de desenvolvimento de software concluídos durante os últimos anos e medidas correspondentes para aquele projeto. Olhando a linha da tabela (Figura 32.2) para o projeto *alfa*: 12.100 linhas de código foram criadas com 24 pessoas-mês de trabalho a um custo de \$168.000. É importante notar que todo o trabalho e

Projeto	LOC	Esforço	\$(000)	Pág. doc.	Erros	Defeitos	Pessoas
alfa	12.100	24	168	365	134	29	3
beta	27.200	62	440	1224	321	86	5
gama	20.200	43	314	1050	256	64	6
•	•	•	•	•	•		
•	•	•	•	•	•		
•	•	•	•	•	•		

FIGURA 32.2 Métricas orientadas a tamanho.

custo registrados na tabela representam todas as atividades de engenharia de software (análise, projeto, código e teste), não apenas o desenvolvimento do código. Outras informações para o projeto *alfa* indicam que foram criadas 365 páginas de documentação, foram registrados 134 erros antes da entrega do software e foram encontrados 29 defeitos após a entrega para o cliente durante o primeiro ano de operação. Três pessoas trabalharam no desenvolvimento do software para o projeto alfa.

A fim de desenvolver métricas que possam ser assimiladas com métricas similares de outros projetos, você pode escolher o número de linhas de código como valor de normalização. A partir dos dados rudimentares contidos na tabela, pode ser desenvolvido um conjunto de métricas simples orientadas a tamanho para cada projeto:

- Erros por KLOC (mil linhas de código)
- Defeitos por KLOC
- \$ por KLOC
- Páginas de documentação por KLOC

Além disso, podem ser calculadas outras métricas interessantes:

- Erros por pessoa-mês
- KLOC por pessoa-mês
- \$ por página de documentação

Métricas orientadas a tamanho são amplamente usadas, mas o debate sobre sua validade e aplicabilidade continua.

Métricas orientadas a tamanho não são aceitas universalmente como a melhor maneira de medir os processos de software. A maior parte da controvérsia gira em torno do uso de linhas de código como medida principal. Os proponentes da medida LOC argumentam que LOC é um “artefato” de todos os projetos de desenvolvimento de software que pode ser facilmente contado, que muitos modelos de estimativa de software existentes usam LOC ou KLOC como dado de entrada principal e que já existe uma grande quantidade de literatura e dados baseados em LOC. Por outro lado, os oponentes argumentam que as medidas LOC são dependentes da linguagem de programação; que, quando é considerada a produtividade, elas penalizam programas bem-projetados, mas menores; que elas não podem facilmente acomodar linguagens não procedurais; e que seu uso nas estimativas requer um nível de detalhe que pode ser difícil de alcançar (isto é, o planejador deve estimar a LOC a ser produzida bem antes que a análise e o projeto estejam concluídos).

32.2.2 Métricas orientadas a função

Métricas de software orientadas a função usam como valor de normalização uma medida da funcionalidade fornecida pela aplicação. A métrica orientada a função mais amplamente usada é a *ponto de função* (FP, *function point*). O cálculo de pontos de função é baseado nas características de domínio de informação e complexidade do software. O mecanismo de cálculo de FP foi discutido no Capítulo 30.³

³ Veja na Seção 30.2.1 uma discussão detalhada sobre cálculo de FP.

O ponto de função, assim como a medida LOC, é controverso. Seus proponentes argumentam que essa função é independente da linguagem de programação, tornando-a ideal para aplicações que usam linguagens convencionais e não procedurais, e que é baseada em dados que têm maior probabilidade de serem conhecidos na evolução de um projeto, o que a torna mais atraente como estratégia de estimativa. Seus oponentes argumentam que o método requer um pouco de “jeitinho”, porque o cálculo é baseado em dados subjetivos, em vez de objetivos, que as contagens do domínio de informações (e outras dimensões) podem ser difíceis de coletar após o fato e que a FP não tem um significado físico direto – é apenas um número.

32.2.3 Harmonizando métricas LOC e FP

A relação entre linhas de código e pontos de função depende da linguagem de programação adotada para implementar o software e da qualidade do projeto. Muitos estudos já tentaram relacionar medidas FP e LOC. A tabela a seguir⁴ [QSM02] fornece estimativas aproximadas da média do número de linhas de código necessárias para criar um ponto de função em várias linguagens de programação:

Linguagem de Programação	LOC por Ponto de Função			
	Média	Mediana	Baixa	Alta
Ada	154	–	104	205
ASP	56	50	32	106
Assembler	337	315	91	694
C	148	107	22	704
C++	59	53	20	178
C#	58	59	51	704
COBOL	80	78	8	400
ColdFusion	68	56	52	105
DBase IV	52	–	–	–
Easytrieve+	33	34	25	41
Focus	43	42	32	56
FORTRAN	90	118	35	-
FoxPro	32	35	25	35
HTML	43	42	35	53
Informix	42	31	24	57
J2EE	57	50	50	67
Java	55	53	9	214
JavaScript	54	55	45	63

(continua)

⁴ Os dados apresentados na tabela representam uma versão abreviada dos dados revelados pelo Quantitative Software Management (www.qsm.com) e são usados com sua permissão; direitos autorais de 2002.

Linguagem de Programação	LOC por Ponto de Função			
	Média	Mediana	Baixa	Alta
JSP	59	–	–	–
Lotus Notes	23	21	15	46
Mantis	71	27	22	250
Natural	51	53	34	60
.NET	60	60	60	60
Oracle	42	29	12	217
OracleDev2K	35	30	23	100
PeopleSoft	37	32	34	40
Perl	57	57	45	60
PL/1	58	57	27	92
Powerbuilder	28	22	8	105
RPG II/III	61	49	24	155
SAS	50	35	33	49
Smalltalk	26	19	10	55
SQL	31	37	13	80
VBScript	38	37	29	50
Visual Basic	50	52	14	276

Uma análise desses dados indica que uma LOC de C++ fornece aproximadamente 2,4 vezes a “funcionalidade” (em média) de uma LOC de C. Além disso, uma LOC de Smalltalk fornece pelo menos quatro vezes a funcionalidade de uma LOC para uma linguagem de programação convencional como Ada, COBOL ou C. Usando as informações da tabela, é possível “retroagir” [Jon98] o software existente para estimar o número de pontos de função, uma vez conhecido o número total de instruções da linguagem de programação.

Medidas de LOC e FP muitas vezes são usadas para derivar *métricas de produtividade*. Isso leva, invariavelmente, a um debate sobre o uso de tais dados. O valor LOC/pessoa-mês (ou FP/pessoa-mês) de um grupo deve ser comparado com dados similares de outro grupo? Os gerentes devem avaliar o desempenho dos indivíduos usando essas métricas? A resposta a essas questões é um enfático “não!”. A razão para essa resposta é que muitos fatores influenciam a produtividade, gerando comparações do tipo “maças com laranjas”, que podem facilmente ser mal interpretadas.

Foi constatado que pontos de função e *métricas baseadas em LOC* são indicadores relativamente precisos do trabalho e do custo do desenvolvimento de software. No entanto, para usar LOC e FP para estimativas (Capítulo 33), deve ser estabelecida uma referência histórica de informações.

Dentro do contexto de métricas de processo e projeto, você deve se preocupar em primeiro lugar com a produtividade e a qualidade – medidas de “resultado” de desenvolvimento de software em função do esforço e tempo aplicados e as medidas da “adequação para uso” dos produtos criados. Para fins de melhoria de processo e planejamento de projeto, seu interesse é his-

tórico. Qual foi a produtividade do desenvolvimento de software nos projetos passados? Qual foi a qualidade do software produzido? Como os dados de produtividade e qualidade do passado podem ser extrapolados para o presente? Como isso pode nos ajudar a melhorar o processo e planejar novos projetos mais precisamente?

32.2.4 Métricas orientadas a objetos

Métricas de projeto de software convencional (LOC ou FP) podem ser usadas para estimar projetos de software orientados a objetos. No entanto, essas métricas não fornecem granularidade suficiente para os ajustes de cronograma e de esforço necessários à medida que são feitas iterações por meio de um processo evolucionário ou incremental. Lorenz e Kidd [Lor94] sugerem o seguinte conjunto de métricas para projetos orientados a objetos:

Número de scripts de cenário. Um script de cenário (análogo a um caso de uso) é uma sequência detalhada de passos que descrevem a interação entre o usuário e a aplicação. Cada script é organizado em trios da forma

{**iniciador**, *ação*, **participante**}

onde **iniciador** é o objeto que solicita algum serviço (que inicia uma mensagem), *ação* é o resultado da solicitação e **participante** é o objeto servidor que atende à solicitação. O número de scripts de cenário está correlacionado diretamente com o tamanho da aplicação e com o número de casos de testes que devem ser desenvolvidos para exercitar o sistema depois que ele é construído.

Número de classes-chave. *Classes-chave* são os “componentes altamente independentes” [Lor94] definidos logo no início em análise orientada a objetos (Capítulo 10).⁵ Como as classes-chave são essenciais ao domínio do problema, a quantidade dessas classes é uma indicação da quantidade de esforço necessário para desenvolver o software e também uma indicação do potencial de reutilização a ser aplicado durante o desenvolvimento do sistema.

Número de classes de apoio. *Classes de apoio* são necessárias para implementar o sistema, mas não estão imediatamente relacionadas ao domínio do problema. Como exemplos podemos citar as classes de interface de usuário (UI), classes de acesso e manipulação de bancos de dados e classes de cálculo. Além disso, podem ser desenvolvidas classes de apoio para cada uma das classes-chave. As classes de apoio são definidas iterativamente ao longo de um processo evolucionário. O número de classes de apoio é uma indicação da quantidade de esforço necessário para desenvolver o software e também uma indicação do potencial de reutilização a ser aplicado durante o desenvolvimento do sistema.

Número médio de classes de apoio para cada classe-chave. Em geral, as classes-chave são conhecidas logo no início do projeto. As classes de apoio são definidas durante o projeto. Se o número médio de classes de apoio para cada classe-chave fosse conhecido para determinado domínio de problema, a estimativa (baseada no número total de classes) seria muito simplificada. Lorenz e

Não é raro que scripts de múltiplos cenários mencionem a mesma funcionalidade ou objetos de dados. Portanto, tenha cuidado ao usar contagem de scripts. Muitos scripts podem ser reduzidos a uma única classe ou conjunto de código.

As classes podem variar em tamanho e complexidade. Portanto, pense em classificar as contagens de classes por tamanho e complexidade.

⁵ Nos referimos às classes-chave como *classes de análise* no Capítulo 10.

Kidd sugerem que as aplicações com uma GUI tenham de duas a três vezes a quantidade de classes de apoio como classes-chave. Aplicações sem GUI têm de uma a duas vezes a quantidade de classes de apoio como classes-chave.

Número de subsistemas. Um *subsistema* é uma agregação de classes que apoia uma função que é visível para o usuário final de um sistema. Uma vez identificados os subsistemas, é mais fácil elaborar um cronograma adequado no qual o trabalho nos subsistemas é dividido entre o pessoal de projeto.

Para serem usadas de modo eficaz em um ambiente de engenharia de software orientado a objetos, métricas similares às aquelas mencionadas acima devem ser coletadas juntamente com as medidas de projeto, tais como o esforço gasto, erros e defeitos descobertos e modelos ou páginas de documentação produzidas. À medida que o banco de dados cresce (após completar um grupo de projetos), as relações entre as medidas orientadas a objetos e as medidas de projeto fornecerão métricas que podem ajudar nas estimativas do projeto.

32.2.5 Métricas orientadas a casos de uso

Os casos de uso⁶ são amplamente usados como método para descrever requisitos no nível dos clientes ou no domínio de negócio que sugerem características e funções de software. Faz sentido usar o caso de uso como uma medida de normalização similar a LOC ou FP. Assim como a FP, o caso de uso é definido no início do processo de software, permitindo que seja usado para estimativas antes de iniciar atividades significativas de modelagem e construção. Os casos de uso descrevem (indiretamente, pelo menos) funções e características visíveis ao usuário que são requisitos básicos para um sistema. O caso de uso é independente da linguagem de programação. Além disso, o número de casos de uso é diretamente proporcional ao tamanho do aplicativo em LOC e ao número de casos de testes que terão de ser projetados para exercitar completamente o aplicativo.

Como os casos de uso podem ser criados em níveis muito diferentes de abstração, não há um “tamanho” padrão para um caso de uso. Sem uma “medida” padronizada do que é um caso de uso, sua aplicação como medida de normalização (por exemplo, esforço gasto por cada caso de uso) é suspeita.

Os pesquisadores têm sugerido os *pontos de casos de uso* (UCPs, *use-case points*) como um mecanismo para estimar trabalho de projeto e outras características. O UCP é uma função do número de atores e das transações deduzidas pelos modelos de casos de uso e é análogo ao FP em alguns aspectos. Se tiver interesse, consulte [Coh05], [Cle06] ou [Col09].

32.2.6 Métricas de projeto de WebApp

O objetivo de todos os projetos de WebApp é fornecer uma combinação de conteúdo e funcionalidade para o usuário. Medidas e métricas usadas para projetos tradicionais de engenharia de software são difíceis de traduzir diretamente para WebApps. No entanto, é possível desenvolver um banco de dados que dê acesso às medidas internas de produtividade e qualidade obtidas em

⁶ Casos de uso são apresentados nos Capítulos 8 e 9.

uma série de projetos. Entre as medidas que podem ser coletadas estão as seguintes:

Número de páginas Web estáticas. Essas páginas representam baixa complexidade relativa e geralmente exigem menos esforços para ser criadas do que as páginas dinâmicas. Essa medida fornece uma indicação do tamanho global da aplicação e do esforço necessário para desenvolvê-la.

Número de páginas Web dinâmicas. Essas páginas representam uma complexidade relativa maior e exigem mais esforço para ser construídas do que as páginas estáticas. Essa medida fornece uma indicação do tamanho global da aplicação e do esforço necessário para desenvolvê-la.

Número de links internos. Essa medida fornece uma indicação do grau de acoplamento arquitetônico dentro da WebApp. À medida que o número de links de páginas aumenta, aumenta também o esforço gasto no projeto navegacional e na construção.

Número de objetos dados persistentes. À medida que cresce o número de objetos dados persistentes (por exemplo, um banco de dados ou um arquivo de dados), a complexidade da WebApp também cresce, e o esforço para implementá-la aumenta proporcionalmente.

Número de interfaces de sistemas externos. À medida que cresce o requisito para interfaces, a complexidade do sistema e esforço de desenvolvimento também aumenta.

Número de objetos de conteúdo estático. Esses objetos representam baixa complexidade relativa e geralmente exigem menos esforço para ser criadas do que as páginas dinâmicas.

Número de objetos de conteúdo dinâmico. Esses objetos representam uma complexidade relativa maior e exigem mais esforço para ser construídas do que as páginas estáticas.

Número de funções executáveis. À medida que o número de funções executáveis (por exemplo, um script ou applet) aumenta, os esforços de modelagem e construção também aumentam.

Cada uma das medidas que acabamos de mencionar pode ser determinada em um estágio relativamente antecipado. Por exemplo, você pode definir uma métrica que reflète o grau de personalização de usuário final exigida para a WebApp e correlacionar isso com o esforço gasto no projeto e/ou os erros descobertos à medida que são feitas as revisões e testes. Para isso, você define

N_{sp} = número de páginas Web estáticas

N_{dp} = número de páginas Web dinâmicas

Então,

$$\text{Índice de personalização, } C = \frac{N_{dp}}{(N_{dp} + N_{sp})}$$

O valor de C varia de 0 a 1. À medida que C se torna maior, o nível de personalização da WebApp se torna um aspecto técnico significativo.

Métricas de WebApp similares podem ser calculadas e correlacionadas com medidas de projeto, tais como esforço gasto, erros e defeitos descobertos e modelos ou páginas de documentação produzidos. À medida que o banco de dados cresce (após a conclusão de certo número de projetos), relações entre as medidas de WebApp e medidas de projeto proporcionarão indicadores que podem ajudar na estimativa do projeto.

FERRAMENTAS DO SOFTWARE



Métricas de projeto e processo

Objetivo: auxiliar na definição, coleta, avaliação e relatórios de medidas e métricas de software.

Mecanismos: cada ferramenta varia em sua aplicação, mas todas elas fornecem mecanismos para coletar e avaliar dados que levam ao cálculo de métricas de software.

Ferramentas representativas:⁷

Function Point WORKBENCH, desenvolvida pela Charismatek (www.charismatek.com.au), oferece uma ampla variedade de métricas orientadas para FP.

DataDrill, desenvolvida pela Distributive Software (www.distributive.com), suporta coleta automática de dados, análise, formatação de gráficos, geração de relatórios e outras tarefas de medição.

PSM Insight, desenvolvida pela Practical Software and Systems Measurement (www.psmc.com), ajuda na criação e subsequente análise de um banco de dados de medições de projeto.

SLIM tool set, desenvolvida pela QSM (www.qsm.com), contém um conjunto abrangente de ferramentas de métricas e estimativas.

SPR tool set, desenvolvida pela Software Productivity Research (www.spr.com), oferece uma coleção abrangente de ferramentas orientadas a FP.

TychoMetrics, desenvolvida pela Predicate Logic (www.predicate.com), é um conjunto de ferramentas para coleta e relato de métricas de gerenciamento.

32.3 Métricas para qualidade de software

A qualidade de um sistema, de uma aplicação ou de um produto é apenas tão boa quanto os requisitos que descrevem o problema, o projeto que modela a solução, o código que leva ao programa executável e os testes que exercitam o software para descobrir os erros. Você pode usar medições para avaliar a qualidade dos requisitos e modelos de projeto, o código-fonte e os casos de teste criados enquanto o software é desenvolvido. Para conseguir essa avaliação em tempo real, aplica-se métricas de produto (Capítulo 30) para avaliar a qualidade dos artefatos de software de maneira objetiva, e não subjetiva.

Um gerente de projeto deve também avaliar a qualidade enquanto o projeto avança. Métricas privadas coletadas por engenheiros de software individuais são combinadas para fornecer os resultados no nível de projeto. Embora muitas medidas de qualidade possam ser coletadas, a principal tendência no nível de projeto é medir erros e defeitos. Métricas derivadas dessas medidas proporcionam uma indicação da efetividade da garantia de qualidade de software individual e de grupo e das atividades de controle.

Software é uma entidade complexa. Portanto, deve-se esperar que ocorram erros à medida que o produto é desenvolvido. As métricas de processo são destinadas a melhorar o processo de software para que os erros sejam descobertos da maneira mais eficiente.

⁷ A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

Métricas, como erros de artefato por ponto de função, erros descobertos por horas de revisão e erros descobertos por horas de teste, proporcionam informações sobre a eficácia de cada uma das atividades sugeridas pela métrica. Os dados sobre os erros também podem ser usados para calcular a *eficiência de remoção de defeitos* (DRE, *defect removal efficiency*) para cada atividade metodológica do processo. A DRE é discutida na Seção 32.3.3.

32.3.1 Medição da qualidade

Embora existam muitas medidas de qualidade de software,⁸ a correção, a manutenibilidade, a integridade e a usabilidade fornecem indicadores úteis para a equipe de projeto. Gilb [Gil88] sugere definições e medidas para cada uma delas.

Correção. A correção é o grau com o qual o software executa sua função. Defeitos (falta de correção) são os problemas relatados por um usuário do programa depois que o programa foi liberado para uso geral. Para fins de avaliação de qualidade, os defeitos são contados durante um período de tempo padrão, em geral um ano. A medida mais comum da correção é o número de defeitos por KLOC, em que um defeito é definido como uma ocorrência de falta de conformidade com os requisitos.

Manutenibilidade. A manutenibilidade é a facilidade com a qual um programa pode ser corrigido se for encontrado um erro, adaptado se o ambiente mudar ou melhorado se o cliente desejar uma alteração nos requisitos. Não há uma maneira de medir a manutenibilidade diretamente; portanto, é preciso usar medidas indiretas. Uma métrica simples orientada por tempo é o *tempo médio de alteração* (MTTC, *mean-time-to-change*), o tempo necessário para analisar a solicitação de alteração, projetar uma modificação apropriada, implementar a alteração, testá-la e distribuí-la para todos os usuários.

Integridade. Esse atributo mede a capacidade de um sistema de resistir aos ataques (tanto acidentais quanto intencionais) à sua segurança. Para medir a integridade, devem ser definidos dois atributos adicionais: ameaça e segurança. *Ameaça* é a probabilidade (que pode ser estimada ou derivada de evidência empírica) de que um ataque de um tipo específico ocorrerá em um dado intervalo de tempo. *Segurança* é a probabilidade (que pode ser estimada ou derivada de evidência empírica) de que um ataque de um tipo específico será repelido. A integridade de um sistema pode, então, ser definida como:

$$\text{Integridade} = \Sigma [1 - (\text{ameaça} \times (1 - \text{segurança}))]$$

Por exemplo, se a ameaça (probabilidade de que um ataque possa ocorrer) for 0,25 e a segurança (a possibilidade de repelir o ataque) for 0,95, a integridade do sistema será 0,99 (muito alta). Por outro lado, se a probabilidade de ameaça for 0,50 e a possibilidade de repelir um ataque for de apenas 0,25, a integridade do sistema será 0,63 (muito baixa e inaceitável).

Uma excelente fonte de informações sobre qualidade de software e tópicos relacionados (incluindo métricas) pode ser encontrada em <http://searchsoftwarequality.techtarget.com/resources>.

⁸ Uma discussão detalhada dos fatores que influenciam na qualidade do software e as métricas que podem ser usadas para avaliar a qualidade do software foi apresentada no Capítulo 30.

Usabilidade. A usabilidade é uma tentativa de quantificar a facilidade de uso e pode ser medida em termos das características apresentadas no Capítulo 15.

Esses quatro fatores são apenas uma amostra dos que foram propostos como medidas para a qualidade do software. O Capítulo 30 considera esse assunto com mais detalhes.

32.3.2 Eficiência na remoção de defeitos

Uma métrica de qualidade que traz vantagens tanto para o projeto quanto para o processo é a *eficiência na remoção de defeitos* (DRE, *defect removal efficiency*). Em essência, a DRE é uma medida da capacidade de filtragem das ações de garantia de qualidade e controle quando são aplicadas em todas as atividades da estrutura de processo.

Quando considerada para um projeto como um todo, a DRE é definida da seguinte maneira:

$$DRE = \frac{E}{E + D}$$

onde E é o número de erros encontrados antes que o software seja fornecido ao usuário final e D é o número de defeitos depois que o software é entregue.

O valor ideal para DRE é 1. Isto é, nenhum defeito é encontrado no software. De modo realista, D será maior do que 0, mas o valor de DRE ainda pode se aproximar de 1. À medida que E aumenta (para um valor de D dado), o valor global de DRE começa a se aproximar de 1. De fato, à medida que E aumenta, é provável que o valor final de D diminua (erros são removidos antes de se tornarem defeitos). Se for usada como uma métrica que fornece um indicador da capacidade de filtragem das atividades de controle de qualidade e segurança, a DRE estimulará a equipe de projeto de software a instituir técnicas para encontrar o maior número possível de erros antes da entrega do software.

A DRE também pode ser usada no projeto para avaliar a habilidade de uma equipe para encontrar erros antes que eles passem para a próxima atividade metodológica ou para a próxima tarefa de engenharia de software. Por exemplo, a análise de requisitos produz um modelo de requisitos que pode ser examinado para encontrar e corrigir erros. Os erros não detectados durante a revisão do modelo de requisitos passam adiante no projeto (onde podem ou não ser encontrados). Quando usada nesse contexto, redefinimos a DRE como

$$DRE_i = \frac{E_i}{E_i + E_{i+1}}$$

onde E_i é o número de erros encontrados durante a ação de engenharia de software i e E_{i+1} é o número de erros encontrados durante a ação de engenharia de software $i + 1$, ligados a erros que não foram descobertos na ação de engenharia de software i .

Um objetivo de qualidade para uma equipe de software (ou um engenheiro de software individual) é conseguir uma DRE_i que se aproxime de 1. Isto é, os erros devem ser filtrados antes que passem para a próxima atividade ou ação.

Se a DRE for baixa quando você fizer a análise e o projeto, dedique algum tempo melhorando a maneira de realizar as revisões técnicas formais.

CASASEGURA



Estabelecendo uma abordagem de métricas

Cena: Escritório de Doug Miller, dois dias após a reunião inicial sobre métricas de software.

Atores: Doug Miller (gerente da equipe de engenharia de software do CasaSegura), Vinod Raman e Jamie Lazar, membros da equipe de engenharia de artefatos de software.

Conversa:

Doug: Vocês dois conseguiram de aprender um pouco sobre métricas de processo e projeto?

Vinod e Jamie: [Ambos acenam a cabeça afirmativamente]

Doug: É sempre uma boa ideia estabelecer metas quando você adota qualquer métrica. Quais são as suas?

Vinod: Nossas métricas devem focar na qualidade. Na verdade, nossa meta geral é manter em um valor mínimo absoluto o número de erros que passamos de uma atividade de engenharia de software para a próxima.

Doug: E ter certeza absoluta de manter o número de defeitos do produto liberado o mais próximo de zero possível.

Vinod (acenando afirmativamente): Naturalmente.

Jamie: Eu gosto da DRE como métrica e acho que podemos usá-la para o projeto inteiro, mas também quando passarmos de uma atividade estrutural para a próxima. Ela nos estimulará a encontrar erros em cada etapa.

Vinod: Eu também gostaria de coletar o número de horas que gastamos em revisões.

Jamie: E o esforço total gasto em cada tarefa de engenharia de software.

Doug: Você pode calcular uma relação entre a revisão e o desenvolvimento... pode ser interessante.

Jamie: Eu gostaria também de monitorar alguns dados de casos de uso. Como, por exemplo, o esforço necessário para desenvolver um caso de uso, o esforço necessário para criar software para implementar um caso de uso e...

Doug (sorrindo): Pensei que iríamos manter tudo simples.

Vinod: Deveríamos, mas, quando se entra nesse negócio de métricas, há muitas coisas interessantes para ver.

Doug: Eu concordo, mas vamos com calma e vamos manter nosso objetivo. Limitem os dados a serem coletados em cinco ou seis itens, e estaremos prontos para começar.

32.4 Integração de métricas dentro do processo de software

A maioria dos desenvolvedores de software ainda não faz medições e, infelizmente, muitos têm pouca vontade de começar. Conforme afirmamos anteriormente neste capítulo, o problema é cultural. Tentar coletar medidas onde nenhuma medida foi coletada no passado muitas vezes causa resistência. “Por que precisamos fazer isso?”, pergunta o gerente de projeto apressado. “Não vejo razão para isso”, concorda um programador muito atarefado.

Nesta seção, vamos considerar alguns argumentos a favor das métricas de software e apresentar uma estratégia para instituir um programa de coleta de métricas em uma empresa de engenharia de software. Mas, antes de começarmos, algumas palavras sábias (agora com quase trinta anos) são oferecidas por Grady e Caswell [Gra87]:

Algumas das coisas que descrevemos aqui parecerão muito fáceis. Na realidade, estabelecer um programa bem-sucedido de métricas de software para a empresa inteira é um trabalho difícil. Quando dizemos que é preciso esperar pelo menos três anos para conhecer as tendências organizacionais, você tem uma ideia da grandeza desse esforço.

Uma advertência sugerida pelos autores é sempre bem-vinda, mas os benefícios das medições são tão atraentes que compensam o esforço.

"Nós controlamos as coisas por números em muitos aspectos da nossa vida... Esses números nos dão uma visão e ajudam a direcionar nossas ações."

**Michael Mah e
Larry Putnam**

32.4.1 Argumentos favoráveis às métricas de software

Por que é tão importante medir o processo de engenharia de software e o artefato que ele produz? A resposta é relativamente clara. Se você não medir, não haverá uma maneira real de determinar se está melhorando. E se você não estiver melhorando, está perdido.

Solicitando e avaliando medidas de produtividade e qualidade, uma equipe de software (e seu gerente) pode estabelecer objetivos claros para melhorias do processo de software. Anteriormente neste livro, destacamos que o software é um assunto estratégico do negócio para muitas empresas. Se o processo por meio do qual ele é desenvolvido pode ser melhorado, disso pode resultar um impacto direto sobre a base. Mas, para estabelecer metas de melhoria, o status atual do desenvolvimento de software deve ser entendido. Logo, a medição é usada para estabelecer um referencial do processo a partir do qual as melhorias podem ser avaliadas.

O rigor do dia a dia do projeto de software deixa pouco tempo para o pensamento estratégico. Os gerentes de projeto de software estão preocupados com problemas mais corriqueiros (mas igualmente importantes): desenvolver estimativas claras de projeto, produzir sistemas de qualidade mais alta, entregar o produto dentro do prazo. Usando medição para estabelecer um referencial de projeto, cada um desses aspectos se torna mais controlável. Já mencionamos que o referencial serve como base para estimativas. Além disso, a coleta de métricas de qualidade permite que a organização "ajuste" seu processo de software para remover as "poucas causas vitais" de defeitos que têm o maior impacto sobre o desenvolvimento do software.⁹

32.4.2 Estabelecimento de um referencial

Estabelecendo-se um referencial para as métricas, outras vantagens podem ser alcançadas nos níveis (técnicos) de processo, projeto e produto. No entanto, as informações coletadas não precisam ser fundamentalmente diferentes. As mesmas métricas podem servir a muitos mestres. O referencial da métrica consiste em dados coletados de projetos de desenvolvimento de software do passado e pode ser tão simples quanto a tabela apresentada na Figura 32.2 ou tão complexo quanto um banco de dados abrangente, contendo dezenas de medidas de projeto e métricas delas derivadas.

Para ajudarem de fato na melhoria do processo e/ou estimativas de custo e esforços, os dados do referencial devem ter os seguintes atributos: (1) os dados devem ser razoavelmente precisos – "chutes" sobre projetos passados devem ser evitados –, (2) devem ser coletados dados de maior número de projetos possível, (3) as medidas devem ser coerentes (por exemplo, uma linha de código deve ser interpretada coerentemente por todos os projetos dos quais são coletados os dados), (4) as aplicações devem ser similares ao trabalho a ser estimado – faz pouco sentido usar um referencial para trabalho com sistemas de informação em lote para estimar uma aplicação em tempo real.

O que é referencial de métricas e quais são os benefícios que ele proporciona a um engenheiro de software?

⁹ Essas ideias foram formalizadas em uma abordagem chamada *garantia estatística de qualidade de software* (*statistical software quality assurance*).

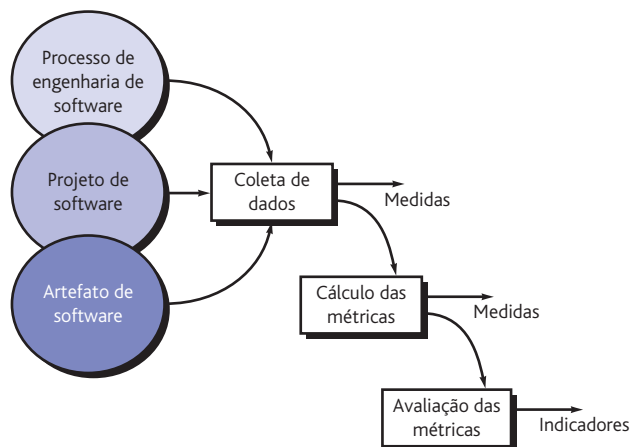


FIGURA 32.3 Processo de coleta de métricas de software.

32.4.3 Coleta, cálculo e avaliação de métricas

O processo para estabelecer um referencial de métricas é ilustrado pela Figura 32.3. Em uma situação ideal, os dados necessários para estabelecer uma referência seriam coletados dinamicamente. Infelizmente, é raro isso acontecer. Portanto, a coleta de dados requer uma investigação histórica de projetos passados para reconstruir os dados necessários. Uma vez coletadas as medidas (sem dúvida, a parte mais difícil), é possível calcular as métricas. Dependendo da amplitude das medidas coletadas, as métricas podem se estender sobre um amplo intervalo de métricas orientadas à aplicação (por exemplo, LOC, FP, orientada a objetos, WebApp), bem como outras métricas orientadas à qualidade e ao projeto. Finalmente, as métricas devem ser avaliadas e aplicadas durante a estimativa, trabalho técnico, controle de projeto e melhoria de processo. A avaliação de métricas se concentra nas razões subjacentes para os resultados obtidos e produz um conjunto de indicadores que guiam o projeto ou o processo.

Se você estiver apenas começando a coletar dados de métricas, lembre-se de mantê-las simples. Se você começar a se aprofundar nos dados, o seu trabalho com métricas não terá sucesso.

32.5 Métricas para empresas pequenas

A grande maioria das empresas de desenvolvimento de software tem menos de 20 profissionais de software. Não é sensato e, na maioria dos casos, não é viável esperar que essas empresas desenvolvam programas abrangentes de métricas de software. No entanto, faz sentido sugerir que organizações de software¹⁰ de todos os tamanhos meçam e depois usem as métricas resultantes para ajudar a melhorar seu processo local de software e a qualidade e prazo dos produtos que produzem.

Uma empresa pequena pode começar focando não na medição, mas sim os resultados. O grupo de software é consultado para definir um objetivo único que exige melhorias. Por exemplo, “reduzir o tempo necessário para

¹⁰ Essa discussão é igualmente relevante para equipes de software que tenham adotado um processo ágil de desenvolvimento de software (Capítulo 5).

Como devemos derivar um conjunto de métricas de software “simples”?

avaliar e implementar solicitações de alterações”. Uma pequena organização pode escolher o seguinte conjunto de medidas, que podem ser obtidas facilmente:

- Tempo (horas ou dias) decorrido desde o instante em que é feita uma solicitação até que a avaliação esteja concluída, t_{fila} .
- Esforço (pessoa/hora) para executar a avaliação, $W_{\text{avaliação}}$.
- Tempo (horas ou dias) decorrido desde o término da avaliação até a atribuição da ordem de alteração para o pessoal, $t_{\text{avaliação}}$.
- Esforço (pessoa/hora) necessário para fazer a alteração, $W_{\text{alteração}}$.
- Tempo (horas ou dias) para fazer a alteração, $t_{\text{alteração}}$.
- Erros descobertos durante o trabalho para fazer a alteração, $E_{\text{alteração}}$.
- Defeitos descobertos depois que a alteração é liberada para o cliente, $D_{\text{alteração}}$.

Uma vez coletadas essas medidas para um conjunto de solicitações da alteração, é possível calcular o tempo total decorrido desde a solicitação da alteração até sua implementação e a porcentagem de tempo gasto na classificação inicial, avaliação e atribuição da mudança e implementação da alteração. De forma semelhante, pode ser determinada a porcentagem do trabalho necessário para a avaliação e implementação. Essas métricas podem ser analisadas no contexto de dados de qualidade, $E_{\text{alteração}}$ e $D_{\text{alteração}}$. A porcentagem permite visualizar onde o processo de solicitação de alteração se retarda e permite conduzir às etapas de melhoria de processo para reduzir t_{fila} , $W_{\text{avaliação}}$, $t_{\text{avaliação}}$, $W_{\text{alteração}}$ e/ou $E_{\text{alteração}}$. Além disso, a eficiência na remoção de defeitos pode ser calculada como

$$\text{DRE} = \frac{E_{\text{alteração}}}{E_{\text{alteração}} + D_{\text{alteração}}}$$

A DRE pode ser comparada com o tempo decorrido e o trabalho total, para determinar o impacto das atividades de garantia de qualidade sobre o tempo e trabalho necessários para fazer uma alteração.

32.6 Estabelecimento de um programa de métricas de software

O Software Engineering Institute desenvolveu um guia [Par96b] para estabelecer um programa de métricas de software “orientado a metas”. O livro sugere as seguintes etapas: (1) identificar suas metas de negócio; (2) identificar o que você quer saber ou aprender; (3) identificar suas submetas; (4) identificar as entidades e atributos relacionados às suas submetas; (5) formalizar suas metas de medição; (6) identificar questões quantificáveis e os indicadores relacionados que vão ser usados para ajudá-lo a atingir as suas metas de medição; (7) identificar os elementos de dados que vão ser coletados para construir os indicadores; (8) identificar as medidas a serem usadas e tornar essas definições operacionais; (9) identificar as ações que você tomará para implementar as medidas; e (10) preparar um plano para

implementar as medidas. Uma discussão detalhada dessas etapas se encontra no guia do SEI. No entanto, temos aqui uma breve visão geral dos pontos principais.

Como o software suporta as funções de negócio, diferencia sistemas ou produtos baseados em computadores ou age como um produto por si mesmo, os objetivos definidos para os negócios quase sempre podem ser ligados a metas específicas em nível de engenharia de software. Por exemplo, considere o produto *CasaSegura*. Trabalhando como uma equipe, a engenharia de software e os gerentes de negócio desenvolvem uma lista de metas comerciais com prioridades:

1. Melhorar a satisfação de nossos clientes com nossos produtos.
2. Tornar os seus produtos mais fáceis de usar.
3. Reduzir o tempo necessário para ter um novo produto no mercado.
4. Tornar o suporte aos nossos produtos mais fácil.
5. Melhorar nossa lucratividade geral.

A empresa de software examina cada meta de negócio e pergunta: “Quais são as atividades que gerenciamos, executamos ou suportamos e o que queremos melhorar nessas atividades?” Para responder a essas perguntas, o SEI recomenda a criação de uma “lista entidade-questão” na qual são arroladas todas as coisas (entidades) dentro do processo de software que são gerenciadas ou influenciadas pela organização de software. Exemplos de entidades incluem recursos de desenvolvimento, artefatos, código-fonte, *test cases*, solicitações de alteração, tarefas de engenharia de software e cronogramas. Para cada entidade listada, os profissionais do software desenvolvem uma série de questões que investigam características quantitativas da entidade (por exemplo, tamanho, custo, tempo para desenvolver). As questões originadas em consequência da criação de uma lista entidade-questão levam à criação de uma série de submetas diretamente relacionada às entidades criadas e às atividades executadas como parte do processo de software.

Considere a quarta meta: “Tornar mais fácil o suporte aos nossos produtos”. Para essa meta, pode ser criada a seguinte lista de questões [Par96b]:

- As solicitações de alterações do cliente contêm as informações de que precisamos para avaliar adequadamente a alteração e então implementá-la dentro do prazo normal?
- Qual é o acúmulo de solicitações de alteração?
- Nosso tempo de resposta para corrigir os erros é aceitável com base nas necessidades do cliente?
- Nosso processo de controle de alterações (Capítulo 29) é seguido?
- As alterações de alta prioridade são implementadas dentro do prazo normal?

Com base nessas questões, a organização de software pode derivar a seguinte submeta: *Melhorar o desempenho do processo de gerenciamento de alterações*. As entidades e atributos do processo de software relevantes à submeta são identificadas e são delineadas as metas de medição associadas a elas.

As métricas de software escolhidas devem ser motivadas pelas metas de negócio e técnicas que deseja atingir.

O SEI IPar96bl fornece instruções detalhadas para as etapas 6 a 10 de sua estratégia de medição motivada por meta. Basicamente, você refina as metas de medição, transformando-as em questões que são ainda mais refinadas, tornando-se entidades e atributos que, então, são refinados, tornando-se métricas.

32.7 Resumo

Medições permitem que gerentes e profissionais melhorem o processo de software; ajudam no planejamento, acompanhamento e controle dos projetos de software; e avaliam a qualidade do artefato de software produzido. Medições de atributos específicos de processo, projeto e produto são usadas para calcular as métricas de software. Essas métricas podem ser analisadas para fornecer indicadores que guiam a gerência e as ações técnicas.

As métricas de processo permitem que a organização tenha uma visão estratégica, fornecendo informações sobre a eficiência de um processo de software. Métricas de projeto são táticas. Elas permitem que o gerente de projeto adapte o fluxo de trabalho do projeto e a abordagem técnica em tempo real.

Na indústria, são usadas métricas orientadas a tamanho e função. Métricas orientadas a tamanho usam a quantidade de linhas de código como fator de normalização para outras medidas, como pessoa/mês ou defeitos. O ponto de função é derivado de medidas do domínio de informações e de uma avaliação subjetiva da complexidade do problema. Além disso, podem ser usadas métricas orientadas a objetos e métricas orientadas a aplicação Web.

Métricas de qualidade de software, como as métricas de produtividade, focam o processo, o projeto e o produto. Desenvolvendo e analisando um referencial de métricas para a qualidade, uma empresa pode corrigir as áreas do processo de software que são a causa dos defeitos de software.

Medições resultam em mudança cultural. Coleta de dados, cálculo das métricas e análise das métricas são as três etapas que devem ser implementadas para se começar um programa de métricas. Em geral, uma estratégia motivada por metas ajuda a empresa a focar nas métricas corretas para seus negócios. Criando um referencial de métricas – um banco de dados contendo medições de processo e produto – os engenheiros de software e seus gerentes podem ter uma visão melhor do trabalho que executam e do produto que produzem.

Problemas e pontos a ponderar

32.1 Descreva com suas próprias palavras a diferença entre métricas de processo e de projeto.

32.2 Por que algumas métricas devem ser mantidas “privadas”? Dê exemplos de três métricas que devem ser privadas. Dê exemplos de três métricas que devem ser públicas.

32.3 O que é uma medida indireta e por que essas medidas são comuns no trabalho com métricas de software?

32.4 Grady sugere uma etiqueta para métricas de software. Você pode acrescentar mais três regras àquelas mencionadas na Seção 32.1.1?

32.5 A Equipe A encontrou 342 erros durante um processo de engenharia de software antes do lançamento. A Equipe B encontrou 184 erros. Que medidas adicionais terão de ser feitas para os projetos A e B para determinar qual das equipes eliminou erros com mais eficiência? Que métricas você poderia propor para ajudar nessa determinação? Que dados históricos podem ser úteis?

32.6 Apresente um argumento contra a adoção do número de linhas de código como medida de produtividade de software. Seu argumento poderá se manter quando forem consideradas dezenas ou centenas de projetos?

32.7 Calcule o valor de pontos de função para um projeto com as seguintes características no domínio de informações:

Número de entradas de usuário: 32

Número de saídas de usuário: 60

Número de consultas de usuário: 24

Número de arquivos: 8

Número de interfaces externas: 2

Suponha que todos os valores de ajuste de complexidade são médios. Use o algoritmo do Capítulo 30.

32.8 Usando a tabela apresentada na Seção 32.2.3, apresente um argumento contra o uso de linguagem Assembler com base na funcionalidade fornecida pelas instruções de código. Novamente referindo-se à tabela, discuta porque C++ seria uma alternativa melhor do que C.

32.9 O software usado para controlar uma fotocopiadora exige 32.000 linhas de C e 4.200 linhas de Smalltalk. Estime o número de pontos de função do software que está na copiadora.

32.10 Uma equipe de engenharia Web criou uma WebApp de e-commerce contendo 145 páginas individuais. Dessas páginas, 65 são dinâmicas, ou seja, são geradas internamente com base em entradas do usuário. Qual é o índice de personalização dessa aplicação?

32.11 Uma WebApp e seu ambiente de suporte não foram totalmente protegidos contra ataques. Os engenheiros Web estimam que a probabilidade de repelir um ataque é de apenas 30%. O sistema não contém informações sigilosas ou comprometedoras; portanto, a probabilidade de ataque é de 25%. Qual é a integridade da WebApp?

32.12 Na conclusão de um projeto foram encontrados 30 erros durante a fase de modelagem e 12 erros durante a fase de construção, ligados a erros que não foram descobertos na fase de modelagem. Qual é a DRE para essas duas fases?

32.13 Uma equipe de software fornece um incremento de software para usuários finais. Os usuários descobrem 8 defeitos durante o primeiro mês de uso. Antes da entrega, a equipe de software encontrou 242 erros durante as revisões técnicas formais e em todas as tarefas de teste. Qual é a DRE geral do projeto após um mês de uso?

Leituras e fontes de informação complementares

A melhoria do processo de software (SPI, software process improvement) tem recebido uma atenção significativa durante as últimas duas décadas. Como as medições e as métricas de software são fundamentais para uma melhora bem-sucedida do processo de software, muitos livros sobre SPI também discutem métricas. Livros de Arban (*Software Metrics and Software Methodology*, Wiley-IEEE Computer Society, 2010) e Rico (*ROI of Software Process Improvement*, J. Ross Publishing, 2004) fornecem uma discussão aprofun-

dada sobre SPI e as métricas que podem ajudar uma organização a atingi-la. Ebert e seus colegas (*Best Practices in Software Measurement*, Springer, 2004) tratam do uso das medições no contexto das normas ISO e CMMI. Kan (*Metrics and Models in Software Quality Engineering*, 2ª ed., Addison-Wesley, 2002) apresenta uma coleção de métricas relevantes.

Ebert e Dumke (*Software Measurement*, Springer, 2007) proporcionam um bom tratamento sobre medições e métricas, conforme devem ser aplicadas para projetos de TI. McGarry e seus colegas (*Practical Software Measurement*, Addison-Wesley, 2001) apresentam conselhos aprofundados para avaliação de processo de software. Uma boa coleção de artigos foi editada por Haug e seus colegas (*Software Process Improvement: Metrics, Measurement, and Process Modeling*, Springer-Verlag, 2001). Florac e Carlton (*Measuring the Software Process*, Addison-Wesley, 1999) e Fenton e Pfleeger (*Software Metrics: A Rigorous and Practical Approach*, Revised, Brooks/Cole Publishers, 1998) discutem como as métricas de software podem ser usadas para proporcionar os indicadores necessários para melhorar o processo de software.

Wohlin e seus colegas (*Experimentation in Software Engineering*, Springer, 2012) discutem o modo de usar medidas na análise de processo de software. Jones (*Applied Software Measurement: Global Analysis of Productivity and Quality*, McGraw-Hill, 2008), Laird e Brennan (*Software Measurement and Estimation*, Wiley-IEEE Computer Society Press, 2006) e Goodman (*Software Metrics: Best Practices for Successful IT Management*, Rothstein Associates, 2004) discutem o uso das métricas de software para gerenciamento de projeto e estimativas. Putnam e Myers (*Five Core Metrics*, Dorset House, 2003) usam um banco de dados de mais de 6.000 projetos de software para demonstrar como cinco métricas fundamentais – tempo, trabalho, tamanho, confiabilidade e produtividade de processo – podem ser usadas para controlar projetos de software. Maxwell (*Applied Statistics for Software Managers*, Prentice Hall, 2003) apresenta técnicas para analisar dados de projeto de software. Munson (*Software Engineering Measurement*, Auerbach, 2003) discute um amplo conjunto de aspectos de medição em engenharia de software. Jones (*Software Assessments, Benchmarks and Best Practices*, Addison-Wesley, 2000) descreve medições quantitativas e qualitativas que ajudam uma empresa a avaliar seu processo e suas práticas de software.

A medição de pontos de função tornou-se uma técnica amplamente utilizada em muitas áreas da engenharia de software. O International Function Point Users Group publicou uma coleção de artigos sobre o uso de métricas de ponto de função (*The IFPUC Guide to IT and Software Measurement*, Auerbach, 2012). Parthasarathy (*Practical Software Estimation: Function Point Methods for Insourced and Outsourced Projects*, Addison-Wesley, 2007) fornece um guia abrangente. Garmus e Herron (*Function Point Analysis: Measurement Practices for Successful Software Projects*, Addison-Wesley, 2000) discutem métricas de processo com ênfase na análise de pontos de função.

Relativamente pouca coisa foi publicada sobre métricas para o trabalho de engenharia para Web. Contudo, Clifton (*Advanced Web Metrics with Google Analytics*, 3ª ed., Sybex, 2012), Kaushik (*Web Analytics 2.0: Accountability and Science of Customer Centricity*, Sybex, 2009; e *Web Analytics: An Hour a Day*, Sybex, 2007), Stern (*Web Metrics: Proven Methods for Measuring Web Site Success*, Wiley, 2002), Inan e Kean (*Measuring the Success of Your Website*, Longman, 2002) e Nobles e Grady (*Web Site Analysis and Reporting*, Premier Press, 2001) tratam das métricas Web do ponto de vista de negócios e marketing.

A mais recente pesquisa na área de métricas está resumida pelo IEEE (*Symposium on Software Metrics*, publicado anualmente). Uma grande variedade de fontes de informações sobre métricas de processo e projeto está disponível na Internet. Uma lista atualizada das referências da Web pode ser encontrada em “recursos de engenharia de software” no site da SEPA: www.mhhe.com/pressman.

Estimativas de projeto de software

33

O gerenciamento do projeto de software começa com um conjunto de atividades chamadas coletivamente de *planejamento de projeto*. Antes de iniciar o projeto, a equipe de software deve fazer uma estimativa do trabalho, dos recursos que serão necessários e do tempo necessário para a sua conclusão. Uma vez cumpridas essas atividades, a equipe de projeto deve estabelecer um cronograma que defina as tarefas e as metas de engenharia de software, deve identificar os responsáveis pela execução de cada tarefa e deve especificar as dependências entre tarefas que podem ter forte influência no progresso do trabalho.

Em um excelente “guia de sobrevivência de projeto de software”, Steve McConnell [McC98] apresenta uma visão do planejamento de projeto na prática:

Muitos profissionais preferem ir direto ao trabalho em vez de investir tempo planejando. Muitos gerentes técnicos não têm treinamento suficiente em gerenciamento para terem confiança de que o seu planejamento vai melhorar os resultados de um projeto. Como nenhuma das partes quer fazer o planejamento, a tarefa em geral acaba não sendo feita.

Conceitos-chave

casos de uso	740
desenvolvimento ágil	746
estimativa	
baseada em problema	735
baseada em processo	739
baseada em FP	738
fazer/comprar	748
harmonização	742
modelos empíricos	743
projetos orientados a objetos	746
técnicas de decomposição	734
terceirização	750
WebApps	747

PANORAMA

O que é? Uma necessidade real de software foi estabelecida; os envolvidos estão incluídos, os engenheiros de software estão prontos para começar e o projeto está para ser iniciado. E como você procede? O planejamento do projeto de software abrange cinco atividades importantes: estimativa, cronograma, análise de risco, planejamento da gestão da qualidade e planejamento do gerenciamento de alterações. Neste capítulo, veremos somente as estimativas – a tentativa de determinar quanto dinheiro, esforço, recursos e tempo serão necessários para criar um sistema ou artefato baseado em software específico.

Quem realiza? Os gerentes de projeto de software – usando informações recebidas dos envolvidos no projeto e dados das métricas de software coletados de projetos anteriores.

Por que é importante? Você construiria uma casa sem saber quanto está disposto a gastar, as tarefas que precisam ser feitas e os prazos para executar o trabalho? É claro que não, e, como a maioria dos sistemas e produtos baseados em computadores custa consideravelmente mais para ser construída do que a construção de uma grande casa, parece bastante sen-

sato desenvolver uma estimativa antes de começar a criar o software.

Quais são as etapas envolvidas? As estimativas começam com uma descrição do escopo do produto. Então, o problema é decomposto em uma série de problemas menores, e cada um desses problemas é estimado usando dados históricos e a experiência como guia. A complexidade e os riscos do problema são considerados antes que a estimativa final seja feita.

Qual é o artefato? É gerada uma tabela simples descrevendo as tarefas a executar, as funções a implementar e o custo, esforço e tempo envolvidos para cada atividade.

Como garantir que o trabalho foi realizado corretamente? Isso é difícil, porque não é possível saber até que o projeto seja finalizado. No entanto, se você tiver experiência e seguir uma estratégia sistemática, se gerar estimativas usando dados históricos confiáveis, se criar dados de estimativa usando pelo menos dois métodos diferentes, se estabelecer um cronograma realista e adaptá-lo continuamente à medida que o projeto avançar, pode ter certeza de que fez a melhor escolha.

dimensionamento	
do software	734
equação do software ...	744
escopo do software	730
planejamento do	
projeto	729
pontos de caso de uso	
(UCPs)	742
recursos	731

Porém, a falta de planejamento é um dos erros mais graves que um projeto pode cometer... é necessário um planejamento eficaz para resolver problemas anteriores [no início do projeto] com custo baixo, em vez de mais adiante [tardamente no projeto], com um alto custo. Os projetos consomem, em média, 80% do tempo em retrabalho – correção de erros cometidos no início.

McConnell alega que todo projeto pode arranjar espaço para planejar (e adaptar o plano durante o projeto) simplesmente reservando-se uma pequena porcentagem do tempo que teria sido gasto no retrabalho porque o planejamento não foi feito.

33.1 Observações sobre as estimativas

O planejamento exige que você assuma um compromisso inicial, mesmo que mais tarde ele venha se mostrar errado. Sempre que forem feitas estimativas, deve-se olhar o futuro e aceitar certo grau de incerteza. Segundo Frederick Brooks [Bro95]:

... nossas técnicas de estimativa são muito mal desenvolvidas. E, mais grave ainda, refletem uma suposição um tanto falsa, não declarada, de que tudo sairá bem... como não temos certeza sobre nossas estimativas, gerentes de software muitas vezes não têm a firmeza necessária para fazer as pessoas esperarem por um bom produto.

"Boas estratégias de estimativa e dados históricos sólidos são a maior esperança de que a realidade vencerá as demandas impossíveis."

Caper Jones

Embora a estimativa seja muito mais arte do que ciência, ela não precisa ser conduzida de maneira aleatória. Existem técnicas úteis para estimar tempo e esforço. As métricas de projeto e processo podem proporcionar perspectivas históricas e informações valiosas para gerar estimativas quantitativas. A experiência (de todos os envolvidos) pode ajudar imensamente à medida que as estimativas são desenvolvidas e revisadas. Por serem a base de todas as outras ações do planejamento de projeto, e pelo fato de o planejamento do projeto fornecer a direção para uma engenharia de software bem-sucedida, seria uma péssima ideia iniciar sem as estimativas.

As estimativas de recursos, custos e cronograma para um trabalho de engenharia de software exigem experiência, acesso a boas informações históricas (métricas) e a coragem de se comprometer com as previsões quantitativas – quando tudo o que existe são apenas informações qualitativas. A estimativa traz um risco inerente,¹ e esse risco leva à incerteza.

A *complexidade do projeto* tem um forte efeito sobre a incerteza inerente ao planejamento. No entanto, é uma medida relativa afetada pela familiaridade com esforços passados. Uma pessoa que desenvolve pela primeira vez uma aplicação sofisticada para comércio eletrônico pode considerá-la excessivamente complexa. No entanto, uma equipe de engenharia para Web desenvolvendo sua décima WebApp para comércio eletrônico consideraria isso um trabalho comum. Já foram propostas várias medidas de complexidade quantitativa de software [Zus97]. Elas são aplicadas em nível de projeto ou código e são, portanto, difíceis de usar durante planejamento de software (antes do projeto e

¹ Técnicas sistemáticas para análise de riscos são apresentadas no Capítulo 35.

do código). No entanto, outras avaliações mais subjetivas de complexidade (por exemplo, fatores de ajuste de complexidade de pontos de função, descritos no Capítulo 30) podem ser estabelecidas no início do processo de planejamento.

O *tamanho do projeto* é outro fator importante que pode afetar a precisão e a eficácia das estimativas. À medida que o tamanho aumenta, a interdependência entre os vários elementos do software cresce rapidamente.² A decomposição do problema, uma estratégia importante para a estimativa, torna-se mais difícil, porque o refinamento dos elementos envolvidos pode ainda ser considerável. Parafraseando a lei de Murphy: “O que pode sair errado, sairá errado” – e, se houver mais coisas que podem falhar, mais coisas falharão.

O *grau de incerteza estrutural* também tem um efeito sobre o risco das estimativas. Nesse contexto, estrutura refere-se ao grau segundo o qual os requisitos foram solidificados, a facilidade com a qual as funções podem ser separadas e a natureza hierárquica das informações a serem processadas.

A disponibilidade de informações históricas tem uma forte influência sobre o risco das estimativas. Utilizar procedimentos que funcionaram pode melhorar as áreas problemáticas. Quando existem métricas de software abrangentes (Capítulo 32) disponíveis de projetos passados, as estimativas podem ser feitas com maior segurança, podem ser estabelecidos os cronogramas para evitar dificuldades passadas e o risco em geral é reduzido.

O risco das estimativas é medido pelo grau de incerteza nas estimativas quantitativas estabelecidas para recursos, custo e cronograma. Se o escopo do projeto é mal entendido ou se os requisitos do projeto sofrem alterações, a incerteza e o risco das estimativas tornam-se perigosamente altos. Como planejador, você (e o cliente) deve reconhecer que variabilidade nos requisitos de software significa instabilidade nos custos e no cronograma.

No entanto, você não deve se tornar obsessivo em relação às estimativas. Estratégias modernas de engenharia de software (por exemplo, modelos incrementais de processo) presumem uma visão iterativa do desenvolvimento. Em tais estratégias, é possível – embora nem sempre politicamente aceitável – voltar à estimativa (conforme mais informações são conhecidas) e revisá-la quando o cliente fizer alterações nos requisitos.

A complexidade, o tamanho e o grau de incerteza estrutural do projeto afetam a confiabilidade das estimativas.

“É característica de uma mente instruída satisfazer-se com o grau de precisão que a natureza do assunto permite e não procurar exatidão quando apenas uma aproximação da verdade é possível.”

Aristóteles

33.2 O processo de planejamento do projeto

O objetivo do planejamento de software é proporcionar uma estrutura, um framework que permita ao gerente fazer estimativas adequadas de recursos, custo e cronograma. Além disso, as estimativas devem tentar definir cenários de melhor e pior caso para que os resultados do projeto possam ser delineados. Embora haja um grau de incerteza inerente, a equipe de software participa de um plano estabelecido como consequência dessas tarefas. O plano deve ser adaptado e atualizado à medida que o projeto avança. Nas próximas seções, discutiremos cada uma das atividades associadas ao planejamento de projeto de software.

Quanto mais você sabe, melhor você estima. Portanto, atualize suas estimativas à medida que o projeto avança.

² O tamanho muitas vezes aumenta devido ao “deslizamento do escopo” que ocorre quando os requisitos do problema mudam. O aumento no tamanho do projeto pode ter um impacto geométrico sobre o custo e o cronograma do projeto (Michael Mah, comunicação pessoal).

CONJUNTO DE TAREFAS



Conjunto de tarefas para planejamento de projeto

1. Estabeleça o escopo do projeto.
2. Determine a viabilidade.
3. Analise os riscos (Capítulo 35).
4. Defina os recursos necessários.
 - a. Determine os recursos humanos necessários.
 - b. Defina os recursos de software reutilizáveis.
 - c. Identifique os recursos ambientais.
5. Estime o custo e a mão de obra.
 - a. Decomponha o problema.
 - b. Desenvolva duas ou mais estimativas usando tamanho, pontos de função, tarefas de processo ou casos de uso.
 - c. Harmonize as estimativas.
6. Desenvolva um cronograma de projeto (Capítulo 34).
 - a. Estabeleça um conjunto significativo de tarefas.
 - b. Defina uma rede de tarefas.
 - c. Use ferramentas de cronograma para desenvolver um diagrama de tempos.
 - d. Defina mecanismos para acompanhamento do cronograma.

33.3 Escopo e viabilidade do software

O *escopo do software* descreve as funções e características que devem ser fornecidas aos usuários; os dados que entram e saem, o “conteúdo” que é apresentado aos usuários como consequência do uso do software e o desempenho, restrições, interfaces e confiabilidade que *limitam* o sistema. O escopo é definido por meio de uma das duas técnicas:

1. Uma descrição narrativa do escopo do software é desenvolvida após comunicação com todos os envolvidos.
2. Um conjunto de casos de uso³ é desenvolvido pelos usuários.

Funções descritas na definição do escopo (ou nos casos de uso) são avaliadas – e, em algumas situações, refinadas – para fornecer mais detalhes, antes de iniciar as estimativas. Como as estimativas de custo e cronograma são ambas funcionalmente orientadas, muitas vezes é aconselhável certo grau de decomposição. As considerações de desempenho abrangem requisitos de processamento e tempo de resposta. As restrições identificam limites colocados no software por hardware externo, memória disponível ou outros sistemas existentes.

Uma vez identificado o escopo (com a participação do cliente), pergunte: “Podemos criar software que atenda a esse escopo? O projeto é viável?”. Com muita frequência, os engenheiros de software passam rapidamente por essas questões (ou são empurrados por gerentes impacientes ou outras pessoas envolvidas), somente para se envolverem em um projeto que já está condenado desde o início.

Putnam e Myers [Put97a] sugerem que apenas delimitar escopo não é suficiente. Uma vez entendido o escopo, deve-se trabalhar para determinar se ele pode ser concluído segundo as dimensões da tecnologia, do orçamento,

A viabilidade do projeto é importante, mas uma consideração das necessidades do negócio é ainda mais importante. Não é uma boa ideia criar um sistema ou produto de alta tecnologia que ninguém quer.

³ Casos de uso foram discutidos em detalhe na Parte II deste livro. Um caso de uso é uma descrição baseada em cenário da interação do usuário com o software sob o ponto de vista do usuário.

do tempo e dos recursos disponíveis. Essa é uma parte crucial do processo de estimativas, que muitas vezes passa despercebida.

33.4 Recursos

A segunda tarefa de planejamento é estimar os recursos necessários para executar o trabalho de desenvolvimento de software. A Figura 33.1 mostra as três principais categorias de recursos de engenharia de software – pessoas, componentes de software reutilizáveis e ambiente de desenvolvimento (hardware e ferramentas de software). Cada recurso é especificado com quatro características: descrição do recurso, uma definição da disponibilidade, instante em que o recurso será necessário e tempo durante o qual o recurso será exigido. As duas últimas podem ser vistas como uma *janela de tempo*. A disponibilidade do recurso para uma janela de tempo especificada deve ser estabelecida o mais cedo possível.

33.4.1 Recursos humanos

O planejador começa avaliando o escopo do software e selecionando as habilidades necessárias para concluir o desenvolvimento. São especificados o cargo (por exemplo, gerente, engenheiro de software sênior) e a especialização (por exemplo, telecomunicações, banco de dados, cliente-servidor). Para projetos relativamente pequenos (poucas pessoas/mês), um só profissional pode executar todas as tarefas de engenharia de software, consultando os especialistas quando necessário. Para projetos maiores, a equipe pode estar geograficamente dispersa em diferentes locais. Portanto, é especificada a localização de cada recurso humano.

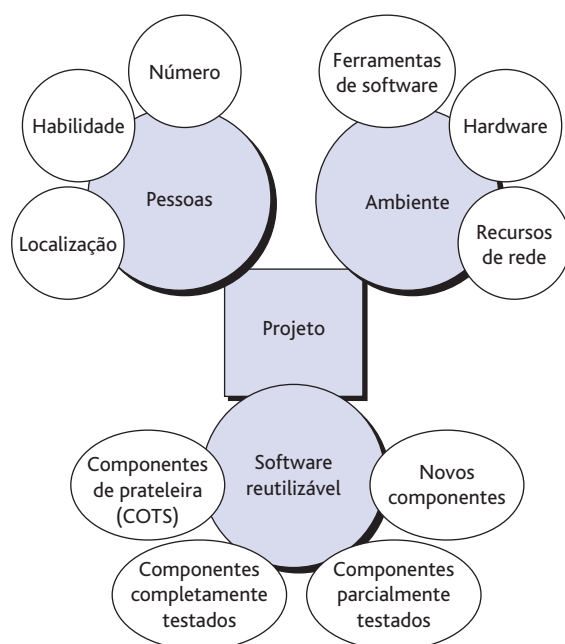


FIGURA 33.1 Recursos de projeto.

O número de pessoas necessárias para um projeto de software pode ser determinado somente após uma estimativa do esforço de desenvolvimento (por exemplo, pessoas-mês). As técnicas para estimar o esforço são discutidas mais adiante neste capítulo.

33.4.2 Recursos de software reutilizáveis

A engenharia de software baseada em componentes (CBSE, component-based software engineering)⁴ enfatiza a capacidade de reutilização – isto é, a criação e reutilização de blocos básicos de software. Esses blocos básicos, chamados também de *componentes*, devem ser catalogados para facilitar a referência, padronizados para facilitar a aplicação e validados para facilitar a integração. Bennatan [Ben00] sugere quatro categorias de recursos de software que deverão ser consideradas conforme o planejamento progride: *componentes de prateleira* (software já existente que pode ser adquirido de terceiros ou de um projeto passado), *componentes completamente experimentados* (especificações, projetos, código ou dados de teste já existentes, desenvolvidos para projetos passados semelhantes ao software a ser construído para o projeto atual), *componentes parcialmente experimentados* (especificações, projetos, código ou dados de teste já existentes, desenvolvidos para projetos passados, relacionados ao software a ser construído para o projeto atual, mas que exigirão modificação significativa) e *componentes novos* (componentes construídos pela equipe de software especificamente para as necessidades do projeto atual).

Ironicamente, os componentes de software reutilizáveis são muitas vezes negligenciados durante o planejamento, só para se tornarem uma preocupação suprema durante a fase de desenvolvimento do processo de software. É aconselhável especificar o quanto antes os requisitos de recursos do software. Dessa maneira, pode ser feita a avaliação técnica das alternativas; e a aquisição, realizada a tempo.

33.4.3 Recursos ambientais

O ambiente que suporta um projeto de software, muitas vezes chamado de *ambiente de engenharia de software* (SEE, *software engineering environment*), incorpora hardware e software. O hardware fornece uma plataforma que suporta as ferramentas (software) necessárias para produzir os artefatos resultantes de uma boa prática de engenharia de software.⁵ Como a maioria das organizações de software tem vários envolvidos no acesso ao SEE, deve-se prescrever a janela de tempo necessária para hardware e software e verificar se esses recursos estarão disponíveis.

Quando um sistema baseado em computador (incorporando hardware e software especializado) precisa ser desenvolvido, a equipe pode necessitar de acesso a elementos de hardware que estão sendo desenvolvidos por outras equipes de engenharia. Por exemplo, o software para um dispositivo robótico usado em uma célula de manufatura pode necessitar de um robô específico

⁴ CBSE foi tratado no Capítulo 14.

⁵ Outro hardware – o *ambiente-alvo* – é o computador no qual o software será executado quando liberado para o cliente.

(por exemplo, um soldador robótico) como parte da etapa de teste de validação; um projeto avançado para layout de página pode necessitar de um sistema digital de impressão de alta velocidade em algum instante durante o desenvolvimento. Cada elemento de hardware precisa ser especificado como parte do planejamento.

33.5 Estimativa do projeto de software

As estimativas de custo e esforço nunca serão uma ciência exata. Muitas variáveis – fatores humanos, técnicos, ambientais e políticos – podem afetar o custo final do software e o esforço necessário para desenvolvê-lo. No entanto, as estimativas de projeto de software podem ser transformadas de algo sobrenatural para uma série de etapas sistemáticas que proporcionam estimativas com um risco aceitável. Para conseguir estimativas confiáveis de custo e esforço, surge uma série de opções:

1. Adiar a estimativa para mais adiante no projeto (obviamente, podemos conseguir uma estimativa com precisão de 100% depois que o projeto estiver concluído!).
2. Fundamentar suas estimativas em projetos similares que já foram concluídos.
3. Usar técnicas de decomposição relativamente simples para gerar estimativas de custo de projeto e esforço.
4. Usar um ou mais modelos empíricos para estimativa de custo e esforço do software.

Infelizmente, a primeira opção, embora atraente, não é prática. As estimativas de custo devem ser feitas no início. Entretanto, é preciso reconhecer que, quanto mais esperar, mais informações você terá da realidade – e menor será a probabilidade de que cometa erros graves nas suas estimativas.

A segunda opção pode funcionar razoavelmente bem se o projeto atual for muito parecido com trabalhos anteriores e outras influências de projeto (por exemplo, o cliente, as condições comerciais, o ambiente de engenharia de software, prazos finais) forem quase equivalentes. Infelizmente, a experiência não tem sido sempre um bom indicador de resultados futuros.

As demais opções são estratégias viáveis para estimativa de projeto de software. Em condições ideais, as técnicas mencionadas para cada opção devem ser aplicadas em paralelo; cada uma sendo usada para verificar a outra. As técnicas de decomposição assumem uma estratégia do tipo “dividir para conquistar” para a estimativa do projeto de software. Decompondo-se um projeto em suas funções principais e atividades de engenharia de software relacionadas, as estimativas de custo e esforço podem ser feitas passo a passo. Podem ser usados modelos empíricos de estimativa para complementar as técnicas de decomposição e oferecer uma estratégia de estimativa potencialmente valiosa por si mesma. Um modelo é baseado na experiência (dados históricos) e assume a seguinte forma:

$$d = f(v_i)$$

“Em uma época de terceirização e concorrência cada vez maiores, a habilidade de estimar com mais precisão... emergiu como um fator crítico de sucesso para muitos grupos de TI.”

Rob Thomsett

“É muito difícil fazer uma defesa vigorosa, plausível e arriscada de uma estimativa não originada de um método quantitativo, suportada por dados escassos e certificada principalmente pelas pressões dos gerentes.”

Fred Brooks

onde d é um entre uma série de valores estimados (por exemplo, esforço, custo, duração do projeto) e v_i são parâmetros independentes selecionados (por exemplo, LOC estimado ou FP).

Ferramentas automáticas de estimativa implementam uma ou mais técnicas de decomposição ou modelos empíricos, sendo uma opção atraente para estimativas. Em tais sistemas, são descritas as características da organização de desenvolvimento (por exemplo, experiência, ambiente) e o software a ser desenvolvido. Com base nesses dados, o custo e o esforço são estimados.

Todas dessas opções viáveis de estimativa de custo de software só são úteis se os dados históricos usados para compor a estimativa são bons. Se não existem dados históricos, os custos se baseiam em uma fundação muito precária. No Capítulo 32, examinamos as características de algumas das métricas de software que proporcionam a base para dados de estimativa histórica.

33.6 Técnicas de decomposição

A estimativa de projeto de software é um método de solução de problemas e, na maioria dos casos, o problema a ser resolvido (desenvolver uma estimativa de custo e esforço para um projeto de software) é muito complexo para ser considerado como uma coisa só. Por essa razão, você deve decompor o problema, redefinindo-o como uma série de problemas menores (e, talvez, mais controláveis).

No Capítulo 31, a abordagem de decomposição foi discutida a partir de dois pontos de vista: decomposição do problema e decomposição do processo. As estimativas usam uma ou ambas as formas de divisão; mas, antes de fazer uma estimativa, entenda o escopo do software a ser criado e faça uma estimativa de seu “tamanho”.

33.6.1 Dimensionamento do software

A precisão de uma estimativa de projeto de software é baseada em vários itens: (1) até que ponto o tamanho do produto a ser construído foi estimado corretamente; (2) a capacidade de transformar a estimativa de tamanho em trabalho humano, calendário e dinheiro (uma função da disponibilidade de métricas de software confiáveis de projetos passados); (3) até que ponto o plano de projeto reflete as habilidades da equipe de software; e (4) a estabilidade dos requisitos do produto e do ambiente que apoia o trabalho de engenharia de software.

Como a estimativa do projeto é apenas tão boa quanto a estimativa do trabalho a ser realizado, o *dimensionamento do software* representa seu primeiro grande desafio como planejador. No contexto do planejamento de projeto, tamanho refere-se a um resultado quantificável do projeto de software. Se for adotada uma abordagem direta, o tamanho pode ser medido em linhas de código (LOC). Se for escolhida uma abordagem indireta, o tamanho é representado como pontos de função (FP). O tamanho pode ser estimado considerando-se o tipo de projeto e seu domínio de aplicação, a funcionalidade entregue (isto é, o número de pontos de função), o número de componentes a ser entregue e até que ponto um conjunto de componentes existentes deve ser modificado para o novo sistema.

O “tamanho” do software a ser criado pode ser estimado usando-se uma medida direta, LOC, ou uma medida indireta, FP.

Putnam e Myers [Put92] sugerem que os resultados de todas essas abordagens de dimensionamento sejam combinados estatisticamente para criar uma estimativa de *três pontos* ou *valor esperado*. Isso é obtido desenvolvendo-se valores otimistas (baixo), mais prováveis e pessimistas (alto) para o tamanho e combinando-os por meio da Equação (33.1), descrita na Seção 33.6.2.

33.6.2 Estimativa baseada em problema

No Capítulo 32, linhas de código (LOC) e pontos de função (PF) foram descritos como medidas a partir das quais as métricas de produtividade podem ser calculadas. Dados de LOC e FP são usados de duas maneiras durante a estimativa do projeto de software: (1) como variáveis de estimativa para “dimensionar” cada elemento do software e (2) como métricas de referência coletadas de projetos anteriores e utilizadas em conjunto com variáveis de estimativa para desenvolver projeções de custo e esforço.

Estimativas LOC e FP são técnicas distintas, mas que têm muitas características em comum. Inicia-se com uma definição delimitada do escopo do software e então tenta-se decompor a definição em funções de problemas que podem ser estimados individualmente. LOC ou FP (a variável de estimativa) é então estimada para cada função. Como alternativa, pode-se escolher outro componente para dimensionamento, como classes ou objetos, alterações ou processos de negócio afetados.

Métricas de produtividade de referência (por exemplo, LOC/pm ou FP/pm)⁶ são aplicadas à variável de estimativa apropriada e, assim, obtém-se o custo ou esforço para a função. As estimativas de função são combinadas para produzir uma estimativa geral para todo o projeto.

É importante observar, porém, que muitas vezes há uma dispersão substancial em métricas de produtividade para uma organização, não sendo aconselhável o uso de uma única métrica de produtividade de referência. Em geral, médias LOC/pm ou FP/pm devem ser calculadas por domínio de projeto. Os projetos devem ser agrupados por tamanho de equipe, por área de aplicação, complexidade e outros parâmetros relevantes. Devem ser calculadas as médias locais do domínio. Quando um novo projeto é estimado, deve primeiramente ser alocado a um domínio e, depois, a média do domínio apropriada para produtividade deve ser usada para gerar a estimativa.

As técnicas de estimativa LOC e FP diferem no nível de detalhe exigido para a decomposição e no alvo da divisão. Quando é usada a LOC como variável de estimativa, a decomposição é absolutamente essencial e muitas vezes é adotada com níveis consideráveis de detalhes. Quanto maior o grau de divisão, maior a probabilidade de serem desenvolvidas estimativas LOC razoavelmente precisas.

Para estimativas FP, a decomposição funciona de forma diferente. Em vez de se concentrar na função, é estimada cada uma das características do domínio de informação – entradas, saídas, arquivos de dados, consultas e interfaces externas –, bem como os 14 valores de ajuste de complexidade discutidos no

O que as estimativas baseadas em LOC e FP têm em comum?

Ao coletar métricas de produtividade para projetos, estabeleça uma classificação de tipos de projeto. Isso permitirá o cálculo de médias específicas do domínio, tornando a estimativa mais precisa.

⁶ O acrônimo *pm* significa pessoa-mês de trabalho.

Capítulo 30. As estimativas resultantes podem, então, ser usadas para derivar um valor de FP que pode ser relacionado a dados anteriores e esse FP pode ser usado para gerar uma estimativa.

Independentemente da variável de estimativa utilizada, deve-se começar estimando um intervalo de valores para cada valor de função ou domínio de informação. Usando dados históricos ou (quando todo o resto falhar) intuição, estimam-se valores de tamanho otimista, mais provável e pessimista para cada função ou contagem para cada valor do domínio de informações. Quando um intervalo de valores é especificado, é fornecida uma indicação implícita do grau de incerteza.

Um valor esperado ou de três pontos pode, então, ser calculado. O *valor esperado* para a variável de estimativa (tamanho) S pode ser calculada como uma média ponderada das estimativas otimista (s_{opt}), mais provável (s_m) e pessimista (s_{pess}). Por exemplo,

$$S = \frac{s_{opt} + 4s_m + s_{pess}}{6} \quad (33.1)$$

oferece a maior credibilidade da estimativa “mais provável” e segue uma distribuição beta de probabilidade. Supomos que há uma probabilidade muito pequena de que o tamanho real fique fora dos valores otimista e pessimista.

Determinado o valor esperado para a variável de estimativa, aplicam-se os dados de produtividade histórica de LOC ou FP. As estimativas estão corretas? A única resposta razoável para essa pergunta é: não temos certeza. Qualquer técnica de estimativa, não importa o quanto seja sofisticada, deve passar por uma verificação cruzada com outra estratégia. Mesmo assim, o bom senso e a experiência devem prevalecer.

33.6.3 Um exemplo de estimativa baseada em LOC

Como exemplo das técnicas de estimativa LOC e FP baseadas em problema, vamos considerar um pacote de software a ser desenvolvido para uma aplicação de projeto auxiliado por computador para componentes mecânicos. O software deve ser executado em uma estação de trabalho de mesa e fazer interface com vários periféricos gráficos, incluindo um mouse, um digitalizador (mesa digitalizadora ou scanner), um monitor colorido de alta resolução e uma impressora a laser. Pode ser formulada uma definição preliminar do escopo do software:

O software CAD mecânico aceitará dados geométricos bidimensionais e tridimensionais fornecidos por um designer. O designer vai interagir e controlar o sistema CAD por meio de uma interface de usuário que vai exibir características de um bom design de interface homem-máquina. Todos os dados geométricos e outras informações de suporte serão mantidos em um banco de dados CAD. Serão desenvolvidos módulos de análise de projeto para produzir a saída exigida, a ser exibida em uma variedade de dispositivos. O software será projetado para controlar e interagir com dispositivos periféricos que incluem um mouse, um digitalizador, uma impressora a laser e um plotter.

Essa definição de escopo é preliminar – não é delimitada. Cada frase deverá ser expandida para proporcionar o detalhe concreto e limites quantitativos. Por exemplo, antes de iniciar a estimativa, o planejador deve determinar

Como calculamos o “valor esperado” para o tamanho do software?

Muitas aplicações modernas residem em uma rede ou fazem parte de uma arquitetura cliente-servidor. Portanto, certifique-se de que suas estimativas incluam a mão de obra exigida para desenvolver software de “infraestrutura”.

Função	LOC estimado
Interface de usuário e recurso de controle	2.300
Análise geométrica bidimensional	5.300
Análise geométrica tridimensional	6.800
Gerenciamento de banco de dados	3.350
Recursos de visualização da computação gráfica	4.950
Função de controle de periféricos	2.100
Módulos de análise do projeto	8.400
<i>Linhas de código estimadas</i>	<i>33.200</i>

FIGURA 33.2 Tabela de estimativas para o método LOC.

o que significa “bom projeto de interface homem-máquina” ou qual deverá ser o tamanho e a sofisticação do “banco de dados CAD”.

Para os propósitos deste livro, suponha que ocorreu um refinamento adicional e que as principais funções do software listadas na Figura 33.2 estão identificadas. Seguindo a técnica de decomposição para LOC, é desenvolvida uma tabela de estimativas (Figura 33.2). Para cada função é desenvolvido um intervalo de estimativas LOC. Por exemplo, o intervalo de estimativas LOC para a função de análise geométrica 3D é otimista, 4.600 LOC; mais provável, 6.900 LOC; e pessimista, 8.600 LOC. Aplicando a Equação (33.1), o valor esperado para a função de análise geométrica 3D é 6.800 LOC. Outras estimativas são obtidas de forma semelhante. Somando verticalmente na coluna de estimativa LOC, obtém-se uma estimativa de 33.200 linhas de código para o sistema CAD.

Um exame de dados históricos indica que a produtividade média organizacional para sistemas desse tipo é de 620 LOC/pm. Com base em um valor bruto da mão de obra de \$ 8 mil por mês, o custo por linha de código é de aproximadamente \$ 13. Com base na estimativa LOC e dados históricos de produtividade, o custo total estimado do projeto é de \$ 431 mil e o esforço estimado é de 54 pessoas-mês.⁷

Não ceda à tentação de usar esse resultado como sua estimativa de projeto. Você deve extrair outro resultado usando uma estratégia diferente.



Estimativa

Cena: Escritório de Doug Miller no início do planejamento.

Atores: Doug Miller (gerente da equipe de engenharia de software do CasaSegura), Vinod Raman, Jamie Lazar e outros membros da equipe.

Conversa:

Doug: Precisamos desenvolver uma estimativa de esforço para o projeto e depois definir um microcronograma para o primeiro incremento e um macrocronograma para os demais incrementos.

CASASEGURA

⁷ As estimativas são arredondadas para o próximo \$ 1.000 e pessoa-mês. Não é necessário, nem realista, ter maior precisão, dadas as limitações da exatidão da estimativa.

Vinod (acenando afirmativamente): Certo, mas não definimos nenhum incremento ainda.

Doug: Sim, mas é por isso que precisamos fazer estimativas.

Jamie (contrariado): Você quer saber quanto tempo vamos gastar para fazer isso?

Doug: Olha só o que eu preciso. Primeiro, temos que decompor o software *CasaSegura* do ponto de vista funcional... depois, temos de estimar o número de linhas de código que cada função terá... depois...

Jamie: Espera aí! Como você acha que vamos fazer isso?

Vinod: Já fizemos em projetos anteriores. Você começa com casos de uso, determina a funcionalidade exigida para implementar cada um e então arrisca um palpite quanto ao número de LOC para cada parte da função. A melhor estra-

tégia é todos fazerem isso de forma independente e depois comparamos os resultados.

Doug: Ou você pode fazer uma decomposição funcional do projeto inteiro.

Jamie: Mas isso vai demorar uma eternidade, e precisamos começar.

Vinod: Não... isso pode ser feito em algumas horas... na verdade, nesta manhã.

Doug: Concordo... não podemos esperar exatidão, apenas uma vaga ideia do tamanho que o *CasaSegura* terá.

Jamie: Acho que deveríamos apenas estimar o esforço... só isso.

Doug: Faremos isso também. Depois usaremos ambas as estimativas como verificação cruzada.

Vinod: Então vamos fazer...

33.6.4 Um exemplo de estimativa baseada em FP

A decomposição para estimativa baseada em FP concentra-se em valores do domínio da informação, em vez de em funções de software. De acordo com a tabela apresentada na Figura 33.3, você estimaria entradas, saídas, consultas, arquivos e interfaces externas para o software CAD. Um valor de FP é calculado usando a técnica discutida no Capítulo 30. Para os propósitos dessa estimativa, assume-se o fator de peso da complexidade como médio. A Figura 33.3 apresenta os resultados dessa estimativa.

Cada um dos fatores de peso da complexidade é estimado; o fator de ajuste do valor é calculado conforme descrito no Capítulo 30:

Fator	Valor
Backup e recuperação	4
Comunicações de dados	2
Processamento distribuído	0
Desempenho crítico	4
Ambiente operacional existente	3
Entrada de dados online	4
Transações de entrada em várias telas	5
Arquivos mestres atualizados online	3
Complexidade dos valores dos domínios de informação	5
Complexidade do processamento interno	5
Código projetado para reutilização	4
Conversão/instalação no projeto	3
Instalações múltiplas	5
Aplicação projetada para alteração	5
Fator de ajuste do valor	1,17

Valor do domínio	Otim.	Mais prov.	Pess.	Est. contagem	Peso	FP contagem
Número de entradas externas	20	24	30	24	4	97
Número de saídas externas	12	15	22	16	5	78
Número de consultas externas	16	22	28	22	5	88
Número de arquivos lógicos internos	4	4	5	4	10	42
Número de arquivos de interface externos	2	2	3	2	7	15
<i>Contagem total</i>						320

FIGURA 33.3 Estimativa de valores do domínio de informações.

Por fim, é obtido o número estimado de FP:

$$FP_{\text{estimado}} = \text{contagem total} \times [0,65 + 0,01 \times \Sigma(F_i)] = 375$$

A produtividade média organizacional para sistemas desse tipo é de 6,5 FP/pm. Com base em um valor bruto de mão de obra de \$ 8 mil por mês, o custo por FP é de aproximadamente \$ 1.230. Com base na estimativa LOC e dados históricos de produtividade, o custo total estimado do projeto é de \$ 461 mil e o esforço estimado é de 58 pessoas-mês.

33.6.5 Estimativa baseada em processo

A técnica mais comum para estimar um projeto é basear a estimativa no processo a ser usado. Isto é, o processo é decomposto em um conjunto relativamente pequeno de atividades, ações e tarefas, e o trabalho necessário para executar cada uma delas é estimado.

Assim como as técnicas baseadas em problemas, a estimativa baseada em processo começa com um delineamento das funções de software obtidas do escopo do projeto. Para cada função, deve ser executada uma série de atividades estruturais. As funções e atividades estruturais relacionadas⁸ podem ser representadas como parte de uma tabela similar à da Figura 33.4.

Estando combinadas as funções do problema e as atividades de processo, você pode estimar o esforço (por exemplo, pessoas-mês) necessário para executar cada atividade do processo de software para cada função do software. Esses dados constituem a matriz central da tabela da Figura 33.4. São então aplicados os valores médios de preço de mão de obra (isto é, custo/unidade de esforço) ao esforço estimado para cada atividade do processo.

Se a estimativa baseada em processo é feita independentemente da estimativa baseada em LOC ou FP, temos agora duas ou três estimativas para custo e esforço que podem ser comparadas e harmonizadas. Se ambos os conjuntos de estimativas apresentarem concordância razoável, há boas razões para acreditar que as estimativas são confiáveis. Por outro lado, se os resultados

Se houver tempo suficiente, use maior detalhamento ao especificar tarefas na Figura 33.4. Por exemplo, subdivida a análise em suas tarefas principais e estime cada uma separadamente.

⁸ As atividades estruturais escolhidas para este projeto diferem um pouco das atividades genéricas discutidas no Capítulo 3. São elas: comunicação com o cliente (CC, customer communication), planejamento, análise de risco, engenharia e construção/liberação das versões.

Atividade →	CC		Análise de risco	Engenharia		Versão da construção		CE	Totais
Tarefa →				Análises	Projeto	Código	Teste		
Função ↓									
UICF				0,50	2,50	0,40	5,00	n/a	8,40
2DGA				0,75	4,00	0,60	2,00	n/a	7,35
3DGA				0,50	4,00	1,00	3,00	n/a	8,50
DBM				0,50	3,00	1,00	1,50	n/a	6,00
PCF				0,50	3,00	0,75	1,50	n/a	5,75
CGDF				0,25	2,00	0,50	1,50	n/a	4,25
DAM				0,50	2,00	0,50	2,00	n/a	5,00
Totais	0,25	0,25	0,25	3,50	20,50	4,50	16,50		46,00
% de mão de obra	1%	1%	1%	8%	45%	10%	36%		

CC = comunicação do cliente CE = avaliação do cliente

FIGURA 33.4 Tabela de estimativa baseada em processo.

dessas técnicas de decomposição mostrarem pouca concordância, deve ser realizada uma investigação e uma análise mais profunda.

33.6.6 Um exemplo de estimativa baseada em processo

Para ilustrar o uso de estimativas baseadas em processo, consideremos novamente o software CAD introduzido na Seção 33.6.3. A configuração do sistema e todas as funções de software permanecem inalteradas e são indicadas pelo escopo do projeto.

De acordo com a tabela baseada em processo mostrada na Figura 33.4, as estimativas de esforço (em pessoas-mês) para cada atividade de engenharia de software são feitas para cada função do software CAD (abreviado para maior simplicidade). As atividades de engenharia e construção subdividem-se nas principais tarefas de engenharia de software da tabela. São feitas estimativas aproximadas de esforço para comunicação com o cliente, planejamento e análise de riscos. Os totais são colocados na parte inferior da tabela. Os totais horizontais e verticais fornecem uma indicação do esforço estimado necessário para análise, projeto, codificação e teste. Devemos observar que 53% de todo o esforço são gastos nas tarefas de engenharia inicial (análise de requisitos e projeto), indicando a importância relativa desse trabalho.

