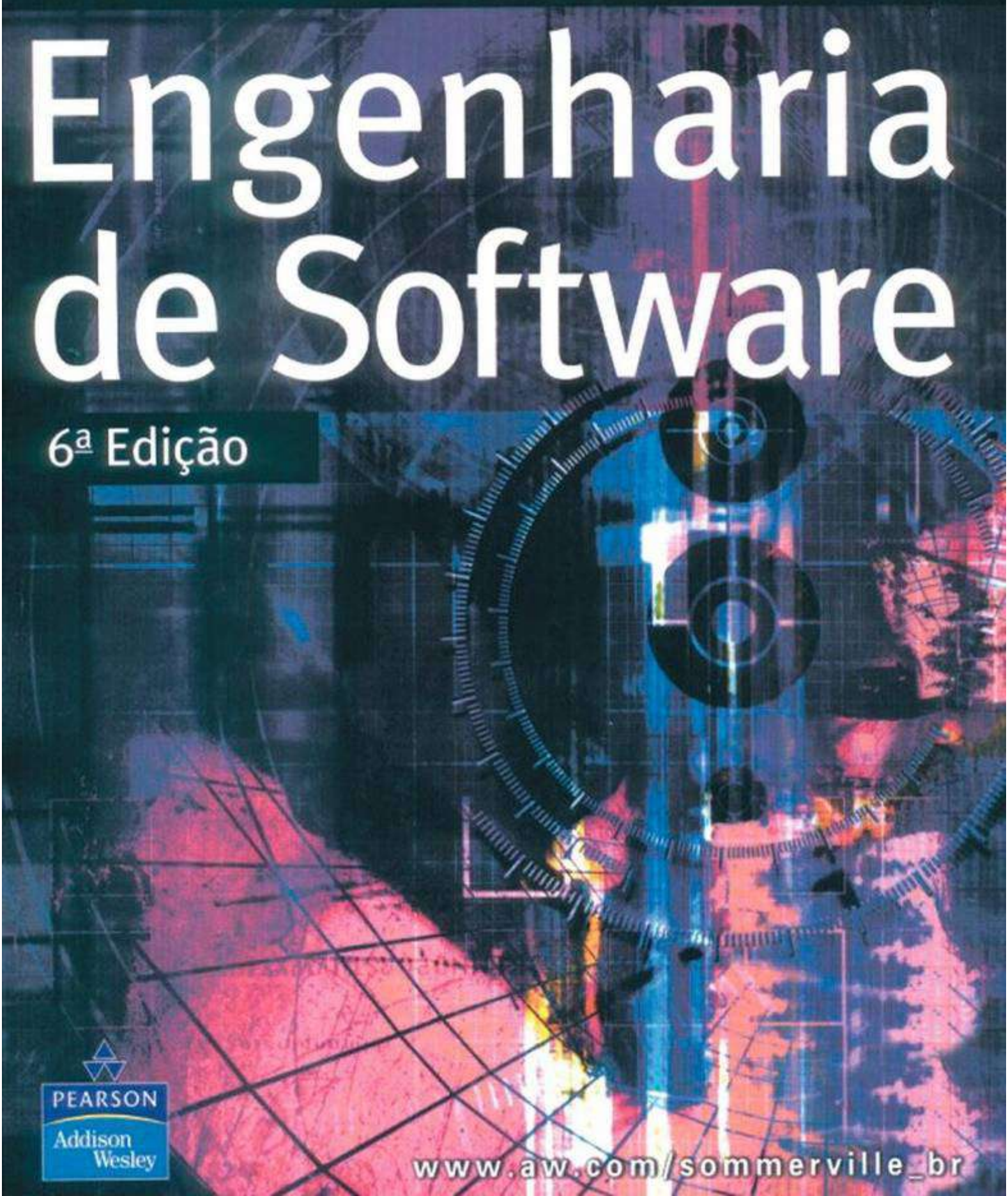
IAN SOMMERVILLE

Engenharia de Software

6ª Edição



PEARSON
Addison Wesley

www.aw.com/sommerville_br

PÁGINA EM BRANCO

Engenharia de Software

6ª Edição

Ian Sommerville

Tradução
Maurício de Andrade

Revisão Técnica
Prof. Dr. Kechi Hirama
Escola Politécnica da Universidade de São Paulo
Departamento de Engenharia de Computação e Sistemas Digitais (PCS)

Addison Wesley

São Paulo

Brasil Argentina Colômbia Costa Rica Chile Espanha Guatemala México Peru Porto Rico Venezuela

© 2003 by Pearson Education do Brasil
Título original: *Software Engineering, 6th Edition*
© Addison-Wesley Publishers Limited 1982, 1984
© Pearson Education Limited 1989, 2001
Publicação autorizada a partir da edição original em inglês,
publicada pela Pearson Education, Inc. sob o selo Addison Wesley
Todos os direitos reservados. Nenhuma parte desta publicação poderá ser reproduzida
ou transmitida de qualquer modo ou por qualquer outro meio, eletrônico ou mecânico,
incluindo fotocópia, gravação ou qualquer outro tipo de sistema de armazenamento
e transmissão de informação, sem prévia autorização, por escrito,
da Pearson Education do Brasil.

Observação sobre marcas registradas:
Java, JavaBeans e Modula-2 são marcas registradas da Sun Microsystems, Inc.;
Lotus Notes é uma marca registrada da Lotus Development Corporation;
Mac OS é uma marca registrada da Apple Computer, Inc.;
Microsoft, PowerPoint, Windows, Visual Basic e Visual C++ são marcas registradas da Microsoft Corporation,
e Unix é uma marca registrada licenciada pela X/Open Company Ltd.

Diretor Editorial: José Martins Braga
Editor: Roger Trimer
Editora de Texto: Rita de Cássia Sam
Preparação: Solange Aparecida Visconte
Revisão: Ivone Aparecida da Silva Andrade e Bete Abreu
Capa: Marcelo da Silva Françozo (Sobre o projeto da editora original)
Diagramação: Figurativa Arte e Projeto Editorial

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Sommerville, Ian
Engenharia de software / Ian Sommerville ; tradução André Maurício de Andrade Ribeiro ; revisão técnica Kechi Hirama. -- São Paulo : Addison Wesley, 2003.

Título original: Software engineering.

Bibliografia.
ISBN 85-88639-07-6

1. Engenharia de software I. Título.

02-5757 CDD-005.1

Índices para catálogo sistemático:

1. Engenharia de software : Computadores
Programação : Processamento de dados 005.1

2004
1ª reimpressão
Direitos exclusivos para a língua portuguesa
cedidos à Pearson Education do Brasil,
uma empresa do grupo Pearson Education
Av. Ermano Marchetti, 1435, Lapa
CEP 05038-001, São Paulo — SP, Brasil
Tel.: (11) 3613-1222, Fax: (11) 3611-0444
e-mail: vendas@pearsoned.com

Prefácio

Hoje em dia, os sistemas de software estão em todos os lugares. Praticamente todo equipamento elétrico apresenta algum tipo de software; o software é utilizado para ajudar nas operações da indústria manufatureira, das escolas e universidades, do setor de assistência à saúde, das finanças e do governo. Muitas pessoas utilizam software de diferentes tipos para fins de entretenimento e de educação. A especificação, o desenvolvimento, o gerenciamento e a evolução desses sistemas de software constituem a disciplina da *engenharia de software*.

Mesmo os sistemas simples de software têm uma alta complexidade inerente e, conseqüentemente, os princípios da engenharia devem ser empregados em seu desenvolvimento. Portanto, a engenharia de software é uma disciplina da engenharia segundo a qual os engenheiros de software utilizam métodos e teorias da ciência da computação e aplicam tudo isso de modo eficaz em relação aos custos, a fim de solucionar problemas difíceis. Esses problemas difíceis significaram que muitos projetos de desenvolvimento de software não foram bem-sucedidos. Contudo, a maioria dos softwares modernos fornece um bom serviço a seus usuários; não deveríamos ter permitido que falhas graves prejudicassem o sucesso real dos engenheiros de software ao longo dos últimos 30 anos.

A engenharia de software foi desenvolvida em resposta a problemas de construção de sistemas de software de grande porte e personalizados, destinados a aplicações industriais, governamentais e para o setor de defesa. Atualmente, desenvolvemos uma variedade muito maior de produtos de software, desde jogos para consoles especializados, passando por produtos para PCs e sistemas baseados na Web, até chegar aos sistemas distribuídos em grande escala. Embora algumas técnicas apropriadas para os sistemas personalizados — como o desenvolvimento orientado a objetos — sejam universais, novas técnicas de engenharia de software estão evoluindo para diferentes tipos de software. Não é possível abordar tudo isso em um único livro; por isso, concentrei-me em técnicas universais e técnicas para o desenvolvimento de sistemas em grande escala, em vez de produtos de software individuais.

Embora este livro seja, a princípio, uma introdução geral à engenharia de software, ele é orientado aos meus próprios interesses na engenharia de requisitos de sistemas e sistemas críticos. Acredito que esses aspectos são particularmente importantes para a engenharia de software no século XXI, quando enfrentamos o desafio de assegurar que nosso software cumpra as reais necessidades de seus usuários, sem causar danos a eles ou ao meio ambiente.

A abordagem utilizada neste livro é a de apresentar uma ampla perspectiva da engenharia de software e não me concentro em quaisquer métodos ou ferramentas específicos. Não me agradam os fanáticos de nenhum tipo, quer sejam acadêmicos pregando os benefícios de métodos formais ou vendedores tentando me convencer de que alguma ferramenta ou algum

método é a resposta aos problemas do desenvolvimento de software. Não existem soluções simples para os problemas da engenharia de software e necessitamos de uma ampla diversidade de instrumentos e técnicas para solucionarmos esses problemas.

Os livros, inevitavelmente, refletem as opiniões e os preconceitos de seus autores. Alguns leitores certamente não concordarão com minhas opiniões e com a escolha de material. Essa discordância é um reflexo saudável da diversidade da disciplina e é essencial para sua evolução. Contudo, espero que todos os engenheiros de software e estudantes de engenharia de software achem este livro interessante.

Características desta edição

Do mesmo modo que muitos sistemas de software, este livro tem evoluído e se modificado desde sua primeira edição, em 1982. Uma de minhas metas na preparação desta edição foi não aumentar o tamanho do livro, o que exigiu alguma reorganização e difíceis decisões sobre o que excluir, enquanto ainda incluía material novo e importante. Eis o que você vai encontrar nesta edição:

- Novos capítulos que abordam processos de software, arquiteturas de sistemas distribuídos, sistemas legados e confiabilidade. A seção sobre especificação formal está em um único capítulo e o material sobre CASE está distribuído em diversos capítulos. A abordagem sobre o projeto funcional é apresentada em um capítulo sobre sistemas legados e a verificação e a validação são tratadas em um único capítulo.
- Todos os capítulos foram atualizados e vários foram reescritos. Agora, o reuso enfoca o desenvolvimento com o reuso, com material sobre padrões e o desenvolvimento baseado em componentes; o projeto orientado a objetos apresenta mais de um enfoque em processo; os requisitos são tratados separadamente em capítulos sobre requisitos propriamente ditos e sobre o processo de engenharia de requisitos, e a estimativa de custos está atualizada para COCOMO 2.
- O material sobre sistemas críticos está integrado e trata de confiabilidade, segurança e disponibilidade em conjunto, além de proteção como um atributo de um sistema crítico.

As leituras sugeridas, no final de cada capítulo, foram atualizadas sempre que possível. Contudo, em muitos casos, os artigos escritos na década de 80 ainda são a melhor introdução para alguns tópicos.

Público-alvo

O livro é destinado a estudantes de cursos de graduação e pós-graduação e a engenheiros de software que atuam no comércio e na indústria. Ele pode ser utilizado em cursos de engenharia de software em geral ou em outros cursos, como os de programação avançada, especificação de software, projeto ou gerenciamento de software. Os profissionais podem achar o livro útil para leitura geral e como um meio de atualizar seu conhecimento em tópicos específicos, como engenharia de requisitos, projeto de arquitetura, desenvolvimento de sistemas de proteção e melhoria de processo. Sempre que possível, apresentei uma visão prática dos exemplos no texto, a fim de refletir o tipo de aplicações que os engenheiros de software devem desenvolver.

Presumi que os leitores tenham alguma familiaridade básica com programação e sistemas de computação modernos e também conheçam estruturas de dados básicas, como pilhas, listas e filas.

Utilizando o livro como um livro-texto

Este livro pode ser utilizado em:

1. *Cursos de introdução geral em engenharia de software* Para estudantes que não tenham experiência prévia em engenharia de software, é possível iniciar com a seção introdutória e, em seguida, escolher capítulos a partir das diferentes seções do livro. Essa opção fornecerá aos estudantes uma visão geral do assunto e possibilitará um estudo mais detalhado para estudantes que tiverem maior interesse.

2. *Cursos introdutórios ou intermediários sobre tópicos específicos de engenharia de software* O livro oferece material de apoio a cursos sobre especificação de requisitos de software, projeto de software, gerenciamento de engenharia de software, desenvolvimento de sistemas confiáveis e evolução de software. Cada uma das partes deste livro pode servir como um texto específico para um curso introdutório ou intermediário sobre o tópico em questão. Sugeri alguma leitura adicional para esses cursos.

3. *Cursos mais avançados sobre tópicos específicos de engenharia de software* Nesse caso, os capítulos do livro formam uma base para o curso, devendo ser complementados com outras leituras que explorem os tópicos com mais detalhes. Todos os capítulos trazem minhas sugestões para outras leituras.

Este texto pode ser utilizado em vários cursos relacionados. Com um único texto, os estudantes têm à sua disposição uma visão abrangente do assunto e não precisam comprar vários livros diferentes.

Este livro abrange todo o material sugerido no componente de engenharia de software do universo de conhecimento da ciência da computação, proposto pelo ACM/IEEE, no documento denominado *Computing Curricula 2001*. Ele também está de acordo com o documento do Software Engineering Body of Knowledge, do IEEE/ACM.

Site Web

O site Web do livro (http://www.aw.com/sommerville_br) traz links para o material de apoio para professores e alunos. Eis os suplementos que estão disponíveis para download:

- Um conjunto de apresentações em PowerPoint para cada capítulo. Os professores podem adaptar e modificar as apresentações conforme desejarem.

- Código-fonte em Java para a maior parte dos exemplos de programas individuais, incluindo código suplementar necessário para compilação.

Agradecimentos

Muitas pessoas contribuíram, durante anos, para a evolução deste livro, e gostaria de agradecer, em primeiro lugar, a todos os que fizeram comentários sobre as primeiras edições e deram sugestões para modificações. Sou grato aos revisores dos rascunhos iniciais deste texto, pela contribuição de seus comentários e suas sugestões, que muito me ajudaram na realização da versão final.

Os revisores do primeiro rascunho foram Andy Gillies e Lindsey Gillies, da University of the West of England; Joe Lambert, da Penn. State University; Frank Maddix, da University of the West of England; Nancy Mead, do Software Institute Engineering, Pittsburgh; Chris Price, da Wales University, Aberystwyth; Gregg Rothermel, da Oregon State University, e Guus Schreiber, da University of Amsterdam. Gostaria de agradecer, em particular, aos meus amigos Ron Morrison, da St. Andrews University, e Ray Welland, da Glasgow University, que revisaram as edições anteriores e se ofereceram novamente para revisar este livro.

Finalmente, minha família suportou minha ausência por mais noites do que gosto de pensar, enquanto finalizava este livro. Meus agradecimentos à minha esposa Anne e às minhas filhas Ali e Jane, pelos cafés e pela paciência.

Ian Sommerville
Lancaster, fevereiro de 2000

Os editores da edição brasileira agradecem a colaboração dos seguintes professores: Adriana Cursino Thomé, da Universidade de Mogi das Cruzes, e Elias Estevão Goulart e Jussara Pimenta Matos, da Faculdade Batista.

Sumário

Parte 1 Visão Geral 1

Capítulo 1 Introdução 3
1.1 Perguntas freqüentes sobre engenharia de software 5
1.2 Responsabilidade profissional e ética 12
Pontos-chave 15
Leituras sugeridas 16
Exercícios 16

Capítulo 2 Engenharia de sistemas com base em computadores 17
2.1 Propriedades emergentes de sistemas 19
2.2 Sistemas e seu ambiente 20
2.3 Modelagem de sistemas 22
2.4 O processo de engenharia de sistemas 25
2.5 Suprimento de sistemas 31
Pontos-chave 33
Leituras sugeridas 33
Exercícios 34

Capítulo 3 Processos de software 35
3.1 Modelos de processo de software 36
3.2 Iteração de processo 42
3.3 Especificação de software 46
3.4 Projeto e implementação de software 47
3.5 Validação de software 50
3.6 Evolução de software 52
3.7 Apoio ao processo automatizado 53
Pontos-chave 57
Leituras sugeridas 57
Exercícios 58

Capítulo 4 Gerenciamento de projetos 59
4.1 Atividades de gerenciamento 61
4.2 Planejamento de projeto 62
4.3 Programação de projeto 65
4.4 Gerenciamento de riscos 70

Pontos-chave 76
Leituras sugeridas 76
Exercícios 77

Parte 2 Requisitos 79

Capítulo 5 Requisitos de software 81
5.1 Requisitos funcionais e não funcionais 83
5.2 Requisitos de usuário 89
5.3 Requisitos de sistema 91
5.4 O documento de requisitos de software 95
Pontos-chave 99
Leituras sugeridas 100
Exercícios 100

Capítulo 6 Processos de engenharia de requisitos 102
6.1 Estudos de viabilidade 103
6.2 Obtenção e análise de requisitos 104
6.3 Validação de requisitos 115
6.4 Gerenciamento de requisitos 117
Pontos-chave 122
Leituras sugeridas 122
Exercícios 123

Capítulo 7 Modelos de sistema 124
7.1 Modelos de contexto 126
7.2 Modelos de comportamento 128
7.3 Modelos de dados 133
7.4 Modelos de objeto 135
7.5 Ferramentas CASE em *workbenches* 139
Pontos-chave 141
Leituras sugeridas 142
Exercícios 142

Capítulo 8 Prototipação de software 144
8.1 Prototipação no processo de software 147
8.2 Técnicas de prototipação rápida 152
8.3 Prototipação de interface com o usuário 159
Pontos-chave 160
Leituras sugeridas 160
Exercícios 161

Capítulo 9 Especificação formal 162
9.1 Especificação formal no processo de software 164
9.2 Especificação de interface 166
9.3 Especificação de comportamento 172
Pontos-chave 177
Leituras sugeridas 177
Exercícios 178

Parte 3 Projeto 179

Capítulo 10 Projeto de arquitetura 181
10.1 Estruturação do sistema 184
10.2 Modelos de controle 189
10.3 Decomposição em módulos 193

10.4 Arquiteturas de domínio específico 196
Pontos-chave 199
Leituras sugeridas 200
Exercícios 200

Capítulo 11 Arquitetura de sistemas distribuídos 202
11.1 Arquiteturas de multiprocessadores 205
11.2 Arquiteturas cliente–servidor 206
11.3 Arquiteturas de objetos distribuídos 211
11.4 CORBA 213
Pontos-chave 217
Leituras sugeridas 218
Exercícios 218

Capítulo 12 Projeto orientado a objetos 220
12.1 Objetos e classes de objetos 222
12.2 Processo de projeto orientado a objetos 226
12.3 Evolução de projeto 238
Pontos-chave 239
Leituras sugeridas 239
Exercícios 240

Capítulo 13 Projeto de software de tempo real 241
13.1 Projeto de sistemas 243
13.2 Executivos de tempo real 246
13.3 Sistemas de monitoramento e controle 250
13.4 Sistemas de aquisição de dados 254
Pontos-chave 256
Leituras sugeridas 256
Exercícios 257

Capítulo 14 Projeto com reuso 259
14.1 Desenvolvimento baseado em componentes 263
14.2 Famílias de aplicações 269
14.3 Padrões de projeto 272
Pontos-chave 275
Leituras sugeridas 275
Exercícios 276

Capítulo 15 Projeto de interface com o usuário 277
15.1 Princípios de projeto de interface com o usuário 279
15.2 Interação dos usuários 282
15.3 Apresentação das informações 284
15.4 Apoio ao usuário 288
15.5 Avaliação da interface 293
Pontos-chave 295
Leituras sugeridas 295
Exercícios 295

Parte 4 Sistemas Críticos 297

Capítulo 16 Confiança 299
16.1 Sistemas críticos 302
16.2 Disponibilidade e confiabilidade 304
16.3 Segurança 308
16.4 Proteção 311

Pontos-chave 313
Leituras sugeridas 313
Exercícios 314

Capítulo 17 Especificação de sistemas críticos 315
17.1 Especificação de confiabilidade de software 316
17.2 Especificação de segurança 322
17.3 Especificação de proteção 329
Pontos-chave 330
Leituras sugeridas 331
Exercícios 331

Capítulo 18 Desenvolvimento de sistemas críticos 333
18.1 Minimização de defeitos 334
18.2 Tolerância a defeitos 339
18.3 Arquiteturas tolerantes a defeitos 348
18.4 Projeto de sistema seguro 351
Pontos-chave 352
Leituras sugeridas 352
Exercícios 352

Parte 5 Verificação e Validação 355

Capítulo 19 Verificação e validação 357
19.1 Planejamento de verificação e validação 361
19.2 Inspeções de software 362
19.3 Análise estática automatizada 367
19.4 Desenvolvimento de software Cleanroom 370
Pontos-chave 373
Leituras sugeridas 373
Exercícios 374

Capítulo 20 Testes de software 375
20.1 Testes para a detecção de defeitos 377
20.2 Testes de integração 385
20.3 Testes orientados a objetos 390
20.4 Bancadas de testes 394
Pontos-chave 396
Leituras sugeridas 396
Exercícios 397

Capítulo 21 Validação de sistemas críticos 398
21.1 Métodos formais e sistemas críticos 399
21.2 Validação de confiabilidade 401
21.3 Garantia de segurança 405
21.4 Avaliação de proteção 411
Pontos-chave 412
Leituras sugeridas 412
Exercícios 413

Parte 6 Gerenciamento 415

Capítulo 22 Gerenciamento de pessoal 417
22.1 Limites para o pensamento 418
22.2 Trabalho em grupo 424

22.3 Seleção e manutenção de pessoal 429
22.4 Modelo de Maturidade de Capacitação de Pessoal 432
Pontos-chave 434
Leituras sugeridas 434
Exercícios 435

Capítulo 23 Estimativa de custo de software 436
23.1 Produtividade 438
23.2 Técnicas de estimativa 442
23.3 Modelagem algorítmica de custos 444
23.4 Duração de projeto e pessoal 453
Pontos-chave 455
Leituras sugeridas 455
Exercícios 456

Capítulo 24 Gerenciamento de qualidade 457
24.1 Garantia e padrões de qualidade 460
24.2 Planejamento de qualidade 465
24.3 Controle de qualidade 466
24.4 Medição e métricas de software 468
Pontos-chave 474
Leituras sugeridas 474
Exercícios 475

Capítulo 25 Melhoria de processo 476
25.1 Qualidade de processo e de produto 479
25.2 Modelagem e análise de processo 480
25.3 Medições de processo 484
25.4 Modelo de Maturidade de Capacitação do SEI 485
25.5 Classificação de processo 490
Pontos-chave 492
Leituras sugeridas 493
Exercícios 493

Parte 7 Evolução 495

Capítulo 26 Sistemas legados 497
26.1 Estruturas dos sistemas legados 499
26.2 Projeto de sistemas legados 503
26.3 Avaliação dos sistemas legados 507
Pontos-chave 512
Leituras sugeridas 512
Exercícios 513

Capítulo 27 Mudanças de software 514
27.1 Dinâmica da evolução de programas 516
27.2 Manutenção de software 517
27.3 Evolução de arquitetura 525
Pontos-chave 530
Leituras sugeridas 531
Exercícios 531

Capítulo 28 Reengenharia de software 532
28.1 Tradução de código-fonte 536
28.2 Engenharia reversa 537
28.3 Melhoria de estrutura de programa 538

28.4 Modularização de programa 541
28.5 Reengenharia de dados 542
Pontos-chave 546
Leituras sugeridas 547
Exercícios 547

Capítulo 29 Gerenciamento de configuração 549
29.1 Planejamento de gerenciamento de configuração 551
29.2 Gerenciamento de mudanças 554
29.3 Gerenciamento de versões e releases 556
29.4 Construção de sistemas 561
29.5 Ferramentas CASE para o gerenciamento de configuração 562
Pontos-chave 566
Leituras sugeridas 566
Exercícios 567

Referências 569

Índice onomástico 581

Índice remissivo 585

PARTE UM

Visão Geral

PÁGINA EM BRANCO

1 Introdução

Objetivos

O objetivo deste capítulo é introduzir o assunto de engenharia de software. Depois de ler este capítulo, você poderá:

- compreender o que é a engenharia de software e por que ela é importante;
- saber as respostas de perguntas importantes que proporcionam uma introdução à engenharia de software, e
- compreender questões profissionais e éticas relevantes para os engenheiros de software.

Sumário

1.1 Perguntas freqüentes sobre engenharia de software
1.2 Responsabilidade profissional e ética

Praticamente todos os países, hoje em dia, dependem de complexos sistemas com base em computadores. Cada vez mais os produtos incorporam, de algum modo, computadores e software de controle. Nesses sistemas, o software representa uma grande e crescente proporção do custo total do sistema. Por isso, produzir software de um modo que apresente uma boa relação custo-benefício é essencial para o funcionamento das economias nacionais e internacionais.

A engenharia de software é uma disciplina da engenharia, cuja meta é o desenvolvimento de sistemas de software com boa relação custo-benefício. O software é abstrato e intangível, não é restrito por materiais nem regido por leis físicas ou por processos industriais. De certa maneira, isso simplifica a engenharia de software, por não haver limitações físicas para o potencial do software. Por outro lado, contudo, essa ausência de limitações naturais significa que o software pode facilmente se tornar bastante complexo e, conseqüentemente, muito difícil de ser compreendido.

A engenharia de software ainda é uma disciplina relativamente nova. A noção de 'engenharia de software' surgiu pela primeira vez em 1968, em uma conferência organizada para discutir a chamada 'crise do software'. Essa crise resultava diretamente da introdução (naquela época) do poderoso hardware de computador de terceira geração. Sua capacidade tornava viáveis aplicações de computador até então inimagináveis. O software resultante era bem maior e mais complexo que os sistemas de software precedentes.

A experiência inicial de construção desses sistemas mostrou que uma abordagem informal do desenvolvimento de software não era o bastante. Projetos importantes sofriam atrasos, às vezes, de alguns anos. Por apresentarem custos muito maiores do que os inicialmente previstos, eles não eram confiáveis, eram de difícil manutenção e tinham desempenho inferior. O desenvolvimento de software estava em crise. Os custos de hardware caíam, enquanto os de software subiam rapidamente. Novas técnicas e novos métodos eram necessários para controlar a complexidade inerente aos grandes sistemas de software.

Essas técnicas se tornaram parte da engenharia de software e são agora amplamente, mas não universalmente, utilizadas. No entanto, ainda há problemas em produzir software complexo, que atenda às expectativas dos usuários e que seja entregue dentro do prazo e do orçamento estabelecido. Muitos projetos de software ainda têm problemas, e isso levou alguns críticos (Pressman, 1997) a sugerirem que a engenharia de software está em um estado de aflição crônica.

À medida que nossa capacidade de produzir software aumentou, também cresceu a complexidade dos sistemas de software requeridos. Novas tecnologias que resultam da convergência de sistemas de computadores e de comunicação trazem novas questões para os engenheiros de software. Por essa razão e pelo fato de muitas empresas não aplicarem as técnicas de engenharia de software de maneira eficaz, ainda temos problemas. A situação não é tão ruim quanto os pessimistas sugerem, mas, com certeza, há espaço para melhorias.

Houve um grande progresso desde 1968 e o desenvolvimento da engenharia de sofware melhorou de modo marcante o software que produzimos. Temos uma compreensão muito melhor das atividades envolvidas no desenvolvimento de software. Desenvolvemos métodos eficazes de especificação, projeto e implementação de software. Novas notações e ferramentas reduzem o esforço necessário para produzir sistemas grandes e complexos.

Os engenheiros de software podem, com razão, se orgulhar de suas realizações. Sem softwares complexos, não teríamos explorado o espaço, não teríamos a Internet e as modernas telecomunicações e todos os meios de viagem seriam mais perigosos e dispendiosos. A engenharia de software tem contribuído muito, em sua curta existência, e estou convencido de que, à medida que essa disciplina amadurecer, sua contribuição no século XXI será ainda maior.

1.1 Perguntas freqüentes sobre engenharia de software

Esta seção tem como propósito responder a algumas perguntas fundamentais sobre engenharia de software, além de fornecer ao leitor o ponto de vista do autor sobre a disciplina. O formato utilizado nesta seção é o de lista de perguntas mais freqüentes (*frequently asked questions* – FAQ). Essa abordagem é comumente utilizada nos grupos de discussões da Internet para proporcionar aos iniciantes respostas a perguntas freqüentemente formuladas. Acredito que essa seja uma maneira muito eficaz de oferecer uma introdução simples à engenharia de software.

As perguntas respondidas nesta seção estão indicadas na Figura 1.1.

1.1.1 O que é software?

Muita gente associa o termo *software* aos programas de computador. Na verdade, essa é uma visão muito restritiva. Software não é apenas o programa mas também toda a documentação associada e os dados de configuração necessários para fazer com que esses programas operem corretamente. Um sistema de software, usualmente, consiste em uma série de programas separados, arquivos de configuração que são utilizados para configurar esses programas, documentação do sistema, que descreve a estrutura desse sistema, e documentação do usuário, que explica como utilizar o sistema e, no caso dos produtos de software, sites Web para os usuários fazerem o download das informações recentes sobre o produto.

Os engenheiros de software se ocupam de desenvolver produtos de software, isto é, software que possa ser vendido a um cliente. Há dois tipos de produtos de software:

1. *Produtos genéricos* São sistemas do tipo *stand-alone*, produzidos por uma organização de desenvolvimento e vendidos no mercado a qualquer cliente capaz de adquiri-los. Algumas vezes, referimo-nos a eles como pacotes de software. Dentre os exemplos desse tipo de produto estão as bases de dados, os processadores de texto, os pacotes de desenho e as ferramentas de gerenciamento de projetos.
2. *Produtos sob encomenda (ou personalizados)* São os sistemas encomendados por um cliente em particular. O software é desenvolvido especialmente para aquele cliente por uma empresa de software. Dentre os exemplos desse tipo de software destacam-se os sistemas de controle para dispositivos eletrônicos, sistemas escritos para serem compatíveis com um processo de negócios específico e sistemas de controle de tráfego aéreo.

Uma diferença importante entre esses diversos tipos de software é que, nos produtos genéricos, a organização que desenvolve o software controla sua especificação. Para os produtos personalizados, a especificação é usualmente desenvolvida e controlada pela organização que está comprando o software. Os desenvolvedores de software devem trabalhar de acordo com essa especificação.

1.1.2 O que é engenharia de software?

A engenharia de software é uma disciplina da engenharia que se ocupa de todos os aspectos da produção de software, desde os estágios iniciais de especificação do sistema até a manutenção desse sistema, depois que ele entrou em operação. Nessa definição, há duas frases importantes:

1. 'Disciplina da engenharia': os engenheiros fazem os produtos funcionarem. Eles aplicam teorias, métodos e ferramentas nas situações apropriadas, de modo seletivo; e sempre

Pergunta	Resposta
O que é software?	São os programas de computador e a documentação associada. Produtos de software podem ser desenvolvidos para um cliente específico ou para o mercado.
O que é engenharia de software?	Engenharia de software é uma disciplina da engenharia que se ocupa de todos os aspectos da produção de software.
Qual é a diferença entre engenharia de software e ciência da computação?	A ciência da computação se ocupa da teoria e dos fundamentos; a engenharia de software trata da praticidade de desenvolver e fornecer software útil.
Qual é a diferença entre engenharia de software e engenharia de sistemas?	A engenharia de sistemas se ocupa de todos os aspectos relacionados ao desenvolvimento de sistemas com base em computadores, incluindo hardware, software e engenharia de processos. A engenharia de software é parte desse processo.
O que é um processo de software?	É um conjunto de atividades, cuja meta é o desenvolvimento ou a evolução do software.
O que é um modelo de processo de software?	É uma representação simplificada de um processo de software, apresentada a partir de uma perspectiva específica.
Quais são os custos da engenharia de software?	Aproximadamente 60 por cento dos custos são relacionados ao desenvolvimento e 40 por cento são custos referentes aos testes. Para o software personalizado, os custos de evolução freqüentemente excedem os custos de desenvolvimento.
O que são métodos de engenharia de software?	São abordagens estruturadas para o desenvolvimento de software, que incluem modelos de sistemas, notações, regras, recomendações de projetos e diretrizes de processos.
O que é CASE (*computer-aided software engineering* – engenharia de software com o auxílio de computador)?	São sistemas de software destinados a proporcionar apoio automatizado às atividades de processo de software. Os sistemas CASE são freqüentemente utilizados para proporcionar apoio aos métodos.
Quais são os atributos de um bom software?	O software deve proporcionar ao usuário a funcionalidade e o desempenho requeridos e deve ser passível de manutenção, confiável e de fácil uso.
Quais são os principais desafios enfrentados pela engenharia de software?	Lidar com sistemas legados, atender à crescente diversidade e atender às exigências quanto a prazos de entrega reduzidos.

Figura 1.1 ▲

Perguntas freqüentes sobre engenharia de software.

procuram descobrir soluções para os problemas, mesmo quando não existem teorias aplicáveis e métodos de apoio. Os engenheiros também reconhecem que precisam trabalhar de acordo com as restrições organizacionais e financeiras e, assim, procuram soluções que estejam dentro dessas restrições.

2. 'Todos os aspectos da produção de software': a engenharia de software não se dedica só aos processos técnicos de desenvolvimento de software, mas também a atividades como o gerenciamento de projetos de software e o desenvolvimento de ferramentas, métodos e teorias que dêem apoio à produção de software.

Em geral, os engenheiros de software adotam uma abordagem sistemática e organizada em seu trabalho, uma vez que essa é, com freqüência, a maneira mais eficaz de produzir software de alta qualidade. No entanto, a engenharia tem a ver, em grande parte, com a questão de selecionar o método mais apropriado para um conjunto de circunstâncias, e uma abordagem mais criativa e informal para o desenvolvimento pode ser eficaz em algumas circunstâncias. O desenvolvimento informal é particularmente apropriado para o desenvolvimento de sistemas de comércio eletrônico (*e-commerce*), com base na Web, que requerem uma combinação de habilidades de software e projetos gráficos.

1.1.3 Qual é a diferença entre engenharia de software e ciência da computação?

Essencialmente, a ciência da computação se preocupa com as teorias e os métodos básicos referentes aos computadores e sistemas de software, enquanto a engenharia de software se dedica aos problemas práticos da produção de software. É fundamental ter algum conhecimento em ciência da computação para os engenheiros de software, da mesma maneira que física é pré-requisito para engenheiros elétricos.

O ideal seria que todos os engenheiros de software se apoiassem em teorias da ciência da computação, mas, na realidade, esse não é o caso. Os engenheiros de software devem, freqüentemente, utilizar abordagens específicas para desenvolver software. Teorias mais refinadas da ciência da computação nem sempre podem ser aplicadas a problemas reais e complexos, que requerem uma solução de software.

1.1.4 Qual é a diferença entre engenharia de software e engenharia de sistemas?

A engenharia de sistemas ou, mais precisamente, a engenharia de sistemas com base em computadores se ocupa de todos os aspectos do desenvolvimento e da evolução de sistemas complexos, em que o software desempenha papel principal. A engenharia de sistemas se ocupa, portanto, não só do desenvolvimento de hardware, do projeto de políticas e processos e da implantação de sistemas, como também da engenharia de software. Os engenheiros de sistemas estão envolvidos na especificação do sistema, na definição da sua arquitetura geral e, em seguida, na integração das diferentes partes necessárias para criar o sistema completo. Eles estão menos preocupados com a engenharia dos componentes do sistema (como hardware e software).

A engenharia de sistemas é uma disciplina mais antiga do que a engenharia de software. As pessoas têm especificado e montado sistemas industriais complexos, como trens e indústrias químicas, por mais de cem anos. Contudo, como o percentual de software em sistemas cresceu, as técnicas de engenharia de software, como, por exemplo, a modelagem de use-case e o gerenciamento de configuração, estão sendo utilizadas no processo de engenharia de sistemas. A engenharia de sistemas será discutida com mais detalhes no Capítulo 2.

1.1.5 O que é um processo de software?

Um processo de software é um conjunto de atividades e resultados associados que geram um produto de software. Essas atividades são, em sua maioria, executadas por engenheiros de software. Há quatro atividades de processo fundamentais (que serão abordadas posteriormente neste livro) comuns a todos os processos de software. Essas atividades são:

1. *Especificação do software* A funcionalidade do software e as restrições em sua operação devem ser definidas.
2. *Desenvolvimento do software* O software deve ser produzido de modo que atenda a suas especificações.
3. *Validação do software* O software tem de ser validado para garantir que ele faz o que o cliente deseja.
4. *Evolução do software* O software deve evoluir para atender às necessidades mutáveis do cliente.

Diferentes processos de software organizam essas atividades de maneiras diversas e são descritos em diferentes níveis de detalhes. Os prazos das atividades variam, do mesmo modo que os resultados de cada atividade. Diferentes organizações podem utilizar processos diferentes para produzir o mesmo tipo de produto. No entanto, alguns processos são mais adequados do que outros para alguns tipos de aplicação. Se um processo inadequado for utilizado,

isso provavelmente reduzirá a qualidade ou a utilidade do produto de software a ser desenvolvido.

Os processos de software serão discutidos mais detalhadamente no Capítulo 3, e o importante tópico da melhoria dos processos de software será discutido no Capítulo 25.

1.1.6 O que é um modelo de processo de software?

Um modelo de processo de software é uma descrição simplificada de um processo de software, que é apresentada a partir de uma perspectiva específica. Os modelos, pela sua natureza, são simplificações; e, assim, um modelo de processo de software é uma abstração do processo real que está sendo descrito. Dentre os modelos de processos destacam-se atividades que são parte do processo de software, produtos de software e o papel das pessoas envolvidas na engenharia de software. Alguns exemplos dos tipos de modelos de processo de software que podem ser produzidos são:

1. *Um modelo de* workflow Ele mostra a seqüência de atividades no processo, juntamente com suas entradas, saídas e dependências. As atividades, nesse modelo, representam ações humanas.
2. *Um modelo de fluxo de dados ou de atividade* Ele representa o processo como um conjunto de atividades, cada uma das quais realiza alguma transformação de dados. Esse modelo mostra como a entrada para o processo, tal como uma especificação, é transformada em uma saída, como um projeto. As atividades, nesse caso, podem estar em um nível inferior ao das atividades em um modelo de *workflow*. Elas podem representar transformações realizadas por pessoas ou por computadores.
3. *Um modelo de papel/ação* Esse modelo representa os papéis das pessoas envolvidas no processo de software e as atividades pelas quais elas são responsáveis.

Existe uma série de diferentes modelos gerais, ou paradigmas, de desenvolvimento de software:

1. *O modelo em cascata* Ele considera as atividades mostradas anteriormente e as representa como fases separadas do processo, como especificações de requisitos, projeto de software, implementação, testes e assim por diante. Após cada estágio ter sido definido, ele é 'aprovado' e o desenvolvimento prossegue para o estágio seguinte.
2. *Desenvolvimento evolucionário* Essa abordagem intercala as atividades de especificação, desenvolvimento e validação. Um sistema inicial é rapidamente desenvolvido a partir de especificações muito abstratas. Em seguida, ele é refinado com as informações do cliente, a fim de produzir um sistema que satisfaça a suas necessidades. O sistema pode, então, ser entregue. Como alternativa, ele pode ser reimplementado, utilizando uma abordagem mais estruturada, para produzir um sistema mais robusto e com maior facilidade de manutenção.
3. *Transformação formal* Essa abordagem se baseia na produção de uma especificação formal matemática do sistema e na transformação dessa especificação, utilizando-se métodos matemáticos, em um programa. Essas transformações 'preservam a correção'. Isso significa que se pode ter certeza de que o programa desenvolvido atende a suas especificações.
4. *Montagem de um sistema a partir de componentes reutilizáveis* Essa técnica supõe que partes do sistema já existam. O processo de desenvolvimento do sistema se concentra na integração dessas partes, em vez de desenvolvê-las a partir do zero. O reuso de software será estudado no Capítulo 14.

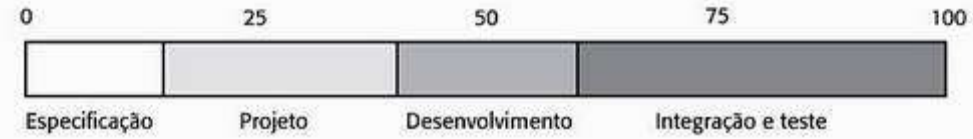
Retornarei a esses modelos de processos genéricos no Capítulo 3.

1.1.7 Quais são os custos da engenharia de software?

Não existe uma resposta simples para essa pergunta, uma vez que a distribuição precisa de custos ao longo do processo de software depende do processo utilizado e do tipo de software que está sendo desenvolvido. Se considerarmos o custo total de desenvolvimento de um complexo sistema de software como de cem unidades de custo, a distribuição dessas unidades provavelmente será semelhante ao que mostra a Figura 1.2.

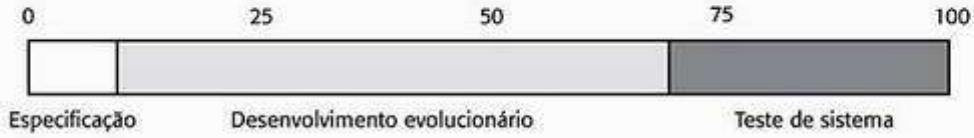
Essa distribuição de custos pode ser empregada quando os custos de especificação, projeto, implementação e integração são medidos separadamente. Observe que a integração e os testes do sistema são as atividades de desenvolvimento mais dispendiosas. A Figura 1.2 sugere que essas atividades representam cerca de 40 por cento dos custos totais de desenvolvimento, mas, para alguns sistemas fundamentais, esse valor pode chegar, provavelmente, perto de 50 por cento dos custos totais do sistema.

Figura 1.2 ▶ Distribuição dos custos de desenvolvimento.



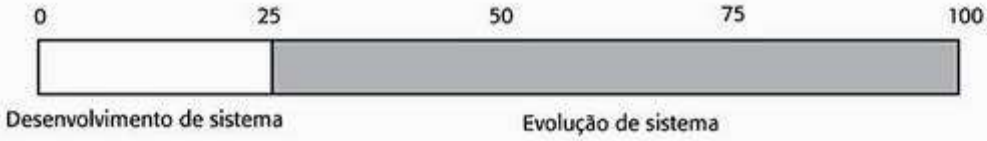
Se o software for desenvolvido utilizando-se uma abordagem evolucionária, não existirá uma linha exata entre especificação, projeto e desenvolvimento. A Figura 1.2 teria de ser modificada para esse tipo de desenvolvimento, como mostra a Figura 1.3. Os custos de especificação ficam reduzidos, porque somente uma especificação de alto nível é produzida antes do desenvolvimento, considerando-se essa abordagem. As fases de especificação, projeto, implementação, integração e testes são realizadas em paralelo, dentro de uma atividade de desenvolvimento. Contudo, ainda há a necessidade de uma atividade separada de testes do sistema, uma vez que a implementação inicial esteja terminada.

Figura 1.3 ▶ Custos de desenvolvimento evolucionário.



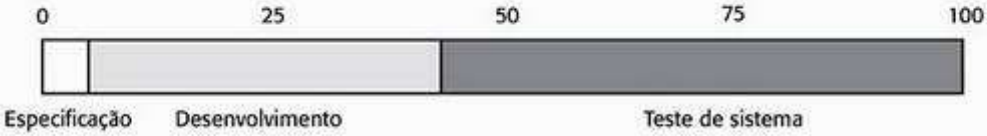
Além dos custos de desenvolvimento, também existem os custos de alteração do software, depois de ele ter sido colocado em uso. Para muitos sistemas de software com um longo ciclo de duração, é provável que esses custos excedam os custos de desenvolvimento em três ou quatro vezes (Figura 1.4).

Figura 1.4 ▶ Custos de evolução.



A distribuição dos custos, exemplificada anteriormente, vale para software personalizado, que é especificado por um cliente e desenvolvido por uma empresa de software contratada. Para os produtos de software que são vendidos (principalmente) para PCs, o perfil de custo é provavelmente diferente. Esses produtos são usualmente desenvolvidos a partir de um esboço de especificação, utilizando uma abordagem de desenvolvimento evolucionário. Os custos de especificação são relativamente baixos. No entanto, por se destinarem ao uso em uma faixa de diferentes configurações, eles devem ser extensivamente testados. A Figura 1.5 mostra o tipo de perfil de custos que pode ser esperado para esses produtos.

Figura 1.5 ▶ Custos de desenvolvimento de produto.



Os custos de evolução para produtos de software genéricos são particularmente difíceis de ser estimados. Em muitos casos, ocorre pouca evolução formal de um produto. Uma vez que uma versão do produto foi lançada, começa o trabalho relacionado à próxima versão e, por razões de marketing, é provável que ela seja apresentada como um produto novo (mas compatível), em vez de como uma versão modificada de um produto que o usuário já comprou. Portanto, os custos de evolução não são registrados separadamente, como ocorre com o software personalizado, mas são simplesmente os custos de desenvolvimento para a próxima versão do sistema.

O modelo de custos para os sistemas de comércio eletrônico com base na Web é, provavelmente, diferente de ambos os anteriores. Esses sistemas, em geral, utilizam software de prateleira para o gerenciamento de informações e têm altos custos de desenvolvimento de interface com o usuário. Quando este livro estava sendo escrito, esses sistemas estavam apenas começando a ser utilizados e não havia quaisquer números confiáveis sobre seus custos de desenvolvimento.

1.1.8 O que são métodos de engenharia de software?

Um método de engenharia de software é uma abordagem estruturada para o desenvolvimento de software, cujo objetivo é facilitar a produção de software de alta qualidade, apresentando uma boa relação custo-benefício. Métodos como análise estruturada (DeMarco, 1978) e JSD (Jackson, 1983) foram inicialmente desenvolvidos na década de 70. Esses métodos procuravam identificar os componentes funcionais básicos de um sistema, e os métodos orientados a funções ainda são amplamente utilizados. Nas décadas de 80 e 90, esses métodos orientados a funções foram complementados por métodos orientados a objetos, como aqueles propostos por Booch (1994) e Rumbaugh (Rumbaugh *et al.*, 1991). Essas diferentes abordagens foram integradas em uma única, construída segundo o UML (*unified modeling language* – linguagem de modelagem unificada) (Fowler e Scott, 1997; Booch *et al.*, 1999; Rumbaugh *et al.*, 1999a, 1999b).

Todos os métodos se baseiam na idéia de desenvolver modelos de um sistema que possam ser representados graficamente e de utilizar esses modelos como uma especificação ou projeto de sistema. Os métodos devem incluir uma série de diferentes componentes (Figura 1.6).

Não existe método ideal, e diferentes métodos têm áreas de aplicação diversificadas. Por exemplo, os métodos orientados a objetos são, com freqüência, apropriados para sistemas interativos, mas não para sistemas com severos requisitos de tempo real.

Figura 1.6 ▼ Componentes do método.

Componente	Descrição	Exemplo
Descrições de modelos de sistema	Descrições dos modelos de sistema a serem desenvolvidos e a notação utilizada para definir esses modelos.	Modelos de objeto, modelos de fluxo de dados, modelos de máquinas de estado etc.
Regras	Restrições que sempre se aplicam a modelos de sistemas.	Cada entidade em um modelo de sistema deve ter um único nome.
Recomendações	Heurísticas que caracterizam (distinguem) a boa prática de projeto nesse método. Seguindo essas recomendações, deve-se chegar a um modelo de sistema bem-organizado.	Nenhum objeto deve ter mais que sete subobjetos associados a ele.
Diretrizes de processo	Descrições das atividades que podem ser seguidas para desenvolver os modelos de sistema e a organização dessas atividades.	Atributos de objetos devem ser documentados, antes de serem definidas as operações associadas a um objeto.

1.1.9 O que é CASE?

O acrônimo CASE significa *computer-aided software engineering* (engenharia de software com o auxílio de computador). Ele se refere a uma ampla gama de diferentes tipos de programas utilizados para apoiar as atividades de processo de software, como a análise de requisitos, a modelagem de sistema, a depuração e os testes. Todos os métodos atualmente são fornecidos com tecnologia CASE associada, como no caso de editores para as notações utilizadas no método, módulos de análise que verificam o modelo do sistema de acordo com as regras do método e geradores de relatórios para ajudar na criação da documentação do sistema. As ferramentas CASE podem também incluir um gerador de códigos que, automaticamente, origina código-fonte a partir do modelo de sistema e alguma orientação de processo, que fornece conselhos ao engenheiro de software sobre o que fazer em seguida.

Esse tipo de ferramenta CASE, destinada a dar apoio à análise e ao projeto, é, algumas vezes, chamada de ferramenta *Upper-CASE*, porque dá apoio às fases iniciais do processo de software. Por outro lado, as ferramentas CASE projetadas para dar apoio à implementação e aos testes, como depuradores, sistemas de análise de programa, geradores de casos de testes e editores de programas, são, às vezes, chamadas de ferramentas *Lower-CASE*.

1.1.10 Quais são os atributos de um bom software?

Da mesma maneira que os serviços que eles fornecem, os produtos de software têm uma série de outros atributos associados, que refletem a qualidade do software. Esses atributos não são diretamente associados ao que o software faz. Em vez disso, eles refletem seu comportamento quando em funcionamento, a estrutura e a organização do programa fonte e também a documentação associada. Exemplos desses atributos (às vezes, chamados de atributos não funcionais) são o tempo de resposta do software à consulta de um usuário e a facilidade de compreensão do código do programa.

O conjunto específico de atributos que se pode esperar de um sistema de software depende obviamente de sua aplicação. Portanto, um sistema bancário tem de ser seguro, um jogo interativo deve ter uma resposta rápida, um sistema de controle de telefonia precisa ser confiável e assim por diante. Isso pode ser generalizado pelo conjunto de atributos mostrado na Figura 1.7, que são, a meu ver, as características essenciais de um sistema de software bem projetado.

As técnicas discutidas neste livro abordam dois desses atributos, a saber: facilidade de manutenção e confiabilidade. A maioria dos métodos de engenharia de software, das ferramentas e das técnicas é direcionada para a produção de software com essas características. A melhoria de desempenho do software geralmente depende do conhecimento de um setor muito específico, e sua facilidade de uso merece um tópico principal e separado, com seu próprio lugar de destaque, que será discutido no Capítulo 15.

1.1.11 Quais são os principais desafios enfrentados pela engenharia de software?

A engenharia de software, no século XXI, enfrenta três principais desafios:

1. *O desafio do legado* A maioria dos grandes sistemas de software em uso atualmente foi desenvolvida há muitos anos e, contudo, opera importantes funções corporativas. O desafio do legado significa poder fazer a manutenção e a atualização desses softwares, para evitar custos excessivos e prosseguir com a prestação de serviços corporativos essenciais.
2. *O desafio da heterogeneidade* Cada vez mais se exige que os sistemas operem como sistemas distribuídos por meio de redes que possuem diferentes tipos de computadores e diferentes tipos de sistemas de apoio. O desafio da heterogeneidade se refere a desen-

Figura 1.7

Atributos essenciais de um bom software.

Característica do produto	Descrição
Facilidade de manutenção	O software deve ser escrito de modo que possa evoluir para atender às necessidades mutáveis dos clientes. Esse é um atributo crucial, porque as modificações em um software são uma conseqüência inevitável de um ambiente de negócios em constante mutação.
Nível de confiança	O nível de confiança do software tem uma gama de características que incluem confiabilidade, proteção e segurança. O software confiável não deve ocasionar danos físicos ou econômicos, no caso de um defeito no sistema.
Eficiência	O software não deve desperdiçar os recursos do sistema, como memória e ciclos do processador. A eficiência, portanto, inclui a rapidez de resposta, o tempo de processamento, a utilização da memória, entre outros.
Facilidade de uso	O software deve ser utilizável, sem esforços indevidos, pelo tipo de usuário para quem foi projetado. Isso significa que ele deve dispor de uma interface apropriada com o usuário e de documentação adequada.

volver técnicas para construir softwares confiáveis, que sejam flexíveis o bastante para lidar com essa heterogeneidade.

3. *O desafio do fornecimento* Muitas técnicas de engenharia de software tradicionais são muito demoradas. O tempo que elas gastam é necessário para que atinjam a qualidade de software exigida. Contudo, as empresas atualmente exigem respostas rápidas e se modificam muito rapidamente. Seu software de apoio deve se modificar com a mesma rapidez. O desafio do fornecimento diz respeito a reduzir o tempo para o fornecimento de sistemas grandes e complexos, sem comprometer a qualidade.

Naturalmente, esses desafios não são independentes. Por exemplo, pode ser necessário fazer mudanças rápidas em um sistema legado, para torná-lo acessível por meio de uma rede. Para resolvermos esses desafios, precisaremos de novas ferramentas e técnicas, bem como maneiras inovadoras de combinar e utilizar os métodos de engenharia de software já existentes.

1.2 Responsabilidade profissional e ética

Assim como ocorre com os outros engenheiros, os engenheiros de software devem aceitar que seu trabalho envolve responsabilidades mais amplas do que a simples aplicação de habilidades técnicas. Seu trabalho é realizado dentro de uma estrutura legal e social. A engenharia de software é, com certeza, delimitada por leis locais, nacionais e internacionais. Os engenheiros de software devem se comportar de maneira responsável, ética e moralmente, para serem respeitados como profissionais.

Nem é preciso dizer que os engenheiros devem manter padrões normais de honestidade e integridade. Eles não devem utilizar suas capacitações e habilidades para se comportarem de maneira desonesta ou que possa trazer descrédito à profissão de engenharia de software. No entanto, existem áreas em que os padrões de comportamento aceitável não estão delimitados por leis, mas pela noção mais tênue de responsabilidade profissional. Eis algumas dessas áreas:

1. *Confidencialidade* Os engenheiros devem, normalmente, respeitar a confidencialidade de seus empregadores ou clientes, quer tenham ou não assinado um acordo formal de confidencialidade.

2. *Competência* Os engenheiros não devem enganar quanto ao seu nível de competência. Não devem conscientemente aceitar serviços que estejam fora do seu limite de competência.

3. *Direitos de propriedade intelectual* Os engenheiros devem estar cientes das leis locais que regulam o uso da propriedade intelectual, como patentes e direitos autorais. Eles devem ser cuidadosos, a fim de assegurar que a propriedade intelectual de empregadores e clientes seja protegida.

4. *Má utilização de computadores* Os engenheiros de software não devem empregar suas habilidades técnicas para o mau uso dos computadores de outras pessoas. O mau uso de computadores abrange desde casos relativamente triviais (por exemplo, jogar no computador do empregador) até casos extremamente sérios (como a disseminação de vírus).

A esse respeito, sociedades e instituições profissionais desempenham um importante papel. Organizações como a ACM (Association for Computing Machinery), o IEEE (Institute of Electrical and Electronic Engineers) e a British Computer Society publicam um código de conduta profissional ou código de ética. Os membros dessas organizações se comprometem a seguir esse código quando se inscrevem como membros. Esses códigos de conduta tratam, em geral, do comportamento ético fundamental.

A ACM e o IEEE cooperaram na produção de um código comum de ética e de prática profissional. Esse código existe tanto na versão resumida, como mostra a Figura 1.8, quanto na mais extensa (Gotterbarn *et al.*, 1999), que acrescenta detalhes e conteúdo à versão compacta. A base lógica de sustentação desse código está resumida nos dois primeiros parágrafos da versão mais detalhada:

> *Os computadores desempenham um papel central e crescente no comércio, na indústria, no governo, na medicina, na educação, no entretenimento e na sociedade em geral. Os engenheiros de software são aqueles que contribuem, por participação direta ou pelo ensino, para a análise, a especificação, o projeto, o desenvolvimento, a certificação, a manutenção e os testes dos sistemas de software. Em razão de seu papel no desenvolvimento de sistemas de software, eles têm oportunidades significativas de praticar o bem ou de causar o mal, de capacitar outras pessoas a fazer o bem ou causar o mal ou de influenciar outros indivíduos a praticar o bem ou causar o mal. Para garantir, tanto quanto possível, que seus esforços sejam utilizados para o bem, eles devem se comprometer a fazer da engenharia de software uma profissão benéfica e respeitada. De acordo com esse compromisso, eles devem aderir ao seguinte Código de Ética e de Prática Profissional.*
>
> *O Código contém oito princípios relacionados ao comportamento e às decisões assumidas pelos engenheiros de software profissionais, inclusive praticantes, educadores, gerentes, supervisores e responsáveis pela definição de políticas, como também estagiários e estudantes da profissão. Os princípios identificam as relações de responsabilidade ética em que indivíduos, grupos e organizações participam, assim como as principais obrigações dentro desse relacionamento. As cláusulas de cada princípio são ilustrações de algumas das obrigações incluídas nessas relações. Essas obrigações estão fundamentadas na humanidade do engenheiro de software, no cuidado especial que é devido ao pessoal afetado pelo trabalho de engenheiros de software e nos elementos singulares da prática de engenharia de software. O Código prescreve essas questões como obrigações de qualquer um que diga ser ou aspira a se tornar um engenheiro de software.*

Em qualquer situação em que pessoas diferentes tenham pontos de vista e objetivos diferentes, é provável que nos defrontemos com dilemas éticos. Por exemplo, se você discordar, em princípio, das políticas dos níveis mais altos de gestão da companhia, como deverá rea-

Código de ética e de prática profissional da engenharia de software

Força tarefa da ACM/IEEE-CS sobre éticas e práticas profissionais da engenharia de software

Preâmbulo

A versão resumida do código apresenta as aspirações em um alto nível de abstração; as cláusulas que estão incluídas na versão integral dão exemplos e detalhes de como essas aspirações modificam o modo pelo qual agimos como profissionais da engenharia de software. Sem essas aspirações, os detalhes podem se tornar muito específicos e tediosos; sem os detalhes, as aspirações podem se tornar aparentemente importantes, mas vazias; juntos, aspirações e detalhes formam um código coeso.

Os engenheiros de software se comprometerão a fazer da análise, da especificação, do projeto, do desenvolvimento, dos testes e da manutenção de software uma profissão benéfica e respeitada. De acordo com seu compromisso com a saúde, a segurança e o bem-estar do público, os engenheiros de software deverão aderir aos seguintes princípios:

1. PÚBLICO – Os engenheiros de software agirão consistentemente com o interesse público.
2. CLIENTE E EMPREGADOR – Os engenheiros de software agirão de uma maneira que esteja em conformidade com os melhores interesses de seus clientes e empregadores e consistente com o interesse público.
3. PRODUTO – Os engenheiros de software deverão assegurar que seus produtos e as alterações a eles relacionadas cumpram o mais alto padrão profissional possível.
4. JULGAMENTO – Os engenheiros de software manterão integridade e independência em seu julgamento profissional.
5. GERENCIAMENTO – Os gerentes e os líderes de engenharia de software adotarão e promoverão uma abordagem ética para o gerenciamento do desenvolvimento e da manutenção do software.
6. PROFISSÃO – Os engenheiros de software fomentarão a integridade e a reputação da profissão, de modo consistente com o interesse público.
7. COLEGAS – Os engenheiros de software serão justos e darão apoio aos seus colegas.
8. PESSOAL – Os engenheiros de software participarão de aprendizado constante com relação à prática de sua profissão e promoverão uma abordagem ética dessa prática.

Figura 1.8 ▲

Código de Ética da ACM/IEEE. (©IEEE/ACM 1999)

gir? É claro que isso depende dos indivíduos em particular e da natureza dessa discordância. É melhor debater um caso defendendo sua posição dentro da organização ou se resignar quanto a esse princípio? Se você sentir que há problemas com um projeto de software, quando deverá revelar isso à gerência? Se discutir a questão enquanto ela for apenas uma suspeita, poderá estar exagerando sua reação a uma situação; se deixar para que a questão seja resolvida muito tarde, então poderá ser impossível resolver as dificuldades.

Esses dilemas éticos permeiam nossa vida profissional e, felizmente, na maioria das vezes, são apenas casos de menor importância ou que podem ser resolvidos sem grandes dificuldades. Quando não podem ser resolvidos, os engenheiros talvez enfrentem um outro problema. Defender seus princípios pode resultar em um pedido de demissão, mas isso pode influenciar outras pessoas, como seu cônjuge ou seus filhos.

Uma situação particularmente difícil para os engenheiros profissionais surge quando o empregador age de um modo que não é ético. Digamos que uma empresa seja responsável pelo desenvolvimento de um sistema crítico de segurança e, devido a pressões quanto a prazos, falsifique os registros de validação de segurança. É responsabilidade do engenheiro manter a confidencialidade ou alertar o cliente ou dar a conhecer, de alguma maneira, que o sistema entregue pode não ser seguro?

O problema nesse caso é que não existem questões absolutas quando se trata de segurança. Embora o sistema possa não ter sido validado de acordo com critérios predeterminados, esses critérios podem ser muito estritos. O sistema pode realmente operar com segurança ao longo de sua vida útil. Há possibilidade, embora adequadamente validado, de o sistema falhar e causar um acidente. A revelação prematura de problemas pode resultar em danos ao empregador e a outros empregados; a falha em não revelar problemas pode resultar em danos a terceiros. Cada um tem de tomar sua própria decisão nesses assuntos. Nesse caso, os danos em potencial, a extensão desses danos e as pessoas que serão afetadas por eles são aspectos que devem influenciar na decisão. Se a situação for muito perigosa, poderá ser justificável divulgar a questão, utilizando a imprensa (digamos). Contudo, sempre devemos tentar resolver a situação visando aos direitos do empregador.

Outra questão ética é a participação no desenvolvimento de sistemas militares e nucleares. Algumas pessoas são profundamente sensíveis a essas questões e não desejam participar de qualquer desenvolvimento de sistema associado a atividades militares. Outras pessoas poderão trabalhar em sistemas militares, mas não em sistemas de armamentos. Por outro lado, há quem considere que a defesa nacional é um princípio primordial e que não tem quaisquer objeções em trabalhar em sistemas de armamentos. A posição ética apropriada, nesse caso, depende inteiramente dos pontos de vista das pessoas envolvidas.

Nessa situação, é importante que tanto o empregador quanto os empregados tornem conhecidos seus pontos de vista, com antecedência. Quando uma organização estiver envolvida em um trabalho no setor militar ou nuclear, a direção precisa especificar que os funcionários devem estar dispostos a aceitar a atribuição de qualquer serviço. Do mesmo modo, se os funcionários são admitidos e deixam claro que não desejam trabalhar nesses sistemas, os empregadores não devem exercer pressão sobre eles para que aceitem esse tipo de trabalho, posteriormente.

A área geral de ética e responsabilidade profissional tem recebido cada vez mais atenção, nos últimos anos. Ela pode ser considerada a partir de uma perspectiva filosófica, em que os princípios básicos de ética são levados em conta e a ética da engenharia de software é discutida com referência a esses princípios básicos. Essa é a abordagem adotada por Laudon (1955) e, com menor ênfase, por Huff e Martin (1995).

Contudo, considero essa abordagem um pouco abstrata e difícil de relacionar com minha experiência do dia-a-dia. Prefiro a abordagem mais concreta incorporada aos códigos de conduta e prática. Acredito que a ética é mais bem discutida em um contexto de engenharia de software, e não como um assunto em sua própria área. Neste livro, portanto, não incluo discussões éticas abstratas, mas, quando apropriado, acrescento exemplos nos exercícios, que possam ser a base de uma discussão ética.

PONTOS-CHAVE

- A engenharia de software é uma disciplina da engenharia que se ocupa de todos os aspectos da produção de software.
- Os produtos de software consistem de programas desenvolvidos e da documentação associada. Os atributos essenciais dos produtos são a facilidade de manutenção, a confiança, a eficiência e a facilidade de uso.
- O processo de software consiste em atividades envolvidas no desenvolvimento de produtos de software. As atividades básicas são a especificação, o desenvolvimento, a validação e a evolução do software.
- Métodos são os meios organizados de produzir software. Eles incluem sugestões sobre o processo a ser seguido, as notações a serem utilizadas, as regras que regem as descrições de sistema produzidas e as diretrizes do projeto.
- As ferramentas CASE são sistemas de software projetados para dar apoio às atividades de rotina no processo de software, tais como editar diagramas de projeto, verificar a consistência de diagramas e manter o controle dos testes de programa que são realizados.
- Os engenheiros de software têm responsabilidades para com a engenharia como profissão e a sociedade. Eles não devem se preocupar exclusivamente com questões técnicas.
- As sociedades profissionais publicam códigos de conduta que estabelecem os padrões comportamentais esperados por seus membros.

LEITURAS SUGERIDAS

Software engineering: an engineering approach. Um texto genérico que inclui uma série de estudos de caso úteis. (J.F. Peters e W. Pedrycz, 2000, John Wiley and Sons.)

"Software engineering Code of Ethics is approved". Um artigo que discute a base do desenvolvimento do Código de Ética da ACM/IEEE e que inclui a versão resumida e a versão completa do Código. (D. Gotterbarn, K. Miller e S. Rogerson, *Comm. ACM*, out. 1999.)

Software engineering: a practitioner's approach. Um livro genérico que examina ampla variedade de tópicos da engenharia de software. (R.S. Pressman, 1997, McGraw-Hill.)

Ethics and computing: living responsibly in a computerized world. Uma boa visão geral do tópico, em conjunto com uma série de artigos mais especializados. (K.W. Bowyer, 1996, IEEE Computer Society Press.)

Professional issues in software engineering. Esse é um excelente livro que discute questões legais e profissionais, assim como questões éticas. (F. Bott, A. Coleman, J. Eaton e D. Rowland, 1995, UCL Press.)

"No silver bullet: essence and accidents of software engineering". Apesar de ser mais antigo, esse artigo é uma boa introdução geral aos problemas da engenharia de software. Sua mensagem essencial não mudou nos últimos 13 anos. (F.P. Brooks, *IEEE Computer*, 20(4), abr. 1987.)

EXERCÍCIOS

1.1 Fazendo referência aos custos de distribuição de software, discutidos na Seção 1.1.7, explique por que é apropriado considerar que o software é mais do que os programas que podem ser executados pelos usuários finais de um sistema.

1.2 Quais são os quatro principais atributos que todos os produtos de software devem ter? Sugira quatro outros atributos que podem ser significativos.

1.3 Qual é a diferença entre um modelo de processo de software e um processo de software? Sugira dois meios pelos quais um modelo de processo de software pode ser útil para a identificação de possíveis melhorias de processo.

1.4 Explique por que os custos dos testes de sistema são particularmente altos para os produtos de software genéricos vendidos para um mercado muito amplo.

1.5 Os métodos de engenharia de software tornaram-se amplamente utilizados apenas quando a tecnologia CASE ficou disponível para oferecer apoio. Sugira cinco tipos de apoio a métodos que podem ser proporcionados pelas ferramentas CASE.

1.6 Além dos desafios dos sistemas legados, da heterogeneidade e da rapidez de fornecimento, identifique outros problemas e desafios que a engenharia de software com certeza deve enfrentar no século XXI.

1.7 Discuta se os engenheiros profissionais devem ter um diploma, assim como médicos e advogados.

1.8 Para cada uma das cláusulas do Código de Ética da ACM/IEEE, mostradas na Figura 1.8, sugira um exemplo apropriado para ilustrá-la.

2 Engenharia de sistemas com base em computadores

Objetivos

O objetivo deste capítulo é introduzir o conceito de engenharia de sistemas com base em computadores e explicar por que o conhecimento da engenharia de sistemas é importante para os engenheiros de software. Depois de ler este capítulo, você poderá:

- saber por que o software em um sistema é afetado por questões de engenharia de sistemas mais amplas;
- obter a introdução ao conceito de propriedades emergentes de sistemas, como confiabilidade, desempenho, segurança e proteção;
- compreender por que o ambiente do sistema deve ser considerado durante o processo de projeto do sistema, e
- compreender a engenharia de sistemas e os processos de suprimento de sistemas.

Sumário

2.1 Propriedades emergentes de sistemas
2.2 Sistemas e seu ambiente
2.3 Modelagem de sistemas
2.4 O processo de engenharia de sistemas
2.5 Suprimento de sistemas

A engenharia de sistemas é a atividade de especificar, projetar, implementar, validar, implantar e manter os sistemas *como um todo*. Os engenheiros de sistemas não se ocupam apenas com o software, mas com as interações de software, hardware e sistema com os usuários e seu ambiente. Eles devem pensar sobre os serviços que o sistema fornece, as restrições dentro das quais o sistema deve ser construído e operado e as interações do sistema com seu ambiente. Os engenheiros de software necessitam de uma compreensão sobre a engenharia de sistemas, porque os problemas de engenharia de software são, freqüentemente, o resultado de decisões da engenharia de sistemas.

Há muitas possíveis definições de um sistema, desde a muito abstrata até a mais concreta, mas uma definição útil seria:

Um sistema é uma coleção significativa de componentes inter-relacionados, que trabalham em conjunto para atingir algum objetivo.

Essa definição geral envolve uma vasta gama de sistemas. Por exemplo, um sistema muito simples, como uma caneta, pode ter apenas três ou quatro componentes de hardware. Por outro lado, um sistema de controle de tráfego aéreo é constituído de milhares de componentes de hardware e software, além dos usuários humanos que tomam decisões com base nas informações sobre o sistema.

Uma característica dos sistemas é que as propriedades e o comportamento dos componentes de sistemas estão intrinsecamente interligados. O funcionamento bem-sucedido de cada componente do sistema depende do funcionamento de alguns outros componentes. Assim, o software somente poderá operar se o processador estiver operacional. O processador poderá realizar computações apenas se o sistema de software, que define essas computações, tiver sido instalado com sucesso.

Os sistemas, freqüentemente, são hierárquicos, no sentido de que eles incluem outros sistemas. Por exemplo, um sistema policial de comando e controle pode ter um sistema de informações geográficas para fornecer detalhes sobre a localização de incidentes. Esses outros sistemas são chamados de *subsistemas*. Uma característica dos subsistemas é que eles podem operar como sistemas independentes. Portanto, o mesmo sistema de informações geográficas pode ser utilizado em diferentes sistemas. Seu comportamento em um sistema específico, contudo, depende de sua relação com outros subsistemas.

As complexas relações entre os componentes em um sistema significam que o sistema é mais do que simplesmente a soma de suas partes. Ele tem propriedades que são do sistema como um todo. Essas *propriedades emergentes* (Checkland, 1981) não podem ser atribuídas a nenhuma parte específica do sistema. Pelo contrário, elas emergem somente quando o sistema como um todo é considerado. Algumas dessas propriedades podem ser derivadas diretamente das propriedades comparáveis de subsistemas, mas, com maior freqüência, elas resultam de complexos inter-relacionamentos de subsistemas, que não podem, na prática, ser compreendidos pela análise de componentes individuais de sistemas.

Alguns exemplos dessas propriedades emergentes são:

1. *O peso total do sistema* Esse é um exemplo de uma propriedade emergente, que pode ser computada a partir de propriedades de componentes individuais.
2. *A confiabilidade do sistema* Depende da confiabilidade dos componentes do sistema e do relacionamento entre eles.
3. *A facilidade de uso de um sistema* Essa é uma propriedade muito complexa, que não depende só do hardware e do software do sistema, mas também dos operadores do sistema e do ambiente em que ele é usado.

Neste livro, preocupo-me com os sistemas com base em computadores, que incluem hardware e software e oferecem uma interface, implementada em software, para usuários huma-

nos. Os engenheiros de software devem ter algum conhecimento de engenharia de sistemas com base em computadores (*computer-based system engineering* – CBSE) (White *et al.*, 1993), devido à importância do software nesses sistemas. Por exemplo, havia menos de 10 megabytes de software no programa espacial US Apollo, que levou o homem à Lua, em 1969; mas há cerca de 100 megabytes de software no programa da estação espacial norte-americana. A engenharia de software é, por conseguinte, fundamental para o sucesso do desenvolvimento de complexos sistemas com base em computadores.

2.1 Propriedades emergentes de sistemas

Como foi discutido na introdução deste capítulo, as propriedades emergentes de um sistema são os atributos do sistema como um todo. Muitas vezes, é difícil prever os valores dessas propriedades emergentes com antecedência. Elas podem ser medidas apenas depois que os subsistemas foram integrados para formar o sistema completo.

Há dois tipos de propriedades emergentes:

1. Propriedades funcionais, que aparecem quando todas as partes de um sistema trabalham em conjunto para atingir algum objetivo. Por exemplo, uma bicicleta tem a propriedade funcional de ser um dispositivo de transporte, uma vez que está montada com todos os seus componentes.

2. Propriedades emergentes não funcionais, como confiabilidade, desempenho, segurança e proteção. Essas propriedades se relacionam com o comportamento do sistema em seu ambiente operacional. Muitas vezes, elas são importantes para os sistemas com base em computadores, uma vez que uma falha em atingir um nível mínimo definido nessas propriedades pode inutilizar o sistema. Algumas funções do sistema podem não ser necessárias para todos os usuários, e, assim, o sistema pode ser aceitável sem elas. Contudo, um sistema que não seja confiável ou que seja muito lento será, provavelmente, rejeitado por todos os usuários.

Para ilustrar a complexidade das propriedades emergentes, considere a confiabilidade de sistema. A confiabilidade é um conceito complexo, que deve sempre ser considerado no nível do sistema, preferivelmente, levando em conta os componentes individuais. Os componentes em um sistema são interdependentes, e, assim, as falhas em um componente podem se propagar pelo sistema e afetar a operação de outros componentes. De modo geral, projetistas de sistemas não podem prever como as conseqüências das falhas podem se propagar pelo sistema, por isso não podem fazer estimativas confiáveis de confiabilidade, a partir de dados sobre a confiabilidade dos componentes do sistema.

Existem três influências estreitamente relacionadas à confiabilidade global de um sistema:

1. *Confiabilidade de hardware* Qual é a probabilidade de um componente de hardware falhar e quanto tempo leva para reparar esse componente?

2. *Confiabilidade de software* Qual é a probabilidade de que um componente de software venha a produzir uma saída incorreta? A falha de software é, usualmente, distinta da falha de hardware, pois o software não se desgasta. Ele pode prosseguir em operação mesmo depois que um resultado incorreto tenha sido produzido. A confiabilidade de software será discutida nos capítulos 16 e 17.

3. *Confiabilidade do operador* Qual é a probabilidade de que o operador de um sistema cometa um erro?

Todos esses itens estão estreitamente ligados. A falha de hardware pode fazer com que sinais falsos sejam gerados, que estão fora da faixa de entradas esperadas pelo software. O software pode, então, se comportar de maneira imprevista. O erro do operador, muitas vezes, ocorre em condições de estresse. Essas condições surgem quando falhas de sistema estão ocorrendo. Os erros de um operador podem, ainda, causar estresse no hardware, gerando mais falhas e assim por diante. Portanto, pode ocorrer uma situação em que uma única falha de subsistema, que seja recuperável, rapidamente desenvolva um problema sério, requerendo a completa interrupção do sistema.

A confiabilidade observada depende do contexto em que o sistema é utilizado. Como já foi discutido, o ambiente do sistema não pode ser especificado com antecedência, nem mesmo os projetistas de sistema podem impor restrições nesses ambientes para os sistemas operacionais. Diferentes sistemas em um ambiente podem reagir a problemas de modo imprevisto, afetando, assim, a confiabilidade do todos esses sistemas. Portanto, mesmo depois que o sistema tenha sido integrado, pode ser difícil fazer medições precisas de sua confiabilidade.

Por exemplo, digamos que um sistema seja projetado para operar à temperatura ambiente normal. Para permitir variações e condições excepcionais, os componentes eletrônicos de um sistema devem ser projetados para operar dentro de uma certa faixa de temperatura, digamos, de 0 a 45 graus. Fora dessa faixa de temperatura, eles se comportarão de maneira imprevisível. Agora, suponha que esse sistema seja instalado perto de um aparelho de ar-condicionado. Se esse aparelho falhar e vazar gás aquecido sobre as peças eletrônicas, esses componentes e todo o sistema poderão falhar.

Se esse sistema tivesse sido instalado em outro local naquele ambiente, não haveria problema algum. Quando o ar-condicionado trabalhou normalmente, não ocorreram problemas. Contudo, em razão da proximidade física dessas máquinas, uma relação de difícil previsão passou a existir entre elas, causando a falha do sistema.

Assim como a confiabilidade, outras propriedades emergentes, como o desempenho e a facilidade de uso, são difíceis de ser avaliadas, mas podem ser medidas depois que o sistema estiver operacional. Propriedades como segurança e proteção, contudo, apresentam problemas diferentes. Nesse caso, não estamos preocupados simplesmente com um atributo relacionado ao comportamento geral do sistema, mas com o comportamento que o sistema não deve exibir. Um sistema seguro é aquele que não permite o acesso não autorizado a seus dados; mas é claramente impossível prever todos os possíveis meios de acesso e proibi-los explicitamente. Portanto, apenas pode ser possível acessar essas propriedades por *default*. Em outras palavras, sabe-se apenas que um sistema não é seguro se alguém o violar.

2.2 Sistemas e seu ambiente

Os sistemas não são entidades independentes, mas existem em um ambiente. Esse ambiente afeta o funcionamento e o desempenho do sistema. Às vezes, o ambiente pode ser considerado um sistema em si mesmo, mas, em geral, ele consiste em uma série de outros sistemas que interagem entre si.

A Figura 2.1 mostra alguns dos sistemas que podem ser incorporados em um edifício de escritórios. Os sistemas de aquecimento, de energia, de iluminação, hidráulicos, de esgoto e de segurança são todos subsistemas dentro do edifício, que é, por sua vez, um sistema. O edifício está localizado em uma rua, que está em uma cidade e assim por diante. O ambiente local de um sistema são os sistemas que estão no mesmo nível. O ambiente total é composto do ambiente local mais o ambiente do sistema que o contém.

Figura 2.1

Hierarquia de sistemas.

Considere o sistema de segurança mostrado no canto inferior esquerdo da Figura 2.1. O ambiente local desse sistema de segurança são os outros sistemas dentro do edifício. O ambiente total inclui todos os outros sistemas do lado de fora do edifício, na rua e na cidade, bem como os sistemas naturais, como o sistema climático.

Há duas razões principais pelas quais o ambiente de um sistema precisa ser compreendido pelos engenheiros de sistema:

1. Em muitos casos, o sistema é concebido para produzir algumas mudanças em seu ambiente. Portanto, um sistema de aquecimento modifica seu ambiente, aumentando ou diminuindo a temperatura. O funcionamento correto do sistema, portanto, somente pode ser avaliado pelos efeitos sobre o ambiente.

2. O funcionamento de um sistema pode ser afetado por mudanças em seu ambiente de um modo que pode ser de difícil previsão. Por exemplo, o sistema elétrico em um edifício pode ser afetado por mudanças ambientais do lado de fora do edifício. Um trabalho realizado na rua pode cortar um cabo de força e o sistema elétrico ficará, portanto, inoperante. Mais sutilmente, uma tempestade pode induzir correntes no sistema elétrico, afetando seu funcionamento normal.

Da mesma maneira que o ambiente físico mostrado na Figura 2.1, os sistemas são também situados em um ambiente organizacional. Isso inclui políticas e procedimentos que são, por sua vez, regidos por questões políticas, econômicas, sociais e ambientais mais amplas. Se o ambiente organizacional não for adequadamente compreendido, os sistemas poderão não atender às necessidades das empresas e ser rejeitados pelos usuários e pelos gerentes organizacionais.

Dentre os fatores humanos e organizacionais provenientes do ambiente do sistema, que afetam o projeto do sistema, destacam-se:

1. *Mudanças no processo* O sistema requer mudanças nos processos de trabalho, no ambiente? Se isso acontecer, certamente será exigido treinamento específico. Se as mudanças forem significativas ou se envolverem pessoas que percam seus empregos, existe o perigo de os usuários resistirem ao sistema.

2. *Mudanças nas tarefas* Os sistemas diminuem a habilidade dos usuários em um ambiente ou faz com que eles modifiquem o modo como trabalham? Se esse for o caso, eles podem resistir ativamente à introdução do sistema na organização. Projetos que envolvem gerentes que precisam mudar seu modo de trabalhar, para a adequação ao sistema de computadores, freqüentemente se ressentem disso. Os gerentes podem pensar que seu status na organização está sendo reduzido, em virtude desse sistema.

3. *Mudanças organizacionais* O sistema modifica a estrutura de poder político em uma organização? Por exemplo, se uma organização depende de um sistema complexo, aqueles que sabem como operar o sistema têm bastante poder político.

Esses fatores humanos, sociais e organizacionais são, muitas vezes, cruciais para determinar se um sistema atinge seus objetivos com sucesso ou não. Infelizmente, a previsão de seus efeitos nos sistemas é muito difícil para os engenheiros que têm pouca experiência em estudos sociais e culturais. Para ajudar a compreender os efeitos dos sistemas nas organizações, várias metodologias têm sido desenvolvidas, como as técnicas sociais de Mumford (Mumford, 1989) e a 'Metodologia de Sistemas Suaves de Checkland' (Checkland, 1981; Checkland e Scholes, 1990). Foram realizados também extensos estudos sociológicos dos efeitos de sistemas com base em computadores no trabalho (Ackroyd *et al.*, 1992).

Como ideal, todo o conhecimento ambiental relevante deve ser incluído na especificação do sistema, de modo que isso seja levado em conta pelos projetistas de sistemas. Na realidade, isso é impossível. Os projetistas de sistemas devem fazer suposições sobre o ambiente com base em outros sistemas equivalentes e no bom senso. Se eles errarem quanto a isso, os sistemas poderão apresentar mau funcionamento, de modo imprevisível. Por exemplo, se os projetistas de um sistema não compreenderem a noção de compatibilidade eletromagnética, o sistema poderá funcionar inadequadamente, quando instalado com outros sistemas que emitam radiação eletromagnética.

2.3 Modelagem de sistemas

Como parte dos requisitos do sistema e da atividade de projetos, o sistema precisa ser modelado como um conjunto de componentes e de relações entre esses componentes. Isso é, normalmente, ilustrado graficamente em um modelo de arquitetura de sistema, que proporciona ao leitor uma visão geral da organização do sistema.

A arquitetura do sistema é, geralmente, retratada como um diagrama de blocos, mostrando os subsistemas principais e as interconexões entre esses subsistemas. Cada subsistema é representado por um retângulo, no diagrama de blocos, e a existência de uma relação entre os subsistemas é indicada por flechas que ligam esses retângulos. Os relacionamentos indicados podem incluir fluxo de dados, uma relação de 'usa'/'usado por' ou algum outro tipo de relação de dependência.

Esse esquema está ilustrado na Figura 2.2, que mostra a decomposição de um sistema de alarme contra intrusos em seus componentes principais. O diagrama de blocos deve ser complementado por breves descrições de cada subsistema, como indica a Figura 2.3.

Figura 2.2 ▶

Um sistema simples de alarme contra intrusos.

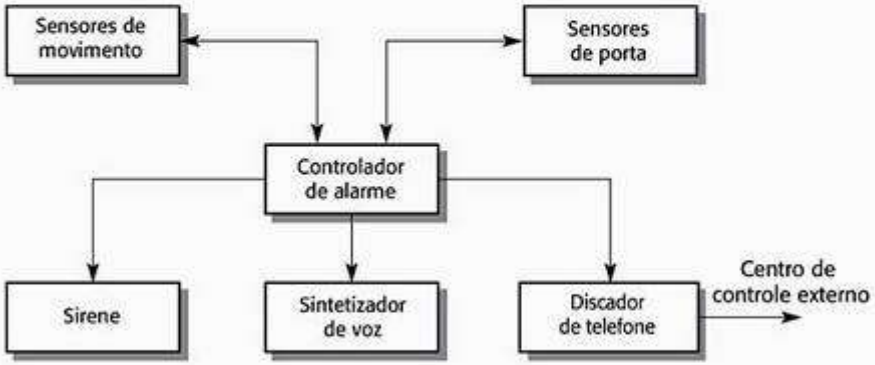


Figura 2.3 ▶

Funcionalidade de subsistemas no sistema de alarme contra intrusos.

Subsistemas	Descrição
Sensor de movimento	Detecta movimento nos cômodos monitorados pelo sistema.
Sensor de porta	Detecta a abertura de porta nas portas externas do edifício.
Controlador de alarme	Controla a operação do sistema.
Sirene	Emite um aviso sonoro quando um intruso é detectado.
Sintetizador de voz	Sintetiza uma mensagem de voz dando a localização do possível intruso.
Discador de telefone	Faz as chamadas externas para avisar a segurança, a polícia etc.

Com esse nível de detalhes, o sistema é decomposto em subsistemas. Cada subsistema pode ser representado da mesma maneira, até que o sistema seja decomposto em componentes funcionais. Os componentes funcionais são aqueles que, quando vistos a partir da perspectiva do subsistema, proporcionam uma única função. Por outro lado, um subsistema usualmente é multifuncional. Naturalmente, quando visto sob outra perspectiva (ou seja, a do fabricante do componente), um componente funcional pode ele próprio ser um sistema individual.

Historicamente, o modelo de arquitetura de sistemas foi utilizado para identificar componentes de hardware e de software que possam ser desenvolvidos em paralelo. Contudo, essa distinção hardware/software está se tornando irrelevante. Atualmente, quase todos os componentes possuem alguns recursos de computação embutidos. Por exemplo, uma rede que conecta máquinas consistirá em cabos físicos, mais os repetidores e gateways de rede. Os repetidores e os gateways incluem processadores e software para dirigir esses processadores, assim como componentes eletrônicos especializados.

No que se refere à arquitetura, atualmente é mais apropriado classificar os subsistemas de acordo com sua função, antes de serem tomadas decisões sobre as vantagens e as desvantagens referentes a hardware/software. A decisão sobre se uma função deve ser fornecida em hardware ou software pode ser regida por fatores não técnicos, como a disponibilidade de componentes COTS (*commercial off-the-shelf* – sistemas comerciais de prateleira) ou o tempo disponível para desenvolver o componente.

Os diagramas de blocos podem ser utilizados para sistema de todos os tamanhos. A Figura 2.4 ilustra a arquitetura de um sistema muito maior para o controle de tráfego aéreo. Existem vários subsistemas importantes, que são por si só grandes sistemas. O fluxo de informações entre esses subsistemas é mostrado pelas setas que conectam esses sistemas.

2.3.1 Componentes funcionais de sistemas

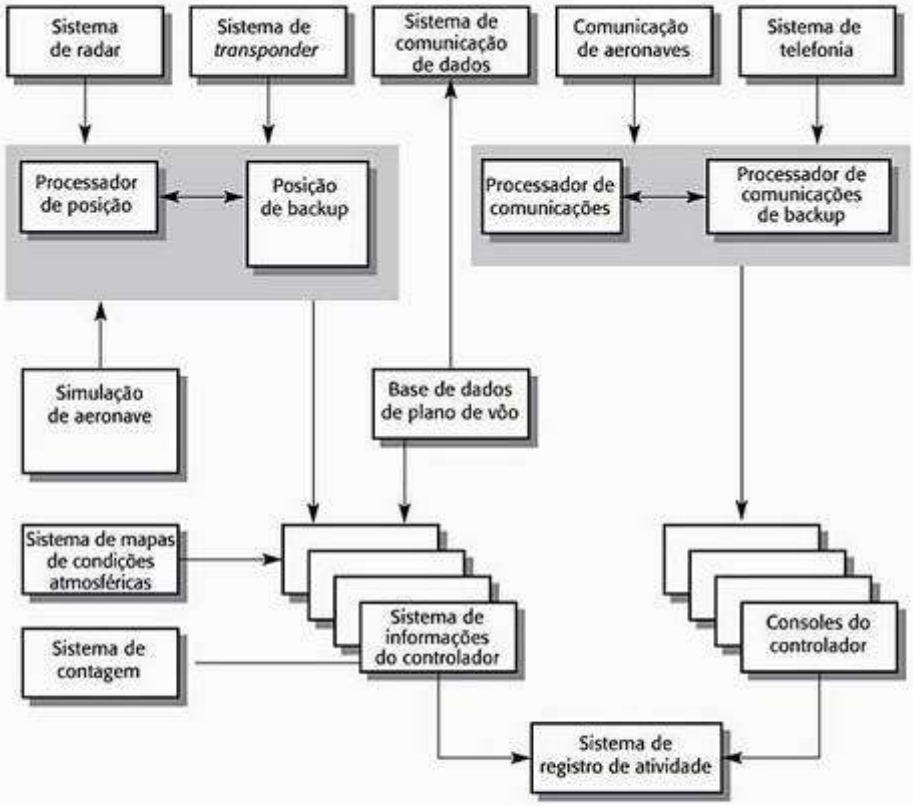
Como foi discutido na seção anterior, uma arquitetura de sistemas deve ser projetada em termos de subsistemas funcionais, sem considerar se eles são subsistemas de hardware ou de software. Mais propriamente, os componentes funcionais em um sistema podem ser classificados de acordo com uma série de diferentes títulos:

1. *Componentes de sensores* coletam informações do ambiente do sistema. Dentre os exemplos de componentes de sensores destacam-se os radares em um sistema de controle de tráfego aéreo, os sensores de posicionamento de papel em uma impressora a laser e um par termoelétrico em uma fornalha.

2. *Componentes de atuadores* causam alguma mudança no ambiente do sistema. Dentre os exemplos de atuadores destacam-se as válvulas que se abrem ou se fecham para aumentar e diminuir a vazão de líquido em um tubo, as superfícies de vôo em uma aeronave, que controlam o ângulo de vôo, e o mecanismo de alimentação de papel, em uma impressora a laser, que move o papel ao longo da luz de escaneamento.

Figura 2.4 ▶

A arquitetura de um sistema de controle de tráfego aéreo.



3. *Componentes de computação* são componentes que, considerando alguma entrada (input), realizam algumas computações sobre essa entrada e produzem uma saída (output). Como exemplo de um componente de computação está o processador de ponto flutuante que realiza computações sobre números reais.

4. *Componentes de comunicação* são componentes cuja função é permitir que outros componentes do sistema se comuniquem entre si. Um exemplo de componente de comunicação é uma Ethernet que conecta diferentes computadores em um edifício.

5. *Componentes de coordenação* são os componentes de sistema cuja função é coordenar a operação de outros componentes. Como exemplo de componente de coordenação está um escalador em um sistema de tempo real, que decide quando os diferentes processos devem ser escalados para serem executados em um processador.

6. *Componentes de interface* são os componentes que transformam a representação utilizada por um componente de sistema em uma representação empregada por outro componente. Como exemplo disso está um componente de interface humana, que considera algum modelo de sistema e o exibe para o operador humano. Outro exemplo é um conversor analógico-digital, que converte uma entrada analógica em uma saída digital.

A Figura 2.5 explica quais desses diferentes tipos de componentes são utilizados na arquitetura do sistema de alarme, ilustrado na Figura 2.2.

Naturalmente, não há nenhuma divisão rígida e rápida entre os diferentes tipos de componentes. Nos tipos de sistemas em que os engenheiros de software estarão provavelmente envolvidos, a maior parte dos componentes incluirá software embutido. O software será, em geral, utilizado para controlar o sistema como um todo.

Essas classificações de componentes são úteis durante o projeto de um sistema. A maioria dos sistemas inclui todos esses tipos de componentes, e é preciso explicitamente identificar cada tipo de componente a partir dos requisitos. Se um ou mais tipos de componentes estiverem faltando, isso sugerirá que algo pode ter sido deixado fora do projeto.

Figura 2.5

Tipos de componentes no sistema de alarme contra intrusos.

Tipos de componentes	Componentes	Função
Sensor	Sensor de movimento, sensor de porta	Detecta movimento em um espaço protegido, detecta abertura de uma porta protegida.
Atuador	Sirene	Aviso sonoro de intrusão.
Comunicação	Discador de telefone	Aciona o centro de controle externo para emitir um aviso de intrusão. Recebe comandos do centro de controle.
Coordenação	Controlador de alarme	Coordena todos os componentes do sistema. Atua nos comandos do painel de controle e do centro de controle.
Interface	Sintetizador de voz	Sintetiza mensagem dando a localização da intrusão.

2.4 O processo de engenharia de sistemas

As fases do processo de engenharia de sistemas são mostradas na Figura 2.6. Esse processo foi uma importante influência no modelo 'em cascata' do processo de software, que discutiremos no Capítulo 3.

Existem importantes distinções entre o processo de engenharia de sistemas e o processo de desenvolvimento de software:

1. *Envolvimento interdisciplinar* Muitas diferentes disciplinas da engenharia podem estar envolvidas na engenharia de sistemas. Há uma imensa possibilidade de mal-entendidos, devido à terminologia diferente utilizada por diferentes engenheiros.

2. *Possibilidade reduzida de refazer o trabalho durante o desenvolvimento de sistemas* Uma vez que algumas decisões da engenharia de sistemas, como a localização de radares em um sistema de controle de tráfego aéreo (*air traffic control* – ATC), sejam tomadas, elas são muito dispendiosas para sofrerem mudanças. Refazer o trabalho de projetar o sistema para resolver esses problemas raramente é possível. Uma razão pela qual o software se tornou tão importante para os sistemas é que ele permite flexibilidade, uma vez que podem ser feitas modificações durante o desenvolvimento de sistema, em resposta a novos requisitos.

A engenharia de sistemas é uma atividade interdisciplinar que envolve equipes com diferentes formações técnicas. As equipes de engenharia de sistemas são necessárias, por causa do amplo conhecimento exigido para considerar todas as implicações das decisões referentes a projetos de sistemas. Considere um sistema de ATC, que utiliza radares e outros sensores para determinar a posição da aeronave (veja a Figura 2.4). A Figura 2.7 mostra algumas das diferentes disciplinas que podem estar envolvidas com a equipe de engenharia de sistemas.

Figura 2.6

O processo de engenharia de sistemas.

Para muitos sistemas, há praticamente infinitas possibilidades de compromissos entre diferentes tipos de subsistemas. Diferentes disciplinas de engenharia devem entrar em acordo, para decidir como a funcionalidade deve ser oferecida. Muitas vezes, não há nenhuma decisão 'correta' sobre como um sistema deve ser decomposto. Em vez disso, existem várias alternativas possíveis. A seleção de uma dessas alternativas pode não necessariamente ser feita por razões técnicas. Digamos que uma alternativa em um sistema de ATC seja construir novos radares, em vez de adequar as instalações já existentes. Se os engenheiros civis envolvidos nesse processo não tiverem outro trabalho importante, eles poderão favorecer essa alternativa, pois isso faz com que mantenham suas atividades. Eles podem, então, explicar essa escolha utilizando argumentos técnicos.

Em razão de o software ser inerentemente flexível, muitos problemas inesperados são deixados para os engenheiros de software resolverem. Suponhamos que a localização de um radar seja tal que um 'ruído' de imagem. É impraticável remover o radar; assim, é necessário um outro meio de remover esse 'ruído'. A solução pode ser aumentar a capacidade de processamento de imagem do software, para remover os 'ruídos' de imagens. Isso pode exigir o aumento da capacidade do processador no sistema, o que pode ser uma tarefa difícil.

Os engenheiros de software, muito freqüentemente, precisam resolver o problema de aumentar a capacidade do software, sem elevar o custo do hardware. Muitas das chamadas 'falhas de software' não foram conseqüência de problemas inerentes ao software, mas resultado da tentativa de modificar o software para adequar modificações nos requisitos da engenharia de sistemas. Um bom exemplo disso foi a falha do sistema de bagagens do aeroporto de Denver (Swartz, 1996).

2.4.1 Definição de requisitos de sistema

A atividade de definição dos requisitos do sistema se destina a levantar os requisitos do sistema como um todo. Assim como na análise dos requisitos do software, o processo envolve consultas com os clientes e usuários finais do sistema. Essa fase de definição de requisitos normalmente se concentra em derivar três tipos de requisitos:

1. *Requisitos funcionais abstratos* As funções básicas que o sistema deve oferecer são definidas em um nível abstrato. A especificação detalhada de requisitos funcionais ocorre no nível do subsistema. Por exemplo, no sistema de controle de tráfego aéreo, essa atividade de definição de requisitos, provavelmente, identificaria a necessidade de uma base de dados de plano de vôo para armazenar os planos de vôos de todas as aeronaves que ingressassem no espaço aéreo controlado. Contudo, os detalhes da base de dados não seriam especificados, a menos que afetassem os requisitos de outros subsistemas.
2. *Propriedades de sistemas* Estas são propriedades emergentes de sistemas não funcionais, como foi discutido anteriormente. Elas podem incluir propriedades como disponibilidade, desempenho, segurança, entre outras. Essas propriedades de sistema não funcionais afetam os requisitos de todos os subsistemas.

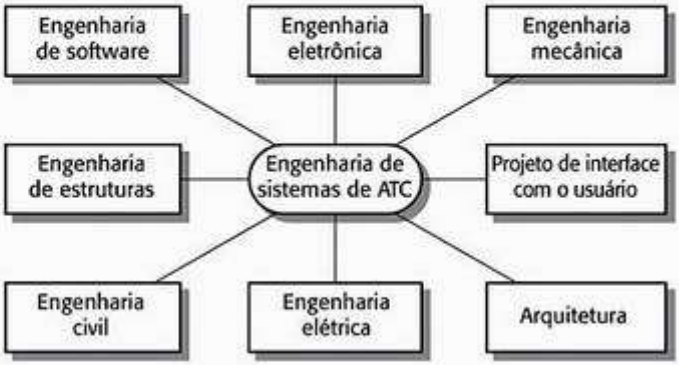


Figura 2.7 Envolvimento interdisciplinar na engenharia de sistemas.

3. *Características que o sistema não deve exibir* Às vezes, é tão importante especificar o que o sistema não deve fazer quanto o que ele deve fazer. Por exemplo, em um sistema de controle de tráfego aéreo, poderia ser especificado que o sistema não deve apresentar muitas informações ao controlador.

Uma parte importante da fase de definição de requisitos é estabelecer um conjunto de objetivos gerais que o sistema deve cumprir. Isso não deve necessariamente ser expresso em termos da funcionalidade do sistema, mas precisa definir por que o sistema está sendo obtido para um ambiente específico.

Para ilustrar a distinção entre esses aspectos, considere um sistema para um prédio de escritórios, que deve oferecer proteção contra incêndios e detecção de intrusos. Uma declaração de objetivos com base na funcionalidade do sistema seria:

Fornecer um sistema de alarme contra incêndios e contra intrusos para o edifício, com o objetivo de divulgar avisos internos e externos referentes a incêndios ou à entrada de pessoa não autorizada.

Esse objetivo declara explicitamente que é preciso existir um sistema de alarmes para avisar sobre ocorrências indesejáveis. Essa declaração poderá ser apropriada, se já existir um sistema de alarme que deverá ser substituído. Por outro lado, uma declaração mais ampla de objetivos pode ser a seguinte:

Assegurar que o trabalho normal realizado no edifício não seja seriamente perturbado por ocorrências como incêndios e a entrada de pessoas não autorizadas.

Enunciando o objetivo desse modo, tanto se amplia quanto se limita algumas opções de projeto, permitindo que a proteção contra intrusos seja feita com o uso de tecnologia sofisticada de fechaduras, sem quaisquer alarmes internos. Essa definição pode excluir o uso de *sprinklers* para proteção contra incêndios e afetar o sistema elétrico do edifício e perturbar seriamente o trabalho em andamento.

Uma dificuldade fundamental para que sejam estabelecidos os requisitos de sistemas é que os problemas pelos quais os sistemas complexos são usualmente construídos, a fim de proporcionar soluções específicas, são em geral 'problemas intrincados' (Rittel e Webber, 1973). Um 'problema intrincado' é aquele que é tão complexo, com o qual há tantas entidades relacionadas, que não existe nenhuma especificação definitiva do problema. A verdadeira natureza do problema somente emerge à medida que uma solução é desenvolvida. Um exemplo extremo de um 'problema intrincado' é o planejamento relacionado a terremotos. Ninguém pode prever precisamente onde será o epicentro de um terremoto, quando ele ocorrerá, que efeito produzirá no ambiente local etc. Portanto, não podemos especificar completamente como lidar com um grande terremoto. O problema só poderá ser enfrentado depois que tiver acontecido.

2.4.2 Projeto de sistemas

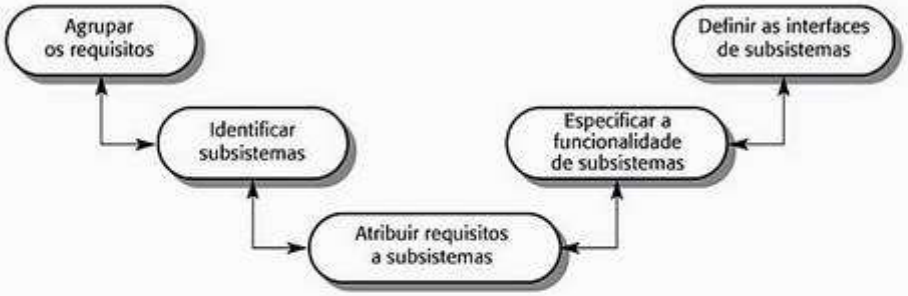
O projeto de sistemas (Figura 2.8) se ocupa de como a funcionalidade de sistemas deve ser fornecida pelos diferentes componentes de sistemas. As atividades envolvidas nesse processo são:

1. *Agrupar os requisitos* Os requisitos são analisados e reunidos em grupos relacionados. Existem, usualmente, várias opções de agrupamento possíveis, e uma série de alternativas podem ser produzidas nesse estágio do processo.

2. *Identificar subsistemas* São identificados diferentes subsistemas que podem, individual ou coletivamente, atender aos requisitos. Os grupos de requisitos são geralmente relacionados a subsistemas, e, assim, essa atividade e o agrupamento dos requisitos podem ser fundidos. Contudo, a identificação de subsistemas também pode ser influenciada por outros fatores organizacionais e ambientais.

Figura 2.8

O processo de projeto de sistemas.



3. *Atribuir requisitos a subsistemas* Os requisitos são atribuídos a subsistemas. Em princípio, isso deverá ser direto, se o agrupamento dos requisitos for utilizado para orientar a identificação de subsistemas. Na prática, nunca existe uma correspondência nítida entre os agrupamentos de requisitos e os subsistemas identificados. As limitações dos subsistemas adquiridos externamente (COTS, veja a Seção 2.4.3) podem significar que os requisitos precisam ser modificados.

4. *Especificar a funcionalidade de subsistemas* As funções específicas fornecidas por cada subsistema são definidas. Isso pode ser visto como parte da fase de projeto de sistemas ou, se o subsistema for um sistema de software, como parte da atividade de especificação de requisitos para esse sistema. Os relacionamentos entre subsistemas devem também ser identificados nesse estágio.

5. *Definir as interfaces de subsistemas* Essa tarefa envolve a definição das interfaces que são fornecidas e exigidas por cada subsistema. Uma vez que essas interfaces tenham sido especificadas, torna-se possível o desenvolvimento paralelo de subsistemas.

Como indicam as setas bidirecionais, na Figura 2.8, existe grande feedback e iteração de um estágio para o outro, nesse processo de projeto. À medida que problemas e questões surgem, freqüentemente é necessário o 'retrabalho' relacionado a estágios anteriores.

Para quase todos os sistemas, existem muitos projetos possíveis, que podem ser desenvolvidos; eles abordam uma gama de soluções com diferentes combinações de hardware, software e operações humanas. A solução escolhida para prosseguir com o desenvolvimento pode ser a solução técnica mais apropriada, que atenda aos requisitos. Contudo, em muitos casos, influências organizacionais e políticas mais amplas influenciam a escolha da solução. Por exemplo, se o sistema for projetado para uso pelo governo, poderão ser escolhidos fornecedores nacionais, em vez de fornecedores estrangeiros, mesmo que o produto nacional seja tecnicamente inferior.

2.4.3 Desenvolvimento de subsistemas

Durante o desenvolvimento de subsistemas, são implementados os subsistemas identificados na fase de projeto do sistema. Isso pode envolver iniciar outro processo de engenharia de sistemas para subsistemas individuais. Quando um subsistema é um sistema de software, pode ser iniciado um processo de software que envolva requisitos, projeto e implementação, entre outros.

Ocasionalmente, o processo de desenvolvimento desenvolverá todos os subsistemas a partir do zero. Contudo, normalmente, alguns subsistemas são sistemas comerciais, de prateleira (COTS – *commercial off-the-shelf*), que são adquiridos para a integração no sistema. De modo geral, é muito mais barato adquirir produtos já existentes do que desenvolver componentes com propósito especial. Nesse estágio, a atividade de projeto pode ser reiniciada, a fim de acomodar a inclusão de um componente adquirido. Os sistemas COTS podem não atender exatamente aos requisitos, mas se 'produtos de prateleira' estão disponíveis, muitas vezes vale a pena o esforço de repensar o projeto.

Diferentes subsistemas são normalmente desenvolvidos em paralelo. Quando são constatados problemas que atravessam os limites de subsistemas, deve ser efetuado um pedido de modificação do sistema. Quando os sistemas envolvem extensiva engenharia de hardware, fazer modificações depois que a fabricação foi iniciada, geralmente, é muito dispendioso. Freqüentemente, é preciso encontrar alternativas que compensem o problema. Essas alternativas, em geral, envolvem modificações no software, em razão de sua flexibilidade. Isso leva a mudanças nos requisitos de software e, assim, como foi discutido no Capítulo 1, é importante projetar software objetivando também a facilidade de realizar mudanças.

2.4.4 Integração de sistemas

A integração de sistemas envolve adotar subsistemas desenvolvidos independentemente e agrupá-los, para que formem um sistema completo. A integração pode ser feita utilizando uma abordagem do tipo 'big bang', em que todos os subsistemas são integrados ao mesmo tempo. Contudo, por razões técnicas e gerenciais, um processo de integração em que os subsistemas são integrados um de cada vez é a melhor abordagem a ser adotada.

Esse processo de integração em fases é a abordagem mais apropriada por duas razões:

1. Em geral, é impossível programar todos os desenvolvimentos de subsistemas, de modo que todos os desenvolvimentos sejam concluídos ao mesmo tempo.
2. A integração incremental reduz os custos de localização de erros. Se muitos subsistemas são integrados simultaneamente, um erro que surgir durante os testes pode estar localizado em qualquer um desses subsistemas. Quando um único subsistema é integrado em um sistema já em operação, os erros que podem ocorrer, provavelmente, estarão no novo subsistema integrado ou nas interações entre os subsistemas existentes e o novo subsistema.

As falhas de subsistemas que são conseqüência de suposições inválidas sobre outros subsistemas, com freqüência, são reveladas durante a integração de sistemas. Isso pode levar a disputas entre os vários fornecedores responsáveis por diferentes subsistemas. Quando são descobertos problemas na interação de subsistemas, os diferentes fornecedores podem discutir sobre quem é responsável pelo problema. As negociações sobre como resolver o problema podem levar várias semanas ou meses, até que se chegue a uma solução.

2.4.5 Instalação de sistemas

Durante a instalação de um sistema, o sistema é colocado no ambiente no qual deverá operar. Embora possa parecer um processo simples, muitos problemas podem surgir, o que significa que a instalação de um sistema complexo pode levar meses, ou mesmo anos.

Exemplos desses problemas são:

1. O ambiente no qual o sistema deve ser instalado não é o mesmo que o ambiente considerado pelos desenvolvedores do sistema. Esse é um problema comum quando sistemas de software são instalados. Por exemplo, o sistema pode utilizar funções fornecidas por uma versão específica do sistema operacional. Essas funções podem não ser idênticas na versão do sistema operacional, no ambiente de instalação. Quando o sistema é instalado, ele pode não operar de maneira nenhuma ou pode operar de um modo que não tenha sido previsto pelos seus desenvolvedores.
2. Os usuários em potencial do sistema podem ser hostis à sua introdução. O sistema pode reduzir suas responsabilidades ou o número de cargos em uma organização. As pessoas podem, portanto, deliberadamente se recusar a cooperar com os instaladores de sistemas. Por exemplo, elas podem se recusar a participar do treinamento de operador ou negar acesso a informações essenciais para a instalação do sistema.

3. Um novo sistema pode ter de coexistir com um sistema anterior, até que a organização esteja certa de que o novo sistema opera adequadamente. Isso causará problemas específicos de instalação, se os sistemas não forem completamente independentes, mas compartilharem alguns componentes. Pode ser impossível instalar o novo sistema sem desinstalar o sistema antigo. Os testes do novo sistema podem, portanto, ocorrer somente nas ocasiões em que o sistema já existente não estiver em uso.

4. Pode haver problemas físicos de instalação. Podem surgir dificuldades para adequar um novo sistema em um edifício já existente, como pode não haver espaço suficiente nos dutos já existentes para os cabos de rede; pode ser necessário instalar aparelhos de ar-condicionado, a mobília pode não ser suficientemente grande e assim por diante. Se a instalação tiver de ocorrer em um prédio histórico, as modificações prediais poderão ser inteiramente proibidas.

2.4.6 Operação de sistemas

Uma vez instalado o sistema, ele é colocado em operação. A operação do sistema pode envolver a organização de sessões de treinamento para operadores e a modificação do processo normal de trabalho, a fim de fazer uso efetivo do novo sistema. Problemas não detectados podem surgir, nesse estágio, porque a especificação do sistema pode conter erros ou omissões. Embora o sistema possa operar segundo as especificações, suas funções podem não atender às necessidades reais de operação. Conseqüentemente, o modo como o sistema é utilizado pode não ser o previsto pelos projetistas do sistema.

Depois que o sistema entrar em operação, poderá surgir o problema de operar o novo sistema com os sistemas já existentes. Podem ocorrer problemas físicos de incompatibilidade. Talvez existam dificuldades em transferir dados de um sistema para outro. Problemas mais sutis podem ocorrer quando interfaces com os usuários radicalmente diferentes são oferecidas por diferentes sistemas. A introdução do novo sistema pode aumentar a taxa de erros de operador nos sistemas já existentes, à medida que os operadores 'misturam' os comandos das interfaces com o usuário.

2.4.7 Evolução de sistemas

Sistemas grandes e complexos têm um longo tempo de duração, e nesse período eles precisam evoluir para corrigir erros nos requisitos de sistema originais e atender aos novos requisitos que surgiram. Os computadores de sistemas, provavelmente, deverão ser substituídos por máquinas mais novas e mais rápidas. A organização que utiliza o sistema pode se reorganizar e, como conseqüência, utilizar o sistema de modo diferente. O ambiente externo do sistema pode ser modificado, forçando, assim, alterações no sistema.

A evolução do sistema, assim como a evolução de software (discutida na Parte 7), é inerentemente dispendiosa por várias razões:

1. As mudanças propostas precisam ser analisadas muito cuidadosamente, tanto sob uma perspectiva de negócios como de acordo com os aspectos técnicos. Essas mudanças devem ser aprovadas por uma série de pessoas antes de serem efetivadas.

2. Como os subsistemas nunca são inteiramente independentes, as mudanças em um subsistema podem afetar adversamente o desempenho e o comportamento de outros subsistemas. Portanto, podem se tornar necessárias outras mudanças nesses subsistemas.

3. As razões das decisões de projeto originais muitas vezes não ficam registradas. Os responsáveis pela evolução de sistemas precisam entender por que foram tomadas determinadas decisões de projeto.

4. À medida que os sistemas 'envelhecem', sua estrutura se torna tipicamente corrompida por mudanças, de modo que aumentam os custos referentes a novas mudanças.

À medida que a sociedade se torna cada vez mais dependente de sistemas de vários tipos, estão aumentado os esforços devotados à evolução de sistemas já existentes, em vez de ao desenvolvimento de novos sistemas. Os sistemas existentes, que precisam ser mantidos, são atualmente chamados, às vezes, de *sistemas legados*. Os sistemas legados serão discutidos no Capítulo 26.

2.4.8 Desativação de sistemas

Desativar um sistema significa tirar o sistema de serviço, após terminar seu tempo de vida útil operacional. Algumas vezes, essa é uma tarefa direta e simples, mas alguns sistemas podem conter materiais que são potencialmente prejudiciais ao meio ambiente. A atividade de engenharia de sistemas deve prever essa desativação e considerar os problemas referentes ao descarte de materiais durante a fase de projeto. Por exemplo, o uso de produtos químicos tóxicos deve ficar limitado a módulos selados, que possam ser removidos como unidades isoladas e reprocessados.

No que diz respeito a software, não existe, com certeza, nenhum problema físico quanto à desativação. Contudo, alguma funcionalidade de software pode ser incorporada em um sistema, a fim de facilitar o processo de desativação. Por exemplo, um software pode ser utilizado para monitorar o estado de outros componentes do sistema. Quando o sistema for desativado, os componentes que não estiverem comprometidos poderão, portanto, ser identificados e reutilizados em outros sistemas.

Se os dados no sistema que está sendo desativado devem ser mantidos pela organização, eles precisam ser convertidos para uso por outro sistema. Isso pode, muitas vezes, envolver custos significativos, pois a estrutura de dados pode estar implicitamente definida no próprio software. No Capítulo 28, serão abordados alguns desses problemas de reengenharia de dados.

2.5 Suprimento de sistemas

Os clientes de complexos sistemas com base em computadores são, em geral, grandes organizações, como as organizações militares, governamentais e de serviços de emergência. O sistema pode ser adquirido como um todo, como partes separadas que vão sendo integradas ou ser especialmente projetado e desenvolvido. Para sistemas de grande porte, escolher entre essas opções pode levar vários meses ou anos. O processo de suprimento de sistemas se ocupa de tomar decisões sobre a melhor maneira de uma organização adquirir um sistema e de decidir sobre os melhores fornecedores desse sistema.

O processo de suprimento está estreitamente relacionado ao processo de engenharia de sistemas. Algumas especificações de sistemas e projetos de arquitetura são feitas antes que essas decisões de suprimento sejam tomadas. Existem duas razões principais para isso:

1. Para comprar ou aceitar um contrato para projetar e construir um sistema, é preciso definir uma especificação de alto nível sobre o que esse sistema deve fazer.
2. Quase sempre é mais barato comprar um sistema do que projetá-lo, fabricá-lo e construí-lo como um projeto separado. Algum projeto de arquitetura é necessário, para identificar os subsistemas que podem ser comprados, em lugar de precisar projetá-los especialmente e fabricá-los.

Os sistemas grandes e complexos normalmente consistem em uma combinação de componentes 'de prateleira' (COTS) e de componentes construídos especialmente. Uma razão pela qual, cada vez mais, os softwares são incluídos em sistemas é que isso propicia maior utilização de componentes de hardware já existentes, com o software atuando como um 'elo' para fazer essas diferentes peças de hardware trabalharem em conjunto de maneira eficiente.

A necessidade de desenvolver esse 'elo' é uma razão pela qual a economia de utilizar componentes de prateleira, às vezes, não é tão grande quanto prevista. O uso de sistemas COTS será discutido no Capítulo 14.

A Figura 2.9 mostra o processo de suprimento, tanto para sistemas existentes quanto para sistemas que precisam ser projetados especialmente. Alguns pontos importantes sobre o processo mostrado nesse diagrama são:

1. Os componentes de prateleira normalmente não atendem exatamente aos requisitos, a menos que os requisitos tenham sido especificados tendo esses componentes em mente. Portanto, escolher um sistema significa encontrar a concordância mais próxima entre os requisitos do sistema e as facilidades oferecidas pelos sistemas de prateleira. Os requisitos têm, então, de ser modificados. Isso pode provocar efeitos em outros subsistemas.
2. Quando um sistema deve ser construído especialmente, a especificação de requisitos atua como a base de um contrato de suprimento do sistema. É, portanto, um documento legal e técnico.
3. Depois de ser escolhido um fornecedor para construir um sistema, há um outro período de negociação de contrato, em que mais mudanças nos requisitos podem ser combinadas e questões como o custo de mudanças discutidas.

A maioria dos subsistemas de hardware e muitos dos subsistemas de software, como os sistemas de gerenciamento de base de dados, não são desenvolvidos especialmente, quando são incluídos em um sistema maior. Preferivelmente, os subsistemas existentes são utilizados como se encontram ou são adaptados para uso naquele sistema.

Muito poucas organizações têm, por si só, a capacidade de projetar, fabricar e testar os componentes de um sistema complexo e grande. Esse fornecedor, que comumente é chamado de contratante principal, pode contratar externamente o desenvolvimento de diferentes subsistemas para uma série de diversos subcontratantes (Figura 2.10). Para grandes sistemas, como os sistemas de controle de tráfego aéreo, um grupo de fornecedores pode formar um consórcio para apresentar uma proposta de contrato. O consórcio deverá incluir todas as capacitações requeridas para esse tipo de sistema e, assim, pode incluir fornecedores de hardware de computador, desenvolvedores de software, fornecedores de periféricos e fornecedores de equipamentos especiais, como radares.

Esse modelo de contratante/subcontratante minimiza o número de organizações com as quais o comprador tem de lidar. Os subcontratantes projetam e constroem partes do sistema, segundo uma especificação definida pelo contratante principal. Uma vez completadas, essas diferentes partes são integradas pelo contratante principal. E, então, elas são entregues ao cliente comprador do sistema. Dependendo do contrato, o comprador pode permitir ao contratante principal uma livre escolha de subcontratantes ou pode exigir que o contratante principal faça a escolha de subcontratantes a partir de uma lista aprovada.

Figura 2.9 ▼

O processo de suprimento de sistemas.

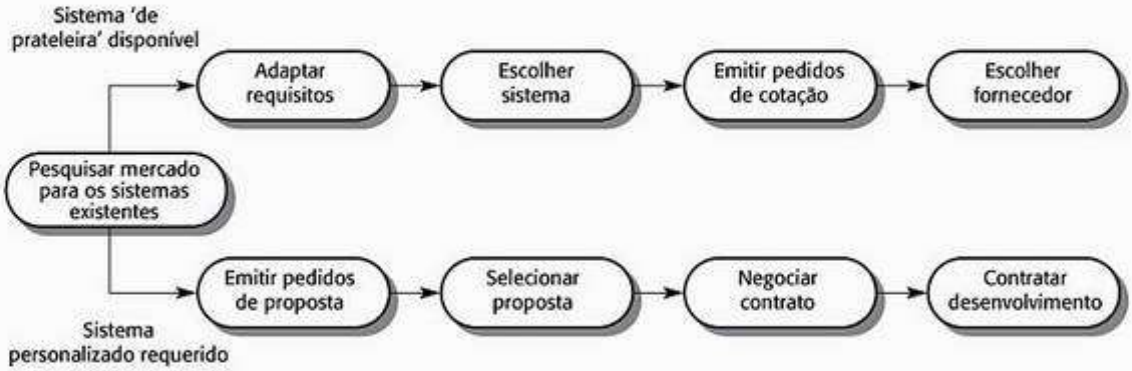


Figura 2.10

O modelo de contratante/subcontratante.

PONTOS-CHAVE

- A engenharia de sistemas é um processo complexo e difícil, que requer a entrada de uma série de disciplinas de engenharia.
- As propriedades emergentes de um sistema são propriedades características do sistema como um todo, e não de suas partes componentes. Elas incluem propriedades como desempenho, confiabilidade, facilidade de uso, segurança e proteção. O sucesso ou o fracasso de um sistema é também dependente dessas propriedades emergentes.
- As arquiteturas de sistema são normalmente descritas com o uso de diagramas de blocos, mostrando os subsistemas principais e suas relações.
- Dentre os tipos de componentes funcionais de sistemas destacam-se os componentes de sensores, os componentes de atuadores, os componentes de computação, os componentes de coordenação, os componentes de comunicação e os componentes de interface.
- O processo de engenharia de sistemas abrange a especificação, o projeto, o desenvolvimento, a integração e os testes. É particularmente importante a integração de sistemas em que subsistemas de diferentes fornecedores devem ser ajustados para trabalharem em conjunto.
- O processo de suprimento de sistemas envolve especificar o sistema, emitir um pedido de propostas, escolher um fornecedor e então estabelecer um contrato para o sistema. Em geral, algumas partes de grandes sistemas com base em computadores são adquiridas como componentes de prateleira (COTS).

LEITURAS SUGERIDAS

Systems engineering: coping with complexity. Na época em que eu estava escrevendo este livro, este era o melhor livro de engenharia de sistemas disponível. Ele aborda os processos de engenharia de sistemas, apresentando bons capítulos sobre requisitos, arquitetura e gerenciamento de projetos. (R. Stevens, P. Brook, K. Jackson e S. Arnold, 1998, Prentice Hall.)

"Airport 95: automated baggage system". Um estudo de caso excelente e de fácil leitura sobre o que pode ocorrer de errado em um projeto de engenharia de sistemas e como o software pode 'levar a culpa' por fracassos mais gerais de sistemas. (*ACM Software Engineering Notes*, 21 mar. 1996.)

"Systems engineering of computer-based systems". Esse artigo oferece boa visão geral da engenharia de sistemas com base em computadores e exige o estabelecimento de uma disciplina de ECBS (*engineering of computer-based systems*), isto é, da engenharia de sistemas com base em computadores. (S. White *et al.*, *IEEE Computer*, 26(11), nov. 1993.)

Systems engineering: principles and practice of computer-based systems engineering. Aborda vários aspectos da CBSE, inclusive o processo de desenvolvimento, o gerenciamento de projetos e os métodos de projeto de sistemas. (B. Thomé (ed.), 1993, John Wiley and Sons.)

EXERCÍCIOS

2.1 Explique por que outros sistemas dentro de um ambiente de sistemas podem ter efeitos não previstos sobre o funcionamento do sistema.

2.2 Modifique a Figura 2.8 para incorporar uma atividade de suprimento explícita, depois que os subsistemas tenham sido identificados. Mostre, em seu diagrama, o feedback que resulta da incorporação dessa atividade.

2.3 Explique por que especificar um sistema a ser utilizado por serviços de emergência, para o gerenciamento de ocorrências de desastres, é inerentemente um 'problema ruim'.

2.4 Sugira como os sistemas de software utilizados em um carro podem ser úteis na desativação do sistema como um todo.

2.5 Explique por que é importante produzir a descrição geral de uma arquitetura de sistemas na fase inicial do processo de especificação de sistemas.

2.6 A Figura 2.1 mostra uma gama de sistemas em um edifício. O sistema de segurança é uma versão ampliada do sistema mostrado na Figura 2.2, que visa à proteção contra intrusos e à detecção de incêndios. Ele incorpora sensores de fumaça, sensores de movimentos e sensores de portas; câmaras de vídeo sob controle de um computador, localizadas em vários pontos do edifício; um console de operador em que o status do sistema é relatado, e recursos de comunicações externas para solicitar os serviços apropriados, como polícia e corpo de bombeiros, entre outros Desenhe um diagrama de blocos de um possível projeto para esse sistema.

2.7 Um sistema de alarme de enchentes deve ser adquirido para avisar antecipadamente sobre possíveis perigos de enchentes em áreas sujeitas a alagamentos. O sistema incluirá um conjunto de sensores para monitorar a taxa de modificação dos níveis de rios, ligações a um sistema meteorológico que produz previsões sobre o tempo, ligações com sistemas de comunicações de serviços de emergência (polícia, guarda-costeira etc.), monitores de vídeo instalados em locais selecionados e uma sala de controle equipada com consoles de operadores e monitores de vídeo.

Os controladores podem acessar as informações da base de dados e iniciar as operações de vídeo. A base de dados do sistema inclui informações sobre sensores, localização de pontos de risco e condições de perigo nesses pontos (maré alta, ventos de sudeste), tabelas de maré para pontos da costa, estoque e localização de equipamentos de controle de enchente, detalhes sobre contatos para serviços de emergência, estações de rádio locais etc.

Desenhe um diagrama de blocos de uma possível arquitetura para esse sistema. Você deve identificar os subsistemas principais e as ligações entre eles.

2.8 Supondo que um sistema encomendado esteja de acordo com sua especificação, descreva, utilizando exemplos associados, três problemas que podem surgir quando ele for instalado em uma organização.

2.9 Quais são os argumentos contrários e favoráveis a considerar a engenharia de sistemas uma profissão específica, como a engenharia elétrica ou a engenharia de software?

2.10 Você é um engenheiro envolvido no desenvolvimento de um sistema financeiro. Durante a instalação, você descobre que esse sistema tornará redundante um grande número de funcionários. O pessoal no ambiente lhe nega o acesso a informações essenciais para completar a instalação do sistema. Em que medida você deverá, como engenheiro de sistemas, se envolver nessa situação? É de sua responsabilidade profissional completar a instalação para a qual foi contratado? Você deve simplesmente abandonar o trabalho até que a organização do cliente tenha resolvido o problema?

3 Processos de software

Objetivos

O objetivo deste capítulo é apresentar a idéia de um processo de software – um conjunto coerente de atividades para a produção de software. Depois de ler este capítulo, você poderá:

- compreender o conceito de um processo de software e de um modelo de processo de software;
- compreender uma série de diferentes modelos de processos de software e quando eles podem ser utilizados;
- compreender, em termos gerais, os modelos de processos para a engenharia de requisitos de software, o desenvolvimento de software, a realização de testes e a evolução, e
- conhecer a tecnologia CASE para apoio ao processo de software.

Sumário

3.1 Modelos de processo de software
3.2 Iteração de processo
3.3 Especificação de software
3.4 Projeto e implementação de software
3.5 Validação de software
3.6 Evolução de software
3.7 Apoio ao processo automatizado

Como foi sugerido no Capítulo 1, um processo de software é um conjunto de atividades e resultados associados que levam à produção de um produto de software. Esse processo pode envolver o desenvolvimento de software desde o início, embora, cada vez mais, ocorra o caso de um software novo ser desenvolvido mediante a expansão e a modificação de sistemas já existentes.

Os processos de software são complexos e, como todos os processos intelectuais, dependem de julgamento humano. Por causa da necessidade de utilizar o julgamento e a criatividade, tentativas de automatizar processos de software têm tido sucesso limitado. As ferramentas CASE (discutidas na Seção 3.7) podem apoiar algumas atividades de processo, mas não existe a possibilidade, pelo menos nos próximos anos, de uma automação mais extensiva, na qual o software assuma o projeto criativo, liberando os engenheiros envolvidos no processo de software.

Uma razão pela qual existe uma abordagem limitada para a automação de processos é a imensa diversidade dos processos de software. Não há um processo ideal, e diferentes organizações desenvolveram abordagens inteiramente diferentes para o desenvolvimento de software. Os processos evoluíram para explorar a capacidade das pessoas em uma organização, assim como as características específicas dos sistemas que estão sendo desenvolvidos. Por conseguinte, até dentro da mesma empresa pode haver muitos processos diferentes utilizados para o desenvolvimento de software.

Embora existam muitos processos de software diferentes, há atividades fundamentais comuns a todos eles, como:

1. *Especificação de software* É preciso definir a funcionalidade do software e as restrições em sua operação.
2. *Projeto e implementação de software* Deve ser produzido o software de modo que cumpra sua especificação.
3. *Validação de software* O software precisa ser validado para garantir que ele faz o que o cliente deseja.
4. *Evolução de software* O software precisa evoluir para atender às necessidades mutáveis do cliente.

Apresento uma visão geral dessas atividades neste capítulo e as discuto com muito mais detalhes em partes posteriores deste livro.

Embora não exista nenhum processo de software 'ideal', existem muitas oportunidades de trabalho para melhorá-lo, em muitas organizações. Os processos podem incluir técnicas desatualizadas ou podem não tirar vantagem das melhores práticas na engenharia de software industrial. Na verdade, muitas organizações ainda confiam em processos criados para atender a um caso específico e não tiram vantagem de métodos de engenharia de software em seu desenvolvimento de software.

A melhoria dos processos de software pode ser implementada de diferentes maneiras. Ela pode ocorrer por meio da padronização de processos, quando a diversidade dos processos de software em uma organização é reduzida. Isso leva à melhoria da comunicação e à redução do tempo de treinamento e faz com que o apoio ao processo automatizado seja mais econômico. A padronização é também uma primeira etapa essencial na introdução de novos métodos e novas técnicas de engenharia de software e também de boas práticas de engenharia de software. Retornarei ao tópico da melhoria de processos de software no Capítulo 25.

3.1 Modelos de processo de software

Como foi discutido no Capítulo 1, um modelo de processo de software é uma representação abstrata de um processo de software. Cada modelo de processo representa um processo

a partir de uma perspectiva particular, de uma maneira que proporciona apenas informações parciais sobre o processo. Nesta seção, apresento uma série de modelos de processo muito genéricos (algumas vezes, chamados de paradigmas de processo) e faço isso a partir de uma perspectiva arquitetural. Em outras palavras, vemos a estrutura do processo, mas não os detalhes de atividades específicas.

Esses modelos genéricos não são descrições definitivas de processos de software. Em vez disso, são abstrações úteis, que podem ser utilizadas para explicar diferentes abordagens do desenvolvimento de software. Para muitos sistemas de grande porte, naturalmente, não existe apenas um processo de software que possa ser utilizado. Processos diferentes são utilizados para desenvolver diferentes partes do sistema.

Os modelos de processo discutidos neste capítulo são:

1. *O modelo em cascata* Esse modelo considera as atividades de especificação, desenvolvimento, validação e evolução, que são fundamentais ao processo, e as representa como fases separadas do processo, como a especificação de requisitos, o projeto de software, a implementação, os testes e assim por diante.
2. *Desenvolvimento evolucionário* Essa abordagem intercala as atividades de especificação, desenvolvimento e validação. Um sistema inicial é rapidamente desenvolvido a partir de especificações abstratas, que são então refinadas com informações do cliente, para produzir um sistema que satisfaça a suas necessidades.
3. *Desenvolvimento formal de sistemas* Essa abordagem se baseia na produção de uma especificação formal matemática do sistema e na transformação dessa especificação, utilizando-se métodos matemáticos, para construir um programa. A verificação de componentes do sistema é realizada mediante argumentos matemáticos, mostrando que eles atendem a suas especificações.
4. *Desenvolvimento orientado a reuso* Essa abordagem tem como base a existência de um número significativo de componentes reutilizáveis. O processo de desenvolvimento de sistemas se concentra na integração desses componentes em um sistema, em vez de proceder ao desenvolvimento a partir do zero.

Os processos com base no modelo em cascata e no modelo de desenvolvimento evolucionário são amplamente utilizados para o desenvolvimento de sistemas práticos. O desenvolvimento formal de sistemas foi utilizado com sucesso em muitos projetos (Mills *et al.*, 1987; Linger, 1994), mas os processos com base nesse modelo ainda são utilizados em poucas organizações. O reuso informal é comum em muitos processos, mas a maioria das organizações não orienta explicitamente seus processos de desenvolvimento de software pelo reuso. Contudo, é provável que essa abordagem se torne muito influente no século XXI, uma vez que montar sistemas a partir de componentes reutilizáveis é essencial para o rápido desenvolvimento de software. O reuso de software será discutido no Capítulo 14.

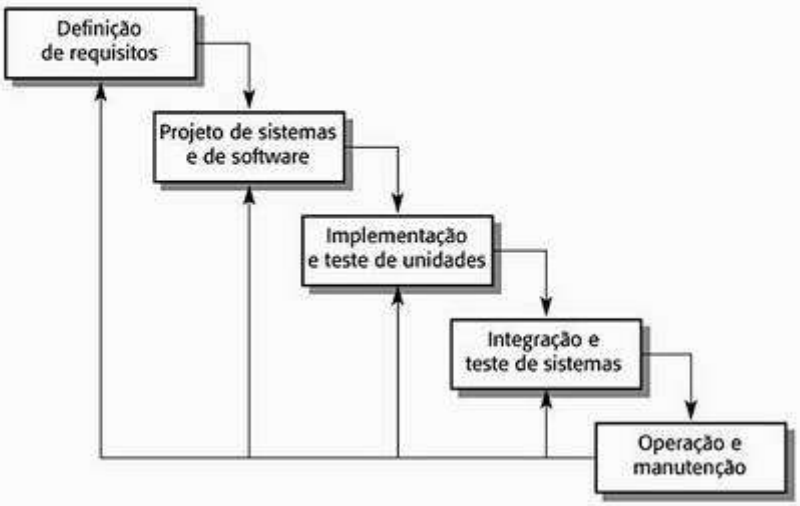
3.1.1 O modelo em 'cascata'

O primeiro modelo publicado do processo de desenvolvimento de software originou-se de outros processos de engenharia (Royce, 1970). Isso é ilustrado na Figura 3.1. Por causa da seqüência em cascata de uma fase para outra, esse modelo é conhecido como 'modelo em cascata', ou ciclo de vida de software. Os principais estágios do modelo retratam as atividades de desenvolvimento fundamentais:

1. *Análise e definição de requisitos* As funções, as restrições e os objetivos do sistema são estabelecidos por meio da consulta aos usuários do sistema. Em seguida, são definidos em detalhes e servem como uma especificação do sistema.
2. *Projeto de sistemas e de software* O processo de projeto de sistemas agrupa os requisitos em sistemas de hardware ou de software. Ele estabelece uma arquitetura do sis-

Figura 3.1

O ciclo de vida do software.



tema geral. O projeto de software envolve a identificação e a descrição das abstrações fundamentais do sistema de software e suas relações.

3. *Implementação e teste de unidades* Durante esse estágio, o projeto de software é compreendido como um conjunto de programas ou unidades de programa. O teste de unidades envolve verificar que cada unidade atenda a sua especificação.

4. *Integração e teste de sistemas* As unidades de programa ou programas individuais são integrados e testados como um sistema completo a fim de garantir que os requisitos de software foram atendidos. Depois dos testes, o sistema de software é entregue ao cliente.

5. *Operação e manutenção* Normalmente (embora não necessariamente), esta é a fase mais longa do ciclo de vida. O sistema é instalado e colocado em operação. A manutenção envolve corrigir erros que não foram descobertos em estágios anteriores do ciclo de vida, melhorando a implementação das unidades de sistema e aumentando as funções desse sistema à medida que novos requisitos são descobertos.

Em princípio, o resultado de cada fase envolve um ou mais documentos que são aprovados (assinados). A fase seguinte não deve se iniciar até que a fase precedente tenha sido concluída. Na prática, esses estágios se sobrepõem e trocam informações entre si. Durante o projeto, são identificados problemas com os requisitos; durante a codificação, são verificados problemas de projeto, e assim por diante. O processo de software não é um modelo linear simples, mas envolve uma seqüência de iterações das atividades de desenvolvimento.

Devido aos custos de produção e aprovação de documentos, as iterações são onerosas e envolvem um 'retrabalho' significativo. Portanto, depois de um pequeno número de iterações, é normal suspender partes do desenvolvimento, como a especificação, e continuar com os estágios posteriores de desenvolvimento. Os problemas são deixados para solução posterior, são ignorados ou programados para serem solucionados. Essa suspensão prematura da definição de requisitos pode significar que o sistema não fará o que o usuário quiser. Pode também levar a sistemas mal-estruturados, uma vez que os problemas de projeto são resolvidos por 'gambiarras' na implementação.

Durante o ciclo de vida final (operação e manutenção), o software é colocado em uso. São descobertos erros e omissões nos requisitos originais do software. Possíveis erros de programa e de projeto emergem, e é identificada a necessidade de nova funcionalidade. Portanto, o sistema deve evoluir para permanecer útil. Fazer essas modificações (manutenção de software) pode envolver a repetição de alguns ou de todos os estágios anteriores do processo.

O problema com o modelo em cascata é sua inflexível divisão do projeto nesses estágios distintos. Os acordos devem ser feitos em um estágio inicial do processo, e isso significa que

é difícil responder aos requisitos do cliente, que sempre se modificam. Portanto, o modelo em cascata deve ser utilizado somente quando os requisitos forem bem compreendidos. Contudo, ele reflete a prática da engenharia. Conseqüentemente, os processos de software com base nessa abordagem ainda são utilizados no desenvolvimento de software, em particular quando fazem parte de um projeto maior de engenharia de sistemas.

3.1.2 Desenvolvimento evolucionário

O desenvolvimento evolucionário tem como base a idéia de desenvolver uma implementação inicial, expor o resultado ao comentário do usuário e fazer seu aprimoramento por meio de muitas versões, até que um sistema adequado tenha sido desenvolvido (Figura 3.2). Em vez de ter as atividades de especificação, desenvolvimento e validação em separado, todo esse trabalho é realizado concorrentemente com um rápido feedback por meio dessas atividades.

Há dois tipos de desenvolvimento evolucionário:

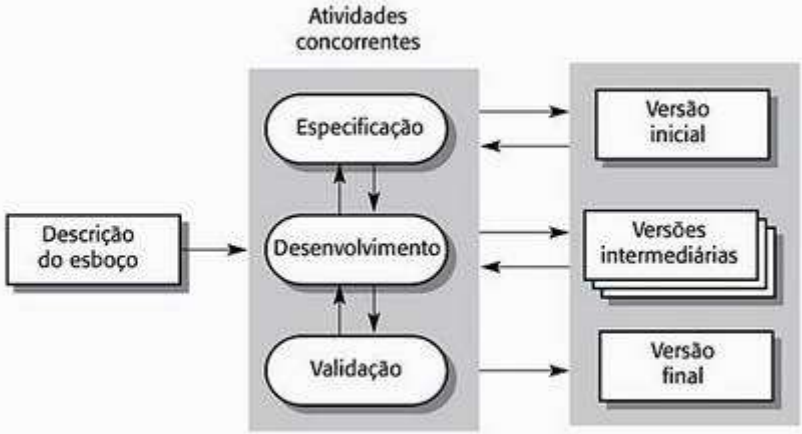
1. *Desenvolvimento exploratório,* em que o objetivo do processo é trabalhar com o cliente, a fim de explorar seus requisitos e entregar um sistema final. O desenvolvimento se inicia com as partes do sistema que são compreendidas. O sistema evolui com o acréscimo de novas características, à medida que elas são propostas pelo cliente.
2. *Fazer protótipos descartáveis,* em que o objetivo do desenvolvimento evolucionário é compreender os requisitos do cliente e, a partir disso, desenvolver uma melhor definição de requisitos para o sistema. O protótipo se concentra em fazer experimentos com partes dos requisitos que estejam mal compreendidas.

Falarei sobre os processos de desenvolvimento evolucionário e suporte a processos no Capítulo 8, em que são descritas várias técnicas de prototipação.

A abordagem evolucionária do desenvolvimento de software, muitas vezes, é mais eficaz do que a abordagem em cascata, no sentido de produzir sistemas que atendam às necessidades imediatas dos clientes. A vantagem de um processo de software com base na abordagem evolucionária é que a especificação pode ser desenvolvida gradativamente. À medida que os usuários desenvolvem uma compreensão melhor de seus problemas, isso pode ser refletido no sistema de software. Contudo, a partir de uma perspectiva de engenharia e de gerenciamento, existem três problemas:

1. *O processo não é visível* Os gerentes necessitam que o desenvolvimento seja regular, para que possam medir o progresso. Se os sistemas são desenvolvidos rapidamente, não é viável produzir documentos que reflitam cada versão do sistema.

Figura 3.2 ▶
Desenvolvimento evolucionário.



2. *Os sistemas freqüentemente são mal-estruturados* A mudança constante tende a corromper a estrutura do software. Incorporar modificações no software torna-se cada vez mais difícil e oneroso.

3. *Podem ser exigidas ferramentas e técnicas especiais* Elas possibilitam rápido desenvolvimento, mas podem ser incompatíveis com outras ferramentas ou técnicas, e relativamente poucas pessoas podem ter a habilitação necessária para utilizá-las.

Para sistemas pequenos (com menos de 100 mil linhas de código) ou para sistemas de porte médio (com até 500 mil linhas de código), com tempo de vida razoavelmente curto, acredito que a abordagem evolucionária de desenvolvimento seja a melhor opção. Contudo, para sistemas de grande porte, de longo tempo de vida, os problemas de desenvolvimento evolucionário se tornam particularmente graves. Para esses sistemas, recomendo um processo misto, que incorpore as melhores características do modelo de desenvolvimento em cascata e do evolucionário.

Isso pode significar desenvolver um protótipo descartável, utilizando uma abordagem evolucionária para resolver dúvidas na especificação do sistema. Esse sistema pode, então, ser reimplementado, utilizando-se uma abordagem mais estruturada. Partes do sistema que são bem compreendidas podem ser especificadas e desenvolvidas utilizando-se o processo com base no modelo em cascata. Outras partes do sistema, como a interface com o usuário, que são difíceis de serem especificadas com antecedência devem ser desenvolvidas utilizando-se uma abordagem de programação exploratória.

3.1.3 Desenvolvimento formal de sistemas

O desenvolvimento formal de sistemas é uma abordagem do desenvolvimento de software que tem algo em comum com o modelo em cascata, mas cujo processo de desenvolvimento tem como base a transformação matemática formal de uma especificação de sistema em um programa executável. Esse processo está ilustrado na Figura 3.3. Por questões de simplicidade, deixei de lado a iteração desse modelo de processo.

As principais distinções entre essa abordagem e o modelo em cascata são:

1. A especificação de requisitos de software é redefinida em uma especificação formal detalhada, que é expressa em uma notação matemática.

2. Os processos de desenvolvimento de projeto, implementação e teste de unidade são substituídos por um processo de desenvolvimento transformacional, em que a especificação

Figura 3.3 ▶
Desenvolvimento formal de sistemas.

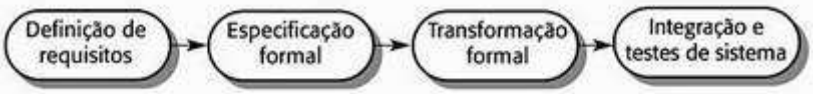
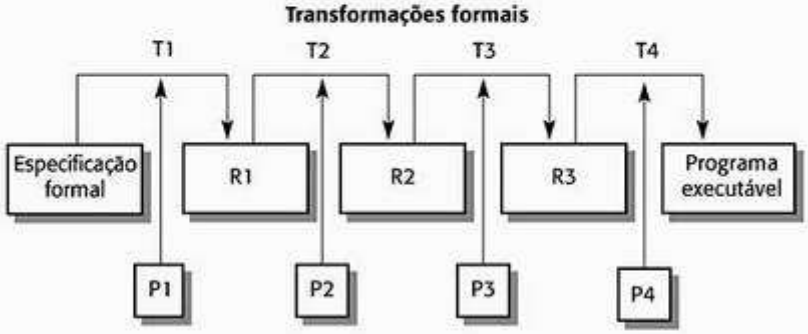


Figura 3.4 ▶
Transformação formal.



formal é refinada por meio de uma série de transformações, em um programa. Esse processo de aperfeiçoamento é ilustrado na Figura 3.4.

No processo de transformação, a representação matemática formal do sistema é sistematicamente convertida em uma representação de sistema mais detalhada, mas ainda matematicamente correta. Cada etapa acrescenta mais detalhes, até que a especificação formal seja convertida em um programa equivalente. As transformações são suficientemente próximas, de modo que o esforço de verificação da transformação não é excessivo. Portanto, podemos garantir, considerando que não haja erros de verificação, que o programa é uma verdadeira implementação da especificação.

A vantagem da abordagem transformacional, comparada a provar que um programa cumpre sua especificação, é que a distância entre cada transformação é menor que a distância entre uma especificação e um programa. As provas de programas são muito longas e impraticáveis para sistemas de grande porte. Uma abordagem transformacional constituída de uma seqüência de etapas menores é mais viável. Contudo, escolher qual transformação deve ser aplicada é uma tarefa que exige habilidade e comprovar a correspondência de transformações é difícil.

O exemplo mais conhecido desse processo de desenvolvimento formal é o processo Cleanroom, que foi originalmente desenvolvido pela IBM (Mills *et al.*, 1987; Selby *et al.*, 1987; Linger, 1994; Prowell *et al.*, 1999). O processo Cleanroom se apóia no desenvolvimento incremental do software, em que cada estágio é desenvolvido e sua correção é demonstrada, utilizando-se uma abordagem formal. Não há testes para detectar defeitos no processo, e o teste do sistema é orientado visando medir a confiabilidade do sistema. Esse processo será discutido no Capítulo 19.

Tanto a abordagem de Cleanroom como qualquer outra abordagem de desenvolvimento formal, com base no método B (Wordsworth, 1996), foram aplicadas com sucesso. Havia poucos defeitos no sistema entregue, e os custos de desenvolvimento não foram significativamente diferentes dos custos de outras abordagens. Essa abordagem é particularmente adequada ao desenvolvimento de sistemas que tenham rigorosas exigências de segurança, confiabilidade e garantia. A abordagem formal simplifica a produção de um caso de segurança ou de confiança, que demonstra aos clientes ou comitês de certificação que o sistema realmente cumpre os requisitos de segurança e confiança.

Fora desses domínios especializados, os processos com base em transformação formal não são amplamente utilizados; eles requerem perícia especializada e, na verdade, para a maioria dos sistemas, esse processo não oferece vantagens significativas de custo ou qualidade em relação a outras abordagens. A razão principal disso é que a interação do sistema não é sensível à especificação formal, o que consome uma parte muito grande do esforço de desenvolvimento para a maioria dos sistemas de software.

3.1.4 Desenvolvimento orientado a reuso

Na maioria dos projetos de software, ocorre algum reuso de software. Isso, em geral, acontece informalmente, quando as pessoas que trabalham no projeto conhecem projetos ou códigos similares àquele exigido. Eles recorrem a esses produtos, fazem as modificações conforme necessário e as incorporam em seus sistemas. Na abordagem evolucionária, descrita na Seção 3.1.2, o reuso é visto freqüentemente como essencial para o rápido desenvolvimento de sistemas.

Esse reuso informal acontece independentemente do processo genérico empregado. Contudo, nos últimos anos, surgiu uma abordagem de desenvolvimento de software (engenharia de software com base em componentes) que conta com o reuso e que tem se tornado cada vez mais amplamente utilizada.

Essa abordagem orientada a reuso conta com uma ampla base de componentes de software reutilizáveis, que podem ser acessados, e com alguma infra-estrutura de integração para esses componentes. Às vezes, esses componentes são sistemas propriamente ditos (os já menciona-

dos COTS, sistemas comerciais de prateleira), que podem ser utilizados para proporcionar funcionalidade específica, como formatação de textos, cálculo numérico, entre outros. O modelo genérico de processo para o desenvolvimento orientado a reuso é mostrado na Figura 3.5.

Embora o estágio inicial de especificação de requisitos e o estágio de validação sejam comparáveis com outros processos, os estágios intermediários em um processo orientado a reuso são diferentes. Esses estágios são:

1. *Análise de componentes* Considerando a especificação de requisitos, é feita uma busca de componentes para implementar essa especificação. Em geral, não existe uma combinação exata e os componentes que podem ser utilizados fornecem somente parte da funcionalidade requerida.

2. *Modificação de requisitos* Durante esse estágio, os requisitos são analisados, utilizando-se as informações sobre os componentes que foram encontrados. Eles são então modificados para refletir os componentes disponíveis. Quando as modificações forem impossíveis, a atividade de análise de componentes poderá ser refeita, a fim de procurar soluções alternativas.

3. *Projeto de sistema com reuso* Durante essa fase, a infra-estrutura do sistema é projetada ou uma infra-estrutura existente é reutilizada. Os projetistas levam em conta os componentes que são reutilizados e organizam a infra-estrutura para lidar com esse aspecto. Um novo software poderá ter de ser projetado, se os componentes reutilizáveis não estiverem disponíveis.

4. *Desenvolvimento e integração* O software que não puder ser comprado será desenvolvido, e os componentes e sistemas COTS serão integrados, a fim de criar um sistema. A integração de sistemas, nesse modelo, pode ser parte do processo de desenvolvimento, em vez de uma atividade separada.

O modelo orientado a reuso tem a vantagem óbvia de reduzir a quantidade de software a ser desenvolvida e, assim, de reduzir custos e riscos. Geralmente, ele também propicia a entrega mais rápida do software. Contudo, as adequações nos requisitos são inevitáveis, e isso pode resultar em um sistema que não atenda às reais necessidades dos usuários. Além disso, algum controle sobre a evolução do sistema se perde, uma vez que novas versões dos componentes reutilizáveis não estão sob o controle da organização que utiliza esses componentes.

3.2 Iteração de processo

Todos os modelos de processo mostrados anteriormente apresentam vantagens e desvantagens. Para a maioria dos grandes sistemas, existe a necessidade de utilizar diferentes abordagens para diferentes partes do sistema, de maneira que um modelo híbrido tem de ser utilizado. Além

Figura 3.5 ▶
Desenvolvimento orientado a reuso.

disso, há também a necessidade de apoiar o processo de iteração, em que partes do processo são repetidas, à medida que os requisitos do sistema evoluem. O projeto de sistema e o trabalho de implementação devem ser refeitos, a fim de implementar os requisitos modificados.

Nesta seção, discuto dois modelos híbridos, que apóiam diferentes abordagens do desenvolvimento e que foram explicitamente projetados para apoiarem a iteração de processo. São eles:

1. desenvolvimento incremental, quando as fases de especificação, projeto e implementação de software são desdobradas em uma série de estágios, que são, por sua vez, desenvolvidos;
2. desenvolvimento em espiral, quando o desenvolvimento do sistema evolui a partir de um esboço inicial, em direção ao sistema final desenvolvido.

A essência dos processos iterativos é que a especificação é desenvolvida em conjunto com o software. Contudo, isso entra em conflito com o modelo de suprimento de muitas organizações, em que a especificação completa do sistema faz parte do contrato para o desenvolvimento do sistema. Na abordagem gradativa, não há uma especificação completa de sistema, até que o estágio final seja especificado. Isso requer um novo tipo de contrato, que os grandes clientes, como os órgãos de governo, podem ter dificuldade em aceitar.

3.2.1 Desenvolvimento incremental

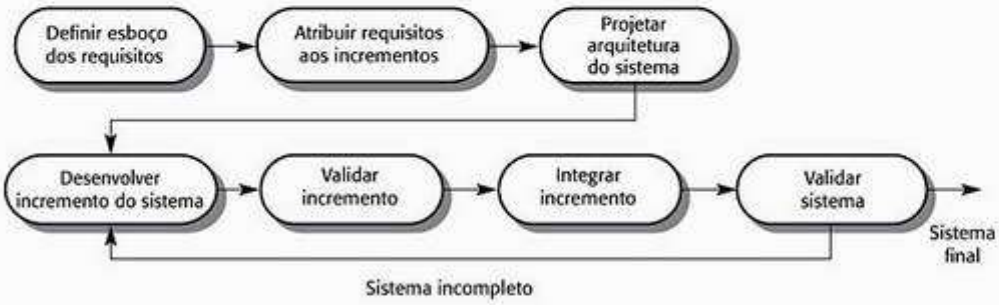
O modelo de desenvolvimento em cascata requer que os clientes de um sistema se comprometam com um conjunto de requisitos específicos, antes do início do projeto, e que os projetistas se comprometam com estratégias de projeto em particular, antes da implementação. As mudanças nos requisitos durante o desenvolvimento requerem o 'retrabalho' referente aos requisitos, ao projeto e à implementação. Contudo, as vantagens do modelo em cascata são as de um modelo simples de gerenciamento, e sua separação de projeto e implementação devem levar a sistemas robustos, que são suscetíveis a mudanças.

Por outro lado, a abordagem do desenvolvimento evolucionário permite que os requisitos e as decisões de projeto sejam postergados, mas também leva a um software que pode ser mal-estruturado e de difícil compreensão e manutenção. O desenvolvimento incremental é uma abordagem intermediária, que combina as vantagens de ambos esses modelos.

A abordagem do desenvolvimento incremental (Figura 3.6) foi sugerida por Mills (Mills *et al.*, 1980) como um meio de reduzir o 'retrabalho' no processo de desenvolvimento e de proporcionar aos clientes algumas oportunidades de adiar decisões sobre seus requisitos detalhados, até que eles tenham alguma experiência com o sistema.

Em um processo de desenvolvimento incremental, os clientes identificam, em um esboço, as funções a serem fornecidas pelo sistema. Eles identificam quais funções são mais importantes e quais são menos importantes para eles. Em seguida é definida uma série de estágios de entrega, com cada estágio fornecendo um subconjunto das funcionalidades do sistema. A alocação de funções aos estágios depende da prioridade da função. As funções prioritárias são entregues primeiramente ao cliente.

Figura 3.6 ▶
Desenvolvimento incremental.



Uma vez identificados os incrementos, os requisitos para as funções a serem entregues no primeiro incremento são definidos em detalhes, e esse incremento é desenvolvido, utilizando-se o processo de desenvolvimento mais apropriado. Durante esse desenvolvimento, outras análises de requisitos para os incrementos seguintes podem ocorrer, mas as mudanças nos requisitos para o incremento atual não são aceitas.

Uma vez que um incremento é concluído e entregue, os clientes podem colocá-lo em operação. Isso significa que eles recebem com antecedência parte da funcionalidade do sistema. Eles podem experimentar o sistema, o que lhes facilita esclarecer seus requisitos para os incrementos subseqüentes e para versões posteriores do estágio atual. À medida que novos incrementos são concluídos, eles são integrados aos estágios existentes, de modo que a funcionalidade do sistema melhora a cada novo estágio que é entregue. As funções comuns podem ser implementadas no início do processo ou gradativamente, à medida que a funcionalidade é exigida por um estágio.

Não existe a necessidade de utilizar o mesmo processo para o desenvolvimento de cada incremento. Quando as funções em um incremento têm uma especificação bem-definida, o modelo de desenvolvimento em cascata pode ser utilizado para aquele incremento. Quando a especificação for mal definida, poderá ser utilizado um modelo de desenvolvimento evolucionário.

Esse processo de desenvolvimento incremental tem diversas vantagens:

1. Os clientes não precisam esperar até que todo o sistema seja entregue, para então tirarem proveito dele. O primeiro estágio satisfaz seus requisitos mais importantes e, assim, o software pode ser imediatamente utilizado.
2. Os clientes podem utilizar os primeiros incrementos como um protótipo e obter uma experiência que forneça os requisitos para estágios posteriores do sistema.
3. Existe um risco menor de fracasso completo do sistema. Embora possam ser encontrados problemas em alguns incrementos, é provável que alguns incrementos sejam entregues com sucesso ao cliente.
4. Como as funções prioritárias são entregues primeiro e incrementos posteriores são integrados a elas, é inevitável que as funções de sistema mais importantes passem pela maior parte dos testes. Isso significa que é menos provável que os clientes encontrem falhas de software na parte mais importante do sistema.

Contudo, existem alguns problemas com o desenvolvimento incremental. Os incrementos devem ser relativamente pequenos (compostos de não mais do que 20 mil linhas de código), e cada incremento deve produzir alguma funcionalidade para o sistema. Pode, portanto, ser difícil mapear os requisitos dos clientes dentro de incrementos de tamanho correto. Além disso, a maioria dos sistemas exige um conjunto de facilidades básicas que são utilizadas por diferentes partes do sistema. Como os requisitos não são definidos em detalhes até que um incremento seja implementado, é difícil identificar facilidades comuns que todos os incrementos exijam.

Uma recente evolução dessa abordagem incremental, chamada de 'programação extrema' (*extreme programming*), foi desenvolvida (Beck, 1999). Ela tem como base o desenvolvimento e a entrega de incrementos de funcionalidade muito pequenos, o envolvimento do cliente no processo, a constante melhoria de código e a programação impessoal, como discutiremos no Capítulo 23. O artigo de Beck inclui vários relatórios de sucesso dessa abordagem, mas ainda é muito cedo para dizer se esta se tornará uma abordagem de uso dominante para o desenvolvimento de software.

3.2.2 Desenvolvimento em espiral

O modelo de processo de desenvolvimento em espiral (Figura 3.7), originalmente proposto por Boehm (1988), é agora bastante conhecido. Em vez representar o processo de software como uma seqüência de atividades com algum retorno de uma atividade para outra,

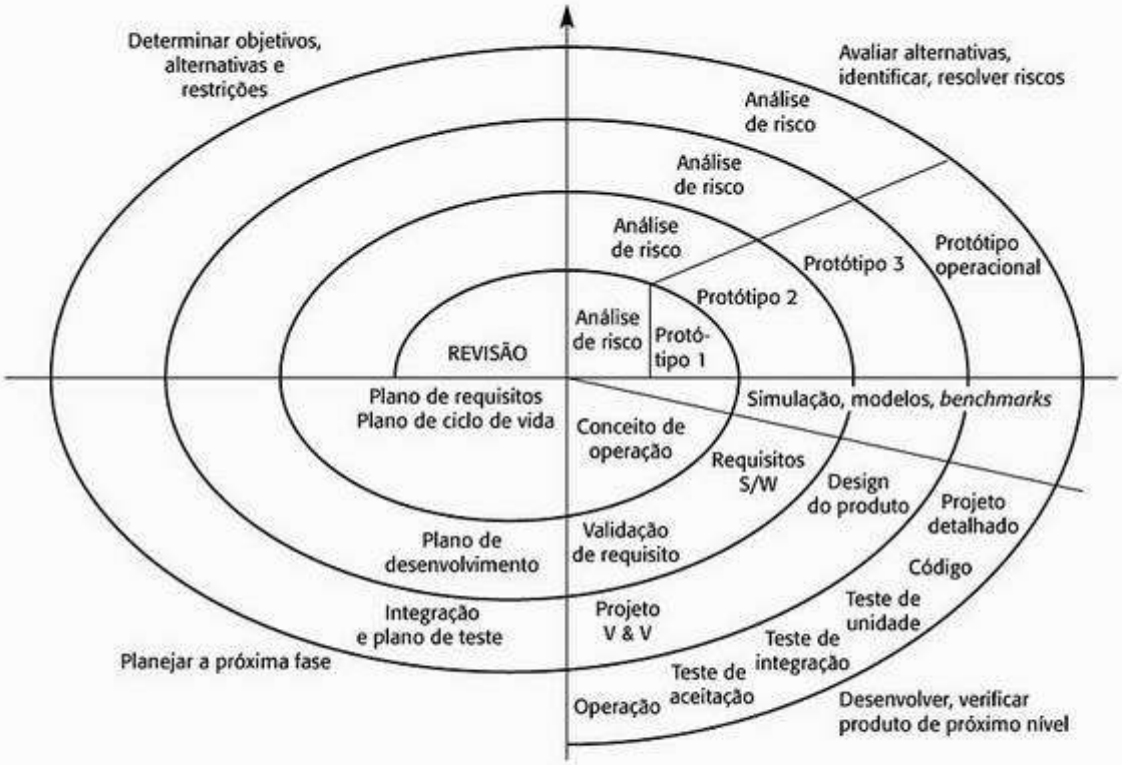


Figura 3.7 ▲

O modelo de processo de software em espiral de Boehm (©1988 IEEE).

o processo é representado como uma espiral. Cada loop na espiral representa uma fase do processo de software. Assim, o loop mais interno pode estar relacionado à viabilidade do sistema; o loop seguinte, à definição de requisitos do sistema; o próximo loop, ao projeto do sistema, e assim por diante.

Cada loop da espiral é dividido em quatro setores:

1. *Definição de objetivos* São definidos os objetivos específicos para essa fase do projeto. São identificadas as restrições para o processo e o produto e é preparado um plano de gerenciamento detalhado. São identificados os riscos do projeto e, dependendo dos riscos, poderão ser planejadas estratégias alternativas.

2. *Avaliação e redução de riscos* Para cada um dos riscos de projeto identificados, é realizada uma análise detalhada e são tomadas providências para reduzir esses riscos. Por exemplo, se houver um risco de os requisitos serem inadequados, poderá ser desenvolvido um protótipo.

3. *Desenvolvimento e validação* Depois da avaliação dos riscos, é escolhido um modelo de desenvolvimento para o sistema. Por exemplo, se forem dominantes os riscos relacionados à interface com o usuário, um modelo apropriado de desenvolvimento pode ser a prototipação evolucionária. Se os riscos de segurança forem a principal consideração, o desenvolvimento com base em transformações formais poderá ser o mais apropriado e assim por diante. O modelo em cascata poderá ser o modelo de desenvolvimento mais apropriado se o risco principal identificado for o da integração de sistemas.

4. *Planejamento* O projeto é revisto e é tomada uma decisão sobre continuar com o próximo loop da espiral. Se a decisão for continuar, serão traçados os planos para a próxima fase do projeto.

A importante distinção entre o modelo em espiral e outros modelos de processo de software é a explícita consideração dos riscos no modelo em espiral. Informalmente, o risco é simples-

mente algo que pode acontecer de errado. Por exemplo, se a intenção for utilizar uma nova linguagem de programação, existirá o risco de que os compiladores disponíveis não sejam confiáveis ou que não produzam código-objeto suficientemente eficiente. Os riscos resultam em problemas no projeto, como a possibilidade de exceder no prazo e no custo previstos; portanto, minimizar riscos é uma atividade de gerenciamento de projeto muito importante. No Capítulo 4, que trata do gerenciamento de projetos, apresento uma discussão detalhada sobre riscos e o gerenciamento de riscos.

Um ciclo da espiral começa com a elaboração de objetivos, como desempenho, funcionalidade etc. Meios alternativos de atingir esses objetivos e as restrições impostas a cada uma dessas alternativas são então enumerados. Cada alternativa é examinada em relação a cada objetivo. Isso normalmente resulta na identificação de causas de riscos para o projeto. A próxima etapa é avaliar esses riscos por atividades, como análises mais detalhadas, prototipação, simulação, entre outras. Uma vez avaliados os riscos, alguma parte do desenvolvimento é realizada, seguida por uma atividade de planejamento para a próxima fase do processo.

Não há fases fixas, como especificação ou projeto, no modelo em espiral. O modelo em espiral abrange outros modelos de processo. A prototipação pode ser utilizada em uma espiral para resolver dúvidas relativas aos requisitos e, portanto, reduzir riscos. Essa fase pode ser seguida por um desenvolvimento em cascata convencional. A transformação formal pode ser utilizada para desenvolver as partes do sistema com muitos requisitos de segurança.

3.3 Especificação de software

Nesta seção e nas três seções seguintes, discutirei as atividades básicas de especificação, desenvolvimento, validação e evolução de software. A primeira dessas atividades, a especificação de software, destina-se a estabelecer quais funções são requeridas pelo sistema e as restrições sobre a operação e o desenvolvimento do sistema. Essa atividade, atualmente, é freqüentemente chamada de engenharia de requisitos; ela é um estágio particularmente importante do processo de software, uma vez que erros nesse estágio inevitavelmente produzem problemas posteriores no projeto e na implementação do sistema.

O processo de engenharia de requisitos é mostrado na Figura 3.8. Esse processo leva à produção de uma documentação de requisitos, que é a especificação para o sistema. Os requisitos geralmente são apresentados em dois níveis de detalhes nesse documento. Os usuários finais e os clientes necessitam de uma declaração de alto nível dos requisitos; os desenvolvedores de sistema precisam de uma especificação mais detalhada do sistema.

Existem quatro fases principais no processo de engenharia de requisitos:

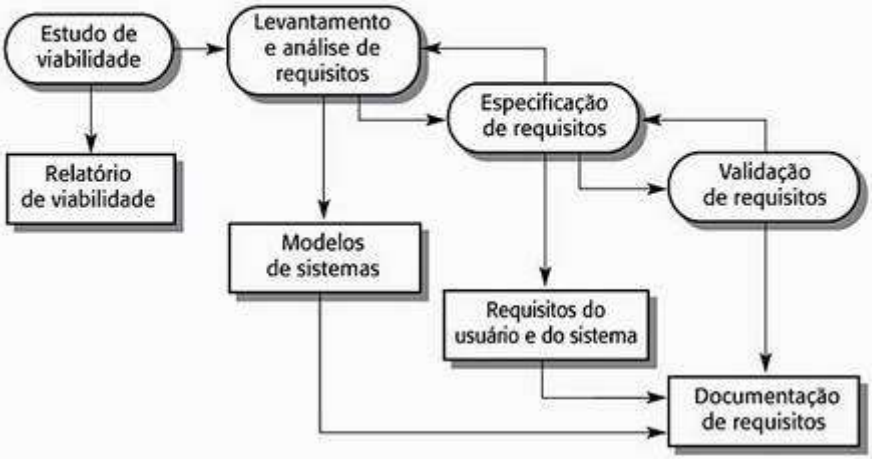
1. *Estudo de viabilidade* É feita uma estimativa para verificar se as necessidades dos usuários que foram identificadas podem ser satisfeitas com a utilização das atuais tecnologias de software e hardware. O estudo decidirá se o sistema proposto será viável, do ponto de vista comercial, e se poderá ser desenvolvido considerando certas restrições orçamentárias existentes. Um estudo de viabilidade deve ser relativamente barato e rápido. O resultado deve informar se a decisão é a de prosseguir com uma análise mais detalhada.

2. *Levantamento e análise de requisitos* Este é o processo de obter os requisitos do sistema pela observação de sistemas existentes, pela conversa com usuários e compradores em potencial, pela análise de tarefas e assim por diante. Ele pode envolver o desenvolvimento de um ou mais diferentes modelos e protótipos de sistema. Isso ajuda o analista a compreender o sistema a ser especificado.

3. *Especificação de requisitos* É a atividade de traduzir as informações coletadas durante a atividade de análise em um documento que defina um conjunto de requisitos. Dois tipos

Figura 3.8

O processo de engenharia de requisitos.



de requisitos podem ser incluídos nesse documento. Os requisitos dos usuários são declarações abstratas dos requisitos de sistema para o cliente e os usuários finais do sistema; os requisitos do sistema são uma descrição mais detalhada da funcionalidade a ser fornecida.

4. *Validação de requisitos* Essa atividade verifica os requisitos quanto a sua pertinência, consistência e integralidade. Durante esse processo, inevitavelmente são descobertos erros na documentação de requisitos. Os requisitos devem então ser modificados, a fim de corrigir esses problemas.

Naturalmente, as atividades no processo de requisitos não são realizadas simplesmente em uma seqüência rigorosa. A análise de requisitos continua durante a definição e a especificação, e novos requisitos surgem ao longo do processo. Portanto, as atividades de análise, definição e especificação são intercaladas.

3.4 Projeto e implementação de software

O estágio de implementação do desenvolvimento de software é o processo de conversão de uma especificação de sistema em um sistema executável. Esse estágio sempre envolve processos de projeto e programação de software, mas, se uma abordagem evolucionária de desenvolvimento for utilizada, ele poderá também envolver o aperfeiçoamento da especificação de software.

Um projeto de software é uma descrição de estrutura de software a ser implementada, dos dados que são parte do sistema, das interfaces entre os componentes do sistema e, algumas vezes, dos algoritmos utilizados. Os projetistas não conseguem que um projeto seja concluído imediatamente, mas desenvolvem o projeto iterativamente por meio de uma série de diferentes versões. O processo de projeto envolve acrescentar formas e detalhes, à medida que o projeto é desenvolvido, com 'retornos' constantes, a fim de corrigir projetos anteriores.

O processo de projeto pode envolver o desenvolvimento de vários modelos do sistema, em diferentes níveis de abstração. Na decomposição de um projeto, são descobertos erros e omissões em estágios anteriores. Essas informações de feedback permitem a melhoria de modelos de projeto anteriores. A Figura 3.9 é um modelo desse processo, que mostra as descrições de projeto produzidas em diferentes estágios desse projeto. Esse diagrama sugere que

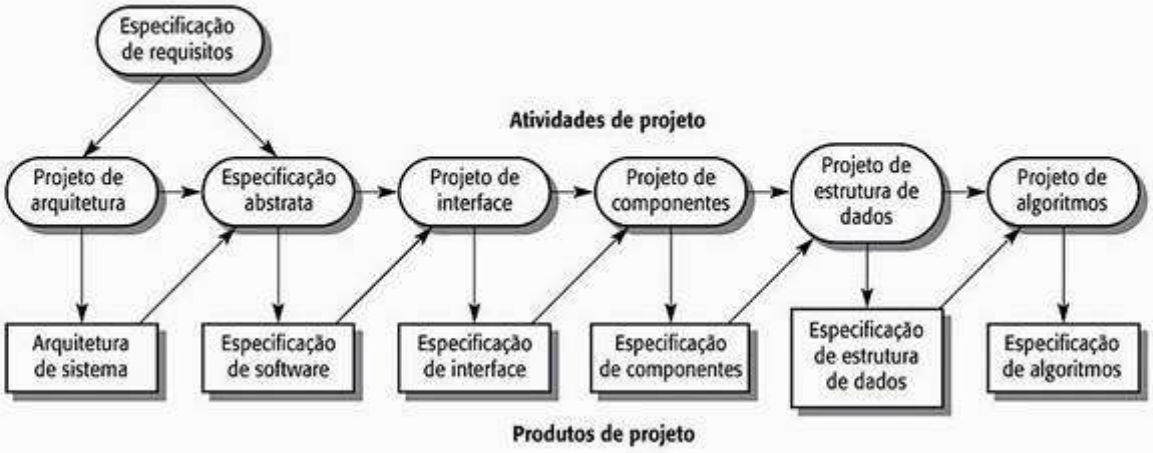


Figura 3.9 ▲
Um modelo geral do processo de projeto.

os estágios de processo de projeto são seqüenciais. Na verdade, as atividades de processo de projeto são intercaladas. O feedback de um estágio para outro e o conseqüente 'retrabalho' do projeto é inevitável em todos os processos de projeto.

Uma especificação para o próximo estágio é a saída de cada atividade de projeto. Essa especificação pode ser uma especificação abstrata, formal, produzida para esclarecer os requisitos, ou pode ser uma especificação de como parte do sistema é implementada. À medida que o processo de projeto continua, essas especificações se tornam mais detalhadas. Os resultados finais de processo são especificações precisas de algoritmos e estruturas de dados a serem implementadas.

As atividades específicas de processo de projeto são:

1. *Projeto de arquitetura* Os subsistemas que constituem o sistema e suas relações são identificados e documentados. Esse importante tópico é abordado no Capítulo 10.
2. *Especificação abstrata* Para cada subsistema, é produzida uma especificação abstrata de suas funções e das restrições dentro das quais deve operar.
3. *Projeto de interface* Para cada subsistema, sua interface com outros subsistemas é projetada e documentada. Essa especificação de interface não pode apresentar ambigüidade, uma vez que ela permite que o subsistema seja utilizado sem conhecimento da operação do subsistema. Os métodos de especificação formal, como os discutidos no Capítulo 9, podem ser utilizados nesse estágio.
4. *Projeto de componentes* Funções são alocadas a diferentes componentes e as interfaces desses componentes são projetadas.
5. *Projeto de estrutura de dados* As estruturas de dados utilizadas na implementação de sistemas são projetadas em detalhe e especificadas.
6. *Projeto de algoritmos* Os algoritmos utilizados para proporcionarem serviços são projetados detalhadamente e especificados.

Esse é um modelo muito genérico do processo de projeto, e processos práticos reais podem adaptá-lo de diferentes maneiras. Por exemplo, os dois últimos estágios, a estrutura de dados e o projeto de algoritmos podem ser parte de projeto ou parte do processo de implementação. Se existem objetos disponíveis para reuso, isso pode restringir a arquitetura do sistema e as interfaces dos módulos do sistema. Isso pode significar que o número de componentes a serem projetados será bastante reduzido. Uma abordagem exploratória do projeto pode ser utilizada, e as interfaces do sistema podem ser projetadas, depois que as estruturas de dados tiverem sido especificadas.

3.4.1 Métodos de projeto

Em muitos projetos de desenvolvimento de software, o projeto de software ainda é um processo utilizado à medida que for necessário. Partindo de um conjunto de requisitos, geralmente em linguagem natural, é preparado um projeto informal. A codificação se inicia, e o projeto é modificado à medida que o sistema é implementado. Existe pouco ou nenhum controle formal dessas modificações ou do gerenciamento de projeto. Quando o estágio de implementação se completa, o projeto normalmente foi tão modificado, depois de sua especificação inicial, que o documento original de projeto é uma descrição incoerente e incompleta do sistema.

Uma abordagem mais metódica do projeto de software é proposta pelos 'métodos estruturados', que são conjuntos de notações e diretrizes para o projeto de software. Dentre os exemplos de métodos estruturados destacam-se o 'Projeto Estruturado' (Constantine e Yourdon, 1979), as 'Análises de Sistemas Estruturados' (Gane e Sarson, 1979), o 'Desenvolvimento de Sistemas' de Jackson (Jackson, 1983) e várias abordagens de projeto orientado a objetos (Robinson, 1992; Booch, 1994; Rumbaugh *et al.*, 1991; Booch *et al.*, 1999; Rumbaugh *et al.*, 1999a, 1999b).

O uso de métodos estruturados normalmente envolve a produção de modelos gráficos de sistemas e resulta em grandes quantidades de documentação de projeto. As ferramentas CASE (veja a Seção 3.7) foram desenvolvidas para apoiar métodos específicos. Os métodos estruturados foram aplicados com sucesso em muitos projetos de grande porte. Eles podem proporcionar significativas reduções de custos, porque utilizam notações padronizadas e asseguram a produção de documentação-padrão para o projeto. Não se pode demonstrar que um método é melhor ou pior do que outros. O sucesso ou fracasso dos métodos depende, com freqüência, de sua adequação ao domínio de uma aplicação.

Um método estruturado inclui um modelo de processo de projeto, notações para representar o projeto, formulários de relatórios, regras e diretrizes de projeto. Embora haja um grande número de métodos, eles têm muito em comum. Os métodos estruturados podem aceitar alguns ou todos os seguintes modelos de um sistema:

1. Um modelo de fluxo de dados, em que o sistema é modelado utilizando-se as transformações de dados que ocorrem à medida que eles são processados.
2. Um modelo de relacionamento de entidades, que é utilizado para descrever as entidades básicas do projeto e as relações entre elas. Os modelos de relacionamento de entidades são a técnica normal utilizada para descrever estruturas de base de dados.
3. Um modelo estrutural, em que são documentados os componentes do sistema e suas interações.
4. Métodos orientados a objetos, que incluem um modelo de herança do sistema, modelos dos relacionamentos estáticos e dinâmicos entre objetos e um modelo de como os objetos interagem entre si quando o sistema está em execução.

Métodos específicos complementam esses com outros modelos de sistemas, como os diagramas de transição de estado; históricos de vida de entidades, que mostram como cada entidade é transformada, à medida que ela é processada, e assim por diante. A maioria dos métodos sugere que um repositório centralizado de informações do sistema, ou um dicionário de dados, deve ser utilizado. Discutirei uma série dessas abordagens da modelagem de sistemas no Capítulo 7.

Na prática, a orientação dada pelos métodos é informal e, assim, diferentes projetistas desenvolverão diferentes projetos. Esses 'métodos', na verdade, são notações padronizadas e incorporações das boas práticas. Seguindo esses métodos e aplicando as diretrizes, um projeto razoável deve surgir como resultado. A criatividade do projetista ainda é exigida para decidir sobre a decomposição do sistema e para assegurar que o projeto 'entenda' adequadamente a especificação do sistema. Estudos empíricos de projetistas (Bansler e Bødker, 1993) mostraram que eles raramente seguem os métodos estritamente. Eles fazem a escolha das diretrizes, dependendo de circunstâncias locais.

3.4.2 Programação e depuração

O desenvolvimento de um programa para implementar o sistema parte naturalmente dos processos de projeto de sistemas. Embora algumas classes de programas, como os sistemas em que a segurança é fundamental, sejam projetadas detalhadamente, antes que qualquer implementação tenha início, é mais comum que os estágios posteriores de desenvolvimento de projeto e de programa sejam intercalados. As ferramentas CASE podem ser utilizadas para gerar um esboço do programa, a partir de um projeto. Isso inclui o código para definir e implementar interfaces, e, em muitos casos, o desenvolvedor precisa apenas acrescentar detalhes da operação de cada componente do programa.

A programação é uma atividade pessoal, e não existe um processo geral que seja normalmente seguido. Alguns programadores começarão com os componentes que eles compreendem bem; desenvolverão esses componentes e, então, prosseguirão com os componentes não tão bem compreendidos. Outros preferem a abordagem oposta, deixando os componentes familiares para o fim, porque sabem como desenvolvê-los. Alguns desenvolvedores gostam de definir os dados no início do processo e, então, utilizam esses dados para dirigir o desenvolvimento do programa; outros deixam os dados sem especificação enquanto for possível.

Normalmente, programadores realizam alguns testes do código que eles mesmos desenvolveram. Isso, muitas vezes, revela defeitos do programa que devem ser removidos. Esse processo é chamado de *depuração*. O teste e a depuração dos defeitos são processos diferentes. O teste estabelece a existência de defeitos. A depuração se ocupa em localizar e corrigir esses defeitos.

A Figura 3.10 ilustra um possível processo de depuração. Defeitos no código devem ser localizados, e o programa precisa ser modificado, a fim de cumprir os requisitos. Os testes terão de ser repetidos, para garantir que a mudança foi feita corretamente. Assim, o processo de depuração é parte tanto do desenvolvimento quanto do teste do software.

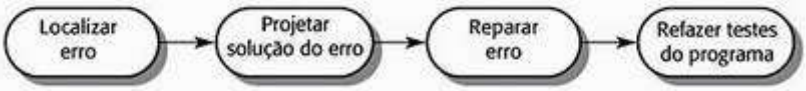
O depurador deve gerar hipóteses sobre o comportamento observável do programa e, então, testá-las, na esperança de encontrar o defeito que causou a anomalia. Testar as hipóteses pode envolver rastrear o código de programa manualmente e pode requerer novos testes para localizar o problema. Ferramentas interativas de depuração, que mostram os valores intermediários das variáveis de programa, e uma lista dos comandos executados podem ser utilizadas como auxílio no processo de depuração.

3.5 Validação de software

A validação de software ou, de modo mais geral, verificação e validação (V & V), destina-se a mostrar que um sistema está de acordo com suas especificações e que ele atende às expectativas do cliente comprador do sistema. Esse processo envolve verificar processos, como por meio de inspeções e revisões (veja o Capítulo 19), em cada estágio do processo de software, desde a definição dos requisitos dos usuários até o desenvolvimento do programa. A maior parte dos custos de validação, contudo, é observada depois da implementação, quando o sistema operacional é testado (veja o Capítulo 20).

Exceto para pequenos programas, os sistemas não devem ser testados como uma unidade isolada, 'monolítica'. Os grandes sistemas são constituídos a partir de subsistemas, que são construídos a partir de módulos, que são compostos por procedimentos e funções. O processo de teste deve, por conseguinte, evoluir em estágios, ou seja, os testes são realizados incrementalmente, em conjunto com a implementação do sistema.

Figura 3.10 ▶
O processo de depuração.



A Figura 3.11 mostra um processo de testes composto de cinco estágios, no qual são testados os componentes do sistema, o sistema integrado e, finalmente, o sistema com os dados do cliente. Como ideal, os defeitos dos componentes são descobertos no início do processo, e os problemas de interface são detectados quando o sistema é integrado. Contudo, à medida que defeitos são descobertos, o programa precisa ser depurado, e isso pode exigir a repetição de outros estágios no processo de testes. Erros nos componentes do programa podem ser verificados durante os testes de integração. O processo é, portanto, iterativo, com as informações sendo fornecidas em feedback a partir dos estágios avançados para as partes iniciais do processo.

Os estágios do processo de testes são:

1. *Teste de unidade* São testados os componentes individuais, para garantir que eles operam corretamente. Cada componente é testado independentemente, sem outros componentes do sistema.

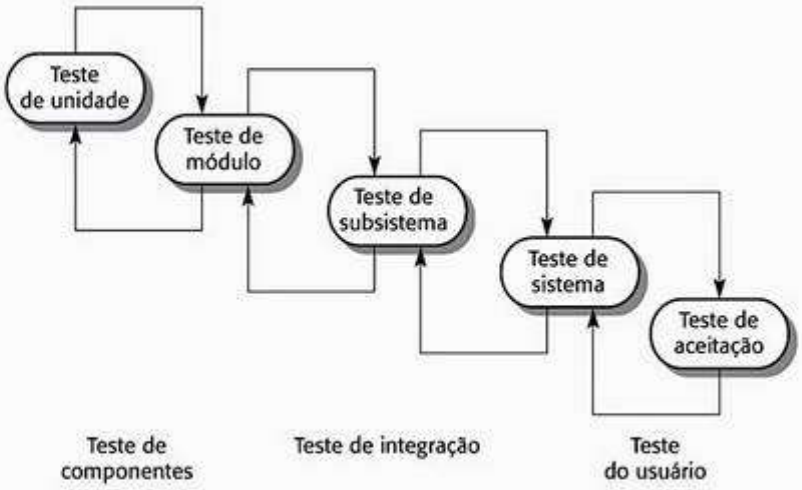
2. *Teste de módulo* Um módulo é uma coleção de componentes dependentes, como uma classe de objetos, um tipo de dado abstrato ou algum conjunto mais amplo de procedimentos e funções. Um módulo abrange componentes relacionados e, assim, pode ser testado sem outros módulos do sistema.

3. *Teste de subsistema* Essa fase envolve testar conjuntos de módulos que foram integrados em subsistemas. Os problemas mais comuns que surgem em grandes sistemas de software são discordâncias referentes a interfaces. O processo de teste de subsistemas deve, portanto, se concentrar na detecção de erros de interfaces de módulos, mediante a utilização rigorosa dessas interfaces.

4. *Teste de sistema* Os subsistemas são integrados para constituírem o sistema. Esse processo se dedica a encontrar erros que resultem de interações não previstas entre subsistemas e de problemas com a interface de subsistemas. Ele também se ocupa de validar que o sistema cumpra seus requisitos funcionais e não funcionais e de testar as propriedades emergentes do sistema.

5. *Teste de aceitação* Esse é o estágio final do processo de testes, antes que o sistema seja aceito para uso operacional. O sistema é testado com os dados fornecidos pelo cliente, no lugar dos dados de teste simulados. O teste de aceitação pode revelar erros e omissões na definição de requisitos de sistema, porque os dados reais 'exercitam' o sistema de modo diferente dos dados de teste. O teste de aceitação pode também revelar problemas com os requisitos, quando os recursos do sistema, na verdade, não atendem às necessidades do usuário ou quando o desempenho do sistema é inaceitável.

Figura 3.11 ▶
O processo de teste.



Os testes de unidade e os testes de módulo normalmente são de responsabilidade dos programadores que desenvolvem o componente. Os programadores produzem seus próprios dados de testes e incrementalmente testam o código, enquanto ele é desenvolvido. Essa é uma abordagem economicamente sensata, uma vez que o programador conhece melhor o componente e é, portanto, a melhor pessoa para gerar os dados de teste. O teste de unidade é parte do processo de implementação, e espera-se que um componente, em conformidade com sua especificação, seja entregue como parte do processo.

Os estágios de teste posteriores envolvem o trabalho de integração a partir de uma série de programadores e devem ser planejados com antecedência. Uma equipe independente de testadores deve trabalhar com planos de testes previamente elaborados, que são desenvolvidos a partir da especificação e do projeto do sistema. A Figura 3.12 ilustra como os planos de testes são o elo entre as atividades de teste e de desenvolvimento.

O teste de aceitação é, às vezes, chamado de *teste alfa*. Os sistemas encomendados são desenvolvidos para um único cliente. O processo de teste alfa continua até que o responsável pelo desenvolvimento do sistema e o cliente concordem que o sistema entregue é uma implementação aceitável dos requisitos do sistema.

Quando um sistema se destina a ser comercializado como um produto de software, um processo de teste, chamado de *teste beta*, é utilizado com freqüência. O teste beta envolve entregar um sistema a uma série de clientes em potencial, que concordam em utilizar esse sistema. Eles relatam os problemas encontrados para os desenvolvedores de sistema. Esse processo expõe o produto ao uso real e detecta erros que podem não ter sido previstos pelos construtores do sistema. Depois desse feedback, o sistema é modificado e liberado para o prosseguimento dos testes beta ou para a venda geral.

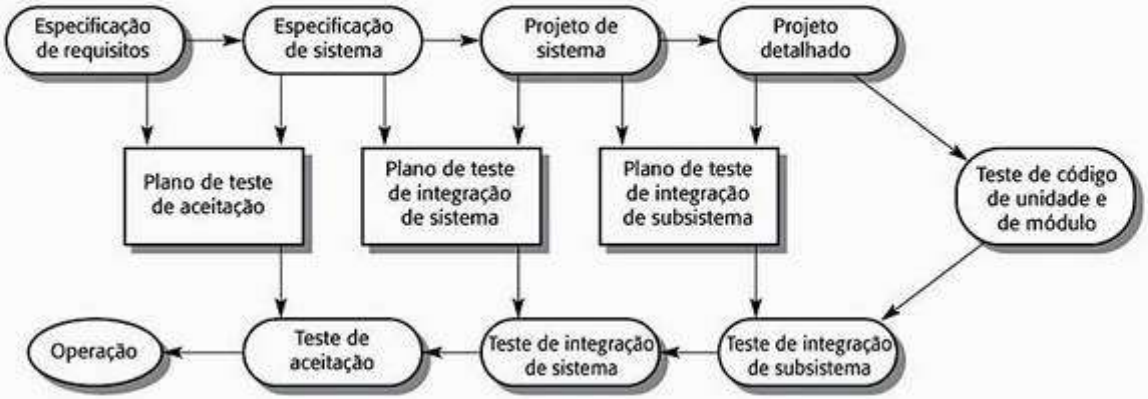
3.6 Evolução de software

A flexibilidade de sistemas de software é uma das principais razões pelas quais, cada vez mais, os softwares estão sendo incorporados em sistemas grandes e complexos. Uma vez tomada uma decisão de fabricar hardware, é muito dispendioso fazer modificações no projeto de hardware. Contudo, quando se trata de software, as mudanças podem ser feitas em qualquer momento, seja durante ou depois do desenvolvimento do sistema. Essas modificações podem ser muito onerosas, mas ainda são muito mais baratas do que as modificações correspondentes no sistema de hardware.

Historicamente, sempre houve um limite entre o processo de desenvolvimento de software e o processo de evolução de software (manutenção de software). O desenvolvimento de software é considerado uma atividade criativa, em que um sistema de software é desenvolvido a partir de um conceito inicial até chegar ao sistema em operação. A manutenção de soft-

Figura 3.12 ▼

Fases de teste no processo de software.



ware é o processo de modificar esse sistema, depois que ele entrou em operação. Embora os custos de 'manutenção' sejam, com freqüência, muitas vezes maiores do que os custos de desenvolvimento inicial, os processos de manutenção são considerados menos desafiadores do que o desenvolvimento de software original.

Esse limite está se tornando cada vez mais irrelevante. Poucos sistemas de software são, atualmente, sistemas completamente novos, e tem muito mais sentido considerar o desenvolvimento e a manutenção como estágios integrados e contínuos. Em vez de dois processos separados, é mais realista pensar na engenharia de software como um processo evolucionário, em que o software é continuamente modificado ao longo de seu tempo de duração, em resposta a requisitos em constante modificação e às necessidades do cliente. Esse processo evolucionário está ilustrado na Figura 3.13.

3.7 Apoio ao processo automatizado

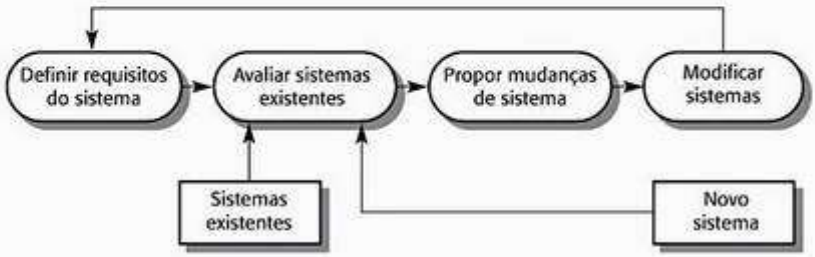
A já mencionada ferramenta CASE, ou engenharia de software com o auxílio de computador, é o nome dado ao software utilizado para apoiar as atividades de processo de software, como a engenharia de requisitos, o projeto, o desenvolvimento de programa e os testes. As ferramentas CASE, portanto, incluem editores de projeto, dicionários de dados, compiladores, depuradores, ferramentas de construção de sistemas, entre outros.

A tecnologia CASE proporciona apoio ao processo de software pela automação de algumas atividades de processo e pelo fornecimento de informações sobre o software que está sendo desenvolvido. Entre os exemplos de atividades que podem ser automatizadas utilizando-se CASE estão:

1. o desenvolvimento de modelos gráficos de sistemas, como parte das especificações de requisitos ou do projeto de software;
2. a compreensão de um projeto utilizando-se um dicionário de dados que contém informações sobre as entidades e sua relação em um projeto;
3. a geração de interfaces com usuários, a partir de uma descrição gráfica da interface, que é criada interativamente pelo usuário;
4. a depuração de programas, pelo fornecimento de informações sobre um programa em execução;
5. a tradução automatizada de programas, a partir de uma antiga versão de uma linguagem de programação, como Cobol, para uma versão mais recente.

A tecnologia CASE está atualmente disponível para a maioria das atividades de rotina no processo de software. Isso proporcionou algumas melhorias na qualidade e na produtividade de software, embora menos do que o previsto pelos primeiros defensores da tecnologia CASE. Eles sugeriram que era provável fazer enormes melhorias, se fossem utilizados ambientes CASE integrados. Na verdade, as verdadeiras melhorias que foram atingidas são da ordem de 40 por cento (Huff, 1992). Embora essa porcentagem seja significativa, a tecnologia CASE não revolucionou a engenharia de software como foi previsto no passado.

Figura 3.13 ▶
Evolução de sistema.



As melhorias possibilitadas pelo uso de CASE são limitadas por dois fatores:

1. A engenharia de software é, essencialmente, uma atividade de projeto que tem como base o pensamento criativo. Os sistemas CASE existentes automatizam atividades de rotina, mas as tentativas de aproveitamento da tecnologia de inteligência artificial para fornecer apoio ao projeto não foram bem-sucedidas.
2. Na maioria das organizações, a engenharia de software é uma atividade realizada em equipe e os engenheiros de software dedicam bastante tempo interagindo com os outros membros da equipe. A tecnologia CASE não é muito compatível com essa situação.

Não está definido se essa situação se modificará no futuro. Meu ponto de vista é de que é improvável que produtos CASE específicos apóiem o trabalho de equipe relacionado ao projeto de software e à engenharia de software. Contudo, sistemas genéricos de apoio à cooperação e a projetos serão desenvolvidos e adaptados para utilização no processo de software.

Atualmente, a tecnologia CASE está plenamente desenvolvida, e as ferramentas e *workbenches* CASE estão disponíveis em uma ampla gama de fornecedores. Contudo, em vez de focalizar alguma ferramenta específica, simplesmente apresentamos aqui uma visão geral, abordando o apoio específico em capítulos relacionados do livro.

3.7.1 A classificação CASE

As classificações CASE nos auxiliam a compreender os diferentes tipos de ferramentas CASE e seu papel em apoiar as atividades de processo de software. Há vários modos de classificar as ferramentas CASE, cada um dos quais nos oferece uma perspectiva diferente sobre essas ferramentas. Nesta seção, discuto as ferramentas CASE sob três dessas perspectivas:

1. *Uma perspectiva funcional* em que as ferramentas CASE são classificadas de acordo com sua função específica.
2. *Uma perspectiva de processo* em que as ferramentas são classificadas de acordo com a atividade de processo que elas apóiam.
3. *Uma perspectiva de integração* em que as ferramentas CASE são classificadas de acordo com a maneira como são organizadas em unidades integradas, que fornecem apoio a uma ou mais atividades de processo.

A Figura 3.14 é uma classificação das ferramentas CASE de acordo com sua função. Essa tabela enumera uma série de diferentes tipos de ferramentas CASE e fornece exemplos específicos de cada ferramenta. Não se trata de uma lista completa das ferramentas CASE, pois não foram incluídas ferramentas especializadas, como as ferramentas de apoio ao reuso.

A Figura 3.15 apresenta uma classificação alternativa das ferramentas CASE e mostra as fases de processo compatíveis com uma série de diferentes tipos de ferramentas CASE. As ferramentas para planejamento e estimativa, edição de textos, preparação de documentos e gerenciamento de configuração podem ser utilizadas ao longo do processo de software.

A amplitude do apoio ao processo de software, proporcionado pela tecnologia CASE, é outra possível dimensão de classificação. Fuggetta (1993) propõe que os sistemas CASE sejam classificados em três categorias:

1. *Ferramentas* apóiam as tarefas de processo individuais, como a verificação da consistência de um projeto, a compilação de um programa, a comparação de resultados de testes, entre outras. As ferramentas podem ser de uso geral, de uso restrito (por exemplo, um processador de textos) ou ser agrupadas em *workbenches*.

Tipo de ferramenta	Exemplos
Ferramentas de planejamento	Ferramentas PERT, ferramentas de estimativa, planilhas de cálculo
Ferramentas de edição	Editores de texto, editores de diagrama, processadores de texto
Ferramentas de gerenciamento de mudança	Ferramentas de controle de requisitos, sistemas de controle de mudança
Ferramentas de gerenciamento de configuração	Sistemas de gerenciamento de versão, ferramentas de construção de sistemas
Ferramentas de prototipação	Linguagens de nível muito alto, geradores de interfaces com o usuário
Ferramentas de apoio a métodos	Editores de projeto, dicionários de dados, geradores de código
Ferramentas de processamento de linguagem	Compiladores, interpretadores
Ferramentas de análise de programa	Geradores de referência cruzada, analisadores estáticos, analisadores dinâmicos
Ferramentas de testes	Geradores de dados de testes, comparadores de arquivos
Ferramentas de depuração	Sistemas interativos de depuração
Ferramentas de documentação	Programas de layout de página, editores de imagem
Ferramentas de reengenharia	Sistemas de referência cruzada, sistemas de reestruturação de programas

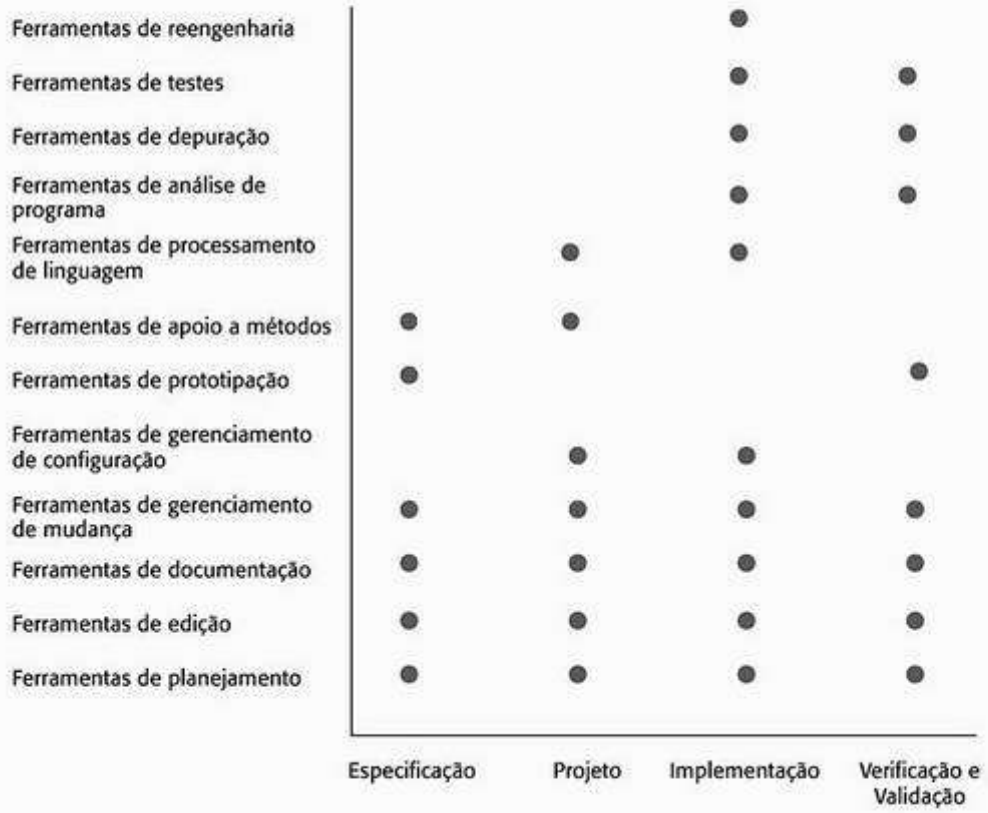
Figura 3.14 ▲

Classificação funcional das ferramentas CASE.

2. *Workbenches* apóiam fases ou atividades de processo, como a especificação e o projeto. Normalmente, consistem em um conjunto de ferramentas com maior ou menor grau de integração.
3. *Ambientes* oferecem apoio a todo ou, pelo menos, a uma parte substancial do processo de software. Normalmente, incluem vários *workbenches* diferentes, que são integradas de alguma maneira.

Figura 3.15 ▶

Classificação de ferramentas CASE com base em atividades.



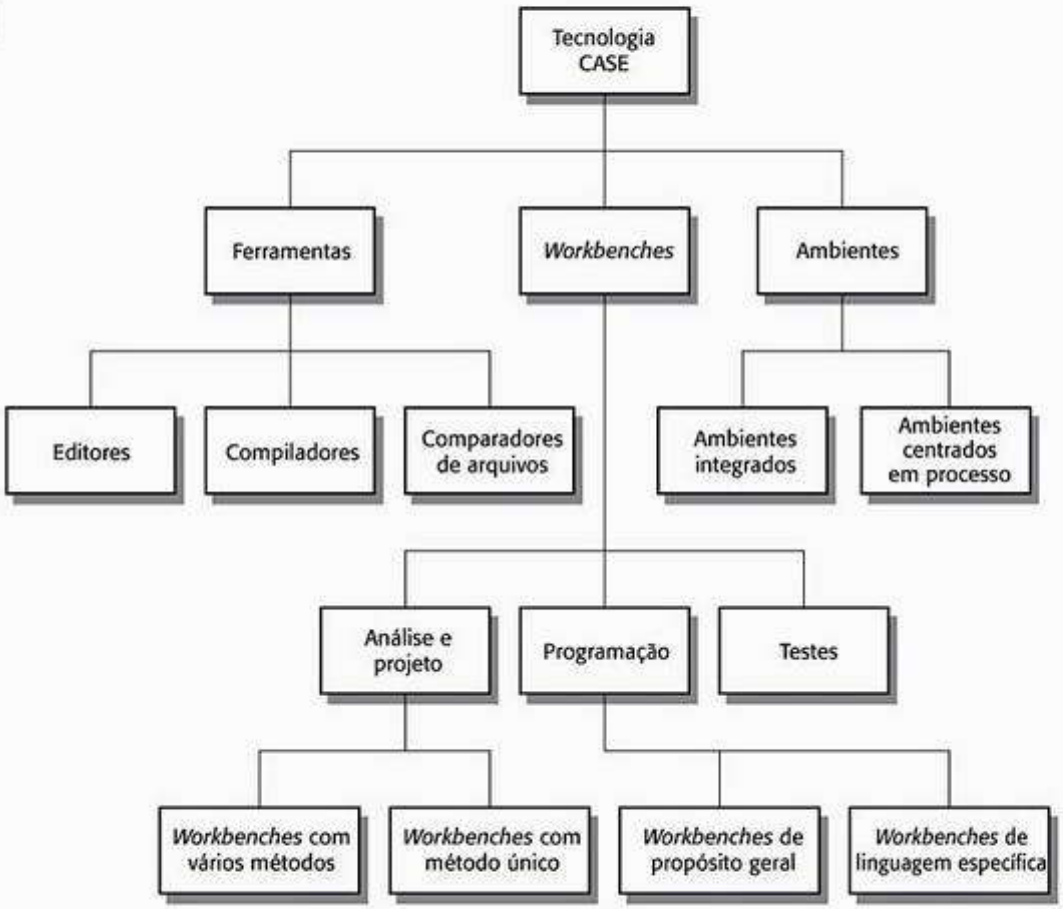
A Figura 3.16 ilustra essa classificação e mostra alguns exemplos dessas diferentes classes de apoio à CASE. Naturalmente, ela é simplesmente um exemplo ilustrativo; muitos tipos de ferramentas e *workbenches* foram deixados de fora nesse diagrama.

As ferramentas de propósito geral são utilizadas com bom senso pelos engenheiros de software, que tomam decisões sobre quando aplicá-las para proporcionar apoio ao processo. As *workbenches*, contudo, normalmente apóiam algum método que inclui um modelo de processo e um conjunto de regras/diretrizes, que se aplicam ao software em desenvolvimento. Classifiquei os ambientes como ambientes integrados ou ambientes centrados em processo. Os ambientes integrados proporcionam apoio à infra-estrutura para a integração de dados, controle e apresentação. Os ambientes centrados em processo são mais genéricos. Eles incluem o conhecimento do processo de software e um mecanismo de processo que utiliza esse modelo para orientar os engenheiros sobre quais ferramentas ou *workbenches* devem ser aplicadas e quando devem ser utilizadas.

Na prática, é tênue o limite entre essas diferentes classes. As ferramentas podem ser vendidas como um produto único, mas podem incluir o apoio a diferentes atividades. Por exemplo, a maioria dos processadores de texto atualmente proporciona um editor de diagramas incorporado. As *workbenches* CASE para projeto cada vez mais oferecem apoio à programação e aos testes e, assim, são mais parecidas com ambientes do que as bancadas especializadas. Portanto, talvez nem sempre seja fácil posicionar um produto utilizando-se uma classificação. Contudo, a classificação proporciona uma primeira etapa útil para ajudar a compreender a extensão do apoio ao processo que uma ferramenta pode proporcionar.

Figura 3.16 ▶

Ferramentas, *workbenches* e ambientes.



PONTOS-CHAVE

- Os processos de software são as atividades envolvidas na produção de um sistema de software. Os modelos de processo de software são representações abstratas desses processos.
- Todos os processos de software incluem especificação de software, projeto e implementação de software, validação de software e evolução de software.
- Os modelos de processo genéricos descrevem a organização dos processos de software. Dentre os exemplos de modelos genéricos destacam-se o modelo em cascata, o desenvolvimento evolucionário, o desenvolvimento formal de sistemas e o desenvolvimento orientado a reuso.
- Os modelos de processo iterativos apresentam o processo de software como um ciclo de atividades. A vantagem dessa abordagem é que ela evita compromissos prematuros com uma especificação ou projeto. Dentre os exemplos de modelos iterativos destacam-se o desenvolvimento incremental e o modelo em espiral.
- A engenharia de requisitos é o processo de desenvolvimento de uma especificação de software. Ela se refere a desenvolver uma especificação que possa ser compreendida pelos usuários do sistema e uma especificação mais detalhada para os desenvolvedores do sistema.
- Os processos de projeto e implementação se ocupam da transformação de uma especificação de requisitos em um sistema executável. Métodos de projeto sistemático podem ser utilizados como parte dessa transformação.
- A validação de software é o processo de verificar se o sistema está em conformidade com sua especificação e se ele atende às reais necessidades dos usuários do sistema.
- A evolução de software se ocupa de modificar os sistemas de software existentes, para que eles atendam a novos requisitos. Isso está se tornando a abordagem normal de desenvolvimento de software para sistemas pequenos e de porte médio.
- A tecnologia CASE proporciona apoio automatizado aos processos de software. As ferramentas CASE oferecem apoio às atividades de processo individuais; as *workbenches* CASE apóiam um conjunto de atividades relacionadas; os ambientes CASE são compatíveis com todas ou com a maioria das atividades de processo de software.

LEITURAS SUGERIDAS

"Embracing change with extreme programming". Uma visão geral interessante desse novo processo gradativo, que foi projetado para uso em conjunto com o desenvolvimento orientado a objetos. (K. Beck, *IEEE Computer*, 32(10), out. 1999.)

Managing software quality and business risk. Trata-se de um livro sobre gerenciamento de software, mas apresenta um excelente capítulo (Capítulo 4) sobre modelos de processo (M. Ould, 1999, John Wiley and Sons.)

"Software process models". Escrevi uma descrição mais detalhada de modelos de processo de software no *Handbook of computer science and engineering*. (A. Tucker (ed.), 1997, CRC Press.)

"A classification of CASE technology". O esquema de classificação proposto neste artigo foi utilizado aqui, mas Fuggetta oferece mais detalhes e ilustra como uma série de produtos comerciais se encaixa nesse esquema. (A. Fuggetta, *IEEE Computer*, 26(12), dez. 1993.)

EXERCÍCIOS

3.1 Justifique suas respostas, com base no tipo de sistema que está sendo desenvolvido, sugerindo o modelo mais apropriado de processo de software genérico que pode ser utilizado como base para o gerenciamento do desenvolvimento dos seguintes sistemas:

- um sistema para controlar o mecanismo contra arrombamento de fechaduras, em um veículo;
- um sistema de realidade virtual para apoiar a manutenção de software;
- um sistema de contabilidade para universidades, que substitua um sistema existente, e
- um sistema interativo para passageiros de ferrovias, que encontre os horários dos trens a partir de terminais instalados nas estações.

3.2 Explique por que programas que são desenvolvidos utilizando o desenvolvimento evolucionário apresentam difícil manutenção.

3.3 Explique como o modelo em cascata do processo de software e o modelo de prototipação podem ser acomodados no modelo de processo em espiral.

3.4 Sugira por que é importante fazer uma distinção entre desenvolver os requisitos do usuário e desenvolver os requisitos do sistema, no processo de engenharia de requisitos.

3.5 Descreva as principais atividades no processo de projeto de software e as saídas dessas atividades. Utilizando um diagrama de relacionamento de entidades, mostre os possíveis relacionamentos entre as saídas dessas atividades.

3.6 Quais são os cinco componentes de um método de projeto? Considere qualquer método que você conheça e descreva seus componentes. Avalie o grau de exatidão do método escolhido.