Com base na taxa média bruta de mão de obra de \$ 8 mil por mês, o custo total estimado do projeto é de \$ 368 mil e o esforço estimado é de 46 pessoas-mês. Se você quiser, as taxas de mão de obra podem ser associadas a cada atividade estrutural ou a cada tarefa de engenharia de software e calculadas separadamente.

33.6.7 Estimativa com casos de uso

Como vimos na Parte II deste livro, os casos de uso fornecem informações sobre o escopo do software e os requisitos à equipe de software. Uma vez desenvolvidos os casos de uso, eles podem ser empregados na estimativa do “tamanho” planejado de um projeto de software. Contudo, o desenvolvimento de

“É melhor entender os fundamentos de uma estimativa antes de utilizá-la.”

Barry Boehm e
Richard Fairley

uma estratégia de estimativa com casos de uso apresenta desafios [Smi99]. Os casos de uso são descritos em muitos formatos e estilos diferentes e representam uma visão externa (a visão do usuário) do software. Portanto, eles podem ser redigidos em muitos níveis diferentes de abstração. Os casos de uso não tratam da complexidade das funções e dos recursos descritos e podem descrever comportamento complexo (por exemplo, interações) que envolvem muitas funções e recursos.

Mesmo com essas restrições, é possível calcular os *pontos de caso de uso* (UCPs, *use case points*) de maneira parecida, com o cálculo de pontos de função (Capítulo 30).

Cohn (Coh05) aponta que o cálculo de pontos de caso de uso deve levar em consideração as seguintes características:

- O número e a complexidade dos casos de uso no sistema.
- O número e a complexidade dos atores no sistema.
- Vários requisitos não funcionais (como portabilidade, desempenho, facilidade de manutenção) que não são redigidos como casos de uso.
- O ambiente no qual o projeto vai ser desenvolvido (por exemplo, a linguagem de programação, a motivação da equipe de software).

Para começar, cada caso de uso é avaliado para determinar sua complexidade relativa. Um caso de uso simples indica uma interface de usuário simples, um único banco de dados e três ou menos transações e cinco ou menos implementações de classe. Um caso de uso médio sinaliza uma interface de usuário mais complexa, dois ou três bancos de dados e de quatro a sete transações, com cinco a 10 classes. Por fim, um caso de uso complexo implica uma interface de usuário complexa, com vários bancos de dados, utilizando oito ou mais transações e 11 ou mais classes. Cada caso de uso é avaliado segundo esses critérios, e a contagem de cada tipo é ponderada por um fator de 5, 10 e 15, respectivamente. O *peso do caso de uso não ajustado* (UUCW, *unadjusted use case weight*) total é a soma de todas as contagens ponderadas [Nun11].

Em seguida, cada ator é avaliado. Atores simples são autômatos (outro sistema, uma máquina ou dispositivo) que se comunicam por meio de uma API. Atores médios são autômatos que se comunicam por meio de um protocolo ou depósito de dados e atores complexos são seres humanos que se comunicam por meio de uma interface gráfica de usuário ou outra interface humana. Cada ator é avaliado segundo esses critérios, e a contagem de cada tipo é ponderada por um fator de 1, 2 e 3, respectivamente. O *peso do ator não ajustado* (UAW, *unadjusted actor weight*) total é a soma de todas as contagens ponderadas.

Esses valores não ajustados são modificados pela consideração de fatores de complexidade técnica (TCFs, *technical complexity factors*) e fatores de complexidade ambiental (ECFs, *environment complexity factors*). Treze fatores contribuem para uma avaliação do TCF final e oito contribuem para o cálculo do ECF final [Coh05]. Uma vez determinados esses valores, o valor de UCP final é calculado da seguinte maneira:

$$\text{UCP} = (\text{UUCW} + \text{UAW}) \times \text{TCF} \times \text{ECF} \quad (33.2)$$

Por que é difícil desenvolver uma técnica de estimativa usando casos de uso?

33.6.8 Um exemplo de estimativa usando pontos de caso de uso

O software CAD apresentado na Seção 33.6.3 é composto de três grupos de subsistemas: subsistema da interface do usuário (inclui UICF), grupo do subsistema de engenharia (inclui os subsistemas 2DGA, 3DGA e DAM) e grupo do subsistema de infraestrutura (inclui os subsistemas CGDF e PCF). Dezesesseis casos de uso descrevem o subsistema de interface de usuário. O grupo do subsistema de engenharia é descrito por 14 casos de uso médio e oito simples, e o subsistema de infraestrutura é descrito com 10 casos de uso simples. Portanto,

$$\begin{aligned} \text{UUCW} &= (16 \text{ casos de uso} \times 15) + [(14 \text{ casos de uso} \times 10) \\ &\quad + (8 \text{ casos de uso} \times 5)] + (10 \text{ casos de uso} \times 5) = 470 \end{aligned}$$

A análise dos casos de uso indica que existem oito atores simples, 12 atores médios e quatro atores complexos. Portanto,

$$\text{UAW} = (8 \text{ atores} \times 1) + (12 \text{ atores} \times 2) + (4 \text{ atores} \times 3) = 44$$

Após a avaliação da tecnologia e do ambiente,

$$\text{TCF} = 1,04$$

$$\text{ECF} = 0,96$$

Usando a relação 33.2,

$$\text{UCP} = (470 + 44) \times 1,04 \times 0,96 = 513$$

Usando-se dados de projetos passados como guia, o grupo de desenvolvimento produziu 85 LOC por UCP. Portanto, uma estimativa do tamanho global do projeto de CAD é de 43.600 LOC. Cálculos semelhantes podem ser feitos para o esforço aplicado ou para a duração do projeto.

Usando 620 LOC/pm como produtividade média para sistemas desse tipo e um custo bruto de mão de obra de \$ 8 mil por mês, o custo por linha de código é de aproximadamente \$ 13. Com base na estimativa LOC e dados históricos de produtividade, o custo total estimado do projeto é de \$ 461 mil, e o esforço estimado é aproximadamente 70 pessoas-mês.

33.6.9 Harmonizando estimativas

As técnicas de estimativa discutidas nas seções anteriores resultam em múltiplas estimativas que devem ser harmonizadas para produzirem uma estimativa única de esforço, duração de projeto ou custo. O esforço total estimado para o software CAD (Seção 33.6.3) varia desde um valor baixo de 46 pessoas-mês (derivado de uma estratégia de estimativa baseada em processo) até um valor alto de 68 pessoas-mês (derivado de uma estimativa de caso de uso). A estimativa média (com as quatro estratégias) é de 56 pessoas-mês. A variação em relação à estimativa média é de aproximadamente 18% para baixo e 21% para cima.

O que acontece quando a concordância entre as estimativas é baixa? A resposta a essa pergunta exige uma reavaliação das informações usadas para fazer as estimativas. Estimativas que divergem muito podem ser atribuídas a

"Métodos complicados podem não produzir uma estimativa exata, particularmente quando os desenvolvedores podem incorporar sua própria intuição na estimativa."

**Philip Johnson
e outros**

uma de duas causas: (1) o escopo do projeto não é entendido adequadamente ou foi mal interpretado pelo planejador ou (2) os dados de produtividade usados para as técnicas de estimativa baseada em problema são inadequados para a aplicação, obsoletos (porque não refletem mais com precisão a organização de engenharia de software) ou foram mal aplicados. Você deve determinar a causa da divergência e harmonizar as estimativas.

INFORMAÇÕES



Técnicas automáticas de estimativa para projetos de software

As ferramentas de estimativa automáticas permitem ao planejador estimar o custo e o esforço e executar análises do tipo “e se...” para variáveis importantes do projeto, como data de entrega ou equipe. Embora existam muitas ferramentas de estimativas automáticas, todas apresentam as mesmas características gerais e executam as seis funções genéricas a seguir [Jon96]:

1. *Dimensionamento dos entregáveis do projeto.* O “tamanho” de um ou mais artefatos de software é estimado. Artefatos incluem a representação externa do software (por exemplo, tela, relatórios), o próprio software (por exemplo, KLOC), funcionalidade fornecida (por exemplo, pontos de função) e informações descritivas (por exemplo, documentos).
2. *Seleção de atividades de projeto.* É selecionada a estrutura de processo apropriada e especificado o conjunto de tarefas de engenharia de software.
3. *Previsão do número de pessoas.* É especificado o número de pessoas que estarão disponíveis para o trabalho. Esse é um dado importante porque a relação entre pessoas disponíveis e trabalho (esforço previsto) é altamente não linear.

4. *Previsão do esforço de software.* As ferramentas de estimativa empregam um ou mais modelos (Seção 33.7) que relacionam o tamanho dos entregáveis do projeto com o esforço necessário para produzi-los.
5. *Previsão do custo do software.* Dados os resultados da etapa 4, os custos podem ser calculados alocando-se valores da mão de obra às atividades de projeto citadas na etapa 2.
6. *Previsão dos cronogramas do software.* Ao conhecermos esforço, número de pessoas e atividades de projeto, podemos produzir um esboço de cronograma alocando mão de obra às atividades de engenharia de software com base nos modelos recomendados para distribuição do esforço discutidos mais adiante neste capítulo.

Quando são aplicadas diferentes ferramentas de estimativa para os mesmos dados de projeto, pode ser encontrada uma variação razoavelmente grande nos resultados estimados. Mais importante ainda, os valores previstos muitas vezes são significativamente diferentes dos valores reais. Isso reforça a noção de que o produto das ferramentas de estimativas deve ser usado como “ponto de dados” por meio do qual são obtidas as estimativas – não como a única fonte para uma estimativa.

33.7 Modelos empíricos de estimativa

Um modelo de estimativa para software utiliza fórmulas derivadas empiricamente para prever o esforço como uma função de LOC ou FP.⁹ Valores para LOC ou FP são estimados usando-se a estratégia descrita nas Seções 33.6.3 e 33.6.4. Em vez de utilizarmos as tabelas descritas naquelas seções, os valores resultantes para LOC ou FP são anexados ao modelo de estimativa.

Os dados empíricos que suportam muitos modelos de estimativa são obtidos de uma amostragem limitada de projetos. Por essa razão, nenhum modelo de estimativa é apropriado para todas as classes de software e todos os

⁹ Um modelo empírico que utiliza casos de uso como variável independente é sugerido na Seção 33.6.6. No entanto, apenas alguns apareceram na literatura até agora.

Um modelo de estimativa reflete a população de projetos com base na qual foi obtido. Portanto, o modelo é sensível ao domínio.

ambientes de desenvolvimento. Portanto, você deve usar os resultados obtidos desses modelos com muito critério.

Um modelo de estimativa deve ser calibrado para refletir condições locais. O modelo deve ser testado aplicando-se dados coletados de projetos completos, anexando-se os dados no modelo e comparando-se resultados reais com os previstos. Se a concordância for baixa, o modelo deve ser ajustado e novamente testado antes de ser usado.

33.7.1 A estrutura dos modelos de estimativa

Um modelo típico de estimativa é obtido pela análise de regressão sobre dados coletados de projetos de software anteriores. A estrutura geral desses modelos assume a forma [Mat94]

$$E = A + B \times (e_v)^C \quad (33.3)$$

onde A , B e C são constantes obtidas empiricamente, E é o esforço em pessoas-mês e e_v é a variável de estimativa (LOC ou FP). Além da relação descrita na Equação (33.3), a maioria dos modelos de estimativa tem alguma forma de componente de ajuste de projeto que permite ajustar E por meio de outras características do projeto (por exemplo, complexidade do problema, experiência da equipe, ambiente de desenvolvimento). Um exame rápido de qualquer modelo derivado empiricamente indica que ele deve ser calibrado de acordo com as necessidades locais.

33.7.2 O modelo COCOMO II

Em seu livro clássico sobre economia da engenharia de software, Barry Boehm [Boe81] apresentou uma hierarquia de modelos de estimativa de software com o nome COCOMO, que significa *CO*nstructive *CO*st *MO*del (modelo construtivo de custo). O COCOMO original tornou-se um dos modelos de estimativa de custo de software mais amplamente usado e discutido no setor. Ele evoluiu para um modelo de estimativa mais abrangente, chamado COCOMO II [Boe00]. Assim como seu predecessor, o COCOMO II é, na realidade, uma hierarquia de modelos de estimativa que trata de diferentes “estágios” do processo de software.

Assim como todos os modelos de estimativa para software, os modelos COCOMO II exigem informações de tamanho. Há disponíveis três diferentes opções como parte da hierarquia de modelo: pontos de objeto,¹⁰ pontos de função e linhas de código-fonte.

33.7.3 A equação do software

A *equação do software* [Put92] é um modelo dinâmico multivariável que assume uma distribuição específica de esforço durante a vida de um projeto de desenvolvimento de software. O modelo foi derivado dos dados de produtivi-

¹⁰ Um *ponto de objeto* é uma medida indireta de software calculada por meio de contagens dos números de (1) telas (na interface do usuário), (2) relatórios e (3) componentes que podem ser necessários para construir a aplicação, junto com fatores de complexidade.

dade coletados em mais de 4 mil projetos de software. Com base nesses dados, derivamos um modelo de estimativa da forma

$$E = \frac{\text{LOC} \times B^{0,333}}{P^3} \times \frac{1}{t^4} \quad (33.4)$$

onde

E = esforço em pessoas-mês ou pessoas-ano

t = duração do projeto em meses ou anos

B = “fator de habilidades especiais”¹¹

P = “parâmetro de produtividade” que reflete: maturidade geral do processo e práticas gerenciais, a amplitude na qual são usadas as boas práticas de engenharia de software, o nível de linguagens de programação usado, o estado do ambiente de software, as habilidades e experiência da equipe de software e a complexidade da aplicação

Os valores típicos podem ser $P = 2$ mil para desenvolvimento de software embarcado em tempo real, $P = 10$ mil para software de telecomunicações e sistemas e $P = 28$ mil para aplicações de sistemas comerciais. O parâmetro de produtividade pode ser derivado para condições locais por meio de dados históricos coletados de trabalhos de desenvolvimento executados no passado.

Você deve observar que a equação do software tem dois parâmetros independentes: (1) uma estimativa do tamanho (em LOC) e (2) uma indicação da duração do projeto em meses corridos ou anos.

Para simplificarem o processo de estimativa e usarem uma forma mais comum para o modelo de estimativa, Putnam e Myers [Put92] sugerem uma série de equações derivadas da equação do software. O tempo de desenvolvimento mínimo é definido como

$$t_{\min} = 8,14 \frac{\text{LOC}}{P^{0,43}} \text{ em meses para } t_{\min} > 6 \text{ meses} \quad (33.5a)$$

$$E = 180 B t^3 \text{ em pessoas-mês para } E \geq 20 \text{ pessoas-mês} \quad (33.5b)$$

Note que t na Equação (33.5b) está representado em anos.

Usando a Equação (33.5) com $P = 12.000$ (o valor recomendado para software científico) para o software CAD discutido anteriormente neste capítulo,

$$t_{\min} = 8,14 \times \frac{33.200}{12.000^{0,43}} = 12,6 \text{ meses}$$

$$E = 180 \times 0,28 \times (1,05)^3 = 58 \text{ pessoas-mês}$$

Os resultados da equação do software correspondem favoravelmente à estimativa desenvolvida na Seção 33.6. Assim como o modelo COCOMO discutido na Seção 33.7.2, a equação do software continua evoluindo. Uma discussão mais detalhada dessa estratégia de estimativa pode ser encontrada em [Put97b].

¹¹ B aumenta lentamente à medida que cresce a “necessidade de integração, teste, garantia de qualidade, documentação e habilidades de gerenciamento” [Put92]. Para programas pequenos (KLOC = 5 a 15), $B = 0,16$. Para programas maiores do que 70 KLOC, $B = 0,39$.

33.8 Estimativa para projetos orientados a objetos

Cabe complementar os métodos convencionais de estimativa de custo de software com uma técnica criada explicitamente para software orientado a objetos. Lorenz e Kidd [Lor94] sugerem a seguinte estratégia:

1. Desenvolva estimativas usando decomposição de esforço, análise FP e qualquer outro método para aplicações convencionais.
2. Usando o modelo de requisitos (Capítulo 10), desenvolva casos de uso e determine uma contagem. Reconheça que o número de casos de uso pode mudar à medida que o projeto avança.
3. Com base no modelo de requisitos, determine o número de classes-chave (chamadas de classes de análise no Capítulo 10).
4. Classifique o tipo de interface para a aplicação e crie um multiplicador para suportar classes, onde os multiplicadores para nenhuma interface gráfica de usuário, uma interface baseada em texto, uma interface gráfica de usuário convencional e uma interface gráfica de usuário complexa são, respectivamente: 2,0; 2,25; 2,5; e 3,0. Multiplique o número de classes-chave (passo 3) pelo multiplicador para obter uma estimativa do número de classes de apoio.
5. Multiplique o número total de classes (classes-chave + classes de apoio) pelo número médio de unidades de trabalho por classe. Lorenz e Kidd sugerem 15 a 20 pessoas-dia por classe.
6. Faça uma verificação cruzada da estimativa baseada em classes, multiplicando o número médio de unidades de trabalho por caso de uso.

33.9 Técnicas de estimativa especializadas

As técnicas de estimativa discutidas nas Seções 33.6 a 33.8 podem ser empregadas em qualquer projeto de software. No entanto, quando uma equipe de software encontra um prazo de projeto extremamente curto (semanas, em vez de meses), no qual é provável que haja uma sequência contínua de alterações, o planejamento do projeto em geral e a estimativa em particular devem ser abreviados.¹² Nas próximas seções, examinaremos duas técnicas especializadas de estimativa.

33.9.1 Estimativa para desenvolvimento ágil

Como os requisitos para um projeto ágil (Capítulo 5) são definidos por um conjunto de cenários de usuário (por exemplo, “histórias” em Programação Extrema), é possível desenvolver uma estratégia de estimativa informal, razoavelmente disciplinada e significativa no contexto do planejamento de projeto

¹² “Abreviado” não significa eliminado. Mesmo os projetos de curta duração devem ser planejados, e a estimativa é a base do planejamento confiável.

para cada incremento de software. A estimativa para projetos ágeis usa uma estratégia de decomposição que abrange os seguintes passos:

1. Cada cenário de usuário (o equivalente a um mini caso de uso criado bem no início de um projeto por usuários ou outros envolvidos) é considerado separadamente para fins de estimativa.
2. O cenário é decomposto em uma série de tarefas de engenharia de software que serão necessárias para desenvolvê-lo.
- 3a. Cada tarefa é estimada separadamente. Observação: a estimativa pode ser baseada em dados históricos, em um modelo empírico ou na “experiência”.
- 3b. Como alternativa, o “volume” do cenário pode ser estimado em LOC, FP ou alguma outra medida orientada por volumes (por exemplo, contagem de caso de uso).
- 4a. As estimativas de cada tarefa são somadas para criar uma estimativa para o cenário.
- 4b. Como alternativa, o volume estimado para o cenário é traduzido em esforço usando dados históricos.
5. As estimativas de esforço para todos os cenários implementados para determinado incremento de software são somadas para desenvolver a estimativa de esforço para o incremento.

Como as estimativas são desenvolvidas quando é aplicado um processo ágil?

No contexto da estimativa para projetos ágeis, “volume” é uma estimativa do tamanho total de um cenário de usuário em LOC ou FP.

Como a duração do projeto exigida para o desenvolvimento de um incremento de software é muito curta (normalmente, de três a seis semanas), essa estratégia de estimativa tem dois propósitos: (1) garantir que o número de cenários a incluir no incremento esteja de acordo com os recursos disponíveis e (2) estabelecer uma base para a alocação de esforço à medida que o incremento é desenvolvido.

33.9.2 Estimativa para projetos de WebApps

Os projetos de WebApps com frequência adotam o modelo de processo ágil. Uma medida modificada de pontos de função, associada às etapas resumidas na Seção 33.9.1, pode ser usada para desenvolver uma estimativa para a WebApp. Roetzheim [Roe00] sugere a seguinte estratégia ao adaptar pontos de função para estimativa de WebApp:

- *Entradas (Inputs)*. Cada tela de entrada ou formulário [form] (por exemplo, CGI ou Java), cada tela de manutenção e, se você usa uma metáfora de conjunto de guias em algum lugar [tab notebook], cada guia [tabl].
- *Saídas (Outputs)*. Cada página Web estática, cada script de página Web dinâmica (por exemplo, ASP, ISAPI ou outro script DHTML) e cada relatório (baseado na Web ou de natureza administrativa).
- *Tabelas (Tables)*. Cada tabela lógica no banco de dados e, se você estiver usando XML para armazenar dados em um arquivo, cada objeto XML (ou coleção de atributos XML).
- *Interfaces (Interfaces)*. Retêm sua definição como arquivos lógicos (por exemplo, formatos únicos de registro) em nossas fronteiras de saída do sistema.

- *Consultas (Queries)*. Publicadas externamente ou usam uma interface orientada por mensagens. Um exemplo típico são as referências externas DCOM ou COM.

Pontos de função (interpretados da maneira descrita) são bons indicadores do volume para uma WebApp.

FERRAMENTAS DO SOFTWARE



Estimativa de mão de obra e custo

Objetivo: o objetivo das ferramentas de estimativa de esforço e custo é fornecer a uma equipe de projeto as estimativas de esforço necessário, duração do projeto e custo, de uma maneira que trate das características específicas do projeto e do ambiente no qual ele deve ser criado.

Mecanismos: em geral, as ferramentas de estimativa de custo usam uma base de dados histórica derivada de projetos e dados locais coletados na prática e um modelo empírico (por exemplo, COCOMO II) utilizado para derivar as estimativas de esforço, duração e custo. As características do projeto e o ambiente de desenvolvimento constituem a entrada, e a ferramenta proporciona como saída uma variedade de estimativas.

Ferramentas representativas:¹³

Costar, desenvolvida pela Softstar Systems (www.softstar-systems.com), usa o modelo COCOMO II para desenvolver estimativas de software.

Cost Xpert, desenvolvida pelo Cost Xpert Group (www.costxpert.com), integra vários modelos de estimativa e uma base de dados históricos de projeto.

Construx Professional, desenvolvida pela Construx (http://www.construx.com/Resources/Construx_Estimate/), tem como base Putnam Model e COCOMO II.

Knowledge Plan, desenvolvida pela Software Productivity Research (www.spr.com), usa entrada de pontos de função como guia principal para um pacote completo de estimativas.

Price S, desenvolvida pela Price Systems (www.pricesystems.com), é uma das ferramentas de estimativa mais antigas e amplamente utilizadas para projetos de desenvolvimento de software de larga escala.

SEER/SEM, desenvolvida pela Galorath (www.galorath.com), fornece recursos de estimativa abrangentes, análise de sensibilidade, avaliação de risco e outras características.

SLIM-Estimate, desenvolvida pela QSM (www.qsm.com), faz uso de "conhecimento industrial" abrangente para proporcionar uma "verificação de plausibilidade" para estimativas derivadas usando dados locais.

33.10 A decisão fazer/comprar

Em muitas áreas de aplicação de software, com frequência é mais viável, em termos de custo, comprar do que desenvolver o software de computador. Os gerentes de engenharia de software precisam tomar a decisão fazer/comprar, que pode ser ainda mais complicada por uma série de opções de aquisição: (1) o software pode ser comprado (ou licenciado) pronto para uso, (2) componentes de software "totalmente testados" ou "parcialmente testados" (consulte a Seção 33.4.2) podem ser adquiridos e depois modificados e integrados para atender às necessidades específicas, ou (3) o software pode ser criado de forma personalizada por um terceiro contratado que atenda às especificações do comprador.

¹³ A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

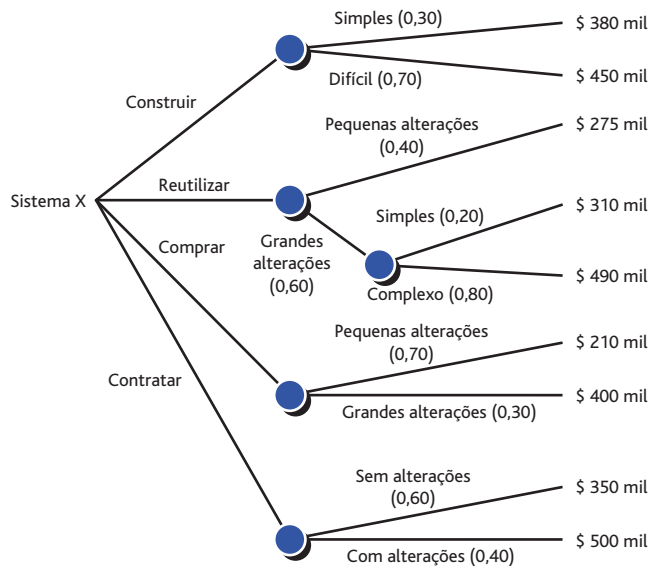


FIGURA 33.5 Uma árvore de decisões para apoiar a decisão fazer/comprar.

As etapas da aquisição de software são definidas pelo aspecto crítico do software a ser comprado e o custo final. Em alguns casos (por exemplo, aplicações de baixo custo), é menos dispendioso comprar e experimentar do que fazer uma demorada avaliação dos pacotes de software em potencial. Na análise final, a decisão fazer/comprar é tomada com base nas seguintes condições: (1) A data de entrega do software será mais adiantada comparada com o software desenvolvido internamente? (2) O custo da aquisição mais o custo da personalização serão inferiores ao custo para desenvolver o software internamente? (3) O custo do suporte externo (por exemplo, um contrato de manutenção) será menor do que o custo do suporte interno? Essas condições se aplicam a cada uma das opções de aquisição.

33.10.1 Criação de uma árvore de decisões

As etapas que acabamos de descrever podem ser ampliadas por meio de técnicas estatísticas, como a árvore de análise de decisões.¹⁴ Por exemplo, a Figura 33.5 mostra uma árvore de decisões para um sistema X baseado em software. Nesse caso, a organização de engenharia de software pode (1) criar o sistema X a partir do zero, (2) reutilizar componentes existentes, parcialmente testados, para criar o sistema, (3) comprar um software disponível e modificá-lo para atender às necessidades locais, ou (4) contratar o desenvolvimento de software externamente.

Se o sistema tem de ser criado desde o início, há uma probabilidade de 70% de que a tarefa será difícil. Usando as técnicas de estimativa discutidas neste capítulo, o planejador do projeto estima que um trabalho de desenvolvimento difícil custará \$ 450 mil. Um trabalho de desenvolvimento “simples” tem

Existe uma maneira sistemática de escolher entre as opções associadas à decisão fazer/comprar?

¹⁴ Uma boa introdução à árvore de análise de decisões pode ser encontrada no endereço http://en.wikipedia.org/wiki/Decision_tree.

um custo estimado de \$ 380 mil. O valor esperado do custo, calculado ao longo de qualquer ramo da árvore de decisões, é

Custo esperado = $\Sigma(\text{probabilidade do caminho})_i \times (\text{custo estimado do caminho})_i$

onde i é o caminho da árvore de decisões. Para o caminho da criação,

$$\text{Custo esperado}_{\text{construção}} = 0,30 (\$ 380\text{K}) + 0,70 (\$ 450\text{K}) = \$ 429\text{K}$$

Seguindo outros caminhos da árvore de decisões, são mostrados os custos projetados para reutilização, compra e contratação sob várias circunstâncias. Os custos esperados para esses caminhos são

$$\text{Custo esperado}_{\text{reutilizar}} = 0,40 (\$ 275\text{K}) + 0,60 [0,20 (\$ 310\text{K}) + 0,80 (\$ 490\text{K})] = \$ 382\text{K}$$

$$\text{Custo esperado}_{\text{comprar}} = 0,70 (\$ 210\text{K}) + 0,30 (\$ 400\text{K}) = \$ 267\text{K}$$

$$\text{Custo esperado}_{\text{contratar}} = 0,60 (\$ 350\text{K}) + 0,40 (\$ 500\text{K}) = \$ 410\text{K}$$

Com base na probabilidade e nos custos projetados da Figura 33.5, o custo esperado mais baixo é o da opção “comprar”.

No entanto, é importante observar que muitos outros critérios – não apenas o custo – devem ser considerados durante o processo de tomada de decisões. A disponibilidade, a experiência do desenvolvedor/fornecedor/contratado, a conformidade com os requisitos, as “políticas” locais e a possibilidade de alterações são apenas alguns dos critérios que podem afetar a decisão final de construir, reutilizar, comprar ou contratar.

33.10.2 Terceirização

Cedo ou tarde, toda empresa que desenvolve software faz a pergunta fundamental: “Há alguma maneira de obtermos o software e os sistemas de que precisamos a um preço mais baixo?”. A resposta não é nada simples, e as discussões emocionais que ocorrem sempre levam a uma única palavra: *terceirização*.

Conceitualmente, a terceirização é muito simples. As atividades de engenharia de software são contratadas de um terceiro que faz o trabalho a um custo menor e, espera-se, com melhor qualidade. O trabalho de software realizado em uma empresa é reduzido a uma atividade de gerenciamento de contrato.¹⁵

A decisão de optar pela terceirização pode ser estratégica ou tática. No nível estratégico, os gerentes de negócio consideram se uma parte significativa de todo o trabalho de software pode ser contratada com outros. No nível tático, um gerente de projeto determina se parte ou todo o projeto pode ser mais bem executado subcontratando o trabalho de software.

Independentemente da amplitude do enfoque, a decisão de terceirizar muitas vezes é financeira. Uma discussão detalhada da análise financeira para a terceirização está fora dos objetivos deste livro; aconselhamos consultar ou-

“Como regra, a terceirização exige uma habilidade de gerenciamento ainda maior do que o desenvolvimento interno.”

Steve McConnell

¹⁵ A terceirização pode ser vista, em geral, como qualquer atividade que leva à aquisição de software ou componentes de software de uma fonte fora da organização de engenharia de software.

tros autores (por exemplo, [Min95]). No entanto, apresentaremos aqui um rápido exame dos prós e contras dessa decisão.

O lado bom é que pode-se reduzir custos diminuindo-se o número de pessoas empregadas e instalações (por exemplo, computadores, infraestrutura) que os suportam. O lado ruim é que a empresa perde um pouco do controle sobre o software de que precisa. Como software é uma tecnologia que diferencia seus sistemas, serviços e produtos, uma empresa corre o risco de colocar o destino de sua competitividade nas mãos de um terceiro.

A tendência para a terceirização certamente continuará. A única maneira de atenuá-la é reconhecer que o trabalho de software é extremamente competitivo em todos os níveis. A única maneira de sobreviver é se tornar tão competitivo quanto os próprios fornecedores de terceirização.



Terceirização

Cena: Sala de reuniões da CPI Corporation no início do projeto.

Atores: Mal Golden, gerente sênior do setor de desenvolvimento de produto; Lee Warren, gerente de engenharia; Joe Camalleri, vice-presidente executivo do setor de desenvolvimento de negócios; e Doug Miller, gerente de projeto de engenharia de software.

Conversa:

Joe: Estamos considerando terceirizar a parte de engenharia de software do CasaSegura.

Doug (chocado): Quando isso ocorreu?

Lee: Recebemos a cotação de um desenvolvedor do exterior. Ela está 30% abaixo daquilo que o seu grupo acha que custará. Veja.

[Passa a cotação para Doug, que a lê.]

Mal: Como você sabe, Doug, estamos tentando baixar os custos, e 30% são 30%. Além disso, esse pessoal é bastante recomendado.

Doug (tomando fôlego e tentando manter-se calmo): Bem, vocês me pegaram de surpresa. Antes de tomarmos a decisão final, gostaria de fazer alguns comentários.

Joe (concordando): Claro, prossiga.

Doug: Nunca trabalhamos com essa empresa, certo?

Mal: Certo, mas...

Doug: E eles afirmam que alterações nas especificações serão cobradas a uma taxa adicional, certo?

Joe (contrariado): Certo, mas esperamos que as coisas se mantenham razoavelmente estáveis.

Doug: Uma péssima suposição, Joe.

Joe: Bem...

Doug: É provável que lançaremos novas versões desse produto nos próximos anos. E é razoável supor que o software trará muitas das novas características, certo?

[Todos acenam afirmativamente.]

Doug: Nós já coordenamos um projeto internacional antes?

Lee (parecendo preocupado): Não, mas me disseram...

Doug (tentando conter sua raiva): Então o que você está me dizendo é que: (1) vamos trabalhar com um fornecedor desconhecido, (2) os custos para fazer isso não são tão baixos quanto parecem, (3) estamos concordando em trabalhar com eles durante muitas versões do produto, não importa o que fizerem na primeira versão, e (4) vamos aprender sobre um projeto internacional "com o bonde andando".

[Todos permanecem em silêncio.]

Doug: Pessoal... eu acho isso um erro. Gostaria que vocês reconsiderassem por mais um dia. Teremos muito mais controle se fizermos o trabalho internamente. Temos a experiência, e posso garantir que não vai nos custar muito mais... o risco será menor, e sei que todos vocês têm aversão a riscos, como eu também tenho.

Joe (contrariado): Você apresentou boas razões, mas tem um interesse claro em manter esse projeto internamente.

Doug: Isso é verdade, mas isso não muda os fatos.

Joe (com um suspiro): Certo, vamos adiar por um ou dois dias, pensar um pouco e fazer uma reunião novamente para uma decisão final. Doug, posso falar em particular com você?

Doug: Sim... eu realmente quero ter certeza de que estamos fazendo a coisa certa.

CASASEGURA

33.11 Resumo

Um planejador de projeto de software deve estimar três itens antes de começar: quanto tempo levará, quanto esforço será necessário e quantas pessoas serão envolvidas. Além disso, o planejador deve prever os recursos (hardware e software) que serão necessários e o risco envolvido.

A definição de escopo ajuda o planejador a desenvolver estimativas usando uma ou mais técnicas que se dividem em duas grandes categorias: decomposição e modelagem empírica. As técnicas de decomposição exigem um delineamento das principais funções do software, seguido de estimativas de (1) número de LOC, (2) valores selecionados dentro do domínio de informações, (3) número de casos de uso, (4) número de pessoas-mês necessário para implementar cada função ou (5) número de pessoas-mês necessário para cada atividade de engenharia de software. Técnicas empíricas usam expressões derivadas empiricamente para esforço e tempo para prever esses valores de projeto. Podem ser utilizadas ferramentas automáticas para implementar um modelo empírico específico.

Estimativas precisas de projeto em geral usam pelo menos duas das três técnicas que acabamos de descrever. Comparando e harmonizando as estimativas desenvolvidas empregando diferentes técnicas, o planejador tem maior possibilidade de derivar uma estimativa precisa. A estimativa de projeto de software nunca será uma ciência exata, mas uma combinação de bons dados históricos e técnicas sistemáticas pode melhorar a precisão da estimativa.

Problemas e pontos a ponderar

33.1 Suponha que você seja o gerente de projeto de uma empresa que cria software para robôs de uso doméstico. Você foi contratado para criar o software para um robô que corta a grama de uma residência. Apresente uma definição de escopo que descreva o software. Certifique-se de que a sua definição de escopo esteja delimitada. Se não estiver familiarizado com robôs, faça algumas pesquisas antes de começar a escrever. Além disso, formule suas hipóteses sobre o hardware necessário. Alternativa: troque o robô cortador de grama por outro problema de seu interesse.

33.2 A complexidade do projeto de software é discutida rapidamente na Seção 33.1. Desenvolva uma lista de características de software (por exemplo, operação concorrente, saída gráfica) que afetam a complexidade de um projeto. Defina prioridades na lista.

33.3 O desempenho é uma consideração importante durante o planejamento. Discuta como pode ser interpretado diferentemente, dependendo da área de aplicação do software.

33.4 Faça uma decomposição funcional do software do robô que você descreveu no Problema 33.1. Estime o tamanho de cada função em LOC. Supondo que a sua organização produza 450 LOC/pm com um valor bruto de mão de obra de \$ 7 mil por pessoa-mês, estime a mão de obra e custo necessário para criar o software usando a técnica de estimativa baseada em LOC descrita neste capítulo.

33.5 Use a equação do software para estimar o software do robô cortador de grama. Suponha que a Equação (33.4) seja aplicável e que $P = 8.000$.

33.6 Desenvolva um modelo de planilha que implemente uma ou mais das técnicas de estimativa descritas neste capítulo. Como alternativa, use um ou mais modelos online para estimativa de fontes da Web.

33.7 Para uma equipe de projeto: desenvolvam uma ferramenta de software que implemente cada uma das técnicas de estimativa desenvolvidas neste capítulo.

33.8 Parece estranho que as estimativas de custo e de cronograma sejam feitas durante o planejamento do projeto de software – antes de fazer a análise detalhada dos requisitos de software ou projeto. Por que você acha que isso é feito? Há circunstâncias nas quais não deveria ser feito?

Leituras e fontes de informação complementares

Muitos livros de gerenciamento de projeto de software contêm discussões de estimativas de projeto. O Project Management Institute (*PMBOK Guide*, PMI, 2001), Wysoki (*Effective Project Management: Traditional, Agile, Extreme*, 6ª ed., Wiley, 2011), Lewis (*Project Planning Scheduling and Control*, 5ª ed., McGraw-Hill, 2010), Kerzner (*Project Management: A Systems Approach to Planning, Scheduling, and Controlling*, 10ª ed., Wiley, 2009), Bennatan (*On Time, Within Budget: Software Project Management Practices and Techniques*, 3ª ed., Wiley, 2000) e Phillips [Phi98] relacionam informações úteis sobre estimativas

McConnell (*Software Estimation: Demystifying the Black Art*, Microsoft Press, 2006) escreveu um guia prático que fornece amplas diretrizes para qualquer um que precise estimar custo de software. Parthasarathy (*Practical Software Estimation*, Addison-Wesley, 2007) destaca pontos de função como uma métrica de estimativa. Hill (*Practical Software Project Estimation*, McGraw-Hill Osborne Media, 2010) e Laird e Brennan (*Software Measurement and Estimation: A Practical Approach*, Wiley-IEEE Computer Society Press, 2006) tratam de medida e seu uso na estimativa de software. Pfleeger (*Software Cost Estimation and Sizing Methods, Issues, and Guidelines*, RAND Corporation, 2005) desenvolveu um guia resumido que trata de muitos fundamentos de estimativas. Jones (*Estimating Software Costs*, 2ª ed., McGraw-Hill, 2007) escreveu um dos mais abrangentes tratamentos de modelos e dados aplicáveis às estimativas de software em todos os domínios de aplicação. Coombs (*IT Project Estimation*, Cambridge University Press, 2002) e Roetzheim e Beasley (*Software Project Cost and Schedule Estimating: Best Practices*, Prentice Hall, 1997) apresentam muitos modelos úteis e sugerem diretrizes passo a passo para gerar as melhores estimativas possíveis.

Há disponível na Internet uma ampla variedade de fontes de informação sobre estimativa de software. Uma lista atualizada das referências da Web pode ser encontrada em “recursos de engenharia de software” no site da SEPA: www.mhhe.com/pressman.

34 Cronograma de projeto

Conceitos-chave

acompanhamento	767
caixa de tempo	768
caminho crítico	757
cronogramas	
princípios dos	758
projetos de WebApps e aplicativos móveis ..	769
distribuição de esforço ..	760
gráfico de Gantt	766
pessoas e esforço	759
rede de tarefas	764
subdivisão do trabalho ..	765
valor agregado	772

No final da década de 1960, um jovem e entusiasmado engenheiro foi selecionado para “escrever” um programa para uma aplicação de manufatura automatizada. O motivo de ter sido escolhido era simples: ele era a única pessoa em seu grupo que havia participado de um seminário sobre programação de computadores. Conhecia em detalhes a linguagem Assembly e a FORTRAN, mas não sabia nada sobre engenharia de software e menos ainda sobre cronograma e acompanhamento de projeto.

Seu chefe lhe entregou os manuais certos e uma descrição do que precisava ser feito. Ele foi informado de que o projeto tinha de ficar pronto em dois meses.

Ele leu os manuais, escolheu sua abordagem e começou a escrever o código. Duas semanas depois, o chefe o chamou ao escritório e perguntou como as coisas estavam andando.

Estão “excelentes”, disse o jovem engenheiro com entusiasmo juvenil. “Era mais simples do que eu pensava. Já estou com quase 75% do trabalho pronto.”

PANORAMA

ria de software que precisam ser feitas, estimou a mão de obra e o número de pessoas, conhece os prazos e até já considerou os riscos. Agora chegou a hora de ligar os pontos, ou seja, de criar uma rede de tarefas de engenharia de software que permita terminar o trabalho no prazo. Uma vez criada a rede, é necessário designar um responsável para cada tarefa, verificar se ela é feita e adaptá-la à medida que os riscos se tornam realidade. Resumindo, isso se chama cronograma e acompanhamento de projeto de software.

Quem realiza? Em nível de projeto, os gerentes de projeto de software, usando informações solicitadas aos engenheiros de software. Em nível individual, os próprios engenheiros de software.

Por que é importante? Para criar um sistema complexo, muitas tarefas de engenharia de software ocorrem em paralelo, e o resultado do trabalho executado durante uma tarefa pode ter um profundo efeito sobre o trabalho a ser executado em outra tarefa. Essas interdependências são

O que é? Você selecionou um modelo de processo apropriado, identificou as tarefas de engenharia

muito difíceis de entender sem um cronograma. É também praticamente impossível avaliar o progresso em um projeto de software de tamanho médio ou grande sem um cronograma detalhado.

Quais são as etapas envolvidas? As tarefas de engenharia de software ditadas pelo modelo de processo de software são refinadas para a funcionalidade a ser criada. Duração e esforço são alocados para cada tarefa e é criada uma rede de tarefas (também chamada de “rede de atividades”) para que a equipe de software cumpra os prazos de entrega estabelecidos.

Qual é o artefato? O cronograma de projeto e as informações relacionadas são produzidas.

Como garantir que o trabalho foi realizado corretamente?

Um cronograma apropriado exige que: (1) todos os riscos apareçam na rede, (2) esforço e duração sejam alocados de modo inteligente para cada tarefa, (3) as interdependências entre as tarefas sejam indicadas corretamente, (4) sejam alocados recursos para o trabalho a ser feito e (5) sejam alocados pontos de verificação bem próximos uns dos outros para que o progresso possa ser acompanhado.

O chefe sorriu e incentivou o jovem engenheiro a continuar seu ótimo trabalho. Marcaram outra reunião para uma semana depois.

Uma semana depois, o chefe chamou o engenheiro ao escritório e perguntou: “Como estamos?”

“Tudo está indo muito bem”, disse o jovem, “mas estou enfrentando alguns obstáculos. Vou resolver tudo isso e voltar logo à rotina normal.”

“Como está o prazo de entrega?”, perguntou o chefe. “Sem problemas”, restou o engenheiro. “Estou com quase 90% do trabalho pronto.”

Se você trabalha no mundo do software há alguns anos, pode imaginar como acaba essa história. Não se surpreenderá em saber que o jovem engenheiro¹ ficou nos 90% do trabalho pronto durante toda a duração do projeto e terminou (com a ajuda de outros) com um mês de atraso.

Essa história tem se repetido dezenas de milhares de vezes durante as últimas cinco décadas. A grande pergunta é: por quê?

34.1 Conceitos básicos

Embora as razões para atrasos na entrega de software sejam inúmeras, muitas delas podem ser atribuídas a uma ou mais das seguintes causas básicas:

- Um prazo de entrega não realista estabelecido por alguém de fora da equipe de software e imposto sobre gerentes e profissionais do grupo.
- Alterações nos requisitos do cliente não refletidas em alterações no cronograma.
- Uma subestimação honesta do esforço e/ou quantidade de recursos que serão necessários para executar o serviço.
- Riscos previsíveis e/ou não previsíveis não considerados quando o projeto foi iniciado.
- Dificuldades técnicas que não puderam ser previstas com antecedência.
- Dificuldades humanas que não puderam ser previstas com antecedência.
- Falha de comunicação entre o pessoal de projeto que resulta em atrasos.
- Falha do gerente de projeto em não perceber o atraso do cronograma do projeto e, desse modo, não executar nenhuma ação para corrigir o problema.

“Cronogramas apertados ou irracionais provavelmente são a influência mais destrutiva em todo o software.”

Capers Jones

Prazos de entrega agressivos (leia-se “não realistas”) são fatos comuns nos negócios de software. Às vezes eles são impostos por motivos legítimos, do ponto de vista daquele que define esses prazos. Mas o bom senso diz que a legitimidade deve ser entendida também pelas pessoas que estão executando o trabalho.

Os métodos de estimativa discutidos no Capítulo 33 e as técnicas de cronograma discutidas neste capítulo são muitas vezes implementados sob a pressão de um prazo de entrega definido. Se as melhores estimativas indicam que o prazo de entrega não é realista, um gerente de projeto competente deve

¹ Caso esteja se perguntando, sim, essa história é autobiográfica (RSP).

"Adoro prazos de entrega. Gosto do som sibilante que fazem quando passam voando."

Douglas Adams

O que devemos fazer quando a gerência nos exige um prazo de entrega impossível?

"proteger sua equipe contra pressões [cronograma] indevidas... [e] enviar a pressão de volta para aqueles que a originaram" [Pag85].

Para ilustrar, suponha que a sua equipe de software tenha sido encarregada de desenvolver um controlador em tempo real para um instrumento de diagnóstico médico que deve ser lançado no mercado em 9 meses. Após cuidadosa estimativa e análise de riscos (Capítulo 35), você chega à conclusão de que o software, da maneira como foi solicitado, levará 14 meses corridos para ser criado com o pessoal disponível. Como deve proceder?

Não faz sentido ir ao escritório do cliente (nesse caso, o provável cliente é o departamento de marketing/vendas) e pedir que a data de entrega seja alterada. As pressões do marketing externo ditaram a data, e o produto tem de ser entregue. É igualmente uma tolice recusar o trabalho (do ponto de vista profissional). Então, o que fazer? Recomendamos tomar as seguintes providências nessa situação:

1. Faça uma estimativa detalhada usando dados históricos de projetos anteriores. Determine o esforço estimado e a duração do projeto.
2. Usando um modelo incremental de processo (Capítulo 4), desenvolva uma estratégia de engenharia de software que forneça a funcionalidade crítica no prazo de entrega imposto, mas deixe outras funcionalidades para depois. Documente o plano.
3. Reúna-se com o cliente e (usando a estimativa detalhada) explique por que o prazo de entrega imposto não é praticável. Não deixe de destacar que todas as estimativas são baseadas no desempenho de projetos anteriores. Não deixe também de indicar a porcentagem de melhoria necessária para cumprir o prazo de entrega da forma como ele está definido.² Cabe fazer o seguinte comentário:

Acho que temos um problema com o prazo de entrega do software controlador XYZ. Já encaminhei a vocês um resumo das taxas de produtividade de desenvolvimento de projetos realizados e uma estimativa que já fizemos de muitas maneiras diferentes. Vocês devem observar que considere uma melhoria de 20% nas taxas de produtividade anteriores, mas ainda temos um prazo de entrega de 14 meses, em vez de 9 meses.

4. Ofereça a estratégia de desenvolvimento incremental como uma alternativa:

Tenho algumas opções a oferecer e gostaria que vocês tomassem a decisão com base nelas. Primeiro, podemos aumentar o orçamento e buscar recursos extras para ficarmos seguros de completar o trabalho em 9 meses. Mas entendo que isso aumentará o risco de má qualidade devido ao prazo apertado.³ Segundo, podemos remover uma série de funções e recursos do software que vocês estão solicitando. Isso tornará a versão preliminar do produto um pouco menos

² Se a melhoria exigida for de 10 a 25%, pode ser possível fazer o trabalho no prazo. Mas é mais provável que a melhoria de desempenho da equipe precisará ser maior do que 50%. Essa é uma expectativa não realista.

³ Você pode argumentar também que o aumento do número de pessoas não reduz proporcionalmente o prazo.

funcional, mas podemos anunciar uma funcionalidade completa e fornecê-la após 14 meses. Terceiro, podemos ignorar toda a realidade e esperar completar o projeto em 9 meses. Vamos acabar não tendo nada a entregar ao cliente. A terceira opção, espero que concordem comigo, é inaceitável. Dados anteriores e nossas melhores estimativas dizem que isso não é realista: é uma receita para o desastre.

Haverá protestos, mas, se forem apresentadas estimativas sólidas, baseadas em bons dados históricos, é provável que sejam escolhidas versões negociadas das opções 1 ou 2. O prazo de entrega não realista é eliminado.

34.2 Cronograma de projeto

Perguntaram certa vez a Fred Brooks como é que os projetos de software acabam atrasando. Sua resposta foi tão simples quanto profunda: “Um dia de cada vez”.

A realidade de um projeto técnico (pode envolver a construção de uma usina hidrelétrica ou o desenvolvimento de um sistema operacional) é que centenas de pequenas tarefas devem ocorrer para se atingir o objetivo maior. Algumas dessas tarefas estão fora da rotina principal e podem ser completadas sem muita preocupação quanto ao impacto na data de entrega do projeto. Outras tarefas estão no *caminho crítico*. Se essas tarefas “críticas” atrasarem, o prazo de entrega do projeto inteiro será ameaçado.

Como gerente de projetos, seu objetivo é definir todas as tarefas, criar uma rede que mostre suas interdependências, identificar as tarefas críticas dentro da rede e acompanhar o progresso para garantir que os atrasos sejam detectados “um dia de cada vez”. Para tanto, deve-se ter um cronograma definido com um grau de resolução que permita monitorar o progresso e controlar o projeto.

Cronograma de projeto de software é uma ação que distribui o esforço estimado por toda a duração planejada do projeto, alocando esse esforço para tarefas específicas de engenharia de software. No entanto, é importante notar que o cronograma evolui com o tempo. Durante os primeiros estágios do planejamento do projeto, desenvolve-se um cronograma macroscópico. Este identifica as principais atividades do processo e as funções do produto para as quais se aplicam. Conforme o projeto caminha, cada item é refinado em um cronograma detalhado. Nesse momento, ações e tarefas de software específicas (necessárias para realizar uma atividade) são identificadas e dispostas em um cronograma.

O cronograma para projetos de engenharia de software pode ser visto sob duas perspectivas bem diferentes. Na primeira, uma data final para entrega de um sistema computacional já foi definida (de maneira irreversível). A organização de software é forçada a distribuir o esforço no prazo prescrito. A segunda perspectiva considera que os limites cronológicos aproximados foram discutidos, mas a data final é definida pela organização de engenharia de software. O esforço é distribuído para fazer o melhor uso dos recursos, e uma data final é refinada após cuidadosa análise do soft-

As tarefas necessárias para que o gerente de projetos atinja seus objetivos não devem ser executadas manualmente. Há muitas ferramentas excelentes para cronogramas. Utilize-as.

“A sobreposição de cronogramas otimistas não resulta em cronogramas reais mais breves, e sim mais longos.”

Steve McConnell

ware. Infelizmente, a primeira situação ocorre com muito mais frequência do que a segunda.

34.2.1 Princípios básicos

Ao desenvolver um cronograma, divida o trabalho, anote as dependências entre as tarefas, atribua esforço e tempo para cada tarefa e defina responsabilidades, resultados e marcos.

Assim como em todas as outras áreas da engenharia de software, existem princípios básicos que guiam os cronogramas de projeto de software:

Divisão do trabalho. O projeto deve ser dividido em uma série de atividades e tarefas gerenciáveis. Para tanto, o produto e o processo são decompostos.

Interdependência. Deve ser determinada a interdependência de cada atividade ou tarefa resultante da divisão. Algumas tarefas devem ocorrer em sequência, enquanto outras podem acontecer em paralelo. Certas atividades não podem começar enquanto o resultado de outra não estiver disponível. Outras atividades podem ocorrer de forma independente.

Alocação do tempo. Para cada tarefa a ser programada, deve ser alocado certo número de unidades de trabalho (por exemplo, pessoas-dias de esforço). Além disso, para cada tarefa devem ser definidas uma data de início e uma data de término, que são uma função das interdependências, e se o trabalho será realizado em tempo integral ou parcial.

Validação do esforço. Cada projeto tem um número definido de pessoas na equipe de software. À medida que ocorre a alocação do tempo, você deve garantir que, em determinado momento, não seja programado mais do que o número alocado de profissionais. Por exemplo, considere um projeto para o qual são designados três engenheiros de software (três pessoas-dia estão disponíveis por dia de esforço atribuído).⁴ Em determinado dia, sete tarefas concomitantes devem ser executadas. Cada tarefa requer 0,50 pessoas-dia de esforço. Foi alocado mais esforço do que pessoas disponíveis para fazer o trabalho.

Definição de responsabilidades. Cada tarefa disposta no cronograma deve ser atribuída a um membro específico da equipe.

Definição dos resultados. Cada tarefa disposta no cronograma deve ter um resultado definido. Para projetos de software, o resultado normalmente é um artefato (por exemplo, o projeto de um componente) ou parte de um artefato. Artefatos muitas vezes são combinados em entregáveis.

Definição dos marcos. Cada tarefa ou grupo de tarefas deve estar associada a um marco no projeto. Um marco é atingido quando um ou mais artefatos teve sua qualidade examinada (Capítulo 19) e foi aprovado.

Cada um desses princípios é aplicado à medida que o projeto evolui.

⁴ Na realidade, há menos de 3 pessoas-dia disponíveis devido a reuniões, ausência por doença, férias e uma variedade de outras razões. Para nossos propósitos, no entanto, vamos considerar uma disponibilidade de 100%.

34.2.2 Relação entre pessoas e esforço

Há um mito conhecido e no qual muitos gerentes responsáveis por trabalhos de desenvolvimento de software ainda acreditam: “Se nos atrasarmos, podemos sempre acrescentar mais programadores e recuperar o tempo perdido mais tarde”. Infelizmente, acrescentar pessoas nas últimas fases de um projeto muitas vezes tem um efeito prejudicial, fazendo o cronograma se arrastar ainda mais. Os profissionais incluídos precisam aprender sobre o sistema e os encarregados de ensiná-los são os mesmos que estavam fazendo o trabalho. Enquanto explicam, nada é feito, e o projeto fica ainda mais atrasado.

Além do tempo necessário para conhecer o sistema, ter mais pessoas aumenta o número de caminhos de comunicação e a complexidade das comunicações em todo o projeto. Embora a comunicação seja essencial para o bom desenvolvimento do software, cada novo caminho requer esforço adicional e, portanto, tempo adicional.

Ao longo dos anos, dados empíricos e análises teóricas demonstraram que os cronogramas de projeto são elásticos. Ou seja, até certo ponto, é possível comprimir uma data de conclusão desejada para um projeto (acrescentando recursos). Também é possível estender uma data de conclusão (reduzindo o número de recursos).

A *Curva Putnam-Norden-Rayleigh* (PNR)⁵ dá uma indicação da relação entre esforço aplicado e prazo de entrega para um projeto de software. Uma versão da curva, representando esforço de projeto em função do prazo de entrega, é apresentada na Figura 34.1. A curva indica um valor mínimo t_o , que representa o custo mínimo para a entrega (o prazo de entrega que resultará no trabalho mínimo despendido). Quando nos movemos para a esquerda de t_o (quando tentamos acelerar a entrega), a curva sobe não linearmente.

Como exemplo, suponhamos que uma equipe de projeto tenha estimado que um nível de esforço E_d será necessário para conseguir um prazo de entrega nominal t_d , que é ótimo em termos de cronograma e recursos dispo-

Se você tiver de acrescentar pessoas a um projeto atrasado, não se esqueça de atribuir-lhes trabalho que já esteja bastante dividido.

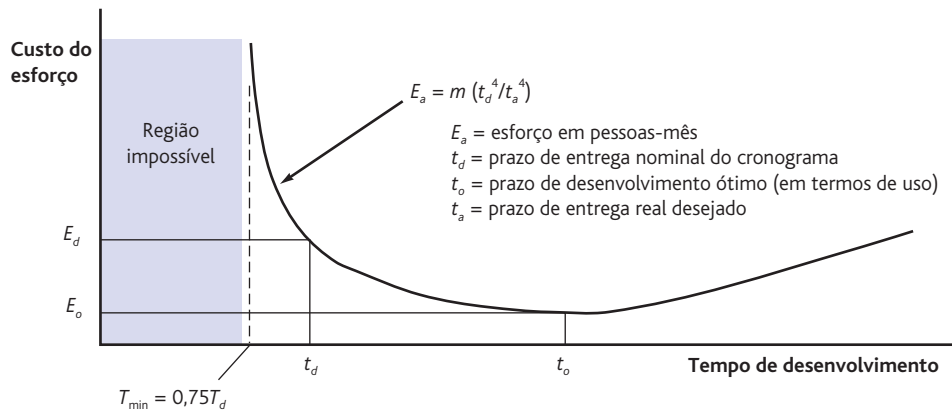


FIGURA 34.1 A relação entre esforço e prazo de entrega.

⁵ As pesquisas originais podem ser encontradas em [Nor70] e [Put78].

Se a entrega pode ser atrasada, a curva PNR indica que os custos do projeto podem ser reduzidos significativamente.

Conforme o prazo de entrega se torna cada vez mais apertado, você chega a um ponto em que o trabalho não pode ser terminado no prazo, independentemente do número de pessoas que estejam trabalhando. Enfrente a realidade e defina uma nova data de entrega.

níveis. Embora seja possível acelerar a entrega, a curva sobe de forma acentuada para a esquerda de t_d . A curva PNR indica, na verdade, que o prazo de entrega do projeto não pode ser comprimido além de $0,75t_d$. Se tentarmos uma compressão maior, o projeto passará para a “região impossível”, e o risco de fracasso se tornará muito alto. A curva PNR indica também que, para a opção de prazo de entrega de custo mais baixo, $t_o = 2t_d$. A implicação aqui é que o atraso na entrega pode reduzir os custos significativamente. Naturalmente, isso deve ser considerado levando-se em conta as consequências comerciais associadas ao atraso.

A equação do software [Put92] introduzida no Capítulo 33 é derivada da curva PNR e demonstra a relação altamente não linear entre o tempo cronológico para completar um projeto e o esforço humano aplicado ao projeto. O número de linhas de código produzidas, L , está relacionado a esforço e tempo de desenvolvimento pela equação:

$$L = P \times E^{1/3} t^{4/3}$$

onde E é o esforço de desenvolvimento em pessoas-mês, P é um parâmetro de produtividade que reflete uma variedade de fatores que levam a uma alta qualidade do trabalho de engenharia de software (valores típicos para P variam entre 2.000 e 12.000) e t é a duração do projeto em meses corridos.

Rearranjando essa equação de software, podemos chegar a uma expressão para trabalho de desenvolvimento E :

$$E = \frac{L^3}{P^3 t^4} \quad (34.1)$$

onde E é o esforço despendido (em pessoas-ano) durante todo o ciclo de vida do desenvolvimento de software e manutenção e t é o tempo de desenvolvimento em anos. A equação para o esforço de desenvolvimento pode ser relacionada ao custo do desenvolvimento pela inclusão de um fator inflacionado da taxa de trabalho (\$/pessoa-ano).

Isso leva a alguns resultados interessantes. Considere um projeto de software complexo, de tempo real, estimado em 33.000 LOC, 12 pessoas-ano de esforço. Se forem atribuídas 8 pessoas para a equipe de projeto, o projeto poderá ser feito em aproximadamente 1,3 ano. No entanto, se estendermos a data final para 1,75 ano, a natureza altamente não linear do modelo descrito na Equação (34.1) nos dará:

$$E = \frac{L^3}{P^3 t^4} \sim 3,8 \text{ pessoas-ano}$$

Isso implica que, estendendo em seis meses a data de entrega, podemos reduzir o número de pessoas de oito para quatro! A validade desses resultados está aberta ao debate, mas a consequência é clara: podem ser obtidos benefícios usando-se menos pessoas durante um período de tempo um pouco mais longo para atingir o mesmo objetivo.

34.2.3 Distribuição de esforço

Cada uma das técnicas de estimativa de projeto de software discutidas no Capítulo 33 leva a estimativas de unidades de trabalho (por exemplo, pessoa-mês) necessárias para completar o desenvolvimento do software. Uma distri-

buição recomendada do trabalho durante o processo de software muitas vezes é conhecida como *regra 40-20-40*. Quarenta por cento de todo o esforço é alocado na análise preliminar e de projeto. Uma porcentagem similar é aplicada ao teste posterior. Você pode inferir corretamente que a codificação (20% do trabalho) está em segundo plano.

Essa distribuição do esforço é usada apenas como guia.⁶ As características de cada projeto ditam a distribuição de esforços. O trabalho despendido em planejamento de projeto raramente passa de 2 a 3% do esforço, a menos que o plano induza uma organização a grandes despesas com alto risco. A comunicação com o cliente e as análises de requisitos podem abranger de 10 a 25% do trabalho do projeto. O esforço gasto em análise e protótipo deve aumentar em proporção direta com o tamanho e complexidade do projeto. Uma faixa de 20 a 25% do esforço é aplicada normalmente ao projeto de software. Deve ser considerado também o tempo gasto para revisão do projeto e subsequente iteração.

Devido ao esforço aplicado no projeto do software, a codificação deve seguir com relativamente pouca dificuldade. Pode-se aceitar um intervalo de 15 a 20% do esforço total. O teste e a subsequente depuração podem totalizar 30 a 40% do esforço de desenvolvimento do software. A criticidade do software muitas vezes determina o volume de teste necessário. Se o software estiver relacionado a vidas humanas (isto é, a falha do software pode resultar em perda de vidas humanas), normalmente as porcentagens podem ser mais altas.

Como o trabalho deve ser distribuído no fluxo do processo de software?

34.3 Definição de um conjunto de tarefas para o projeto de software

Seja qual for o modelo de processo escolhido, o trabalho que uma equipe executa é obtido com um conjunto de tarefas que permitem definir, desenvolver e, por fim, suportar software de computador. Não há um conjunto único de tarefas que seja adequado para todos os projetos. O conjunto de tarefas que seria apropriado para um sistema grande e complexo provavelmente seria considerado exagerado para um software pequeno e razoavelmente simples. Portanto, um processo de software eficaz definiria uma coleção de conjuntos de tarefas, cada uma projetada para atender às necessidades de diferentes tipos de projeto.

Conforme mencionamos no Capítulo 3, um conjunto de tarefas é uma coleção de tarefas de engenharia de software, marcos, artefatos e filtros de garantia de qualidade que precisam ser obtidos para completar um projeto em particular. O conjunto de tarefas deve proporcionar disciplina suficiente para se obter um software de alta qualidade, mas, ao mesmo tempo, não deve sobrecarregar a equipe com trabalho desnecessário.

⁶ Hoje, a regra 40-20-40 é criticada. Alguns acreditam que mais de 40% do esforço total deveria ser despendido durante a análise e o projeto. Por outro lado, alguns defensores do desenvolvimento ágil (Capítulo 5) argumentam que deveria ser gasto menos tempo “no início” e que uma equipe deveria passar rapidamente para a construção.

Para desenvolver um cronograma de projeto, um conjunto de tarefas deve ser distribuído ao longo da duração do projeto. O conjunto vai variar dependendo do tipo de projeto e do grau de rigor com que a equipe decide fazer seu trabalho. Embora seja difícil desenvolver uma classificação abrangente dos tipos de projeto, muitas empresas de software encontram os seguintes projetos:

1. *Projetos de desenvolvimento de conceito* iniciados para explorar algum conceito novo de negócio ou a aplicação de uma nova tecnologia.
2. *Projetos de desenvolvimento de novas aplicações* feitos a partir da solicitação de um cliente específico.
3. *Projetos de aperfeiçoamento de aplicação* que ocorrem quando um software existente passa por grandes modificações em sua função, desempenho ou interfaces observáveis pelo usuário.
4. *Projetos de manutenção de aplicação* corrigem, adaptam ou ampliam software existente de maneira não muito evidente ao usuário.
5. *Projetos de reengenharia* empreendidos com a intenção de recriar um sistema existente (legado) no todo ou em parte.

Mesmo dentro de um tipo de projeto, muitos fatores influenciam o conjunto de tarefas a ser selecionado. Esses fatores incluem [Pre05]: tamanho do projeto, número de usuários em potencial, importância da missão, longevidade da aplicação, estabilidade dos requisitos, facilidade de comunicação cliente/desenvolvedor, maturidade da tecnologia aplicável, restrições de desempenho, características internas e não internas, pessoal de projeto e fatores de reengenharia. Quando tomados de forma combinada, esses fatores fornecem uma indicação do *grau de rigor* com que o processo de software deve ser aplicado.

34.3.1 Um exemplo de conjunto de tarefas

Projetos de desenvolvimento de conceito ocorrem quando é preciso explorar o potencial de uma nova tecnologia. Não há certeza de que a tecnologia será aplicável, mas um cliente (por exemplo, o marketing) acredita que existem vantagens em potencial. Projetos de desenvolvimento de conceito são abordados aplicando-se estas tarefas principais:

- 1.1 **Definição do escopo do conceito** determina o escopo geral do projeto.
- 1.2 **Planejamento preliminar do conceito** estabelece a capacidade da empresa de assumir o trabalho sugerido pelo escopo do projeto.
- 1.3 **Avaliação do risco da tecnologia** avalia o risco associado à tecnologia a ser implementada como parte do escopo do projeto.
- 1.4 **Prova de conceito** demonstra a viabilidade de uma nova tecnologia no contexto de software.
- 1.5 **Implementação do conceito** implementa a representação do conceito de maneira que possa ser examinada por um cliente e usada para finalidades de “marketing”, quando um conceito deve ser vendido para outros clientes ou gerentes.

Um modelo de processo adaptável (APM, adaptable process model) foi desenvolvido para ajudar na definição de conjuntos de tarefas para vários projetos de software. Uma descrição completa do APM pode ser encontrada em www.rspa.com/apm.

1.6 Reação do cliente ao conceito solicita feedback sobre um novo conceito de tecnologia e foca-se nas aplicações especiais do cliente.

Um rápido exame dessas tarefas deve despertar algumas surpresas. Na verdade, o fluxo da engenharia de software para projetos de desenvolvimento de conceito (e também para todos os outros tipos de projeto) nada mais é do que bom senso.

34.3.2 Refinamento das tarefas principais

As tarefas principais (as ações de engenharia de software) descritas na seção anterior podem ser usadas para definir um cronograma macroscópico para um projeto. No entanto, ele deve ser refinado para criar um cronograma de projeto detalhado. O refinamento começa decompondo-se cada tarefa principal em uma série de tarefas (com os artefatos correspondentes e seus marcos).

Como exemplo de decomposição de tarefa, considere a Tarefa 1.1, Escopo do Conceito. O refinamento da tarefa pode ser obtido usando-se um formato de esboço, mas neste livro empregamos a abordagem de uma linguagem de projeto de processo para ilustrar o fluxo da atividade de escopo de conceito:

Definição de tarefa: Tarefa 1.1 Escopo do Conceito

- 1.1.1 Identifique as necessidades, benefícios e clientes em potencial;
 - 1.1.2 Defina saída/controle desejado e eventos de entrada que orientam a aplicação;
Início da Tarefa 1.1.2
 - 1.1.2.1 RT: Reveja a descrição da necessidade⁷
 - 1.1.2.2 Faça uma lista de saídas/entradas visíveis ao cliente
 - 1.1.2.3 RT: Examine saídas/entradas com o cliente e revise conforme necessário;Fim da tarefa 1.1.2
 - 1.1.3 Defina a funcionalidade/comportamento para cada função principal;
Início da Tarefa 1.1.3
 - 1.1.3.1 RT: Examine os objetos de saída e entrada de dados produzidos na tarefa 1.1.2;
 - 1.1.3.2 Crie um modelo de funções/comportamentos;
 - 1.1.3.3 RT: Examine as funções/comportamentos com o cliente e revise conforme necessário;Fim da tarefa 1.1.3
 - 1.1.4 Isole os elementos da tecnologia a ser implementados em software;
 - 1.1.5 Pesquise a disponibilidade de software existente;
 - 1.1.6 Defina a viabilidade técnica;
 - 1.1.7 Faça uma estimativa rápida do tamanho;
 - 1.1.8 Crie uma definição do escopo;
- Fim da definição: Tarefa 1.1

As tarefas e subtarefas mencionadas no refinamento da linguagem de projeto de processo formam a base de um cronograma detalhado para a atividade de escopo de conceito.

⁷ RT indica uma revisão técnica (Capítulo 20) que deve ser feita.

34.4 Definição de uma rede de tarefas

As tarefas e subtarefas têm interdependências baseadas em sua sequência. Além disso, quando há mais de uma pessoa envolvida em um projeto de engenharia de software, é provável que as atividades e tarefas de desenvolvimento sejam executadas em paralelo. Quando isso ocorre, tarefas concomitantes devem ser coordenadas para que estejam prontas quando outras mais adiante necessitarem de seus artefatos.

Uma *rede de tarefas*, também chamada de *rede de atividades*, é uma representação gráfica do fluxo de tarefas de um projeto. Às vezes é usada como um mecanismo por meio do qual a sequência e as dependências de tarefa são colocadas em uma ferramenta automática de cronograma de projeto. Em sua forma mais simples (usada ao se criar um cronograma macroscópico), a rede de tarefas representa as principais tarefas da engenharia de software. A Figura 34.2 mostra uma rede de tarefas esquemática para um projeto de desenvolvimento de conceito.

A natureza concomitante das atividades de engenharia de software gera um número de requisitos importantes para a elaboração do cronograma. Como tarefas paralelas ocorrem de forma assíncrona, você deve determinar as dependências entre elas para garantir o progresso contínuo até o término do trabalho. Preste atenção, também, nas tarefas que ficam no *caminho crítico*. Isto é, tarefas que devem ser finalizadas no prazo para que o projeto como um todo possa terminar no prazo. Esses problemas serão discutidos com mais detalhes ainda neste capítulo.

É importante observar que a rede de tarefas da Figura 34.2 é macroscópica. Em uma rede de tarefas detalhada (precursora de um cronograma detalhado), cada atividade na figura seria expandida. Por exemplo, a Tarefa 1.1 seria expandida para mostrar todas as tarefas detalhadas no refinamento da Tarefa 1.1 mostrada na Seção 34.3.2.

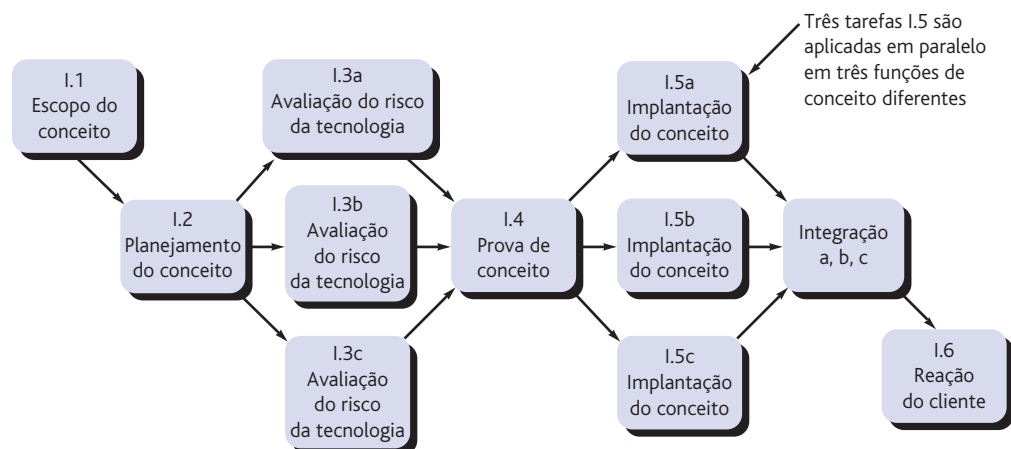


FIGURA 34.2 Uma rede de tarefas para desenvolvimento de conceito.

A rede de tarefas é um mecanismo útil para mostrar as dependências entre elas e determinar o caminho crítico.

34.5 Cronograma

O cronograma de um projeto de software não difere muito do cronograma de qualquer esforço de engenharia multitarefa. Portanto, ferramentas e técnicas generalizadas de cronogramas podem ser aplicadas com poucas modificações aos projetos de software.

A *técnica de avaliação e revisão de programa* (PERT, *program evaluation and review technique*) e o *método do caminho crítico* (CPM, *critical path method*) são dois métodos de elaboração de cronograma de projetos que podem ser aplicados ao desenvolvimento de software. As duas técnicas são controladas por informações já desenvolvidas em atividades anteriores de planejamento de projeto: estimativa de esforço, decomposição da função do produto, seleção do modelo de processo apropriado e do conjunto de tarefas e decomposição das tarefas selecionadas.

As interdependências entre as tarefas podem ser definidas por uma rede de tarefas. Também chamadas de *estrutura de subdivisão do trabalho* (WBS, *work breakdown structure*) do projeto, as tarefas são definidas para o produto como um todo ou para funções individuais.

Tanto PERT quanto CPM fornecem ferramentas quantitativas que permitem (1) determinar o caminho crítico – a cadeia de tarefas que determinam a duração do projeto, (2) estabelecer estimativas de tempo “mais prováveis” para tarefas individuais aplicando modelos estatísticos e (3) calcular “tempos-limite” que definem uma “janela” de tempo para uma tarefa em particular.

“Tudo o que temos de decidir é o que fazer com o tempo que nos é concedido.”

**Gandalf em
O Senhor dos Anéis:
A Sociedade do Anel**

FERRAMENTAS DO SOFTWARE



Cronograma de projeto

Objetivo: O objetivo das ferramentas de cronograma de projeto é permitir que o gerente defina as tarefas, estabeleça suas dependências, atribua recursos humanos às tarefas e desenvolva uma variedade de cartas, diagramas e tabelas que ajudem a acompanhar e controlar o projeto de software.

Mecanismos: Em geral, as ferramentas de cronograma de projeto exigem a especificação de uma estrutura de subdivisão do trabalho das tarefas ou a geração de uma rede de tarefas. Uma vez definido o desmembramento (um esboço) ou rede de tarefas, datas de início e fim, recursos humanos, prazos de entrega e outros dados são anexados a cada uma delas. A ferramenta gera, então, uma variedade de cartas de tempos e outras tabelas que permitem ao gerente avaliar o fluxo de tarefas. Esses dados podem ser atualizados continuamente no decorrer do projeto.

Ferramentas representativas:⁸

AMS Realtime, desenvolvida pela Advanced Management Systems (www.amsusa.com), tem recursos de cronograma para projetos de todos os tamanhos e tipos.

Microsoft Project, desenvolvida pela Microsoft (www.microsoft.com), é a ferramenta de cronograma de projeto baseada em PC mais amplamente utilizada.

4C, desenvolvida pela *4C Systems* (www.4csys.com), suporta todos os aspectos do planejamento de projeto, incluindo cronograma.

Uma lista abrangente de fornecedores e artefatos de software de gerenciamento de projeto pode ser encontrada em www.infogoal.com/pmc/pmcswr.htm.

⁸ A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

34.5.1 Gráfico de Gantt

Para criar o cronograma de um projeto de software, você deve começar com um conjunto de tarefas (a estrutura de subdivisão do trabalho). Se forem usadas ferramentas automáticas, a subdivisão do trabalho entra como uma rede de tarefas ou resumo de tarefas. Dados de esforço, duração e data de início são definidos para cada tarefa. Além disso, as tarefas podem ser atribuídas a indivíduos específicos.

Um gráfico de Gantt permite determinar que tarefas serão executadas em determinado ponto no tempo.

Como resultado dessas informações, é gerado um *gráfico de Gantt*. O gráfico de Gantt pode ser desenvolvido para o projeto inteiro. Outra opção é desenvolver gráficos separados para cada função do projeto ou para cada pessoa que trabalha no projeto.

A Figura 34.3 ilustra o formato de um gráfico de Gantt. Ela mostra parte de um cronograma de projeto de software que destaca a tarefa de **escopo do conceito** para um software processador de texto (WP, word processor). Todas as tarefas do projeto (para escopo do conceito) são listadas na coluna da esquerda. As barras horizontais indicam a duração de cada tarefa. Quando ocorrem várias barras ao mesmo tempo no calendário, é sinal de que há concomitância de tarefas. Os losangos indicam marcos.

Uma vez introduzidas as informações necessárias para a geração de uma carta de tempo, a maioria das ferramentas de cronograma de projeto de soft-

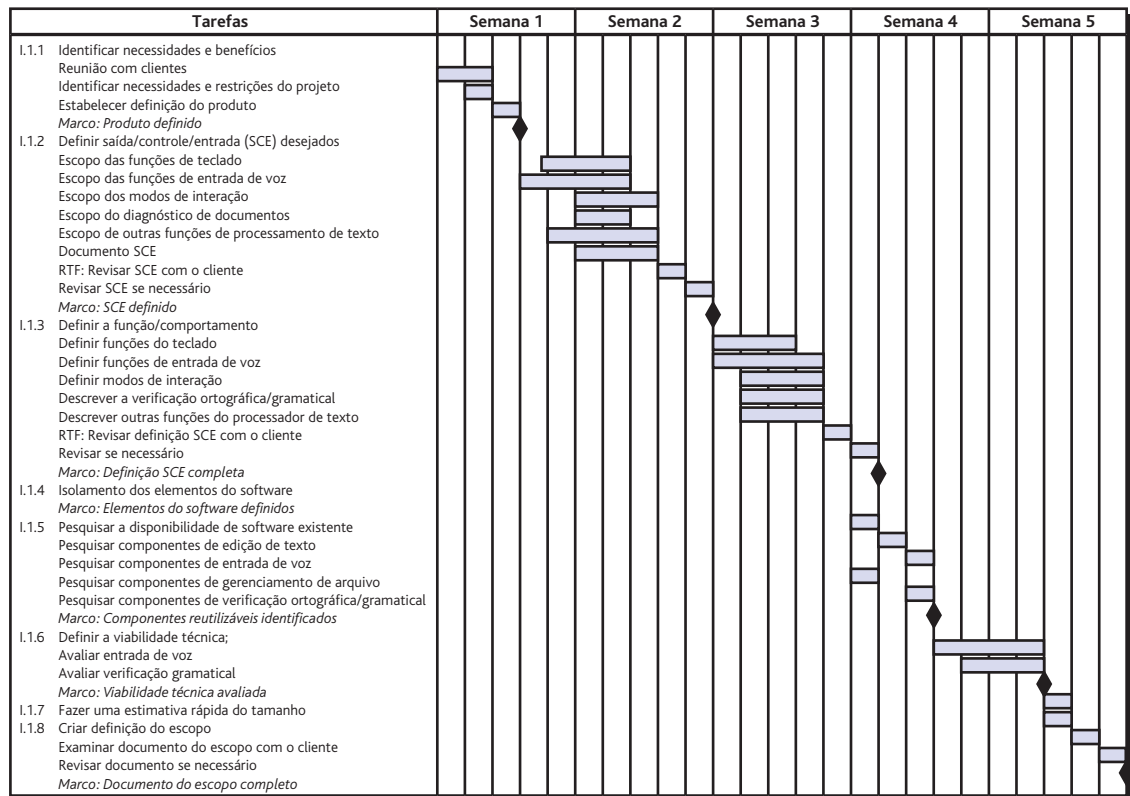


FIGURA 34.3 Exemplo de gráfico de Gantt.

Tarefas	Início planejado	Início real	Término planejado	Término real	Pessoa designada	Esforço alocado	Observações
I.1.1 Identificar necessidades e benefícios							
Reunião com clientes	sem1, d1	sem1, d1	sem1, d2	sem1, d2	BLS	2 p-d	O escopo necessitará mais esforço/tempo
Identificar necessidades e restrições do projeto	sem1, d2	sem1, d2	sem1, d2	sem1, d2	JPP	1 p-d	
Estabelecer definição do produto	sem1, d3	sem1, d3	sem1, d3	sem1, d3	BLS/JPP	1 p-d	
Marco: Produto definido	sem1, d3	sem1, d3	sem1, d3	sem1, d3			
I.1.2 Definir saída/controle/entrada (SCE) desejados							
Escopo das funções de teclado	sem1, d4	sem1, d4	sem2, d2		BLS	1,5 p-d	
Escopo das funções de entrada de voz	sem1, d3	sem1, d3	sem2, d2		JPP	2 p-d	
Escopo dos modos de interação	sem2, d1		sem2, d3		MLL	1 p-d	
Escopo do diagnóstico de documentos	sem2, d1		sem2, d2		BLS	1,5 p-d	
Escopo de outras funções de processamento de texto	sem1, d4	sem1, d4	sem2, d3		JPP	2 p-d	
Documento SCE	sem2, d1		sem2, d3		MLL	3 p-d	
RTF: Revisar SCE com o cliente	sem2, d3		sem2, d3		todos	3 p-d	
Revisar SCE se necessário	sem2, d4		sem2, d4		todos	3 p-d	
Marco: SCE definido	sem2, d5		sem2, d5				
I.1.3 Definir a função/comportamento							

FIGURA 34.4 Exemplo de tabela de projeto.

ware produz *tabelas de projeto* – uma listagem tabular de todas as tarefas de projeto, suas datas de início e fim planejadas e atuais, e uma variedade de informações relacionadas (Figura 34.4). Usadas em conjunto com a carta de tempo, as tabelas de projeto permitem acompanhar o progresso.

34.5.2 Acompanhamento do cronograma

Se tiver sido bem desenvolvido, o cronograma de projeto torna-se um roteiro que define as tarefas e os marcos a serem acompanhados e controlados à medida que o projeto avança. O acompanhamento pode ser feito de várias maneiras:

- Fazendo reuniões periódicas sobre o estado do projeto nas quais cada membro da equipe relata o progresso e os problemas;
- Avaliando os resultados de todas as revisões feitas durante o processo de engenharia de software;
- Determinando se os marcos formais do projeto (os losangos da Figura 34.3) foram atingidos na data programada;
- Comparando a data de início real com a data de início programada para cada tarefa de projeto listada na tabela de recursos (Figura 34.4);
- Reunindo-se informalmente com os profissionais para obter sua avaliação subjetiva do progresso até o momento e os problemas previstos;
- Usando análise de valor agregado (Seção 34.6) para avaliar o progresso quantitativamente.

Na realidade, todas essas técnicas de acompanhamento são empregadas por gerentes de projeto experientes.

O controle é empregado por um gerente de projeto de software para administrar os recursos do projeto, enfrentar os problemas e dirigir a equi-

“A regra básica do relatório de status de software pode ser resumida em uma única frase: ‘Sem surpresas’.”

Capers Jones

A melhor indicação do progresso é a conclusão e revisão bem-sucedidas de um artefato de software definido.

pe. Se tudo estiver bem (isto é, o projeto dentro do prazo e do orçamento, as revisões indicando que há progresso real e os marcos estão sendo alcançados), o controle é fácil. Entretanto, se ocorrem problemas, você deve exercer o seu controle para conciliar todos os itens o mais rápido possível. Depois de diagnosticado um problema, recursos adicionais podem se concentrar na área problemática: o pessoal pode ser realocado, ou o cronograma do projeto, redefinido.

Quando enfrentam pressões severas de prazo de entrega, gerentes de projeto experientes às vezes usam uma técnica de cronograma e controle de projeto chamada *caixa de tempo* (*time-boxing*) [Jal04]. A estratégia caixa de tempo reconhece que o produto completo pode não estar pronto no prazo de entrega predefinido. Então, é escolhido um paradigma de software incremental (Capítulo 4) e gerado um cronograma para cada entrega incremental.

As tarefas associadas a cada incremento são limitadas no tempo. Isso significa que o cronograma de cada tarefa é ajustado retroativamente a partir da data de entrega para o incremento. Uma “caixa” é traçada ao redor de cada tarefa. Quando uma tarefa chega ao limite de sua caixa de tempo (mais ou menos 10%), o trabalho é interrompido e inicia-se a próxima tarefa.

A reação inicial à abordagem caixa de tempo quase sempre é negativa: “Se o trabalho não está pronto, como podemos prosseguir?”. A resposta está na maneira como o trabalho é feito. Quando se atinge o limite da caixa de tempo, é provável que 90% da tarefa já tenha sido feita.⁹ Os 10% restantes, embora importantes, podem (1) ser adiados até o próximo incremento ou (2) ser concluídos mais tarde, se for necessário. Em vez de ficar “preso” em uma tarefa, o projeto prossegue em direção à data de entrega.

34.5.3 Acompanhamento do progresso de um projeto orientado a objetos

Embora um modelo iterativo seja o melhor framework para um projeto orientado a objetos, o paralelismo de tarefas torna o acompanhamento do projeto difícil. Você pode ter dificuldades para estabelecer marcos significativos para um projeto orientado a objetos porque muitas coisas diferentes acontecem ao mesmo tempo. Em geral, os marcos principais a seguir podem ser considerados “completos” quando os seguintes critérios forem atingidos:

Marco técnico: Análise orientada a objetos concluída

- Todas as classes e a hierarquia de classes foram definidas e revisadas.
- Os atributos de classe e operações associados a uma classe foram definidos e revisados.
- Os relacionamentos entre classes (Capítulo 10) foram estabelecidos e revisados.

⁹ Um cético pode se recordar do ditado: “Os primeiros 90% do sistema tomam 90% do tempo; os 10% restantes tomam 90% do tempo”.

- Um modelo comportamental (Capítulo 11) foi criado e revisado.
- As classes reutilizáveis foram identificadas.

Marco técnico: Projeto orientado a objetos concluído

- O conjunto de subsistemas foi definido e revisado.
- Classes foram alocadas a subsistemas e revisadas.
- A alocação de tarefas foi estabelecida e revisada.
- As responsabilidades e colaborações foram identificadas.
- Os atributos e operações foram atribuídos e revisados.
- O modelo de comunicação foi criado e revisado.

Marco técnico: Programação orientada a objetos concluída

- Cada nova classe foi implementada em código por meio do modelo de projeto.
- As classes extraídas (de uma biblioteca de reutilização) foram implementadas.
- Foi criado o protótipo ou incremento.

Marco técnico: Teste orientado a objetos

- Foi examinada a exatidão e totalidade dos modelos de análise e projeto orientado a objetos.
- Foi desenvolvida e revisada a rede responsabilidade-colaboração de classe (Capítulo 10).
- Foram criados os casos de teste e feitos os testes em nível de classe (Capítulo 24) para cada classe.
- Os casos de teste foram criados e o teste de conjunto (Capítulo 24) está concluído e as classes integradas.
- Foram finalizados os testes de sistema.

Depuração e teste ocorrem em harmonia. O status da depuração muitas vezes é avaliado considerando-se o tipo e número de erros (bugs) "pendentes".

Lembrando que o modelo de processo orientado a objetos é iterativo, cada um desses marcos deve ser revisitado conforme diferentes incrementos são entregues ao cliente.

34.5.4 Cronograma para projetos de WebApps e aplicativos móveis

O cronograma para projeto de WebApps e aplicativos móveis distribui o esforço estimado ao longo do tempo planejado (duração) para criar cada incremento. Isso é feito alocando-se o esforço para tarefas específicas. Entretanto, é importante notar que o cronograma geral evolui com o tempo. Durante a primeira iteração, é desenvolvido um cronograma macroscópico, que identifica todos os incrementos de WebApps ou aplicativos móveis e projeta as datas nas quais cada um será entregue. À medida que o desenvolvimento de um incremento progride, o item para o incremento no cronograma macroscópico é refinado em um cronograma detalhado. Nesse momento, são identificadas e agendadas tarefas de desenvolvimento específicas (necessárias para executar uma atividade).

Vamos considerar o **CasaSeguraGarantida.com** para entendermos melhor o cronograma macroscópico. Recordando discussões anteriores do **CasaSegura-Garantida.com**, podem ser identificados sete incrementos para o componente baseado na Web do projeto:

Incremento 1: Informações básicas da empresa e do produto

Incremento 2: Informações detalhadas do produto e downloads

Incremento 3: Cotações do produto e processamento de pedidos do produto

Incremento 4: Layout do espaço e projeto do sistema de segurança

Incremento 5: Informações e ordem dos serviços de monitoramento

Incremento 6: Controle online do equipamento de monitoramento

Incremento 7: Acesso às informações das contas

A equipe consulta e negocia com os envolvidos e desenvolve um cronograma *preliminar* de entrega para os sete incrementos. Um gráfico de Gantt para esse cronograma está na Figura 34.5.