3.7 Projete um modelo de processo para efetuar testes de sistema e registrar seus resultados.

3.8 Explique por que um sistema de software utilizado em um ambiente real deve ser modificado ou, progressivamente, se tornará menos útil.

3.9 Sugira como um esquema de classificação da tecnologia CASE pode ser útil para os gerentes responsáveis pela aquisição de sistemas CASE.

3.10 Pesquise a disponibilidade de ferramentas em seu ambiente local de desenvolvimento e classifique as ferramentas de acordo com os parâmetros (função, atividade, amplitude de apoio) sugeridos aqui.

3.11 Historicamente, a introdução de tecnologia tem provocado profundas mudanças no mercado de trabalho e, pelo menos temporariamente, fez com que algumas pessoas perdessem o emprego. Discuta se a introdução da tecnologia CASE pode ter as mesmas conseqüências para os engenheiros de software. Se você achar que não, explique por quê. Se achar que essa tecnologia reduzirá as oportunidades de emprego, será ético que os engenheiros que podem ser afetados resistam à introdução dessa tecnologia, passiva e ativamente?

4 Gerenciamento de projetos

Objetivos

O objetivo deste capítulo é proporcionar uma visão geral do gerenciamento de projetos de software. Depois de ler este capítulo, você poderá:

- compreender as diferenças entre o gerenciamento de projetos de software e outros tipos de gerenciamento de projetos de engenharia;
- conhecer as principais tarefas dos gerentes de projeto de software;
- compreender por que o planejamento de projetos é essencial em todos os projetos de software;
- compreender como representações gráficas (diagramas de barras, diagramas de atividades) são utilizadas pelos gerentes de projeto para representar programações de projeto, e
- compreender o processo de gerenciamento de riscos e alguns dos riscos que podem surgir em projetos de software.

Sumário

4.1 Atividades de gerenciamento
4.2 Planejamento de projeto
4.3 Programação de projeto
4.4 Gerenciamento de riscos

O fracasso de muitos grandes projetos de software, na década de 60 e no início da década de 70, foi a primeira indicação das dificuldades de gerenciamento de software. O software era entregue com atraso, não era confiável, custava várias vezes mais do que previam as estimativas originais e, muitas vezes, exibia características precárias de desempenho (Brooks, 1975). Esses projetos não fracassaram porque os gerentes ou os programadores eram incompetentes. Ao contrário, esses projetos grandes e desafiadores atraíam pessoas de capacitação acima da média. A falha residia na abordagem de gerenciamento utilizada. Técnicas de gerenciamento provenientes de outras disciplinas da engenharia eram aplicadas e mostravam-se ineficazes para o desenvolvimento de software.

A necessidade de gerenciamento é uma importante distinção entre o desenvolvimento profissional de software e a programação em nível amador. Necessitamos do gerenciamento de projetos de software porque a engenharia de software profissional está sempre sujeita a restrições de orçamento e de prazo. Essas restrições são estabelecidas pela organização que desenvolve o software. O trabalho do gerente de projeto de software é garantir que o projeto de software cumpra essas restrições e entregar um produto de software que contribua para as metas da empresa.

Os gerentes de software são responsáveis por planejar e programar o desenvolvimento do projeto. Eles supervisionam o trabalho para assegurar que ele seja realizado em conformidade com os padrões requeridos e monitoram o progresso para verificar se o desenvolvimento está dentro do prazo e do orçamento. O bom gerenciamento não pode garantir o sucesso do projeto. Contudo, o mau gerenciamento geralmente resulta no fracasso do projeto. O software é entregue com atraso, custa mais do que originalmente foi estimado e apresenta falha no cumprimento de seus requisitos.

Os gerentes de software fazem o mesmo tipo de trabalho que outros gerentes de projeto de engenharia. Contudo, a engenharia de software é distinta de outros tipos de engenharia, em uma variedade de modos que podem tornar o gerenciamento de software particularmente difícil. Algumas dessas diferenças são:

1. *O produto é intangível* O gerente do projeto de construção de um navio ou de um projeto de engenharia civil pode ver o produto sendo desenvolvido. Se há um atraso na programação, o efeito no produto é visível. Partes da estrutura estão obviamente inacabadas. O software é intangível; não pode ser visto ou tocado. Os gerentes de projeto de software não podem ver o progresso, eles dependem de outras pessoas para produzir a documentação necessária, a fim de examinar o progresso.

2. *Não há processo de software-padrão* Não temos uma compreensão clara das relações entre o processo de software e os tipos de produto. Nas disciplinas de engenharia com longo histórico, o processo é experimentado e testado. O processo de engenharia para determinados tipos de sistema, como uma ponte, é bem compreendido. Nossa compreensão do processo de software se desenvolveu significativamente nos últimos anos. Contudo, ainda não podemos prever com certeza quando um processo de software específico poderá causar problemas de desenvolvimento.

3. *Grandes projetos de software são, freqüentemente, projetos únicos* Os grandes projetos de software são normalmente diferentes de projetos anteriores. Os gerentes, portanto, na verdade, têm ampla experiência anterior, que pode ser utilizada para reduzir a incerteza nos planos. Conseqüentemente, é mais difícil prever problemas. Além disso, as rápidas mudanças tecnológicas em computadores e nas comunicações desatualizam as experiências prévias. As lições aprendidas com essas experiências podem não ser transferíveis para novos projetos.

Por causa desses problemas, não é de surpreender que alguns projetos de software sofram atrasos e ultrapassem o orçamento previsto. Os sistemas de software são, freqüentemente, novos e tecnicamente inovadores. Os projetos de engenharia (como novos sistemas de transporte), muitas vezes, têm problemas relacionados ao cronograma. Em face das dificuldades

envolvidas, talvez seja até notável que tantos projetos de software sejam entregues dentro do prazo e do orçamento.

O gerenciamento de projeto de software é um tópico amplo e não pode ser abordado em um único capítulo. Portanto, neste capítulo, simplesmente faço uma introdução ao assunto e descrevo três importantes atividades de gerenciamento, como: planejamento de projeto, programação de projeto e gerenciamento de riscos. Os capítulos posteriores (na Parte 6) tratam de outros aspectos do gerenciamento de software, entre eles gestão de pessoal, estimativa de custos de software e gerenciamento de qualidade.

4.1 Atividades de gerenciamento

É impossível dar uma descrição do trabalho-padrão de um gerente de software. O trabalho varia muito, dependendo da organização e do produto a ser desenvolvido. Contudo, a maioria dos gerentes assume a responsabilidade, em algum estágio, por algumas das seguintes atividades ou por todas elas:

- elaboração de propostas
- planejamento e programação de projetos
- custo do projeto
- monitoramento e revisões de projetos
- seleção e avaliação de pessoal
- elaboração de relatórios e apresentações

O primeiro estágio de um projeto de software pode envolver a elaboração de uma proposta para executar o projeto. A proposta descreve os objetivos do projeto e como ele será realizado. Em geral, inclui também estimativas de custo e programação. Isso pode justificar por que o contrato do projeto deve ser delegado a uma organização ou equipe específica.

A elaboração da proposta é uma tarefa crucial, uma vez que a existência de muitas empresas de software depende de um número suficiente de propostas aceitas e contratos firmados. Pode não haver diretrizes definidas para essa tarefa; a elaboração de propostas é uma habilidade adquirida com a experiência. Aron (1983) apresenta uma argumentação sobre esse aspecto do trabalho de um gerente de projeto que ainda é relevante atualmente.

O planejamento de projeto se ocupa de identificar as atividades, os marcos e os documentos a serem produzidos em um projeto. Um plano deve, então, ser traçado para guiar o desenvolvimento em direção aos objetivos do projeto. A estimativa de custos é uma atividade que se ocupa de estimar os recursos requeridos para realizar o projeto. Esses assuntos serão abordados com mais detalhes posteriormente, neste capítulo e no Capítulo 23.

O monitoramento de projeto é uma atividade contínua. O gerente deve manter o acompanhamento do andamento do projeto e comparar os progressos e custos reais com os que foram planejados. Embora a maioria das organizações tenha mecanismos formais para o monitoramento, um gerente hábil pode formar um quadro mais nítido do que está acontecendo mediante discussões informais com a equipe de projetos.

O monitoramento informal pode prever problemas em potencial no projeto, no sentido de que pode revelar dificuldades à medida que elas ocorrem. Por exemplo, discussões diárias com a equipe de projetos podem revelar um problema específico ao encontrar uma falha de software. Em vez de esperar que um atraso na programação seja relatado, o gerente de software pode designar um especialista para resolver o problema ou pode decidir que o projeto seja convenientemente reprogramado.

Durante um projeto, é normal ocorrer uma série de revisões formais pelo gerenciamento de projetos. Essas revisões se ocupam de examinar o progresso geral, o desenvolvimento técnico do projeto, considerando o status do projeto em relação aos objetivos da organização que autoriza a operação do software.

O prazo para o desenvolvimento de um grande projeto de software pode ser de vários anos. Durante esse tempo, os objetivos organizacionais muito provavelmente serão modificados. Essas mudanças podem significar que o software não será mais necessário ou que os requisitos iniciais do projeto são inadequados. O gerenciamento pode decidir interromper o desenvolvimento do projeto ou modificá-lo a fim de acomodar as mudanças aos objetivos da organização.

Os gerentes de projeto, geralmente, precisam selecionar pessoal para trabalhar em seu projeto. O ideal é que uma equipe hábil e com experiência apropriada esteja disponível para trabalhar no projeto. Contudo, na maioria dos casos, os gerentes têm de se contentar com menos do que uma equipe ideal de projeto. As razões disso são:

1. O orçamento do projeto pode não ser suficiente para contratar uma equipe bem paga, e pode ser necessário utilizar uma equipe menos experiente e que exija menor remuneração.
2. Uma equipe com a experiência apropriada pode não estar disponível, dentro ou fora da organização. Pode ser impossível recrutar uma equipe nova para o projeto. Dentro da organização, as melhores pessoas talvez já estejam alocadas em outros projetos.
3. A organização pode querer desenvolver as habilidades de seus funcionários, por isso pode ser designada uma equipe inexperiente para um projeto, a fim de que essa equipe aprenda e ganhe experiência.

O gerente de software precisa trabalhar dentro dessas limitações quando seleciona a equipe de projeto. Contudo, é bem provável que ocorram problemas, a não ser que pelo menos um membro do projeto tenha alguma experiência no tipo de sistema a ser desenvolvido. Sem essa experiência, muitos erros simples podem ocorrer. Discutirei a constituição da equipe de projeto e a seleção de pessoal no Capítulo 22.

O gerente de projeto geralmente é o responsável pela preparação dos relatórios sobre o projeto, tanto para a organização do cliente como para as organizações dos fornecedores. Os gerentes de projeto devem redigir documentos concisos e coerentes, que resumam as informações fundamentais a partir de relatórios de projeto detalhados. Eles devem ser capazes de apresentar essas informações durante as revisões de andamento. Conseqüentemente, a capacidade de se comunicar de modo eficaz, tanto verbalmente como por escrito, é uma habilidade essencial para um gerente de projeto.

4.2 Planejamento de projeto

O gerenciamento eficaz de um projeto de software depende de um planejamento acurado do andamento do projeto. O gerente de projeto deve prever os problemas que podem surgir e preparar soluções experimentais para esses problemas. Um plano traçado no início do projeto deve ser utilizado como guia para esse projeto. O plano inicial deve ser o melhor possível, em face das informações disponíveis. Ele deve evoluir à medida que o projeto seja desenvolvido e melhores informações se tornem disponíveis.

Uma estrutura para um plano de desenvolvimento de software é descrita na Seção 4.2.1. Assim como um plano de projeto, os gerentes podem também ter de traçar outros tipos de planos. Esses planos estão brevemente descritos na Figura 4.1 e são abordados com mais detalhes no capítulo referente ao assunto, em outra parte do livro.

Figura 4.1 ▶

Tipos de plano.

Plano	Descrição
Plano de qualidade	Descreve os procedimentos para teste de qualidade que serão utilizados em um projeto. Veja o Capítulo 24.
Plano de validação	Descreve a abordagem, os recursos e o método utilizados para a validação do sistema. Veja o Capítulo 19.
Plano de gerenciamento de configuração	Descreve os procedimentos de gerenciamento e as estruturas a serem utilizadas. Veja o Capítulo 29.
Plano de manutenção	Prevê os requisitos de manutenção do sistema, os custos de manutenção e o esforço necessário. Veja o Capítulo 27.
Plano de desenvolvimento de equipe	Descreve como as habilidades e a experiência dos membros da equipe de projeto serão desenvolvidas. Veja o Capítulo 22.

O pseudocódigo, mostrado na Figura 4.2, descreve o processo de planejamento do projeto para o desenvolvimento de software. Ele mostra que o planejamento é um processo iterativo, que só termina quando o próprio projeto for concluído. À medida que as informações sobre o projeto se tornam disponíveis durante seu desenvolvimento, o plano deve ser regularmente revisado. Os objetivos gerais da empresa são um importante fator que deve ser considerado quando se formula o plano de projeto. Quando esses objetivos são modificados, são necessárias mudanças no plano do projeto.

O processo de planejamento se inicia com uma avaliação das restrições (a data de entrega estabelecida, o pessoal disponível, o orçamento total, entre outras) que afetam o projeto. Essa avaliação é realizada em conjunto com uma estimativa dos parâmetros para o projeto, como sua estrutura, seu tamanho e sua distribuição de funções. Os marcos de progresso e os produtos a serem entregues são então definidos. O processo entra, então, em um loop. Uma programação para o projeto é traçada e as atividades definidas na programação são iniciadas ou liberadas para prosseguimento. Depois de algum tempo (normalmente, depois de duas a três semanas), o andamento é examinado e as discrepâncias são observadas. Como as estimativas dos parâmetros de projeto são experimentais, o plano sempre precisará ser modificado.

Os gerentes de projeto revisam as suposições sobre o projeto, à medida que mais informações se tornam disponíveis, refazendo a programação do projeto. Se o projeto estiver atrasado, talvez tenham de renegociar com o cliente as restrições do projeto e os produtos a serem entregues. Se essa renegociação não for bem-sucedida e a programação não puder ser cumprida, poderá ser considerada uma revisão técnica do projeto. O objetivo dessa revisão

Figura 4.2 ▶

Planejamento de projeto.

```
Estabeleça as restrições do projeto
Faça a avaliação inicial dos parâmetros do projeto
Defina os marcos do projeto e os produtos a serem entregues
while projeto não estiver terminado ou cancelado loop
    Faça a programação do projeto
    Inicie as atividades de acordo com a programação
    Aguarde (por um período)
    Examine o progresso do projeto
    Revise as estimativas de parâmetros do projeto
    Atualize a programação do projeto
    Reanalise as restrições do projeto e os produtos a serem entregues
    If surgirem problemas, then
      Inicie revisão técnica
    end if
end loop
```

é encontrar uma abordagem de desenvolvimento alternativa, que se limite às restrições do projeto e cumpra a programação.

Naturalmente, gerentes de projeto experientes não supõem que tudo correrá bem, pois sempre surgirão problemas de algum tipo durante um projeto. As hipóteses e as programações assumidas inicialmente devem ser pessimistas, em vez de otimistas. É preciso tentar prever todas as possíveis eventualidades no plano de projeto, para que as restrições e os marcos do projeto não tenham de ser renegociados a todo tempo no loop de planejamento.

4.2.1 O plano de projeto

O plano de projeto define os recursos disponíveis para o projeto, a estrutura analítica do trabalho e uma programação para realizar o trabalho. Em algumas organizações, o plano de projeto é um documento único que incorpora todos os diferentes tipos de plano apresentados anteriormente. Em outros casos, o plano de projeto se ocupa exclusivamente do processo de desenvolvimento. São incluídas referências a outros planos, mas os planos são separados entre si.

A estrutura de plano que descrevo aqui é adequada a esse último tipo de plano. Os detalhes do plano de projeto variam, dependendo do tipo de projeto e da empresa. Contudo, a maioria dos planos deve incluir as seguintes seções:

1. *Introdução* Descreve com brevidade os objetivos do projeto e define as restrições (por exemplo, orçamento e prazo) que afetam o gerenciamento do projeto.
2. *Organização de projeto* Descreve o modo como a equipe de desenvolvimento é organizada, as pessoas envolvidas e seus papéis na equipe.
3. *Análise de riscos* Descreve possíveis riscos de projeto, a probabilidade de surgir esses riscos e as estratégias propostas para a redução deles. O gerenciamento de riscos é abordado na Seção 4.4.
4. *Requisitos necessários de hardware e software* Descreve o hardware e o software de apoio exigidos para realizar o desenvolvimento. Se o hardware tiver de ser comprado, deverão ser incluídos os prazos de entrega e as estimativas de preço.
5. *Estrutura analítica* Descreve a divisão do trabalho em atividades e identifica os marcos e os produtos a serem entregues com cada atividade. Os marcos e os produtos são discutidos na Seção 4.2.2.
6. *Programação de projeto* Descreve as dependências entre atividades, o tempo estimado requerido para atingir cada marco e a alocação de pessoas nas atividades.
7. *Mecanismos de monitoramento e de elaboração de relatórios* Descreve os relatórios de gerenciamento que devem ser produzidos, quando eles devem ser produzidos e quais mecanismos de monitoramento são utilizados.

O plano de projeto deve ser revisado regularmente durante o projeto. Algumas partes, como a programação do projeto, serão freqüentemente modificadas; outras partes serão mais estáveis. Deve ser utilizada uma organização de documentos que permita a substituição direta de seções.

4.2.2 Marcos e produtos a serem entregues

Os gerentes precisam de informações. Como o software é intangível, essas informações somente podem ser fornecidas como documentos que descrevem o estado do software que está sendo desenvolvido. Sem essas informações, é impossível julgar o progresso e as estimativas de custos e as programações não podem ser atualizadas.

Quando se planeja um projeto, uma série de *marcos* deve ser estabelecida; um marco é o ponto final de uma atividade de processo de software. A cada marco deve haver uma saída formal, como um relatório, que possa ser apresentado para a gerência. Os relatórios de marcos não precisam ser documentos extensos, eles podem simplesmente ser um breve relatório das realizações em uma atividade de projeto. Os marcos devem representar o final de um estágio distinto e lógico de um projeto. Marcos indefinidos, como, por exemplo, 'codificação 80 por cento concluída', que são de impossível validação, são inúteis para o gerenciamento do projeto.

Um *produto a ser entregue* é o resultado do projeto entregue ao cliente. Em geral, ele é entregue no término de alguma fase importante do projeto, como a especificação, o projeto etc.

Normalmente, os produtos a serem entregues são marcos, mas os marcos não precisam ser produtos a serem entregues. Os marcos podem ser resultados internos do projeto, que são utilizados pelo gerente de projeto na verificação do andamento do projeto, sem serem entregues ao cliente.

Para estabelecer marcos, o processo de software deve ser dividido em atividades básicas com saídas associadas. Por exemplo, a Figura 4.3 mostra atividades envolvidas na especificação de requisitos, quando se utiliza a prototipação para ajudar a validar os requisitos. As principais saídas para cada atividade (os marcos de projeto) são mostradas. Os produtos a serem entregues do projeto são a definição e a especificação de requisitos.

4.3 Programação de projeto

A programação de projeto é uma tarefa particularmente necessária para os engenheiros de projeto. Os gerentes estimam o tempo e os recursos exigidos para completar as atividades e os organizam em uma seqüência coerente. A menos que o projeto em programação seja similar a um projeto anterior, estimativas precedentes são uma base incerta para uma nova programação de projeto. A estimativa de programação é ainda mais complicada pelo fato de que diferentes projetos podem utilizar diferentes métodos de projeto e linguagens de implementação.

Se o projeto for tecnicamente avançado, as estimativas iniciais certamente serão otimistas, mesmo quando os gerentes tentam considerar todas as eventualidades. A esse respeito, a programação de software não é diferente da programação de qualquer outro tipo de grandes projetos avançados. O desenvolvimento de novas aeronaves, pontes e mesmo novos modelos de automóveis freqüentemente sofre algum atraso devido a problemas que não foram previstos. As programações, portanto, devem ser continuamente atualizadas, à medida que melhores informações sobre o progresso se tornam disponíveis.

A programação de projeto (Figura 4.4) envolve dividir o trabalho total de um projeto em atividades distintas e avaliar o tempo necessário para completar essas atividades. De modo

Figura 4.3 ▼

Marcos no processo de requisitos.

Figura 4.4 ▲
O processo de programação de projeto.

geral, algumas dessas atividades são realizadas em paralelo. Os programadores de projeto devem coordenar essas atividades paralelas e organizar o trabalho, de modo que a força de trabalho seja otimizada. Eles devem evitar uma situação em que todo o projeto é atrasado porque uma tarefa importante não foi concluída.

As atividades de projeto devem normalmente durar pelo menos uma semana. Subdivisões menores significam que um tempo desproporcional deve ser dedicado para a estimativa e a revisão do diagrama. Também é útil estabelecer um tempo máximo de duração para qualquer atividade, que vai de oito a dez semanas. Se uma atividade demorar mais tempo do que isso, ela deverá ser subdividida para a programação e o planejamento do projeto.

Ao estimar as programações, os gerentes não devem presumir que todos os estágios do projeto estarão livres de problemas. As pessoas que trabalham no projeto podem ficar doentes ou pedir demissão; algum hardware pode apresentar defeito, e o software ou hardware de suporte, que são essenciais, podem ser entregues com atraso. Se o projeto for novo e tecnicamente avançado, determinadas partes dele poderão ser mais difíceis e exigirem mais tempo do que foi originalmente previsto.

Os gerentes também devem estimar os recursos necessários para completar cada tarefa, além dos prazos de execução. O recurso principal é o esforço humano requerido. Outros recursos podem ser o espaço em disco exigido em um computador, o tempo de uso necessário de um hardware especializado, como um simulador, e o orçamento de viagens para o pessoal da equipe de projetos. Discutirei a estimativa com mais detalhes no Capítulo 23.

Uma boa regra prática geral é fazer a estimativa como se nada fosse acontecer de errado e, então, aumentar a estimativa, a fim de poder resolver possíveis problemas. Outras eventualidades destinadas a resolver problemas não previstos também podem ser acrescentadas à estimativa. Essas eventualidades extras dependem do tipo de projeto, dos parâmetros do processo (prazo final, padrões etc.) e do nível de experiência dos engenheiros de software que trabalham no projeto. Como regra geral, sempre acrescento 30 por cento à minha estimativa original, para a solução de possíveis problemas, e mais 20 por cento, para lidar com outros aspectos que não pudemos prever.

A programação de projeto é normalmente representada como um conjunto de diagramas, mostrando a estrutura analítica do trabalho, dependências das atividades e alocação de pessoal. Esses tópicos serão discutidos na seção seguinte. Ferramentas de gerenciamento de software, como o Microsoft Project, são, hoje em dia, comumente utilizadas para automatizar a produção de diagramas.

4.3.1 Diagramas de barras e redes de atividades

Os diagramas de barras e as redes de atividades são notações gráficas utilizadas para ilustrar a programação de projeto. Os diagramas de barras mostram quem é responsável por cada atividade e para quando está programado o início e o término da atividade. As redes de atividades mostram a dependência entre as diferentes atividades que constituem o projeto. Os diagramas de barras e as redes de atividades podem ser gerados automaticamente, a partir de um banco de dados de informações do projeto, com a utilização de uma ferramenta de gerenciamento de projeto.

Considere o conjunto de atividades indicado na Figura 4.5. Essa tabela mostra atividades, suas durações e suas interdependências. De acordo com a Figura 4.5, pode-se ver que a tarefa T3 depende da tarefa T1. Isso significa que T1 deve ser concluída antes que T3 inicie. Por exemplo, T1 pode ser a preparação do projeto de um componente e T3, a implementação desse projeto. Antes de iniciar a implementação, a preparação precisa estar terminada.

Considerando as dependências e as durações estimadas para as atividades, pode ser produzida uma rede de atividades que mostre as seqüências dessas atividades (Figura 4.6). Essa rede mostra quais atividades podem ser realizadas em paralelo e quais devem ser executadas em seqüência, devido à dependência em relação a uma atividade anterior. As atividades são representadas como retângulos. Os marcos e os produtos a serem entregues são indicados com cantos arredondados. As datas nesse diagrama mostram a data de início da atividade. O diagrama deve ser lido da esquerda para a direita e de cima para baixo.

Na ferramenta de gerenciamento de projeto utilizada para produzir esse diagrama, todas as atividades devem terminar em marcos. Uma atividade pode iniciar quando seu marco precedente (que pode depender de diversas atividades) tiver sido atingido. Portanto, na terceira coluna da Figura 4.5, mostrei também o marco correspondente (por exemplo, M5), que é atingido quando as atividades naquela coluna são concluídas (veja a Figura 4.6).

Antes de prosseguir de um marco para outro, todos os caminhos que levam até ele precisam estar completos. Por exemplo, a tarefa T9, mostrada na Figura 4.6, não pode ser iniciada até que as tarefas T3 e T6 estejam concluídas. A chegada ao marco M4 indica que essas tarefas foram completadas.

O tempo mínimo necessário para terminar o projeto pode ser estimado considerando o caminho mais longo no diagrama de atividades (o caminho principal). Nesse caso, o tempo é de 11 semanas ou 55 dias trabalhados. Na Figura 4.6, o caminho crítico é mostrado como uma seqüência de blocos com contorno em negrito. A programação total do projeto depende do caminho principal. Qualquer atraso na execução de qualquer atividade importante ocasiona o atraso do projeto.

Todavia, os atrasos ocorridos nas atividades que não estão no caminho principal não precisam causar um atraso geral da programação. Contanto que esses atrasos não estendam as atividades a ponto de o tempo total exceder o tempo no caminho principal, a programação do projeto não será afetada. Por exemplo, se a tarefa T8 se atrasar, talvez isso não afete a data

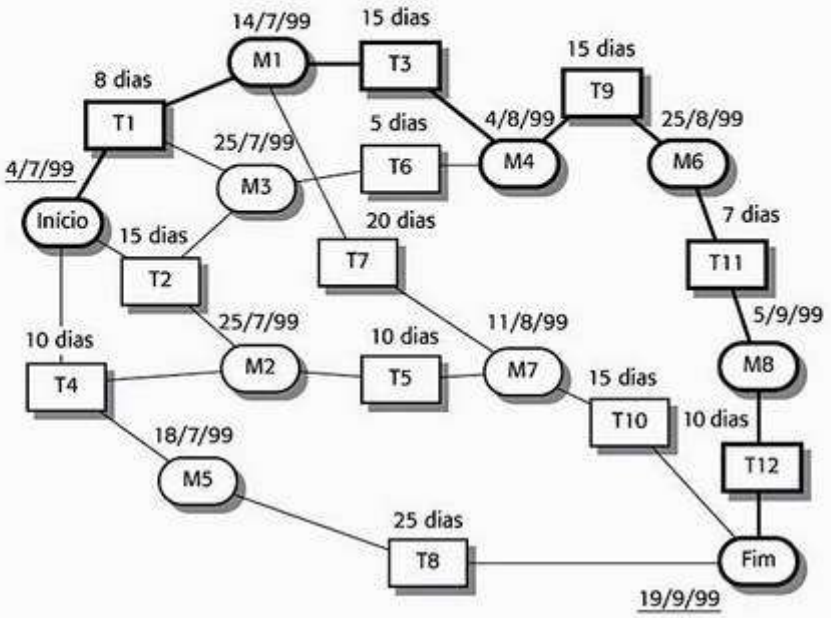
Figura 4.5 ▶

Duração e dependências das tarefas.

Tarefa	Duração (dias)	Dependências
T1	8	
T2	15	
T3	15	T1 (M1)
T4	10	
T5	10	T2, T4 (M2)
T6	5	T1, T2 (M3)
T7	20	T1 (M1)
T8	25	T4 (M5)
T9	15	T3, T6 (M4)
T10	15	T5, T7 (M7)
T11	7	T9 (M6)
T12	10	T11 (M8)

Figura 4.6

Uma rede de atividades.



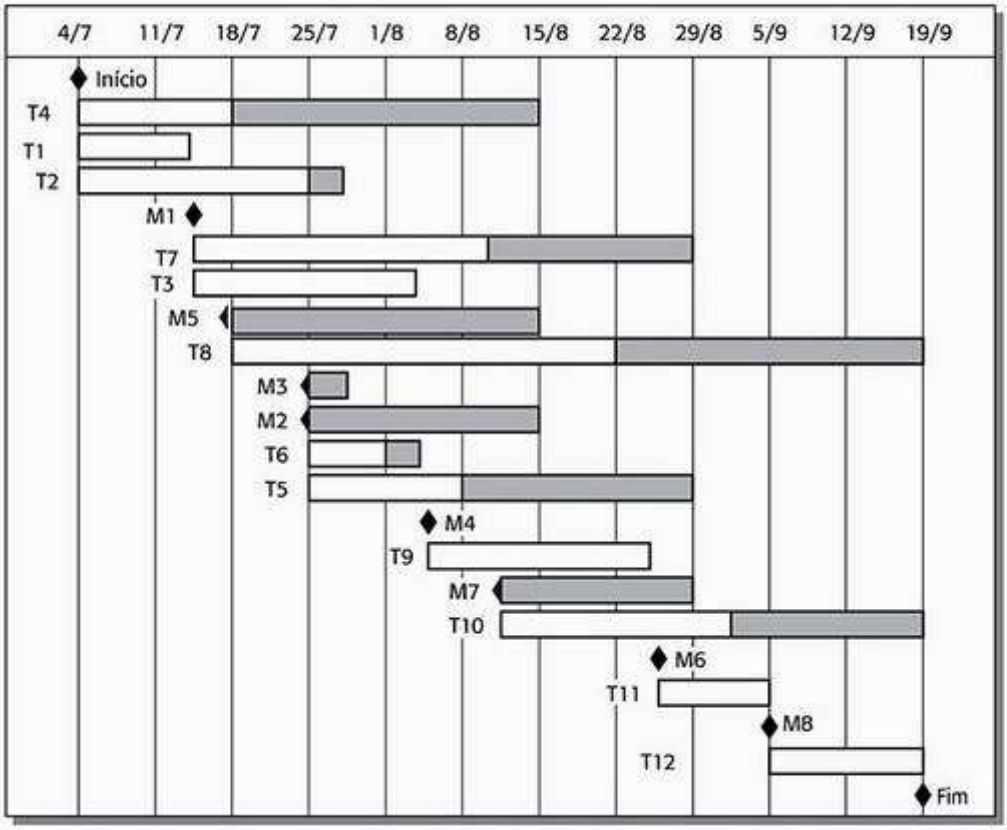
final de término do projeto, uma vez que ela não está no caminho principal. O diagrama de barras do projeto (Figura 4.7) mostra a amplitude de atraso possível com uma barra sombreada.

Os gerentes também utilizam as redes de atividades quando alocam o trabalho de projeto. Elas podem fornecer esclarecimentos sobre dependências que não são intuitivamente óbvias. Pode ser possível modificar o projeto de sistema, de modo que o caminho principal seja encurtado. O período de programação do projeto pode ser diminuído pela redução do tempo gasto na espera pela finalização de atividades.

Figura 4.7

Diagrama de barras de atividades.

A Figura 4.7 é uma maneira alternativa de representar as informações sobre a programação do projeto. Trata-se de um diagrama de barras (algumas vezes, chamado de diagrama de



Gantt, em homenagem ao seu inventor) mostrando um calendário de projeto e a data de início e término das atividades.

Algumas das atividades na Figura 4.7 são seguidas por uma barra sombreada, cuja duração é calculada pela ferramenta de programação. Isso mostra que existe alguma flexibilidade na data de conclusão dessas atividades. Se uma atividade não for concluída a tempo, o caminho principal não será afetado até o fim do período marcado pela barra sombreada. As atividades que estão no caminho principal não têm nenhuma margem de erro e podem ser identificadas porque não existe nenhuma barra sombreada associada a elas.

Além de considerar as programações, os gerentes de projeto devem também considerar a alocação de recursos e, em particular, a alocação de pessoal para as atividades de projeto. A Figura 4.8 sugere uma alocação de pessoal para as atividades ilustradas na Figura 4.7.

A Figura 4.8 também pode ser processada por ferramentas de apoio ao gerenciamento de projeto, e um diagrama de barras produzido, que mostra os períodos nos quais o pessoal está designado para o projeto (Figura 4.9). A equipe não precisa estar designada para um projeto durante o tempo todo. Em alguns períodos de intervalo, as pessoas podem estar de férias, trabalhando em outros projetos, assistindo a cursos de treinamento ou realizando alguma outra atividade.

As grandes empresas geralmente empregam uma série de especialistas que trabalham em um projeto, conforme necessário. Isso pode causar problemas de programação. Se um projeto se atrasar enquanto um especialista estiver trabalhando nele, isso poderá ter um efeito prejudicial sobre os outros projetos, que também poderão se atrasar, porque o especialista não está disponível.

Inevitavelmente, as programações iniciais de projetos estarão incorretas. À medida que um projeto se desenvolve, as estimativas devem ser comparadas com o tempo real empregado. Essa comparação pode ser utilizada como base para revisar a programação das partes posteriores do projeto. Quando os números reais são conhecidos, o diagrama de atividades deve ser reexaminado. As atividades de projeto seguintes podem então ser reorganizadas, a fim de reduzir a extensão do caminho principal.

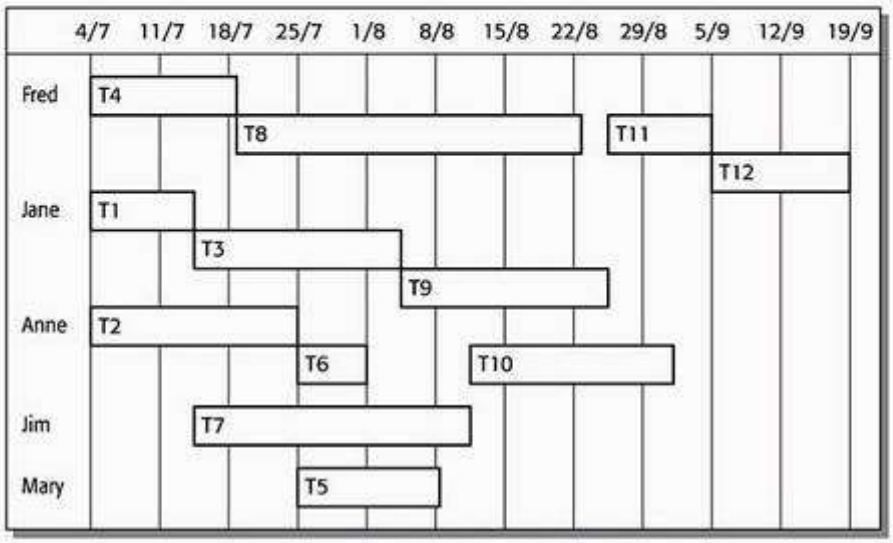
Figura 4.8 ▶

Alocação de pessoas para as atividades.

Tarefa	Engenheiro
T1	Jane
T2	Anne
T3	Jane
T4	Fred
T5	Mary
T6	Anne
T7	Jim
T8	Fred
T9	Jane
T10	Anne
T11	Fred
T12	Fred

Figura 4.9

Alocação de equipe *versus* diagrama de tempo.



4.4 Gerenciamento de riscos

Uma importante tarefa de um gerente de projeto é prever os riscos que podem afetar a programação do projeto ou a qualidade do software em desenvolvimento e tomar as medidas necessárias para evitar esses riscos. Os resultados da análise de riscos devem ser documentados no plano de projeto, juntamente com uma análise das conseqüências da ocorrência de algum fator de risco. Identificar riscos e traçar planos para minimizar seus efeitos sobre o projeto é o que se chama de gerenciamento de riscos (Hall, 1998; Ould, 1999).

De modo simplificado, podemos pensar no risco como uma probabilidade de que alguma circunstância adversa realmente venha a ocorrer. Os riscos podem ameaçar o projeto, o software que está sendo desenvolvido ou a organização. Essas categorias de riscos podem ser definidas como se segue:

1. *Riscos relacionados ao projeto* são os riscos que afetam a programação ou os recursos do projeto.
2. *Riscos relacionados ao produto* são os riscos que afetam a qualidade ou o desempenho do software que está em desenvolvimento.
3. *Riscos para os negócios* são os riscos que afetam a organização que está desenvolvendo ou adquirindo o software.

Naturalmente, essa não é uma classificação exclusiva. Se um programador experiente abandonar um projeto, isso poderá representar um risco de projeto (porque a entrega do sistema pode atrasar), um risco relacionado ao produto (porque um substituto pode não ser tão experiente e, portanto, pode cometer erros) e um risco para os negócios (porque aquela experiência não está disponível para elaborar propostas para negócios futuros).

O gerenciamento de riscos é particularmente importante para projetos de software, devido às incertezas inerentes que a maioria dos projetos enfrenta. Os riscos podem surgir como decorrência de requisitos maldefinidos, de dificuldades em estimar o prazo e os recursos necessários para o desenvolvimento de software, da dependência de habilidades individuais e de mudanças nos requisitos, em razão de modificações nas necessidades do cliente. O gerente de projeto deve prever riscos, compreender o impacto desses riscos no projeto, no produto e nos negócios e tomar providências para evitar esses riscos. Planos de contingência podem ser traçados

para que, se os riscos realmente ocorrerem, seja possível uma ação imediata que vise à recuperação.

Os tipos de risco que podem afetar um projeto dependem do projeto e do ambiente organizacional em que o software está sendo desenvolvido. Contudo, muitos riscos são considerados universais e alguns deles são descritos na Figura 4.10.

O processo de gerenciamento de riscos está ilustrado na Figura 4.11 e envolve vários estágios:

1. *Identificação de riscos* São identificados os possíveis riscos de projeto, produto e negócios.
2. *Análise de riscos* São avaliadas as possibilidades e as conseqüências da ocorrência desses riscos.
3. *Planejamento de riscos* São traçados planos para enfrentar os riscos, seja evitando-os seja minimizando seus efeitos sobre o projeto.
4. *Monitoramento de riscos* O risco é constantemente avaliado e os planos para a diminuição de riscos são revisados, à medida que mais informações sobre eles se tornam disponíveis.

O processo de gerenciamento de riscos, como todos os outros planejamentos de projeto, é um processo iterativo, que continua ao longo do projeto. Uma vez traçado um conjunto inicial de planos, a situação é monitorada. À medida que mais informações sobre os riscos se tornam disponíveis, eles têm de ser reavaliados e novas prioridades devem ser estabelecidas. Os planos para evitar riscos e os planos de contingência podem ser modificados com o surgimento de novas informações sobre os riscos.

Os resultados do processo de gerenciamento de riscos devem ser documentados em um plano de gerenciamento de riscos. Essa fase deve incluir uma discussão sobre os riscos apre-

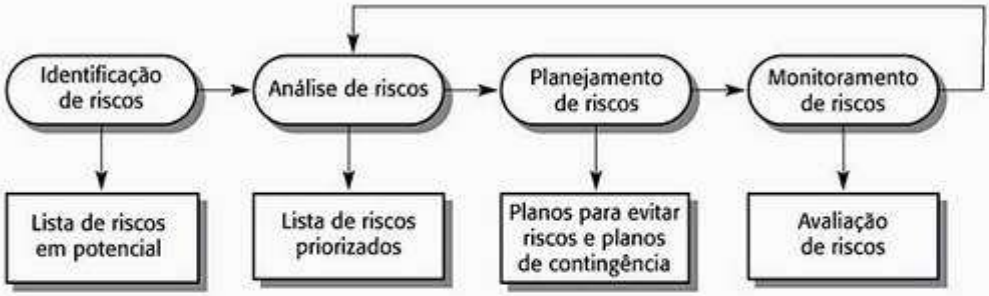
Figura 4.10 ▶

Possíveis riscos de software.

Risco	Tipo de risco	Descrição
Rotatividade de pessoal	Projeto	O pessoal experiente deixará o projeto antes do término.
Mudança de gerenciamento	Projeto	Haverá uma mudança no gerenciamento organizacional, com a definição de prioridades diferentes.
Indisponibilidade de hardware	Projeto	O hardware essencial ao projeto não será entregue dentro do prazo.
Alteração nos requisitos	Projeto e produto	Haverá maior número de mudanças nos requisitos do que o previsto.
Atrasos na especificação	Projeto e produto	As especificações de interfaces essenciais não estavam disponíveis dentro dos prazos.
Tamanho subestimado	Projeto e produto	O tamanho do sistema foi subestimado.
Baixo desempenho de ferramentas CASE	Produto	As ferramentas CASE que apóiam o projeto não apresentam desempenho conforme o previsto.
Mudanças na tecnologia	Negócios	A tecnologia básica sobre a qual o sistema está sendo construído foi superada por nova tecnologia.
Concorrência com o produto	Negócios	Um produto concorrente foi lançado no mercado, antes que o sistema fosse concluído.

Figura 4.11

O processo de gerenciamento de riscos.



sentados pelo projeto, uma análise desses riscos e os planos que são necessários para gerenciá-los. Quando for apropriado, pode também incluir alguns resultados do gerenciamento de riscos, isto é, planos específicos de contingência a serem ativados caso o risco venha a ocorrer.

4.4.1 Identificação de riscos

A identificação de riscos é o primeiro estágio do gerenciamento de riscos; ela se preocupa em descobrir os possíveis riscos existentes no projeto. Em princípio, esses riscos não devem ser avaliados ou priorizados nesse estágio, embora, na prática, os riscos com conseqüências muito pequenas ou com muito pouca probabilidade não sejam normalmente levados em consideração.

A identificação de riscos pode ser realizada como um processo em equipe, utilizando-se uma abordagem de brainstorming (intensiva troca de idéias), ou pode simplesmente ter como base a experiência de um gerente. Para ajudar nesse processo, uma lista de possíveis tipos de riscos pode ser utilizada. Dentre esses tipos destacam-se:

1. *Riscos quanto à tecnologia* São os riscos que se originam de tecnologias de software ou de hardware, que são utilizadas como parte do sistema em desenvolvimento.
2. *Riscos quanto ao pessoal* São os riscos associados às pessoas da equipe de desenvolvimento.
3. *Riscos organizacionais* São os riscos que derivam do ambiente organizacional em que o software está sendo desenvolvido.
4. *Riscos quanto às ferramentas* São os riscos que derivam de ferramentas CASE e de outros tipos de software de apoio, utilizados para desenvolver o sistema.
5. *Riscos quanto aos requisitos* São os riscos que derivam de modificações nos requisitos do cliente e do processo de gerenciamento da modificação de requisitos.
6. *Riscos quanto à estimativa* São os riscos que derivam das estimativas feitas pelo gerenciamento sobre as características do sistema e os recursos necessários para construí-lo.

A Figura 4.12 apresenta alguns exemplos de possíveis riscos em cada uma dessas categorias. O resultado desse processo deve ser uma longa lista dos riscos que podem ocorrer e que podem afetar o produto, o processo e os negócios.

4.4.2 Análise de riscos

Durante o processo de análise de riscos, cada risco identificado é considerado individualmente e é feito um julgamento sobre a probabilidade e a seriedade desse risco. Não existe uma maneira fácil de fazer isso – tudo depende do julgamento e da experiência do gerente

de projeto. Em geral, não é preciso uma avaliação numérica exata, mas essa análise deve ter como base uma análise com utilização de intervalos:

1. A probabilidade do risco pode ser determinada como muito baixa (menor do que 10 por cento), baixa (10–25 por cento), moderada (25–50 por cento), alta (50–75 por cento) ou muito alta (maior do que 75 por cento).
2. Os efeitos do risco podem ser determinados como catastróficos, sérios, toleráveis ou insignificantes.

Os resultados desse processo de análise devem, então, ser apresentados na tabela ordenada de acordo com a seriedade do risco. A Figura 4.13 ilustra esse aspecto para os riscos identificados na Figura 4.12. Obviamente, nesse caso, a avaliação de probabilidade e seriedade é arbitrária. Na prática, você precisa de informações detalhadas sobre o projeto, o processo, a equipe de desenvolvimento e a organização, para poder fazer essa avaliação.

Naturalmente, tanto a probabilidade quanto a avaliação dos efeitos de um risco podem se modificar, à medida que mais informações sobre o risco se tornam disponíveis e os planos de gerenciamento de risco são implementados. Portanto, essa tabela deve ser atualizada durante cada iteração do processo de risco.

Uma vez que os riscos tenham sido analisados e priorizados, deve ser feito um julgamento sobre quais são os riscos mais importantes a serem considerados durante o projeto. Esse julgamento deve depender de uma combinação da probabilidade de o risco surgir e dos efeitos daquele risco. Em geral, todos os riscos catastróficos devem sempre ser considerados, como também todos os riscos sérios que tenham mais do que uma probabilidade moderada de vir a ocorrer.

Boehm (1988) recomenda identificar e monitorar os 'dez maiores riscos', mas esse número é um tanto arbitrário. O número certo de riscos a serem monitorados deve depender do projeto; e eles podem ser um total de cinco ou quinze. Contudo, o número de riscos escolhido para monitorar deve ser gerenciável. Um número muito grande de riscos simplesmente pode-

Figura 4.12 ▶

Riscos e tipos de risco.

Tipos de risco	Riscos possíveis
Tecnologia	O banco de dados utilizado no sistema não pode processar tantas transações por segundo, como esperado. Componentes do software que deviam ser reutilizados contêm defeitos que limitam sua funcionalidade.
Pessoal	É impossível recrutar pessoal com a habilidade requerida. Pessoas importantes estão doentes e não disponíveis em períodos cruciais. O treinamento necessário para o pessoal não está disponível.
Organizacional	A organização está estruturada de maneira que diferentes gerências são responsáveis pelo projeto. Problemas financeiros organizacionais forçam reduções no orçamento do projeto.
Ferramentas	O código gerado pelas ferramentas CASE é ineficiente. As ferramentas CASE não podem ser integradas.
Requisitos	São propostas mudanças nos requisitos, que exigem significativo retrabalho. Os clientes não compreendem o impacto das mudanças nos requisitos.
Estimativa	O tempo requerido para desenvolver o software é subestimado. A taxa de solução de defeitos é subestimada. O tamanho do software é subestimado.

Figura 4.13 ▶
Análise de riscos.

Risco	Probabilidade	Efeitos
Problemas financeiros organizacionais forçam reduções no orçamento do projeto.	Baixa	Catastróficos
É impossível recrutar pessoal com as habilidades requeridas para o projeto.	Alta	Catastróficos
Pessoas-chave estão doentes em períodos cruciais do projeto.	Moderada	Sérios
Componentes de software que deviam ser reutilizados contêm defeitos que limitam sua funcionalidade.	Moderada	Sérios
São propostas mudanças nos requisitos, que exigem significativo retrabalho.	Moderada	Sérios
A organização está estruturada de maneira que diferentes gerências são responsáveis pelo projeto.	Alta	Sérios
O banco de dados utilizado no sistema não pode processar tantas transações por segundo, como esperado.	Moderada	Sérios
O tempo requerido para desenvolver o software é subestimado.	Alta	Sérios
As ferramentas CASE não podem ser integradas.	Alta	Toleráveis
Os clientes não compreendem o impacto das mudanças nos requisitos.	Moderada	Toleráveis
O treinamento necessário para o pessoal não está disponível.	Moderada	Toleráveis
A taxa de solução de defeitos é subestimada.	Moderada	Toleráveis
O tamanho do software é subestimado.	Alta	Toleráveis
O código gerado pelas ferramentas CASE é ineficiente.	Moderada	Insignificantes

ria exigir que muitas informações fossem coletadas. A partir dos riscos identificados na Figura 4.13, é apropriado considerar todos os oito riscos que podem ter conseqüências catastróficas ou sérias.

4.4.3 Planejamento de riscos

O processo de planejamento de riscos considera cada um dos riscos mais importantes que foram identificados e define estratégias para gerenciá-lo. Mais uma vez, não há nenhum processo simples que possa ser seguido para estabelecer planos de gerenciamento de riscos. Isso depende do julgamento e da experiência do gerente de projeto. A Figura 4.14 mostra possíveis estratégias que foram identificadas para os principais riscos, mostrados na Figura 4.13.

As estratégias classificam-se em três categorias:

1. *Estratégias preventivas* Seguir essas estratégias significa que a probabilidade de o risco surgir será reduzida. Um exemplo de estratégia preventiva é a utilizada para lidar com componentes defeituosos, mostrada na Figura 4.14.
2. *Estratégias de minimização* Seguir essas estratégias significa que o impacto do risco será reduzido. Um exemplo de estratégia de minimização de riscos é a utilizada quando alguém está doente, mostrada na Figura 4.14.

Figura 4.14 ▶

Estratégias de gerenciamento de riscos.

Risco	Estratégia
Problemas financeiros organizacionais	Prepare um documento informativo para a alta gerência, mostrando como o projeto presta uma contribuição muito importante para os objetivos da empresa.
Problemas de recrutamento	Alerte o cliente sobre as dificuldades em potencial e a possibilidade de atrasos; investigue a compra de componentes.
Doença de pessoas da equipe	Reorganize a equipe de maneira que haja mais sobreposição de trabalho e, portanto, as pessoas compreendam as tarefas umas das outras.
Componentes defeituosos	Substitua componentes potencialmente defeituosos por componentes comprados e que tenham confiabilidade reconhecida.
Alterações nos requisitos	Extraia informações que podem ser rastreadas, para avaliar o impacto das mudanças nos requisitos, maximize a inclusão de informações no projeto.
Reestruturação organizacional	Prepare um documento informativo para a alta gerência, mostrando como o projeto presta uma contribuição muito importante para os objetivos da empresa.
Desempenho do banco de dados	Investigue a possibilidade de comprar um banco de dados com maior desempenho.
Prazo de desenvolvimento subestimado	Investigue a compra de componentes e verifique o uso de um gerador de programas.

3. *Planos de contingência* Seguir essas estratégias significa que, se o pior acontecer, você está preparado e tem uma estratégia pronta para lidar com o caso. Um exemplo de estratégia de contingência é a estratégia aplicada a problemas financeiros organizacionais, mostrada da Figura 4.14.

4.4.4 Monitoramento de riscos

O monitoramento de riscos envolve avaliar regularmente cada um dos riscos individuais, a fim de decidir se esse risco está se tornando mais ou menos provável e se seus efeitos decorrentes se modificaram. É claro que, em geral, isso não pode ser observado de modo direto; portanto, devem ser examinados outros fatores, para serem levantados indícios sobre a probabilidade de risco e seus efeitos. Esses fatores são, obviamente, dependentes dos tipos de risco. A Figura 4.15 apresenta alguns exemplos de fatores que podem ser úteis para avaliar esses tipos de risco.

Figura 4.15 ▶

Fatores de risco.

Tipo de risco	Indicadores em potencial
Tecnologia	Atraso na entrega de hardware ou software de apoio, muitos problemas de tecnologia são relatados.
Pessoal	Pessoal pouco motivado, relacionamento insatisfatório entre os membros da equipe, disponibilidade de trabalho.
Organizacional	Fofocas na empresa, falta de iniciativa por parte da alta gerência.
Ferramentas	Relutância de membros da equipe em utilizar ferramentas, reclamações sobre ferramentas CASE, solicitações de estações de trabalho com maior capacidade.
Requisitos	Muitos pedidos de modificações nos requisitos, reclamações do cliente.
Estimativa	Falha no cumprimento do programa estabelecido, falha em eliminar defeitos registrados.

O monitoramento de riscos deve ser um processo contínuo e, a cada análise de progresso feita pela gerência, cada um dos riscos principais deve ser considerado separadamente e discutido em uma reunião.

PONTOS-CHAVE

- Um bom gerenciamento de projeto de software é essencial para que os projetos de engenharia de software sejam desenvolvidos dentro do prazo e do orçamento.
- O gerenciamento de software é diferente dos outros gerenciamentos de engenharia. O software é intangível. Os projetos podem ser novos ou inovadores, de modo que não haja nenhuma experiência prévia para orientar seu gerenciamento. Os processos de software não são bem compreendidos.
- Os gerentes de software têm diversos papéis. Suas atividades mais significativas são: planejamento de projeto, estimativa e programação. O planejamento e a estimativa são processos iterativos, que continuam ao longo do projeto. À medida que mais informações se tornam disponíveis, os planos e as programações precisam ser revistos.
- Um marco de projeto é o resultado previsto de uma atividade em que algum relatório formal de progresso deve ser apresentado à gerência. Os marcos devem ocorrer regularmente ao longo de um projeto de software. Um produto a ser entregue é um marco que é disponibilizado ao cliente do projeto.
- A programação de projeto envolve a criação de várias representações gráficas de parte do plano de projeto. Isso inclui diagramas de atividades, que mostram o inter-relacionamento de atividades de projeto, e diagramas de barras, que mostram a duração das atividades.
- Os principais riscos de projeto devem ser identificados e avaliados, a fim de estabelecer sua probabilidade e suas conseqüências para o projeto. Para os riscos prováveis e potencialmente sérios, devem ser feitos planos preventivos, de gerenciamento de riscos, ou planos para administrar esse risco quando ele surgir. Os riscos devem ser explicitamente discutidos em cada reunião sobre o progresso do projeto.

LEITURAS SUGERIDAS

Managing software quality and business risk. O Capítulo 3 desse livro é simplesmente a melhor discussão sobre riscos que já vi. O livro é orientado à questão dos riscos, e penso que ele seja, provavelmente, o melhor livro disponível sobre esse tópico no momento em que escrevo este livro. (M. Ould, 1999, John Wiley and Sons.)

The mythical man month. Os problemas do gerenciamento de software permaneceram imutáveis desde a década de 60, e esse é um dos melhores livros sobre o assunto. Trata-se de um relato interessante e de fácil leitura sobre o gerenciamento de um dos primeiros projetos de software de grande porte, o sistema operacional IBM OS/360. A segunda edição traz outros artigos clássicos, de autoria de Brooks. (F.P. Brooks, 1975, Addison Wesley.)

Assessment and control of software risks. Atualmente, existem vários livros disponíveis sobre o gerenciamento de riscos. Esse apresenta uma discussão particularmente abrangente dos tipos de riscos, por que eles surgem e como evitá-los. (C. Jones, 1994, Prentice Hall.)

Principles of software engineering management. Esse é um relato muito peculiar sobre o gerenciamento de software, mas ele traz boas recomendações. Ele foi escrito de um modo que proporciona fácil leitura. (T. Gilb, 1988, Addison Wesley.)

Veja a Parte 6 deste livro, para verificar outras referências de leitura sobre o gerenciamento.

EXERCÍCIOS

4.1 Explique por que a intangibilidade dos sistemas de software gera problemas especiais para o gerenciamento de projetos de software.

4.2 Explique por que os melhores programadores nem sempre se tornam os melhores gerentes de software. Você pode considerar útil basear sua resposta na lista de atividades de gerenciamento apresentada na Seção 4.1.

4.3 Explique por que o processo de planejamento de projetos é iterativo e por que um plano deve ser revisto continuamente, durante um projeto de software.

4.4 Explique brevemente o propósito de cada uma das seções em um plano de projeto de software.

4.5 Qual é a principal distinção entre um marco e um produto a ser entregue?

4.6 A Figura 4.16 discrimina uma série de atividades, durações e dependências. Faça um diagrama de atividades mostrando a programação de projeto.

4.7 A Figura 4.5 fornece a duração de tarefas para atividades de projeto de software. Suponha que um problema sério e não previsto ocorra e, em vez de levar dez dias para ser realizada, a tarefa T5 leve 45 dias. Revise a rede de atividades de acordo com essa ocorrência, evidenciando o novo caminho principal. Desenhe novos diagramas de barras mostrando como o projeto pode ser reorganizado.

4.8 Utilizando exemplos relatados de problemas com projetos, na literatura disponível, faça uma lista das dificuldades de gerenciamento que ocorreram nesses projetos de programação que fracassaram. (Comece com o livro de Brooks, recomendado em "Leituras sugeridas".)

4.9 Além dos riscos mostrados na Figura 4.12, identifique seis outros riscos que, provavelmente, podem surgir em projetos de software.

Figura 4.16 ▶

Durações e dependências de tarefas.

Tarefa	Duração (dias)	Dependências
T1	10	
T2	15	T1
T3	10	T1, T2
T4	20	
T5	10	
T6	15	T3, T4
T7	20	T3
T8	35	T7
T9	15	T6
T10	5	T5, T9
T11	10	T9
T12	20	T10
T13	35	T3, T4
T14	10	T8, T9
T15	20	T12, T14
T16	10	T15

4.10 Seu gerente pede que seja entregue um software dentro de uma programação que você sabe que somente será cumprida caso sua equipe trabalhe durante um período de horas extras, sem remuneração. Todos os membros da equipe têm filhos pequenos. Discuta se você deve aceitar essa solicitação de seu gerente e se deve persuadir a equipe a se dedicar desse modo à organização, em vez de se dedicar a seus familiares. Que fatores podem ser importantes em sua decisão?

4.11 Como programador, você é promovido ao cargo de gerente de projeto, mas você sente que pode prestar melhor contribuição em uma posição técnica do que em um cargo administrativo. Discuta se você deve aceitar essa promoção.

PARTE DOIS

Requisitos

PÁGINA EM BRANCO

5 Requisitos de software

Objetivos

O objetivo deste capítulo é apresentar uma introdução sobre os requisitos de sistema de software e explicar diferentes maneiras de expressar esses requisitos. Depois de ler este capítulo, você poderá:

- compreender os conceitos dos requisitos do usuário e dos requisitos de sistema e por que esses requisitos podem ser expressos utilizando-se diferentes notações;
- compreender as diferenças entre os requisitos funcionais e não funcionais;
- compreender duas técnicas para descrever os requisitos de sistema, como a descrição em linguagem natural estruturada e a descrição com base em linguagem de programação, e
- compreender como os requisitos podem ser organizados em um documento de requisitos de software.

Sumário

5.1 Requisitos funcionais e não funcionais
5.2 Requisitos do usuário
5.3 Requisitos de sistema
5.4 O documento de requisitos de software

Os problemas que os engenheiros de software têm para solucionar são, muitas vezes, imensamente complexos. Compreender a natureza dos problemas pode ser muito difícil, especialmente se o sistema for novo. Conseqüentemente, é difícil estabelecer com exatidão o que o sistema deve fazer. As descrições das funções e das restrições são os *requisitos* para o sistema; e o processo de descobrir, analisar, documentar e verificar essas funções e restrições é chamado de *engenharia de requisitos*. Neste capítulo, concentro-me nos requisitos propriamente ditos e em como descrevê-los. Fiz uma introdução ao processo de engenharia de requisitos no Capítulo 3 e discutirei o assunto com mais detalhes no Capítulo 6.

O termo *requisito* não é utilizado pela indústria de software de modo consistente. Em alguns casos, um requisito é visto como uma declaração abstrata, de alto nível, de uma função que o sistema deve fornecer ou de uma restrição do sistema. No outro extremo, ele é uma definição detalhada, matematicamente formal, de uma função do sistema. Davis (1993) explica por que essas diferenças existem:

> Se uma empresa deseja estabelecer um contrato para o desenvolvimento de um grande projeto de software, ela tem de definir suas necessidades de maneira suficientemente abstrata para que uma solução não seja predefinida. Os requisitos devem ser redigidos de modo que os diversos fornecedores possam apresentar propostas, oferecendo, talvez, diferentes maneiras de atender às necessidades organizacionais do cliente. Uma vez estabelecido um contrato, o fornecedor precisa preparar uma definição de sistema para o cliente, com mais detalhes, de modo que o cliente compreenda e possa validar o que o software fará. Esses dois documentos podem ser chamados de *documentos de requisitos* do sistema.

Alguns dos problemas que surgem durante o processo de engenharia de requisitos são resultantes da falta de uma nítida separação entre esses diferentes níveis de descrição. Faço essa distinção utilizando o termo *requisitos do usuário* para designar os requisitos abstratos de alto nível, e o termo *requisitos de sistema*, para indicar a descrição detalhada do que o sistema deverá fazer. Além desses dois níveis de detalhes, uma descrição mais detalhada (uma especificação de projeto de software) pode ser produzida para associar a engenharia de requisitos e as atividades de projeto. Os requisitos do usuário, os requisitos de sistema e a especificação de projeto de software podem ser definidos como se segue:

1. *Requisitos do usuário* são declarações, em linguagem natural e também em diagramas, sobre as funções que o sistema deve fornecer e as restrições sob as quais deve operar.
2. *Requisitos de sistema* estabelecem detalhadamente as funções e as restrições de sistema. O documento de requisitos de sistema, algumas vezes chamado de especificação funcional, deve ser preciso. Ele pode servir como um contrato entre o comprador do sistema e o desenvolvedor do software.
3. *Especificação de projeto de software* é uma descrição abstrata do projeto de software; que é uma base para o projeto e a implementação mais detalhados. Essa especificação acrescenta mais detalhes à especificação de requisitos do sistema.

Diferentes níveis de especificação de sistema são úteis porque comunicam informações sobre o sistema para diferentes tipos de leitores. A Figura 5.1 ilustra a distinção entre os requisitos de usuários e os de sistemas. Ela mostra como um requisito de usuário pode ser expandido em diversos requisitos de sistema.

Os requisitos do usuário devem ser escritos para gerentes do cliente e dos fornecedores, que não tenham um conhecimento técnico detalhado do sistema (Figura 5.2). A especificação de requisitos de sistema deve ter como alvo os profissionais técnicos de nível sênior e os gerentes de projeto. Novamente, ela será utilizada pelo gerente do cliente e do fornecedor. Os usuários finais de sistemas podem ler ambos os documentos. Por fim, a especificação de projeto de software é um documento orientado à implementação. Ele deve ser escrito para os engenheiros de software que desenvolverão o sistema.

Figura 5.1

Requisitos do usuário e do sistema.

Definição dos requisitos do usuário

1. O software deve oferecer um meio de representar e acessar arquivos externos criados por outras ferramentas.

Especificação dos requisitos de sistema

1.1 O usuário deve dispor de recursos para definir o tipo dos arquivos externos.
1.2 Cada tipo de arquivo externo pode ter uma ferramenta associada que pode ser aplicada a ele.
1.3 Cada tipo de arquivo externo pode ser representado como um ícone específico na tela do usuário.
1.4 Devem ser fornecidos recursos para o ícone que representa um arquivo externo, a ser definido pelo usuário.
1.5 Quando um usuário seleciona um ícone que representa um arquivo externo, o efeito dessa seleção é aplicar a ferramenta associada com o tipo de arquivo externo ao arquivo representado pelo ícone selecionado

Figura 5.2

Leitores de diferentes tipos de especificação.

5.1 Requisitos funcionais e não funcionais

Os requisitos de sistema de software são, freqüentemente, classificados como funcionais ou não funcionais ou como requisitos de domínio:

1. *Requisitos funcionais* São declarações de funções que o sistema deve fornecer, como o sistema deve reagir a entradas específicas e como deve se comportar em determinadas situações. Em alguns casos, os requisitos funcionais podem também explicitamente declarar o que o sistema não deve fazer.

2. *Requisitos não funcionais* São restrições sobre os serviços ou as funções oferecidos pelo sistema. Entre eles destacam-se restrições de tempo, restrições sobre o processo de desenvolvimento, padrões, entre outros.

3. *Requisitos de domínio* São requisitos que se originam do domínio de aplicação do sistema e que refletem características desse domínio. Podem ser requisitos funcionais ou não funcionais.

Na realidade, a distinção entre esses diferentes tipos de requisitos não é tão clara como sugerem essas definições simples. Um requisito de usuário relacionado à proteção, digamos, parece ser um requisito não funcional. Contudo, quando desenvolvido com mais detalhes, pode levar a outros requisitos que são claramente funcionais, como a necessidade de incluir recursos de autorização de usuários no sistema. Portanto, embora seja útil classificar os requisitos dessa maneira quando os discutimos, devemos lembrar que essa é, na verdade, uma distinção artificial.

5.1.1 Requisitos funcionais

Os requisitos funcionais para um sistema descrevem a funcionalidade ou os serviços que se espera que o sistema forneça. Eles dependem do tipo de software que está sendo desenvolvido, dos usuários de software que se espera verificar e do tipo de sistema que está sendo desenvolvido. Quando expressos como requisitos de usuário, eles são normalmente descritos de um modo bastante geral, mas os requisitos funcionais de sistema descrevem a função de sistema detalhadamente, suas entradas e saídas, exceções etc.

Os requisitos funcionais de um sistema de software podem ser expressos de diversas maneiras. Aqui estão vários requisitos funcionais do sistema de biblioteca de universidade (Kotonya e Sommerville, 1998) para que os estudantes e a faculdade possam pedir livros e documentos de outras bibliotecas:

1. O usuário deverá ser capaz de buscar todo o conjunto inicial de banco de dados ou selecionar um subconjunto a partir dele.
2. O sistema fornecerá telas apropriadas para o usuário ler documentos no repositório de documentos.
3. Cada pedido será alocado a um único identificador (ORDER_ID), que o usuário poderá copiar para a área de armazenagem permanente da conta.

Esses requisitos funcionais de usuário definem recursos específicos que devem ser fornecidos pelo sistema. Eles foram observados no documento de requisitos de usuário do sistema e ilustram que os requisitos funcionais podem ser escritos em diferentes níveis de detalhes (compare os requisitos 1 e 3).

Muitos problemas de engenharia de software se originam da imprecisão na especificação de requisitos. É natural para um desenvolvedor de sistemas interpretar um requisito ambíguo para simplificar sua implementação. Muitas vezes, contudo, isso não é o que o cliente quer. Novos requisitos devem ser estabelecidos e é necessário realizar mudanças no sistema. Naturalmente, isso atrasa a entrega do sistema e aumenta os custos.

Considere o requisito do segundo exemplo para o sistema de biblioteca, na lista de requisitos mostrada anteriormente, que se refere a 'telas apropriadas' fornecidas pelo sistema. O sistema de biblioteca pode fornecer documentos em uma série de formatos, e a intenção desse requisito é que as telas para todos esses formatos estejam disponíveis. Contudo, ele está redigido de maneira ambígua, uma vez que não torna claro que as telas para cada formato de documento devem ser disponibilizadas. Um desenvolvedor que está sob pressão quanto à programação pode simplesmente fornecer uma tela de texto e argumentar que os requisitos foram cumpridos.

Em princípio, a especificação de requisitos funcionais de um sistema deve ser completa e consistente. A completeza significa que todas as funções requeridas pelo usuário devem estar definidas. A consistência significa que os requisitos não devem ter definições contraditórias. Na prática, para sistemas complexos e grandes, é quase impossível atingir a consistência e a completeza dos requisitos. A razão disso é, em parte, por causa da complexidade inerente ao sistema e, em parte, porque diferentes pontos de vista (veja o Capítulo 6) apresentam necessidades inconsistentes. Essas inconsistências podem não ser óbvias quando os requisitos são especificados primeiramente. Os problemas somente emergem depois de

uma análise mais profunda. À medida que os problemas são descobertos durante as revisões ou em fases posteriores do ciclo de vida, os problemas no documento de requisitos devem ser corrigidos.

5.1.2 Requisitos não funcionais

Os requisitos não funcionais, como o nome sugere, são aqueles que não dizem respeito diretamente às funções específicas fornecidas pelo sistema. Eles podem estar relacionados a propriedades de sistema emergentes, como confiabilidade, tempo de resposta e espaço em disco. Como alternativa, eles podem definir restrições para o sistema, como a capacidade dos dispositivos de E/S (entrada/saída) e as representações de dados utilizadas nas interfaces de sistema.

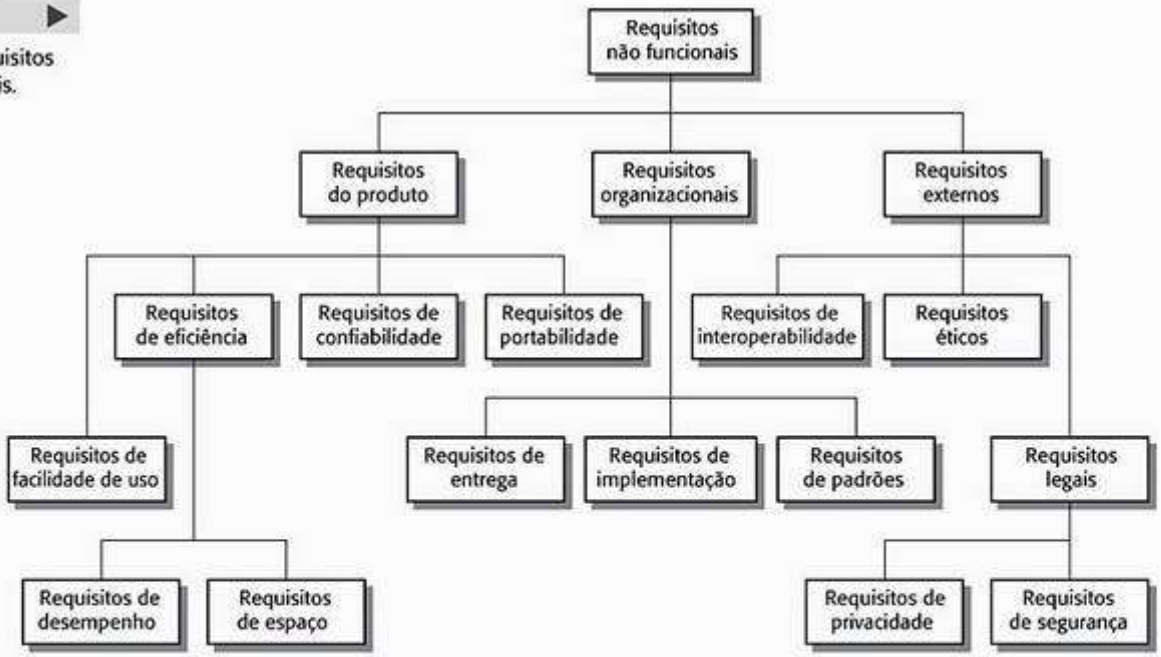
Muitos requisitos não funcionais dizem respeito ao sistema como um todo, e não a características individuais do sistema. Isso significa que eles são, freqüentemente, mais importantes do que os requisitos funcionais individuais. Enquanto a falha em cumprir com um requisito funcional individual pode degradar o sistema, a falha em cumprir um requisito não funcional de sistema pode tornar todo o sistema inútil. Por exemplo, se um sistema de aviação não atender a seus requisitos de confiabilidade, ele não será atestado como seguro para operação; se um sistema de controle em tempo real falhar em cumprir com seus requisitos de desempenho, as funções de controle não operarão corretamente.

Contudo, os requisitos não funcionais nem sempre dizem respeito ao sistema de software a ser desenvolvido. Alguns requisitos não funcionais podem restringir o processo que pode ser utilizado para desenvolver o sistema. São exemplos de requisitos de processo uma especificação dos padrões de qualidade, que deve ser utilizada no processo, uma especificação de que o projeto deve ser produzido com um conjunto especificado de ferramentas CASE e uma descrição de processo a ser seguido.

Os requisitos não funcionais surgem conforme a necessidade dos usuários, em razão de restrições de orçamento, de políticas organizacionais, pela necessidade de interoperabilidade com outros sistemas de software ou hardware ou devido a fatores externos, como por exemplo regulamentos de segurança e legislação sobre privacidade. A Figura 5.3 é uma classificação dos diferentes tipos de requisitos não funcionais que podem surgir.

Figura 5.3 ▶

Tipos de requisitos não funcionais.



Classifiquei os diferentes tipos de requisitos não funcionais, mostrados na Figura 5.3, de acordo com sua procedência:

1. *Requisitos de produtos* São os requisitos que especificam o comportamento do produto. Entre os exemplos estão os requisitos de desempenho sobre com que rapidez o sistema deve operar e quanta memória ele requer, os requisitos de confiabilidade, que estabelecem a taxa aceitável de falhas, os requisitos de portabilidade e os requisitos de facilidade de uso.
2. *Requisitos organizacionais* São procedentes de políticas e procedimentos nas organizações do cliente e do desenvolvedor. Entre os exemplos estão os padrões de processo que devem ser utilizados, os requisitos de implementação, como a linguagem de programação ou o método de projeto utilizado, e os requisitos de fornecimento, que especificam quando o produto e seus documentos devem ser entregues.
3. *Requisitos externos* Esse amplo tópico abrange todos os requisitos procedentes de fatores externos ao sistema e a seu processo de desenvolvimento. Dentre eles destacam-se os requisitos de interoperabilidade, que definem como o sistema interage com sistemas em outras organizações, os requisitos legais, que devem ser seguidos para assegurar que o sistema opera de acordo com a lei, e os requisitos éticos. Os requisitos éticos são definidos em um sistema para garantir que este será aceitável para seus usuários e o público em geral.

A Figura 5.4 mostra exemplos de requisitos de produto, organizacionais e externos. O requisito de produto diz respeito a um ambiente de apoio à programação para a linguagem Ada (um APSE). Ele restringe a liberdade dos projetistas do APSE na sua escolha de símbolos utilizados na interface com o usuário, do APSE; não diz nada sobre a funcionalidade do APSE, e claramente identifica uma restrição do sistema, em vez de uma função. O requisito organizacional especifica que o sistema deve ser desenvolvido de acordo com um processo-padrão da empresa, definido como XYZCo-SP-STAN-95. O requisito externo é derivado da necessidade de o sistema atender à legislação de privacidade. Ele especifica que os operadores de sistema não devem ter acesso a qualquer dado de que não necessitem.

Um problema comum com os requisitos não funcionais é que eles são, às vezes, de difícil verificação; eles podem ser escritos para refletir os objetivos gerais do cliente, como a facilidade de uso, a habilidade de o sistema se recuperar de uma falha ou a rapidez de resposta ao usuário. Esses requisitos causam problemas para os desenvolvedores de sistema, à medida que eles deixam o enfoque aberto à interpretação e à conseqüente discussão, quando o sistema é entregue. Como ilustração desse problema, considere a Figura 5.5. Ela mostra uma meta do sistema, relacionada com sua facilidade de uso, e como isso pode ser expresso de maneira verificável, como um requisito não funcional. Esse requisito não funcional pode ser verificado por testes e, assim, é possível checar se o sistema atendeu ao objetivo do cliente.

Como ideal, os requisitos não funcionais devem ser expressos quantitativamente, utilizando-se métricas que possam ser objetivamente testadas. A Figura 5.6 mostra uma série de possíveis métricas, que podem ser utilizadas para especificar propriedades de sistema não funcionais. As medições podem ser feitas durante o teste de sistema, para determinar se o sistema cumpre com esses requisitos ou não.

Figura 5.4 ▶

Exemplos de requisitos não funcionais.

Requisito de produto
4.C.8 Deve ser possível que toda a comunicação necessária entre o APSE e o usuário seja expressa no conjunto-padrão de caracteres Ada.

Requisito organizacional
9.3.2 O processo de desenvolvimento de sistema e os documentos a serem entregues deverão estar de acordo com o processo e os produtos a serem entregues, definidos em XYZCo-SP-STAN-95.

Requisito externo
7.6.5 O sistema não deverá revelar aos operadores nenhuma informação pessoal sobre os clientes, além de seus nomes e o número de referência.

Figura 5.5

Metas do sistema e requisitos verificáveis.

Uma meta do sistema
O sistema deve ser fácil de utilizar por controladores experientes e deve ser organizado de modo que os erros dos usuários sejam minimizados.

Um requisito não funcional verificável
Controladores experientes devem ser capazes de utilizar todas as funções do sistema depois de um total de duas horas de treinamento. Depois desse treinamento, o número médio de erros feitos por usuários experientes não deve exceder a dois por dia.

Figura 5.6

Métricas para especificar requisitos não funcionais

Propriedade	Métrica
Velocidade	Transações processadas/segundo Tempo de resposta ao usuário/evento Tempo de *refresh* da tela
Tamanho	K bytes Número de chips de RAM
Facilidade de uso	Tempo de treinamento Número de frames de ajuda
Confiabilidade	Tempo médio para falhar Probabilidade de indisponibilidade Taxa de ocorrência de falhas Disponibilidade
Robustez	Tempo de reinício depois de uma falha Porcentagem de eventos que causam falhas Probabilidade de que dados sejam corrompidos por falhas
Portabilidade	Porcentagem de declarações dependentes de sistemas-alvo Número de sistemas-alvo

Na prática, em geral, a especificação quantitativa de requisitos é difícil. Os clientes podem não ser capazes de traduzir suas metas em requisitos quantitativos; para algumas metas, como as metas de facilidade de manutenção, não há métricas que possam ser utilizadas; o custo de verificar objetivamente requisitos quantitativos não funcionais pode ser muito alto. Portanto, os documentos de requisitos, muitas vezes, incluem declarações de metas misturadas com requisitos. Essas metas podem ser úteis para os desenvolvedores porque fornecem algumas pistas sobre as prioridades do cliente. Contudo, os clientes devem saber que essas metas estão sujeitas a más interpretações e não podem ser objetivamente verificadas.

Os requisitos não funcionais freqüentemente entram em conflito e interagem com outros requisitos funcionais do sistema. Por exemplo, pode ser requerido que o máximo de espaço ocupado por um sistema seja de 4 Mbytes, porque todo o sistema deve ser adequado à memória do tipo *read-only* (memória 'somente para leitura') e instalado em uma espaçonave. Um requisito adicional pode ser que o sistema seja escrito utilizando-se Ada, uma linguagem de programação projetada para o desenvolvimento de software crítico e em tempo real. Contudo, pode não ser possível compilar um programa Ada com a funcionalidade exigida, com menos de 4 Mbytes. É preciso encontrar certa 'compensação' entre esses requisitos. Uma linguagem alternativa de desenvolvimento pode ser utilizada, ou também é possível acrescentar memória ao sistema.

Em princípio, os requisitos funcionais e não funcionais devem ser diferenciados em um documento de requisitos; na prática, isso é difícil. Se os requisitos não funcionais forem definidos separadamente dos requisitos funcionais, algumas vezes será difícil ver o relacionamento entre eles. Se eles forem definidos com os requisitos funcionais, poderá ser difícil separar considerações funcionais e não funcionais e identificar os requisitos que correspondem ao

sistema como um todo. É preciso encontrar um equilíbrio adequado e isso depende do tipo de sistema que está sendo especificado. Contudo, requisitos claramente relacionados às propriedades emergentes do sistema devem ser explicitamente destacados. Isso pode ser feito colocando-os em uma seção separada do documento de requisitos ou distinguindo-os, de alguma maneira, dos outros requisitos de sistema.

5.1.3 Requisitos de domínio

Os requisitos de domínio são derivados do domínio da aplicação do sistema, em vez de serem obtidos a partir das necessidades específicas dos usuários do sistema. Eles podem ser novos requisitos funcionais em si, podem restringir os requisitos funcionais existentes ou estabelecer como devem ser realizados cálculos específicos. Os requisitos de domínio são importantes porque, muitas vezes, refletem fundamentos do domínio da aplicação. Se esses requisitos não forem satisfeitos, poderá ser impossível fazer o sistema operar satisfatoriamente.

Para ilustrar os requisitos de domínio, considere os seguintes requisitos do sistema de biblioteca:

1. Deve haver uma interface-padrão com o usuário para todos os bancos de dados, que terá como base o padrão Z39.50.
2. Em razão das restrições referentes a direitos autorais, alguns documentos devem ser excluídos imediatamente ao serem fornecidos. Dependendo dos requisitos dos usuários, esses documentos serão impressos localmente no servidor do sistema para serem encaminhados manualmente ao usuário ou direcionados para uma impressora de rede.

O primeiro desses requisitos é uma restrição sobre uma exigência funcional de sistema. Ele especifica que a interface com o usuário para o banco de dados deve ser implementada de acordo com um padrão específico da biblioteca. O segundo requisito foi introduzido devido às leis de direitos autorais, que se aplicam aos materiais utilizados em bibliotecas. Ele especifica que o sistema deve incluir um recurso automático de excluir impressão, para algumas classes de documentos.

Para ilustrar os requisitos de domínio que especificam como um cálculo é realizado, considere a Figura 5.7, que tem como base a especificação de um sistema automatizado de proteção de trens. Esse sistema parará automaticamente um trem se ele atravessar um sinal vermelho. Esse requisito define como a desaceleração do trem é calculada pelo sistema e utiliza terminologia específica do domínio. Para compreendê-lo, o engenheiro precisa ter algum conhecimento de operação de sistemas de estrada de ferro e das características dos trens.

Os requisitos para o sistema de trens ilustram o problema principal com os requisitos de domínio. Esses requisitos são expressos com o uso de uma linguagem que é específica do domínio da aplicação e, muitas vezes, é difícil para os engenheiros de software compreender essa linguagem. Os especialistas em um domínio podem deixar de fornecer informações em um requisito, simplesmente porque para eles essas informações são muito óbvias. Contudo, elas podem não ser óbvias para os desenvolvedores de sistemas, que, como resultado disso, talvez venham a implementar o requisito de maneira insatisfatória.

Figura 5.7 ▶

Um requisito de domínio de um sistema de proteção de trens.

A desaceleração do trem será computada como:

$$D_{trem} = D_{controle} + D_{gradiente},$$

onde $D_{gradiente}$ é 9,81 ms^2 * gradiente compensado/alfa e onde os valores de 9,81 ms^2/alfa são conhecidos para diferentes tipos de trens.

5.2 Requisitos de usuário

Os requisitos de usuários para um sistema devem descrever os requisitos funcionais e não funcionais de modo compreensível pelos usuários do sistema que não têm conhecimentos técnicos detalhados. Eles devem especificar somente o comportamento externo do sistema, evitando tanto quanto possível as características do projeto de sistema. Conseqüentemente, não devem ser definidos utilizando-se um modelo de implementação. Eles podem ser escritos com o uso de linguagem natural, formulários e diagramas intuitivos simples.

Contudo, vários problemas podem surgir quando os requisitos são escritos em linguagem natural:

1. *Falta de clareza* Às vezes, é difícil utilizar a linguagem de maneira precisa e sem ambigüidade, sem produzir um documento de difícil leitura.
2. *Confusão de requisitos* Os requisitos funcionais e os não funcionais, os objetivos do sistema e as informações sobre o projeto podem não estar claramente definidos.
3. *Fusão de requisitos* Vários requisitos diferentes podem ser expressos juntos como um único requisito.

Como ilustração de alguns desses problemas, considere um dos requisitos para um ambiente de programação em linguagem Ada, mostrado na Figura 5.8.

Esse requisito inclui informações conceituais e detalhadas. Ele expressa o conceito de que deve haver recursos de controle de configuração fornecidos como uma parte inerente do APSE. Contudo, ele também inclui o detalhe de que os recursos de controle de configuração devem permitir o acesso aos objetos em um grupo de versão, sem especificar seu nome completo. Teria sido melhor deixar esse detalhe para a especificação de requisitos de sistema.

É uma boa prática separar os requisitos de usuário dos requisitos mais detalhados do sistema, em um documento de requisitos. Do contrário, leitores que não têm conhecimentos técnicos dos requisitos de usuário podem ser sobrecarregados com detalhes que, na verdade, somente são relevantes para os técnicos. A Figura 5.9 ilustra essa confusão. Esse exemplo tem como base o documento de requisitos para uma ferramenta CASE destinada a editar modelos de projeto de software. O usuário pode especificar que uma grade deve ser exibida de modo que entidades possam ser precisamente posicionadas em um diagrama.

A primeira sentença mistura três diferentes tipos de requisitos:

1. Um requisito funcional conceitual especifica que o sistema de edição deve fornecer uma grade e apresenta uma base lógica para isso.

Figura 5.8 ▶

Um requisito sobre um banco de dados para um ambiente de programação.

4.A.5 O banco de dados deve aceitar a geração e o controle de objetos de configuração; ou seja, os objetos que são agrupamentos de outros objetos no banco de dados. Os recursos de controle de configuração devem permitir o acesso aos objetos em um grupo de versão, pelo uso de um nome incompleto.

Figura 5.9 ▶

Um requisito de usuário para uma grade de editor.

2.6 Recursos de grade Para ajudar no posicionamento de entidades em um diagrama, o usuário pode acionar uma grade em centímetros ou em polegadas, por meio de uma opção no painel de controle. Inicialmente, a grade está desativada. Ela pode ser ligada e desligada a qualquer momento durante uma sessão de edição e pode ser alternada entre polegadas e centímetros a qualquer momento. Uma opção de grade será fornecida na visão reduzida do diagrama, mas o número de linhas da grade mostrado diminuirá, para evitar preencher o diagrama menor com linhas de grade.

2. Um requisito não funcional, que fornece informações detalhadas sobre as unidades da grade (centímetros ou polegadas).
3. Um requisito não funcional de interface com o usuário, que define como essa grade é ligada e desligada pelo usuário.

O requisito na Figura 5.9 também apresenta algumas informações, mas não todas as de inicialização. Ele especifica que a grade está inicialmente desligada, contudo não define suas unidades quando está ligada. Ele fornece informações detalhadas, por exemplo, que o usuário pode alternar entre unidades, mas não o espaçamento entre as linhas da grade.

Quando os requisitos do usuário apresentam muitas informações, isso restringe a liberdade do desenvolvedor do sistema para oferecer soluções inovadoras aos problemas do usuário e faz com que os requisitos sejam de difícil compreensão. Os requisitos do usuário devem simplesmente mostrar os recursos principais a serem fornecidos. Isso é ilustrado na Figura 5.10, em que reescrevi o requisito para a grade de editor, focalizando apenas as características essenciais do sistema.

A lógica associada com os requisitos é importante. Ela ajuda os desenvolvedores de sistema e os responsáveis pela sua manutenção a compreender por que os requisitos foram incluídos e a avaliar o impacto da mudança nos requisitos. Por exemplo, na Figura 5.10, a lógica reconhece que pode ser útil uma grade ativa, em que objetos posicionados automaticamente saltam para uma linha da grade. Contudo, ela rejeita deliberadamente essa opção em favor de um posicionamento manual. Se uma mudança nessa opção for proposta em algum estágio posterior, ficará claro que o uso de uma grade passiva foi deliberado, em vez de ser uma decisão de implementação.

Outro exemplo de um requisito de usuário mais específico, que também define parte do sistema de edição, é mostrado na Figura 5.11. Essa é uma especificação mais detalhada de uma função. Nesse caso, a definição inclui uma lista de ações do usuário. Essa opção é, algumas vezes, necessária, de modo que todas as funções possam ser fornecidas de maneira consistente. Os detalhes de implementação não devem ser incluídos nessas informações adicionais.

Portanto, a definição não estabelece como o cursor e os símbolos são movimentados ou como o tipo é selecionado.

Para minimizar possíveis divergências quando estiver redigindo os requisitos do usuário, recomendo que o leitor siga algumas diretrizes simples:

1. Invente um formato-padrão e certifique-se de que todas as definições de requisitos estejam conforme esse formato. Padronizar o formato significa que as omissões podem ser menos freqüentes e faz com que os requisitos sejam verificados com mais facilidade. "O formato que utilizo inclui 'reforçar' o requisito inicial, considerando uma declaração da lógica, com cada requisito de usuário e uma referência à especificação mais detalhada de requisitos de sistema."
2. Utilize a linguagem de modo consistente. Em particular, faça uma distinção entre os requisitos obrigatórios e os que são desejáveis. É prática usual definir os requisitos obrigató-

Figura 5.10 ▶

Uma definição de um recurso de grade de editor.

2.6 Recursos de grade

2.6.1 O editor deverá fornecer um recurso de grade, em que uma matriz de linhas horizontais e verticais constitua um fundo da janela do editor. Essa grade deverá ser uma tela passiva em que o alinhamento de entidades é de responsabilidade do usuário.
Lógica: uma grade ajuda o usuário a criar um diagrama 'limpo', com entidades bem espaçadas. Embora uma grade ativa, em que as entidades saltam as linhas de grade, possa ser útil, o posicionamento é impreciso. O usuário é a melhor pessoa para decidir onde as entidades devem ser posicionadas.

Especificação: ECLIPSE/WS/Ferramentas/DE/FS. Seção 5.6

Figura 5.11 ▶

Requisitos de usuário para criação de nó.

3.5.1 Adicionando nós a um desenho

3.5.1.1 **O editor deve fornecer um recurso aos usuários para adicionar nós de um tipo especificado a seu desenho.**

3.5.1.2 A seqüência de ações para acrescentar um nó deve ser como se segue:

1. O usuário deve selecionar o tipo de nó a ser acrescentado.
2. O usuário deve mover o cursor para a posição aproximada do nó no diagrama e indicar que o símbolo do nó deve ser adicionado naquele ponto.
3. O usuário deve então arrastar o símbolo do nó para sua posição final.
   Lógica: o usuário é a melhor pessoa para decidir onde posicionar um nó no diagrama. Essa abordagem dá ao usuário o controle direto sobre a seleção do tipo de nó e seu posicionamento.

Especificação: ECLIPSE/WS/Ferramentas/DE/FS. Seção 3.5.1

rios utilizando-se o verbo 'deve' e os requisitos desejáveis, utilizando-se o verbo 'deveria', como podemos ver na Figura 5.11. Portanto, é obrigatório que o sistema inclua um recurso para acrescentar nós em um desenho. É desejável que a seqüência de ações seja feita como foi especificado, mas isso não será absolutamente essencial, se houver boas razões para fazê-la de outra maneira.

3. Utilize destaque no texto (negrito ou itálico) para ressaltar partes importantes dos requisitos.

4. Evite, tanto quanto possível, o uso de jargões de informática. Contudo, inevitavelmente, ocorrerá que termos técnicos detalhados, utilizados no domínio da aplicação do sistema, sejam incluídos nos requisitos do usuário.

5.3 Requisitos de sistema

Os requisitos de sistema são descrições mais detalhadas dos requisitos do usuário. Eles podem servir como base para um contrato destinado à implementação do sistema e, portanto, devem ser uma especificação completa e consistente de todo o sistema. Eles são utilizados pelos engenheiros de software como ponto de partida para o projeto de sistema.

A especificação de requisitos de sistema pode incluir diferentes modelos do sistema, como um modelo de objeto ou um modelo de fluxo de dados. Descrevo diferentes modelos que podem ser utilizados na especificação de requisitos de sistema no Capítulo 7.

Em princípio, os requisitos de sistema deveriam definir o que o sistema deveria fazer, e não como ele teria de ser implementado. Contudo, no que se refere aos detalhes exigidos para especificar o sistema completamente, é quase impossível excluir todas as informações de projeto. Há várias razões para isso:

1. Uma arquitetura inicial do sistema pode ser definida para ajudar a estruturar a especificação de requisitos. Os requisitos de sistema são organizados de acordo com os diferentes subsistemas que constituem o sistema.

2. Na maioria dos casos, os sistemas devem interoperar com outros sistemas existentes. Isso restringe o projeto, e essas restrições geram requisitos para o novo sistema.

3. O uso de um projeto específico (como a programação de N-versões para atingir a confiabilidade, que será discutida no Capítulo 18) pode ser um requisito externo de sistema.

A linguagem natural é utilizada freqüentemente para escrever especificações de requisitos de sistema. Contudo, da mesma maneira que os problemas identificados na Seção 5.2, outros problemas com a linguagem natural podem surgir, quando ela é utilizada para uma especificação mais detalhada:

1. A compreensão da linguagem natural depende do uso das mesmas palavras para o mesmo conceito, pelos leitores e por quem escreve as especificações. Isso leva a divergências, devido à ambigüidade da linguagem natural. Jackson (1995) apresenta um excelente exemplo disso quando discute os sinais exibidos em uma escada rolante. Eles diziam "É preciso estar calçado" e "Os cachorros devem ser carregados". Deixo a seu encargo analisar a interpretação conflitante dessas frases.
2. Uma especificação de requisitos em linguagem natural é muito flexível. Pode-se dizer a mesma coisa de modos completamente diferentes. Fica por conta do leitor descobrir quando os requisitos são os mesmos e quando são diferentes.
3. Não existe nenhum meio fácil de padronizar os requisitos de linguagem natural. Pode ser difícil encontrar todos os requisitos relacionados. Para descobrir as conseqüências de uma mudança, talvez seja preciso examinar cada requisito, em vez de simplesmente um grupo de requisitos relacionados.

Devido a esses problemas, as especificações de requisitos escritas em linguagem natural estão sujeitas a divergências. Muitas vezes, elas somente são descobertas em fases posteriores do processo de software e, então, a solução para elas pode ser muito dispendiosa.

Existem várias alternativas para o uso da linguagem natural, que adicionam estrutura à especificação e que ajudam a reduzir a ambigüidade. Elas são mostradas na Figura 5.12.

Outras abordagens, como linguagens especializadas de requisitos (Teichrow e Hershey, 1977; Alford, 1977; Bell *et al.*, 1977; Alford, 1985), também foram desenvolvidas, mas atualmente elas são pouco utilizadas. Davis (1990) resume e compara algumas dessas diferentes abordagens da especificação de requisitos. Neste capítulo, são focalizadas as duas primeiras dessas abordagens, ou seja, a linguagem natural estruturada e o uso de linguagens de descrição de projeto.

Figura 5.12 ▶

Notações para a especificação de requisitos.

Notação	Descrição
Linguagem natural estruturada	Essa abordagem depende da definição de formulários-padrão ou *templates* para expressar a especificação de requisitos.
Linguagem de descrição de projeto	Essa abordagem utiliza uma linguagem como uma linguagem de programação, mas com recursos mais abstratos para especificar os requisitos pela definição de um modelo operacional do sistema.
Notações gráficas	Uma linguagem gráfica, complementada com anotações de texto, é utilizada para definir os requisitos funcionais do sistema. Um exemplo anterior dessa linguagem gráfica foi SADT (Ross, 1977; Schoman e Ross, 1977). Mais recentemente, as descrições de use-case (Jacobson *et al.*, 1993) foram utilizadas. Essas linguagens serão discutidas no próximo capítulo.
Especificações matemáticas	São notações com base em conceitos matemáticos, como uma máquina de estados finitos e conjuntos. Essas especificações não ambíguas reduzem as discussões entre cliente e fornecedor sobre a funcionalidade do sistema. Contudo, a maioria dos clientes não compreende as especificações formais e reluta em aceitá-las no momento de uma contratação de sistema. Discutirei as especificações formais no Capítulo 9.

5.3.1 Especificações em linguagem estruturada

A linguagem natural estruturada é uma forma restrita da linguagem natural, que se destina a escrever requisitos de sistema. A vantagem dessa abordagem é que ela mantém a maior parte da facilidade de expressão e compreensão da linguagem natural, mas garante que algum grau de uniformidade seja imposto à especificação. As notações de linguagem estruturada podem limitar a terminologia utilizada e utilizar templates para especificar os requisitos de sistema. Elas também podem incorporar princípios de controle procedentes de linguagens de programação e destaques gráficos para dividir a especificação.

Um projeto que utilizou linguagem natural estruturada para especificar requisitos de sistema é descrito por Heninger (1980). Formulários com propósito especial foram projetados para descrever a entrada, a saída e as funções do sistema de software de uma aeronave. Os requisitos de sistema foram especificados com o uso desses formulários.

Para utilizar uma abordagem com base em formulários para especificar os requisitos de sistema, é preciso definir um ou mais formulários ou templates-padrão para expressar os requisitos. A especificação pode ser estruturada em torno dos objetos manipulados pelo sistema, das funções realizadas ou dos eventos processados por ele. Um exemplo dessa especificação com base em formulários é mostrado na Figura 5.13. Essa é uma definição mais detalhada da função de nó Add para o sistema de projeto-edição, definido na Figura 5.11.

Quando um formulário-padrão é utilizado para especificar os requisitos funcionais, as seguintes informações devem ser incluídas:

1. uma descrição da função ou entidade que está sendo especificada;
2. uma descrição de suas entradas e de onde elas se originam;
3. uma descrição de suas saídas e para onde elas prosseguirão;
4. uma indicação de que outras entidades são utilizadas (a parte referente a 'requer');
5. se uma abordagem funcional for utilizada, será 'chamada' uma pré-condição estabelecendo o que deve ser verdadeiro antes da função; e também será 'chamada' uma pós-condição especificando o que é verdadeiro depois da função, e
6. uma descrição dos efeitos colaterais (se existirem) da operação.

Figura 5.13 ▶

Especificação de requisitos de sistema utilizando-se um formulário-padrão.

ECLIPSE/Workstation/Ferramentas/DE/FS/3.5.1

Função Adicionar nós.

Descrição Adiciona um nó em um desenho existente. O usuário seleciona o tipo de nó e seu posicionamento. Quando adicionado ao desenho, o nó se torna a seleção atual. O usuário escolhe a posição do nó movimentando o cursor para a área em que o nó será adicionado.

Entradas Tipo de nó, Posição do nó, Identificador do desenho.

Origem Tipo de nó e Posição do nó são entradas fornecidas pelo usuário; Identificador de desenho se origina da base de dados.

Saídas Identificador do desenho.

Destino O banco de dados do desenho. O desenho é designado para a base de dados, no término da operação.

Requer Gráfico de desenho associado ao identificador de desenho de entrada.

Pré-condição O desenho é aberto e exibido na tabela do usuário.

Pós-condição O desenho é imutável, a não ser pela adição de um nó do tipo especificado em dada posição.

Efeitos colaterais Nenhum.

Definição: ECLIPSE/Workstation/Ferramentas/DE/RD/3.5.1

Utilizar especificações formatadas elimina alguns dos problemas de especificação em linguagem natural, uma vez que há menos variação na especificação e os requisitos são agrupados de modo mais eficaz. Contudo, pode permanecer alguma ambigüidade na especificação. Métodos alternativos, que utilizam notações mais estruturadas, como uma PDL (descrita a seguir), avançam um pouco em direção à resolução dos problemas de ambigüidade da especificação. Contudo, quem não é especialista geralmente a considera de difícil compreensão.

5.3.2 Especificação de requisitos com o uso de uma PDL

Ao contrário das ambigüidades inerentes à especificação em linguagem natural, é possível descrever os requisitos operacionalmente com o uso de uma PDL (*program description language* – linguagem de descrição de programa). Uma PDL é uma linguagem derivada de uma linguagem de programação como Java ou Ada. Ela pode conter princípios mais abstratos, adicionais, para aumentar seu poder de expressão. A vantagem de utilizar uma PDL é que ela pode ser verificada sintática ou semanticamente por ferramentas de software. Omissões e inconsistências de requisitos podem ser inferidas a partir dos resultados dessas verificações.

As PDLs resultam em especificações muito detalhadas e, algumas vezes, estão muito perto da implementação para sua inclusão em um documento de requisitos. Contudo, recomendo sua utilização em duas situações:

1. Quando uma operação é especificada como uma seqüência de ações mais simples e a ordem de execução é importante. As descrições dessas seqüências em linguagem natural são, muitas vezes, confusas, particularmente se condicionais aninhadas e loops estiverem envolvidos. Isso é ilustrado na Figura 5.14, que define parte da especificação de uma operação de um caixa automático (*automatic teller machine* – ATM). Utilizei Java como a PDL, mas deliberadamente exclui partes da descrição, para poupar espaço. Uma descrição completa em Java da ATM pode ser obtida por download a partir das páginas Web deste livro.
2. Quando interfaces de hardware e software tiverem de ser especificadas. Em muitos casos, as interfaces entre subsistemas são definidas na especificação de requisitos de sistema. O uso de uma PDL permite que tipos e objetos de interface sejam especificados.

Se o leitor dos requisitos de sistema estiver familiarizado como a PDL utilizada, especificar os requisitos dessa maneira pode torná-los menos ambíguos e mais fácil de ser compreendidos. Se a PDL tiver como base a linguagem de implementação, existirá uma transição natural dos requisitos para o projeto. A possibilidade de má interpretação fica reduzida. Os especificadores não precisam ser treinados em outra linguagem descritiva.

Há desvantagens nessa abordagem de especificação de requisitos:

1. A linguagem utilizada para escrever a especificação pode não ser suficientemente expressiva para descrever a funcionalidade do sistema.
2. A notação só é compreensível para pessoas que tenham algum conhecimento de linguagem de programação.
3. O requisito pode ser considerado um esboço de especificação de projeto, em vez de um modelo para ajudar o usuário a compreender o sistema.

Uma maneira efetiva de utilizar essa abordagem de especificação é combiná-la com o uso da linguagem natural estruturada. Uma abordagem com base em formulários pode ser utilizada para especificar o sistema completo. Uma PDL pode, então, ser utilizada para definir seqüências de controle ou interfaces mais detalhadamente.

5.3.3 Especificação de interface

A grande maioria dos sistemas de software deve operar com outros sistemas que já foram implementados e instalados em um ambiente. Se o novo sistema e os sistemas existentes devem trabalhar em conjunto, as interfaces de sistemas existentes precisam ser especificadas com precisão. Essas especificações devem ser definidas no início do processo e incluídas (talvez como um apêndice) no documento de requisitos.

Existem três diferentes tipos de interfaces que podem precisar ser definidas:

1. Interfaces de procedimento, em que subsistemas existentes oferecem uma gama de funções, que são acessadas ao chamar procedimentos de interface.
2. Estruturas de dados, que são transmitidas de um subsistema para outro. Uma PDL baseada em Java pode ser utilizada para isso, com a estrutura de dados sendo descrita com o uso de uma definição de classe com atributos representando campos da estrutura. Contudo, acho que os diagramas de relacionamento de entidades (descritos no Capítulo 7) são melhores para esse tipo de descrição.
3. Representações de dados (como a ordenação de bits) que foram estabelecidas para um subsistema existente. Java não é compatível com essa especificação de representação detalhada; assim, não recomendo o uso da PDL baseada em Java, nesse caso.

As notações formais, discutidas no Capítulo 9, permitem que as interfaces sejam definidas de maneira não ambígua, mas sua natureza especializada significa que elas não são compreensíveis sem um treinamento especial. Elas raramente são utilizadas na prática para a especificação de interfaces, embora, a meu ver, são perfeitamente adequadas para esse propósito. Menos formais, as descrições de interfaces em PDL são uma conciliação entre a facilidade de compreensão e a precisão; mas, geralmente, elas são mais precisas do que a especificação de interfaces em linguagem natural.

A Figura 5.15 é um exemplo de uma definição do primeiro desses tipos de interfaces. Nesse caso, trata-se da interface procedural, oferecida por um servidor de impressão. Ela gerencia uma fila de pedidos de impressão de arquivos em diferentes impressoras. Os usuários podem examinar a fila associada a uma impressora e remover seus serviços de impressão dessa fila. Eles podem também transferir o serviço de uma impressora para outra.

A especificação na Figura 5.15 é um modelo abstrato do servidor de impressão, sem revelar quaisquer detalhes da interface. Essa funcionalidade das operações de interface pode ser definida utilizando-se linguagem natural estruturada (Figura 5.10), PDL com base em Java (Figura 5.14) ou uma notação formal, como será discutido no Capítulo 9.

5.4 O documento de requisitos de software

O documento de requisitos de software – às vezes, chamado de SRS (*software requirements specification*) ou especificação de requisitos de software – é a declaração oficial do que é exigido dos desenvolvedores de sistema. Ele deve incluir os requisitos de usuário para um sistema e uma especificação detalhada dos requisitos de sistema. Em alguns casos, os requisitos de usuário e de sistema podem ser integrados em uma única descrição. Em outros casos, os requisitos de usuário são definidos em uma introdução à especificação dos requisitos de sistema. Se houver um grande número de requisitos, os requisitos detalhados de sistema poderão ser apresentados como documentos separados.