As datas de entrega (representadas por losangos na carta de tempo) são preliminares e podem mudar conforme os cronogramas dos incrementos são detalhados. No entanto, esse cronograma macroscópico fornece ao gerente uma indicação de quando o conteúdo e a funcionalidade estarão disponíveis e quando o projeto inteiro estará concluído. Como estimativa preliminar, a equipe trabalhará para entregar todos os incrementos em um prazo de 12 semanas. Deve-se notar que alguns dos incrementos serão desenvolvidos em paralelo (por exemplo, incrementos 3, 4 e 7). Isso implica que a equipe terá pessoas em número suficiente para fazer todo o trabalho em paralelo.

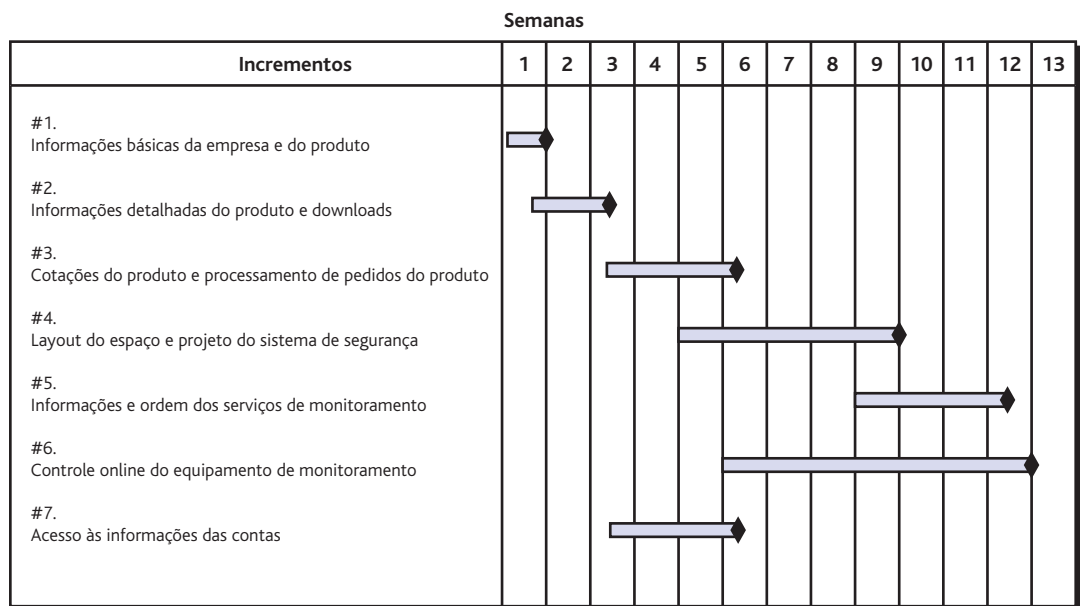


FIGURA 34.5 Gráfico de Gantt para cronograma macroscópico de projeto.

Uma vez desenvolvido o cronograma macroscópico, a equipe está pronta para agendar as tarefas para um incremento específico. Para tanto, pode-se usar a metodologia genérica de processo aplicável a todos os incrementos. Uma *lista de tarefas* é criada usando-se as tarefas genéricas derivadas como parte do framework, como ponto de partida, e depois adaptando-as, considerando conteúdo e funções a serem derivadas para um incremento de WebApp específico.

Cada ação do framework (e suas tarefas relacionadas) pode ser adaptada por uma dentre quatro maneiras: (1) uma tarefa é aplicada como ela está, (2) uma tarefa é eliminada porque não é necessária para o incremento, (3) uma nova tarefa (personalizada) é acrescentada e (4) uma tarefa é refinada (elaborada) em uma série de subtarefas, e cada uma delas se torna parte do cronograma.

Para ilustrar, considere uma ação genérica de *modelagem de projeto* para WebApps que pode ser executada aplicando-se uma ou todas as tarefas para WebApps discutidas no Capítulo 17. Como exemplo, considere a tarefa genérica *Projeto da Interface* da forma como é aplicada ao quarto incremento do **CasaSeguraGarantida.com**. Lembre-se de que o quarto incremento implementa o conteúdo e a função para descrever o espaço residencial ou comercial a ser protegido pelo sistema de segurança *CasaSegura*. De acordo com a Figura 34.5, o quarto incremento começa no início da quinta semana e termina no fim da nona semana.

Não há dúvida de que a tarefa *Projeto da Interface* deve ser executada. A equipe reconhece que o projeto da interface é fundamental para o sucesso do incremento e decide refinar (elaborar) a tarefa. As subtarefas a seguir são derivadas para a tarefa *Projeto da Interface* para o quarto incremento:

- Desenvolver um esboço do layout para a página de projeto do espaço.
- Rever o layout com os envolvidos.
- Projetar os mecanismos de navegação do layout do espaço.
- Projetar o layout “prancheta de desenho”.¹⁰
- Desenvolver detalhes procedurais para a função de layout da parede gráfica.
- Desenvolver detalhes procedurais para cálculo do comprimento da parede e função de display.
- Desenvolver detalhes procedurais para a função de layout da janela gráfica.
- Desenvolver detalhes procedurais para a função de layout da porta gráfica.
- Projetar mecanismos para selecionar componentes do sistema de segurança (sensores, câmeras, microfones etc.).

¹⁰ Nesse estágio, a equipe imagina criar o espaço literalmente desenhando as paredes, janelas e portas por meio das funções gráficas. As linhas das paredes vão se “encaixar” nos pontos de fixação. As dimensões da parede serão mostradas automaticamente. Janelas e portas serão posicionadas graficamente. O usuário pode também selecionar sensores, câmeras etc. específicos e posicioná-los quando o espaço estiver definido.

- Desenvolver detalhes procedurais para o layout gráfico dos componentes do sistema de segurança.
- Conduzir revisões em pares, se necessário.

Essas tarefas se tornam parte do cronograma para o quarto incremento da WebApp e são alocadas no cronograma de desenvolvimento do incremento. Elas podem ser colocadas no software de cronograma (por exemplo, Microsoft Project) e usadas para acompanhamento e controle.

CASASEGURA



Acompanhamento do cronograma

Cena: Escritório de Doug Miller antes do início do projeto do software CasaSegura.

Atores: Doug Miller (gerente da equipe de engenharia de software do CasaSegura) e Vinod Raman, Jamie Lazar e outros membros da equipe.

Conversa:

Doug (observando um slide no PowerPoint): O cronograma para o primeiro incremento do CasaSegura parece adequado, mas vamos ter problemas para acompanhar o andamento.

Vinod (com cara de preocupado): Por quê? Temos as tarefas dispostas no cronograma dia a dia, muitos artefatos, e temos certeza de que não estamos alocando recursos demais.

Doug: Tudo bem, mas como saberemos quando o modelo de requisitos para o primeiro incremento estará pronto?

Jamie: As coisas são iterativas; por isso, difíceis.

Doug: Compreendo, mas... bem, por exemplo, considere "classes de análise definidas". Você indicou isso como um marco.

Vinod: Sim.

Doug: Quem determina isso?

Jamie (sério): Elas estarão prontas quando estiverem prontas.

Doug: Isso não é suficiente, Jamie. Temos que agendar RTs [revisões técnicas, Capítulo 20], e você não fez isso. Uma revisão bem-sucedida do modelo de análise, por exemplo, é um marco razoável. Entendeu?

Jamie (contrariado): Ok, voltemos para a prancheta.

Doug: Não vai levar mais de uma hora para fazer as correções... todos os demais podem começar já.

34.6 Análise de valor agregado

O valor de retorno proporciona uma indicação quantitativa do progresso.

Na Seção 34.5, discutimos algumas abordagens qualitativas para o acompanhamento de projeto. Cada uma delas dá ao gerente uma indicação do progresso, mas uma avaliação das informações fornecidas é um tanto subjetiva. Será que há uma técnica quantitativa para avaliar o progresso, conforme a equipe de software avança nas tarefas alocadas para o cronograma do projeto? Na verdade, existe sim uma técnica para fazer a análise quantitativa do progresso. Ela é chamada de *análise de valor agregado* (EVA, *earned value analysis*). Humphrey [Hum95] discute o valor agregado da seguinte maneira:

O sistema de valor agregado proporciona uma escala de valores comum para todas as tarefas [de projeto de software], independentemente do tipo de trabalho executado. É estimado o total de horas para concluir o projeto e, a cada tarefa, é dado um valor agregado com base em sua porcentagem estimada do total.

Colocando de modo mais simples, o valor agregado é uma medida do progresso. Ele permite avaliar a "porcentagem de conclusão" de um projeto usando análise quantitativa, em vez de depender de suposições. Fleming e Koppleman [Fle98] argumentam que, na verdade, a análise do valor agregado

“proporciona leituras precisas e confiáveis do desempenho a partir dos 15% do projeto”. Para determinar o valor agregado, são feitos os seguintes passos:

1. O *custo orçado do trabalho programado* (BCWS, *budgeted cost of work scheduled*) é determinado para cada tarefa representada no cronograma. Durante a estimativa, é planejado o trabalho (em pessoas-horas ou pessoas-dias) para cada tarefa de engenharia de software. Desse modo, $BCWS_i$ é o trabalho planejado para a tarefa i . Para determinar o progresso em determinado ponto ao longo do cronograma de projeto, o valor de BCWS é a soma dos valores de $BCWS_i$ para todas as tarefas que devem ter sido concluídas naquele ponto no tempo no cronograma do projeto.
2. Os valores de BCWS para todas as tarefas são somados para derivar o *orçamento no final* (BAC, *budget at completion*). Assim,

$$BAC = \sum(BCWS_k)$$
 para todas as tarefas k
3. Em seguida, é calculado o valor para *custo orçado do trabalho executado* (BCWP, *budgeted cost of work performed*). O valor de BCWP é a soma dos valores de BCWS para todas as tarefas que foram realmente concluídas em certo momento no cronograma de projeto.

Wilkins [Wil99] observa que “a distinção entre BCWS e BCWP é que o primeiro representa o orçamento das atividades planejadas para serem concluídas e o último representa o orçamento das atividades que realmente foram concluídas”. Ao darmos valores para BCWS, BAC e BCWP, podemos calcular importantes indicadores de progresso:

Índice de desempenho do cronograma
(schedule performance index), $SPI = \frac{BCWP}{BCWS}$

Variância do cronograma (*schedule variance*), $SV = BCWP - BCWS$

O SPI é uma indicação da eficiência com a qual o projeto está utilizando os recursos programados. Um valor SPI próximo de 1,0 indica execução eficiente do cronograma de projeto. O SV é apenas uma indicação absoluta da variância em relação ao cronograma planejado.

Porcentagem programada para conclusão = $\frac{BCWS}{BAC}$

fornece uma indicação da porcentagem do trabalho que deve ter sido concluída no tempo t .

Porcentagem concluída = $\frac{BCWP}{BAC}$

fornece uma indicação quantitativa da porcentagem concluída de um projeto em dado instante t .

É possível também calcular o *custo real do trabalho executado* (ACWP, *actual cost of work performed*). O valor de ACWP é a soma do trabalho realmente despendido em tarefas concluídas até determinado instante no cronograma de projeto. É possível então calcular

Índice de desempenho de custo (*cost performance index*), $CPI = \frac{BCWP}{ACWP}$

Variância do custo (*cost variance*), $CV = BCWP - ACWP$

Como calcular o valor agregado e utilizá-lo para avaliar o progresso?

Uma grande variedade de recursos de análise de valor agregado pode ser encontrada em <http://www.acq.osd.mil/evm/>.

Um valor de CPI próximo de 1,0 fornece uma forte indicação de que o projeto está de acordo com o seu orçamento. CV é uma indicação absoluta de economia de custos (em relação aos custos planejados) ou déficit em determinado estágio de um projeto.

Como um radar de longo alcance, a análise de valor agregado esclarece as dificuldades do cronograma antes que elas possam se tornar aparentes. Isso permite tomar as ações corretivas antes que uma crise no projeto se desenvolva.

34.7 Resumo

O cronograma é o resultado da atividade de planejamento, que é um dos componentes principais do gerenciamento de projetos de software. Quando associado a métodos de estimativa e análise de riscos, o cronograma estabelece um mapa para o gerente de projeto.

O cronograma começa com a decomposição do processo. As características do projeto são empregadas para adaptar um conjunto de tarefas apropriado para o trabalho a ser feito. Uma rede de tarefas mostra cada tarefa de engenharia, sua dependência de outras tarefas e a duração projetada. A rede de tarefas é utilizada para calcular o caminho crítico, um gráfico de Gantt e uma variedade de informações de projeto. Usando o cronograma como guia, você pode acompanhar e controlar cada etapa no processo de software.

Problemas e pontos a ponderar

34.1 Prazos de entrega “não razoáveis” são um fato real no negócio de software. Como você procederá se tiver de enfrentar uma situação dessas?

34.2 Qual a diferença entre cronograma macroscópico e cronograma detalhado? É possível gerenciar um projeto se houver apenas o cronograma macroscópico? Por quê?

34.3 Pode haver um caso em que o marco de um projeto de software não esteja vinculado a uma revisão? Em caso afirmativo, dê um ou mais exemplos.

34.4 “Sobrecarga de comunicação” pode ocorrer quando várias pessoas trabalham em um projeto de software. O tempo gasto em comunicação reduz a produtividade individual (LOC/mês), e o resultado pode ser menor produtividade para a equipe. Ilustre (quantitativamente) como os engenheiros que são bem versados em boas práticas de engenharia de software e usam revisões técnicas podem aumentar a taxa de produção de uma equipe (quando comparada com a soma das taxas de produção individuais). Dica: você pode considerar que as revisões reduzem o retrabalho e que o retrabalho pode ser responsável por 20 a 40% do tempo de um profissional.

34.5 Embora acrescentar pessoas a um projeto de software em atraso possa retardá-lo ainda mais, há circunstâncias em que isso não é verdade. Descreva-as.

34.6 A relação entre pessoas e tempo é altamente não linear. Usando a equação do software de Putnam (descrita na Seção 34.2.2), desenvolva uma tabela que relacione número de pessoas com duração de projeto para um projeto de software que exige 50.000 LOC e 15 pessoas-ano de esforço (o parâmetro de produtividade é 5.000 e $B = 0,37$). Considere que o software deve ser entregue em 24 meses somando-se ou subtraindo-se 12 meses.

34.7 Suponha que você foi contratado por uma universidade para desenvolver um sistema de registro de curso online (*online course registration system* - OLCRS). Primeiro, aja como o cliente (se você é um estudante, isso é fácil) e especifique as características de um bom sistema. (Como alternativa, o professor lhe fornecerá uma série de requisitos preliminares para o sistema.) Usando os métodos de estimativa discutidos no Capítulo 33, desenvolva uma estimativa de esforço e duração para OLCRS. Sugira como você faria para:

- Definir atividades paralelas durante o projeto OLCRS.
- Distribuir o esforço ao longo do projeto.
- Estabelecer marcos para o projeto.

34.8 Selecione uma série de tarefas apropriadas para o projeto OLCRS.

34.9 Defina uma rede de tarefas para OLCRS descrita no Problema 34.7 ou, como alternativa, para outro projeto de software que lhe interesse. Não deixe de mostrar as tarefas e os marcos e faça estimativas de trabalho e duração para cada tarefa. Se possível, utilize uma ferramenta de cronograma automática para esse trabalho.

34.10 Se uma ferramenta de cronograma automática estiver à disposição, determine o caminho crítico para a rede definida no Problema 34.9.

34.11 Usando uma ferramenta de cronograma (se estiver disponível) ou papel e lápis (se necessário), desenvolva uma carta de tempo para o projeto OLCRS.

34.12 Suponha que você seja o gerente de projeto de software e que lhe pediram para calcular estatísticas de valor agregado para um pequeno projeto de software. O projeto tem 56 tarefas planejadas que, segundo as estimativas, exigem 582 pessoas-dia para ser concluída. No instante em que lhe pediram para fazer a análise de valor agregado, 12 tarefas já estavam prontas. No entanto, o cronograma do projeto indica que 15 tarefas deveriam ter sido concluídas. Estão disponíveis os seguintes dados de cronograma (em pessoas-dia):

Tarefa	Esforço planejado	Esforço real
1	12,0	12,5
2	15,0	11,0
3	13,0	17,0
4	8,0	9,5
5	9,5	9,0
6	18,0	19,0
7	10,0	10,0
8	4,0	4,5
9	12,0	10,0
10	6,0	6,5
11	5,0	4,0
12	14,0	14,5
13	16,0	—
14	6,0	—
15	8,0	—

Calcule o SPI, variância de cronograma, porcentagem programada para conclusão, porcentagem concluída, CPI e variância de custo para o projeto.

Leituras e fontes de informação complementares

Praticamente todos os livros escritos sobre gerenciamento de projetos de software contêm uma discussão sobre cronograma. O Project Management Institute (*PMBOK Guide*, 5ª ed., PMI, 2013), Wysoki (*Effective Project Management: Traditional, Agile, Extreme*, 6ª ed., Wiley, 2011), Lewis (*Project Planning Scheduling and Control*, 5ª ed., McGraw-Hill, 2010), Kerzner (*Project Management: A Systems Approach to Planning, Scheduling, and Controlling*, 10ª ed., Wiley, 2009), Chemuturi e Cagley (*Mastering Software Project Management: Best Practices, Tools, and Techniques*, J. Ross Publishing, 2010), Hughes e Cotterel (*Software Project Management*, 5ª ed., McGraw-Hill, 2009), Luckey e Phillips (*Software Project Management for Dummies*, For Dummies, 2006), Lewin (*Better Software Project Management*, Wiley, 2001) e Bennatan (*On Time, Within Budget: Software Project Management Practices and Techniques*, 3ª ed., Wiley, 2000) contêm interessantes discussões sobre o assunto. Embora seja específico da aplicação, Harris (*Planning and Scheduling Using Microsoft Office Project 2010*, Eastwood Harris Pty Ltd., 2010) fornece uma discussão útil sobre como ferramentas de cronograma podem ser usadas para monitorar e controlar com sucesso um projeto de software.

Fleming e Koppelman (*Earned Value Project Management*, 3ª ed., Project Management Institute Publications, 2010), Budd (*A Practical Guide to Earned Value Project Management*, 2ª ed., Management Concepts, 2009) e Webb e Wake (*Using Earned Value: A Project Manager's Guide*, Ashgate Publishing, 2003) discutem o uso de técnicas de valor agregado para planejamento, monitoramento e controle de projetos com detalhes consideráveis.

Uma ampla gama de fontes de informação sobre cronograma de projeto de software se encontra à disposição na Internet. Uma lista atualizada das referências da Web pode ser encontrada em “recursos de engenharia de software” no site da SEPA: www.mhhe.com/pressman.

Gestão de riscos 35

Em seu livro sobre análise e gestão de riscos, Robert Charette [Cha89] apresenta uma definição conceitual de risco:

Primeiro, risco diz respeito a acontecimentos futuros. Hoje e ontem já estão além da preocupação ativa, já que estamos colhendo o que plantamos com nossas ações passadas. A pergunta é, mudando nossas ações hoje, podemos criar uma oportunidade para uma situação diferente e, esperamos, melhor para nós mesmos amanhã? Segundo, isso significa que o risco envolve mudanças, como de ideia, opinião, ação ou lugar... [Terceiro,] risco envolve escolha e a incerteza que a própria escolha traz. Paradoxalmente, o risco, assim como a morte e os impostos, é uma das poucas certezas da vida.

Quando se considera risco no contexto da engenharia de software, os três fundamentos conceituais de Charette estão sempre em evidência. O futuro é a sua preocupação – quais riscos podem fazer o projeto de software dar errado? A mudança é sua preocupação – como as alterações nos requisitos do cliente, nas tecnologias de desenvolvimento, nos ambientes-alvo e em todas as outras entidades conectadas ao projeto afetam a cadência e o

Conceitos-chave

avaliação	781
categorias de risco	779
estratégias	778
proativas	778
reativas	778
exposição ao risco	786
identificação	780
lista de itens de risco	780
previsão	782
refinamento	787
RMMM	790
segurança e imprevistos	790
tabela de riscos	783

PANORAMA

O que é? Análise e gestão de risco são uma série de passos que ajudam uma equipe de software a entender e gerenciar a incerteza. Muitos problemas podem perturbar um projeto de software. O risco é um problema em potencial – ele pode ocorrer ou não. Independentemente do resultado, é aconselhável identificá-lo, avaliar sua probabilidade de ocorrência, estimar seu impacto e estabelecer um plano de contingência caso o problema realmente ocorra.

Quem realiza? Todos os envolvidos na gestão da qualidade – gerentes, engenheiros de software e outros – participam da análise e gestão de risco.

Por que é importante? Pense naquele lema dos escoteiros: “Sempre alerta”. Software é uma empreitada difícil. Muitas coisas podem dar errado e, francamente, frequentemente dão. Por essa razão, estar sempre alerta – entender os riscos e tomar medidas proativas para evitá-los ou administrá-los – é um elemento-chave do bom gerenciamento de projeto de software.

Quais são as etapas envolvidas? Reconhecer o que pode dar errado é o primeiro passo, chamado de “identificação do risco”. Em seguida, analisar o risco para determinar a probabilidade de que ocorra e o dano que causará se ocorrer. Levantadas essas informações, os riscos são classificados por probabilidade e por impacto. Por fim, é desenvolvido um plano para gerenciar os riscos de alta probabilidade e alto impacto.

Qual é o artefato? É produzido um plano de mitigação, monitoramento e gestão de risco (RMMM, risk mitigation, monitoring and management) ou um conjunto de formulários de informações sobre o risco.

Como garantir que o trabalho foi realizado corretamente? Os riscos analisados e gerenciados resultam de um estudo completo das pessoas, produto, processo e projeto. O RMMM deve ser sempre examinado à medida que o projeto avança, para garantir que os riscos se mantenham atualizados. Os planos de contingência para gestão de risco devem ser realistas.

sucesso geral? Por último, deve-se levar a sério as escolhas – que métodos e ferramentas usar, quantas pessoas envolver, quanta ênfase na qualidade é “suficiente”?

Peter Drucker [Dru75] certa vez disse: “Embora seja tolice tentar eliminar o risco e questionável tentar minimizá-lo, é essencial que os riscos assumidos sejam os certos”. Para que você consiga identificar os “riscos certos” a serem assumidos durante um projeto de software, é importante identificar todos os riscos evidentes, tanto para os gerentes quanto para os profissionais.

35.1 Estratégias de risco reativas *versus* proativas

“Se você não atacar
ativamente os
riscos, eles atacam
ativamente você.”

Tom Gilb

Estratégias de risco *reativas* são chamadas pejorativamente de “Escola Indiana Jones de gestão de riscos” [Tho92]. Em seus filmes, Indiana Jones, ao enfrentar uma enorme dificuldade, invariavelmente diz: “Não se preocupe, vou pensar em alguma coisa!”. Nunca se preocupando com os problemas até eles acontecerem, Indy reage de alguma forma heroica.

Infelizmente, o gerente de projeto de software não é o Indiana Jones, e os membros de sua equipe de projeto de software não são seus fiéis seguidores. Ainda assim, a maioria das equipes de software apoia-se apenas em estratégias de risco reativas. No melhor dos casos, uma estratégia reativa monitora o projeto à procura de riscos possíveis. São reservados recursos para enfrentar os riscos, caso se tornem problemas reais. Normalmente, a equipe de software não faz nada a respeito dos riscos até que alguma coisa dê errado. Desse modo, a equipe corre, na tentativa de corrigir o problema rapidamente. Isso costuma ser chamado de *modo de combate ao incêndio*. Quando falha, os “gestores de crises” [Cha92] assumem, e o projeto fica realmente ameaçado.

Uma estratégia consideravelmente mais inteligente para a gestão de riscos é ser proativo. Uma estratégia *proativa* começa muito antes do trabalho técnico. São identificados os riscos em potencial, avaliados a probabilidade e o impacto, e os riscos são classificados por ordem de importância. Então, a equipe de software estabelece um plano para gerenciar o risco. O objetivo primário é evitar o risco, mas, como nem todos os riscos podem ser evitados, o grupo trabalha para desenvolver um plano de contingência que lhe permita responder de maneira controlada e eficaz. Durante todo o restante deste capítulo, discutiremos uma estratégia proativa de gestão de risco.

35.2 Riscos de software

Embora haja muito debate sobre qual é a melhor definição para risco de software, há um consenso geral de que o risco sempre envolve duas características: *incerteza* – o risco pode ou não ocorrer; ou seja, não existem riscos

com 100% de probabilidade¹ – e *perda* – se o risco se tornar realidade, consequências ou perdas indesejadas ocorrerão [Hig95]. Quando os riscos são analisados, é importante quantificar o nível de incerteza e o grau de perda associados a cada risco. Para tanto, consideram-se diferentes categorias de risco.

Riscos de projeto ameaçam o plano do projeto. Isto é, se os riscos do projeto se tornarem reais, é possível que o cronograma fique atrasado e os custos aumentem. Os riscos de projeto identificam problemas potenciais de orçamento, cronograma, pessoal (equipes e organização), recursos, clientes e requisitos e seu impacto sobre o projeto de software. No Capítulo 33, a complexidade do projeto, seu tamanho e grau de incerteza estrutural também foram definidos como fatores de risco de projeto (e de estimativa).

Riscos técnicos ameaçam a qualidade e a data de entrega do software a ser produzido. Se um risco técnico em potencial se torna realidade, a implementação pode se tornar difícil ou impossível. Os riscos técnicos identificam problemas em potencial de projeto, implementação, interface, verificação e manutenção. Além disso, a ambiguidade de especificações, a incerteza técnica, a obsolescência técnica e a tecnologia “de ponta” também são fatores de risco. Riscos técnicos ocorrem porque o problema é mais difícil de resolver do que se pensava.

Riscos de negócio ameaçam a viabilidade do software a ser criado e muitas vezes ameaçam o projeto ou o produto. Candidatos aos cinco principais riscos de negócio são (1) criar um excelente produto ou sistema que ninguém realmente quer (risco de mercado), (2) criar um produto que não se encaixa mais na estratégia geral de negócios da empresa (risco estratégico), (3) criar um produto que a equipe de vendas não sabe como vender (risco de vendas), (4) perder o apoio da alta gerência devido à mudança no foco ou mudança de profissionais (risco gerencial) e (5) perder o orçamento ou o comprometimento dos profissionais (riscos de orçamento).

É extremamente importante observar que uma simples classificação de risco nem sempre funcionará. Alguns riscos são impossíveis de prever.

Outra classificação geral dos riscos foi proposta por Charette [Cha89]. *Riscos conhecidos* são aqueles que podem ser descobertos após uma cuidadosa avaliação do plano do projeto, do ambiente comercial e técnico no qual o projeto está sendo desenvolvido e de outras fontes de informação confiáveis (por exemplo, data de entrega irreal, falta de documentação dos requisitos ou do escopo do software, ambiente de desenvolvimento ruim). *Riscos previsíveis* são extrapolados da experiência de projetos anteriores (por exemplo, rotatividade do pessoal, comunicação deficiente com o cliente, diluição do esforço da equipe conforme as solicitações de manutenção vão sendo atendidas). *Riscos imprevisíveis* são o curinga do baralho. Eles podem ou não ocorrer, mas são extremamente difíceis de identificar com antecedência.

Que tipos de risco provavelmente encontraremos ao criar software?

“Projetos sem nenhum risco real são fracassos. Eles quase sempre são desprovidos de benefícios; e é por isso que não foram feitos anos atrás.”

Tom DeMarco e
Tim Lister

¹ Um risco 100% provável é uma restrição no projeto de software.

INFORMAÇÕES

**Sete princípios da gestão de riscos**

O Software Engineering Institute (SEI) (www.sei.cmu.edu) identifica sete princípios que "oferecem um framework para uma gestão de risco eficaz". São eles:

Mantenha uma perspectiva global – encare os riscos de software no contexto de um sistema no qual ele é um componente e o problema de negócio que se pretende resolver.

Tenha uma visão antecipada – considere os riscos que podem surgir no futuro (por exemplo, devido a mudanças no software); estabeleça planos de contingência para que os eventos futuros sejam controláveis.

Estimule a comunicação aberta – se alguém apontar um risco em potencial, não o menospreze. Se um risco é proposto de uma maneira informal, considere-o. Esti-

mule todos os envolvidos e usuários a sugerir riscos a qualquer momento.

Integre – uma consideração do risco deve ser integrada na gestão de qualidade.

Enfatize um processo contínuo – a equipe deve estar vigilante em toda a gestão da qualidade, modificando os riscos identificados à medida que mais informações forem conhecidas e acrescentando novos quando uma visão melhor for obtida.

Desenvolva uma visão compartilhada do produto – se todos os envolvidos compartilham da mesma visão do software, é provável que se tenha uma melhor identificação e avaliação do risco.

Estimule o trabalho de equipe – os talentos, habilidades e conhecimento de todos os envolvidos devem ser examinados quando se executam atividades de gestão de risco.

35.3 Identificação do risco

A identificação do risco é uma tentativa sistemática de especificar ameaças ao plano do projeto (estimativas, cronograma, recursos etc.). Ao identificar os riscos conhecidos e previsíveis, o gerente de projeto dá o primeiro passo para evitá-los quando possível e controlá-los quando necessário.

Há dois tipos de riscos para cada categoria apresentada na Seção 35.2. *Riscos genéricos* são ameaças em potencial a todo projeto de software. *Riscos específicos de produto* podem ser identificados somente por aqueles que têm uma visão clara da tecnologia, das pessoas e do ambiente específico para o qual o software está sendo desenvolvido. Para identificar riscos específicos de produto, são examinados o plano do projeto e a definição de escopo do projeto e procura-se uma resposta para a seguinte pergunta: "Que características especiais desse produto podem ameaçar o plano do nosso projeto?".

Um método para identificar riscos é criar uma *checklist* dos itens de risco. Ela pode ser usada para identificação do risco e concentra-se em alguns dos subconjuntos dos riscos conhecidos e previsíveis nas seguintes subcategorias genéricas:

- **Tamanho do produto** – riscos associados ao tamanho geral do software a ser criado ou modificado.
- **Impacto do negócio** – riscos associados às restrições impostas pela gerência ou pelo mercado.
- **Características do envolvido** – são riscos associados à sofisticação dos clientes e à habilidade do desenvolvedor em se comunicar com os envolvidos oportunamente.

Embora seja importante considerar os riscos genéricos, são os riscos específicos do produto que causam os maiores problemas. Dedique tempo suficiente para identificar tantos riscos de produto quantos forem possíveis.

- *Definição do processo* – riscos associados ao grau em que a gestão de qualidade foi definida e é seguida pela organização de desenvolvimento.
- *Ambiente de desenvolvimento* – riscos associados à disponibilidade e qualidade das ferramentas a serem usadas para criar o produto.
- *Tecnologia a ser criada* – riscos associados à complexidade do sistema a ser criado e à “novidade” da tecnologia que está embutida no sistema.
- *Quantidade de pessoas e experiência* – riscos associados à experiência técnica em geral e de projeto dos engenheiros de software que farão o trabalho.

A lista de itens de risco pode ser organizada de diversas maneiras. Questões relevantes a cada um dos tópicos podem ser respondidas para cada projeto de software. Com as respostas a essas questões, é possível estimar o impacto do risco. Outro formato de *checklist* de itens de risco simplesmente lista as características relevantes a cada subcategoria genérica. Por fim, um conjunto de “componentes e fatores de risco” [AFC88] é listado, com sua probabilidade de ocorrência. Fatores de desempenho, suporte, custo e cronograma são discutidos em resposta às últimas questões.

Existem muitas *checklists* abrangentes para riscos de projeto de software disponível na Web (por exemplo, [Baa07], [NAS07], [Wor04]). Você pode usá-las para ter uma visão dos riscos genéricos para projetos de software. Além do uso de *checklists*, foram propostos *padrões de risco* [Mil04] como uma abordagem sistemática para identificação de riscos.

35.3.1 Avaliação do risco geral do projeto

As questões a seguir foram extraídas dos dados de risco obtidos a partir de entrevistas com gerentes de projeto de software experientes em diversas partes do mundo [Kei98]. As questões estão ordenadas por sua importância relativa ao sucesso de um projeto.

1. A alta gerência e o cliente estão formalmente comprometidos em apoiar o projeto?
2. Os usuários estão bastante comprometidos com o projeto e o sistema/produto a ser criado?
3. Os requisitos são amplamente entendidos pela equipe de engenharia de software e seus clientes?
4. Os clientes foram totalmente envolvidos na definição dos requisitos?
5. Os usuários têm expectativas realistas?
6. O escopo do projeto é estável?
7. A equipe de engenharia de software tem a combinação de aptidões adequada?
8. Os requisitos de projeto são estáveis?
9. A equipe de projeto tem experiência com a tecnologia a ser implementada?

O projeto de software em que estamos trabalhando está em sério risco?

Risk Radar (radar de risco) é um banco de dados e ferramentas que ajudam gerentes a identificar, classificar e comunicar riscos do projeto. Pode ser encontrado em www.spmn.com.

10. O número de pessoas na equipe de projeto é adequado para o trabalho?
11. Todos os clientes e usuários concordam com a importância do projeto e com os requisitos do sistema/produto a ser criado?

Se alguma dessas questões for respondida negativamente, devem ser providenciados, imediatamente, processos de mitigação, monitoramento e gerenciamento. O grau de risco do projeto é diretamente proporcional ao número de respostas negativas a essas questões.

35.3.2 Componentes e fatores de risco

A Força Aérea Americana [AFC88] publicou um livreto contendo excelentes diretrizes para identificação e combate a riscos de software. A abordagem da Força Aérea exige que o gerente de projeto identifique os fatores de risco que afetam os componentes de risco de software – desempenho, custo, suporte e cronograma. No contexto dessa discussão, os componentes de risco são definidos da seguinte maneira:

"Gestão de riscos é gerenciamento de projeto para adultos."

Tim Lister

- *Risco de desempenho* – é o grau de incerteza de que o produto atenderá aos seus requisitos e será adequado para o uso que se pretende.
- *Risco de custo* – é o grau de incerteza de que o orçamento do projeto será mantido.
- *Risco de suporte* – é o grau de incerteza de que o software resultante será fácil de corrigir, adaptar e melhorar.
- *Risco de cronograma* — é o grau de incerteza de que o cronograma do projeto será mantido e que o produto será entregue a tempo.

O impacto de cada elemento motivador de risco sobre o componente de risco é dividido em quatro categorias de impacto – negligenciável, marginal, crítico ou catastrófico. Na Figura 35.1 [Boe89], uma caracterização das possíveis consequências dos erros (linhas 1) ou uma falha em obter um resultado desejado (linhas 2) são descritas. A categoria de impacto é escolhida com base na caracterização que melhor se adapta à descrição na tabela.

35.4 Previsão de risco

A *previsão de risco*, também chamada de *estimativa de risco*, tenta classificar cada risco de duas maneiras – (1) a possibilidade ou probabilidade de que o risco seja real e ocorrerá e (2) as consequências dos problemas associados ao risco, caso ele ocorra. Você trabalha com outros gerentes e pessoal técnico para executar quatro etapas de projeção de risco:

1. Estabelecer uma escala que reflita a possibilidade detectada de um risco.
2. Esboçar as consequências do risco.
3. Estimar o impacto do risco sobre o projeto e o produto.

Componentes		Desempenho	Suporte	Custo	Cronograma
Categoria					
Catastrófico	1	Falha em satisfazer o requisito resultaria em falha da missão		A falha resulta em aumento de custos e atrasos no cronograma com valores previstos que excedem \$ 500 mil	
	2	Degradação significativa até não cumprimento do desempenho técnico	Software que não responde com agilidade ou ao qual é difícil dar suporte	Dificuldades financeiras significativas, provável estouro no orçamento	Data de entrega não exequível
Crítico	1	Falha em atender o requisito degradará o desempenho do sistema até um ponto no qual o sucesso da missão é questionável		Falha resulta em atrasos operacionais e/ou aumento de custos com valores estimados entre \$ 100 mil e \$ 500mil	
	2	Alguma redução no desempenho técnico	Pequenos atrasos nas modificações de software	Alguma falta de recursos financeiros, possíveis estouros de orçamento	Possível atraso na data de entrega
Marginal	1	Falha em atender o requisito resultaria na degradação de missão secundária		Custos, impactos e/ou atrasos de cronograma recuperáveis com valores estimados de \$ 1 mil a \$ 100 mil	
	2	De mínima a pequena redução no desempenho técnico	Suporte responsivo de software	Recursos financeiros suficientes	Cronograma realista e possível
Negligenciável	1	Falha em atingir o requisito criaria inconveniência ou impacto não operacional		Erro resulta em pequeno impacto nos custos e/ou cronograma com valor esperado de menos de \$ 1 mil	
	2	Nenhuma redução do desempenho técnico	Software ao qual é fácil dar suporte	Possível sobra no orçamento	Data de entrega pode ser antecipada

Observação: (1) Possível consequência de erros ou falhas de software não detectadas.
 (2) Possível consequência se o resultado desejado não é obtido.

FIGURA 35.1 Avaliação de impacto.

Fonte: [Boe89]

4. Avaliar a exatidão geral da projeção de risco para que não haja mal-entendidos.

O objetivo dessas etapas é pensar os riscos de uma maneira que leve à definição de prioridades. Nenhuma equipe de software tem recursos para resolver todos os riscos possíveis com o mesmo grau de rigor. Priorizando os riscos, você pode alocar recursos onde eles terão maior impacto.

35.4.1 Desenvolvimento de uma tabela de riscos

Elaborar uma tabela de riscos é uma técnica simples para a projeção de riscos.² Um exemplo é apresentado na Figura 35.2.

² A tabela de risco pode ser implementada como um modelo de planilha. Isso permite fácil manipulação e ordenação dos valores.

Riscos	Categoria	Probabilidade	Impacto	RMMM
A estimativa de tamanho pode ser significativamente baixa	PS	60%	2	
Número de usuários maior do que o planejado	PS	30%	3	
Reutilização menor do que a planejada	PS	70%	2	
Os usuários resistem ao sistema	BU	40%	3	
O prazo de entrega será apertado	BU	50%	2	
Financiamento será perdido	CU	40%	1	
O cliente mudará os requisitos	PS	80%	2	
A tecnologia não atingirá as expectativas	TE	30%	1	
Falta de treinamento no uso das ferramentas	DE	80%	3	
Pessoal sem experiência	ST	30%	2	
A rotatividade do pessoal será alta	ST	60%	2	
Σ				
Σ				
Σ				

Valores de impacto:

- 1 – catastrófico
- 2 – crítico
- 3 – marginal
- 4 – negligenciável

FIGURA 35.2 Exemplo de tabela de riscos antes da ordenação.

Pense seriamente sobre o software que você vai criar e pergunte a si mesmo "o que pode dar errado?". Crie sua lista e peça a outros membros da equipe que façam o mesmo.

Primeiro, são listados todos os riscos (não importa quão remotos sejam) na primeira coluna da tabela. As *checklists* de itens de risco mencionadas na Seção 35.3 podem ajudar com essa tarefa. Cada risco é caracterizado na segunda coluna (por exemplo, PS significa risco de tamanho de projeto, BU significa risco de negócio). A probabilidade de cada risco é colocada na próxima coluna da tabela. O valor da probabilidade para cada risco pode ser estimado pelos membros da equipe individualmente. Para tanto, pode-se consultar os membros da equipe em ordem aleatória até que suas avaliações coletivas de risco comecem a convergir.

Em seguida, avalia-se o impacto de cada risco. É investigado cada componente de risco usando a caracterização apresentada na Figura 35.1 e determina-se uma categoria de impacto. São tiradas as médias³ das categorias de cada um dos quatro componentes de risco – desempenho, suporte, custo e cronograma – para determinar um valor de impacto global.

Uma vez preenchidas as quatro primeiras colunas da tabela, ela é ordenada por probabilidade e por impacto. Riscos de alta probabilidade e alto impacto se situam no topo da tabela, e riscos de baixa probabilidade posicionam-se no fim. Com isso, conclui-se a priorização de risco de primeira ordem.

Você pode estudar a tabela resultante e definir uma linha de corte. A *linha de corte* (traçada horizontalmente em algum ponto na tabela) implica que somente riscos que ficam acima da linha receberão mais atenção. Riscos que se

A tabela de riscos é ordenada por probabilidade e impacto para classificar os riscos.

³ Pode ser usada uma média ponderada se um componente de risco tiver mais significado para um projeto.

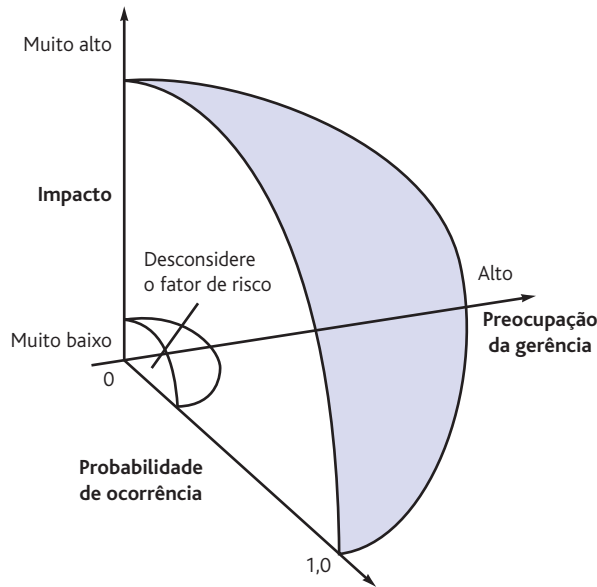


FIGURA 35.3 Riscos e preocupação da gerência.

posicionam abaixo da linha são reavaliados para uma priorização de segunda ordem. Como mostra a Figura 35.3, o impacto e a probabilidade do risco influem de modos distintos na preocupação do gerente. Um fator de risco com alto impacto, mas uma probabilidade de ocorrência muito baixa, não deve tomar um tempo significativo da gerência. No entanto, riscos de alto impacto com probabilidade entre moderada e alta e riscos de baixo impacto com alta probabilidade devem ser encaminhados para as etapas de análise de risco a seguir.

Todos os riscos que ficam acima da linha de corte devem ser gerenciados. A coluna com o título RMMM contém um ponteiro que aponta para um plano de *mitigação, monitoramento e gestão do risco* ou, como alternativa, um conjunto de formulários de informações desenvolvido para todos os riscos que se posicionam acima da linha de corte. O plano RMMM e os formulários de informações de risco são discutidos nas Seções 35.5 e 35.6.

A probabilidade do risco pode ser determinada com estimativas individuais e, depois, pelo desenvolvimento de um valor de consenso. Embora essa abordagem funcione, foram desenvolvidas técnicas mais sofisticadas para determinar a probabilidade do risco (por exemplo, [McC09]).

"[Hoje] ninguém se dá ao luxo de conhecer uma tarefa tão bem que não possa ter surpresas, e surpresa significa risco."

Stephen Grey

35.4.2 Avaliação do impacto do risco

Três fatores afetam as prováveis consequências caso um risco ocorra: sua natureza, seu escopo e sua época. A natureza do risco indica os problemas que podem surgir se ele ocorrer. Por exemplo, uma interface externa para o hardware do cliente mal definida (um risco técnico) logo atrapalhará o projeto e os testes, e provavelmente causará problemas na integração do sistema no fim do projeto. O escopo de um risco relaciona a gravidade (quão sério é ele?) com sua distribuição geral (quanto do projeto será afetado ou quantos clientes serão prejudicados?). Por fim, a época do risco considera quando e por quanto

Como avaliamos as consequências de um risco?

tempo o impacto será sentido. Em muitos casos, você vai querer que as “más notícias” ocorram o mais cedo possível, mas em alguns, quanto mais tarde, melhor.

Retornando mais uma vez à abordagem de análise de risco proposta pela Força Aérea Americana [AFC88], podem-se aplicar os seguintes procedimentos para determinar as consequências gerais de um risco: (1) determinar o valor médio da probabilidade de ocorrência para cada componente de risco, (2) usando a Figura 35.1, determinar o impacto para cada componente com base no critério mostrado e (3) completar a tabela de risco e analisar os resultados conforme descrito nas seções anteriores.

A *exposição ao risco* (RE, *risk exposure*) geral é determinada pela seguinte relação [Hal98]:

$$RE = P \times C$$

onde P é a probabilidade de ocorrência de um risco e C é o custo para o projeto, caso o risco ocorra.

Por exemplo, suponha que a equipe de software defina o risco de um projeto da seguinte maneira:

Identificação do risco. Somente 70% dos componentes de software programados para reutilização serão realmente integrados na aplicação. A funcionalidade restante terá de ser desenvolvida de maneira personalizada.

Probabilidade do risco. 80% (provavelmente).

Impacto do risco. Foram planejados 60 componentes de software reutilizáveis. Se somente 70% puderem ser usados, 18 componentes terão de ser desenvolvidos do zero (além de outros softwares personalizados que foram planejados para serem desenvolvidos). Como cada componente tem, em média, 100 LOC e os dados locais indicam que o custo de engenharia de software para cada LOC é de \$ 14, o custo total (impacto) para desenvolver os componentes seria $18 \times 100 \times 14 = \$25.200$.

Exposição ao risco. $RE = 0,80 \times 25.200 \sim \$20.200$.

Compare a RE de todos os riscos com a estimativa de custos para o projeto. Se RE for maior do que 50% do custo do projeto, a viabilidade do projeto deve ser avaliada.

Feita a estimativa do custo do risco, a exposição ao risco pode ser calculada para cada risco na tabela de riscos. A exposição total para todos os riscos (acima da linha de corte na tabela de riscos) pode ser um meio de ajustar a estimativa de custo final para um projeto. Ela também pode ser usada para prever o aumento provável nos recursos de pessoal necessários em vários pontos durante o cronograma do projeto.

As técnicas de projeção e análise de risco descritas nas Seções 35.4.1 e 35.4.2 são aplicadas iterativamente à medida que o projeto de software avança.⁴ A equipe deve rever a tabela de risco a intervalos regulares, reavaliando cada risco para determinar quando novas circunstâncias causam mudanças na probabilidade e no impacto. Como consequência dessa atividade, pode ser necessário acrescentar novos riscos à tabela, remover alguns que não são mais relevantes e mudar as posições relativas dos que restarem.

⁴ Caso tenha interesse, um tratamento mais matemático do custo do risco é apresentado em [Ben10].

CASASEGURA

**Análise de risco**

Cena: Escritório de Doug Miller antes do início do projeto do software *CasaSegura*.

Atores: Doug Miller (gerente da equipe de engenharia de software do *CasaSegura*) e Vinod Raman, Jamie Lazar e outros membros da equipe.

Conversa:

Doug: Gostaria de dedicar um tempo para um brainstorming sobre os riscos do projeto *CasaSegura*.

Jamie: Do tipo “o que pode dar errado”?

Doug: Sim. Aqui estão algumas categorias em que as coisas podem dar errado.

[Ele mostra a todos as categorias listadas na introdução da Seção 35.3.]

Vinod: Humm... você quer apenas chamar a nossa atenção para elas ou...

Doug: Não, veja o que acho que devemos fazer. Cada um faz uma lista de riscos... agora mesmo...

[Dez minutos para todos escreverem.]

Doug: Ok, parem.

Jamie: Mas eu ainda não terminei!

Doug: Tudo bem. Veremos a lista novamente. Agora, para cada item da sua lista, atribua uma porcentagem de pro-

babilidade de que o risco venha a ocorrer. Depois, atribua um impacto ao projeto em uma escala de 1 (pequeno) a 5 (catastrófico).

Vinod: Se achar que o risco é alto, especifique uma probabilidade de 50% e se achar que ele tem um impacto moderado sobre o projeto, especifique um 3, certo?

Doug: Exatamente.

[Cinco minutos, todos escrevendo.]

Doug: Ok, parem. Agora vamos fazer uma lista do grupo no quadro branco. Eu escrevo; vou pegar um item por vez de cada lista de vocês em uma sequência de rodadas.

[Quinze minutos depois, a lista está criada.]

Jamie (apontando para o quadro e rindo): Vinod, aquele risco (apontando para um item do quadro) é ridículo. É mais fácil sermos atingidos por um raio. Devemos removê-lo.

Doug: Não, vamos deixar por enquanto. Consideraremos todos os riscos; não importa que sejam absurdos. Depois, vamos limpar a lista.

Jamie: Mas já temos mais de 40 riscos... como poderemos controlar todos eles?

Doug: Não podemos. É por esse motivo que definiremos um ponto de corte após ordenarmos todos os riscos. Farei isso depois, e iremos nos reunir amanhã novamente. Por ora, voltemos ao trabalho... e, nos intervalos de folga, pensem sobre qualquer outro risco que tenham esquecido.

35.5 Refinamento do risco

Durante os estágios iniciais do planejamento de projeto, um risco pode ser especificado de forma bem geral. À medida que o tempo passa e se conhece mais sobre o projeto e o risco, talvez seja possível refinar o risco em um conjunto de riscos mais detalhados, cada um deles mais fácil de mitigar, monitorar e gerenciar.

Uma maneira de fazer isso é representar o risco em um formato *condição-transição-consequência* (CTC) [Glu94]. O risco é definido da seguinte maneira:

Considerando que <condição>, então há a preocupação de que (possivelmente) <consequência>.

Usando o formato CTC para o risco de reutilização descrito na Seção 35.4.2, podemos escrever:

Considerando que todos os componentes de software reutilizáveis devem estar em conformidade com padrões específicos de projeto e que alguns deles não se enquadram nesses padrões, há uma preocupação de que (possivelmente) somente

Qual é uma boa maneira de descrever um risco?

70% dos módulos que se planejava reutilizar possam realmente ser integrados na montagem do sistema, resultando na necessidade de criar de forma personalizada os 30% dos componentes restantes.

Essa condição geral pode ser refinada da seguinte maneira:

Subcondição 1. Certos componentes reutilizáveis foram desenvolvidos por uma equipe terceirizada que não conhecia os padrões internos de projeto.

Subcondição 2. O padrão de projeto para as interfaces de componente não foi completamente estabelecido e pode não estar em conformidade com certos componentes reutilizáveis existentes.

Subcondição 3. Certos componentes reutilizáveis foram implementados em uma linguagem não suportada no ambiente em que serão usados.

As consequências associadas a essas subcondições refinadas permanecem as mesmas (30% dos componentes de software devem ser criados de forma personalizada), mas o refinamento ajuda a isolar os riscos subjacentes e pode levar a uma análise e resposta mais fáceis.

35.6 Mitigação, monitoramento e gestão de riscos (RMMM)

"Se tomo tantas medidas de precaução, é porque não quero dar chance ao azar."

Napoleão

Todas as atividades de análise de risco apresentadas até aqui têm um único objetivo: ajudar a equipe de projeto a desenvolver uma estratégia para lidar com o risco. Uma estratégia eficaz deve considerar três aspectos: como evitar o risco, como monitorar o risco e como gerenciar o risco e planejar a contingência.

Se a equipe de software adota uma abordagem proativa ao risco, evitar o risco é sempre a melhor estratégia. Para tanto, desenvolve-se um plano de *mitigação de risco*. Por exemplo, suponha que a alta rotatividade de pessoal seja o risco r_1 de um projeto. Com base no histórico passado e na intuição do gerente, a probabilidade l_1 de alta rotatividade é estimada em 0,70 (70%, um tanto alta) e o impacto x_1 é projetado como crítico. A alta rotatividade terá um impacto crítico sobre o custo e sobre o cronograma do projeto.

Para mitigar esse risco, é preciso desenvolver uma estratégia para reduzir a rotatividade. Entre as providências possíveis a serem tomadas, citamos:

O que podemos fazer para mitigar um risco?

- Reunir-se com o pessoal para determinar as causas da rotatividade (por exemplo, más condições de trabalho, salário baixo, mercado de trabalho competitivo).
- Mitigar as causas que estão sob o seu controle antes do início do projeto.
- Uma vez iniciado o projeto, assumir que a rotatividade acontecerá e desenvolver técnicas para garantir a continuidade quando as pessoas saírem.
- Organizar equipes de projeto para que as informações sobre cada atividade de desenvolvimento sejam amplamente difundidas.
- Definir padrões para os produtos do projeto e estabelecer mecanismos para assegurar que todos os modelos e documentos sejam desenvolvidos a tempo.

- Realizar revisões em pares de todo o trabalho (para que mais de uma pessoa esteja “por dentro”).
- Designar um substituto para cada profissional cujo trabalho seja crítico.

Com o avanço do projeto, começam as atividades de *monitoramento de risco*. O gerente de projeto monitora fatores que podem indicar se o risco está se tornando mais ou menos possível. No caso da alta rotatividade de pessoal, a atitude geral dos membros da equipe baseada nas pressões do projeto, o grau de coesão da equipe, as relações pessoais entre os membros da equipe, os problemas em potencial com remuneração e benefícios e a disponibilidade de empregos dentro e fora da empresa são monitorados.

Além de monitorar esses fatores, um gerente de projeto deve monitorar a efetividade das providências para a mitigação do risco. Por exemplo, uma providência citada recomendava a definição de padrões para o produto e mecanismos para assegurar que os produtos sejam desenvolvidos a tempo. Esse é um mecanismo para garantir a continuidade se um elemento crítico deixar o projeto. O gerente de projeto deve monitorar os produtos cuidadosamente para garantir que cada um seja autossuficiente e forneça as informações necessárias se um novato precisar entrar na equipe de software no meio do projeto.

A *gestão de riscos e o plano de contingência* consideram que os esforços de mitigação do risco falharam e que o risco se tornou uma realidade. Continuando o exemplo, o projeto está em andamento, e um grupo de pessoas avisa que vai sair. Se a estratégia de mitigação foi utilizada, existe pessoal substituto disponível, as informações estão documentadas e todo o conhecimento é compartilhado dentro da equipe. Além disso, pode-se mudar temporariamente o foco dos recursos (e reajustar o cronograma do projeto) para aquelas funções que estão com a equipe completa, permitindo que os novatos a serem acrescentados à equipe “entrem logo no ritmo”. As pessoas que estão saindo devem interromper todo o trabalho e passar suas últimas semanas envolvidas em atividades de “transferência de conhecimentos”. Isso pode incluir captação de conhecimento por meio de vídeo, desenvolvimento de “documentos de comentários ou Wikis” e/ou reuniões com outros membros da equipe que permanecerão no projeto.

É importante observar que as etapas de mitigação, monitoramento e gestão de risco (RMMM, risk mitigation, monitoring, and management) acarretam custo adicional ao projeto. Por exemplo, o tempo gasto para incluir um substituto para cada técnico essencial custa dinheiro. Parte da gestão de risco é avaliar quando os benefícios acumulados pelas providências de RMMM são superados pelos custos associados a sua implementação. Basicamente, executa-se uma análise de custo-benefício clássica. Se as providências para evitar os riscos de alta rotatividade aumentarem o custo e a duração do projeto em uma estimativa de 15%, mas o fator de custo predominante for a “substituição do profissional”, o gerente pode decidir não implementar essa etapa. Por outro lado, se houver uma projeção de que as providências para evitar o risco aumentarão os custos em 5% e a duração em apenas 3%, o gerente provavelmente vai tomar essas providências.

Se a exposição a um risco específico for menor do que seu custo de mitigação, não tente mitigar o risco, mas continue monitorando-o.

Para um grande projeto, 30 ou 40 riscos podem ser identificados. Se para cada um deles forem identificados de 3 a 7 passos de gestão de riscos, ela pode se tornar um projeto em si mesma. Por essa razão, deve-se adaptar a regra 80-20 de Pareto para o risco de software. A experiência indica que 80% do risco geral de projeto (80% do potencial de falha do projeto) podem ser responsáveis por apenas 20% dos riscos identificados. O trabalho executado durante as primeiras etapas de análise de risco o ajudará a determinar quais dos riscos estão incluídos nesses 20% (por exemplo, riscos que levam à mais alta exposição ao risco). Desse modo, alguns dos riscos identificados, avaliados e projetados podem não entrar no plano RMMM – eles não estão incluídos nos 20% críticos (os riscos com prioridade mais alta no projeto).

O risco não está limitado ao próprio projeto de software. Riscos podem ocorrer depois que o software foi desenvolvido com sucesso e entregue ao cliente. Esses riscos estão tipicamente associados às consequências da falha no software em campo.

Segurança do software e análise de imprevistos (por exemplo, [Dun02], [Her00], [Lev95]) são atividades de garantia da qualidade de software (Capítulo 21) que se concentram na identificação e avaliação de imprevistos em potencial que podem afetar negativamente o software e fazer o sistema inteiro falhar. Se imprevistos puderem ser identificados antecipadamente no processo de engenharia de software, poderão ser especificadas características do projeto do software que servirão para eliminar ou controlar os imprevistos em potencial.

35.7 O plano RMMM

Uma estratégia de gestão de risco pode ser incluída no plano de projeto de software, ou as etapas de gestão de risco podem ser organizadas em um *plano de mitigação, monitoramento e gestão* separado. O plano RMMM documenta todo o trabalho executado como parte da análise de risco e é usado pelo gerente de projeto como parte do plano geral de projeto.

Algumas equipes de software não desenvolvem um documento RMMM formal. Em vez disso, cada risco é documentado individualmente usando-se um *formulário de informações de risco* (RIS, *risk information sheet*) [Wil97]. Em muitos casos, o RIS é mantido por meio de um sistema de banco de dados para que a criação e introdução de informações, ordem de prioridade, pesquisas e outras análises possam ser feitas facilmente. O formato do RIS está ilustrado na Figura 35.4.

Uma vez documentado o RMMM e começado o projeto, iniciam-se as etapas de mitigação e monitoramento de risco. Conforme já discutimos, mitigação de risco é uma atividade para evitar problemas. O monitoramento de risco é uma atividade de acompanhamento de projeto com três objetivos primários: (1) avaliar se os riscos previstos vão ocorrer de fato; (2) assegurar que as etapas de mitigação ao risco definidas para o risco estejam sendo aplicadas adequadamente; e (3) coletar informações que podem ser usadas para futuras análises de riscos. Em muitos casos, os problemas que ocorrem durante um projeto podem estar ligados a mais de um risco. Outra função do monitoramento de risco é tentar definir a origem [quais riscos causaram quais problemas durante o projeto].

Formulário de informações de risco			
ID do risco: P02-4-32	Data: 09/05/09	Prob: 80%	Impacto: alto
Descrição: Somente 70% dos componentes de software programados para reutilização serão, de fato, integrados na aplicação. A funcionalidade restante terá de ser desenvolvida de maneira personalizada.			
Refinamento/contexto: Subcondição 1: Certos componentes reutilizáveis foram desenvolvidos por uma equipe terceirizada que não conhecia os padrões internos de projeto. Subcondição 2: O padrão de projeto para as interfaces de componente não foi completamente estabelecido e pode não estar em conformidade com certos componentes reutilizáveis existentes. Subcondição 3: Certos componentes reutilizáveis foram implementados em uma linguagem não suportada no ambiente em que serão usados.			
Mitigação/monitoramento: 1. Contate a empresa terceirizada para determinar a conformidade com os padrões de projeto. 2. Pressione para que haja padronização da interface; considere a estrutura de componente ao decidir sobre o protocolo de interface. 3. Determine o número de componentes que estão na categoria da subcondição 3; determine se pode ser adquirido o suporte de linguagem.			
Gerenciamento/plano de contingência/disparo: Foi calculada a exposição ao risco: \$ 20.200. Reserve esse valor no custo de contingência do projeto. Desenvolva um cronograma revisado assumindo que 18 componentes adicionais terão de ser criados de forma personalizada; defina a equipe de maneira correspondente. Disparador: as providências para mitigação improdutivas em 01/07/09.			
Estado atual: 12/05/09: iniciadas as etapas de mitigação.			
Autor: D. Gagne		Designado: B. Laster	

FIGURA 35.4 Formulário de informações de risco.

Fonte: [Wil97]

FERRAMENTAS DO SOFTWARE

**Gestão de riscos**

Objetivo: O objetivo das ferramentas de gestão de riscos é ajudar a equipe de projeto na definição dos riscos, avaliar seu impacto e probabilidade e acompanhar os riscos durante todo o projeto de software.

Mecanismos: Em geral, as ferramentas de gestão de riscos ajudam na identificação genérica dos riscos fornecendo uma lista de riscos típicos de projeto e comerciais, proporcionando *checklists* ou outras técnicas de "entrevista" que ajudam a identificar riscos específicos de projeto, atribuindo probabilidade e impacto a cada risco, mantendo estratégias

de mitigação de risco e gerando vários relatórios diferentes relacionados aos riscos.

Ferramentas representativas:⁵

@risk, desenvolvida pela Palisade Corporation (www.palisade.com), é uma ferramenta genérica de análise de risco que usa simulação Monte Carlo para controlar seu mecanismo analítico.

Riskman, distribuída pela ABS Consulting (www.absconsulting.com/riskmansoftware/index.html), é um sistema especializado de avaliação de risco que identifica riscos relacionados a projeto.

⁵ A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

Risk Radar, desenvolvida pela SPMN (www.spmn.com), auxilia os gerentes de projeto na identificação e gerenciamento de riscos de projeto.

ARM, desenvolvida pela Deltek (www.deltek.com), é uma ferramenta baseada na Web que permite a fornecedores, clientes e equipes de projeto geograficamente distribuídos colaborar em conhecimento de risco básico.

X:PRIMER, desenvolvida pela GrafP Technologies (www.grafp.com), é uma ferramenta genérica baseada na Web que prevê o que pode dar errado em um projeto e identifica as principais causas para falhas em potencial e as ações eficazes a tomar.

35.8 Resumo

Sempre que um projeto de software estiver em execução, o bom senso manda que se faça a análise de risco. No entanto, a maioria dos gerentes de projeto de software a faz informal e superficialmente, quando a faz. O tempo que se gasta identificando, analisando e controlando os riscos traz retornos de muitas maneiras – menos pressão durante o projeto, mais capacidade de acompanhar e controlar um projeto e a confiança que resulta do planejamento para os problemas antes que eles ocorram.

A análise de riscos pode absorver um grande esforço de planejamento do projeto. A identificação, projeção, avaliação, gestão e monitoramento, tudo toma tempo. Mas o esforço é recompensado. Citando Sun Tzu, general chinês que viveu há 2.500 anos: “Se você conhece o inimigo e conhece a si mesmo, não precisa temer o resultado de uma centena de batalhas”. Para o gerente de projeto de software, o inimigo é o risco.

Problemas e pontos a ponderar

35.1 Dê cinco exemplos de outros campos que ilustram os problemas associados à estratégia reativa de riscos.

35.2 Descreva a diferença entre “riscos conhecidos” e “riscos previsíveis”.

35.3 Acrescente três questões ou tópicos a cada uma das *checklists* de itens de risco apresentadas no site da SEPA.

35.4 Você foi designado para criar um software de suporte para um sistema de edição de vídeo de baixo custo. O sistema aceita vídeo digital como entrada, armazena vídeo em disco e depois permite que o usuário faça uma grande variedade de edições no vídeo digitalizado. O resultado pode, então, ser gravado em DVD ou outro tipo de mídia. Faça algumas pesquisas sobre sistemas desse tipo e depois produza uma lista dos riscos tecnológicos que enfrentaria ao empreender um projeto desse tipo.

35.5 Você é o gerente de projeto de uma grande empresa de software. Foi designado para liderar uma equipe que está desenvolvendo um software processador de texto “avançado”. Crie uma tabela de riscos para o projeto.

35.6 Descreva a diferença entre componentes de risco e fatores de risco.

35.7 Desenvolva uma estratégia de mitigação de risco e especifique atividades específicas de mitigação de risco para três dos riscos descritos na Figura 35.2.

35.8 Desenvolva uma estratégia de monitoramento de risco e especifique as atividades de monitoramento de risco para três dos riscos descritos na Figura 35.2. Não se esqueça

de identificar os fatores que estará monitorando para determinar se o risco está se tornando mais ou menos possível.

35.9 Desenvolva uma estratégia de gestão de risco e especifique atividades de gestão para três dos riscos descritos na Figura 35.2.

35.10 Tente refinar três dos riscos descritos na Figura 35.2 e depois crie formulários de informações de risco para cada um deles.

35.11 Represente três dos riscos mostrados na Figura 35.2 usando um formato CTC.

35.12 Recalcule a exposição de risco discutida na Seção 35.4.2 quando o custo por LOC é de \$ 16 e a probabilidade é de 60%.

35.13 Você pode pensar em uma situação na qual um risco de alta probabilidade e alto impacto não seria considerado parte do seu plano RMMM?

35.14 Descreva cinco áreas de aplicação de software nas quais a análise de segurança e imprevistos do software seriam uma preocupação importante.

Leituras e fontes de informação complementares

A literatura sobre gestão de risco de software se expandiu significativamente nas últimas décadas. Mun (*Modeling Risk*, Wiley, 2ª ed., 2010) apresenta um tratamento matemático detalhado da análise de risco que pode ser aplicada aos projetos de software. Mulcahy (*Risk Management, Tricks of the Trade for Project Managers*, 2ª ed., RMC Publications, 2010), Kendrick (*Identifying and Managing Project Risk*, 2ª ed., American Management Association, 2009), Crohy e seus colegas (*The Essentials of Risk Management*, McGraw-Hill, 2006) e Marrison (*The Fundamentals of Risk Measurement*, McGraw-Hill, 2002) apresentam métodos e ferramentas úteis que todo gerente de projeto pode usar. Jindal e seus colegas (*Risk Management in Software Engineering*, Create Space Independent Publishing, 2012) discutem a incorporação da avaliação de risco de segurança como parte do desenvolvimento de sistemas.

DeMarco e Lister (*Dancing with Bears*, Dorset House, 2003) escreveram um livro interessante e esclarecedor que orienta os gerentes e profissionais de software na gestão de riscos. Moynihan (*Coping with IT/IS Risk Management*, Springer-Verlag, 2002) apresenta opiniões pragmáticas de gerentes de projeto que tratam de riscos continuamente. Royer (*Project Risk Management*, Management Concepts, 2002) e Smith e Merritt (*Proactive Risk Management*, Productivity Press, 2002) sugerem um processo proativo para gestão de risco. Karolak (*Software Engineering Risk Management*, Wiley, 2002) escreveu um guia que introduz um modelo de análise de risco fácil de usar, com *checklists* e questionários suportados por um pacote de software.

Capers Jones (*Assessment and Control of Software Risks*, Prentice Hall, 1994) apresenta uma discussão detalhada dos riscos de software, incluindo dados coletados de centenas de projetos de software. Jones define 60 fatores de risco que podem afetar o resultado dos projetos de software. Boehm [Boe89] sugere um excelente questionário e formatos de *checklist* que podem ser valiosos na identificação do risco. Charette [Cha89] apresenta uma abordagem detalhada da mecânica da análise de risco, usando teoria de probabilidades e técnicas estatísticas para analisar os riscos. Em outro volume, Charette (*Application Strategies for Risk Analysis*, McGraw-Hill, 1990) discute o risco no contexto de sistema e engenharia de software e sugere estratégias pragmáticas para gestão de risco. Gilb (*Principles of Software Engineering Management*, Addison-Wesley, 1988) apresenta uma série de “princípios” (frequentemente são divertidos e às vezes profundos) que podem servir de excelente guia para gestão de risco.

Ewusi-Mensah (*Software Development Failures: Anatomy of Abandoned Projects*, MIT Press, 2003) e Yourdon (*Death March*, Prentice Hall, 1997) discutem o que acontece quando os riscos engolfam a equipe de projeto de software. Bernstein (*Against the Gods*, Wiley, 1998) apresenta uma história interessante do risco que remonta aos tempos antigos.

O Software Engineering Institute publicou muitos relatórios e guias detalhados sobre análise e gestão de riscos. O panfleto do Air Force Systems Command AFSCP 800-45 [AFC88] descreve técnicas de identificação e redução de riscos. Todas as edições do *ACM Software Engineering Notes* têm uma seção denominada “Risks to the Public” (editor, P. G. Neumann). Se quiser conhecer as mais recentes e melhores histórias de horror do software, esse é o lugar.

Há disponível na Internet uma grande variedade de fontes de informação sobre gestão de riscos de software. Uma lista atualizada das referências da Web pode ser encontrada em “recursos de engenharia de software” no site da SEPA: **www.mhhe.com/pressman**.

Seja qual for seu domínio de aplicação, tamanho ou complexidade, o software continuará a evoluir com o tempo. As mudanças motivam esse processo. No âmbito do software, alterações ocorrem quando erros são corrigidos, quando há adaptação a um novo ambiente, quando o cliente solicita novas características ou funções e quando a aplicação passa por um processo de reengenharia para se atualizar a um contexto moderno. Nos últimos 40 anos, Manny Lehman (por exemplo, Leh97a) e seus colegas fizeram análises detalhadas de software e sistemas industriais na tentativa de desenvolver uma *teoria unificada para a evolução do software* (*unified theory for software evolution*). Os detalhes desse

Conceitos-chave

análise de inventário . . .	803
engenharia direta	811
engenharia reversa	805
dados	807
interfaces de usuário	808
processamento	807
manutenção de software	796
manutenibilidade	797

PANORAMA

O que é? Considere qualquer produto de tecnologia que tenha lhe servido bem. Você o utiliza regularmente, mas ele está ficando obsoleto. Apresenta problemas com muita frequência, leva mais tempo para ser reparado e já não representa a tecnologia mais atualizada. O que fazer? Por um tempo, você tenta consertá-lo, remendá-lo ou até ampliar sua funcionalidade. Isso se chama manutenção. Mas a manutenção se torna cada vez mais difícil à medida que os anos passam. Chega, então, um momento em que é preciso reconstruí-lo. Você criará um produto com mais funcionalidades, melhor desempenho e confiabilidade e de manutenção mais fácil. Isso é o que chamamos de reengenharia.

regularmente, mas ele está ficando obsoleto. Apresenta problemas com muita frequência, leva mais tempo para ser reparado e já não representa a tecnologia mais atualizada. O que fazer? Por um tempo, você tenta consertá-lo, remendá-lo ou até ampliar sua funcionalidade. Isso se chama manutenção. Mas a manutenção se torna cada vez mais difícil à medida que os anos passam. Chega, então, um momento em que é preciso reconstruí-lo. Você criará um produto com mais funcionalidades, melhor desempenho e confiabilidade e de manutenção mais fácil. Isso é o que chamamos de reengenharia.

Quem realiza? No nível organizacional, a manutenção é feita por pessoal de suporte que faz parte da empresa de engenharia de software. A reengenharia é executada por especialistas de negócio (muitas vezes empresas de consultoria) e, no nível de software, por engenheiros de software.

Por que é importante? Vivemos em um mundo que muda rapidamente. As demandas por funções de negócio e a tecnologia da informação que as suporta estão mudando em um ritmo que impõe enorme pressão competitiva sobre todas as organizações comerciais. É por essa razão que o software deve ser mantido continuamente e, no momento apropriado, deve passar por reengenharia para acompanhar o ritmo.

Quais são as etapas envolvidas? A manutenção corrige defeitos, adapta o software para atender a um ambiente em mudança e melhora a funcionalidade para atender às necessidades dos clientes. Em um nível estratégico, a reengenharia de processos de negócio (BPR, *business process reengineering*) define objetivos comerciais, identifica e avalia processos comerciais existentes e cria processos de negócio revisados que melhor atendem aos objetivos atuais. A reengenharia de software abrange análise de inventário, reestruturação de documentos, engenharia reversa, reestruturação de programas e dados e engenharia direta. O objetivo dessas atividades é criar versões dos programas que apresentem uma qualidade mais alta e melhorar a manutenibilidade.

Qual é o artefato? Uma variedade de produtos de manutenção e reengenharia (por exemplo, casos de uso, modelos de análise e projeto, procedimentos de teste) é produzida. O resultado final é um software atualizado.

Como garantir que o trabalho foi realizado corretamente? Use as mesmas práticas de SQA aplicadas a todos os processos de engenharia de software – revisões técnicas avaliam os modelos de análise e projeto; revisões especializadas consideram a aplicabilidade e compatibilidade de negócio; e são aplicados testes para descobrir erros em conteúdo, funcionalidade e interoperabilidade.

reengenharia de processos de negócio (BPR)	799
reengenharia de software	802
reestruturação	809
código	809
dados	810
reestruturação dos documentos	804
suportabilidade	798

Por que os sistemas legados evoluem com o tempo?

trabalho estão além dos objetivos deste livro, mas é válido conhecer as leis que daí se originaram [Leh97b]:

A Lei da Mudança Contínua (1974): Software implementado em um contexto de computação real e que, portanto, vai evoluir com o tempo (chamado de *sistema tipo E*) deve ser adaptado continuamente ou se tornará cada vez menos satisfatório.

A Lei da Complexidade Crescente (1974): À medida que um sistema tipo E evolui, sua complexidade aumenta, a menos que seja feito um trabalho para mantê-la ou reduzi-la.

A Lei da Autorregulação (1974): O processo de evolução do sistema tipo E é autorregulado com distribuição do produto e medidas de processo próximas do normal.

A Lei da Conservação da Estabilidade Organizacional (1980): A taxa de atividade efetiva média em um sistema tipo E em evolução é invariante durante o tempo de vida do produto.

A Lei da Conservação da Familiaridade (1980): Conforme um sistema tipo E evolui, tudo o que está associado a ele (por exemplo, desenvolvedores, pessoal de vendas, usuários) deve manter o domínio de seu conteúdo e comportamento para uma evolução satisfatória. Um crescimento excessivo diminui esse domínio. Portanto, o crescimento incremental médio permanece invariante à medida que o sistema evolui.

A Lei do Crescimento Contínuo (1980): O conteúdo funcional dos sistemas tipo E deve ser continuamente ampliado durante toda a sua existência para manter a satisfação do usuário.

A Lei da Qualidade em Declínio (1996): A qualidade dos sistemas tipo E vai parecer diminuir a menos que eles sejam rigorosamente mantidos e adaptados às mudanças do ambiente operacional.

A Lei do Sistema de Realimentação (1996): Processos tipo E em evolução constituem sistemas de realimentação multinível, multilaço, multiagente, e devem ser tratados como tais para que se obtenha uma melhoria significativa em qualquer base razoável.

As leis que Lehman e seus colegas definiram fazem parte da realidade do engenheiro de software. Neste capítulo, discutiremos o desafio das atividades de manutenção e reengenharia de software necessárias para ampliar a vida dos sistemas legados.

36.1 Manutenção de software

A manutenção começa quase imediatamente. O software é liberado para os usuários e, em alguns dias, os relatos de erros começam a chegar à empresa de engenharia de software. Em algumas semanas, uma classe de usuários indica que o software deve ser mudado para se adaptar às necessidades especiais de seus ambientes. E, em alguns meses, outro grupo corporativo, ainda não estava interessado no software quando ele foi lançado, reconhece que ele pode trazer vantagens. Eles precisarão de algumas melhorias para fazer o software funcionar em seu mundo.

O desafio da manutenção do software começou. Enfrentamos uma fila cada vez maior de correções de erros, solicitações de adaptação e melhorias que devem ser planejadas, programadas e, por fim, executadas. Logo, a fila já cresceu muito, e o trabalho ameaça devorar os recursos disponíveis. Com o passar do tempo, sua empresa descobre que está gastando mais tempo e dinheiro com a manutenção dos programas do que criando novas aplicações. Não é raro uma empresa de software despende de 60% a 70% de todos os recursos com manutenção de software.

Você pode se perguntar por que é necessária tanta manutenção e por que tanto esforço deve ser despendido. Osborne e Chikofsky [Os90] dão uma resposta parcial à questão:

Muitos softwares dos quais dependemos hoje têm de 10 a 15 anos, em média. Mesmo quando esses programas foram criados, usando as melhores técnicas de projeto e codificação conhecidas na época (e muitos não foram), o tamanho do programa e o espaço de armazenamento eram as preocupações principais. Eles então migraram para novas plataformas, foram ajustados para mudanças nas máquinas e na tecnologia dos sistemas operacionais e aperfeiçoados para atender a novas necessidades dos usuários – tudo isso sem grande atenção à arquitetura geral. O resultado são estruturas mal projetadas, mal codificadas, de lógica deficiente e mal documentadas em relação aos sistemas de software, e que devemos manter funcionando...

Outra razão para o problema de manutenção do software é a mobilidade dos profissionais. É provável que a equipe (ou pessoa) responsável pelo trabalho original não esteja mais por perto. Pior ainda, outras gerações de profissionais de software podem ter modificado o sistema e já se foram. E hoje pode não ter restado ninguém que tenha conhecimento direto do sistema legado.

Conforme observamos no Capítulo 29, a natureza ubíqua das alterações permeia todo o trabalho de software. Mudanças são inevitáveis quando sistemas baseados em computador são criados; portanto, você deve desenvolver mecanismos para avaliar, controlar e fazer modificações.

Neste livro, destacamos a importância de entender o problema (análise) e desenvolver uma solução bem estruturada (projeto). A Parte II deste livro é dedicada aos mecanismos dessas ações de engenharia de software, e a Parte III concentra-se nas técnicas para garantir que você as tenha feito corretamente. Tanto a análise quanto o projeto levam a uma importante característica do software que chamamos de manutenibilidade. Em essência, *manutenibilidade* é um indicativo qualitativo¹ da facilidade de corrigir, adaptar ou melhorar o software. Grande parte das funções da engenharia de software é criar sistemas que apresentem alta manutenibilidade.

Mas o que é manutenibilidade? Software “manutenível” apresenta uma modularidade eficaz (Capítulo 12). Utiliza padrões de projeto (Capítulo 16) que permitem entendê-lo facilmente. Foi construído usando padrões e convenções de codificação bem definidos, levando a um código-fonte autodocumentado e inteligível. Passou por técnicas de garantia de qualidade (Parte III deste livro)

“Manutenibilidade e clareza de programa são conceitos paralelos: quanto mais difícil for entender um programa, mais difícil será mantê-lo.”

Gerald Berns

¹ Existem muitas medidas quantitativas que fornecem uma indicação indireta da manutenibilidade (por exemplo, [Sch99], [SEI02]).

que descobriram problemas de manutenção em potencial antes que o software fosse lançado. Foi criado por engenheiros de software que reconhecem que não estarão por perto quando as alterações tiverem de ser feitas. Portanto, o projeto e a implementação do software devem “ajudar” a pessoa que for fazer a alteração.

36.2 Suportabilidade do software

A fim de fornecer efetivamente suporte a software de classe industrial, a empresa (ou seus projetistas) deve ser capaz de fazer correções, adaptações e melhorias inerentes à atividade de manutenção. Além disso, a empresa deve desempenhar outras atividades importantes, que incluem suporte operacional continuado, suporte ao usuário e atividades de reengenharia durante toda a vida útil do software. Uma boa definição da *suportabilidade do software* é esta:

... a capacidade de fornecer suporte a um sistema de software durante toda a vida útil do produto. Isso implica satisfazer qualquer necessidade ou requisito, mas também prover de equipamento, infraestrutura de suporte, software adicional, serviços de conveniências, mão de obra ou qualquer outro recurso necessário para manter o software operacional e capaz de satisfazer suas funções. [SSO08]

Em essência, a suportabilidade é um dos muitos fatores de qualidade que devem ser considerados durante a análise e o projeto que são parte da gestão da qualidade. Ela deve ser tratada como parte do modelo de requisitos (ou especificações) e considerada conforme o projeto evolui e a construção inicia.

Por exemplo, a necessidade de software “antidefeito” em nível de componente e código foi discutida anteriormente neste livro. O software deve conter recursos para ajudar o pessoal de suporte quando for encontrado um defeito no ambiente operacional (e não se iluda, defeitos *serão* encontrados). Além disso, o pessoal de suporte deve ter acesso a um banco de dados que contenha os registros de todos os defeitos já detectados – suas características, causa e solução. Isso permitirá que o pessoal de suporte examine defeitos “similares” e possibilitará diagnóstico e correção mais rápidos.

Embora os erros encontrados em uma aplicação sejam um problema crítico de suporte, a suportabilidade também exige que sejam providenciados recursos para resolver os problemas diários dos usuários. A função do pessoal de suporte é responder às dúvidas dos usuários sobre instalação, operação e uso da aplicação.

36.3 Reengenharia

Em um artigo seminal escrito para a *Harvard Business Review*, Michael Hammer [Ham90] lançou os fundamentos para uma revolução no pensamento gerencial sobre processos de negócio e computação:

Uma grande quantidade de documentos sobre suportabilidade de software pode ser encontrada em www.softwaresupportability.org/Downloads.html.

Está na hora de parar de fazer as trilhas das vacas. Em vez de incorporar processos ultrapassados em hardware e software, devemos rejeitá-los e começar de novo. Devemos fazer uma “reengenharia” em nossos negócios: usar o poder da tecnologia da informação moderna para redesenhar radicalmente nossos processos de negócio para obter melhorias significativas no desempenho.

O alvoroço em torno da reengenharia diminuiu, mas o processo em si continua em empresas grandes e pequenas. A conexão entre reengenharia nos negócios e reengenharia no software baseia-se em uma “visão de sistema”.

Ao passo que os gerentes trabalham para modificar as regras do negócio para obter maior eficiência e competitividade, o software deve acompanhar o ritmo. Em alguns casos, isso significa a criação de grandes sistemas baseados em computador,² mas, em outros, significa a modificação ou reforma de aplicações existentes.

Nas próximas seções, examinaremos a reengenharia de maneira descendente (*top-down*), começando com uma rápida visão geral da reengenharia de processos de negócio e passando para uma discussão mais detalhada das atividades técnicas que ocorrem quando o software sofre reengenharia.

“Encarar o amanhã pensando em usar métodos de ontem é ver a vida estagnada.”

James Bell

36.4 Reengenharia de processos de negócio

A reengenharia de processos de negócio (BPR, business process reengineering) algumas vezes vai muito além do escopo das tecnologias de informação e da engenharia de software. Entre as muitas definições (um tanto abstratas) sugeridas para a BPR, destaca-se a publicada na revista *Fortune* [Ste93]: “A busca e a implementação de mudanças radicais nos processos de negócio para obter resultados excelentes”. E como é feita a busca e como é obtida a implementação? Mais importante ainda, como podemos assegurar que a “mudança radical” sugerida levará de fato a “resultados inovadores”, e não ao caos organizacional?

A reengenharia de processos de negócio (BPR) muitas vezes resulta em nova funcionalidade de software, enquanto a reengenharia de software trabalha para substituir funcionalidade existente por um software melhor e mais fácil de manter.

36.4.1 Processos de negócio

Um processo de negócio é “um conjunto de tarefas relacionadas de maneira lógica, desempenhadas para obter um resultado de negócio definido” [Dav90]. No processo de negócio, pessoas, equipamentos, recursos materiais e procedimentos são combinados para produzir um resultado específico. Exemplos de processos de negócio incluem projeto de um novo produto, compra de serviços e suprimentos, contratação de funcionários e pagamento dos fornecedores. Cada um desses demanda uma série de tarefas e cada um utiliza diversos recursos na empresa.

Todo processo de negócio tem um cliente definido – uma pessoa ou grupo que recebe o resultado (por exemplo, uma ideia, um relatório, um projeto, um serviço, um produto). Além disso, os processos de negócio cruzam as fronteiras da organização. Exigem que diferentes grupos organizacionais participem das “tarefas logicamente relacionadas” que definem o processo.

Como engenheiro de software, seu trabalho ocorre na base dessa hierarquia. Porém, você deve se certificar de que alguém tenha dado séria atenção ao nível acima. Se isso não foi feito, seu trabalho está em risco.

² A explosão de aplicações e sistemas baseados na Web e aplicativos móveis é uma indicação dessa tendência.

Cada sistema é, na realidade, uma hierarquia de subsistemas. Um negócio não é exceção. De forma geral, o negócio é segmentado da seguinte maneira:

O negócio → sistemas de negócio → processos de negócio → subprocessos de negócio

Cada sistema de negócio (também chamado de *funções de negócio*) é composto de um ou mais processos, e cada processo de negócio é definido por uma série de subprocessos.

A BPR pode ser aplicada a qualquer nível da hierarquia, mas, à medida que o escopo da BPR se amplia (conforme subimos na hierarquia), os riscos associados a ela crescem significativamente. Por essa razão, muitos esforços de BPR concentram-se em processos individuais e subprocessos.

36.4.2 Um modelo de BPR

Como a maioria das atividades de engenharia, a reengenharia dos processos de negócio é iterativa. As metas de negócio e os processos para atingi-las devem ser adaptados a um ambiente de negócio em mutação. Por essa razão, não há um início e um fim para a BPR – é um processo evolucionário. Na Figura 36.1 um modelo para reengenharia de processo de negócio é apresentado. O modelo define seis atividades:

1. **Definição do negócio.** As metas do negócio são identificadas no contexto dos quatro motivadores principais: redução do custo, redução do tempo, melhoria da qualidade e desenvolvimento pessoal. As metas podem ser definidas no nível do negócio ou para um componente específico do negócio.
2. **Identificação do processo.** São identificados os processos críticos para atingir as metas estabelecidas na definição do negócio. Eles podem, então, ser classificados por importância, necessidade de mudança ou de qualquer outra maneira apropriada para a reengenharia.

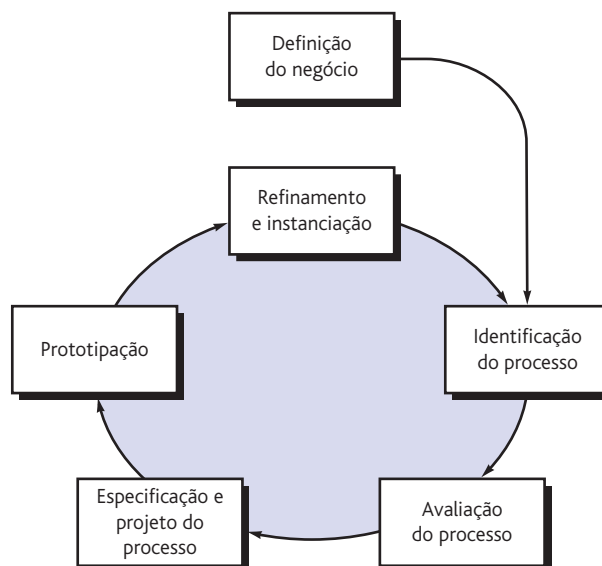


FIGURA 36.1 Um modelo de BPR.

3. **Avaliação do processo.** O processo existente é amplamente analisado e medido. Identificam-se as tarefas do processo, registram-se os custos e o tempo consumido por elas e isolam-se os problemas de qualidade/desempenho.
4. **Especificação e projeto do processo.** Com base nas informações obtidas durante as três primeiras atividades da BPR, são preparados casos de uso (Capítulos 8 e 9) para cada processo que deve ser reprojetoado. No contexto da BPR, os casos de uso identificam um cenário que fornece algum resultado a um cliente. Com o caso de uso como especificação do processo, um novo conjunto de tarefas é projetado para o processo.
5. **Prototipação.** Um processo de negócio reprojetoado deve passar por um protótipo antes de ser totalmente integrado aos negócios. Essa atividade “testa” o processo de modo que possam ser feitos os refinamentos.
6. **Refinamento e instanciação.** Com base nas informações obtidas do protótipo, o processo do negócio é refinado e instanciado em um sistema de negócio.

“Tão logo identificamos algo conhecido em uma coisa nova, nos tranquilizamos.”

F. W. Nietzsche

Essas atividades de BPR às vezes são utilizadas com ferramentas de análise de fluxo de trabalho. A finalidade das ferramentas é criar um modelo do fluxo de trabalho existente, na tentativa de melhor analisar os processos.³

FERRAMENTAS DO SOFTWARE



Reengenharia de processos de negócio (BPR)

Objetivo: O objetivo das ferramentas de BPR é auxiliar a análise e a avaliação de processos de negócio existentes e a especificação de projetos de novos.