O documento de requisitos tem um conjunto diversificado de usuários, abrangendo desde a alta gerência da organização, que está pagando pelo sistema, até os engenheiros respon-

Figura 5.14

Uma descrição em PDL da operação de um ATM.

```
class ATM {
  // declarations here
  public static void main (String args[]) throws InvalidCard {
      try {
          thisCard.read () ; // may throw InvalidCard exception
          pin = KeyPad.readPin () ; attempts = 1 ;
          while ( !thisCard.pin.equals (pin) & attempts < 4 )
              {    pin = KeyPad.readPin () ; attempts = attempts + 1 ;
              }
              if (!thisCard.pin.equals (pin))
                  throw new InvalidCard ("Bad PIN") ;
          thisBalance = thisCard.getBalance () ;
          do { Screen.prompt (" Please select a service ") ;
              service = Screen.touchKey () ;
              switch (service) {
                  case Services.withdrawalWithReceipt:
                          receiptRequired = true ;
                  case Services.withdrawalNoReceipt:
                          amount = KeyPad.readAmount () ;
                          if (amount > thisBalance)
                          {    Screen. printmsg ("Balance insufficient") ;
                               break ;
                          }
                          Dispenser.deliver (amount) ;
                          newbalance = thisBalance – amount ;
                          if (receiptRequired)
                              Receipt.print (amount, newBalance) ;
                          break ;
                  // other service descriptions here
                  default: break ;
              }
          }
          while (service != Services.quit) ;
          thisCard.returnToUser ("Please take your card") ;
      }
      catch (InvalidCard e)
      {    Screen.printmsg ("Invalid card or PIN") ;
      }
      // other exception handling here
  } //main ()
} //ATM
```

sáveis pelo desenvolvimento do software. A Figura 5.16 (Kotonya e Sommerville, 1998) ilustra os possíveis usuários do documento e como eles o utilizam.

Heninger (1980) sugere que existem seis requisitos aos quais um documento de requisitos de software deveria satisfazer:

- Deveria especificar somente o comportamento externo do sistema.

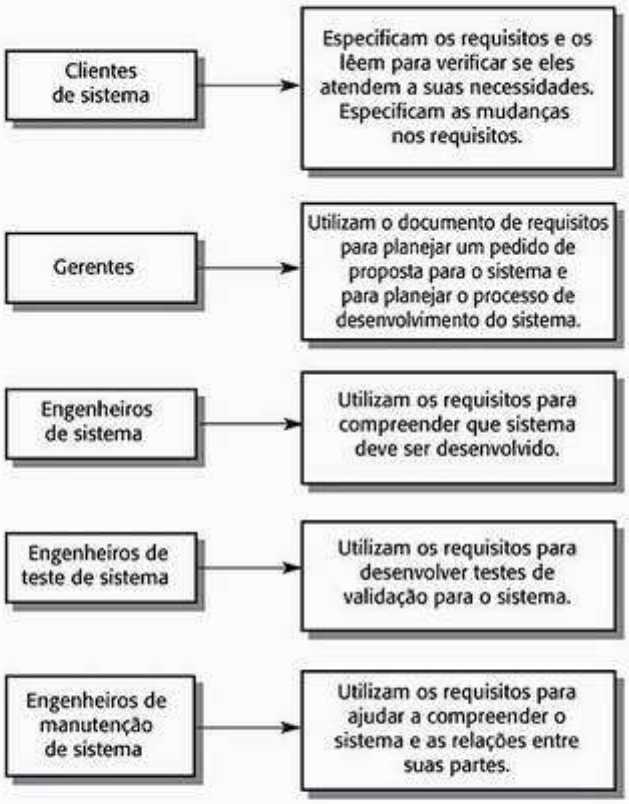
Figura 5.15

A descrição em PDL baseada em Java de uma interface de servidor de impressão (PrintServer).

```
interface PrintServer {
// define um servidor de impressora abstrato
// requer: impressora de interface Printer, interface PrintDoc
// fornece: inicialize, print, displayPrintQueue, cancelPrintJob, switchPrinter
  void initialize ( Printer p ) ;
  void print ( Printer p, PrintDoc d ) ;
  void displayPrintQueue ( Printer p ) ;
  void cancelPrintJob (Printer p, PrintDoc d) ;
  void switchPrinter (Printer p1, Printer p2, PrintDoc d) ;
} //PrintServer
```

Figura 5.16

Usuários de um documento de requisitos.



- Deveria especificar as restrições à implementação.
- Deveria ser fácil de ser modificado.
- Deveria servir como uma ferramenta de referência para os responsáveis pela manutenção do sistema.
- Deveria registrar a estratégia sobre o ciclo de vida do sistema.
- Deveria caracterizar respostas aceitáveis para eventos indesejáveis.

Embora tenham sido sugeridos há mais de 20 anos, esses são bons conselhos. Contudo, algumas vezes é difícil especificar sistemas em termos do que eles farão (seu comportamento externo). Inevitavelmente, devido às restrições originárias de sistemas existentes, o projeto de sistema é restringido, e isso deve ser refletido no documento de requisitos. Outro conselho, como a necessidade de registrar uma estratégia sobre o ciclo de vida do sistema, é muito bem aceito, mas não amplamente seguido, quando são escritos os documentos de requisitos.

Uma série de diferentes organizações de grande porte, como o Departamento de Defesa dos Estados Unidos e o IEEE, definiram padrões para os documentos de requisitos. Davis (1993) discute alguns desses padrões e compara seu conteúdo. O padrão mais amplamente conhecido é o padrão IEEE/ANSI 830-1993 (IEEE, 1993). Thayer e Dorfman (1997), em sua excelente coleção de artigos sobre engenharia de requisitos, incluem uma especificação completa desse padrão. Esse padrão do IEEE sugere a seguinte estrutura para os documentos de requisitos:

1. **Introdução**
   - 1.1 Propósito do documento de requisitos
   - 1.2 Escopo do produto
   - 1.3 Definições, acrônimos e abreviações
   - 1.4 Referências
   - 1.5 Visão geral do restante do documento

2. **Descrição geral**
   2.1 Perspectiva do produto
   2.2 Funções do produto
   2.3 Características do usuário
   2.4 Restrições gerais
   2.5 Suposições e dependências
3. **Requisitos específicos** que abrangem os requisitos funcionais, não funcionais e de interface. Essa é, obviamente, a parte mais substancial do documento, mas, devido à ampla variabilidade na prática organizacional, não é apropriado definir uma estrutura-padrão para essa seção. Os requisitos podem documentar interfaces externas, descrever funcionalidade e desempenho de sistema, especificar requisitos lógicos de banco de dados, restrições de projeto, propriedades emergentes do sistema e características de qualidade.
4. **Apêndices**
5. **Índice**

Embora o padrão do IEEE não seja ideal, ele contém uma grande quantidade de boas orientações sobre como escrever os requisitos e como evitar problemas. Ele é muito geral para ser considerado um padrão organizacional por si só. Contudo, ele pode ser adaptado para definir um padrão dirigido às necessidades de uma organização em particular. A Figura 5.17 ilustra uma possível organização para um documento de requisitos, que se baseia no padrão IEEE. Contudo, fiz uma ampliação para incluir informações sobre a evolução prevista para o sistema, como foi recomendado por Heninger.

Naturalmente, as informações incluídas em um documento de requisitos devem depender do tipo de software que está sendo desenvolvido e da abordagem de desenvolvimento que é utilizada. Se uma abordagem evolucionária for adotada para um produto de software (digamos), o documento de requisitos se omitirá em relação a muitos dos capítulos detalhados, sugeridos anteriormente. Nesse caso, os projetistas e os programadores utilizam seu julgamento para decidir como atender aos requisitos dos usuários para o sistema.

Por outro lado, quando o software é parte de um grande projeto de engenharia de sistemas, que inclui sistemas de hardware e software que interagem entre si, freqüentemente, é essencial definir os requisitos com um refinado nível de detalhes. Isso significa que os documentos de requisitos provavelmente serão muito extensos e que eles devem incluir a maioria dos capítulos mostrados na Figura 5.17. Para documentos extensos, é particularmente importante incluir uma tabela abrangente de conteúdo e um índice de documentos, de modo que os leitores possam encontrar as informações de que precisam.

Figura 5.17 ▶

A estrutura de um documento de requisitos.

Capítulo	Descrição
Prefácio	Ele deve definir o público a que se destina o documento e descrever seu histórico da versão, incluindo uma lógica para a criação de uma nova versão e um sumário das mudanças feitas em cada versão.
Introdução	Deve descrever a necessidade do sistema. Deve descrever brevemente suas funções e explicar como ele deverá operar com outros sistemas. Deve descrever como o sistema se ajusta aos negócios em geral ou aos objetivos estratégicos da organização que está encomendando o software.
Glossário	Deve definir os termos técnicos utilizados no documento. Não se deve fazer suposições sobre a experiência ou o conhecimento apurado do leitor.
Definição de requisitos do usuário	Os serviços fornecidos para o usuário e os requisitos não funcionais do sistema devem ser descritos nesta seção. Essa descrição pode utilizar linguagem natural, diagramas ou outras notações que sejam compreensíveis pelos clientes. Padrões de produtos e de processos a serem seguidos devem ser especificados.

Arquitetura de sistemas	Esse capítulo deve apresentar uma visão geral de alto nível da arquitetura de sistema prevista, mostrando a distribuição de funções por meio de módulos de sistemas. Os componentes de arquitetura que estão sendo reutilizados devem ser destacados.
Especificação de requisitos do sistema	Deve descrever os requisitos funcionais e não funcionais com mais detalhes. Se for necessário, outros detalhes podem também ser adicionados aos requisitos não funcionais; por exemplo, podem ser definidas interfaces com outros sistemas.
Modelos de sistema	Devem ser estabelecidos um ou mais modelos de sistemas, mostrando o relacionamento entre os componentes de sistema e o sistema e seu ambiente. Esses podem ser os modelos de objeto, os modelos de fluxo de dados e os modelos semânticos de dados.
Evolução de sistema	Devem ser descritas as suposições fundamentais nas quais o sistema se baseia e as mudanças previstas, devidas à evolução de hardware, mudanças nas necessidades de usuário etc.
Apêndices	São fornecidos detalhes e informações específicas relacionadas à aplicação que está sendo desenvolvida. Exemplos de apêndices que podem ser incluídos são descrições de hardware e bases de dados. Os requisitos de hardware definem as configurações mínima e ótima para o sistema. Os requisitos de base de dados definem a organização lógica dos dados utilizados pelo sistema e os relacionamentos entre esses dados.
Índice	Podem ser incluídos vários índices no documento. Da mesma maneira que um índice normal alfabético, pode haver um índice de diagramas, um índice de funções, entre outros.

PONTOS-CHAVE

- Os requisitos para um sistema de software estabelecem o que o sistema deve fazer e definem restrições sobre sua operação e implementação.
- Os requisitos funcionais são declarações de funções que o sistema deve fornecer ou são descrições de como alguns cálculos devem ser realizados. Os requisitos de domínio são requisitos funcionais, provenientes de características do domínio de aplicação.
- Os requisitos não funcionais são os requisitos do produto, que restringem o sistema a ser desenvolvido, os requisitos de processo que se aplicam ao processo de desenvolvimento e os requisitos externos. Eles freqüentemente se relacionam às propriedades emergentes do sistema e, portanto, se aplicam ao sistema como um todo.
- Os requisitos de usuário se destinam às pessoas envolvidas no uso e na aquisição do sistema. Eles devem ser escritos utilizando-se linguagem natural, tabelas e diagramas, de modo que sejam compreensíveis.
- Os requisitos de sistema se destinam a comunicar, de modo preciso, as funções que o sistema tem de fornecer. Para reduzir a ambigüidade, eles podem ser escritos em uma linguagem estruturada de algum tipo, que pode ser um formulário estruturado de linguagem natural, uma linguagem com base em uma linguagem de programação de alto nível ou uma linguagem especial para a especificação de requisitos.
- O documento de requisitos de software é a declaração estabelecida dos requisitos do sistema. Ele deve ser organizado de modo que possa ser utilizado pelos clientes de sistema e pelos desenvolvedores de software.

LEITURAS SUGERIDAS

Requirements engineering: processes and techniques. Esse livro aborda todos os aspectos do processo de engenharia de requisitos e discute técnicas específicas de especificação de requisitos. (G. Kotonya e I. Sommerville, 1998, John Wiley and Sons.)

Software requirements engineering. É uma coleção de artigos sobre engenharia de requisitos, que apresenta diversos artigos relevantes, incluindo "Recommended practice for software requirements specification". É uma discussão do padrão do IEEE para documentos de requisitos. (R.H. Thayer e M. Dorfman (eds.), 1997, IEEE Computer Society Press.)

EXERCÍCIOS

5.1 Discuta os problemas da utilização de linguagem natural para definir os requisitos do usuário e do sistema e mostre, utilizando pequenos exemplos, como a linguagem natural estruturada em formulários pode ajudar a evitar algumas dessas dificuldades.

5.2 Descubra ambigüidades ou omissões no seguinte documento de requisitos de uma parte de um sistema de emissão de passagens.

> Um sistema automático de emissão de passagens vende passagens de trem. Os usuários escolhem seu destino e apresentam um cartão de crédito e um número de identificação pessoal. A passagem é emitida e o custo dessa passagem é incluído em sua conta do cartão de crédito. Quando o usuário pressiona o botão para iniciar, uma tela de menu com os possíveis destinos é ativada, juntamente com uma mensagem para que o usuário selecione um destino. Uma vez selecionado um destino, pede-se que os usuários insiram seu cartão de crédito. A validade do cartão é checada e o usuário, então, deve fornecer um número de identificação pessoal. Quando a transação de crédito é validada, a passagem é emitida.

5.3 Reescreva a descrição mostrada anteriormente utilizando a abordagem estruturada, descrita neste capítulo. Resolva as ambigüidades identificadas de maneira apropriada.

5.4 Escreva requisitos de sistema para o sistema em questão, utilizando uma notação com base em Java. Você pode fazer quaisquer suposições razoáveis sobre o sistema. Preste atenção, em particular, à especificação de erros do usuário.

5.5 Utilizando a técnica sugerida aqui, em que a linguagem natural é apresentada de maneira-padrão, escreva requisitos plausíveis do usuário para as seguintes funções:

- Um sistema de bomba de gasolina para auto-atendimento, que inclui uma leitora de cartão de crédito. O cliente passa o cartão pela leitora e registra a quantidade de combustível solicitada. O combustível é fornecido e a conta do cliente é debitada.
- A função de fornecer dinheiro em um caixa automático de banco.
- A função de verificar e corrigir a ortografia, em um processador de textos.

5.6 Descreva três tipos diferentes de requisitos não funcionais que possam ser colocados em um sistema. Dê exemplos de cada um desses três tipos de requisitos.

5.7 Escreva um conjunto de requisitos não funcionais para o sistema de emissão de passagens, descrito anteriormente, estabelecendo sua confiabilidade e seu tempo de resposta.

5.8 Quais são os requisitos para uma linguagem de programação, de modo que ela seja adequada para definir as especificações de interface? Comente sobre a adequação das linguagens C, Java e Ada para esse propósito.

5.9 Sugira como um engenheiro responsável pelo esboço de uma especificação de requisitos poderia manter o acompanhamento dos relacionamentos entre os requisitos funcionais e não funcionais.

5.10 Você conseguiu um emprego com um usuário de software que contratou seu funcionário anterior para desenvolver um sistema para eles. Você descobre que a interpretação de sua empresa dos requisitos é diferente da interpretação dada pelo seu funcionário anterior. Discuta o que você deveria fazer nessa situação. Você sabe que os custos para seu atual funcionário deverão aumentar, se as ambigüidades não forem resolvidas, e tem também uma responsabilidade de confidencialidade para com seu funcionário anterior.

6 Processos de engenharia de requisitos

Objetivos

O objetivo deste capítulo é descrever uma série de processos genéricos da engenharia de requisitos. Depois de ler este capítulo, você poderá:

- ❒ compreender as principais atividades da engenharia de requisitos e suas relações;
- ❒ conhecer várias técnicas de obtenção e análise de requisitos;
- ❒ compreender a importância da validação de requisitos e como as revisões de requisitos são utilizadas nesse processo, e
- ❒ compreender por que o gerenciamento de requisitos é necessário e como ele é compatível com outras atividades da engenharia de requisitos.

Sumário

6.1 Estudos de viabilidade
6.2 Obtenção e análise de requisitos
6.3 Validação de requisitos
6.4 Gerenciamento de requisitos

A engenharia de requisitos é um processo que envolve todas as atividades exigidas para criar e manter o documento de requisitos de sistema. Existem quatro atividades genéricas de processo de engenharia de requisitos que são de alto nível, ou seja, o estudo da viabilidade do sistema, a obtenção e a análise de requisitos, a especificação de requisitos e sua documentação e, finalmente, a validação desses requisitos. Neste capítulo, abordo todas essas atividades, com exceção da especificação e da documentação de requisitos, sobre as quais já discutimos no Capítulo 5. A Figura 6.1 ilustra a relação entre essas atividades e mostra também os documentos produzidos em cada estágio do processo de engenharia de requisitos (*requirements engineering* – RE).

As atividades de engenharia de requisitos, mostradas na Figura 6.1, dizem respeito à obtenção, à documentação e à verificação dos requisitos. Contudo, praticamente em todos os sistemas, os requisitos se modificam. As pessoas interessadas desenvolvem melhor compreensão do que elas querem que o software faça; a organização compradora do sistema sofre modificações; e são feitas alterações no hardware, no software e no ambiente organizacional do sistema. O gerenciamento de requisitos é uma atividade adicional da engenharia de requisitos, que se dedica a gerenciar as modificações nos requisitos. Abordo esse aspecto na seção final deste capítulo.

Algumas pessoas consideram a engenharia de requisitos como o processo de aplicação de um método estruturado, como a análise orientada a objetos (Rumbaugh *et al.*, 1991; Booch, 1994). Isso envolve analisar o sistema e desenvolver um conjunto de modelos gráficos de sistema, que, então, atua como uma especificação do sistema. O conjunto de modelos descreve o comportamento do sistema e recebe informações adicionais, descrevendo, por exemplo, seu desempenho ou sua confiabilidade requerida.

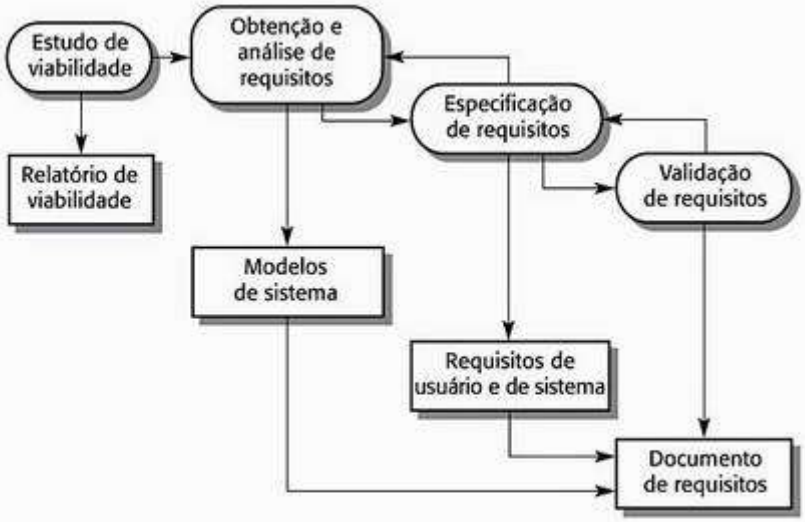
Embora os métodos estruturados tenham um papel a desempenhar no processo de engenharia de requisitos, existem muito mais aspectos na engenharia de requisitos do que os que são abordados por esses métodos. Eles não fornecem apoio efetivo aos estágios iniciais do processo de RE, como a obtenção de requisitos. Portanto, aqui, enfoco abordagens mais gerais da engenharia de requisitos e discuto métodos de análise estruturada e modelos de sistema no Capítulo 7.

6.1 Estudos de viabilidade

Para todos os sistemas novos, o processo de engenharia de requisitos de sistema deve começar com um estudo de viabilidade. A entrada para o estudo de viabilidade é uma descrição

Figura 6.1 ▶

Processo de engenharia de requisitos.



geral do sistema e de como ele será utilizado dentro de uma organização. Os resultados do estudo de viabilidade devem ser um relatório que recomenda se vale a pena ou não realizar o processo de engenharia de requisitos e o processo de desenvolvimento de sistemas.

Um estudo de viabilidade é um estudo breve, direcionado, que se destina a responder a algumas perguntas:

1. O sistema contribui para os objetivos gerais da organização?
2. O sistema pode ser implementado com a utilização de tecnologia atual dentro das restrições de custo e de prazo?
3. O sistema pode ser integrado com outros sistemas já em operação?

A questão sobre se o sistema contribui ou não para os objetivos da empresa é fundamental. Se um sistema não for compatível com esses objetivos, ele não terá nenhum valor real para a empresa. Embora isso possa parecer óbvio, muitas organizações desenvolvem sistemas que não contribuem para seus objetivos, seja porque não existe uma declaração clara desses objetivos ou porque outros fatores políticos ou organizacionais influenciam na aquisição do sistema.

Preparar um estudo de viabilidade envolve avaliar e coletar informações e redigir relatórios. A fase de avaliação identifica as informações exigidas para responder às três perguntas apresentadas anteriormente. Uma vez identificadas as informações, é preciso questionar as fontes de informação, a fim de encontrar as respostas para essas perguntas. Eis alguns exemplos das possíveis perguntas que devem ser feitas:

1. Como a organização se comportaria, se esse sistema não fosse implementado?
2. Quais são os problemas com os processos atuais e como um novo sistema ajudaria a diminuir esses problemas?
3. Que contribuição direta o sistema trará para os objetivos da empresa?
4. As informações podem ser transferidas para outros sistemas organizacionais e também podem ser recebidas a partir deles?
5. O sistema requer tecnologia que não tenha sido utilizada anteriormente na organização?
6. O que precisa e o que não precisa ser compatível com o sistema?

Entre as fontes de informação estão os gerentes de departamentos em que o sistema será utilizado, os engenheiros de software que estão familiarizados com o tipo de sistema proposto, peritos em tecnologia, usuários finais de sistema etc. Eles devem ser entrevistados durante o estudo de viabilidade, a fim de coletar as informações exigidas.

Quando as informações estiverem disponíveis, o relatório do estudo de viabilidade será preparado. Esse relatório deve recomendar se o desenvolvimento do sistema deve continuar ou não. Ele pode propor mudanças no enfoque, no orçamento e no cronograma, além de sugerir outros requisitos de alto nível para o sistema.

6.2 Levantamento e análise de requisitos

Depois dos estudos iniciais de viabilidade, o próximo estágio do processo de engenharia de requisitos é o levantamento e a análise de requisitos. Nessa atividade, os membros da equipe técnica de desenvolvimento de software trabalham com o cliente e os usuários finais do sistema para descobrir mais informações sobre o domínio da aplicação, que serviços o sistema deve fornecer, o desempenho exigido do sistema, as restrições de hardware e assim por diante.

O levantamento e a análise de requisitos podem envolver diferentes tipos de pessoas em uma organização. O termo *stakeholder* é utilizado para se referir a qualquer pessoa que terá alguma influência direta ou indireta sobre os requisitos do sistema. Dentre os *stakeholders* desta-

cam-se os usuários finais que interagirão com o sistema e todo o pessoal, em uma organização, que venha a ser por ele afetado. Os engenheiros que estão desenvolvendo o sistema ou fazendo a manutenção de outros sistemas relacionados, os gerentes de negócios, os especialistas nesse domínio, os representantes de sindicato, entre outros, podem ser também os *stakeholders* do sistema.

O levantamento e a análise compõem um processo difícil, por diversas razões:

1. Os *stakeholders* freqüentemente não sabem na realidade o que querem do sistema computacional, a não ser em termos muito gerais; eles podem achar difícil articular o que desejam do sistema; eles podem fazer pedidos não realistas, por não terem noção do custo de suas solicitações.

2. Os *stakeholders* em um sistema expressam naturalmente os requisitos em seus próprios termos e com o conhecimento implícito de sua área de atuação. Os engenheiros de requisitos que não têm experiência no domínio do cliente devem compreender esses requisitos.

3. Diferentes *stakeholders* têm em mente diferentes requisitos e podem expressá-los de maneiras distintas. Os engenheiros de requisitos precisam descobrir todas as possíveis fontes de requisitos e encontrar os pontos comuns e os conflitos.

4. Fatores políticos podem influenciar os requisitos do sistema. Eles podem provir de gerentes que definem requisitos específicos de sistema para aumentar sua influência na organização.

5. O ambiente econômico e de negócios, no qual a análise de requisitos ocorre, é dinâmico. Inevitavelmente, ele se modifica durante o processo de análise. Como conseqüência, a importância dos requisitos específicos pode mudar. Novos requisitos podem surgir por parte dos novos *stakeholders*, que não haviam sido consultados inicialmente.

Um modelo genérico do processo de levantamento e análise é mostrado na Figura 6.2. Cada organização terá sua própria versão ou uma versão mais definida desse modelo geral, dependendo de fatores locais, como a perícia da equipe, o tipo de sistema em desenvolvimento, os padrões utilizados, entre outros.

As atividades de processo são:

1. *Compreensão do domínio* Os analistas devem desenvolver sua compreensão do domínio da aplicação. Por exemplo, se for exigido um sistema para um supermercado, o analista deverá descobrir como operam os supermercados.

2. *Coleta de requisitos* É o processo de interagir com os *stakeholders* do sistema para descobrir seus requisitos. Obviamente, a compreensão do domínio se desenvolve mais durante essa atividade.

3. *Classificação* Essa atividade considera o conjunto não estruturado dos requisitos e os organiza em grupos coerentes.

4. *Resolução de conflitos* Quando múltiplos *stakeholders* estão envolvidos, os requisitos apresentarão conflitos. Essa atividade se ocupa de encontrar e solucionar esses conflitos.

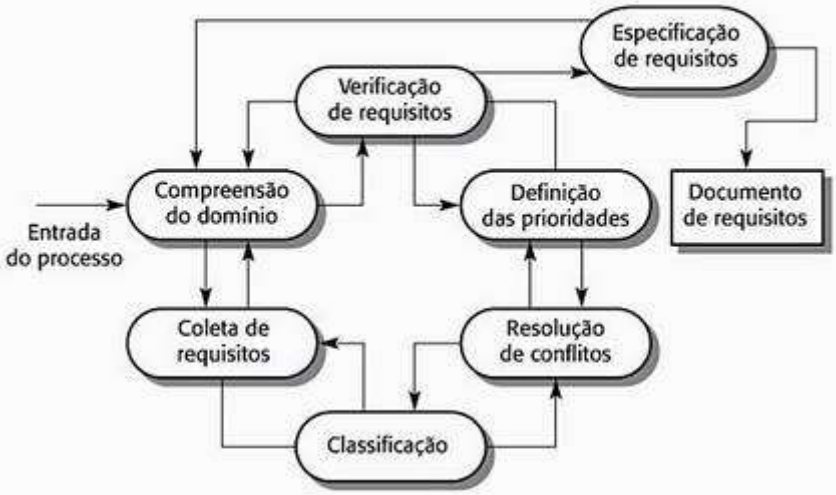
5. *Definição das prioridades* Em qualquer conjunto de requisitos, alguns serão mais importantes do que outros. Esse estágio envolve a interação com os *stakeholders*, para descobrir os requisitos mais importantes.

6. *Verificação de requisitos* Os requisitos são verificados, a fim de se descobrir se eles são completos e consistentes e se estão em concordância com o que os *stakeholders* realmente desejam do sistema.

A Figura 6.2 mostra que o levantamento e a análise de requisitos é um processo iterativo, com feedback contínuo de cada atividade para as outras. O ciclo começa com a com-

Figura 6.2

Processo de levantamento e análise de requisitos.



preensão do domínio e termina com a verificação dos requisitos. A compreensão dos requisitos pelo analista melhora a cada fase do ciclo.

Nesta seção, abordo três técnicas para o levantamento e a análise de requisitos, que são: o levantamento orientado a pontos de vista, os cenários e a etnografia. Outras técnicas que podem ser utilizadas durante essa fase do processo de engenharia de requisitos são os métodos de análise estruturada, abordados no Capítulo 7, e a prototipação, que será vista no Capítulo 8. Não existe uma abordagem perfeita e universalmente aplicável para a análise de requisitos. Normalmente, é preciso utilizar várias dessas abordagens para desenvolver compreensão e análise completas dos requisitos.

6.2.1 Levantamento orientado a pontos de vista

Para qualquer sistema, de tamanho médio ou grande, normalmente há diferentes tipos de usuário final. Muitos *stakeholders* têm algum tipo de interesse nos requisitos de sistema. Por exemplo, dentre os *stakeholders* de sistema, para o caso de um sistema de caixa automático de banco (ATM), destacam-se:

1. *clientes do banco* que recebem os serviços do sistema;
2. *representantes de outros bancos* que têm acordos de reciprocidade que permitem o uso de ATMs uns dos outros;
3. *gerentes de agências bancárias* que obtêm informações do sistema;
4. *equipes de atendimento de balcão em agências bancárias* envolvidas na operação diária do sistema, que atendem às reclamações de clientes etc.;
5. *administradores de bancos de dados* que são responsáveis pela integração do sistema com o banco de dados do cliente do banco;
6. *gerentes de segurança bancária* que devem assegurar que o sistema não apresente nenhum tipo de falha de segurança;
7. *departamento de marketing do banco* que provavelmente estará interessado em utilizar o sistema como um instrumento de marketing do banco, e
8. *engenheiros de manutenção de hardware e software* que têm a responsabilidade de fazer a manutenção e a atualização de hardware e software.

Essa relação mostra que, mesmo para um sistema relativamente simples, existem muitos *pontos de vista* diferentes que devem ser considerados. Os diferentes pontos de vista a respeito de um problema 'vêem' o problema de modos diferentes. Contudo, suas perspectivas não são inteiramente independentes, mas em geral apresentam alguma duplicidade, de modo que apresentem requisitos comuns.

As abordagens orientadas a pontos de vista, na engenharia de requisitos, reconhecem esses diferentes pontos de vista e os utilizam para estruturar e organizar o processo de levantamento e os próprios requisitos. Uma importante capacidade da análise orientada a pontos de vista é que ela reconhece a existência de várias perspectivas e oferece um *framework* para descobrir conflitos nos requisitos propostos por diferentes *stakeholders*.

Diferentes métodos apresentam diferentes idéias sobre o que significa um 'ponto de vista'. Um ponto de vista pode ser considerado:

1. *Uma fonte ou 'drenos' de dados* Nesse caso, os pontos de vista são responsáveis pela produção ou consumo de dados. A análise envolve identificar todos esses pontos de vista, identificar quais dados são produzidos ou consumidos e que processamento é realizado. Métodos como o SADT (Ross, 1977; Schoman e Ross, 1977) e o CORE (Mullery, 1979), o primeiro método a propor pontos de vista explícitos, utilizam esse tipo de ponto de vista.

2. *Um framework de representação* Nesse caso, um ponto de vista é considerado um tipo particular de modelo de sistema (Finkelstein *et al.*, 1990; Nuseibeh *et al.*, 1994). Por exemplo, diferentes engenheiros devem desenvolver um modelo de relacionamento de entidades, um modelo de máquina de estados, entre outros. Cada abordagem de análise descobre diferentes aspectos sobre o sistema que está sendo analisado.

3. *Um receptor de serviços* Nesse caso, os pontos de vista são externos ao sistema e dele recebem serviços (Kotonya e Sommerville, 1992, 1996). Eles podem fornecer dados para esses serviços ou sinais de controle. A análise envolve examinar os serviços recebidos por diferentes pontos de vista, coletando esses serviços e resolvendo conflitos.

Cada um desses modelos de ponto de vista apresenta diferentes aspectos positivos e negativos. Os pontos de vista como fontes de dados ou 'drenos' e os pontos de vista como representações são particularmente valiosos para a descoberta de conflitos detalhados entre os requisitos (Easterbrook e Nuseibeh, 1996). Contudo, eles são menos adequados para estruturar o processo de análise de requisitos, uma vez que não há nenhuma relação simples entre os pontos de vista e os *stakeholders* do sistema.

Os sistemas interativos fornecem serviços aos usuários finais. Conseqüentemente, a abordagem mais eficaz orientada a pontos de vista, para a análise de sistemas interativos, utiliza pontos de vista externos. Esses pontos de vista interagem com o sistema, recebendo serviços dele e fornecendo dados e sinais de controle para ele.

As vantagens desse tipo de ponto de vista são:

1. Os pontos de vista são externos ao sistema e, assim, são uma maneira natural de estruturar o processo de levantamento de requisitos.

2. É relativamente fácil decidir se alguma coisa é um ponto de vista válido. Os pontos de vista devem interagir com o sistema de alguma maneira.

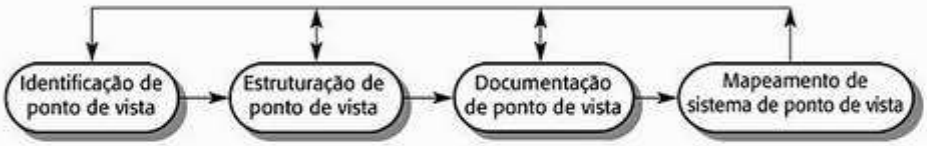
3. Os pontos de vista e os serviços são um meio útil de estruturar os requisitos não funcionais. Cada serviço pode ter requisitos não funcionais associados. Os pontos de vista múltiplos permitem que o mesmo serviço tenha diferentes requisitos não funcionais em diferentes pontos de vista.

O método VORD (*viewpoint-oriented requirements definition* – definição de requisitos orientada a pontos de vista) (Kotonya e Sommerville, 1996, 1998) foi projetado como um *framework* orientado a serviços, para o levantamento e a análise de requisitos. Os estágios principais do método VORD, mostrado na Figura 6.3, são:

1. Identificação de pontos de vista, que envolve descobrir os pontos de vista que utilizam serviços do sistema e identificar os serviços específicos fornecidos para cada ponto de vista.

Figura 6.3

O método VORD.



2. Estruturação de pontos de vista, que envolve agrupar pontos de vista relacionados, segundo uma hierarquia. Serviços comuns estão localizados nos níveis mais altos da hierarquia e herdados por pontos de vista de nível inferior.
3. A documentação do ponto de vista, que envolve refinar a descrição dos pontos de vista e serviços identificados.
4. Mapeamento de sistema conforme pontos de vista, que envolve identificar objetos em um projeto orientado a objetos, utilizando as informações de serviço que estão encapsuladas nos pontos de vista.

As informações sobre o ponto de vista e serviços no VORD são coletadas utilizando-se formulários-padrão. Os formulários utilizados para as informações do ponto de vista (*template* de ponto de vista) e as informações de serviços (*template* de serviços) são mostrados na Figura 6.4. O VORD também utiliza várias notações diagramáticas, inclusive diagramas de hierarquia de ponto de vista (veja o exemplo na Figura 6.8) e cenários de eventos (veja o exemplo na Figura 6.10).

Ilustro a aplicação do VORD empregando as três primeiras etapas na análise de requisitos para um sistema de controle de caixa automático de banco (ATM). Os ATMs têm um sistema de software embutido, destinado a dirigir o hardware da máquina e a se comunicar com a central de dados do banco.

O ATM aceita as solicitações do cliente e fornece dinheiro, informações sobre a conta, atualizações de informações, entre outros. Os clientes podem fazer retiradas e pagamentos em dinheiro, conferir o saldo, transferir dinheiro de uma conta para outra, pedir extrato de conta, talões de cheque etc. As máquinas fornecidas por um banco podem permitir que clientes de outros bancos utilizem um subconjunto de seus recursos (geralmente, retiradas em dinheiro e consultas de saldo de conta).

A primeira etapa da análise de ponto de vista é identificar os possíveis pontos de vista. Como em todos os métodos, essa identificação inicial é provavelmente o estágio mais difícil. Uma das abordagens é o brainstorming, no qual são identificados os serviços em potencial e as entidades que interagem com o sistema. Os *stakeholders* se reúnem e sugerem possíveis pontos de vista, que são anotados em um diagrama de bolhas, como mostra a Figura 6.5. Nesse diagrama, os pontos de vista em potencial para um sistema de ATM são mostrados em bolhas separadas.

Figura 6.4

Formulários de *templates* de pontos de vista e serviços.

Template de ponto de vista	
Referência:	o nome do ponto de vista.
Atributos:	atributos que fornecem informações sobre o ponto de vista.
Eventos:	uma referência a um conjunto de cenários de eventos que descreve como o sistema reage a eventos do ponto de vista.
Serviços:	uma referência a um conjunto de descrições de serviços.
Subpontos de vista:	os nomes de subpontos de vista.

Template de serviço	
Referência:	o nome do serviço.
Razão:	razão pela qual o serviço é fornecido.
Especificação:	referência a uma lista de especificações de serviço, que podem ser expressas em diferentes notações.
Pontos de vista:	lista de nomes de pontos de vista que recebem o serviço.
Requisitos não funcionais:	referência a um conjunto de requisitos não funcionais que impõem restrições ao serviço.
Provedores:	referência a uma lista de objetos de sistema que fornecem o serviço.

Em uma reunião de brainstorming, deve-se tentar identificar os pontos de vista em potencial, os serviços do sistema, as entradas de dados, os requisitos não funcionais, os eventos de controle e as exceções. Nesse estágio da análise, não se deve tentar impor uma estrutura ao diagrama. As fontes de informações que podem ser utilizadas para criar essa visão inicial do sistema podem ser os documentos que estabelecem os objetivos de alto nível do sistema, o conhecimento dos engenheiros de software sobre projetos precedentes ou a experiência como clientes de banco. As entrevistas podem ser feitas com gerentes de banco, pessoal de caixa, consultores, engenheiros e clientes.

O próximo estágio do processo é identificar os pontos de vista (mostrados como bolhas pretas, na Figura 6.5) e os serviços (mostrados como bolhas cinza). Os serviços devem ser alocados aos pontos de vista. Serviços não alocados podem sugerir pontos de vista que não foram identificados na sessão inicial de brainstorming. Por exemplo, os serviços de 'atualização de software remoto' e 'diagnósticos remotos', na Figura 6.5, implicam que pode haver a necessidade de um ponto de vista de manutenção de software.

A Figura 6.6 ilustra, para alguns dos pontos de vista identificados na Figura 6.5, a alocação de serviços. O mesmo serviço pode estar alocado a diversos pontos de vista.

Além de receber serviços, os pontos de vista também fornecem entradas para esses serviços. Por exemplo, os usuários de caixas eletrônicos devem especificar a quantia de dinheiro que desejam, quando fazem uma retirada. Os pontos de vista também fornecem informações sobre o controle, a fim de determinar se e quando os serviços serão efetuados.

Durante esse estágio inicial do processo, essas informações de dados e controle são simplesmente identificadas pelo nome. A Figura 6.7 mostra essas informações de controle para o ponto de vista do titular de conta, cujos serviços estão definidos na Figura 6.6. As informações de controle são fornecidas por meio de botões da máquina e os dados, pelo cartão do usuário ou pelo teclado da máquina.

As informações de ponto de vista são utilizadas para preencher os *templates* de ponto de vista e organizar os pontos de vista em uma hierarquia de herança. Para mostrar pontos de vista em comum e reutilizar as informações de ponto de vista, os fatores de hierarquia de herança salientam os pontos de vista que fornecem serviços comuns. As informações de serviços, dados e controle são herdadas por subpontos de vista.

A Figura 6.8 mostra parte da hierarquia de ponto de vista para o sistema de ATM. Para evitar o acúmulo, mostramos apenas os serviços associados com dois pontos de vista e omitimos uma série de pontos de vista do pessoal de banco. Como os serviços são herdados ao 'descer' na hierarquia de pontos de vista, os serviços gerais associados com o ponto de vista do 'cliente' ('consulta de saldo' e 'retirada de dinheiro') são herdados pelo 'titular da conta' e pelo 'não-titular da conta'.

Figura 6.5

Brainstorming para a identificação de pontos de vista.

Figura 6.6

Informações do serviço para o ponto de vista.

Figura 6.7

Dados de ponto de vista e informações de controle.

TITULAR DA CONTA

Entrada de controle	Entrada de dados
Iniciar transação Cancelar transação Encerrar transação Selecionar serviço	Detalhes do cartão PIN* Quantia solicitada Mensagem

*PIN = *personal identification number* (número de identificação pessoal)

O próximo estágio de processo é descobrir mais informações detalhadas sobre os serviços oferecidos, os dados que eles requerem e como são controlados. Os requisitos são coletados em consulta aos *stakeholders* associados com cada ponto de vista. Serão discutidas as necessidades de serviço de cada ponto de vista diferente, seja com os usuários finais ou com os especialistas em pontos de vista, se o ponto de vista for outro sistema automatizado. A Figura 6.9 mostra o exemplo de um *template* de pontos de vista completado conforme o ponto de vista do cliente e um *template* para o serviço de retirada de dinheiro.

Observe que um dos campos do *template* de ponto de vista é utilizado como referência a cenários de eventos que descrevem como o sistema se comporta em resposta a vários eventos. Esses cenários de eventos são discutidos na seção seguinte.

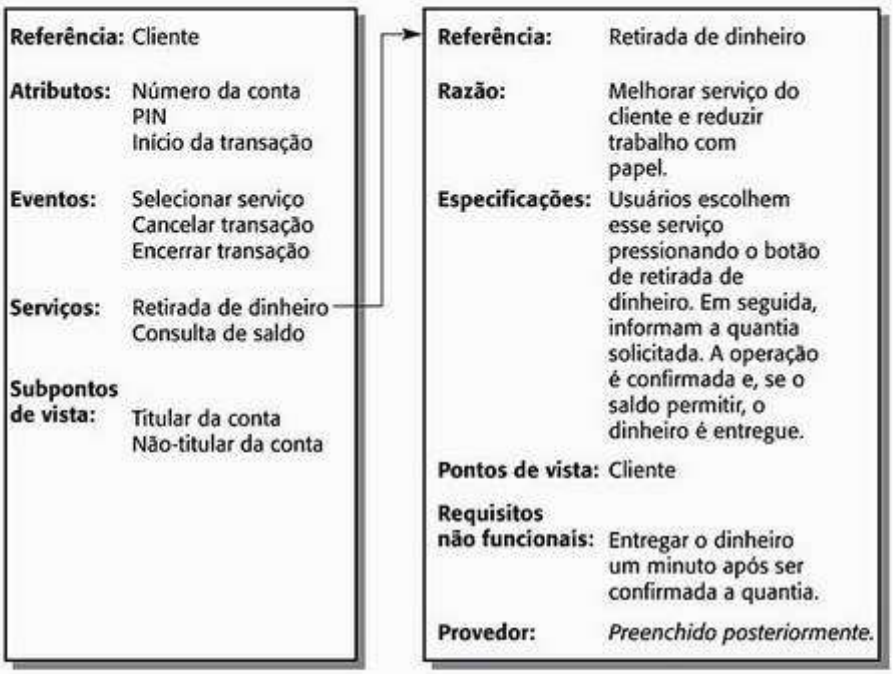
Templates de pontos de vista e de serviços, assim como cenários de eventos, são desenvolvidos para todos os pontos de vista e serviços identificados. As informações podem, então, sofrer uma verificação cruzada para descobrir erros na análise e conflitos de requisitos. Como isso gera uma grande quantidade de informações, o VORD, como os outros métodos de análise, somente é de uso prático com o apoio de ferramentas CASE. Um conjunto de ferramentas CASE para VORD está disponível e seu download pode ser feito por meio das páginas Web do livro.

Figura 6.8

Hierarquia de pontos de vista.

Figura 6.9

Descrições de ponto de vista do cliente e retirada de dinheiro.



6.2.2 Cenários

As pessoas geralmente acham mais fácil relacionar exemplos da vida real do que descrições abstratas. Elas podem compreender e criticar um cenário de como poderiam interagir com um sistema de software. Os engenheiros de requisitos podem utilizar as informações obtidas com essa discussão para formular os requisitos reais do sistema.

Os cenários podem ser particularmente úteis para acrescentar detalhes a um esboço da descrição de requisitos. Estas são descrições de exemplos de sessões de interação. Cada cenário aborda um ou um pequeno número de possíveis interações. Diferentes tipos de cenários foram desenvolvidos, e eles fornecem diferentes tipos de informações, com diferentes níveis de detalhes sobre o sistema.

O cenário começa com um esboço da interação e, durante o levantamento de requisitos, são acrescentados detalhes para criar uma descrição completa dessa interação. De modo geral, o cenário pode incluir:

1. uma descrição do estado do sistema no início do cenário;
2. uma descrição do fluxo normal de eventos no cenário;
3. uma descrição do que pode sair errado e de como lidar com isso;
4. informações sobre outras atividades que possam estar em andamento ao mesmo tempo, e
5. uma descrição do estado do sistema no final do cenário.

A obtenção de requisitos com base em cenários pode ser realizada informalmente, e o engenheiro de requisitos trabalha com os *stakeholders* para identificar cenários e captar detalhes desses cenários. Como alternativa, uma abordagem mais estruturada, como os cenários de eventos ou use-cases, pode ser empregada.

Cenários de eventos

Os cenários de eventos são utilizados em VORD para documentar o comportamento do sistema quando apresentado com eventos específicos. Cada evento distinto de interação, como inserir um cartão em um ATM ou selecionar um serviço de ATM, pode ser documentado com um cenário de evento separado. Entre os cenários de eventos estão a descrição do fluxo de dados e as ações do sistema. Além disso, esses cenários documentam as exceções que possam surgir. Para ilustrar esses cenários, considere a Figura 6.10, que mostra o cenário para

um evento 'iniciar transação'. Esse evento é iniciado por um cliente que insere seu cartão na máquina. Essa é uma descrição mais abstrata do que a descrição em Java-PDL, apresentada no Capítulo 5.

As convenções diagramáticas utilizadas em cenários de evento são:

1. Os dados fornecidos a partir de um ponto de vista ou entregues a um ponto de vista são mostrados em elipses.
2. As informações de controle entram e saem da parte superior de cada bloco.
3. Os dados saem da direita de cada bloco. Se eles não forem mencionados, isso significará que são internos ao sistema.
4. As exceções são mostradas na parte inferior do bloco. Onde houver diversas opções possíveis, elas serão incluídas em um bloco, como é mostrado à esquerda da Figura 6.10.
5. O nome do próximo evento esperado, depois do término do cenário, é mostrado em um bloco sombreado.

A Figura 6.10 mostra que, quando um cartão é inserido, o número de identificação pessoal (PIN) do cliente é solicitado. O cliente insere seu cartão (o cartão deve estar presente para acionar a solicitação) e seu PIN. Se o cartão for válido, ele poderá ser processado pela máquina e o controle poderá passar para o próximo estágio.

No primeiro estágio, há três exceções possíveis:

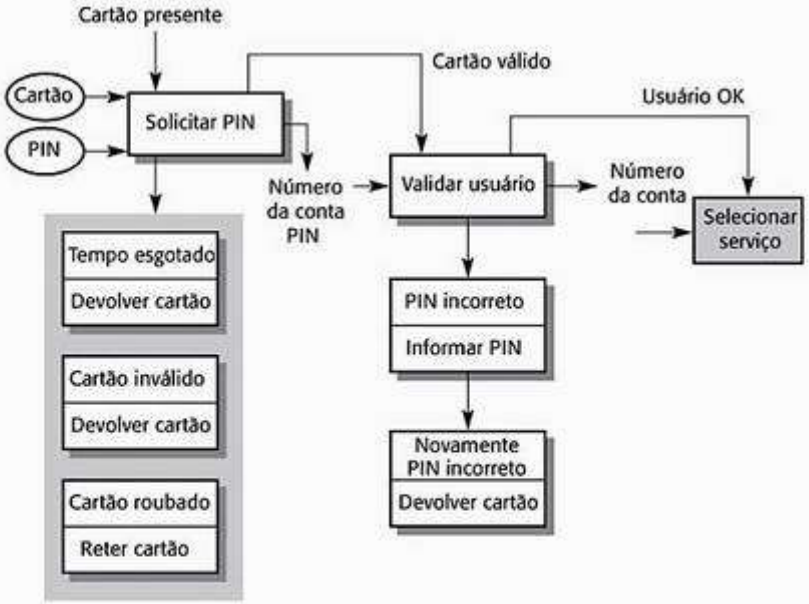
1. *Tempo esgotado* O cliente pode não fornecer o PIN dentro do limite de tempo permitido. O cartão é devolvido.
2. *Cartão inválido* O cartão não é reconhecido e é devolvido.
3. *Cartão roubado* O cartão é reconhecido como um cartão roubado e é retido pela máquina.

Cada exceção pode ser definida com mais detalhes, construindo-se um diagrama separado de análise de dados e de controle, que mostre como a exceção ocorre. Como alternativa, o diagrama geral pode ter a anotação de informações extras, que explicam quando as exceções ocorrem e as ações que devem ser realizadas.

O estágio de 'validar usuário' verifica se o PIN está associado ao número da conta do cliente e, em caso afirmativo, aciona o próximo estágio (selecionar serviço) com o evento 'usuário OK'. O número da conta é também uma saída desse estágio. Entre as possíveis exceções estão a entrada de um PIN incorreto e, nesse caso, o PIN é novamente solicitado. O diagrama mostra que essa solicitação repetida pode também ter uma exceção. Se um PIN incorreto for novamente fornecido, o cartão será devolvido.

Figura 6.10

Cenário do evento: iniciar transação.



Use-cases

Os use-cases são técnicas baseadas em cenários para a obtenção de requisitos, que foram introduzidos pela primeira vez no método Objectory (Jacobson *et al.*, 1993). Eles se tornaram, atualmente, uma característica fundamental da notação em UML (*unified modeling language* – linguagem de modelagem unificada) para descrever modelos de sistemas orientados a objetos. Na sua forma mais simples, um use-case identifica os agentes envolvidos em uma interação e especifica o tipo de interação. Isso está ilustrado na Figura 6.11, que descreve um use-case de serviços de biblioteca, em que um usuário pode pegar emprestado um livro de uma biblioteca ou devolvê-lo.

A Figura 6.11 ilustra os pontos essenciais da notação de use-case. Os agentes no processo são representados por bonecos e cada classe de interação é representada por uma elipse com um nome. O conjunto de use-cases representa todas as possíveis interações que serão representadas nos requisitos de sistema. Isso é mostrado na Figura 6.12, que desenvolve o exemplo da biblioteca e mostra outros use-cases nesse ambiente.

Às vezes, existe confusão sobre se um use-case é ou não um cenário em si; ou como foi sugerido por Fowler (Fowler e Scott, 1997), um use-case engloba um conjunto de cenários em que cada cenário é um traço isolado dentro do use-case. Nesse caso, haveria um cenário para a interação normal, mais os cenários para cada exceção possível.

Dentro da UML, diagramas de seqüência podem ser utilizados para acrescentar informações a um use-case. Esses diagramas de seqüência mostram os agentes envolvidos na interação, os objetos dentro do sistema com os quais eles interagem e as operações que estão associadas a esses objetos. Para ilustrar essa situação, a Figura 6.13 mostra a interação envolvida em adquirir e catalogar um livro para a biblioteca.

A Figura 6.13 ilustra que existem dois objetos das classes Item de Biblioteca e Catálogo envolvidos no use-case de Gerenciamento de Catálogo. A seqüência de ações é realizada de cima para baixo, e os rótulos nas setas entre os agentes e os objetos indicam os nomes das operações. Portanto, quando um livro é comprado, a operação *Novo* é feita em Catálogo e a operação *Adquirir*, em Item. Uma vez disponível o livro, a operação *Catalogar Item* é feita em Item.

A UML é um *verdadeiro* padrão para a modelagem orientada a objetos; assim, os use-cases e a obtenção de requisitos com base em use-case são cada vez mais utilizados para obter requisitos. Discuto outros aspectos da UML no Capítulo 7, que aborda a modelagem de sistemas e, no Capítulo 12, que trata de projetos orientados a objetos.

Figura 6.11 ▶

Um use-case de empréstimo.

Figura 6.12 ▶

Use-cases de biblioteca.

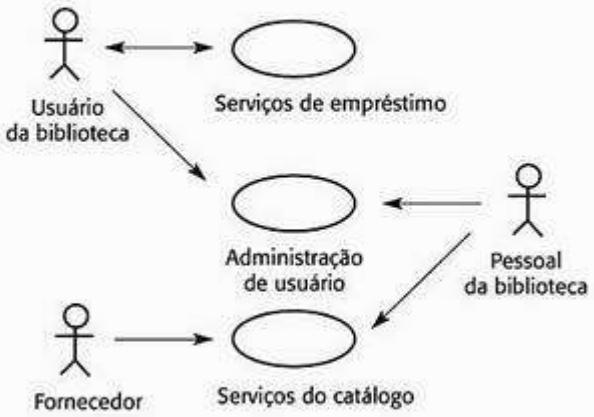
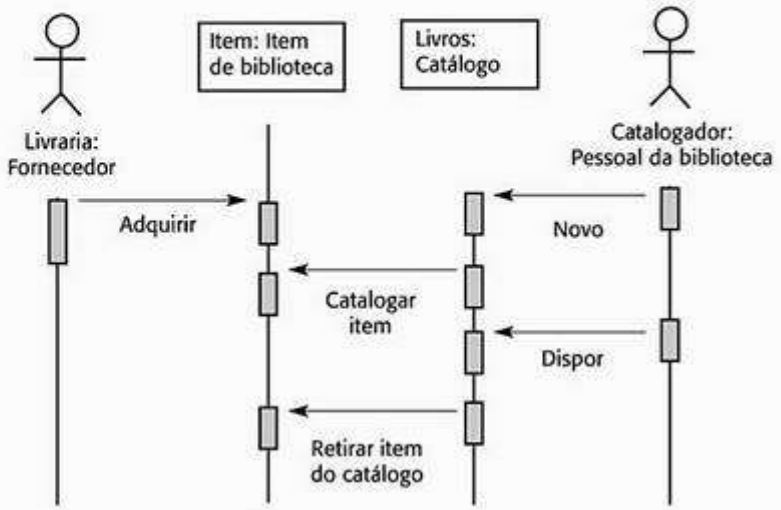


Figura 6.13

Diagrama de seqüência para o gerenciamento de catálogo.



6.2.3 Etnografia

Os sistemas de software não existem isoladamente. Eles são utilizados em um contexto social e organizacional, e os requisitos de sistema de software podem ser derivados ou limitados por esse contexto. Freqüentemente, satisfazer a esses requisitos sociais e organizacionais é fundamental para o sucesso do sistema. Uma razão pela qual muitos sistemas de software são entregues, mas nunca utilizados, é que eles não consideram adequadamente a importância desse tipo de requisito de sistema.

A etnografia é uma técnica de observação que pode ser utilizada para compreender os requisitos sociais e organizacionais. Um analista se insere no ambiente de trabalho em que o sistema será utilizado. O trabalho diário é observado e são anotadas as tarefas reais em que os participantes estão envolvidos. O valor da etnografia é que ela ajuda a descobrir requisitos de sistema implícitos, que refletem os processos reais, em vez de os processos formais, em que as pessoas estão envolvidas.

As pessoas geralmente acham muito difícil articular detalhes de seu trabalho, porque o trabalho para elas é sua segunda natureza. Elas compreendem seu próprio trabalho, mas podem não compreender a relação dele com outras atividades na organização. Os fatores sociais e organizacionais que afetam o trabalho, mas que não são óbvios para os indivíduos, somente podem ficar claros quando examinados por um observador imparcial.

Suchman (1983) utilizou a etnografia para estudar o trabalho em escritório e descobriu que as práticas de trabalho reais eram muito mais ricas, mais complexas e dinâmicas do que os simples modelos supostos pelos sistemas de automação de escritórios. A diferença entre o trabalho suposto e o real foi a razão mais importante pela qual esses sistemas de escritório não tinham nenhum efeito significativo na produtividade. Outros estudos etnográficos para a compreensão de requisitos de sistema incluíram o trabalho sobre controle de tráfego aéreo (Bentley *et al.*, 1992; Hughes *et al.*, 1992), as salas de controle de metrô (Heath e Luff, 1991), os sistemas financeiros (Hughes *et al.*, 1994) e várias atividades de projeto (Pycock e Bowers, 1996; Button e Sharrock, 1997).

A etnografia é particularmente eficaz na descoberta de dois tipos de requisitos:

1. Os requisitos derivados da maneira como as pessoas realmente trabalham, em vez da maneira pela qual as definições de processo dizem como elas deveriam trabalhar. Por exemplo, os controladores de tráfego aéreo podem desligar um sistema de alerta de conflito aéreo, que detecta aeronaves voando em rota de colisão, muito embora os procedimentos de controle normais especifiquem que o sistema deve ser utilizado. A estratégia de controle é projetada para garantir que essas aeronaves sejam afastadas, antes que

Figura 6.14 ▶
Etnografia e prototipação para análise de requisitos.

ocorram problemas, e os controladores acham que o alarme de alerta de conflito tira sua atenção do trabalho.

2. Os requisitos derivados da cooperação e conscientização das atividades de outras pessoas. Por exemplo, os controladores de tráfego aéreo podem utilizar a conscientização do trabalho de outros controladores para prever o número de aeronaves que entrarão no seu setor de controle. Em seguida, eles modificam suas estratégias de controle, dependendo da sobrecarga prevista. Portanto, um sistema de ATC (*air traffic control* – controle de tráfego aéreo) automatizado deve permitir que os controladores em um setor possam ter alguma visibilidade do trabalho em setores adjacentes.

A etnografia pode ser combinada com a prototipação (Figura 6.14). A etnografia informa o desenvolvimento do protótipo, de modo que um número menor de ciclos de refinamento do protótipo seja necessário. Além disso, a prototipação enfoca a etnografia pela identificação de problemas e questões que podem então ser discutidas com o etnógrafo. O etnógrafo deve então procurar respostas para essas questões, durante a nova fase de estudo do sistema (Sommerville *et al.*, 1993).

Os estudos de etnografia podem revelar importantes detalhes de processo que, freqüentemente, são omitidos por outras técnicas de obtenção de requisitos. Contudo, devido ao seu enfoque no usuário final, essa abordagem não é apropriada para descobrir requisitos organizacionais e de domínio. Ela nem sempre pode identificar novas características que deveriam ser acrescentadas a um sistema. A etnografia não é, portanto, uma abordagem completa para a obtenção de requisitos e deveria ser utilizada com outras abordagens, como a análise de use-case.

6.3 Validação de requisitos

A validação de requisitos se ocupa de mostrar que os requisitos realmente definem o sistema que o cliente deseja. Ela tem muito em comum com a análise de requisitos, uma vez que se preocupa em descobrir problemas nos requisitos. Contudo, esses são processos distintos, já que a validação deve se ocupar da elaboração de um esboço completo do documento de requisitos, enquanto a análise envolve trabalhar com requisitos incompletos.

A validação de requisitos é importante porque a ocorrência de erros em um documento de requisitos pode levar a grandes custos relacionados ao retrabalho, quando esses erros são descobertos durante o desenvolvimento ou depois que o sistema estiver em operação. O custo de fazer uma modificação no sistema, resultante de um problema de requisito, é muito maior do que reparar erros de projeto ou de codificação. A razão disso é que uma mudança nos requisitos, em geral, significa que o projeto do sistema e a implementação também devem ser modificados e que o sistema tem de ser novamente testado.

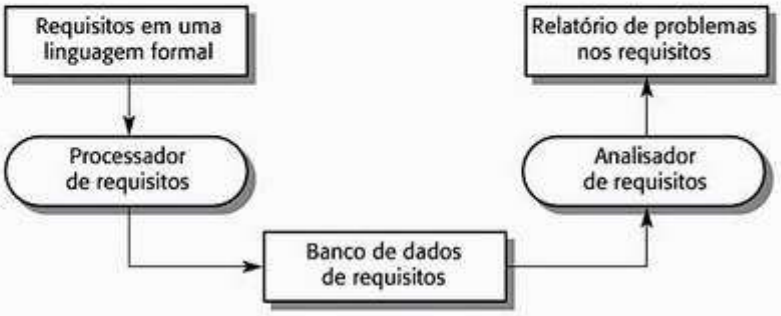
Durante o processo de validação de requisitos, diferentes tipos de verificação devem ser realizados sobre os requisitos no documento de requisitos. Dentre as verificações destacam-se:

1. *Verificações de validade* Um usuário pode pensar que um sistema é necessário para realizar certas funções. Contudo, mais estudos e análises podem identificar funções adicionais ou diferentes, que são exigidas. Os sistemas têm diversos usuários com necessidades diferentes e qualquer conjunto de requisitos é inevitavelmente uma solução conciliatória da comunidade de usuários.
2. *Verificações de consistência* Os requisitos em um documento não devem ser conflitantes, ou seja, não devem existir restrições contraditórias ou descrições diferentes para uma mesma função do sistema.
3. *Verificações de completeza* O documento de requisitos deve incluir requisitos que definam todas as funções e restrições exigidas pelo usuário do sistema.
4. *Verificações de realismo* Utilizando o conhecimento da tecnologia existente, os requisitos devem ser verificados, a fim de assegurar que eles realmente podem ser implementados. Essas verificações devem também levar em conta o orçamento e os prazos para o desenvolvimento do sistema.
5. *Facilidade de verificação* Para reduzir o potencial de divergências entre cliente e fornecedor, os requisitos do sistema devem sempre ser escritos de modo que possam ser verificados. Isso significa que um conjunto de verificações pode ser projetado para mostrar que o sistema entregue cumpre com esses requisitos.

Existe uma série de técnicas de validação de requisitos que podem ser utilizadas em conjunto ou individualmente:

1. *Revisões de requisitos* Os requisitos são analisados sistematicamente por uma equipe de revisores. Esse processo será discutido na seção a seguir.
2. *Prototipação* Nessa abordagem de validação, um modelo executável do sistema é mostrado aos usuários finais e clientes. Eles podem experimentar o modelo para verificar se ele atende às suas necessidades. Discutirei as técnicas de prototipação no Capítulo 8.
3. *Geração de casos de teste* Como modelo ideal, os requisitos deveriam ser testáveis. Se os testes para os requisitos são criados como parte do processo de validação, isso, muitas vezes, revela problemas com os requisitos. Se um teste é difícil ou impossível de ser projetado, isso freqüentemente significa que os requisitos serão de difícil implementação e devem ser reconsiderados.
4. *Análise automatizada da consistência* Se os requisitos são expressos como um modelo de sistema em uma notação estruturada ou formal, então as ferramentas CASE podem ser utilizadas para verificar a consistência do modelo. Isso está ilustrado na Figura 6.15. Para verificar a consistência, a ferramenta CASE deve construir uma base de dados de requisitos e, então, utilizando-se as regras do método ou da notação, verificar todos os requisitos na base de dados. Uma análise de requisitos produz um relatório das inconsistências que foram descobertas.

Figura 6.15 ▶
Verificação automatizada da consistência dos requisitos.



As dificuldades da validação de requisitos não devem ser subestimadas. É difícil demonstrar que um conjunto de requisitos atende às necessidades de um usuário. Os usuários devem pensar no sistema em operação e imaginar como esse sistema se adequaria ao seu trabalho. Não é fácil para profissionais habilitados de computação conseguir realizar esse tipo de análise abstrata, o que é ainda mais difícil para os usuários de sistema. Como resultado, a validação de requisitos raramente descobre todos os problemas com os requisitos, e as modificações para corrigir omissões e falhas de compreensão, depois que o documento de requisitos foi aceito, são inevitáveis.

6.3.1 Revisões de requisitos

Uma revisão de requisitos é um processo manual, que envolve muitos leitores, tanto do pessoal do cliente como do fornecedor, que verifica o documento de requisitos a fim de detectar anomalias ou omissões. O processo de revisão pode ser gerenciado da mesma maneira que as inspeções de programa (veja o Capítulo 19). Como alternativa, esse processo pode ser organizado em maior escala, com muitos participantes envolvidos na verificação de diferentes partes do documento.

As revisões de requisitos podem ser informais ou formais. As revisões informais simplesmente envolvem os fornecedores que discutem os requisitos com tantos *stakeholders* quantos forem possíveis. É surpreendente como muitas vezes a comunicação entre desenvolvedores e *stakeholders* termina depois da obtenção de requisitos, e não existe nenhuma confirmação de que os requisitos documentados são os que os *stakeholders* realmente solicitaram. Muitos problemas podem ser detectados simplesmente conversando sobre o sistema com os *stakeholders*, antes de iniciar uma revisão formal.

Em uma revisão formal de requisitos, a equipe de desenvolvimento deve 'conduzir' o cliente pelos requisitos de sistema, explicando as implicações de cada um. A equipe de revisão deve verificar cada requisito, em termos de sua consistência, e checar os requisitos como um todo, sob o ponto de vista de sua completeza. Os revisores podem também checar:

1. *Facilidade de verificação* O requisito é realmente passível de ser testado, como foi definido?
2. *Facilidade de compreensão* O requisito pode ser adequadamente compreendido pelos compradores ou usuários finais do sistema?
3. *Facilidade de rastreamento* A origem do requisito é claramente definida? Pode ser preciso retornar à origem do requisito, a fim de avaliar o impacto de uma mudança. A facilidade de rastreamento é importante porque permite avaliar o impacto de uma mudança no restante do sistema. Isso será discutido com mais detalhes na próxima seção.
4. *Adaptabilidade* O requisito é adaptável? Em outras palavras, ele pode ser modificado, sem que isso provoque efeitos em grande escala em outros requisitos do sistema?

Conflitos, contradições, erros e omissões nos requisitos devem ser destacados durante a revisão e formalmente registrados. Fica, então, por conta dos usuários, do comprador do sistema e dos desenvolvedores do sistema negociar a solução para esses problemas identificados.

6.4 Gerenciamento de requisitos

Os requisitos para grandes sistemas de software estão sempre sendo modificados. Uma das razões para isso é que esses sistemas são geralmente desenvolvidos para lidar com problemas 'intrincados' (como foi discutido no Capítulo 2). Como o problema não pode ser intei-

ramente definido, os requisitos de sistema são necessariamente incompletos. Durante o processo de software, a compreensão dos desenvolvedores sobre o problema está constantemente se modificando, e essas mudanças se refletem nos requisitos.

Além disso, grandes sistemas de software são geralmente necessários para melhorar o status atual. O sistema existente pode ser um sistema manual ou um sistema de computador desatualizado. Embora as dificuldades com o sistema atual possam ser conhecidas, é difícil prever que efeitos o sistema 'aperfeiçoado' terá sobre a organização. Depois que os usuários finais se familiarizam com um sistema, novos requisitos surgem, pelas seguintes razões:

1. Grandes sistemas geralmente têm uma comunidade de usuários diversificada. Diferentes usuários têm diferentes requisitos e prioridades, que podem ser conflitantes ou contraditórios. Os requisitos finais do sistema são, inevitavelmente, uma conciliação entre eles; e, com a experiência, muitas vezes é constatado que o equilíbrio do apoio dado a diferentes usuários precisa ser mudado.

2. O pessoal que paga por um sistema e os usuários desse sistema raramente são as mesmas pessoas. Os clientes do sistema impõem requisitos em razão de restrições organizacionais e orçamentárias, e esses requisitos podem ser conflitantes com os requisitos dos usuários finais.

3. A empresa e o ambiente técnico do sistema se modificam, e isso tem de ser refletido no próprio sistema. Um novo hardware pode ser implementado, pode ser necessário fazer a interface do sistema com outros sistemas, as prioridades da empresa podem se modificar, acarretando conseqüentes mudanças no suporte necessário ao sistema, e novas legislações e regulamentos podem ser criados, e estes devem ser implementados pelo sistema. Os requisitos não funcionais são, particularmente, afetados por mudanças na tecnologia de hardware.

O gerenciamento de requisitos é o processo de compreender e controlar as mudanças nos requisitos de sistemas. O processo de gerenciamento de requisitos é realizado em conjunto com outros processos da engenharia de requisitos. O planejamento se inicia ao mesmo tempo que o levantamento inicial de requisitos, e o gerenciamento ativo dos requisitos deve começar assim que um esboço da versão do documento de requisitos estiver disponível.

Discuto a seguir cada uma dessas atividades. Todavia, antes de passar para essa descrição, discutirei por que os requisitos inevitavelmente se modificam e explico por que alguns tipos de requisitos são mais suscetíveis de serem modificados do que outros.

6.4.1 Requisitos permanentes e voláteis

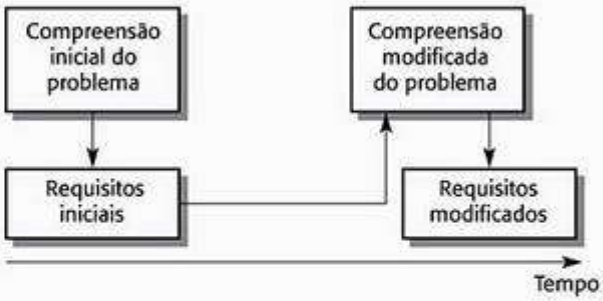
O desenvolvimento de requisitos de software focaliza a atenção nos recursos do software, nos objetivos da empresa e em outros sistemas da empresa. Enquanto a definição de requisitos é desenvolvida, uma melhor compreensão das necessidades dos usuários é alcançada. Isso fornece informações para o usuário, causando modificações nos requisitos (Figura 6.16). Pode levar vários anos para se especificar e desenvolver um sistema de grande porte. Ao longo desse tempo, o ambiente do sistema e os objetivos da empresa certamente deverão ser modificados. Os requisitos devem, portanto, evoluir, a fim de refletir essas mudanças.

A partir de uma perspectiva de evolução, os requisitos são divididos em duas classes:

1. *Requisitos permanentes* São os requisitos relativamente estáveis, que derivam da atividade principal da organização e que se relacionam diretamente com o domínio do sistema. Por exemplo, em um hospital, sempre haverá requisitos relativos aos pacientes, aos médicos, às enfermeiras e aos tratamentos. Esses requisitos podem ser derivados dos modelos de domínio, que mostram as entidades e os relacionamentos que caracterizam um domínio de aplicação (Prieto-Díaz e Arango, 1991; Easterbrook, 1993).

Figura 6.16

Evolução de requisitos.



2. *Requisitos voláteis* São os requisitos que provavelmente vão se modificar durante o desenvolvimento do sistema ou depois que o sistema estiver em operação. Exemplos de requisitos voláteis são aqueles resultantes de políticas governamentais sobre assistência médica.

Harker *et al.* (1993) sugeriram que os requisitos voláteis são divididos em cinco classes. Contudo, acredito que duas dessas classes estão relacionadas e prefiro, então, uma classificação como a que mostra a Figura 6.17.

6.4.2 Planejamento do gerenciamento de requisitos

Planejar é o primeiro estágio essencial ao processo de gerenciamento de requisitos, que é muito dispendioso; e para cada projeto, o estágio de planejamento estabelece o nível de detalhes exigido para o gerenciamento de requisitos. Durante o estágio de gerenciamento de requisitos, temos de decidir sobre os seguintes aspectos:

1. *Identificação de requisitos* Cada requisito precisa ser identificado de modo único, para que possa ser feita a referência cruzada deste com os outros requisitos e para que ele possa ser utilizado nas avaliações de facilidade de rastreamento.
2. *Processo de gerenciamento de mudanças* Trata-se do conjunto de atividades que avalia o impacto e o custo das mudanças. Discutiremos isso com mais detalhes na próxima seção.

Figura 6.17

Classificação dos requisitos voláteis.

Tipo de requisito	Descrição
Requisitos mutáveis	Requisitos que se modificam por causa das mudanças do ambiente no qual a organização está operando. Por exemplo, em sistemas de hospitais, o financiamento do tratamento de pacientes pode se modificar e, assim, exigir que diferentes informações sobre o tratamento sejam coletadas.
Requisitos emergentes	Requisitos que surgem à medida que a compreensão do cliente do sistema se desenvolve durante o desenvolvimento do sistema. O processo de projeto pode revelar novos requisitos emergentes.
Requisitos conseqüentes	Requisitos que resultam da introdução do sistema de computação. A introdução do sistema de computação pode modificar os processos da organização e criar novos meios de trabalho, que geram novos requisitos de sistema.
Requisitos de compatibilidade	Requisitos que dependem de sistemas ou processos de negócios específicos dentro da organização. À medida que eles se modificam, os requisitos de compatibilidade no sistema encomendado ou entregue podem também ter de evoluir.

3. *Políticas de facilidade de rastreamento* Essas políticas definem as relações entre os requisitos e entre os requisitos e o projeto de sistema que devem ser registrados e também como esses registros devem ser mantidos.

4. *Suporte de ferramentas CASE* O gerenciamento de requisitos envolve processar uma grande quantidade de informações sobre os requisitos. As ferramentas que podem ser utilizadas vão desde sistemas especializados de gerenciamento de requisitos até planilhas de cálculo e sistemas simples de bancos de dados.

Existem muitas relações entre requisitos e outros requisitos e entre os requisitos e o projeto do sistema. Há também elos entre os requisitos e as razões básicas da proposição desses requisitos. Quando são propostas modificações, é preciso verificar o impacto dessas mudanças sobre outros requisitos e o projeto de sistema. A facilidade de rastreamento é uma propriedade geral de uma especificação de requisito, que reflete a facilidade de se encontrar requisitos relacionados.

Existem três tipos de informações sobre a facilidade de rastreamento, que podem ser mantidas:

1. *Informações sobre a facilidade de rastreamento da origem* vinculam os requisitos aos *stakeholders* que propuseram esses requisitos. Quando uma mudança é proposta, essa informação é utilizada para descobrir os *stakeholders*, de modo que eles possam ser consultados sobre a mudança.

2. *Informações sobre a facilidade de rastreamento de requisitos* vinculam requisitos dependentes dentro de seu respectivo documento. Essa informação é utilizada para avaliar quantos requisitos provavelmente serão afetados por uma mudança proposta e a extensão das mudanças conseqüentes nos requisitos, que podem ser necessárias

3. *Informações sobre a facilidade de rastreamento de projeto* vinculam os requisitos aos módulos de projeto em que esses requisitos são implementados. Essa informação é utilizada para avaliar o impacto das mudanças nos requisitos, as propostas para o projeto e a implementação do sistema.

As informações sobre a facilidade de rastreamento são, freqüentemente, representadas com o uso de matrizes de facilidade de rastreamento, que relacionam os requisitos aos *stakeholders*, os requisitos entre si ou aos módulos de projeto. Se considerarmos as matrizes de facilidade de rastreamento que vinculam os requisitos a outros requisitos, cada requisito será representado por uma linha e por uma coluna na matriz. Onde existe uma dependência entre requisitos, ela é registrada na célula, na intersecção linha/coluna.

Figura 6.18 ▶

Uma matriz de facilidade de rastreamento.

ID do requisito	1.1	1.2	1.3	2.1	2.2	2.3	3.1	3.2
1.1		U	R					
1.2			U			R		U
1.3	R			R				
2.1			R		U			U
2.2								U
2.3		R		U				
3.1								R
3.2							R	

Isso está ilustrado na Figura 6.18, que mostra uma matriz simples de facilidade de rastreamento, que registra as dependências entre os requisitos. Um U na intersecção linha/coluna ilustra que o requisito na linha utiliza os recursos especificados no requisito nomeado na coluna; um R significa que existe uma relação fraca entre os requisitos. Por exemplo, eles podem ambos definir os requisitos para partes do mesmo subsistema.

As matrizes de facilidade de rastreamento podem ser utilizadas quando um pequeno número de requisitos precisa ser gerenciado, mas elas se tornam muito difíceis de serem manuseadas e são de manutenção dispendiosa em grandes sistemas com muitos requisitos. Para esses sistemas, temos de obter as informações da facilidade de rastreamento em um banco de dados de requisitos, em que cada requisito é explicitamente vinculado a requisitos relacionados. O impacto das mudanças pode, então, ser avaliado pelo uso dos recursos de visualização dos bancos de dados. Como alternativa, pode ser possível gerar as matrizes de facilidade de rastreamento automaticamente.

O gerenciamento de requisitos precisa de algum apoio automatizado e as ferramentas CASE utilizadas devem ser escolhidas durante a fase de planejamento. O apoio de ferramentas é necessário para:

1. *Armazenamento de requisitos* Os requisitos devem ser mantidos em um depósito de dados seguro, gerenciado, que seja acessível por todos os envolvidos no processo de engenharia de requisitos.
2. *Gerenciamento de mudanças* Esse processo (Figura 6.19) é simplificado se o apoio de ferramentas ativas estiver disponível.
3. *Gerenciamento de facilidade de rastreamento* Como foi discutido antes, o apoio de ferramentas para a rastreabilidade permite que sejam descobertos requisitos relacionados. Estão disponíveis algumas ferramentas que utilizam técnicas de processamento de linguagem natural, para ajudar a descobrir as possíveis relações entre os requisitos.

Para sistemas pequenos, pode não ser necessário utilizar ferramentas especializadas de gerenciamento de requisitos. Esse processo pode ter apoio utilizando os recursos disponíveis em processadores de texto, planilhas de cálculo e bancos de dados de PCs. Contudo, para sistemas maiores, o apoio de ferramentas mais especializadas é necessário. Inclui links às informações sobre as ferramentas de gerenciamento de requisitos, como DOORS e 'Requisite Pro', nas páginas Web referentes ao livro.

6.4.3 Gerenciamento de mudanças de requisitos

O gerenciamento de mudanças de requisitos (Figura 6.19) deve ser aplicado a todas as mudanças propostas para os requisitos. A vantagem de utilizar um processo formal para o gerenciamento de mudanças é que todas as propostas de mudança são tratadas de modo consistente e que as mudanças no documento de requisitos são feitas de maneira controlada. Há três estágios principais em um processo de gerenciamento de mudanças:

1. *Análise do problema e especificação da mudança* O processo começa com a identificação de um problema com os requisitos ou, algumas vezes, com uma proposta específica de mudança. Nesse estágio, é realizada a análise do problema ou da proposta de mudança, a fim de verificar sua validade. Uma proposta mais específica de mudança nos requisitos pode então ser efetuada.
2. *Análise e custo da mudança* O efeito da mudança proposta é avaliado, utilizando-se informações sobre a facilidade de rastreamento e o conhecimento geral dos requisitos do sistema. O custo da mudança é estimado em termos das modificações no documento de requisitos e, se apropriado, no projeto de sistemas e na implementação. Uma vez concluída essa análise, é tomada uma decisão sobre prosseguir com a alteração de requisitos ou não.

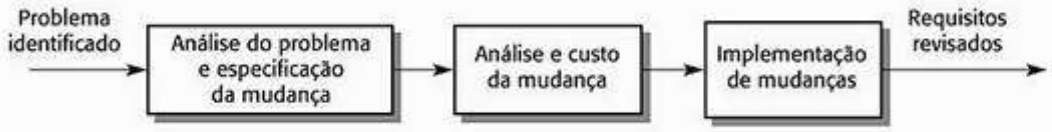


Figura 6.19 ▲ Gerenciamento de mudanças de requisitos.

3. *Implementação de mudanças* O documento de requisitos e, quando for necessário, o projeto de sistema e a implementação são modificados. O documento de requisitos deve ser organizado de maneira que as mudanças possam ser acomodadas sem muito esforço. Assim como ocorre com os programas, a facilidade de modificações em documentos é alcançada ao se minimizar referências externas e ao tornar as seções do documento tão modulares quanto possível.

Se uma mudança nos requisitos de um sistema for requerida urgentemente, sempre existe a tentação de fazer essa mudança no sistema e, depois, retrospectivamente modificar o documento de requisitos. Isso quase inevitavelmente faz com que a especificação de requisitos e a implementação do sistema se desajustem. Uma vez feitas as mudanças no sistema, as modificações no documento de requisitos podem ser esquecidas ou feitas de uma maneira não consistente com as mudanças no sistema.

PONTOS-CHAVE

- O processo de engenharia de requisitos inclui um estudo de viabilidade, o levantamento e a análise de requisitos, a especificação de requisitos, a validação de requisitos e o gerenciamento de requisitos.
- A análise de requisitos é um processo iterativo, que envolve a compreensão do domínio, assim como a coleta, a classificação, a estruturação, a priorização e a validação dos requisitos.
- Diferentes *stakeholders* do sistema têm diferentes requisitos. Todos os sistemas complexos devem, portanto, ser analisados a partir de uma série de diferentes pontos de vista. Os pontos de vista podem ser fontes ou 'drenos' de dados; diferentes representações ou entidades do sistema estão fora do sistema e recebem serviços a partir dele.
- Os fatores sociais e organizacionais têm forte influência sobre os requisitos do sistema e podem determinar se o software será realmente utilizado ou não.
- A validação dos requisitos é o processo de verificar os requisitos quanto a sua validade, consistência, completeza, seu realismo e sua facilidade de verificação. As revisões de requisitos e a prototipação são as principais técnicas utilizadas para a validação de requisitos.
- As modificações organizacionais, técnicas e de negócios inevitavelmente levam a mudanças nos requisitos, em um sistema de software. O gerenciamento de requisitos é o processo de gerenciar e controlar essas mudanças.
- O processo de gerenciamento de requisitos inclui o planejamento do gerenciamento, em que são especificados os procedimentos e as políticas para o gerenciamento de requisitos, e o gerenciamento de mudanças, em que as mudanças são analisadas e seu impacto é avaliado.

LEITURAS SUGERIDAS

Mastering the requirements process. Um livro de fácil leitura, dirigido a engenheiros de requisitos com alguma prática. Ele fornece orientações específicas sobre o desenvolvimento de um efetivo processo de engenharia de requisitos. (S. Robertson e J. Robertson, 1999, Addison Wesley Longman.)

Requirements engineering: processes and techniques. Esse livro inclui uma visão mais detalhada das atividades do processo de engenharia de requisitos e discute o método VORD e sua aplicação. (G. Kotonya e I. Sommerville, 1998, John Wiley and Sons.)

Software requirements: objects, functions and states. Esse livro tem um capítulo razoável sobre a análise em geral e inclui um bom levantamento de métodos de análise de requisitos. Contudo, não fala sobre abordagens da análise com base em múltiplos pontos de vista. (A.M. Davis, 1993, Prentice Hall.)

Exploring requirements: quality before design. Um livro excelente e muito fácil de ler, que oferece conselhos práticos sobre a realidade da engenharia de requisitos. (D.C. Gause e G.M. Weinberg, 1989, Dorset House.)

EXERCÍCIOS

6.1 Sugira quem podem ser os *stakeholders* em um sistema de registro de estudantes de uma universidade. Explique por que é quase sempre inevitável que os requisitos de diferentes *stakeholders* sejam, de alguma maneira, conflitantes.

6.2 Um sistema de software deve ser desenvolvido para automatizar um catálogo de biblioteca. Esse sistema conterá informações sobre todos os livros da biblioteca e será utilizado por seus funcionários, leitores e pessoal que empresta livros da biblioteca. O sistema deverá permitir 'navegar' no catálogo e fazer consultas, além de fornecer recursos que possibilitem aos usuários enviar mensagens ao pessoal da biblioteca, solicitando a reserva de livros que estão emprestados. Identifique os principais pontos de vista que devem ser levados em consideração na especificação desse sistema e mostre suas relações, utilizando um diagrama de hierarquia de pontos de vista.

6.3 Para três dos pontos de vista identificados no sistema de catálogo da biblioteca, sugira os serviços que podem ser fornecidos para cada ponto de vista, os dados que o ponto de vista pode fornecer e os eventos que controlam o fornecimento desses serviços.

6.4 Para os serviços identificados no Exercício 6.3, especifique quais poderiam ser as restrições não funcionais mais importantes.

6.5 Empregando seu próprio conhecimento de como um ATM é utilizado, desenvolva um conjunto de use-cases que possa ser utilizado para derivar os requisitos para um sistema de ATM.

6.6 Discuta o exemplo de um tipo de sistema em que os fatores sociais e políticos podem influenciar muito os requisitos do sistema. Explique por que esses fatores são importantes em seu exemplo.

6.7 Quem deve estar envolvido em uma revisão de requisitos? Desenhe um modelo de processo mostrando como uma revisão de requisitos pode ser organizada.

6.8 Por que as matrizes de facilidade de rastreamento se tornam difíceis de gerenciar quando existem muitos requisitos de sistema? Projete um mecanismo de estruturação de requisitos, com base em pontos de vista, que possa ajudar a reduzir a escala desse problema.

6.9 Quando mudanças de emergência têm de ser feitas em sistemas, o software de sistema pode precisar ser modificado, antes que as mudanças nos requisitos tenham sido aprovadas. Sugira um modelo de um processo para fazer essas modificações, que assegure que o documento de requisitos e a implementação do sistema não se tornem inconsistentes.

6.10 Sua empresa utiliza um método de análise-padrão que é normalmente aplicado em todas as análises de requisitos. Em seu trabalho, você acha que esse método não pode representar fatores sociais que sejam significativos no sistema que está sendo analisado. Você expõe isso a seu gerente, que deixa claro que o padrão deve ser seguido. Discuta o que você deveria fazer nessa situação.

7 Modelos de sistema

Objetivos

O propósito deste capítulo é introduzir uma série de diferentes tipos de modelos de sistema, os quais podem ser desenvolvidos durante o processo de engenharia de requisitos. Depois de ler este capítulo, você poderá:

- compreender por que é importante modelar o contexto de um sistema;
- compreender os conceitos de modelagem de comportamento, modelagem de dados e modelagem de objeto;
- conhecer algumas das notações definidas na UML (*unified modeling language* – linguagem de modelagem unificada) e como essas notações podem ser utilizadas para desenvolver diferentes tipos de modelos de sistema, e
- compreender como as ferramentas CASE em *workbenches* apóiam a modelagem de sistemas.

Sumário

7.1 Modelos de contexto
7.2 Modelos de comportamento
7.3 Modelos de dados
7.4 Modelos de objeto
7.5 Ferramentas CASE em *workbenches*

Os requisitos de usuário devem ser escritos em linguagem natural porque precisam ser compreendidos por pessoas que não são peritos técnicos. Contudo, requisitos de sistema mais detalhados podem ser expressos de maneira mais técnica. Uma técnica amplamente utilizada é documentar a especificação do sistema como um conjunto de modelos de sistema. Esses modelos são representações gráficas que descrevem o problema a ser resolvido e o sistema a ser desenvolvido. Devido às representações gráficas utilizadas, os modelos são freqüentemente mais compreensíveis do que as descrições detalhadas em linguagem natural dos requisitos de sistema. Eles são também uma importante ponte entre o processo de análise e de projeto.

Os modelos podem ser utilizados no processo de análise para desenvolver uma compreensão do sistema existente a ser substituído ou melhorado ou para especificar o sistema requerido. Eles podem ser utilizados para representar o sistema, a partir de diferentes perspectivas:

1. uma perspectiva externa, na qual o contexto ou o ambiente do sistema é modelado;
2. uma perspectiva de comportamento, em que o comportamento do sistema é modelado, e
3. uma perspectiva estrutural, em que a arquitetura do sistema ou a estrutura dos dados processados pelo sistema é modelada.

Abordamos essas três perspectivas neste capítulo e também discutimos a modelagem de objetos, que combina, até certo ponto, as modelagens de comportamento e estrutural.

Os métodos estruturados, como a análise estruturada de sistema (DeMarco, 1978) e a análise orientada a objetos (Rumbaugh *et al.*, 1991; Booch, 1994), fornecem um *framework* para a modelagem detalhada de sistemas, como parte da obtenção e análise de requisitos. A maioria dos métodos estruturados tem seu próprio conjunto preferido de modelos de sistema. Eles geralmente definem um processo que pode ser utilizado para derivar esses modelos e um conjunto de regras e diretrizes que se aplicam aos modelos. A documentação padronizada é produzida para o sistema. As ferramentas CASE, discutidas na Seção 7.5, geralmente estão disponíveis para apoio aos métodos. Entre essas ferramentas estão os editores de modelos e documentação automatizada de sistemas, que fornecem alguns recursos de verificação de modelo.

Contudo, os métodos de análise estruturada apresentam uma série de deficiências:

1. Eles não fornecem apoio efetivo para a compreensão ou modelagem de requisitos de sistemas não funcionais.
2. Eles são confusos por não incluírem de modo geral diretrizes que ajudem os usuários a decidir se um método é apropriado ou não para um problema específico nem incluem normalmente alguma recomendação sobre como podem ser adaptados para uso em um ambiente específico.
3. Freqüentemente, eles produzem muita documentação. A essência dos requisitos de sistema pode ficar oculta pelo volume de detalhes que são incluídos.
4. Os modelos produzidos são muito detalhados e os usuários freqüentemente acham que eles são de difícil compreensão. Portanto, esses usuários, na verdade, não podem verificar o realismo desses modelos.

Na prática, os engenheiros de requisitos não precisam se restringir aos modelos propostos em qualquer método específico. Por exemplo, os métodos orientados a objetos não sugerem, geralmente, que os modelos de fluxo de dados devem ser desenvolvidos. Contudo, de acordo com minha experiência, esses modelos são úteis, algumas vezes, como parte de um processo de análise orientado a objetos. Freqüentemente, eles refletem o entendimento do sistema pelos usuários finais e também podem contribuir diretamente para a identificação do objeto (os dados que fluem) e a identificação de operações nesses objetos.

O aspecto mais importante de um modelo de sistemas é que ele omite detalhes. Um modelo de sistema é uma abstração do sistema em estudo, e não de uma representação alternativa desse sistema. Como ideal, a *representação* de um sistema deve manter todas as informações a respeito da entidade que está sendo representada. Uma *abstração* deliberadamente

simplifica e realça as características mais importantes. Por exemplo, considerando o improvável evento de este livro ser resumido em *Seleções* do *Reader's Digest*, a apresentação, nesse caso, seria uma abstração dos pontos mais importantes. Essa tradução do inglês para o português seria uma representação alternativa. A intenção do tradutor é manter todas as informações como elas são apresentadas em inglês.

Diferentes tipos de modelos de sistema têm como base diferentes abordagens de abstração. Um modelo de fluxo de dados, por exemplo, se concentra no fluxo de dados e nas transformações funcionais nesses dados, excluindo detalhes de suas estruturas. Por outro lado, um modelo de relacionamento de entidades tem o propósito de documentar os dados do sistema e seus relacionamentos, sem se preocupar com as funções do sistema.

Entre os exemplos dos diferentes tipos de modelo de sistema que podem ser produzidos durante o processo de análise estão:

1. *Modelo de processamento de dados* Os diagramas de fluxo de dados mostram como os dados são processados em diferentes estágios no sistema.
2. *Modelo de composição* Os diagramas de relacionamento de entidades mostram como as entidades no sistema são compostas de outras entidades.
3. *Modelo de arquitetura* Os modelos de arquitetura mostram os subsistemas principais que constituem um sistema.
4. *Modelo de classificação* Os diagramas de classe/herança mostram como as entidades têm características comuns.
5. *Modelo de estímulo-resposta* Os diagramas de transição de estado mostram como o sistema reage a eventos externos e internos.

Todos esses tipos de modelos são abordados neste capítulo. Sempre que possível, utilizo notações em UML, que está emergindo como uma linguagem padrão de modelagem, particularmente para a modelagem orientada a objetos (Booch *et al.*, 1999; Rumbaugh *et al.*, 1999a). Quando a UML não inclui uma notação apropriada, utilizo notações intuitivas simples para a descrição do modelo.

7.1 Modelos de contexto

Em um estágio inicial do processo de obtenção e análise de requisitos, é preciso decidir sobre os limites do sistema. Isso envolve trabalhar com os *stakeholders* do sistema para distinguir o que é o sistema e qual é seu ambiente. Essas decisões devem ser tomadas logo no início do processo, a fim de limitar os custos do sistema e o tempo necessário para a análise.

Em alguns casos, o limite entre um sistema e seu ambiente é relativamente claro. Por exemplo, quando um sistema automatizado está substituindo um sistema computadorizado ou manual já existente, o ambiente do novo sistema geralmente é o mesmo que o do sistema existente. Em outros casos, existe maior flexibilidade. Você decide sobre o limite entre o sistema e seu ambiente, durante o processo de engenharia de requisitos.

Por exemplo, o ambiente de um conjunto de ferramentas CASE pode incluir um banco de dados já existente, cujos serviços são utilizados pelo sistema, ou o conjunto de ferramentas pode definir seu próprio banco de dados interno. Quando um banco de dados já existe, o posicionamento do limite entre esses sistemas pode ser um difícil problema gerencial e técnico; somente é possível tomar uma decisão sobre o que é e o que não é parte do sistema, depois de ser feita alguma análise.

A definição do contexto de um sistema não é um julgamento sem valor. As preocupações sociais e organizacionais podem significar que a posição de um limite de sistema pode ser determinada por fatores que não são técnicos. Por exemplo, um limite do sistema pode ser posicio-

nado de modo que o processo de análise seja totalmente realizado em um local, pode ser escolhido de maneira que um gerente particularmente difícil não precise ser consultado e pode ser posicionado de modo que o custo do sistema termine sofrendo um aumento, e a divisão de desenvolvimento de sistemas, portanto, tenha de ser expandida para projetar e implementar o sistema.

Uma vez tomadas algumas decisões sobre os limites do sistema, parte da atividade de análise é a definição desse contexto e das dependências que um sistema tem em seu ambiente. Normalmente, produzir um modelo simples de arquitetura é o primeiro passo nessa atividade.

Isso está ilustrado na Figura 7.1. Trata-se de um modelo de arquitetura que mostra a estrutura do sistema de informações que inclui uma rede de caixas automáticos de banco (ATMs). Os modelos de arquitetura de alto nível são geralmente expressos como simples diagramas de blocos, em que cada subsistema é representado por um retângulo com um nome e as linhas indicam que existem algumas associações entre os subsistemas.

Na Figura 7.1, vemos que cada máquina de caixa automático está conectada a um banco de dados de contas, a um sistema de contabilidade da agência local, a um sistema de proteção e a um sistema de apoio à manutenção da máquina. O sistema também está conectado a um banco de dados de operação que monitora como a rede de ATMs é utilizada e a um sistema de balcão da agência local. Esse sistema de balcão fornece serviços como backup e impressão. Esses serviços precisam, portanto, ser incluídos no próprio caixa automático.

Os modelos de arquitetura descrevem o ambiente de um sistema. Contudo, eles não mostram a relação entre os outros sistemas no ambiente e o sistema que está sendo especificado. Os sistemas externos podem produzir dados para o sistema ou consumir dados a partir dele. Eles podem compartilhar dados com o sistema e ser conectados diretamente, por meio de uma rede, ou não serem conectados de maneira alguma. Eles podem estar fisicamente localizados em um mesmo lugar ou em prédios separados. Todas essas relações podem afetar os requisitos do sistema em definição e devem ser levadas em consideração.

Portanto, os modelos simples de arquitetura são normalmente complementados por outros modelos, como os modelos de processo, que mostram as atividades de processo apoiadas pelo sistema, e os modelos de fluxo de dados (descritos na seção seguinte), que mostram como os dados são transferidos entre o sistema e outros sistemas em seu ambiente.

A Figura 7.2 ilustra um modelo de processo para o suprimento de equipamentos em uma organização, que envolve especificar o equipamento requerido, procurar e escolher fornecedores, fazer o pedido de compra do equipamento, recebê-lo e testá-lo depois da entrega. Ao especificar o suporte técnico de computador para este processo, é preciso decidir quais dessas atividades serão realmente realizadas. As outras atividades estão fora do limite do sistema. Na Figura 7.2, a linha pontilhada agrupa as atividades que estão dentro do limite do sistema.

Figura 7.1 ▶

Contexto do sistema de ATM.

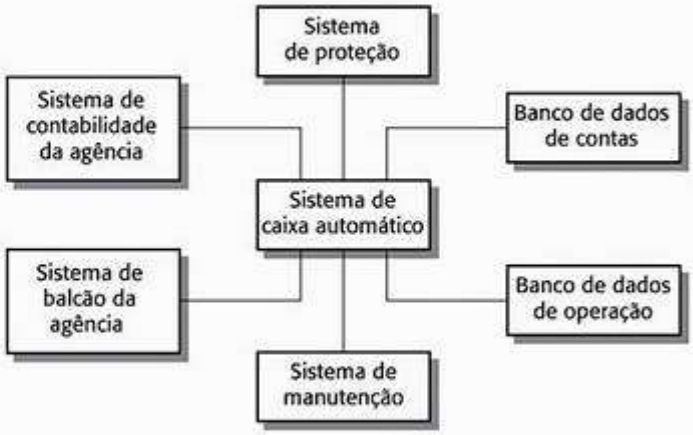
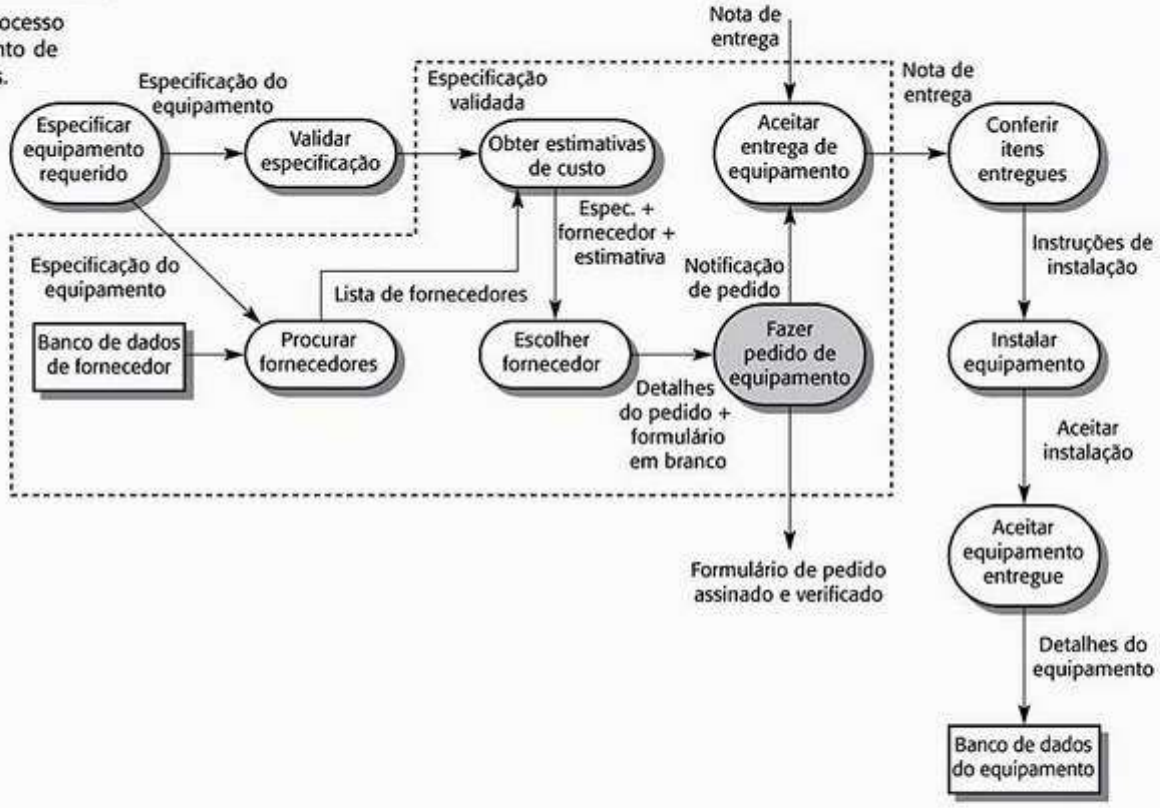


Figura 7.2

Modelo de processo para suprimento de equipamentos.



7.2 Modelos de comportamento

Os modelos de comportamento são utilizados para descrever o comportamento geral do sistema. Discuto dois tipos de modelos de comportamento, que são: os modelos de fluxo de dados, que mostram o processamento de dados no sistema, e os modelos de máquinas de estado, que mostram como o sistema reage a eventos. Esses modelos podem ser utilizados separadamente ou em conjunto, dependendo do tipo de sistema que está sendo desenvolvido.

A maioria dos sistemas de negócios é principalmente direcionada a dados. Eles são controlados pelas entradas de dados no sistema, e relativamente com pouco processamento de eventos externos. Um modelo de fluxo de dados pode ser tudo o que é necessário para representar o comportamento desses sistemas. Por outro lado, os sistemas de tempo real são, freqüentemente, dirigidos a eventos, com um mínimo de processamento de dados. Um modelo de máquina de estado, discutido na Seção 7.2.2, é o modo mais eficaz de representar seu comportamento. Outras classes de sistema podem ser tanto dirigidas a dados como também a eventos e, assim, ambos os tipos de modelo devem ser desenvolvidos.

7.2.1 Modelos de fluxo de dados

Os modelos de fluxo de dados são uma maneira intuitiva de mostrar como os dados são processados por um sistema. No que se refere à análise, eles devem ser utilizados para modelar o modo como os dados são processados no sistema existente. A notação utilizada nesses modelos representa o processamento funcional, o repositório de dados e os movimentos de dados entre funções. O uso de modelos de fluxo de dados para análise se tornou comum depois

da publicação do livro de DeMarco (1978) sobre a análise estruturada de sistemas. Esses modelos são uma parte intrínseca dos métodos desenvolvidos a partir desse livro.

Os modelos de fluxo de dados são utilizados para mostrar como os dados fluem por meio de uma seqüência de etapas de processamento. Os dados são transformados a cada etapa, antes de seguirem para o próximo estágio. Essas etapas de processamento, ou transformações, são funções de programa quando diagramas de fluxo de dados são utilizados para documentar um projeto de software. Contudo, em um modelo de análise, o processamento pode ser realizado por pessoas ou computadores.

Um modelo de fluxo de dados está ilustrado na Figura 7.3, que mostra as etapas envolvidas no processamento de um pedido de compra de produtos (como um computador) em uma organização. Esse modelo específico descreve o processamento de dados na atividade 'Colocar pedido de equipamento' no modelo de processo geral, mostrado na Figura 7.2. Ele mostra como o pedido de compra de produtos 'se move' de processo para processo e mostra também os repositórios de dados envolvidos no processo.

Na notação que utilizo para descrever diagramas de fluxo de dados, na Figura 7.3, os retângulos com cantos arredondados representam etapas de processamento, as setas rotuladas com o nome do dado representam fluxos e os retângulos representam repositórios de dados ou fontes de dados.

Os modelos de fluxo de dados são valiosos, porque acompanhar e documentar como os dados associados com um processo em particular se movimentam pelo sistema ajuda o analista a compreender o que está acontecendo. Os diagramas de fluxo de dados têm a vantagem de serem simples e intuitivos, diferentemente de outras notações de modelagem. Muitas vezes, é possível explicá-los a usuários de sistema em potencial, os quais podem, portanto, participar da validação da análise.