Mecanismos: A mecânica das ferramentas varia. Em geral, as ferramentas de BPR permitem que um analista modele os processos de negócio existentes, em um esforço para avaliar as ineficiências do fluxo de trabalho ou problemas funcionais. Uma vez identificados os problemas, existem ferramentas que permitem a análise do protótipo e/ou simulação de processos de negócio revisados.

Ferramentas representativas:³

ExtendSim, desenvolvida pela ImagineThat (www.imagine-thatinc.com), é uma ferramenta de simulação para modelar processos existentes e explorar novos processos. A ferramenta Extend fornece um recurso amplo do tipo “e se” que permite a um analista de negócio explorar diferentes cenários de processo.

Metastrom BPM, desenvolvida pela Opentext (<http://bps.opentext.com/>), fornece suporte para gerenciamento de processos de negócio tanto manuais quanto automáticos.

IceTools, desenvolvida pela Blue Ice (<http://www.icetools.com/home.html>), é uma coleção de modelos de BPR para Microsoft Office e Microsoft Visio.

OMNIBUS, desenvolvida pela Kovair (<http://www.kovair.com/>), é uma das muitas ferramentas que permitem a uma organização modelar o fluxo de trabalho de processo (neste caso, fluxo de trabalho de TI).

ProcessMaker, conjunto de fluxo de trabalho de código aberto, desenvolvido pela Colosa (<http://www.process-maker.com/>), incorpora um conjunto de ferramentas para modelagem, simulação e cronograma de fluxo de trabalho.

Uma boa lista de links para ferramentas de BPR pode ser encontrada em www.opfro.org/index.html?Components/Producers/Tools/BusinessProcessReengineeringTools.html~Contents.

³ A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

36.5 Reengenharia de software

O cenário é muito comum. Uma aplicação atendeu às necessidades de negócio de uma empresa por 10 ou 15 anos. Durante esse tempo, ela foi corrigida, adaptada e aperfeiçoada muitas vezes. Profissionais realizaram esse trabalho com as melhores intenções, mas as boas práticas de engenharia de software foram sempre deixadas de lado (devido à urgência de outros aspectos). Agora a aplicação está instável. Ainda funciona, mas sempre que se tenta fazer uma alteração, ocorrem efeitos colaterais sérios e inesperados. No entanto, a aplicação deve continuar evoluindo. O que fazer?

Software que não pode ser mantido não é um problema novo. Na verdade, a ênfase cada vez maior na reengenharia de software foi gerada pelos problemas de manutenção de quase meio século.

36.5.1 Um modelo de processo de reengenharia de software

Reengenharia toma tempo, tem um custo financeiro alto e absorve recursos que poderiam ser usados em necessidades mais imediatas. Por essas razões, a reengenharia não é realizada em alguns meses ou mesmo anos; a reengenharia dos sistemas de informação absorverá recursos da tecnologia da informação por muitos anos. Todas as empresas precisam de uma estratégia pragmática para a reengenharia de software.

Uma estratégia viável faz parte de um modelo de processo de reengenharia. Discutiremos o modelo mais adiante nesta seção; primeiro, vejamos alguns princípios básicos.

A reengenharia é um trabalho de reforma. Para melhor entendê-la, considere uma atividade análoga: a reforma de uma casa. Imagine a seguinte situação. Você acaba de comprar uma casa em outro estado, mas nunca viu o imóvel. Comprou-o por um preço extremamente baixo, tendo sido alertado de que a casa poderia precisar de uma reforma completa. Como você agiria?

- Antes de iniciar a reforma, seria bom inspecionar a casa. Para determinar se ela precisa de reforma completa, você (ou um profissional de construção) criaria uma lista de critérios para que a inspeção fosse sistemática.
- Antes de colocar a casa abaixo, você verificaria se a estrutura está boa. Se a estrutura estivesse em bom estado, talvez fosse possível “remodelar” sem reformar (a um custo bem mais baixo e em menos tempo).
- Antes de começar a reforma, você procuraria entender como a casa foi construída. Daria uma olhada entre as paredes. Verificaria a fiação elétrica, a tubulação hidráulica e as partes internas da estrutura. Mesmo que você resolvesse descartar tudo, as informações obtidas teriam utilidade quando iniciasse a reconstrução.
- Na reforma, usaria somente os materiais mais modernos e mais duráveis. Isso poderia custar um pouco mais agora, mas ajudaria a evitar uma manutenção cara e demorada mais tarde.
- Se você decidisse reformar, seria disciplinado. Usaria práticas que resultariam na mais alta qualidade – hoje e no futuro.

Uma excelente fonte de informações sobre reengenharia de software pode ser encontrada em reengineer.org.

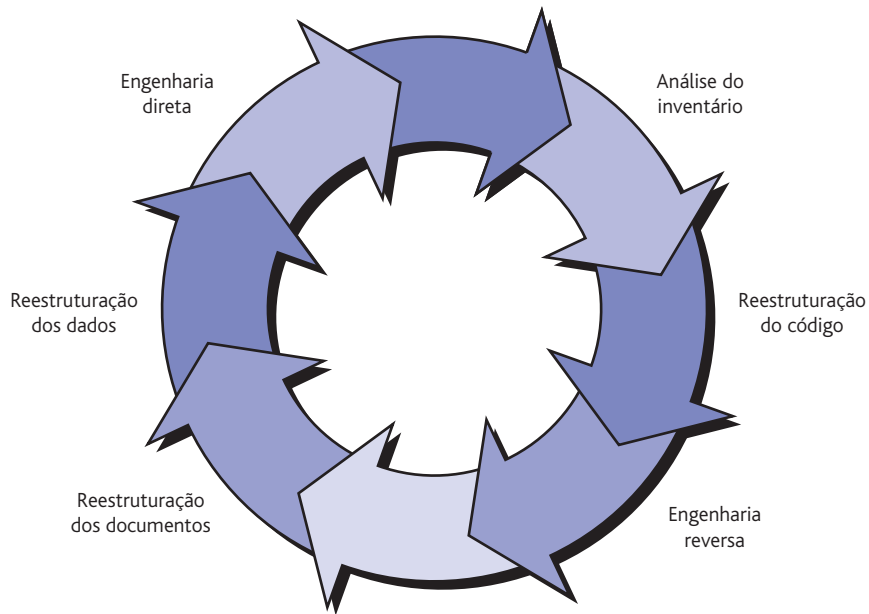


FIGURA 36.2 Um modelo de processo de reengenharia de software.

Embora esses princípios concentrem-se na reforma de uma casa, eles se aplicam igualmente bem à reengenharia de sistemas e aplicações baseados em computador.

Para implementar esses princípios, podemos usar um modelo de processo de reengenharia de software que define seis atividades, apresentado na Figura 36.2. Em alguns casos, essas atividades ocorrem em sequência linear, mas nem sempre isso acontece. Por exemplo, pode acontecer de a engenharia reversa (entendimento do funcionamento interno de um programa) ter de ocorrer antes do início da reestruturação dos documentos.

36.5.2 Atividades de reengenharia de software

O paradigma da reengenharia mostrado na Figura 36.2 é um modelo cíclico. Ou seja, cada uma das atividades apresentadas como parte do paradigma pode ser revisitada. Para qualquer ciclo, o processo pode terminar após qualquer uma dessas atividades.

Análise de inventário. Toda organização de software deve ter um inventário de todas as aplicações. O inventário pode ser nada mais do que uma planilha com informações detalhadas (por exemplo, tamanho, idade, criticalidade nos negócios) para cada aplicação ativa. Ordenando essas informações de acordo com a criticalidade de negócio, longevidade, manutenibilidade atual e suportabilidade e outros critérios localmente importantes, surgem os candidatos à reengenharia. Recursos podem, então, ser alocados às aplicações candidatas ao trabalho de reengenharia.

É importante observar que o inventário deverá ser revisto regularmente. O status das aplicações (por exemplo, criticalidade de negócio) pode mudar em função do tempo e, como resultado, as prioridades para a reengenharia também podem mudar.

Se o tempo e os recursos forem escassos, você pode aplicar o princípio de Pareto ao software que vai passar por reengenharia. Aplique o processo de reengenharia aos 20% do software, responsáveis por 80% dos problemas.

Crie apenas a documentação necessária para entender melhor o software, nem uma página a mais.

Reestruturação dos documentos. Documentação deficiente é a marca registrada de muitos sistemas legados. O que se pode fazer quanto a isso? Quais são as suas opções? Em alguns casos, criar documentação quando não existe nenhuma é simplesmente dispendioso demais. Se o software funciona, não toque nele! Em outros casos, alguma documentação deve ser criada, mas somente quando forem feitas alterações. Se ocorrer uma modificação, documente-a. Por fim, existem situações nas quais um sistema fundamental deve ser totalmente documentado, mas, mesmo assim, os documentos devem atingir um mínimo essencial. Sua organização de software deve escolher a opção de documentação mais apropriada para cada caso.

Engenharia reversa. A engenharia reversa para software é o processo de analisar um programa na tentativa de criar uma representação dele em um nível mais alto de abstração do que o código-fonte. A engenharia reversa é um processo de *recuperação do projeto*. As ferramentas de engenharia reversa extraem informações do projeto de dados, da arquitetura e procedural com base em um programa existente.

Reestruturação do código. O tipo mais comum de reengenharia (atualmente, o uso do termo reengenharia é questionável neste caso) é a *reestruturação de código*.⁴ Alguns sistemas legados têm uma arquitetura de programa razoavelmente sólida, mas os módulos individuais foram codificados de uma maneira que dificulta entendê-los, testá-los e mantê-los. Nesses casos, o código dentro dos módulos suspeitos pode ser reestruturado.

Nessa atividade, o código-fonte é analisado com uma ferramenta de reestruturação. As violações das construções de programação estruturada são registradas, e o código é, então, reestruturado (isso pode ser feito automaticamente) ou até mesmo reescrito em uma linguagem de programação mais moderna. O código reestruturado resultante é revisado e testado para garantir que não tenham sido introduzidas anomalias. A documentação interna do código é atualizada.

Reestruturação dos dados. Um programa com uma arquitetura de dados fraca será difícil de adaptar e melhorar. Na verdade, para muitas aplicações, a arquitetura das informações tem mais a ver com a viabilidade de longo prazo de um programa do que o próprio código-fonte.

Diferentemente da reestruturação de código, que ocorre em um nível de abstração relativamente baixo, a reestruturação de dados é uma atividade de reengenharia completa. Em muitos casos, a reestruturação dos dados começa com uma atividade de engenharia reversa. A arquitetura de dados atual é dissecada, e os modelos de dados necessários são definidos. Identificam-se os objetos de dados e atributos, e a qualidade das estruturas de dados existentes é revisada.

Quando a estrutura de dados é fraca (por exemplo, estão implementados arquivos de texto simples, quando uma abordagem relacional simplificaria muito o processamento), os dados passam por reengenharia.

⁴ A reestruturação de código tem alguns dos elementos de “refatoração”, um conceito de redesign introduzido no Capítulo 12 e discutido em outras partes deste livro.

Como a arquitetura de dados tem forte influência sobre a arquitetura do programa e os algoritmos que a constituem, mudanças nos dados resultarão, invariavelmente, em mudanças de arquitetura ou em nível de código.

Engenharia direta. Em um mundo ideal, as aplicações seriam recriadas por meio de um “motor de reengenharia” automatizado. O programa antigo seria colocado nesse motor, analisado, reestruturado e regenerado em uma forma que apresentasse os melhores aspectos da qualidade do software. Resumindo, é bastante improvável que tal “motor” apareça algum dia, mas os fabricantes de software introduziram ferramentas que fornecem um subconjunto limitado desses recursos que se destinam a domínios de aplicação específicos (por exemplo, aplicações implementadas por meio de um sistema de banco de dados específico). Mais importante, essas ferramentas de reengenharia estão se tornando cada vez mais sofisticadas.

A engenharia direta não apenas recupera as informações do projeto do software existente, como também as utiliza para alterar ou reconstituir o sistema, em um esforço para melhorar sua qualidade geral. Em muitos casos, o software que passou pela reengenharia recria a função do sistema existente e acrescenta novas funções e/ou melhora o desempenho geral.

36.6 Engenharia reversa

A engenharia reversa evoca uma imagem do “*slot* mágico”. Você coloca no *slot* uma listagem de código não documentada, criada de qualquer jeito, e do outro lado sai uma descrição completa do projeto (e uma documentação completa) para o programa de computador. Infelizmente, o *slot* mágico não existe. A engenharia reversa pode extrair informações do projeto a partir do código-fonte, mas o nível de abstração, a completude da documentação, o grau segundo o qual as ferramentas e um analista humano trabalham juntos e a direcionalidade do processo são altamente variáveis.

O *nível de abstração* de um processo de engenharia reversa e as ferramentas utilizadas para realizá-lo refere-se à sofisticação das informações de projeto que podem ser extraídas do código-fonte. Idealmente, o nível de abstração deve ser o mais alto possível. Isto é, o processo de engenharia reversa deve ser capaz de derivar representações de projeto procedural (em uma abstração de baixo nível), informações de programa e de estrutura de dados (em um nível de abstração um tanto mais alto), modelos de objeto, dados e/ou modelos de fluxo de controle (em um nível de abstração relativamente alto) e modelos de dados (em um nível alto de abstração). Conforme o nível de abstração aumenta, você recebe informações que permitirão entender o programa mais facilmente.

A *completude* de um processo de engenharia reversa refere-se ao nível de detalhe fornecido em um nível de abstração. Em muitos casos, a completude diminui conforme o nível de abstração aumenta. Por exemplo, dada uma listagem de código-fonte, é relativamente fácil desenvolver uma representação completa do projeto procedural. Podem ser derivadas também representações simples do projeto arquitetural, mas é muito mais difícil desenvolver um conjunto completo de diagramas UML ou modelos.

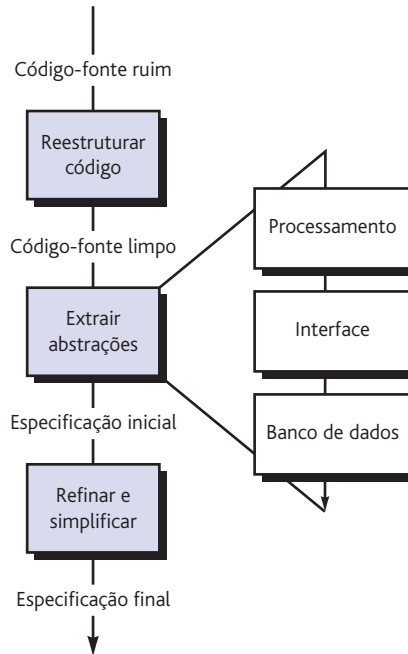


FIGURA 36.3 O processo de engenharia reversa.

Três problemas de engenharia reversa devem ser tratados: nível de abstração, completude e direcionalidade.

A completude melhora em proporção direta com a quantidade de análise realizada pela pessoa que está fazendo a engenharia reversa. *Interatividade* refere-se ao grau segundo o qual as pessoas são “integradas” às ferramentas automáticas para criar um processo eficaz de engenharia reversa. Em muitos casos, à medida que o nível de abstração aumenta, a interatividade deve aumentar para que a completude não seja prejudicada.

Se a *direcionalidade* do processo de engenharia reversa for unidirecional, todas as informações extraídas do código-fonte serão fornecidas ao engenheiro de software, que pode então usá-las durante qualquer atividade de manutenção. Se a direcionalidade for bidirecional, as informações serão colocadas em uma ferramenta de reengenharia que tenta reestruturar ou regenerar o sistema antigo.

O processo de engenharia reversa está representado na Figura 36.3. Antes de começar as atividades de engenharia reversa, o código-fonte não estruturado (“ruim”) é reestruturado (Seção 36.5.1) para que contenha somente as construções de programação estruturada.⁵ Isso torna o código-fonte mais fácil de ler e proporciona a base para todas as atividades subsequentes de engenharia reversa.

O centro da engenharia reversa é uma atividade chamada *extrair abstrações*. Você deve avaliar o programa antigo e, com base no código-fonte (muitas vezes não documentado), desenvolver uma especificação significativa do processamento executado, da interface de usuário aplicada e das estruturas de dados de programa ou do banco de dados utilizadas.

Recursos úteis para “recuperação do projeto e entendimento do programa” podem ser encontrados em http://www.softpanorama.net/SE/reverse_engineering_links.shtml.

⁵ O código pode ser reestruturado usando um *motor de reestruturação* – uma ferramenta que reestrutura código-fonte.

36.6.1 Engenharia reversa para entender os dados

A engenharia reversa dos dados ocorre em diferentes níveis de abstração; frequentemente, é a primeira tarefa da reengenharia. Em nível de programa, as estruturas internas de dados de programa devem muitas vezes passar por engenharia reversa como parte de um trabalho de reengenharia total. Em nível de sistema, estruturas de dados globais (por exemplo, arquivos, bancos de dados) passam muitas vezes por reengenharia para acomodar novos paradigmas de gerenciamento de banco de dados (por exemplo, a mudança de arquivos de texto para sistemas de bancos de dados relacionais ou orientados a objetos). A engenharia reversa das estruturas de dados globais atuais define o cenário para a introdução de um novo banco de dados para todo o sistema.

Estruturas de dados internas. As técnicas de engenharia reversa para dados internos de programa focam-se na definição de classes de objetos. Isso é feito examinando-se o código do programa para agrupar variáveis de programa relacionadas. Em muitos casos, a organização dos dados dentro do código identifica tipos de dados abstratos. Por exemplo, estruturas de registro, arquivos, listas e outras estruturas de dados muitas vezes fornecem um indicador inicial das classes.

Estrutura de banco de dados. Seja qual for sua organização lógica e estrutura física, um banco de dados permite a definição de objetos de dados e suporta algum método para estabelecer relações entre os objetos. Portanto, para fazer a reengenharia de um esquema de banco de dados para outro é necessário entender os objetos e suas relações.

Os passos a seguir [Pre94] podem ser usados para definir o modelo de dados existente, como um precursor para a reengenharia de um novo modelo de banco de dados: (1) criar um modelo do objeto inicial, (2) determinar chaves candidatas (são examinados os atributos para determinar se elas são usadas para apontar para outro registro ou tabela; aquelas que servem como ponteiros tornam-se chaves candidatas), (3) refinar as classes provisórias, (4) definir generalizações e (5) descobrir associações usando técnicas análogas à abordagem CRC. Uma vez conhecidas as informações definidas nos passos anteriores, pode ser aplicada uma série de transformações [Pre94] para mapear a estrutura de banco de dados antiga em uma nova estrutura de banco de dados.

Em alguns casos, a primeira atividade de reengenharia tenta construir um diagrama de classes UML.

A abordagem de engenharia reversa de dados para software convencional segue um caminho análogo: (1) criar um modelo de dados, (2) identificar atributos dos objetos de dados e (3) definir relações.

36.6.2 Engenharia reversa para entender o processamento

A engenharia reversa para entender o processamento começa com um esforço de entender e extrair abstrações procedurais representadas pelo código-fonte. Para entender as abstrações procedurais, o código é analisado em vários níveis de abstração: sistema, programa, componente, padrão e instruções.

A funcionalidade global de toda a aplicação deve ser entendida antes do trabalho detalhado de engenharia reversa. Isso estabelece um contexto para análise posterior e dá uma ideia dos problemas de interoperabilidade entre as aplicações de um sistema maior. Cada um dos programas que formam o sistema representa uma abstração funcional em um alto nível de detalhe. É criado um diagrama de blocos, representando a interação entre essas abstrações funcionais. Cada componente executa alguma subfunção e representa uma abstração procedural definida. É desenvolvida uma narrativa de processamento

"Existe uma paixão pela compreensão, assim como uma paixão pela música. Essa paixão é muito comum nas crianças, mas em muitas pessoas acaba se perdendo mais tarde."

Albert Einstein

para cada componente. Em algumas situações, já existem especificações de sistema, programa e componente. Quando esse é o caso, as especificações são revistas para verificar a conformidade com o código existente.⁶

A situação torna-se mais complexa quando o código é considerado dentro de um componente. Você deve procurar seções de código que representem padrões procedurais genéricos. Em quase todos os componentes, uma seção do código prepara dados para processamento (dentro do módulo), outra seção faz o processamento e outra prepara os resultados do processamento para exportação do componente. Em cada uma dessas seções, podem ser encontrados padrões menores; por exemplo, validação de dados e verificações de limites, que muitas vezes ocorrem na seção do código que prepara os dados para processamento.

Para sistemas grandes, a engenharia reversa em geral é feita usando-se uma abordagem semiautomática. Podem ser empregadas ferramentas automatizadas para ajudá-lo a entender a semântica do código existente. O resultado desse processo é, então, passado para as ferramentas de reestruturação e engenharia direta para completar o processo de reengenharia.

36.6.3 Engenharia reversa das interfaces de usuário

Interfaces gráficas do usuário sofisticadas tornaram-se itens obrigatórios para produtos e sistemas de todos os tipos baseados em computadores. Portanto, o desenvolvimento de interfaces do usuário tornou-se um dos tipos mais comuns de atividade de reengenharia. Antes de recriar uma interface de usuário, porém, a engenharia reversa deverá ser feita.

Para entender completamente uma interface de usuário, a estrutura e o comportamento da interface devem ser especificados. Merlo e seus colegas [Mer93] sugerem três questões básicas que devem ser respondidas quando se começa a engenharia reversa de uma interface do usuário (UI):

- Quais são as ações básicas (por exemplo, teclas e cliques de mouse) que a interface deve processar?
- Qual é a descrição compacta da resposta comportamental do sistema para essas ações?
- O que significa “substituição” ou, mais precisamente, que conceito de equivalência de interfaces é relevante aqui?

A notação de modelagem comportamental (Capítulo 11) pode proporcionar um meio para desenvolver respostas para as duas primeiras questões. Grande parte das informações necessárias para criar um modelo comportamental pode ser obtida observando-se a manifestação externa da interface. Mas as informações adicionais necessárias para criar o modelo comportamental devem ser extraídas do código.

É importante notar que uma interface gráfica de usuário substituta pode não refletir exatamente a interface antiga (na verdade, pode ser radicalmente diferente). Muitas vezes compensa desenvolver uma nova metáfora de interação. Por exemplo, uma interface de usuário antiga exige que o usuário forneça

Como podemos entender o funcionamento de uma interface de usuário existente?

⁶ Frequentemente, especificações escritas no início da vida do programa não são atualizadas. À medida que são feitas as atualizações, o código não está mais em conformidade com a especificação.

um fator de escala (variando de 1 a 10) para reduzir ou ampliar uma imagem gráfica. Uma interface gráfica de usuário que passou por reengenharia pode usar uma barra de rolagem *touch-screen* para executar a mesma função.⁷

FERRAMENTAS DO SOFTWARE



Engenharia reversa

Objetivo: Ajudar os engenheiros de software a entender a estrutura de projeto interna de programas complexos.

Mecanismos: Em muitos casos, as ferramentas de engenharia reversa aceitam código-fonte como entrada e produzem uma variedade de representações de projeto estrutural, procedural, de dados e comportamental.

Ferramentas representativas:⁷

Imagix 4D, desenvolvida pela Imagix (www.imagix.com), "ajuda os desenvolvedores de software a entender

software em C e C++ complexo ou legado" pela engenharia reversa e a documentação do código-fonte.

Understand, desenvolvida pela Scientific Toolworks (www.scitools.com), analisa Ada, Fortran, C, C++, C#, PHP, HTML, JavaScript, Python e Java "para fazer a engenharia reversa, documentar automaticamente, calcular métricas de código e ajudá-lo a entender, navegar e manter o código-fonte".

Uma lista de ferramentas de engenharia reversa pode ser encontrada em <http://www.eclipse.org/gmt/modisco/relatedProjects.php>.

36.7 Reestruturação

A reestruturação de software modifica o código-fonte e/ou os dados para torná-lo mais amigável para futuras alterações. Em geral, a reestruturação não modifica a arquitetura geral do programa; ela tende a se concentrar nos detalhes de projeto dos módulos e nas estruturas de dados locais definidas nos módulos. Se a reestruturação vai além dos limites dos módulos e abrange a arquitetura do software, ela passa a ser engenharia direta (Seção 36.8).

A reestruturação ocorre quando a arquitetura básica de uma aplicação é sólida, mesmo que as partes técnicas internas necessitem de retrabalho. Ela ocorre quando partes importantes do software são reparáveis e somente um subconjunto de todos os módulos e dados necessita de uma modificação mais extensa.⁸

36.7.1 Reestruturação de código

A *reestruturação de código* é feita para gerar um projeto que produz a mesma função, mas com mais qualidade do que o programa original. Em geral, as técnicas de reestruturação de código (por exemplo, as técnicas de simplificação lógica de Warnier [War74]) modelam a lógica de programação usando álgebra booleana e aplicam uma série de regras de transformação que resulta em lógica reestruturada. O objetivo é pegar um código "emaranhado" e dele derivar um projeto procedural que esteja em conformidade com a filosofia de programação estruturada (Capítulo 19).

Embora a reestruturação de código possa aliviar problemas imediatos associados à depuração ou a pequenas alterações, ela não é a reengenharia. O verdadeiro benefício é obtido somente quando os dados e a arquitetura são reestruturados.

⁷ A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

⁸ Às vezes, é difícil fazer a distinção entre reestruturação extensiva e redesenvolvimento. Ambos os casos são reengenharia.

Outras técnicas de reestruturação também foram propostas para uso com as ferramentas de reengenharia. Um diagrama de intercâmbio de recursos mapeia cada módulo de programa e os recursos (tipos de dados, procedimentos e variáveis) permutados entre ele e outros módulos. Criando representações do fluxo de recursos, a arquitetura do programa pode ser reestruturada para obter o mínimo acoplamento entre os módulos.

36.7.2 Reestruturação de dados

Antes de iniciar a reestruturação de dados, deve ser feita uma atividade de engenharia reversa chamada *análise do código-fonte*. São avaliadas todas as instruções da linguagem de programação que contenham definições de dados, descrições de arquivo, E/S (I/O) e descrições de interface. A finalidade é extrair itens de dados e objetos para obter informações sobre fluxo de dados e para entender as estruturas de dados existentes que precisam ser implementadas. Essa atividade às vezes é chamada de *análise de dados*.⁹

Uma vez concluída a análise de dados, inicia-se o *reprojeto dos dados*. Em sua forma mais simples, uma etapa de *padronização de registro de dados* esclarece as definições de dados para obter consistência entre nomes de itens de dados ou formatos de registros físicos em uma estrutura de dados ou formato de arquivo existente. Outra forma de reprojeto, chamada de *racionalização de nomes de dados*, garante que todas as convenções de nomes de dados estejam de acordo com os padrões locais e que os pseudônimos (*aliases*) sejam eliminados à medida que os dados fluem pelo sistema.

Quando a reestruturação vai além da padronização e da racionalização, são feitas modificações físicas nas estruturas de dados existentes para tornar

FERRAMENTAS DO SOFTWARE



Reestruturação de software

Objetivo: O objetivo das ferramentas de reestruturação é transformar software não estruturado e mais antigo em linguagens de programação e estruturas de projeto modernas.

Mecanismos: Em geral, o código-fonte é o elemento de entrada e ele é transformado em um programa mais bem estruturado. Em alguns casos, a transformação ocorre na mesma linguagem de programação. Em outros, uma linguagem de programação mais antiga é transformada em uma mais moderna.

Ferramentas representativas:⁹

DMS Software Reengineering Toolkit, desenvolvida pela Semantic Design (www.semdesigns.com), fornece uma variedade de recursos de reestruturação para COBOL, C/C++, Java, Fortran 90 e VHDL.

Clone Doctor, desenvolvida pela Semantic Designs (www.semdesigns.com), analisa e transforma programas escritos em C, C++, Java ou COBOL ou qualquer outra linguagem de programação de computador baseada em texto.

plusFORT, desenvolvida pela Polyhedron (www.polyhedron.com), é um conjunto de ferramentas FORTRAN contendo recursos para reestruturar programas FORTRAN mal projetados em programas modernos padrão FORTRAN ou C.

Links para uma variedade de ferramentas de reengenharia e engenharia reversa podem ser encontrados em <http://www.comp.lancs.ac.uk/projects/renaissance/RenaissanceWeb/Reengineering/Tools.html> e <http://www.fujaba.de/projects.html>.

⁹ A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nessa categoria. Na maioria dos casos, seus nomes são marcas registradas pelos respectivos desenvolvedores.

o projeto de dados mais eficaz. Isso pode significar a transformação de um formato de arquivo em outro ou, em alguns casos, a transformação de um tipo de banco de dados em outro.

36.8 Engenharia direta

Um programa com um fluxo de controle que equivale graficamente a um prato de espaguete, com módulos de 2 mil instruções, algumas poucas linhas de comentários úteis em 290 mil instruções de código-fonte e nenhuma outra documentação, deve ser modificado para acomodar alterações e requisitos de usuário. As opções são as seguintes:

1. Você pode trabalhar arduamente, fazendo modificação após modificação, enfrentando problemas de projeto improvisado e de código-fonte confuso para implementar as mudanças necessárias.
2. Você pode tentar entender os detalhes internos do programa em um esforço para tornar as modificações mais eficazes.
3. Você pode reprojeter, recodificar e testar as partes do software que exigem modificação, aplicando uma abordagem de engenharia de software a todos os segmentos revisados.
4. Você pode reprojeter completamente, recodificar e testar o programa usando ferramentas de reengenharia para nos ajudar a entender o projeto atual.

Que opções existem quando encontramos um programa mal projetado e mal implementado?

Não há uma opção “correta”. As circunstâncias podem recomendar a primeira opção, mesmo que as outras sejam mais desejáveis.

Em vez de esperar até que a manutenção seja solicitada, a organização de desenvolvimento ou suporte usa os resultados da análise de inventário para selecionar um programa que (1) permanecerá em uso por certo número de anos, (2) está sendo utilizado no momento com sucesso e (3) pode passar por modificações importantes ou melhoramentos em futuro próximo. Então é aplicada a opção 2, 3 ou 4.

Em uma primeira impressão, a sugestão para que você redesenvolva um programa grande quando uma versão ainda funciona pode ser bastante extravagante. Mas antes de julgar o trabalho, considere os seguintes argumentos: o custo para manter uma linha de código-fonte pode ser de 20 a 40 vezes o do desenvolvimento inicial daquela linha. Além disso, o reprojeto da arquitetura de software (programa e/ou estrutura de dados) usando conceitos modernos de projeto pode facilitar muito a manutenção futura. Como já existe um protótipo do software, a produtividade do desenvolvimento deverá ser muito mais alta do que a média. O usuário agora tem experiência com o software. Portanto, novos requisitos e a direção das mudanças podem ser definidos com grande facilidade. Ferramentas automatizadas de reengenharia facilitarão algumas partes do trabalho. E, por último, existirá uma configuração de software completa (documentos, programas e dados) depois de uma manutenção preventiva.

Um grande departamento de desenvolvimento de software interno (por exemplo, um grupo de desenvolvimento de sistemas comerciais para uma grande empresa de produtos de consumo) pode ter de 500 a 2 mil programas

A reengenharia é bem semelhante à limpeza de seus dentes. Você pode pensar em mil razões para adiar, e conseguirá um jeito de retardar por um tempo. Mas, no fim, sua tática de adiamento se voltará contra você, causando-lhe dor.

em produção sob sua responsabilidade. Esses programas podem ser classificados por importância e examinados como candidatos para engenharia direta.

Em muitos casos, a engenharia direta não cria só um equivalente moderno de um programa antigo. Em vez disso, novos requisitos de usuário e tecnologia são integrados ao trabalho de reengenharia. O programa redesenvolvido amplia a capacidade da aplicação antiga.

36.8.1 Engenharia direta para arquiteturas cliente-servidor

Nas últimas décadas, recursos de computação centralizados (incluindo software) foram distribuídos entre muitas plataformas clientes. Embora diferentes ambientes distribuídos possam ser projetados, a aplicação centralizada típica que passou por reengenharia para se transformar em uma arquitetura cliente-servidor tem estas características: a funcionalidade da aplicação migra para cada computador cliente; novas interfaces gráficas de usuário são implementadas nas instalações do cliente; funções de banco de dados são atribuídas ao servidor; funcionalidade especializada (por exemplo, análise que demanda muitos cálculos) podem permanecer no servidor; e novos requisitos de comunicação, segurança, arquivamento e controle devem ser estabelecidos tanto no lado do cliente quanto no do servidor. É importante observar que a migração da computação centralizada para cliente-servidor exige reengenharia, tanto de negócio quanto de software.

A reengenharia para aplicações cliente-servidor começa com uma análise completa do ambiente comercial, que abrange o mainframe existente. Três leis de abstração podem ser identificadas. O *banco de dados* estabelece a base de uma arquitetura cliente-servidor e gerencia transações e consultas das aplicações do servidor. No entanto, essas transações e consultas devem ser controladas com base em um conjunto de regras de negócio (definidas por um processo de negócio existente ou que passou por reengenharia). Aplicações cliente fornecem funcionalidade destinada à comunidade de usuários.

As funções do sistema de gerenciamento de banco de dados e a arquitetura de dados do banco de dados existentes devem passar por engenharia reversa antes do reprojeto da camada fundamental do banco de dados. O banco de dados cliente-servidor passa por reengenharia para garantir que as transações sejam executadas consistentemente, que todas as atualizações sejam executadas somente por usuários autorizados, que as regras de negócio básicas sejam seguidas (por exemplo, antes que o registro de um fornecedor seja excluído, o servidor assegura que não existem contas a pagar, contratos, ou comunicações para aquele fornecedor), que as consultas possam ser acomodadas eficientemente e que seja estabelecida uma capacidade total de arquivamento.

A camada das regras de negócio representa o software residente tanto no cliente quanto no servidor. Esse software executa tarefas de controle e coordenação para assegurar que as transações e consultas entre a aplicação cliente e o banco de dados estejam em conformidade com processos de negócio estabelecidos.

A camada das aplicações clientes implementa funções de negócio exigidas por grupos específicos de usuários. Em muitas ocasiões, uma aplicação centralizada mais antiga é segmentada em uma série de aplicações para desktop menores e modificadas por meio da reengenharia. A comunicação

Em alguns casos, a migração para uma arquitetura cliente-servidor deverá ser considerada não como uma reengenharia, mas como um novo trabalho de desenvolvimento. A reengenharia entra em cena apenas quando uma funcionalidade específica do sistema antigo vai ser integrada em uma nova arquitetura.

entre as aplicações para desktop (quando necessária) é controlada pela camada de regras de negócio.

Uma discussão ampla sobre projeto e reengenharia de software cliente-servidor pode ser encontrada em livros dedicados ao assunto. Se tiver interesse, consulte [Van02], [Cou00], ou [Orf99].

36.8.2 Engenharia direta para arquiteturas orientadas a objetos

A reengenharia de software orientada a objetos tornou-se o paradigma de desenvolvimento adotado por muitas organizações. Mas o que dizer das aplicações desenvolvidas com métodos convencionais? Em alguns casos, a resposta é deixar essas aplicações “como estão”. Em outros, as aplicações mais antigas devem passar por reengenharia para que possam ser facilmente integradas em sistemas orientados a objetos de grande porte.

A reengenharia de software convencional para uma implementação orientada a objetos usa muitas das técnicas discutidas na Parte II deste livro. Primeiro, o software passa por engenharia reversa para que possam ser criados modelos de dados, modelos funcionais e modelos comportamentais apropriados. Se o sistema ao qual se aplica a reengenharia amplia a funcionalidade ou o comportamento da aplicação original, são criados casos de uso (Capítulos 8 e 9). Os modelos de dados desenvolvidos durante a engenharia reversa são então utilizados em conjunto com a modelagem CRC (Capítulo 10) para estabelecer a base para a definição de classes. Hierarquias de classe, modelos de relacionamento de objetos, modelos de comportamento de objeto e subsistemas são definidos e inicia-se o projeto orientado a objetos.

À medida que a engenharia direta orientada a objetos avança da análise para o projeto, um modelo de processo CBSE (Capítulo 10) pode ser invocado. Se a aplicação existente reside dentro de um domínio que já seja constituído por muitas aplicações orientadas a objetos, é provável que exista uma biblioteca de componentes robusta e que possa ser empregada durante a engenharia direta.

Para as classes que devem ser criadas desde o início, talvez seja possível reutilizar algoritmos e estruturas de dados da aplicação convencional existente. No entanto, eles devem ser reprojitados para ficar em conformidade com a estrutura orientada a objetos.

“Você pode gastar um pouco agora ou gastar muito mais tarde.”

Cartaz em uma revendedora de automóveis sugerindo manutenção no carro

36.9 Aspectos econômicos da reengenharia

Em um mundo perfeito, todo programa não manutenível seria aposentado imediatamente para ser substituído por aplicações de alta qualidade, desenvolvidas com modernas práticas de engenharia de software. Mas vivemos em um mundo de recursos limitados; a reengenharia consome recursos que podem ser utilizados para outras finalidades de negócio. Portanto, antes de pensar em fazer a reengenharia de uma aplicação existente, uma empresa deve fazer uma análise de custo-benefício.

Um modelo de análise de custo-benefício para reengenharia foi proposto por Sneed [Sne95]. São definidos nove parâmetros:

P_1 = Custo anual corrente de manutenção para uma aplicação

P_2 = Custo anual corrente das operações para uma aplicação

P_3 = Valor de negócio anual corrente de uma aplicação
 P_4 = Custo de manutenção anual previsto após a reengenharia
 P_5 = Custo anual previsto das operações após a reengenharia
 P_6 = Valor de negócio anual previsto após a reengenharia
 P_7 = Custos estimados da reengenharia
 P_8 = Prazo estimado para fazer a reengenharia
 P_9 = Fator de risco da reengenharia ($P_9 = 1,0$ é nominal)
 L = Expectativa de vida do sistema

O custo associado à manutenção continuada de uma aplicação candidata à reengenharia (a reengenharia ainda não foi feita) pode ser definido como

$$C_{\text{manut}} = [P_3 - (P_1 + P_2)] \times L \quad (36.1)$$

Os custos associados à reengenharia são definidos usando a seguinte relação:

$$C_{\text{reeng}} = P_6 - (P_4 + P_5) \times (L - P_8) - (P_7 \times P_9) \quad (36.2)$$

Usando os custos apresentados nas Equações (36.1) e (36.2), o benefício total da reengenharia pode ser calculado como

$$\text{Custo-benefício} = C_{\text{reeng}} - C_{\text{manut}} \quad (36.3)$$

A análise de custo-benefício apresentada nessas equações pode ser feita para todas as aplicações de alta prioridade identificadas durante a análise de inventário (Seção 36.4.2). As aplicações que apresentam o custo-benefício mais alto podem ser identificadas para reengenharia, enquanto o trabalho com as outras aplicações pode ser adiado até que os recursos estejam disponíveis.

36.10 Resumo

A manutenção e o suporte de software são atividades contínuas que ocorrem em todo o ciclo de vida de uma aplicação. Durante essas atividades, defeitos são corrigidos, aplicações são adaptadas a um ambiente operacional ou de negócio em mudança, melhorias são implementadas por solicitação dos envolvidos e suporte é fornecido aos usuários quando eles integram uma aplicação em seu fluxo de trabalho pessoal ou corporativo.

A reengenharia ocorre em dois níveis de abstração. No nível do negócio, ela concentra-se nos processos de negócio para melhorar a competitividade em alguma área do negócio. No nível do software, a reengenharia examina os sistemas de informação e as aplicações, com a finalidade de reestruturá-las para que tenham melhor qualidade.

A reengenharia de processos de negócio (BPR) define metas de negócio; identifica e avalia processos de negócio existentes (no contexto das metas definidas); especifica e projeta processos revisados; cria protótipos, refina e os viabiliza em um negócio. A BPR tem um foco que se estende além do software. O resultado da BPR é muitas vezes a definição de maneiras pelas quais a tecnologia da informação pode fornecer um melhor suporte ao negócio.

A reengenharia de software engloba uma série de atividades que incluem análise de inventário, reestruturação da documentação, engenharia reversa, reestruturação de programas e dados e engenharia direta. A finalidade dessas atividades é criar versões dos programas existentes que tenham

melhor qualidade e melhor manutenibilidade – programas que serão viáveis para o século 21.

O custo-benefício da reengenharia pode ser determinado quantitativamente. O custo do estado atual, isto é, o custo associado ao suporte e à manutenção continuados de uma aplicação existente é comparado aos custos projetados da reengenharia e à redução resultante nos custos de manutenção e suporte. Em quase todos os casos nos quais um programa tem uma vida longa e em certo momento apresenta manutenibilidade ou suportabilidade ruim, a reengenharia representa uma estratégia de negócio eficiente em termos de custo.

Problemas e pontos a ponderar

36.1 Considere um trabalho que tenha realizado nos últimos cinco anos. Descreva os processos de negócio nos quais você tomou parte. Use o modelo BPR descrito na Seção 36.4.2 para recomendar alterações no processo em um esforço para torná-lo mais eficiente.

36.2 Pesquise sobre a eficiência da reengenharia dos processos de negócio. Apresente argumentos a favor e contra essa abordagem.

36.3 Seu professor selecionará um dos programas que todos na classe tenham desenvolvido durante o curso. Troque o seu programa de forma aleatória com algum outro aluno na classe. Não explique nem dê detalhe sobre o programa. Agora, implemente uma melhoria (especificada por seu professor) no programa que você recebeu.

- Execute todas as tarefas de engenharia de software, incluindo um breve detalhamento (mas não com o autor do programa).
- Mantenha um controle preciso de todos os erros encontrados durante o teste.
- Discuta sua experiência na classe.

36.4 Explore a *checklist* da análise de inventário apresentada no site da SEPA e tente desenvolver um sistema de classificação quantitativa de software que poderia ser aplicado a programas existentes, em um trabalho para escolher programas candidatos para a reengenharia. O seu sistema deve se estender além da análise econômica apresentada na Seção 36.9.

36.5 Sugira alternativas em documentação impressa ou eletrônica que poderiam servir de base para a reestruturação da documentação. (Dica: pense em novas tecnologias descritivas que poderiam ser usadas para mostrar a finalidade do software.)

36.6 Algumas pessoas acreditam que a tecnologia de inteligência artificial aumentará o nível de abstração do processo de engenharia reversa. Pesquise sobre esse assunto, isto é, o uso de inteligência artificial (AI, artificial intelligence) para a engenharia reversa, e escreva um pequeno artigo que destaque esse ponto.

36.7 Por que a completude é difícil de ser atingida à medida que o nível de abstração aumenta?

36.8 Por que a interatividade deve aumentar se a completude tiver de aumentar?

36.9 Usando informações obtidas na Web, apresente para a sua classe características de três ferramentas de engenharia reversa.

36.10 Há uma diferença sutil entre reestruturação e engenharia direta. Qual é?

36.11 Pesquise a literatura e/ou fontes na Internet para encontrar um ou mais artigos que discutem estudos de casos de reengenharia de mainframe para cliente-servidor. Apresente um resumo.

36.12 Como você determinaria P_4 até P_7 no modelo de custo-benefício apresentado na Seção 36.9?

Leituras e fontes de informação complementares

É irônico o fato de a manutenção e o suporte de software representarem as atividades mais dispendiosas na vida útil de uma aplicação e, no entanto, terem sido escritos menos livros sobre manutenção e suporte do que sobre qualquer outro tópico importante de engenharia de software. Dentre as recentes adições à literatura estão os livros de Reifer (*Software Maintenance Success Recipes*, Auerbach, 2011), Jarzabek (*Effective Software Maintenance and Evolution*, Auerbach, 2007), Grubb e Takang (*Software Maintenance: Concepts and Practice*, World Scientific Publishing Co., 2ª ed., 2003) e Pigoski (*Practical Software Maintenance*, Wiley, 1996). Esses livros abrangem as práticas básicas de manutenção e suporte e apresentam diretrizes úteis de gerenciamento. As técnicas de manutenção que focalizam ambientes cliente-servidor são discutidas por Schneberger (*Client/Server Software Maintenance*, McGraw-Hill, 1997). A pesquisa atual sobre “evolução de software” é apresentada em uma antologia editada por Mens e Demeyer (*Software Evolution*, Springer, 2008).

Como muitos tópicos atuais na área de negócios, a propaganda que envolve a reengenharia de processos de negócio deu origem a uma visão mais pragmática do assunto. Hammer e Champy (*Reengineering the Corporation*, HarperBusiness, edição revisada, 2003) fomentaram o interesse inicial com sua obra, que se tornou *best-seller*. Outros livros de Jacka e Keller (*Business Process Mapping: Improving Customer Satisfaction*, 2ª ed., Wiley, 2009), Sharp e McDermott (*Workflow Modeling*, 2ª ed., Artech House, 2008), Andersen (*Business Process Improvement Toolbox*, 2ª ed., American Society for Quality, 2007), Smith e Fingar (*Business Process Management (BPM): The Third Wave*, Meghan-Kiffer Press, 2003) e Harrington *et al.* (*Business Process Improvement Workbook*, McGraw-Hill, 1997) apresentam estudos de casos e diretrizes detalhadas para BPR.

Abfalter (*Software Reengineering*, VDM Verlag, 2008), Fong (*Information Systems Reengineering and Integration*, Springer, 2006) descrevem técnicas de conversão de banco de dados, engenharia reversa e engenharia direta da forma como são aplicadas para a maioria dos sistemas de informação. Nierstrasz e seus colegas (*Object Oriented Reengineering Patterns*, Square Bracket Associates, 2009) proporcionam uma visão baseada em padrões sobre como refazer e/ou recriar sistemas orientados a objetos. Secord e seus colegas (*Modernizing Legacy Systems*, Addison-Wesley, 2003), Ulrich (*Legacy Systems: Transformation Strategies*, Prentice Hall, 2002), Valenti (*Successful Software Reengineering*, IRM Press, 2002) e Rada (*Reengineering Software: How to Reuse Programming to Build New, State-of-the-Art Software*, Fitzroy Dearborn Publishers, 1999) concentram-se em estratégias e práticas para reengenharia em nível técnico. Miller (*Reengineering Legacy Software Systems*, Digital Press, 1998) “proporciona uma estrutura para manter sistemas de aplicações sincronizados com as estratégias de negócio e com as mudanças da tecnologia”.

Cameron (*Reengineering Business for Success in the Internet Age*, Computer Technology Research, 2000) e Umar (*Application (Re)Engineering: Building Web-Based Applications and Dealing with Legacies*, Prentice Hall, 1997) fornecem orientações para organizações que querem transformar sistemas legados em um ambiente baseado na Web. Cook (*Building Enterprise Information Architectures: Reengineering Information Systems*, Prentice Hall, 1996) aborda a ligação entre BPR e tecnologia da informação. Aiken (*Data Reverse Engineering*, McGraw-Hill, 1996) discute como reunir, reorganizar e reutilizar dados organizacionais. Arnold (*Software Reengineering*, IEEE Computer Society Press, 1993) montou uma excelente antologia dos primeiros artigos que focalizavam as tecnologias de reengenharia de software.

Uma grande variedade de fontes de informação sobre reengenharia de software está disponível na Internet. Uma lista atualizada das referências da Web pode ser encontrada em “recursos de engenharia de software”, no site da SEPA: www.mhhe.com/pressman.

PARTE V

Tópicos avancados

Nesta parte do livro, veremos vários tópicos avançados que ampliarão seu entendimento sobre a engenharia de software. Nos próximos capítulos serão tratadas as seguintes questões:

- O que é a melhoria do processo de software e como pode ser usada para melhorar as práticas de engenharia de software?
- Quais são as tendências emergentes que podem ter uma influência significativa sobre as práticas de engenharia de software na próxima década?
- Qual é a expectativa para os engenheiros de software?

Uma vez respondidas essas perguntas, você entenderá de assuntos que podem ter um profundo impacto sobre a engenharia de software nos próximos anos.

37 Melhoria do processo de software

Conceitos-chave

avaliação	823
mensuração	827
CMMI	828
educação e treinamento	825
gerenciamento de risco ..	827
instalação/migração	826
justificação	825
melhoria do processo de software (SPI)	
aplicabilidade	822
definição de	819
frameworks	819
processo	823
modelos de maturidade	821
People-CMM	832
retorno sobre investimento	834
seleção	825

Muito antes de a expressão “melhoria do processo de software” estar na moda, RSP trabalhou em grandes corporações tentando melhorar a qualidade de suas práticas de engenharia de software. Como resultado de suas experiências, escreveu o livro *Making Software Engineering Happen* [Pre88]. O prefácio apresentava o seguinte comentário:

Nos últimos 10 anos, tive a oportunidade de ajudar muitas grandes empresas a implementar práticas de engenharia de software. O trabalho é difícil e raramente ocorre tão tranquilamente quanto se gostaria – mas quando é bem-sucedido, os resultados são profundos. Os projetos de software têm maior probabilidade de serem concluídos a tempo. A comunicação entre todas as partes envolvidas no desenvolvimento de software melhoram. O nível de confusão e caos que com frequência prevalece para grandes projetos de software é reduzido substancialmente. O número de erros encontrados pelo cliente diminui muito. A credibilidade da organização de software melhora. E a gerência tem um problema a menos com que se preocupar.

Mas nem tudo é um mar de rosas. Muitas empresas tentam implementar práticas de engenharia de software e, frustradas, desistem. Outras fazem tudo pela metade e nunca desfrutarão dos benefícios que acabamos de descrever. Há ainda outras que o fazem da forma linha-dura, impositiva, o que resulta em rebelião declarada entre o pessoal técnico e os gerentes e na subsequente perda de moral.

PANORAMA

rão a um melhor processo de software e, em consequência, uma maior qualidade do software fornecido, dentro do prazo de entrega.

Quem realiza? As pessoas que defendem a SPI provêm de três grupos: gerentes técnicos, engenheiros de software e pessoas responsáveis pela garantia da qualidade.

Por que é importante? Algumas empresas de software têm pouco mais do que um processo de software específico. À medida que trabalham para melhorar suas práticas, devem resolver os pontos fracos do processo e melhorar sua abordagem para o trabalho de software.

O que é? A melhoria do processo de software engloba um conjunto de atividades que leva-

Quais são as etapas envolvidas? A SPI é iterativa e contínua, mas pode ser vista em cinco etapas: (1) avaliação do processo de software atual, (2) educação e treinamento dos profissionais e gerentes, (3) seleção e justificação de elementos do processo, métodos de engenharia de software e ferramentas, (4) implementação do plano da SPI e (5) avaliação e ajuste com base nos resultados do plano.

Qual é o artefato? Embora haja muitos produtos SPI intermediários, o resultado final é um processo de software melhorado que leva a uma qualidade mais alta do software.

Como garantir que o trabalho foi realizado corretamente? O software que a sua empresa produz será fornecido com menos defeitos, o retrabalho em cada estágio do processo de software será reduzido e a entrega no prazo se tornará muito mais possível.

Embora essas palavras tenham sido escritas há quase três décadas, elas permanecem válidas nos dias atuais.

Hoje, praticamente todas as grandes organizações têm tentado “fazer a engenharia de software acontecer”. Algumas implementaram práticas isoladas que ajudaram a melhorar a qualidade do produto e o prazo de entrega. Outras estabeleceram um processo de software “consolidado” que orienta suas atividades técnicas e de gerenciamento de projeto. Mas há aquelas que continuam a enfrentar dificuldades. Suas práticas são às vezes boas e outras não, e o processo é improvisado. Ocasionalmente, o trabalho é espetacular, mas, na maior parte, cada projeto é uma aventura, e ninguém sabe se terminará bem ou mal.

Sendo assim, qual desses dois grupos precisa de melhoria do processo de software? A resposta (que pode surpreendê-lo) é *ambos*. Os que tiveram sucesso ao implementar a engenharia de software não podem se tornar complacentes; devem trabalhar continuamente para melhorar sua abordagem de engenharia de software. E os que enfrentam dificuldades devem começar a tomar seu rumo em direção à melhoria.

37.1 O que é SPI?

O termo *melhoria do processo de software* (SPI, *software process improvement*) implica muitas coisas. Primeiro, implica que os elementos de um processo de software eficaz podem ser definidos de maneira eficaz; segundo, que uma abordagem organizacional existente para o desenvolvimento de software pode ser avaliada em relação aos elementos; e terceiro, que uma estratégia significativa pode ser definida para melhoria. A abordagem SPI transforma a abordagem existente para o desenvolvimento de software em algo mais focado, com melhor repetibilidade e mais confiável (em termos de qualidade de produto e prazo de entrega).

Como a SPI não é gratuita, ela deve produzir um retorno sobre o investimento. O trabalho e o tempo necessários para implementar uma estratégia SPI devem compensar de alguma maneira mensurável. Para tanto, os resultados do processo e prática melhorados devem levar a uma redução nos “problemas” de software que custam tempo e dinheiro. Eles devem reduzir: o número de defeitos que passam para os usuários, o volume de retrabalho causado por problemas de qualidade, os custos associados à manutenção e suporte do software (Capítulo 36) e os custos indiretos que ocorrem quando o software é entregue com atraso.

37.1.1 Abordagens para SPI

Embora a empresa possa escolher uma abordagem relativamente informal para SPI, a grande maioria escolhe um entre vários frameworks SPI. O *framework SPI* define (1) inúmeras características que devem estar presentes quando se deseja obter um processo de software eficaz, (2) um método para determinar se aquelas características estão presentes, (3) um mecanismo para

SPI implica um processo de software definido, uma abordagem organizacional e estratégia para melhoria.

“Grande parte da crise de software é autoimposta, segundo um Diretor-Executivo de Informação, ‘Prefiro errar do que atrasar. Sempre podemos corrigir depois’.”

Mark Paulk

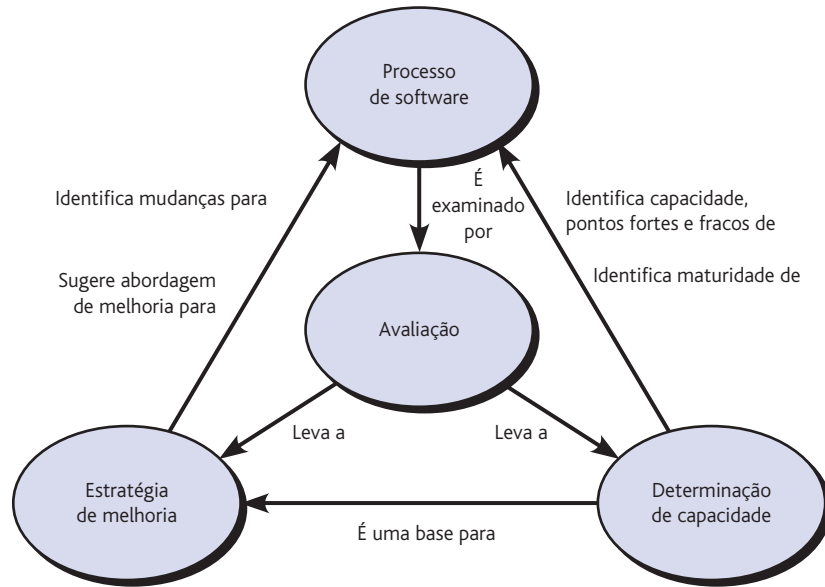


FIGURA 37.1 Elementos de um framework SPI.

Fonte: Adaptado de [Rou02].

resumir os resultados de qualquer avaliação e (4) uma estratégia para ajudar a organização na implementação das características de processo consideradas fracas ou ausentes.

Um framework SPI avalia a “maturidade” do processo de software de uma empresa e fornece uma indicação qualitativa do nível de maturidade – na verdade, o termo “modelo de maturidade” (Seção 37.1.2) é bastante aplicado. Em essência, o framework SPI abrange um modelo de maturidade que, por sua vez, incorpora uma série de indicadores de qualidade de processo que fornecem indicação geral da qualidade do processo que levará à qualidade do produto.

A Figura 37.1 fornece uma visão geral de um framework SPI padrão. São mostrados os elementos-chave do framework e suas relações uns com os outros.

É importante observar que não há um framework SPI universal. Na verdade, o framework SPI escolhido por uma empresa reflete o grupo de pessoas que apoia o trabalho da SPI. Conradi [Con96] define seis grupos de suporte a SPI:

Certificadores de qualidade. Os esforços para melhoria de processo defendidos por esse grupo concentram-se na seguinte relação:

$$\text{Qualidade(Processo)} \Rightarrow \text{Qualidade(Produto)}$$

A abordagem desse grupo enfatiza métodos de avaliação e examina uma série bem definida de características que lhes permitem determinar se o processo apresenta qualidade. São mais propensos a adotar uma metodologia de processo do tipo CMMI, SPICE, TickIT ou Bootstrap.¹

¹ Cada um desses frameworks SPI será discutido mais adiante neste capítulo.

Formalistas. Este grupo quer entender (e, quando possível, otimizar) o fluxo de trabalho de processo. Para tanto, usa linguagens de modelagem de processo (PMLs, process modeling languages) para criar um modelo do processo existente e então projetar extensões ou modificações que tornarão o processo mais eficaz.

Defensores das ferramentas. É um grupo que insiste em uma abordagem assistida por ferramenta para SPI, que modela o fluxo de trabalho e outras características de processo de maneira que possa ser analisada para melhoria.

Profissionais. Usa uma abordagem pragmática, “enfazando gerenciamento básico de projeto, qualidade e produto, aplicando planejamento e métricas em nível de projeto, mas com pouca modelagem formal de processo ou suporte” [Con96].

Reformadores. O objetivo deste grupo é a mudança organizacional que pode levar a um melhor processo de software. Eles tendem a se concentrar mais nos aspectos humanos (Seção 37.5) e enfatizam medidas de capacidade humana e estrutura.

Ideólogos. Este grupo foca na adequação de um modelo particular de processo para um domínio de aplicação ou estrutura organizacional específicos. Em vez de modelos de processo de software típicos (por exemplo, modelos iterativos), os ideólogos teriam um interesse maior em um processo que poderia, digamos, suportar a reutilização ou a reengenharia.

Quando um framework SPI é aplicado, os grupos que o apoiam (independentemente de seu foco geral) devem estabelecer mecanismos para: (1) suportar transição de tecnologia, (2) determinar o grau segundo o qual uma organização está pronta para absorver as mudanças de processo propostas e (3) medir o grau segundo o qual as mudanças foram adotadas.

37.1.2 Modelos de maturidade

Um *modelo de maturidade* é aplicado dentro do contexto de um framework SPI. Sua finalidade é dar uma indicação geral da “maturidade do processo” manifestada por uma empresa de software. Ou seja, a indicação da qualidade do processo de software, o grau segundo o qual os profissionais entendem e aplicam o processo e o estado geral da prática de engenharia de software. Para tanto, utiliza-se algum tipo de escala ordinal.

Por exemplo, o *Capability Maturity Model* (Seção 37.3) do Software Engineering Institute sugere cinco níveis de maturidade, variando de *inicial* (processo de software rudimentar) a *otimizado* (um processo que leva às boas práticas).² Infelizmente, algumas organizações de software apresentam níveis de “imaturidade de processo” que enfraquecem qualquer tentativa racional de melhorar as práticas de engenharia de software. Schorsch [Sch06] sugere quatro níveis de imaturidade organizacional, muitas vezes encontrados no universo do desenvolvimento de software:

O modelo de maturidade define níveis de competência no processo de software e implementação.

² O CMM original foi atualizado e é discutido na Seção 37.3.

Nível 0, Negligente – Não permite o processo de desenvolvimento bem-sucedido. Todos os problemas são considerados técnicos. As atividades gerenciais e de garantia da qualidade são consideradas complicadas e supérfluas para a tarefa do processo de desenvolvimento de software. Dependem de amuletos.

Nível 1, Obstrutivo – Impõe processos contraproducentes. Os processos são rigidamente definidos e há grande ênfase para que sejam seguidos. Cerimônias ritualísticas ocorrem em abundância. O gerenciamento coletivo inviabiliza a atribuição de responsabilidade. *Status quo über alles* (status quo acima de tudo).

Nível 2, Arrogante – Despreza completamente a boa engenharia de software. Uma separação total entre atividades de desenvolvimento de software e atividades de melhoria do processo de software. Ausência completa de um programa de treinamento.

Nível 3, Sabotador – Negligencia totalmente o próprio compromisso, descredita conscientemente os esforços de melhoria do processo de software da organização. Falta de incentivo e mau desempenho.

Os níveis de imaturidade de Schorsch são prejudiciais para qualquer empresa de software. Se você encontrar qualquer um deles, as tentativas de SPI estarão fadadas ao fracasso.

A questão fundamental é se as escalas de maturidade, como a proposta como parte do CMM, proporcionam qualquer benefício real. Pensamos que sim. A escala de maturidade proporciona uma visão clara da qualidade do processo que pode ser utilizado pelos profissionais e gerentes, como um *benchmark* a partir do qual podem ser planejadas as estratégias de melhoria.

37.1.3 A SPI é para todos?

Por muitos anos, a SPI foi vista como uma atividade “corporativa” – eufemismo para algo que apenas grandes empresas executam. Atualmente, porém, um porcentual significativo de todo o desenvolvimento de software é executado por empresas que empregam menos de 100 pessoas. Uma empresa pequena pode implementar atividades de SPI e executá-las de maneira bem-sucedida?

Há diferenças culturais importantes entre empresas de desenvolvimento de software grandes e pequenas. Não surpreende o fato de que pequenas organizações são mais informais, aplicam menos práticas padronizadas e tendem a ser auto-organizadas. Elas também tendem a se vangloriar da “criatividade” de seus membros e encaram, inicialmente, um framework SPI como demasiadamente burocrático e de grande peso. No entanto, a melhoria do processo é tão importante para a empresa pequena quanto para a grande.

Em pequenas organizações, a implementação de um framework SPI exige recursos que podem ser escassos. Os gerentes precisam alocar pessoas e dinheiro para colocar a engenharia de software em ação. Portanto, independentemente do tamanho da empresa de software, é válido considerar a motivação comercial para a SPI.

A SPI será aprovada e implementada somente depois que seus proponentes demonstrarem uma *alavancagem financeira* [Bir98]. A alavancagem financeira é demonstrada examinando-se as vantagens técnicas (por exemplo,

Se um modelo de processo específico ou uma abordagem SPI parecem ser opressivos para sua organização, provavelmente são.

menos defeitos que surgem após a entrega, retrabalho reduzido, menores custos de manutenção ou chegada mais rápida ao mercado) e traduzindo-as em valores monetários. Em essência, você deve mostrar um retorno realista sobre o investimento (Seção 37.7) para os custos SPI.

37.2 O processo de SPI

A parte difícil da SPI não é definir as características que determinam um processo de software de alta qualidade ou criar um modelo de maturidade de processo. Isso é relativamente fácil. A parte difícil é estabelecer um consenso para iniciar a SPI e definir uma estratégia contínua para implementá-la em uma empresa de software.

O Software Engineering Institute desenvolveu a IDEAL – “modelo de melhoria organizacional que serve de roteiro para iniciar, planejar e implementar ações de melhoria” [SEI08]. A IDEAL representa os muitos modelos de processo para SPI, definindo cinco atividades distintas – iniciar, diagnosticar, estabelecer, agir e aprender – que orientam a empresa tendo como base as atividades de SPI.

Neste livro, apresentamos um roteiro um pouco diferente para a SPI, baseado no modelo de processo para SPI proposto originalmente em [Pre88]. Ele aplica uma filosofia de bom senso que requer organização para que a empresa (1) se autoavalie, (2) se torne mais inteligente para que possa fazer escolhas inteligentes, (3) selecione o modelo de processo (e os elementos de tecnologia relacionados) que melhor satisfaça às suas necessidades, (4) crie uma instância do modelo em seu ambiente operacional e sua cultura e (5) avalie o que foi feito. Essas cinco atividades (discutidas nas próximas subseções)³ são aplicadas de maneira iterativa (cíclica) para fortalecer a melhoria contínua do processo.

37.2.1 Avaliação e análise de lacunas

Qualquer tentativa de melhorar o processo de software atual sem antes avaliar a eficácia das atividades metodológicas e práticas de engenharia de software associadas seria o mesmo que iniciar uma longa jornada sem ter ideia do ponto de partida. Você iniciará com grande entusiasmo, tentará encontrar seu rumo, gastará muita energia e suportará grandes doses de frustração e, provavelmente, decidirá que realmente não quer ir a lugar nenhum. Resumindo, antes de começar sua jornada, é aconselhável saber precisamente onde você está.

A primeira atividade do roteiro, chamada de *avaliação*, permite que você se oriente. A finalidade da avaliação é revelar os pontos fortes e fracos na maneira como sua organização aplica processos de software existentes e as práticas de engenharia de software que compõem o processo.

Procure entender seus pontos fortes e fracos. Se for inteligente, trabalhará nos pontos fortes.

³ Parte do conteúdo dessas seções foi adaptada de [Pre88] com permissão.

A avaliação examina uma ampla variedade de ações e tarefas que levarão a um processo de alta qualidade. Por exemplo, seja qual for o modelo de processo escolhido, a organização deve estabelecer mecanismos genéricos como: abordagens definidas para comunicação com o cliente; métodos estabelecidos para representar os requisitos do usuário; um framework de gerenciamento de projeto que inclua definição de escopo, estimativa, cronograma e rastreamento de projeto; métodos de análise de risco; procedimentos de gerenciamento de alterações; garantia de qualidade e atividades de controle, incluindo revisões; e muitas outras. Cada um desses mecanismos é considerado no contexto das atividades metodológicas (Capítulo 3) estabelecidas e avaliado para determinar se todas as questões a seguir podem ser resolvidas:

- O objetivo da atividade está claramente definido?
- Os produtos exigidos como entrada e produzidos como saída estão identificados e descritos?
- As tarefas a serem realizadas estão claramente descritas?
- As pessoas que devem desempenhar a atividade estão identificadas de acordo com suas funções?
- Os critérios de entrada e saída foram estabelecidos?
- As métricas para a atividade foram estabelecidas?
- Há ferramentas disponíveis para suportar a atividade?
- Há um programa de treinamento explícito que trata da atividade?
- A atividade é executada uniformemente para todos os projetos?

Embora as questões apresentadas demandem respostas de *sim* ou *não*, o papel da avaliação é examinar a resposta para determinar se a atividade em questão está sendo executada conforme as boas práticas.

À medida que é conduzida a avaliação do processo, você (ou aqueles que foram contratados para fazer a avaliação) também deve se concentrar nas seguintes questões:

Consistência. As atividades, ações e tarefas importantes são aplicadas consistentemente em todos os projetos de software e por todas as equipes de software?

Sofisticação. As ações técnicas e de gerência são executadas com um nível de sofisticação que demanda total entendimento das boas práticas?

Aceitação. O processo de software e as práticas de engenharia de software são amplamente aceitos pela gerência e pelo pessoal técnico?

Comprometimento. A gerência forneceu os recursos necessários para obter consistência, sofisticação e aceitação?

A diferença entre aplicação local e melhor prática representa uma “lacuna” que oferece oportunidades para melhoria. O grau segundo o qual consis-

tência, sofisticação, aceitação e compromisso são atingidos indica o nível de mudança cultural necessário para atingir uma melhora significativa.

37.2.2 Educação e treinamento

Embora poucos dos que trabalham software questionem as vantagens de um processo de software ágil e organizado ou de práticas de engenharia de software sólidas, muitos profissionais e gerentes não sabem o bastante sobre qualquer um dos assuntos.⁴ Consequentemente, percepções incorretas de processos e práticas levam a decisões inadequadas quando um framework SPI é introduzido. Daí é possível concluir que os elementos-chave de qualquer estratégia SPI são a educação e o treinamento para os profissionais, gerentes técnicos e gerentes seniores que têm contato direto com a organização de software. Três tipos de educação e treinamento devem ser promovidos: conceitos e métodos genéricos de engenharia de software, tecnologia e ferramentas específicas e comunicação e tópicos relacionados à qualidade. Em um contexto moderno, a educação e o treinamento podem ser fornecidos de diversas maneiras diferentes. Tudo, desde podcasts, vídeos curtos no YouTube, treinamento mais abrangente baseado na Internet (por exemplo, [QAI08]), até DVDs e cursos em sala de aulas, pode ser oferecido como parte de uma estratégia SPI.

Tente providenciar treinamento "just-in-time" dirigido para as reais necessidades da equipe de software.

37.2.3 Seleção e justificção

Terminada a atividade de avaliação⁵ e iniciada a de educação, a empresa de software deve começar a fazer suas escolhas. Essas escolhas ocorrem durante a *atividade de seleção e justificção*, na qual se optam por características de processo e métodos e ferramentas de engenharia de software específicos para preencher o processo de software.

Primeiro você deve escolher o modelo de processo (Capítulos 3 a 5) que melhor se adapte à sua organização, seus envolvidos e o software que você produz. Você deve decidir quais atividades do conjunto de atividades metodológicas vão ser aplicadas, os principais artefatos que serão produzidos e os pontos de verificação de garantia de qualidade que permitirão à sua equipe acompanhar o progresso. Se a atividade de avaliação de SPI indicar que você tem uma deficiência específica (por exemplo, não possui funções de SQA formais), concentre-se nas características do processo que tratam diretamente dessas deficiências.

Em seguida, faça uma divisão do trabalho adaptável para cada atividade metodológica (por exemplo, modelagem), definindo o conjunto de tarefas que seriam aplicadas a um projeto típico. Considere também os métodos de

Ao fazer suas escolhas, não deixe de considerar a cultura da sua organização e o nível de aceitação que cada escolha poderá atingir.

⁴ Se você está lendo este livro, não será um deles!

⁵ Atualmente, a avaliação é uma atividade contínua. Ela é conduzida periodicamente para determinar se a estratégia SPI atingiu seus objetivos imediatos e preparar o cenário para melhorias futuras.

engenharia de software que podem ser aplicados para executar essas tarefas. Conforme as escolhas forem feitas, a educação e o treinamento deverão ser coordenados para que o entendimento seja reforçado.

De maneira ideal, todos trabalham juntos para selecionar vários elementos de processo de tecnologia e poder direcionar-se para a atividade de instalação ou migração (Seção 37.2.4). Na realidade, a seleção pode ser um caminho complicado. Muitas vezes é difícil obter consenso entre os diferentes grupos. Se os critérios de seleção forem estabelecidos por meio de discussões, participantes poderão argumentar indefinidamente sobre sua adequação e se determinada escolha realmente atende aos critérios estabelecidos.

É verdade que uma má escolha pode prejudicar mais do que ajudar, mas “paralisia por análise” indica pouco progresso (se houver algum) e que os problemas de processo permanecem. Contanto que a característica de processo ou elemento de tecnologia tenha uma boa chance de atender às necessidades de uma organização, muitas vezes é melhor ser objetivo e fazer uma escolha do que esperar por uma solução perfeita.

37.2.4 Instalação/migração

Instalação é o primeiro ponto em que uma empresa de software sente os efeitos das mudanças implementadas pelo roteiro de SPI. Em alguns casos, é recomendado um processo inteiramente novo para a empresa. Atividades estruturais, ações de engenharia de software e tarefas de trabalho individual devem ser definidas e instaladas como parte de uma nova cultura de engenharia de software. Essas mudanças representam uma transição organizacional e tecnológica importante e devem ser administradas com muito cuidado.

Em outros casos, mudanças associadas à SPI são relativamente menos importantes, representando modificações pequenas, mas significativas, a um modelo de processo existente. Essas mudanças muitas vezes são chamadas de *migração de processo*. Hoje, muitas organizações de software têm um “processo” em andamento – o problema é que ele não funciona de maneira eficaz. Sendo assim, a *migração* incremental de um processo (que não funciona tão bem quanto se desejaria) para outro é uma estratégia mais eficaz.

A instalação e a migração são, na realidade, atividades de *redesenho de processo de software* (SPR, *software process redesign*). Scacchi [Sca00] afirma que o “SPR está ligado à identificação, aplicação e refinamento de novas maneiras de melhorar radicalmente e transformar o processo de software”. Quando é iniciada uma abordagem formal ao SPR, consideram-se três diferentes modelos de processo: (1) o processo existente (“da forma como está”), (2) um processo de transição (“daqui para lá”) e o processo-alvo (“o novo”). Se o processo-alvo for significativamente diferente do existente, a única abordagem racional para a instalação é uma estratégia incremental na qual o processo de transição é implementado em etapas. O processo de transição proporciona uma série de pontos intermediários que permitem à cultura da organização de software se adaptar a pequenas alterações durante um período.

37.2.5 Mensuração

Embora esteja listada como a última atividade no roteiro de SPI, a *mensuração* ocorre durante toda a SPI. A atividade de mensuração mede o grau segundo o qual as alterações foram criadas e adotadas, o grau segundo o qual essas alterações resultam em software de melhor qualidade ou outros benefícios de processo perceptíveis e o estado geral do processo e a cultura da organização conforme a SPI progride.

Durante a atividade de mensuração, são considerados fatores qualitativos e métricas quantitativas. Do ponto de vista qualitativo, as atitudes da gerência e dos profissionais sobre o processo de software no passado podem ser comparadas com as escolhidas após instalação das mudanças de processo. Métricas quantitativas (Capítulo 32) são coletadas de projetos que usaram o processo de transição ou “o novo” e comparadas com métricas similares coletadas para projetos executados de acordo com o processo “da forma como está”.

37.2.6 Gestão de riscos para SPI

A SPI é uma atividade de risco. Na verdade, mais da metade de todos os empreendimentos de SPI terminam em fracasso. As razões do fracasso variam muito e são específicas da organização. Entre os riscos mais comuns estão: falta de suporte gerencial, resistência cultural por parte do pessoal técnico, estratégia de SPI mal planejada, abordagem excessivamente formal à SPI, escolha de um processo inadequado, falta de interesse por parte dos principais envolvidos, orçamento inadequado, falta de treinamento do pessoal, instabilidade organizacional e uma infinidade de outros fatores. O papel dos responsáveis pela SPI é analisar os possíveis riscos e desenvolver uma estratégia interna para controlá-los.

Uma empresa de software deve administrar o risco em três pontos-chave no processo de SPI [Sta97b]: antes de iniciar o roteiro da SPI, durante a execução das atividades de SPI (avaliação, educação, seleção, instalação) e durante a atividade de avaliação que se segue à ocorrência de alguma característica de processo. Em geral, as seguintes categorias podem ser identificadas [Sta97b] para os fatores de risco da SPI: orçamento e custos, conteúdo e resultados práticos, cultura, manutenção de resultados práticos de SPI, missão e metas, gerenciamento organizacional, estabilidade organizacional, envolvidos no processo, cronograma para o desenvolvimento da SPI, ambiente de desenvolvimento da SPI, processo de desenvolvimento da SPI, gerenciamento de projeto da SPI e pessoal para a SPI.

Em cada categoria, há muitos fatores de risco genéricos. Por exemplo, a cultura organizacional tem forte influência sobre o risco. Podem ser identificados estes fatores⁶ de risco genéricos para a categoria cultural [Sta97b]:

- Atitude em relação à mudança, com base em esforços anteriores para mudar
- Experiência com programas de qualidade, nível de sucesso

A SPI muitas vezes falha porque os riscos não foram considerados adequadamente e não houve planejamento de contingência.

⁶ Fatores de risco para cada uma das categorias de risco descritas nesta seção podem ser encontrados em [Sta97b].

- Orientação de ações para resolver os problemas *versus* debates políticos
- Uso de fatos para administrar a organização e os negócios
- Paciência com as mudanças, habilidade em empregar o tempo para a socialização
- Orientação para as ferramentas – a expectativa de que as ferramentas possam resolver os problemas
- Nível de “totalidade de planejamento” – habilidade da organização em planejar
- Habilidade dos membros da organização em participar abertamente nas reuniões com vários níveis da organização
- Habilidade dos membros da organização em administrar as reuniões eficientemente
- Nível de experiência em organização com processos definidos

Usando-se os fatores de risco e atributos genéricos como guia, uma tabela de riscos (Capítulo 35) pode ser desenvolvida para isolar os riscos que merecem mais atenção.

37.3 O CMMI

O CMM original foi desenvolvido e atualizado pelo Software Engineering Institute na década de 1990 como um framework SPI completo. Hoje, evoluiu tornando-se o *Capability Maturity Model Integration* (CMMI) [CMM07], um metamodelo de processo abrangente, qualificado em uma série de capacidades de sistema e engenharia de software que devem estar presentes à medida que as organizações alcançam diferentes níveis de capacitação e maturidade de processo.

O CMMI representa um metamodelo de processo de duas maneiras diferentes: (1) como um modelo “contínuo” e (2) como um modelo “por estágio”. O metamodelo CMMI contínuo descreve um processo em duas dimensões, conforme está ilustrado na Figura 37.2. Cada área de processo (por exemplo, planejamento de projeto ou gerenciamento de requisitos) é formalmente avaliada em relação a metas e práticas específicas e classificada de acordo com os seguintes níveis de capacidade:

Nível 0: *Incompleto* – A área de processo (por exemplo, gerenciamento de requisitos) não funciona ou não atinge todas as metas e objetivos definidos pelo CMMI para a capacidade nível 1 para a área de processo.

Nível 1: *Executado* – Todas as metas específicas da área de processo (definida pelo CMMI) foram satisfeitas. Estão sendo executadas as tarefas necessárias para produzir os artefatos definidos.

Nível 2: *Controlada* – Todos os critérios do nível de capacidade 1 foram satisfeitos. Além disso, todo o trabalho associado à área de processo está de acordo com uma política definida em termos de organização, todas as pessoas que estão fazendo o trabalho têm acesso a recursos adequados

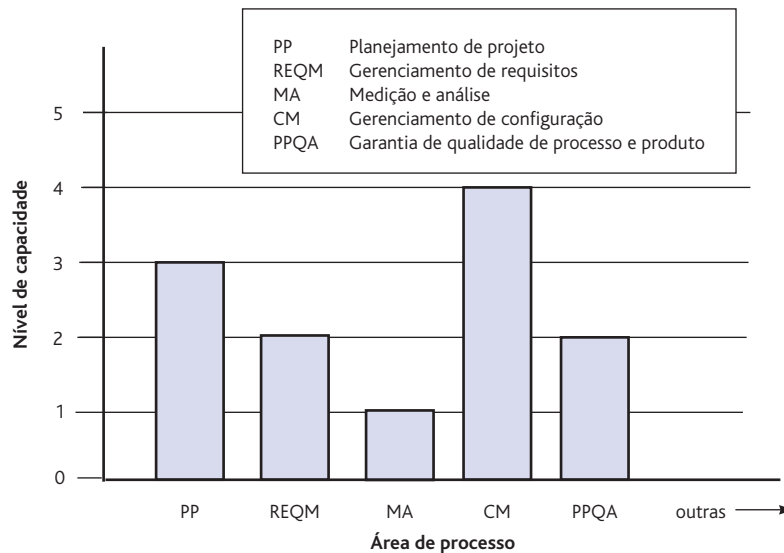


FIGURA 37.2 Perfil da capacidade da área de processo CMMI.

Fonte: [Phi02].

para executar o trabalho, os envolvidos agem ativamente na área de processo conforme necessário, todas as tarefas e produtos são “monitorados, controlados, revisados e avaliados quanto à conformidade com a descrição de processo” [CMM07].

Nível 3: Definido – Todos os critérios do nível de capacidade 2 foram satisfeitos. Além disso, o processo é “adaptado com base no conjunto de processos padronizados da organização, de acordo com as regras de adaptação da organização e dos produtos acabados, medidas e outras informações de melhoria de processo para agregar valores ao processo organizacional” [CMM07].

Nível 4: Controlado quantitativamente – Todos os critérios do nível de capacidade 3 foram satisfeitos. Além disso, a área de processo é controlada e melhorada usando medição e avaliação quantitativa. “São estabelecidos objetivos quantitativos para qualidade e desempenho de processo e utilizados como critérios no controle do processo” [CMM07].

Nível 5: Otimizado – Todos os critérios do nível de capacidade 4 foram satisfeitos. Além disso, a área de processo é adaptada e otimizada usando meios quantitativos (estatísticos) para atender à mudança de necessidades do cliente e melhorar continuamente a eficiência da área de processo em consideração.

O CMMI define cada área de processo em termos de “metas específicas” e as “práticas específicas” necessárias para atingir essas metas. As *metas específicas* estabelecem as características que devem existir para que as atividades envolvidas por uma área de processo sejam eficazes. *Práticas específicas* refinam uma meta, transformando-a em uma série de atividades relacionadas ao processo.

Toda organização deve lutar para atingir o objetivo do CMMI. No entanto, a implementação de todo o aspecto do modelo pode ser desastrosa na sua situação.

Informações completas, bem como download de uma versão do CMMI, podem ser obtidas em <http://cmmiinstitute.com/resources/>.

Por exemplo, **planejamento de projeto** é uma das oito áreas de processo definidas pelo CMMI para a categoria “gerenciamento de projeto”.⁷ As metas específicas (SG, specific goals) e as práticas específicas (SP, specific practices) associadas, definidas para o **planejamento de projeto**, são [CMM07]:

SG 1 Estabelecer Estimativa

SP 1.1-1 Estimar o Escopo do Projeto

SP 1.2-1 Estabelecer Estimativas de Atributos de Produto e Tarefa

SP 1.3-1 Definir o Ciclo de Vida do Projeto

SP 1.4-1 Determinar Estimativas de Trabalho e Custo

SG 2 Desenvolver um Plano de Projeto

SP 2.1-1 Estabelecer o Orçamento e o Cronograma

SP 2.2-1 Identificar os Riscos do Projeto

SP 2.3-1 Planejar o Gerenciamento de Dados

SP 2.4-1 Elaborar Plano para Recursos de Projeto

SP 2.5-1 Elaborar Plano para Conhecimento e Habilidades Necessárias

SP 2.6-1 Elaborar Plano para Participação dos Envolvidos

SP 2.7-1 Estabelecer o Plano de Projeto

SG 3 Obter Comprometimento com o Plano

SP 3.1-1 Rever Planos Que Afetam o Projeto

SP 3.2-1 Reconciliar Trabalho e Níveis de Recursos

SP 3.3-1 Obter Comprometimento com o Plano

Além das metas e práticas específicas, o CMMI também define um conjunto de cinco metas genéricas e práticas relacionadas a cada área de processo. Cada meta corresponde a um dos cinco níveis de capacidade. Portanto, para atingir determinado nível de capacidade, a meta genérica para aquele nível e as práticas genéricas que correspondem àquela meta devem ser atingidas.

O modelo CMMI por estágio define as mesmas áreas de processo, metas e práticas do modelo contínuo. A principal diferença é que o modelo por estágio define cinco níveis de maturidade, em vez de cinco níveis de capacidade. Para atingir um nível de maturidade, as metas específicas e as práticas associadas a um conjunto de áreas de processo devem ser atingidas. A relação entre níveis de maturidade e áreas de processo é mostrada na Figura 37.3

⁷ Outras áreas de processo para “gerenciamento de projeto” incluem: monitoramento e controle de projeto, gerenciamento de acordos com fornecedores, gerenciamento integrado de projeto para IPPD, gestão de risco, equipe integrada, gerenciamento integrado de fornecedor e gerenciamento quantitativo de projeto.

Nível	Foco	Áreas de Processo
Otimizante	<i>Melhoria contínua do processo</i>	Inovação organizacional e entrega Análise causal e resolução
Controlado quantitativamente	<i>Gerenciamento quantitativo</i>	Desempenho de processo organizacional Gerenciamento quantitativo de projeto
Definido	<i>Padronização de processo</i>	Desenvolvimento de requisitos Solução técnica Integração de produto Verificação Validação Foco no processo organizacional Definição de processo organizacional Treinamento organizacional Gerenciamento de projeto integrado Gerenciamento de fornecimento integrado Gestão de risco Análise de decisão e resolução Ambiente organizacional para integração Equipe integrada
Repetível	<i>Gerenciamento básico de projeto</i>	Gerenciamento de requisitos Planejamento de projeto Monitoração e controle de projeto Gerenciamento de acordo com fornecedor Medição e análise Garantia de qualidade de processo e produto Gerenciamento de configuração
Executado		

FIGURA 37.3 Áreas de processo necessárias para atingir um nível de maturidade.

Fonte: [Phi02].

INFORMAÇÕES



CMMI – Fazer ou não fazer?

CMMI é um metamodelo de processo. Ele define (em mais de 700 páginas) as características de processo que devem existir caso uma empresa de software queira estabelecer um processo de software completo. A questão debatida há quase duas décadas é: “O CMMI é opressivo?”. Como muitas coisas na vida (e em software), a resposta não é um simples sim ou não.

O espírito do CMMI sempre deve ser adotado. Com o risco da simplificação, o desenvolvimento de software deve ser encarado com seriedade – deve ser planejado amplamente, controlado com uniformidade, acompanhado com precisão e conduzido profissionalmente. Deve concentrar-se nas necessidades dos patrocinadores/envolvidos no projeto, nas habilidades dos engenheiros de software e na qualidade do produto final. Essas ideias são indiscutíveis.

Os requisitos detalhados do CMMI devem ser considerados seriamente se uma empresa cria sistemas grandes

e complexos que envolvem dezenas ou centenas de pessoas por muitos meses ou anos. Pode ocorrer que o CMMI esteja “na medida certa” em tais situações, se a cultura organizacional for favorável a modelos de processo-padrão e a gerência estiver comprometida em torná-la um sucesso. No entanto, em outras situações, o CMMI pode ser demais para uma organização assimilá-lo. Isso significa que o CMMI é “ruim” ou “demasiadamente burocrática” ou “à moda antiga?”. Não... não é. Significa que o que é certo para uma cultura organizacional pode não ser certo para outra.

O CMMI é uma conquista importante na engenharia de software. Proporciona uma discussão abrangente das atividades e ações que devem existir quando uma organização cria software de computador. Mesmo que a empresa prefira não adotar seus detalhes, toda equipe de software deveria aderir a esse espírito e aprender com a discussão de processo e prática de engenharia de software.

O People-CMM sugere práticas que melhoram a competência e a cultura da força de trabalho.

37.4 People-CMM

Não importa quão bem seja concebido, um processo de software não será bem-sucedido sem profissionais talentosos e motivados. O *Modelo de Maturidade de Capacitação de Pessoas* (People-CMM, *People Capability Maturity Model*) “é um roteiro para implementar práticas de trabalho que aperfeiçoam continuamente a capacidade dos profissionais de uma organização” [Cur01]. Desenvolvido no decorrer da década de 1990 e refinado nos anos seguintes, o objetivo do People-CMM é estimular a melhoria contínua do conhecimento genérico da força de trabalho (chamado de “competências núcleo”), habilidades específicas de engenharia de software e gerenciamento de projeto (chamado de “competências da força de trabalho”) e habilidades relacionadas ao processo.

Assim como o CMM, o CMMI e frameworks SPI relacionados, o People-CMM define um conjunto de cinco níveis de maturidade organizacional que proporcionam uma indicação da sofisticação relativa das práticas e processos da força de trabalho. Esses níveis de maturidade [CMM08] estão ligados à existência (dentro de uma organização) de um conjunto de áreas-chave de processo (KPA, key process areas). Uma visão geral dos níveis organizacionais e KPAs relacionadas está na Figura 37.4.

O People-CMM complementa qualquer framework SPI, estimulando a empresa a cultivar e melhorar seu bem mais precioso – as pessoas. Sendo importante, estabelece uma atmosfera na força de trabalho que permite à organização de software “atrair, desenvolver e preservar talentos notáveis” [CMM08].

37.5 Outros frameworks SPI

Embora o CMM e o CMMI do Software Engineering Institute (SEI) sejam os frameworks SPI mais amplamente aplicados, muitas alternativas foram propostas e estão em uso.⁸ Fornecemos uma breve visão geral desses frameworks nos parágrafos a seguir.⁹

Além do CMM, há outros frameworks SPI que podem ser considerados?

SPICE. O modelo SPICE (*Software Process Improvement and Capability Determination* – Melhoria de Processo de Software e Determinação de Capacitação) [SPI99] proporciona um framework de avaliação SPI compatível com a ISO 15504:2003 e ISO 12207. O conjunto de documentos SPICE [SDS08] apresenta um framework SPI completo, incluindo um modelo para gerenciamento de processo, diretrizes para conduzir uma avaliação e classificação do processo em consideração, construção, seleção e uso dos instrumentos e ferramentas de avaliação e treinamento para os avaliadores.

⁸ Alguns desses frameworks não são exatamente “alternativas”, já que são abordagens complementares à SPI. Uma tabela abrangente com muito mais frameworks SPI pode ser encontrada em <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.13.4787&rep=rep1&type=pdf>.

⁹ Se houver interesse, para cada uma delas há uma grande variedade de material impresso e baseado na Web.

Nível	Foco	Áreas de Processo
Otimizante	<i>Melhoria contínua</i>	Inovação contínua da força de trabalho Alinhamento do desempenho organizacional Melhora contínua da capacidade
Previsível	<i>Identifica e desenvolve conhecimento, prática e habilidades</i>	Tutela Gerenciamento da capacidade organizacional Gerenciamento do desempenho quantitativo Propriedades baseadas na competência Grupos de trabalho fortalecidos Integração de competência
Definido	<i>Quantifica e administra conhecimento, prática e habilidades</i>	Cultura participativa Desenvolvimento de grupo de trabalho Práticas baseadas na competência Desenvolvimento de carreira Desenvolvimento de competência Planejamento da força de trabalho Análise de competência
Repetível	<i>Práticas básicas de gerenciamento de pessoas que podem ser repetidas</i>	Compensação Treinamento e desenvolvimento Gerenciamento de desempenho Ambiente de trabalho Comunicação e coordenação Pessoal
Inicial	<i>Práticas inconsistentes</i>	

FIGURA 37.4 Áreas de processo para People-CMM.

Bootstrap. O framework SPI *Bootstrap* “foi desenvolvido para assegurar a conformidade com a norma ISO emergente para avaliação e melhoria de processo de software (SPICE) e para alinhar a metodologia com a ISO 12207” [Boo06]. O objetivo do Bootstrap é examinar um processo de software usando um conjunto de boas práticas de engenharia de software como base para a avaliação. Assim como o CMMI, o Bootstrap proporciona um nível de maturidade de processo por meio dos resultados de questionários que reúnem informações sobre o processo de software e projetos de software “tal como estão”. As diretrizes SPI baseiam-se no nível de maturidade e nas metas organizacionais.

PSP e TSP. Embora a SPI seja caracterizada em geral como uma atividade organizacional, não há razão para uma melhoria de processo não ser conduzida em nível individual ou de equipe. Tanto o PSP quanto o TSP (Capítulo 4) enfatizam a necessidade de coletar continuamente dados sobre o trabalho que está em execução e usar esses dados para desenvolver estratégias para melhorias. Watts Humphrey ([Hum97], [Hum00]), o desenvolvedor de ambos os métodos, comenta:

"As empresas de software têm mostrado grande deficiência em sua habilidade de tirar proveito das experiências adquiridas com projetos executados."

NASA

O PSP [e TSP] mostrará como planejar e acompanhar o seu trabalho e como produzir consistentemente software de alta qualidade. O uso de PSP [e TSP] fornecerá os dados que indicam a eficácia do seu trabalho e que identificam seus pontos fortes e fracos... Para uma carreira bem-sucedida e bem remunerada, você precisa conhecer suas experiências e habilidades, lutar para melhorá-las e concentrar seu talento exclusivamente no trabalho.

TickIT. O método de auditoria Ticket [Tic05] assegura a conformidade com a norma *ISO 9001:2000 para Software* – uma norma genérica que se aplica a qualquer organização que queira melhorar a qualidade geral dos produtos, sistemas ou serviços que fornece. Portanto, a norma é diretamente aplicável a organizações e empresas de software.

A estratégia subjacente sugerida pela ISO 9001:2000 é descrita da seguinte maneira [ISO00]:

A ISO 9001:2000 destaca a importância para uma organização em identificar, implementar, administrar e melhorar continuamente a eficácia dos processos necessários para o sistema de gerenciamento de qualidade e para administrar as interações desses processos com o fim de atingir os objetivos da organização... A eficácia e eficiência do processo podem ser estimadas por meio de processos de revisão interna e externa e podem ser avaliadas em uma escala de maturidade.

Um excelente resumo da ISO 9001:2008 pode ser encontrado em <http://praxiom.com/iso-9001.htm>.

A norma ISO 9001:2008 adotou um ciclo “planejar-fazer-verificar-agir” aplicado aos elementos de gerenciamento de qualidade de um projeto de software. Em um contexto de software, o “planejar” estabelece os objetivos do processo, atividades e tarefas necessárias para obter software de alta qualidade e, como resultado, a satisfação do cliente. “Fazer” implementa o processo de software (incluindo tanto atividades de estrutura quanto de apoio). “Verificar” monitora e mede o processo para garantir que todos os requisitos estabelecidos para gerenciamento da qualidade tenham sido atingidos. “Agir” inicia as atividades de melhoria do processo de software que contribuem continuamente para aprimorar o processo. TickIT pode ser usado em todo o ciclo “planejar-fazer-verificar-agir” para assegurar que o progresso SPI esteja ocorrendo. Os auditores TickIT avaliam a aplicação do ciclo como um precursor da certificação ISO 9001:2008. Para uma discussão detalhada da ISO 9001:2008 e TickIT, consulte [Ant06], [Tri05] ou [Sch03].

37.6 Retorno sobre investimento em SPI

A SPI é um trabalho pesado e requer investimento substancial em dinheiro e profissionais. Os administradores que aprovam o orçamento e recursos destinados à SPI invariavelmente farão a seguinte pergunta: “Como posso saber se obtivemos um retorno razoável do dinheiro que gastamos?”.

Em nível qualitativo, os proponentes da SPI argumentam que um processo de software melhorado levará a uma qualidade superior. Eles sustentam que um processo melhorado resultará na implementação de filtros de qualidade superiores (resultando em menor propagação de defeitos), melhor controle das alterações (resultando em menor caos no projeto) e menos retrabalho téc-

nico (resultando em menor custo e melhor prazo de entrega para o mercado). Mas podem esses benefícios qualitativos ser traduzidos em resultados quantitativos? A equação clássica do retorno sobre o investimento (ROI, return on investment) é:

$$\text{ROI} = \left[\frac{|\Sigma(\text{benefícios}) - \Sigma(\text{custos})|}{\Sigma(\text{custos})} \right] \times 100\%$$

onde

benefícios incluem as economias em custos associadas à melhor qualidade do produto (menos defeitos), menos retrabalho, redução do trabalho associado a alterações e lucro proveniente de uma entrega mais rápida ao mercado;

custos incluem custos SPI diretos (por exemplo, treinamento, medição) e custos indiretos associados à maior ênfase no controle de qualidade e atividades de gerenciamento de mudanças e uma aplicação mais rigorosa dos métodos de engenharia de software (por exemplo, a criação de um modelo de projeto).

Na prática, esses benefícios quantitativos e custos são muitas vezes difíceis de medir com precisão, e todos dependem de uma interpretação. Mas isso não significa que uma organização de software deve executar um programa SPI sem cuidadosa análise dos custos e benefícios que se acumulam. Um tratamento abrangente do retorno de investimento para a SPI pode ser encontrado em um livro especial de David Rico [Ric04].

37.7 Tendências da SPI

Nos últimos 25 anos, muitas empresas tentaram melhorar suas práticas de engenharia de software aplicando um framework SPI para promover mudança organizacional e transição de tecnologia. Conforme observamos antes neste capítulo, mais da metade fracassou nessa tentativa. Independentemente do sucesso ou falha, tudo custa muito. David Rico [Ric04] relata que uma aplicação típica de um framework SPI como o SEI CMM pode custar entre \$ 25 mil e \$ 70 mil por pessoa e levar anos para se completar! Não é surpresa o fato de que o futuro da SPI deve dar ênfase a uma abordagem menos dispendiosa e menos demorada.

Para serem eficazes no mundo do desenvolvimento de software do século 21, os futuros frameworks SPI devem se tornar significativamente mais ágeis. Em vez de um foco organizacional (que pode levar anos para se completar com sucesso), os esforços de SPI contemporâneos devem se concentrar no nível de projeto, trabalhando para melhorar um processo de equipe em semanas, não em meses ou anos. Para obter resultados significativos (mesmo em nível de projeto) em curto prazo, modelos complexos de framework podem ser substituídos por modelos mais simples. Em lugar de dezenas de práticas-chave e centenas de práticas suplementares, um framework SPI ágil deve dar ênfase apenas a poucas práticas críticas (por exemplo, análogas às atividades de framework discutidas neste livro).

Qualquer tentativa em SPI demanda uma força de trabalho com conhecimento, mas as despesas com educação e treinamento podem ser altas e devem ser minimizadas (e enxugadas). Em vez de cursos em salas de aula (caros e demorados), trabalhos de SPI futuros deverão utilizar treinamento baseado na Web, destinado a práticas críticas. Ao contrário de tentativas avançadas para mudar a cultura organizacional (com todos os perigos políticos que surgem), a mudança cultural deverá ocorrer com um pequeno grupo de cada vez até se alcançar o ponto crucial.

O trabalho de SPI das duas últimas décadas tem um mérito significativo. Os frameworks e modelos desenvolvidos representam valores intelectuais importantes para a comunidade de engenharia de software. Mas esses valores direcionam futuras tentativas em SPI, não se tornando dogma recorrente, mas servindo como base para modelos SPI melhores, mais simples e mais ágeis.

37.8 Resumo

Um framework de melhoria do processo de software define as características que devem estar presentes quando se quer atingir um processo de software eficaz, um método de avaliação que ajude a determinar se essas características estão presentes e uma estratégia para ajudar a empresa de software a modificar as características de processo consideradas fracas ou ausentes. Independentemente do grupo que apoia a SPI, o objetivo é melhorar a qualidade do processo e, como consequência, melhorar a qualidade do software e os prazos.

O modelo de maturidade de processo dá uma indicação geral da “maturidade de processo” exibida por uma empresa de software. Ele proporciona um senso de qualidade sobre a eficiência relativa do processo de software que está em uso no momento.

O roteiro da SPI começa com a avaliação – uma série de atividades que revelam os pontos fracos e fortes da aplicação do processo de software e das práticas de engenharia de software que fazem parte do processo. Desse modo, a empresa pode desenvolver um plano geral de SPI.

Um dos elementos-chave de qualquer plano SPI é educação e treinamento, uma atividade que se concentra na melhoria do nível de conhecimento dos gerentes e profissionais. Depois que o pessoal se torna experiente nas tecnologias de software atuais, começa a seleção e a justificação. Essas tarefas levam a escolhas sobre a arquitetura do processo de software, os métodos que fazem parte dela e as ferramentas que a suportam. Instalação e avaliação são atividades de SPI que causam mudanças no processo e examinam sua eficácia e impacto.

Para melhorar o processo de software, a empresa deve ter as seguintes características: comprometimento e suporte da gerência para com a SPI, envolvimento do pessoal durante todo o processo de SPI, integração do processo na cultura organizacional geral, uma estratégia SPI personalizada para as necessidades locais e uma administração sólida do projeto de SPI.

Hoje há uma série de frameworks SPI em uso: CMM e CMMI do SEI são amplamente utilizadas. O People-CMM foi personalizado para avaliar

a qualidade da cultura organizacional e os profissionais envolvidos. SPICE, Bootstrap, PSP, TSP e TickIT são frameworks adicionais que podem levar a uma SPI eficaz.

SPI é trabalho sério e requer investimento substancial em dinheiro e pessoas. Para garantir a obtenção de um retorno sobre o investimento razoável, a organização deve medir os custos associados à SPI e os benefícios que podem ser atribuídos a ela.

Problemas e pontos a ponderar

37.1 Por que as organizações de software usualmente lutam com dificuldade quando aderem a um esforço para melhorar o processo local de software?

37.2 Descreva o conceito de “maturidade de processo” com suas próprias palavras.

37.3 Faça uma pesquisa (verifique no site do SEI) e determine a distribuição da maturidade de processo para organizações de software nos Estados Unidos e no mundo todo.

37.4 Você trabalha para uma pequena organização de software – apenas 11 pessoas estão envolvidas no desenvolvimento de software. A SPI é adequada para sua empresa? Justifique sua resposta.

37.5 Avaliação é semelhante a um exame anual de saúde. Usando essa metáfora, descreva a atividade de avaliação SPI.

37.6 Qual a diferença entre um processo “da forma como está”, um processo “daqui para lá” e um processo “novo”?

37.7 Como é aplicada a gestão de risco no contexto da SPI?

37.8 Selecione um dos fatores críticos de sucesso na Seção 37.2.7. Faça uma pesquisa e escreva um pequeno artigo sobre como pode ser alcançado.

37.9 Faça uma pesquisa e explique como a CMMI difere de sua antecessora, a CMM.

37.10 Escolha um dos frameworks SPI discutidos na Seção 37.5 e redija um pequeno artigo descrevendo-o em mais detalhes.

Leituras e fontes de informação complementares

Um dos recursos mais facilmente acessíveis e abrangentes de informações sobre SPI foi desenvolvido pelo Software Engineering Institute e está disponível em www.sei.cmu.edu e www.cmmiinstitute.com. Os sites do SEI contêm centenas de artigos, estudos e descrições detalhadas de frameworks SPI.

Durante os últimos anos, muitos bons livros foram acrescentados a uma ampla literatura desenvolvida durante as duas últimas décadas. Chrissis e seus colegas (*CMMI for Development: Guidelines for Process Integration and Product Improvement*, 3ª ed., Addison-Wesley, 2011) e McMahon (*Integrating CMMI and Agile Development*, Addison-Wesley, 2010) discutem a aplicação de CMMI em desenvolvimento moderno de software. Land (*Jumpstart CMM/CMMI Software Process Improvements*, Wiley-IEEE Computer Society, 2007) apresenta os requisitos definidos como parte dos padrões de engenharia de software SEI CMM e CMMI com IEEE, com ênfase na interseção de processo e prática. Micklewright (*Lean ISO 9001*, ASQ Quality Press, 2010) e Cianfrani e seus colegas (*ISO 9001:2008 Explained*, 3ª ed., ASQ Quality Press, 2009) descrevem o significado e a finalidade da ISO 9001:2008. Mutafelija e Stromberg (*Systematic Process Improvement Using ISO 9001:2000 and CMMI*, Artech House Publishers, 2007) discutem os frameworks ISO

9001:2000 e CMMI SPI e a “sinergia” entre eles. Conradi e seus colegas (*Software Process Improvement: Results and Experience from the Field*, Springer, 2006) apresentam os resultados de uma série de estudos de caso e experimentos relacionados à SPI.

McKay e Black (*Improving the Software Process*, RBCS, 2012) oferecem um exame detalhado das estratégias e tendências da SPI. Fauzi e seus colegas (*Software Process Improvement: Approaches and Tools for Practical Development*, IGI Global, 2011), Van Loon (*Process Assessment and Improvement: A Practical Guide for Managers, Quality Professionals and Assessors*, Springer, 2006) discutem a SPI no contexto da ISO/IEC 15504. Watts Humphrey (*PSP*, Addison-Wesley, 2005 e *TSP*, Addison-Wesley, 2005) trata de seus frameworks SPI Personal Team Process e Team Software Process em dois livros. Fantina (*Practical Software Process Improvement*, Artech House Publishers, 2004) traz instruções pragmáticas com ênfase em CMMI/CMM.

Há disponível na Internet uma grande variedade de fontes de informação sobre melhoria de processo de software. Uma lista atualizada das referências da Web pode ser encontrada em “recursos de engenharia de software”, no site da SEPA: **www.mhhe.com/pressman**.

Tendências emergentes na engenharia de software

38

Ao longo da história relativamente breve da engenharia de software, profissionais e pesquisadores desenvolveram inúmeros modelos de processo, métodos técnicos e ferramentas automatizadas em um esforço de apoiar a mudança na maneira como criamos software para computador. Apesar de a experiência indicar o contrário, há um desejo implícito de se encontrar a “solução mágica” – o processo mágico ou tecnologia transcendente que nos permitirá criar facilmente sistemas grandes e complexos, sem confusão, enganos e atrasos – sem os vários problemas que continuam a povoar o trabalho de software.

A história, no entanto, indica que nossa busca pela solução mágica parece estar fadada ao fracasso. Novas tecnologias são introduzidas regularmente, apresentadas como a “solução” para muitos dos problemas que os engenheiros de software enfrentam e incorporadas a projetos grandes e pequenos. Críticos do setor exageram a importância dessas “novas” tecnologias de software, os especialistas da comunidade de software as adotam com entusiasmo e, por fim, elas desempenham um papel no mundo da engenharia de software. Mas tendem a não produzir o resultado esperado e, como consequência, a busca continua.

Em edições anteriores deste livro (ao longo dos últimos 35 anos), discutimos tecnologias emergentes e seu impacto sobre a engenharia de software. Algumas foram amplamente adotadas, outras nunca alcançaram seu poten-

Conceitos-chave

blocos básicos	847
ciclo de excelência	842
ciclo de vida de	
inovação	840
código aberto	848
complexidade	845
desenvolvimento	
controlado por modelo	853
desenvolvimento	
guiado por teste	854
desenvolvimento	
colaborativo	852
engenharia de	
requisitos	852
evolução da	
tecnologia	840
ferramentas	855
projeto pós-moderno	854
requisitos emergentes	846
direções da tecnologia	849
software aberto	846
tendências leves	843

PANORAMA

O que é? Ninguém pode prever o futuro com certeza absoluta, mas é possível avaliar tendências na área de engenharia de software e, a partir delas, sugerir direções possíveis para a tecnologia. É isso que tentamos fazer neste capítulo.

Quem realiza? Qualquer um que deseje dedicar seu tempo para se manter atualizado sobre os problemas da engenharia de software pode tentar prever a direção futura da tecnologia.

Por que é importante? Por que os reis na antiguidade consultavam os adivinhos? Por que grandes corporações multinacionais contratam firmas de consultoria e se esforçam para fazer previsões? Por que uma grande parte do público lê horóscopos? Queremos sempre saber o que vai acontecer para nos prepararmos.

Quais são as etapas envolvidas? Não há uma fórmula para prever o caminho à frente. Tentamos fazer isso coletando dados, organizando esses dados para que nos forneçam informações úteis, examinando associações sutis para extrair conhecimento e, desse conhecimento, sugerir prováveis tendências que prevejam como as coisas serão em algum tempo futuro.

Qual é o artefato? Uma visão do futuro próximo que pode ser ou não correta.

Como garantir que o trabalho foi realizado corretamente? Prever o caminho à frente é uma arte, não uma ciência. De fato, é muito raro que uma previsão séria sobre o futuro esteja absolutamente certa ou completamente errada (com exceção, felizmente, das previsões do fim do mundo). Procuramos tendências e tentamos extrapolá-las. Podemos avaliar a precisão dessa extrapolação somente com o passar do tempo.

cial. Nossa conclusão: as tecnologias vêm e vão; as tendências que devemos explorar são mais flexíveis. Em outras palavras, o progresso na engenharia de software será orientado pelas tendências nos negócios, organizações, mercado e tendências culturais. Todas elas levam à inovação tecnológica.

Neste capítulo, examinaremos algumas tendências tecnológicas na engenharia de software, mas nossa ênfase será sobre algumas tendências nos negócios, nas organizações, no mercado e culturais que podem ter influência importante sobre a tecnologia de engenharia de software nos próximos 10 ou 20 anos.

38.1 Evolução da tecnologia

Em um livro fascinante que traz uma visão atraente sobre como as tecnologias de computação (e outras tecnologias relacionadas) vão evoluir, Ray Kurzweil [Kur05] afirma que a evolução tecnológica é similar à biológica, mas ocorre a uma velocidade de magnitude muitas vezes maior. A evolução (seja ela biológica, seja tecnológica) ocorre em resultado de uma realimentação positiva – “os métodos mais capacitados resultantes de um estágio do progresso evolucionário são usados para criar o próximo estágio” [Kur05].

As grandes questões para o século 21 são: (1) Quão rápido uma tecnologia evolui? (2) Quão significativos são os efeitos da realimentação positiva? (3) Quão profundas serão as mudanças resultantes?

Quando uma tecnologia bem-sucedida é introduzida, o conceito inicial transforma-se em um “ciclo de vida da inovação” razoavelmente previsível [Gai95], ilustrado na Figura 38.1. Na fase de *avanço*, um problema é identificado e tentativas repetidas em busca de uma solução viável são realizadas. Em algum ponto, aparece uma promessa de solução. O trabalho de avanço inicial é reproduzido na fase *replicador* e ganha um uso mais amplo. O *empirismo* leva à criação de regras que regem o uso da tecnologia, e o sucesso repetido leva a uma *teoria* de uso mais amplo, seguida pela criação de ferramentas automatizadas durante a fase da *automação*. Por fim, a tecnologia amadurece e passa a ser amplamente utilizada.

Quais são as “grandes questões” quando consideramos a evolução da tecnologia?

A tecnologia da computação está evoluindo a uma taxa exponencial, e seu crescimento pode logo se tornar explosivo.

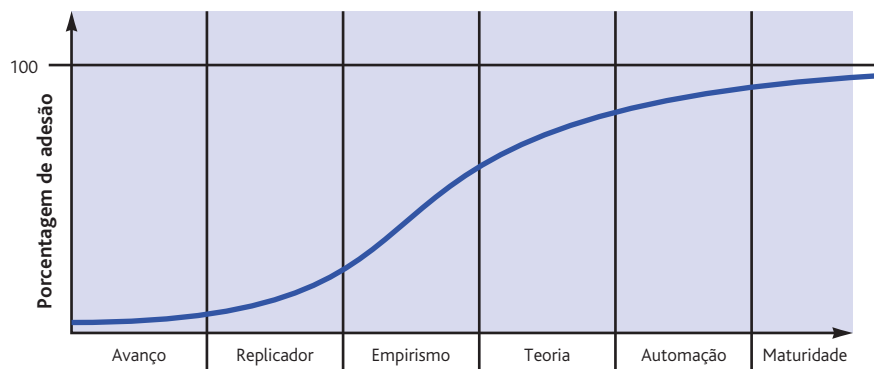


FIGURA 38.1 Um ciclo de vida da evolução tecnológica.

É importante notar que muitas pesquisas e tendências tecnológicas nunca atingem a maturidade. Na verdade, a grande maioria das tecnologias “promissoras” no domínio da engenharia de software suscita um grande interesse por alguns anos e depois passa a ser usada por um grupo específico de usuários. Isso não significa que elas não tenham mérito, mas que o caminho da inovação é longo e difícil.

Kurzweil [Kur05] concorda que as tecnologias de computação evoluem em uma “curva-S”, que apresenta um crescimento relativamente lento durante os anos de formação da tecnologia, uma rápida aceleração durante a fase de crescimento e depois um período de nivelamento, quando atinge seus limites. Hoje, estamos acelerando rapidamente no joelho da curva-S das modernas tecnologias de computação – na transição entre o crescimento inicial e o crescimento explosivo que deve se seguir. A implicação é que, durante os próximos 20 a 40 anos, veremos mudanças significativas (até mesmo impressionantes) na capacidade de computação. Kurzweil sugere que, em 20 anos, a evolução tecnológica vai acelerar a um ritmo cada vez mais rápido, chegando finalmente à era da inteligência não biológica que se combinará com a inteligência humana e a ampliará de maneira fascinante.

E tudo isso, não importa como evolua, precisará de software e sistemas que fazem nossos esforços atuais parecerem infantis. Por volta de 2040, uma combinação de computação extrema, nanotecnologia, redes com larguras de banda extremamente altas e robótica nos levará a um mundo diferente.¹ O software, possivelmente de formas que ainda não podemos compreender, continuará existindo no centro desse mundo novo. A engenharia de software não vai acabar.

38.2 Perspectivas para uma verdadeira disciplina de engenharia

Por quase 50 anos, muitos pesquisadores acadêmicos e profissionais do setor têm clamado por uma verdadeira disciplina de engenharia para software. Em um importante seguimento de seu artigo clássico de 1990 sobre o tema, Mary Shaw [Sha09] comenta sobre essa busca contínua:

As disciplinas de engenharia normalmente evoluem de exercícios práticos de uma tecnologia, suficientes para uso local ou improvisado. Quando a tecnologia se torna economicamente importante, ela exige técnicas de produção estáveis e controle gerenciado. O mercado comercial resultante se baseia na experiência e não em um entendimento aprofundado da tecnologia... uma profissão de engenharia surge quando... a ciência se torna suficientemente madura para suportar prática intencional e evolução de projeto, com resultados previsíveis.

¹ Kurzweil [Kur05] apresenta um argumento técnico que prevê forte inteligência artificial (que passará pelo teste de Turing) em 2029, e sugere que a evolução de humanos e máquinas começará a se combinar em 2045. A grande maioria dos leitores deste livro viverá para ver se isso de fato ocorrerá.

Diríamos que a indústria atingiu a “prática intencional”, mas que os “resultados previsíveis” permaneceram ilusórios.

À medida que aplicativos móveis e WebApps começam a dominar o cenário do software, Shaw identifica desafios que “surtem das profundas interdependências entre sistemas muito complexos e seus usuários” [Sha09]. Ela argumenta que a base de conhecimento que leva à “prática intencional” foi democratizada pelas redes sociais especializadas que agora povoam a Web. Por exemplo, em vez de consultar um manual de engenharia de software controlado de forma centralizada, um engenheiro de software pode apresentar um problema em um fórum apropriado e obter uma solução com contribuições de todo o mundo, extraída da experiência de muitos outros desenvolvedores. A solução proposta é frequentemente criticada em tempo real, com alternativas e adaptações oferecidas como opções.

Mas esse não é o nível de disciplina que muitos exigem. Conforme diz Shaw: “IOsl problemas enfrentados pelos engenheiros de software estão cada vez mais situados em contextos sociais complexos e é cada vez mais difícil delinear os limites do problema” [Sha09]. Como resultado, isolar os fundamentos científicos de uma disciplina continua a ser um desafio. Neste ponto da história de nossa área, é razoável dizer que “a descoberta de novas ideias de engenharia de software é, a esta altura, naturalmente incremental e evolucionária” [Erd10].

38.3 Observação de tendências na engenharia de software

Barry Boehm [Boe08] sugere que “os engenheiros de software enfrentarão os desafios, frequentemente enormes, de lidar com rápidas mudanças, incerteza e emergência, confiabilidade, diversidade e interdependência, mas também terão oportunidades de fazer contribuições significativas”. Mas quais são as tendências que vão permitir enfrentar esses desafios nos próximos anos?

Na introdução deste capítulo, afirmamos que “as tendências leves” têm um impacto significativo sobre a direção geral da engenharia de software. Mas outras tendências (“mais pesadas”) orientadas para a tecnologia e pesquisa permanecem importantes. As tendências nas pesquisas “são motivadas por percepções gerais do estado da arte e da prática, por percepções do pesquisador sobre as necessidades dos profissionais, por programas de fundos nacionais que buscam estratégias específicas e por puro interesse técnico” [Mil00b]. Tendências tecnológicas ocorrem quando pesquisas são extrapoladas para atender às necessidades da indústria e demandas do marketing.

Na Seção 38.1, discutimos o modelo da curva-S para a evolução da tecnologia. A curva-S é apropriada para considerar os efeitos de longo prazo das tecnologias básicas, à medida que evoluem. Mas o que ocorre com as inovações, ferramentas e métodos mais modestos e de curto prazo? O Gartner Group [Gar08] – uma consultoria que estuda tendências da tecnologia em muitos ramos de atividade – desenvolveu um *ciclo da excelência para tecnologias emergentes*, representado na Figura 38.2. Nem toda tecnologia

“Acredito que possa haver um mercado mundial talvez para cinco computadores.”

Thomas Watson,
presidente da IBM,
1943

O “ciclo da excelência” apresenta uma visão realista da integração da tecnologia no curto prazo. No entanto, a tendência no longo prazo é exponencial.

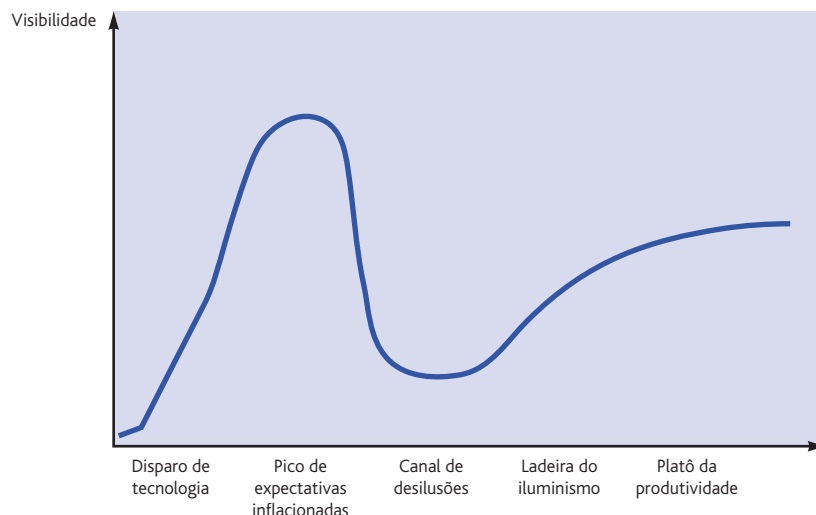


FIGURA 38.2 O ciclo da excelência para tecnologias emergentes do Gartner Group [Gar08].

de engenharia de software segue esse caminho ao longo do ciclo da excelência. Em alguns casos, a decepção é justificada e a tecnologia fica relegada à obscuridade.

38.4 Identificação das “tendências leves”

Todo país com uma indústria de TI considerável tem um conjunto único de características que definem o modo como os negócios são conduzidos, as dinâmicas organizacionais que surgem dentro de uma empresa, os aspectos de marketing que se aplicam aos clientes locais e a cultura dominante que governa toda a interação humana. No entanto, algumas tendências em cada uma dessas áreas são universais e têm tanto a ver com sociologia, antropologia e psicologia de grupo (muitas vezes chamadas de “ciências leves”) quanto com pesquisa acadêmica ou industrial.

Conectividade e colaboração (possibilitadas pelas comunicações em banda larga) já permitem a existência de equipes de software que não ocupam o mesmo espaço físico (trabalho à distância e emprego de meio turno em um contexto local). Uma equipe colabora com outras que estão separadas por fuso horário, linguagem e cultura. A engenharia de software deve responder com um modelo de processo abrangente para “equipes distribuídas”, que seja ágil o bastante para atender às demandas de urgência, mas disciplinado o suficiente para coordenar diferentes grupos.

A *globalização* resulta em uma força de trabalho diversificada (em termos de idioma, cultura, solução de problemas, filosofia de administração, prioridades de comunicação e interação entre as pessoas). Isso, por sua vez, exige uma estrutura organizacional flexível. Equipes diferentes (em países diferentes) devem responder a problemas de engenharia da maneira que melhor atenda às suas necessidades especiais, enquanto proporciona uniformidade para a execução geral de um projeto global. Esse tipo de organização sugere menos níveis de administração e mais ênfase na tomada de

“640K deve ser suficiente para todos.”

Bill Gates,
presidente da
Microsoft, 1981

decisões em equipe. Ele pode levar a uma maior agilidade, mas apenas se mecanismos de comunicação forem estabelecidos para que cada equipe possa entender o projeto e o status técnico (via conexão em rede) a qualquer momento. Métodos e ferramentas da engenharia de software podem contribuir para a obtenção de algum nível de uniformidade (equipes falam a mesma “linguagem” implementada por métodos e ferramentas específicos). O processo de software pode proporcionar o framework para a existência desses métodos e ferramentas.

Em algumas regiões (nos Estados Unidos e na Europa, por exemplo), a população está envelhecendo. Essa inegável tendência demográfica (e cultural) indica que muitos engenheiros de software e gerentes experientes estarão deixando o mercado de trabalho durante a próxima década. A comunidade de engenharia de software deve responder com mecanismos viáveis que capturem o conhecimento desses gerentes e técnicos – por exemplo, o uso de *padrões* (Capítulo 16) é um passo na direção certa – para que eles fiquem à disposição das gerações futuras de profissionais. Em outras regiões do mundo, o número de jovens disponíveis para a indústria de software está explodindo. É a oportunidade de moldar uma cultura de engenharia de software sem o ônus de 50 anos de prejuízos da “velha escola”.

Estima-se que mais de um bilhão de novos consumidores entrarão no mercado mundial na próxima década. Os gastos dos consumidores nas economias emergentes passarão para mais de \$ 12 trilhões em 10 anos [ATK12]. Não há dúvida de que uma porcentagem significativa desse gasto será com produtos e serviços que tenham um componente digital – que são baseados em software ou controlados por software. A implicação é uma demanda crescente por novos softwares. A questão então é: “Novas tecnologias de engenharia de software podem ser desenvolvidas para atender a essa demanda mundial?”. As tendências do mercado moderno muitas vezes são controladas pelo fornecimento.² Em outros casos, requisitos da demanda controlam o mercado. Em qualquer caso, um ciclo de inovação e demanda ocorre de maneira que muitas vezes dificulta determinar quem vem primeiro!

Por fim, a própria cultura humana terá impacto na direção da engenharia de software. Toda geração deixa sua própria marca na cultura local, e a nossa não será diferente. Faith Popcorn [Pop08], conhecido consultor especializado em tendências culturais, caracteriza-as da seguinte maneira: “Nossas Tendências não são modismos. Nossas Tendências permanecem. Nossas Tendências evoluem. Elas representam forças subjacentes, causas principais, necessidades humanas básicas, atitudes, aspirações. Elas nos ajudam a navegar pelo mundo, entender o que está acontecendo e por quê, e a nos prepararmos para aquilo que ainda virá”. Uma discussão detalhada sobre como as tendências culturais modernas terão impacto sobre a engenharia de software é mais bem apresentada por aqueles que se especializaram nas “ciências leves”.

² O lado fornecedor adota nos mercados uma abordagem do tipo “faça e eles comprarão”. Tecnologias especiais são criadas, e os consumidores se aglomeram para adotá-las – às vezes!

38.4.1 Gestão da complexidade

Quando a 1ª edição deste livro foi escrita (em 1982), produtos digitais da forma como conhecemos hoje não existiam, e os sistemas baseados em mainframe contendo um milhão de linhas de código-fonte (LOC, lines of source code) eram considerados muito grandes. Hoje, não é raro encontrar pequenos dispositivos digitais com 60 mil a 200 mil linhas de código-fonte de software personalizado, com alguns milhões de linhas de código para recursos do sistema operacional. Sistemas modernos baseados em computador contendo de 10 a 50 milhões de linhas de código não são incomuns.³ Em um futuro relativamente próximo, começarão a surgir sistemas⁴ que exigirão mais de 1 bilhão de linhas de código.⁵

Pense nisso por um instante!

Considere as interfaces para um sistema de 1 bilhão de linhas de código, para o mundo exterior, para outros sistemas interoperáveis, para a Internet (ou seu sucessor) e para os milhões de componentes internos que devem funcionar todos juntos para fazer esse monstro da computação operar corretamente. Há uma maneira confiável de garantir que todas essas conexões permitam que as informações fluam adequadamente?

Considere o próprio projeto. Como gerenciamos o fluxo do trabalho e acompanhamos o progresso? As estratégias convencionais poderão ser escaladas muitas vezes em ordem de grandeza?

Considere o número de pessoas (e sua localização) que estará fazendo o trabalho, a coordenação dos profissionais e da tecnologia, o fluxo ininterrupto de alterações, a possibilidade de ambiente de sistema multiplataforma, multioperacional. Há uma maneira de gerenciar e coordenar indivíduos que estão trabalhando em um projeto enorme?

Considere o desafio da engenharia. Como podemos analisar dezenas de milhares de requisitos, limitações e restrições de uma forma que garanta que as inconsistências e ambiguidades, omissões e erros sejam imediatamente descobertos e corrigidos? Como podemos criar uma arquitetura de projeto que seja robusta o bastante para lidar com um sistema desse tamanho? Como os engenheiros de software poderão estabelecer um sistema de controle de alterações que terá de manipular centenas de milhares de alterações?

Considere o desafio da garantia da qualidade. Como podemos realizar verificação e validação de modo significativo? Como você testa um sistema de 1 bilhão de LOC?

Nos primórdios da engenharia de software, as tentativas de gerenciar a complexidade só poderiam ser descritas como *ad hoc*. Hoje, usamos processos, métodos e ferramentas para manter a complexidade sob controle. Mas o que acontecerá no futuro? Nossa abordagem atual estará à altura da tarefa a ser realizada?

"Não há razão para uma pessoa querer ter um computador em casa."

Ken Olson,
presidente e
fundador da Digital
Equipment Corp.,
1977

³ Os sistemas operacionais modernos dos PCs (por exemplo, Linux, Mac OS e Windows) têm de 30 a 60 milhões de LOC. Sistemas operacionais de dispositivos móveis podem ter mais de 2 milhões de LOC.

⁴ Na realidade, esse "sistema" será um sistema de sistemas – centenas de aplicativos interoperáveis trabalhando juntos para atingir algum objetivo comum.

⁵ Nem todos os sistemas complexos são grandes. Um aplicativo relativamente pequeno (digamos, menos de 100 mil LOC) pode ainda ser bastante complexo.

Software aberto
abrange inteligência
ambiente, aplicações
sensíveis ao contexto
e computação
generalizada.

38.4.2 Software aberto

Conceitos como inteligência ambiente,⁶ aplicações sensíveis ao contexto e computação pervasiva/ubíqua – todos focam a integração de sistemas baseados em software em um ambiente mais amplo do que um PC, um dispositivo de computação móvel ou qualquer outro dispositivo digital. Essas visões separadas do futuro próximo da computação sugerem coletivamente o “software aberto” – software projetado para se adaptar a um ambiente em contínua mudança “auto-organizando sua estrutura e autoadaptando seu comportamento” [Bar06b].

Para ajudar a ilustrar os desafios que os engenheiros de software enfrentarão em um futuro previsível, considere a noção de *inteligência ambiente* (amI, *ambient intelligence*). Ducatel [Duc01] define a amI da seguinte maneira: “Pessoas estão rodeadas por interfaces inteligentes e intuitivas que estão embarcadas em todos os tipos de objetos. O espaço da inteligência ambiente é capaz de reconhecer e responder à presença de diferentes indivíduos enquanto trabalham de uma maneira contínua e sem obstrução”.

Com o uso difundido de smartphones de baixo custo, e, ainda assim, cada vez mais poderosos, estamos a caminho dos sistemas amI onipresentes. O desafio para os engenheiros de software é desenvolver aplicativos que forneçam funcionalidade cada vez maior em produtos de todos os tipos – funcionalidade que se adapte às necessidades do usuário e, ao mesmo tempo, proteja a privacidade e ofereça segurança.

38.4.3 Requisitos emergentes

No início de um projeto de software, há um clichê que se aplica igualmente a todos os envolvidos: “Você não sabe o que não sabe”. Isso significa que os clientes raramente definem requisitos “estáveis”. Indica também que os engenheiros de software não podem prever onde ocorrerão as ambiguidades e inerências. Os requisitos mudam – mas isso não é novidade.

À medida que os sistemas se tornam mais complexos, até mesmo uma tentativa rudimentar de definir requisitos abrangentes está destinada ao fracasso. Uma definição de objetivos globais pode ser possível, um esboço dos objetivos intermediários pode ser alcançado, mas requisitos estáveis... sem chance! Os requisitos surgirão conforme todos os envolvidos na engenharia e construção de um sistema complexo aprenderem mais sobre esse sistema, sobre o ambiente no qual ele reside e sobre os usuários que vão interagir.

Essa realidade implica uma série de tendências de engenharia de software. Primeiro, devem ser projetados modelos de processo para aderirem à mudança e adotarem os princípios básicos da filosofia ágil (Capítulo 5). Em seguida, métodos que resultem em modelos elaborados (por exemplo, modelos de requisito e de projeto) devem ser usados criteriosamente, porque esses modelos mudarão repetidamente à medida que mais conhecimento sobre o

*Como os requisitos
emergentes já são
uma realidade, sua
organização deve
pensar em adotar um
modelo de processo
incremental.*

⁶ Uma introdução muito boa e detalhada sobre *inteligência ambiente* pode ser encontrada em www.emergingcommunication.com/volume6.html. Mais informações podem ser obtidas em www.ambientintelligence.org/.

sistema for adquirido. Por fim, ferramentas que suportem tanto os processos quanto os métodos devem facilitar a adaptação e a mudança.

Há outro aspecto dos requisitos emergentes, no entanto. Para a grande maioria do software desenvolvido até hoje, a fronteira entre o sistema baseado em software e seu ambiente externo é estável. A fronteira pode mudar, mas isso ocorrerá de maneira controlada, permitindo que o software seja adaptado como parte de um ciclo de manutenção. Essa opinião está começando a mudar. O software aberto (Seção 38.4.2) precisa “se adaptar e reagir dinamicamente às mudanças, mesmo que sejam imprevistas” [Bar06b].

Por sua natureza, os requisitos emergentes levam à mudança. Como controlamos a evolução de um aplicativo ou sistema amplamente usado durante toda a sua vida útil, e que efeito isso tem sobre a maneira como projetamos software?

Conforme o número de alterações aumenta, a probabilidade de efeitos colaterais não desejados também aumenta. Isso deverá ser motivo de preocupação quando sistemas complexos com requisitos emergentes se tornarem comuns. A comunidade de engenharia de software deve desenvolver métodos que ajudem as equipes a prever o impacto das mudanças no sistema inteiro, moderando, assim, efeitos colaterais indesejados. Hoje, nossa habilidade para tanto é seriamente limitada.

38.4.4 O mix de talentos

A natureza de uma equipe de engenharia de software pode mudar à medida que os sistemas baseados em software ficarem mais complexos, as comunicações e a colaboração entre equipes globais tornarem-se lugar-comum e os requisitos emergentes (com o fluxo de mudanças resultantes) se tornarem a norma. Cada equipe de software deve contribuir com talento criativo e habilidades técnicas para sua parte de um sistema complexo, e o processo todo deve permitir que o resultado dessas ilhas de talento se combine de modo eficiente.

Alexandra Weber-Morales [Mor05] sugere o mix de talentos de uma “equipe de software dos sonhos”. *Brain* é o arquiteto-chefe, capaz de lidar com as demandas dos envolvidos e mapeá-las em um framework tecnológico extensível e implementável. *Data Grrl* é a guru do banco de dados e da estrutura de dados que “ataca vigorosamente as linhas e colunas, com profundo conhecimento da lógica de predicados e da teoria dos conjuntos no que se refere ao modelo relacional”. *Blocker* é um líder técnico (gerente) que permite que a equipe trabalhe sem interferências e, ao mesmo tempo, garante que a colaboração esteja ocorrendo. *Hacker* é o programador perfeito que conhece padrões e linguagens e pode usar ambas de modo eficaz. O *Gatherer* “descobre habilmente requisitos de sistema com... visão antropológica” e os expressa com clareza.

38.4.5 Blocos básicos de software

Todos nós que adotamos uma filosofia de engenharia de software já enfatizamos a necessidade de reutilização – de código-fonte, classes orientadas a objetos, componentes, padrões e software de prateleira. Embora a comunidade de engenharia tenha feito progresso em sua tentativa de capturar conheci-

mento e reutilizar soluções aprovadas, uma parcela significativa do software continua a ser criada “do zero”. Parte da razão para isso é um contínuo desejo (por parte dos envolvidos e dos profissionais de engenharia de software) por “soluções únicas”.

No mundo do hardware, os OEMs (original equipment manufacturers) dos dispositivos digitais usam quase que exclusivamente produtos-padrão específicos de aplicativo (ASSPs, application-specific standard products), produzidos por fabricantes de circuitos integrados. Esse “hardware comercial” fornece os blocos básicos necessários para implementar tudo, desde um smartphone até um dispositivo de computação vestíveis. Cada vez mais, os mesmos OEMs usam o “software comercial” – blocos básicos de software projetados especificamente para um domínio de aplicação único (por exemplo, dispositivos VoIP). Michael Ward [War07] comenta:

Uma vantagem do uso de componentes de software é que o OEM pode alavancar a funcionalidade proporcionada pelo software sem ter de desenvolver especialidades internas nas funções específicas ou investir tempo de desenvolvedor no trabalho de implementação e validação dos componentes. Outras vantagens incluem a capacidade de adquirir e fornecer apenas o conjunto específico de funcionalidades necessárias ao sistema, bem como de integrar esses componentes em uma arquitetura já existente.

Além dos componentes empacotados como software comercial, há uma tendência crescente em adotar *soluções de plataforma de software* que “incorporam conjuntos de funcionalidades relacionadas, fornecidas tipicamente em um framework de software integrado” [War07]. Uma plataforma de software isenta um OEM do trabalho associado ao desenvolvimento de funcionalidade básica e, em vez disso, permite que ele dedique trabalho de software para as características que diferenciam seu produto.

38.4.6 Mudança na percepção de “valor”

Nos últimos 25 anos do século 20, a pergunta importante que os homens de negócio faziam ao discutirem software era: “Por que custa tão caro?”. Essa pergunta raramente é feita hoje, e foi substituída por outra: “Por que não podemos obtê-lo (software e/ou produto baseado em software) mais rapidamente?”.

Considerando software para computador, nota-se que a percepção moderna está mudando do valor nos negócios (custo e lucratividade) para os valores de clientes, que incluem: agilidade na entrega, riqueza de funcionalidade e qualidade geral do produto.

38.4.7 Código aberto

Quem é o proprietário do software que você ou sua organização utiliza? Cada vez mais, a resposta é “todos”. O movimento “código aberto” (*open source*) tem sido descrito da seguinte maneira [OSO12]: “Código aberto é um método de desenvolvimento de software que utiliza o poder da revisão em dupla distribuída e a transparência do processo. A premissa do código aberto é melhor qualidade,

maior confiabilidade, maior flexibilidade, menor custo e o fim do aprisionamento tecnológico predatório”. O termo *código aberto*, quando aplicado a software de computador, implica que os produtos de engenharia de software (modelos, código-fonte, conjuntos de teste) são abertos ao público e podem ser revistos e ampliados (com controles) por qualquer um com interesse e permissão.

Se tiver mais interesse, Weber [Web05] fornece uma introdução valiosa, Feller e seus colegas [Fel07] editaram uma antologia abrangente e objetiva que considera os benefícios e problemas associados a código aberto e Brown [Bro12] oferece uma discussão mais técnica.

“Mas para que serve?”

Engenheiro da divisão de computação avançada da IBM, 1968, comentando sobre um microchip

38.5 Rumos da tecnologia

Parece que sempre pensamos que a engenharia de software mudará mais rapidamente do que de fato muda. Uma nova tecnologia “da moda” (poderia ser um novo processo, um método especial ou uma ferramenta interessante) é introduzida, e os especialistas sugerem que “tudo” mudará. Mas a engenharia de software é muito mais do que tecnologia – ela envolve pessoas e suas habilidades em comunicar necessidades e inovar para tornar aquelas necessidades uma realidade. Sempre que há pessoas envolvidas, as mudanças ocorrem lentamente, aos trancos e barrancos. Apenas quando se atinge um “ponto de virada” [Gla02] é que uma tecnologia se propaga pela comunidade de engenharia de software e realmente ocorre uma ampla mudança.

“A resposta criativa para a tecnologia digital é adotá-la como uma nova janela sobre tudo o que é eternamente humano e usá-la com paixão, sabedoria, bravura e alegria.”

Ralph Lombreglia

Nesta seção, examinaremos algumas tendências em processos, métodos e ferramentas que podem ter alguma influência sobre a engenharia de software durante a próxima década. Elas conduzirão a um ponto de virada? Temos de esperar para ver.

38.5.1 Tendências de processo

Podemos dizer que todas as tendências de negócios, organizacionais e culturais discutidas na Seção 38.4 reforçam a necessidade de processo. Mas os frameworks abordados no Capítulo 37 fornecem um roteiro para o futuro? As metodologias de processo vão evoluir para buscar um melhor equilíbrio entre a disciplina e a criatividade? Os processos de software se adaptarão às diferentes necessidades dos envolvidos que solicitam o software, aqueles que o criam e aqueles que o utilizam? O software pode proporcionar um meio de reduzir o risco dos três componentes ao mesmo tempo?

Essas e outras questões permanecem pendentes. Nos próximos parágrafos, adaptaremos seis ideias propostas por Conradi e Fuggetta [Con02] para sugerir possíveis tendências de processo.

1. À medida que os frameworks SPI evoluírem, darão ênfase a “estratégias que focalizam a orientação e inovação de produto” [Con02]. Em um mundo de rápidas mudanças no desenvolvimento de software, estratégias SPI de longo prazo raramente sobrevivem em um ambiente de negócios dinâmicos. Há muitas mudanças ocorrendo rapidamente. Isso significa que um roteiro estável, passo a passo, para SPI talvez precise ser substituído

Quais são as tendências de processo mais prováveis na próxima década?

por um framework que enfatize os objetivos de curto prazo com uma orientação para produto.

2. *Como os engenheiros de software têm uma boa noção do ponto frágil do processo, as mudanças em geral deverão ser motivadas por suas necessidades e começar de baixo para cima.* Conradi e Fuggetta [Con02] sugerem que as atividades de SPI no futuro deverão “usar critérios de avaliação simples e focalizados para começar, e não uma ampla avaliação”. Concentrando-se nos esforços de SPI de forma restrita e trabalhando de baixo para cima, os profissionais começarão logo a ver mudanças substanciais em como é conduzido o trabalho de engenharia de software.
3. *A tecnologia de processo automatizado de software (SPT, automated software process technology) se distanciará do gerenciamento global de processo (suporte com base ampla de todo o processo de software) para concentrar-se nos aspectos que podem se beneficiar mais da automação.* Ninguém é contra ferramentas e automação, mas em muitas situações a SPT não atingiu seu objetivo (veja a Seção 38.3). Para maior eficácia, ela deverá focalizar atividades de apoio (Capítulo 3) – os elementos mais estáveis do processo de software.
4. *Haverá mais ênfase ao retorno sobre o investimento das atividades de SPI.* No Capítulo 37, você aprendeu que o retorno sobre o investimento (ROI, return on investment) pode ser definido como:

$$\text{ROI} = \frac{\Sigma (\text{benefícios}) - \Sigma (\text{custos})}{\Sigma (\text{custos})} \times 100\%$$

Até hoje, as organizações de software têm se empenhado para delinear claramente os “benefícios” de maneira quantitativa. Pode-se afirmar [Con02] que “precisamos, portanto, de um modelo padronizado de valor de mercado... para levar em conta as iniciativas de melhorias de software”.

5. *À medida que o tempo passa, a comunidade do software pode vir a entender que a especialização em sociologia e antropologia pode ter tanto ou muito mais a ver com uma SPI bem-sucedida quanto outras disciplinas mais técnicas.* Além disso, a SPI muda a cultura organizacional, e a mudança cultural envolve indivíduos e grupos de profissionais. Conradi e Fuggetta [Con02] observam corretamente que “os desenvolvedores de software são trabalhadores de conhecimento. Tendem a responder negativamente a ordem de cima sobre como fazer o trabalho ou como mudar processos”. Pode-se aprender muito examinando a sociologia de grupos para melhor entender as maneiras eficazes de introduzir uma mudança.
6. *Novos modos de aprender podem facilitar a transição para um processo de software mais eficaz.* Nesse contexto, “aprender” implica aprender com sucessos e erros. Uma organização de software que coleta métricas (Capítulos 30 e 32) permite a si mesma entender como os elementos de um processo afetam a qualidade do produto final.

38.5.2 O grande desafio

Existe uma tendência que é inegável: os sistemas baseados em software se tornarão maiores e mais complexos com o passar do tempo. É a engenharia desses sistemas grandes e complexos, independentemente da plataforma ou do domínio de aplicação, que apresenta o “grande desafio” [Bro06] para os engenheiros de software. Manfred Broy [Bro06] afirma que os engenheiros de software podem enfrentar “o desafio assustador do desenvolvimento de complexos sistemas de software”, criando novas abordagens para entender modelos de sistema e usando esses modelos como base para a construção de software de alta qualidade da nova geração.

Conforme a comunidade de engenharia de software desenvolve novas abordagens motivadas por modelo (assunto discutido brevemente mais adiante nesta seção) para a representação de requisitos de sistema e projeto, as seguintes características devem ser tratadas [Bro06]:

- *Multifuncionalidade* – À medida que os dispositivos digitais evoluíram, começaram a apresentar um amplo conjunto de funções às vezes não relacionadas. O telefone celular, antes considerado um aparelho de comunicação simples, tornou-se um poderoso computador de bolso que executa um amplo espectro de funções comprovadamente mais importantes do que fazer uma ligação telefônica. Conforme observa Broy [Bro06], “[os] engenheiros devem descrever o contexto detalhado no qual as funções serão fornecidas e, mais importante, devem identificar as interações potencialmente perigosas entre diferentes características do sistema”.
- *Reatividade e temporização (timeliness)* – Os dispositivos digitais interagem cada vez mais com o mundo real e devem reagir aos estímulos externos no tempo. Eles devem estabelecer interface com um amplo conjunto de sensores e devem responder em um intervalo de tempo apropriado para a tarefa em questão. Devem ser desenvolvidos novos métodos que (1) ajudem os engenheiros de software a prever a cadência das várias características reativas e (2) implementem características menos dependentes da máquina e mais portáteis.
- *Novos modos de interação do usuário* – As tendências abertas para software significam que novos modos de interação devem ser modelados e implementados. Independentemente do fato de essas novas abordagens usarem interfaces multitoque, reconhecimento de voz ou interfaces mentais⁷ diretas, as novas gerações de software para dispositivos digitais devem acomodá-las.
- *Arquiteturas complexas* – Um automóvel de luxo tem mais de duas mil funções controladas por software residente em uma arquitetura de hardware complexa, incluindo múltiplos processadores, estrutura de barramento sofisticada, atuadores, sensores e interfaces humanas cada vez

Que características de sistema os analistas e projetistas devem considerar para futuras aplicações?

⁷ Uma breve discussão sobre interfaces mentais diretas pode ser encontrada em http://en.wikipedia.org/wiki/Brain-computer_interface e um exemplo comercial que continua a evoluir é descrito em <http://au.gamespot.com/news/6166959.html>

mais sofisticados e muitos componentes relacionados à segurança. Sistemas ainda mais complexos estão no horizonte próximo, apresentando desafios significativos para os projetistas de software.

- *Sistemas heterogêneos distribuídos* – Os componentes de tempo real de qualquer sistema embarcado moderno podem ser conectados por meio de um barramento interno, de uma rede sem fio ou da Internet (ou as três coisas).
- *Criticidade* – O software tornou-se o componente central em todos os sistemas críticos nos negócios e em muitos sistemas em termos de segurança. Contudo, a comunidade de engenharia de software apenas começou a aplicar os princípios mais básicos de segurança de software.
- *Variabilidade de manutenção* – A vida do software em um dispositivo digital raramente dura além de 3 a 5 anos, mas os sistemas complexos de avionica instalados em uma aeronave têm uma vida útil de pelo menos 20 anos. O software dos automóveis fica em algum ponto intermediário. Isso deverá ter um impacto sobre o projeto?

Broy [Bro06] afirma que essas e outras características do software podem ser gerenciadas somente se a comunidade desenvolver uma filosofia de engenharia de software distribuída e colaborativa mais eficaz, melhores abordagens de requisitos de engenharia, uma abordagem mais robusta do desenvolvimento motivado por modelo e melhores ferramentas de software. Nas próximas seções, exploraremos rapidamente cada uma dessas áreas.

38.5.3 Desenvolvimento colaborativo

A colaboração envolve a disseminação ao longo do tempo e um processo eficaz para comunicação e criação de projeto.

Parece muito óbvio, mas diremos mesmo assim: *a engenharia de software é uma tecnologia de informação*. Desde o início de qualquer projeto de software, todos os envolvidos devem compartilhar informações – sobre metas e objetivos básicos, sobre requisitos básicos de sistema, sobre problemas de projeto de arquitetura, sobre quase todos os aspectos do software a ser criado.

Hoje, os engenheiros de software colaboram em diferentes fusos horários e diferentes países. Todos devem compartilhar informações. O mesmo vale para projetos abertos, nos quais centenas ou milhares de desenvolvedores de software trabalham para criar uma aplicação aberta. Novamente, as informações devem ser disseminadas para que possa ocorrer a colaboração aberta.

38.5.4 Engenharia de requisitos

As ações básicas de engenharia de requisitos (RE, *requirements engineering*) – levantamento, elaboração, negociação, especificação e validação – foram apresentadas nos Capítulos 8 a 11. O sucesso ou fracasso dessas ações têm forte influência sobre o sucesso ou fracasso de todo o processo de engenharia de software. Contudo, a RE tem sido comparada a “tentar colocar uma braçadeira de mangueira em gelatina” [Gon04]. Conforme já mencionamos em várias partes deste livro, os requisitos de software tendem a continuar mudando e, com o surgimento dos sistemas abertos, requisitos emergentes (e mudanças quase contínuas) podem se tornar algo comum.

Hoje, muitas abordagens “informais” de RE começam com a criação de cenários de usuário (por exemplo, casos de uso). Abordagens mais informais criam um ou mais modelos de requisitos e os utilizam como base para o projeto. Métodos formais permitem que o engenheiro de software represente requisitos usando uma notação matemática que pode ser verificada. Tudo pode funcionar razoavelmente bem quando os requisitos são estáveis, mas não resolve imediatamente o problema dos requisitos dinâmicos ou emergentes.

Existem várias e distintas direções de pesquisa de engenharia de requisitos, incluindo processamento de linguagem natural com base em descrições textuais traduzidas para representações mais estruturadas, maior confiabilidade em bancos de dados para estruturação e entendimento de requisitos de software, uso de padrões de RE para descrever problemas típicos e soluções quando são executadas as tarefas de engenharia de requisitos e engenharia de requisitos orientada para metas. No entanto, no nível industrial, ações de RE permanecem relativamente informais e surpreendentemente básicas. Para melhorar a maneira como os requisitos são definidos, a comunidade de engenharia de software provavelmente implementará três subprocessos enquanto a RE é executada [Gli07]: (1) melhoria da aquisição de conhecimentos e compartilhamento de conhecimentos que possibilite o entendimento mais completo das restrições do domínio de aplicação e necessidades dos envolvidos, (2) maior ênfase na iteração quando os requisitos são definidos e (3) ferramentas mais eficazes de comunicação e coordenação que permitam a todos os envolvidos colaborar eficazmente.

Os subprocessos de RE descritos no parágrafo anterior só terão sucesso se forem integrados adequadamente a uma estratégia evolutiva da engenharia de software. Como a solução de problemas baseada em padrões e as soluções baseadas em componentes começam a conquistar muitos domínios de aplicação, a RE deve conciliar o desejo de agilidade (entrega incremental rápida) e os requisitos emergentes inerentes resultantes. A noção de uma “especificação de software” estática está começando a desaparecer, para ser substituída por “requisitos motivados por valor” [Som05] derivados quando os envolvidos respondem às características e funções fornecidas nos primeiros incrementos de software.

38.5.5 Desenvolvimento de software controlado por modelo

Os engenheiros de software lidam com abstração em quase todas as etapas no processo de engenharia de software. Quando o projeto começa, as abstrações em nível de arquitetura e de componente são representadas e julgadas. Elas devem então ser traduzidas para uma representação de linguagem de programação que transforma o projeto (uma abstração de nível relativamente alto) em um sistema operável com um ambiente de computação específico (baixo nível de abstração). O *desenvolvimento de software controlado por modelo*⁸ acopla linguagens de modelagem específicas de domínio com mecanismo de transformação e geradores para facilitar a representação da abstração em altos níveis e, então, transforma-as em níveis mais baixos [Sch06].

Abordagens controladas por modelo tratam de um desafio contínuo para todos os desenvolvedores de software – como representar o software em um nível de abstração mais alto do que o código.

⁸ O termo *model-driven engineering* (MDE) também é usado.

Linguagens de modelagem específicas de domínio (DSMLs, *domain-specific modeling languages*) representam “estrutura de aplicativo, comportamento e requisitos dentro de domínios de aplicação particulares” e são descritas com metamodelos que “definem as relações entre conceitos no domínio e especificam precisamente a semântica fundamental e as restrições associadas a esses conceitos de domínio” [Sch06]. A principal diferença entre uma DSML e uma linguagem de modelagem de uso geral como a UML (Apêndice 1) é que a DSML está ajustada aos conceitos de projeto inerentes ao domínio de aplicação e pode, portanto, representar relações e restrições entre elementos de projeto de modo eficiente.

38.5.6 Projeto pós-moderno

Em um artigo interessante sobre projeto de software na “era pós-moderna”, Philippe Kruchten [Kru05] faz a seguinte observação:

A ciência da computação ainda não atingiu o contexto geral que consiga explicar sua *totalidade* – ainda não descobrimos as leis fundamentais do software que teriam o mesmo papel das leis fundamentais da física em outras disciplinas de engenharia. Ainda sentimos o gosto amargo do estouro da bolha da Internet e do pesadelo Y2K. Assim, nesta era pós-moderna, em que parece que tudo importa um pouco, embora quase nada importe, quais são os próximos rumos para o projeto de software?

Parte de qualquer tentativa de entender tendências em projeto de software é estabelecer fronteiras para o projeto. Onde termina a engenharia de requisitos e começa o projeto? Onde termina o projeto e começa a geração de código? A resposta a essas questões não é tão fácil como pode parecer. Embora o modelo de requisitos devesse concentrar-se no “que”, não em “como”, todo analista faz um pouco de projeto e quase todos os projetistas fazem um pouco de análise. De forma semelhante, conforme o projeto de componentes de software se aproxima do detalhe algorítmico, um projetista começa a representar o componente em um nível de abstração que se aproxima do código.

38.5.7 Desenvolvimento guiado por teste

Os requisitos controlam o projeto, e o projeto estabelece uma base para a construção. Essa simples realidade da engenharia de software funciona razoavelmente bem e é essencial na criação de uma arquitetura de software. No entanto, uma mudança sutil pode trazer vantagens significativas quando são considerados projeto e construção em nível de componente.

No *desenvolvimento guiado por teste* (TDD, *test-driven development*), requisitos para um componente de software servem de base para a criação de uma série de casos de teste que exercitam a interface e tentam encontrar erros nas estruturas de dados e na funcionalidade fornecida pelo componente. O TDD não é realmente uma nova tecnologia, mas sim uma tendência que enfatiza o projeto de casos de teste *antes* da criação do código-fonte.⁹

O processo TDD segue o fluxo procedural simples ilustrado na Figura 38.3. Antes de ser criado o primeiro segmento de código, um engenheiro de

⁹ Lembre-se de que a Extreme Programming (Capítulo 5) enfatiza essa abordagem como parte de seu modelo de processo ágil.

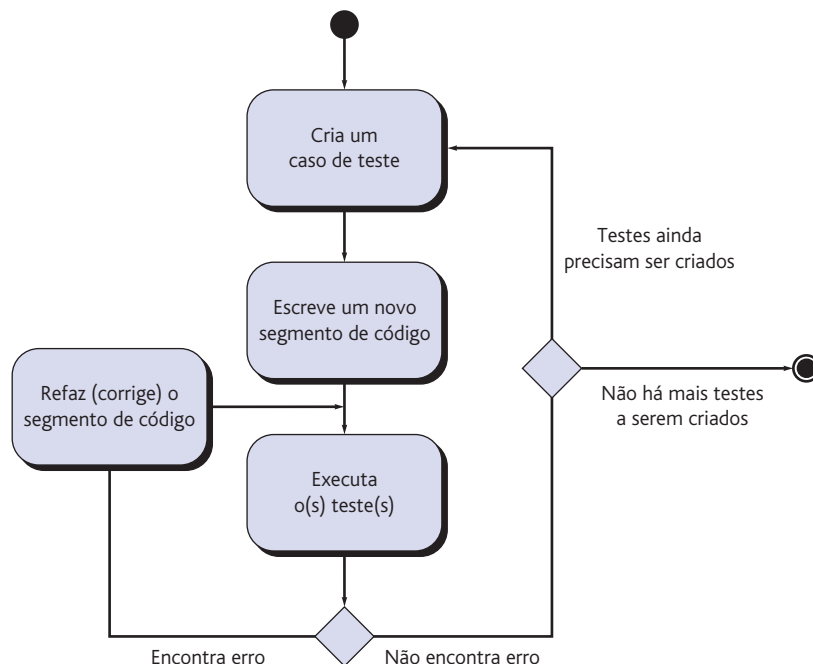


FIGURA 38.3 O fluxo de desenvolvimento controlado por teste.

software cria um teste para exercitar o código (tentando fazer o código falhar). O código é então escrito para satisfazer ao teste. Se passar, um novo teste é criado para o próximo segmento de código a ser desenvolvido. O processo continua até que o componente esteja completamente codificado e todos os testes executem sem erro. No entanto, se algum teste consegue descobrir um erro, o código existente é refeito (corrigido), e todos os testes criados até aquele ponto são executados novamente. Esse fluxo iterativo continua até que não haja mais teste a ser criado, implicando que o componente satisfaz a todos os requisitos definidos para ele.

Durante o TDD, o código é desenvolvido em incrementos muito pequenos (uma subfunção de cada vez) e nenhum código é escrito enquanto não houver um teste para experimentá-lo. Cada iteração resulta em um ou mais novos testes, os quais são acrescentados a um conjunto de testes de regressão que é executado a cada mudança. Isso é feito para garantir que o novo código não tenha gerado efeitos colaterais que causem erros no código anterior. Se tiver mais interesse em TDD, consulte [Bec04b], [Ste10] ou [Whi12].

38.6 Tendências relacionadas a ferramentas

Centenas de ferramentas de engenharia de software de nível industrial são introduzidas no mercado todo ano. A maioria é fornecida por empresas que afirmam que aquela ferramenta vai melhorar o gerenciamento de projeto, ou a análise de requisitos, ou a modelagem do projeto, ou a geração de código, ou o teste, ou gerenciamento de mudanças, ou qualquer uma das muitas atividades, ações e tarefas de engenharia de software discutidas neste livro. Outras foram desenvolvidas como ofertas de código aberto. A maioria das ferramen-

tas de código aberto concentra-se nas atividades de “programação”, com ênfase específica na construção (particularmente a geração de código). Há ainda outras que resultam de esforços de pesquisas em laboratórios de universidades ou do governo. Embora tenham apelo para aplicações bastante limitadas, grande parte não está pronta para a aplicação mais ampla da indústria.

Em nível industrial, os pacotes mais abrangentes formam os *ambientes de engenharia de software* (SEE, *software engineering environments*)¹⁰ que integram um conjunto de ferramentas ao redor de um banco de dados central (repositório). Quando considerado como um todo, um SEE integra informações por meio do processo de software e auxilia na colaboração necessária para muitos sistemas grandes e complexos baseados em software. Mas os ambientes atuais não são facilmente extensíveis (é difícil integrar uma ferramenta COTS que não faz parte do pacote) e tendem a ser de uso geral (não são específicos do domínio de aplicação). Há também um retardo de tempo significativo entre a introdução de novas soluções de tecnologia (por exemplo, desenvolvimento de software controlado por modelo) e a disponibilidade de SEEs viáveis que suportam a nova tecnologia.

As tendências futuras em ferramentas de software seguirão dois caminhos: um *caminho focado no homem*, que responde a algumas das “tendências leves” discutidas na Seção 38.4, e um caminho centrado na tecnologia, que trata de novas tecnologias (Seção 38.5) à medida que são introduzidas e adotadas.

As tendências leves discutidas na Seção 38.4 – a necessidade de gerenciar a complexidade, acomodar requisitos emergentes, estabelecer modelos de processo que aceitam mudanças, coordenar equipes globais com um mix de talentos variável, entre outras coisas – sugerem uma nova era em que suporte de ferramentas para colaboração de envolvidos se tornará tão importante quanto o suporte de ferramentas para tecnologia.

A agilidade na engenharia de software (Capítulo 5) é obtida quando os envolvidos trabalham em equipe. Portanto, a tendência para os SEEs colaborativos produzirá benefícios mesmo quando o software for desenvolvido localmente. Mas qual das ferramentas de tecnologia que complementa o sistema e os componentes que favorecem uma melhor colaboração?

Uma das tendências dominantes é a criação de um conjunto de ferramentas que suporta desenvolvimento controlado por modelo (Seção 38.5.5), com ênfase no projeto controlado por arquitetura. Oren Novotny [Nov04] sugere que o modelo, e não o código-fonte, se torne o foco da engenharia de software:

Modelos independentes de plataforma são criados em UML e então passam por vários níveis de transformação para se transformarem em código-fonte para uma plataforma específica. Pode-se concluir, então, que o modelo, não o arquivo, deverá se tornar a nova unidade de saída. Um modelo tem muitas visões diferentes em diferentes níveis de abstração. No nível mais alto, componentes independentes de plataforma podem ser especificados na análise; no nível mais baixo, há uma implementação específica de plataforma que se reduz a um conjunto de classes no código.

¹⁰ É usado também o termo *ambiente de desenvolvimento integrado* (IDE, *integrated development environment*).

Novotny afirma que uma nova geração de ferramentas funcionará com um repositório para criar modelos em todos os níveis de abstração necessários para estabelecer relações entre os vários modelos, transformar modelos de um nível de abstração em outro (por exemplo, transformar um modelo de projeto em código-fonte), gerenciar alterações e versões e coordenar ações de controle e garantia de qualidade nos modelos de software.

Além dos ambientes completos de engenharia de software, as ferramentas de solução pontual que resolvem tudo, desde reunião de requisitos até refatoração de projeto/código e teste, continuarão a evoluir e terão mais funcionalidades. Em algumas situações, ferramentas de modelagem e teste voltadas para um domínio de aplicação específico proporcionarão um melhor benefício quando comparadas com seus equivalentes genéricos.

38.7 Resumo

As tendências que têm efeito sobre a tecnologia de engenharia de software muitas vezes se originam de cenários de negócios, organizacionais, mercadológicos e culturais. Essas “tendências leves” podem influir na direção da pesquisa e da tecnologia derivada em consequência da pesquisa.

Quando uma nova tecnologia é introduzida, ela passa por um ciclo de vida que nem sempre leva a uma aceitação ampla, apesar de as expectativas originais serem elevadas. O grau segundo o qual qualquer tecnologia de engenharia de software ganha aceitação ampla está ligado à sua capacidade de resolver os problemas apresentados pelas tendências, tanto as leves quanto as pesadas.

Tendências leves – a necessidade cada vez maior de conectividade e colaboração, projetos globais, transferência de conhecimento, o impacto das economias emergentes e a influência da própria cultura humana – levam a uma série de desafios que abrangem desde o gerenciamento de complexidade e requisitos emergentes até a manipulação de um mix de talentos sempre em mudanças entre equipes de software geograficamente dispersas.

Tendências pesadas – o ritmo sempre acelerado da mudança da tecnologia – surgem do âmbito das tendências leves e afetam a estrutura do software e o escopo dos processos, e a maneira pela qual uma metodologia de processo é caracterizada. Desenvolvimento colaborativo, novas formas de engenharia de requisitos, desenvolvimento baseado em modelo e controlado por teste e projeto pós-moderno mudarão o cenário dos métodos. Os ambientes de ferramentas responderão a uma necessidade cada vez maior de comunicação e colaboração e, ao mesmo tempo, integrarão soluções pontuais, específicas do domínio, que poderão mudar a natureza atual das tarefas de engenharia de software.

Problemas e pontos a ponderar

38.1 Obtenha uma cópia do *best-seller* *The tipping point*, de Malcolm Gladwell (disponível via Google Book Search) e discuta como suas teorias se aplicam à adoção de novas tecnologias de engenharia de software.

38.2 Por que o software aberto apresenta um desafio às abordagens convencionais de engenharia de software?

38.3 Analise o *ciclo da excelência para tecnologias emergentes* do Gartner Group. Selecione um produto de tecnologia bem conhecido e apresente um breve histórico que ilustre como ele se comportou ao longo da curva. Selecione outro produto de tecnologia bem conhecido que não seguiu o caminho sugerido pela curva da excelência.

38.4 O que é uma “tendência leve”?

38.5 Você enfrenta um problema extremamente complexo que vai exigir uma solução demorada. Como trata a complexidade desse problema e como propõe uma solução?

38.6 O que são “requisitos emergentes” e por que eles representam um desafio para os engenheiros de software?

38.7 Selecione um trabalho de desenvolvimento de código aberto (que não seja o Linux) e apresente um breve histórico de sua evolução e sucesso relativo.

38.8 Descreva como o processo de software mudará durante a próxima década.

38.9 Você trabalha em Los Angeles e participa de uma equipe global de engenharia de software. Você e colegas em Londres, Mumbai, Hong Kong e Sydney devem editar uma especificação de requisitos de 245 páginas para um grande sistema. A primeira edição deve ser feita em três dias. Descreva o conjunto ideal de ferramentas online que lhe possibilitaria colaborar eficazmente.

38.10 Descreva o desenvolvimento de software controlado por modelo. Faça o mesmo para o desenvolvimento controlado por teste.

Leituras e fontes de informação complementares

Livros que discutem o caminho para software e computação abrangem uma grande variedade de assuntos técnicos, científicos, econômicos, políticos e sociais. Kurzweil (*The Singularity Is Near*, Penguin Books, 2005; e *How to Create a Mind*, Viking, 2012) apresentam uma visão atraente de um mundo que mudará de formas muito profundas em meados deste século. Sterling (*Tomorrow Now*, Random House, 2002) lembra-nos de que o progresso real raramente é ordenado e eficiente. Livros de Nanz (*The Future of Software Engineering*, Springer, 2010) e Draheim e seus colegas (*Software Engineering Tools: Trends of Software Engineering Tools and Platforms*, 2010) discutem tendências no desenvolvimento de software. Meisel (*The Software Society: Cultural and Economic Impact*, Trafford, 2013), Saylor (*The Mobile Wave: How Mobile Intelligence Will Change Everything*, Vanguard Press, 2012), Dourish e Bell (*Divining a Digital Future: Mess and Mythology in Ubiquitous Computing*, MIT Press, 2011) e Teich (*Technology and the Future*, 12ª ed., Wadworth, 2012) apresentam ensaios refletidos sobre o impacto social da tecnologia e como a cultura mutante molda a tecnologia. Philips e Naisbitt (*High Tech/High Touch*, Nicholas Brealey, 2001) observam que muitos de nós nos tornamos “intoxicados” com alta tecnologia e que a “grande ironia da era da alta tecnologia é que nos tornamos escravos de dispositivos que se destinavam a nos dar a liberdade”. Zey (*The Future Factor*, Transaction Publishers, 2004) discute as forças que definirão o destino humano durante este século. A obra de Negroponte (*Being Digital*, Alfred A. Knopf, 1995) foi um *best-seller* nos meados da década de 1990 e continua a fornecer uma visão esclarecedora da computação e seu impacto geral.

À medida que o software se torna parte do cenário de quase todas as facetas de nossas vidas, a “cyber ética” evoluiu como um importante assunto de discussão. Livros de Quninn (*Ethics for the Information Age*, 5ª ed., Addison-Wesley, 2012), Spinello (*Cyberethics: Morality and Law in Cyberspace*, 4ª ed., Jones & Bartlett Publishers, 2010), Tavini

(*Ethics and Technology*, 3ª ed., Wiley, 2010), Halbert e Ingulli (*Cyberethics*, 2ª ed., South-Western College Publishers, 2004) e Baird e seus colegas (*Cyberethics: Social and Moral Issues in the Computer Age*, Prometheus Books, 2000) consideram o tema em detalhe. O governo dos Estados Unidos publicou um volumoso relatório em CD-ROM (*21st Century Guide to Cybercrime*, Progressive Management, 2003) que considera todos os aspectos do crime no computador, problemas de propriedade intelectual e o National Infrastructure Protection Center (NIPC).

Uma grande variedade de fontes de informação sobre os rumos futuros das tecnologias relacionadas ao software e engenharia de software está disponível na Internet. Uma lista atualizada das referências da Web pode ser encontrada em “recursos de engenharia de software” no site da SEPA: www.mhhe.com/pressman.

39

Comentários finais

Conceitos-chave

comunicação.....	862
conhecimento.....	863
espectro de	
informações.....	864
ética.....	865
futuro.....	864
inteligência artificial.....	863
peçoas.....	861
responsabilidade.....	865
software revisitado.....	861

Nos 38 capítulos que precedem este, exploramos um processo para engenharia de software que abrange procedimentos de gestão, métodos técnicos, conceitos e princípios básicos, técnicas especializadas, atividades orientadas a pessoas e tarefas passíveis de automação, anotações com lápis e papel e ferramentas de software. Argumentamos que medições, disciplina e um foco na agilidade e qualidade resultarão em software que atende às necessidades do cliente, é confiável, manutenível – ou seja, é um software melhor. Contudo, nunca foi dito que a engenharia de software é um remédio para todas as dificuldades.

As tecnologias de software e de sistemas continuam a ser um desafio para todo profissional de software e toda empresa que constrói sistemas de computador. Embora tenhamos escrito essas palavras com a visão do século 20, Max Hopper [Hop90] descreve com precisão o estado atual das coisas:

Porque as mudanças na tecnologia da informação estão se tornando tão rápidas e intolerantes e as consequências de ficar para trás são tão irreversíveis, ou as empresas dominam a tecnologia ou morrem... Pense nisso como uma escravidão tecnológica. As empresas terão de lutar cada vez mais para manter sua posição.

As mudanças na tecnologia de engenharia de software sem dúvida são “rápidas e inclementes”, ao mesmo tempo que o progresso real é, com frequência, bastante lento. Até que seja tomada a decisão de adotar um novo processo, método ou ferramenta, que seja realizado o treinamento necessário para entender sua aplicação e seja introduzida a tecnologia na cultura de desenvolvimento de software, algo mais novo (e até melhor) surge, e o processo todo começa novamente.

PANORAMA

software, é hora de colocarmos os fatos em perspectiva e tecer alguns comentários finais.

Quem realiza? Autores como nós. Quando você chega ao fim de um livro longo e desafiador, é ótimo poder reunir os fatos de maneira clara.

Por que é importante? É sempre bom lembrar onde estivemos e pensar para onde vamos.

O que é? Ao chegarmos ao fim de uma jornada relativamente longa pela engenharia de software,

Quais são as etapas envolvidas? Faremos considerações sobre o que vimos e trataremos de alguns dos assuntos importantes e algumas direções para o futuro.

Qual é o artefato? Uma discussão que o ajudará a entender o contexto real de desenvolvimento de software.

Como garantir que o trabalho foi realizado corretamente? Isso é difícil de conseguir em tempo real. Somente daqui a alguns anos é que algum de nós poderá dizer se os conceitos, princípios, métodos e técnicas de engenharia de software discutidos neste livro o ajudaram a tornar-se um engenheiro de software melhor.

Uma coisa que aprendemos durante muitos anos nesse campo é que os profissionais de engenharia de software são “conscientes de sua situação”. O caminho à frente estará entulhado de carcaças de novas e excitantes tecnologias (a última moda) que nunca tiveram sucesso (apesar da quantidade de “novidades”). O futuro será formado por tecnologias mais modestas que, de certa forma, modificam a direção e a amplitude do caminho. Abordamos algumas dessas tecnologias no Capítulo 38.

Neste capítulo de conclusão vamos adotar uma visão mais ampla e considerar onde estivemos e para onde estamos indo a partir de uma perspectiva mais filosófica.

39.1 A importância do software – revisitada

A importância do software pode ser definida de várias maneiras. No Capítulo 1, o software foi caracterizado como um diferenciador. A função proporcionada pelo software diferencia produtos, sistemas e serviços e traz uma vantagem competitiva no mercado. Entretanto, o software é mais do que um diferenciador. Quando considerados como um todo, os produtos de engenharia de software geram o bem de consumo mais importante que qualquer indivíduo, negócio ou governo pode adquirir: a informação.

No Capítulo 38, abordamos rapidamente a computação aberta – uma tecnologia que está mudando fundamentalmente nossa percepção dos computadores, das coisas que fazemos com eles (e que eles fazem para nós) e nossa percepção das informações como um guia, um bem e uma necessidade. Destacamos também que o software necessário para dar suporte à computação aberta apresentará novos desafios para os engenheiros de software. Porém, mais importante ainda, a crescente invasão do software de computador apresentará desafios ainda mais significativos para a sociedade como um todo. Sempre que uma tecnologia tem um amplo impacto – que pode salvar vidas ou colocá-las em perigo, criar negócios ou destruí-los, informar os líderes dos governos ou confundi-los –, ela deve ser “manuseada com cuidado”.

39.2 Pessoas e a maneira como constroem sistemas

O software necessário para os sistemas de alta tecnologia torna-se cada vez mais complexo a cada ano que passa, e o tamanho dos programas resultantes aumenta proporcionalmente. O rápido crescimento do tamanho dos programas “médios” não causaria problemas se não fosse por um simples fato: à medida que aumenta o tamanho do programa, o número de pessoas envolvidas com o trabalho também aumenta.

A experiência mostra que, quando o número de profissionais em uma equipe de projeto de software aumenta, a produtividade geral do grupo pode ficar prejudicada. Uma maneira de contornar o problema é criar várias equipes de engenharia de software, dividindo as pessoas em grupos de trabalho. No entanto, à medida que o número de equipes de engenharia de software cresce, a comunicação entre elas se torna tão difícil e morosa quanto a es-

“Choque do futuro é o ambiente de estresse e desorientação que criamos ao redor dos indivíduos, submetendo-os a inúmeras mudanças em um período de tempo muito curto.”

Alvin Toffler

tabelecida entre os indivíduos. Pior ainda, a comunicação (entre pessoas ou equipes) tende a ser ineficiente – muito tempo é gasto transferindo-se pouco conteúdo e, com frequência, informações importantes “caem no vazio”.

Já que a comunidade de engenharia de software tem de enfrentar o dilema da comunicação, o caminho para os engenheiros de software deve incluir mudanças radicais na maneira como os profissionais e as equipes se comunicam uns com os outros. No Capítulo 38, abordamos os ambientes colaborativos que podem proporcionar melhorias significativas na maneira como as equipes se comunicam.

Em última instância, a comunicação é a transferência de conhecimento, e a aquisição (e transferência) de conhecimento está mudando profundamente. À medida que os mecanismos de busca se tornarem cada vez mais sofisticados, as redes sociais e o *crowdsourcing* se transformarem em ferramentas de desenvolvimento e as aplicações para Web 2.0 fornecerem melhor sinergia, a velocidade e a qualidade de transferência de conhecimento aumentará exponencialmente.

Se a história pode servir de referência, podemos afirmar que as pessoas em si não mudarão. No entanto, a maneira como elas se comunicam, o ambiente no qual trabalham, o modo como adquirem conhecimento, os métodos e ferramentas que utilizam, a disciplina que aplicam e, portanto, a cultura geral para o desenvolvimento futuro mudarão de forma significativa e ainda mais profunda.

CASASEGURA



Conclusão?

Cena: Escritório de Doug Miller.

Atores: Doug Miller (gerente do grupo de engenharia do software CasaSegura) e Vinod Raman (membro da equipe de engenharia de software do produto).

Conversa:

Doug: Estou muito contente por termos concluído isso sem muito drama.

Vinod (suspirando e inclinando-se na cadeira): É, mas o projeto cresceu, não foi?

Doug: E você está surpreso? Quando iniciamos o CasaSegura, o marketing achava que um aplicativo de área de trabalho resolveria e, então...

Vinod (sorrindo): E então a Web e a mobilidade assumiram o controle.

Doug: Mas todos nós aprendemos muito.

Vinod: Aprendemos. O material técnico era interessante, mas provavelmente foi o de engenharia de software que nos permitiu concluir no prazo.

Doug: Sim, isso e o esforço de todos vocês. O que você está recebendo do suporte ao cliente? Como está a qualidade no campo?

Vinod: Existem alguns problemas, mas nada realmente sério. Estou vendo isso. Na verdade, daqui a cinco minutos vou me reunir com Jamie para falar a respeito de um deles.

Doug: Antes de ir...

Vinod (a caminho da porta): Já sei... mais trabalho, certo?

Doug: A engenharia desenvolveu um novo sensor... de alta tecnologia... precisamos integrá-lo ao CasaSegura II.

Vinod: CasaSegura II?

Doug: É, CasaSegura II. Vamos iniciar o planejamento na semana que vem.

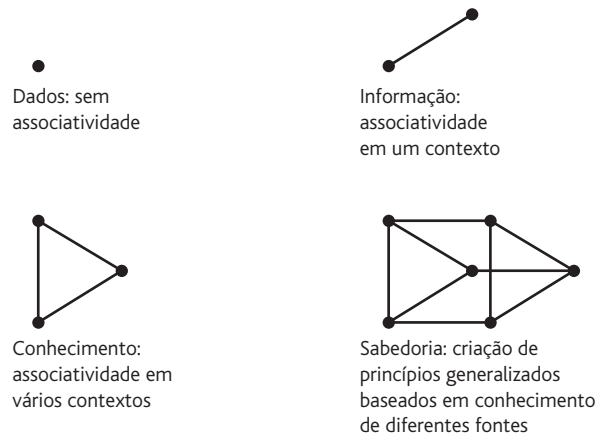


FIGURA 39.1 Um espectro de “informações”.

39.3 Novos modos de representar a informação

Na história da computação, houve uma mudança sutil na terminologia usada para descrever o trabalho de desenvolvimento de software para a comunidade dos negócios. Há 50 anos, o termo *processamento de dados* era a expressão técnica para descrever o uso dos computadores em um contexto comercial. Hoje, processamento de dados deu lugar a outra expressão, *tecnologia da informação*, que significa a mesma coisa, mas apresenta uma mudança sutil no foco. A ênfase não está apenas no processamento de grandes quantidades de dados e sim na extração de informações importantes desses dados. Obviamente, essa sempre foi a intenção, mas a transformação na tecnologia reflete uma alteração muito mais importante na filosofia de gerenciamento.

Quando as aplicações de software são discutidas hoje, as palavras *dados*, *informações* e *conteúdo* vêm à tona repetidamente. Encontramos a palavra *conhecimento* em algumas aplicações de *inteligência artificial*, mas seu uso é relativamente raro. Praticamente ninguém discute *sabedoria* no contexto de aplicações de software.

Dados são informações em estado bruto – conjuntos de fatos que devem ser processados para ter um significado. As informações surgem associando-se fatos em determinado contexto. Conhecimento associa informações obtidas em um contexto às informações obtidas em outro. Por fim, a sabedoria ocorre quando princípios generalizados são derivados de conhecimentos dispersos. Cada uma dessas quatro visões da “informação” é representada esquematicamente na Figura 39.1.

Até hoje, grande parte do software tem sido criada para processar dados ou informações. Engenheiros de software estão agora igualmente preocupados com sistemas que processam conhecimento.¹ Conhecimento é bidimensional. Informações coletadas sobre uma variedade de tópicos relacionados e não relacionados são conectadas para formar um conjunto de fatos que chamamos de *conhecimento*. A chave é nossa habilidade em associar informações de uma

“A melhor preparação para fazer um bom trabalho amanhã é fazer um bom trabalho hoje.”

Elbert Hubbard

¹ O rápido crescimento das tecnologias de obtenção de dados e tratamento de dados reflete essa tendência em crescimento.