Em princípio, o desenvolvimento de modelos, como modelos de fluxo de dados, deve ser um processo top-down (de cima para baixo). Nesse exemplo, isso implica que o processo geral de aquisição deve ser analisado, primeiramente. A análise de subprocessos, como fazer pedidos de compra, deve então ser realizada. Na prática, ela nunca é assim. As informações sobre o sistema são normalmente adquiridas em vários níveis diferentes, ao mesmo tempo. Os modelos de nível inferior podem ser desenvolvidos primeiramente e então resumidos para criar um modelo mais geral.

Os modelos de fluxo de dados mostram uma perspectiva funcional em que cada transformação representa uma única função. Da mesma maneira como se descreve o processamento dentro de um sistema, algumas vezes é útil utilizar descrições de fluxo de dados para ilustrar o contexto do sistema. O modelo de fluxo de dados pode mostrar como diferentes sistemas e subsistemas trocam informações. Esses subsistemas não precisam ser funções únicas. Por exemplo, um subsistema pode ser um servidor de armazenamento de dados, com uma

Figura 7.3 ▼

Diagrama de fluxo de dados do processamento de pedido de compra.

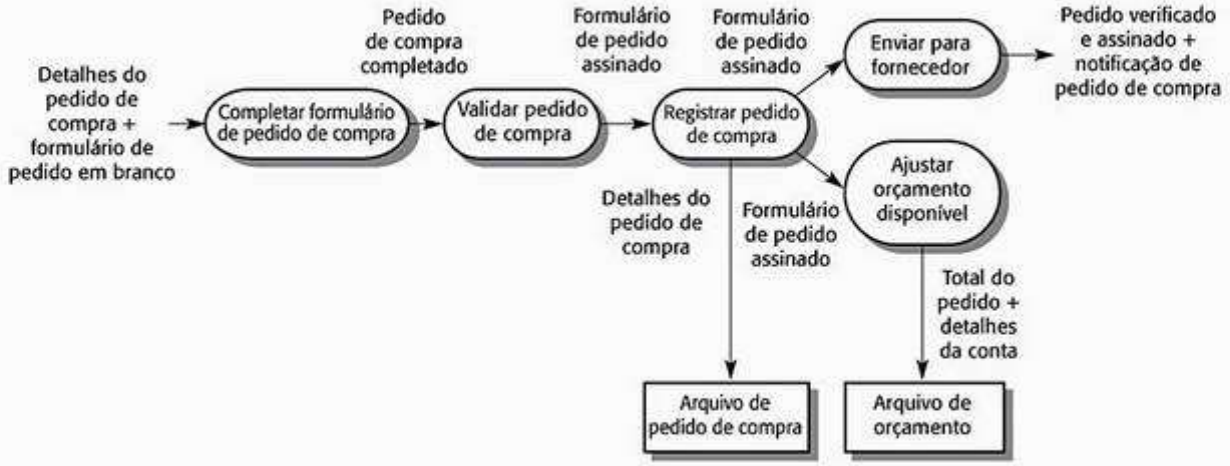


Figura 7.4

Diagrama de fluxo de dados de um conjunto de ferramentas CASE.

interface razoavelmente complexa. A Figura 7.4 mostra um exemplo de um diagrama de fluxo de dados utilizado dessa maneira. Nesse exemplo, os retângulos com cantos arredondados representam os subsistemas.

7.2.2 Modelos de máquinas de estado

Os modelos de máquinas de estado são utilizados para modelar o comportamento de um sistema, em resposta a eventos externos e internos. O modelo de máquina de estado mostra estados de sistema e eventos que provocam as transições de um estado para outro, mas não mostra o fluxo de dados dentro do sistema. Esse tipo de modelo é particularmente útil para modelar sistemas em tempo real, porque esses sistemas são, com freqüência, dirigidos por estímulos do ambiente do sistema. Por exemplo, o sistema de alarme em tempo real, que será discutido no Capítulo 13, responde a estímulos de sensores de movimento, sensores de abertura de porta etc.

Os modelos de máquinas de estado são parte integral dos métodos de projeto em tempo real, como o que foi proposto por Ward e Mellor (1985) e Harel (1987, 1988). O método de Harel utiliza uma notação chamada *statecharts*, que foi a base para a notação de modelagem de máquina de estado na UML.

Um modelo de máquina de estado de um sistema presume que, a qualquer momento, o sistema está em um estado, dentre uma série de possíveis estados. Quando um estímulo é recebido, isso pode dar início a uma transição para um estado diferente. Por exemplo, um sistema que controla uma válvula pode passar do estado 'Válvula aberta' para o estado 'Válvula fechada', quando um comando (estímulo) do operador é recebido.

Essa abordagem de modelagem de sistema está ilustrada na Figura 7.5. Esse diagrama mostra um modelo de máquina de estado de um forno de microondas simples, equipado com botões para ajustar a potência e o relógio e iniciar o sistema. Os fornos de microondas reais são, na verdade, muito mais complexos do que o sistema descrito aqui. Contudo, o modelo inclui as características essenciais do sistema. Para simplificar o modelo, presumi que a seqüência de ações no uso do forno de microondas é:

1. Selecione o nível de potência (potência média ou potência total).
2. Informe o tempo de cozimento.
3. Pressione 'iniciar' e o alimento será cozido durante o tempo especificado.

Por razões de segurança, o forno não deve operar quando a porta estiver aberta e, ao completar o cozimento, um alarme é acionado. Ele tem um visor alfanumérico muito simples, utilizado para exibir várias mensagens de aviso e de alerta.

A notação em UML que utilizo para descrever os modelos de máquina de estado é projetada para modelar o comportamento de objetos. Contudo, é uma notação de propósito geral, que pode ser utilizada para qualquer tipo de modelagem de máquina de estado. Os retângu-

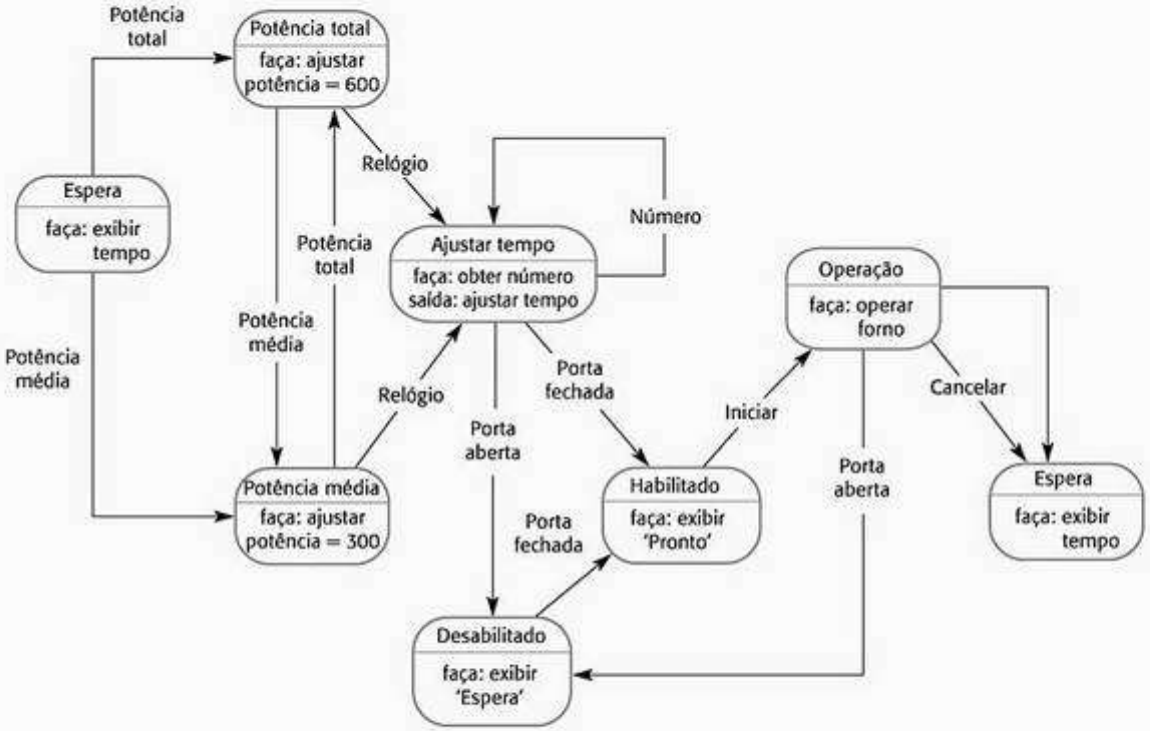


Figura 7.5 ▲

Modelo de máquina de estado de um forno de microondas.

los com cantos arredondados em um modelo representam os estados do sistema; eles apresentam uma breve descrição (depois de 'faça') da ação tomada naquele estado. As setas com legenda representam os estímulos, que forçam a transição de um estado para outro.

Portanto, na Figura 7.5, podemos ver que o sistema responde inicialmente ao botão de potência total ou ao de média potência. Os usuários podem mudar de idéia depois de selecionar um desses botões e escolher a outra alternativa. O tempo é especificado e, então, desde que a porta esteja fechada, o botão iniciar será habilitado. Apertando esse botão, a operação do forno se inicia e o cozimento é realizado durante o tempo especificado.

A notação em UML permite indicar a atividade que ocorre em um estado. Contudo, em uma especificação de sistema detalhada, é preciso fornecer mais detalhes, tanto sobre o estímulo como sobre o estado do sistema (Figura 7.6). Essas informações podem ser mantidas em um dicionário de dados, como será discutido posteriormente nesta seção.

O problema com a abordagem de máquina de estado é que o número de estados possíveis aumenta rapidamente. Para grandes modelos de sistema, portanto, é necessário fazer alguma estruturação desses modelos de estado. Um modo de fazer isso é utilizar a noção de um superestado, que engloba um número de estados separados. Esse superestado se parece com um estado único em um modelo de alto nível, mas ele é ampliado com mais detalhes, em um diagrama separado. Para ilustrar esse conceito, considere o estado Operação, na Figura 7.5. Trata-se de um superestado que pode ser expandido, como ilustra a Figura 7.7.

O estado Operação inclui uma série de subestados; ele mostra que a operação se inicia com uma verificação de status e, se qualquer problema for descoberto, um alarme é indicado e a operação é desabilitada. O cozimento envolve operar o gerador de microondas durante o tempo especificado e, ao término, um alarme sonoro é acionado. Se a porta for aberta durante a operação, o sistema passará para o estado desabilitado, como mostra a Figura 7.5.

Figura 7.6

Descrição de estado e estímulos do forno de microondas.

Estado	Descrição
Espera	O forno está aguardando a programação. O visor exibe a hora atual.
Potência média	A potência do forno está ajustada para 300 watts. O visor exibe 'Potência média'.
Potência total	A potência do forno está ajustada para 600 watts. O visor exibe 'Potência total'.
Ajustar tempo	O tempo de cozimento é ajustado pelo usuário. O visor mostra o tempo de cozimento selecionado e é atualizado quando o tempo é ajustado.
Desabilitado	A operação do forno está desabilitada, por questões de segurança. A luz interna do forno está acesa. O visor exibe 'Não está pronto'.
Habilitado	A operação do forno está habilitada. A luz interna do forno está apagada. O visor exibe 'Pronto para cozinhar'.
Operação	Forno em operação. A luz interna do forno está acesa. O visor exibe a contagem de tempo do relógio. Ao término do cozimento, um alarme soa por cinco segundos. A luz do forno está acesa. O visor exibe 'Cozimento concluído', enquanto o alarme soa.
Estímulo	**Descrição**
Potência média	O usuário pressionou o botão de potência média.
Potência total	O usuário pressionou o botão de potência total.
Relógio	O usuário pressionou um dos botões do relógio.
Número	O usuário pressionou uma tecla numérica.
Porta aberta	A porta do microondas está aberta.
Porta fechada	A porta do microondas está fechada.
Iniciar	O usuário pressionou o botão Iniciar.
Cancelar	O usuário pressionou o botão Cancelar.

Figura 7.7

Microondas em operação.

7.3 Modelos de dados

A maioria dos grandes sistemas de software utiliza um grande banco de dados de informações. Em alguns casos, esse banco de dados existe, independentemente do sistema de software. Em outros, ele é criado para o sistema que está sendo desenvolvido. Uma parte importante da modelagem de sistemas é definir o modo lógico dos dados processados pelo sistema.

A técnica de modelagem de dados mais amplamente utilizada é a modelagem denominada ERA (*entity-relation-attribute* – entidade-relação-atributo), que mostra as entidades de dados, seus atributos associados e as relações entre essas entidades. Essa abordagem da modelagem foi proposta primeiramente em meados da década de 70 por Chen (1976), com diversas variantes desenvolvidas desde então (Codd, 1979; Hammer e McLeod, 1981; Hull e King, 1987). Contudo, todas elas têm a mesma forma básica.

A UML não inclui uma notação específica para esse tipo de modelagem de dados, uma vez que ela presume um processo de desenvolvimento orientado a objetos e modela os dados utilizando objetos e seus relacionamentos. Contudo, podemos pensar em entidades como classes de objetos simplificadas (elas não têm nenhuma operação) e podemos utilizar modelos de classe em UML com associações identificadas entre as classes como modelos de dados. Embora os modelos de dados resultantes não sejam UML de boa qualidade, isso é compensado pela conveniência de se utilizar uma notação-padrão.

Os modelos de dados são, freqüentemente, utilizados em conjunto com os modelos de fluxo de dados, para descrever a estrutura da informação que está sendo processada. Ilustrei esse processo incluindo um modelo de dados de um projeto de software (Figura 7.8). Esse projeto pode ser processado pelos vários componentes do conjunto de ferramentas CASE, mostrado na Figura 7.4.

Os projetos são grafos dirigidos. Eles consistem de um conjunto de nós de diferentes tipos, conectados por links que representam os relacionamentos entre os nós. Existe uma representação em tela desse grafo, que é um diagrama de projeto e uma representação de banco de dados correspondente. O sistema de edição executa um mapeamento da representação do banco de dados para a representação em tela, cada vez que um diagrama é desenhado. As informações produzidas pelo editor para outras ferramentas de análise de projeto devem incluir a representação lógica do grafo de projeto. Contudo, essas ferramentas de análise não estão voltadas para os detalhes da representação física em tela. Elas processam as entidades, seus atributos lógicos (como seus nomes) e seus relacionamentos.

A Figura 7.8 mostra que um projeto tem atributos cujos valores são o nome do projeto, a descrição do projeto, a data de criação (data C) e a data de modificação (data M). O projeto é constituído de um conjunto de nós e de um conjunto de links. Os nós e os links têm nomes e atributos de tipo. Eles podem ter um conjunto de rótulos associados, que armazenam outras informações descritivas. Cada rótulo tem um ícone e texto associados.

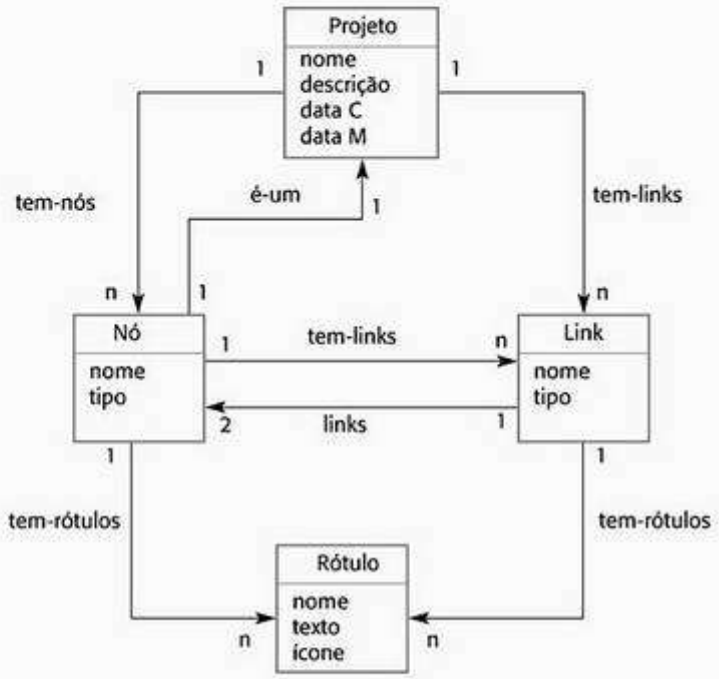
Os modelos de relacionamento de entidades têm sido amplamente utilizados em projetos de banco de dados. Os esquemas de bancos de dados relacionais anteriores a esses modelos estão em terceira forma normal, que é uma característica desejável (Barker, 1989). Por causa dos tipos explícitos e do reconhecimento de subtipos e supertipos, é também simples e direto implementar esses modelos utilizando-se bancos de dados orientados a objetos.

Como todos os modelos gráficos, os modelos ERA não têm detalhes e devem ser complementados com mais descrições detalhadas das entidades, das relações e dos atributos incluídos no modelo. Essas descrições mais detalhadas podem ser coletadas em um repositório ou dicionário de dados. Os dicionários de dados geralmente são úteis quando se desenvolvem modelos de sistema e podem ser utilizados para gerenciar todas as informações de todos os tipos de modelos de sistema.

Um dicionário de dados é, simplesmente, uma lista alfabética dos nomes inclusos nos diferentes modelos do sistema. Além do nome, o dicionário deve incluir uma descrição associada da entidade citada e, se o nome representar um objeto composto, pode haver uma

Figura 7.8

Modelo de dados semântico de um projeto de software.



descrição da composição. Outras informações, como a data da criação, o criador e a representação da entidade, podem também ser incluidas, dependendo do tipo de modelo que está sendo desenvolvido.

Eis as vantagens de utilizar um dicionário de dados:

1. Este é um mecanismo para o gerenciamento de nomes. Muitas pessoas diferentes podem precisar inventar nomes para as entidades e os relacionamentos, quando um grande modelo de sistema é desenvolvido. Esses nomes devem ser utilizados de modo consistente e não devem coincidir. O software de dicionário de dados pode fazer a verificação, a fim de assegurar a exclusividade do nome e informar aos analistas de requisitos sobre a possível duplicidade de nomes.

2. Ele serve como um depósito de informações organizacionais, que pode vincular a análise, o projeto, a implementação e a evolução. À medida que o sistema é desenvolvido, as informações são utilizadas para esclarecimentos sobre o desenvolvimento. Toda nova informação é acrescentada ao repositório e toda informação sobre uma entidade está em um único lugar.

Todos os nomes de sistemas, quer sejam nomes de entidades, tipos, relações, atributos ou serviços, devem fazer parte do dicionário. O software de apoio deve estar disponível para criar, fazer a manutenção e fazer a consulta no dicionário. Esse software pode ser integrado com outras ferramentas, de modo que a criação do dicionário seja parcialmente automatizada. A maioria das ferramentas CASE que apóiam a modelagem de sistemas também inclui a compatibilidade com dicionários de dados.

Um exemplo de entradas do dicionário de dados é mostrado na Figura 7.9. Utilizamos nomes encontrados no modelo de dados semântico do projeto mostrado na Figura 7.8. Simplifiquei a apresentação desse exemplo, excluindo alguns nomes e abreviando as informações associadas. Uma entrada mais completa do dicionário de dados incluiria links para a representação da informação (por exemplo, uma declaração de tipo) e links reversos para o ponto em que o nome é utilizado.

Figura 7.9 ▶

Exemplos de entradas em um dicionário de dados.

Nome	Descrição	Tipo	Data
tem-rótulos	Relação de 1:N entre entidades do tipo nó ou link e entidades do tipo Rótulo.	Relação	5.10.1998
Rótulo	Guarda informações estruturadas ou não estruturadas sobre nós ou links. Os rótulos são representados por um ícone (que pode ser uma caixa transparente) e pelo texto associado.	Entidade	8.12.1998
Link	Uma relação de 1:1 entre as entidades de projeto representadas como nós. Os links têm tipos e podem ser nomeados.	Relação	8.12.1998
nome (rótulo)	Cada rótulo tem um nome que identifica o tipo de rótulo. O nome deve ser único dentro do conjunto de tipos de rótulos utilizados em um projeto.	Atributo	8.12.1998
nome (nó)	Cada nó tem um nome que deve ser único dentro de um projeto. O nome pode ter até 64 caracteres.	Atributo	15.11.1998

7.4 Modelos de objeto

Uma abordagem orientada a objetos para todo o desenvolvimento de software é atualmente de uso comum, particularmente para o desenvolvimento de sistemas interativos. Isso significa expressar os requisitos de sistemas utilizando um modelo de objeto, projetar utilizando objetos e desenvolver o sistema em uma linguagem de programação orientada a objetos, como Java ou C++.

Os modelos de objeto desenvolvidos durante a análise de requisitos podem ser utilizados para representar os dados do sistema e seu processamento. A esse respeito, eles combinam alguns usos de modelos de fluxo de dados e modelos de dados semânticos. Eles também são úteis para mostrar como as entidades no sistema podem ser classificadas e compostas de outras entidades.

Para algumas classes de sistemas, os modelos de objeto são meios naturais de refletir as entidades do mundo real, que são manipuladas pelo sistema. Isso é particularmente verdadeiro quando o sistema processa informação sobre entidades concretas, como por exemplo carros, aeronaves e livros, que já têm atributos claramente identificáveis. As entidades mais abstratas, de nível superior, como o conceito de uma biblioteca, um sistema de registro médico ou um processador de textos, são mais difíceis de modelar como classes de objetos. Elas não têm necessariamente uma interface simples, consistindo em atributos e operações independentes.

Os modelos de objeto desenvolvidos durante a análise de requisitos certamente simplificam a transição para o projeto e a programação orientados a objetos. Contudo, constatei que os usuários finais de um sistema, freqüentemente, consideram que os modelos de objeto não são naturais e são de difícil compreensão. Muitas vezes, eles preferem adotar uma visão mais funcional de processamento de dados. Portanto, algumas vezes, é útil complementar modelos de objeto com modelos de fluxo de dados, que mostram o processamento de dados de ponta a ponta no sistema.

Uma classe de objeto é uma abstração em relação a um conjunto de objetos que identificam atributos comuns (como em um modelo de dados semântico) e os serviços ou as operações que são fornecidos por cada objeto. Os objetos são entidades executáveis, com os atributos e os serviços da classe de objeto. Eles são instâncias da classe de objeto, e muitos objetos diferentes podem ser criados a partir de uma classe. De modo geral, os modelos desenvolvidos que utilizam análise focalizam as classes de objeto e suas relações.

Modelos de sistemas desenvolvidos durante a análise de requisitos devem modelar as entidades do mundo real utilizando classes de objeto e não devem incluir detalhes dos objetos individuais no sistema. Vários tipos de modelos de objeto podem ser produzidos, mostrando como as classes de objetos são relacionadas umas às outras, como os objetos são agregados para formar outros objetos, como os objetos interagem com outros objetos e assim por diante. Tudo isso pode contribuir para a nossa compreensão de um sistema que está sendo especificado.

O processo de análise para identificar objetos e classes de objetos é reconhecido como uma das áreas mais difíceis do desenvolvimento orientado a objetos. A identificação de objetos é basicamente a mesma para a análise e o projeto. Podem ser utilizados os métodos de identificação de objetos, apresentados no Capítulo 12, que discute o projeto orientado a objetos. Aqui, concentro-me em alguns modelos de objeto que podem ser gerados durante o processo de análise.

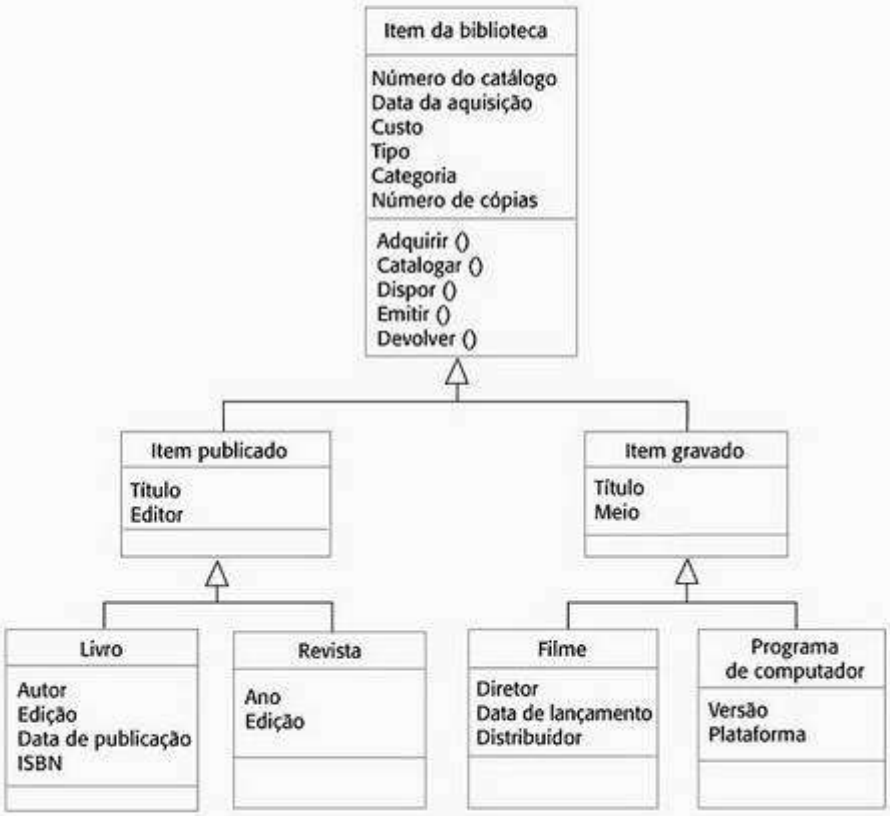
Vários métodos de análise orientados a objetos foram propostos na década de 90 (Booch, 1994; Coad e Yourdon, 1990; Rumbaugh *et al.*, 1991). Esses métodos têm muito em comum, e três dos principais desenvolvedores (Booch, Rumbaugh e Jacobson) decidiram integrar suas abordagens para produzir um método unificado (Rumbaugh *et al.*, 1999b). A linguagem de modelagem unificada (UML), utilizada nesse método unificado, tornou-se um padrão efetivo para a modelagem de objetos. Ela inclui notações para diferentes tipos de modelo de sistemas. Já vimos os modelos de use-case e os diagramas de seqüência, no Capítulo 5, e também os modelos de máquina de estado, no início deste capítulo.

Uma classe de objetos em UML, como ilustram os exemplos da Figura 7.10, é representada como um retângulo orientado verticalmente, com três seções:

1. O nome da classe do objeto está na seção superior.
2. Os atributos de classe estão na seção intermediária.
3. As operações associadas com a classe de objeto estão na seção inferior do retângulo.

Figura 7.10 ▶

Parte de uma hierarquia de classes para um sistema de biblioteca.



Neste livro, não é possível abordar todos os aspectos da UML, por isso detenho-me nos modelos de objeto, a fim de mostrar como os objetos podem ser classificados e herdar atributos e operações de outros objetos, nos modelos de agregação, que mostram como os objetos são compostos, e nos modelos simples de comportamento, que mostram as interações de objetos.

7.4.1 Modelos de herança

A modelagem orientada a objetos envolve identificar as classes de objetos que são importantes no domínio em estudo, e elas são organizadas em uma taxonomia. Uma taxonomia é um esquema de classificação que mostra como uma classe de objetos está relacionada com outras classes, por meio de atributos e serviços comuns.

Para mostrar essa taxonomia, as classes são organizadas em uma hierarquia de herança, em que as classes de objetos mais gerais são apresentadas no topo da hierarquia. Os objetos mais especializados herdam seus atributos e serviços. Esses objetos especializados podem ter seus próprios atributos e serviços.

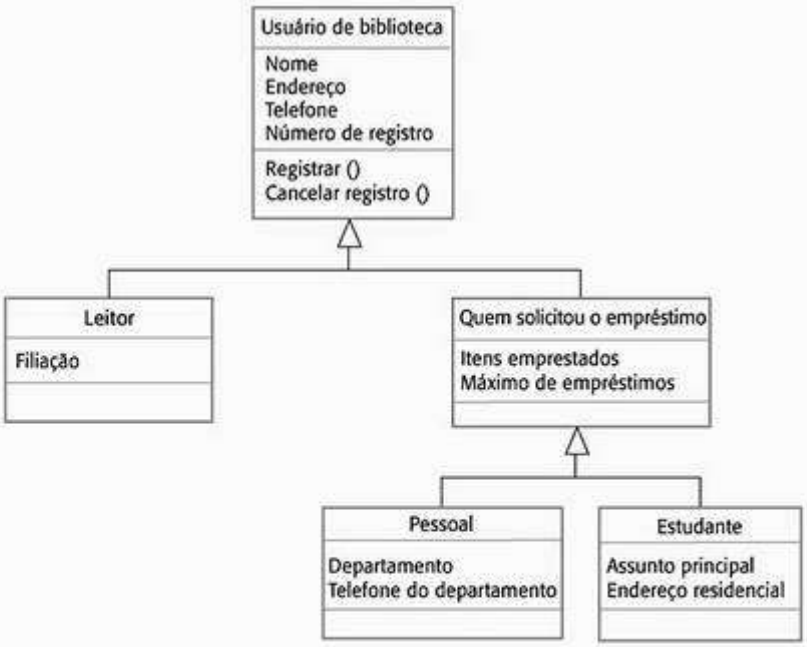
A Figura 7.10 ilustra parte de uma hierarquia de classes simplificada para um modelo de sistema de biblioteca. Essa hierarquia fornece informações sobre os itens pertencentes à biblioteca. A biblioteca tem vários tipos de itens, como livros, músicas, filmes, revistas e jornais. Na Figura 7.10, o item mais geral está no topo da árvore e tem um conjunto de atributos e serviços que são comuns a todos os itens de bibliotecas. Esses itens são herdados pelas classes Item publicado e Item gravado, que acrescentam seus próprios atributos, que são então herdados por itens de nível inferior.

A Figura 7.11 é um exemplo de outra hierarquia de herança, que pode ser parte de um modelo de biblioteca. Nesse caso, são mostrados os usuários de uma biblioteca. Existem duas classes de usuários: aqueles que têm permissão para pegar livros emprestados e aqueles que somente podem ler os livros na biblioteca, sem levá-los.

Na notação em UML, a herança é mostrada no sentido ascendente, em vez de no sentido descendente, como acontece em algumas notações orientadas a objetos. Ou seja, as setas (mostradas como um triângulo) apontam a partir da classe que herda atributos e operações para as classes superiores. Em vez de utilizar o termo herança, a UML se refere ao relacionamento de generalização.

Figura 7.11 ▶

Hierarquia de classe de usuário.



O projeto de hierarquia de classes não é fácil. Uma vantagem de desenvolver esses modelos é que o analista precisa compreender detalhadamente o domínio em que o sistema será instalado. Como um exemplo da sutileza do problema que surge na prática, considere a hierarquia de itens de biblioteca. Aparentemente, o atributo 'Título' poderia ser colocado no item mais geral e, então, herdado por itens de nível inferior.

Embora tudo em uma biblioteca deva ter algum tipo de identificador ou número de registro, isso não indica que tudo deva ter um título. Por exemplo, uma biblioteca pode ter os artigos pessoais de um político aposentado. Muitos desses itens podem não estar explicitamente identificados. Esses documentos serão classificados utilizando-se outra classe (não mostrada aqui) que tenha um conjunto diferente de atributos.

As figuras 7.10 e 7.11 mostram hierarquias de herança de classe em que cada classe de objeto herda seus atributos e operações a partir de uma única classe materna. Também podem ser construídos modelos de herança múltipla, em que uma classe tem várias classes de origem. Seus atributos e serviços herdados são um conjunto daqueles herdados de cada superclasse. A Figura 7.12 mostra um exemplo de um modelo de herança múltipla, que também pode ser parte do modelo de biblioteca.

O problema principal com a herança múltipla é projetar um grafo de herança em que os objetos não herdem atributos desnecessários. Entre outros problemas estão a dificuldade de reorganizar o grafo de herança, quando são necessárias mudanças, e solucionar a coincidência de nomes, quando duas ou mais superclasses têm atributos com o mesmo nome, mas com diferentes significados. No que se refere à modelagem de sistemas, essas coincidências são relativamente fáceis de serem resolvidas, alterando manualmente o modelo de objeto. Elas causam mais problemas na programação orientada a objetos.

7.4.2 Agregação de objetos

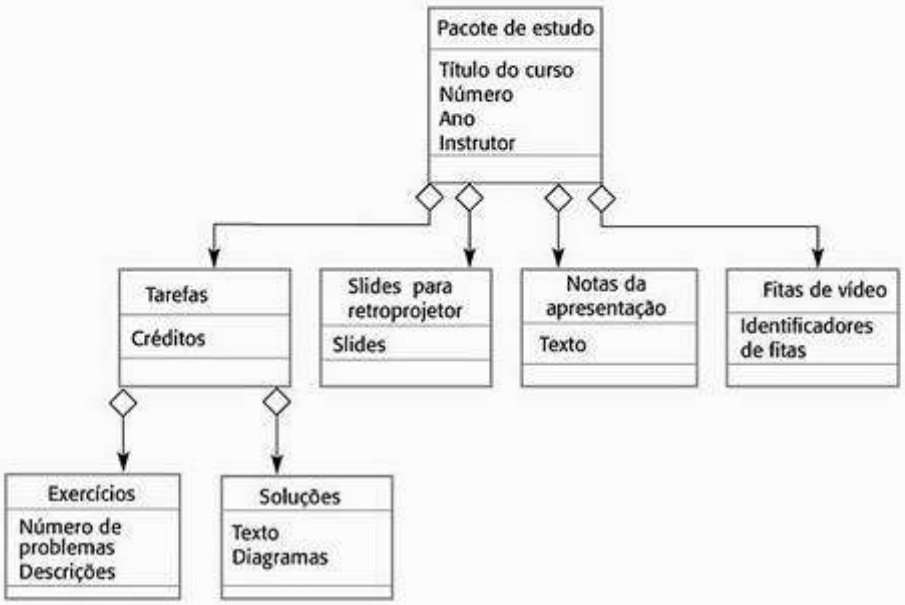
Além de adquirir atributos e serviços por meio de um relacionamento de herança com outros objetos, alguns objetos são constituídos por outros objetos. Ou seja, um objeto é um agregado de um conjunto de outros objetos. As classes que representam esses objetos podem ser modeladas com o uso de um modelo de agregação de objetos, como mostra a Figura 7.13. Nesse exemplo, modelamos um item de biblioteca que é um pacote de estudo para um curso universitário. Esse pacote de estudo contém as notas obtidas em aulas expositivas, exercícios, resoluções de testes, cópias de transparências utilizadas em apresentações e fitas de vídeo.

A notação em UML para a agregação se destina a representar a composição pela inclusão de um formato de losango na origem do link. Portanto, a Figura 7.13 pode ser lida como: "Um pacote de estudo é composto de uma ou mais tarefas, pacotes de slides para retroprojetor, notas referentes a apresentações e fitas de vídeo".

Figura 7.12 ▶
Herança múltipla.

Figura 7.13 ▶

Um objeto agregado representando um curso.



7.4.3 Modelagem de comportamento de objetos

Para modelar o comportamento de objetos, temos de mostrar como as operações fornecidas pelos objetos são utilizadas. Em UML, os comportamentos são modelados com o uso de cenários baseados em use-cases, como foi discutido no Capítulo 6. Já vimos um exemplo de modelagem de comportamento, no Capítulo 6, em que a Figura 6.13 ilustra o gerenciamento de um catálogo de biblioteca. Além dos diagramas de seqüência, a UML também contém diagramas de colaboração, que mostram a seqüência das mensagens trocadas pelos objetos. Esses diagramas são semelhantes aos diagramas de seqüência, mas não abordarei agora esse assunto.

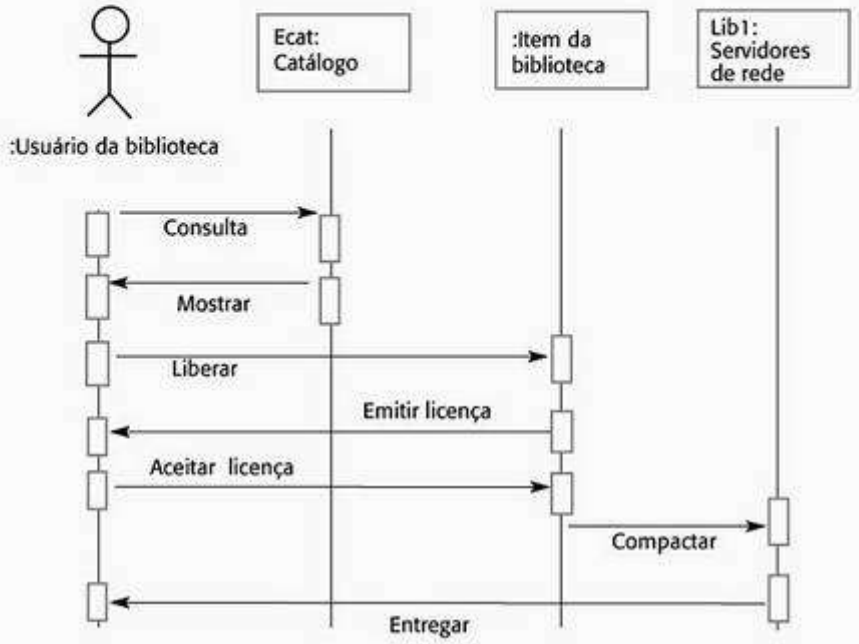
Ilustro como os diagramas de seqüência podem ser utilizados para a modelagem de comportamento, descrevendo um cenário em que os usuários retiram itens da biblioteca utilizando um formulário eletrônico. Por exemplo, imagine uma situação em que os pacotes de estudo, mostrados na Figura 7.13, possam ser mantidos eletronicamente e transferidos para o computador do estudante por download.

A Figura 7.14 mostra um diagrama de seqüência com objetos ao longo do topo do diagrama. As operações estão indicadas por setas rotuladas e a seqüência das operações é descendente. Nesse cenário, o usuário da biblioteca acessa o catálogo para verificar se o item solicitado está disponível eletronicamente e, caso esteja, pede a liberação eletrônica desse item. Por razões de direito de propriedade, essa operação precisa ser licenciada, de modo que exista uma transação entre o item e o usuário, em que a licença é aceita. O item a ser liberado é remetido para um objeto servidor de rede, para que os dados sejam compactados, antes de serem remetidos para o usuário da biblioteca.

7.5 Ferramentas CASE em *workbenches*

Uma *workbench* CASE é um conjunto de ferramentas que são utilizadas em determinada fase do processo de software, como a fase de projeto, implementação ou teste. A vantagem de agrupar as ferramentas CASE em uma *workbench* é que as ferramentas podem trabalhar em conjunto, para oferecer um apoio mais abrangente. Serviços comuns podem ser imple-

Figura 7.14 ▶
Liberação eletrônica de itens.



mentados e solicitados por todas as outras ferramentas. As ferramentas de *workbench* podem ser integradas por meio de arquivos compartilhados, de um repositório compartilhado ou de estruturas de dados compartilhadas.

As *workbenches* de análise e projeto são projetadas para dar apoio à modelagem de sistemas durante os estágios de análise e projeto do processo de software. Essas *workbenches* fornecem apoio à criação, à edição e à análise das notações gráficas utilizadas em métodos estruturados. As *workbenches* de análise e projeto podem apoiar um método específico de análise e projeto, como análise e projeto orientados a objetos. Como alternativa, elas podem ser sistemas mais gerais de edição de diagramas, com o conhecimento dos tipos de diagramas que são utilizados em vários métodos. As *workbenches* orientadas a métodos incorporam regras e diretrizes de métodos. Portanto, é possível realizar alguma verificação automática dos diagramas.

A Figura 7.15 mostra as ferramentas que podem ser incluídas em uma *workbench* de análise e projeto. Essas ferramentas são, normalmente, integradas por meio de um repositório compartilhado, cuja estrutura é de propriedade do fornecedor da *workbench*. Portanto, as *workbenches* de análise e projeto são, geralmente, ambientes fechados. É difícil para os usuários adicionar suas próprias ferramentas ou modificar as ferramentas fornecidas.

Figura 7.15 ▶
Uma *workbench* de projeto e análise.

A *workbench* de análise e projeto, ilustrada na Figura 7.15, inclui:

1. *Editores de diagramas*, para criar diagramas de fluxo de dados, hierarquias de objetos, diagramas de relacionamento de entidades etc. Esses editores não são apenas ferramentas de desenho, mas associam o tipo de entidades no diagrama. Eles capturam informações sobre essas entidades e armazenam essas informações no repositório central.
2. *Ferramentas de análise, de projeto e de verificação*, que processam o projeto e produzem um relatório de erros e anomalias. Elas podem ser integradas com o sistema de edição, de modo que os erros do usuário sejam percebidos em um estágio inicial do processo.
3. *Linguagens de pesquisa de repositório*, que permitem ao projetista encontrar no repositório projetos e informações associadas a projetos.
4. *Dicionário de dados*, que mantém informações sobre as entidades utilizadas em um projeto de sistema.
5. *Ferramentas de definição e geração de relatórios*, que captam informações do depósito central e automaticamente geram documentação do sistema.
6. *Ferramentas de definição de formulários*, que permitem especificar formatos de telas e de documentos.
7. *Recursos de importação/exportação*, que permitem o intercâmbio de informações do repositório central com outras ferramentas de desenvolvimento.
8. *Geradores de código*, que geram código ou estruturas de código automaticamente a partir do projeto capturado no repositório central.

Em alguns casos, é possível gerar um programa ou um fragmento de programa com as informações fornecidas no modelo de sistema. Os geradores de código que estão incluídos em *workbenches* de análise e projeto podem gerar códigos em linguagens como Java, C++ ou C. Como os modelos excluem detalhes de nível inferior, o gerador de código em uma *workbench* de projeto pode não ser capaz de gerar o sistema completo. De modo geral, é necessário fazer alguma codificação manualmente, a fim de completar o programa gerado.

Algumas *workbenches* de análise e projeto se destinam a dar apoio a métodos que são destinados ao desenvolvimento de aplicações de sistemas de negócios. A plataforma de desenvolvimento e a plataforma de aplicações são as mesmas, de modo que um sistema de banco de dados, como Sybase ou Oracle, é utilizado para implementar o repositório compartilhado da ferramenta. Entre essas *workbenches* estão a maior parte dos recursos de um ambiente de programação em 4GL e, assim, em vez de gerarem uma linguagem de programação convencional a partir de um projeto, podem gerar uma linguagem de banco de dados em 4GL.

PONTOS-CHAVE

- Um modelo é uma visão abstrata de um sistema, que ignora alguns de seus detalhes. Modelos de sistema complementares podem ser desenvolvidos, os quais apresentam informações diferentes sobre o sistema.
- Os modelos de contexto mostram como o sistema que está sendo modelado está posicionado em um ambiente com outros sistemas e processos. Os modelos de arquitetura, os modelos de processo e os modelos de fluxo de dados podem ser utilizados como modelos de contexto.

- Os diagramas de fluxo de dados podem ser utilizados para modelar o processamento de dados realizado por um sistema. O sistema é modelado como um conjunto de transformação de dados, com funções que atuam sobre eles.
- Os modelos de máquinas de estado são utilizados para modelar o comportamento de um sistema, em resposta a eventos internos ou externos.
- Os modelos semânticos de dados descrevem a estrutura lógica dos dados que são importados e exportados pelo sistema. Eles mostram as entidades de sistema, seus atributos e os relacionamentos em que elas participam. Eles podem ser complementados com dicionários de dados, que fornecem uma descrição mais detalhada dos dados.
- Os modelos de objetos descrevem as entidades do sistema lógico e sua classificação e agregação. Eles combinam um modelo de dados com um modelo de processamento. Entre os possíveis modelos de objetos que podem ser desenvolvidos estão os modelos de herança, os modelos de agregação e os modelos de comportamento.
- As ferramentas CASE em *workbenches* dão apoio ao desenvolvimento de modelos de sistema, fornecendo ferramentas de edição, verificação, relatório e documentação.

LEITURAS SUGERIDAS

Structured analysis and system specification. Essa foi talvez a primeira e é certamente a descrição mais conhecida da modelagem de fluxo de dados. Esse tópico é também abordado em muitos outros livros modernos, mas eles não necessariamente trouxeram melhorias em relação ao original. (T. DeMarco, 1978, Yourdon Press.)

Software requirements: objects, functions and states. Esse livro focaliza a modelagem de sistemas no processo de engenharia de requisitos. A análise da modelagem de sistemas é seu mérito. (A.M. Davis, 1993, Prentice Hall.)

The unified modeling language user guide. Esse livro compõe uma série de três livros escritos pelos mesmos autores que criaram a UML e o Método Unificado. Esse livro e os textos correspondentes, dos mesmos autores, constituem os livros definitivos sobre UML. (G. Booch, J. Rumbaugh e I. Jacobson, 1999, Addison Wesley.)

EXERCÍCIOS

7.1 Desenhe um modelo de contexto para um sistema de informações sobre pacientes em um hospital. Você pode fazer qualquer suposição razoável sobre os outros sistemas de hospital que estiverem disponíveis, mas seu modelo deve conter um sistema de admissão de pacientes e um sistema de armazenamento de imagens para a produção de chapas de raios X.

7.2 Com base em sua experiência com um ATM, desenhe um diagrama de fluxo de dados modelando o processamento de dados envolvido quando um cliente efetua um saque na máquina.

7.3 Modele o processamento de dados que pode ocorrer em um sistema de correio eletrônico. Você deve modelar separadamente o processamento de envio e de recebimento de mensagens.

7.4 Projete os modelos de máquina de estado do software de controle para:

- uma máquina de lavar automática, que tem diferentes programas para diferentes tipos de roupas;
- o software de um aparelho de reprodução de CDs, e
- uma secretária eletrônica, que grava as mensagens recebidas e indica o número dessas mensagens em um visor luminoso. O sistema deve permitir ao proprietário do telefone discar, digitar uma seqüência de números (identificados como tons) e ouvir as mensagens que foram gravadas.

7.5 Amplie o modelo de um projeto de software, mostrado na Figura 7.8, para incluir informações de layout físico. Os nós podem ser representados por símbolos compostos, que são uma combinação de símbolos simples, como retângulos e linhas, e são exibidos em uma coordenada específica. Os links são constituídos por uma série de segmentos de linhas, que podem ser cheias ou pontilhadas. O texto é exibido em uma fonte especificada. O nome associado a um nó, link ou rótulo é posicionado dentro ou perto do nome do item.

7.6 Modele as classes de objeto que podem ser utilizadas em um sistema de correio eletrônico. Se você fez o Exercício 7.3, descreva as semelhanças e as diferenças entre o modelo de processamento de dados e o modelo de objeto.

7.7 Utilizando uma abordagem de relacionamento de entidades, descreva um possível modelo de dados para o sistema do catálogo de biblioteca, descrito neste capítulo. Suponha que os itens da biblioteca sejam aqueles modelados na Figura 7.10.

7.8 Desenvolva um modelo de objeto, com um diagrama de hierarquia de classes e um diagrama de agregação, mostrando os componentes principais de um sistema de computador pessoal e seu sistema de software.

7.9 Desenvolva um diagrama de seqüência, mostrando as interações envolvidas quando um estudante se inscreve para um curso em uma universidade. Os cursos podem ter um número limitado de vagas e, assim, o processo de inscrição deve incluir a verificação da disponibilidade dessas vagas. Suponha que o estudante tenha acesso a um catálogo eletrônico, para que possa se informar sobre os cursos disponíveis.

7.10 Descreva três atividades de modelagem que podem receber apoio de uma ferramenta CASE em *workbench* para algum método de análise. Sugira três atividades que não podem ser prontamente automatizadas.

8 Prototipação de software

Objetivos

Os objetivos deste capítulo são explicar como a prototipação de software é utilizada no processo de software e descrever diferentes abordagens para o desenvolvimento de protótipos. Depois de ler este capítulo, você poderá:

- compreender o papel da prototipação em diferentes tipos de projetos de desenvolvimento;
- compreender a diferença entre a prototipação evolucionária e a prototipação descartável;
- conhecer três diferentes técnicas de prototipação, que são: o desenvolvimento de linguagem de nível muito elevado, a programação de banco de dados e o reuso de componentes e aplicações, e
- compreender por que a prototipação é a única técnica viável para o projeto e o desenvolvimento de interface com o usuário.

Sumário

8.1 Prototipação no processo de software
8.2 Técnicas de prototipação rápida
8.3 Prototipação de interface com o usuário

Os clientes e os usuários finais de software acham muito difícil expressar seus reais requisitos. É quase impossível prever como um sistema afetará práticas de trabalho, como interagirá com outros sistemas e que operações dos usuários devem ser automatizadas. A análise cuidadosa e as revisões sistemáticas de requisitos ajudam a reduzir as incertezas sobre o que o sistema deve fazer. Contudo, não há um substituto real para experimentar um requisito antes de concordar com ele. Isso será possível se um protótipo de sistema estiver disponível.

Um protótipo é uma versão inicial de um sistema de software, que é utilizada para mostrar conceitos, experimentar opções de projeto e, em geral, para conhecer mais sobre os problemas e suas possíveis soluções. O desenvolvimento rápido de um protótipo é essencial para que os custos sejam controlados e os usuários possam fazer experiências com o protótipo no início do processo de software.

Um protótipo de software apóia duas atividades do processo de engenharia de requisitos:

1. *Levantamento de requisitos* Os protótipos de sistema permitem que os usuários realizem experiências para ver como o sistema apóia seu trabalho. Eles obtêm novas idéias para os requisitos e podem identificar pontos positivos e negativos do software. Eles podem, então, propor novos requisitos de sistema.

2. *Validação de requisitos* O protótipo pode revelar erros e omissões nos requisitos propostos. Uma função descrita em uma especificação pode parecer útil e bem-definida. Contudo, quando essa função é utilizada com outras, os usuários muitas vezes acham que sua visão inicial era incorreta e incompleta. A especificação de sistema pode então ser modificada para refletir sua compreensão alterada dos requisitos.

A prototipação pode ser utilizada como uma técnica de análise e redução de riscos (veja o Capítulo 4). Um risco significativo no desenvolvimento de software são os erros e as omissões de requisitos. Os custos da correção de erros de requisitos em fases posteriores do processo podem ser muito altos. As experiências mostraram (Boehm *et al.*, 1984) que a prototipação reduz o número de problemas com a especificação de requisitos. Além disso, os custos totais de desenvolvimento poderão ser mais baixos se um protótipo for desenvolvido.

Portanto, a prototipação é parte do processo de engenharia de requisitos. Contudo, a distinção entre a prototipação como uma atividade separada e o desenvolvimento massivo de software ficou obscurecida nos últimos anos. Muitos sistemas são, atualmente, desenvolvidos com o uso de uma abordagem evolucionária, em que uma versão inicial é criada rapidamente e modificada para produzir um sistema final. Um modelo de processo iterativo, como o desenvolvimento incremental (discutido no Capítulo 3), pode ser utilizado em conjunto com uma linguagem projetada para o rápido desenvolvimento de aplicações. Portanto, as técnicas utilizadas para desenvolver um protótipo para a validação de requisitos podem também ser utilizadas para desenvolver o próprio sistema de software. Discutirei isso na Seção 8.1.

Além de permitir que os usuários melhorem a especificação de requisitos, desenvolver um protótipo de sistema pode ter outros benefícios:

1. Possíveis equívocos entre desenvolvedores de software e usuários podem ser identificados à medida que as funções do sistema são apresentadas.
2. A equipe de desenvolvimento de software pode encontrar requisitos incompletos e/ou inconsistentes quando o protótipo é desenvolvido.
3. Um sistema operante, embora limitado, se torna rapidamente disponível, a fim de mostrar a viabilidade e a utilidade da aplicação para a gerência.
4. O protótipo pode ser utilizado como uma base para escrever a especificação para um sistema com qualidade de produção.

Desenvolver um protótipo normalmente leva a melhorias na especificação do sistema. Uma vez disponível um protótipo, ele pode também ser utilizado para outros propósitos (Ince e Hekmatpour, 1987):

1. *Treinamento de usuário* Um sistema protótipo pode ser utilizado para treinar os usuários antes que o sistema final seja entregue.

2. *Teste de sistema* Os protótipos podem executar testes *back-to-back*. Os mesmos casos de testes são submetidos ao protótipo e ao sistema que está em teste. Se ambos os sistemas apresentarem o mesmo resultado, o caso de teste não terá detectado um defeito. Se os resultados forem diferentes, isso poderá significar que há um defeito no sistema e as razões dessa diferença deverão ser investigadas.

Em um estudo de 39 diferentes projetos de prototipação, Gordon e Bieman (1995) constataram que os benefícios do uso da prototipação no processo de software eram:

1. melhoria na facilidade de uso do sistema;
2. maior aproximação do sistema com as necessidades dos usuários;
3. melhoria da qualidade do projeto;
4. melhoria na facilidade de manutenção, e
5. esforço de desenvolvimento reduzido.

O estudo sugere, portanto, que as melhorias na facilidade de uso e melhores requisitos de usuário, provenientes do uso de protótipo, não significam necessariamente um aumento geral dos custos de desenvolvimento de sistema. A prototipação geralmente aumenta os custos nos primeiros estágios do processo de software, mas reduz os custos posteriores. A razão principal disso é que o retrabalho durante o desenvolvimento é evitado, quando os clientes solicitam menos mudanças no sistema. Contudo, eles constataram que uma conseqüência negativa da prototipação é que o desempenho geral do sistema é, por vezes, degradado quando um código de protótipo ineficiente é reutilizado.

Um modelo de processo para o desenvolvimento de protótipo é mostrado na Figura 8.1. Os objetivos da prototipação devem ficar explícitos desde o princípio do processo. Esses objetivos podem ser: desenvolver um sistema para o protótipo de interface com o usuário, desenvolver um sistema para validar os requisitos funcionais do sistema ou desenvolver um sistema para mostrar a viabilidade da aplicação para a gerência. O mesmo protótipo não pode atender a todos os objetivos. Se os objetivos ficam implícitos, a gerência ou os usuários finais podem entender mal a função do protótipo. Conseqüentemente, eles podem não receber os benefícios que esperavam do desenvolvimento do protótipo.

O estágio seguinte no processo é decidir o que incluir no sistema protótipo e, talvez o mais importante, o que deve ser excluído do sistema. Para reduzir os custos de prototipação e acelerar a programação da entrega, pode-se excluir alguma funcionalidade do protótipo. É possível decidir diminuir os requisitos não funcionais, como o tempo de resposta e o uso de memória. O gerenciamento e o tratamento de erros podem ser ignorados ou ser rudimentares, a menos que o objetivo do protótipo seja estabelecer uma interface com o usuário. Os padrões de confiabilidade e qualidade de programa podem ser reduzidos.

O estágio final do processo é a avaliação do protótipo. Ince e Hekmatpour sugerem que esse é o estágio mais importante da prototipação. Deve ser feita uma provisão durante esse estágio para o treinamento de usuário, e os objetivos do protótipo devem ser utilizados na preparação de um plano para avaliação. Os usuários precisam de algum tempo para se sentirem à vontade com um sistema novo e para estabelecerem um padrão normal de uso. Quando estiverem utilizando o sistema normalmente, eles descobrirão erros e omissões nos requisitos.

Figura 8.1 ▶

Processo de desenvolvimento de protótipo.

8.1 Prototipação no processo de software

Como já foi discutido anteriormente, é difícil para os usuários finais prever como vão utilizar novos sistemas de software para dar apoio ao seu trabalho diário. Se esses sistemas forem grandes e complexos, provavelmente será impossível fazer essa avaliação antes de o sistema ser construído e colocado em operação.

Uma maneira de lidar com essa dificuldade é utilizar uma abordagem evolucionária para o desenvolvimento do sistema. Isso significa fornecer ao usuário um sistema que é incompleto e, então, modificá-lo e aumentá-lo até que os requisitos do usuário se tornem claros. Como alternativa, é possível decidir deliberadamente construir um protótipo descartável para ajudar na análise e na validação de requisitos. Depois da avaliação, o protótipo é descartado e um sistema com qualidade de produção é construído. A Figura 8.2 ilustra ambas as abordagens de desenvolvimento de protótipos.

A prototipação evolucionária começa com um sistema relativamente simples, que implementa os requisitos mais importantes do usuário. Esse sistema é ampliado e alterado à medida que novos requisitos vão sendo descobertos. Por fim, ele se torna o sistema que foi requerido. Não existe nenhuma especificação detalhada de sistema e, em muitos casos, pode não haver um documento formal de requisitos. A prototipação evolucionária é, atualmente, a técnica utilizada para o desenvolvimento de sites Web e aplicações de comércio eletrônico.

Por outro lado, a abordagem de prototipação descartável se destina a ajudar no aprimoramento e na classificação da especificação do sistema. O protótipo é escrito, avaliado e modificado. A avaliação do protótipo informa o desenvolvimento da especificação detalhada do sistema, que é incluída no documento de requisitos de sistema. Uma vez escrita a especificação, o protótipo não tem mais utilidade e é descartado.

Existe uma importante diferença entre os objetivos da programação evolucionária e da programação descartável:

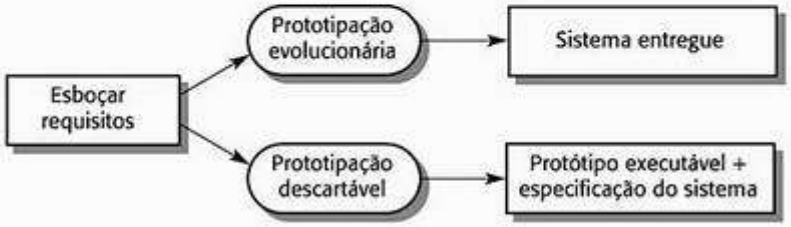
1. O objetivo da prototipação evolucionária é fornecer um sistema funcional aos usuários finais. Isso significa que, normalmente, o processo se inicia com os requisitos do usuário que são mais bem compreendidos e com os que têm maior prioridade. Os requisitos de menor prioridade e os mais vagos são implementados quando e se eles forem solicitados pelos usuários.

2. O objetivo da prototipação descartável é validar ou derivar os requisitos do sistema. É preciso começar com aqueles requisitos que não são bem compreendidos, porque é preciso saber mais sobre eles. Os requisitos que são diretos podem nunca ser prototipados.

Outra distinção importante entre essas abordagens é quanto ao gerenciamento da qualidade dos sistemas. Por definição, os protótipos descartáveis têm duração muito curta. Deve ser possível modificá-los muito rapidamente durante o desenvolvimento, mas a facilidade de manutenção a longo prazo não é exigida. O baixo nível de desempenho e confiabilidade pode ser aceitável em um protótipo descartável, desde que ele cumpra com sua função principal de ajudar no entendimento dos requisitos.

Figura 8.2 ▶

A prototipação evolucionária e a prototipação descartável.



Por outro lado, os protótipos que evoluem para o sistema final devem ser desenvolvidos com os mesmos padrões de qualidade organizacional de qualquer outro software. Eles devem ter sólida estrutura, de modo que possam receber manutenção por muitos anos. Eles devem ser confiáveis e eficientes e se adequar a padrões organizacionais relevantes.

8.1.1 Prototipação evolucionária

A prototipação evolucionária se baseia na idéia de desenvolver uma implementação inicial, expondo-a aos comentários dos usuários e aperfeiçoando-a ao longo de muitos estágios, até que um sistema adequado tenha sido desenvolvido (Figura 8.3). Essa abordagem de desenvolvimento foi inicialmente utilizada para os sistemas (por exemplo, os sistemas de AI – *artificial intelligence* ou inteligência artificial) que são difíceis ou impossíveis de especificar. Contudo, atualmente essa se tornou uma técnica importante de desenvolvimento de software. A prototipação evolucionária faz parte das técnicas de RAD (*rapid application development* – desenvolvimento rápido de aplicações) e de JAD (*joint application development* – desenvolvimento conjunto de aplicações) (Millington e Stapleton, 1995; Wood e Silver, 1995; Stapleton, 1997) ou tem muito em comum com essas técnicas.

Existem duas vantagens principais em adotar essa abordagem para o desenvolvimento de software:

1. *Rápido fornecimento do sistema* Como discuti na introdução do livro, o ritmo das mudanças nos negócios significa que é essencial que o apoio ao software seja disponibilizado rapidamente. Em alguns casos, o rápido fornecimento e a facilidade de uso são mais importantes do que os detalhes de funcionalidade ou a facilidade de manutenção de software a longo prazo.
2. *Compromisso do usuário com o sistema* O envolvimento de usuários com o processo de desenvolvimento não quer dizer apenas que o sistema tem maior possibilidade de atender a seus requisitos, mas também significa que os usuários finais do sistema assumiram um compromisso e provavelmente vão querer contribuir para que ele funcione.

Há diferenças de detalhes entre os métodos particulares de rápido desenvolvimento de software, mas todos compartilham algumas características fundamentais:

1. O processo de especificação, projeto e implementação são intercalados. Não existe especificação detalhada de sistema e a documentação de projeto produzida depende normalmente das ferramentas utilizadas para implementar o sistema. O documento de requisitos do usuário somente define as características mais importantes do sistema.
2. O sistema é desenvolvido em uma série de estágios. Os usuários finais e outros *stakeholders* são envolvidos no projeto e na avaliação, a cada estágio. Eles podem propor mudanças no software e novos requisitos, que devem ser implementados em uma versão posterior do sistema.

Figura 8.3 ▶
Prototipação evolucionária.

3. As técnicas para o desenvolvimento rápido de sistema são utilizadas (veja a Seção 8.2). Elas podem conter ferramentas CASE e linguagens de quarta geração (4GL).

4. As interfaces com o usuário do sistema são, geralmente, desenvolvidas utilizando-se um sistema de desenvolvimento interativo (veja a Seção 8.3), que permite que o projeto de interface seja criado rapidamente, mediante o desenho e a colocação de ícones nas interfaces.

As abordagens com base na prototipação evolucionária e na especificação para o desenvolvimento de software diferem em sua visão da verificação e da validação. A verificação é o processo de conferir se um programa atende a suas especificações. Como não existe especificação detalhada para o protótipo, a verificação é, por conseguinte, impossível.

A validação deve mostrar que o programa é adequado ao propósito previsto, em vez de sua conformidade com uma especificação. Isso também é difícil sem uma especificação detalhada, uma vez que não há nenhuma declaração explícita de propósito. Os usuários finais envolvidos no processo podem estar felizes com o sistema, mas outros *stakeholders* podem se sentir excluídos e insatisfeitos com o sistema quanto ao atendimento de seus propósitos.

Portanto, a verificação e a validação de um sistema que foi desenvolvido com o uso de prototipação evolucionária podem somente verificar se o sistema é adequado, isto é, se ele é suficientemente bom para o propósito previsto. A adequação, naturalmente, não é mensurável de imediato, e somente podem ser feitos julgamentos subjetivos da adequação de um programa. Isso não invalida sua utilidade; o desempenho humano não pode ser garantido como correto, mas ficamos satisfeitos se o desempenho for adequado à tarefa em questão. Contudo, como será discutido a seguir, isso não causa problemas nos casos em que os sistemas são desenvolvidos para clientes por um fornecedor externo de desenvolvimento de software.

Existem três problemas principais com a prototipação evolucionária, que são particularmente importantes quando são desenvolvidos sistemas grandes, com longo tempo de duração:

1. *Problemas de gerenciamento* As estruturas de gerenciamento de software para sistemas de grande porte são estabelecidas para lidar com um modelo de processo de software que gera produtos regulares, a fim de avaliar o progresso. Os protótipos evoluem tão rapidamente que não é eficaz, sob o ponto de vista do custo, produzir um grande volume de documentação para o sistema. Além disso, o desenvolvimento rápido de protótipos pode requerer o uso de tecnologias que não são familiares. Os gerentes podem achar difícil utilizar os funcionários já existentes, porque eles não apresentam essas habilidades.

2. *Problemas de manutenção* A continuidade das mudanças tende a corromper a estrutura do protótipo de sistema. Isso significa que qualquer um que não seja do grupo de desenvolvedores originais provavelmente achará difícil compreendê-la. Além disso, se uma tecnologia especializada for utilizada para apoiar o desenvolvimento rápido de um protótipo, ela pode se tornar obsoleta. Portanto, encontrar pessoas que tenham o conhecimento necessário para fazer a manutenção do sistema pode ser difícil.

3. *Problemas contratuais* O modelo normal de contrato entre um cliente e um desenvolvedor de software se baseia em uma especificação de sistema. Quando não houver essa especificação, pode ser difícil estabelecer um contrato para o desenvolvimento do sistema. Os clientes podem não se sentir à vontade com um contrato que simplesmente paga os desenvolvedores pelo tempo gasto no projeto, uma vez que isso pode levar a serviços prolongados e ultrapassar o orçamento definido; os desenvolvedores certamente não estarão dispostos a aceitar um contrato com preço fixo, uma vez que não podem controlar as mudanças requeridas pelos usuários finais.

Essas dificuldades significam que os clientes devem estar cientes do uso da prototipação evolucionária como uma técnica de desenvolvimento. Ela possibilita o rápido desenvolvimento e o fornecimento de sistemas pequenos e de porte médio. Os custos de desenvolvimento de sistema podem ser reduzidos e a facilidade de uso é melhorada. Se os usuários estiverem envolvidos no desenvolvimento, o sistema provavelmente deverá estar adequado a suas verdadeiras necessidades. Contudo, as organizações que utilizam essa abordagem devem aceitar que o tempo de duração do sistema deverá ser relativamente curto. À medida que os problemas de manutenção aumentam, o sistema terá de ser substituído ou completamente reescrito. Para grandes sistemas, que podem envolver uma série de diferentes fornecedores subcontratados, os problemas de gerenciamento da prototipação evolucionária se tornam intratáveis. Quando são desenvolvidos protótipos para partes desses grandes sistemas, eles devem ser protótipos descartáveis.

O desenvolvimento incremental (Figura 8.4) evita alguns problemas da mudança constante que caracterizam a prototipação evolucionária. Uma arquitetura geral de sistema é estabelecida no início do processo, a fim de atuar como um *framework*. Os componentes de sistema são desenvolvidos de maneira incremental e entregues dentro desse *framework*. Uma vez validados e entregues, nem o *framework* nem os componentes são modificados, a menos que sejam descobertos erros. O feedback dos usuários sobre os componentes fornecidos, contudo, pode influenciar no projeto de componentes programados para entrega posterior.

O desenvolvimento incremental é mais fácil de ser gerenciado do que a prototipação evolucionária, quando os padrões normais de processo de software são seguidos. Planos e documentação devem ser produzidos para cada novo estágio do sistema. Esse processo permite algum feedback dos usuários no início do processo e limita os erros de sistema, quando a equipe de desenvolvimento não está preocupada com interações entre partes inteiramente diferentes do sistema de software. Uma vez concluído um novo estágio do sistema, suas interfaces são 'congeladas'. Os estágios posteriores têm de se adaptar a essas interfaces e podem ser testados em relação a elas.

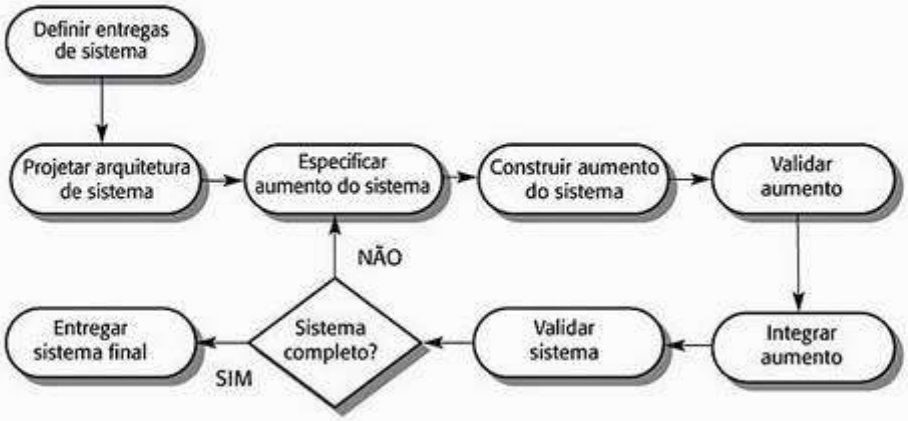
8.1.2 Prototipação descartável

Um modelo de processo de software com base em um estágio inicial de prototipação descartável está ilustrado na Figura 8.5. Essa abordagem amplia o processo de análise de requisitos, com a intenção de reduzir os custos totais do ciclo de vida. A função principal do protótipo é esclarecer os requisitos e fornecer informações adicionais para os gerentes avaliarem riscos de processo. Depois da avaliação, o protótipo é descartado. Ele não é utilizado como base para o desenvolvimento posterior de sistema.

Essa abordagem para a prototipação de sistemas é comumente utilizada para os sistemas de hardware. O protótipo é utilizado para verificar o projeto, antes que dispendiosos encar-

Figura 8.4 ▶

Um processo de desenvolvimento incremental.



gos referentes à fabricação do sistema tenham sido considerados. Um protótipo de sistema eletrônico pode ser desenvolvido utilizando-se componentes de prateleira, antes que sejam feitos investimentos em circuitos integrados de aplicação especial para implementar a versão de produção do sistema.

Contudo, o protótipo descartável de software não é normalmente utilizado para a validação de projeto, mas para ajudar a desenvolver os requisitos do sistema. O projeto de protótipo, freqüentemente, é bem diferente daquele do sistema final. O sistema deve ser desenvolvido tão rapidamente quanto possível, de modo que os usuários possam fornecer o feedback de sua experiência com o protótipo, para que este seja utilizado no desenvolvimento da especificação do sistema. A funcionalidade pode ser retirada do protótipo descartável quando essas funções são bem compreendidas, os padrões de qualidade podem ser reduzidos e os critérios de desempenho, ignorados. A linguagem de prototipação, muitas vezes, será diferente da linguagem final de implementação do sistema.

O modelo de processo, na Figura 8.5, considera que o protótipo é desenvolvido a partir de um esboço da especificação de sistema, fornecido para experimentação e, em seguida, modificado, até que o cliente esteja satisfeito com sua funcionalidade. Nesse estágio, um modelo de processo de software por fases é iniciado, uma especificação é derivada do protótipo e o sistema é reimplementado em uma versão final de produção. Os componentes do protótipo podem ser reutilizados no sistema com qualidade de produção e, assim, os custos de produção podem ser reduzidos.

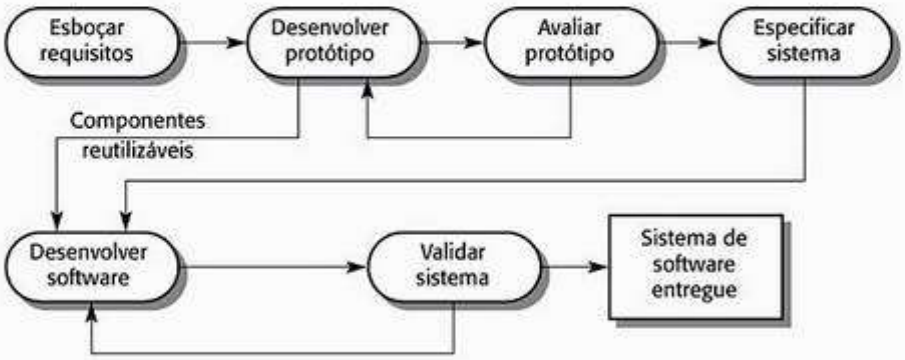
Em vez de derivar uma especificação do protótipo, algumas vezes, sugere-se que a especificação do sistema possa ser a própria implementação do protótipo. A instrução dada ao fornecedor do software poderia ser simplesmente 'escreva um sistema como este'. Existem vários problemas com essa abordagem:

1. Características importantes podem ter sido excluídas do protótipo, para simplificar uma implementação rápida. Na verdade, pode não ser possível fazer o protótipo de algumas das partes mais importantes do sistema, como as funções cruciais de segurança.
2. Uma implementação não tem apoio legal como um contrato entre o cliente e o fornecedor.
3. Os requisitos não funcionais, como os relativos à confiabilidade, à capacidade e à segurança, não podem ser adequadamente testados em uma implementação de protótipo.

Um problema geral da prototipação descartável executável é que o modo de utilização do protótipo pode não corresponder à maneira como o sistema final entregue é utilizado. O responsável pelo teste do protótipo pode estar particularmente interessado no sistema e pode não ser um típico representante de usuário do sistema. O tempo de treinamento durante a avaliação do protótipo pode ser insuficiente. Se o protótipo for lento, os avaliadores poderão ajustar seu modo de trabalho e evitar essas características de sistema que tenham tempos de resposta demorados. Quando é fornecido com melhor tempo de resposta no sistema final, os avaliadores podem utilizá-lo de modo diferente.

Figura 8.5 ▶

Um processo de software com prototipação descartável.



Às vezes, os desenvolvedores sofrem pressões dos gerentes para que entreguem protótipos descartáveis para uso, particularmente quando há atrasos na entrega da versão final do software. Contudo, de modo geral isso não é aconselhável pelas seguintes razões:

1. Pode ser impossível ajustar o protótipo para atender aos requisitos não funcionais, como os requisitos de desempenho, proteção, robustez e confiabilidade, que foram ignorados durante o desenvolvimento do protótipo.
2. Mudanças rápidas durante a prototipação inevitavelmente significam que o protótipo não está documentado. A única especificação de projeto é o código do protótipo. Isso não é suficientemente bom para a manutenção a longo prazo.
3. As mudanças feitas durante o desenvolvimento do protótipo provavelmente terão degradado a estrutura do sistema. O sistema terá uma manutenção difícil e dispendiosa.
4. Os padrões de qualidade organizacional são, normalmente, diminuídos para o desenvolvimento do protótipo.

Os protótipos descartáveis não precisam ser protótipos executáveis de software para que sejam úteis no processo de engenharia de requisitos. Os modelos com base em papel, da interface com o usuário de sistema (Rettig, 1994), se mostraram efetivos em ajudar usuários a aprimorar um projeto de interface e trabalhar em cenários de uso comum. Seu desenvolvimento é muito barato e eles podem ser construídos em poucos dias. Uma extensão dessa técnica é um protótipo do tipo 'Wizard of Oz' ('Mágico de Oz'), em que somente a interface com o usuário é desenvolvida. Os usuários interagem com essa interface, mas seus pedidos são transmitidos para uma pessoa que os interpreta e fornece a resposta apropriada. Essas abordagens de prototipação são discutidas em Sommerville e Sawyer (1997).

8.2 Técnicas de prototipação rápida

Técnicas de prototipação rápida são técnicas de desenvolvimento que enfatizam a rapidez do fornecimento, em vez de outras características do sistema, como o desempenho, a facilidade de manutenção ou a confiabilidade. Existem três técnicas de desenvolvimento rápido que são práticas para a prototipação de nível industrial:

1. desenvolvimento com linguagem dinâmica de alto nível;
2. programação de banco de dados, e
3. montagem de componentes e aplicações.

Por conveniência, descrevo essas técnicas em seções separadas. Na prática, contudo, todas são freqüentemente utilizadas no desenvolvimento de um protótipo de sistema. Por exemplo, uma linguagem de programação de banco de dados pode ser utilizada para extrair dados, com processamento detalhado que empregue componentes reutilizáveis. A interface com o usuário de sistema pode ser definida utilizando-se programação visual. Luqi (1992) descreve como essa abordagem mista foi utilizada para criar um protótipo de sistema de comando e controle.

O desenvolvimento de protótipos é, hoje em dia, normalmente apoiado por um conjunto de ferramentas que contêm, pelo menos, duas dessas técnicas. Por exemplo, o sistema Smalltalk VisualWorks aceita uma linguagem de nível muito alto e fornece muitos componentes reutilizáveis que podem ser incluídos em aplicações. O Lotus Notes inclui apoio à programação de banco de dados utilizando uma linguagem de nível muito elevado e componentes reutilizáveis, que podem ser vinculados a operações de banco de dados.

Atualmente, a maioria dos sistemas de prototipação é compatível com uma abordagem de programação visual, na qual alguma parte, ou todo o protótipo, é desenvolvida interativamente. Em vez de escrever um programa seqüencial, o desenvolvedor do protótipo manipula ícones gráficos que representam funções, dados ou componentes de interface com o usuário e associa scripts de processamento com esses ícones. Um programa executável é gerado automaticamente a partir da representação visual do sistema. Isso simplifica o desenvolvimento do programa e reduz custos de prototipação. Discuto mais detalhadamente a programação visual na Seção 8.2.3, que aborda o reuso de componentes e de aplicações.

8.2.1 Desenvolvimento com linguagem dinâmica de alto nível

As linguagens dinâmicas de alto nível são linguagens de programação que incluem poderosos recursos de gerenciamento de dados em *run-time*. Elas simplificam o desenvolvimento de programas porque reduzem muitos problemas de alocação de armazenamento e gerenciamento. O sistema de linguagens inclui recursos que normalmente precisam ser construídos a partir de construções mais primitivas, em linguagens como Ada ou C. Exemplos de linguagem de nível muito alto são Lisp (com base em estruturas de lista), Prolog (com base em lógica) e Smalltalk (com base em objetos).

Até relativamente pouco tempo atrás, as linguagens dinâmicas de nível muito alto não eram largamente utilizadas para o desenvolvimento de grandes sistemas, porque elas necessitavam de um grande sistema de suporte de *run-time*. Esse suporte de *run-time* aumenta as necessidades de armazenamento e reduz as velocidades de execução de programas escritos nessa linguagem. Contudo, o aumento da capacidade e o custo reduzido de hardware de computador fizeram com que esses fatores passassem a ter menos importância.

Isso significa que, para muitas aplicações de negócios, essas linguagens podem substituir linguagens de programação imperativas, como C, Cobol e Ada. Java é claramente uma linguagem de programação importante, com raízes em C++, e ainda incorpora muitas características de Smalltalk, como a independência de plataforma e o gerenciamento automático de armazenamento. Java proporciona muitas vantagens das linguagens de nível muito alto, com o rigor e as oportunidades de otimização de desempenho, oferecidos pelas linguagens convencionais de terceira geração. Muitos componentes Java reutilizáveis estão disponíveis e, assim, ela é claramente uma linguagem muito adequada para a prototipação evolucionária.

A Figura 8.6 mostra as linguagens dinâmicas que são mais comumente utilizadas para a prototipação. Ao escolher uma linguagem de prototipação, é preciso ter em mente uma série de questões:

1. *Qual é o domínio de aplicação do problema?* Como mostra a Figura 8.6, diferentes linguagens são mais adequadas para diferentes domínios de aplicação. Se você deseja fazer protótipos de aplicações que envolvam o processamento de linguagem natural (digamos), uma linguagem como Lisp ou Prolog é mais adequada do que Java ou Smalltalk.

2. *Que interação com o usuário é exigida?* Diferentes linguagens fornecem diferentes níveis de apoio para a interação com o usuário. Algumas linguagens, como Smalltalk e Java, são bem integradas com Web browsers, enquanto outras, como Prolog, são mais adequadas a interfaces com base em texto.

3. *Que ambiente de apoio é fornecido com a linguagem?* Ambientes de apoio bem desenvolvidos, com muitas ferramentas e fácil acesso a componentes reutilizáveis, simplificam o processo de desenvolvimento de protótipos.

As linguagens dinâmicas de alto nível podem ser utilizadas em combinação, a fim de criar um protótipo de sistema. Diferentes partes do sistema podem ser programadas em diferentes linguagens e um *framework* de comunicação pode ser estabelecido entre essas partes. Zave (1989)

Figura 8.6 ▶

Linguagens de alto nível para prototipação.

Linguagem	Tipo	Domínio de aplicação
Smalltalk	Orientada a objetos	Sistemas interativos
Java	Orientada a objetos	Sistemas interativos
Prolog	Lógica	Processamento simbólico
Lisp	Com base em listas	Processamento simbólico

descreve essa abordagem de desenvolvimento na prototipação de um sistema de rede de telefonia. Quatro diferentes linguagens foram utilizadas: Prolog, para prototipação de banco de dados; Awk (Aho *et al.*, 1988), para faturamento; CSP (Hoare, 1985), para especificação de protocolo, e PAISLey (Zave e Schell, 1986), para simulação de desempenho.

Não existe uma linguagem ideal para a prototipação de grandes sistemas quando as diferentes partes do sistema são tão diversificadas. A vantagem de uma abordagem com linguagens combinadas é que a linguagem mais apropriada para uma parte lógica da aplicação pode ser escolhida, apressando, assim, o desenvolvimento do protótipo. A desvantagem é que pode ser difícil estabelecer um *framework* de comunicação que permita que várias linguagens se comuniquem. As entidades utilizadas nas diferentes linguagens são muito diferentes. Conseqüentemente, seções longas de código podem ser necessárias para traduzir uma entidade de uma linguagem para outra.

8.2.2 Programação de banco de dados

O desenvolvimento evolucionário é, atualmente, uma técnica-padrão para implementar aplicações pequenas e de porte médio no domínio de sistemas de negócios. A maioria das aplicações de negócios envolve a manipulação de dados a partir de um banco de dados e a produção de saídas que implicam organizar e formatar esses dados.

Para oferecer suporte ao desenvolvimento dessas aplicações, todos os sistemas de gerenciamento de bancos de dados comerciais, atualmente, aceitam a programação de bancos de dados. A programação de bancos de dados é realizada utilizando-se uma linguagem especial, que envolve o conhecimento desse banco de dados e inclui operações que são ajustadas à manipulação de banco de dados. O ambiente de apoio à linguagem fornece ferramentas para proporcionar compatibilidade com a definição de interface com o usuário, a computação numérica e a geração de relatórios. O termo *linguagem de quarta geração* (4GL) é utilizado para se referir à linguagem de programação de bancos de dados, como também a seu ambiente de apoio.

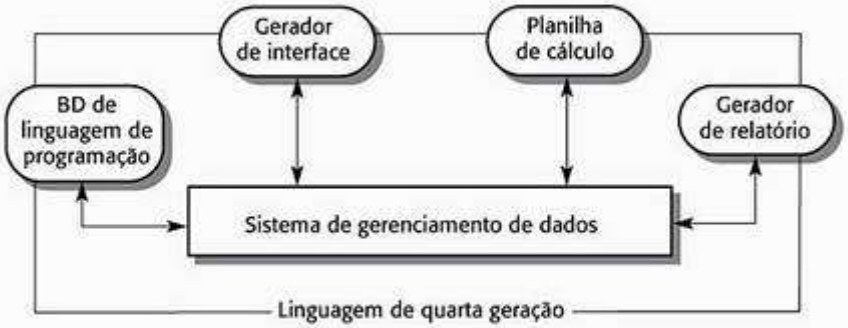
As linguagens de quarta geração são bem-sucedidas, porque há grande quantidade de aspectos comuns nas aplicações de processamento de dados. Essencialmente, essas aplicações se ocupam de atualizar bancos de dados e de produzir relatórios a partir das informações em bancos de dados. Formulários-padrão são utilizados para entradas e saídas. As 4GLs são ajustadas para produzirem aplicações interativas, que são compatíveis para abstrair informações obtidas em um banco de dados organizacional, apresentando o resultado a usuários finais em seu terminal ou estação de trabalho e depois atualizando o banco de dados com mudanças introduzidas pelos usuários. A interface com o usuário geralmente consiste em um conjunto de formulários-padrão ou planilhas de cálculo.

As ferramentas incluídas em um ambiente 4GL (Figura 8.7) são:

1. Uma linguagem de consulta de banco de dados que, hoje em dia, é geralmente SQL (Date e Darwen, 1997). Ela pode ser fornecida diretamente ou gerada automaticamente a partir de formulários preenchidos por um usuário final.

Figura 8.7

Componentes de linguagem de quarta geração.



2. Um gerador de interface, que é utilizado para criar formulários para a entrada e a exibição de dados.
3. Uma planilha de cálculos, para a análise e a manipulação de informações numéricas.
4. Um gerador de relatórios, que é utilizado para definir e criar relatórios a partir de informações em bancos de dados.

A maioria das aplicações de negócios utiliza formulários estruturados para entrada e saída, de modo que as 4GLs fornecem poderosos recursos para a definição de tela e a geração de relatórios. As telas são, freqüentemente, definidas como uma série de relatórios vinculados (em uma aplicação que estudamos, havia 137 diferentes definições de formulários); portanto, o sistema de geração de telas deve oferecer:

1. *definição interativa de formulários*, em que o desenvolvedor define os campos a serem exibidos e como eles devem ser organizados;
2. *vinculação de formulários*, em que o desenvolvedor pode especificar que determinadas entradas acarretam a exibição de outros formulários, e
3. *verificação de campo*, em que o desenvolvedor define a variação permitida para a inserção de valores em campos de formulários.

A maioria das 4GLs, atualmente, aceita o desenvolvimento de interfaces de banco de dados com base em Web browsers. Isso permite que o banco de dados seja acessado a partir de qualquer lugar, com uma conexão válida de Internet, reduzindo os custos de treinamento e os custos de software e permitindo que os usuários externos tenham acesso a um banco de dados. Contudo, as limitações inerentes aos Web browsers e aos protocolos da Internet significam que essa abordagem pode ser inadequada para sistemas em que seja necessário obter respostas muito rápidas e interativas.

O desenvolvimento com base nas 4GLs pode ser utilizado para a prototipação evolucionária ou em conjunto com uma análise com base em métodos, na qual modelos de sistema são utilizados para gerar o protótipo de sistema. A estrutura que as ferramentas CASE impõem na aplicação e na documentação associada significa que os protótipos evolucionários desenvolvidos com o uso dessa abordagem devem ser de manutenção mais fácil do que os sistemas desenvolvidos manualmente. As ferramentas CASE podem gerar SQL ou código em uma linguagem de nível mais inferior, como Cobol. Forte (1992) descreve uma série de ferramentas desse tipo em uma breve análise sobre as linguagens de quarta geração.

Embora as 4GLs sejam muito adequadas para o desenvolvimento de protótipos, existem algumas desvantagens em utilizá-las para sistemas de produção. Os programas escritos em uma 4GL são, em geral, mais lentos do que programas similares em linguagens convencionais de programação e normalmente requerem muito mais memória. Por exemplo, estive envolvido em um experimento no qual reescrever um programa 4GL em C++ resultou em uma redução de 50 por cento nos requisitos de memória. O programa em C também foi executado dez vezes mais rápido do que o sistema em 4GL.

Embora elas claramente reduzam os custos de desenvolvimento de sistemas, o efeito das 4GLs nos custos totais do ciclo de vida de grandes sistemas de processamento de dados ainda não está claro. Os programas tendem a ser desestruturados e de difícil manutenção, e as 4GLs não são padronizadas. Os custos pós-implantação podem, portanto, ser altos. Problemas específicos podem surgir quando sistemas que utilizem 4GL precisam passar pelo processo de reengenharia. Não existe padronização ou uniformidade entre as linguagens de quarta geração. Isso significa que os usuários podem ter de reescrever programas, porque a linguagem em que eles foram escritos originalmente está obsoleta.

8.2.3 Montagem de componentes e aplicações

O tempo necessário para desenvolver um sistema poderá ser reduzido, se muitas partes desse sistema puderem ser reutilizadas, em vez de serem projetadas e implementadas. Protótipos poderão ser construídos rapidamente, se tivermos um conjunto de componentes reutilizáveis e um mecanismo para compor esses componentes em sistemas. O mecanismo de composição deve conter recursos de controle e um mecanismo para a comunicação de componentes. Essa abordagem é ilustrada na Figura 8.8.

A prototipação com componentes reutilizáveis envolve desenvolver uma especificação de sistema, levando em conta quais componentes reutilizáveis estão disponíveis, e isso pode significar que é preciso obter alguma conciliação entre os requisitos. A funcionalidade dos componentes disponíveis pode não ter uma adequação precisa com os requisitos dos usuários. Contudo, os requisitos de usuário são freqüentemente bastante flexíveis e, assim, na maioria dos casos, essa abordagem pode ser utilizada para o desenvolvimento de protótipos.

O desenvolvimento de protótipos que emprega o reuso pode ser aceito em dois níveis:

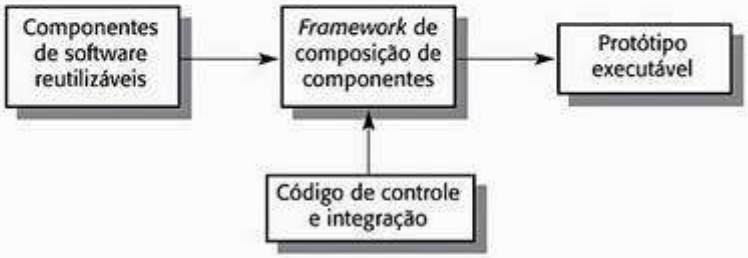
1. No nível de aplicações, em que sistemas inteiros de aplicações são integrados com o protótipo, de modo que sua funcionalidade possa ser compartilhada. Por exemplo, se o protótipo requer uma capacidade de processamento de textos, isso pode ser fornecido pela integração de um sistema-padrão de processamento de textos. Aplicações como o Microsoft Office suportam ligações de aplicativos.

2. No nível de componentes, em que componentes individuais são integrados dentro de um *framework*-padrão a fim de implementar o sistema. Esse *framework*-padrão pode ser uma linguagem de *scripting*, que é projetada para o desenvolvimento evolucionário, como Visual Basic, TCL/TK (Ousterhout, 1994), Python (Lutz, 1996) ou Perl (Wall *et al.*, 1996). Como alternativa, pode ser um *framework* de integração de componentes mais genérico, com base em CORBA, DCOM ou JavaBeans (Sessions, 1997; Orfali e Harkey, 1998; Pope, 1998).

O reuso de aplicações permite o acesso a toda a funcionalidade de uma aplicação. Se a aplicação também fornece recursos de *scripting* ou de composição (por exemplo, macros do Excel), esses recursos podem ser utilizados para desenvolver alguma funcionalidade de protótipo. Uma metáfora de documento composto é útil para compreender essa abordagem para

Figura 8.8 ▶

Composição de componente reutilizável.



o desenvolvimento de protótipos. Os dados processados pelo sistema do protótipo são organizados em um documento composto, que atua como um contêiner para vários objetos diferentes. Esses objetos contêm diferentes tipos de dados (como uma tabela, um diagrama e um formulário), que podem ser processados por diferentes aplicações. Eles são vinculados e tipificados, de maneira que o acesso a um objeto resulta em que a aplicação associada seja iniciada.

A Figura 8.9 ilustra a noção de um protótipo de sistema constituído de um documento composto, que inclui elementos de texto, elementos de planilha de cálculo e arquivos de som. Os elementos de texto são processados pelo processador de textos, as tabelas, por aplicativos de planilhas de cálculo, e os arquivos de som, por um reprodutor de áudio. Quando um usuário do sistema acessa um objeto de um tipo particular, o aplicativo associado é chamado para fornecer funcionalidade ao usuário. Por exemplo, quando objetos do tipo som são acessados, o programa de reprodução de áudio é chamado para processar esses objetos.

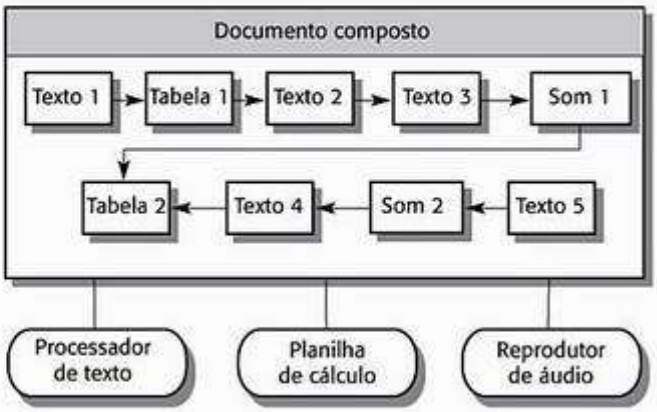
Para ilustrar o tipo de protótipo que pode ser desenvolvido utilizando-se essa abordagem, considere o processo de gerenciamento de requisitos, discutido no Capítulo 6. Um sistema de apoio ao gerenciamento de requisitos necessita de um meio de captar requisitos, armazenar esses requisitos, produzir relatórios, descobrir relações entre os requisitos e gerenciar essas relações como tabelas de rastreamento. Isso pode ser desenvolvido em protótipos, por meio da vinculação de um banco de dados (para armazenar os requisitos), um processador de textos (para obter os requisitos e formatar os relatórios), uma planilha de cálculos (para gerenciar as tabelas de facilidade de rastreamento) e um código especialmente escrito para encontrar relações entre os requisitos.

A principal vantagem dessa abordagem é que uma porção da funcionalidade do protótipo pode ser implementada rapidamente, a um custo muito baixo. Se os usuários do protótipo já estiverem familiarizados com as aplicações que o constituem, eles não precisarão aprender como utilizar esses novos recursos. Contudo, se eles não souberem utilizar as aplicações, o aprendizado poderá ser difícil, especialmente quando eles puderem ser confundidos com uma funcionalidade que não seja necessária. É possível que haja problemas de desempenho com o protótipo, por causa da necessidade de trocar de um sistema de aplicações para outro. Isso depende do apoio fornecido ao sistema operacional. O apoio mais amplamente utilizado para essa abordagem é o mecanismo OLE (*object linking and embedding* – vinculação e inclusão de objetos) da Microsoft (Sessions, 1997).

Nem sempre é possível ou sensato reutilizar aplicações inteiras. O desenvolvimento com reuso depende de sofisticados componentes reutilizáveis, que podem ser funções ou objetos que realizam operações específicas, como classificação, busca e exibição. O sistema de protótipos é desenvolvido com a definição de uma estrutura geral de controle e, então, com a integração dos componentes com essa estrutura. Se os componentes que realizam a função requerida não

Figura 8.9

Vinculação de aplicativos.



estiverem disponíveis, será desenvolvido um código de propósito especial; se for apropriado, ele estará disponível para futuro reuso.

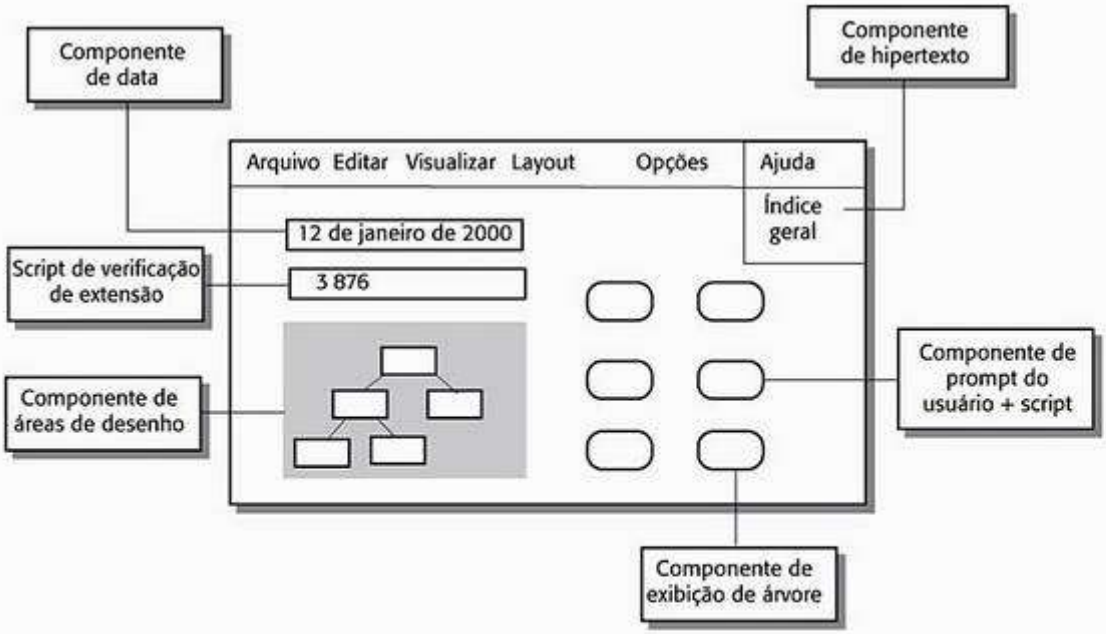
Os sistemas de desenvolvimento visual, como o Visual Basic, são compatíveis com essa abordagem baseada no reuso, para o desenvolvimento de aplicações. Os programadores de aplicações constroem o sistema interativamente, pela definição da interface em termos de telas, campos, botões e menus. Esses itens são denominados e scripts de processamento são associados com partes individuais da interface (por exemplo, um botão denominado 'Simular'). Esses scripts podem ser chamados de componentes reutilizáveis, código de propósito especial ou uma mistura de ambos.

Ilustro essa abordagem na Figura 8.10, que mostra uma tela da aplicação com menus na parte superior, campos de entrada (os campos brancos à esquerda do display), campos de saída (o campo sombreado à esquerda do display) e botões, que são os retângulos com cantos arredondados, à direita do display. Quando essas entidades são posicionadas no display pelo sistema de construção de protótipos, o desenvolvedor define que componente reutilizável deve ser associado a eles ou escreve um fragmento de programa para realizar o processamento necessário. Na Figura 8.10, mostro o componente associado a alguns elementos do display.

O Visual Basic é um exemplo de uma família de linguagens chamada de linguagens de *scripting* (Ousterhout, 1998). As linguagens de *scripting* são linguagens *typeless* de alto nível, que são projetadas para integrar componentes e criar sistemas. Um exemplo inicial de uma linguagem de *scripting* foi o shell do Unix (Bourne, 1978) e, desde seu desenvolvimento, uma série de outras linguagens de *scripting* mais poderosas foram criadas (Ousterhout, 1994; Lutz, 1996; Wall *et al.*, 1996). As linguagens de *scripting* incluem estruturas de controle e caixas de ferramentas gráficas, e Ousterhout (1998) ilustra que elas reduzem drasticamente o tempo necessário para o desenvolvimento de sistema.

Essa abordagem para o desenvolvimento de sistemas permite o rápido desenvolvimento de aplicações relativamente simples e pequenas, que podem ser construídas por uma pessoa ou por uma pequena equipe. Para grandes sistemas, que devem ser desenvolvidos por equipes maiores, a organização é mais difícil. Não existe uma arquitetura explícita de sistema e, freqüentemente, ocorrem complexas dependências entre as diferentes partes do sistema. Isso causa dificuldades, quando são necessárias mudanças. Essa abordagem também é limitada a um conjunto específico de objetos de interação e, assim, pode ser difícil implementar interfaces com

Figura 8.10 ▼
Programação visual com reuso.



o usuário que não sejam padronizadas. O desenvolvimento mais genérico com base em componentes em torno de uma arquitetura de *framework*, como será discutido no Capítulo 14, é mais apropriado para grandes sistemas.

8.3 Prototipação de interface com o usuário

As interfaces gráficas com o usuário se tornaram, atualmente, a norma para sistemas interativos. O esforço envolvido na especificação, no projeto e na implementação de uma interface com o usuário representa parte significativa dos custos de desenvolvimento de aplicações. Como será discutido no Capítulo 15, os projetistas não devem impor seus pontos de vista a respeito de uma interface com o usuário que seja aceitável pelos usuários. O usuário deve tomar parte do processo de projeto de interface. Essa compreensão leva a uma abordagem de projeto chamada de projeto centrado no usuário (Norman e Draper, 1986), que depende da prototipação de interface e do envolvimento do usuário com o processo de projeto de interface.

Do ponto de vista da engenharia de software, a prototipação é parte essencial do processo de projeto de interface. Devido à natureza dinâmica das interfaces com o usuário, as descrições textuais e os diagramas não são suficientemente bons para exprimir os requisitos da interface com o usuário. Portanto, a prototipação evolucionária, com o envolvimento do usuário final, é a única maneira sensata de desenvolver interfaces gráficas com o usuário, destinadas a sistemas de software.

Os geradores de interface são sistemas de projeto de telas gráficas, em que componentes de interface, como menus, campos, ícones e botões, são relacionados a partir de um menu e posicionados em uma interface. Como já foi discutido, os sistemas desse tipo são parte essencial de um sistema de programação de bancos de dados, e o Visual Basic (discutido anteriormente) baseou sua técnica-padrão de desenvolvimento nesse sistema. Shneiderman (1998) discute uma série desses sistemas. Os geradores de interface criam um programa bem estruturado, gerado a partir de uma especificação de interface. As iterações, que são parte inerente do desenvolvimento evolucionário, não degradam a estrutura do software, e a reimplementação não é necessária.

Milhões de pessoas têm acesso aos Web browsers. Eles aceitam uma linguagem de definição de páginas (HTML), que foi estendida a partir de uma simples linguagem de marcação de texto para uma notação abrangente, destinada à especificação de interface com o usuário. Botões, campos, formulários e tabelas podem ser incluídos nas páginas Web, assim como objetos de multimídia, que permitem o acesso a sons, vídeo e exibições de realidade virtual. Os scripts de processamento podem ser associados com objetos de interface com o usuário, com processamento realizado no cliente Web ou, de modo centralizado, no servidor Web.

Devido à grande disponibilidade de Web browsers e à eficiência do HTML e de suas capacidades de processamento associadas, cada vez mais as interfaces com o usuário estão sendo construídas como interfaces com base na Web. Como discutirei no Capítulo 26, que aborda os sistemas legados, essas interfaces não estão simplesmente limitadas a novos sistemas. As interfaces com base na Web estão substituindo as interfaces com base em formulários, para uma grande variedade de sistemas legados.

As interfaces com o usuário, com base na Web, podem ser prototipadas utilizando-se um editor-padrão de site Web, que é essencialmente um construtor de interface com o usuário. As entidades de página Web são definidas e posicionadas e ações são relacionadas com elas, seja com a utilização de recursos de HTML instalados (por exemplo, vinculação com outra página), seja com a utilização de scripts Java ou CGI.

PONTOS-CHAVE

- Um protótipo de sistema pode ser desenvolvido para dar aos usuários finais de sistema uma impressão concreta das capacidades desse sistema. O protótipo pode, portanto, ajudar a estabelecer e validar os requisitos de sistema.
- À medida que aumenta a necessidade de um rápido fornecimento de software, a prototipação está se tornando cada vez mais comum, como a técnica-padrão de desenvolvimento para sistemas pequenos e de porte médio, especialmente no domínio de negócios.
- A prototipação descartável envolve o desenvolvimento de um protótipo para compreender os requisitos do sistema. Na prototipação evolucionária, um protótipo evolui de várias versões até o sistema final.
- Ao implementar um protótipo descartável, primeiramente são desenvolvidas as partes menos compreendidas do sistema; em um protótipo evolucionário, são desenvolvidas as partes do sistema que são mais bem compreendidas.
- O desenvolvimento rápido é importante para os protótipos de sistemas. A fim de entregar um protótipo rapidamente, talvez seja necessário excluir algumas funcionalidades de sistema ou diminuir os requisitos não funcionais, como a rapidez de resposta e a confiabilidade.
- Entre as técnicas de prototipação estão o uso de linguagens de nível muito elevado, a programação de bancos de dados e a construção de um protótipo a partir de componentes reutilizáveis. Muitos ambientes de prototipação são compatíveis com uma abordagem de programação visual de desenvolvimento.
- As interfaces com o usuário deveriam sempre ser desenvolvidas com a prototipação, quando não é possível especificar essas interfaces efetivamente utilizando um modelo estático. Os usuários deveriam estar envolvidos na avaliação e na evolução do protótipo.