"Sabedoria é o poder que nos permite usar o conhecimento para nosso próprio benefício e para o benefício dos outros."

Thomas J. Watson

variedade de origens que podem não ter qualquer conexão óbvia e combiná-las de maneira que nos proporcione um benefício distinto.²

Para ilustrar a progressão dos dados para o conhecimento, considere os dados do senso indicando que o número de nascimentos em 1996 nos Estados Unidos foi de 4,9 milhões. Esse número representa o valor de um dado. Relacionando esse dado com os números de nascimentos nos 40 anos anteriores, podemos gerar uma informação útil – os *baby boomers* dos anos 1950 e início dos anos 1960, que agora estão envelhecendo, fizeram um último esforço para ter seus filhos antes do fim da idade fértil. Além disso, a geração X entrou em sua idade fértil. Os dados do senso podem então ser ligados a outras informações aparentemente não relacionadas. Por exemplo, o número atual de professores da escola primária que se aposentarão durante a próxima década, o número de estudantes que estarão se formando no primário e secundário, a pressão sobre os políticos para baixar os impostos e, portanto, limitar os aumentos de salários dos professores.

Todas essas informações podem ser combinadas para formular uma representação do conhecimento – haverá uma pressão significativa sobre o sistema educacional nos Estados Unidos no início do século 21 e essa pressão continuará por muitas décadas. Usando esse conhecimento, pode surgir uma oportunidade de negócios. Pode haver uma oportunidade significativa de desenvolver novos modos de aprendizado mais eficazes e menos dispendiosos do que a forma atual.

O futuro do software leva a sistemas que processam conhecimento. Estivemos processando dados com computadores por mais de 50 anos e extraíndo informações por mais de três décadas. Um dos desafios mais importantes para a comunidade de engenharia de software é criar sistemas que possibilitem o próximo passo ao longo do espectro – sistemas que extraem conhecimento dos dados e informações de maneira prática e benéfica.

39.4 A visão no longo prazo

Na Seção 39.3, sugerimos que o futuro leva a sistemas que “processam conhecimento”. No entanto, o futuro da computação em geral e dos sistemas baseados em software em particular pode levar a eventos consideravelmente mais profundos.

Em um fascinante livro – leitura obrigatória para todo profissional envolvido em tecnologias da computação –, Ray Kurzweil [Kur05] afirma que já chegamos a uma época em que “o ritmo das mudanças tecnológicas será tão rápido, seu impacto tão profundo, que a vida humana será transformada irreversivelmente”. Kurzweil³ apresenta um argumento convincente de que esta-

² A semântica da Web (Web 2.0) permite a criação de “mashups” que podem proporcionar um mecanismo fácil para conseguir isso.

³ É importante notar que Kurzweil não é um escritor de ficção científica qualquer ou um futurista sem portfólio. Ele é um tecnologista sério que (segundo a Wikipedia) “tem sido pioneiro nos campos do reconhecimento óptico de caracteres (OCR), síntese texto-para-fala, tecnologia de reconhecimento de voz e instrumentos de teclado eletrônicos”.

mos atualmente no “joelho” de uma curva de crescimento exponencial que nos levará a enormes avanços na capacidade de computação durante as próximas duas décadas. Se combinados com avanços equivalentes em nanotecnologia, genética e robótica, podemos chegar a uma época, em meados deste século, em que a distinção entre humanos (como conhecemos hoje) e máquinas começará a se confundir – época na qual a evolução humana se acelerará de modo assustador (para alguns) e espetacular (para outros).

Kurzweil afirma que, em algum momento na próxima década, a capacidade de computação e os requisitos de software serão suficientes para modelar todos os aspectos do cérebro humano [Kur13] – todas as conexões físicas, processos analógicos e camadas químicas. Quando isso ocorrer, os seres humanos darão o primeiro passo para obter “IA (inteligência artificial) forte” e, em consequência disso, máquinas que realmente pensam (usando os termos convencionais de hoje). Mas haverá uma diferença fundamental. Os processos do cérebro humano são excessivamente complexos e fracamente ligados às fontes externas de informação. Eles também são lentos em computação, mesmo em comparação com a tecnologia atual. Se ocorrer a emulação completa do cérebro humano, o “pensamento” ocorrerá a velocidades milhares de vezes maiores do que seu correspondente humano, com conexões íntimas a um mar de informações (pense na Web de hoje como um exemplo primitivo). O resultado é... bem... tão fantástico, que é melhor deixar que o próprio Kurzweil descreva.

É importante notar que nem todos acreditam que o futuro que Kurzweil descreve é uma coisa boa. Em um ensaio agora famoso intitulado “O futuro não precisa de nós” (*The Future Doesn't Need Us*), Bill Joy [Joy00], um dos fundadores da Sun Microsystems, diz que a “robótica, a engenharia genética e a nanotecnologia estão fazendo dos humanos uma espécie ameaçada”. Seus argumentos prevendo uma antiutopia tecnológica representam um contra-argumento ao futuro utópico previsto por Kurzweil. Ambos devem ser considerados seriamente quando os engenheiros de software assumem um dos papéis importantes que definem o futuro da raça humana.

“Você não pode ligar os pontos olhando para frente; você só pode ligá-los olhando para trás. Portanto, você precisa acreditar que os pontos serão ligados de algum modo em seu futuro.”

Steve Jobs

39.5 A responsabilidade do engenheiro de software

A engenharia de software evoluiu, tornando-se uma profissão mundialmente respeitada. Como profissionais, os engenheiros de software devem se orientar por um código de ética que rege o trabalho que fazem e os produtos que criam. Uma força-tarefa da ACM/IEEE-CS (ACM/ IEE-CS Joint Task Force) produziu um código de ética e práticas profissionais para engenharia de software (*Software Engineering Code of Ethics and Professional Practices*) (Versão 5.1). O código [ACM12] declara:

Engenheiros de software deverão se comprometer a fazer da análise, especificação, projeto, desenvolvimento, teste e manutenção de software uma profissão benéfica e respeitada. De acordo com seu compromisso com a saúde, segurança e bem-estar do público, os engenheiros de software deverão adotar os oito princípios a seguir:

Uma discussão completa sobre o código de ética ACM/IEEE pode ser encontrada em seeri.etsu.edu/Codes/default.shtm.

1. PÚBLICO – Os engenheiros de software deverão agir em consonância com o interesse público.
2. CLIENTE E EMPREGADOR – Os engenheiros de software deverão agir de acordo com o melhor interesse de seu cliente e empregador em consonância com o interesse público.
3. PRODUTO – Os engenheiros de software deverão garantir que seus produtos e modificações relacionadas atinjam os mais altos padrões profissionais possíveis.
4. JULGAMENTO – Os engenheiros de software deverão manter a integridade e independência de seu julgamento profissional.
5. GESTÃO – Os gerentes de engenharia de software e líderes deverão aderir e promover uma abordagem ética para a gestão do desenvolvimento e manutenção de software.
6. PROFISSÃO – Os engenheiros de software deverão prestigiar a integridade e a reputação da profissão em consonância com o interesse público.
7. COLEGAS – Os engenheiros de software deverão ser justos e solidários com seus colegas.
8. INDIVIDUALMENTE – Os engenheiros de software deverão participar do contínuo aprendizado referente à prática de sua profissão e promover uma abordagem ética à prática da profissão.

Embora cada um desses oito princípios seja igualmente importante, surge um tema mais importante: um engenheiro de software deve trabalhar no interesse público. Em nível pessoal, um engenheiro de software deve se guiar pelas seguintes regras:

- Nunca roubar dados para obter vantagens pessoais.
- Nunca distribuir ou vender informações proprietárias obtidas como parte de seu trabalho em um projeto de software.
- Nunca destruir ou modificar de forma mal-intencionada os programas, arquivos ou dados de outra pessoa.
- Nunca violar a privacidade de um indivíduo, grupo ou organização.
- Nunca invadir um sistema por esporte ou lucro.
- Nunca criar ou promover um vírus ou verme (*worm*) de computador.
- Nunca usar a tecnologia de computação para facilitar a discriminação ou assédio.

Durante a última década, certos membros da indústria de software formaram um *lobby* por uma legislação de proteção que [SEE03]: (1) permite às empresas liberar software sem revelar os defeitos conhecidos, (2) isenta os desenvolvedores da responsabilidade por quaisquer danos resultantes desses defeitos conhecidos, (3) impede que outros descubram defeitos sem permissão do desenvolvedor original, (4) permite a incorporação de software “autoajuda” dentro do produto que pode desabilitar (via comando remoto) a operação do produto e (5) isenta os desenvolvedores do software com “autoajuda” de danos se o software for desabilitado por um terceiro.

Como toda legislação, o debate em geral concentra-se em aspectos políticos, não tecnológicos. No entanto, muitas pessoas (até mesmo nós) pensam que a legislação protetora, se formulada incorretamente, entra em conflito com o código de ética da engenharia de software, isentando indiretamente os engenheiros de sua responsabilidade de produzir software de alta qualidade.

39.6 Comentário final de RSP

Já faz quase três décadas e meia que o trabalho na 1ª edição deste livro começou. Ainda me lembro de quando estava sentado à minha mesa como jovem professor, escrevendo o manuscrito de um livro sobre um assunto com o qual poucas pessoas se importavam e ainda menos pessoas realmente entendiam. Lembro-me das cartas de rejeição dos editores, que argumentavam (educadamente, mas com firmeza) que nunca haveria um mercado para um livro sobre “engenharia de software”. Felizmente, a McGraw-Hill decidiu tentar⁴, e o resto, como se costuma dizer, é história.

Desde a 1ª edição, este livro mudou significativamente – em escopo, tamanho, estilo e conteúdo. Assim como a engenharia de software, cresceu e (espero) amadureceu durante os últimos anos.

Uma abordagem de engenharia para o desenvolvimento de software de computador é hoje senso comum. O debate sobre o “paradigma certo”, a importância da agilidade, o grau de automação e os métodos mais eficazes ainda continua, mas os princípios básicos da engenharia de software são agora aceitos em toda a indústria. Por que, então, só recentemente temos visto uma adoção mais ampla?

A resposta, penso, está na dificuldade da transição tecnológica e na mudança cultural que a acompanha. Apesar de muitos de nós apreciarmos a necessidade de uma disciplina de engenharia para software, lutamos contra a inércia da prática anterior e nos defrontamos com novos domínios de aplicação (e os desenvolvedores que trabalham nelas) que parecem prontos para repetir os mesmos erros do passado. Para facilitarmos a transição, precisamos de muitas coisas – um processo de software ágil, adaptável e sensível; métodos mais eficazes; ferramentas mais poderosas; melhor aceitação pelos profissionais e apoio dos gerentes; e uma boa dose de educação.

Você pode não concordar com todas as abordagens descritas neste livro. Algumas das técnicas e opiniões são controversas; outras devem ser ajustadas para funcionar bem em diferentes ambientes de desenvolvimento de software. No entanto, espero sinceramente que *Engenharia de software: uma abordagem profissional* tenha delineado os problemas que enfrentamos, demonstrado o poder dos conceitos de engenharia de software e proporcionado um framework de métodos e ferramentas.

⁴ Na realidade, o mérito vai para Peter Freeman e Eric Munson, que convenceram a McGraw-Hill de que valia a pena tentar. Depois de quase dois milhões de cópias, é justo dizer que tomaram uma boa decisão.

À medida que avançamos no século 21, o software continua a ser o produto e a indústria mais importantes no cenário mundial. Seu impacto e importância têm amplo alcance. Contudo, uma nova geração de desenvolvedores de software deve enfrentar muitos dos mesmos desafios que as gerações anteriores enfrentaram. Esperemos que as pessoas que enfrentam esse desafio – os engenheiros de software – tenham a sabedoria para desenvolver sistemas que melhorem a condição humana.

Introdução à UML¹

1

A *Unified Modeling Language* (UML, *Linguagem de Modelagem Unificada*) é “uma linguagem-padrão para descrever/documentar projeto de software. A UML pode ser usada para visualizar, especificar, construir e documentar os artefatos de um sistema de software intensivo” [Boo05]. Em outras palavras, assim como os arquitetos criam plantas e projetos para serem usados por uma empresa de construção, os arquitetos de software criam diagramas UML para ajudar os desenvolvedores de software a construir o software. Se você entender o vocabulário da UML (os elementos visuais do diagrama e seus significados), poderá facilmente entender e especificar um sistema e explicar o projeto desse sistema para outros interessados.

Grady Booch, Jim Rumbaugh e Ivar Jacobson desenvolveram a UML na década de 1990, com muitas opiniões da comunidade de desenvolvimento de software. A UML combinou um grupo de notações de modelagem concorrentes usadas pela indústria do software na época. Em 1997, a UML 1.0 foi apresentada ao OMG (Object Management Group), uma associação sem fins lucrativos dedicada a manter especificações para serem usadas pela indústria de computadores. A UML 1.0 foi revisada tornando-se a UML 1.1 e adotada mais tarde naquele ano. O padrão atual é a UML 2.3² e agora é um padrão ISO. Como esse padrão é novo, muitas referências mais antigas, como [Gam95], não usam a notação UML.

A UML 2.3 fornece 13 diagramas diferentes para uso na modelagem de software. Neste apêndice, discutiremos apenas os diagramas de *classe*, *implantação*, *caso de uso*, *sequência*, *comunicação*, *atividade* e *estado*. Esses diagramas são usados nesta edição do livro.

Você notará que há muitas características opcionais em diagramas UML. A linguagem UML proporciona essas opções (às vezes obscuras) para que você possa expressar todos os aspectos importantes de um sistema. Ao mesmo tempo, é possível suprimir partes não relevantes ao aspecto que está sendo modelado para não congestionar o diagrama com detalhes irrelevantes. Portanto, a omissão de uma característica particular não significa que ela esteja ausente, mas sim que foi suprimida. Neste apêndice, não apresen-

Conceitos-chave

dependência	872
diagrama de	
comunicação	880
diagrama de	
implantação	874
diagrama de atividade ..	881
diagrama de caso	
de uso	875
diagrama de classe	870
diagrama de estado	884
diagrama de sequência ..	876
frames de interação	878
generalização	871
multiplicidade	872
Object Constraint	
Language	887
raias	883
stereotype	871

¹ Este apêndice teve a contribuição de Dale Skrien e foi adaptado de seu livro, *An introduction to object-oriented design and design patterns in Java* (McGraw-Hill, 2008). Todo o conteúdo é usado com permissão.

² <http://www.omg.org/spec/UML/2.3/Infrastructure/PDF/> e <http://www.omg.org/spec/UML/2.3/Superstructure/PDF/>. Essas duas especificações complementares constituem uma especificação completa da linguagem de modelagem UML 2.

teremos uma discussão exaustiva de todas as características dos diagramas UML. Em vez disso, nos concentraremos nas opções-padrão, especialmente as usadas neste livro.

Diagramas de classe

Para modelar classes, incluindo seus atributos, operações e relações e associações com outras classes,³ a UML tem um *diagrama de classe*. Um diagrama de classe fornece uma visão estática ou estrutural do sistema. Ele não mostra a natureza dinâmica das comunicações entre os objetos das classes no diagrama.

Os elementos principais são caixas, ou seja, ícones usados para representar classes e interfaces. Cada caixa é dividida em partes horizontais. A parte superior contém o nome da classe. A seção do meio lista os atributos da classe. Os atributos podem ser valores que a classe calcula a partir de suas variáveis de instância ou valores que a classe pode obter de outros objetos dos quais é composta. Por exemplo, um objeto sempre pode saber a hora atual e ser capaz de retorná-la quando for solicitada, no caso em que seria apropriado listar a hora atual como um atributo daquela classe de objetos. No entanto, o objeto muito provavelmente não teria a hora armazenada em uma de suas variáveis de instância, porque precisaria continuamente atualizar aquele campo. Em vez disso, o objeto poderia calcular a hora atual (por exemplo, por meio da consulta a objetos de outras classes) no momento em que a hora fosse requisitada. A terceira seção do diagrama de classes contém as operações ou comportamentos da classe. Uma *operação* refere-se ao que os objetos da classe podem fazer. Usualmente é implementada como um *método* da classe.

A Figura A1.1 apresenta um exemplo simples de uma classe **Thoroughbred** que modela cavalos puro-sangue. Ela mostra três atributos – *mother*, *father* e *birthyear*. Os diagramas também mostram três operações: *getCurrentAge()*, *getFather()* e *getMother()*. Pode haver outros atributos e operações suprimidos, não mostrados no diagrama.

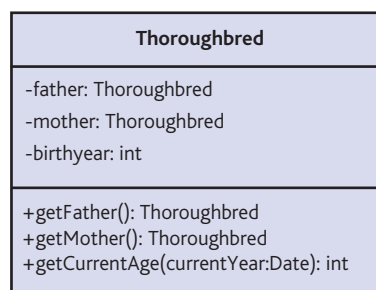


FIGURA A1.1 Um diagrama para a classe Thoroughbred.

³ Se você não estiver familiarizado com os conceitos de orientação a objetos, uma breve introdução é apresentada no Apêndice 2.

Cada atributo pode ter um nome, um tipo e um nível de visibilidade. O tipo e a visibilidade são opcionais. O tipo vem após o nome e é separado por dois-pontos. A visibilidade é indicada precedendo pelo sinal -, #, ~ ou +, que indicam, respectivamente, visibilidade *private*, *protected*, *package* ou *public*. Na Figura A1.1, todos os atributos têm visibilidade *private*, conforme indica o sinal de menos (-). Você pode também especificar que um atributo é estático ou de classe, usando um sublinhado. Cada operação também pode ser mostrada com um nível de visibilidade, parâmetros com nomes e tipos e um tipo de retorno.

Uma classe abstrata ou método abstrato é indicado pelo uso de itálico no nome da classe no diagrama de classes. Como exemplo, veja a classe **Horse** na Figura A1.2. Uma interface é indicada acrescentando-se a expressão “<<interface>>” (chamada de *stereotype*) acima do nome. Veja a interface **OwnedObject** na Figura A1.2. Uma interface também pode ser representada graficamente por um círculo vazio.

Vale mencionar que o ícone que representa uma classe pode ter outras partes opcionais. Por exemplo, uma quarta seção na parte inferior da caixa de classe pode ser usada para listar as responsabilidades da classe. Essa seção é particularmente útil quando se faz a transição dos cartões CRC (Capítulo 10) para diagramas de classe, pois as responsabilidades listadas nos cartões CRC podem ser acrescentadas à quarta seção na caixa da classe no diagrama UML antes que os atributos e operações que executam essas responsabilidades sejam criados. Essa quarta seção não é mostrada em nenhuma das figuras neste apêndice.

Os diagramas de classe também podem exibir relações entre classes. Uma classe que seja subclasse de outra classe é conectada a ela por uma seta com uma linha cheia como eixo e com uma ponta triangular vazia. A seta aponta da subclasse para a superclasse. Em UML, uma relação como essa é chamada de *generalização*. Por exemplo, na Figura A1.2, as classes **Thoroughbred** e **QuarterHorse** são exibidas como subclasses da classe abstrata **Horse**. Uma seta tendo como eixo uma linha tracejada indica implementação de

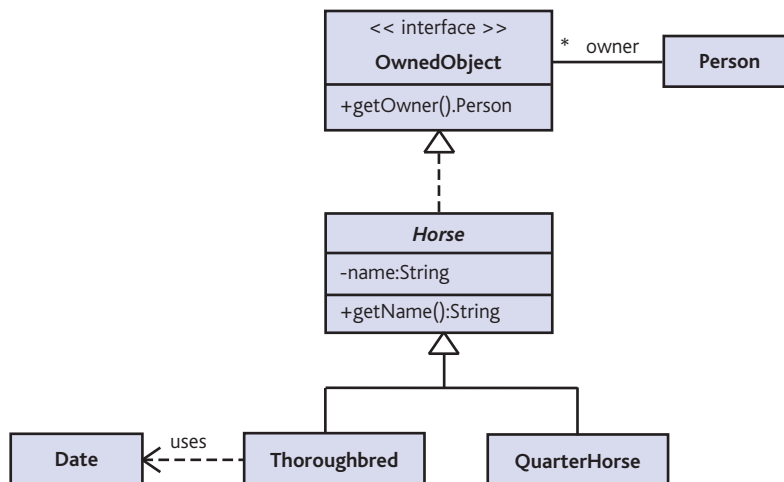


FIGURA A1.2 Um diagrama de classe referente a cavalos.

interface. Em UML, esse tipo de relação é chamada de *realização*. Na Figura A1.2, a classe **Horse** implementa ou realiza a interface **OwnedObject**.

A *associação* entre duas classes indica que há uma relação estrutural entre elas. Associações são representadas por linhas cheias. Uma associação tem muitas partes opcionais. Ela pode ser rotulada, assim como cada uma de suas extremidades, para indicar o papel de cada classe na associação. Por exemplo, na Figura A1.2, há uma associação entre **OwnedObject** e **Person** na qual **Person** desempenha o papel de proprietário (owner). Setas em qualquer uma ou em ambas as extremidades de uma linha de associação indicam navegabilidade. Além disso, cada extremidade da linha de associação pode ter um valor de multiplicidade. Navegabilidade e multiplicidade são explicadas em detalhes mais à frente nesta seção. Uma associação pode também conectar uma classe com ela própria, usando um laço. Desse modo, uma associação indica a conexão de um objeto da classe com outros objetos da mesma classe.

A associação com uma seta em uma extremidade indica navegabilidade unidirecional. A seta significa que de uma classe pode-se facilmente acessar a segunda classe associada para a qual a associação aponta. A partir da segunda classe, não se pode necessariamente acessar com facilidade a primeira classe. Outra maneira de pensar sobre isso é que a primeira classe tem conhecimento da segunda classe, enquanto o objeto da segunda classe não conhece necessariamente a primeira classe. Uma associação sem setas em geral indica uma associação bidirecional, que é o que se pretendia na Figura A1.2, mas poderia também significar apenas que a navegabilidade não é importante e foi deixada de lado.

Deve-se notar que um atributo de uma classe é muito parecido com uma associação da classe com o tipo de classe do atributo. Isto é, para indicar que uma classe tem uma propriedade chamada “nome” do tipo **String**, poderíamos mostrar aquela propriedade como um atributo, como na classe **Horse** da Figura A1.2. Como alternativa, poderíamos criar uma associação unidirecional da classe **Horse** para a classe **String**, sendo “nome” o papel da classe **String**. A abordagem de atributo é melhor para tipos de dados primitivos, enquanto a abordagem de associação muitas vezes é melhor se a classe da propriedade desempenha um papel importante no projeto, caso em que é muito bom ter uma caixa de classe para aquele tipo.

Uma relação de *dependência* representa outra conexão entre classes e é indicada por uma linha tracejada (com setas opcionais nas extremidades e com rótulos opcionais). Uma classe depende de outra se alterações na segunda classe podem exigir alterações na primeira classe. Uma associação de uma classe para outra indica automaticamente uma dependência. Não é necessária uma linha pontilhada entre classes se já houver uma associação entre elas. No entanto, para uma relação transiente (uma classe que não mantém relação de longo prazo com outra classe, mas usa aquela classe ocasionalmente), podemos colocar uma linha tracejada da primeira classe para a segunda. Por exemplo, na Figura A1.2, a classe **Thoroughbred** usa a classe **Date** sempre que é invocado o método `getCurrentAge()` e, assim, a dependência é chamada de “uses”.

A *multiplicidade* de uma extremidade de uma associação indica o número de objetos daquela classe associados à outra classe. Uma multiplicidade é especificada por um valor inteiro não negativo ou por um intervalo de valores

inteiros. Uma multiplicidade especificada como “0..1” significa que há 0 ou 1 objeto na extremidade da associação. Por exemplo, cada pessoa no mundo tem um número de Seguro Social ou não tem tal número (especialmente se não forem cidadãos americanos) e, assim, uma multiplicidade de 0..1 poderia ser usada em uma associação entre uma classe **Person** e uma classe **Social-SecurityNumber** no diagrama de classes. A multiplicidade especificada por “1..*” significa um ou mais, e a multiplicidade especificada por “0..*”, ou apenas “*”, significa zero ou mais. Usou-se um * como multiplicidade na extremidade **OwnedObject** da associação com a classe **Person** na Figura A1.2 porque uma **Person** pode possuir zero ou mais objetos.

Se uma extremidade de uma associação apresenta multiplicidade maior do que 1, os objetos da classe aos quais se faz referência na extremidade da associação provavelmente estão armazenados em uma coleção, como um conjunto ou uma lista ordenada. Poderíamos também incluir a própria classe de coleção no diagrama UML, mas uma classe desse tipo usualmente é desconsiderada e assume-se, implicitamente, que esteja lá devido à multiplicidade da associação.

Uma *agregação* é um tipo especial de associação representada por um losango vazio em uma extremidade do ícone. Ela indica uma relação “todo/parte”, em que a classe para a qual a seta aponta é considerada uma “parte” da classe na extremidade do losango da associação. Uma *composição* é uma agregação indicando forte relação de propriedade entre as partes. Em uma composição, as partes vivem e morrem com o proprietário porque não têm um papel a desempenhar no sistema de software independente do proprietário. Veja na Figura A1.3 exemplos de agregação e composição.

Uma classe **College** tem uma agregação de objetos **Building** que representam os edifícios que formam o campus. A universidade (**College**) tem também uma coleção de cursos. Mesmo que a universidade fechasse, os edifícios ainda continuariam a existir (supondo que a universidade não fosse fisicamente destruída) e poderiam ser usados para outros propósitos, mas um objeto **Course** não tem utilidade fora da universidade na qual está sendo oferecido. Se a universidade deixasse de existir como uma entidade de negócios, o objeto **Course** não teria mais utilidade e, portanto, também deixaria de existir.

Outro elemento comum em diagramas de classes é uma anotação (*note*), representada por uma caixa com um canto dobrado e conectada a outros ícones por uma linha pontilhada. Ela pode ter conteúdo arbitrário (texto e gráficos) e é similar a um comentário de linguagem de programação. Pode ter informações sobre o papel de uma classe ou restrições que todos os objetos daquela classe devem satisfazer. Se o conteúdo for uma restrição, estará entre chaves. Observe a restrição anexada à classe **Course** na Figura A1.3.

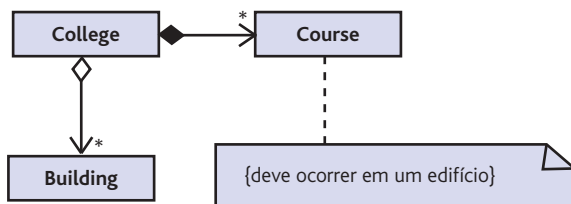


FIGURA A1.3 A relação entre College, Course e Building.

Diagramas de implantação

Diagramas de implantação focalizam a estrutura do sistema de software e são úteis para mostrar a distribuição física de um sistema de software entre plataformas de hardware e ambientes de execução. Por exemplo, suponha que você esteja desenvolvendo um pacote de renderização gráfica baseado na Web. Os usuários do seu pacote de software usarão o navegador Web para acessar o seu site e introduzir as informações de renderização. O seu site vai renderizar uma imagem gráfica de acordo com as especificações do usuário e a enviará de volta ao usuário. Como a renderização gráfica pode ser cara em termos de computação, você decide tirar a renderização do servidor Web, colocando-a em uma plataforma separada. Portanto, haverá três dispositivos de hardware envolvidos no seu sistema: o cliente Web (o computador do usuário executando um navegador), o computador que está hospedando o servidor Web e o computador que está hospedando o dispositivo de renderização.

A Figura A1.4 mostra o diagrama de implantação para um pacote de software como esse. Neles, os componentes de hardware são desenhados como caixas com o título “<<device>>”. Os caminhos de comunicação entre os componentes de hardware são traçados com linhas com títulos opcionais. Na Figura A1.4, os caminhos são identificados com o protocolo de comunicação e o tipo de rede usada para conectar os dispositivos.

Cada nó em um diagrama de implantação pode também ser anotado com detalhes sobre o dispositivo. Por exemplo, na Figura A1.4, é desenhado o navegador cliente para mostrar que contém um artefato, que é o software do navegador Web. Artefato é tipicamente um arquivo que contém software executando em um dispositivo. Você pode também especificar valores rotulados, como mostra a Figura A1.4 no nó do servidor Web. Esses valores definem o fornecedor do servidor Web e o sistema operacional usado pelo servidor.

Diagramas de implantação também podem mostrar os nós do ambiente de execução, desenhados em caixas contendo o rótulo “<<execution environment>>”. Esses nós representam sistemas, como os sistemas operacionais, que podem hospedar outros programas de software.

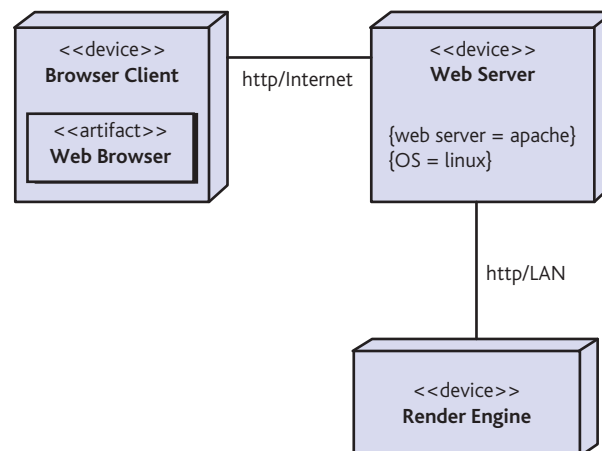


FIGURA A1.4 Um diagrama de implantação.

Diagramas de caso de uso

Casos de uso (Capítulos 8 e 9) e o *diagrama de caso de uso* ajudam a determinar a funcionalidade e as características do software sob o ponto de vista do usuário. Para dar uma ideia de como os casos de uso e os diagramas de caso de uso funcionam, vamos criar alguns deles para uma aplicação de software de gerenciamento para uma loja de música digital online. Algumas das coisas que o software pode fazer são:

- Baixar um arquivo de música MP3 e armazená-lo na biblioteca da aplicação.
- Capturar a música e armazená-la na biblioteca da aplicação.
- Gerenciar a biblioteca da aplicação (por exemplo, excluir músicas ou organizá-las em listas de execução).
- Gravar em um CD uma lista de músicas da biblioteca.
- Carregar uma lista de músicas da biblioteca para um iPod ou MP3 player.
- Converter uma música do formato MP3 para o formato AAC e vice-versa.

Essa não é uma lista completa, mas suficiente para entendermos a função dos casos de uso e diagramas de caso de uso.

Um caso de uso descreve como um usuário interage com o sistema, definindo os passos necessários para atingir um objetivo específico (por exemplo, gravar uma lista de músicas em um CD). Variações na sequência de passos descrevem vários cenários (por exemplo, o que acontece se as músicas da lista não couberem em um CD?).

Um diagrama UML de caso de uso é uma visão geral de todos os casos de uso e de como eles estão relacionados. Fornece uma visão geral da funcionalidade do sistema. Um diagrama de caso de uso para a aplicação de música digital é mostrado na Figura A1.5.

Nesse diagrama, a figura do usuário representa um *ator* (Capítulo 8) que está associado a uma categoria de usuário (ou outro elemento de interação). Sistemas complexos normalmente possuem mais de um ator. Por exemplo, uma aplicação “máquina de venda automática” pode ter três atores representando clientes, pessoal de manutenção e os fornecedores que abastecem a máquina.

No diagrama de caso de uso, estes são mostrados como elipses. Os atores são conectados por linhas aos casos de uso que eles executam. Note que nenhum dos detalhes dos casos de uso é incluído no diagrama e precisa ser armazenado separadamente. Observe também que os casos de uso são colocados em um retângulo, mas os atores não. Esse retângulo serve para lembrar visualmente as fronteiras do sistema e que os atores estão fora do sistema.

Alguns casos de uso em um sistema podem estar relacionados uns com os outros. Por exemplo, há passos similares para gravar uma lista de músicas em um CD e para carregar uma lista de músicas em um iPod ou smartphone. Em ambos os casos, o usuário primeiro cria uma lista vazia e, em seguida, acrescenta as músicas da biblioteca na lista. Para evitar duplicação, normalmente é melhor criar um novo caso de uso representando a atividade duplicada e

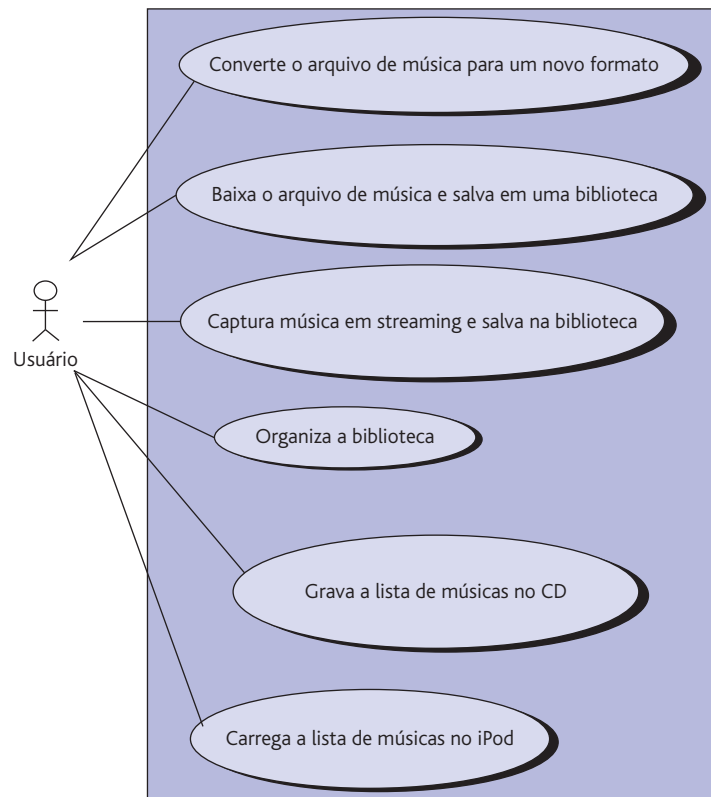


FIGURA A1.5 Um diagrama de caso de uso para o sistema de música.

depois deixar que outros casos incluam esse novo caso de uso como um de seus passos. A inclusão é indicada nos diagramas de caso de uso, como mostra a Figura A1.6, por meio de uma seta tracejada identificada como «include», conectando um caso de uso a outro.

Um diagrama de caso de uso, por mostrar todos os casos, é um bom auxílio para assegurar a inclusão de toda a funcionalidade do sistema. Em nosso organizador de música digital, certamente desejaríamos ter mais casos de uso, como, por exemplo, um para tocar uma música da biblioteca. Mas tenha em mente que a maior contribuição dos casos de uso para o processo de desenvolvimento de software é a descrição textual de cada caso e não o diagrama geral de casos de uso [Fow04]. É por meio das descrições que você consegue formar uma ideia clara dos objetivos do sistema que está desenvolvendo.

Diagramas de sequência

Em contraste com os diagramas de classe e de implantação, que mostram a estrutura estática de um componente de software, o *diagrama de sequência* é utilizado para indicar as comunicações dinâmicas entre objetos durante a execução de uma tarefa. Ele mostra a ordem temporal na qual as mensagens são enviadas entre os objetos para executar aquela tarefa. Podemos usar um

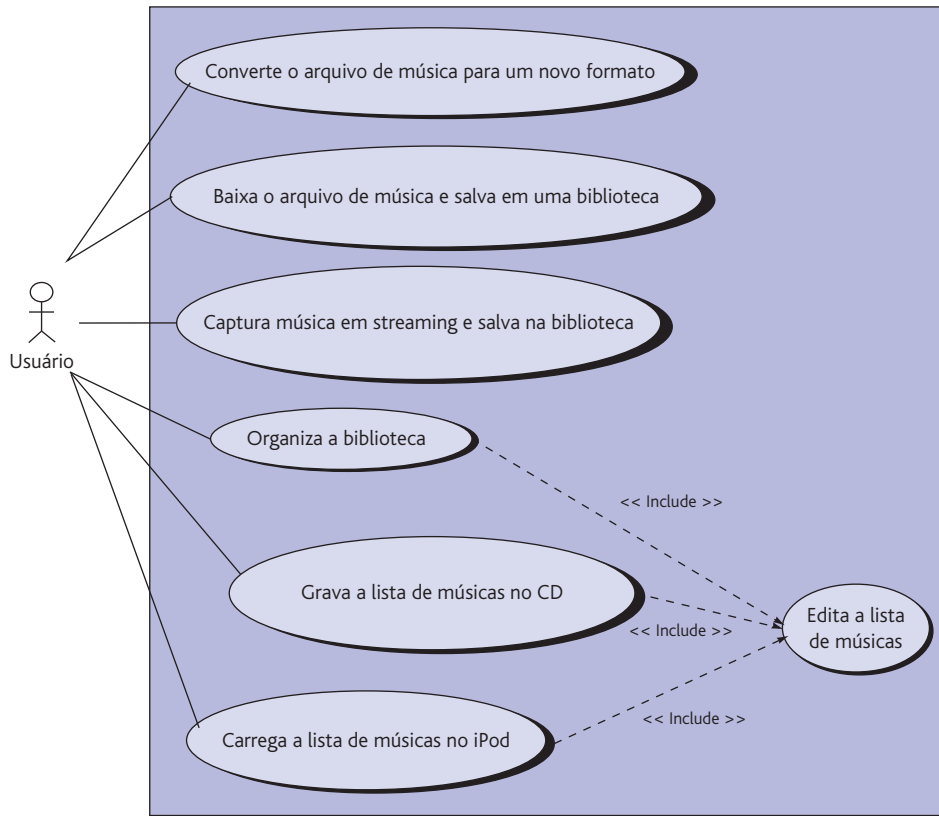


FIGURA A1.6 Um diagrama de caso de uso com casos de uso incluídos.

diagrama de sequência para mostrar as interações em um caso de uso ou em um cenário do sistema de software.

Na Figura A1.7 há um diagrama de sequência para um programa de desenho. O diagrama mostra os passos envolvidos para destacar uma figura no desenho quando é clicada. Cada caixa na linha do topo do diagrama em geral corresponde a um objeto, embora seja possível fazer as caixas modelarem outras coisas, como classes, por exemplo. Se a caixa representa um objeto (como é o caso em todos os nossos exemplos), dentro da caixa pode-se opcionalmente declarar o tipo do objeto, precedido de dois-pontos. Você pode também colocar antes dos dois-pontos e do tipo o nome do objeto, conforme mostra a terceira caixa na Figura A1.7. Abaixo de cada caixa há uma linha tracejada chamada de *linha de vida* do objeto. O eixo vertical no diagrama de sequência corresponde ao tempo, e o tempo aumenta à medida que se caminha para baixo.

Um diagrama de sequência mostra chamadas de método usando setas horizontais do *chamador* para o *chamado*, identificadas com o nome do método e, opcionalmente, incluindo seus parâmetros, seus tipos e o tipo de retorno. Por exemplo, na Figura A1.7, **MouseListener** chama o método *getFigureAt()* de **Drawing**. Quando um objeto está executando um método (quando tem uma ativação de um método na pilha), você pode, opcionalmente, mostrar uma barra branca, conhecida como *barra de ativação*, ao longo da linha de vida do ob-

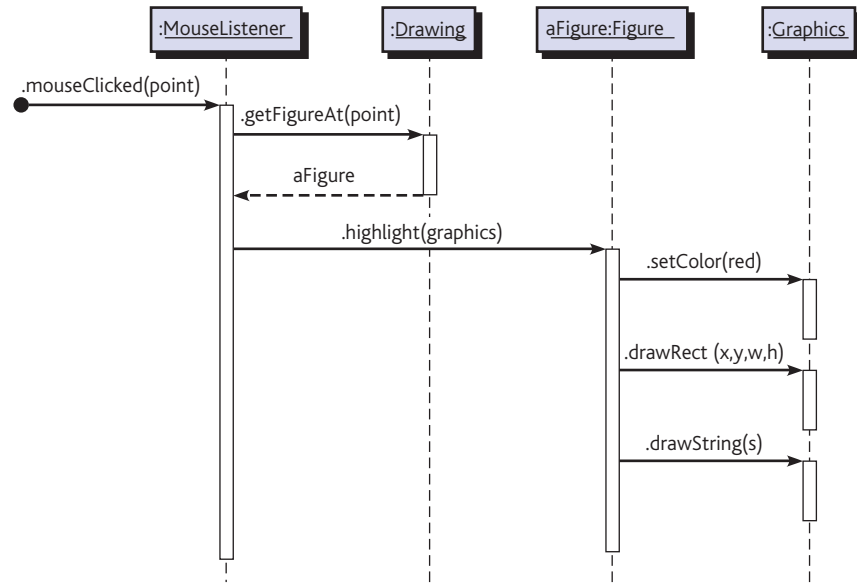


FIGURA A1.7 Exemplo de diagrama de sequência.

jeto. Na Figura A1.7, há barras de ativação para todas as chamadas de método. O diagrama também pode mostrar, opcionalmente, o retorno de uma chamada de método, com uma seta pontilhada e um rótulo opcional. Na Figura A1.7, o retorno da chamada do método `getFigureAt()` é identificado com o nome do objeto retornado. Uma prática comum, como fizemos na Figura A1.7, é omitir a seta de retorno quando um método não retorna nada (*void*), porque isso complicaria o diagrama, fornecendo informações de pouca importância. Um círculo preto com uma seta indica uma *mensagem encontrada* cuja origem é desconhecida ou irrelevante.

Agora você será capaz de entender a tarefa que a Figura A1.7 está mostrando. Uma origem desconhecida chama o método `mouseClicked()` de um **MouseListener**, passando como argumento o ponto onde o clique ocorreu. O **MouseListener** por sua vez chama o método `getFigureAt()` de um **Drawing**, que retorna um **Figure**. Então, **MouseListener** chama o método destacado de **Figure**, passando um objeto **Graphics** como argumento. Em resposta, **Figure** chama três métodos do objeto **Graphics** para traçar a figura em vermelho.

O diagrama na Figura A1.7 é muito claro e não contém condicionais ou laços. Se forem necessárias estruturas lógicas de controle, provavelmente será melhor traçar um diagrama de sequência separado para cada caso. Isto é, se o fluxo de mensagem puder tomar dois caminhos diferentes, dependendo de uma condição, trace dois diagramas de sequência separados, um para cada possibilidade.

Se você ainda quer incluir laços, condicionais e outras estruturas de controle em um diagrama de sequência, use frames de interação (*interaction frames*), que são retângulos que envolvem as partes do diagrama e são identificados com o tipo de estrutura de controle que representam. A Figura A1.8 ilustra isso, mostrando o processo envolvido para destacar todas as figu-

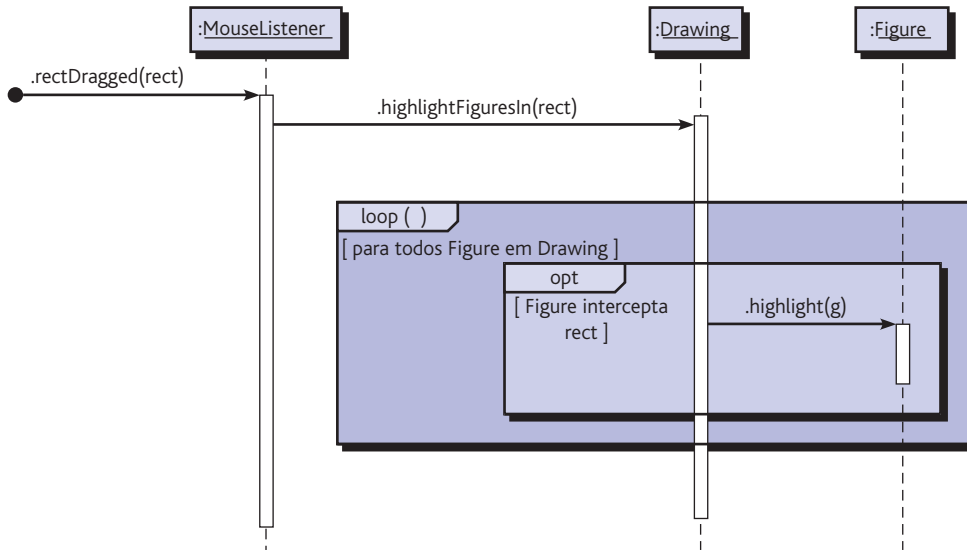


FIGURA A1.8 Um diagrama de sequência com dois frames de interação.

ras em um retângulo. Para **MouseListener** é enviada a mensagem `rectDragged`. O **MouseListener** então manda o desenho destacar todas as figuras no retângulo, chamando o método `highlightFigures()`, passando o retângulo como argumento. O método passa em laço por todos os objetos **Figure** no objeto **Drawing** e, se o objeto **Figure** intercepta o retângulo, é solicitado a **Figure** que se destaque. As frases entre colchetes são chamadas de *guardas*, que são condições booleanas que devem ser verdadeiras se a ação dentro do frame de interação deve continuar.

Há muitas outras características que podem ser incluídas em um diagrama de sequência. Por exemplo:

1. Você pode distinguir entre mensagens síncronas e assíncronas. Mensagens síncronas são exibidas com pontas de setas cheias, enquanto mensagens assíncronas são mostradas com pontas de seta em traço.
2. Você pode mostrar um objeto fazendo-o enviar a si próprio uma mensagem, com uma seta saindo do objeto, virando para baixo e, em seguida, apontando de volta para o mesmo objeto.
3. Você pode mostrar a criação do objeto traçando uma seta identificada de forma apropriada (por exemplo, com um rótulo «create») para a caixa de um objeto. Nesse caso, a caixa aparecerá no diagrama abaixo das caixas que correspondem a objetos que já existiam quando a ação começa.
4. Você pode representar a destruição de um objeto com um X grande no fim da sua linha de vida. Outros objetos podem destruir um objeto e, nesse caso, uma seta aponta do outro objeto para o X. Um X é útil também para indicar que um objeto não é mais utilizável e está pronto para ser enviado à coleta de lixo.

As três últimas características são mostradas no diagrama de sequência da Figura A1.9.

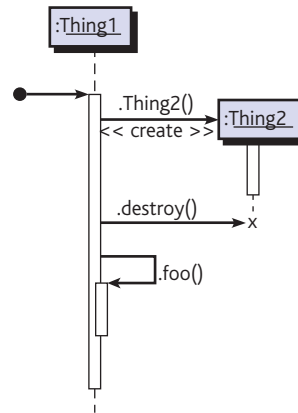


FIGURA A1.9 Criação, destruição e laços em diagramas de sequência.

Diagramas de comunicação

O *diagrama de comunicação* UML (conhecido como “diagrama de colaboração” na UML 1.X) fornece outra indicação da ordem temporal das comunicações, mas dá ênfase às relações entre os objetos e classes em vez da ordem temporal. O diagrama de comunicação está ilustrado na Figura A1.10, a qual mostra as mesmas ações do diagrama de sequência da Figura A1.7.

Em um diagrama de comunicação, os objetos que interagem são representados por retângulos. Associações entre objetos são representadas por linhas ligando os retângulos. Normalmente, há uma seta apontando para um objeto no diagrama, que inicia a sequência de passagem de mensagens. A seta é identificada com um número e um nome de mensagem. Se a mensagem que chega for identificada com o número 1 e se ela faz o objeto receptor invocar outras mensagens em outros objetos, aquelas mensagens são representadas por setas do emissor para o receptor com uma linha de associação e recebem números, 1.1, 1.2 e assim por diante, na ordem em que são chamadas. Se

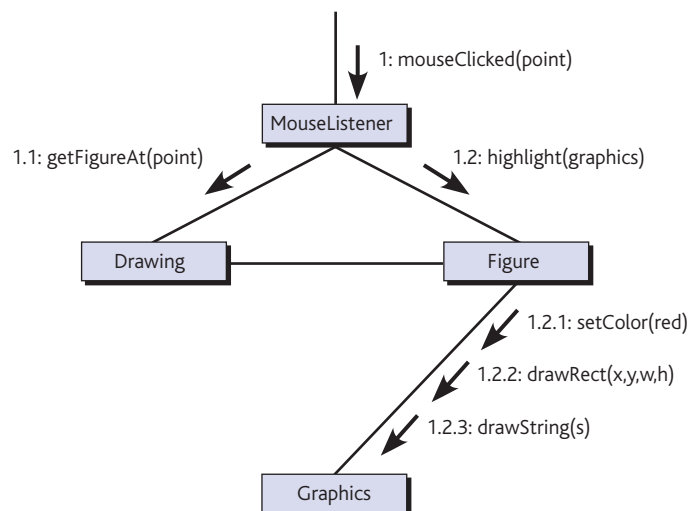


FIGURA A1.10 Um diagrama de comunicação UML.

aquelas mensagens por sua vez invocam outras, é acrescentado outro ponto e outro número ao número que as identifica, para indicar novo aninhamento da mensagem passada.

Na Figura A1.10, você vê que a mensagem **mouseClicked** chama o método *getFigureAt()* e depois *highlight()*. A mensagem *highlight()* chama três outras mensagens: *setColor()*, *drawRect()* e *drawstring()*. A numeração em cada rótulo mostra os aninhamentos, bem como a natureza sequencial de cada mensagem.

Há muitas características opcionais que podem ser acrescentadas aos rótulos das setas. Por exemplo, você pode colocar uma letra na frente do número. Uma seta chegando poderia ser marcada como A1: mouseClicked(point), indicando a execução de uma sequência de comandos (*thread*), A. Se outras mensagens são executadas em outras *threads*, o rótulo seria precedido por uma letra diferente. Por exemplo, se o método *mouseClicked()* é executado na *thread* A, mas cria uma nova *thread* B e chama *highlight()* naquela *thread*, então a seta de **MouseListener** para **Figure** seria rotulada como 1.B2: highlight(graphics).

Se você estiver interessado em mostrar as relações entre os objetos, além das mensagens que estão sendo enviadas entre eles, o diagrama de comunicação provavelmente é uma opção melhor do que o diagrama de sequência. Se estiver mais interessado na ordem temporal da mensagem enviada, o diagrama de sequência provavelmente será melhor.

Diagramas de atividade

O *diagrama de atividade* mostra o comportamento dinâmico de um sistema ou de parte de um sistema por meio do fluxo de controle entre ações que o sistema executa. Ele é similar a um fluxograma, exceto que pode mostrar fluxos concorrentes.

O componente principal de um diagrama de atividade é um nó *ação*, representado por um retângulo arredondado, que corresponde a uma tarefa executada por um sistema de software. Setas que vão de um nó ação para outro indicam o fluxo de controle. Isto é, uma seta entre dois nós ação significa que, depois que a primeira ação é completada, a segunda ação começa. Um ponto preto cheio forma o *nó inicial* que representa o ponto inicial da atividade. Um ponto preto envolvido por um círculo preto é o *nó final*, indicando o fim da atividade.

Um *fork* representa a separação de atividades em duas ou mais atividades concorrentes. Ela é desenhada como uma barra preta horizontal com uma seta apontando para ela e duas ou mais setas apontando para fora dela. Cada seta de saída representa um fluxo de controle que pode ser executado concorrentemente com os fluxos que correspondem às outras setas que saem. Essas atividades concorrentes podem ser executadas em um computador por meio de diferentes *threads* ou até mesmo usando diferentes computadores.

A Figura A1.11 mostra um exemplo de um diagrama de atividade envolvendo a confecção de um bolo. O primeiro passo é procurar a receita. Uma vez encontrada, os ingredientes secos e molhados podem ser medidos e mis-

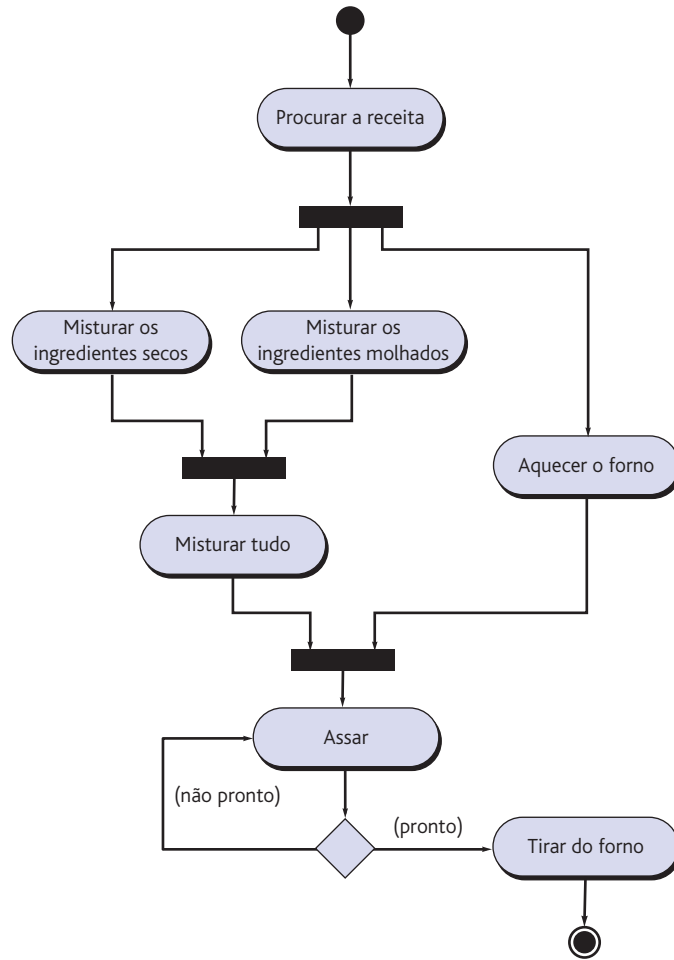


FIGURA A1.11 Um diagrama de atividade UML demonstrando como fazer um bolo.

turados e o forno pode ser pré-aquecido. A mistura dos ingredientes secos pode ser feita em paralelo com a mistura dos ingredientes molhados e com o preaquecimento do forno.

Uma *junção* (*join*) é uma maneira de sincronizar fluxos de controle concorrentes. Ela é representada por uma barra preta horizontal com duas ou mais setas chegando e uma seta saindo. O fluxo de controle representado pela seta que sai não pode iniciar a execução até que todos os outros fluxos representados pelas setas que chegam tenham sido completados. Na Figura A1.11, temos uma junção antes da ação de mistura dos ingredientes secos e molhados. Ela indica que todos os ingredientes secos e todos os ingredientes molhados devem ser misturados antes que as duas misturas possam ser combinadas. A segunda junção na figura informa que, antes de começar a assar o bolo, todos os ingredientes devem estar misturados, e o forno deve estar na temperatura correta.

Um nó *decisão* corresponde a uma ramificação no fluxo de controle baseada em uma condição. Um nó desse tipo é mostrado como um triângulo branco com uma seta chegando e duas ou mais saindo. Cada seta que sai é

identificada com uma condição entre colchetes (*guard*). O fluxo de controle segue a seta que sai cuja condição é verdadeira (*true*). É bom certificar-se de que as condições abrangem todas as possibilidades, de forma que exatamente uma delas seja verdadeira sempre que um nó de decisão for encontrado. A Figura A1.11 apresenta um nó de decisão após assar o bolo. Se o bolo está pronto, ele é retirado do forno. Caso contrário, permanece no forno por mais um tempo.

Uma das coisas que o diagrama de atividades da Figura A1.11 não mostra é quem executa cada uma das ações. Muitas vezes, a divisão exata do trabalho não importa. Mas, se você quiser indicar como as ações são divididas entre os participantes, decore o diagrama de atividades com raias (*swimlanes*), como mostra a Figura A1.12. As *raias*, como o nome diz, são formadas dividindo-se o diagrama em tiras ou “faixas”, e cada uma dessas faixas corresponde a um dos participantes. Todas as ações em uma faixa são executadas pelo participante correspondente. Na Figura A1.12, Jennie é responsável pela mistura dos ingredientes secos e depois pela mistura dos ingredientes secos e molhados juntos, Helen é responsável por aquecer o forno e tirar o bolo, e Mary, por todo o restante.

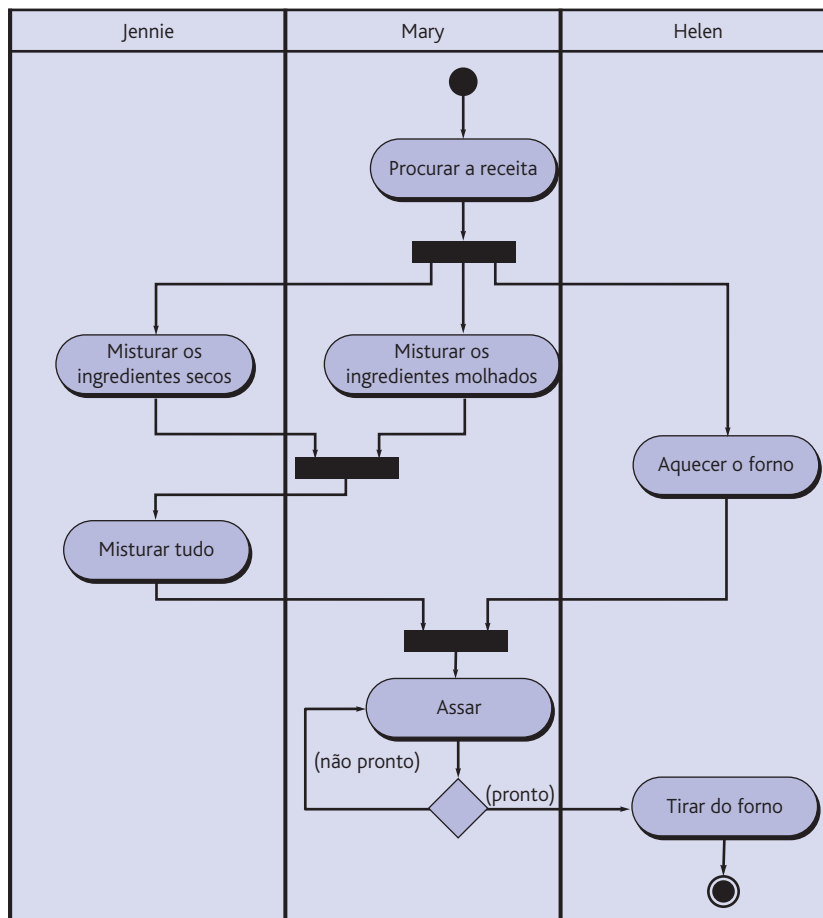


FIGURA A1.12 O diagrama de atividades da confecção do bolo com a inclusão de raias.

Diagramas de estado

O comportamento de um objeto em determinado instante frequentemente depende do seu estado, ou seja, dos valores de suas variáveis naquele instante. Como um exemplo trivial, considere um objeto com uma variável de instância booleana. Quando solicitado a executar uma operação, o objeto pode realizar algo se a variável for verdadeira (*true*) e realizar outra coisa se for falsa (*false*).

Um *diagrama de estado* modela os estados de um objeto, as ações executadas dependendo daqueles estados e as transições entre os estados do objeto.

Como exemplo, considere o diagrama de estado para uma parte de um compilador Java. A entrada do compilador é um arquivo de texto, que pode ser considerado uma longa sequência (*string*) de caracteres. O compilador lê os caracteres, um de cada vez, e a partir deles determina a estrutura do programa. Uma pequena parte desse processo consiste em ignorar caracteres “espaço em branco” (por exemplo, os caracteres *espaço*, *tabulação*, *nova linha* e *return*) e caracteres dentro de um comentário.

Suponha que o compilador delegue ao objeto **WhiteSpaceAndCommentEliminator** a tarefa de avançar sobre os caracteres espaço em branco e caracteres dentro de comentários. A tarefa desse objeto é ler os caracteres de entrada até que todos os espaços em branco e os caracteres entre comentários tenham sido lidos. Nesse ponto, ele retorna o controle para o compilador para ler e processar caracteres que não são espaços em branco e não são caracteres de comentários. Pense em como o objeto **WhiteSpaceAndCommentEliminator** lê os caracteres e determina se o próximo caractere é um espaço em branco ou se é parte de um comentário. O objeto pode verificar espaços em branco testando o próximo caractere para saber se é “ ”, “\t”, “\n”, e “\r”. Mas como ele determina se o próximo caractere faz parte de um comentário? Por exemplo, quando vê uma “/” pela primeira vez, ele ainda não sabe se aquele caractere representa um operador de divisão, parte do operador /= ou o início de uma linha ou comentário de bloco. Para determinar isso, **WhiteSpaceAndCommentEliminator** precisa registrar o fato de que viu uma “/” e então passar para o próximo caractere. Se o caractere que vem depois da “/” for outra “/” ou um “*”, então **WhiteSpaceAndCommentEliminator** saberá que está agora lendo um comentário e pode avançar até o fim sem processar ou salvar qualquer caractere. Se o caractere que vem em seguida à primeira “/” for alguma coisa diferente de uma “/” ou um “*”, o objeto **WhiteSpaceAndCommentEliminator** saberá que a “/” representa o operador divisão ou parte do operador /= e, assim, para de avançar sobre os caracteres.

Resumindo, à medida que o objeto **WhiteSpaceAndCommentEliminator** lê os caracteres, ele precisa controlar vários aspectos, incluindo se o caractere atual é um espaço em branco, se o caractere anterior que ele leu era uma “/”, se está no momento lendo caracteres de um comentário, se chegou ao fim de um comentário e assim por diante. Tudo isso corresponde a diferentes estados do objeto **WhiteSpaceAndCommentEliminator**. Em cada um desses estados, **WhiteSpaceAndCommentEliminator** se comporta de modo diferente com relação ao próximo caractere a ser lido.

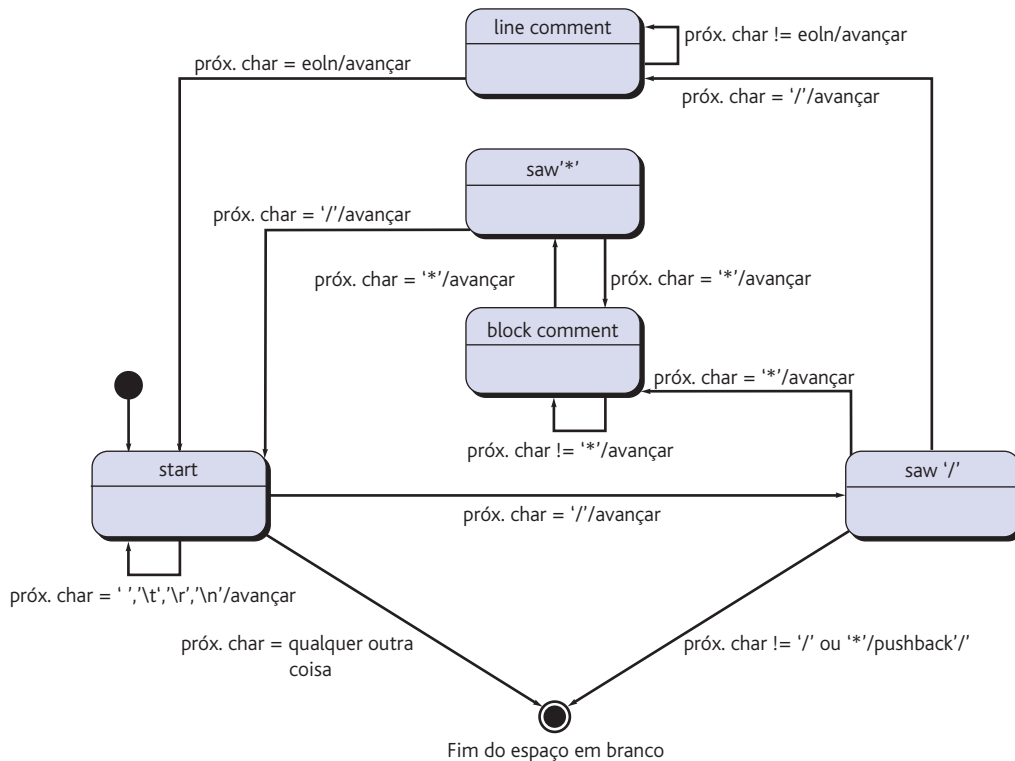


FIGURA A1.13 Um diagrama de estado para avançar além do espaço em branco e comentários em Java.

Para ajudar a visualizar todos os estados desse objeto e como ele muda o estado, você pode usar um diagrama de estado, conforme indica a Figura A1.13. Um diagrama de estado mostra os estados por meio de retângulos arredondados, cada um dos quais com um nome em sua metade superior. Há também um círculo preto, chamado de “pseudoestado inicial”, que não é realmente um estado, mas apenas um ponto para o estado inicial. Na Figura A1.13, o estado **start** é o estado inicial. Setas de um estado para outro representam transições ou mudanças no estado do objeto. Cada transição é identificada com um evento disparo, uma barra (/) e uma atividade. Todas as partes dos rótulos de transição são opcionais nos diagramas de estado. Se o objeto está em um estado e ocorre o disparo de evento para uma de suas transições, a atividade daquela transição é executada, e o objeto assume o novo estado indicado pela transição. Por exemplo, na Figura A1.13, se o objeto **WhiteSpaceAndCommentEliminator** está no estado **start** e o próximo caractere é “/”, **WhiteSpaceAndCommentEliminator** avança além daquele caractere e muda para o estado **saw ‘/’**. Se o caractere depois de “/” é outra “/”, o objeto avança para o estado **line comment** e permanece lá até ler um caractere fim de linha (eoln, end-of-line). Se, por outro lado, o próximo caractere após a “/” é um “*”, o objeto avança para o estado **block comment** e permanece lá até encontrar outro “*” seguido de uma “/”, que indica o fim de um comentário de bloco. Estude o diagrama para ter certeza de que o entendeu. Note que, após avançar além do espaço em branco ou de um comentário, o objeto **WhiteSpaceAndCommentEliminator** volta para o estado **start** e

começa tudo novamente. Esse comportamento é necessário, já que pode haver vários comentários ou caracteres de espaço em branco sucessivos antes de encontrar quaisquer outros caracteres do código-fonte Java.

Um objeto pode fazer uma transição para um estado final, indicado por um círculo preto com um círculo branco ao redor dele, que informa que não há mais transições. Na Figura A1.13, o objeto **WhiteSpaceAndCommentEliminator** é encerrado quando o próximo caractere não é um espaço em branco nem parte de um comentário. Note que todas as transições, exceto as duas que levam ao estado final, têm atividades que consistem em avançar para o próximo caractere. As duas transições para o estado final não avançam sobre o próximo caractere porque o próximo faz parte de uma palavra ou símbolo de interesse para o compilador. Note que se o objeto está no estado **saw '/'**, mas o próximo caractere não é uma **"/** ou ******, então o caractere **"/** é um operador divisão ou parte do operador **/=** e, portanto, não queremos avançar. Na verdade, queremos retroceder um caractere para tornar a **"/** o próximo caractere, para que assim o caractere **"/** possa ser usado pelo compilador. Na Figura A1.13, essa atividade de retrocesso é chamada de *pushback '/'*.

Um diagrama de estado o ajudará a descobrir situações perdidas ou inesperadas. Com um diagrama de estado, é relativamente fácil garantir que todos os eventos possíveis para todos os estados possíveis foram levados em conta. Na Figura A1.13, você pode facilmente verificar que cada estado incluiu transições para todos os caracteres possíveis.

Os diagramas de estado UML podem conter muitas outras características não incluídas na Figura A1.13. Por exemplo, quando um objeto está em um estado, ele usualmente não faz nada e espera até que ocorra um evento. No entanto, há um tipo especial de estado, denominado *estado de atividade*, no qual o objeto executa alguma atividade, chamada de *do-activity*, enquanto está naquele estado. Para indicarmos no diagrama que um estado é um estado de atividade, incluímos na metade inferior do retângulo arredondado de estado a palavra **do/**, seguida da atividade que deve ser executada enquanto estiver naquele estado. A *do-activity* pode terminar antes que ocorram quaisquer transições de estado, após as quais o estado de atividade se comporta como um estado de espera normal. Se ocorrer uma transição fora do estado atividade, antes que a *do-activity* tenha terminado, a *do-activity* será interrompida.

Como um evento é opcional quando ocorre uma transição, é possível que nenhum evento possa estar listado como parte do rótulo de uma transição. Em casos assim para estados de espera normais, o objeto fará imediatamente uma transição daquele estado para o novo estado. Para estados atividade, uma transição dessas ocorre logo que *do-activity* termina.

A Figura A1.14 ilustra essa situação usando estados para um telefone comercial. Quando uma chamada é colocada em espera, vai para o estado **on hold with music** (toca música suave por 10 segundos). Após 10 segundos, a *do-activity* do estado é completada, e o estado se comporta como um estado normal de não atividade. Se a pessoa que está chamando pressiona a tecla # quando a chamada está no estado **on hold with music**, a chamada faz uma transição para o estado **canceled**, e o telefone passa imediatamente para o estado **dial tone**. Se a tecla # é pressionada antes de completar os 10 segundos de música de espera, a *do-activity* é interrompida, e a música para imediatamente.

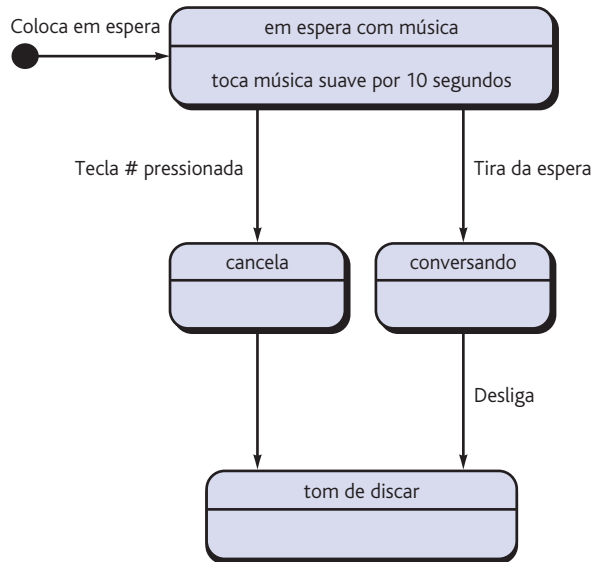


FIGURA A1.14 Um diagrama de estado com um estado de atividade e uma transição sem evento.

Object Constraint Language – um panorama

A grande variedade de diagramas disponíveis como parte da UML proporciona um excelente conjunto de formas de representação para o modelo de projeto. No entanto, as representações gráficas muitas vezes não são suficientes. Talvez você precise de um mecanismo para representar explícita e formalmente informações que restringem alguns elementos do modelo de projeto. É possível, naturalmente, descrever as restrições em uma linguagem natural como o inglês, mas essa abordagem invariavelmente leva à inconsistência e à ambiguidade. Por essa razão, parece ser apropriada uma linguagem mais formal – uma que use a teoria dos conjuntos e linguagens de especificação formal (consulte o Capítulo 28 e o Apêndice 3), mas tenha de certa forma a sintaxe menos matemática de uma linguagem de programação.

A *Object Constraint Language* (OCL) complementa a UML, permitindo que você use uma gramática e sintaxe formais para construir instruções não ambíguas sobre vários elementos do modelo de projeto (por exemplo, classes e objetos, eventos, mensagens, interfaces). As instruções OCL mais simples são criadas em quatro partes: (1) um *contexto* que define a situação limitada na qual a instrução é válida, (2) uma *propriedade* que representa algumas características do contexto (por exemplo, se o contexto é uma classe, uma propriedade pode ser um atributo), (3) uma *operação* (por exemplo, aritmética, orientada a conjunto) que manipula ou qualifica uma propriedade e (4) palavras-chave (por exemplo, **if**, **then**, **else**, **and**, **or**, **not**, **implies**) usadas para especificar expressões condicionais.

Como um exemplo simples de uma expressão OCL, considere o sistema de impressão discutido no Capítulo 14. A condição de guarda (*guard condition*) colocada no evento **jobCostAccepted** que causa uma transição entre os estados *computingJobCost* e *formingJob* dentro do diagrama de estado (*statechart*

diagram) para o objeto **PrintJob** (Figura 14.9). No diagrama (Figura 14.9), a condição de guarda é expressa em linguagem natural e implica que a autorização só pode ocorrer se o cliente está autorizado a aprovar o custo do trabalho. Em OCL, a expressão pode assumir a forma:

```
customer
  self.authorizationAuthority = 'yes'
```

onde um atributo booleano, **authorizationAuthority**, da classe (na realidade uma instância específica da classe) denominada **Customer** deve ser definida como **yes** para que a condição de guarda seja satisfeita.

À medida que o modelo de projeto é criado, muitas vezes há instâncias nas quais pré ou pós-condições devem ser satisfeitas antes de completar alguma ação especificada pelo projeto. A OCL fornece uma poderosa ferramenta para especificar pré e pós-condições de maneira formal. Como exemplo, considere uma ampliação do sistema de impressão (discutido como exemplo no Capítulo 14), no qual o cliente estabelece um limite superior de custo para a impressão e uma data final de entrega no instante em que são especificadas outras características da impressão. Se as estimativas de custo e prazos de entrega excederem esses limites, o trabalho não é aceito, e o cliente deve ser notificado. Em OCL, um conjunto de pré e pós-condições pode ser especificado da seguinte maneira:

```
context PrintJob::validate(upperCostBound : Integer, custDeliveryReq :
  Integer)
pre: upperCostBound > 0
  and custDeliveryReq > 07
  and self.jobAuthorization = 'no'
post: if self.totalJobCost <= upperCostBound
  and self.deliveryDate <= custDeliveryReq
then
  self.jobAuthorization = 'yes'
endif
```

Essa instrução OCL define uma invariante (**inv**) – condições que devem existir antes de (pre) e após (post) algum comportamento. Inicialmente, uma pré-condição estabelece que um limite de custo e de data de entrega deve ser especificado pelo cliente e a autorização deve ser definida como “no”. Depois que os custos e o prazo de entrega são determinados, é aplicada a pós-condição especificada. Deve-se notar também que, para a expressão

```
self.jobAuthorization = 'yes'
```

não é atribuído o valor “yes”, mas está declarando que **jobAuthorization** deve ser definido como “yes” quando a operação terminar. Uma descrição completa da OCL está fora dos objetivos deste apêndice. A especificação OCL completa pode ser obtida em www.omg.org/technology/documents/formal/ocl.htm.⁴

⁴ Uma descrição da versão mais recente da OCL pode ser encontrada em <http://www.omg.org/spec/OCL/2.3.1/>.

Leituras e fontes de informação complementares

Há dezenas de livros com discussões sobre UML. Livros de Hay (*UML and Data Modeling: A Reconciliation*, Technics Publications, 2011; e *Enterprise Model Patterns: Describing the World*, Technics Publications, 2011), Gomaa (*Software Modeling and Design: UML, Use Cases, Patterns, and Software Architecture*, Cambridge University Press, 2011; e *Requirements Engineering: From System Goals to UML Models to Software Specifications*, Wiley, 2009) e Podesa (*UML for the IT Business Analyst*, Course Technology, 2009) discutem o uso de UML em desenvolvimento de software. Outros livros que fornecem informações úteis incluem: Miles e Hamilton (*Learning UML 2.0*, O'Reilly Media, 2006), Booch, Rumbaugh e Jacobson (*Unified Modeling Language User Guide*, 2ª ed., Addison-Wesley, 2005), Ambler (*The Elements of UML 2.0 Style*, Cambridge University Press, 2005) e Pilone e Pitman (*UML 2.0 in a Nutshell*, O'Reilly Media, 2005).

Há disponível na Internet uma ampla variedade de fontes de informação sobre o uso da UML na modelagem de engenharia de software. Uma lista atualizada das referências da Web pode ser encontrada em “análise” e “projeto” no site: www.mhhe.com/pressman.

Esta página foi deixada em branco intencionalmente.

Conceitos de orientação a objetos

2

O que é um ponto de vista orientado a objetos? Por que um método é considerado orientado a objetos? O que é um objeto? À medida que os conceitos de orientação a objetos ganharam ampla aceitação, durante as décadas de 1980 e 1990, surgiram muitas opiniões diferentes sobre as respostas corretas a essas perguntas, mas atualmente há uma visão mais sólida dos conceitos de orientação a objetos. Este apêndice fornecerá uma visão rápida do assunto e apresentará os conceitos básicos e a terminologia.

Para entender o ponto de vista orientado a objetos, considere um exemplo prático – o objeto sobre o qual você está sentado(a) agora –, uma cadeira (*chair*). **Chair** é uma subclasse de uma classe muito maior que chamamos de **PieceOfFurniture** (PeçaDeMobília). Cadeiras individuais são membros (usualmente chamados de instâncias) da classe **Chair**. Um conjunto de atributos genéricos pode ser associado a cada objeto da classe **PieceOfFurniture**. Por exemplo, todos os móveis têm um custo, dimensões, peso, localização e cor, entre muitos outros atributos possíveis. Isso se aplica independentemente de estarmos falando de uma mesa (*table*) ou uma cadeira (*chair*), um sofá (*sofa*) ou um armário (*armoire*). Como **Chair** é membro de **PieceOfFurniture**, **Chair** herda todos os atributos definidos para a classe.

Tentamos dar uma definição bem-humorada de uma classe descrevendo seus atributos, mas algo está faltando. Todo objeto da classe **PieceOfFurniture** pode ser manipulado de várias maneiras. Ele pode ser comprado e vendido, modificado fisicamente (por exemplo, você pode cortar uma perna da cadeira ou pintar o objeto de roxo) ou movido de um lugar para outro. Cada uma dessas *operações* (outros termos são *serviços* ou *métodos*) modificará um ou mais atributos do objeto. Por exemplo, se o atributo **localização** for um item de dado composto definido como

localização = edifício + piso + sala

uma operação denominada *move()* modificaria um ou mais dos itens de dados (**edifício**, **piso** ou **sala**) que formam o atributo **localização**. Para isso, *move* deve ter “conhecimento” desses itens de dados. A operação *move()* poderia ser usada para uma cadeira ou mesa, já que ambas são instâncias da classe **PieceOfFurniture**. Operações válidas para a classe **PieceOfFurniture** – *buy()*, *sell()*, *weigh()* – são especificadas como parte da definição de classe e herdadas por todas as instâncias da classe.

A classe **Chair** (e todos os objetos em geral) encapsula dados (os valores de atributo que definem a cadeira), operações (as ações aplicadas para mudar os atributos da cadeira), outros objetos, constantes (configuração de valores) e outras informações relacionadas. *Encapsulamento* significa que todas essas

Conceitos-Chave

atributos.....	893
classes.....	892
características das ..	897
controladoras	894
definição de	892
de entidade.....	894
de fronteira.....	894
encapsulamento	891
herança.....	894
mensagens.....	895
métodos.....	893
operações.....	893
polimorfismo.....	896
serviços.....	893
subclasse	892
superclasse.....	892

informações são empacotadas sob um nome e podem ser reutilizadas como uma especificação ou componente de programa.

Agora que introduzimos alguns conceitos básicos, terá mais sentido uma definição formal de *orientado a objeto*. Coad e Yourdon [Coa91] definem o termo da seguinte maneira:

Orientado a objeto = objetos + classificação + herança + comunicação

Três desses conceitos já foram apresentados. A comunicação será discutida mais adiante neste apêndice.

Classes e objetos

Classe é um conceito de orientação a objeto que encapsula os dados e as abstrações procedurais necessárias para descrever o conteúdo e o comportamento de alguma entidade do mundo real. Abstrações de dados que descrevem a classe são envolvidas por uma “parede” de abstrações procedurais [Tay90] (representada na Figura A2.1) capazes de manipular os dados de certa maneira. Em uma classe bem projetada, a única maneira de atingir os atributos (e operar sobre eles) é passando por um dos métodos que formam a “parede” ilustrada na figura. Portanto, a classe encapsula dados (dentro da parede) e o processamento que manipula os dados (os métodos que formam a parede). Isso possibilita o encapsulamento de informações (Capítulo 12) e reduz o impacto de efeitos colaterais associados às mudanças. Como os métodos tendem a manipular um número limitado de atributos, sua coesão é melhorada, e como a comunicação ocorre somente por meio dos métodos que formam a “parede”, a classe tende a ser menos fortemente acoplada em relação a outros elementos de um sistema.¹

Em outras palavras, podemos dizer que classe é uma descrição generalizada (por exemplo, um modelo ou esquema) que descreve uma coleção de objetos similares. Por definição, objetos são instâncias de uma classe específica e herdam seus atributos e propriedades disponíveis para manipular os atributos. Uma *superclasse* (muitas vezes chamada de *classe base*) é a generalização de um conjunto de classes relacionadas a ela. Uma *subclasse* é uma especialização da superclasse. Por exemplo, a superclasse **MotorVehicle** é uma generalização da classe **Truck**, **SUV**, **Automobile** e **Van**. A subclasse **Automobile** herda todos os atributos de **MotorVehicle**, mas, além disso, incorpora outros atributos específicos apenas aos automóveis.

Essas definições implicam a existência de uma hierarquia de classes na qual atributos e operações da superclasse são herdados por subclasses que podem, cada uma delas, acrescentar atributos e métodos “privados” adicionais. Por exemplo, as operações *sitOn()* e *turn()* podem ser privadas à subclasse **Chair**.

¹ Deve-se notar, no entanto, que o acoplamento pode se tornar um sério problema em sistemas orientados a objetos. Ele surge quando classes de várias partes do sistema são usadas como tipos de dados de atributos e argumentos para métodos. Apesar do acesso aos objetos poder ser apenas por meio de chamadas de procedimento, isso não significa que o acoplamento seja necessariamente baixo, mas apenas menor do que seria se fosse permitido o acesso direto às partes internas do objeto.

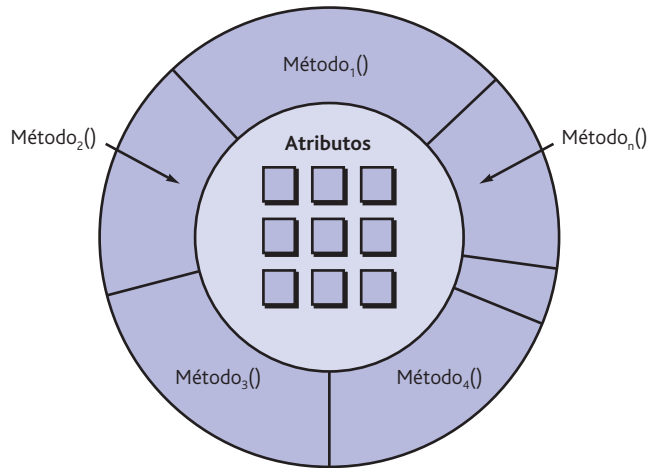


FIGURA A2.1 Representação esquemática de uma classe.

Atributos

Você já aprendeu que os atributos são anexados às classes e que eles as descrevem de certa maneira. O atributo pode assumir um valor definido por um *domínio* enumerado. Em muitos casos, domínio é apenas um conjunto de valores específicos. Suponha que uma classe **Automobile** tenha o atributo *cor*. O domínio dos valores de *cor* é {**white**, **black**, **silver**, **gray**, **blue**, **red**, **yellow**, **green**}. Em situações mais complexas, o domínio pode ser uma classe. Continuando o exemplo, a classe **Automobile** também tem um atributo *powerTrain* (conjunto de tração) que por si só é uma classe. A classe **PowerTrain** teria os atributos que descrevem o motor e a transmissão específicos para determinado carro.

As *características* (valores do domínio) podem ser ampliadas atribuindo-se um valor-padrão (característica) a um atributo. Por exemplo, por padrão, o atributo *cor* é **white**. Também pode ser útil associar uma probabilidade a determinada característica atribuindo-se pares {valor, probabilidade}. Considere o atributo *cor* para automóvel. Em algumas aplicações (planejamento de manufatura), pode ser necessário atribuir uma probabilidade a cada uma das cores (branco e preto são altamente prováveis como cores de automóveis).

Operações, métodos e serviços

Um objeto encapsula dados (representados como uma coleção de atributos) e algoritmos que processam os dados. Esses algoritmos são chamados de *operações*, *métodos* ou *serviços*² e podem ser vistos como componentes de processamento.

Cada uma das operações encapsulada por um objeto proporciona uma representação de um dos comportamentos do objeto. A operação *GetColor()* para o objeto **Automobile** extrairá a cor armazenada no atributo *cor*. A impli-

² No contexto desta discussão, é usado o termo *operações*, mas os termos *métodos* e *serviços* são igualmente populares.

cação da existência dessa operação é que a classe **Automobile** foi projetada para receber um estímulo (chamamos o estímulo de *mensagem*) que solicita a cor da instância particular de uma classe. Sempre que um objeto recebe um estímulo, ele inicia algum comportamento. Isso pode ser algo simples como obter a cor do automóvel ou complexo como o início de uma cadeia de estímulos passados entre uma variedade de objetos diferentes. Neste último caso, considere um exemplo no qual o estímulo inicial recebido por **Object 1** resulta na geração de dois outros estímulos enviados para **Object 2** e **Object 3**. Operações encapsuladas pelo segundo e terceiro objetos agem sobre o estímulo, retornando as informações necessárias para o primeiro objeto. **Object 1** usa as informações retornadas para satisfazer o comportamento exigido pelo estímulo inicial.

Conceitos de análise e projeto orientado a objetos

A modelagem de requisitos (também chamada de modelagem de análise) concentra-se primeiro nas classes extraídas diretamente do enunciado do problema. Essas *classes de entidade* tipicamente representam itens que devem ser armazenados em um banco de dados e persistem durante toda a aplicação (a menos que sejam excluídas especificamente).

O projeto refina e amplia o conjunto de classes de entidade. Classes de fronteira e controladoras são desenvolvidas e/ou refinadas durante o projeto. *Classes de fronteira* criam a interface (por exemplo, telas interativas e relatórios impressos) que o usuário vê e com os quais interage à medida que o software é usado. Classes de fronteira são projetadas com a responsabilidade de controlar a maneira como os objetos entidade são representados para os usuários.

Classes controladoras são projetadas para controlar (1) a criação ou atualização de objetos entidade, (2) a instanciação de objetos de fronteira, já que obtêm informações dos objetos entidade, (3) a comunicação complexa entre conjuntos de objetos e (4) a validação dos dados transferidos entre objetos ou entre o usuário e a aplicação.

Os conceitos discutidos nos próximos parágrafos podem ser úteis no trabalho de análise e projeto.

Herança. É um dos diferenciadores-chave entre sistemas convencionais e orientados a objeto. Uma subclasse **Y** herda todos os atributos e operações associados à sua superclasse **X**. Isso significa que todas as estruturas de dados e algoritmos originalmente projetados e implementados para **X** ficam imediatamente disponíveis para **Y** – nenhum trabalho adicional precisa ser feito. A reutilização foi conseguida diretamente.

Qualquer alteração nos atributos ou nas operações contidas dentro de uma superclasse é imediatamente herdada por todas as subclasses. Portanto, a hierarquia de classes torna-se um mecanismo pelo qual alterações (em altos níveis) podem ser imediatamente propagadas em um sistema.

É importante notar que, em cada nível de hierarquia de classe, novos atributos e operações podem ser acrescentados àqueles que foram herdados de

níveis mais altos na hierarquia. De fato, sempre que uma nova classe precisa ser criada, você tem um conjunto de opções:

- A classe pode ser projetada e construída a partir do zero. Isto é, não é usada a herança.
- A hierarquia de classes pode ser pesquisada para determinar se uma classe mais alta na hierarquia contém grande parte dos atributos e operações necessários. A nova classe herda da classe mais alta e podem ser feitas adições quando necessário.
- A hierarquia de classes pode ser reestruturada para que os atributos e operações necessários possam ser herdados pela nova classe.
- As características de uma classe existente podem ser anuladas, e diferentes versões de atributos ou operações são implementadas para a nova classe.

Como todos os conceitos fundamentais de projeto, a herança pode proporcionar benefício significativo ao projeto, mas, se ela for usada de forma não apropriada,³ pode complicar um projeto desnecessariamente e levar a um software passível de erros e difícil de manter.