LEITURAS SUGERIDAS

"Scripting: higher-level programming for the 21st century". Uma visão geral das linguagens de *scripting*, pelo inventor da TCL/TK, que discute as vantagens dessa abordagem para o desenvolvimento rápido de aplicações. (J.K. Ousterhout, *IEEE Computer*, mar. 1998.)

"Rapid prototyping: lessons learned". Esse é um bom resumo das vantagens e das desvantagens da prototipação rápida de software. Tem como base uma análise empírica de uma série de diferentes tipos de sistema de software. (V. Scott Gordon e J. Bielman, *IEEE Software*, jan. 1995.)

"Twenty-two tips for a happier, healthier prototype". Conselhos muito importantes sobre a prototipação, apresentados sob a perspectiva de um projetista de interface com o usuário. (James Rudd e Scott Isensee, *ACM Interactions*, 1.1, jan. 1994.)

EXERCÍCIOS

8.1 Foi solicitado que você investigasse a viabilidade da prototipação no processo de desenvolvimento de software em sua organização. Escreva um relatório para seu gerente, discutindo as classes de projeto em que a prototipação deve ser utilizada e apresentando os custos e benefícios esperados a partir do uso da prototipação.

8.2 Explique por que, para o desenvolvimento de grandes sistemas, é recomendado que os protótipos sejam descartáveis.

8.3 Que características de linguagens, como Smalltalk e Lisp, contribuem para o apoio à prototipação rápida?

8.4 Sob quais circunstâncias você recomendaria que a prototipação deva ser utilizada como meio de validação dos requisitos de sistema?

8.5 Sugira as dificuldades que podem surgir quando se utiliza a prototipação em sistemas integrados de computador em tempo real.

8.6 Um gerente de software está envolvido no desenvolvimento de projeto de um sistema de apoio ao projeto de software, que aceita a tradução de requisitos de software para uma especificação formal de software. Comente as vantagens e as desvantagens das seguintes estratégias de desenvolvimento:

a) Desenvolva um protótipo descartável utilizando uma linguagem de prototipação, como Smalltalk. Avalie esse protótipo e revise os requisitos. Desenvolva o sistema final utilizando a linguagem C.

b) Desenvolva o sistema a partir dos requisitos existentes utilizando Java e modifique-o a fim de adaptá-lo a quaisquer requisitos modificados do usuário.

c) Desenvolva o sistema utilizando a prototipação evolucionária e empregando uma linguagem de prototipação como Smalltalk. Modifique o sistema de acordo com os requisitos do usuário e entregue o protótipo modificado.

8.7 Discuta a prototipação com a utilização de componentes reutilizáveis e sugira os problemas que podem surgir quando se emprega essa abordagem. Qual é o modo mais efetivo de especificar os componentes reutilizáveis?

8.8 Quais são as vantagens e as desvantagens de utilizar o mecanismo OLE, da Microsoft, para o desenvolvimento rápido de aplicações?

8.9 Uma entidade beneficente pediu que você fizesse o protótipo de um sistema que controle todas as doações recebidas. Esse sistema precisa manter os nomes e os endereços dos doadores, seus interesses específicos, a quantia doada e quando foi feita a doação. Quando a doação é superior a um certo valor, o doador pode estabelecer condições associadas à doação (por exemplo, ela deve ser gasta em um projeto específico), e o sistema deve manter o controle dessas doações e como elas foram empregadas. Discuta como você faria esse protótipo, tendo em mente que na entidade há trabalhadores pagos e voluntários. Muitos voluntários são aposentados que têm pouca ou nenhuma experiência com computadores.

8.10 Você desenvolveu um sistema de protótipo descartável para um cliente, que está muito feliz com ele. Contudo, ele sugere que não existe nenhuma necessidade de desenvolver outro sistema, mas que você deve fornecer o protótipo, e oferece um ótimo preço pelo sistema. Você sabe que pode haver futuros problemas com relação à manutenção do sistema. Discuta como você poderia responder a esse cliente.

9 Especificação formal

Objetivos

O objetivo principal deste capítulo é apresentar as técnicas de especificação formal que podem ser utilizadas para acrescentar detalhes à especificação de requisitos de um sistema. Depois de ler este capítulo, você poderá:

- compreender por que as técnicas de especificação formal ajudam a descobrir problemas em requisitos de sistema;
- compreender o uso de técnicas algébricas de especificação formal para definir especificações de interface, e
- compreender como técnicas formais baseadas em modelos podem ser utilizadas para a especificação de comportamento.

Sumário

9.1 Especificação formal no processo de software
9.2 Especificação de interface
9.3 Especificação de comportamento

Em disciplinas tradicionais de engenharia, como a engenharia civil e a elétrica, o progresso geralmente envolveu o desenvolvimento de melhores técnicas matemáticas. A indústria de engenharia não teve nenhuma dificuldade em aceitar a necessidade da análise matemática e em incorporá-la a seus processos. A análise matemática é uma parte rotineira do processo de desenvolvimento e validação de um projeto de produto.

Contudo, a engenharia de software não seguiu o mesmo caminho. Embora já tenham passado mais de 25 anos de pesquisa sobre o uso de técnicas matemáticas no processo de software, essas técnicas tiveram um impacto limitado. Os denominados 'métodos formais' de desenvolvimento de software não são amplamente utilizados no desenvolvimento de software industrial. A maioria das empresas de desenvolvimento de software não considera que seja eficaz, em termos de custo, aplicá-los em seus processos de software.

O termo 'métodos formais' envolve uma série de diferentes atividades, inclusive a especificação formal de sistemas, a análise e a comprovação da especificação, o desenvolvimento transformacional (discutido no Capítulo 3) e a verificação do programa. Todas essas atividades dependem de uma especificação formal do software, que é a especificação expressa em uma linguagem cujo vocabulário, sintaxe e semântica são formalmente definidos. Essa necessidade de uma definição formal significa que as linguagens de especificação precisam ter como base conceitos matemáticos, cujas propriedades tenham sido investigadas e sejam bem compreendidas. O ramo da matemática utilizado é chamado de matemática discreta e os conceitos matemáticos são obtidos a partir do conjunto formado por teoria, lógica e álgebra.

Na década de 80, a especificação formal e os métodos formais mais gerais eram vistos por muitos pesquisadores como o caminho mais provável para importantes melhorias na qualidade de software. Eles argumentavam que a análise detalhada e rigorosa, que é uma parte essencial dos métodos formais, levaria a programas com menos erros e que fossem mais adequados às necessidades dos usuários. Além disso, eles previram que, no século XXI, grande parte dos produtos de software seria desenvolvida com a utilização dos métodos formais.

É claro que essa previsão não foi cumprida e existe uma série de razões para isso:

1. *Sucesso da engenharia de software* O uso de métodos de engenharia de software, como os métodos estruturados, o gerenciamento de configuração e o encapsulamento de informações, nos processos de projeto e desenvolvimento, resultou em melhorias na qualidade de software. Isso contradiz as previsões no sentido de que as provas de programa são essenciais para a melhoria da qualidade.

2. *Mudanças de mercado* Na década de 80, a questão da qualidade de software era vista como o problema fundamental da engenharia de software. Contudo, desde então, a questão principal para muitas classes de desenvolvimento de software não é a qualidade, mas o tempo de disponibilização para o mercado. O software tem de ser entregue rapidamente e os clientes, com freqüência, estarão dispostos a aceitá-lo com algumas falhas, se sua entrega for rápida. Técnicas para o rápido desenvolvimento de software não são uma boa interface com as especificações formais. Naturalmente, a qualidade é um fator importante, mas ela deve ser alcançada dentro do contexto da entrega rápida.

3. *Abordagem limitada dos métodos formais* Os métodos formais não são, em geral, muito adequados para especificar as interfaces e as interações com o usuário. Como os componentes de interface com o usuário se tornaram uma parte cada vez maior de muitos sistemas, quaisquer benefícios obtidos pelo uso de métodos formais são limitados.

4. *Escalabilidade limitada dos métodos formais* Os métodos formais não se aplicam bem ao sistema em toda a sua extensão. Projetos bem-sucedidos que utilizaram essas técnicas têm seu uso limitado a importantes sistemas de kernel, que são relativamente pequenos. Esse problema foi exacerbado pela falta de apoio de ferramentas para essas técnicas.

Esses fatores significam que os riscos de se adotarem métodos formais na maioria dos projetos de software superam os possíveis benefícios de sua utilização. São muito grandes os cus-

tos e os problemas relacionados à introdução de métodos formais em processos de software. Contudo, a especificação formal é um excelente modo de descobrir erros de especificação e de apresentar a especificação do sistema sem encontrar ambigüidades. Todos os projetos bem-sucedidos, que utilizaram métodos formais, relataram menos erros no software fornecido.

Portanto, nos sistemas em que as falhas precisam ser evitadas, o emprego de métodos formais pode ser justificável e, provavelmente, é eficaz em termos de custo. O uso de métodos formais está aumentando em áreas especializadas de desenvolvimento de sistemas críticos, em que propriedades emergentes de sistema, como segurança, confiabilidade e proteção, são muito importantes. Esses sistemas fundamentais, como será discutido na Parte 4, têm custos de validação muito altos, e os custos de falhas no sistema são grandes e estão aumentando. Os métodos formais estão sendo utilizados porque eles podem reduzir esses custos.

Dentre os sistemas críticos, nos quais métodos formais foram aplicados com sucesso, destacam-se o sistema de informação de controle de tráfego aéreo (Hall, 1996), os sistemas de sinalização de estrada de ferro (Dehbonei e Mejia, 1995), os sistemas de naves espaciais (Easterbrook *et al.*, 1998) e os sistemas de controle médico (Jacky, 1995; Jacky *et al.*, 1997). Eles também são usados para a especificação de ferramentas de software (Neil *et al.*, 1998), a especificação de parte do sistema CICS, da IBM (Wordsworth, 1991), e um núcleo de sistema de tempo real (Spivey, 1990). O método Cleanroom, para o desenvolvimento de software (Mills *et al.*, 1987; Linger, 1994; Prowell *et al.*, 1999), que será discutido no Capítulo 19, utiliza argumentos formais de que o código cumpre com sua especificação. No Reino Unido, o uso de métodos formais é obrigatório, conforme definido pelo Ministério da Defesa, para sistemas em que a segurança é fundamental (MOD, 1995).

9.1 Especificação formal no processo de software

Na discussão sobre o processo de engenharia de requisitos, apresentada no Capítulo 6, sugeri que há três níveis de especificação de software que podem ser desenvolvidos: os requisitos de usuário, os requisitos de sistema e a especificação de projeto de software. Os requisitos de usuário são a especificação mais abstrata e a especificação de projeto de software, a mais detalhada. Em geral, as especificações matemáticas formais encontram-se em algum ponto entre os requisitos de sistema e as especificações de projeto de software. Elas não incluem os detalhes da implementação, mas devem apresentar um completo modelo matemático do sistema.

O envolvimento do cliente com o software diminui, enquanto o envolvimento do fornecedor aumenta à medida que a especificação é desenvolvida. Nos estágios iniciais do processo, é essencial que a especificação seja 'orientada ao cliente'. Ela deve ser escrita de modo que seja compreensível para o cliente e deve fazer o menor número de suposições possíveis sobre o projeto do software. Contudo, o estágio final do processo, que é a construção de uma especificação completa, consistente e precisa, é principalmente destinado ao fornecedor do software. Ele serve como base para a implementação do sistema. Essa especificação precisa pode ser uma especificação formal.

A Figura 9.1 mostra os estágios da especificação de software e suas interfaces com o processo de projeto. Os estágios de especificação, mostrados na Figura 9.1, não são independentes nem são necessariamente desenvolvidos na seqüência indicada. A Figura 9.2 mostra que as atividades de especificação e de projeto podem ser realizadas paralelamente. Existe uma relação biunívoca entre cada estágio no processo. As informações são fornecidas da especificação para o processo de projeto e vice-versa.

À medida que a especificação é desenvolvida detalhadamente, aumenta a compreensão do especificador sobre essa especificação. Criar uma especificação formal força uma análise detalhada dos sistemas, o que geralmente revela erros e inconsistências na especificação informal

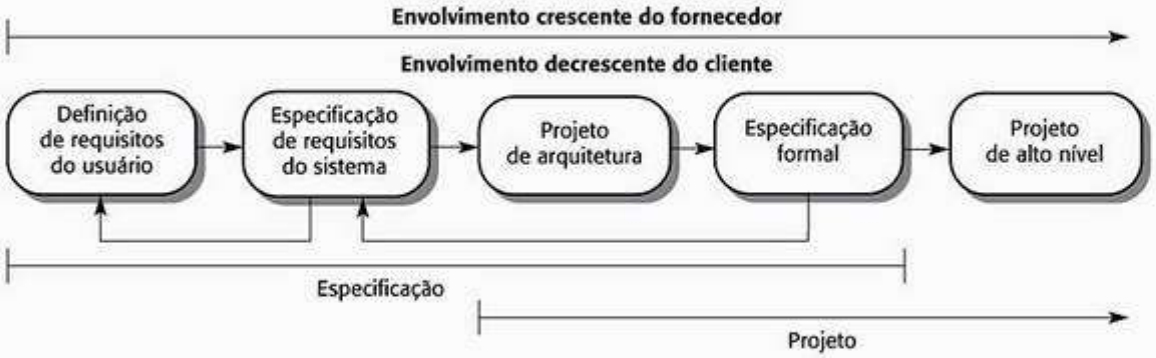


Figura 9.1 ▲

Especificação e projeto.

de requisitos. Essa detecção de erros é o argumento mais forte para o desenvolvimento de uma especificação formal (Hall, 1990). De modo geral, é muito dispendioso corrigir os problemas com requisitos que não são detectados até os estágios mais avançados do processo de software.

Dependendo do processo utilizado, os problemas de especificação descobertos durante a análise formal podem influenciar nas mudanças na especificação de requisitos, se isso já tiver sido estabelecido. Se a especificação de requisitos foi aceita como base de um contrato, os problemas descobertos devem ser levados ao cliente, para resolução, antes de iniciar o projeto de sistema.

Desenvolver e analisar uma especificação formal sobrecarrega os custos do desenvolvimento de software. A Figura 9.3 mostra como os custos do processo de software são afetados pelo uso da especificação formal. Quando um processo convencional é utilizado, os custos de validação são aproximadamente 50 por cento dos custos de desenvolvimento e os custos de implementação e projeto, duas vezes maiores do que os custos de especificação. Com a especificação formal, os custos de especificação e implementação são equivalentes e os custos de validação de sistema são significativamente reduzidos. Como o desenvolvimento da especificação formal descobre problemas com os requisitos, evita-se o retrabalho necessário para corrigir esses problemas, depois que o sistema foi projetado.

Existem duas abordagens para a especificação formal, que têm sido utilizadas para escrever a especificação detalhada de sistemas de software não triviais. São elas:

1. uma abordagem algébrica, em que o sistema é descrito em termos de operações e suas relações e
2. uma abordagem com base em modelos, em que um modelo do sistema é construído, utilizando-se construções matemáticas, como conjuntos e seqüências, e as operações de sistema são definidas em função de como elas modificam o estado do sistema.

Diferentes linguagens, nessas famílias, foram desenvolvidas para especificar sistemas seqüenciais e concorrentes. A Figura 9.4 mostra exemplos das linguagens em cada uma dessas classes. Neste capítulo, são apresentadas ambas as técnicas. Os exemplos mostrados neste texto dão uma idéia de como a especificação formal resulta em uma especificação deta-

Figura 9.2 ▶

Especificação formal no processo de software.

Figura 9.3

Custos de desenvolvimento de software com especificação formal.

Figura 9.4

Linguagens de especificação formal.

	Seqüencial	Concorrente
Algébrica	Larch (Guttag *et al.*, 1985, 1993) OBJ (Futatsugi *et al.*, 1985)	Lotos (Bolognesi e Brinksma, 1987)
Com base em modelos	Z (Spivey, 1992) VDM (Jones, 1980) B (Wordsworth, 1996)	CSP (Hoare, 1985) Petri Nets (Peterson, 1981)

lhada e precisa, mas não são discutidos detalhes da linguagem ou das técnicas de especificação.

9.2 Especificação de interface

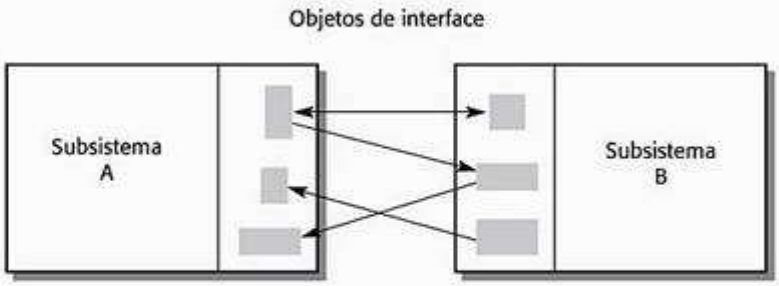
Os grandes sistemas são, em geral, decompostos em subsistemas, que são desenvolvidos independentemente. Os subsistemas utilizam outros subsistemas e, assim, uma parte essencial do processo de especificação é definir interfaces de subsistemas. Uma vez estabelecidas e definidas as interfaces, o subsistema pode ser desenvolvido independentemente.

As interfaces de subsistemas são, com freqüência, definidas como um conjunto de tipos abstratos de dados ou objetos (Figura 9.5). Elas descrevem os dados e as operações que podem ser acessados por meio da interface do subsistema. Uma especificação de interface de subsistema pode, portanto, ser elaborada combinando as especificações dos componentes que constituem a interface do sistema.

Especificações precisas da interface de subsistemas são importantes porque os desenvolvedores de subsistemas devem escrever o código que utilize os serviços de outros subsistemas, antes que esses sejam implementados. A especificação da interface fornece informações para os desenvolvedores de subsistemas, de modo que eles saibam que serviços estarão disponíveis em outros subsistemas e como poderão ser acessados. Especificações de interface de subsistema que sejam claras e não apresentem ambigüidade reduzem as chances de equívoco entre o subsistema que fornece um serviço e o subsistema que o utiliza.

Figura 9.5

Objetos de interface de subsistema.



A abordagem algébrica foi originalmente projetada para a definição de interfaces de tipo abstrato de dados. Em um tipo abstrato de dados, o tipo é definido especificando-se as operações, em vez da representação de tipo. Portanto, ele é muito similar a uma classe de objetos. O método algébrico de especificação formal define o tipo abstrato de dados em termos de suas relações entre as operações de tipo.

Guttag (1977) foi o primeiro a discutir essa abordagem, na especificação de tipos abstratos de dados. Cohen *et al.* (1986) mostram como a técnica pode ser estendida para a especificação completa de sistemas, utilizando-se o exemplo de um sistema de recuperação de documentos. Liskov e Guttag (1986) também tratam da especificação algébrica de tipos abstratos de dados. Larch (Guttag *et al.*, 1993) é provavelmente a linguagem mais conhecida para a especificação algébrica.

A estrutura de uma especificação de objetos é mostrada na Figura 9.6. O corpo da especificação tem quatro componentes:

- Uma introdução, que declara o **sort** (o nome do tipo) da entidade que está sendo especificada. Sort é o nome de um conjunto de objetos e geralmente é implementado como um tipo. A introdução pode também incluir uma declaração **imports**, em que são declarados os nomes de especificações que definem outros sorts. Importar uma especificação torna esses sorts disponíveis para uso.
- Uma parte da descrição, em que as operações são descritas informalmente. Isso torna a especificação formal mais fácil de ser compreendida. A especificação formal complementa essa descrição, fornecendo sintaxe e semântica isentas de ambigüidade para as operações de tipo.
- A parte da assinatura, que define a sintaxe da interface para a classe do objeto ou o tipo abstrato de dados. Os nomes das operações que são definidas, o número e os tipos de parâmetros e os sorts de resultados de operação são descritos na assinatura.
- A parte referente a axiomas, que especifica a semântica das operações, definindo um conjunto de axiomas que caracteriza o comportamento do tipo abstrato de dados. Esses axiomas se relacionam a operações utilizadas para construir entidades da classe definida com operações utilizadas para inspecionar seus valores.

Figura 9.6

Estrutura de uma especificação algébrica.

< NOME DA ESPECIFICAÇÃO >

sort < nome >
imports < LISTA DE NOMES DA ESPECIFICAÇÃO >

Descrição informal do sort e de suas operações.

Assinaturas da operação, que estabelecem os nomes e os tipos de parâmetros para as operações definidas sobre o sort.

Axiomas que definem as operações sobre o sort.

O processo de desenvolvimento de uma especificação formal de interface de subsistemas deve incluir as seguintes atividades:

1. *Estruturação da especificação* Organizar a especificação informal de interface em um conjunto de tipos abstratos de dados ou em classes de objeto. Deve-se, informalmente, definir as operações associadas a cada classe.
2. *Nomeação da especificação* Estabelecer um nome para cada especificação de tipo abstrato, decidir se eles exigem ou não parâmetros genéricos e decidir sobre os nomes para as classes identificadas.
3. *Seleção de operação* Escolher um conjunto de operações para cada especificação, com base na funcionalidade da interface identificada. É preciso incluir operações para criar instâncias da classe, a fim de modificar e inspecionar os valores das instâncias. Talvez seja preciso acrescentar funções àquelas inicialmente identificadas na definição informal da interface.
4. *Especificação informal de operação* Escrever uma especificação informal de cada operação e descrever como as operações afetam a classe definida.
5. *Definição de sintaxe* Definir a sintaxe das operações e os parâmetros para cada uma delas. Essa é a parte de assinatura da especificação formal. Atualizar a especificação informal nesse estágio, se necessário.
6. *Definição de axioma* Definir os nomes de suas operações e descrever que condições são sempre verdadeiras para diferentes combinações de operações.

Para explicar a técnica de especificação algébrica, especificarei uma estrutura simples de dados (uma lista ligada) com um número limitado de operações associadas a ela, conforme ilustra a Figura 9.7.

Para desenvolver o exemplo de especificação da lista, considere que o primeiro estágio, ou seja, a estruturação da especificação, foi realizado e que a necessidade de uma lista foi identificada. O nome da especificação e o nome do sort podem ser os mesmos, embora seja útil distinguir entre eles com o uso de alguma convenção. Serão utilizadas letras maiúsculas para o nome da especificação (LIST) e letras minúsculas com a inicial da palavra em maiúscula para o nome do sort (List). Como as listas são coleções de outros tipos, a especificação

Figura 9.7 ▶

Uma simples especificação de lista.

LIST (Elem)

sort List
imports INTEGER

Define uma lista em que elementos são adicionados ao seu final e removidos de seu início. As operações são Create, que busca uma lista vazia; Cons, que cria uma nova lista com a adição de um membro; Lenght, que avalia o tamanho da lista; Head, que avalia o primeiro elemento da lista, e Tail, que cria uma lista removendo o Head de sua lista de entrada. Undefined representa um valor indefinido do tipo Elem.

Create → List
Cons (List, Elem) → List
Head (List) → Elem
Length (List) → Integer
Tail (List) → List

Head (Create) = Undefined **exception** (empty list)
Head (Cons (L, v)) = **if** L = Create **then** v **else** Head (L)
Length (Create) = 0
Length (Cons (L, v)) = Length (L) + 1
Tail (Create) = Create
Tail (Cons (L, v)) = **if** L = Create **then** Create **else** Cons (Tail (L), v)

tem um parâmetro genérico (Elem). O tipo Elem pode representar integer, string, list e assim por diante.

Em geral, para cada tipo abstrato de dados, as operações necessárias devem conter uma operação para possibilitar instâncias do tipo (Create) e para construir o tipo a partir de seus elementos (Cons). No caso das listas, deve haver uma operação para avaliar o primeiro elemento da lista (Head), uma operação que retorna a lista criada pela remoção do primeiro elemento (Tail) e uma operação para contar o número de elementos da lista (Lenght).

Para definir a sintaxe de cada uma dessas operações, é preciso decidir que parâmetros são exigidos para a operação e os resultados da operação. Em geral, os parâmetros de entrada são o sort que está sendo definido (List) ou o sort genérico. Os resultados das operações podem ser originários desses sorts ou de algum outro sort, como um Integer ou Boolean. No exemplo da lista, a operação Lenght retorna um integer. Uma declaração imports, que afirme que a especificação de integer é utilizada, deve, portanto, ser incluída na especificação.

Os axiomas que definem a semântica de um tipo abstrato de dados são escritos com o uso de operações definidas na parte da assinatura. Eles especificam a semântica, estabelecendo o que é sempre verdadeiro sobre o comportamento de entidades com aquele tipo abstrato.

Operações de um tipo abstrato de dados são classificadas, em geral, em:

1. *Operações de construtor*, que criam ou modificam entidades da classe definida na especificação. De modo geral, elas recebem nomes como Create, Update, Add ou, nesse caso, Cons, que significa construir.
2. *Operações de inspeção*, que avaliam atributos da classe definida na especificação. Tipicamente, elas recebem nomes que correspondem a nomes de atributos ou nomes como Eval, Get, entre outros.

Uma boa regra prática para escrever uma especificação algébrica é estabelecer as operações de construtor e registrar um axioma para cada operação de inspeção sobre cada construtor. Isso sugere que se houver m operações de construtor e n operações de inspeção, haverá m * n axiomas definidos.

Contudo, as operações de construtor associadas com um tipo abstrato podem não ser todas de construtores primitivos. É possível defini-las utilizando-se outras operações de construtores e de inspeção. Se uma operação de construtor puder ser definida com o uso de outros construtores, serão necessárias somente as operações de inspeção, utilizando-se os construtores primitivos.

Na especificação de lista, as operações de construtor são Create, Cons e Tail, que constroem listas. As operações de inspeção são Head e Lenght, que são utilizadas para descobrir atributos de lista. A operação Tail, contudo, não é um construtor primitivo. Portanto, não há nenhuma necessidade de definir axiomas sobre a operação Tail para operações de Head e Lenght, mas Tail tem de ser definida, utilizando-se os construtores primitivos.

Avaliar o cabeçalho de uma lista vazia resulta em um valor indefinido. As especificações de Head e Tail mostram que Head avalia o início da lista e Tail avalia a lista de entrada com seu cabeçalho removido. A especificação de Head declara que o cabeçalho de uma lista criada utilizando Cons é o valor adicionado à lista (se a lista inicial for vazia) ou é o mesmo cabeçalho do parâmetro inicial de lista para Cons. Adicionar um elemento a uma lista não afeta seu cabeçalho, a menos que a lista esteja vazia.

A recursão é comumente utilizada ao escrever especificações algébricas. O valor da operação Tail é a lista que é formada considerando-se a lista de entrada e removendo seu cabeçalho. A definição de Tail mostra como a recursão é utilizada na construção de especificações algébricas. A operação é definida em listas vazias e, então, recursivamente em listas não vazias, com o término da recursão quando o resultado é uma lista vazia.

Às vezes, é mais fácil compreender especificações recursivas desenvolvendo-se um pequeno exemplo. Digamos que temos uma lista [5, 7], em que 5 é o início da lista e 7 é o fim dela. A operação Cons ([5, 7], 9) deve retornar uma lista [5, 7, 9] e uma operação Tail aplicada a isso deve retornar a lista [7, 9]. A seqüência de equações que resulta da substituição de parâ-

metros na especificação anterior com esses valores é:

```
Tail ([5, 7, 9]) =
  Tail (Cons ([5, 7], 9)) =
  Cons (Tail ([5, 7]), 9) =
  Cons (Tail (Cons ([5], 7)), 9) =
  Cons (Cons (Tail ([5]), 7), 9) =
    Cons (Cons (Tail (Cons ([], 5)), 7), 9) =
    Cons (Cons ([Create], 7), 9) =
  Cons ([7], 9) =
    [7, 9]
```

A repetição da escrita sistemática do axioma para Tail ilustra que isso, na verdade, produz o resultado previsto. O axioma para Head pode ser verificado de maneira similar.

Agora, vejamos como essa técnica pode ser utilizada em um sistema crítico. Considere que, em um sistema de controle de tráfego aéreo, um objeto foi projetado para representar um setor de espaço aéreo controlado. Cada setor controlado pode incluir uma série de aeronaves e cada um deles tem um diferente identificador de aeronaves. Por razões de segurança, todas as aeronaves precisam estar distantes, pelo menos, a uma altitude de 300 metros. O sistema alertará o controlador se houver uma tentativa de posicionar uma aeronave de modo que essa restrição seja desrespeitada.

Para simplificar a descrição, defini apenas um número limitado de operações no objeto do setor. Em um sistema real, haverá provavelmente muitas outras operações e mais condições complexas de segurança, relacionadas à separação horizontal da aeronave. As principais operações no objeto são:

1. *Enter* Essa operação adiciona uma aeronave (representada por um identificador) ao espaço aéreo, em uma altitude específica. Não deve haver outra aeronave naquela altitude ou em um raio de 300 metros.
2. *Leave* Essa operação remove a aeronave especificada do setor controlado e é utilizada quando a aeronave se move para um setor adjacente.
3. *Move* Essa operação move uma aeronave de uma altitude para outra. Novamente, é verificada a restrição de segurança de que a separação vertical entre aeronaves tem de ser de, pelo menos, 300 metros.
4. *Lookup* Considerando um identificador de aeronaves, essa operação retorna a altitude atual daquela aeronave no setor.

Ficará mais fácil especificar essas operações, se algumas operações de interface forem definidas:

1. *Create* Essa é uma operação-padrão para um tipo abstrato de dados. Ela provoca uma instância vazia do tipo a ser criado. Nesse caso, representa um setor em que não há nenhuma aeronave.
2. *Put* Essa é uma versão mais simples da operação *Enter*. Ela simplesmente acrescenta uma aeronave ao setor, sem qualquer verificação de restrição associada.
3. *In-space* Dado um sinal de chamada de uma aeronave, essa operação booleana retornará o resultado verdadeiro, se a aeronave estiver no setor controlado, e, caso contrário, retornará o resultado falso.
4. *Occupied* Dada uma altitude, essa operação booleana retornará o resultado verdadeiro, se houver uma aeronave dentro de 300 metros daquela altitude, e, caso contrário, retornará o resultado falso.

A vantagem de definir essas operações mais simples é que, então, é possível utilizá-las como blocos de construção para definir as operações mais complexas no sort de Sector. A especificação algébrica dessa classe é mostrada na Figura 9.8.

Essencialmente, as operações básicas de construtor são Create e Put, que são utilizadas na especificação das outras operações. Occupied e In-space são operações de verificação, que são definidas com a utilização de Create e Put e, então, utilizadas em outras especificações. Aqui, não é possível explicar todas as operações detalhadamente, mas discutirei duas delas (Occupied e Move). Com essas informações, o leitor poderá compreender a especificação das outras operações.

1. A especificação de Occupied declara que, em um espaço aéreo vazio (Create), um nível está sempre vago. Quando aplicado a um espaço aéreo que tenha sido colocado utilizando-se uma operação Put, Occupied verifica se a altitude especificada está dentro dos 300 metros de altitude da aeronave acrescentada ao setor pela operação Put. Em caso positivo, a altitude já está ocupada e, assim, o valor de Occupied é verdadeiro. Caso contrário, a opera-

Figura 9.8 ▶

Especificação de um setor controlado.

```
SECTOR
sort Sector
imports INTEGER, BOOLEAN

Enter – adiciona uma aeronave ao setor, se as condições de segurança forem satisfeitas.
Leave – remove uma aeronave do setor.
Move – move uma aeronave de uma altitude para outro setor que seja seguro.
Lookup – encontra a altitude da aeronave no setor.

Create – cria um setor vazio.
Put – adiciona uma aeronave ao setor sem verificar a restrição.
In-space – verifica se uma aeronave já está no setor.
Occupied – verifica se uma altitude especificada está disponível.

Enter (Sector, Call-sign, Height) → Sector
Leave (Sector, Call-sign) → Sector
Move (Sector, Call-sign, Height) → Sector
Lookup (Sector, Call-sign) → Height

Create → Sector
Put (Sector, Call-sign, Height) → Sector
In-space (Sector, Call-sign) → Boolean
Occupied (Sector, Height) → Boolean

Enter (S, CS, H) =
   if      In-space (S, CS ) then S exception (Aeronave já está no setor)
   elsif   Occupied (S, H) then S exception (Conflito relacionado à altitude)
   else    Put (S, CS, H)

Leave (Create, CS) = Create exception (Aeronave não está no setor)
Leave (Put (S, CS1, H1), CS) =
    if CS = CS1 then S else Put (Leave (S, CS), CS1, H1)

Move (S, CS, H) =
   if      S = Create then Create  exception (Sem aeronave no setor)
   elsif   not In-space (S, CS) then S  exception (Aeronave não está no setor)
   elsif   Occupied (S, H) then S exception (Conflito relacionado à altitude)
   else    Put (Leave (S, CS), CS, H)

-- NO-HEIGHT é uma constante que indica que uma altitude válida não pode ser retornada

Lookup (Create, CS) = NO-HEIGHT  exception (Aeronave não está no setor)
Lookup (Put (S, CS1, H1), CS) =
    if CS = CS1  then H1 else Lookup (S, CS)

Occupied (Create, H) = false
Occupied (Put (S, CS1, H1), H) =
    if      (H1 > H and H1 - H ≤ 300) or (H > H1 and  H - H1 ≤ 300)  then true
   else     Occupied (S, H)

In-space (Create, CS) = false
In-space (Put (S, CS1, H1), CS ) =
    if CS = CS1 then true else In-space (S, CS)
```

ção verifica o setor recursivamente. O leitor pode imaginar essa verificação sendo realizada na última aeronave colocada no setor. Se a altitude não estiver dentro do alcance da altitude daquela aeronave, a operação fará a verificação em relação à aeronave precedente que foi colocada dentro do setor e assim por diante. É possível que, se não houver nenhuma aeronave dentro do alcance da altitude especificada, a verificação seja realizada em relação a um setor vazio e, assim, o resultado retornado será falso.

2. A especificação de Move declara que, se uma operação de Move for aplicada a um espaço aéreo vazio (Create), o espaço aéreo fica imutável e surge uma exceção para indicar que a aeronave especificada não está no espaço aéreo. Quando aplicada a um espaço aéreo que é preenchido utilizando-se Put, a operação primeiramente verifica (utilizando In-space) se a aeronave está no setor. Se não estiver, ela gerará uma exceção. Se estiver no setor, ela então verificará se a altitude definida está disponível (utilizando Occupied), gerando uma exceção, caso haja uma aeronave naquela altitude. Se o espaço estiver disponível, a operação Move será equivalente à aeronave especificada, que deixa o espaço aéreo e é colocada dentro do setor na nova altitude.

9.3 Especificação de comportamento

As técnicas algébricas simples, descritas na seção anterior, são particularmente adequadas para descrever interfaces em que as operações de objeto são independentes do estado do objeto. Ou seja, os resultados da aplicação de uma operação não devem depender dos resultados de operações anteriores. Quando essa condição não prevalece, as técnicas algébricas podem se tornar complicadas. Além disso, penso que as descrições algébricas de sistema de comportamento são, muitas vezes, artificiais e de difícil compreensão.

Uma abordagem alternativa para a especificação formal, que tem sido mais amplamente utilizada em projetos industriais, é a especificação baseada em modelos. A especificação baseada em modelos é uma abordagem da especificação formal, em que a especificação de sistemas é expressa como um modelo de estado do sistema. As operações de sistemas são especificadas, definindo-se como elas afetam o estado do sistema. Desse modo, o comportamento do sistema pode ser definido.

Notações bem desenvolvidas para o desenvolvimento de especificações com base em modelos são: VDM (Jones, 1980; Jones, 1986), B (Wordsworth, 1996) e Z (Hayes, 1987; Spivey, 1992). Optei pelo uso de Z. Em Z, os sistemas são modelados utilizando-se conjuntos e relações entre conjuntos. Contudo, Z aumentou esses conceitos matemáticos com construções que aceitam particularmente a especificação de software.

Em uma introdução à especificação baseada em modelos, posso apresentar apenas uma visão geral de como uma especificação pode ser desenvolvida. Uma descrição completa da notação Z seria mais longa do que este capítulo. Em vez disso, apresento alguns exemplos para ilustrar a técnica e apresentar a notação assim que ela é requerida. Uma descrição completa da notação Z é fornecida em livros, como o de Diller (1994), Woodcock (Woodcock e Davies, 1996) e Jacky (1997).

As especificações formais podem ser cansativas e de difícil leitura, especialmente quando são apresentadas como grandes fórmulas matemáticas. É compreensível, portanto, que isso tenha inibido muitos engenheiros de software de investigar seu potencial no desenvolvimento de sistemas. Os projetistas da notação Z devotaram atenção especial a esse problema. As especificações são apresentadas como texto informal, complementado com descrições formais. As descrições formais são incluídas como pequenas partes (chamadas de esquemas), que são de fácil leitura, e diferenciadas do texto associado com o emprego do destaque gráfico. Os esquemas são utilizados para introduzir variáveis de estado e definir restrições e operações no

estado. Entre as operações de esquemas estão a composição, a renomeação e a ocultação de esquemas. Essas operações permitem que os esquemas sejam manipulados. Elas são um poderoso mecanismo para a especificação de sistemas.

Para ser mais eficaz, uma especificação formal deve ser complementada por uma descrição informal de apoio. A apresentação do esquema Z foi projetada para se destacar do texto a seu redor (veja a Figura 9.9).

A assinatura do esquema define as entidades que constituem o estado do sistema e o predicado do esquema estabelece condições que devem sempre ser verdadeiras para essas entidades. Quando um esquema define uma operação, o predicado pode estabelecer pré-condições ou pós-condições. A diferença entre essas pré e pós-condições define a ação especificada no esquema de operação.

Para ilustrar o uso de Z na especificação de um sistema importante, utilizo um exemplo simplificado de um sistema de bomba de insulina, utilizada por diabéticos. Os diabéticos não podem metabolizar a glicose naturalmente e necessitam de injeções de insulina geneticamente produzida, um hormônio essencial para metabolizar a glicose. Esse sistema monitora o nível de glicose no sangue de diabéticos e automaticamente injeta insulina quando necessário. Utilizo esse exemplo também na Parte 4, em que abordo o desenvolvimento de sistemas críticos.

A Figura 9.10 ilustra a estrutura dessa bomba de insulina. A glicose do sangue é monitorada em intervalos regulares e, se seu nível estiver aumentando, a insulina será injetada para reduzir o nível de glicose.

1. *Caixa da agulha* Conectada à bomba. Utilizada para fornecer insulina ao corpo do diabético.
2. *Sensor* Mede o nível de glicose no sangue do usuário. A entrada do sensor é representada por reading? na especificação formal.
3. *Bomba* Bombeia a insulina de um reservatório para a caixa da agulha. O valor referente ao número de doses de insulina a ser administrado é representado por dose na especificação formal.

Figura 9.9 ▶

Estrutura de um esquema Z.

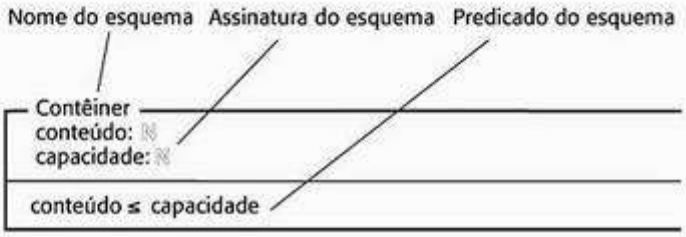
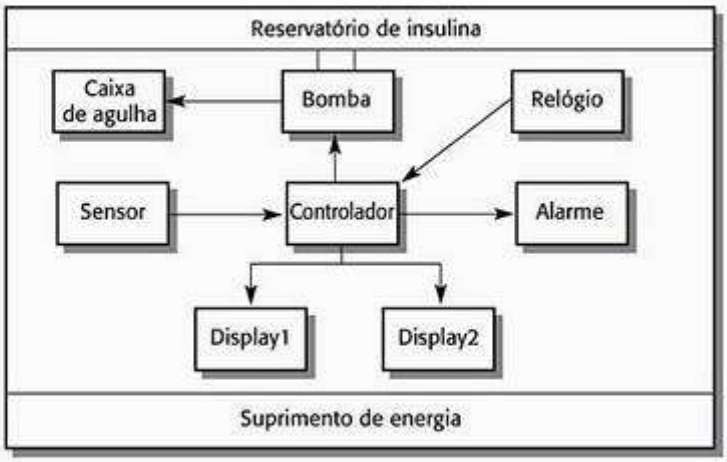


Figura 9.10 ▶

Um diagrama em blocos de uma bomba de insulina.



4. *Controlador* Controla todo o sistema. Tem uma chave para ligar/desligar, um botão para anular e um botão para ajustar a quantidade a ser fornecida. Para simplificar, não modelei essa capacidade de anulação na descrição formal.
5. *Alarme* Dispara um alarme, caso ocorra algum problema. O valor fornecido para o alarme é representado por alarm! na especificação seguinte.
6. *Displays* Existem dois displays. Um deles mostra a última leitura de açúcar no sangue medida e, o outro, mensagens de status. Esses displays são representados por display1! e display2! na especificação formal.
7. *Relógio* Fornece ao controlador a hora atual.

Mesmo para um pequeno sistema como a bomba de insulina, a especificação formal é bem longa. Embora a operação básica do sistema seja simples, há muitas condições de alarme possíveis que devem ser consideradas. Portanto, incluo aqui apenas alguns dos esquemas principais da especificação e explico o que eles significam.

O esquema básico que modela o estado da bomba de insulina é mostrado na Figura 9.11. Podemos ver aqui como as duas partes básicas do esquema são utilizadas. Na parte superior, nomes e tipos são declarados e, na parte inferior, são estabelecidas algumas condições que são sempre verdadeiras.

O estado é modelado utilizando-se uma série de variáveis de estado. Por convenção, os nomes em Z seguidos de uma ? (interrogação) representam as entradas e os nomes seguidos de uma ! (exclamação) representam as saídas. Contudo, essas são convenções de denominação simples e as variáveis representadas são tratadas exatamente da mesma maneira que qualquer outra variável de estado. Os nomes declarados no esquema são:

1. reading? Este é um número natural (isto é, um número não negativo) que representa a leitura a partir do sensor de glicose no sangue. É um valor de entrada.
2. dose, cumulative_dose São também números naturais que representam a dose de insulina a ser ministrada e a dosagem cumulativa ministrada durante certo período.
3. r0, r1, r2 Representam as três últimas leituras e são utilizados para calcular a taxa de alteração do nível de glicose no sangue.
4. capacity Um número natural que representa a capacidade do reservatório de insulina na bomba.
5. alarm! Essa saída representa o alarme da máquina que assinala alguma condição excepcional.
6. pump! Este é um número natural que representa os sinais de controle transmitidos ao conjunto físico da bomba. É um valor de saída.
7. display 1!, display 2! Esses valores de saída do tipo *string* representam dois displays de texto na bomba de insulina. Um display (display1) é utilizado para exibir mensagens de texto e o outro (display2), para mostrar a dose de insulina aplicada.

Figura 9.11 ▶

Esquema da bomba de insulina.

```
┌─ Insulin_pump ──────────────────────────────────────────
│ reading? : ℕ
│ dose, cumulative_dose: ℕ
│ r0, r1, r2: ℕ      // utilizado para gravar as últimas três leituras
│ capacity: ℕ
│ alarm!: {off, on}
│ pump!: ℕ
│ display1!, display2!: STRING
├─────────────────────────────────────────────────────────
│ dose ≤ capacity ∧ dose ≤ 5 ∧ cumulative_dose ≤ 50
│ capacity ≥ 40 ⇒ display1! = " "
│ capacity ≤ 39 ∧ capacity ≥ 10 ⇒ display1! = "Insulin low"
│ capacity ≤ 9 ⇒ alarm! = on ∧ display1! = "Insulin very low"
│ r2 = reading?
└─────────────────────────────────────────────────────────
```

O predicado de esquema define invariáveis que são sempre verdadeiras com um implícito 'e' entre cada linha do predicado. As condições sobre o estado do sistema, que são sempre verdadeiras para a bomba de insulina, são:

1. A dose deve ser menor ou igual à capacidade. Ou seja, é impossível aplicar mais insulina do que a que está no reservatório.

2. Nenhuma dose única pode ser maior do que 5 unidades de insulina e a dose total aplicada em determinado período não pode exceder a 50 unidades. Essas são condições de segurança que serão discutidas no Capítulo 17.

3. O display1! mostra uma mensagem que indica o status do reservatório de insulina. É possível pensar na frase Z <expressão lógica 1> → <expressão lógica 2> como sendo o mesmo que if <expressão lógica 1> then <expressão lógica 2>. Se o reservatório contém 40 unidades ou mais, então o display está em branco; se contém entre 10 e 40 unidades, então um aviso é exibido, e se contém menos de 10 unidades, então um alarme é acionado e um aviso de urgência é exibido.

A bomba de insulina operará verificando a glicose no sangue a cada dez minutos e (de maneira simplificada) a insulina será aplicada, se a taxa de alteração da glicose no sangue estiver aumentando. A quantidade de insulina a ser aplicada é calculada como mostra a Figura 9.12, que apresenta o esquema DOSAGE. Se a taxa de alteração for estática, uma quantidade fixa de insulina será aplicada. Pode-se ver, nesse esquema, que há várias situações diferentes que afetam a dose aplicada. Os detalhes dessas situações não são realmente relevantes em nosso caso.

Não modelei o comportamento temporal do sistema (isto é, o fato de que o sensor de glicose é verificado a cada dez minutos) utilizando Z. Embora essa opção certamente seja possível, ela é um tanto inoportuna e, na minha opinião, uma descrição informal, na verdade, comunica a especificação mais concisamente do que uma especificação formal.

A Figura 9.12 ilustra uma característica de Z comumente utilizada: o esquema delta. Se um nome de esquema for incluído na parte das declarações, isso será equivalente a incluir na declaração todos os nomes declarados naquele esquema e as condições estarão na parte do predicado. Elas são associadas aos predicados que são declarados no esquema DOSAGE.

Figura 9.12 ▶

Esquema DOSAGE.

```
┌─ DOSAGE ──────────────────────────────────────
│ ΔInsulin_Pump
├───────────────────────────────────────────────
│ (
│ dose = 0 ∧
│   (
│         ((r1 ≥ r0) ∧ (r2 = r1)) ∨
│         ((r1 > r0) ∧ (r2 ≤ r1)) ∨
│         ((r1 < r0) ∧ ((r1 − r2) > (r0 − r1)))
│   ) ∨
│  dose = 4 ∧
│   (
│         ((r1 ≤ r0) ∧ (r2 = r1))∨
│         ((r1 < r0) ∧ ((r1 − r2) ≤ (r0 − r1)))
│   ) ∨
│ dose =(r2 − r1) * 4 ∧
│   (
│         ((r1 ≤ r0) ∧ (r2 > r1)) ∨
│         ((r1 > r0) ∧ ((r2 − r1) ≥ (r1 − r0)))
│   )
│ )
│ capacity' = capacity − dose
│ cumulative_dose' = cumulative_dose + dose
│ r0' = r1 ∧ r1' = r2
└───────────────────────────────────────────────
```

Se o nome do esquema for precedido de um Δ (símbolo de delta), será introduzido um novo conjunto de valores, cujos nomes são os mesmos daqueles declarados no esquema, mas os quais são indicados por um símbolo (′). Esses valores representam os valores das variáveis de estado associadas após uma operação. Portanto, se uma operação modifica um valor val (digamos), o valor depois da operação é referido como val′. O esquema delta em DOSAGE, portanto, introduz os nomes capacity′, cumulative-dose′ e assim por diante.

Os esquemas que modelam as saídas do dispositivo da bomba de insulina são mostrados na Figura 9.13. Eles modelam os displays da máquina e do alarme inserido. Novamente, um esquema delta é utilizado para introduzir os nomes das variáveis indicadas, embora, nesse caso, somente as variáveis de saída sejam referenciadas. O esquema DISPLAY declara que display2! mostra a dose calculada (Nat_to_string é uma função de conversão) e que display1! mostra uma mensagem de aviso ou 'OK'. O esquema ALARM estabelece as condições quando o alarme é ativado. Ele estará ligado se a leitura de glicose do sangue for muito baixa (menos do que 3) ou muito alta (mais do que 30).

Os predicados em todos os esquemas Z têm de ser consistentes, ou seja, não deve haver uma declaração feita em um esquema que contradiga o predicado em outro esquema. Quando existem inconsistências, freqüentemente elas indicam problemas de especificação e várias técnicas matemáticas podem ser aplicadas a uma especificação Z para a análise de inconsistências. Não vou entrar nesse assunto agora. Contudo, examinando os quatro esquemas Z, podemos perceber uma inconsistência que introduzi deliberadamente, mas que poderia ter sido introduzida por acidente, em uma especificação real.

No esquema total Insulin_pump, é declarado que display1! deve mostrar o estado do reservatório de insulina. Contudo, o esquema DISPLAY declara que esse display deve mostrar um tipo diferente de mensagem de aviso ou deve indicar que o nível de glicose do sangue está dentro de limites aceitáveis. Há um conflito aqui. O que, portanto, deve ser mostrado nesse display? Essa questão precisa ser resolvida discutindo-se o problema com especialistas médicos e usuários do sistema em potencial.

Destacar problemas que devam ser resolvidos é uma das principais vantagens de utilizar uma especificação formal. Com uma especificação informal, é muito mais fácil ignorar esses conflitos, que devem ser resolvidos em um estágio posterior do processo de desenvolvimento.

Figura 9.13 ▶
Esquemas de saídas.

```
┌─ DISPLAY ──────────────────────────────────
│ ΔInsulin_Pump
├────────────────────────────────────────────
│ display2!′ = Nat_to_string (dose) ∧
│ (reading? < 3 ⇒ display1!′ = "Sugar low" ∨
│ reading? > 30 ⇒ display1!′ = "Sugar high" ∨
│ reading? ≥ 3 and reading? ≤ 30 ⇒ display1!′ = "OK")
└────────────────────────────────────────────

┌─ ALARM ────────────────────────────────────
│ ΔInsulin_Pump
├────────────────────────────────────────────
│ ( reading? < 3 ∨ reading? > 30 ) ⇒ alarm!′ = on ∨
│ ( reading? ≥ 3 ∧ reading? ≤ 30 ) ⇒ alarm!′ = off
└────────────────────────────────────────────
```

PONTOS-CHAVE

- Os métodos de especificação formal de sistema complementam as técnicas de especificação informal. Eles podem ser utilizados para aprimorar uma especificação detalhada de requisitos de sistema, de maneira informal. Portanto, eles ajudam a diminuir a distância entre requisitos e projeto.
- As especificações formais são precisas e isentas de ambigüidade. Elas eliminam dúvidas em uma especificação e evitam alguns problemas de má interpretação de linguagem. Contudo, os não-especialistas podem considerar as especificações formais difíceis de serem compreendidas.
- O valor principal da utilização de métodos formais no processo de software é que essa opção força uma análise dos requisitos de sistema em um estágio inicial. Corrigir erros nesse estágio é mais barato do que modificar um sistema já entregue.
- As técnicas de especificação formal são mais aplicáveis ao desenvolvimento de sistemas críticos, em que a segurança, a confiabilidade e a proteção são particularmente importantes. Elas também podem ser utilizadas para especificar padrões.
- As técnicas algébricas de especificação formal são particularmente adequadas para especificar interfaces, quando a interface é definida como um conjunto de classes de objeto ou tipos abstratos de dados. Essas técnicas ocultam o estado do sistema e especificam o sistema em termos de relações entre as operações de interfaces.
- As técnicas baseadas em modelos apresentam o sistema utilizando construções matemáticas, como conjuntos e funções, e podem expor o estado do sistema, o que simplifica alguns tipos de especificação de comportamento.

LEITURAS SUGERIDAS

IEEE transactions on software engineering, jan. 1998. Essa edição do jornal apresenta uma seção especial sobre os usos práticos de métodos formais na engenharia de software. Traz também artigos sobre Z e LARCH.

"Formal methods: promises and problems". Esse artigo é uma discussão realista das vantagens em potencial da utilização de métodos formais e das dificuldades de integrar o uso de métodos formais ao desenvolvimento prático de software. (Luqi e J. Goguen, *IEEE Software*, 14(1), jan. 1997.)

"Strategies for incorporating formal specifications in software development". Nesse artigo, os autores classificam diferentes abordagens da especificação formal e sugerem que a falta de apoio metodológico e de ferramentas é um fator importante para a não-utilização de métodos formais. (M.D. Fraser, K. Kumar e V. Vaishnavi, *Comm. ACM*, 37(10), out. 1994.)

"A specifier's introduction to formal methods". Embora esse artigo tenha sido publicado há mais de dez anos, as técnicas fundamentais de especificação não mudaram muito. Esse é um bom artigo introdutório, que descreve os princípios da especificação formal e introduz uma série de diferentes abordagens. (J.M. Wing, *IEEE Computer,* 23(9), set. 1990.)

EXERCÍCIOS

9.1 Sugira por que o projeto de arquitetura de um sistema deve preceder o desenvolvimento de uma especificação formal.

9.2 Você foi incumbido da tarefa de 'vender' técnicas formais de especificação para uma organização de desenvolvimento de software. Faça um esboço de como você explicaria as vantagens da especificação formal para engenheiros de software práticos e céticos.

9.3 Explique por que é particularmente importante definir interfaces de subsistema de maneira precisa e por que a especificação algébrica é especialmente apropriada para a especificação de interfaces de subsistema.

9.4 Um tipo abstrato de dados, representando uma pilha, tem as seguintes operações associadas:

New: gera uma nova pilha.
Push: adiciona um elemento ao topo da pilha.
Top: avalia o elemento no topo da pilha.
Retract: remove o elemento de cima da pilha e retorna a pilha modificada.
Empty: verdadeiro, se não houver nenhum elemento na pilha.

Defina esse tipo abstrato de dados, utilizando uma especificação algébrica.

9.5 No exemplo de um setor de espaço aéreo controlado, a condição de segurança é que a aeronave não pode estar dentro de um espaço de 300 m de altitude no mesmo setor. Modifique a especificação mostrada na Figura 9.8, a fim de permitir que a aeronave ocupe a mesma altitude no setor, desde que estejam separadas em pelo menos 8 km, no sentido horizontal. Você pode ignorar aeronaves em setores adjacentes. *Dica: é* preciso modificar as operações de construtor de modo que incluam a posição da aeronave e sua altitude. Você também precisa definir uma operação que, dadas as duas posições, retorne a separação entre elas.

9.6 As máquinas de caixa de banco contam com o uso de informações do cartão de usuário, que fornece o identificador do banco, o número da conta e o identificador pessoal do usuário. Elas também extraem informações sobre contas de um banco de dados central e atualizam esse banco de dados ao término de uma transação. Utilizando seu conhecimento da operação de ATMs, escreva esquemas Z definindo o estado do sistema, a validação de cartão (quando o identificador do usuário é verificado) e o saque.

9.7 Modifique o esquema Insulin_pump, mostrado na Figura 9.11, e o esquema DOSAGE, mostrado na Figura 9.12, de modo que o usuário de sistema possa anular a dose calculada e especificar a dose a ser ministrada. As condições de segurança especificadas no esquema Insulin_pump devem ainda permanecer. O fato de o sistema estar operando no modo cancelamento deve ser indicado em um display.

9.8 Você é um engenheiro de sistemas e lhe pedem para sugerir a melhor maneira de desenvolver um software destinado a um marcapasso de coração no qual a segurança é fundamental. Você sugere especificar formalmente o sistema, mas sua sugestão não é aceita pelo seu gerente. Você acha que as razões dele não têm sentido e são preconceituosas. É ético desenvolver o sistema utilizando métodos que você acha que são inadequados?

PARTE TRÊS

Projeto

PÁGINA EM BRANCO

10 Projeto de arquitetura

Objetivos

O objetivo deste capítulo é introduzir os conceitos de arquitetura de software e projeto de arquitetura. Depois de ler este capítulo, você poderá:

- compreender por que o projeto de arquitetura de software é importante;
- compreender que diferentes modelos podem ser requeridos para documentar uma arquitetura de sistema;
- conhecer diferentes tipos de arquitetura de software, que abrangem estrutura de sistema, controle e decomposição modular, e
- compreender como os modelos de arquitetura de domínio específico podem ser utilizados como base para arquiteturas tipo produto de linha e para comparar implementações de arquitetura.

Sumário

10.1 Estruturação do sistema
10.2 Modelos de controle
10.3 Decomposição em módulos
10.4 Arquiteturas de domínio específico

Os grandes sistemas são sempre decompostos em subsistemas, que fornecem algum conjunto de serviços relacionados. O processo inicial de projeto, que consiste em identificar esses subsistemas e estabelecer um *framework* para o controle e a comunicação de subsistemas, é chamado de *projeto de arquitetura* e a saída desse processo de projeto é uma descrição da *arquitetura do software.*

A estrutura geral do processo de projeto foi discutida na Seção 3.4. Na Figura 3.9, apresentei um modelo do processo de projeto, em que o projeto de arquitetura é o primeiro estágio e representa um vínculo importante entre o projeto e os processos de engenharia de requisitos. O ideal seria uma especificação que não inclua quaisquer informações sobre o projeto. Na prática, isso é irreal, exceto para sistemas muito pequenos. A decomposição da arquitetura é necessária para estruturar e organizar a especificação. Um bom exemplo disso foi apresentado na Figura 2.4, que mostra a arquitetura de um sistema de controle de tráfego aéreo. O modelo de arquitetura é, com freqüência, o ponto de partida para a especificação das várias partes do sistema.

O processo de projeto de arquitetura se preocupa em estabelecer um *framework* estrutural básico para um sistema. Ele envolve a identificação dos componentes principais do sistema e das comunicações entre esses componentes. Bass *et al.* (1998) discutem três vantagens de projetar e documentar explicitamente uma arquitetura de software:

1. *Comunicação com os stakeholders* A arquitetura é uma apresentação de alto nível do sistema, que pode ser utilizada como um ponto de discussão para uma gama de diferentes *stakeholders.*

2. *Análise de sistema* Tornar explícita a arquitetura de sistemas em um estágio inicial do desenvolvimento do sistema significa que alguma análise pode ser realizada. As decisões de projeto de arquitetura têm um profundo efeito sobre se o sistema pode ou não cumprir requisitos importantes, como desempenho, confiabilidade e facilidade de manutenção.

3. *Reutilização em larga escala* A arquitetura de sistemas é uma descrição compacta e administrável de como um sistema é organizado e de como os componentes operam entre si. A arquitetura pode ser transferida por meio de sistemas com requisitos similares e, dessa maneira, pode fornecer apoio ao reuso de software em grande escala. É possível desenvolver arquiteturas tipo produto de linha, em que a mesma arquitetura é utilizada por meio de uma gama de sistemas relacionados. Isso será discutido na Seção 10.4 e no Capítulo 14.

Diferentes projetistas enfocam o processo de projeto de arquitetura de modos diferentes. O processo utilizado depende da aplicação do conhecimento, da habilidade e da intuição do arquiteto do sistema. Contudo, as seguintes atividades são comuns a todos os processos de projeto de arquitetura:

1. *Estruturação de sistema* O sistema é estruturado em uma série de subsistemas principais, em que cada subsistema é uma unidade de software independente. As comunicações entre subsistemas são identificadas. Esse assunto é abordado na Seção 10.1.

2. *Modelagem de controle* É estabelecido um modelo geral dos relacionamentos de controle entre as partes do sistema. Esse aspecto será abordado na Seção 10.2.

3. *Decomposição modular* Cada subsistema identificado é decomposto em módulos. O arquiteto deve decidir sobre o tipo de módulo e suas interconexões. Esse tema será abordado na Seção 10.3.

Essas atividades são, geralmente, intercaladas, em vez de realizadas em seqüência. Durante qualquer um desses processos, é possível que se desenvolva o projeto mais detalhadamente, para averiguar se as decisões de projeto de arquitetura permitem que o sistema cumpra seus requisitos.

Não existe uma distinção nítida entre subsistemas e módulos, mas acredito ser útil pensar neles como se segue:

1. *Um subsistema* é um sistema cuja operação não depende dos serviços fornecidos por outros subsistemas. Os subsistemas são compostos de módulos e têm interfaces definidas, as quais são utilizadas para a comunicação entre subsistemas.
2. *Um módulo* é geralmente um componente de sistema que fornece um ou mais serviços para outros módulos. Ele utiliza serviços fornecidos por outros módulos e normalmente não é considerado um sistema independente. Os módulos são compostos de uma série de outros componentes de sistema mais simples.

A saída do processo de projeto de arquitetura é um documento de projeto de arquitetura. Esse documento consiste em uma série em representações gráficas dos modelos de sistema, juntamente com texto descritivo a ele associado. Ele deve descrever como o sistema é estruturado em subsistemas e como cada subsistema é estruturado em módulos. Os diferentes modelos gráficos do sistema apresentam diferentes perspectivas sobre a arquitetura. Entre os modelos de arquitetura que podem ser desenvolvidos estão:

1. Um modelo estrutural estático que mostra os subsistemas ou componentes que devem ser desenvolvidos como unidades separadas.
2. Um modelo de processo dinâmico que mostra como o sistema é organizado em processos em *run-time* (tempo de execução). Esse processo pode ser diferente do modelo estático.
3. Um modelo de interface que define os serviços oferecidos por cada subsistema em sua interface pública.
4. Modelos de relacionamento que mostram relacionamentos como o fluxo de dados entre os subsistemas.

Um grupo de pesquisadores propuseram o uso de linguagens de descrição de arquitetura (*architectural description languages* – ADLs) para descrever arquiteturas de sistemas. Bass *et al.* (1998) descrevem as principais características dessas linguagens. Os elementos básicos das ADLs são componentes e conectores, e elas incluem regras e diretrizes para arquiteturas bem formadas. Contudo, como todas as linguagens especializadas, elas sofrem a desvantagem de que somente podem ser compreendidas por peritos em linguagem – são inacessíveis aos especialistas de domínio e de aplicação. Isso faz com que elas sejam de difícil análise, a partir de uma perspectiva prática. Portanto, é provável que só sejam utilizadas em um pequeno número de aplicações. No entanto, os modelos e as notações informais, como a UML, serão, de preferência, mais utilizados para a descrição de arquitetura.

O projeto de arquitetura pode ter como base um modelo ou um estilo de arquitetura específico (Garlan e Shaw, 1993). É importante uma conscientização desses modelos, de suas aplicações, de seus pontos fortes e fracos. Neste capítulo, são descritos diferentes modelos estruturais, modelos de controle e modelos de decomposição.

Contudo, as arquiteturas da maioria dos sistemas de grande porte não obedecem a um único modelo. Diferentes partes do sistema podem ser projetadas com o uso de diferentes modelos de arquitetura. Além disso, em alguns casos a arquitetura de sistema pode ser composta, ou seja, criada pela combinação de diferentes modelos de arquitetura. Os projetistas precisam encontrar o modelo mais apropriado e, então, modificá-lo de acordo com os requisitos do problema. Um exemplo desse caso é mostrado na Seção 10.4, na discussão sobre a arquitetura de um compilador, em que um modelo de repositório é combinado com um modelo de fluxo de dados.

A arquitetura do sistema afeta o desempenho, a robustez e a facilidade de distribuição e de manutenção de um sistema. A estrutura e o estilo específico escolhidos para uma aplicação podem, portanto, depender dos requisitos não funcionais do sistema:

1. *Desempenho* Se o desempenho for um requisito importante, isso sugere que a arquitetura deve ser projetada para restringir as operações mais importantes dentro de um pequeno

número de subsistemas com a menor comunicação possível entre esses subsistemas. Isso pode significar a utilização de componentes com granularidade relativamente maior, em vez de componentes com menor granularidade, a fim de reduzir a comunicação entre componentes.

2. *Proteção* Se a proteção for um requisito fundamental, isso vai sugerir que uma estrutura em camadas para a arquitetura deve ser utilizada com os itens mais importantes protegidos nas camadas mais internas e com um alto nível de validação de proteção aplicado a essas camadas.

3. *Segurança* Se a segurança for um requisito de grande importância, isso vai sugerir que a arquitetura deve ser projetada de modo que as operações relacionadas com a segurança fiquem todas localizadas em um único subsistema ou em um pequeno número de subsistemas. Essa opção reduz os custos e os problemas de validação de segurança e torna possível fornecer os sistemas de proteção relacionados.

4. *Disponibilidade* Se a disponibilidade for um requisito fundamental, isso vai sugerir que a arquitetura do sistema deve ser projetada para incluir componentes redundantes, para que seja possível substituir e atualizar componentes, sem a interrupção do sistema. As arquiteturas de sistema tolerantes a falhas, para sistemas com grande disponibilidade, serão abordadas no Capítulo 18.

5. *Facilidade de manutenção* Se a facilidade de manutenção for um requisito fundamental, isso vai sugerir que a arquitetura deve ser projetada utilizando-se componentes encapsulados de menor granularidade, que possam ser rapidamente modificados. Os produtores de dados devem ser separados dos consumidores e as estruturas de dados compartilhadas devem ser evitadas.

Existe, com certeza, um conflito em potencial entre algumas dessas arquiteturas. Por exemplo, o desempenho é melhorado pelo uso de componentes de maior granularidade e a facilidade de manutenção, pelo uso de componentes de menor granularidade. Se ambas as características forem importantes requisitos de sistema, será preciso encontrar alguma solução intermediária. Como foi discutido anteriormente, pode-se, às vezes, conseguir isso pelo uso de diferentes estilos de arquitetura para diferentes partes do sistema.

10.1 Estruturação do sistema

A primeira fase da atividade de projeto de arquitetura é geralmente voltada para a decomposição do sistema em um conjunto de subsistemas que interagem. Em seu nível de maior abstração, um projeto de arquitetura pode ser representado como um diagrama de blocos, em que cada bloco do diagrama representa um subsistema. Blocos dentro de blocos indicam que o subsistema foi, por sua vez, decomposto em subsistemas. As setas significam que dados e/ou controles são passados de subsistema para subsistema, na direção das setas. Um diagrama de blocos de arquitetura apresenta uma visão geral da estrutura do sistema. De modo geral, ele é compreensível a vários engenheiros que estejam envolvidos no processo de desenvolvimento de sistema.

A Figura 10.1 é um modelo estrutural de arquitetura para um sistema robotizado de embalagem. Esse sistema de robô pode embalar diferentes tipos de objetos. Ele utiliza um subsistema de visão para apanhar objetos em uma transportadora, identifica o tipo de objeto e seleciona o tipo certo de embalagem. Em seguida, ele retira objetos da transportadora de entrega para serem embalados. Os objetos embalados são colocados em outra transportadora. Outros exemplos de projetos de arquitetura deste nível são mostrados nas figuras 2.2 e 2.4.

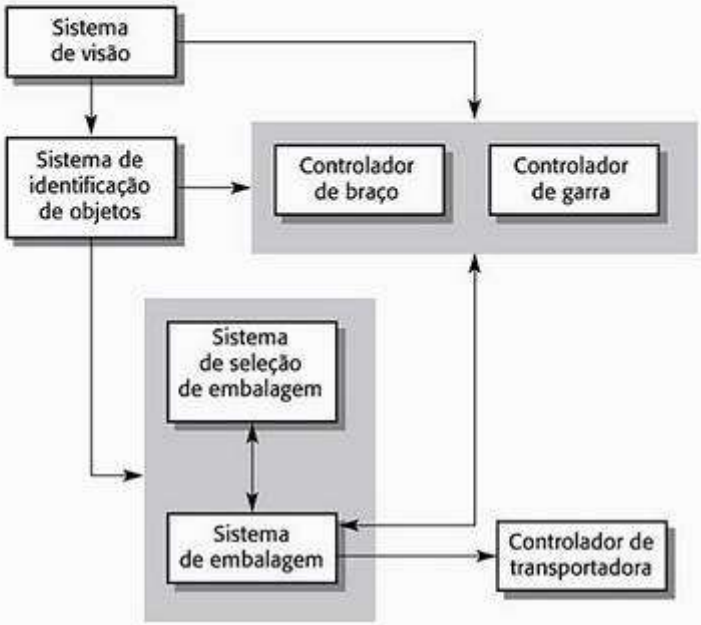


Figura 10.1 ▶ Diagrama de blocos de um sistema de controle robotizado de embalagem.

Bass *et al.* (1998) argumentam que os diagramas simples de blocos e linhas não são representações úteis de arquitetura, uma vez que não mostram a natureza das relações entre componentes de sistema nem suas propriedades externas visíveis. A partir da perspectiva de um projetista de software, isso é absolutamente correto. Contudo, esse tipo de modelo é eficaz para a comunicação com os *stakeholders* do sistema e para planejar o projeto. Como ele não é muito detalhado, os *stakeholders* podem relacioná-lo ao projeto e ter uma visão abstrata do sistema. Ele identifica os principais subsistemas que são desenvolvidos independentemente, de modo que os gerentes possam começar a designar pessoas para planejar o desenvolvimento desses sistemas. Os diagramas de blocos e linhas certamente não devem ser as únicas representações de arquitetura a serem utilizadas; contudo, acho que elas fazem parte de uma série de modelos úteis de arquitetura.

Modelos mais específicos da estrutura podem ser desenvolvidos, os quais mostram como os subsistemas compartilham dados, como estão distribuídos e como atuam como interface entre si. São discutidos, nesta seção, três desses modelos-padrão: modelo de repositório, modelo cliente-servidor e modelo de máquina abstrata.

10.1.1 O modelo de repositório

Os subsistemas que constituem um sistema precisam trocar informações, para que possam trabalhar em conjunto de modo eficaz. Existem duas maneiras fundamentais para fazê-lo:

1. Todos os dados compartilhados são mantidos em um banco de dados central, que pode ser acessado por todos os subsistemas. Um modelo de sistema com base em um banco de dados compartilhado é, algumas vezes, chamado de *modelo de repositório*.
2. Cada subsistema mantém seu próprio banco de dados. Os dados são intercambiados com outros subsistemas, transmitindo mensagens para eles.

A maioria dos sistemas que utilizam grandes quantidades de dados é organizada em termos de um banco de dados compartilhado ou de um repositório. Esse modelo é, portanto, adequado a aplicações em que os dados são gerados por um subsistema e utilizados por outro. São exemplos desse tipo de sistema os sistemas de comando e controle, os sistemas de infor-

mações gerenciais, os sistemas de CAD (*computer aided design* – projeto auxiliado por computador) e os conjuntos de ferramentas CASE.

A Figura 10.2 é um exemplo de uma arquitetura de conjunto de ferramentas CASE, com base em um repositório compartilhado. O primeiro repositório compartilhado para ferramentas CASE foi provavelmente desenvolvido, no início da década de 70, por uma empresa inglesa chamada ICL, a fim de apoiar seu desenvolvimento de sistemas operacionais (McGuffin *et al.*, 1979). Esse modelo tornou-se amplamente conhecido quando Buxton (1980) apresentou propostas para o ambiente Stoneman, com o propósito de apoiar o desenvolvimento de sistemas escritos em Ada. Desde então, muitos conjuntos de ferramentas CASE foram desenvolvidos com base em um repositório compartilhado.

As vantagens e as desvantagens de um repositório compartilhado são as seguintes:

1. É uma maneira eficiente de compartilhar grandes quantidades de dados. Não há necessidade de transmitir os dados explicitamente de um subsistema para outro.
2. Contudo, os subsistemas devem concordar com o modelo de dados de repositório. Inevitavelmente, este é um meio de conciliar as necessidades específicas de cada ferramenta. O desempenho pode ser adversamente afetado por essa conciliação. Pode ser difícil ou impossível integrar novos subsistemas, se seus modelos de dados não se adequarem ao esquema estabelecido.
3. Os subsistemas que produzem dados não precisam saber como esses dados são utilizados pelos outros subsistemas.
4. Contudo, a evolução pode ser difícil, uma vez que um grande volume de informações é gerado de acordo com um modelo de dados estabelecido. Traduzir isso para um novo modelo será certamente dispendioso e poderá ser difícil ou mesmo impossível.
5. As atividades como backup, segurança, controle de acesso e recuperação a partir de erros são centralizadas e de responsabilidade do gerente de repositório. As ferramentas podem focalizar suas principais funções, em vez de se ocuparem com essas questões.
6. Contudo, diferentes subsistemas podem ter diferentes requisitos para políticas de proteção, recuperação e backup. O modelo de repositório impõe a mesma política a todos os subsistemas.
7. O modelo de compartilhamento é visível por meio do esquema de repositório. É simples e direto integrar novas ferramentas, considerando que elas são compatíveis com o modelo de dados estabelecido.
8. Contudo, pode ser difícil distribuir o repositório em uma série de máquinas. Embora seja possível distribuir um repositório logicamente centralizado, pode haver problemas de redundância e inconsistência de dados.

Figura 10.2 ▶

Arquitetura de um conjunto de ferramentas CASE integrado.

No modelo mostrado anteriormente, o repositório é passivo e o controle é de responsabilidade do subsistema que utiliza o repositório. Uma abordagem alternativa foi derivada para os sistemas de AI, que utilizam um modelo de 'quadro negro' que ativa subsistemas, quando determinados dados se tornam disponíveis. Isso é apropriado quando a forma dos dados do repositório é menos estruturada. As decisões sobre quais ferramentas devem ser ativadas só podem ser tomadas quando os dados tiverem sido analisados. Esse modelo é discutido por Nii (1986).

10.1.2 O modelo cliente–servidor

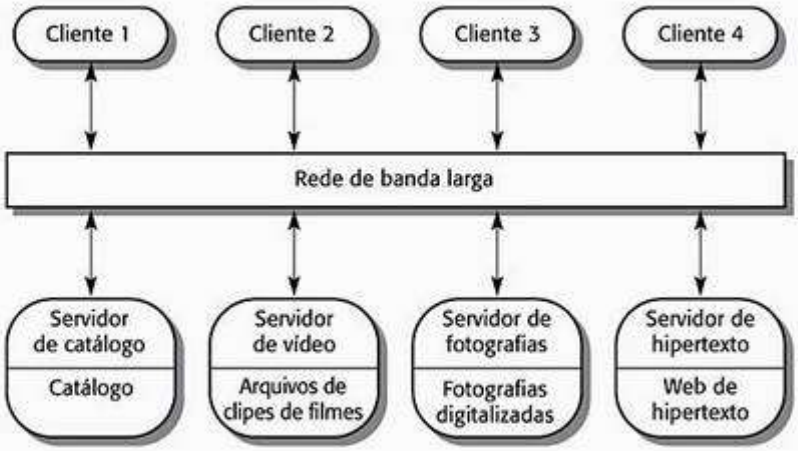
O modelo de arquitetura de cliente-servidor é um modelo de sistema distribuído, que mostra como os dados e o processamento são distribuídos em uma série de processadores. Os componentes principais desse modelo são:

1. Um conjunto de servidores em *stand-alone*, que oferece serviços a outros subsistemas. Podemos citar como exemplo os servidores de impressoras, que oferecem serviços de impressão; os servidores de arquivo, que oferecem serviços de gerenciamento de arquivos, e os servidores de compilação, que oferecem serviços de compilação de linguagem de programação.
2. Um conjunto de clientes que solicita os serviços oferecidos pelos servidores. Estes são, normalmente, subsistemas em si. É possível que haja várias instâncias de um programa cliente sendo executadas simultaneamente.
3. Uma rede que permite aos clientes acessar esses serviços. Em princípio, isso não é realmente necessário, uma vez que tanto clientes quanto servidores poderiam ser executados em uma única máquina. Na prática, contudo, esse modelo não seria utilizado nessa situação.

É possível que os clientes tenham de saber os nomes dos servidores disponíveis e dos serviços que eles fornecem. Contudo, os servidores não precisam saber a identidade de clientes ou quantos clientes existem. Os clientes têm acesso aos serviços fornecidos por um servidor por meio de chamadas remotas.

Um exemplo de um sistema construído com base em um modelo cliente-servidor é mostrado na Figura 10.3. Este é um sistema de hipertexto para vários usuários, destinado a fornecer uma biblioteca de filmes e fotografias. Nesse sistema, há vários servidores que gerenciam e exibem os diferentes tipos de mídia. Os *frames* do filme precisam ser transmitidos com rapidez e em sincronia, mas com resolução relativamente baixa. Eles precisam ser compactados em um repositório. Contudo, as imagens precisam ser transmitidas com alta resolução. O catá-

Figura 10.3 ▶
Arquitetura de um sistema de biblioteca de filmes e fotografias.



logo deve ser capaz de lidar com uma variedade de consultas e fornecer links com os sistemas de informações de hipertexto. O programa cliente é simplesmente uma interface com o usuário, integrada a esses serviços.

A abordagem cliente-servidor pode ser utilizada para implementar um sistema com base em repositórios, em que o repositório é fornecido como um servidor de sistemas. Os subsistemas que têm acesso ao repositório são clientes. No entanto, normalmente, cada subsistema gerencia seus próprios dados. Servidores e clientes trocam dados para o processamento. Isso pode resultar em problemas de desempenho, quando grandes quantidades de dados são trocadas. No entanto, à medida que redes mais rápidas são desenvolvidas, esse problema está se tornando menos importante.

A vantagem mais importante do modelo cliente-servidor é que ele é uma arquitetura distribuída, permitindo o uso efetivo de sistemas de rede com muitos processadores distribuídos. É fácil incluir um novo servidor e integrá-lo com o restante do sistema ou fazer a atualização dos servidores de modo transparente, sem afetar outras partes do sistema. Serão discutidas as arquiteturas distribuídas, bem como as arquiteturas cliente-servidor e as arquiteturas de objetos distribuídos, com mais detalhes no Capítulo 11.

Contudo, mudanças nos clientes e servidores existentes podem ser necessárias, a fim de se obter os benefícios plenos da integração de um novo servidor. Não há nenhum modelo de dados compartilhados e os subsistemas normalmente organizam seus dados de diferentes modos. Isso significa que modelos específicos de dados podem ser estabelecidos para cada servidor, o que permite otimizar seu desempenho. A falta de um modelo de referência compartilhada para dados pode significar que é difícil prever problemas na integração de dados a partir de um novo servidor. Cada servidor deve assumir a responsabilidade por atividades de gerenciamento de dados, como backup e recuperação.

10.1.3 O modelo de máquina abstrata

O modelo de máquina abstrata de uma arquitetura (às vezes, chamado de modelo em camadas) modela a interface de subsistemas e organiza um sistema em uma série de camadas, cada uma das quais fornece um conjunto de serviços. Cada camada define uma máquina abstrata, cuja linguagem de máquina (os serviços fornecidos pela camada) é utilizada para implementar o próximo nível de máquina abstrata. Por exemplo, uma maneira comum de implementar uma linguagem é definir uma 'máquina de linguagem' ideal e compilar a linguagem no código para essa máquina. Uma etapa adicional de tradução converte esse código de máquina abstrata em código de máquina real.

Um exemplo bem conhecido dessa abordagem é o modelo de referência OSI de protocolos de redes (Zimmermann, 1980), que será discutido na Seção 10.4.2. Outro exemplo influente foi proposto por Buxton (1980), que sugeriu um modelo de três camadas para um APSE (*Ada programming support environment* – ambiente de apoio à programação em Ada). A Figura 10.4 tem algo em comum com isso e mostra como um sistema de gerenciamento de versões pode ser integrado com o uso dessa abordagem de máquina abstrata.

Figura 10.4 ▶

Modelo de máquina abstrata de um sistema de gerenciamento de versões.

O sistema de gerenciamento de versões se dedica a gerenciar versões de objetos e fornece recursos gerais de gerenciamento de configuração, como será discutido no Capítulo 29. Para apoiar esses recursos de gerenciamento de configuração, ele utiliza um sistema de gerenciamento de objetos que fornece serviços de armazenamento e gerenciamento de informações para objetos. Esse sistema emprega um sistema de banco de dados para fornecer armazenamento e serviços de dados básicos, como o gerenciamento de transações, o *rollback* e a recuperação, além de controle de acesso. O gerenciamento de banco de dados utiliza os recursos básicos de sistema operacional e de repositório de arquivos em sua implementação.

A abordagem em camadas apóia o desenvolvimento gradativo de sistemas. À medida que uma camada é desenvolvida, alguns serviços fornecidos por aquela camada podem se tornar disponíveis para os usuários. Essa arquitetura também pode ser modificada e apresenta portabilidade. Se sua interface for preservada, uma camada poderá ser substituída por outra camada. Quando as interfaces em camadas se modificam, somente a camada adjacente é afetada. Como os sistemas em camadas localizam dependências da máquina nas camadas mais internas, eles podem ser implementados em outros computadores de modo bastante acessível. Somente as camadas internas, dependentes de máquina, precisam ser novamente implementadas.

Uma desvantagem da abordagem em camadas é que pode ser difícil estruturar sistemas dessa maneira. Recursos básicos, como o gerenciamento de arquivos, que são exigidos por todas as máquinas abstratas, podem ser fornecidos por camadas internas. Os serviços requeridos pelo usuário podem, portanto, requerer acesso a uma máquina abstrata, que está várias camadas abaixo da camada mais externa. Isso subverte o modelo, quando uma camada externa não é mais simplesmente dependente de sua camada imediatamente anterior.

O desempenho também pode ser comprometido devido aos vários níveis de interpretação de comando que são, às vezes, requeridos. Se houver muitas camadas, algum custo indireto sempre será associado com o gerenciamento de camadas. Para evitar esses problemas, as aplicações podem ter de se comunicar diretamente com as camadas internas, em vez de utilizar os recursos fornecidos na máquina abstrata.

10.2 Modelos de controle

Os modelos para estruturar um sistema se ocupam de como um sistema é decomposto em subsistemas. Para trabalhar como um sistema, os subsistemas devem ser controlados de modo que seus serviços sejam fornecidos no local certo e no tempo certo. Os modelos estruturais não incluem (e não devem fazê-lo) informações de controle. Em vez disso, o arquiteto deve organizar os subsistemas de acordo com algum modelo de controle que complemente o modelo estrutural utilizado. Os modelos de controle em nível de arquitetura dizem respeito ao controle de fluxo entre subsistemas.

Duas abordagens gerais de controle podem ser identificadas:

1. *Controle centralizado* Um subsistema tem a responsabilidade geral pelo controle e inicia e interrompe os outros subsistemas. Ele pode também devolver o controle para outro subsistema, mas aguardará que essa responsabilidade pelo controle seja devolvida a ele.

2. *Controle baseado em eventos* Em vez de ter as informações de controle embutidas em um subsistema, cada subsistema pode responder a eventos gerados externamente. Esses eventos podem provir de outros subsistemas ou do ambiente do sistema.

Os modelos de controle complementam os modelos estruturais. Todos os modelos estruturais mostrados anteriormente podem ser realizados com o uso do controle centralizado ou do controle baseado em eventos.

10.2.1 Controle centralizado

Em um modelo de controle centralizado, um subsistema é escolhido como o controlador de sistema e tem a responsabilidade de gerenciar a execução dos outros subsistemas. Os modelos de controle centralizado se dividem em duas classes, que dependem de os subsistemas controlados operarem seqüencialmente ou em paralelo.

1. *Modelo retorno de chamadas* Esse é o familiar modelo de sub-rotina top-down, em que o controle começa na parte superior de uma hierarquia de sub-rotina e, por meio de chamadas de sub-rotina, passa para níveis mais inferiores na árvore. O modelo de sub-rotina aplica-se apenas a sistemas seqüenciais.
2. *Modelo gerenciador* Pode ser aplicado a sistemas concorrentes. Um componente de sistema é designado como um gerenciador do sistema e controla o início, a interrupção e a coordenação de outros processos de sistema. Um processo é um subsistema ou um módulo que pode ser executado em paralelo com outros processos. Uma variação desse modelo também pode ser aplicada em sistemas seqüenciais, em que uma rotina de gerenciamento chama determinados subsistemas, dependendo dos valores de algumas variáveis de estado. Isso é geralmente implementado como uma declaração Case.

O modelo retorno de chamadas está ilustrado na Figura 10.5. O programa principal pode chamar as rotinas 1, 2 e 3; a Rotina 1 pode chamar as rotinas 1.1 ou 1.2; a Rotina 3 pode chamar as rotinas 3.1 ou 3.2, e assim por diante. Esse é um modelo de dinâmica do programa, e não um modelo estrutural. Não existe nenhuma necessidade de a Rotina 1.1, por exemplo, ser parte da Rotina 1.

Esse modelo familiar está embutido em linguagens de programação, como Ada, Pascal e C. O controle passa de uma rotina de alto nível na hierarquia para uma rotina de nível inferior e, então, retorna ao ponto em que a rotina foi chamada. A sub-rotina atualmente em execução tem responsabilidade pelo controle e tanto pode chamar outras rotinas como devolver o controle para seu antecessor. A opção de retornar a algum outro ponto no programa consiste em um estilo de programação inadequado.

Esse modelo retorno de chamadas pode ser utilizado em nível de módulo para controlar funções ou objetos. As sub-rotinas em uma linguagem de programação, que são chamadas por outras sub-rotinas, são naturalmente funcionais. Contudo, em muitos sistemas orientados a objetos, as operações em objetos (métodos) são implementadas como procedimentos ou funções. Por exemplo, quando um objeto Java solicita um serviço de outro objeto, ele faz isso chamando um método associado.

A natureza rígida e restrita desse modelo é, ao mesmo tempo, um ponto forte e fraco. É um ponto forte porque é relativamente simples analisar fluxos de controle e concluir como o sistema responderá a entradas específicas. É um ponto fraco porque, como será discutido no Capítulo 18, é difícil lidar com as exceções feitas à operação normal.

Figura 10.5 ▶
Modelo de retorno de chamadas.

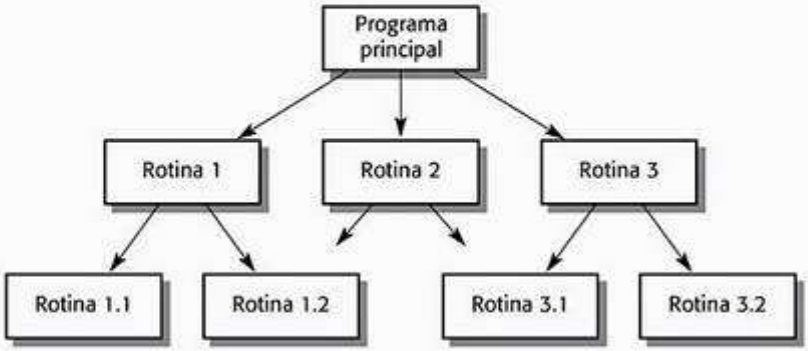
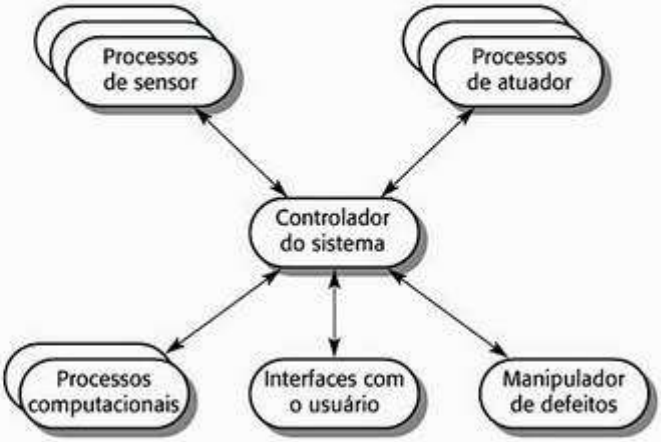


Figura 10.6 ▶

Modelo de controle centralizado para sistema de tempo real.



A Figura 10.6 é uma ilustração de um modelo de gerenciamento centralizado de controle para um sistema concorrente. Esse modelo é, freqüentemente, utilizado em sistemas em tempo real simples, que não têm restrições de tempo muito rígidas. O controlador central gerencia a execução de um conjunto de processos associados com sensores e atuadores. O sistema de monitoramento de edifícios, que será discutido no Capítulo 13, segue esse modelo de controle.

O processo controlador de sistemas decide quando os processos devem ser iniciados ou interrompidos, dependendo das variáveis de estado do sistema. Ele verifica se outros processos produziram informações que devem ser processadas ou se é preciso transmitir informações a eles para processamento. De modo geral, o controlador está em um loop contínuo, verificando sensores e outros processos para eventos e mudanças de estado. Por essa razão, esse modelo é, algumas vezes, chamado de loop de eventos.

10.2.2 Sistemas orientados a eventos

Nos modelos de controle centralizado, as decisões de controle são normalmente determinadas pelos valores de algumas variáveis de estado do sistema. Por outro lado, os modelos de controle orientados a eventos são baseados em eventos gerados externamente. O termo *evento*, nesse contexto, não significa somente um sinal binário; ele pode ser um sinal capaz de assumir uma gama de valores. A distinção entre um evento e uma entrada simples é que a ocorrência do evento está fora do controle do processo que manipula esse evento. Um subsistema pode precisar acessar as informações de estado, a fim de manipular esses eventos, mas essas informações de estado em geral não determinam o fluxo de controle.

Existem tipos diferentes de sistemas orientados a eventos, que podem ser desenvolvidos. Eles podem ser planilhas de cálculo, em que a mudança do valor de uma célula faz com que outras células sejam modificadas; sistemas de produção com base em regras, como é utilizado na AI, em que uma condição que se torna verdadeira ocasiona o início de uma ação, e objetos ativos, em que a mudança de um valor de atributo de um objeto inicia algumas ações. Garlan *et al.* (1992) discutem esses diferentes tipos de sistemas.

Nesta seção, são discutidos dois modelos de controle orientados a eventos:

1. *Modelos de transmissão* Nesses modelos, um evento é, em princípio, transmitido para todos os subsistemas. Qualquer subsistema que puder manipular esse evento responderá a ele.

2. *Modelos orientados a interrupções* Eles são exclusivamente utilizados em sistemas de tempo real, em que interrupções externas são detectadas por um manipulador de interrupções. Eles são, então, passados para algum outro componente para processamento.

Os modelos de transmissão são eficazes na integração de subsistemas distribuídos em diferentes computadores em rede. Modelos dirigidos por interrupções são utilizados em sistemas de tempo real com rigorosos requisitos de tempo.

Em um modelo de transmissão (Figura 10.7), os subsistemas registram um interesse em eventos específicos. Quando esses eventos ocorrem, o controle é transferido para o subsistema, que pode manipular o evento. A distinção entre esse modelo e o modelo centralizado, mostrado na Figura 10.6, é que a política de controle não está incluída no evento e no manipulador de mensagens. Os subsistemas decidem quais eventos são requeridos e o manipulador de eventos e mensagens assegura que esses eventos sejam enviados para eles.

Todos os eventos podem ser transmitidos para todos os subsistemas, mas isso exige uma grande quantidade de processamento adicional. Mais freqüentemente, o manipulador de eventos e mensagens mantém um registro de subsistemas e eventos que lhe interessa. Os subsistemas geram eventos que indicam, talvez, que alguns dados estão disponíveis para processamento. O manipulador de eventos detecta os eventos, consulta o registrador de eventos e passa o evento para aqueles subsistemas que declararam interesse.

De modo geral, o manipulador de eventos também aceita a comunicação ponto a ponto. Um subsistema pode explicitamente enviar uma mensagem para outro sistema. Existe uma série de variações desse modelo, como o ambiente de Campo (Reiss, 1990) e o Softbench, da Hewlett-Packard (Fromme e Walker, 1993). Ambos foram utilizados para controlar as interações de ferramentas em ambientes de engenharia de software. Os requisitores de objetos (*object request brokers*), discutidos no Capítulo 11, também são compatíveis com esse modelo de controle para a comunicação de objetos distribuídos.

A vantagem dessa abordagem de transmissão é que a evolução é relativamente simples. Um novo subsistema para manipular determinadas classes de eventos pode ser integrado ao registrar seus eventos com o manipulador de eventos. Qualquer subsistema pode ativar qualquer outro subsistema, sem saber seu nome ou sua localização. Os subsistemas podem ser implementados em máquinas distribuídas e essa distribuição é transparente para outros subsistemas.

A desvantagem desse modelo é que os subsistemas não sabem se ou quando os eventos serão manipulados. Quando um subsistema gera um evento, ele não sabe que outros subsistemas registraram um interesse nesse evento. É bem possível que diferentes subsistemas se registrem para os mesmos eventos. Isso pode gerar conflitos, quando os resultados da manipulação de eventos se tornam disponíveis.

Os sistemas de tempo real que requerem eventos gerados externamente para serem manipulados muito rapidamente devem ser orientados a eventos. Por exemplo, se um sistema de tempo real for utilizado para controlar o sistema de segurança em um carro, ele deverá detectar uma possível colisão e, talvez, inflar um *air bag* antes que a cabeça do motorista bata no volante. Para proporcionar essa resposta rápida a eventos, o controle dirigido por interrupções tem de ser utilizado.

Um modelo de controle dirigido por interrupções é ilustrado na Figura 10.8. Existe um número conhecido de tipos de interrupção, com um manipulador definido para cada um deles. Cada tipo de interrupção é associado com a locação de memória em que o seu endereço de manipulador está armazenado. Quando ocorre uma interrupção de um tipo específico, um comutador de hardware provoca a transferência de controle imediatamente para seu manipulador. Esse manipulador de interrupção pode, então, iniciar ou interromper outros processos, em resposta ao evento assinalado pela interrupção.

Figura 10.7 ▶

Modelo de controle com base em transmissão (broadcasting) seletiva.

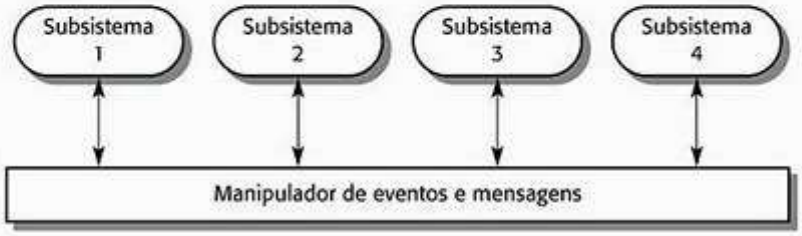
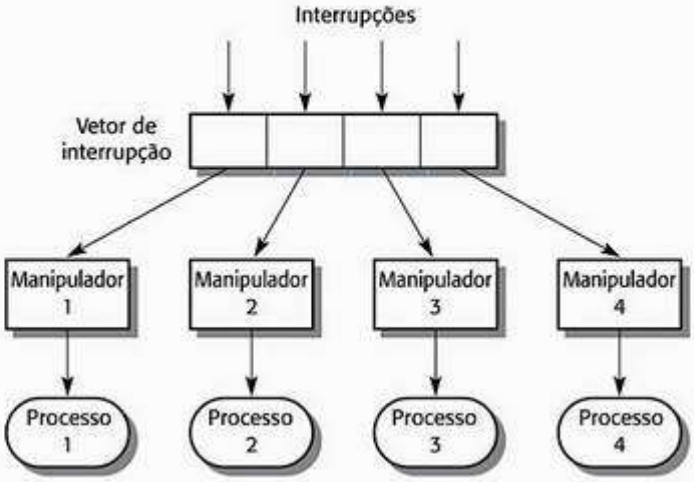


Figura 10.8 ▶
Modelo de controle dirigido por interrupções.



Esse modelo deve ser utilizado somente em sistemas especiais de tempo real, em que a resposta imediata a algum evento seja necessária. Ele pode ser combinado com o modelo de gerenciamento centralizado. O gerenciador central cuida da operação normal do sistema, com o controle baseado em interrupções para emergências.

A vantagem dessa abordagem do controle é que ela permite respostas muito rápidas para os eventos a serem implementados. A desvantagem é que sua programação é complexa e sua validação, difícil. Pode ser impossível repetir padrões de tempo de interrupção durante o processo de teste do sistema e pode ser difícil modificar sistemas desenvolvidos utilizando-se esse modelo, se o número de interrupções for limitado pelo hardware. Uma vez atingido esse limite, nenhum outro tipo de evento pode ser manipulado. Essa limitação pode, algumas vezes, ser contornada, mapeando-se vários tipos de eventos dentro de uma única interrupção e, então, deixando o manipulador resolver qual evento ocorreu. Contudo, essa poderá ser uma abordagem impraticável, se uma resposta muito rápida à interrupção for exigida.

10.3 Decomposição em módulos

Depois que uma arquitetura estrutural foi projetada, o próximo estágio do processo de projeto de arquitetura é a decomposição dos subsistemas em módulos. Não há uma distinção rígida entre a decomposição de sistema e a decomposição em módulos. Os modelos discutidos na Seção 10.1 podem ser aplicados nesse nível. Contudo, os componentes em módulos são geralmente menores do que os subsistemas, e isso permite que sejam utilizados modelos alternativos de decomposição.

Considero dois modelos que podem ser utilizados na decomposição de um subsistema em módulos:

1. *Modelo orientado a objetos* O sistema é decomposto em um conjunto de objetos que se comunicam.
2. *Modelo de fluxo de dados* O sistema é decomposto em módulos funcionais que aceitam a entrada de dados e os transformam, de alguma maneira, em dados de saída. Isso é também chamado de abordagem de *pipeline*.

No modelo orientado a objetos, os módulos são objetos com estado privativo e operações definidas nesse estado. No modelo de fluxo de dados, os módulos são transformações fun-

cionais. Em ambos os casos, os módulos podem ser implementados como componentes seqüenciais ou como processos.

Os projetistas devem evitar, se possível, tomar decisões prematuras sobre a simultaneidade. A vantagem de evitar um projeto de sistema em paralelo é que programas seqüenciais são mais fáceis de projetar, implementar, verificar e testar do que os sistemas paralelos. As dependências de tempo entre processos são difíceis de formalizar, controlar e verificar. É melhor decompor sistemas em módulos e, então, decidir durante a implementação se eles devem ser executados em seqüência ou em paralelo.

10.3.1 Modelos de objetos

Um modelo orientado a objetos de uma arquitetura de sistemas estrutura o sistema em um conjunto de objetos inadequadamente acoplados, com interfaces bem definidas. Os objetos chamam os serviços oferecidos por outros objetos. Já foi discutido sobre os modelos de objetos no Capítulo 7, e o assunto será discutido, com mais detalhes, no Capítulo 12.

A Figura 10.9 é um exemplo de modelo de arquitetura orientada a objetos de um sistema de processamento de faturas. Esse sistema pode emitir faturas para clientes, receber pagamentos e emitir recibos desses pagamentos e avisos de faturas que não foram pagas. É utilizada a notação em UML, apresentada no Capítulo 7, em que as classes de objetos têm nomes e um conjunto de atributos associados. As operações, caso ocorram, são definidas na parte inferior do retângulo que representa o objeto. Setas tracejadas indicam que um objeto utiliza os atributos ou serviços fornecidos por outro objeto.

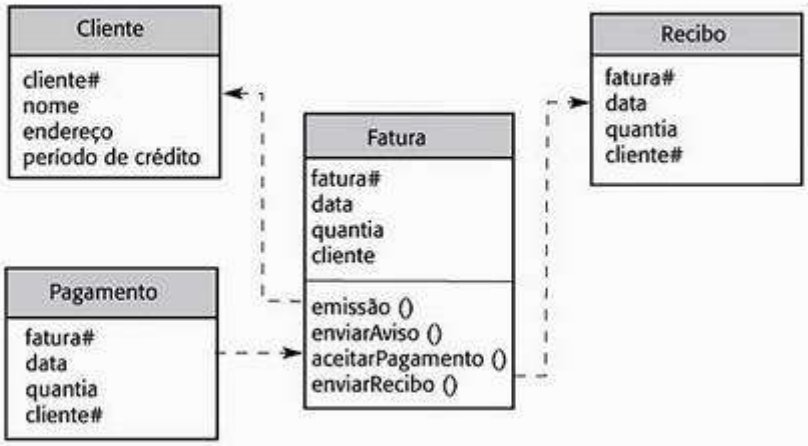
Uma decomposição orientada a objetos diz respeito a classes de objetos, seus atributos e suas operações. Quando ela é implementada, os objetos são criados a partir dessas classes e algum modelo de controle é utilizado para coordenar as operações de objetos – nesse exemplo específico, é a classe 'Fatura', que tem várias operações associadas, que implementam a funcionalidade do sistema. Essa classe utiliza outras classes que representam clientes, pagamentos e recibos.

As vantagens da abordagem orientada a objetos são bem conhecidas. Uma vez que os objetos são inadequadamente acoplados, a implementação de objetos pode ser modificada, sem afetar outros objetos. Os objetos são, freqüentemente, representações de entidades do mundo real e, assim, a estrutura do sistema é realmente compreensível. Uma vez que essas entidades do mundo real são utilizadas em diferentes sistemas, os objetos podem ser reutilizados. Foram desenvolvidas linguagens de programação orientadas a objetos, que fornecem a implementação direta de componentes de arquitetura.

Contudo, a abordagem orientada a objetos tem também desvantagens. Para utilizar serviços, os objetos devem explicitamente fazer referência ao nome e à interface de outros objetos. Se

Figura 10.9 ▶

Modelo de objeto de um sistema de processamento de faturas.



uma modificação na interface for requerida para satisfazer as mudanças propostas no sistema, deverá ser avaliado o efeito dessa mudança em todos os usuários do objeto modificado. Embora os objetos possam mapear claramente até as entidades do mundo real, em pequena escala, às vezes, as entidades mais complexas são difíceis de ser representadas como objetos.

10.3.2 Modelos de fluxo de dados

Em um modelo de fluxo de dados, as transformações funcionais processam suas entradas e produzem saídas. Os dados fluem de uma para outra e são transformados à medida que se movimentam ao longo da seqüência. Cada etapa do processamento é implementada como uma transformação. Os dados de entrada fluem por essas transformações até serem convertidos em saída. As transformações podem ser executadas seqüencialmente ou em paralelo. Os dados podem ser processados por cada transformação item por item ou em um único lote.

Quando as transformações são apresentadas como processos separados, esse modelo, às vezes, é chamado de modelo de duto (pipe) e filtro, segundo a terminologia utilizada no sistema Unix. O sistema Unix proporciona dutos que atuam como condutores de dados e um conjunto de comandos que são transformações funcionais. Os sistemas que se adequam a esse modelo podem ser implementados pela combinação de comandos Unix com o uso de dutos e dos recursos de controle do shell do Unix. O termo 'filtro' é utilizado porque uma transformação 'filtra' os dados que pode processar a partir de seu fluxo de dados de entrada.

Variantes desse modelo de fluxo de dados têm sido empregadas desde que os computadores foram inicialmente utilizados para o processamento automático de dados. Quando as transformações são seqüenciais, com dados processados em lotes, esse modelo de arquitetura é um modelo seqüencial em lotes. Essa é uma arquitetura comum para algumas classes de sistemas de processamento de dados, como os sistemas de cobrança, que geram grandes números de relatórios de saída, que são extraídos de cálculos simples em um grande número de registros de entrada.

Um exemplo desse tipo de arquitetura de sistemas é mostrado na Figura 10.10. Uma organização emitiu faturas para clientes. Uma vez por semana, os pagamentos que foram recebidos são conciliados com as faturas. Para as faturas que foram pagas, um recibo é emitido; para as que não foram pagas dentro do prazo de pagamento permitido, um aviso é emitido.