Mensagens. As classes devem interagir umas com as outras para atingir os objetivos do projeto. Uma mensagem estimula a ocorrência de algum comportamento no objeto receptor. O comportamento ocorre quando uma operação é executada.

A interação entre objetos é apresentada na Figura A2.2. Uma operação dentro de **SenderObject** gera uma mensagem da forma *mensagem* (<parâmetros>) em que os parâmetros identificam **ReceiverObject** como o objeto a ser estimulado pela mensagem, a operação dentro de **ReceiverObject** que deve receber a mensagem e os itens de dados que fornecem as informações necessárias para que a operação seja bem-sucedida. A colaboração definida entre classes como parte do modelo de análise fornece diretrizes úteis na criação das mensagens.

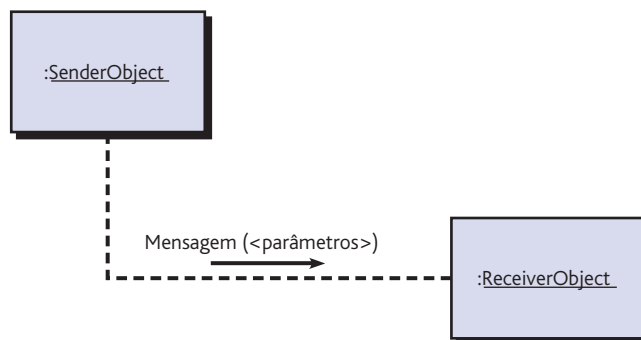


FIGURA A2.2 Passagem de mensagens entre objetos.

³ Projetar uma subclasse que herda atributos e operações de mais de uma superclasse (às vezes chamada de “herança múltipla”) não é visto com simpatia por muitos projetistas.

Cox [Cox86] descreve o intercâmbio entre classes da seguinte maneira:

É solicitado a um objeto `lclassel` que execute uma de suas operações enviando-lhe uma mensagem com as informações do que fazer. O receptor `lobjetol` responde à mensagem primeiro escolhendo a operação que implementa o nome da mensagem, executando essa operação e, então, retornando o controle para o chamador. As mensagens unem um sistema orientado a objetos. As mensagens fornecem informações sobre o comportamento dos objetos individuais e do sistema orientado a objetos como um todo.

Polimorfismo. Característica que reduz bastante o esforço necessário para ampliar o projeto de um sistema orientado a objetos. Para entender o polimorfismo, considere uma aplicação convencional que deve traçar quatro tipos diferentes de gráficos: gráficos de colunas, gráficos de pizza, histogramas e diagramas Kiviat. De maneira ideal, uma vez coletados os dados para um tipo particular de gráfico, o gráfico será traçado. Para conseguir isso em uma aplicação convencional (e manter a coesão do módulo), seria necessário desenvolver módulos de desenho para cada tipo de gráfico. Então, no projeto, estaria embutida lógica de controle similar a esta a seguir:

```
case of graphtype:
    if graphtype = linegraph then DrawLineGraph (data);
    if graphtype = piechart then DrawPieChart (data);
    if graphtype = histogram then DrawHisto (data);
    if graphtype = kiviatic then DrawKiviat (data);
end case;
```

Embora esse projeto seja razoavelmente simples, seria complicado adicionar novos tipos de gráficos. Um novo módulo de desenho teria de ser criado para cada tipo de gráfico, e a lógica de controle teria de ser atualizada para refletir o novo tipo de gráfico.

Para resolver esse problema, todos os gráficos se tornam subclasses de uma classe geral denominada **Graph**. Usando um conceito chamado *sobrecarga* [Tay90], cada subclasse define uma operação *draw*. O objeto pode enviar uma mensagem *draw* a qualquer um dos objetos instanciados a partir de qualquer uma das subclasses. O objeto que está recebendo a mensagem chamará sua própria operação *draw* para criar o gráfico apropriado. Portanto, o projeto é reduzido a

```
draw <graphtype>
```

Quando um novo tipo de gráfico é acrescentado ao sistema, cria-se uma subclasse com sua própria operação *draw*. Mas não são necessárias alterações em qualquer objeto que queira que um gráfico seja desenhado, pois a mensagem *draw <graphtype>* permanece inalterada. Resumindo, o polimorfismo permite que várias operações diferentes tenham o mesmo nome. Isso, por sua vez, desacopla os objetos uns dos outros, tornando-os mais independentes.

Classes de projeto. O modelo de requisitos define um conjunto completo de classes de análise. Cada uma descreve algum elemento do domínio do problema, focalizando os aspectos visíveis ao usuário ou ao cliente. O nível de abstração de uma classe de análise é relativamente alto.

Conforme o modelo de projeto evolui, a equipe de software deve definir um conjunto de *classes de projeto* que (1) refina as classes de análise, fornecendo detalhe de projeto que permitirá que sejam implementadas e (2) crie um novo conjunto de classes de projeto que implemente uma infraestrutura de software que suporte a solução de negócio. São sugeridos cinco tipos diferentes de classes de projeto, cada um representando uma camada diferente da arquitetura de projeto [Amb01]:

- *Classes de interfaces do usuário* definem todas as abstrações necessárias para a interação humano-computador (*human-computer interaction*, HCI).
- *Classes de domínio de negócio* normalmente são refinamentos das classes de análise definidas anteriormente. As classes identificam os atributos e serviços (métodos) necessários para implementar algum elemento do domínio de negócio.
- *Classes de processos* implementam as abstrações de aplicação de baixo nível necessárias para a completa gestão das classes de domínio de negócio.
- *Classes persistentes* representam repositórios de dados (por exemplo, um banco de dados) que persistirá depois da execução do software.
- *Classes de sistema* implementam funções de gerenciamento e controle de software que permitem ao sistema operar e comunicar-se em seu ambiente computacional e com o mundo exterior.

À medida que o projeto arquitetural evolui, a equipe de software deve desenvolver um conjunto completo de atributos e operações para cada classe de projeto. O nível de abstração é reduzido conforme cada classe de análise é transformada em uma representação de projeto. Isto é, classes de análise representam objetos (e métodos associados aplicados a eles), usando o jargão do domínio de negócio. As classes de projeto apresentam um detalhe significativamente mais técnico como guia para a implementação.

Arlow e Neustadt [Arl02] sugerem que cada classe de projeto seja revista para garantir sua “boa formação”. Eles definem quatro características de uma classe de projeto bem formada:

Completa e suficiente. Uma classe de projeto deve ser o encapsulamento completo de todos os atributos e métodos exigidos para uma classe (com base em uma interpretação reconhecível do nome da classe). Por exemplo, a classe **Cena** definida para software de edição de vídeo será completa apenas se contiver todos os atributos e métodos que podem ser razoavelmente associados à criação de uma cena de vídeo. A suficiência garante que a classe de projeto contenha somente os métodos necessários para atingir a finalidade da classe, nem mais nem menos.

Primitivismo. Os métodos associados a uma classe de projeto devem se concentrar na execução de uma função específica para a classe. Uma vez que a função tenha sido implementada com um método, a classe não deve proporcionar nenhuma outra maneira de fazê-lo. Por exemplo, a classe **VideoClip** do software de edição de vídeo pode ter atributos *start-point* e *end-point* para indicar os pontos de início e fim do videoclipe (note que o

vídeo “bruto” carregado no sistema pode ser maior do que o videoclipe usado). Os métodos, *estabelecerPontoInicial()* e *estabelecerPontoFinal()*, fornecem os únicos meios para estabelecer os pontos de início e fim do clipe.

Alta coesão. Uma classe de projeto coesa é limitada. Ela tem um conjunto de responsabilidades pequeno e concentrado e aplica atributos e métodos de forma simples para implementar essas responsabilidades. Por exemplo, a classe **VideoClip** do software de edição de vídeo pode conter um conjunto de métodos para editar o videoclipe. Contanto que cada método se concentre somente nos atributos associados ao videoclipe, a coesão é mantida.

Baixo acoplamento. No modelo de projeto, é necessário que as classes de projeto colaborem umas com as outras. No entanto, a colaboração deve ser mantida em um nível mínimo aceitável. Se um modelo de projeto é altamente acoplado (todas as classes de projeto colaboram com todas as outras classes de projeto), o sistema é difícil de implementar, testar e manter com o decorrer do tempo. Em geral, classes de projeto em um subsistema devem ter apenas um conhecimento limitado das outras classes. Essa restrição, chamada *lei de Demeter* [Lie03], sugere que um método deve enviar mensagens apenas para métodos de classes vizinhas.⁴

Leituras e fontes de informação complementares

Durante as últimas três décadas, centenas de livros foram escritos sobre programação, análise e projeto orientados a objetos. Weisfeld (*The Object-Oriented Thought Process*, 4ª ed., Addison-Wesley, 2013) apresenta um excelente tratamento dos conceitos e princípios gerais orientados a objetos. Wong e Nguyen (*Principles of Object-Oriented Programming*, Amazon Digital Services, 2011) fornece um tutorial prático sobre muitos conceitos importantes orientados a objetos. McLaughlin e seus colegas (*Head First Object-Oriented Analysis and Design: A Brain Friendly Guide to OOA&D*, O'Reilly Media, 2006) escreveram uma abordagem acessível e leve sobre estratégias de análise e projeto orientados a objetos. Keogh e Gianni (*OOP Demystified*, McGraw-Hill, 2004) oferecem um guia útil sobre o assunto.

Um tratamento mais aprofundado sobre análise e projeto orientados a objetos é apresentado por Booch e seus colegas (*Object-Oriented Analysis and Design with Applications*, 3ª ed., Addison-Wesley, 2007). Clark (*Beginning C# Object-Oriented Programming*, Apress, 2013), Kochen (*Programming in Objective-C*, 5ª ed., Developer's Library, 2012), Phillips (*Python 3 Object Oriented Programming*, Packt Publishing, 2011), Khurana (*Object-Oriented Programming with C++*, Vikas Publishing, 2010) e Wu (*An Introduction to Object-Oriented Programming with Java*, 2ª ed., McGraw-Hill, 2009) são representativos de dezenas de títulos sobre orientação a objetos escritos para muitas linguagens de programação diferentes.

Uma grande variedade de fontes de informação sobre tecnologias orientadas a objeto está disponível na Internet. Uma lista atualizada das referências da Web pode ser encontrada em “análise” e “projeto” no site: www.mhhe.com/pressman.

⁴ Uma maneira menos formal de expressar a lei de Demeter seria: “Cada unidade deve conversar apenas com seus amigos; não converse com estranhos”.

Aplicação de notação matemática¹ para especificação formal

Para ilustrarmos o uso da notação matemática na especificação formal de um componente de software, examinamos novamente o exemplo de tratador de blocos apresentado no Capítulo 28. O sistema de gerenciamento de blocos está representado esquematicamente na Figura 28.8 e deve ser examinado antes de continuarmos aqui.

Um conjunto denominado *BLOCKS* será formado por todos os números de bloco. *AllBlocks* é um conjunto de blocos que estão entre 1 e *MaxBlocks*. O estado será modelado por dois conjuntos e uma sequência. Os dois conjuntos são *used* e *free*. Ambos contêm blocos – o conjunto *used* contém os blocos que estão atualmente em uso em arquivos e o conjunto *free* contém blocos que estão disponíveis para novos arquivos. A sequência vai conter conjuntos de blocos que estão prontos para ser liberados dos arquivos que foram excluídos. O estado pode ser descrito como

$$used, free : \mathbb{P} BLOCKS$$

$$BlockQueue : seq \mathbb{P} BLOCKS$$

Isso é muito semelhante à declaração de variáveis de programa. Estamos dizendo que *used* e *free* serão conjuntos de blocos e que *BlockQueue* será uma sequência em que cada elemento será um conjunto de blocos. A invariante de dados pode ser escrita como

$$used \cap free = \emptyset \wedge$$

$$used \cup free = AllBlocks \wedge$$

$$\forall i: \text{dom } BlockQueue \bullet BlockQueue \subseteq used \wedge$$

$$\forall i, j: \text{dom } BlockQueue \bullet i \neq j \Rightarrow BlockQueue i \cap BlockQueue j = \emptyset$$

A primeira linha da invariante de dados informa que não haverá blocos comuns nas coleções de blocos usados e blocos livres. A segunda linha diz que a coleção de blocos usados e blocos livres sempre será igual à coleção total de blocos no sistema. A terceira linha indica que o *i*-ésimo elemento na fila de blocos sempre será um subconjunto dos blocos usados. A linha final diz que, para quaisquer dois elementos da fila de blocos que não sejam o mesmo bloco, não haverá blocos comuns nesses dois elementos.

Conceitos-chave

linguagem de	
especificação Z	904
exemplo de	906
notação	905
linguagens de	
especificação formal	900
Object Constraint	
Language (OLC)	901
exemplo de	903
notação	902

¹ Escrevemos esta seção do Apêndice 3 baseando-nos na hipótese de que você conhece a notação matemática associada a conjuntos e sequências e a notação lógica usada no cálculo de predicados. Se precisar de uma revisão, há um breve resumo apresentado como recurso suplementar no site (em inglês) da 8ª edição. Para informações mais detalhadas, veja [Jec06] ou [Pot04].

A primeira operação a ser definida é aquela que remove um elemento do início da fila de blocos. A pré-condição é que deve haver pelo menos um item na fila:

$$\#BlockQueue > 0,$$

A pós-condição é que o início da fila deve ser removido e colocado na coleção de blocos livres e a fila deve ser ajustada para mostrar remoção:

$$used' = used \setminus head\ BlockQueue \wedge$$

$$free' = free \cup head\ BlockQueue \wedge$$

$$BlockQueue' = tail\ BlockQueue$$

Uma convenção usada em muitos métodos formais é que o valor de uma variável após uma operação recebe uma indicação por meio de um apóstrofo. Portanto, o primeiro componente da expressão anterior diz que os novos blocos usados (*used'*) serão iguais aos blocos usados anteriores menos os blocos removidos. O segundo componente diz que os novos blocos livres (*free'*) serão os blocos livres anteriores com o início da fila de blocos acrescentado a eles. O terceiro componente diz que a nova fila de blocos será igual à cauda do valor anterior da fila de blocos, isto é, todos os elementos na fila exceto o primeiro. Uma segunda operação adiciona uma coleção de blocos, *Ablocks*, à fila de blocos. A pré-condição é que *Ablocks* seja atualmente um conjunto de blocos usados:

$$Ablocks \subseteq used$$

A pós-condição é que o conjunto de blocos seja acrescentado ao fim da fila de blocos e que o conjunto de blocos usados e livres permaneça inalterado:

$$BlockQueue' = BlockQueue \smallfrown (Ablocks) \wedge$$

$$used' = used \wedge$$

$$free' = free$$

Não há dúvida de que a especificação matemática da fila de blocos é consideravelmente mais rigorosa do que uma narrativa em linguagem natural ou um modelo gráfico. O rigor adicional exige esforço, mas os benefícios obtidos da melhor consistência e completude podem ser justificados para alguns domínios de aplicação.

Linguagens de especificação formal

Uma linguagem de especificação formal em geral é composta por três componentes principais: (1) uma sintaxe que define a notação especial com a qual a especificação é representada, (2) o domínio semântico para ajudar a definir um “universo de objetos” [Win90] que será usado para descrever o sistema e (3) um conjunto de relações que definem as regras semânticas que indicam como os objetos podem ser manipulados adequadamente e satisficam à especificação.

Como represento pré e pós-condições?

O domínio sintático de uma linguagem de especificação formal muitas vezes é baseado em sintaxe derivada da notação padrão da teoria dos conjuntos e do cálculo de predicados. O *domínio semântico* de uma linguagem de especificação indica como a linguagem representa os requisitos do sistema.

Há em uso hoje uma variedade de linguagens de especificação formal. OCL [OMG03b], Z [ISO02], LARCH [Gut93] e VDM [Jon91] são linguagens de especificação formal representativas que possuem as características já mencionadas. Neste apêndice, apresentamos uma breve discussão sobre OCL e Z.

Object Constraint Language (OCL)²

Object Constraint Language (OCL) é uma notação formal desenvolvida de forma que os usuários da UML possam dar mais precisão às suas especificações. Todos os poderes da lógica e da matemática discreta estão disponíveis na linguagem. No entanto, os projetistas da OCL decidiram que apenas os caracteres ASCII (em vez da notação matemática convencional) deveriam ser usados em instruções OCL.

Para usar a OCL, começa-se com um ou mais diagramas UML – mais comumente diagramas de classe, estado ou atividade (Apêndice 1). São acrescentadas as expressões OCL e declaram-se fatos sobre elementos dos diagramas. Essas expressões são chamadas de *restrições (constraints)*; qualquer implementação derivada do modelo deve assegurar que cada uma das restrições permaneça sempre verdadeira.

Como uma linguagem de programação orientada a objetos, uma expressão OCL envolve operadores atuando sobre objetos. No entanto, o resultado de uma expressão completa deve ser sempre booleano, isto é, verdadeiro ou falso. Os objetos podem ser instâncias da classe **Collection** da OCL, da qual **Set** e **Sequence** são duas subclasses.

O objeto **self** é o elemento do diagrama UML em cujo contexto a expressão OCL está sendo avaliada. Outros objetos podem ser obtidos por *navegação*, usando-se o símbolo . (ponto) do objeto **self**. Por exemplo:

- Se **self** é a classe **C**, com atributo **a**, **self.a** é avaliado como o objeto armazenado em **a**.
- Se **C** tem uma associação um-para-muitos chamada *assoc* com outra classe **D**, **self.assoc** é avaliado como um **Set** cujos elementos são do tipo **D**.
- Por fim (e um pouco mais sutilmente), se **D** tem atributo **b**, a expressão **self.assoc.b** é avaliada como o conjunto de todos os **bs** pertencentes a todos os **Ds**.

A OCL oferece operações internas implementando operadores de conjunto e de lógica, especificação construtiva e matemática relacionada. Um pequeno exemplo é apresentado na Tabela A3.1.

² Esta seção teve a contribuição do professor Timothy Lethbridge, da Universidade de Ottawa, e foi apresentada aqui com sua permissão.

Tabela A3.1 Resumo da notação ocl fundamental

$x.y$	Obtém a propriedade y do objeto x . Uma propriedade pode ser um atributo, o conjunto de objetos no fim de uma associação, o resultado da avaliação de uma operação ou outros itens, dependendo do tipo de diagrama UML. Se x for um Set, y será aplicado a todo elemento de x ; os resultados são reunidos em um novo Set.
$c \rightarrow f()$	Aplica a operação f interna de OCL à própria Collection c (ao contrário de cada um dos objetos em c). A seguir estão listados exemplos de operações nativas.
and, or, =, <>	Operação lógica e, operação lógica ou, igual, diferente.
$p \text{ implies } q$	Verdadeiro se q for verdadeiro ou p for falso.

Exemplos de operações sobre Collection (incluindo Sets e Sequences)

$C \rightarrow \text{size}()$	O número de elementos na Collection c .
$C \rightarrow \text{isEmpty}()$	Verdadeiro se c não tiver elementos, falso caso contrário.
$c1 \rightarrow \text{includesAll}(c2)$	Verdadeiro se todo elemento de $c2$ for encontrado em $c1$.
$c1 \rightarrow \text{excludesAll}(c2)$	Verdadeiro se nenhum elemento de $c2$ for encontrado em $c1$.
$C \rightarrow \text{forAll}(\text{elem} \mid \text{boolexpr})$	Verdadeiro se boolexpr for verdadeiro quando aplicado a cada elemento de c . Quando um elemento está sendo avaliado, é acoplado à variável elem , a qual pode usar boolexpr . Isso implementa a qualificação universal, discutida anteriormente.
$C \rightarrow \text{forAll}(\text{elem1}, \text{elem2} \mid \text{boolexpr})$	O mesmo que a anterior, exceto que boolexpr é avaliada para todo possível <i>par</i> de elementos tomados de c , incluindo casos em que o par consiste no mesmo elemento.
$C \rightarrow \text{isUnique}(\text{elem} \mid \text{expr})$	Verdadeiro se expr é avaliada a um valor diferente quando aplicada a cada elemento de c .

Exemplos de operações em Sets

$s1 \rightarrow \text{intersection}(s2)$	O conjunto dos elementos encontrados em $s1$ e também em $s2$.
$s1 \rightarrow \text{union}(s2)$	O conjunto dos elementos encontrados em $s1$ ou $s2$.
$s1 \rightarrow \text{excluding}(x)$	O conjunto $s1$ com o objeto x omitido.

Exemplo de operação específica para Sequences

$\text{Seq} \rightarrow \text{first}()$	$\text{Seq} \rightarrow \text{first}()$ O objeto que é o primeiro elemento na sequência seq .

Para ilustrar o uso da OCL na especificação, reexaminamos o exemplo do tratador de blocos, introduzido no Capítulo 28. O primeiro passo é desenvolver um modelo UML (Figura A3.1). Esse diagrama de classe especifica muitas relações entre os objetos envolvidos. No entanto, são acrescentadas expressões OCL para permitir que os implementadores do sistema saibam mais precisamente o que deve permanecer verdadeiro enquanto o sistema processa.

As expressões OCL que complementam o diagrama de classes correspondem às seis partes da invariante discutida na Seção 28.6. No exemplo a seguir, a invariante é repetida em inglês e então é escrita a expressão OCL correspondente. É aconselhável inserir texto em linguagem natural junto com a lógica formal; isso ajuda a entender a lógica e auxilia os revisores a descobrir os erros, por exemplo, situações em que não há correspondência entre o inglês e a lógica.

1. Nenhum bloco será marcado como não usado e usado.

context BlockHandler inv:

$(\text{self.used} \rightarrow \text{intersection}(\text{self.free})) \rightarrow \text{isEmpty}()$

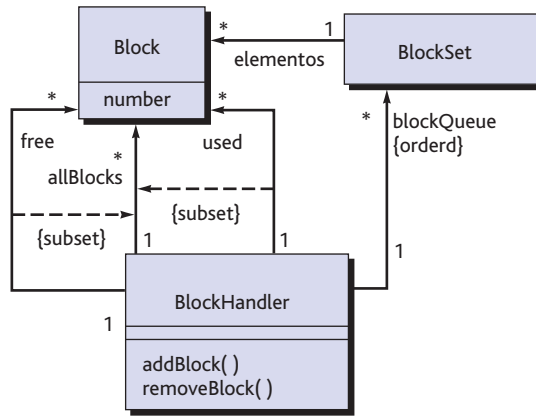


FIGURA A3.1 Diagrama de classe para um tratador de blocos.

Note que cada expressão começa com a palavra-chave **context**. Isso indica o elemento do diagrama UML que a expressão restringe. A palavra-chave **self** aqui se refere à instância de **BlockHandler**; no próximo item, como permite a OCL, vamos omitir o **self**.

2. *Todos os conjuntos de blocos mantidos na fila serão subconjuntos da coleção de blocos usados no momento.*

context BlockHandler inv:

blockQueue→forAll(aBlockSet | used→includesAll(aBlockSet))

3. *Nenhum elemento da fila terá o mesmo número de bloco.*

context BlockHandler inv:

blockQueue→forAll(blockSet1, blockSet2 |
 blockSet1 <> blockSet2 implies
 blockSet1.elements.number→excludesAll(blockSet2.elements.number))

A expressão antes de **implies** é necessária para assegurar que ignoramos pares nos quais ambos os elementos são o mesmo bloco.

4. *A coleção de blocos usados e blocos não usados será a coleção total de blocos que compõem os arquivos.*

context BlockHandler inv:

allBlocks 5 used→union(free)

5. *A coleção de blocos não usados não terá números de bloco duplicados.*

context BlockHandler inv:

free→isUnique(aBlock | aBlock.number)

6. *A coleção de blocos usados não terá números de bloco duplicados.*

context BlockHandler inv:

used→isUnique(aBlock | aBlock.number)

A OCL também pode ser utilizada para especificar pré e pós-condições de operações. Por exemplo, a notação a seguir descreve operações que removem e acrescentam conjuntos de blocos à fila. Observe que a notação **x@pre** indica o objeto **x** da maneira como ele existia *antes* da operação; isso é o oposto à notação matemática discutida anteriormente, em que é o **x** *depois* da operação que é especialmente designado (como **x'**).

```
context BlockHandler::removeBlocks()
  pre: blockQueue->size() >0
  post: used 5 used@pre-blockQueue@pre->first() and
        free 5 free@pre->union(blockQueue@pre->first()) and
        blockQueue 5 blockQueue@pre->excluding(blockQueue@pre->first)

context BlockHandler::addBlocks(aBlockSet :BlockSet)
  pre: used->includesAll(aBlockSet.elements)
  post: (blockQueue.elements = blockQueue.elements@pre
        -> append (aBlockSet.elements) and
        used 5 used@pre and free = free@pre
```

A OCL é uma linguagem de modelagem, mas tem todos os atributos de uma linguagem formal. A OCL permite a expressão de várias restrições, pré e pós-condições, proteções e outras características relacionadas aos objetos representados em vários modelos UML.

A linguagem de especificação Z

Z (que se pronuncia como “zed”) é uma linguagem de especificação amplamente usada na comunidade de métodos formais. A linguagem Z aplica conjuntos tipados, relações e funções no contexto da lógica de predicados de primeira ordem para criar *esquemas* – um meio de estruturar a especificação formal.

As especificações Z são organizadas como um conjunto de esquemas. Esquemas são usados para estruturar uma especificação formal da mesma maneira que os componentes são utilizados para estruturar um sistema.

Um esquema descreve os dados armazenados que um sistema acessa e altera. No contexto da linguagem Z, isso é chamado de “estado”. O uso do termo *estado* em Z é ligeiramente diferente do empregado no restante do livro.³ A estrutura genérica de um esquema assume a forma:

```
———nomeDoEsquema———
  declarações
  —————
  invariante
  —————
```

onde declarações identificam as variáveis que formam o estado do sistema e invariante impõe restrições sobre a maneira pela qual o estado pode evoluir. A Tabela A3.2 apresenta um resumo da notação da linguagem Z.

³ Lembre-se de que, em outros capítulos, *estado* foi empregado para identificar um modo de comportamento observável externamente para um sistema.

Tabela A3.2 Resumo da notação Z

A notação Z baseia-se na teoria de conjuntos tipados e na lógica de primeira ordem. Z fornece uma construção, chamada de esquema, para descrever o espaço de estado e operações de uma especificação. Um esquema agrupa declarações de variáveis com uma lista de predicados que restringem o valor possível de uma variável. Em Z, o esquema X é definido pela forma

X
declarações
predicados

Funções globais e constantes são definidas pela forma

declarações
predicados

A declaração fornece o tipo da função ou constante, enquanto o predicado dá seu valor. Nesta tabela há apenas um conjunto abreviado de símbolos Z.

Conjuntos:

$S : \mathbb{P}X$	S é declarada como um conjunto de X s.
$X \in S$	x é membro de S .
$x \notin S$	x não é membro de S .
$S \subseteq T$	S é um subconjunto de T : todo membro de S também está em T .
$S \cup T$	A união de S e T : contém cada membro de S ou T ou ambos.
$S \cap T$	A interseção de S e T : contém cada membro de S e T .
$S \setminus T$	A diferença de S e T : contém cada membro de S , exceto os que também estão em T .
$\emptyset$	Conjunto vazio: não contém nenhum membro.
$\{x\}$	Conjunto único (singleton): contém apenas x .
$\mathbb{N}$	O conjunto de números naturais 0, 1, 2,...
$S : \mathbb{F}X$	S é declarada como um conjunto finito de X s.
$\max(S)$	O máximo do conjunto não vazio de números S .

Funções:

$f: X \rightarrow Y$	f é declarada como uma injeção parcial de X em Y .
$\text{dom } f$	O domínio de f : o conjunto de valores x para os quais $f(x)$ é definida.
$\text{ran } f$	O intervalo de f : o conjunto de valores tomados por $f(x)$ quando x varia sobre o domínio de f .
$f \oplus \{x, y\}$	Função que concorda com f , exceto que x é mapeado em y
$\{x\} \triangleleft f$	Uma função como f , exceto que x é removido de seu domínio.

Lógica:

$P \wedge Q$	P e Q : é verdadeiro se P e Q são verdadeiros.
$P \Rightarrow Q$	P implica Q : verdadeiro se Q for verdadeiro ou P for falso.
$\theta S' = \theta S$	Nenhum componente do esquema S muda em uma operação.

O exemplo a seguir de um esquema descreve o estado de um tratador de blocos e a invariante de dados:

BlockHandler

$used, free : \mathbb{P} BLOCKS$
 $BlockQueue : seq \mathbb{P} BLOCKS$
 $used \cap free = \emptyset \wedge$
 $used \cup free = AllBlocks \wedge$
 $\forall i: \text{dom } BlockQueue \bullet BlockQueue\ i \subseteq used \wedge$
 $\forall i, j: \text{dom } BlockQueue \bullet i \neq j \Rightarrow BlockQueue\ i \cap BlockQueue\ j = \emptyset$

Conforme observamos, o esquema consiste em duas partes. A parte acima da linha central representa as variáveis do estado, enquanto a parte abaixo da linha central descreve a invariante de dados. Sempre que o esquema especificar operações que mudam o estado, ela é precedida pelo símbolo (Δ). O exemplo de um esquema, a seguir, descreve a operação que remove um elemento da fila de blocos:

RemoveBlocks

$\Delta BlockHandler$

$\#BlockQueue > 0,$
 $used' = used \setminus head\ BlockQueue \wedge$
 $free' = free \cup head\ BlockQueue \wedge$
 $BlockQueue' = tail\ BlockQueue$

A inclusão de $\Delta BlockHandler$ resulta em que todas as variáveis que formam o estado tornam-se disponíveis para o esquema *RemoveBlocks* e assegura que a invariante de dados seja mantida antes e depois da execução da operação.

A segunda operação, que acrescenta uma coleção de blocos ao fim da fila, é representada por

AddBlocks

$\Delta BlockHandler$
 $Ablocks? : BLOCKS$

$Ablocks? \subseteq used$
 $BlockQueue' = BlockQueue \smallfrown \langle Ablocks? \rangle \wedge$
 $used' = used \wedge$
 $free' = free$

Por convenção, em Z, uma variável de entrada que é lida, mas não forma parte do estado, é encerrada por um ponto de interrogação. Assim, *Ablocks?*, que age como um parâmetro de entrada, é encerrada por um ponto de interrogação.

Leituras e fontes de informação complementares

No domínio dos métodos formais, livros de Gabbar (*Modern Formal Methods and Applications*, Springer, 2010), Casey (*A Programming Approach to Formal Methods*, McGraw-Hill, 2000), Hinchey e Bowan (*Industrial Strength Formal Methods*, Springer-Verlag, 1999), Hussmann (*Formal Foundations for Software Engineering Methods*, Springer-Verlag, 1997) e Sheppard (*An Introduction to Formal Specification with Z and VDM*, McGraw-Hill, 1995) fornecem diretrizes úteis. Além disso, livros sobre linguagens específicas, como Liu (*Formal Engineering for Industrial Software Development: Using SOFL Method*, 2010), Warmer e Kleppe (*The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley, 2005; e *Object Constraint Language*, Addison-Wesley, 1998), Jacky (*The Way of Z: Practical Programming with Formal Methods*, Cambridge University Press, 1997), Harry (*Formal Methods Fact File: VDM and Z*, Wiley, 1997) e Cooper e Barden (*Z in Practice*, Prentice Hall, 1995) fornecem informações úteis para métodos formais, bem como uma variedade de linguagens de modelagem.

Há uma grande variedade de recursos de informação sobre métodos formais disponível na Internet. Uma lista atualizada das referências da Web pode ser encontrada em “recursos de engenharia de software” no site da SEPA: www.mhhe.com/pressman.

Esta página foi deixada em branco intencionalmente.

Referências

- [Abb83] Abbott, R., "Program Design by Informal English Descriptions," *CACM*, vol. 26, no. 11, November 1983, pp. 892–894.
- [Abr09] Abrial, J., "Faultless Systems: Yes We Can!" *IEEE Computer*, vol. 42, no. 9, September 2009, pp. 30–36.
- [ACM12] ACM/IEEE-CS Joint Task Force, *Software Engineering Code of Ethics and Professional Practice*, 2012, disponível em www.acm.org/serving/se/code.htm.
- [Ada93] Adams, D., *Mostly Harmless*, Macmillan, 1993.
- [AFC88] *Software Risk Abatement*, AFCS/AFLC Pamphlet 800-45, U.S. Air Force, September 30, 1988.
- [Agi03] The Agile Alliance Home Page, disponível em www.agilealliance.org/home.
- [Air99] Airlie Council, "Performance Based Management: The Program Manager's Guide Based on the 16-Point Plan and Related Metrics," Draft Report, March 8, 1999.
- [Aka04] Akao, Y., *Quality Function Deployment*, Productivity Press, 2004.
- [Ale11] Alexander, I. "Gore, Sore, or What?" *IEEE Software*, vol. 28, no. 1, January–February 2011, pp. 8–10.
- [Ale77] Alexander, C., *A Pattern Language*, Oxford University Press, 1977.
- [Ale79] Alexander, C., *The Timeless Way of Building*, Oxford University Press, 1979.
- [All08] Allen, J. H., et al., *Software Security Engineering: A Guide for Project Managers*, Addison-Wesley, 2008.
- [Amb01] Ambler, S., *The Object Primer*, 2nd ed., Cambridge University Press, 2001.
- [Amb02a] Ambler, S., "What Is Agile Modeling (AM)?" 2002, disponível em www.agilemodeling.com/index.htm.
- [Amb02b] Ambler, S., and R. Jeffries, *Agile Modeling*, Wiley, 2002.
- [Amb02c] Ambler, S., "UML Component Diagramming Guidelines," 2002, disponível em www.modelingstyle.info/.
- [Amb04] Ambler, S., "Examining the Cost of Change Curve," in *The Object Primer*, 3rd ed., Cambridge University Press, 2004.
- [Amb06] Ambler, S., "The Agile Unified Process (AUP), 2006, disponível em www.ambysoft.com/unifiedprocess/agileUP.html.
- [Amb95] Ambler, S., "Using Use-Cases," *Software Development*, July 1995, pp. 53–61.
- [Amb98] Ambler, S., *Process Patterns: Building Large-Scale Systems Using Object Technology*, Cambridge University Press/SIGS Books, 1998.
- [And05] Andreou, A., et al., "Key Issues for the Design and Development of Mobile Commerce Services and Applications," *International Journal of Mobile Communications*, vol. 3, no. 3, March 2005, pp. 303–323.
- [And06] Andrews, M., and J. Whittaker, *How to Break Web Software: Functional and Security Testing of Web Applications and Web Services*, Addison-Wesley, 2006.
- [ANS87] ANSI/ASQC A3-1987, *Quality Systems Terminology*, 1987.
- [Ant06] Anton, D., and C. Anton, *ISO 9001 Survival Guide*, 3rd ed., AEM Consulting Group, 2006.
- [AOS07] AOSD.net (Aspect-Oriented Software Development), glossary, disponível em <http://aosd.net/wiki/index.php?title=Glossary>.
- [App00] Appleton, B., "Patterns and Software: Essential Concepts and Terminology," February 2000, disponível em www.cmcrossroads.com/bradapp/docs/patterns-intro.html.

- [App13] Apple Computer, *Accessibility*, 2013, disponível em www.apple.com/accessibility/.
- [Arl02] Arlow, J., and I. Neustadt, *UML and the Unified Process*, Addison-Wesley, 2002.
- [Arn89] Arnold, R. S., "Software Restructuring," *Proceedings of the IEEE*, vol. 77, no. 4, April 1989, pp. 607–617.
- [Art97] Arthur, L. J., "Quantum Improvements in Software System Quality," *CACM*, vol. 40, no. 6, June 1997, pp. 47–52.
- [Ast04] Astels, D., *Test Driven Development: A Practical Guide*, Prentice Hall, 2004.
- [ATK12] ATKearney, "A.T. Kearney Study of Global Wealth and Spending," 2012, disponível em http://www.atkearney.com/news-media/news-releases/news-release/-/asset_publisher/00OIL7Jc67KL/content/id/387464.
- [Baa07] de Baar, B., "Project Risk Checklist," 2007, disponível em www.softwareprojects.org/project_riskmanagement_starting62.htm.
- [Baa10] Baaz, A., et al., "Appreciating Lessons Learned," *IEEE Software*, vol. 27, no. 4, July–August, 2010, pp. 72–79.
- [Bab09] Babar, M., and I. Groton, "Software Architecture Review: The State of Practice," *IEEE Computer*, vol. 42, no. 6, June 2009, pp. 1–8.
- [Bab86] Babich, W. A., *Software Configuration Management*, Addison-Wesley, 1986.
- [Bac97] Bach, J., "Good Enough Quality: Beyond the Buzzword," *IEEE Computer*, vol. 30, no. 8, August 1997, pp. 96–98.
- [Bac98] Bach, J., "The Highs and Lows of Change Control," *Computer*, vol. 31, no. 8, August 1998, pp. 113–115.
- [Bae98] Baetjer, Jr., H., *Software as Capital*, IEEE Computer Society Press, 1998, p. 85.
- [Bak72] Baker, F. T., "Chief Programmer Team Management of Production Programming," *IBM Systems Journal*, vol. 11, no. 1, 1972, pp. 56–73.
- [Ban06a] Baniassad, E., et al., "Discovering Early Aspects," *IEEE Software*, vol. 23, no. 1, January–February, 2006, pp. 61–69.
- [Bar06b] Baresi, L., E. DiNitto, and C. Ghezzi, "Toward Open-World Software: Issues and Challenges," *IEEE Computer*, vol. 39, no. 10, October 2006, pp. 36–43.
- [Bas03] Bass, L., P. Clements, and R. Kazman, *Software Architecture in Practice*, 2nd ed., Addison-Wesley, 2003.
- [Bas84] Basili, V. R., and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," *IEEE Trans. Software Engineering*, vol. SE-10, 1984, pp. 728–738.
- [Bea11] Beaird, J., *The Principles of Beautiful Web Design*, 2nd ed., Sitepoint, 2011.
- [Bec00] Beck, K., *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 1999.
- [Bec01] Beck, K., et al., "Manifesto for Agile Software Development," www.agilemanifesto.org/.
- [Bec04a] Beck, K., *Extreme Programming Explained: Embrace Change*, 2nd ed., Addison-Wesley, 2004.
- [Bec04b] Beck, K., *Test-Driven Development: By Example*, 2nd ed., Addison-Wesley, 2004.
- [Bee99] Beedle, M., et al., "SCRUM: An Extension Pattern Language for Hyperproductive Software Development," included in: *Pattern Languages of Program Design 4*, Addison-Wesley Longman, Reading MA, 1999, disponível para download em http://jeffsutherland.com/scrums/scrums_plop.pdf.
- [Beg10] Begel, A., R. DeLine, and T. Zimmermann, "Social Media for Software Engineering," *Proc. FoSER 2010*, ACM, November, 2010.
- [Bei84] Beizer, B., *Software System Testing and Quality Assurance*, Van Nostrand-Reinhold, 1984.
- [Bei90] Beizer, B., *Software Testing Techniques*, 2nd ed., Van Nostrand-Reinhold, 1990.
- [Bei95] Beizer, B., *Black-Box Testing*, Wiley, 1995.
- [Bel81] Belady, L., Foreword to *Software Design: Methods and Techniques* (L. J. Peters, author), Yourdon Press, 1981.
- [Bel95] Bellinzona R., M. G. Gugini, and B. Pernici, "Reusing Specifications in OO Applications," *IEEE Software*, March 1995, pp. 65–75.

- [Ben00] Bennatan, E., *Software Project Management: A Practitioner's Approach*, 3rd ed., McGraw-Hill, 2000.
- [Ben99] Bentley, J., *Programming Pearls*, 2nd ed., Addison-Wesley, 1999.
- [Ben02] Bennett, S., S. McRobb, and R. Farmer, *Object-Oriented Analysis and Design*, 2nd ed., McGraw-Hill, 2002.
- [Ben10] Benaroch, M., and A. Appari, "Financial Pricing of Software Development Risk Factors," *IEEE Software*, vol. 27, no. 3, September–October, 2010, pp. 65–73.
- [Ber80] Bersoff, E., V. Henderson, and S. Siegel, *Software Configuration Management*, Prentice Hall, 1980.
- [Ber93] Berard, E., *Essays on Object-Oriented Software Engineering*, vol. 1, Addison-Wesley, 1993.
- [Bes04] Bessin, J., "The Business Value of Quality," IBM developerWorks, June 15, 2004, disponível para download em <http://cm.techwell.com/articles/original/business-value-quality-and-testing>.
- [Bie94] Bieman, J. M., and L. M. Ott, "Measuring Functional Cohesion," *IEEE Trans. Software Engineering*, vol. SE-20, no. 8, August 1994, pp. 308–320.
- [Bin93] Binder, R., "Design for Reuse Is for Real," *American Programmer*, vol. 6, no. 8, August 1993, pp. 30–37.
- [Bin94a] Binder, R., "Testing Object-Oriented Systems: A Status Report," *American Programmer*, vol. 7, no. 4, April 1994, pp. 23–28.
- [Bin94b] Binder, R. V., "Object-Oriented Software Testing," *Communications of the ACM*, vol. 37, no. 9, September 1994, p. 29.
- [Bin99] Binder, R., *Testing Object-Oriented Systems: Models, Patterns, and Tools*, Addison-Wesley, 1999.
- [Bir98] Biró, M., and T. Remzső, "Business Motivations for Software Process Improvement," ERCIM News No. 32, January 1998, disponível em www.ercim.org/publication/Ercim_News/enw32/biro.html.
- [Bla09] Black, S., et al. "Formal Versus Agile: Survival of the Fittest?" *IEEE Computer*, vol. 42, no. 9, September 2009, pp. 37–45.
- [Bla10] Blair, S., et al., "Responsibility-Driven Architecture," *IEEE Software*, vol. 27, no. 3, March–April 2010, pp. 26–32.
- [Bod09] Bode, S., et al., "Software Architectural Design Meets Security Engineering," *Proceedings of 16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, 2009.
- [Boe00] Boehm, B., et al., *Software Cost Estimation in COCOMO II*, Prentice Hall, 2000.
- [Boe01a] Boehm, B., "The Spiral Model as a Tool for Evolutionary Software Acquisition," *Crosstalk*, May 2001, disponível em www.stsc.hill.af.mil/crosstalk/2001/05/boehm.html.
- [Boe01b] Boehm, B., and V. Basili, "Software Defect Reduction Top 10 List," *IEEE Computer*, vol. 34, no. 1, January 2001, pp. 135–137.
- [Boe08] Boehm, B., "Making a Difference in the Software Century," *IEEE Computer*, vol. 41, no. 3, March 2008, pp. 32–38.
- [Boe81] Boehm, B., *Software Engineering Economics*, Prentice Hall, 1981.
- [Boe88] Boehm, B., "A Spiral Model for Software Development and Enhancement," *Computer*, vol. 21, no. 5, May 1988, pp. 61–72.
- [Boe89] Boehm, B. W., *Software Risk Management*, IEEE Computer Society Press, 1989.
- [Boe96] Boehm, B., "Anchoring the Software Process," *IEEE Software*, vol. 13, no. 4, July 1996, pp. 73–82.
- [Boe98] Boehm, B., "Using the WINWIN Spiral Model: A Case Study," *Computer*, vol. 31, no. 7, July 1998, pp. 33–44.
- [Boh00] Bohl, M., and M. Rynn, *Tools for Structured Design: An Introduction to Programming Logic*, 5th ed., Prentice Hall, 2000.
- [Boh66] Bohm, C., and G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," *CACM*, vol. 9, no. 5, May 1966, pp. 366–371.
- [Boi04] Boiko, B., *Content Management Bible*, 2nd ed., Wiley, 2004.

- [Boo02] Booch, G., and A. Brown, "Collaborative Development Environments," Rational Software Corp., October 28, 2002.
- [Boo05] Booch, G., J. Rumbaugh, and I. Jacobsen, *The Unified Modeling Language User Guide*. 2nd ed., Addison-Wesley, 2005.
- [Boo06] Bootstrap-institute.com, 2006, www.cse.dcu.ie/espinode/directory/directory.html.
- [Boo08] Booch, G., *Handbook of Software Architecture*, 2008, disponível em www.booch.com/architecture/systems.jsp.
- [Boo11a] Booch, G., "Dominant Design," *IEEE Software*, vol. 28, no. 2, January–February 2011, pp. 8–9.
- [Boo11b] Booch, G., "Draw Me a Picture," *IEEE Software*, vol. 28, no. 1, January–February 2011, pp. 6–7.
- [Boo94] Booch, G., *Object-Oriented Analysis and Design*, 2nd ed., Benjamin Cummings, 1994.
- [Bor01] Borchers, J., *A Pattern Approach to Interaction Design*, Wiley, 2001.
- [Bos00] Bosch, J., *Design & Use of Software Architectures*, Addison-Wesley, 2000.
- [Bos11] Bose, B., et al., "Morphing Smartphones into Automotive Application Platforms," *IEEE Computer*, vol. 44, no. 5, May 2011, pp. 28–29.
- [Bra94] Bradac, M., D. Perry, and L. Votta, "Prototyping a Process Monitoring Experiment," *IEEE Trans. Software Engineering*, vol. 20, no. 10, October 1994, pp. 774–784.
- [Bre02] Breen, P., "Exposing the Fallacy of 'Good Enough' Software," informit.com, February 1, 2002, disponível em www.informit.com/articles/article.asp?p=25141&rl=1.
- [Bre03] Breu, R., et al., *Key Issues of a Formally Based Process Model for Security Engineering*, Proceedings of the 16th International Conference on Software & Systems Engineering and their Applications, 2003.
- [Bre10] Breu, R., *Ten Principles for Living Models—A Manifesto of Change-Driven Software Engineering*, International Conference on Complex, Intelligent and Software Intensive Systems (CISIS), Krakow, Poland, February 2010, pp. 1–8.
- [Bro01] Brown, B., *Oracle9i Web Development*, 2nd ed., McGraw-Hill, 2001.
- [Bro03] Brooks, F., "Three Great Challenges for Half-Century-Old Computer Science," *JACM*, vol. 50, no. 1, January 2003, pp. 25–26.
- [Bro06] Broy, M., "The 'Grand Challenge' in Informatics: Engineering Software Intensive Systems," *IEEE Computer*, vol. 39, no. 10, October 2006, pp. 72–80.
- [Bro10a] Brown, N., R. Nord, and I. Ozkaya, "Enabling Agility through Architecture," *Crosstalk*, November–December 2010, disponível em www.crosstalkonline.org/storage/issue-archives/.../201011-Brown.pdf.
- [Bro10b] Broy, M., and R. Reussner, "Software Architecture Review: The State of Practice," *IEEE Computer*, vol. 43, no. 10, June 2010, pp. 88–91.
- [Bro11] Brown, M., "The Best Tools for Mobile App Testing," disponível em <http://www.mobile-app-testing.com/the-best-tools-for-mobile-app-testing/2011/08/>.
- [Bro12] Brown, A., *The Architecture of Open Source Applications*, lulu.com, 2012.
- [Bro95] Brooks, F., *The Mythical Man-Month*, Silver Anniversary edition, Addison-Wesley, 1995.
- [Bro96] Brown, A., and K. Wallnau, "Engineering of Component Based Systems," *Component-Based Software Engineering*, IEEE Computer Society Press, 1996, pp. 7–15.
- [Buc99] Bucanac, C., "The V-Model," University of Karlskrona/Ronneby, January 1999, disponível para download em www.bucanac.com/documents/The_V-Model.pdf.
- [Bud96] Budd, T., *An Introduction to Object-Oriented Programming*, 2nd ed., Addison-Wesley, 1996.
- [Bus07] Buschmann, F., et al., *Pattern-Oriented Software Architecture, A System of Pattern*, Wiley, 2007.
- [Bus10] Buschmann, F., "Learning from Failure, Part 2: Featuritis, Performatitis, and Other Diseases," *IEEE Software*, vol. 27, no. 1, January–February 2010, pp. 10–11.
- [Bus10a] Buschmann, F., "On Architecture Styles and Paradigms," *IEEE Software*, vol. 27, no. 5, September–October 2010, pp. 92–94.

- [Bus10b] Buschmann, F., and K. Henley, "Five Considerations for Software Architecture, Part 1," *IEEE Software*, vol. 27, no. 3, May–June 2010, pp. 63–65.
- [Bus10c] Buschmann, F., and K. Henley, "Five Considerations for Software Architecture, Part 2," *IEEE Software*, vol. 27, no. 4, July–August 2010, pp. 12–14.
- [Bus96] Buschmann, F., et al., *Pattern-Oriented Software Architecture*, Wiley, 1996.
- [Cac02] Cachero, C., et al., "Conceptual Navigation Analysis: a Device and Platform Independent Navigation Specification," *Proceedings of the Second International Workshop on Web-Oriented Technology*, June 2002, disponível em www.dsic.upv.es/~west/iwwost02/papers/cachero.pdf.
- [Car08] Carrasco, M., "7 Key Attributes of High Performance Software Development Teams," June 30, 2008, disponível em <http://www.realsoftwaredevelopment.com/7-keyattributes-of-high-performance-software-development-teams/>.
- [Car90] Card, D., and R. Glass, *Measuring Software Design Quality*, Prentice Hall, 1990.
- [Cas06] Casey, V., and I. Richardson, "Uncovering the Reality within Virtual Software Teams," *Proc. GSD'06*, ACM, May 23, 2006.
- [Cas89] Cashman, M., "Object Oriented Domain Analysis," *ACM Software Engineering Notes*, vol. 14, no. 6, October 1989, p. 67.
- [Cav78] Cavano, J., and J. McCall, "A Framework for the Measurement of Software Quality," *Proc. ACM Software Quality Assurance Workshop*, November 1978, pp. 133–139.
- [CCS02] CS3 Consulting Services, 2002, disponível em www.cs3inc.com/DSDM.htm.
- [Cec06] Cechich, A., et al., "Trends on COTS Component Identification," *Proc. Fifth Intl. Conf. on COTS-Based Software Systems*, IEEE, 2006.
- [Cha89] Charette, R., *Engineering Risk Analysis and Management*, McGraw-Hill/Intertext, 1989.
- [Cha92] Charette, R., "Building Bridges over Intelligent Rivers," *American Programmer*, vol. 5, no. 7, September 1992, pp. 2–9.
- [Cha93] de Champeaux, D., D. Lea, and P. Faure, *Object-Oriented System Development*, Addison-Wesley, 1993.
- [Chi94] Chidamber, S., and C. Kemerer, "A Metrics Suite for Object-Oriented Design," *IEEE Trans. Software Engineering*, vol. SE-20, no. 6, June 1994, pp. 476–493.
- [Cho89] Choi, S., and W. Scacchi, "Assuring the Correctness of a Configured Software Description," *Proceedings of the Second International Workshop on Software Configuration Management*, ACM, Princeton, NJ, October 1989, pp. 66–75.
- [Chu09] Chung, L., and J. Leite, "On Non-Functional Requirements in Software Engineering," published in *Conceptual Modeling: Foundations and Applications*, Springer-Verlag, 2009.
- [Chu95] Churcher, N., and M. Shepperd, "Towards a Conceptual Framework for Object-Oriented Metrics," *ACM Software Engineering Notes*, vol. 20, no. 2, April 1995, pp. 69–76.
- [Cia09] Ciafrani, C., et al., *ISO 9000:2008 Explained*, 3rd ed., ASQ Quality Press, 2009.
- [Cig07] Cigital, Inc., "Case Study: Finding Defects Earlier Yields Enormous Savings," 2007, disponível em www.cigital.com/solutions/roi-cs2.php.
- [Cla05] Clark, S., and E. Baniasaad, *Aspect-Oriented Analysis and Design*, Addison-Wesley, 2005.
- [Cle03] Clements, P., R. Kazman, and M. Klein, *Evaluating Software Architectures: Methods and Case Studies*, Addison-Wesley, 2003.
- [Cle06] Clemmons, R., "Project Estimation with Use Case Points," *CrossTalk*, February 2006, pp. 18–222, disponível em www.stsc.hill.af.mil/crosstalk/2006/02/0602Clemmons.pdf.
- [Cle10] Clements, P., and L. Bass, "The Business Goals Viewpoint," *IEEE Software*, vol. 27, no. 6, November–December 2010, pp. 38–45.
- [CMM07] *Capability Maturity Model Integration (CMMI)*, Software Engineering Institute, 2007, disponível em www.sei.cmu.edu/cmmi/.
- [CMM08] *People Capability Maturity Model Integration (People CMM)*, Software Engineering Institute, 2008, disponível em www.sei.cmu.edu/cmm-p/.
- [Coa91] Coad, P., and E. Yourdon, *Object-Oriented Analysis*, 2nd ed., Prentice Hall, 1991.

- [Coa99] Coad, P., E. Lefebvre, and J. DeLuca, *Java Modeling in Color with UML*, Prentice Hall, 1999.
- [Coc01a] Cockburn, A., and J. Highsmith, "Agile Software Development: The People Factor," *IEEE Computer*, vol. 34, no. 11, November 2001, pp. 131–133.
- [Coc01b] Cockburn, A., *Writing Effective Use-Cases*, Addison-Wesley, 2001.
- [Coc02] Cockburn, A., *Agile Software Development*, Addison-Wesley, 2002.
- [Coc04] Cockburn, A., "What the Agile Toolbox Contains," *CrossTalk*, November 2004, disponível em www.stsc.hill.af.mil/crosstalk/2004/11/0411Cockburn.html.
- [Coc05] Cockburn, A., *Crystal Clear*, Addison-Wesley, 2005.
- [Coc11] Cochran, C., *ISO 9001 in Plain English*, Paton Professional, 2011.
- [Coh05] Cohn, M., "Estimating with Use Case Points," *Methods & Tools*, Fall, 2005, disponível em <http://www.mountangoatsoftware.com/articles/estimating-with-use-case-points>.
- [Col09] Collaris, R-A., and E. Dekker, "Software Cost Estimation Using Use Case Points," IBM Developer Works, March 15, 2009, disponível em http://www.ibm.com/developerworks/rational/library/edge/09/mar09/collaris_dekker/.
- [Con93] Constantine, L., "Work Organization: Paradigms for Project Management and Organization," *CACM*, vol. 36, no. 10, October 1993, pp. 34–43.
- [Con95] Constantine, L., "What DO Users Want? Engineering Usability in Software," *Windows Tech Journal*, December 1995, disponível em www.forUse.com.
- [Con96] Conradi, R., "Software Process Improvement: Why We Need SPIQ," NTNU, October 1996, disponível para download em www.idi.ntnu.no/grupper/su/publ/pdf/nik96-spiq.pdf.
- [Con99] Constantine, L., and L. Lockwood, "Learning the Laws of Usability," *Software Development*, vol. 7, no. 10, October, 1999.
- [Con02] Conradi, R., and A. Fuggetta, "Improving Software Process Improvement," *IEEE Software*, July–August 2002, pp. 2–9, disponível em <http://citeseer.ist.psu.edu/conradi02improving.html>.
- [Cop05] Coplien, J., "Software Patterns," 2005, disponível em <http://hillside.net/patterns/definition.html>.
- [Cou00] Coulouris, G., J. Dollimore, and T. Kindberg, *Distributed Systems: Concepts and Design*, 3rd ed., Addison-Wesley, 2000.
- [Cri92] Christel, M., and K. Kang, "Issues in Requirements Elicitation," Software Engineering Institute, CMU/SEI-92-TR-12 7, September 1992.
- [Crn11] Crnkovic, I., et al., "A Classification Framework for Software Component Models," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, September–October 2011, pp. 593–615.
- [Cro79] Crosby, P., *Quality Is Free*, McGraw-Hill, 1979.
- [Cur01] Curtis, B., W. Heffley, and S. Miller, *People Capability Maturity Model*, Addison-Wesley, 2001.
- [Cur86] Curritt, P., M. Dyer, and H. Mills, "Certifying the Reliability of Software," *IEEE Trans, Software Engineering*, vol. SE-12, no. 1, January 1994.
- [Cur90] Curtis, B., and D. Walz, "The Psychology of Programming in the Large: Team and Organizational Behavior," *Psychology of Programming*, Academic Press, 1990.
- [CVS07] Concurrent Versions System, Ximbiot, 2007, disponível em http://ximbiot.com/cvs/wiki/index.php?title=Main_Page.
- [CVS12] Open CVS, 2012, disponível em <http://web.archive.org/web/20041220041434/>, <http://www.opencvs.org/>.
- [IDAC03] "An Overview of Model-Based Testing for Software," Data and Analysis Center for Software, CR/TA 12, June 2003, disponível em www.goldpractices.com/download/practice/pdf/Model_Based_Testing.pdf.
- [IDah72] Dahl, O., E. Dijkstra, and C. Hoare, *Structured Programming*, Academic Press, 1972.
- [IDan09] Dang, A., "How Correct is Security by Correctness?" Tom's Hardware, July 16, 2009, disponível em <http://www.tomshardware.com/reviews/joanna-rutkowska-rootkit,2356-7.html>.

- [Dar01] Dart, S., *Spectrum of Functionality in Configuration Management Systems*, Software Engineering Institute, 2001, disponível em www.sei.cmu.edu/legacy/scm/tech_rep/TR11_90/TOC_TR11_90.html.
- [Dar91] Dart, S., "Concepts in Configuration Management Systems," *Proc. Third International Workshop on Software Configuration Management*, ACM SIGSOFT, 1991, disponível em www.sei.cmu.edu/legacy/scm/abstracts/abscm_concepts.html.
- [Dar99] Dart, S., "Change Management: Containing the Web Crisis," *Proc. Software Configuration Management Symposium*, Toulouse, France, 1999, disponível em www.perforce.com/perforce/conf99/dart.html.
- [Dav90] Davenport, T. H., and J. E. Young, "The New Industrial Engineering: Information Technology and Business Process Redesign," *Sloan Management Review*, Summer 1990, pp. 11–27.
- [Dav93] Davis, A., et al., "Identifying and Measuring Quality in a Software Requirements Specification," *Proceedings of the First International Software Metrics Symposium*, IEEE, Baltimore, MD, May 1993, pp. 141–152.
- [Dav95a] Davis, M., "Process and Product: Dichotomy or Duality," *Software Engineering Notes*, ACM Press, vol. 20, no. 2, April 1995, pp. 17–18.
- [Dav95b] Davis, A., *201 Principles of Software Development*, McGraw-Hill, 1995.
- [Day99] Dayani-Fard, H., et al., "Legacy Software Systems: Issues, Progress, and Challenges," IBM Technical Report: TR-74.165-k, April 1999, disponível em www.cas.ibm.com/toronto/publications/TR-74.165/k/legacy.html.
- [DeM02] DeMarco, T., and B. Boehm, "The Agile Methods, Fray," *IEEE Computer*, vol. 35, no. 6, June 2002, pp. 90–92.
- [DeM79] DeMarco, T., *Structured Analysis and System Specification*, Prentice Hall, 1979.
- [DeM95] DeMarco, T., *Why Does Software Cost So Much?* Dorset House, 1995.
- [DeM98] DeMarco, T., and T. Lister, *Peopleware*, 2nd ed., Dorset House, 1998.
- [Dem86] Deming, W., *Out of the Crisis*, MIT Press, 1986.
- [Den73] Dennis, J., "Modularity," in *Advanced Course on Software Engineering* (F. L. Bauer, ed.), Springer-Verlag, 1973, pp. 128–182.
- [Des08] de Sá, M., and L. Carriço, "Lessons from Early Stages Design of Mobile Applications," *Proceedings of 10th International Conference on Human Computer Human with Mobile Services and Devices*, September 2008, pp. 127–136.
- [deS09] de Sousa, C., et al., "Cooperative and Human Aspects of Software Engineering," *IEEE Software*, vol. 26, no. 6, pp. 17–19.
- [Dev00] Devanbu, P., and S. Stubblebine, "Software Engineering for Security: A Roadmap," *Proc. ICSE*, IEEE, 2000, disponível em <http://www0.cs.ucl.ac.uk/staff/A.Finkelstein/fose/finaldevanbu.pdf>.
- [Dev01] Devedzik, V., "Software Patterns," in *Handbook of Software Engineering and Knowledge Engineering*, World Scientific Publishing Co., 2001.
- [Dha95] Dhama, H., "Quantitative Metrics for Cohesion and Coupling in Software," *Journal of Systems and Software*, vol. 29, no. 4, April 1995.
- [Dij65] Dijkstra, E., "Programming Considered as a Human Activity," in *Proc. 1965 IFIP Congress*, North-Holland Publishing Co., 1965.
- [Dij72] Dijkstra, E., "The Humble Programmer," 1972 ACM Turing Award Lecture, CACM, vol. 15, no. 10, October 1972, pp. 859–866.
- [Dij76a] Dijkstra, E., "Structured Programming," in *Software Engineering, Concepts and Techniques* (J. Buxton et al., ed.), Van Nostrand-Reinhold, 1976.
- [Dij76b] Dijkstra, E., *A Discipline of Programming*, Prentice Hall, 1976.
- [Dij82] Dijkstra, E., "On the Role of Scientific Thought," *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, 1982.
- [Dix99] Dix, A., "Design of User Interfaces for the Web," *Proc. User Interfaces to Data Systems Conference*, September 1999, disponível em www.comp.lancs.ac.uk/computing/users/dixa/topics/webarch/.

- [Don99] Donahue, G., S. Weinschenck, and J. Nowicki, "Usability Is Good Business," Compuware Corp., July 1999, disponível em www.compuware.com.
- [Dre99] Dreilinger, S., "CVS Version Control for Web Site Projects," 1999, disponível em www.durak.org/cvswebsites/howto-cvs/howto-cvs.html.
- [Dru75] Drucker, P., *Management*, W. H. Heinemann, 1975.
- [DSi08] D'Silva, V., et al., "Interview: Software Security in the Real World," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7, July 2008, pp. 1165–1178.
- [Duc01] Ducatel, K., et al., *Scenarios for Ambient Intelligence in 2010*, ISTAG-European Commission, 2001, disponível para download em <ftp://ftp.cordis.europa.eu/pub/ist/docs/istagscenarios2010.pdf>.
- [Dun01] Dunaway, D., and S. Masters, *CMM-Based Appraisal for Internal Process Improvement (CBA IPI Version 1,2 Method Description)*, Software Engineering Institute, 2001, disponível em www.sei.cmu.edu/publications/documents/01.reports/01tr033.html.
- [Dun02] Dunn, W., *Practical Design of Safety-Critical Computer Systems*, William Dunn, 2002.
- [Dun82] Dunn, R., and R. Ullman, *Quality Assurance for Computer Software*, McGraw-Hill, 1982.
- [Duy02] VanDuyne, D., J. Landay, and J. Hong, *The Design of Sites*, Addison-Wesley, 2002.
- [Dye92] Dyer, M., *The Cleanroom Approach to Quality Software Development*, Wiley, 1992.
- [Edg95] Edgemon, J., "Right Stuff: How to Recognize It When Selecting a Project Manager," *Application Development Trends*, vol. 2, no. 5, May 1995, pp. 37–42.
- [Eis01] Eisenstein, J., et al., "Applying Model-Based Techniques to the Development of UIs for Mobile Computers," *Proceedings of Intelligent User Interfaces*, January 2001.
- [Eji91] Ejiogu, L., *Software Engineering with Formal Metrics*, QED Publishing, 1991.
- [Elr01] Elrad, T., R. Filman, and A. Bader (ed.), "Aspect Oriented Programming," *Comm. ACM*, vol. 44, no. 10, October 2001, special issue.
- [Erd09] Ergomus, H., "The Seven Traits of Superprofessionals," *IEEE Software*, vol. 26, no. 4, July–August, 2009, pp. 4–6.
- [Erd10] Ergogmus, H., "Déjà vu: The Life of Software Engineering Ideas," *IEEE Software*, vol. 27, no. 1, January–February 2010, pp. 2–3.
- [Eri05] Ericson, C., *Hazard Analysis Techniques for System Safety*, Wiley-Interscience, 2005.
- [Eri08] Erickson, T., *The Interaction Design Patterns Page*, May 2008, disponível em www.vision.com/~snowfall/InteractionPatterns.html.
- [Eva04] Evans, E., *Domain Driven Design*, Addison-Wesley, 2004.
- [Eve09] Everett, G., and B. Meyer, "Point/Counterpoint," *IEEE Software*, vol. 26, no. 4, July–August 2009, pp. 62–65.
- [Fag86] Fagan, M., "Advances in Software Inspections," *IEEE Trans. Software Engineering*, vol. 12, no. 6, July 1986.
- [Fal10] Falessi, D., et al., "Peaceful Coexistence: Agile Developer Perspectives on Software Architecture," *IEEE Software*, vol. 27, no. 3, March–April 2010, pp. 23–25.
- [Fel07] Feller, J., et al. (eds.), *Perspectives on Free and Open Source Software*, The MIT Press, 2007.
- [Fel89] Felican, L., and G. Zalateu, "Validating Halstead's Theory for Pascal Programs," *IEEE Trans. Software Engineering*, vol. SE-15, no. 2, December 1989, pp. 1630–1632.
- [Fen91] Fenton, N., *Software Metrics*, Chapman and Hall, 1991.
- [Fen94] Fenton, N., "Software Measurement: A Necessary Scientific Basis," *IEEE Trans. Software Engineering*, vol. SE-20, no. 3, March 1994, pp. 199–206.
- [Fer00] Fernandez, E. B., and X. Yuan, "Semantic Analysis Patterns," *Proceedings of the 19th International Conference on Conceptual Modeling, ER2000*, Lecture Notes in Computer Science 1920, Springer, 2000, pp. 183–195. Também disponível em www.cse.fau.edu/~ed/SAPpaper2.pdf.
- [Fer97] Ferguson, P., et al., "Results of Applying the Personal Software Process," *IEEE Computer*, vol. 30, no. 5, May 1997, pp. 24–31.

- [Fer98] Ferdinandi, P. L., "Facilitating Communication," *IEEE Software*, September 1998, pp. 92–96.
- [Fil05] Filman, R., et al., *Aspect-oriented Software Development*, Addison-Wesley, 2005.
- [Fir12] Firesmith, D., *Security and Safety Requirements for Software-Intensive Systems*, Auerbach, 2012.
- [Fir93] Firesmith, D. G., *Object-Oriented Requirements Analysis and Logical Design*, Wiley, 1993.
- [Fis11] Fisher, R., et al., *Getting to Yes: Negotiating without Giving In*, Penguin Books, 2011.
- [Fle98] Fleming, Q., and J. Koppelman, "Earned Value Project Management," *CrossTalk*, vol. 11, no. 7, July 1998, p. 19.
- [Fok10] Fokaefs, M., et al., WikiDev 2.0: Facilitating Software Development Teams, *Proceedings of the 14th European Conference on Software Maintenance and Reengineering*, March 15–18, 2010, pp. 276–277.
- [Fos06] Foster, E., "Quality Culprits," InfoWorld Grip Line Weblog, May 2, 2006, disponível em http://weblog.infoworld.com/gripline/2006/05/02_a395.html.
- [Fow00] Fowler, M., et al., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 2000.
- [Fow01] Fowler, M., and J. Highsmith, "The Agile Manifesto," *Software Development Magazine*, August 2001, disponível em www.sdmagazine.com/documents/s=844/sdm0108a/0108a.htm.
- [Fow02] Fowler, M., "The New Methodology," June 2002, disponível em www.martinfowler.com/articles/newMethodology.html#N8B.
- [Fow03] Fowler, M., et al., *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003.
- [Fow04] Fowler, M., *UML Distilled*, 3rd ed., Addison-Wesley, 2004.
- [Fow97] Fowler, M., *Analysis Patterns: Reusable Object Models*, Addison-Wesley, 1997.
- [Fra93] Frankl, P., and S. Weiss, "An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow," *IEEE Trans. Software Engineering*, vol. SE-19, no. 8, August 1993, pp. 770–787.
- [Fre80] Freeman, P., "The Context of Design," in *Software Design Techniques*, 3rd ed. (P. Freeman and A. Wasserman, eds.), IEEE Computer Society Press, 1980, pp. 2–4.
- [Fre90] Freedman, D., and G. Weinberg, *Handbook of Walkthroughs, Inspections and Technical Reviews*, 3rd ed., Dorset House, 1990.
- [Fri10] Fricker, S., "Handshaking with Implementation Proposals: Negotiating Requirements Understanding," *IEEE Software*, vol. 27, no. 2, March–April 2010, pp. 72–80.
- [Gag04] Gage, D., and J. McCormick, "We Did Nothing Wrong," *Baseline Magazine*, March 4, 2004, disponível em www.baselinemag.com/article2/0,1397,1544403,00.asp.
- [Gai95] Gaines, B., "Modeling and Forecasting the Information Sciences," Technical Report, University of Calgary, Calgary, Alberta, September 1995.
- [Gam95] Gamma, E., et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
- [Gar08] GartnerGroup, "Understanding Hype Cycles," 2008, disponível em www.gartner.com/pages/story.php.id.8795.s.8.jsp.
- [Gar09] Gardner, D., "Can Software Development Aspire to the Cloud?" *ZDNet.com*, April 28, 2009, disponível em <http://www.zdnet.com/blog/gardner/can-software-developmentaspire-to-the-cloud/2915>.
- [Gar09a] Garlan, D., et al., "Architectural Mismatch: Why Reuse is Still So Hard," *IEEE Software*, vol. 26, no. 4, July–August 2009, pp. 66–69.
- [Gar10] Garcia-Crespo, A. et al., "A Qualitative Study of Hard Decision Making in Managing Global Software Development Teams," *Journal of Management Information Systems*, vol. 27, no. 3, June 2010, pp. 247–252.
- [Gar12] GartnerGroup, "Gartner Says Worldwide Media Tablets Sales to Reach 119 Million Units in 2012," April 2012, disponível em <http://www.gartner.com/it/page.jsp?id=1980115>.

- [Gar84] Garvin, D., "What Does 'Product Quality' Really Mean?" *Sloan Management Review*, Fall 1984, pp. 25–45.
- [Gar87] Garvin D., "Competing on the Eight Dimensions of Quality," *Harvard Business Review*, November 1987, pp. 101–109. Um resumo está disponível em www.acm.org/crossroads/xrds6-4/software.html.
- [Gar95] Garlan, D., and M. Shaw, "An Introduction to Software Architecture," *Advances in Software Engineering and Knowledge Engineering*, vol. I (V. Ambriola and G. Tortora, eds.), World Scientific Publishing Company, 1995.
- [Gau89] Gause, D., and G. Weinberg, *Exploring Requirements: Quality Before Design*, Dorset House, 1989.
- [Gav11] Gavalas, D., and D. Economou, "Development Platforms for Mobile Applications," *IEEE Software*, vol. 28, no. 1, January–February, 2011, pp. 77–86.
- [Gey01] Geyer-Schulz, A., and M. Hahsler, "Software Engineering with Analysis Patterns," Technical Report 01/2001, Institut für Informationsverarbeitung und -wirtschaft, Wirtschaftsuniversität Wien, November 2001, disponível para download em <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.70.8415&rep=rep1&type=pdf>.
- [Gho01] Ghoshm, A., and Swaminatha, T. "Software Security and Privacy Risks in Mobile E-Commerce," *Communication of the ACM*, vol. 44, no. 2, February 2001, pp. 51–57.
- [Gil06] Gillis, D., "Pattern-Based Design," *tehan + lax blog*, September 14, 2006, disponível em www.teehanlax.com/blog/?p=96.
- [Gil88] Gilb, T., *Principles of Software Project Management*, Addison-Wesley, 1988.
- [Gil95] Gilb, T., "What We Fail to Do in Our Current Testing Culture," *Testing Techniques Newsletter* (online edition, ttn@soft.com), Software Research, January 1995.
- [Gla02] Gladwell, M., *The Tipping Point*, Back Bay Books, 2002.
- [Gla98] Glass, R., "Defining Quality Intuitively," *IEEE Software*, May 1998, pp. 103–104, 107.
- [Gli07] Glinz, M., and R. Wieringa, "Stakeholders in Requirements Engineering," *IEEE Software*, vol. 24, no. 2, March–April 2007, pp. 18–20.
- [Gli09] Glinz, M., "A Risk-Base, Value-Oriented Approach to Quality Requirements," *IEEE Software*, vol. 26, no. 5, March–April 2009, pp. 34–41.
- [Glu94] Gluch, D., "A Construct for Describing Software Development Risks," CMU/SEI-94-TR-14, Software Engineering Institute, 1994.
- [Gna99] Gnaho, C., and F. Larcher, "A User-Centered Methodology for Complex and Customizable Web Engineering," *Proceedings of the First ICSE Workshop on Web Engineering*, ACM, Los Angeles, May 1999.
- [Gon04] Gonzales, R., "Requirements Engineering," Sandia National Laboratories, uma apresentação de slides, disponível em www.incose.org/enchantment/docs/04AprRequirementsEngineering.pdf.
- [Gor06] Gorton, I., *Essential Software Architecture*, Springer, 2006.
- [Got11] Gotel, O., and S. Morris, "Requirements Tracery," *IEEE Software*, vol. 28, no. 5, September–October 2011, pp. 92–94.
- [Gra03] Gradecki, J., and N. Lesiecki, *Mastering AspectJ: Aspect-Oriented Programming in Java*, Wiley, 2003.
- [Gra87] Grady, R. B., and D. L. Caswell, *Software Metrics: Establishing a Company-Wide Program*, Prentice Hall, 1987.
- [Gra92] Grady, R. G., *Practical Software Metrics for Project Management and Process Improvement*, Prentice Hall, 1992.
- [Gru00] Gruia-Catalin, R., et al., "Software Engineering for Mobility: A Roadmap," *Proceedings of the 22nd International Conference on the Future of Software Engineering*, 2000.
- [Gru02] Grundy, J., "Aspect-Oriented Component Engineering," 2002, www.cs.auckland.ac.nz/~john-g/aspects.html.
- [Gub09] Gube, J., "40+ Helpful Resources on User Interface Design Patterns," June 15, 2009, <http://www.smashingmagazine.com/2009/06/15/40-helpful-resources-on-userinterface-design-patterns/>.

- [Gus89] Gustavsson, A., "Maintaining the Evolution of Software Objects in an Integrated Environment," *Proceedings of the Second International Workshop on Software Configuration Management*, ACM, Princeton, NJ, October 1989, pp. 114–117.
- [Gut93] Guttag, J., and J. Horning, *Larch: Languages and Tools for Formal Specification*, Springer-Verlag, 1993.
- [Hac98] Hackos, J., and J. Redish, *User and Task Analysis for Interface Design*, Wiley, 1998.
- [Hai02] Hailpern, B., and P. Santhanam, "Software Debugging, Testing and Verification," *IBM Systems Journal*, vol. 41, no. 1, 2002, disponível em www.research.ibm.com/journal/sj/411/hailpern.html.
- [Hal77] Halstead, M., *Elements of Software Science*, North-Holland, 1977.
- [Hal90] Hall, A., "Seven Myths of Formal Methods," *IEEE Software*, September 1990, pp. 11–20.
- [Hal98] Hall, E. M., *Managing Risk: Methods for Software Systems Development*, Addison-Wesley, 1998.
- [Ham90] Hammer, M., "Reengineer Work: Don't Automate, Obliterate," *Harvard Business Review*, July–August 1990, pp. 104–112.
- [Han95] Hanna, M., "Farewell to Waterfalls," *Software Magazine*, May 1995, pp. 38–46.
- [Har11] Harris, N., and P. Avgeriou, "Pattern-Based Architecture Reviews," *IEEE Software*, vol. 28, no. 6, November–December 2011, pp. 66–71.
- [Har12] Hardy, T., *Software and System Safety*, Authorhouse, 2012.
- [Har98b] Harrison, R., S. Counsell, and R. Nithi, "An Evaluation of the MOOD Set of Object-Oriented Software Metrics," *IEEE Trans. Software Engineering*, vol. SE-24, no. 6, June 1998, pp. 491–496.
- [Hen10] Hendler, J., "Web 3.0: The Dawn of Semantic Search," *IEEE Computer*, January, 2010, p. 77.
- [Her00] Herrmann, D., *Software Safety and Reliability*, Wiley-IEEE Computer Society Press, 2000.
- [Het84] Hetzel, W., *The Complete Guide to Software Testing*, QED Information Sciences, 1984.
- [Het93] Hetzel, W., *Making Software Measurement Work*, QED Publishing, 1993.
- [Hev93] Hevner, A., and H. Mills, "Box Structure Methods for System Development with Objects," *IBM Systems Journal*, vol. 31, no. 2, February 1993, pp. 232–251.
- [IFP01] *Function Point Counting Practices Manual*, 2001, disponível para download em: [perun.pmf.uns.ac.rs/old/repository/.../se/functionpoints.pdf](http://pmf.uns.ac.rs/old/repository/.../se/functionpoints.pdf).
- [IFP12] *Function Point Bibliography/Reference Library*, International Function Point Users Group, 2012, disponível em http://www.ifpug.org/?page_id=237.
- [Isk08] Iskold, A., "Top Ten Concepts that Every Software Engineer Should Know," Read-Write, July 2008, disponível em http://readwrite.com/2008/07/22/top_10_concepts_that_every_software_engineer_should_know.
- [ISO00] *ISO 9001: 2000 Document Set*, International Organization for Standards, 2000, disponível em www.iso.ch/iso/en/iso9000-14000/iso9000/iso9000index.html.
- [ISO02] *Z Formal Specification Notation—Syntax, Type System and Semantics*, ISO/IEC 13568:2002, Intl. Standards Organization, 2002.
- [ISO08] ISO SPICE, 2008, disponível em www.isospice.com/categories/SPICE-Project/.
- [Ivo01] Ivory, M., R. Sinha, and M. Hearst, "Empirically Validated Web Page Design Metrics," *ACM SIGCHI'01*, March 31–April 4, 2001, disponível em <http://webtango.berkeley.edu/papers/chi2001/>.
- [Jac02a] Jacobson, I., "A Resounding 'Yes' to Agile Processes—But Also More," *Cutter IT Journal*, vol. 15, no. 1, January 2002, pp. 18–24.
- [Jac02b] Jacyntho, D., D. Schwabe, and G. Rossi, "An Architecture for Structuring Complex Web Applications," 2002, disponível em www2002.org/CDROM/alternate/478/.
- [Jac04] Jacobson, I., and P. Ng, *Aspect-Oriented Software Development*, Addison-Wesley, 2004.
- [Jac75] Jackson, M. A., *Principles of Program Design*, Academic Press, 1975.
- [Jac92] Jacobson, I., *Object-Oriented Software Engineering*, Addison-Wesley, 1992.
- [Jac98] Jackman, M., "Homeopathic Remedies for Team Toxicity," *IEEE Software*, July 1998, pp. 43–45.

- [Jac99] Jacobson, I., G. Booch, and J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, 1999.
- [Jal04] Jalote, P., et al., "Timeboxing: A Process Model for Iterative Software Development," *Journal of Systems and Software*, vol. 70, issue 2, 2004, pp. 117–127. Disponível em www.cse.iitk.ac.in/users/jalote/papers/Timeboxing.pdf.
- [Jec06] Jech, T., *Set Theory*, 3rd ed., Springer, 2006.
- [Jon04] Jones, C., "Software Project Management Practices: Failure Versus Success," *Cross-Talk*, October 2004. Disponível em www.stsc.hill.af.mil/crossTalk/2004/10/0410Jones.html.
- [Jon86] Jones, C., *Programming Productivity*, McGraw-Hill, 1986.
- [Jon91] Jones, C., *Systematic Software Development Using VDM*, 2nd ed., Prentice Hall, 1991.
- [Jon96] Jones, C., "How Software Estimation Tools Work," *American Programmer*, vol. 9, no. 7, July 1996, pp. 19–27.
- [Jon98] Jones, C., *Estimating Software Costs*, McGraw-Hill, 1998.
- [Joy00] Joy, B., "The Future Doesn't Need Us," *Wired*, vol. 8, no. 4, April 2000.
- [Kai02] Kaiser, J., "Elements of Effective Web Design," *About, Inc.*, 2002, disponível em <http://webdesign.about.com/library/weekly/aa091998.htm>.
- [Kan01] Kaner, C., "Pattern: Scenario Testing" (draft), 2001, disponível em www.testing.com/test-patterns/patterns/pattern-scenario-testing-kaner.html.
- [Kan93] Kaner, C., J. Falk, and H. Q. Nguyen, *Testing Computer Software*, 2nd ed., Van Nostrand-Reinhold, 1993.
- [Kan95] Kaner, C., "Lawyers, Lawsuits, and Quality Related Costs," 1995, disponível em www.badsoftware.com/plaintif.htm.
- [Kan11] Kannan, N., "Mobile Testing: Nine Strategy Tests You'll Want to Perform," 2011, disponível em <http://searchsoftwarequality.techtarget.com/tip/Mobile-testing-Nine-strategytests-youll-want-to-perform>.
- [Kar94] Karten, N., *Managing Expectations*, Dorset House, 1994.
- [Kau95] Kauffman, S., *At Home in the Universe*, Oxford, 1995.
- [Kaz03] Kazman, R., and A. Eden, "Defining the Terms Architecture, Design, and Implementation," *news@sei interactive*, Software Engineering Institute, vol. 6, no. 1, 2003, disponível em www.sei.cmu.edu/news-at-sei/columns/the_architect/2003/1q03/architect-1q03.htm.
- [Kaz98] Kazman, R., et al., *The Architectural Tradeoff Analysis Method*, Software Engineering Institute, CMU/SEI-98-TR-008, July 1998, resumo em <http://www.sei.cmu.edu/architecture/tools/evaluate/atam.cfm>.
- [Kea07] Keane, "Testing Mobile Business Applications," a white paper, 2007, disponível em www.keane.com.
- [Kei98] Keil, M., et al., "A Framework for Identifying Software Project Risks," *CACM*, vol. 41, no. 11, November 1998, pp. 76–83.
- [Kel00] Kelly, D., and R. Oshana, "Improving Software Quality Using Statistical Techniques, Information and Software Technology," *Elsevier*, vol. 42, August 2000, pp. 801–807, disponível em www.eng.auburn.edu/~kchang/comp6710/readings/Improving_Quality_with_Statistical_Testing_InfoSoftTech_August2000.pdf.
- [Ker05] Kerievsky, J., *Industrial XP: Making XP Work in Large Organizations*, Cutter Consortium, Executive Report, vol. 6., no. 2, 2005, disponível em www.cutter.com/content-and-analysis/resource-centers/agile-project-management/sample-our-research/apmr0502.html.
- [Ker11] Kershbaum, F., et al., "Secure Collaborative Supply-Chain Management," *IEEE Computer*, vol. 44, no. 9, September 2011, pp. 38–43.
- [Ker78] Kernighan, B., and P. Plauger, *The Elements of Programming Style*, 2nd ed., McGraw-Hill, 1978.
- [Kho12a] Khode, A., "Getting Started with Mobile Apps Testing," 2012, disponível em <http://www.mobileappstesting.com/getting-started-with-mobile-apps-testing/>.

- [Kho12b] Khode, A., “Checklist for Mobile Test Automation Tools,” 2012, disponível em <http://www.mobileappstesting.com/getting-started-with-mobile-apps-testing/>
- [Kir94] Kirani, S., and W. Tsai, “Specification and Verification of Object-Oriented Programs, Technical Report TR 94-64, Computer Science Department, University of Minnesota, December 1994.
- [Kiz05] Kizza, J., *Computer Network Security*, Springer, 2005.
- [Knu98] Knuth, D., *The Art of Computer Programming*, three volumes, Addison-Wesley, 1998.
- [Koe12] Koester, J., “The Seven Deadly Sins of MobileApp Design,” *Venture Beat/Mobile*, May 31, 2012, disponível em <http://venturebeat.com/2012/05/31/the-7-deadly-sins-of-mobileapp-design/>
- [Kon02] Konrad, S., and B. Cheng, “Requirements Patterns for Embedded Systems,” *Proceedings of the 10th Anniversary IEEE Joint International Conference on Requirements Engineering*, IEEE, September 2002, pp. 127–136, disponível em <http://citeseer.ist.psu.edu/669258.html>.
- [Kor03] Korpipaa, P., et al., “Managing Context Information in Mobile Devices,” *IEEE Pervasive Computing*, vol. 2, no. 3, July–September 2003, pp. 42–51.
- [Kra88] Krasner, G., and S. Pope, “A Cookbook for Using the Model-View Controller User Interface Paradigm in Smalltalk-80,” *Journal of Object-Oriented Programming*, vol. 1, no. 3, August–September 1988, pp. 26–49.
- [Kra95] Kraul, R., and L. Streeter, “Coordination in Software Development,” *CACM*, vol. 38, no. 3, March 1995, pp. 69–81.
- [Kru05] Krutchen, P., “Software Design in a Postmodern Era,” *IEEE Software*, vol. 22, no. 2, March–April 2005, pp. 16–18.
- [Kru06] Kruchten, P., H. Obbink, and J. Stafford (eds.), “Software Architectural” (special issue), *IEEE Software*, vol. 23, no. 2, March–April 2006.
- [Kru09] Kruchten, P., et al. “The Decision View’s Role in Software Architecture Practice,” *IEEE Software*, vol. 26, no. 2, March–April 2009, pp. 70–72.
- [Kur05] Kurzweil, R., *The Singularity Is Near*, Penguin Books, 2005.
- [Kur13] Kurzweil, R., *How to Create a Mind*, Viking, 2013.
- [Kyb84] Kyburg, H., *Theory and Measurement*, Cambridge University Press, 1984.
- [Laa00] Laakso, S., et al., “Improved Scroll Bars,” *CHI 2000 Conf. Proc.*, ACM, 2000, pp. 97–98, disponível em www.cs.helsinki.fi/u/salaakso/patterns/.
- [Lag10] Lago, P., et al., “Software Architecture: Framing Stakeholders’ Concerns,” *IEEE Software*, vol. 27, no. 6, November–December 2010, pp. 20–24.
- [Lai02] Laitenberger, A., “A Survey of Software Inspection Technologies,” in *Handbook on Software Engineering and Knowledge Engineering*, World Scientific Publishing Company, 2002.
- [Lam01a] Lam, W., “Testing E-Commerce Systems: A Practical Guide,” *IEEE IT Pro*, March–April 2001, pp. 19–28.
- [Lam01b] Lamsweerde, A. “Goal-Oriented Requirements Engineering: A Guided Tour,” *Proceedings of 5th IEEE International Symposium on Requirements Engineering*, Toronto, August 2009, pp. 249–263.
- [Lan01] Lange, M., “It’s Testing Time! Patterns for Testing Software,” June 2001, disponível em www.testing.com/test-patterns/patterns/index.html.
- [Lan02] Land, R., “A Brief Survey of Software Architecture,” technical report, Dept. of Computer Engineering, Mälardalen University, Sweden, February 2002.
- [Lan10] Lanubile, F., C. Ebert, R. Prikladnicki, and A. Vizcaino, “Collaboration Tools for Global Software Engineering,” *IEEE Software*, vol. 27, 2010, pp. 52–55.
- [Laz11] Lazzaroni, M., et al., *Reliability Engineering*, Springer, 2011.
- [Leh97a] Lehman, M., and L. Belady, *Program Evolution: Processes of Software Change*, Academic Press, 1997.
- [Leh97b] Lehman, M., et al., “Metrics and Laws of Software Evolution—The Nineties View,” *Proceedings of the 4th International Software Metrics Symposium (METRICS '97)*, IEEE, 1997, disponível em www.ece.utexas.edu/~perry/work/papers/feast1.pdf.