Este é um modelo de apenas uma parte do sistema de processamento de faturas; transformações alternativas seriam utilizadas para a emissão de faturas. Observe a diferença entre esse modelo e seu equivalente orientado a objetos, discutido na seção anterior. O modelo de objetos é mais abstrato, uma vez que não traz informações sobre a seqüência de operações.

As vantagens dessa arquitetura são:

1. Ela suporta o reuso de transformações.
2. Ela é intuitiva, no sentido de que muitas pessoas pensam em seu trabalho em termos de processamento de entrada e saída.

Figura 10.10 ▼

Modelo de fluxo de dados de um sistema de processamento de faturas.

3. A evolução do sistema pela adição de novas transformações é geralmente simples e direta.
4. Ela é simples de ser implementada, como um sistema simultâneo ou seqüencial.

A principal desvantagem do modelo surge da necessidade de um formato comum para a transferência de dados, que possa ser reconhecido por todas as transformações. Cada transformação deve concordar com sua comunicação de transformações no formato dos dados que serão processados ou, então, um formato-padrão para todos os dados trocados tem de ser imposto. A última abordagem é a única viável, quando as transformações são independentes e reutilizáveis. Em Unix, o formato-padrão é simplesmente uma seqüência de caracteres. Cada transformação precisa analisar sua entrada e ajustar sua saída à maneira estabelecida. Isso aumenta o tempo (overhead) do sistema e pode significar que é impossível integrar transformações que utilizam formatos de dados incompatíveis.

É difícil escrever sistemas interativos utilizando o modelo de fluxo de dados, devido à necessidade de uma seqüência de dados a ser processada. Embora as entradas e as saídas de texto simples possam ser modeladas dessa maneira, as interfaces gráficas com o usuário têm formatos de entrada/saída mais complexos e um controle que tem como base eventos como cliques de mouse ou seleção de menu. É difícil traduzir isso de maneira compatível com o modelo de fluxo de dados.

10.4 Arquiteturas de domínio específico

Os modelos de arquitetura mostrados anteriormente são modelos gerais. Eles podem ser aplicados a muitas classes de aplicação diferentes. Assim como os modelos gerais, os modelos de arquitetura, que são específicos de um determinado domínio de aplicação, também podem ser utilizados. Embora as instâncias desses sistemas sejam distintas nos detalhes, a estrutura comum de arquitetura pode ser reutilizada para o desenvolvimento de novos sistemas. Esses modelos de arquitetura são chamados de *arquiteturas de domínio específico*.

Existem dois tipos de modelo de arquitetura de domínio específico:

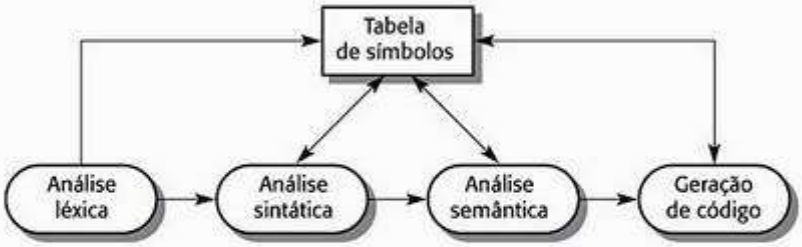
1. *Modelos genéricos*, que são abstrações a partir de uma série de sistemas reais. Eles englobam as características principais desses sistemas. Por exemplo, em sistemas em tempo real, é possível que haja modelos de arquitetura genéricos de diferentes tipos de sistemas, como sistemas de coleta de dados, sistemas de monitoramento, entre outros.
2. *Modelos de referência*, que são mais abstratos e descrevem uma classe maior de sistemas. Eles são um meio de informar os projetistas sobre a estrutura geral daquela classe de sistema. Por exemplo, Rockwell e Gera (1993) propuseram um modelo de referência para fábricas de software.

Obviamente, não há uma distinção rígida entre os diferentes tipos de modelos. Os modelos genéricos podem também, algumas vezes, servir como modelos de referência. Aqui, é feita uma distinção entre eles, porque os modelos genéricos podem ser reutilizados diretamente em um projeto. Os modelos de referência são, normalmente, utilizados para comunicar conceitos de domínio e comparar possíveis arquiteturas.

Essa condição reflete a ordem de derivação desses modelos. Os modelos genéricos, que são representações abstratas do sistema, são geralmente derivados bottom-up, a partir de sistemas existentes, enquanto os modelos de referência são derivados top-down. Os modelos de referência não refletem necessariamente a arquitetura real dos sistemas existentes no domínio.

Figura 10.11 ▶

Modelo de fluxo de dados de um compilador.



10.4.1 Modelos genéricos

Talvez, o exemplo mais conhecido de um modelo genérico de arquitetura seja um modelo de compilador. Milhares de compiladores já foram escritos e, atualmente, está estabelecido, de modo geral, que eles devem ter os seguintes módulos:

1. Um analisador léxico, que identifica as marcas (tokens) de entrada de linguagem e as converte em algum formulário interno.
2. Uma tabela de símbolos, construída pelo analisador léxico, que mantém as informações sobre nomes e tipos utilizados no programa.
3. Um analisador sintático, que verifica a sintaxe da linguagem que está sendo compilada. Ele utiliza uma gramática definida da linguagem e constrói uma árvore de sintaxe.
4. Uma árvore de sintaxe, que é uma estrutura interna que representa o programa que está sendo compilado.
5. Um analisador semântico, que utiliza as informações da árvore de sintaxe e da tabela de símbolos para verificar a correção semântica do programa de entrada.
6. Um gerador de códigos, que 'percorre' a árvore de sintaxe e gera o código de máquina.

Também podem ser incluídos outros componentes que transformam a árvore de sintaxe, a fim de melhorar a eficiência e remover a redundância do código de máquina gerado.

Os componentes que constituem um compilador podem ser organizados de acordo com diferentes modelos de arquitetura. Como Garlan e Shaw (1993) indicaram, os compiladores podem ser implementados com o uso de um modelo composto. Uma arquitetura de fluxo de dados pode ser utilizada com a tabela de símbolos que atua como um repositório para dados compartilhados. As fases da análise léxica, sintática e semântica são organizadas seqüencialmente, como mostra a Figura 10.11.

Esse modelo ainda é amplamente utilizado e é eficaz em ambientes de lotes, em que os programas são compilados e executados, sem a interação do usuário. Contudo, ele é menos eficaz quando o compilador deve ser integrado com outras ferramentas de processamento de linguagem, como um sistema de edição estruturado, um depurador interativo, uma impressora de programas, entre outras. Os componentes genéricos do sistema podem, então, ser organizados em um modelo com base em repositório, como mostra a Figura 10.12.

Nesse modelo de compilador, a tabela de símbolos e a árvore de sintaxe atuam como um repositório central de informações. Ferramentas ou fragmentos de ferramentas comunicam-se por meio dele. Outras informações, como a definição gramatical e a definição de formato de saída para o programa, foram obtidas a partir das ferramentas e colocadas no repositório.

Na prática, existe um número muito grande de modelos de arquitetura de domínio específico. Alguns outros exemplos de arquiteturas de processo genérico para sistemas de tempo real estão ilustrados no Capítulo 13.

Contudo, poucos modelos genéricos, de domínio específico, estão disponíveis publicamente. As organizações que desenvolvem esses modelos os vêem como uma valiosa propriedade intelectual, de que elas precisam para o futuro desenvolvimento de sistemas. Com freqüência, esses modelos representam a arquitetura tipo produto de linha em que produtos relacionados são implementados utilizando-se a mesma arquitetura básica. Por exemplo, drivers de impressoras com diferentes capacidades (velocidade, opções de papel etc.) podem, todos eles,

Figura 10.12 Modelo de repositório de um sistema de processamento de linguagens.

utilizar a mesma arquitetura básica. As arquiteturas tipo produto de linha serão discutidas no Capítulo 14.

10.4.2 Arquiteturas de referência

Os modelos genéricos de arquitetura refletem a arquitetura dos sistemas existentes. Por outro lado, os modelos de referência são, de modo geral, derivados de um estudo do domínio da aplicação. Eles representam uma arquitetura idealizada que inclui todas as características que o sistema pode incorporar.

As arquiteturas de referência podem ser utilizadas como uma base para a implementação de sistemas. Esta foi a intenção básica do modelo de referência OSI (Zimmermann, 1980) para a interconexão de sistemas abertos. O modelo foi imaginado como um padrão. Se um sistema fosse compatível com esse modelo, seria capaz de se comunicar com outros sistemas também compatíveis. Assim, um sistema de controle de estoque em um supermercado, que seguisse o modelo OSI, poderia intercambiar dados diretamente com o sistema de pedidos do fornecedor.

Contudo, os modelos de referência não devem normalmente ser considerados um caminho para a implementação. Em vez disso, sua função principal é servir como meio de comparar diferentes sistemas em um domínio. Um modelo de referência fornece um vocabulário para comparação, que atua como um padrão, em relação ao qual os sistemas podem ser avaliados.

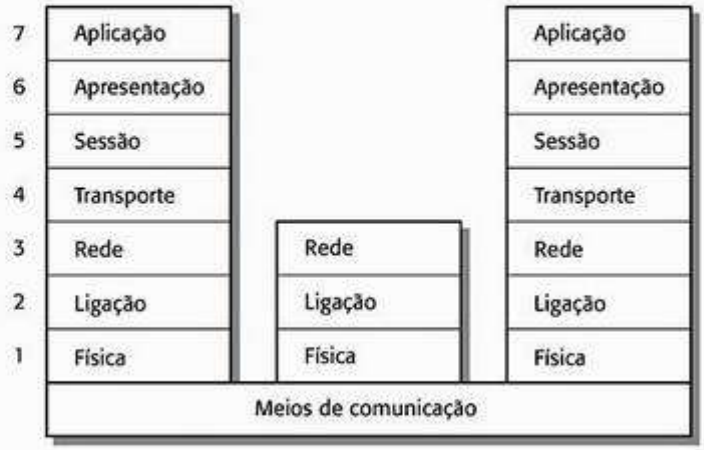
O modelo OSI é composto de sete camadas para a interconexão de sistemas abertos, conforme ilustra a Figura 10.13. As funções exatas das diferentes camadas não são importantes no momento. Essencialmente, as camadas inferiores se referem à interconexão, as camadas intermediárias, à transferência de dados, e as camadas superiores, à transferência de informações semanticamente significativas da aplicação, como documentos padronizados e assim por diante.

Os projetistas do modelo OSI tinham o objetivo muito prático de definir um padrão, de modo que os sistemas compatíveis pudessem se comunicar uns com os outros. Cada camada deveria depender somente da camada inferior a ela. À medida que a tecnologia se desenvolvesse, uma camada poderia ser reimplementada, de modo transparente, sem afetar os sistemas que utilizavam as outras camadas.

Na prática, contudo, os problemas da abordagem com base em camadas para a modelagem de arquitetura comprometeram esse objetivo. Devido às grandes diferenças entre redes, a simples interconexão pode ser impossível. Embora as características funcionais de cada camada estejam bem definidas, as características não funcionais não estão especificadas. Os desenvolvedores de sistemas têm de implementar seus próprios recursos de alto nível e 'pular' camadas. Como alternativa, eles devem projetar características não padronizadas para melhorar o desempenho do sistema.

Figura 10.13 ▶

Arquitetura do modelo de referência OSI.



Conseqüentemente, a substituição transparente de uma camada no modelo é, na prática, impossível. Contudo, essa condição não diminui a utilidade do modelo, uma vez que fornece uma base para a estruturação abstrata e a implementação sistemática de comunicações entre sistemas.

Outros modelos de referência que foram propostos incluem um modelo para ambientes CASE (ECMA, 1991) e um modelo para fábricas de software (Rockwell e Gera, 1993). Alguns padrões de projeto (*design patterns*) (Gamma *et al.*, 1995), que serão discutidos no Capítulo 14, podem também ser considerados arquiteturas de referência.

PONTOS-CHAVE

- A arquitetura de software é o *framework* fundamental para a estruturação do sistema. Diferentes modelos de arquitetura, como um modelo estrutural, um modelo de controle e um modelo de decomposição, podem ser desenvolvidos durante o projeto de arquitetura.
- Os grandes sistemas raramente são compatíveis com um único modelo de arquitetura. Eles são heterogêneos e incorporam diferentes modelos em diferentes níveis de abstração.
- Os modelos de decomposição de sistemas incluem modelos de repositório, modelos cliente–servidor e modelos de máquina abstrata. Os modelos de repositório compartilham dados por meio de um repositório comum. Os modelos cliente–servidor geralmente distribuem dados. Os modelos de máquina abstrata são organizados em camadas, em que cada camada é implementada com o uso dos recursos fornecidos pela sua camada-base.
- Os modelos de controle incluem os modelos de controle e de evento centralizados. Nos modelos centralizados, as decisões de controle são feitas e dependem do estado do sistema e nos modelos de evento, os eventos externos controlam o sistema.
- Os modelos de decomposição em módulos incluem os modelos de fluxo de dados e de objeto. Os modelos de fluxo de dados são funcionais, enquanto os modelos de objeto têm como base entidades fracamente associadas, que mantêm seu próprio estado e suas operações.
- Os modelos de arquitetura de domínio específico são abstrações em relação a um domínio de aplicação. Os modelos de domínio específico podem ser modelos genéricos construídos bottom-up a partir de sistemas existentes ou podem ser modelos de referência, que são os modelos abstratos e idealizados do domínio.

LEITURAS SUGERIDAS

A arquitetura de software é um tópico importante e existe um grande número de livros disponíveis. Os livros que considero úteis são:

Software architecture in practice. Trata-se de uma discussão prática sobre arquiteturas de software, que não exagera na abordagem e apresenta uma visão corporativa racional dos motivos pelos quais as arquiteturas são importantes. (L. Bass, P. Clements e R. Kazman, 1998, Addison Wesley.)

Systems engineering: coping with complexity. O Capítulo 4 desse livro sobre engenharia de sistemas discute o projeto de arquitetura de sistemas, em vez de simplesmente abordar o tema software, apresentando uma perspectiva diferente e agradável. (R. Stevens, P. Brook, K. Jackson e S. Arnold, 1998, Prentice Hall.)

Software architecture: perspectives on an emerging discipline. Esse foi o primeiro livro publicado sobre arquitetura de software e traz uma boa discussão sobre diferentes estilos de arquitetura. (M. Shaw e D. Garlan, 1996, Prentice Hall.)

EXERCÍCIOS

10.1 Explique por que pode ser necessário projetar a arquitetura de sistemas antes que as especificações sejam escritas.

10.2 Construa uma tabela mostrando as vantagens e as desvantagens dos diferentes modelos estruturais discutidos neste capítulo.

10.3 Justifique sua resposta, sugerindo um modelo estrutural apropriado para os seguintes sistemas:

- um sistema automatizado de emissão de passagens, utilizado pelos passageiros de uma estação ferroviária;
- um sistema de videoconferência controlado por computador, que permite que dados de computadores, de áudio e de vídeo sejam visíveis para vários participantes ao mesmo tempo, e
- uma máquina robotizada de limpar chão, que se destina a limpar espaços relativamente abertos, como corredores. A máquina deve ser capaz de identificar paredes e outras obstruções.

10.4 Projete uma arquitetura para o sistema acima com base em sua escolha de modelo. Faça suposições razoáveis sobre os requisitos de sistema.

10.5 Explique por que um modelo de controle de retorno de chamadas geralmente não é adequado para sistemas em tempo real que controlam algum processo.

10.6 Sugira um modelo de controle apropriado para os seguintes sistemas e justifique sua resposta:

- um sistema de processamento em lotes, que capta as informações sobre horas trabalhadas e pagamento e imprime holerites e informações sobre transferência de crédito bancário;
- um conjunto de ferramentas de software que são produzidas por diferentes fabricantes, mas que devem operar em conjunto, e
- um controlador de televisão que responde a sinais de uma unidade de controle remoto.

10.7 Discuta as vantagens e as desvantagens da facilidade de distribuição do modelo de fluxo de dados e do modelo de objetos. Suponha que sejam necessárias uma máquina única e as versões distribuídas de uma aplicação.

10.8 Você deve comparar dois conjuntos de ferramentas CASE integradas. Explique como é possível utilizar um modelo de referência CASE (Brown *et al.*, 1992) para fazer essa comparação.

10.9 Na sua opinião, deve haver uma profissão separada de 'arquiteto de software', cujo papel seria trabalhar independentemente com um cliente para projetar uma arquitetura de software? Essa arquitetura seria, então, implementada por alguma empresa de software. Quais poderiam ser as dificuldades para estabelecer essa profissão?

11 Arquiteturas de sistemas distribuídos

Objetivos

O objetivo deste capítulo é descrever, com mais detalhes, as arquiteturas de sistemas de software distribuídos. Depois de ler este capítulo, você poderá:

- compreender as principais vantagens e desvantagens das arquiteturas de sistemas distribuídos;
- compreender diferentes abordagens do desenvolvimento de arquiteturas cliente–servidor;
- compreender as diferenças entre as arquiteturas cliente–servidor e de objetos distribuídos, e
- compreender o conceito de um requisitor de objeto (*object request broker*) e os princípios básicos dos padrões CORBA.

Sumário

11.1 Arquiteturas de multiprocessadores
11.2 Arquiteturas cliente–servidor
11.3 Arquiteturas de objetos distribuídos
11.4 CORBA

Praticamente todos os sistemas com base em grandes computadores são, hoje em dia, sistemas distribuídos. Um sistema distribuído é aquele em que as informações em fase de processamento são distribuídas para vários computadores, em vez de ficarem confinadas a uma única máquina. Com certeza, a engenharia de sistemas distribuídos tem muito em comum com a engenharia de software, mas existem questões específicas que devem ser levadas em conta quando se projeta esse tipo de sistema. Já foram apresentadas algumas dessas questões na introdução à arquitetura cliente-servidor, no Capítulo 10, e neste capítulo esse assunto é abordado com mais detalhes.

Os engenheiros de software devem estar cientes dessas questões relacionadas a projetos, porque os sistemas distribuídos são amplamente utilizados. Até recentemente, a maioria dos grandes sistemas era composta de sistemas centralizados executados em um único mainframe com terminais conectados a ele. Esses terminais tinham pouca ou nenhuma capacidade de processamento, de modo que todo o processamento de informações era de responsabilidade do computador mainframe. Os projetistas não tinham de pensar nas questões de processamento distribuído. Atualmente, há três tipos principais de sistema:

1. Sistemas pessoais, que não são distribuídos e são projetados para serem executados em um computador pessoal ou em uma workstation (estação de trabalho). Entre os exemplos desse tipo de sistema estão os processadores de texto, as planilhas de cálculo e os sistemas gráficos.
2. Sistemas embutidos, que são executados em um único processador ou em um grupo integrado de processadores. Entre os exemplos desse tipo de sistema estão os sistemas de controle para dispositivos domésticos e os sistemas de gerenciamento de instrumentos.
3. Sistemas distribuídos, em que o software de sistema é executado em um grupo de processadores fracamente integrados, que cooperam entre si conectados por uma rede. Dentre os exemplos desses sistemas destacam-se os sistemas de ATMs, os sistemas de reserva, os sistemas de *groupware* etc.

Hoje em dia, existem limites bem distintos entre esses tipos de sistema, mas esses limites provavelmente vão desaparecer no futuro. À medida que os sistemas de redes sem fio de alta velocidade se tornarem amplamente disponíveis, será possível integrar dinamicamente produtos com sistemas embutidos, como agendas eletrônicas com sistemas mais genéricos.

Coulouris *et al.* (1994) identificam seis importantes características dos sistemas distribuídos:

1. *Compartilhamento de recursos* Um sistema distribuído permite o compartilhamento de recursos de hardware e de software, como discos, impressoras, arquivos e compiladores, que estão associados a diferentes computadores em uma rede. O compartilhamento de recursos é, com certeza, também possível em sistemas para multiusuários, mas, nesse caso, todos os recursos têm de ser fornecidos e gerenciados por um computador central.
2. *Abertura* A abertura de um sistema diz até que ponto ele pode ser ampliado, adicionando-se novos recursos não proprietários a ele. Os sistemas distribuídos são sistemas abertos, que normalmente incluem hardware e software de diferentes fabricantes.
3. *Concorrência* Em um sistema distribuído, vários processos podem operar ao mesmo tempo em diferentes computadores na rede. Esses processos podem (mas não precisam) se comunicar uns com os outros, durante sua operação normal.
4. *Escalabilidade* Em princípio, pelo menos, os sistemas distribuídos são escalonáveis, no sentido de que a capacidade do sistema pode ser aumentada pelo acréscimo de novos recursos, a fim de atender a novas demandas relacionadas ao sistema. Na prática, a escalabilidade pode ser limitada pela rede que conecta os computadores individuais ao sistema. Se muitos computadores novos forem acrescentados, então a capacidade da rede poderá se tornar inadequada.

5. *Tolerância a defeitos* A disponibilidade de vários computadores e o potencial de duplicar informações significa que os sistemas distribuídos podem ser tolerantes a algumas falhas de hardware e software (veja o Capítulo 18). Na maioria dos sistemas distribuídos, um serviço de má qualidade pode resultar da ocorrência de falhas; a perda completa de serviço tende a ocorrer somente quando existe uma falha de rede.

6. *Transparência* A transparência significa que o usuário não necessita saber da natureza distribuída do sistema. Um objetivo do projeto de sistema pode ser que os usuários tenham acesso inteiramente transparente aos recursos e não precisem saber nada sobre a distribuição do sistema. Contudo, em muitos casos, possibilitar que os usuários tenham algum conhecimento da organização do sistema permite que eles façam melhor uso dos recursos.

Naturalmente, os sistemas distribuídos apresentam algumas desvantagens:

- *Complexidade* Os sistemas distribuídos são mais complexos do que os sistemas centralizados, dificultando a compreensão de suas propriedades emergentes e a realização de testes desses sistemas. Por exemplo, em vez de o desempenho do sistema depender da velocidade de um processador, ele depende da largura de banda da rede e da velocidade dos diferentes processadores na rede. Mover recursos de uma parte do sistema para outra pode afetar radicalmente o desempenho do sistema.

- *Proteção* O sistema pode ser acessado a partir de vários computadores diferentes e o tráfego na rede pode estar sujeito à interceptação. Isso torna mais difícil gerenciar a proteção em um sistema distribuído.

- *Facilidade de gerenciamento* Os diferentes computadores em um sistema podem ser de diferentes tipos e podem executar diferentes versões do sistema operacional. Os defeitos em uma máquina podem se propagar para outras máquinas, o que resulta em conseqüências inesperadas. Isso tudo significa que a necessidade de maior esforço para gerenciar e manter o sistema em operação é solicitada.

- *Imprevisibilidade* Como todos os usuários da Web sabem, os sistemas distribuídos são imprevisíveis em suas respostas. Essas respostas dependem da carga total sobre o sistema, de sua organização e da carga na rede. Como tudo isso pode mudar, em curto prazo, o tempo de resposta à solicitação de um usuário pode variar repentinamente de uma solicitação para outra.

Além de discutir as vantagens e as desvantagens dos sistemas distribuídos, Coulouris *et al.* (1994) identificam as principais questões de projeto para os sistemas distribuídos. Essas questões são mostradas na Figura 11.1. Neste capítulo, são abordadas as arquiteturas de software distribuídas, porque na minha opinião esse assunto é muito importante para a engenharia de software. Livros especializados sobre os sistemas distribuídos tratam desses outros tópicos.

O desafio para os projetistas de sistemas distribuídos está em projetar software e hardware para que eles forneçam características desejáveis dos sistemas distribuídos e, ao mesmo tempo, minimizem os problemas inerentes a esses sistemas. Para fazer isso, é preciso que eles compreendam as vantagens e as desvantagens das diferentes arquiteturas de sistemas distribuídos. Neste texto, são analisados dois tipos genéricos de arquitetura de sistemas distribuídos:

1. *Arquiteturas cliente-servidor* Nessa abordagem, podemos pensar no sistema como um conjunto de serviços que são fornecidos aos clientes que os utilizam. Servidores e clientes são tratados diferentemente nesses sistemas.

2. *Arquiteturas de objetos distribuídos* Nesse caso, não há nenhuma distinção entre servidores e clientes e podemos pensar no sistema como um conjunto de objetos que interagem, cuja localização é irrelevante. Não existe distinção entre um provedor de serviços e um usuário desses serviços.

Figura 11.1 ▶

Questões no projeto de sistemas distribuídos.

Questão de projeto	Descrição
Identificação de recursos	Os recursos em um sistema distribuído estão espalhados em diferentes computadores e um esquema de nomes tem de ser constituído para que os usuários possam descobrir e referir-se aos recursos de que necessitam. Um exemplo desse esquema de denominações é o URL (*uniform resource locator* – localizador uniforme de recursos), que é utilizado para identificar páginas Web. Se um esquema de identificação significativo e universalmente conhecido não for utilizado, muitos desses recursos serão inacessíveis aos usuários de sistemas.
Comunicações	A disponibilidade universal da Internet e a eficiente implementação dos protocolos de comunicação TCP/IP, da Internet, propõem que, para a maioria dos sistemas distribuídos, este é o modo mais eficaz de os computadores se comunicarem. Contudo, quando houver requisitos específicos de desempenho, confiabilidade, entre outros, poderão ser utilizadas abordagens alternativas para as comunicações.
Qualidade do serviço	A qualidade do serviço oferecido por um sistema reflete seu desempenho, sua disponibilidade e sua confiabilidade. Ela é afetada por uma série de fatores, como a alocação de processos em processos no sistema, a distribuição de recursos no sistema, a rede e o hardware do sistema e a adaptabilidade do sistema.
Arquiteturas de software	A arquitetura de software descreve como a funcionalidade do aplicativo é distribuída em uma série de componentes lógicos e como esses componentes são distribuídos nos processadores. Escolher a arquitetura certa para um aplicativo é essencial para atingir a qualidade de serviço desejada.

Os diferentes componentes de um sistema distribuído podem ser implementados em diferentes linguagens de programação e podem ser executados em tipos de processadores inteiramente diferentes. Os modelos de dados, a representação das informações e os protocolos de comunicação podem, todos eles, ser diferentes. Portanto, um sistema distribuído exige um software que possa gerenciar essas diversas partes e garantir que elas se comuniquem e troquem dados. O termo *middleware* é utilizado para se referir a esse software – pois ele é o intermediário dos diferentes componentes distribuídos do sistema.

Bernstein (1996) resume os diferentes tipos de *middleware* que estão disponíveis para possibilitar a computação distribuída. O *middleware* é um software de propósito geral, que normalmente é adquirido no mercado, em vez de ser escrito especialmente por desenvolvedores de aplicações. Entre os exemplos de *middleware* estão os softwares para gerenciamento de comunicações com banco de dados, os gerentes de transações, os conversores de dados e os controladores de comunicações. Posteriormente, neste capítulo, serão descritas as estruturas dos sistemas distribuídos, que são uma classe muito importante de *middleware*.

De modo geral, os sistemas distribuídos são desenvolvidos com o uso de uma abordagem orientada a objetos. Eles são constituídos de partes independentes, pouco integradas, e cada uma delas pode interagir diretamente com os usuários ou com outras partes do sistema. As partes do sistema podem precisar responder a eventos independentes. Os objetos de software refletem essas características e, assim, são uma abstração natural para os componentes de sistemas distribuídos. Se o leitor não estiver familiarizado com o conceito de objetos, será recomendável a leitura do Capítulo 12 antes de prosseguir neste capítulo.

11.1 Arquiteturas de multiprocessadores

O modelo mais simples de sistema distribuído é um sistema de multiprocessador, que consiste em uma série de diferentes processos que podem (mas não precisam) ser executados em processadores separados. Esse modelo é comum em grandes sistemas em tempo real. Como será discutido no Capítulo 13, esses sistemas coletam informações, tomam decisões utilizando essas informações e, então, enviam sinais para os atuadores que modificam o ambiente do sistema.

Logicamente, os processos referentes à coleta de informações, à tomada de decisões e ao controle de atuadores podem, todos eles, ser executados em um único processador sob o controle de um escalador (*scheduler*). A utilização de vários processadores melhora o desempenho e a capacidade de recuperação do sistema. A distribuição de processos para os processadores pode ser predeterminada (isso é comum em sistemas importantes) ou ficar sob o controle de um escalador, que decide qual processo alocar para cada processador.

Um exemplo desse tipo de sistema é mostrado na Figura 11.2, que é um modelo simplificado de um sistema de controle de tráfego. Um conjunto de sensores distribuídos coleta informações sobre o fluxo de tráfego e as processa localmente, antes de enviá-las para a sala de controle. Os operadores tomam decisões utilizando essas informações e dão instruções para um processo separado de controle de luzes de tráfego. Nesse exemplo, há processos lógicos separados para gerenciar os sensores, a sala de controle e as luzes de tráfego. Esses processos lógicos podem ser processos individuais ou um grupo de processos. Nesse exemplo, eles são executados em processadores separados.

Os sistemas de software compostos de vários processos não são necessariamente sistemas distribuídos. Se mais de um processador estiver disponível, então a distribuição poderá ser implementada, mas os projetistas de sistema não precisam sempre considerar as questões de distribuição durante o processo de projeto. A abordagem de projeto para esse tipo de sistema é essencialmente aquela utilizada em sistemas de tempo real, como será discutido no Capítulo 13.

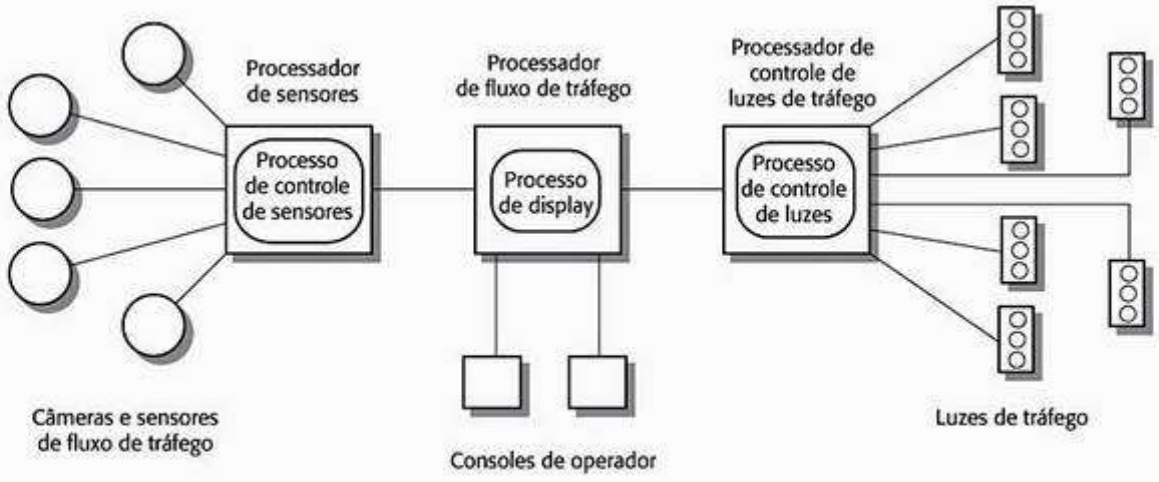
11.2 Arquiteturas cliente–servidor

Já apresentei o conceito de arquiteturas cliente–servidor no Capítulo 10. Em uma arquitetura cliente–servidor, uma aplicação é modelada como um conjunto de serviços que são fornecidos por servidores e um conjunto de clientes que utilizam esses serviços (Orfali e Harkey, 1998). Os clientes precisam estar cientes dos servidores que estão disponíveis, mas, em geral, eles não sabem da existência de outros clientes. Clientes e servidores são processos diferentes, como mostra a Figura 11.3, que é um modelo lógico de uma arquitetura cliente–servidor distribuída.

Figura 11.2 ▼

Um sistema de controle de tráfego multiprocessado.

Não existe necessariamente um mapeamento de 1:1 entre os processos e os processadores do sistema. A Figura 11.4 mostra a arquitetura física de um sistema com seis computadores clientes e dois computadores servidores. Eles podem executar os processos cliente e servidor, mostrados



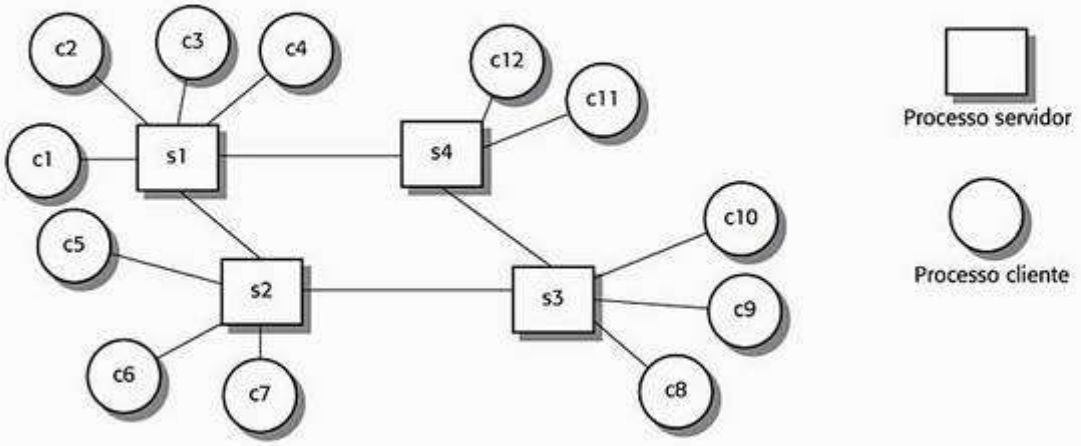


Figura 11.3 ▲

Um sistema cliente–servidor.

na Figura 11.3. Em geral, quando me refiro a clientes e servidores, refiro-me a esses processos lógicos e não a computadores físicos nos quais esses processos são executados.

O projeto de sistema cliente–servidor deve refletir a estrutura lógica da aplicação que está sendo desenvolvida. Um modo de ver uma aplicação está ilustrado na Figura 11.5, que mostra uma aplicação estruturada em três camadas. A camada de apresentação se ocupa de apresentar informações aos usuários e todas as interações com eles. A camada de processamento de aplicações, por sua vez, se ocupa de implementar a lógica da aplicação e a camada de gerenciamento de dados, de todas as operações de bancos de dados. Em sistemas centralizados, essas operações não precisam ser nitidamente separadas. Contudo, quando se está projetando um sistema distribuído, deve-se fazer uma clara distinção entre elas, pois, então, torna-se possível distribuir cada camada para um computador diferente.

A arquitetura cliente–servidor mais simples é chamada de arquitetura cliente–servidor de duas camadas, em que uma aplicação é organizada como um servidor (ou vários servidores idênticos) e um conjunto de clientes. Como ilustra a Figura 11.6, as arquiteturas cliente–servidor de duas camadas podem ser de dois tipos:

1. *Modelo cliente-magro* Nesse modelo, todo o processamento da aplicação e gerenciamento de dados é realizado no servidor. O cliente é responsável simplesmente por executar o software de apresentação.

2. *Modelo cliente-gordo* Nesse modelo, o servidor é responsável somente pelo gerenciamento de dados. O software no cliente implementa a lógica da aplicação e as interações com o usuário do sistema.

Figura 11.4 ▶

Computadores em uma rede cliente–servidor.

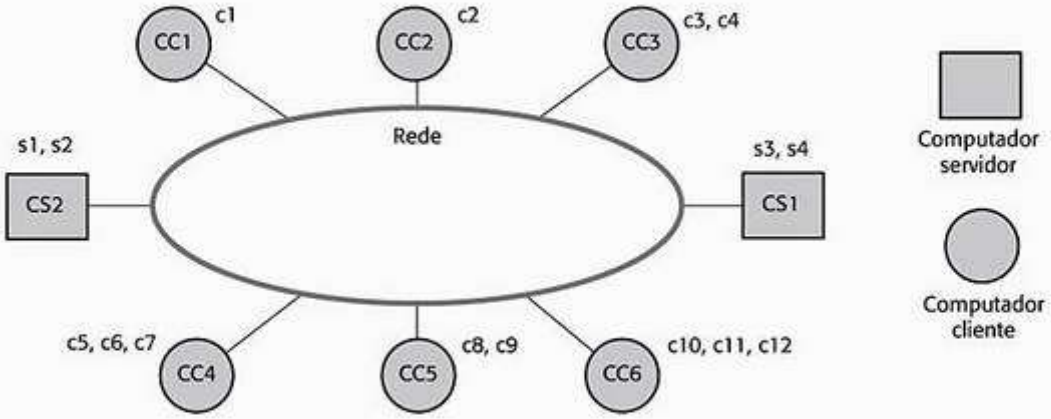


Figura 11.5

Camadas da aplicação.

Uma arquitetura do tipo cliente-magro, composta de duas camadas, é a abordagem mais simples para se utilizar quando sistemas legados centralizados (veja o Capítulo 26) evoluem para uma arquitetura cliente-servidor. A interface com o usuário para esses sistemas migra para PCs e a aplicação em si atua como um servidor e controla todo o processamento de aplicações e o gerenciamento de dados. Um modelo cliente-magro pode também ser implementado quando os clientes forem simples dispositivos de rede, no lugar de PCs ou workstations. O dispositivo de rede executa um browser de Internet e a interface com o usuário é implementada por meio desse sistema.

Uma grande desvantagem do modelo cliente-magro é que ele atribui uma grande carga de processamento ao servidor e à rede. O servidor é responsável por todo o cálculo, o que pode resultar em um grande tráfego de rede entre o cliente e o servidor. Existe uma grande capacidade de processamento disponível nos PCs modernos, que fica, na maior parte, sem uso quando se opta pela abordagem do tipo cliente-magro.

Em contrapartida, o modelo cliente-gordo utiliza essa capacidade de processamento disponível e distribui o processamento lógico da aplicação e a apresentação ao cliente. O servidor é, essencialmente, um servidor de transações que gerencia todas as transações do banco de dados. Um exemplo bem conhecido desse tipo de arquitetura são os sistemas ATM, em que o ATM é o cliente e o servidor é um mainframe que executa o banco de dados da conta do cliente.

Esse sistema de rede ATM está ilustrado na Figura 11.7. Observe que os ATMs não se conectam diretamente ao banco de dados do cliente, mas sim a um monitor de teleprocessamento. Um monitor de teleprocessamento (*teleprocessing* – TP) é um sistema de *middleware*, que organiza as comunicações com os clientes remotos e coloca em série as transações dos clientes para o processamento pelo banco de dados. O uso de transações em série significa que o sistema pode se recuperar a partir de falhas, sem corromper os dados do sistema.

Enquanto o modelo cliente-gordo distribui o processamento de modo mais eficiente do que o modelo cliente-magro, o sistema de gerenciamento é mais complexo. A funcionalidade da aplicação está distribuída em vários computadores diferentes. Quando o software da aplicação precisa ser modificado, isso envolve a reinstalação em cada computador cliente, o que pode representar um custo significativo, se houver centenas de clientes no sistema.

Figura 11.6

Clientes magros e gordos.

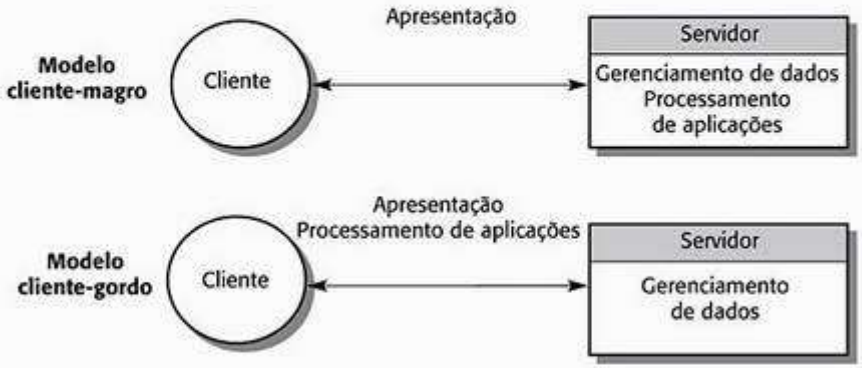
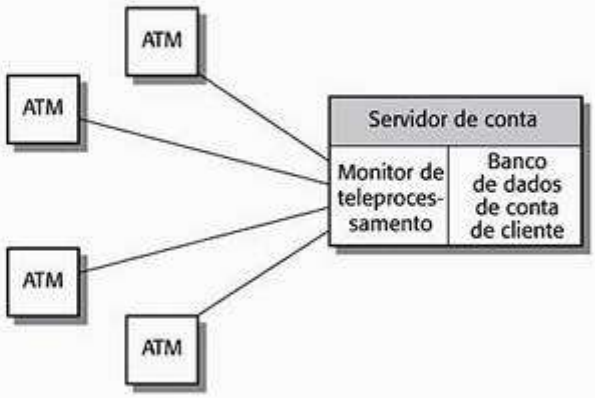


Figura 11.7

Um sistema ATM cliente–servidor.



O advento da linguagem Java e da possibilidade de recorrer a applets (pequenos aplicativos) por download tem permitido o desenvolvimento de um modelo cliente–servidor que pode ser considerado intermediário entre os modelos cliente-magro e cliente-gordo. Alguns softwares de processamento de aplicações podem ser transferidos para o cliente por download, como applets Java, o que alivia, desse modo, a carga do servidor. A interface com o usuário é construída utilizando-se um Web browser que pode executar applets Java. Contudo, os Web browsers de diferentes fornecedores, e mesmo diferentes versões de Web browsers do mesmo fornecedor, nem sempre se apresentam da mesma maneira. Versões anteriores em computadores mais antigos podem não conseguir executar Java. Portanto, é preciso utilizar essa abordagem somente quando se tem certeza de que todos os usuários do sistema têm browsers compatíveis, habilitados para Java.

O problema essencial com a abordagem cliente–servidor de duas camadas é que as três camadas lógicas – apresentação, processamento de aplicações e gerenciamento de dados – devem ser mapeadas em dois sistemas de computadores. Poderão ocorrer problemas com a escalabilidade e o desempenho, se o modelo cliente-magro for escolhido, ou problemas de gerenciamento de sistema, se o modelo cliente-gordo for utilizado. Para evitar esses problemas, uma abordagem alternativa é utilizar uma arquitetura cliente–servidor de três camadas (Figura 11.8). Nessa arquitetura, a apresentação, o processamento de aplicações e o gerenciamento de dados são processos logicamente separados.

Uma arquitetura de software cliente–servidor de três camadas não significa necessariamente que haja três sistemas de computador conectados à rede. Um único servidor pode executar tanto o processamento de aplicações quanto o gerenciamento de dados de aplicações, como servidores lógicos separados. Contudo, se a necessidade aumentar, será relativamente simples separar o processamento de aplicações e o gerenciamento de dados e executá-los em processadores separados.

Um sistema de operações bancárias na Internet é um exemplo do tipo de sistema que pode ser implementado com o uso de uma arquitetura cliente–servidor de três camadas. O banco de dados do cliente do banco (em geral hospedado em um computador mainframe) fornece serviços de gerenciamento de dados; um servidor Web fornece os serviços de aplicativos, como recursos para transferir dinheiro, gerar extratos e pagar contas, e o computador do usuário com um browser de Internet é o cliente. Esse modelo está ilustrado na Figura 11.9; trata-se de um sistema escalonável, uma vez que é relativamente fácil adicionar novos servidores Web, à medida que o número de clientes aumenta.

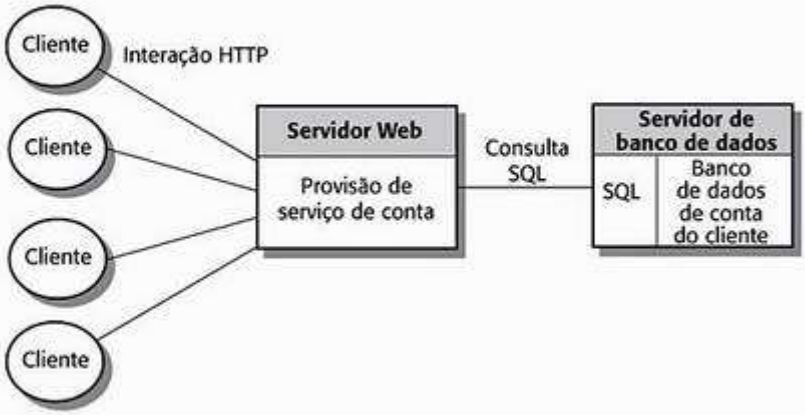
O uso de uma arquitetura de três camadas, nesse caso, permite otimizar a transferência de informações entre o servidor Web e o servidor de banco de dados. As comunicações entre esses

Figura 11.8

Uma arquitetura cliente–servidor de três camadas.

Figura 11.9

Arquitetura de distribuição de um sistema de operações bancárias na Internet.



sistemas não precisam ter como base os padrões da Internet, mas podem utilizar protocolos de comunicações de nível inferior e mais rápidos. Um *middleware* eficaz que aceite consultas de banco de dados em SQL (*structured query language* – linguagem de consulta estruturada) é utilizado para manejar a recuperação de informações a partir do banco de dados.

Em alguns casos, é apropriado ampliar o modelo cliente-servidor de três camadas para uma variante com várias camadas, em que servidores adicionais são acrescentados ao sistema. Os sistemas com múltiplas camadas podem ser utilizados quando as aplicações precisam acessar e utilizar dados de diferentes bancos de dados. Nesse caso, um servidor de integração é colocado entre o servidor de aplicações e os servidores de bancos de dados acessados. O servidor de integração coleta os dados distribuídos e os apresenta à aplicação, como se eles estivessem disponíveis em um único banco de dados.

As arquiteturas cliente-servidor de três camadas e as variantes com várias camadas, que distribuem o processamento de aplicações em diversos servidores, são inerentemente mais escalonáveis do que as arquiteturas de duas camadas. O tráfego de rede fica reduzido, em comparação com as arquiteturas de cliente-magro, compostas de duas camadas. O processamento de aplicações é a parte mais volátil do sistema e pode ser facilmente atualizada porque está localizada no centro. Em alguns casos, o processamento pode ser distribuído entre os servidores de lógica da aplicação e de gerenciamento de dados, levando, assim, à resposta mais rápida para os pedidos dos clientes.

Figura 11.10

Uso de diferentes arquiteturas cliente-servidor (C/S).

Os projetistas de arquiteturas cliente-servidor devem levar em conta uma série de fatores ao escolherem a arquitetura mais apropriada. A Figura 11.10 mostra situações em que as arquiteturas cliente-servidor que foram discutidas são, provavelmente, mais apropriadas.

Arquitetura	Aplicações
Arquitetura C/S de duas camadas, com clientes-magros	Aplicações de sistemas legados ou de processamento de aplicações e o gerenciamento de dados é impraticável. Aplicações de computação intensiva, como compiladores com pouco ou nenhum gerenciamento de dados. Aplicações de uso intensivo de dados (*browsing* e consultas) com pouco ou nenhum processamento de aplicativos.
Arquitetura C/S com clientes-gordos	Aplicações em que o processamento de aplicativos é fornecido por COTS, sistemas comerciais de prateleira (por exemplo, Microsoft Excel), no cliente. Aplicações em que é exigido o processamento de dados de computação intensiva (por exemplo, visualização de dados). Aplicações com funcionalidade relativamente estável para o usuário final, utilizadas em um ambiente de gerenciamento de sistemas bem-estabelecido.
Arquitetura C/S com três ou múltiplas camadas	Aplicações em grande escala, com centenas ou milhares de clientes. Aplicações em que os dados e as aplicações são voláteis. Aplicações em que os dados de múltiplas fontes são integrados.

11.3 Arquiteturas de objetos distribuídos

No modelo cliente–servidor de um sistema distribuído, clientes e servidores são diferentes. Os clientes recebem serviços dos servidores, e não de outros clientes; os servidores podem atuar como clientes, recebendo serviços de outros servidores, mas não solicitam serviços de clientes, e os clientes devem conhecer os serviços que são oferecidos por servidores específicos e saber como contatar esses servidores. Esse modelo funciona bem para muitos tipos de aplicações. Contudo, ele limita a flexibilidade dos projetistas de sistemas, pelo fato de que eles devem decidir onde os serviços serão fornecidos. Eles devem também planejar a escalabilidade e, assim, oferecer um meio de distribuir a carga nos servidores, à medida que mais clientes forem acrescentados ao sistema.

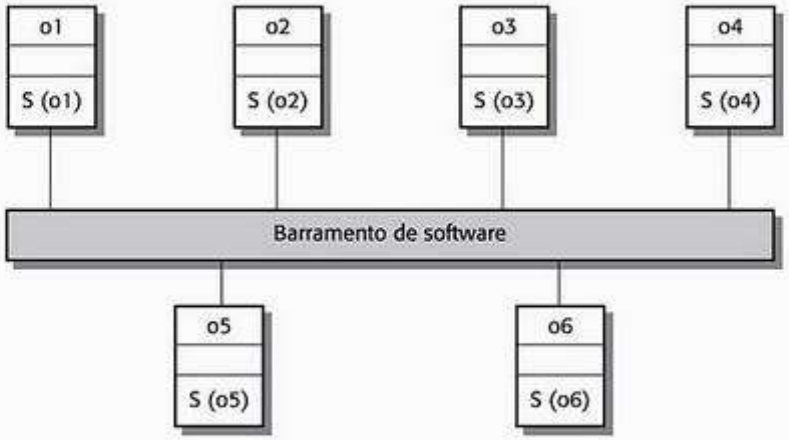
Uma abordagem mais geral para o projeto de sistemas distribuídos é eliminar a distinção entre cliente e servidor e projetar a arquitetura de sistema como uma arquitetura de objetos distribuídos. Em uma arquitetura de objetos distribuídos (Figura 11.11), os componentes fundamentais do sistema são objetos que oferecem uma interface para um conjunto de serviços que eles fornecem. Outros objetos solicitam esses serviços sem fazer nenhuma distinção lógica entre um cliente (quem recebe o serviço) e um servidor (o provedor do serviço).

Os objetos podem ser distribuídos em uma série de computadores em uma rede e podem se comunicar por meio do *middleware*. Por analogia a um barramento de hardware, que permite que diferentes placas sejam conectadas a ele e que aceita comunicação entre dispositivos de hardware, podemos pensar nesse *middleware* como um barramento de software. Ele fornece um conjunto de serviços que permitem que os objetos se comuniquem e sejam acrescentados ao sistema ou removidos dele. Esse *middleware* é chamado de requisitor de objetos e seu papel é fornecer uma interface contínua entre os objetos. Os requisitores de objetos serão discutidos na Seção 11.4.

As vantagens desse modelo de arquitetura de sistema distribuído são:

- Ele permite ao projetista de sistema adiar decisões sobre onde ou como os serviços devem ser fornecidos. Os objetos provedores de serviços podem ser executados em qualquer nó da rede. Portanto, a distinção entre os modelos cliente-gordo e cliente-magro torna-se irrelevante, uma vez que não existe nenhuma necessidade de decidir antecipadamente onde os objetos de lógica de aplicações estão localizados.
- É uma arquitetura de sistemas muito aberta, que permite que novos recursos sejam acrescentados, conforme necessário. Como será discutido na seção seguinte, foram desenvolvidos e implementados padrões de barramento de software que permitem que objetos escritos em diferentes linguagens de programação se comuniquem e forneçam serviços uns aos outros.

Figura 11.11 ▶
Arquitetura de objetos distribuídos.



- O sistema é flexível e extensível. Diferentes instâncias do sistema, com o mesmo serviço fornecido por diferentes objetos ou por objetos duplicados, podem ser criadas para lidar com diferentes cargas do sistema. Novos objetos podem ser acrescentados, à medida que a carga do sistema aumenta, sem interferir em outros objetos do sistema.
- É possível reconfigurar o sistema dinamicamente com objetos que migrem pela rede, conforme necessário. Essa opção pode ser importante, quando existem padrões variáveis de demanda por serviços. Um objeto provedor de serviços pode migrar para o mesmo processador que os objetos solicitantes de serviços, melhorando, assim, o desempenho do sistema.

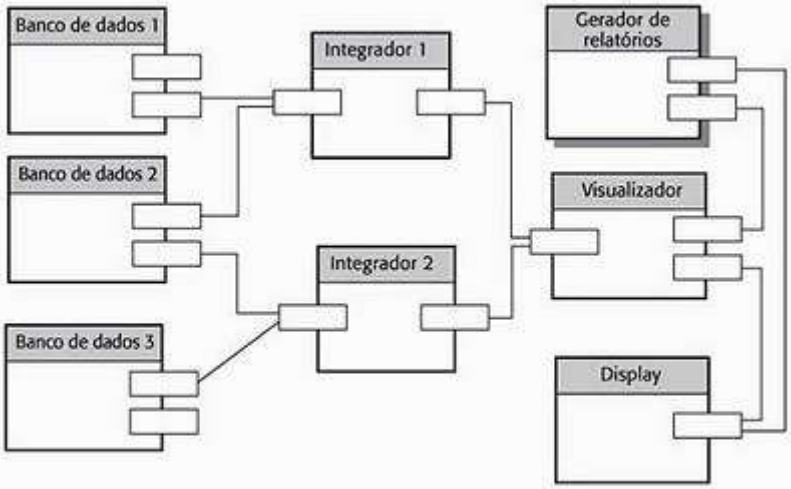
Uma arquitetura de sistema distribuído pode ser utilizada de duas maneiras, em projetos de sistema:

1. Como um modelo lógico que permite estruturar e organizar o sistema. Nesse caso, pense em como oferecer a funcionalidade de aplicativos somente em termos de serviços e combinações de serviços. Então, é possível resolver como fornecer esses serviços, utilizando uma série de objetos distribuídos. Nesse nível, os objetos projetados são, geralmente, objetos com alta granularidade (algumas vezes, chamados de objetos de negócios), que fornecem serviços de domínio específico. Por exemplo, em uma aplicação destinada ao comércio varejista, é possível que haja objetos de negócios voltados para o controle de estoque, as comunicações com clientes, a solicitação de produtos etc. Podemos, então, naturalmente, imaginar esse modelo lógico como um modelo de implementação.
2. Como uma abordagem flexível da implementação de sistemas cliente-servidor. Nesse caso, o modelo lógico do sistema é um modelo cliente-servidor, mas os clientes e os servidores são imaginados como objetos distribuídos, que se comunicam por meio de um barramento de software. Isso torna possível modificar o sistema facilmente, por exemplo, de um sistema de duas camadas para um de múltiplas camadas. Nesse caso, o servidor ou o cliente não pode ser implementado como um único objeto distribuído, mas pode consistir em uma série de objetos menores, dos quais cada um fornece serviços especializados.

Um exemplo de um tipo de sistema em que a arquitetura de objetos distribuídos pode ser apropriada é um sistema de mineração de dados (*data mining system*), que procura por relações entre os dados que estão armazenados em uma série de diferentes bancos de dados (Figura 11.12). Um exemplo de aplicação de mineração de dados pode estar onde um comércio no varejo apresenta diferentes tipos de lojas (ou seja, lojas de alimentos e lojas de utensílios domésticos), que tentam encontrar relações entre as compras de diferentes tipos de alimentos e diferentes tipos de utensílios. Por exemplo, as pessoas que compram alimentos para recém-nascidos

Figura 11.12 ▶

Arquitetura de distribuição de um sistema de mineração de dados.



podem também comprar determinados tipos de papel de parede. Ao saber disso, a loja pode se direcionar especificamente aos compradores de alimentos para recém-nascidos, com ofertas combinadas.

Nesse exemplo, cada banco de dados pode ser encapsulado como um objeto distribuído, com uma interface que fornece acesso *read-only* ('somente para leitura') aos seus dados. Os objetos integradores se ocupam, cada um deles, com tipos específicos de relacionamentos e coletam informações de todos os bancos de dados para tentar deduzi-los. É possível que haja um objeto integrador que se ocupe com as variações sazonais nas mercadorias vendidas e outro que se ocupe com as relações entre os diferentes tipos de mercadorias.

Os objetos de visualização interagem com os objetos integradores, a fim de produzirem uma visualização ou um relatório sobre as relações descobertas. Devido aos grandes volumes de dados manipulados, os objetos de visualização normalmente utilizarão representações gráficas das relações descobertas. A apresentação de informações gráficas será discutida no Capítulo 15.

Uma arquitetura de objetos distribuídos, em vez de uma arquitetura cliente-servidor, é apropriada para esse tipo de aplicação por três razões:

1. Diferentemente de um sistema ATM bancário, por exemplo, o modelo lógico do sistema não é o de provisão de serviços, em que há serviços distintos de gerenciamento de dados.
2. Ela permite que o número de bancos de dados acessados seja aumentado, sem interferir no sistema. Cada banco de dados é, simplesmente, um outro objeto distribuído. Os objetos de bancos de dados podem fornecer uma interface simplificada, que controla o acesso aos dados. Os bancos de dados acessados podem residir em diferentes máquinas.
3. Ela permite que novos tipos de relacionamentos sejam selecionados pela adição de novos objetos integradores. Partes dos negócios que estão interessadas em relacionamentos específicos podem ampliar o sistema pela adição de objetos integradores, que operam em seus computadores sem exigir o conhecimento de qualquer outro integrador utilizado em outro local.

A maior desvantagem das arquiteturas de objetos distribuídos é que elas são mais complexas de ser projetadas do que os sistemas cliente-servidor, que parecem ser uma maneira bem natural de pensar a respeito dos sistemas. Esses sistemas refletem muitas transações humanas, em que as pessoas solicitam e recebem serviços de outras pessoas, que se especializam em fornecer esses serviços. É mais difícil pensar no fornecimento geral de serviços e ainda não temos muita experiência com o projeto e o desenvolvimento de objetos de negócios com alta granularidade.

11.4 CORBA

Como indiquei na seção anterior, a implementação de uma arquitetura de objetos distribuídos requer *middleware* (requisitores de objetos) para lidar com as comunicações entre os objetos distribuídos. Em princípio, os objetos no sistema podem ser implementados com o uso de diferentes linguagens de programação, podem ser executados em diferentes plataformas e seus nomes não precisam ser conhecidos por todos os outros objetos no sistema. Portanto, o 'elemento de ligação' do *middleware* precisa fazer um bom trabalho para garantir as contínuas comunicações dos objetos.

No momento em que este livro foi escrito, existiam dois padrões principais para o *middleware* trabalhar com a computação de objetos distribuídos:

1. *CORBA (common object request broker architecture* – arquitetura de requisitor de objeto comum) é um conjunto de padrões para *middleware*, definido pela OMG. A OMG é um consórcio de empresas, que inclui empresas de destaque como a Sun, a Hewlett-Packard

e a IBM. Os padrões CORBA definem uma abordagem genérica de máquina independente para a computação de objetos distribuídos. Foi desenvolvida uma série de implementações desse padrão por diferentes fabricantes. As implementações CORBA estão disponíveis para sistemas operacionais Unix e Microsoft.

2. *DCOM (distributed component object model* – modelo de objeto de componentes distribuídos) é um padrão desenvolvido e implementado pela Microsoft e é integrado com o sistema operacional da Microsoft. Seu modelo de computação distribuída é menos genérico do que o modelo CORBA e o DCOM oferece compatibilidade mais limitada com a interoperabilidade. No momento em que este livro foi escrito, o uso do DCOM estava limitado aos sistemas operacionais da Microsoft.

Neste texto, CORBA é destacado, porque é o modelo mais genérico. Na minha opinião, é provável que CORBA, DCOM e outras abordagens, como o RMI (*remote method invocation* – chamada de método remoto) do Java, convirjam, e essa convergência terá como base os padrões CORBA. Não há necessidade de mais do que um padrão nessa área, e diferentes padrões impedem o progresso de longo prazo.

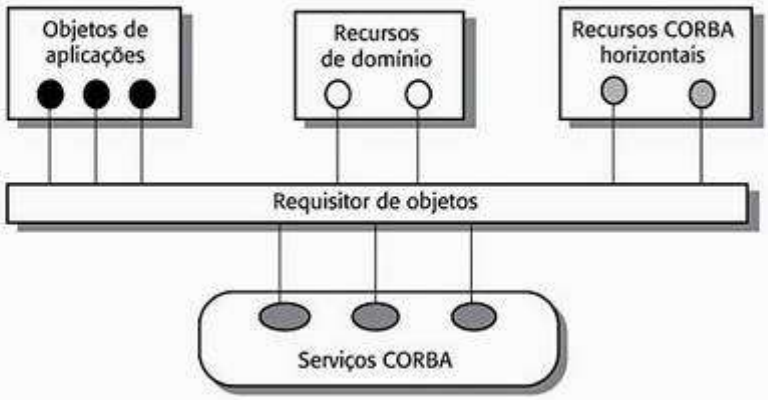
Os padrões CORBA foram definidos pela OMG, que é constituída de mais de 500 empresas, a fim de promover o desenvolvimento orientado a objetos. O papel da OMG é definir padrões para o desenvolvimento orientado a objetos, mas não para fornecer implementações específicas desses padrões. Os padrões definidos estão publicamente disponíveis, gratuitamente, no site Web da OMG. A OMG não se ocupa apenas com CORBA, mas define uma ampla gama de outros padrões, inclusive a UML (Rumbaugh *et al.*, 1999a) e objetos comuns de negócios. Acrescentei um link para esses padrões a partir das páginas Web do livro.

A visão da OMG de uma aplicação distribuída é mostrada na Figura 11.13, que foi adaptada do diagrama de Siegel sobre 'Arquitetura de Gerenciamento de Objetos' (Siegel, 1998). Ele propõe que uma aplicação distribuída deve ser constituída de uma série de componentes:

1. Objetos de aplicações, que são projetados e implementados para essa aplicação.
2. Objetos-padrão, que são definidos pela OMG para um domínio específico. No momento em que este livro foi escrito, uma série de forças-tarefa foram constituídas para definir padrões de objetos de domínio nas áreas de finanças/seguros, comércio eletrônico, serviços de saúde e várias outras áreas.
3. Serviços CORBA fundamentais, que fornecem serviços básicos de computação distribuída, como diretórios, gerenciamento de proteção, entre outros.
4. Recursos CORBA horizontais, como os recursos de interface com o usuário e os recursos de gerenciamento de sistemas. O termo 'recursos horizontais' sugere que esses recursos são comuns a muitos domínios de aplicações e, portanto, utilizados em muitas aplicações diferentes.

Figura 11.13 ▶

Estrutura de uma aplicação distribuída com base no CORBA.



Os padrões CORBA abrangem todos os aspectos dessa visão. Existem quatro elementos principais nesses padrões:

1. Um modelo de objetos para objetos de aplicação, em que um objeto CORBA é um encapsulamento de estado, com uma interface bem definida, de linguagem neutra, descrita em uma IDL (*interface definition language* – linguagem de definição de interface).
2. Um requisitor de objeto (ORB), que gerencia solicitações de serviços de objetos. Ele localiza o objeto que fornece o serviço, prepara-o para a requisição, envia a requisição de serviço e retorna o resultado ao solicitante.
3. Um conjunto de serviços de objetos, que são serviços gerais que provavelmente serão requisitados por muitas aplicações distribuídas. Por exemplo, serviços de diretórios, serviços de transações e serviços de persistência.
4. Um conjunto de componentes comuns construído tendo como base esses serviços básicos, que pode ser solicitado por aplicações. Esses componentes podem ser componentes verticais de domínio específico ou componentes horizontais de propósito geral, que são utilizados por muitas aplicações. Eles serão discutidos no Capítulo 14.

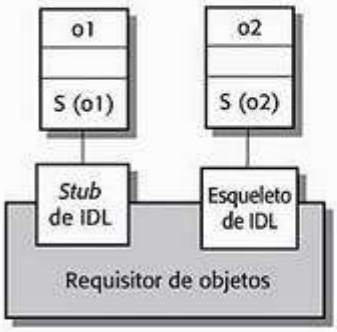
O modelo de objetos CORBA considera que um objeto é um encapsulamento de atributos e serviços, como é normal para objetos. Contudo, os objetos CORBA precisam ter uma definição de interface separada, que especifica os atributos públicos e as operações do objeto. As interfaces dos objetos CORBA são definidas com o uso de uma linguagem de definição de interface padrão, independente da linguagem (IDL). Se um objeto deseja utilizar serviços fornecidos por outro objeto, então ele acessa esses serviços por meio da interface IDL. Os objetos CORBA têm um identificador único, denominado IOR (*interoperable object reference* – referência a objeto interoperável), utilizado quando um objeto requisita serviços de outro objeto.

O requisitor de objetos conhece os objetos que estão requisitando serviços e também suas interfaces. O ORB manipula a comunicação entre os objetos. Os objetos em comunicação não precisam saber a localização de outros objetos nem sobre sua implementação. À medida que a interface IDL isola os objetos em relação ao ORB, é possível modificar a implementação de objeto de um modo inteiramente transparente. A localização do objeto pode mudar entre chamadas, e isso é transparente para outros objetos no sistema.

Essa opção está ilustrada na Figura 11.14, que mostra como dois objetos, o1 e o2, podem se comunicar por meio de um ORB. O objeto chamador o1 tem um *stub* de IDL associado, que define a interface do objeto que está fornecendo o serviço solicitado. O implementador de o1 inclui chamadas para esse *stub* em sua implementação de objetos, quando um serviço é requerido. A IDL é um superconjunto da linguagem C++, de modo que é muito fácil acessar esse *stub*, se a programação estiver sendo feita em C++; e é relativamente fácil, em C ou Java. Mapeamentos de linguagem para IDL também foram definidos para outras linguagens, como Ada e COBOL. Nesses casos, a compatibilidade com a ferramenta é, geralmente, necessária para oferecer suporte ao link para a IDL.

Figura 11.14 ▶

Comunicações de objetos por meio de um ORB.



O objeto que está fornecendo o serviço tem um esqueleto de IDL associado, que vincula a interface à implementação dos serviços. Simplificando, quando um serviço é chamado pela interface, o esqueleto de IDL traduz essa ação em uma chamada para o serviço em qualquer implementação de linguagem que tenha sido utilizada. Quando o método ou o procedimento tiver sido executado, o esqueleto de IDL traduz os resultados para IDL, de modo que eles possam ser acessados pelo objeto que fez a chamada. Quando um objeto fornece serviços para outros objetos e utiliza serviços que são fornecidos em outro local, ele precisa de um esqueleto de IDL e de *stub* de um IDL. Um *stub* de IDL é necessário para cada objeto utilizado.

Os requisitores de objetos não são, geralmente, implementados como processos separados, mas são um *framework* (veja o Capítulo 14), que pode ser vinculado com implementações de objetos. Portanto, em um sistema distribuído, cada computador que estiver executando objetos distribuídos terá seu próprio requisitor de objetos, que manipulará todas as chamadas locais de objetos. Contudo, quando uma requisição é feita para um serviço a ser fornecido por um objeto remoto, isso requer comunicações entre ORBs.

Essa situação está ilustrada na Figura 11.15. Nesse exemplo, se o objeto o1 ou o2 solicitar um serviço de o3 ou o4, será necessário que os ORBs associados se comuniquem. Uma implementação CORBA aceita a comunicação ORB a ORB, fornecendo a todos os ORBs o acesso a todas as definições de interfaces IDL e implementando o GIOP (*generic inter-ORB protocol* – protocolo genérico entre ORBs) padrão do OMG. Esse protocolo define mensagens-padrão que os ORBs podem trocar para implementar a chamada de objetos remotos e a transferência de informações. Quando for combinado com protocolos TCP/IP da Internet de nível inferior, o GIOP permitirá que os ORBs se comuniquem pela Internet.

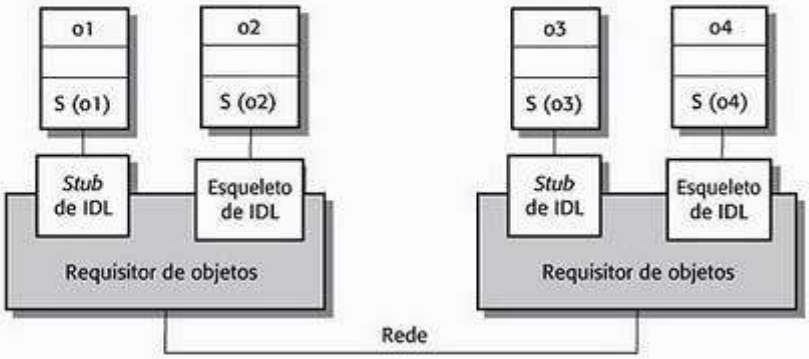
A iniciativa CORBA está em andamento desde a década de 80 e suas primeiras versões apenas se ocupavam em oferecer compatibilidade com objetos distribuídos. Contudo, à medida que os padrões evoluíram, eles se tornaram mais amplos. Além de um mecanismo para as comunicações de objetos distribuídos, os padrões CORBA atualmente definem alguns serviços-padrão, que podem ser fornecidos para aceitar aplicações orientadas a objetos distribuídas.

É possível pensar nos serviços CORBA como recursos que provavelmente deverão ser requeridos por muitos sistemas distribuídos. Os padrões definem aproximadamente 15 serviços comuns. Alguns exemplos desses serviços genéricos são:

1. Serviços de identificação e de negociação, que permitem que objetos se refiram a outros e os descubram na rede. O serviço de identificação é um serviço de diretório, que permite que os objetos sejam nomeados. Outros objetos podem descobrir a referência a objeto interoperável de objetos que desejam utilizar por meio do serviço de identificação. Isso é semelhante às páginas de uma lista telefônica: os objetos que utilizam o serviço precisam saber os nomes registrados de outros objetos. Os serviços de negociação são como as páginas amarelas: os objetos podem descobrir o que outros objetos registraram no serviço de negociação e podem acessar a especificação desses objetos.

2. Serviços de notificação, que permitem aos objetos notificar outros objetos sobre a ocorrência de algum evento. Os objetos podem registrar seu interesse em um evento especí-

Figura 11.15 ▶
Comunicações entre ORBs.



fico com o serviço e, quando esse evento ocorrer, eles serão automaticamente notificados. Por exemplo, digamos que o sistema contém um *spooler* de impressora, que enfileira documentos a serem impressos, e uma série de objetos de impressora. O *spooler* de impressora registra que está interessado no evento 'fim de impressão' de um objeto da impressora. O serviço de notificação informará quando a impressão for concluída e poderá, então, programar o próximo documento na impressora.

3. Serviços de transação, que aceitam transações atômicas e *rollback* em falha. As transações são um recurso da tolerância a defeitos (veja o Capítulo 18), que aceita a recuperação a partir de erros, durante uma operação de atualização. Se uma operação de atualização de objeto falhar, então o estado do objeto poderá ser retornado para o seu estado, antes do início da atualização.

Pretende-se que os padrões CORBA devam incluir definições de interface para uma ampla variedade de componentes que possam ser utilizados por construtores de aplicações distribuídas. Estes podem ser componentes verticais ou horizontais. Os componentes verticais são aqueles específicos de um domínio de aplicação. Como já foi comentado, várias forças-tarefa de diferentes setores da indústria foram estabelecidas para definir esses componentes. Os componentes horizontais são aqueles de propósito geral, como os componentes de interface com o usuário. Uma força-tarefa também é responsável pelo desenvolvimento dessas especificações.

No momento em que este livro foi escrito, essas especificações de componentes ainda estavam em desenvolvimento e não haviam sido estabelecidas. De acordo com meu ponto de vista, é provável que esta seja a área mais problemática dos padrões CORBA e que leve muitos anos para as especificações e as implementações padronizadas de componentes estarem disponíveis.

PONTOS-CHAVE

- Praticamente, todos os grandes sistemas novos são, hoje em dia, sistemas distribuídos, em que o software de sistema é executado em um grupo fracamente integrado de processadores em rede.
- Os sistemas distribuídos podem aceitar o compartilhamento de recursos, a abertura, a concorrência, a escalabilidade, a tolerância a falhas e a transparência.
- Os sistemas cliente–servidor são sistemas distribuídos, modelados como um conjunto de serviços que são fornecidos por servidores para processos de clientes.
- Em um sistema cliente–servidor, a interface com o usuário sempre é executada em um cliente e o gerenciamento de dados sempre é fornecido por um servidor compartilhado. A funcionalidade da aplicação pode ser implementada no computador cliente ou no servidor.
- Em uma arquitetura de objetos distribuídos, não há distinção entre clientes e servidores. Os objetos fornecem serviços gerais que podem ser solicitados por outros objetos. Essa abordagem pode ser utilizada para implementar sistemas cliente–servidor.
- Sistemas de objetos distribuídos requerem *middleware* para manipular as comunicações entre objetos e para permitir que eles sejam adicionados ao sistema e removidos dele. Conceitualmente, podemos pensar nesse *middleware* como um barramento de software em que os objetos se conectam.
- Os padrões CORBA são um conjunto de padrões para *middleware*, que aceita arquiteturas de objetos distribuídos. Entre elas estão as definições de modelos de objetos, as definições de um requisitor de objetos e as definições de serviços comuns. Várias implementações dos padrões CORBA estão disponíveis.

LEITURAS SUGERIDAS

Communications of the ACM, out. 1998. Essa edição especial traz uma série de artigos sobre CORBA, escritos para leitores não especializados. Particularmente, recomendo o artigo de introdução, de autoria de Siegel.

Client–server programming with Java and CORBA. Um guia prático sobre o desenvolvimento de sistemas cliente–servidor, que mostra como desenvolver esses sistemas em Java e discute a abordagem Java à chamada de método remoto e seus relacionamentos com o CORBA. (R. Orfali e D. Harkey, 1998, John Wiley and Sons.)

"Frameworks for component-based client–server computing". Esse trabalho discute as arquiteturas cliente–servidor e também CORBA e DCOM e como eles podem ser utilizados como base para implementar os sistemas cliente–servidor. (S.M. Lewandowski, *Computer Surveys,* 30(1), mar. 1998.)

"Middleware: a model for distributed systems services". Esse trabalho é uma excelente visão geral que resume o papel do *middleware* em sistemas distribuídos e discute a variedade de serviços de *middleware* que podem ser fornecidos. (P.A. Bernstein, *Comm. ACM*, 39(2), fev. 1996.)

EXERCÍCIOS

11.1 Explique por que os sistemas distribuídos são inerentemente mais escalonáveis do que os sistemas centralizados. Quais são os prováveis limites na escalabilidade do sistema?

11.2 Qual é a diferença fundamental entre a abordagem cliente-gordo e cliente-magro para o desenvolvimento de sistemas cliente–servidor? Explique por que o uso de Java como uma linguagem de implementação reduz a distinção entre essas abordagens.

11.3 É proposto o desenvolvimento de um sistema para informações sobre bolsa de valores, em que os negociantes podem acessar informações sobre empresas e avaliar várias possibilidades de investimento, utilizando um sistema de simulação. Diferentes negociantes utilizam essa simulação de diferentes maneiras, de acordo com suas experiências e o tipo de ações com que estejam trabalhando. Sugira uma arquitetura cliente–servidor para esse sistema, que mostre onde está a funcionalidade. Justifique o modelo de sistema cliente–servidor que você escolheu.

11.4 Fazendo referência ao modelo de aplicação mostrado na Figura 11.5, discuta os problemas que podem surgir na conversão de um sistema de mainframe legado, da década de 80, que processa políticas de seguros para uma arquitetura cliente–servidor.

11.5 Os sistemas distribuídos com base em um modelo cliente–servidor têm sido desenvolvidos desde a década de 80, mas só recentemente as arquiteturas de sistema com base em objetos distribuídos foram implementadas. Proponha três razões pelas quais isso ocorreu.

11.6 Explique por que o uso de objetos distribuídos com um requisitor de objetos simplifica a implementação de sistemas cliente–servidor extensíveis. Justifique sua resposta com um exemplo.

11.7 Como é utilizada a IDL para CORBA, a fim de lidar com a comunicação entre objetos que foram implementados em diferentes linguagens de programação? Explique por que essa abordagem poderá causar problemas de desempenho, se houver diferenças radicais entre as linguagens utilizadas para a implementação de objetos.

11.8 Quais são os recursos básicos que devem ser fornecidos por um requisitor de objetos?

11.9 Podemos argumentar que desenvolver padrões CORBA para componentes horizontais e verticais é anticompetitivo. Se esses componentes forem desenvolvidos e adotados, isso inibirá o desenvolvimento de componentes melhores por empresas menores. Discuta o papel geral da padronização em apoiar e restringir a concorrência no mercado de software.

12 Projeto orientado a objetos

Objetivos

O objetivo deste capítulo é introduzir uma abordagem de projeto de software em que o projeto é estruturado como objetos que interagem entre si. Depois de ler este capítulo, você poderá:

- compreender como um projeto de software pode ser representado como um conjunto de objetos que interagem entre si, que gerenciam seus próprios estados e suas operações;
- conhecer as atividades mais importantes em um processo geral de projeto orientado a objetos;
- compreender os diferentes modelos que podem ser utilizados para documentar um projeto orientado a objetos, e
- conhecer a representação desses modelos em UML (linguagem de modelagem unificada).

Sumário

12.1 Objetos e classes de objetos
12.2 Processo de projeto orientado a objetos
12.3 Evolução de projeto

O projeto orientado a objetos é uma estratégia de projeto em que os projetistas de sistema pensam em termos de 'coisas', em vez de operações ou funções. O sistema em funcionamento é constituído de objetos que interagem entre si, que mantêm seu próprio estado local e fornecem operações com base nessas informações de estado (Figura 12.1). Eles ocultam informações sobre a representação do estado e, então, limitam o acesso a ele. Um processo de projeto orientado a objetos envolve projetar as classes de objetos e as relações entre essas classes. Quando o projeto é compreendido como um programa em execução, os objetos requeridos são criados dinamicamente, utilizando-se as definições de classe.

O projeto orientado a objetos é parte do *desenvolvimento orientado a objetos*, em que uma estratégia orientada a objetos é utilizada em todo o processo de desenvolvimento:

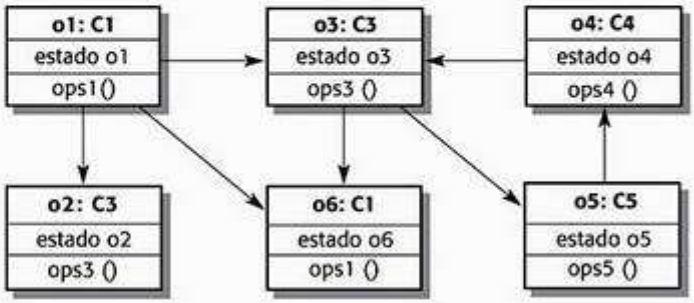
- *A análise orientada a objetos* se dedica a desenvolver um modelo orientado a objetos do domínio de aplicação. Os objetos identificados refletem entidades e operações que estão associadas com o problema a ser resolvido.
- *O projeto orientado a objetos* se dedica a desenvolver um modelo orientado a objetos de um sistema de software para implementar os requisitos identificados. Os objetos em um projeto orientado a objetos estão relacionados à solução do problema que está sendo resolvido. É possível que haja relacionamentos próximos entre alguns objetos do problema e alguns objetos da solução, mas o projetista, inevitavelmente, precisa adicionar novos objetos e transformar objetos do problema, a fim de implementar a solução.
- *A programação orientada a objetos* se ocupa de realizar um projeto de software utilizando uma linguagem de programação orientada a objetos, por exemplo, a linguagem Java, que aceita a implementação direta de objetos e fornece recursos para definir as classes de objetos.

A transição entre esses estágios de desenvolvimento deve ser contínua e direta, com a mesma notação utilizada em cada estágio. Mover para o próximo estágio envolve aprimorar o estágio anterior, adicionando detalhes às classes existentes de objetos e inventando novas classes, a fim de oferecer funcionalidade adicional. Embora as informações estejam ocultas dentro de objetos, as decisões detalhadas de projeto sobre a representação de dados podem ser adiadas, até que o sistema esteja implementado. Em alguns casos, as decisões sobre a distribuição de objetos e se os objetos podem ou não ser seqüenciais ou simultâneos também podem ser adiadas. Isso significa que os projetistas de software não estão limitados pelos detalhes da implementação do sistema. Eles podem imaginar projetos que podem ser adaptados a diferentes ambientes de execução.

Os sistemas orientados a objetos devem ser de fácil manutenção, uma vez que os objetos são independentes. Eles podem ser compreendidos e modificados como entidades *stand-alone*. A modificação da implementação de um objeto ou a adição de serviços não deve afetar outros objetos de sistemas. Devido ao fato de os objetos serem associados com coisas, existe sempre um nítido mapeamento entre as entidades do mundo real (como componentes de hardware) e seus objetos de controle no sistema. Isso melhora a facilidade de compreensão e, portanto, a facilidade de manutenção do projeto.

Figura 12.1 ▶

Um sistema constituído de objetos que interagem entre si.



Os objetos são componentes potencialmente reutilizáveis porque são encapsulamentos independentes de estado e de operações. Os projetos podem ser desenvolvidos com o uso de objetos que tenham sido criados em projetos precedentes. Essa alternativa reduz os custos de projeto, programação e validação e também pode levar ao uso de objetos-padrão (o que, conseqüentemente, aumenta a facilidade de compreensão do projeto). Ela também reduz os riscos envolvidos no desenvolvimento de software. Contudo, como será discutido no Capítulo 14, o reuso, às vezes, é mais bem implementado utilizando-se coleções de objetos (componentes ou estruturas), em vez de objetos individuais.

Vários métodos de projeto orientado a objetos foram propostos (Coad e Yourdon, 1990; Robinson, 1992; Jacobson *et al.*, 1993; Booch, 1994; Graham, 1994). Uma unificação das notações utilizadas nesses métodos foi definida (UML) juntamente com um processo de projeto associado (Rumbaugh *et al.*, 1999b). Não foi descrito neste texto nenhum método de projeto específico, mas os conceitos genéricos e as atividades de projeto estão descritos na Seção 12.2. A notação em UML é utilizada em todo o capítulo.

12.1 Objetos e classes de objetos

Hoje em dia, os termos 'objeto' e 'orientado a objetos' são amplamente utilizados e aplicados a diferentes tipos de entidades, métodos de projeto, sistemas e linguagens de programação. Contudo, existe uma aceitação geral de que um objeto é um encapsulamento de informações, e isso se reflete na definição de um objeto e de uma classe de objeto a seguir:

> Um objeto é uma entidade que possui um estado e um conjunto definido de operações que operam nesse estado. O estado é representado por um conjunto de atributos de objeto. As operações associadas com o objeto fornecem serviços para outros objetos (clientes), que requisitam esses serviços quando alguma computação é necessária.
>
> Os objetos são criados de acordo com uma definição de classe de objetos, que serve como um *template* para criar objetos. Essa classe apresenta declarações de todos os atributos e operações que devem ser associados a um objeto dessa classe.

A notação utilizada neste texto para as classes de objetos é a definida em UML. Uma classe de objetos é representada como um retângulo identificado com duas seções. Os atributos do objeto são relacionados na seção superior. As operações que estão associadas ao objeto são registradas na seção inferior. A Figura 12.2 ilustra essa notação utilizando uma classe de objetos que modela um funcionário em uma organização. Em UML, o termo 'operação' é a especificação de uma ação e o termo 'método' é utilizado para se referir à implementação da operação.

A classe Funcionário define uma série de atributos que mantêm informações sobre os funcionários, além dos seus nomes e endereços, do número da previdência social, do código de imposto, entre outros. As reticências (...) indicam que há mais atributos associados à classe, que não são mostrados aqui. As operações associadas ao objeto podem ser join (chamada quando um funcionário entra na organização), leave (chamada quando um funcionário deixa a organização), retire (chamada quando um funcionário se aposenta pela organização) e changeDetails (chamada quando alguma informação de um funcionário precisa ser modificada).

Os objetos se comunicam ao solicitar serviços (métodos de chamada) de outros objetos e, se necessário, trocar as informações requeridas pelo fornecimento de serviço. As cópias das informações necessárias para executar o serviço e os resultados da execução do serviço são fornecidos como parâmetros. Eis alguns exemplos desse estilo de comunicação:

Figura 12.2 ▶

Um objeto funcionário.

```
// Chamar um método associado com um objeto buffer que retorna o próximo valor
// no buffer
v = circular.Buffer.Get ();
// Chamar o método associado a um objeto de termostato que ajuste a
// temperatura a ser mantida
thermostat.setTemp (20);
```

Em alguns sistemas distribuídos, as comunicações com objetos são implementadas diretamente como mensagens de texto que os objetos trocam entre si. O objeto receptor analisa gramaticalmente a mensagem, identifica o serviço e os dados associados e realiza o serviço requisitado. Contudo, quando os objetos coexistem no mesmo programa, as chamadas de método são implementadas da mesma maneira que as chamadas de procedimento ou de função, em uma linguagem como C ou Ada.

Quando as requisições de serviços são implementadas dessa maneira, a comunicação entre objetos é síncrona, ou seja, o objeto que chama espera que a requisição de serviço seja concluída. Contudo, se os objetos forem implementados como processos simultâneos ou *threads*, a comunicação do objeto poderá ser assíncrona. O objeto que chama pode continuar em operação enquanto o serviço requisitado está sendo executado. Será explicado, ainda nesta seção, como os objetos podem ser implementados como processos simultâneos.

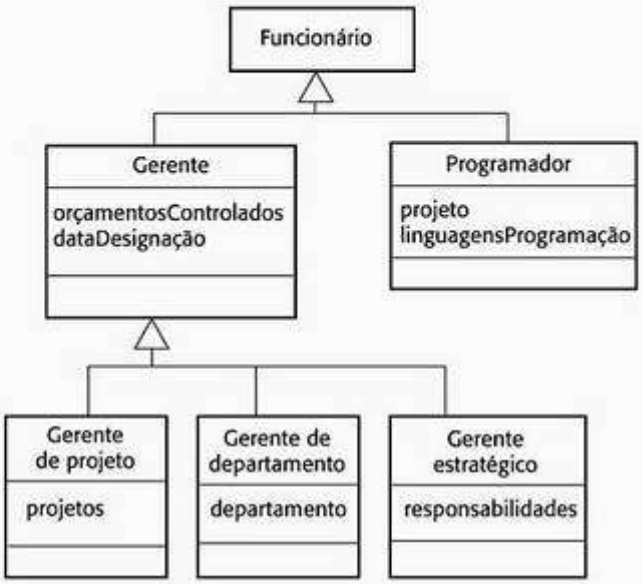
Como foi discutido no Capítulo 7, em que foi descrita uma série de possíveis modelos de objeto, as classes de objetos podem ser organizadas em uma hierarquia de generalização ou de herança, que mostra o relacionamento entre as classes de objeto gerais ou mais específicas. A classe de objeto mais específica é completamente consistente com a classe de objeto geral, mas inclui outras informações. Em UML, a generalização é indicada por uma seta que aponta para a classe-mãe. Em linguagens de programação orientadas a objetos, a generalização é em geral implementada com o uso do mecanismo de herança. A classe-filha herda atributos e operações da classe-mãe.

A Figura 12.3 mostra um exemplo dessa hierarquia, em que são mostradas diferentes classes de funcionários. As classes inferiores da hierarquia têm os mesmos atributos e as operações que suas classes-mãe, mas podem adicionar novos atributos e operações ou modificar alguns desses atributos a partir de suas classes-mãe. Isso significa que existe a possibilidade de intercâmbio em uma direção. Se o nome de uma classe-mãe é utilizado em um modelo, isso significa que o objeto no sistema pode ser definido como pertencente àquela classe ou a qualquer um de seus descendentes.

A classe Gerente, na Figura 12.3, tem todos os atributos e operações da classe Funcionário, além de dois novos atributos que registram os orçamentos controlados pelo gerente e a data

Figura 12.3 ▶

Uma hierarquia de generalização.



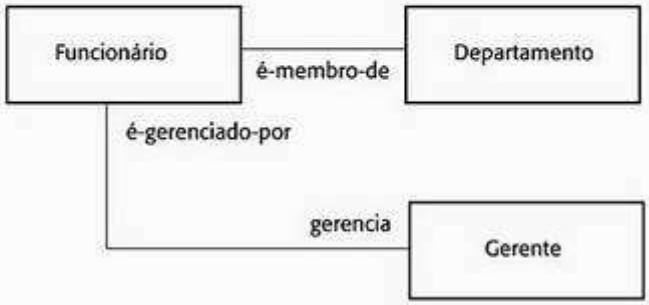
em que ele foi designado para uma função específica de gerência. Da mesma maneira, a classe Programador adiciona novos atributos que definem o projeto em que o programador está trabalhando e as habilidades da linguagem de programação que ele tem. Os objetos da classe Gerente ou Programador podem, portanto, ser utilizados em qualquer lugar em que um objeto da classe Funcionário for requisitado.

Os objetos que são membros de uma classe de objetos participam nos relacionamentos com outros objetos. Esses relacionamentos podem ser modelados descrevendo-se as associações entre as classes de objetos. Em UML, as associações são denotadas por uma linha entre as classes de objeto, que podem, opcionalmente, ser anotadas com informações sobre a associação. Essa condição está ilustrada na Figura 12.4, que mostra a associação entre objetos da classe Funcionário e objetos da classe Departamento e entre objetos da classe Funcionário e objetos da classe Gerente.

A associação é um relacionamento muito geral e, freqüentemente, é utilizada em UML para indicar que o atributo de um objeto é um objeto associado ou que a implementação de um método de objeto conta com o objeto associado. Contudo, em princípio, pelo menos, qualquer tipo de associação é possível. Uma das associações mais comuns é a agregação, que ilustra como objetos podem ser compostos de outros objetos. Veja o Capítulo 7 para saber mais sobre esse tipo de associação.

Figura 12.4 ▶

Um modelo de associação.



12.1.1 Objetos concorrentes

Conceitualmente, um objeto requisita um serviço de outro objeto, enviando uma mensagem de 'requisição de serviço' para aquele objeto. Não há nenhum requisito para a execução serial, em que um objeto aguarda pelo término de um serviço requisitado. Como conseqüência, o modelo geral de interação de objetos permite que os objetos sejam executados simultaneamente como processos paralelos. Esses objetos podem ser executados no mesmo computador ou, como objetos distribuídos, em diferentes máquinas.

Na prática, a maioria das linguagens de programação orientadas a objetos tem como *default* um modelo de execução serial em que as requisições de objetos de serviço são implementadas do mesmo modo que as chamadas de função. Portanto, quando um objeto chamado theList é criado a partir de uma classe normal de objeto, escrevemos em Java:

```
theList.append (17)
```

Essa notação chama o método append associado ao objeto theList para acrescentar o elemento 17 a theList e a execução do objeto que chama é suspensa, até que a operação append seja concluída. Contudo, Java inclui um mecanismo muito simples (*thread*), que permite criar objetos que são executados simultaneamente. Portanto, é fácil considerar um projeto orientado a objetos e produzir uma implementação em que os objetos sejam processos simultâneos.

Existem dois tipos de implementação de objetos simultâneos:

1. *Servidores*, em que o objeto é implementado como um processo paralelo, com métodos correspondentes às operações definidas de objeto. Os métodos iniciam-se em resposta a uma mensagem externa e podem ser executados em paralelo com métodos associados a outros objetos. Quando eles completam sua operação, o objeto interrompe sua execução e aguarda por outras requisições de serviço.
2. *Objetos ativos*, em que o estado do objeto pode ser modificado por operações internas em execução dentro do próprio objeto. O processo que representa o objeto executa continuamente essas operações e, assim, nunca interrompe a si próprio.

Os servidores são muito úteis em um ambiente distribuído, em que o objeto que chama e o objeto chamado são executados em diferentes computadores. O tempo de resposta para o serviço requisitado é imprevisível e, assim, sempre que possível, é preciso projetar o sistema de modo que o objeto que tenha requisitado um serviço não necessite aguardar a conclusão do serviço. Eles podem também ser utilizados em uma única máquina, quando um serviço leva algum tempo para ser concluído (por exemplo, a impressão de um documento), e o serviço pode ser requisitado por vários objetos diferentes.

Os objetos ativos são utilizados quando um objeto precisa atualizar seu próprio estado em intervalos especificados. Isso é comum em sistemas em tempo real, em que os objetos são associados a dispositivos de hardware, que coletam informações sobre o ambiente do sistema. Os métodos de objetos permitem o acesso de outros objetos a suas informações de estado.

A Figura 12.5 mostra como um objeto ativo pode ser definido e implementado em Java. Essa classe de objetos representa o *transponder* de uma aeronave. O *transponder* mantém o controle da posição da aeronave utilizando um sistema de navegação por satélite. Ele pode responder a mensagens de computadores de controle de tráfego aéreo. Ele fornece a posição atual da aeronave em resposta a uma requisição do método givePosition.

Esse objeto é implementado como um *thread*, no qual um loop contínuo no método run traz o código para calcular a posição da aeronave, utilizando sinais de satélite. Os *threads* são criados em Java, utilizando a classe Thread inserida como uma classe-mãe em uma declaração de classe. Eles devem incluir um método chamado run, que é iniciado pelo sistema *run-time* Java, quando os objetos definidos como *threads* são criados. Esse exemplo é mostrado na Figura 12.5.

Figura 12.5 ▶

Implementação de um objeto ativo com utilização de threads em Java.

```
class Transponder extends Thread {
        Position currentPosition ;
        Coords c1, c2 ;
        Satellite sat1, sat2 ;
        Navigator theNavigator ;
        public Position givePosition ()
        {
            return currentPosition ;
        }
        public void run ()
        {
            while (true)
            {
                c1 = sat1.position () ;
                c2 = sat2.position () ;
                currentPosition = theNavigator.compute (c1, c2) ;
            }
        }
} //Transponder
```

12.2 Processo de projeto orientado a objetos

Nesta seção, ilustro o processo de projeto orientado a objetos, ao desenvolver um projeto para o software de controle, que é embutido em uma estação meteorológica automatizada. Como foi discutido na introdução, há vários métodos de projeto orientado a objetos, sem nenhum processo ou método de projeto definitivo que possa ser considerado o 'melhor'. O processo abordado neste texto é um processo geral, que incorpora atividades comuns à maioria dos processos de OOD (*object-oriented design* – projeto orientado a objetos). A esse respeito, ele é comparável ao processo de UML proposto (Rumbaugh *et al.*, 1999b), mas foi simplificado significativamente para ser apresentado aqui.

O processo geral utilizado neste texto para o projeto orientado a objetos engloba uma série de estágios:

1. Compreender e definir o contexto e os modos de utilização do sistema.
2. Projetar a arquitetura do sistema.
3. Identificar os principais objetos do sistema.
4. Desenvolver os modelos de projeto.
5. Especificar as interfaces dos objetos.

Deliberadamente, isso não foi ilustrado como um simples diagrama de processo, uma vez que isso implicaria uma nítida seqüência de atividades nesse processo. Na verdade, todas as atividades citadas podem ser consideradas atividades intercaladas que influenciam umas às outras. Os objetos são identificados e as interfaces, especificadas inteira ou parcialmente, enquanto a arquitetura do sistema é definida. À medida que os objetos são produzidos, essas definições individuais de objetos podem ser aprimoradas e isso pode significar mudanças na arquitetura do sistema.

Esse método será discutido, ainda nesta seção, como estágios separados no processo de projeto. Contudo, não se deve considerar, a partir disso, que o projeto seja um processo simples, bem-estruturado. Na realidade, desenvolve-se um projeto ao propor soluções e aperfeiçoá-las, à medida que as informações se tornam disponíveis. No entanto, quando surgem problemas, é preciso retomar o processo e tentar novamente. Algumas vezes, certas opções

são exploradas detalhadamente, para verificar se funcionam; outras vezes, os detalhes são ignorados até a fase posterior do processo.

Essas atividades de processo são ilustradas com o exemplo de um projeto orientado a objetos. Esse exemplo é parte de um sistema destinado a criar mapas meteorológicos, com o emprego de dados meteorológicos coletados automaticamente. Os requisitos detalhados para esse sistema de mapeamento climático ocupariam muitas páginas. Contudo, uma arquitetura geral de sistema poderá ser desenvolvida a partir de uma descrição de sistema relativamente simples:

> Um sistema de mapeamento metereológico é necessário para gerar mapas meteorológicos regularmente, utilizando dados coletados a partir de estações meteorológicas remotas, sem que seus funcionários estejam presentes, e de outras fontes de dados, como observadores de tempo, balões e satélites meteorológicos. As estações meteorológicas transmitem seus dados ao computador da área em resposta a uma requisição dessa máquina.
>
> O sistema de computador da área valida os dados coletados e faz a integração dos dados a partir de diferentes fontes. Os dados integrados são arquivados e, com os dados desse arquivo e um banco de dados de mapas digitalizados, é criado um conjunto de mapas meteorológicos locais. Os mapas podem ser impressos para distribuição em uma impressora especial ou ser exibidos em diversos formatos.

Essa descrição mostra que parte do sistema geral se ocupa da coleta de dados, outra parte, da integração de dados a partir de diferentes fontes, parte, do arquivamento desses dados, e parte, da criação de mapas meteorológicos. A Figura 12.6 ilustra uma possível arquitetura de sistema, que pode ser derivada a partir dessa descrição. Trata-se de uma arquitetura em camadas (discutida no Capítulo 10), que reflete os diferentes estágios de processamento do sistema, como a coleta de dados, a integração de dados, o arquivamento de dados e a criação de mapas. Uma arquitetura em camadas é apropriada nesse caso porque cada estágio depende apenas do processamento do estágio precedente para sua operação.

Na Figura 12.6, foram mostradas as diferentes camadas, inclusive o nome da camada em um símbolo de *package* UML, que foi denotada como um subsistema. Um *package* UML representa uma coleção de objetos e outros *packages*. Essa opção é utilizada para mostrar que cada camada inclui uma série de outros componentes.

Na Figura 12.7, esse modelo abstrato de arquitetura foi ampliado ao mostrar os componentes dos subsistemas. Novamente, os componentes são muito abstratos e foram derivados das informações contidas na descrição do sistema. Dou seqüência ao exemplo de projeto ao enfocar o subsistema da estação meteorológica, que é parte da camada de coleta de dados.

Figura 12.6 ▶

Arquitetura em camadas para um sistema de mapeamento meteorológico.

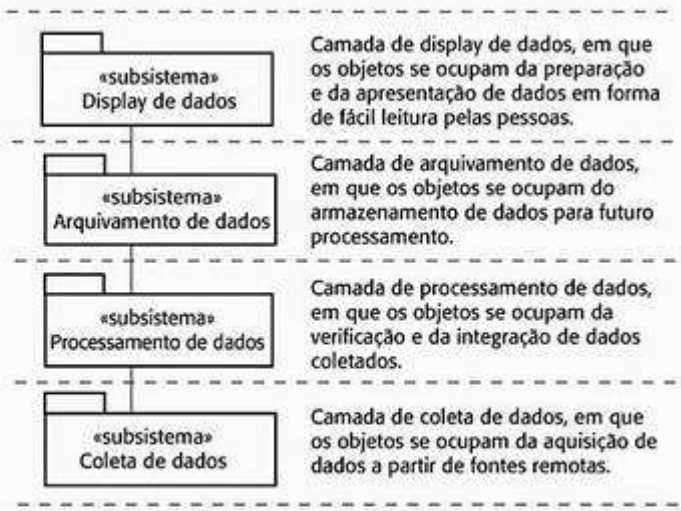
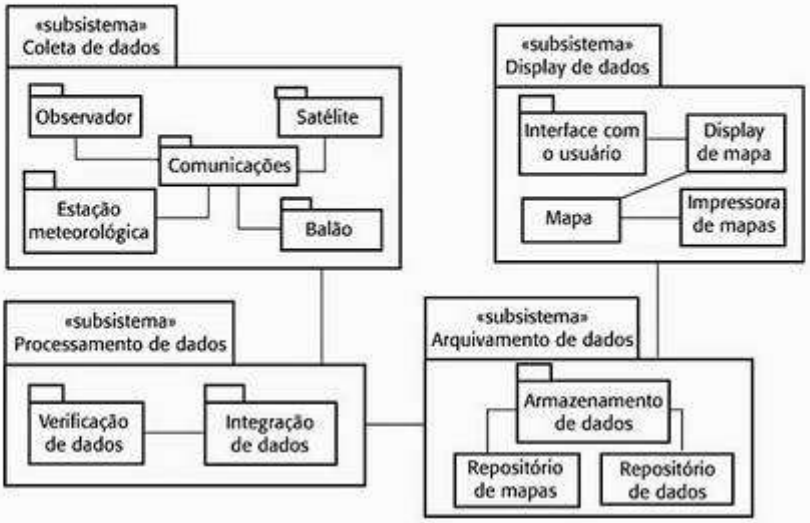


Figura 12.7 ▶

Subsistemas em um sistema de mapeamento meteorológico.



12.2.1 Contexto do sistema e modelos de uso

O primeiro estágio de qualquer processo de projeto de software consiste em desenvolver uma compreensão das relações entre o software que está sendo projetado e seu ambiente externo. Desenvolver essa compreensão ajuda a decidir como fornecer a funcionalidade necessária para o sistema e como estruturar o sistema de modo que ele possa se comunicar eficazmente com seu ambiente.

O contexto e o modelo de uso do sistema representam dois modelos complementares das relações entre o sistema e seu ambiente:

1. O contexto do sistema é um modelo estático, que descreve os outros sistemas naquele ambiente.
2. O modelo de uso do sistema é um modelo dinâmico, que descreve como o sistema realmente interage com seu ambiente.

O modelo de contexto de um sistema pode ser representado com o uso de associações (veja a Figura 12.4), em que, essencialmente, é produzido um simples diagrama de blocos da arquitetura do sistema geral. Esse diagrama pode ser expandido pela representação de um modelo de subsistema, utilizando-se *packages* UML, como mostra a Figura 12.7. Essa figura ilustra que o contexto do sistema da estação meteorológica está dentro de um subsistema, que se ocupa da coleta de dados, além de outros subsistemas que constituem o sistema de mapeamento meteorológico.

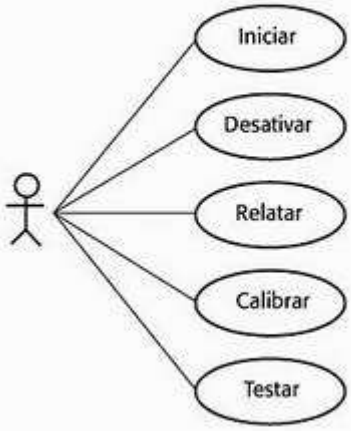
Quando se modelam as interações de um sistema com seu ambiente, deve-se utilizar uma abordagem abstrata, que não inclua muitos detalhes dessas interações. A abordagem proposta em UML é desenvolver um modelo de use-case, em que cada um deles represente uma interação com o sistema. Em modelos de use-case (também discutido no Capítulo 6), cada interação possível tem seu nome em uma elipse e a entidade externa envolvida na interação é representada pela figura de um boneco. No caso do sistema de estação meteorológica, essa entidade externa é o sistema de processamento de dados para os dados sobre o clima.

Um modelo de use-case para a estação meteorológica é descrito na Figura 12.8, que mostra que a estação meteorológica interage com entidades externas para iniciar e desativar, relatar os dados climáticos que foram coletados e testar e calibrar instrumentos.