- [Let01] Lethbridge, T., and R. Laganier, *Object-Oriented Software Engineering: Practical Software Development Using UML and Java*, McGraw-Hill, 2001.
- [Let03a] Lethbridge, T., Personal communication on domain analysis, May 2003.
- [Let03b] Lethbridge, T., Personal communication on software metrics, June 2003.
- [Lev01] Levinson, M., "Let's Stop Wasting \$78 billion a Year," *CIO Magazine*, October 15, 2001, disponível em www.cio.com/archive/101501/wasting.html.
- [Lev95] Leveson, N., *Safeware: System Safety and Computers*, Addison-Wesley, 1995.
- [Lew09] Lewicki, R., B. Barry, and D. Saunders, *Negotiation*, McGraw-Hill, 2009.
- [Lie03] Lieberherr, K., "Demeter: Aspect-Oriented Programming," May 2003, disponível em www.ccs.neu.edu/home/lieber/LoD.html.
- [Lin79] Linger, R., H. Mills, and B. Witt, *Structured Programming*, Addison-Wesley, 1979.
- [Lin88] Linger, R., and H. Mills, "A Case Study in Cleanroom Software Engineering: The IBM COBOL Structuring Facility," *Proc. COMPSAC '88*, Chicago, October 1988.
- [Lin94] Linger, R., "Cleanroom Process Model," *IEEE Software*, vol. 11, no. 2, March 1994, pp. 50–58.
- [Lip10] Lipner, S., "The Security Development Life Cycle," June 24, 2010, disponível em https://www.owasp.org/images/7/78/OWASP_AppSec_Research_2010_Keynote_2_by_Lipner.pdf.
- [Lis88] Liskov, B., "Data Abstraction and Hierarchy," *SIGPLAN Notices*, vol. 23, no. 5, May 1988.
- [Liu98] Liu, K., et al., "Report on the First SEBPC Workshop on Legacy Systems," Durham University, February 1998, disponível em www.dur.ac.uk/CSM/SABA/legacy-wksp1/report.html.
- [Lon02] Longstreet, D., "Fundamental of Function Point Analysis," Longstreet Consulting, Inc., 2002, disponível em www.ifpug.com/fpafund.htm.
- [Lor94] Lorenz, M., and J. Kidd, *Object-Oriented Software Metrics*, Prentice Hall, 1994.
- [Lup08] Lupton, E., and J. Cole, *Graphic Design: The New Basics*, Princeton Architectural Press, 2008.
- [Maa07] Maassen, O., and S. Stelling, "Creational Patterns: Creating Objects in an OO System," 2007, disponível em www.informit.com/articles/article.asp?p=26452&rl=1.
- [Mad10] Madison, J., "Agile-Architecture Interactions," *IEEE Software*, vol. 27, no. 3, March–April 2010, pp. 41–48.
- [Mai10a] Maiden, N., and S. Jones, "Agile Requirements—Can We Have Our Cake and Eat It Too?" *IEEE Software*, vol. 27, no. 3, May–June 2010, pp. 20–24.
- [Mai10b] Maiden, N., "Service Design: It's All in the Brand," *IEEE Software*, vol. 27, no. 5, September–October 2010, pp. 18–19.
- [Man81] Mantai, M., "The Effect of Programming Team Structures on Programming Tasks," *CACM*, vol. 24, no. 3, March 1981, pp. 106–113.
- [Man97] Mandel, T., *The Elements of User Interface Design*, Wiley, 1997.
- [Mar00] Martin, R., "Design Principles and Design Patterns," 2000, disponível em www.objectmentor.com.
- [Mar01] Marciniak, J. (ed.), *Encyclopedia of Software Engineering*, 2nd ed., Wiley, 2001.
- [Mar02a] Marick, B., "Software Testing Patterns," 2002, disponível em www.testing.com/test-patterns/index.html.
- [Mar94] Marick, B., *The Craft of Software Testing*, Prentice Hall, 1994.
- [Mas02] Mascolo, C., et al., "Mobile Computing Middleware," contido em *Advanced Lectures on Networking* (E. Georgi, ed.), Springer-Verlag, 2002.
- [Mat94] Matson, J., et al., "Software Cost Estimation Using Function Points," *IEEE Trans. Software Engineering*, vol. SE-20, no. 4, April 1994, pp. 275–287.
- [McC04] McConnell, S., *Code Complete*, 2nd. ed., Microsoft Press, 2004.
- [McC09] McCaffrey, J., "Analyzing Risk Exposure and Risk using PERIL," *MSDN Magazine*, January 2009, disponível em <http://msdn.microsoft.com/en-us/magazine/dd315417.aspx>.
- [McC76] McCabe, T., "A Software Complexity Measure," *IEEE Trans. Software Engineering*, vol. SE-2, December 1976, pp. 308–320.

- [McC77] McCall, J., P. Richards, and G. Walters, "Factors in Software Quality," three volumes, NTIS AD-A049-014, 015, 055, November 1977.
- [McC94] McCabe, T. J., and A. H. Watson, "Software Complexity," *CrossTalk*, vol. 7, no. 12, December 1994, pp. 5–9.
- [McC96] McConnell, S., "Best Practices: Daily Build and Smoke Test," *IEEE Software*, vol. 13, no. 4, July 1996, pp. 143–144.
- [McC98] McConnell, S., *Software Project Survival Guide*, Microsoft Press, 1998.
- [McC99] McConnell, S., "Software Engineering Principles," *IEEE Software*, vol. 16, no. 2, March–April 1999, disponível em www.stevemcconnell.com/ieeesoftware/eic04.htm.
- [McD93] McDermid, J., and P. Rook, "Software Development Process Models," in *Software Engineer's Reference Book*, CRC Press, 1993, pp. 15/26–15/28.
- [McG91] McGlaughlin, R., "Some Notes on Program Design," *Software Engineering Notes*, vol. 16, no. 4, October 1991, pp. 53–54.
- [McG94] McGregor, J., and T. Korson, "Integrated Object-Oriented Testing and Development Processes," *Communications of the ACM*, vol. 37, no. 9, September, 1994, pp. 59–77.
- [McN10] McNeil, P., *The Web Designer's Idea Book*, vol. 2, How Publishers, 2010.
- [Mea10] Mead, N., and Jarzombek, J., "Advancing Software Assurance with Public-Private Collaboration," *IEEE Computer*, vol. 43, no. 9, September 2010, pp. 21–30.
- [Mei06] Meier, J., "Web Application Security Engineering," *IEEE Security and Privacy*, July–August 2006, pp. 16–24.
- [Mei09] Meier, J., et al., *Microsoft Application Architecture Guide*, 2nd ed., Microsoft Press, 2009, disponível em <http://msdn.microsoft.com/en-us/library/ff650706>.
- [Mei12] Meier, J., et al., "Chapter 19: Mobile Applications," *Application Architecture Guide, 2.0*, 2012, disponível em <http://apparchguide.codeplex.com/wikipage?title=Chapter%2019%20-%20Mobile%20Applications>
- [Men01] Mendes, E., N. Mosley, and S. Counsell, "Estimating Design and Authoring Effort," *IEEE Multimedia*, vol. 8, no. 1, January–March 2001, pp. 50–57.
- [Mer93] Merlo, E., et al., "Reengineering User Interfaces," *IEEE Software*, January 1993, pp. 64–73.
- [Mes08] Messeguer, R., et al. "Communication and Coordination Patterns to Support Mobile Collaboration," *Proceedings of 12th International Conference on Collaborative and Cooperative Work*, April 2008, pp. 565–570.
- [Mey09] Meyer, B., et al., "Programs that Test Themselves," *IEEE Computer*, vol. 42, no. 9, September 2009, pp. 46–55.
- [Mic04] Microsoft, "Prescriptive Architecture: Integration and Patterns," *MSDN*, May 2004, disponível em <http://msdn2.microsoft.com/en-us/library/ms978700.aspx>.
- [Mic12] "Principles of Service-Oriented Design," Microsoft, 2012, disponível em <http://msdn.microsoft.com/en-us/library/bb972954.aspx>
- [Mic13a] Microsoft, "Patterns and Practices," *MSDN*, disponível em <http://msdn.microsoft.com/en-us/library/ff647589.aspx>.
- [Mic13b] *Microsoft Accessibility Technology for Everyone*, 2013, disponível em www.microsoft.com/enable/.
- [Mil00a] Miller, E., "WebSite Testing," 2000, disponível em www.soft.com/eValid/Technology/WhitePapers/website.testing.html.
- [Mil04] Miler, J., and J. Gorski, "Risk Identification Patterns for Software Projects," *Foundations of Computing and Decision Sciences*, vol. 29, no. 1, 2004, pp. 115–131, disponível em http://iag.pg.gda.pl/iag/download/Miler-Gorski_Risk_Identification_Patterns.pdf.
- [Mil72] Mills, H., "Mathematical Foundations for Structured Programming," Technical Report FSC 71-6012, IBM Corp., Federal Systems Division, Gaithersburg, MD, 1972.
- [Mil77] Miller, E., "The Philosophy of Testing," in *Program Testing Techniques*, IEEE Computer Society Press, 1977, pp. 1–3.
- [Mil87] Mills, H., M. Dyer, and R. Linger, "Cleanroom Software Engineering," *IEEE Software*, September 1987, pp. 19–25.

- [Mil88] Mills, H., "Stepwise Refinement and Verification in Box Structured Systems," *Computer*, vol. 21, no. 6, June 1988, pp. 23–35.
- [Min95] Minoli, D., *Analyzing Outsourcing*, McGraw-Hill, 1995.
- [Mob11] Mobile Labs, "Mobile Application Test Automation: Best Practices for Best Results," a white paper, 2011, disponível em <http://mobilelabsinc.com/wp-content/uploads/2012/01/Mobile-Application-Test-Automation-Best-Practices-White-Paper.pdf>.
- [Mob12] "Mobile UI Patterns," 2012, disponível em <http://mobile-patterns.com/>.
- [Mor05] Morales, A., "The Dream Team," *Dr. Dobbs Portal*, March 3, 2005, disponível em www.ddj.com/dept/global/184415303.
- [Mor81] Moran, T., "The Command Language Grammar: A Representation for the User Interface of Interactive Computer Systems," *Intl. Journal of Man-Machine Studies*, vol. 15, pp. 3–50.
- [Mus87] Musa, J., A. Iannino, and K. Okumoto, *Engineering and Managing Software with Reliability Measures*, McGraw-Hill, 1987.
- [Mye78] Myers, G., *Composite Structured Design*, Van Nostrand, 1978.
- [Mye79] Myers, G., *The Art of Software Testing*, Wiley, 1979.
- [Myl11] Malavarapu, V., and M. Inanamdar, "Taking Testing to the Cloud," 2012, disponível em <http://www.cognizant.com/InsightsWhitepapers/Taking-Testing-to-the-Cloud.pdf>.
- [NAS07] NASA, *Software Risk Checklist*, Form LeR-F0510.051, March 2007, disponível em <http://osat-ext.grc.nasa.gov/rmo/spa/SoftwareRiskChecklist.doc>.
- [Nei11] Neil, T., "A Look Inside Mobile Design Patterns," UXBooth.com, November 12, 2011, disponível em <http://www.uxbooth.com/blog/mobile-design-patters/>.
- [Nei12] Neil, T., *Mobile Design Pattern Gallery*, O'Reilly Media, 2012.
- [Ngu00] Nguyen, H., "Testing Web-Based Applications," *Software Testing and Quality Engineering*, May–June 2000, disponível em www.stqemagazine.com.
- [Ngu01] Nguyen, H., *Testing Applications on the Web*, Wiley, 2001.
- [Nie00] Nielsen, J., *Designing Web Usability*, New Riders Publishing, 2000.
- [Nie92] Nierstrasz, O., S. Gibbs, and D. Tschritzis, "Component-Oriented Software Development," *CACM*, vol. 35, no. 9, September 1992, pp. 160–165.
- [Nie94] Nielsen, J., and J. Levy, "Measuring Usability: Preference vs. Performance," *CACM*, vol. 37, no. 4, April 1994, pp. 65–75.
- [Nie96] Nielsen, J., and A. Wagner, "User Interface Design for the WWW," *Proc. CHI '96 Conf. on Human Factors in Computing Systems*, ACM Press, 1996, pp. 330–331.
- [Nir10] Niranjana, P., and C. V. Guru Rao, "A Mockup Tool for Software Component Reuse Library," *Intl. J. Software Engineering & Applications*, vol. 1, no. 2, April 2010, disponível em <http://airccse.org/journal/ijsea/papers/0410ijsea1.pdf>.
- [Nok13] "Category: Mobile Design," Nokia Developer, 2013, disponível em http://www.developer.nokia.com/Community/Wiki/Category:Mobile_Design_Patterns.
- [Nog00] Nogueira, J., C. Jones, and Luqi, "Surfing the Edge of Chaos: Applications to Software Engineering," Command and Control Research and Technology Symposium, Naval Post Graduate School, Monterey, CA, June 2000, disponível em www.dodccrp.org/2000CCRTS/cd/html/pdf_papers/Track_4/075.pdf.
- [Nor70] Norden, P., "Useful Tools for Project Management" in *Management of Production*, M. K. Starr (ed.), Penguin Books, 1970.
- [Nor86] Norman, D. A., "Cognitive Engineering," in *User Centered Systems Design*, Lawrence Earlbaum Associates, 1986.
- [Nor88] Norman, D., *The Design of Everyday Things*, Doubleday, 1988.
- [Nov04] Novotny, O., "Next Generation Tools for Object-Oriented Development," *The Architecture Journal*, January 2005, disponível em <http://msdn2.microsoft.com/en-us/library/aa480062.aspx>.
- [Noy02] Noyes, B., "Rugby, Anyone?" *Managing Development* (an online publication of Fawcette Technical Publications), June 2002, disponível em www.fawcette.com/resources/managingdev/methodologies/scrum/.

- [Nun11] Nunes, N., L. Constantine, and R. Kazman, “iUCP: Estimating Interactive Software Project Size with Enhanced Use Case Points,” *IEEE Software*, vol. 28, no. 4, July–August 2011, pp. 64–73.
- [Obj10] “The Dependency Inversion Principle,” *Objectmentor.com*, 2010, disponível em www.objectmentor.com/resources/articles/dip.pdf.
- [Off02] Offutt, J., “Quality Attributes of Web Software Applications,” *IEEE Software*, March–April 2002, pp. 25–32.
- [Ols06] Olsen, G., “From COM to Common,” *Component Technologies*, ACM, vol. 4, no. 5, June 2006, disponível em <http://acmqueue.com/modules.php?name=Content&pa=showpage&pid=394>.
- [Ols99] Olsina, L., et al., “Specifying Quality Characteristics and Attributes for Web Sites,” *Proc. 1st ICSE Workshop on Web Engineering*, ACM, Los Angeles, May 1999.
- [OMG03a] Object Management Group, *OMG Unified Modeling Language Specification*, version 1.5, March 2003, disponível em www.rational.com/uml/resources/documentation/.
- [OMG03b] “Object Constraint Language Specification,” in *Unified Modeling Language*, v2.0, Object Management Group, September 2003, disponível em www.omg.org.
- [Orf99] Orfali, R., D. Harkey, and J. Edwards, *Client/Server Survival Guide*, 3rd ed., Wiley, 1999.
- [Os90] Osborne, W. M., and E. J. Chikofsky, “Fitting Pieces to the Maintenance Puzzle,” *IEEE Software*, January 1990, pp. 10–11.
- [OSO12] *OpenSource.org*, 2012, disponível em www.opensource.org/.
- [Pag85] Page-Jones, M., *Practical Project Management*, Dorset House, 1985, p. vii.
- [Par72] Parnas, D., “On Criteria to Be Used in Decomposing Systems into Modules,” *CACM*, vol. 14, no. 1, April 1972, pp. 221–227.
- [Par96a] Pardee, W., *To Satisfy and Delight Your Customer*, Dorset House, 1996.
- [Par96b] Park, R. E., W. B. Goethert, and W. A. Florac, *Goal Driven Software Measurement—A Guidebook*, CMU/SEI-96-BH-002, Software Engineering Institute, Carnegie Mellon University, August 1996.
- [Par10] Parnas, D., “Interview: Software Security in the Real World,” *IEEE Computer*, vol. 43, no. 1, January 2010, pp. 28–34.
- [Par11] Pardo, C., et al., “Harmonizing Quality Assurance Processes and Product Characteristics,” *IEEE Computer*, June 2011, pp. 94–96.
- [Pas10] Passos, L., et al., “Static Architecture-Conformance Checking: An Illustrative Overview,” *IEEE Software*, vol. 27, no. 5, September–October 2010, pp. 82–89.
- [Pat07] Patton, J., “Understanding User Centricity,” *IEEE Software*, vol. 24, no. 6, November–December 2007, pp. 9–11.
- [Pau94] Paulish, D., and A. Carleton, “Case Studies of Software Process Improvement Measurement,” *Computer*, vol. 27, no. 9, September 1994, pp. 50–57.
- [Pea11] Pearson, S., and M. Mont, “Sticky Policies: An Approach for Managing Privacy across Multiple Parties,” *IEEE Computer*, vol. 44, no. 9, September 2011, pp. 60–68.
- [Pee11] Peeters, J., “Agile Security Requirements Engineering,” *Proceedings of First International Workshop on Empirical Requirements Engineering*, 2011.
- [Pir74] Pirsig, R., *Zen and the Art of Motorcycle Maintenance*, Bantam Books, 1974.
- [Pha89] Phadke, M., *Quality Engineering Using Robust Design*, Prentice Hall, 1989.
- [Pha97] Phadke, M., “Planning Efficient Software Tests,” *CrossTalk*, vol. 10, no. 10, October 1997, pp. 11–15.
- [Phi02] Phillips, M., “CMMI V1.1 Tutorial,” April 2002, disponível em www.sei.cmu.edu/cmmi/.
- [Phi98] Phillips, D., *The Software Project Manager's Handbook*, IEEE Computer Society Press, 1998.
- [Pol45] Polya, G., *How to Solve It*, Princeton University Press, 1945.
- [Poo88] Poore, J., and H. Mills, “Bringing Software Under Statistical Quality Control,” *Quality Progress*, November 1988, pp. 52–55.
- [Poo93] Poore, J., H. Mills, and D. Mutchler, “Planning and Certifying Software System Reliability,” *IEEE Software*, vol. 10, no. 1, January 1993, pp. 88–99.
- [Pop08] Popcorn, F., *Faith Popcorn's Brain Reserve*, 2008, disponível em www.faithpopcorn.com/.

- [Pot04] Potter, M., *Set Theory and Its Philosophy: A Critical Introduction*, Oxford University Press, 2004.
- [Pow02] Powell, T., *Web Design*, 2nd ed., McGraw-Hill/Osborne, 2002.
- [Pow98] Powell, T., *Web Site Engineering*, Prentice Hall, 1998.
- [Pra07] Pratt, M., "Five Tips for Building an Incident Response Plan," *Computerworld*, May 16, 2007, disponível em http://www.computerworld.com/s/article/9019558/Five_tips_for_building_an_incident_response_pla.
- [Pre05] Pressman, R., *Adaptable Process Model*, Version 2.0, R. S. Pressman & Associates, 2005, disponível em www.rspa.com/apm/index.html.
- [Pre08] Pressman, R., and D. Lowe, *Web Engineering: A Practitioner's Approach*, McGraw-Hill, 2008.
- [Pre88] Pressman, R., *Making Software Engineering Happen*, Prentice Hall, 1988.
- [Pre94] Premerlani, W., and M. Blaha, "An Approach for Reverse Engineering of Relational Databases," *CACM*, vol. 37, no. 5, May 1994, pp. 42–49.
- [Pri10] Prince, B., "10 Most Dangerous Web App Security Flaws," *eWeek.com*, April, 19, 2010, disponível em <http://www.eweek.com/c/a/Security/10-Most-Dangerous-Web-App-Security-Risks-730757/>.
- [Put78] Putnam, L., "A General Empirical Solution to the Macro Software Sizing and Estimation Problem," *IEEE Trans. Software Engineering*, vol. SE-4, no. 4, July 1978, pp. 345–361.
- [Put92] Putnam, L., and W. Myers, *Measures for Excellence*, Yourdon Press, 1992.
- [Put97a] Putnam, L., and W. Myers, "How Solved Is the Cost Estimation Problem?" *IEEE Software*, November 1997, pp. 105–107.
- [Put97b] Putnam, L., and W. Myers, *Industrial Strength Software: Effective Management Using Measurement*, IEEE Computer Society Press, 1997.
- [Pyz03] Pyzdek, T., *The Six Sigma Handbook*, McGraw-Hill, 2003.
- [QAI08] *A Software Engineering Curriculum*, QAI, 2008, informações disponíveis em www.qaieschool.com/innerpages/offer.asp.
- [QSM02] "QSM Function Point Language Gearing Factors," Version 2.0, *Quantitative Software Management*, 2002, disponível em www.qsm.com/FPGearing.html.
- [Qur09] Qureshi N., and A. Perini, "Engineering Adaptive Requirements," *Proceedings of Workshop on Software Engineering for Adaptive and Self-Managing Systems*, Vancouver, May 2009, pp. 126–131.
- [Rad02] Radice, R., *High-Quality Low Cost Software Inspections*, Paradoxicon Publishing, 2002.
- [Rai06] Raiffa, H., *The Art and Science of Negotiation*, Belknap Press, 2005.
- [Rat12] Raatikainen, M., et al., "Mobile Content as a Service: A Blueprint for a Vendor-Neutral Cloud of Mobile Devices," *IEEE Software*, vol. 29, no. 4, July–August, 2012, pp. 28–32.
- [Red10] Redwine, S., "Fitting Software Assurance into Higher Education," *IEEE Computer*, vol. 43, no. 9, September 2010, pp. 41–66.
- [Ree99] Reel, J., "Critical Success Factors in Software Projects," *IEEE Software*, May 1999, pp. 18–23.
- [Reu12] Reuveni, D. "Crowdsourcing Provides Answer to App Testing Dilemma," 2012, disponível em <http://www.wirelessweek.com/Articles/2010/02/Mobile-Content-Crowdsourcing-Answer-App-Testing-Dilemma-Mobile-Applications/>.
- [Ric01] Ricadel, A., "The State of Software Quality," *InformationWeek*, May 21, 2001, disponível em www.informationweek.com/838/quality.htm.
- [Ric04] Rico, D., *ROI of Software Process Improvement*, J. Ross Publishing, 2004. Um resumo pode ser encontrado em <http://davidfrico.com/rico03a.pdf>.
- [Rob10] Robinson, W., "A Roadmap for Comprehensive Requirements Monitoring," *IEEE Computer*, vol. 43, no. 5, May 2010, pp. 64–72.
- [Roc06] *Graphic Design That Works*, Rockport Publishers, 2006.
- [Roc11] Rocha, F., S. Abreus, and M. Correia, "The Final Frontier: Confidentiality and Privacy in the Cloud," *IEEE Computer*, vol. 44, no. 9, September 2011, pp. 44–50.

- [Roc94] Roche, J. M., "Software Metrics and Measurement Principles," *Software Engineering Notes*, ACM, vol. 19, no. 1, January 1994, pp. 76–85.
- [Rod12] Rodriguez, S., "Half of Americans Now Have Smartphones," *The Los Angeles Times*, August 14, 2012, disponível em <http://www.latimes.com/business/technology/la-fi-tna-americans-smartphones-20120814,0,6077673.story>.
- [Rod98] Rodden, T., et al., "Exploiting Context in HCI Design for Mobile Systems," *Proceedings of Workshop on Human Computer Interaction with Mobile Devices*, 1998.
- [Roe00] Roetzheim, W., "Estimating Internet Development," *Software Development*, August 2000, disponível em www.sdmagazine.com/documents/s=741/sdm0008d/0008d.htm.
- [Rog12] Rogers, A., "Software Quality Assurance Engineers Are the Happiest Workers in America," *Business Insider*, April 16, 2012, disponível em <http://www.businessinsider.com/happiest-jobs-in-america-2012-4>.
- [Rom11] Roman, R., et al., "Securing the Internet of Things," *IEEE Computer*, vol. 44, no. 9, September 2011, pp. 51–58.
- [Roo09] Rooksby, J., et al., "Testing in the Wild: The Social and Organizational Dimensions of Real World Practice," *Journal of Computer Supported Work*, vol. 18, no. 5–6, December 2009, pp. 559–580.
- [Roo96] Roos, J., "The Poised Organization: Navigating Effectively on Knowledge Landscapes," 1996, disponível em www.imd.ch/fac/roos/paper_po.html.
- [Ros04] Rosenhainer, L., "Identifying Crosscutting Concerns in Requirements Specifications," 2004, disponível em <http://trese.cs.utwente.nl/workshops/oopsla-early-aspects-2004/Papers/Rosenhainer.pdf>.
- [Ros75] Ross, D., J. Goodenough, and C. Irvine, "Software Engineering: Process, Principles and Goals," *IEEE Computer*, vol. 8, no. 5, May 1975.
- [Rot02] Roth, J., "Seven Challenges for Developers of Mobile Groupware," in *Proceedings of Computer Human Interaction Workshop on Mobile Ad Hoc Collaboration*, 2002.
- [Rou02] Rout, T. (project manager), *SPICE: Software Process Assessment—Part 1: Concepts and Introductory Guide*, 2002, disponível em www.sqi.gu.edu.au/spice/suite/download.html.
- [Roy70] Royce, W., "Managing the Development of Large Software Systems: Concepts and Techniques," *Proc. WESCON*, August 1970.
- [Roz11] Rozanski, N., and E. Woods, *Software Systems Architecture*, 2nd ed., Addison-Wesley, 2011.
- [Rum91] Rumbaugh, J., et al., *Object-Oriented Modeling and Design*, Prentice Hall, 1991.
- [Rya11a] Ryan, M., "Cloud Computing Privacy Concerns on Our Doorstep," *Communication of the ACM*, vol. 44, no. 2, January 2011, pp. 36–38.
- [Sae11] Saeed, A., T. Chen, and O. Alzubi, "Malicious and Spam Posts in Online Social Networks," *IEEE Computer*, vol. 44, no. 9, September 2011, pp. 23–28.
- [Saf08] Safanov, V., *Using Aspect-Oriented Programming for Trustworthy Software Development*, Wiley-Interscience, 2008.
- [Sai11] Saieddian, H., and D. Broyles, "Security Vulnerabilities in the Same Origin Policy Implications and Alternatives," *IEEE Computer*, vol. 44, no. 9, September 2011, pp. 29–36.
- [Sal09] Saleh, K., and G. Elshahry, "Modeling Security Requirements for Trustworthy Systems," *Encyclopedia of Information Science and Technology*, 2nd ed., 2009.
- [Sar06] Sarwate, A., "Hot or Not: Web Application Vulnerabilities," *SC Magazine*, December 27, 2006, disponível em <http://www.scmagazine.com/hot-or-not-web-applicationvulnerabilities/article/34315/>.
- [Saw08] Sawyer, S., et al., "Social Interactions of Information Systems Development Teams: A Performance Perspective," *Information Systems Journal*, vol. 20, 2008, pp. 81–107.
- [Sca00] Scacchi, W., "Understanding Software Process Redesign Using Modeling, Analysis, and Simulation," *Software Process Improvement and Practice*, Wiley, 2000, pp. 185–195, disponível em www.ics.uci.edu/~wscacchi/Papers/Software_Process_Redesign/SPIP-ProSim99.pdf.
- [Sce02] Sceppa, D., *Microsoft ADO.NET*, Microsoft Press, 2002.

- [Sch01b] Schwaber, K., and M. Beedle, *Agile Software Development with SCRUM*, Prentice Hall, 2001.
- [Sch03] Schlickman, J., *ISO 9001: 2000 Quality Management System Design*, Artech House Publishers, 2003.
- [Sch06] Schmidt, D., "Model-Driven Engineering," *IEEE Computer*, vol. 39, no. 2, February 2006, pp. 25–31.
- [Sch09] Schumacher, R. (ed.), *Handbook of Global User Research*, Morgan-Kaufmann, 2009.
- [Sch11] Schilit, B. "Mobile Computing: Looking to the Future," *IEEE Computer*, vol. 44, no. 5, May 2011, pp. 28–29.
- [Sch98a] Schneider, G., and J. Winters, *Applying Use Cases*, Addison-Wesley, 1998.
- [Sch98c] Schulmeyer, G., and J. McManus (eds.), *Handbook of Software Quality Assurance*, 3rd ed., Prentice Hall, 1998.
- [Sch99] Schneidewind, N., "Measuring and Evaluating Maintenance Process Using Reliability, Risk, and Test Metrics," *IEEE Trans. SE*, vol. 25, no. 6, November–December 1999, pp. 768–781, disponível em www.dacs.dtic.mil/topics/reliability/IEEETrans.pdf.
- [SDS08] Spice Document Suite, "The SPICE and ISO Document Suite," *ISO-Spice*, 2008, disponível em www.isospice.com/articles/9/1/SPICE-Project/Page1.html.
- [SEE03] The Software Engineering Ethics Research Institute, "UCITA Updates," 2003, disponível em <http://seeri.etsu.edu/default.htm>.
- [SEI00] SCAMPI, *V1.0 Standard CMMI® Assessment Method for Process Improvement: Method Description*, Software Engineering Institute, Technical Report CMU/SEI-2000-TR-009, disponível em www.sei.cmu.edu/publications/documents/00.reports/00tr009.html.
- [SEI02] "Maintainability Index Technique for Measuring Program Maintainability," SEI, 2002, disponível em www.sei.cmu.edu/str/descriptions/mitmpm_body.html.
- [SEI08] "The Ideal Model," Software Engineering Institute, 2008, disponível em www.sei.cmu.edu/ideal/.
- [SEI13] "Software Product Lines—Overview," Software Engineering Institute, 2013, disponível em www.sei.cmu.edu/productlines/.
- [Sha05] Shalloway, A., and J. Trott, *Design Patterns Explained*, 2nd ed., Addison-Wesley, 2005.
- [Sha09] Shaw, M., "Continuing Prospects for an Engineering Discipline of Software," *IEEE Software*, vol. 26, no. 8, November–December 2009, pp. 64–67.
- [Sha95a] Shaw, M., and D. Garlan, "Formulations and Formalisms in Software Architecture," *Volume 1000—Lecture Notes in Computer Science*, Springer-Verlag, 1995.
- [Sha95b] Shaw, M., et al., "Abstractions for Software Architecture and Tools to Support Them," *IEEE Trans. Software Engineering*, vol. SE-21, no. 4, April 1995, pp. 314–335.
- [Sha96] Shaw, M., and D. Garlan, *Software Architecture*, Prentice Hall, 1996.
- [She10] Sheldon, F., and Vishik, C. "Moving Toward Trustworthy Systems: R&D Essentials," *IEEE Computer*, vol. 43, no. 9, September 2010, pp. 31–40.
- [Shn04] Shneiderman, B., and C. Plaisant, *Designing the User Interface*, 4th ed., Addison-Wesley, 2004.
- [Shn09] Shneiderman, B., et al., *Designing the User Interface*, 5th ed., Addison-Wesley, 2009.
- [Shn80] Shneiderman, B., *Software Psychology*, Winthrop Publishers, 1980, p. 28.
- [Sho83] Shooman, M., *Software Engineering*, McGraw-Hill, 1983.
- [Shu12] Shull, F., "Designing a World at Your Fingertips: A Look at Mobile User Interfaces," *IEEE Software*, vol. 29, no. 4, July–August, 2012, pp. 4–7.
- [Sin08] Sinn, R. H., *Software Security Technologies: A Programmatic Approach*, Addison-Wesley, 2008.
- [Sne03] Snee, R., and R. Hoerl, *Leading Six Sigma*, Prentice Hall, 2003.
- [Sne95] Sneed, H., "Planning the Reengineering of Legacy Systems," *IEEE Software*, January 1995, pp. 24–25.
- [Soa10] Soares, G., et al., "Making Program Refactoring Safer," *IEEE Software*, vol. 37, no. 4, July–August 2010, pp. 52–57.
- [Soa11] Soasta, "Five Strategies for Performance Testing Mobile Applications," a white paper, 2011, disponível em <http://info.soasta.com/performance-testing-mobile-apps-sem2.html>.

- [Sob10] Sobel, A., and G. McGraw, "Interview: Software Security in the Real World," *IEEE Computer*, vol. 43, no. 9, September 2010, pp. 47–53.
- [Sol99] van Solingen, R., and E. Berghout, *The Goal/Question/Metric Method*, McGraw-Hill, 1999.
- [Som05] Somerville, I., "Integrating Requirements Engineering: A Tutorial," *IEEE Software*, vol. 22, no. 1, January–February 2005, pp. 16–23.
- [Som97] Somerville, I., and P. Sawyer, *Requirements Engineering*, Wiley, 1997.
- [Sou08] de Sousa, C., and D. Redmiles, "An Empirical Study of Software Developer's Management of Dependencies and Changes," *ICSE Proceedings*, May 2008, disponível em <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.158.427&rep=rep1&type=pdf>.
- [Spa11] Spagnolli, B., et al., "Eco-Feedback on the Go: Motivating Energy Awareness," *IEEE Computer*, vol. 44, no. 5, May 2011, pp. 38–45.
- [SPI99] "SPICE: Software Process Assessment, Part 1: Concepts and Introduction," Version 1.0, ISO/IEC JTC1, 1999.
- [Spl01] Splaine, S., and S. Jaskiel, *The Web Testing Handbook*, STQE Publishing, 2001. [Spo02] Spolsky, J., "The Law of Leaky Abstractions," November 2002, disponível em www.joeonsoftware.com/articles/LeakyAbstractions.html.
- [Spr04] Spriestersbach, A., and T. Springer, "Quality Attributes in Mobile Web Application Development," 2004, disponível em <http://atlas.tk.informatik.tu-darmstadt.de/Publications/2004/profes.pdf>.
- [Sri01] Sridhar, M., and N. Mandyam, "Effective Use of Data Models in Building Web Applications," 2001, disponível em www2002.org/CDROM/alternate/698/.
- [SSO08] Software-Supportability.org, 2008, disponível em www.software-supportability.org/.
- [Sta10] Stafford, T., and R. Poston, "Online Security Threats and Computer User Intentions," *IEEE Computer*, vol. 43, no. 1, January 2010, pp. 58–64.
- [Sta97] Stapleton, J., *DSDM—Dynamic System Development Method: The Method in Practice*, Addison-Wesley, 1997.
- [Sta97b] Statz, J., D. Oxley, and P. O'Toole, "Identifying and Managing Risks for Software Process Improvement," *CrossTalk*, April 1997, disponível em www.stsc.hill.af.mil/crosstalk/1997/04/identifying.asp.
- [Ste93] Stewart, T., "Reengineering: The Hot New Managing Tool," *Fortune*, August 23, 1993, pp. 41–48.
- [Sto05] Stone, D., et al., *User Interface Design and Evaluation*, Morgan Kaufman, 2005.
- [Str08] Stickland, J., "How Cloud Computing Works," <http://computer.howstuffworks.com/cloud-computing/cloud-computing.htm>, April, 2008.
- [Ste10] Stephens, M., and D. Rosenberg, *Design Driven Testing*, Apress, 2010.
- [Tai89] Tai, K., "What to Do Beyond Branch Testing," *ACM Software Engineering Notes*, vol. 14, no. 2, April 1989, pp. 58–61.
- [Tai12] Taivalsaari, A., and K. Systa, "Mobile Content as a Service: A Blueprint for a Vendor-Neutral Cloud of Mobile Devices," *IEEE Software*, vol. 29, no. 4, July–August 2012, pp. 28–33.
- [Tan01] Tandler, P., "Aspect-Oriented Model-Driven Development for Mobile Context-Aware Computing," *Proceedings of UbiComp 2001: Ubiquitous Computing*, 2001.
- [Tay09] Taylor, R., N. Medvidovic, and E. Dashofy, *Software Architecture*, Wiley, 2009.
- [Tay90] Taylor, D., *Object-Oriented Technology: A Manager's Guide*, Addison-Wesley, 1990.
- [The01] Thelin, T., H. Petersson, and C. Wohlin, "Sample Driven Inspections," *Proc. of Workshop on Inspection in Software Engineering (WISE'01)*, Paris, France, July 2001, pp. 81–91. Disponível em <http://www.cas.mcmaster.ca/wise/wise01/ThelinPeterssonWohlin.pdf>.
- [The13] "The Emergence of Cloud Computing," 2013, disponível em <http://www.opensourcery.com/clouddev.htm>. [Tho04] Thomas, J., et al., *Java Testing Patterns*, Wiley, 2004.
- [Tho92] Thomsett, R., "The Indiana Jones School of Risk Management," *American Programmer*, vol. 5, no. 7, September 1992, pp. 10–18.
- [Tic05] *TickIT*, 2005, disponível em www.tickit.org/. [Tid02] Tidwell, J., "IU Patterns and Techniques," May, 2002, disponível em <http://designinginterfaces.com/>.

- [Til00] Tillman, H., "Evaluating Quality on the Net," Babson College, May 30, 2000, disponível em www.hopetillman.com/findqual.html#2.
- [Tog01] Tognozzi, B., "First Principles," *askTOG*, 2001, disponível em www.asktog.com/basics/firstPrinciples.html.
- [Tra95] Tracz, W., "Third International Conference on Software Reuse—Summary," *ACM Software Engineering Notes*, vol. 20, no. 2, April 1995, pp. 21–22.
- [Tyr05] Tyree, J., and A. Akerman, "Architectural Decisions: Demystifying Architecture," *IEEE Software*, vol. 22, no. 2, March–April, 2005.
- [Uem99] Uemura, T., S. Kusumoto, and K. Inoue, "A Function Point Measurement Tool for UML Design Specifications," *Proc. of Sixth International Symposium on Software Metrics*, IEEE, November 1999, pp. 62–69.
- [Ull97] Ullman, E., *Close to the Machine: Technophilia and its Discontents*, City Lights Books, 2002.
- [Uni03] Unicode, Inc., *The Unicode Home Page*, 2003, disponível em www.unicode.org/.
- [USA87] *Management Quality Insight*, AFCSP 800-14 (U.S. Air Force), January 20, 1987.
- [Ute12] UTest, E-book: *Essential Guide to Mobile App Testing, 2012*, disponível em <http://www.utest.com/landing-blog/essential-guide-mobile-app-testing>.
- [UXM10] "Four Best User Interface Pattern Libraries," *UX Movement.com*, disponível em <http://uxmovement.com/resources/4-best-design-pattern-libraries/>.
- [Vac06] Vacca, J., *Practical Internet Security*, Springer, 2006.
- [Van02] Van Steen, M., and A. Tanenbaum, *Distributed Systems: Principles and Paradigms*, Prentice Hall, 2002.
- [Van89] Van Vleck, T., "Three Questions About Each Bug You Find," *ACM Software Engineering Notes*, vol. 14, no. 5, July 1989, pp. 62–63.
- [Ven03] Venners, B., "Design by Contract: A Conversation with Bertrand Meyer," *Artima Developer*, December 8, 2003, disponível em www.artima.com/intv/contracts.html.
- [Vin11] Vinson, L., "Mobile Application Testing: Process, Tools, and Techniques," 2011, disponível em <http://threeminds.organic.com/2011/05/mobile-application-testing-process-toolstechniques.html>.
- [Voa12] Voas, J., et al., "Mobile Software App Takeover," *IEEE Software*, vol. 29, no. 4, July–August 2012, pp. 25–27.
- [Wal03] Wallace, D., I. Raggett, and J. Aufgang, *Extreme Programming for Web Projects*, Addison-Wesley, 2003.
- [Wal12] Walker, J., "Computer Programmers Learn Tough Lesson in Sharing," *The Wall Street Journal*, vol. 260, no. 48, August 27, 2012, p. 1.
- [War07] Ward, M., "Using VoIP Software Building zBlocks—A Look at the Choices," TMNNet, 2007, disponível em www.tmcnet.com/voip/0605/featurearticle-using-voip-software-building-blocks.htm.
- [War74] Warnier, J. D., *Logical Construction of Programs*, Van Nostrand-Reinhold, 1974.
- [Was10] Wasserman, A., "Software Engineering Issues for Mobile Application Development," *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, 2010.
- [Web05] Weber, S., *The Success of Open Source*, Harvard University Press, 2005.
- [Web13] Web Application Security Consortium, 2013, disponível em <http://www.webappsec.org/>.
- [Wee11] Weevers, I., "Seven Guidelines for Designing High Performance Mobile User Experiences," *Smashing Magazine*, July 18, 2011, disponível em <http://uxdesign.smashing-magazine.com/2011/07/18/seven-guidelines-for-designing-high-performancemobile-user-experiences/>.
- [Wei86] Weinberg, G., *On Becoming a Technical Leader*, Dorset House, 1986.
- [Wel01] vanWelie, M., "Interaction Design Patterns," 2001, disponível em www.welie.com/patterns/.
- [Wel99] Wells, D., "XP—Unit Tests," 1999, disponível em www.extremeprogramming.org/rules/unittests.html.

- [Wev11] Wever, A., and N. Maiden, "Requirements Analysis: The Next Generation," *IEEE Software*, vol. 28, no. 2, March–April 2011, pp. 22–23.
- [Whi07] Whitehead, J., "Collaboration in Software Engineering: A Roadmap," in *Future of Software Engineering 2007*, L. Briand and A. Wolf (eds.), IEEE-CS Press, 2007.
- [Whi08] White, J., "Start Your Engines: Mobile Application Development," April 22, 2008, disponível em <http://www.devx.com/SpecialReports/Article/37693>.
- [Whi12] Whittaker, J., et al., *How Google Tests Software*, Addison-Wesley, 2012.
- [Whi97] Whitmire, S., *Object-Oriented Design Measurement*, Wiley, 1997.
- [Wie02] Wiegers, K., *Peer Reviews in Software*, Addison-Wesley, 2002.
- [Wie03] Wiegers, K., *Software Requirements*, 2nd ed., Microsoft Press, 2003.
- [Wik12] Wikipedia, "Data Modeling," 2012, disponível em http://en.wikipedia.org/wiki/Data_modeling.
- [Wik13] "Cloud Computing," January 2013, disponível em http://en.wikipedia.org/wiki/Cloud_computing.
- [Wil00] Williams, L., and R. Kessler, "All I Really Need to Know about Pair Programming I Learned in Kindergarten," *CACM*, vol. 43, no. 5, May 2000, disponível em <http://collaboration.csc.ncsu.edu/laurie/Papers/Kindergarten.PDF>.
- [Wil05] Willoughby, M., "Q&A: Quality Software Means More Secure Software," *Computerworld*, March 21, 2005, disponível em www.computerworld.com/securitytopics/security/story/0,10801,91316,00.html.
- [Wil97] Williams, R., J. Walker, and A. Dorofee, "Putting Risk Management into Practice," *IEEE Software*, May 1997, pp. 75–81.
- [Wil99] Wilkens, T., "Earned Value, Clear and Simple," *Primavera Systems*, April 1, 1999, p. 2.
- [Win90] Wing, J., "A Specifier's Introduction to Formal Methods," *IEEE Computer*, vol. 23, no. 9, September 1990, pp. 8–24.
- [Wir71] Wirth, N., "Program Development by Stepwise Refinement," *CACM*, vol. 14, no. 4, 1971, pp. 221–227.
- [Wir90] Wirfs-Brock, R., B. Wilkerson, and L. Weiner, *Designing Object-Oriented Software*, Prentice Hall, 1990.
- [WMT02] *Web Mapping Testbed Tutorial*, 2002, disponível em www.webmapping.org/vcgdocuments/vcgTutorial/.
- [Woh94] Wohlin, C., and P. Runeson, "Certification of Software Components," *IEEE Trans. Software Engineering*, vol. SE-20, no. 6, June 1994, pp. 494–499.
- [Wor04] World Bank, *Digital Technology Risk Checklist*, 2004, disponível em www.moonv6.org/lists/att-0223/WWBANK_Technology_Risk_Checklist_Ver_6point1.pdf.
- [Wri11] Wright, A., "Lessons Learned: Architects Are Facilitators, Too!" *IEEE Software*, vol. 28, no. 2, January–February 2011, pp. 70–72.
- [W3C03] World Wide Web Consortium, *Web Content Accessibility Guidelines*, 2003, disponível em www.w3.org/TR/2003/WD-WCAG20-20030624/.
- [Yac03] Yacoub, S., et al., *Pattern-Oriented Analysis and Design*, Addison-Wesley, 2003.
- [Yah13] Yahoo Developer Network, Yahoo! Design Pattern Library, 2013, disponível em <http://developer.yahoo.com/ypatterns/>.
- [Yau11] Yau, S., and H. An, "Software Engineering Meets Services and Cloud Computing," *IEEE Computer*, vol. 44, no. 10, October 2011, pp. 47–53.
- [You01] Young, R., *Effective Requirements Practices*, Addison-Wesley, 2001.
- [You75] Yourdon, E., *Techniques of Program Structure and Design*, Prentice Hall, 1975.
- [You95] Yourdon, E., "When Good Enough Is Best," *IEEE Software*, vol. 12, no. 3, May 1995, pp. 79–81.
- [Yau02] Yaun, M., "Best Tools for Mobile Application Development," *Java World*, disponível em www.javaworld.com/javaworld/jw-10-2002/jw-1018-wireless.html.
- [Zah90] Zahniser, R., "Building Software in Groups," *American Programmer*, vol. 3, no. 7–8, July–August 1990.
- [Zah94] Zahniser, R., "Timeboxing for Top Team Performance," *Software Development*, March 1994, pp. 35–38.

- [Zha05] Zhang, W., and S. Jarzabek, "Reuse without Compromising Performance: Industrial Experience from RPG Software Product Line for Mobile Devices," *Proceedings of 9th Software Product Line Conference*, September 2005, pp. 57–69.
- [Zim11] Zimmermann, O., "Architectural Decisions as Reusable Design Assets," *IEEE Software*, vol. 28, no. 1, January–February 2011, pp. 64–69.
- [Zul92] Zultner, R., "Quality Function Deployment for Software: Satisfying Customers," *American Programmer*, February 1992, pp. 28–41.
- [Zus90] Zuse, H., *Software Complexity: Measures and Methods*, DeGruyter, 1990.
- [Zus97] Zuse, H., *A Framework of Software Measurement*, DeGruyter, 1997.

Índice

- Aberto-fechado, princípio, 292
- Abstração, 108, 231-232
- Ação, 16
- Acessibilidade, 336
- Acompanhamento de dependência, 631
- Acoplamento, 298
 - classe, 898
- Agilidade,
 - custo da alteração, 69
 - definição de, 68
 - princípios, 70
- Agilidade arquitetural, 280
- Agregação, 873
- Aleatório, teste, 532
- Além de fronteiras, teste (*ver também* Internacionalização), 578
- Alfa, teste, 485
- Aliança ágil, 66
- Ameaça,
 - análise, 585
 - modelagem, 594
 - probabilidade, 591
- Amostragem, modelo de, 612
- Amostragem, revisões por, 444
- Análise,
 - ambiente de trabalho, 331
 - ameaça, 585
 - conteúdo exibido, 331
 - defeitos, 450
 - interface do usuário, 325
 - regras práticas, 168-169
 - tarefas, 326
 - usuário, 325
- Análise, classes de,
 - características das, 187
 - diagramas de estado, 204
 - identificação, 185
- Análise, modelo de,
 - construção, 154
 - elementos de, 172
- Análise, pacotes de, 199
- Análise de viabilidade, aplicativos móveis, 571
- Análise estruturada, 171
- Aperfeiçoamento de processos contínuo, 449
- Aplicações Web/aplicativos móveis, 7
- Argumentos favoráveis às métricas, 720
- Arquétipos, 269
- Arquitetura, 118, 253
 - alternativas, 274
 - centralizada em dados, 258-259
 - chamadas e retornos, 260
 - definição de, 253
 - divergência, 311
 - estilos, 258-259
 - fluxo de dados, 259
 - gêneros, 257
 - importância da, 254
 - orientada a serviço, 266
 - padrões, 278
 - projeto para WebApp, 381
 - propriedades estruturais, 232-233
 - prós e contras, 275
 - template de descrição, 257
 - verificação de conformidade, 279
- Artefatos, teste de integração, 483
- Árvore de dados, 217
- Árvore de decisões, 749
- Ascendente (*bottom-up*),
 - integração, 477
- Aspectos, 54-55, 237
- Associações, 198, 872
- Ataque, 585
- Ataque, padrões de, 589
- Atividades de apoio, 17
- Ativo, 585
- Ativo, valor do, 591
- Atores, 149
- Atributos, 188, 892-893
 - métricas, 654, 657
 - qualidade, 455
- Auditorias, qualidade de software, 450
- Autoadaptativos, sistemas, 158
- Auto-organizadas, equipes, 691-693
- Avaliação, riscos, 777
- Avaliações *post-mortem* (PME), 445
- Banco de dados, teste de, 546-547
- Beta, teste, 485
- Big bang, teste de integração,
 - abordagem, 476
- Blocos básicos, 848
- Bootstrap, framework SPI, 832-833
- Bugs, (*ver também* Falhas, Defeitos), 432
- Caixa de estado, 604
 - especificação de, 606
- Caixa-branca, teste, 500
- Caixa-clara, 605
 - especificação, 607
- Caixa-preta, 604
 - especificação, 605
 - teste, 509
- Caminho básico, teste do, 500
- Caminhos de programa independentes, 502
- Campos de aplicação, 6
- Capacidade de sobrevivência, 591
- Carga, teste de, 562
- CasaSegura, 143, 150
 - acoplamento em ação, 298
 - análise de domínio, 171
 - análise sintática, 186
 - aplicação de padrões, 362
 - árvore de dados, 217
 - avaliação de arquitetura, 276
 - casos de uso para projeto de UI, 327
 - cenário preliminar de usuário, 174
 - cenários de usuário, 147
 - classes de projeto, 241
 - coesão em ação, 297
 - como tudo começa, 26
 - conceitos de projeto, 239
 - conclusão, 862-863
 - considerações sobre processo ágil, 76
 - contexto de arquitetura,
 - diagrama, 269
 - decisões de arquitetura, 265
 - design gráfico, 377
 - diagrama de atividades, 220
 - diagrama de caso de uso, 153
 - diagrama de estado, 205
 - diagrama de raias, 180-181
 - diagrama de sequência, 206
 - erros de comunicação, 111
 - escolha de uma arquitetura, 262
 - estruturas da equipe, 93
 - instâncias de, 272
 - layout da tela, 334
 - levantamento dos requisitos, 145
 - modelagem comportamental, 157
 - modelo de casos de uso, 178
 - modelos CRC, 197
 - modelos de classes, 190
 - negociação, 160
 - padrões de análise, 209
 - padrões de projeto, 356
 - princípio aberto-fechado (OCP), 292-293
 - projeto *versus* código, 227
 - questões de qualidade, 424
 - regras de ouro da interface, 320
 - requisitos de aplicativos móveis, 396
 - revisão do projeto de interface, 340

- seleção de um modelo, 46-47, 49-50
- Cascata, modelo, 41
 - problemas com, 42
- Caso de teste, derivação de, 504
- Caso de teste, projeto de, 515
- Casos de uso, 146, 326, 875
 - criação, 173
 - desenvolvimento, 149
 - diagrama, 153, 179, 875
 - estimativa, 740
 - eventos, 203
 - ferramentas, 154
 - formal, 177
 - métricas, 714
 - pergunta a ser respondida, 150
 - refinamento dos, 176
- Cenários de uso, 146
- Centrada no usuário, engenharia de segurança, 589
- Certificação, 612
 - equipes, 611
 - modelo, 612
 - teste, 488-489
 - teste para aplicativos móveis, 570
- Ciclo, teste de, 507
- Ciclo de excelência, 843
- Ciclo de vida clássico, 42
- Ciclo rápido, teste de, 473
- CK, conjunto de métricas, 667
- Classe-Responsabilidade-Colaborador (CRC), 192
 - cartões, 74
- Classes (*ver também* Objeto), 892-893
 - acoplamento, 898
 - agregação, 196
 - análise, 185
 - atributos, 188
 - características, 897
 - coesão, 898
 - controlador, 894
 - diagramas de, 156, 191, 870
 - fronteiras, 892-893
 - interface do usuário, 897
 - negócio, 897
 - operações (*ver também* Métodos), 189
 - persistente, 897
 - processo, 897
 - projeto, 239
 - teste, 481, 528
- Clientes, 111
 - mitos sobre, 24
- CMM, 37-38
- CMMI, 37-38, 828
- COCOMO II, modelo, 744
- Codificação, princípios de, 122
- Código, qualidade do, 454
- Código, reestruturação de, 809
- Código, revisões, 433
- Código aberto, 848
- Código-fonte, métricas de, 675
- Coesão, 296, 898
- Colaboração, 140, 195
 - diagrama, 880
 - ferramentas, 98
- Combate ao incêndio, modo de, 778
- Compatibilidade de dispositivo, teste de, 483
- Complexidade,
 - elementos, 626
 - gestão da, 845
- Complexidade ciclomática, 503
- Componente, modelo de, 612
- Componentes,
 - adaptação, 310
 - baseados em classe, 291
 - classificação, 312
 - composição, 310
 - definição de, 286
 - diretrizes para o projeto, 295
 - elaboração, 287
 - projeto de, 290
 - projeto para WebApp, 387
 - recuperação, 312
 - visão orientada a objetos, 286
 - visão relacionada a processo, 291
 - visão tradicional, 288
- Componentes arquiteturais, 270
- Composição, 873
- Computação na nuvem, 10, 97, 405
- Comunicação, atividade de, 17
 - conjunto de tarefas, 695-696
 - diagrama, 880
 - equipe, 691-693
 - princípios da, 110
- Concepção, 133
- Conclusão (do teste), 484
- Concorrentes, modelos 48-49
- Condição, teste de, 507
- Conectividade, teste de, 483
- Confiabilidade,
 - medida, 459
 - software, 459
- Confiabilidade, 591
- Confiabilidade, verificação de, 591
- Configuração, auditoria de, 639
- Configuração, gestão de, 626
- Configuração, modelos de, 220
- Configuração, objetos de, 642
- Configuração, revisão de, 484
- Configuração, teste de, 558
- Conjunto, teste de, 482, 529
- Conjunto de tarefas, comunicação do, 695-696
- Consistência, 526
- Construção, atividade de, 17
 - princípios, 121
- Conteúdo,
 - ferramentas de teste, 645
 - gestão de, 643
 - modelo, 216
 - objetos, 379
 - repositório, 630
 - teste, 544-545
- Contexto, 693-694
- Controladora, classe, 894
- Controle de alterações, 626
 - processo, 636
- Controle de versão, 637
- Coordenação, equipe, 691-693
- Coordenação e comunicação, 691-693
- Correção, prova de, 609
- Correção, verificação de (*ver também* Projeto, verificação de), 608
 - modelo OOA, 525
 - segurança, 591
- CRC, revisão do modelo, 527
- Credibilidade, 591
- Custos de avaliação, 422
- Dados históricos, 659
- Decisão fazer/comprar, 748
- Decisões de arquitetura, 256, 266
- Decomposição,
 - problema da, 693-694
- Defeitos (*ver também* Bugs, Falhas), 432
- Defeitos, amplificação de, 433
- Defeitos, análise de, 450
- Defeitos, curva de,
 - hardware, 5
 - software, 6
- Dependências, 198, 872
- acompanhamento, 631
 - gestão de, 628-629
 - inversão, 241
- Depuração, 488-489
 - automática, 492-493
 - considerações psicológicas, 490-492
 - estratégias, 490-492
 - táticas, 490-492
- Depuração automática, 490-492
- Descendente (*top-down*),
 - integração, 476
- Descrições de arquitetura, 255
- Desempenho, teste de, 487-488, 580
- Desenvolvimento ágil (*ver também* Processo ágil), 71
 - políticas de, 71
- Desenvolvimento baseado em componentes, 52-53, 308
- Desenvolvimento baseado em teste, 854
- Desenvolvimento colaborativo, 852
- Desenvolvimento de software orientado a aspectos, 53-54
- Design gráfico, 378
- Diagrama,
 - atividade, 869
 - casos de uso, 875
 - classe, 870
 - colaboração, 880
 - comunicação, 880
 - disponibilização, 874
 - estado, 884
 - sequência, 876
- Diagrama de atividade, 155, 179-180, 303, 869
- Diagrama de contexto arquitetural (ACD), 267
- Diagramas de sequência, 205, 876
- Disparador, 178
- Disponibilização, atividades de, 17, 124-125
 - diagrama, 874
 - teste, 487-488

- Documentos,
 reestruturação, 804
 teste, 517
- Domínio, análise de, 169-170
- Domínio, engenharia de, 308
- Domínio de negócio, classes de, 897
- Eficácia de custos, 436
- Eficiência na remoção de defeitos (DRE), 718
- Elaboração, 135, 237
 componente, 287
 tarefa, 327
- Elementos humanos, 626
- Eliminação de causa, 490-492
- Encapsulamento, 892-893
- Encapsulamento de informações, 234-235
- Engenharia,
 direta, 811
 reversa, 805
 segurança, 588
- Engenharia de segurança, 588
 centrada no usuário, 589
 ferramentas, 598
- Engenharia de software,
 baseada em componentes, 279
 camadas de, 16
 definição de, 15
 desafios, 851
 ética, 865
 games, 1
 impacto da mídia social, 95
 prática, 19
 práticas de trabalho, 125-126
 princípios, 20-21, 104
 princípios fundamentais, 106
 projeto, 225
 psicologia, 89
 rumos da tecnologia, 849
 tendências, 839
 usando a nuvem, 97
 visão no longo prazo, 864
- Engenharia de software baseada em componentes (CBSE), 308
- Engenharia direta, 811
- Engenharia reversa, 805
 dados, 807
 ferramentas, 809
 interfaces do usuário, 808
 processamento, 807
- Engenheiro de software,
 características do, 88
 funções, 89
- Envolvido, 687
 definição de, 17
 identificação, 139
- Equipes, 90
 ágeis, 93, 690-691
 auto-organizadas, 94, 691-693
 código aberto, 688-689
 consistentes, 90
 estruturas de, 92
 globais, 99
 mix de talentos, 847
 paradigma aleatório, 688-689
 paradigma fechado, 688-689
 paradigma sincronizado, 689-690
 paradigmas organizacionais, 92
 toxicidade, 91
 XP, 94
- Equipes ágeis, 690-691
- Erros, 432
 ambiente WebApp, 541-542
 correção, 492-493
 custo de, 423
 densidade de, 435
- Escopo, 113, 134
 software, 693-694
- Especificação, 135
 caixa de estado, 606
 caixa-clara, 607
 caixa-preta, 605
 estrutura de caixa, 604
 métricas de qualidade, 663
- Especificação de estrutura de caixa, 604
- Específico de produto, risco, 780
- Espiral, modelo, 46-47
- Estado, diagrama de, 884
- Estado finito, modelagem de, 511
- Estados, diagrama de, 304
- Estados, representações de, 204
- Estatística da garantia de qualidade, 455-456
- Estimativa,
 baseada em FP, 738
 baseada em problemas, 735
 baseada em processo, 739
 caso de uso, 740
 conceitos, 750
 desenvolvimento ágil, 746
 modelos empíricos, 743
 projetos orientados a objeto, 746
 reconciliação, 742
 risco (*ver também* Risco, projeção de), 782
 software, 727
 técnicas de decomposição, 734
 WebApp, 747
- Estimativa de custos, ferramentas de, 748
- Estratégia,
 depuração, 490-492
 teste, 471
- Estratégias de teste para software convencional, 475
- Estrutura, diagrama de, 289
- Estrutura de controle, teste de, 507
- Evitando problemas de gestão, 696-697
- Exceções, 178
- Expectativa de perda anual (ALE), 594
- Experiência do usuário, teste de, 483
- Exposição ao risco, 786
- Falhas (*ver também* Bugs, Defeitos), 432
- Falhas, custos de, 422
- Fatores de ajuste do valor (VAF), 660
- Ferramentas,
 casos de uso, 154
- CBSE, 313
- colaboração, 98
- conformidade da arquitetura, 280
- controle de versão, 635
- desenvolvimento de interface do usuário, 337
- engenharia de requisitos, 138
- engenharia de segurança, 598
- engenharia reversa, 809
- estimativa de custos, 748
- estimativa de esforço, 748
- estimativa de mão de obra e custo, 748
- extração de teste, 614
- ferramentas de navegação Web, 556-557
- geração de teste, 614
- gestão de alterações para WebApp, 647
- gestão de conteúdo, 645
- gestão de projeto, 698-699
- gestão de riscos, 791
- integração de teste manual, 614
- linguagens de descrição da arquitetura, 277
- métodos formais, 614
- métricas de produto, 678
- métricas de projeto e processo, 716
- métricas para WebApp, 675
- modelagem de processos, 61-62
- projeto arquitetural, 273
- reengenharia de processos de negócio, 801
- reestruturação de software, 810
- SCM, 640
- teste de aplicativo móvel, 580
- teste de configuração Web, 559
- teste de conteúdo Web, 546-547
- teste de desempenho Web, 563
- teste de interface do usuário, 553-554
- teste de segurança Web, 560
- Fluxo de dados, modelagem de 511
- Fluxo de dados, teste de, 507
- Fluxo de trabalho, análise de, 328
- Fluxo de transação, modelagem de, 511
- Força bruta, depuração, 490-492
- Formulário de informações de risco (RIS), 790
- Frames de interação, 878
- Framework, 17, 291, 351-352
 atividades, 16, 17, 32
 SPI, 819
- Fronteira, classes de, 892-893
- Funcional, decomposição, 693-694
- Funcional, independência, 235-236
- Funcional, modelo, 218
- Funcional, teste, 509
- Garantia da qualidade de software (SQA), 448
 elementos de, 450
- Garantia da segurança, 592
- Generalização, 871
- Gêneros, 257

- Gerenciamento de mudanças, 451, 631
- Gestão,
 - espectro, 685
 - mitos, 23
 - projeto, 684
 - riscos, 451, 777
 - segurança, 451
- Gestão de configuração de software (SCM), 623
 - aplicativos móveis, 640
 - ferramentas, 640
 - padrões, 649
 - processo, 632
 - repositório, 630
 - WebApps, 640
- Gestão de impacto, 638
 - riscos, 785
- Gestão de impacto adiante, 638
- Gestão de impacto retroativo, 639
- Gestos, teste de, 575
- Grafo de fluxo, 500
- Grafos, teste baseado em, 509
- Grupo de teste independente (ITG), 475
- Herança, 894
- Horda mongol, conceito de, 24
- Identificação, 633
 - riscos, 780
- Impacto do risco, 785
- Implantação, projeto de, 247-248
- Implantação da função de qualidade (QFD), 146
- Incerteza, 779
- Incrementos, 44
- Indicador, 655
- Informação, 3
 - objetivos, 693-694
 - representação, 862-863
- Inspeções, 432, 437
- Integração,
 - ascendente (*bottom-up*), 481
 - descendente (*top-down*), 473
- Integração, teste de, 475
 - artefatos, 480
 - orientado a objeto, 529
- Integração de teste manual, ferramentas de, 614
- Integração do problema com o processo, 694-695
- Inteligência artificial, software de, 7
- Interface, análise de, 325
- Interface do usuário, classes de, 897
- Interface do usuário, métricas de projeto de, 672
- Interface do usuário, projeto de (*ver* Projeto de interface)
- Interface do usuário, teste da, 548-549
- Internacionalização, 336, 578
- Internet das Coisas, 588
- Invariante, 616
- Inventário, análise de, 803
- ISO 9001:2000, 37-38
- ISO 9001:2008, 462
- Itens de configuração de software (SCI), 628-629
- Lacunas, análise de, 823
- Layout, 378
- Levantamento, 134
 - ágil, 148
 - artefatos, 147
- Líder de equipe, 688
- Linguagem,
 - especificação formal, 900
 - especificação Z, 904
 - Object Constraint Language, 886-887
 - semântica, 901
 - sintaxe, 901
- Linguagem de descrição de arquitetura (ADL), 232-233, 276
- Linguagem de especificação formal, 900
- Linha de produtos, software para, 7, 11
- Listas de controle,
 - aplicativos móveis, 570
 - item de risco, 781
 - revisão, 439
- LOC, métricas baseadas em, 712
- Lorenz e Kidd, métricas OO, 671
- Manifesto ágil, 66
- Manutenção, 108
 - software, 796
- Manutenibilidade, 797
- Matriz ortogonal, teste, 513
- Matrizes de grafos, 506
- Maturidade,
 - modelos de, 821
 - nível de, 831
- Medição, 654, 708
 - princípios da, 656
- Medidas, 654
 - confiabilidade, 460
 - diretas, 708
 - disponibilidade, 460, 461
- Melhora do processo de software (SPI) (*ver* Melhoria do processo)
- Melhoria estatística do processo de software (SSPI), 707
- Mensagens, 895
- Meta/Questão/Métrica (GQM), 656
- Método de Desenvolvimento de Sistemas Dinâmicos (DSDM), 78-79
- Metodologia, 17, 291, 351-352
- Métodos (*ver também* Operações), 16, 892-893
- Métodos formais, 52-53, 602
 - ferramentas, 614
- Métricas, 654
 - aplicativos móveis, projeto de, 673
 - argumentos, 720
 - atributos, 654, 657
 - baseadas em função, 659, 710
 - baseadas em LOC, 712
 - caso de uso, 714
 - código-fonte, 675
 - coleta, 721
 - empresas pequenas, 721
 - estabelecimento de um programa, 722
 - metas de negócio, 724
 - modelo de requisitos, 659
 - nível de componente, 671
 - orientadas a objetos, 666, 713
 - orientadas a tamanho, 709
 - orientadas a classes, 667
 - privadas, 706
 - processo, 704
 - produtividade, 712
 - projeto, 663, 707
 - projeto arquitetural, 663
 - projeto de interface do usuário, 672
 - projeto para WebApp, 673
 - públicas, 706
 - qualidade, 455-456
 - qualidade de especificação, 663
 - qualidade de software, 716
 - referencial, 720
 - revisões, 534
 - teste, 676
 - WebApp, 714
- Middleware, 405
- Mídia social, 95
- Mídia social e privacidade, 587
- Mitigação, Monitoramento e Gestão de Riscos (RMMM), 788, 790
- Mitos, 32
 - cliente, 24
 - gestão, 23
 - profissional, 25
- MobileApps (aplicativos móveis), 9
 - atividades de desenvolvimento, 395
 - checklist, 570
 - checklist da qualidade, 397
 - desafios de projeto, 392
 - diretrizes de teste, 568
 - engenharia de software, 407
 - erros de projeto, 401
 - estratégias de teste, 569
 - ferramentas, 404
 - ferramentas de teste, 579
 - matriz de teste, 572
 - melhores práticas de projeto, 401
 - métricas de projeto, 673
 - modelagem de requisitos, 214
 - padrões, 366
 - projeto, 391
 - projeto arquitetural de, 274
 - projeto de interface, 398
 - projeto de nível de componentes, 305-306
 - SCM, 640
 - sensível ao contexto, 398-399
 - teste, 483
 - teste de esforço (stress), 573
 - teste de usabilidade, 575
 - teste em tempo real, 578
- Mobilidade, ambientes de, 403
- Modelagem, 17
 - ágil, 79-80
 - ameaça, 594
 - baseada em cenário, 173

- CRC, 192
- estado finito, 511
- fluxo de dados, 511
- no tempo, 511
- princípios, 114
- segurança, 590
- transação, 511
- Modelagem ágil, princípios de, 81
- Modelagem de processos, ferramentas de, 61-62
- Modelo,
 - amostragem, 612
 - certificação, 612
 - COCOMO II, 744
 - componente, 612
 - segurança, 590
- Modelo, desenvolvimento de software controlado por, 853
- Modelo, teste baseado em, 514
- Modelos,
 - comportamentais, 203
 - orientados a objetos, 525
 - projeto, 242-243
 - requisitos, 167
- Modelos empíricos,
 - estimativa, 743
- Modelos vivos, 120
- Model-View-Controller (MVC), 384
- Modularidade, 233-234
- Módulo (*ver também* Componente), 288
- MOI, modelo, de liderança, 688
- MOOD, conjunto de métricas, 670
- Mudanças, tipos de, 8
- Múltiplas classes, teste de, 534
- Multiplicidade, 872
- Navegação,
 - semântica, 385
 - sintaxe, 387
 - unidade semântica, 387
- Navegação, modelo de, 220
- Navegação, teste de, 555-556
- Negligência, 425-426
- Negociação, 135, 159
- Negócio, processo de,
 - ferramentas, 801
 - reengenharia, 799
- Nível de componente, 671
 - teste em, 554-555
- Object Constraint Language (OCL), 886-887, 901
 - exemplo de, 903
 - notação, 902
- Objetivos da informação, 693-694
- Objeto (*ver também* Classe), 892-893
- Objetos agregados, 633
- OO, métricas, Lorenz e Kidd, 671
- Operações (*ver também* Métodos), 189, 892-893
- Orientado(a) a objetos,
 - análise, 172, 184
 - consistência do modelo, 526
 - estimativa de projeto, 746
 - estratégias de teste, 528
 - métricas, 713
 - métricas de projeto, 666
 - modelos, 525
 - OOA, exatidão do modelo, 525
 - OOD, exatidão do modelo, 525
 - projeto de caso de teste, 530
 - software, 481
 - teste de integração, 529
 - teste de unidade, 528
 - teste de validação, 529
- Padrões,
 - ISO 9001:2008, 462
 - qualidade de software, 450
 - SCM, 649
- Padrões, 109
 - análise, 157, 207
 - aplicativos móveis, 366
 - arquiteturais, 263, 278, 359
 - contexto, 348
 - descrição, 352
 - exemplo de, 209
 - linguagens, 353
 - modelagem de requisitos, 207
 - nível de componente, 360
 - processo, 34-35
 - projeto, 348
 - projeto de interface, 362
 - repositório, 353, 360
 - templates para, 352
 - teste, 519
 - tipos de, 348-349
 - WebApps, 364
- Padrões baseados em componentes, 291
- Paradigma aberto, equipe de, 688-689
- Paradigma fechado, equipe de, 688-689
- Paradigma randômico, equipe de, 688-689
- Paradigma sincronizado, equipe de, 689-690
- Paradigmas organizacionais, 92
- Partição, 693-694
- Partição, teste de, 533
- Particionamento de equivalência, 511
- People Capability Maturity Model, 685, 831-832
- Perigos, 463, 790
- Persistentes, classes, 897
- Pessoas, 687
- Phishing, 587
- Pistas de auditoria, 632
- Planejamento, atividade de, 17
 - princípios, 112
 - teste Web, 542-543
- Plano de contingência, 789
- Plano de SQA, 453, 463
- Plano RMMM, 790
- Polimorfismo, 896
- Ponto de função (FP), 659, 710
 - estimativa, 738
- Pontos de Caso de Uso (UCPs), 742
- Pontos de conexão, 351-352
- Pontos de prioridade, 140
- Pontos de vista, vários, 139
- Pós-condições, 616
- Post-mortem*, avaliações (PME), 445
- Prática (engenharia de software),
 - essência da, 19
 - princípios fundamentais, 108
- Práticas vitais, 698-699
- Precondições, 178, 616
- Preocupações transversais, 53-54, 237, 589
- Prevenção, custos de, 422
- Previsíveis, riscos, 779
- Primeiro-em-profundidade,
 - integração, 480
- Privacidade, 586
 - computação na nuvem, 587
 - mídia social, 587
- Privadas, métricas, 706
- Proativas, estratégias de risco, 778
- Problema, decomposição do, 693-694
- Problema, elaboração do, 693-694
- Problema, estimativa baseada em, 735
- Processo, 16, 686
 - adaptação, 18
 - ágil, 69
 - avaliação, 36-37
 - classes, 897
 - componentes do, 16
 - decomposição, 693-694
 - depuração, 490-491
 - elementos, 626
 - fluxo, 31, 33
 - genérico, 31
 - imaturidade, 822
 - iterativo, 31
 - metodologia, 17, 32
 - modelo de padrão, 34-35
 - padrões, 34-35, 36-37
 - princípios, 106
 - programação extrema (XP), 72
 - relação com produto, 61-62
 - SCM, 632
- Processo, estimativa baseada em, 339
- Processo, maturidade do, 821
- Processo, melhoria do, 36-37, 818
 - abordagens, 819
 - análise de lacunas, 823
 - avaliação, 827
 - contínuo, 449
 - educação, 825
 - etapas, 823
 - instalação, 826
 - justificação, 825
 - outros frameworks, 831-832
 - ROI, 833-834
 - tendências, 834-835
- Processo, métricas do, 704
- Processo, modelos de, 40
 - concorrentes, 48-51
 - dirigido a riscos, 47-48
 - especializado, 51-52
 - evolucionário, 44-45
 - incremental, 43
 - prescritivo, 41

- reengenharia de software, 803
- sala limpa, 603
- Processo, tecnologia do, 60-61
- Processo ágil, 70
- Processo de Software de Equipe (TSP), 59-60, 832-833
- Processo de Software Pessoal (PSP), 58-59, 832-833
- Processo Unificado, 54-55
 - ágil, 82
 - fases, 55-56
 - fluxo de trabalho, 57-58
 - história do, 55-56
- Produtividade, medidas de, 712
- Produtividade, métricas de, 712
- Produto, 686, 692-693
- Produto, métricas de, 653
 - ferramentas, 678
- Produtor, 441
- Programação em pares, 75-76, 439-440
- Programação Extrema (XP), 72
 - atividades, 72
 - equipe, 94
 - industrial, 75-76
 - teste, 75-76
- Programador-chefe, equipe com, 92
- Projeção, risco (*ver também* Risco, estimativa de), 782
- Projeto, 686, 692-693
 - arquitetural, 267
 - baseado em cenário, 532
 - baseado em padrões, 347, 354
 - caso de teste, 242, 515
 - classes, 239
 - componentes tradicionais, 306-307
 - conceitos, 230-231
 - conceitos de orientação a objetos, 238
 - conjunto de tarefas, 230-231
 - conteúdo, 379
 - dados, 243-244
 - diretrizes de qualidade, 228, 454
 - elementos arquiteturais, 243-244
 - estético, 377
 - evolução do, 229-230
 - formal, 603
 - granularidade, 365
 - interfaces, 244-245, 317
 - métricas, 663
 - modelo, 226
 - navegação, 385
 - nível de componente, 246-247, 290, 299
 - nível de disponibilização, 247-248
 - pós-moderno, 854
 - processo, 228
 - refatoração, 74
 - refinamento, 237, 608
 - reutilização, 312
 - revisões técnicas, 228-229
 - sala limpa, 607
 - verificação, 608
 - WebApps, 371
- Projeto, banco de dados do, 627
- Projeto, complexidade do, 728
 - Projeto, gestão do, 684
 - ferramentas, 698-699
 - Projeto, planejamento do, 729
 - conjunto de tarefas, 730
 - processo, 729
 - Projeto, risco de, 779
 - Projeto, tamanho do, 729
 - Projeto, velocidade de, 73
 - Projeto arquitetural, ferramentas de, 273
 - Projeto arquitetural, métricas do, 663
 - Projeto de caso de teste entre classes, 534
 - Projeto de interface, 244-245, 317
 - aplicativos móveis, 340-341, 398-399
 - avaliação de, 341-342
 - erros, 335
 - etapas, 332
 - modelos, 323
 - padrões, 362
 - processo, 323
 - regras de ouro, 318
 - WebApp, 337, 376
 - Projetos, como tudo começa, 26
 - Prototipação, 44-45
 - problemas com, 45-46
 - Pseudocontrolado, 486-487
 - Pseudocontroladores, 468-469
 - Públicas, métricas, 706
- Qualidade,
 - ações administrativas, 426-427
 - aplicativos móveis, 397
 - árvore de requisitos, 373
 - atributos, 455
 - "bom o suficiente", 421
 - código, 455
 - conceitos, 412
 - custo de, 422
 - definição de, 413
 - fatores de McCall, 416
 - Garvin, dimensões de, 415
 - ISO 9126, 418
 - métodos para alcançar, 427-428
 - métricas, 455
 - projeto, 454
 - projeto para WebApp, 372
 - requisitos, 454
 - segurança, 425-426
 - visão quantitativa, 420
- Qualidade, controle da, 427-428
- Qualidade, garantia da, 428-429
 - estatística, 455-456
- Qualidade, gestão da, 449
 - recursos, 452
- Raias, diagrama de, 180-181, 330
- Rastreabilidade, 118, 142
 - matriz, 142, 628-629
- Rastreamento (*backtracking*), depuração, 490-492
- Reativas, estratégias de risco, 778
- Reconciliação,
 - estimativa, 742
 - métricas LOC e FP, 711
- Recuperação, teste, 486-487
- Recursos, 731
 - ambientais, 732
 - gestão da qualidade, 452
 - humanos, 731
 - software reutilizável, 732
- Recursos ambientais, 732
- Recursos de ajuda, teste de, 516
- Recursos de software reutilizáveis, 732
- Recursos humanos, 731
- Reengenharia,
 - aspectos econômicos, 813
 - processo de negócio, 799
 - software, 802
- Reengenharia de software, 802
 - modelo de processo, 803
- Reestruturação, 809
 - código, 809
 - dados, 810
 - documentos, 804
- Reestruturação de dados, 810
- Refatoração, 74, 238, 301
- Referenciais, 626, 720
- Refinamento (*ver também* Elaboração), 237
- Refinamento horizontal, 613
- Refinamento vertical, 613
- Registrador, 442
- Registro pendente de trabalhos (*backlog*), 78-79
- Registros, manutenção dos, 442
- Regressão, teste de, 478
- Relatório de status, configuração, 639
- Repositório, 630
 - conteúdo, 630
 - hipermídia, 365
 - padrões de projeto, 360
 - SCM, 630
- Requisitos,
 - controle de, 631
 - emergentes, 846
 - entendendo, 131
 - não funcionais, 141
 - negociação de, 159
 - qualidade dos, 454
 - segurança, 585
 - validação, 161
- Requisitos, coleta dos, colaborativo, 143
- Requisitos, engenharia de, 132, 852
 - ágil, 158
 - erros comuns, 162
 - ferramentas, 138
 - orientada a metas, 134
 - questões iniciais, 140
- Requisitos, especificação dos, templates para, 136
- Requisitos, levantamento de, segurança, 589
- Requisitos, modelagem de,
 - abordagens a, 171
 - aplicativos móveis, 213
 - objetivos, 167-168
 - princípios, 116
 - WebApps, 213
- Requisitos, modelos de (*ver também* Análise, modelos de), 114
 - métricas, 659
 - tipos de, 167

- Requisitos, tarefas de,
 - concepção, 133
 - elaboração, 135
 - especificação, 135
 - levantamento, 134, 142
 - negociação, 135
 - validação, 136
- Responsabilidade civil, 425-426
- Responsabilidades, 193
- Reuniões,
 - informais, 439
 - revisão, 441
- Reutilização, 312
- Revisão,
 - arquitetural, 277
 - checklist, 439
 - código, 433
 - configuração, 488-489
 - diretrizes, 442
 - informal, 439
 - líder, 441
 - lista de problemas, 442
 - métricas, 435
 - relatório, 442
 - reuniões, 441
 - partitória, 432
 - por amostragem, 444
 - qualidade de software, 450
 - técnica, 441
- Revisões técnicas, modelo de referência para, 438
- Revisões técnicas formais (FTR), 432, 441
- Risco,
 - avaliação, 777
 - categorias, 779
 - checklist de itens, 781
 - componente, 782
 - estimativa (*ver também* Projeção), 782
 - exposição, 786
 - fatores, 782
 - formulário de informações, 790
 - identificação, 780
 - impacto, 785
 - padrões, 781
 - projeção (*ver também* Estimativa), 782
 - refinamento, 787
- Risco, estratégias de,
 - proativas, 778
 - reativas, 778
- Risco, tabela de, 783
- Riscos, 424
 - conhecidos, 779
 - específicos de produto, 780
 - imprevisíveis, 779
 - negócio, 779
 - previsíveis, 779
 - projeto, 779
 - técnicos, 779
- Riscos, gestão de, 777
 - ferramentas, 791
 - princípios, 780
 - SPI, 827
- Riscos de negócio, 779
- Sala limpa (*clean room*), projeto 607
 - modelo de processo, 603
- teste (*ver também* Teste de uso estatístico), 610
- Scrum, 78
 - reuniões, 78-79
- Segurança,
 - análise de requisitos, 585, 589
 - caso, 591
 - fatores de qualidade, 425-426
 - garantia da, 592
 - gestão, 451
 - modelo, 590
 - objetivos, 590
 - preocupações transversais, 589
 - software, 460
 - verificações de correção, 591
- Segurança, caso de garantia (*ver também* Credibilidade), 592
- Segurança e privacidade, 586
- Seis Sigma, 458
- Semântica, linguagem, 901
- Sensíveis ao contexto, aplicativos, 398-399
- Separação de interesses, 108, 233-234
- Sequência de execução (*thread*), teste baseado em, 482, 529
- Serviços, 892-893
- Serviços, métodos orientados a, 148
- Sincronização, controle de, 637
- Sintaxe, 901
- Sistema, classes de, 897
- Sistema, software de, 6
- Sistema, teste de, 486-487
- Sistema, vulnerabilidade do, 591
- Sistema de forças, 348
- Software, arquitetura de (*ver* Arquitetura)
- Software, auditoria de configuração de, 639
- Software, campos de aplicação, 6
 - “bom o suficiente”, 421
 - como capital, 30
 - definição de, 4
 - importância do, 861
 - natureza do, 3
 - orientado a objeto, 487-488
 - questões-chave, 4
 - realidades, 14
- Software, componente de (*ver* Componente)
- Software, confiabilidade do, 459
- Software, defeitos de, impacto dos, 432
- Software, desenvolvimento de, mitos do, 23
- Software, dimensionamento, 734
- Software, equação do, 744
- Software, equipes de (*ver também* Equipes), 688-689
- Software, escopo do (*ver também* Escopo), 693-694, 730
- Software, estimativa do projeto de, 727
- Software, ferramentas de reestruturação, 810
- Software, manutenção do (*ver também* Manutenção), 796
- Software, processo de, 16
- Software, projeto de (*ver* Projeto)
- Software, proteção do, 460
- Software, qualidade do (*ver também* Qualidade)
 - auditoria, 450
 - métricas, 716
 - padrões, 450
 - revisões, 450
- Software aberto, 846
- Software de aplicação, 7
- Software de engenharia/científico, 7
- Software embarcado, 7
- Software legado, 7
- Solução de problemas, 19
- Solução pontual, 74
- SPICE, 37-38, 832-833
- Sprints, 78-79
- SQA, grupo de, 450
- SQA, plano de, 454
- Stereotype*, 871
- Subclasse, 892-893
- Superclasse, 892-893
- Superfície de ataque, 596
- Superfície de ataque, 596
- Suportabilidade, 798
- Tabela da voz do cliente, 146
- Tabela para organização de padrões, 358
- Tamanho, métricas orientadas a, 709
- Tarefa, 16
 - análise de, 326
- Tarefas, conjunto de, 31
 - identificação de, 33-34
 - planejamento do projeto, 730
- Teclado virtual, entrada por, 577
- Técnicas automáticas de estimativa, 743
- Técnicas de decomposição, estimativa, 734
- Técnico, risco, 779
- Tecnologia, evolução da, 840
- Tempo real, teste em,
 - aplicativos móveis, 578
 - sistema, 517
- Tendências, processo, 849
 - ferramentas, 855
- Terceirização, 750
- Testabilidade, 497
- Testagem automatizada, 571
- Teste,
 - alfa, 485
 - beta, 477
 - características, 498
 - estratégias para software convencional, 473
 - ferramentas de extração, 614
 - ferramentas de geração, 614
 - matriz de, 572
 - orientados a objetos, 530
- Teste baseado em cenário, 532
- Teste baseado em falhas, 531-532
- Teste baseado em uso, 529
- Teste comportamental, 509
- Teste de esforço (*stress*), 487-488, 562
 - aplicativos móveis, 573

- Teste de mesa, 439
- Teste de uso estatístico, 604
- Teste em condições naturais, 483, 573
- Teste exaustivo, 500
- Teste fumaça, 479
- Testes,
 - aleatórios, 532
 - aplicativos móveis, 482
 - automatizados, 571
 - banco de dados, 546-547
 - baseados em cenário, 532
 - baseados em falhas, 531-532
 - baseados em grafo, 509
 - baseados em modelo, 514
 - baseados em sequência de execução (*thread*), 476, 529
 - baseados em uso, 529
 - caixa-branca, 500
 - caixa-preta, 509
 - certificação, 570
 - classe, 482, 528
 - comportamentais, 509
 - condição, 507
 - conjunto, 472, 529
 - critérios, 472
 - desempenho, 486-487
 - diretrizes, 568
 - disponibilização, 475
 - do caminho básico, 500
 - documentação, 517
 - esforço (*stress*), 475
 - estratégias, 470, 528, 542-543, 569
 - estrutura de controle, 507
 - exaustivos, 500
 - ferramentas, 579
 - fluxo de dados, 507
 - fumaça, 487-488
 - funcionais, 509
 - gestos, 575
 - integração, 481
 - loop, 507
 - matriz ortogonal, 513
 - métodos, 529, 676
 - múltiplas classes, 534
 - nível de componente, 554-555
 - orientados a objetos, 528
 - padrões, 519
 - partição, 533
 - princípios, 123
 - processo, 543-544
 - recuperação, 478
 - recursos de ajuda, 516
 - regressão, 486-487
 - sala limpa (*ver também* Teste de uso estatístico), 610
 - segurança, 479
 - sistema, 483
 - sistema em tempo real, 517
 - unidade, 467-468
 - uso estatístico, 604
 - validação, 467-468
 - WebApp, 482
- Testes de compatibilidade, 553-554
- TickIt, 833-834
 - planejamento do projeto, 730
- Toxicidade, equipe, 689-690
- Trabalho, formas de, 125-126
- UML, notação, 869
 - diagrama de atividade, 155, 179-180, 303
 - diagrama de caso de uso, 153
 - diagrama de classe, 156
 - diagrama de distribuição, 247-248
 - diagrama de estado, 156
 - diagrama de raias, 180-181
 - modelagem de análise, 207
 - representação de interface, 245-246
- Unidade, teste de, 473
 - orientado a objetos, 528
- Usabilidade, 322
 - teste, 551-552, 575
- Usuários, 111
- Usuários, histórias de, 73
- V, modelo, 43
- Validação, 136, 161, 483
 - Validação, teste de, 483
 - checklist, 137
 - orientado a objetos, 529
- Valor limite, análise de, 512
- Verificação, 470
 - confiabilidade, 591
 - correção, 608
 - projeto, 608
- Versão, controle de, 634
 - ferramentas, 635
- Versões, 631
- Volatilidade, 134
- Voz, entrada de, 576
- Voz, reconhecimento de, 576
- W⁵HH, princípio, 697-698
- Walkthroughs, 432
- Web,
 - dimensões de qualidade, 541
 - estratégias de teste, 542-543
 - ferramentas de teste da interface do usuário, 553-554
 - ferramentas de teste de configuração, 58-59
 - ferramentas de teste de desempenho, 563
 - ferramentas de teste de navegação, 556-557
 - ferramentas de teste de segurança, 560
 - planejamento de teste, 542-543
 - processo de teste, 543-544
- WebApps (aplicações Web), 7, 9
 - arquitetura, 381
 - características, 379
 - erros em um ambiente, 541-542
 - estimativa, 747
 - ferramentas de gestão de alterações, 647
 - ferramentas para métricas, 675
 - gestão de alterações, 647
 - metas de projeto, 374
 - métricas, 714
 - métricas de projeto, 673
 - modelagem de requisitos, 213
 - modelo de conteúdo, 216
 - modelo de navegação, 220
 - modelo funcional, 218
 - pirâmide de projeto, 375
 - projeto, 371
 - projeto arquitetural de, 273
 - projeto de componente, 387
 - projeto de interface, 376
 - projeto de nível de componentes, 305
 - qualidade da, 372
 - SCM, 640
 - teste, 482
- Z, linguagem de especificação, 904
 - exemplo de, 906
 - notação, 905