Cada um desses use-cases pode ser descrito por meio de uma descrição simples em linguagem natural. Isso ajuda os projetistas a identificar objetos no sistema e a compreender o que o sistema deve fazer. É utilizada uma forma estilizada dessa descrição que identifica claramente quais informações são trocadas, como a interação é iniciada etc. Esse exemplo é mostrado na Figura 12.9, em que é descrito o use-case Relatar da Figura 12.8.

Figura 12.8

Use-cases para a estação meteorológica.



Naturalmente, é possível utilizar qualquer técnica para descrever os use-cases, desde que a descrição seja breve e facilmente compreensível. É preciso desenvolver descrições para cada um dos use-cases mostrados no modelo.

A descrição de use-case ajuda a identificar objetos e operações no sistema. A partir da descrição do use-case Relatar, é óbvio que serão necessários objetos que representem instrumentos que coletam os dados sobre o clima e também um objeto que represente o resumo desses dados. Também são necessárias as operações para requisitar dados sobre o clima e para enviar esses dados.

12.2.2 Projeto de arquitetura

Uma vez que as interações entre o sistema de software que está sendo projetado e o ambiente do sistema tenham sido definidas, é possível utilizar essas informações como base para projetar a arquitetura do sistema. É claro que é preciso combinar essas informações com o conhe-

Figura 12.9

Descrição do use-case Relatar.

Sistema	Estação meteorológica.
Use-case	Relatar.
Agentes	Sistema de coleta de dados sobre o clima, Estação meteorológica.
Dados	A estação meteorológica envia para o sistema de coleta de dados climáticos um resumo dos dados sobre o clima, que foram coletados a partir de instrumentos, no período de coleta. Os dados enviados referem-se às temperaturas máximas, mínimas e médias do solo e do ar; à pressão máxima, mínima e média do ar; às velocidades máxima, mínima e média do vento; à precipitação total das chuvas, e à direção do vento, conforme a amostragem a cada intervalo de cinco minutos.
Estímulo	O sistema de coleta de dados sobre o clima estabelece um link de modem com a estação meteorológica e requisita a transmissão dos dados.
Resposta	Os dados resumidos são enviados para o sistema de coleta de dados sobre o clima.
Comentários	Em geral, as estações meteorológicas recebem um pedido de relatório por hora, mas essa freqüência pode diferir de uma estação para outra e ser modificada no futuro.

cimento geral dos princípios do projeto de arquitetura e com o conhecimento mais detalhado do domínio.

A estação meteorológica automatizada é um sistema relativamente simples e sua arquitetura pode também ser representada como um modelo em camadas. Esse exemplo está ilustrado na Figura 12.10 como três *packages* UML dentro do package mais geral, denominado Estação meteorológica. Observe como foram utilizadas anotações em UML (texto em caixas com cantos dobrados) para fornecer mais informações.

As três camadas no software da estação meteorológica são:

1. a camada de interface, que se ocupa de todas as comunicações com outras partes do sistema e com o fornecimento das interfaces externas do sistema;
2. a camada de coleta de dados, que se ocupa de gerenciar a coleta de dados a partir dos instrumentos e de resumir os dados climáticos antes da transmissão para o sistema de mapeamento, e
3. a camada dos instrumentos, que é um encapsulamento de todos os instrumentos utilizados para coletar os dados brutos sobre as condições do tempo.

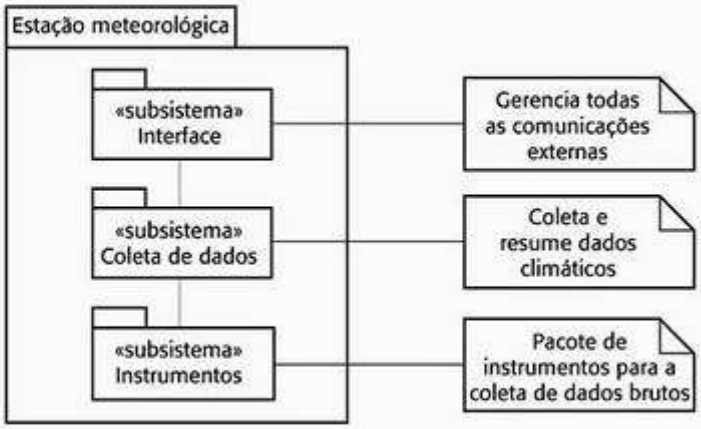
Em geral, deve-se tentar decompor o sistema, de modo que as arquiteturas sejam o mais simples possível. Uma boa regra prática é a de que não deve haver mais do que sete entidades fundamentais incluídas em um modelo de arquitetura. Cada uma dessas entidades pode ser descrita separadamente, mas, com certeza, pode-se escolher revelar a estrutura das entidades, como foi feito na Figura 12.7.

12.2.3 Identificação de objetos

Nesse estágio do processo de projeto, já deveriam ter sido formuladas algumas idéias sobre os objetos essenciais no sistema que está sendo projetado. No sistema de estação meteorológica, está claro que os instrumentos são objetos e é preciso pelo menos um objeto em cada um dos níveis de arquitetura. Isso reflete o princípio geral de que os objetos tendem a emergir durante o processo de projeto. Contudo, de modo geral, também é necessário procurar e documentar outros objetos que possam ser relevantes.

Embora o título desta seção seja identificação de *objetos*, na prática, esse processo, se ocupa de identificar as *classes de objetos*. O projeto é descrito em termos dessas classes. É inevitável a necessidade de refinar as classes de objetos que inicialmente foram identificadas e repetir novamente esse estágio do processo, enquanto se desenvolve uma compreensão mais profunda do projeto.

Figura 12.10 ▶
Arquitetura da estação meteorológica.



Existem várias propostas sobre como identificar as classes de objetos:

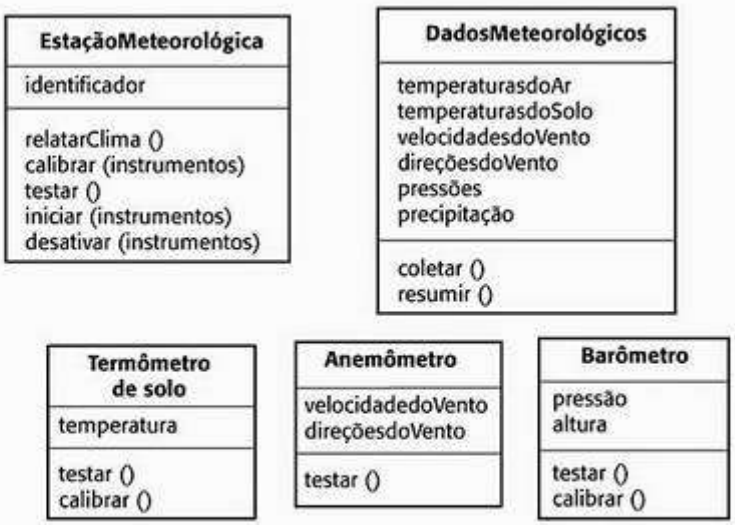
1. Utilize a análise gramatical de uma descrição em linguagem natural de um sistema. Objetos e atributos são nomes e operações e serviços são verbos (Abbott, 1983). Essa abordagem foi incorporada ao método HOOD para o projeto orientado a objetos (Robinson, 1992), que foi amplamente utilizado na indústria aeroespacial européia.
2. Utilize entidades tangíveis (coisas) no domínio da aplicação, como aeronaves; funções, como gerente; eventos, como solicitações; interações, como reuniões; locais, como escritórios; unidades organizacionais, como empresas, e assim por diante. (Shlaer e Mellor, 1988; Coad e Yourdon, 1990; Wirfs-Block *et al.*, 1990). Considere isso identificando estruturas de repositório de dados (estruturas de dados abstratos) no domínio da solução, que possam ser requeridas para lidar com esses objetos.
3. Utilize uma abordagem comportamental em que o projetista primeiramente compreenda o comportamento geral do sistema. Os vários comportamentos são atribuídos a diferentes partes do sistema e uma compreensão é derivada de quem inicia e participa desses comportamentos. Os participantes que desempenham papéis significativos são reconhecidos como objetos (Rubin e Goldberg, 1992).
4. Utilize uma análise baseada em cenários, em que vários cenários de uso do sistema são identificados e analisados de cada vez. À medida que cada cenário é analisado, a equipe responsável pela análise deve identificar objetos, atributos e operações que são requeridos. Um método de análise, chamado de cartões CRC, em que analistas e projetistas assumem o papel de objetos, é eficaz no apoio a essa abordagem baseada em cenários (Beck e Cunningham, 1989).

Essas abordagens nos ajudam a iniciar a identificação de objetos. Na prática, muitas diferentes fontes de conhecimento têm de ser utilizadas para descobrir objetos e classes de objetos. Objetos e operações que são inicialmente identificados a partir da descrição informal do sistema podem ser um ponto de partida para o projeto. Outras informações, obtidas a partir do conhecimento do domínio de aplicação ou da análise de cenário, podem, então, ser utilizadas para refinar e ampliar os objetos iniciais. Essas informações podem ser coletadas em documentos de requisitos, em discussões com usuários e em uma análise dos sistemas existentes.

Foi utilizada uma abordagem híbrida, nesse caso, para identificar os objetos da estação meteorológica. Não há espaço para descrever todos os objetos, mas são apresentadas cinco classes de objetos na Figura 12.11. O Termômetro de solo, o Anemômetro e o Barômetro representam objetos de domínio da aplicação e os objetos EstaçãoMeteorológica e DadosMeteorológicos foram identificados na descrição do sistema e na descrição do cenário (use-case).

Figura 12.11 ▶

Exemplos de classes de objetos no sistema da estação meteorológica.



Esses objetos estão relacionados a diferentes níveis na arquitetura do sistema:

1. A classe de objetos denominada EstaçãoMeteorológica fornece a interface básica da estação meteorológica com seu ambiente. Portanto, suas operações refletem as interações mostradas na Figura 12.8. Nesse caso, é utilizada uma única classe de objetos para encapsular todas essas interações, mas em outros projetos pode ser mais apropriado utilizar várias classes para fornecer a interface com o sistema.

2. A classe de objeto denominada DadosMeteorológicos encapsula os dados resumidos dos diferentes instrumentos na estação meteorológica. Suas operações associadas se ocupam de coletar e resumir os dados que são requeridos.

3. As classes de objeto TermômetrodeSolo, Anemômetro e Barômetro são diretamente relacionadas a instrumentos no sistema. Elas refletem entidades tangíveis de hardware no sistema e as operações se ocupam de controlar esse hardware.

Nesse estágio do processo de projeto, o conhecimento do domínio da aplicação pode ser utilizado para identificar outros objetos e serviços. Nesse caso, sabemos que as estações meteorológicas são geralmente localizadas em lugares remotos. Elas contêm vários instrumentos que, às vezes, apresentam defeitos. As falhas de instrumentos devem ser relatadas automaticamente, o que implica a necessidade de atributos e operações para verificar o funcionamento correto dos instrumentos. Existem com certeza muitas estações meteorológicas remotas. Por conseguinte, é preciso alguma maneira de identificar os dados coletados de cada estação, de modo que cada uma delas deve ter seu próprio identificador.

Decidi não tornar os objetos associados a cada objeto ativo do instrumento. A operação coletar em DadosMeteorológicos chama objetos de instrumento para fazer leituras, quando necessário. Os objetos ativos trazem seus próprios controles e, nesse caso, isso quer dizer que o instrumento decide quando fazer as leituras. Contudo, a desvantagem dessa alternativa é que, se uma decisão foi tomada para alterar a freqüência da coleta de dados ou se diferentes estações coletam dados de modo diferente, novas classes de objetos teriam de ser introduzidas. Ao fazer com que os objetos de instrumentos realizem as leituras quando necessário, quaisquer mudanças na estratégia de coleta podem ser facilmente implementadas, sem modificar os objetos associados aos instrumentos.

12.2.4 Modelos de projeto

Os modelos de projeto mostram os objetos ou as classes de objeto em um sistema e, quando apropriado, os diferentes tipos de relações entre essas entidades. Eles são essencialmente o próprio projeto. Eles são a ponte entre os requisitos para o sistema e a implementação do sistema. Isso significa que existem requisitos conflitantes nesses modelos. Eles têm de ser abstratos, de modo que detalhes desnecessários não ocultem uma relação entre eles e os requisitos de sistema. Contudo, eles também devem incluir detalhes suficientes para que os programadores possam tomar decisões de implementação.

Em geral, esse conflito é contornado pelo desenvolvimento em diferentes modelos, com diferentes níveis de detalhes. Quando houver vínculos diretos entre os engenheiros de requisitos, os projetistas e os programadores, apenas os modelos abstratos poderão ser exigidos. As decisões específicas de projeto podem ser feitas à medida que o sistema é implementado. Quando os vínculos entre os especificadores de sistemas, os projetistas e os programadores forem indiretos (ou seja, quando um sistema estiver sendo projetado em uma parte da organização, mas tiver de ser implementado em outro lugar), mais modelos detalhados poderão ser requeridos.

Uma importante etapa no processo de projeto, portanto, é decidir quais modelos de projeto são necessários e o nível de detalhes desses modelos. Essa decisão também depende do tipo de sistema que está sendo desenvolvido. Um sistema de processamento seqüencial de dados será projetado de modo diferente de um sistema em tempo real embutido e, portanto, diferentes modelos de projeto serão utilizados. Existem muito poucos sistemas em que todos os modelos são necessários, e minimizar o número de modelos produzidos reduz os custos de projeto e o tempo necessário para completar o processo de projeto.

Há dois tipos de modelos de projeto que, normalmente, devem ser produzidos para descrever um projeto orientado a objetos:

1. Os modelos estáticos, que descrevem a estrutura estática do sistema em termos das classes de objetos de sistema e de seus relacionamentos. Relacionamentos importantes que podem ser documentados nesse estágio são os relacionamentos de generalização, relacionamentos utiliza/é-utilizado-por e relacionamentos de composição.
2. Os modelos dinâmicos, que descrevem a estrutura dinâmica do sistema e mostram as interações entre os objetos do sistema (e não as classes de objetos). Dentre as interações que podem ser documentadas destacam-se a seqüência de requisições de serviços feitas por objetos e o modo como o estado do sistema é relacionado a essas interações de objetos.

A UML fornece um grande número de possíveis modelos estáticos e dinâmicos, que podem ser produzidos para documentar um projeto. Booch *et al.* (1999) propõem nove tipos diferentes de diagramas para representar esses modelos. Aqui, não tenho espaço para tratar de todos eles nem todos são apropriados para o exemplo da estação meteorológica. Os modelos discutidos nesta seção são:

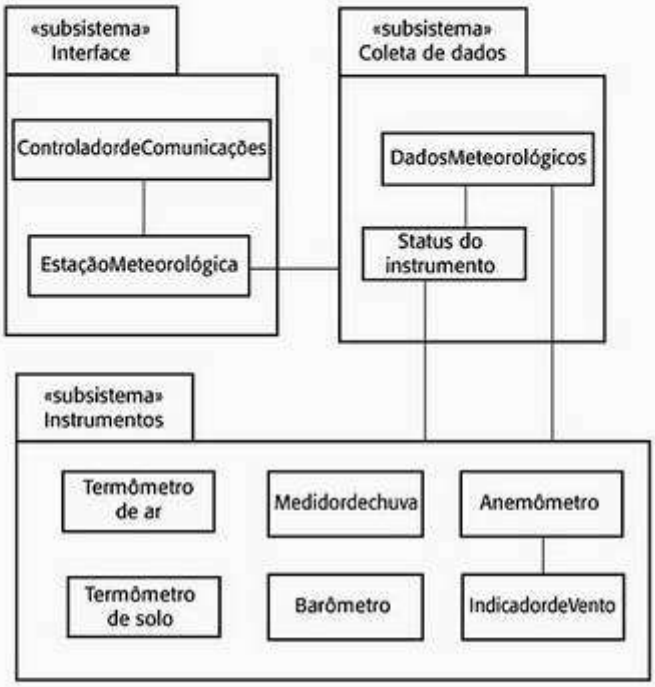
1. Modelos de subsistema, que mostram agrupamentos lógicos de objetos em subsistemas coerentes. Eles são representados com o uso de uma forma de diagrama de classes, em que cada subsistema é mostrado como um pacote. Os modelos de subsistema são modelos estáticos.
2. Modelos de seqüência, que mostram a seqüência das interações entre objetos. Eles são representados com o uso de um diagrama de seqüência de colaboração em UML. Os modelos de seqüência são modelos dinâmicos.
3. Modelos de máquina de estado, que mostram como objetos individuais modificam seu estado em resposta aos eventos. Eles são representados em UML com o uso de diagramas *statechart*. Os modelos de máquina de estado são modelos dinâmicos.

Já foram discutidos anteriormente, neste e em outros capítulos, outros tipos de modelos que podem ser desenvolvidos. Os modelos de use-case mostram interações com o sistema (Figura 12.8 e figuras 6.12 e 6.13), os modelos de objeto descrevem as classes de objetos (Figura 12.2), os modelos de generalização e de herança mostram como classes podem ser generalizações de outras classes (figuras 7.10, 7.11 e 7.12) e os modelos de agregação mostram como coleções de outros objetos estão relacionadas (Figura 7.13).

Na minha opinião, um modelo de subsistema é um dos modelos estáticos mais úteis, uma vez que mostra como o projeto pode ser organizado em grupos de objeto relacionados logicamente. Já vimos exemplos desse tipo de modelo na Figura 12.7, que mostra os subsistemas no sistema de mapeamento meteorológico. Em UML, os pacotes são construções encapsuladas e não refletem diretamente em entidades no sistema que é desenvolvido. Contudo, eles podem ser refletidos em construções estruturais, como bibliotecas Java.

A Figura 12.12 mostra os objetos nos subsistemas da estação meteorológica. Também apresento algumas associações nesse modelo. Por exemplo, o objeto ControladordeComunicações é associado com o objeto EstaçãoMeteorológica e o objeto EstaçãoMeteorológica, com o pacote ColetadeDados. Isso significa que esse objeto é associado a um ou mais objetos nesse pacote. Um modelo de *package* e um modelo de classe de objeto devem descrever os agrupamentos lógicos no sistema.

Figura 12.12 ▶
Packages da estação meteorológica.



Um dos modelos dinâmicos mais úteis e fáceis de compreender que podem ser produzidos é o modelo de seqüência, que documenta, para cada modo de interação, a seqüência de interações de objetos que acontecem. Em um modelo de seqüência:

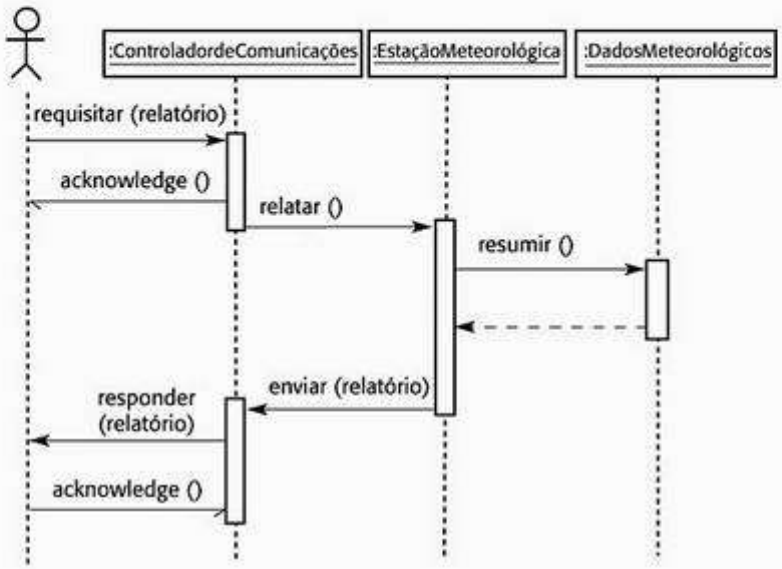
1. Os objetos envolvidos na interação são organizados horizontalmente, com uma linha vertical ligada a cada objeto.
2. O tempo é representado verticalmente, de modo que progrida para baixo nas linhas verticais. Portanto, a seqüência de operações pode ser lida facilmente no modelo.
3. As interações entre objetos são representadas por setas rotuladas que ligam as linhas verticais. Elas *não* são fluxos de dados, mas representam mensagens ou eventos, que são fundamentais para a interação.
4. O retângulo estreito, na linha da vida do objeto, representa o tempo pelo qual o objeto é o objeto controlador no sistema. Um objeto assume o controle na parte superior desse retângulo e o transfere para outro objeto na base do retângulo. Se houver uma hierarquia de chamadas, o controle não será cedido até que o último retorno à chamada inicial de método tenha sido completado.

Esse exemplo está ilustrado na Figura 12.13, que mostra a seqüência de interações quando o sistema de mapeamento externo requisita os dados da estação meteorológica. Esse diagrama pode ser lido como se segue:

1. Um objeto que é uma instância do ControladordeComunicações (:ControladordeComunicações) recebe um pedido de seu ambiente para enviar um relatório meteorológico. Ele acusa o recebimento do pedido. A meia ponta da seta indica que o remetente da mensagem não espera uma resposta.
2. Esse objeto envia uma mensagem a um objeto que é uma instância de EstaçãoMeteorológica para criar um relatório do clima. A instância de ControladordeComunicações, então, interrompe a si mesmo (seu bloco de controle termina). O estilo de ponta de seta utilizado indica que a instân-

Figura 12.13

Seqüência de operações – coleta de dados.



cia do objeto ControladordeComunicações e a instância do objeto EstaçãoMeteorológica são objetos que podem ser executados concorrentemente.

3. O objeto que é uma instância de EstaçãoMeteorológica envia uma mensagem para um objeto DadosMeteorológicos, que resume os dados sobre o clima. Nesse caso, o estilo diferente de ponta de seta utilizado indica que a instância de EstaçãoMeteorológica aguarda uma resposta.

4. Esse resumo é computado e o controle retorna para o objeto EstaçãoMeteorológica. A seta tracejada indica um retorno de controle.

5. Esse objeto envia uma mensagem para ControladordeComunicações solicitando que ele transfira os dados para o sistema remoto. O objeto EstaçãoMeteorológica, então, interrompe a si mesmo.

6. O objeto ControladordeComunicações envia os dados resumidos para o sistema remoto, recebe a confirmação (*acknowledgement*) e, então, interrompe a si mesmo, aguardando pelo próximo pedido.

A partir do diagrama de seqüência, é possível ver que o objeto ControladordeComunicações e o objeto EstaçãoMeteorológica são, na realidade, processos concorrentes, em que a execução pode ser interrompida e retomada. Essencialmente, a instância do objeto ControladordeComunicações escuta mensagens do sistema externo, decodifica essas mensagens e inicia as operações da estação meteorológica.

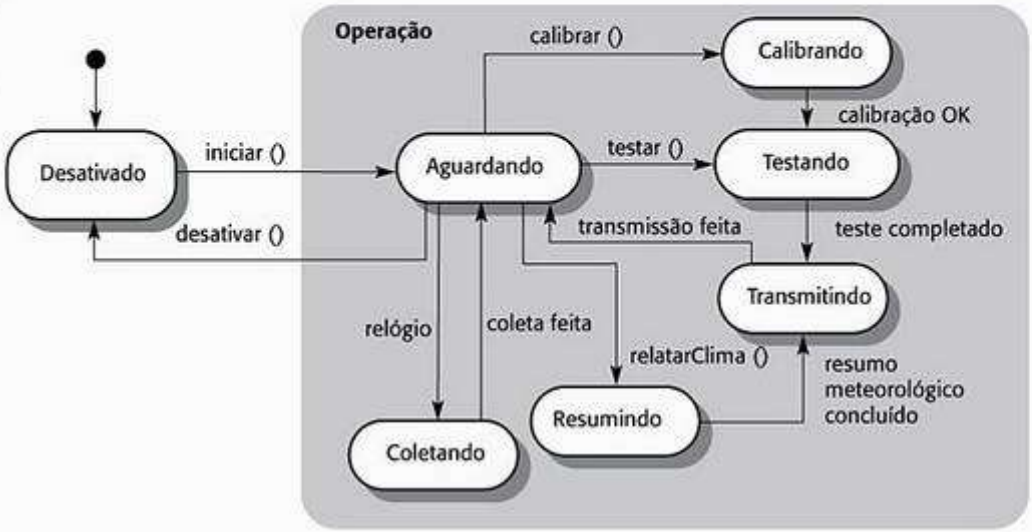
Ao documentar um projeto, é preciso produzir um diagrama de seqüência para cada interação significativa. Se um modelo de use-case foi desenvolvido, então deve haver um diagrama de seqüência para cada use-case identificado.

Os diagramas de seqüência são utilizados para modelar o comportamento combinado de um grupo de objetos, mas também pode-se desejar resumir o comportamento de um único objeto em resposta a diferentes mensagens que ele pode processar. Para fazer isso, é possível utilizar um modelo de máquina de estado, que mostra como a instância do objeto muda de estado, dependendo das mensagens que ele recebe. A UML utiliza *statecharts*, inicialmente inventados por Harel (1987), para descrever modelos de máquina de estado.

A Figura 12.14 é um *statechart* para o objeto EstaçãoMeteorológica, que mostra como ele responde a pedidos de vários serviços.

Figura 12.14

Diagrama de estado da EstaçãoMeteorológica



Podemos ler esse diagrama como se segue:

1. Se o estado do objeto é 'Desativado', ele só pode responder a uma mensagem iniciar (). Então, ele passa para um estado em que está aguardando por outras mensagens. A seta sem rótulo com o ponto preto indica que o estado 'Desativado' é o estado inicial.
2. No estado 'Aguardando', o sistema espera outras mensagens. Se uma mensagem Desativar () for recebida, o objeto retorna ao estado desativado.
3. Se uma mensagem relatarClima () é recebida, o sistema passa para um estado em que está resumindo e, então, quando o resumo é concluído, passa para o estado transmitindo, no qual as informações são transmitidas por meio do ControladordeComunicações. Ele, então, volta para o estado aguardando.
4. Se uma mensagem calibrar () é recebida, o sistema passa para o estado calibrando, para o estado testando e, então, para o estado transmitindo, antes de retornar ao estado esperando. Se uma mensagem testar () é recebida, o sistema passa diretamente para o estado testando.
5. Se um sinal do relógio é recebido, o sistema passa para um estado de coleta, em que está coletando dados dos instrumentos. Por sua vez, cada instrumento é instruído para coletar seus dados.

De modo geral, não é normalmente necessário produzir um *statechart* para todos os objetos definidos. Muitos objetos em um sistema são objetos relativamente simples, e um modelo de máquina de estado não ajudaria os implementadores a compreender esses objetos.

12.2.5 Especificação de interface entre objetos

Uma parte importante de qualquer processo de projeto é a especificação das interfaces entre os diferentes componentes no projeto. Precisamos especificar as interfaces para que os objetos e outros componentes possam ser projetados em paralelo. Uma vez especificada uma

interface, os desenvolvedores de outros objetos podem considerar que aquela interface será implementada.

Os projetistas devem evitar informações de representação de interface em seus projetos de interface. Em vez disso, a representação deve ser oculta e as operações de objeto devem ser fornecidas para acessar e atualizar os dados. Se a representação estiver oculta, ela poderá ser modificada sem afetar os objetos que utilizam esses atributos. Isso leva a um projeto que é inerentemente de manutenção mais fácil. Por exemplo, uma representação em vetor de uma pilha pode ser modificada para uma representação em lista, sem afetar outros objetos que utilizem a pilha. Em contrapartida, sempre faz sentido expor os atributos em um modelo estático de projeto, uma vez que essa é a maneira mais compacta de ilustrar as características essenciais dos objetos.

Não existe, necessariamente, uma relação simples de 1:1 entre objetos e interfaces. O mesmo objeto pode ter várias interfaces que são, cada uma delas, pontos de vista dos métodos que elas fornecem. Essa condição é diretamente compatível com Java, em que as interfaces são declaradas separadamente dos objetos e os objetos 'implementam' interfaces. Da mesma maneira, todo um grupo de objetos pode ser acessado por meio de uma única interface.

O projeto de interface de objetos se ocupa de especificar os detalhes da interface para um objeto ou para um grupo de objetos. Isso significa definir as assinaturas e a semântica dos serviços fornecidos pelo objeto ou por um grupo de objetos. As interfaces podem ser especificadas em UML, utilizando a mesma notação de um diagrama de classes. Contudo, não há nenhuma seção de atributo e o estereótipo em UML <<interface>> deve ser incluído na parte do nome.

Uma abordagem alternativa é utilizar uma linguagem de programação para definir a interface. Essa opção está ilustrada na Figura 12.15, que mostra a especificação em Java da interface da estação meteorológica. À medida que as interfaces ficam mais complexas, essa abordagem se torna mais eficaz, porque os recursos de conferência de sintaxe no compilador podem ser utilizados para descobrir erros e inconsistências na descrição da interface. A descrição em Java pode mostrar que alguns métodos podem apresentar diferentes números de parâmetros. Portanto, o método **desativar** poderá ser aplicado à estação como um todo, se não possuir parâmetros, ou poderá desativar um único instrumento.

Figura 12.15 ▶

Descrição em Java da interface da estação meteorológica.

```
interface EstaçãoMeteorológica {
  public void EstaçãoMeteorológica () ;
  public void iniciar () ;
  public void iniciar (Instrumento i) ;
  public void desativar () ;
  public void desativar (Instrumento i) ;
  public void relatarClima () ;
  public void testar () ;
  public void testar (Instrumento i) ;
  public void calibrar (Instrumento i) ;
  public int obterID () ;
} //EstaçãoMeteorológica
```

12.3 Evolução de projeto

Uma importante vantagem de uma abordagem orientada a objetos para projeto é que ela simplifica o problema de fazer mudanças no projeto. A razão disso é que a representação do estado do objeto não influencia o projeto. Modificar os detalhes internos de um objeto provavelmente não afeta quaisquer outros objetos do sistema. Além disso, como os objetos são associados de maneira inadequada, em geral, é fácil e direto introduzir novos objetos sem provocar efeitos significativos no restante do sistema.

Para ilustrar a 'robustez' da abordagem orientada a objetos, considere que algumas capacidades de monitoração de poluição devem ser adicionadas a cada estação meteorológica. Isso envolve adicionar um medidor de qualidade do ar para computar a concentração de vários poluentes na atmosfera. As leituras de poluição são transmitidas ao mesmo tempo que os dados meteorológicos. Para modificar o projeto, as seguintes alterações devem ser feitas:

1. A classe de objeto chamada Qualidade do Ar deve ser introduzida como parte da EstaçãoMeteorológica no mesmo nível que DadosMeteorológicos.
2. Uma operação relatarQualidadedoAr deve ser adicionada à EstaçãoMeteorológica para enviar as informações sobre poluição para o computador central. O software de controle da estação meteorológica tem de ser modificado, de modo que as leituras de poluição sejam automaticamente coletadas quando solicitadas pelo objeto de nível superior EstaçãoMeteorológica.
3. Objetos que representam os tipos de instrumentos de monitoração de poluição devem ser adicionados. Nesse caso, podem ser medidos os níveis de óxido nitroso (NO), fumaça e benzeno.

A Figura 12.16 mostra a EstaçãoMeteorológica e os novos objetos adicionados ao sistema. Com exceção do nível mais alto do sistema (EstaçãoMeteorológica), nenhuma mudança de software é requerida nos objetos originais da estação meteorológica. A adição de coleta de dados sobre poluição não afeta, de maneira alguma, a coleta de dados meteorológicos.

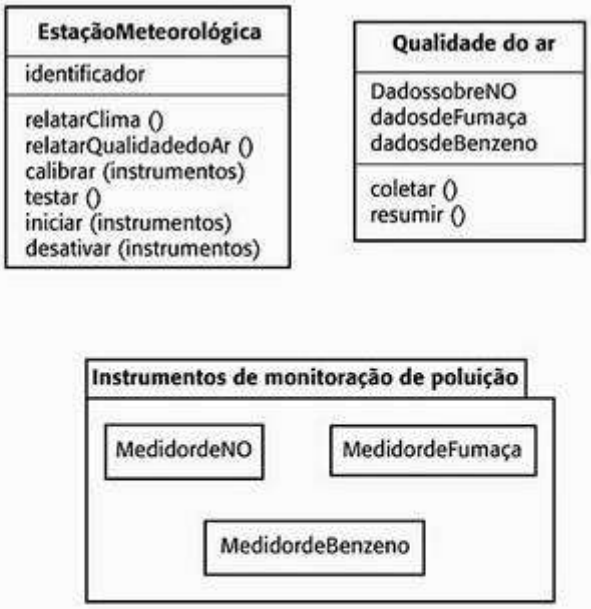


Figura 12.16 ▶ Novos objetos para apoiar a monitoração de poluição.

PONTOS-CHAVE

- O projeto orientado a objetos é um meio de projetar software, de modo que os componentes fundamentais do projeto representem objetos com seus próprios estados e operações, em vez de funções.
- Um objeto deve ter operações de construção e de inspeção, permitindo que seu estado seja inspecionado e modificado. O objeto fornece serviços (operações que utilizam informações de estado) a outros objetos. Os objetos são criados em *run-time*, utilizando uma especificação contida em uma definição de classe do objeto.
- Os objetos podem ser implementados na seqüência ou concorrentemente. Um objeto concorrente pode ser passivo, cujo estado só é modificado por meio de sua interface, ou ativo, que pode mudar seu próprio estado sem intervenção externa.
- A linguagem de modelagem unificada (UML) foi projetada para oferecer uma gama de diferentes notações, que podem ser utilizadas para documentar um projeto orientado a objetos.
- O processo de projeto orientado a objetos inclui atividades para projetar a arquitetura do sistema, identificar objetos do sistema, descrever o projeto, utilizando diferentes modelos de objeto, e documentar as interfaces de objeto.
- Uma série de diferentes modelos pode ser produzida durante um processo de projeto orientado a objetos. Eles incluem modelos estáticos (modelos de classe, de generalização e de associação) e modelos dinâmicos (modelos de seqüência e de máquina de estado).
- As interfaces com objetos devem ser definidas precisamente, de maneira que possam ser utilizadas por outros objetos. Uma linguagem de programação como Java pode ser utilizada para documentar as interfaces de objetos.
- Uma importante vantagem do projeto orientado a objetos é que ele simplifica a evolução do sistema.

LEITURAS SUGERIDAS

Comm. ACM, out. 1999. Essa edição especial inclui uma série de artigos sobre o uso prático da UML em projetos. O artigo de Bell e Schmidt sobre o uso de um estudo de caso de projetos é particularmente recomendado.

Using UML: software engineering with objects and components. Uma boa introdução, de fácil leitura, ao projeto orientado a objetos com o emprego de UML. (R. Pooley e P. Stevens, 1999, Addison Wesley Longman.)

The unified modeling language user guide. Um texto definitivo sobre UML e seu uso para descrever projetos orientados a objetos. Existem dois outros textos associados – um deles é um manual de referência sobre UML e o outro propõe um processo de desenvolvimento orientado a objetos. (G. Booch, I. Jacobson e J. Rumbaugh, 1999, Addison Wesley.)

EXERCÍCIOS

12.1 Explique por que adotar uma abordagem de projeto com base em objetos associados de maneira inadequada, que ocultam informações sobre suas representações, deve levar a um projeto que pode ser rapidamente modificado.

12.2 Por meio de exemplos, explique as diferenças entre um objeto e uma classe de objetos.

12.3 Sob quais circunstâncias deve ser apropriado desenvolver um projeto em que os objetos são executados simultaneamente?

12.4 Utilizando a notação gráfica em UML para as classes de objeto, projete as seguintes classes de objeto, com a identificação de atributos e operações. Use sua própria experiência para decidir sobre os atributos e as operações que devem ser associados com esses objetos:

- um telefone;
- uma impressora para uso doméstico;
- um aparelho de som;
- uma conta bancária, e
- um catálogo de biblioteca.

12.5 Desenvolva o projeto da estação meteorológica em detalhes, propondo descrições de interface com os objetos mostrados na Figura 12.11. Esse projeto pode ser expresso em Java, C++ ou UML.

12.6 Desenvolva o projeto da estação meteorológica para mostrar a interação entre o subsistema de coleta de dados e os instrumentos que coletam dados sobre o clima. Utilize os diagramas de seqüência para mostrar essa interação.

12.7 Identifique os possíveis objetos nos seguintes sistemas e desenvolva um projeto orientado a objetos para eles. Você pode fazer quaisquer suposições razoáveis sobre esses sistemas, ao derivar o projeto.

- Um sistema de gerenciamento de agenda e horário de um grupo de pessoas que trabalham juntas destina-se a dar apoio a uma agenda de reuniões e compromissos. Quando um compromisso, que envolve certo número de pessoas, deve ser marcado, o sistema encontra um horário disponível em comum, em cada uma de suas agendas, e marca o compromisso para aquele horário. Se não houver nenhum horário disponível em comum, o sistema interagirá com os usuários, a fim de reorganizar suas agendas pessoais e abrir espaço para o compromisso.
- Um posto de gasolina deve ser organizado para operação inteiramente automatizada. Os motoristas passam seus cartões de crédito em uma leitora conectada à bomba; o cartão é verificado por meio da comunicação com um computador da empresa de crédito e um limite de combustível é estabelecido. O motorista pode, então, abastecer com a quantidade de combustível solicitada. Quando a quantidade de combustível estiver completa e a mangueira da bomba for colocada de volta, o valor do combustível fornecido será debitado do cartão de crédito do motorista. O cartão de crédito é devolvido depois do débito. Se o cartão for inválido, será devolvido pela bomba antes que o combustível seja transferido.

12.8 Escreva definições precisas de interface em Java ou C+++ para os objetos que você definiu no Exercício 12.7.

12.9 Faça um diagrama de seqüência mostrando as interações de objetos em um sistema de agenda em grupo, quando o grupo de pessoas marca uma reunião.

12.10 Prepare um *statechart* mostrando as possíveis mudanças de estado em um ou mais dos objetos que você definiu no Exercício 12.7.

13 Projeto de software de tempo real

Objetivos

Este capítulo visa apresentar as técnicas utilizadas no projeto de sistemas de tempo real e descrever algumas das arquiteturas de sistema de tempo real. Depois de ler este capítulo, você poderá:

- compreender o conceito de um sistema de tempo real e por que esses sistemas são, em geral, implementados como um conjunto de processos concorrentes;
- conhecer um processo de projeto para sistemas de tempo real;
- compreender o papel de um executivo de tempo real, e
- compreender arquiteturas comuns de processos para sistemas de monitoramento e controle e sistemas de aquisição de dados.

Sumário

13.1 Projeto de sistemas
13.2 Executivos de tempo real
13.3 Sistemas de monitoramento e controle
13.4 Sistemas de aquisição de dados

Os computadores são utilizados para controlar uma ampla gama de sistemas, que abrangem desde simples máquinas domésticas até completas instalações de produção. Eles interagem diretamente com os dispositivos de hardware. O software nesses sistemas é um sistema de tempo real integrado, que tem de reagir a eventos gerados pelo hardware e emitir sinais de controle, em resposta a esses eventos. Ele está *embutido* em um sistema de hardware de maior porte e deve responder, de *tempo real*, a eventos do ambiente do sistema.

Os sistemas de tempo real são diferentes de outros tipos de sistemas de software; seu funcionamento correto depende da resposta do sistema aos eventos dentro de um determinado intervalo de tempo (geralmente, curto). Eis a definição de um sistema de tempo real:

> Um sistema de tempo real é um sistema de software cujo funcionamento correto depende dos resultados produzidos por ele e do tempo em que esses resultados são produzidos. Um sistema de tempo real leve é aquele cuja operação será *degradada*, se os resultados não forem produzidos de acordo com os requisitos de tempo especificados. Um sistema de tempo real rígido é aquele cuja operação será *incorreta* se os resultados não forem produzidos de acordo com a especificação de tempo.

Uma maneira de ver um sistema em tempo real é como um sistema de estímulo/resposta. Considerando um determinado estímulo fornecido, o sistema deve produzir uma resposta correspondente. O comportamento de um sistema de tempo real, portanto, pode ser definido relacionando os estímulos que são recebidos pelo sistema, as respostas associadas e o tempo em que a resposta deve ser produzida.

Os estímulos são de duas classes:

1. *Estímulo periódico* Eles ocorrem em intervalos de tempo previsíveis. Por exemplo, o sistema pode examinar um sensor a cada 50 milissegundos e reagir (responder) de acordo com o valor (o estímulo) desse sensor.
2. *Estímulo aperiódico* Eles ocorrem irregularmente e, em geral, são assinalados utilizando-se o mecanismo de interrupção do computador. Um exemplo desse estímulo seria uma interrupção que indica que uma transferência de E/S (entrada/saída) foi concluída e que os dados estavam disponíveis no buffer.

Os estímulos periódicos em um sistema de tempo real são normalmente gerados por sensores associados ao sistema. Eles fornecem informações sobre o estado do ambiente do sistema. As respostas são direcionadas para um conjunto de atuadores, que controlam alguma unidade de hardware que, então, influencia o ambiente do sistema. Os estímulos aperiódicos podem ser gerados por atuadores ou por sensores. Eles sempre indicam alguma condição excepcional, como uma falha de hardware, que tem de ser resolvida pelo sistema. Esse modelo sensor-sistema-atuador de um sistema de tempo real integrado está ilustrado na Figura 13.1.

Figura 13.1 ▶
Modelo geral de um sistema de tempo real.

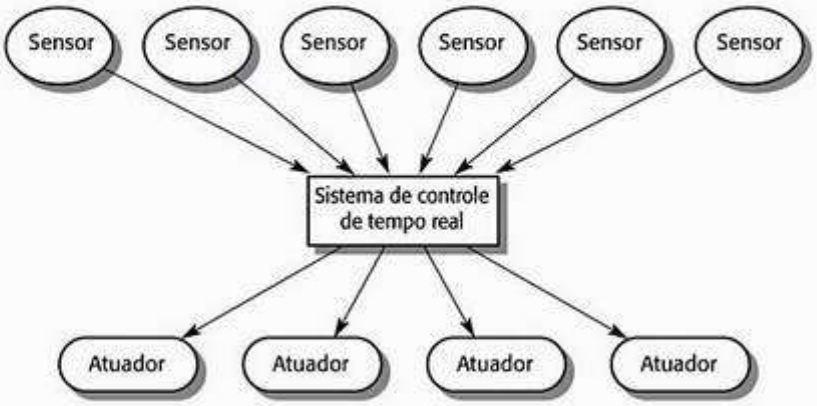


Figura 13.2 ▶

Processos de controle sensor/atuador.

Um sistema de tempo real tem de responder a estímulos que ocorrem em diferentes momentos. Sua arquitetura deve, portanto, ser organizada de modo que o controle seja transferido ao manipulador apropriado para aquele estímulo, assim que ele for recebido. Isso é impraticável em programas seqüenciais. Os sistemas de tempo real são, portanto, normalmente projetados como um conjunto de processos concorrentes e cooperantes. Parte do sistema de tempo real (o executivo de tempo real) é dedicado ao gerenciamento desses processos.

A generalidade desse modelo de estímulo/resposta de um sistema de tempo real leva ao modelo de arquitetura genérica, em que existem três tipos de processos (veja a Figura 13.2). Para cada tipo de sensor, há um processo de gerenciamento de sensor Os processos computacionais calculam a resposta requerida para o estímulo recebido pelo sistema e os processos de controle de atuador gerenciam a operação do atuador. Esse modelo permite que os dados sejam coletados com rapidez a partir do sensor (antes que a próxima entrada se torne disponível) e possibilita que o processamento e a resposta do atuador associado sejam realizados posteriormente.

13.1 Projeto de sistemas

Como foi discutido no Capítulo 2, parte do processo de projeto de sistema envolve decidir quais capacitações do sistema devem ser implementadas em software e quais em hardware. As restrições de tempo e outros requisitos podem significar que certas funções de sistema, como o processamento de sinais, devem ser implementadas utilizando-se hardware especialmente projetado. O processo de projeto de sistema pode, portanto, envolver o projeto de hardware de propósito especial, bem como o projeto de software de tempo real.

Os componentes de hardware apresentam desempenho muito melhor do que o software equivalente. Gargalos de processamento de sistema podem ser identificados e substituídos por hardware, evitando, desse modo, a dispendiosa otimização de software. Fornecer desempenho de hardware significa que o projeto de software pode ser estruturado para ter facilidade de adaptação e as considerações de desempenho podem ficar em segundo lugar.

As decisões sobre a divisão hardware/software devem, se possível, ficar em segundo plano durante o tempo que for possível, ao longo do processo de projeto. Isso implica que a arquitetura de sistema deve se constituir de componentes *stand-alone*, que possam ser implementados tanto em hardware como em software. Felizmente, construir um projeto dessa maneira é exatamente a meta do projetista que tenta projetar um sistema de fácil manutenção. Um bom processo de projeto de sistema deve, portanto, resultar em um sistema que possa ser implementado tanto em hardware quanto em software.

O processo de projeto para sistemas de tempo real difere de outros processos de projeto de software, porque os tempos de resposta do sistema devem ser considerados logo no início no processo. Os eventos (estímulos), em vez dos objetos ou das funções, devem ser a parte central do processo de projeto. Há vários estágios nesse processo de projeto:

1. Identificar os estímulos que o sistema deve processar e as respostas associadas.
2. Para cada estímulo e cada resposta associada, identificar as restrições de tempo que se aplicam ao processamento de estímulo e de resposta.
3. Agregar o processamento de estímulo e de resposta em uma série de processos simultâneos. Um bom modelo geral para a arquitetura do sistema é associar um processo a cada classe de estímulo e resposta, como mostra a Figura 13.2.
4. Para cada estímulo e resposta, projetar algoritmos para realizar os cálculos requeridos. Os projetos de algoritmos, com freqüência, têm de ser desenvolvidos relativamente no início do processo de projeto, para dar uma indicação da quantidade de processamento requerida e do tempo necessário para completar esse processamento.
5. Projetar um sistema de escalonamento que assegure que os processos sejam iniciados a tempo de atender a seus prazos.
6. Integrar o sistema sob o controle de um executivo de tempo real.

Naturalmente, esse é um processo iterativo. Uma vez estabelecido um processo de arquitetura e decidida uma política de escalonamento, são necessárias extensas avaliações e simulações, a fim de verificar se o sistema atenderá às suas restrições de tempo. Essa análise pode revelar que o sistema não terá um desempenho adequado. A arquitetura do processo, a política de escalonamento, o executivo ou todos esses pontos podem, então, precisar ser reprojetados para melhorar o desempenho do sistema.

Analisar a sincronia de um sistema de tempo real é difícil. Devido à natureza imprevisível dos estímulos aperiódicos, os projetistas devem fazer certas suposições quanto à probabilidade de que esses estímulos ocorram (e, portanto, solicitem serviços) em qualquer momento específico. Essas suposições podem estar incorretas e o desempenho do sistema, depois da entrega, pode não ser adequado. Dasarthy (1985) e Burns (1991) discutem questões gerais da validação de tempo. O livro de Gomaa (1993) traz uma boa discussão sobre os métodos de análise de desempenho.

Os processos de um sistema de tempo real devem ser coordenados. Os mecanismos de coordenação de processos garantem a exclusão mútua de recursos compartilhados. Quando um processo estiver modificando um recurso compartilhado, os outros processos não devem ser capazes de modificar esse recurso. Os mecanismos para assegurar a exclusão mútua incluem semáforos (Dijkstra, 1968b), monitores (Hoare, 1974) e regiões críticas (Brinch-Hansen, 1973). Esses mecanismos não são abordados neste livro, uma vez que estão bem documentados em textos sobre sistemas operacionais (Tanenbaum e Woodhull, 1997; Silberschaltz e Galvin, 1998).

Como os sistemas de tempo real precisam respeitar suas restrições de tempo, pode não ser prático utilizar estratégias de projeto para sistemas de tempo real rígidos, que envolvam prazos de implementação adicionais. Por exemplo, o projeto orientado a objetos envolve ocultar representações de dados e acessar valores de atributos por meio de operações definidas com o objeto. Isso envolve atrasos inevitáveis e a conseqüente perda de desempenho.

13.1.1 Modelagem de sistemas de tempo real

Os sistemas de tempo real devem responder a eventos que ocorrem em intervalos irregulares. Esses eventos (estímulos) freqüentemente levam o sistema a passar para um estado diferente. Por essa razão, a modelagem de máquinas de estado, descrita no Capítulo 7, pode ser utilizada como uma maneira de descrever um sistema de tempo real.

Um modelo de estado de um sistema considera que, a qualquer momento, o sistema está em um estado, dentre uma série de possíveis estados. Quando um estímulo é recebido, isso pode dar início a uma transição para um estado diferente. Por exemplo, um sistema que controla uma válvula pode passar do estado 'Válvula aberta' para o estado 'Válvula fechada', quando um comando (estímulo) do operador é recebido.

Essa abordagem de modelagem de sistema é ilustrada com o uso do modelo de um forno de microondas simples, apresentado no Capítulo 7. A Figura 13.3 mostra um modelo de máquina de estado de um forno de microondas simples, equipado com botões para ajustar a potência e o relógio e iniciar o sistema. Os retângulos com cantos arredondados representam os estados de sistema e os rótulos com setas representam os estímulos que forçam a transição de um estado para outro. Os nomes escolhidos no diagrama da máquina de estado são descritivos e as informações associadas indicam as ações tomadas pelos atuadores do sistema ou as informações que são exibidas.

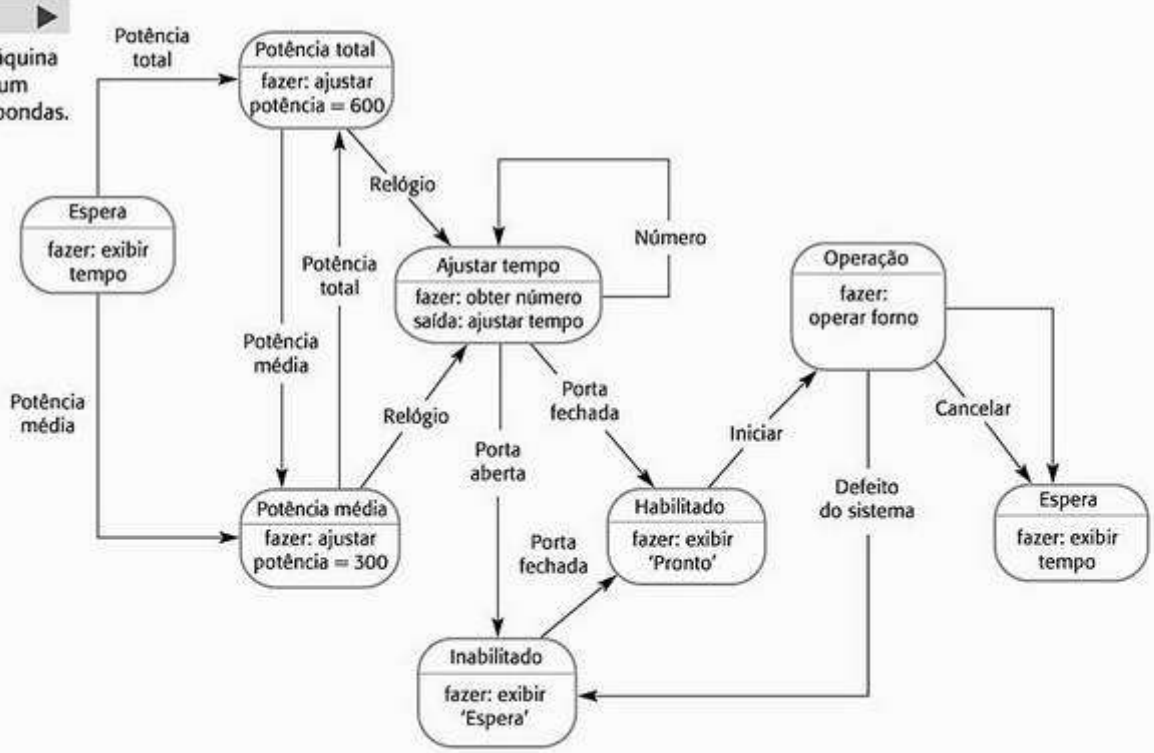
É possível traçar a operação do forno pela leitura do modelo da esquerda para a direita. No estado inicial (Espera), o usuário pode selecionar potência total ou média. O próximo estado inicia quando o usuário pressiona o botão do relógio e define o tempo. A operação pode, então, ser habilitada quando a porta do forno é fechada e o alimento é cozido no estado de Operação. Por fim, quando o cozimento estiver terminado, o forno retornará ao estado de Espera.

Os modelos de máquina de estado são uma boa opção para representar o projeto de sistema em tempo real e também são independentes da linguagem. Por essa razão, eles são parte integrante de métodos de projeto de tempo real (Ward e Mellor, 1985; Harel, 1987, 1988). O método de Harel, que tem como base uma notação chamada *statecharts*, abordou o problema da inerente complexidade dos modelos de máquina de estado. Os *statecharts* estruturam os modelos de estado, de modo que os grupos de estado possam ser considerados uma entidade única. A notação também permite que sistemas concorrentes sejam representados como modelos de estado. Os modelos de estado são também compatíveis com a linguagem de modelagem unificada (Rumbaugh *et al.*, 1999a) e, neste texto, é utilizada a notação em UML.

13.1.2 Programação de tempo real

A linguagem de programação utilizada para implementar um sistema de tempo real pode também influenciar o projeto. Os sistemas de tempo real rígidos ainda são, algumas vezes, programados em linguagem assembly, de modo que prazos curtos podem ser cumpridos. As

Figura 13.3 ▶
Modelo de máquina de estado de um forno de microondas.



linguagens de nível de sistemas, como a linguagem C, que permitem que códigos eficazes sejam gerados também podem ser utilizadas.

A vantagem de utilizar uma linguagem de programação de sistemas de baixo nível, como C, é que ela permite o desenvolvimento de programas muito eficientes. Contudo, a linguagem não inclui nenhum princípio para lidar com a concorrência ou com o gerenciamento de recursos compartilhados. Ela se baseia em sistemas operacionais ou recursos de executivos e, por isso, existe um enfoque maior nos erros de programação. Muitas vezes, os programas também são mais difíceis de ser entendidos.

A linguagem Ada foi originalmente projetada para a implementação de sistemas embutidos e tem características como execução de tarefas, exceções e cláusulas de representação. Sua capacidade de *rendezvous* é um bom mecanismo de propósito geral para a sincronização de tarefas (Burns e Wellings, 1990; Barnes, 1994). Infelizmente, a versão original da linguagem Ada (Ada 83) era inadequada para a implementação de sistemas de tempo real rígidos. Era impossível especificar prazos para tarefas, não havia nenhuma exceção integrada, se um prazo de conclusão não fosse cumprido, e era imposta uma política de primeiro a entrar/primeiro a sair (*first in/first out*) para o atendimento a uma fila de entrada de tarefas. Uma revisão do padrão Ada, de 1983 (Barnes, 1998), atendeu a algumas dessas questões. A versão revisada da linguagem forneceu tipos protegidos, que permitiram a implementação mais fácil de estruturas protegidas de dados compartilhados e maior controle sobre o escalonamento de tarefas e tempo. Isso melhorou a linguagem Ada como uma linguagem de programação de tempo real, mas ainda não fornece suficiente controle para sistemas complexos em tempo real.

As versões iniciais de Java (então chamada de *Oak*) foram projetadas para escrever sistemas integrados de pequena escala, como aqueles de controladores de equipamentos domésticos. Os projetistas Java incluíram, portanto, alguma compatibilidade para processos concorrentes, na forma de objetos concorrentes (*threads*) e métodos sincronizados. Contudo, como esses sistemas não têm restrições rígidas de tempo, Java não inclui recursos para controlar o escalonamento de *threads* ou para especificar que os threads devam ser executados em momentos específicos.

A linguagem Java, portanto, não é adequada para programar sistemas de tempo real rígidos ou sistemas em que os processos tenham prazos rígidos. Os problemas fundamentais com Java como uma linguagem de programação em tempo real são:

1. Não é possível especificar o momento em que os threads devam ser executados.
2. A coleta de lixo (*garbage*) não pode ser controlada – ela pode ser iniciada a qualquer momento. Portanto, o comportamento dos threads relacionado ao tempo é imprevisível.
3. Não é possível descobrir os tamanhos das filas associadas com recursos compartilhados.
4. A implementação da Máquina Virtual Java varia de um computador para outro, de modo que o mesmo programa pode ter diferentes comportamentos relacionados ao tempo.
5. A linguagem não permite a análise detalhada de espaço de *run-time* ou de processador.

No momento em que este livro foi escrito, estava sendo realizado um trabalho para resolver alguns desses problemas e para definir uma versão de tempo real para Java (Nilsen, 1998). Contudo, não está claro como isso pode ser separado da implementação básica da Máquina Virtual Java. Existe, portanto, um conflito inevitável entre a portabilidade da linguagem e as características de tempo real.

13.2 Executivos de tempo real

Um executivo de tempo real é análogo a um sistema operacional em um computador de propósito geral. Ele gerencia a alocação de processos e recursos em um sistema de tempo real. Ele

inicia e interrompe os processos apropriados, de modo que os estímulos possam ser manipulados, e aloca a memória e os recursos do processador. No entanto, não inclui normalmente mais recursos complexos do sistema operacional, como o gerenciamento de arquivo.

Baker e Scallon (1986) apresentam uma boa discussão dos recursos requeridos nos executivos em tempo real. Cooling (1991) também aborda esse tópico e discute brevemente os produtos comerciais dos executivos de tempo real. Embora existam vários produtos disponíveis de executivos de tempo real, os requisitos especializados de muitos sistemas de tempo real exigem freqüentemente que o executivo seja projetado como parte do sistema.

Os componentes de um executivo (Figura 13.4) dependem do tamanho e da complexidade do sistema de tempo real que está em desenvolvimento. Normalmente, para todos os casos, exceto os sistemas mais simples, eles incluirão:

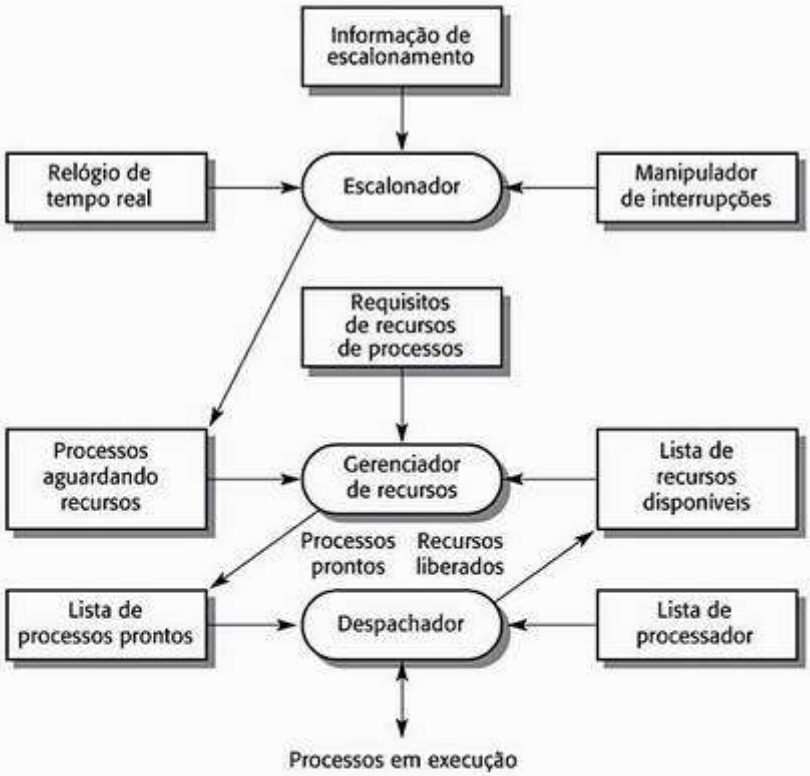
1. *Um relógio de tempo real* Esse relógio fornece informações para escalonar os processos periodicamente.
2. *Um manipulador de interrupções* Gerencia os pedidos aperiódicos de serviços.
3. *Um escalonador* Esse componente é o responsável por examinar os processos que podem ser executados e escolher um deles para execução.
4. *Um gerenciador de recursos* Considerando um processo que esteja programado para execução, o gerenciador de recursos aloca a memória apropriada e os recursos de processador.
5. *Um despachador* Esse componente é responsável por iniciar a execução de um processo.

Os sistemas que fornecem um serviço contínuo, como os sistemas de telecomunicação e de monitoramento, com altos níveis de requisitos de confiabilidade, também podem incluir outras capacidades executivas:

- *Um gerenciador de configuração* Esse gerenciador é responsável pela configuração dinâmica (Kramer e Magee, 1985) do hardware do sistema. Os módulos de hardware podem ser tirados de serviço e o sistema pode ser atualizado pela adição de novo hardware, sem interromper o sistema.

Figura 13.4 ▶

Componentes de um executivo de tempo real.



- *Um gerenciador de defeitos* Esse componente é responsável por detectar defeitos de hardware e de software e tomar as medidas apropriadas para a recuperação a partir desses defeitos. Os princípios de tolerância a defeitos e a recuperação a partir de defeitos serão discutidos no Capítulo 18.

Os estímulos processados por um sistema de tempo real, geralmente, têm diferentes níveis de prioridade. Para alguns estímulos, como aqueles associados com certos eventos excepcionais, é essencial que seu processamento seja concluído dentro do limite de tempo especificado. Outros processos podem ser adiados com segurança, se um processo mais importante solicitar serviço. Conseqüentemente, o executivo para um sistema de tempo real deve ser capaz de gerenciar pelo menos dois níveis de prioridades para os processos do sistema:

1. *Nível de interrupção* Este é o nível de mais alta prioridade. Ele é alocado para processos que precisam de uma resposta muito rápida. Um desses processos será o processo de relógio de tempo real.
2. *Nível de relógio* Esse nível de prioridade é alocado aos processos periódicos.

É possível que haja um outro nível de prioridade alocado para os processos de background (como um processo de autoverificação) que não tenham de cumprir prazos de conclusão em tempo real. Esses processos são escalonados para execução quando o processador estiver disponível.

Dentro de cada um desses níveis de prioridade, diferentes classes de processo podem ser alocadas em diferentes prioridades. Por exemplo, é possível que haja várias linhas de interrupção. Uma interrupção a partir de um dispositivo muito rápido pode ter de prever o processamento da interrupção de um dispositivo mais lento, para evitar a perda de informações. A alocação de prioridades de processo, de modo que todos os processos sejam atendidos em tempo, geralmente requer ampla análise e simulação.

13.2.1 Gerenciamento de processos

O gerenciamento de processos em um executivo de tempo real se ocupa de gerenciar o conjunto de processos concorrentes que fazem parte do sistema de tempo real. O gerente de processos deve escolher um processo para execução, alocar memória e recursos de processador para esse processo e iniciar sua execução em um processador.

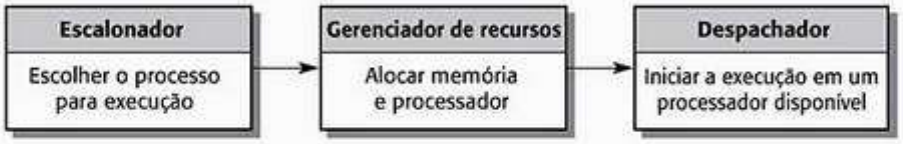
Os processos periódicos são aqueles que precisam ser executados em intervalos de tempo previamente especificados para a aquisição de dados e o controle de atuador. O executivo utiliza seu relógio de tempo real para determinar quando um processo deve ser executado. Na maioria dos sistemas de tempo real, haverá várias classes de processos periódicos. Elas terão diferentes períodos (o tempo entre a execução de processos), tempos de execução e prazos de conclusão (o tempo em que o processamento deve ser concluído). O executivo precisa escolher o processo apropriado para a execução em qualquer momento.

O relógio de tempo real é configurado para fazer o som de um 'tique' periodicamente, em geral, em poucos milissegundos. O 'tique' do relógio dá início a um processo de nível de interrupção que, por sua vez, programa o gerenciador de processos para processos periódicos. O processo de nível de interrupção não é normalmente responsável por gerenciar processos periódicos, porque o processamento de interrupção tem de ser concluído o mais rápido possível.

As ações realizadas pelo executivo para o gerenciamento de processos periódicos são mostradas na Figura 13.5. A lista de processos periódicos é examinada pelo escalonador, que seleciona um processo para ser executado. A escolha depende da prioridade do processo, dos períodos de processo, dos tempos de execução esperados e dos prazos de conclusão dos processos prontos. Algumas vezes, dois processos com diferentes prazos devem ser executados ao mesmo

Figura 13.5 ▶

Ações de um executivo de tempo real requeridas para iniciar um processo.



tempo estabelecido pelo 'tique' do relógio. Nesse caso, um deles tem de ser adiado, desde que seu prazo de conclusão possa ser atendido.

Quando uma interrupção é detectada pelo executivo, isso indica que algum serviço está sendo requerido. O mecanismo de interrupção do computador faz com que o controle seja transferido para uma locação de memória predeterminada, que contém uma instrução para passar para uma rotina de serviço de interrupção. As rotinas de serviço de interrupção têm de ser simples e curtas, além de apresentar tempos de execução rápidos. Enquanto a interrupção está sendo atendida, outras interrupções são desabilitadas e serão ignoradas pelo sistema. Para tornar a probabilidade de perda de informações tão pequena quanto possível, o tempo gasto nesse estado deve ser minimizado.

A rotina de serviço deve desabilitar outras interrupções, a fim de evitar que ela mesma seja interrompida. É preciso descobrir a causa da interrupção e iniciar, com alta prioridade, o processo para manipular o estímulo causador da interrupção. Em alguns sistemas de aquisição de dados com grande velocidade, o manipulador de interrupção salva os dados que a interrupção assinalou como disponíveis em um buffer, para processamento posterior. As interrupções são habilitadas novamente e o controle é devolvido ao executivo.

A qualquer momento, é possível que haja vários processos, com diferentes prioridades, que devem ser executados. Fica por conta do escalonador decidir a ordem de execução. O escalonamento efetivo é essencial, caso os requisitos de tempo real para um sistema devam ser cumpridos. Existem duas estratégias fundamentais de escalonamento:

1. *Escalonamento não preemptivo (non pre-emptive)* Uma vez escalonado um processo para execução, ele é executado até ser concluído ou até que seja interrompido, por alguma razão, como no caso de ter de aguardar por uma entrada. Isso causa problemas quando existem processos com diferentes prioridades e um processo de alta prioridade precisa aguardar pelo término de um processo de baixa prioridade.

2. *Escalonamento preemptivo (pre-emptive)* A execução de um processo pode ser interrompida se um processo de prioridade mais alta solicitar serviço. O processo com prioridade mais alta interrompe o processo com prioridade mais baixa e é alocado para um processador.

Dentro dessas estratégias diferentes algoritmos de escalonamento foram desenvolvidos. Dentre eles destacam-se o escalonamento *round-robin*, em que cada processo é executado por vez, o escalonamento monotônico, em que a prioridade é dada ao processo de período e ao prazo de término mais curtos (Burns e Wellings, 1997). Cada uma dessas estratégias tem diferentes vantagens e desvantagens, mas neste livro não as abordarei.

As informações sobre o processo a ser executado são fornecidas ao gerente de recursos, que aloca memória e, em um sistema com vários processadores, um processador para esse processo. Em seguida, o processo é colocado na 'lista de prontos', uma lista de processos que estão prontos para execução. Quando um processador termina de executar um processo e se torna disponível, o despachador é chamado. Ele faz a leitura da 'lista de prontos', a fim de encontrar um processo que possa ser executado no processador disponível e inicia sua execução.

13.3 Sistemas de monitoramento e controle

Existe um grande número de classes de sistemas-padrão para os sistemas de tempo real, como, por exemplo, sistemas de monitoramento, sistemas de aquisição de dados e sistemas de controle e comando. Cada um desses sistemas tem uma arquitetura de processo característica; assim, quando um sistema é desenvolvido, sua arquitetura pode, geralmente, derivar de um desses tipos-padrão. Portanto, em vez de discutir questões gerais de projeto, neste livro é ilustrado o projeto de sistemas de tempo real com o uso desses modelos genéricos.

Os sistemas de monitoramento e controle são uma classe importante de sistemas de tempo real. Eles verificam os sensores que fornecem informações sobre o ambiente do sistema e realizam ações, dependendo da leitura do sensor. Os sistemas de monitoramento realizam uma ação quando algum valor excepcional é detectado pelo sensor. Os sistemas de controle controlam continuamente os atuadores de hardware, dependendo do valor dos sensores associados.

Considere o seguinte exemplo:

> Um sistema de alarme contra roubo deve ser implementado em um edifício. Ele utiliza tipos diferentes de sensores, incluindo detectores de movimento em aposentos individuais, sensores de janela nas janelas do andar térreo, que detectam se uma janela foi quebrada, e sensores de porta, que detectam a abertura de portas no corredor. Existem 50 sensores de janelas, 30 sensores de portas e 200 detectores de movimento no sistema.
>
> Quando um sensor detecta a presença de um intruso, o sistema automaticamente chama a polícia local e, com um sintetizador de voz, relata a localização do alarme. Ele liga as luzes nas salas ao redor do sensor ativado e aciona um alarme audível. O sistema de alarme é normalmente alimentado pelo sistema de energia principal, mas é equipado com uma bateria de backup. A perda de energia é detectada utilizando-se um monitor de circuito de energia separado, que monitora a voltagem principal. Ele interrompe o sistema de alarme quando uma queda de voltagem é detectada.

Este é um sistema de tempo real leve, que não tem rigorosos requisitos de tempo. Os sensores não precisam detectar eventos de grande velocidade e, assim, precisam ser verificados somente duas vezes por segundo.

O processo de projeto tem início pela identificação de estímulos aperiódicos que o sistema recebe e das respostas associadas. O projeto foi simplificado ignorando os estímulos gerados por procedimentos de autoverificação de sistema e os estímulos externos gerados para testar o sistema ou para desligá-lo, no caso de um alarme falso. Isso significa que existem apenas duas classes de estímulos que precisam ser processadas:

1. *Falha de energia* Esse estímulo é gerado pelo monitor de circuito. A resposta requerida consiste em alternar o circuito para o circuito de backup de energia, emitindo sinal para um dispositivo eletrônico de ativação de energia.
2. *Alarme contra intrusos* Este é um estímulo gerado por um dos sensores do sistema. A resposta a esse estímulo consiste em verificar o número da sala do sensor ativo, fazer um chamado para a polícia, iniciar o sintetizador de voz para gerenciar o chamado e ligar o alarme sonoro contra intrusos e as luzes do prédio na área.

A etapa seguinte no processo de projeto é considerar as restrições de tempo associadas a cada estímulo e às respostas relativas. Essas restrições de tempo são mostradas na Figura 13.6. Nesse diagrama, as diferentes classes de sensores que podem gerar um estímulo de alarme foram relacionadas separadamente, uma vez que elas têm diferentes requisitos de tempo.

A alocação das funções de sistema aos processos simultâneos é o próximo estágio de projeto. Existem três tipos diferentes de sensores que devem ser verificados periodicamente; assim,

Figura 13.6 ▶

Requisitos de tempo de estímulo/resposta.

Estímulo/resposta	Requisitos de tempo
Interrupção por falta de energia	A transferência para a fonte de energia reserva deve ser concluída dentro de um prazo de 50 milissegundos.
Alarme de porta	Cada alarme de porta deve ser verificado duas vezes por segundo.
Alarme de janela	Cada alarme de janela deve ser verificado duas vezes por segundo
Detector de movimento	Cada detector de movimento deve ser verificado duas vezes por segundo.
Alarme sonoro	O alarme sonoro deve ser ligado dentro de meio segundo a partir do momento em que um alarme é ativado por um sensor.
Interruptores de luz	As luzes devem ser ligadas dentro de meio segundo a partir do momento em que um alarme é ativado por um sensor.
Comunicações	O chamado para a polícia deve ser iniciado dentro de dois segundos a partir do momento em que um alarme é ativado por um sensor.
Sintetizador de voz	Uma mensagem sintetizada deve estar disponível dentro de quatro segundos a partir do momento em que um alarme é ativado por um sensor.

cada um desses tipos de sensores tem um processo associado. Há um sistema dirigido por interrupção para manipular a falta de energia e o chaveamento, um sistema de comunicação, um sintetizador de voz, um sistema de alarme sonoro e um sistema de chaveamento de energia para ligar as luzes em volta do sensor. Cada um desses sistemas é controlado por um processo independente. Isso sugere a arquitetura de sistema mostrada na Figura 13.7.

Na notação utilizada na Figura 13.7, as setas rotuladas que associam processos indicam fluxos de dados entre os processos, com o rótulo indicando o tipo de fluxo de dados. A seta associada a cada processo na parte superior à direita indica controle. As setas em um processo periódico utilizam linhas cheias, com o número mínimo de vezes que um processo deve ser executado por segundo conforme o rótulo.

Figura 13.7 ▼

Arquitetura de processo do sistema de alarme contra roubo.

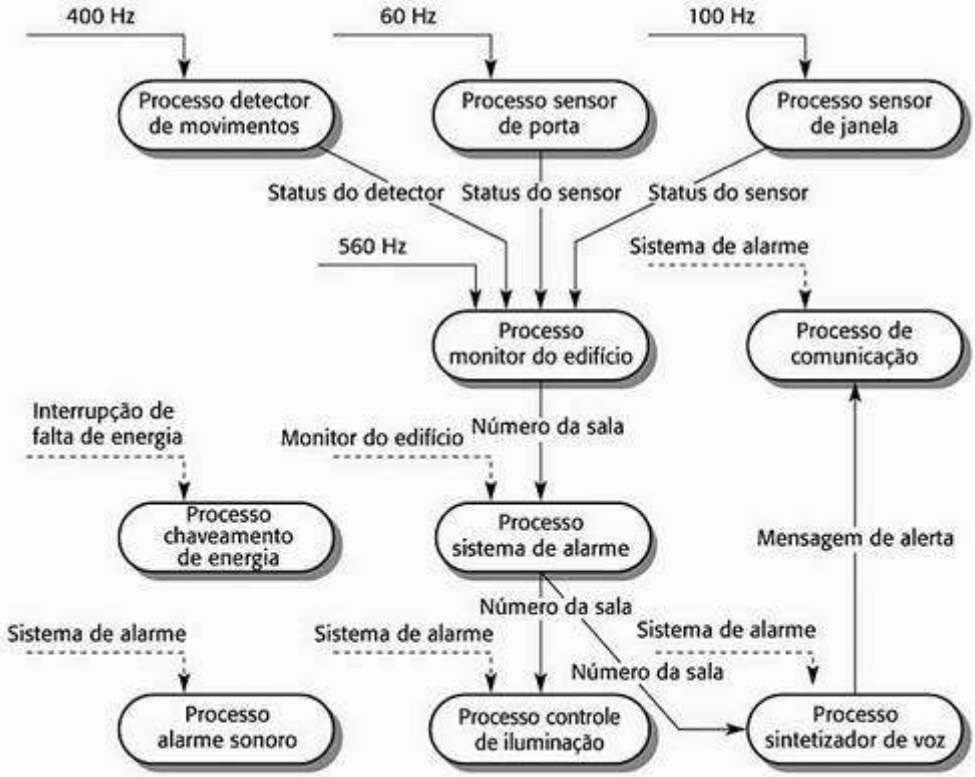


Figura 13.8 ▶

Implementação em Java do processo de monitoramento do prédio.

```
// Veja http://www.software-engin.com/para obter links do código
// Java completo para esse exemplo

class BuildingMonitor extends Thread {

  BuildingSensor win, door, move ;

  Siren siren = new Siren () ;
  Lights lights = new Lights () ;
  Synthesizer synthesizer = new Synthesizer () ;
  DoorSensors doors = new DoorSensor (30) ;
  WindowSensors windows = new WindowSensors (50) ;
  MovementSensors movements = new MovementSensors (200) ;
  PowerMonitor pm = new PowerMonitor () ;

  BuildingMonitor ()
  {
      // iniciar todos os sensores e dar início aos processos
      siren.start () ; lights.start () ;
      synthesizer.start () ; windows.start () ;
      doors.start () ; movements.start () ; pm.start () ;
  }

  public void run ()
  {
      int room = 0 ;
      while (true)
      {
          // varrer os sensores de movimento pelo menos duas vezes por segundo (400 Hz)
          move = movements.getVal () ;
          // varrer os sensores de janela pelo menos duas vezes por segundo (100 Hz)
          win = windows.getVal () ;
          // varrer os sensores de porta pelo menos duas vezes por segundo (60 Hz)
          door = doors.getVal () ;
          if (move.sensorVal == 1 | door.sensorVal == 1 | win.sensorVal == 1)
              {
                  // um sensor indicou um intruso
                  if (move.sensorVal == 1)  room = move.room ;
                  if (door.sensorVal == 1)  room = door.room ;
                  if (win.sensorVal == 1)   room = win.room ;

                  lights.on (room) ; siren.on () ; synthesizer.on (room) ;
                  break ;
              }
      }
      lights.shutdown () ; siren.shutdown () ; synthesizer.shutdown () ;
      windows.shutdown () ; doors.shutdown () ; movements.shutdown () ;

  } // run
} //BuildingMonitor
```

A freqüência de escalonamento de período é determinada pelo número de sensores e pelos requisitos de tempo do sistema. Por exemplo, existem 30 sensores de porta que devem ser interrogados duas vezes por segundo. Isso significa que o processo de sensor de porta associado deve ser executado 60 vezes por segundo (60 Hz). Da mesma maneira, o processo de detector de movimento deve ser executado 400 vezes por segundo.

Os processos aperiódicos têm linhas pontilhadas nas setas de controle. As linhas são rotuladas com o evento que faz com que o processo seja escalonado. As informações de controle nos processos de atuador (ou seja, o controlador de alarme sonoro, o controlador de iluminação, entre outros) indicam que seu início vem de um comando explícito a partir do processo Sistema de alarme. Nem todos os processos precisam receber dados de outros processos.

Por exemplo, o processo responsável por gerenciar uma falha de energia necessita de dados de qualquer outra parte no sistema.

Esses processos podem ser implementados em Java com o uso de *threads*. A Figura 13.8 mostra o código Java que implementa o processo BuildingMonitor, que verifica os sensores do sistema. Se esses sensores assinalarem um intruso, o software ativará o sistema de alarme associado. Supõe-se, nesse projeto, que os requisitos de tempo (incluídos como comentários) possam ser cumpridos. Conforme foi comentado anteriormente, Java 2.0 não inclui recursos para permitir que a freqüência de execução de thread seja especificada.

Como esse sistema não tem requisitos rígidos de tempo, é possível implementá-lo em Java. Naturalmente, não existe uma maneira de se garantir, em Java 2.0, que as especificações de tempo serão cumpridas. Nesse exemplo, todos os sensores são varridos o mesmo número de vezes. Isso não causaria problemas práticos nesse sistema.

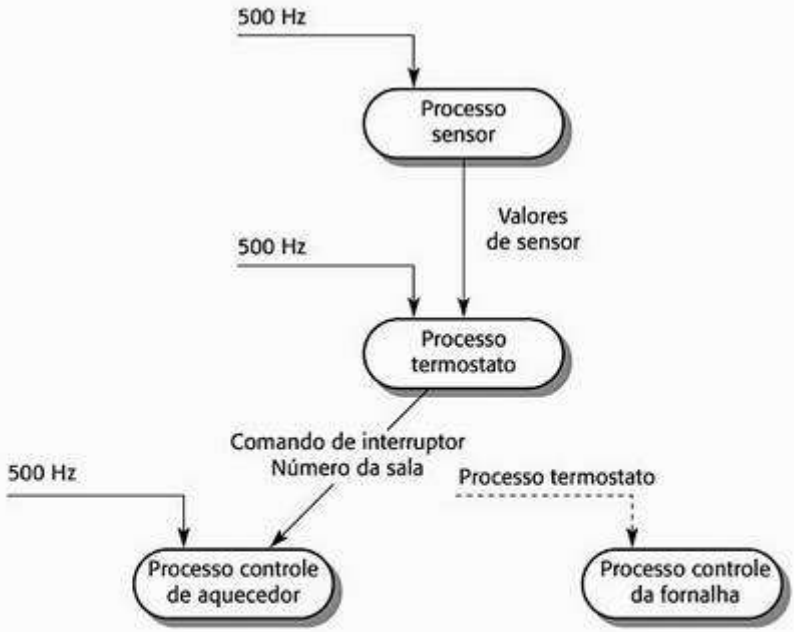
Uma vez estabelecida a arquitetura do processo de sistema, devem ser projetados os algoritmos para o processamento de estímulos e a geração de respostas. Como foi discutido na Seção 13.1, esse estágio de projeto detalhado é necessário no início do processo de projeto, a fim de assegurar que o sistema pode cumprir com suas restrições de tempo especificadas. Se os algoritmos associados forem complexos, poderão ser necessárias mudanças nos requisitos de tempo. A menos que o processamento de sinais seja requerido, os algoritmos de sistema em tempo real são freqüentemente muito simples. Eles apenas exigem uma alocação de memória a ser verificada, alguns cálculos simples a serem realizados ou um sinal a ser emitido. Esse é o caso do sistema de alarme contra ladrões, por isso não foi mencionado o estágio de projeto do algoritmo.

A etapa final do processo de projeto é projetar um sistema de escalonamento que assegure que um processo sempre será escalonado para atender aos seus prazos de conclusão. Nesse exemplo, os prazos não são rígidos. As prioridades de processo devem ser organizadas, de modo que todos os processos de verificação de sensores tenham a mesma prioridade. O processo para manipular uma falha de energia deve ser um processo de nível de interrupção de prioridade mais alta. As prioridades dos processos que gerenciam o sistema de alarme devem ser as mesmas dos processos de sensores. Se o sistema foi implementado em Java, então seu sistema de *run-time* gerenciaria o escalonamento do processo.

O sistema de alarme contra roubo é um sistema de monitoramento, e não um sistema de controle, uma vez que não inclui atuadores que sejam diretamente afetados por valores de sen-

Figura 13.9 ▶

Arquitetura de processo de um sistema de controle de temperatura.



sores. Um exemplo de sistema de controle é o sistema de controle de aquecimento de um edifício. Esse sistema monitora sensores de temperatura em diferentes salas no edifício e desliga e liga um aquecedor, dependendo da temperatura real e da temperatura ajustada no termostato da sala. O termostato também controla o chaveamento da fornalha no sistema.

A arquitetura de processo desse sistema é mostrada na Figura 13.9. Fica claro que sua maneira geral é semelhante ao sistema de alarme contra roubo. O desenvolvimento posterior desse exemplo fica por conta do leitor, como um exercício.

13.4 Sistemas de aquisição de dados

Os sistemas de aquisição de dados são uma outra classe de sistemas em tempo real que, geralmente, têm como base um modelo de arquitetura geral. Eles coletam dados de sensores para subseqüente processamento e análise.

Para ilustrar essa classe de sistema, considere o modelo de sistema mostrado na Figura 13.10, que representa um sistema que coleta dados de sensores monitorando o fluxo de nêutrons em um reator nuclear. Os dados do sensor são colocados em um buffer, a partir do qual são extraídos e processados, e o nível médio de fluxo é exibido no display do operador.

Cada sensor tem um processo associado que converte o nível de fluxo de entrada analógica para um sinal digital. Ele passa esse nível de fluxo, com o identificador do sensor, para o buffer de dados do sensor. O processo responsável pelo processamento de dados capta os dados desse buffer, faz o processamento e os transmite para um processo de display para a saída em um console do operador.

Em sistemas de tempo real que envolvem a aquisição e o processamento de dados, as velocidades de execução e os períodos do processo de aquisição e de processamento podem estar não sincronizados. Quando um processamento significativo é necessário, a aquisição de dados pode ser mais rápida do que o processamento de dados. Se somente cálculos simples precisam ser realizados, o processamento pode ser mais rápido do que a aquisição de dados.

Para minimizar essas diferenças de velocidade, a maioria dos sistemas de aquisição de dados controla a entrada de dados utilizando um buffer circular ou em anel. O processo que produz os dados (produtor) adiciona informações a esse buffer e o processo que utiliza os dados (consumidor) obtém informações do buffer (Figura 13.11).

Obviamente, uma exclusão mútua deve ser implementada, a fim de impedir que os processos produtor e consumidor acessem simultaneamente o mesmo elemento no buffer. O sistema deve também assegurar que o produtor não tente adicionar informações a um buffer cheio e que o consumidor não obtenha informações de um buffer vazio.

Na Figura 13.12, é mostrada uma possível implementação em Java de um buffer. Os valores no buffer são do tipo SensorRecord e existem duas operações definidas, chamadas get e put. A operação get obtém um item do buffer e a operação put adiciona um item ao buffer.

Figura 13.10 ▶

Arquitetura de um sistema de monitoramento de fluxo.

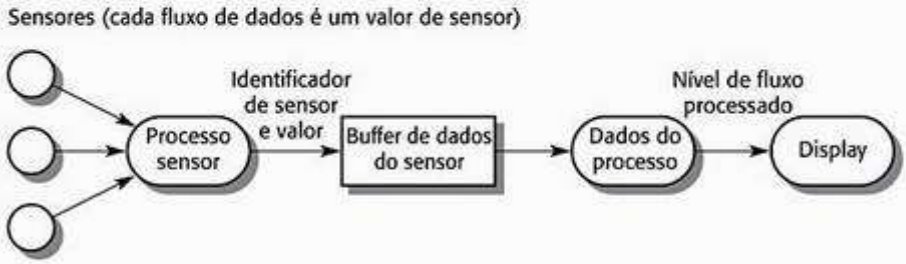
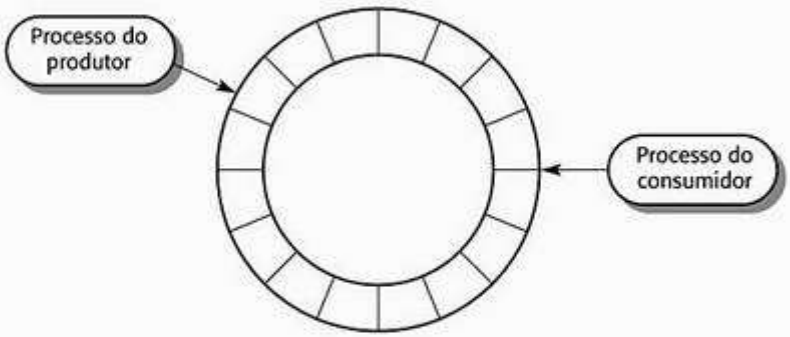


Figura 13.11

Um buffer em anel para aquisição de dados.



O construtor do buffer estabelece o tamanho, quando os objetos do tipo CircularBuffer são declarados.

O modificador synchronized associado aos métodos get e put indica que esses métodos não devem operar concorrentemente. Quando um desses métodos é chamado, o sistema de *run-time* obtém um bloqueio na instância de objeto, para garantir que o outro método não possa ser chamado ao mesmo tempo e, assim, manipular a mesma entrada no buffer. As chamadas aos métodos wait e notify, dentro desses métodos, garantem que as entradas não possam ser colocadas em um buffer cheio ou retiradas de um buffer vazio. O método wait acarreta que o *thread*

Figura 13.12

Uma implementação em Java de um buffer em anel.

```
class CircularBuffer
{
  int bufsize ;
  SensorRecord [ ] store ;
  int numberOfEntries = 0 ;
  int front = 0, back = 0 ;

  CircularBuffer (int n) {
      bufsize = n ;
      store = new SensorRecord [bufsize] ;
  } // CircularBuffer

  synchronized void put (SensorRecord rec) throws InterruptedException
  {
      if (numberOfEntries = = bufsize)
          wait () ;
      store [back] = new SensorRecord (rec.sensorId, rec.sensorVal) ;
      back = back + 1 ;
      if (back = = bufsize)
          back = 0 ;
      numberOfEntries = numberOfEntries + 1 ;
      notify () ;
  } // put

  synchronized SensorRecord get () throws InterruptedException
  {
      SensorRecord result = new SensorRecord (– 1, – 1) ;
      if (numberOfEntries == 0)
              wait () ;
      result = store [front] ;
      front = front + 1 ;
      if (front == bufsize)
          front = 0 ;
      numberOfEntries = numberOfEntries – 1 ;
      notify () ;
      return result ;
  } // get
} // CircularBuffer
```

que chama faça sua própria suspensão até que outro *thread* diga para parar de esperar chamando o método notify. Quando wait é chamado, o bloqueio no objeto protegido é liberado. O método notify ativa um dos *threads* que estiverem esperando e provoca o reinício da operação.

PONTOS-CHAVE

- Um sistema em tempo real é um sistema de software que precisa responder a eventos em tempo real. Sua precisão não depende apenas dos resultados que ele produz, mas também do momento em que esses resultados são produzidos.
- Um modelo geral para a arquitetura de sistemas de tempo real envolve associar um processo com cada classe de dispositivo de sensor e de atuador. Outros processos de coordenação também podem ser requeridos.
- O projeto de arquitetura de um sistema de tempo real geralmente envolve organizar o sistema como um conjunto de processos concorrentes que interagem entre si.
- Um executivo em tempo real é responsável pelo gerenciamento de processo e recursos. Ele sempre inclui um escalonador, que é o componente responsável por decidir qual processo deve ser escalonado para ser executado. As decisões de escalonamento são tomadas utilizando-se prioridades de processo.
- Os sistemas de monitoramento e controle varrem periodicamente um conjunto de sensores que capturam informações do ambiente de sistema. Eles tomam ações, dependendo das leituras do sensor, emitindo comandos para os atuadores.
- Os sistemas de aquisição de dados são geralmente organizados de acordo com um modelo produtor–consumidor. O processo produtor coloca os dados em um buffer circular, em que são consumidos pelo processo consumidor. O buffer é também implementado como um processo, de modo que os conflitos entre o produtor e o consumidor sejam eliminados.
- Embora a linguagem Java tenha recursos de apoio à concorrência, esta não é uma linguagem adequada para o desenvolvimento de sistemas de tempo real em que o tempo é fundamental. Faltam recursos para controlar a execução e é impossível analisar o comportamento relacionado ao tempo nos programas em Java.

LEITURAS SUGERIDAS

Doing hard time: developing real-time systems with UML, objects frameworks and patterns. Esse livro discute como as técnicas orientadas a objetos podem ser utilizadas no projeto de sistemas em tempo real. À medida que as velocidades de hardware aumentam, essa abordagem se torna incrivelmente viável para os projetos de sistemas em tempo real. (B.P. Douglass, 1999, Addison Wesley.)

"Adding real-time capabilities to Java". Uma boa descrição de por que a linguagem Java não é adequada para o desenvolvimento de sistemas de tempo real. (K. Nilsen, *Comm. ACM*, 41(6), jun. 1998.)

Real-time systems and programming languages, 2. ed. Um texto excelente e abrangente que fornece ampla cobertura de todos os aspectos dos sistemas em tempo real. (A. Burns e A. Wellings, 1997, Addison Wesley.)

Advances in real-time systems. Esse é um excelente volume tutorial do IEEE, que aborda vários aspectos do projeto e da implementação de sistemas de tempo real. Alguns capítulos pressupõem que o leitor esteja familiarizado com o assunto, mas outros apresentam boas introduções. (J.A. Stankovic e K. Ramamritham, 1993, IEEE Press.)

EXERCÍCIOS

13.1 Explique, com exemplos, por que os sistemas de tempo real geralmente têm de ser implementados com a utilização de processos concorrentes.

13.2 Explique por que uma abordagem orientada a objetos para o desenvolvimento de software pode não ser adequada para os sistemas de tempo real.

13.3 Desenhe modelos de máquina de estado do software de controle para os seguintes sistemas:

- Uma máquina de lavar roupa automática que tem diferentes programas para diferentes tipos de roupas.
- O software para um aparelho de reprodução de CDs.
- Uma secretária eletrônica que registra as mensagens recebidas e exibe o número de mensagens aceitas em um visor. O sistema deve permitir que o proprietário do telefone possa telefonar, digitar uma seqüência de números (identificados como tons) e ouvir as mensagens recebidas reproduzidas pelo telefone.
- Uma máquina de venda de bebidas que pode fornecer café com ou sem leite e açúcar. O usuário deposita uma moeda e faz sua escolha pressionando um botão na máquina. Essa ação provoca a saída de uma xícara com café em pó. O usuário coloca a xícara sob uma torneira, pressiona outro botão e a água quente é fornecida.

13.4 Utilizando as técnicas de projeto de sistema de tempo real, discutidas neste capítulo, refaça o projeto do sistema de coleta de dados da estação meteorológica, abordado no Capítulo 12, como um sistema de estímulo/resposta.

13.5 Projete uma arquitetura de processo para um sistema de monitoramento ambiental, que coleta dados a partir de um conjunto de sensores de qualidade do ar, situado nos arredores de uma cidade. Existem cinco mil sensores organizados em cem regiões. Cada sensor deve ser interrogado quatro vezes por segundo. Quando mais de 30 por cento dos sensores, em determinada região, indicarem que a qualidade do ar está abaixo do nível aceitável, luzes locais de aviso serão ativadas. Todos os sensores retornam as leituras para um computador central, que gera relatórios, a cada 15 minutos, sobre a qualidade do ar na cidade.

13.6 Discuta as vantagens e as desvantagens de Java como uma linguagem de programação para os sistemas de tempo real.

13.7 Um sistema de proteção de trens aciona automaticamente os freios, caso o limite de velocidade para um trecho seja excedido ou caso o trem entre em um trecho de trilhos que esteja, no momento, assinalado com uma luz vermelha (isto é, o trem não deveria ter entrado nesse trecho). Os detalhes são mostrados na Figura 13.13. Identifique os estímulos que devem ser processados pelo sistema de controle a bordo do trem e as respostas associadas a esses estímulos.

Figura 13.13 ▶ Descrição do sistema de proteção de trens.

- O sistema recebe informações sobre o limite de velocidade de um trecho, a partir de um transmissor da via férrea que, continuamente, transmite o identificador do trecho e seu limite de velocidade. O mesmo transmissor também transmite informações sobre o status do sinal que controla o trecho da linha. O tempo requerido para transmitir as informações de sinais e de controle do trecho é de 50 milissegundos.
- O trem pode receber informações do transmissor da via férrea quando estiver a 10 metros de um transmissor.
- A velocidade máxima do trem é de 180 km/h.
- Os sensores do trem fornecem informações sobre sua velocidade atual (atualizadas a cada 250 milissegundos) e sobre o status dos freios (atualizadas a cada 100 milissegundos).
- Se a velocidade do trem exceder o limite definido para o trecho atual em mais de 5 km/h, será emitido um aviso na cabine do maquinista. Se o trem exceder o limite de velocidade atual do trecho em mais de 10 km/h, os freios serão automaticamente acionados, até que a velocidade caia para a velocidade-limite do trecho. Os freios devem ser acionados dentro de 100 milissegundos, a partir do momento em que o excesso de velocidade for detectado.
- Se o trem entrar em um trecho assinalado com luz vermelha, o sistema de proteção acionará os freios e reduzirá a velocidade para zero. Os freios do trem devem ser acionados dentro de 100 milissegundos, a partir do momento em que o sinal de luz vermelha for recebido.
- O sistema atualiza continuamente um display de status na cabine do maquinista.

13.8 Sugira uma possível arquitetura de processo para esse sistema.

13.9 Se um processo periódico do sistema de proteção a bordo do trem for utilizado para coletar dados do transmissor da via férrea, qual será sua freqüência de escalonamento de tempo para assegurar que o sistema, com certeza, colete informações do transmissor? Explique como você chegou a essa resposta.

13.10 Pediram que você trabalhasse em um projeto de desenvolvimento de tempo real para um aplicativo militar, mas você não tem nenhuma experiência anterior em projetos dessa área. Discuta o que você, como engenheiro de software profissional, deve fazer, antes de começar a trabalhar no projeto.

14 Projeto com reuso

Objetivos

O objetivo deste capítulo é explicar como o reuso de software já existente pode ser incorporado no processo de projeto de sistemas. Depois de ler este capítulo, você poderá:

- compreender os benefícios do reuso de componentes de software e alguns problemas que podem surgir com o reuso;
- compreender diferentes tipos de componentes que podem ser reutilizados e os processos de projeto destinados ao reuso;
- ter uma noção das famílias de aplicações e compreender como elas são uma maneira eficiente de reutilizar software, e
- compreender como os padrões são abstrações de alto nível que promovem o reuso de projetos no desenvolvimento orientado a objetos.

Sumário

14.1 Desenvolvimento baseado em componentes
14.2 Famílias de aplicações
14.3 Padrões de projeto

O processo de projeto, na maioria das disciplinas de engenharia, tem como base o reuso de componentes. Engenheiros mecânicos e elétricos não especificam um projeto em que cada componente tenha de ser fabricado especialmente. Eles baseiam o projeto em componentes que já foram experimentados e testados em outros sistemas. Esses componentes não são apenas pequenos componentes, como flanges e válvulas, mas incluem subsistemas principais, como motores, condensadores ou turbinas.

Atualmente, em geral, é aceito o princípio de que precisamos de uma abordagem comparável para o desenvolvimento de software. O software deve ser considerado um ativo e o reuso desses ativos é essencial para aumentar o retorno de seus custos de desenvolvimento. Os requisitos relativos a menores custos de produção e manutenção de software, à maior rapidez na entrega de sistemas e ao aumento da qualidade só podem ser atendidos pelo reuso generalizado e sistemático de software.

O reuso de software deve ser considerado durante o projeto de engenharia ou de requisitos. O reuso oportuno é possível durante a programação, quando se descobrem componentes que por acaso atendam a um requisito. Contudo, o reuso sistemático requer um processo de projeto que considere como os projetos existentes podem ser reutilizados e que organize explicitamente o projeto em torno de componentes de software disponíveis.

A engenharia de software baseado no reuso é uma abordagem para o desenvolvimento que tenta maximizar o reuso do software já existente. As unidades de software que são reutilizadas podem ser de tamanhos radicalmente diferentes. Por exemplo:

1. *O reuso de sistemas de aplicações* Todo o sistema de aplicações pode ser reutilizado pela sua incorporação, sem mudança, em outros sistemas (por exemplo, o reuso de produtos COTS, Seção 14.1.2) ou pelo desenvolvimento de famílias de aplicações, que podem ser executadas em plataformas diferentes ou ser especializadas para as necessidades de determinados clientes (Seção 14.2).

2. *Reuso de componentes* Os componentes de uma aplicação, que variam em tamanho incluindo desde subsistemas até objetos isolados, podem ser reutilizados. Por exemplo, um sistema de combinação de padrões, desenvolvido como parte de um sistema de processamento de texto, pode ser reutilizado em um sistema de gerenciamento de banco de dados. Esse tópico será abordado na Seção 14.3.

3. *Reuso de funções* Os componentes de software que implementam uma única função, como uma função matemática, podem ser reutilizados. Esse tipo de reuso com base em bibliotecas de padrões tem sido comum nos últimos 40 anos.

O reuso de sistemas de aplicações tem sido amplamente praticado há muitos anos, à medida que as empresas de software implementam seus sistemas em uma série de máquinas e fazem ajustes para diferentes ambientes. Também o reuso de funções está bem estabelecido por meio de bibliotecas de funções reutilizáveis, como gráficos e bibliotecas de matemática. Contudo, embora tenha havido interesse no reuso de componentes desde o início da década de 80, foi somente nos últimos anos que ele se tornou aceito como uma abordagem prática para o desenvolvimento de sistemas de software.

Uma vantagem óbvia do reuso de software é que os custos gerais de desenvolvimento ficam reduzidos. Menos componentes de software têm de ser especificados, projetados, implementados e validados. Contudo, a redução de custos é apenas uma vantagem em potencial do reuso, pois existe uma série de outras vantagens no reuso de ativos de software, como mostra a Figura 14.1.

Existem três requisitos fundamentais para o projeto e o desenvolvimento de software baseado em reuso:

1. Deve ser possível encontrar componentes reutilizáveis apropriados. As organizações necessitam de uma base de componentes reutilizáveis adequadamente catalogados e documentados. Deve ser fácil encontrar componentes nesse catálogo, se ele existir.

Benefícios	Explicação
Maior confiabilidade	Os componentes reutilizados que são empregados nos sistemas em operação devem ser mais confiáveis do que os componentes novos. Eles já foram experimentados e testados em diferentes ambientes. Os defeitos de projeto e de implementação são descobertas e eliminadas no uso inicial dos componentes, reduzindo, assim, o número de falhas quando o componente é reutilizado.
Redução dos riscos de processo	Se recorrermos a um componente já existente, serão menores as incertezas sobre os custos relacionados ao reuso desse componente do que sobre custos de desenvolvimento. Esse é um fator importante para o gerenciamento de projetos, pois reduz as incertezas nas estimativas de custos de projeto. É particularmente verdadeiro quando reutilizamos componentes grandes, como subsistemas.
Uso efetivo de especialistas	Em vez de os especialistas em aplicações fazerem o mesmo trabalho em diferentes projetos, eles podem desenvolver componentes reutilizáveis, que englobam seu conhecimento.
Conformidade com padrões	Alguns padrões, como os padrões de interface com o usuário, podem ser implementados como um conjunto de componentes-padrão. Por exemplo, os componentes reutilizáveis podem ser desenvolvidos para implementar menus em uma interface com o usuário. Todas as aplicações apresentam o mesmo formato de menu para os usuários. O uso de interfaces-padrão com o usuário melhora a confiabilidade, uma vez que os usuários possivelmente cometem menos enganos quando utilizam uma interface familiar.
Desenvolvimento acelerado	De modo geral, é mais importante fornecer um sistema para o mercado o mais rápido possível do que se prender aos custos gerais de desenvolvimento. O reuso de componentes acelera a produção, porque o tempo de desenvolvimento e o de validação devem ser reduzidos.

Figura 14.1 ▲
Benefícios do reuso de software.

2. O responsável pelo reuso dos componentes precisa ter certeza de que os componentes se comportarão como especificado e de que serão confiáveis. Idealmente, todos os componentes no catálogo de uma organização devem estar certificados, a fim de confirmar que atingiram determinados padrões de qualidade. Na prática, essa situação não é realista e as pessoas em uma empresa aprendem de maneira informal sobre componentes confiáveis.

3. Os componentes devem ter a documentação associada para ajudar o usuário a compreendê-los e adaptá-los a uma nova aplicação. A documentação deve incluir informações sobre onde os componentes foram reutilizados e sobre quaisquer problemas de reuso que tenham sido encontrados.

O uso bem-sucedido de Visual Basic e Visual C++ com componentes e de Java com JavaBeans demonstrou o valor do reuso. A engenharia de software baseada em componentes (Szyperski, 1998) está se tornando amplamente aceita como uma abordagem eficaz em termos de custos para o desenvolvimento de software.

Contudo, existem alguns custos e problemas associados com o reuso (Figura 14.2), que podem inibir a introdução desse método e significar que as reduções no custo total de desenvolvimento, com o reuso, serão menores do que as previstas.

Essas dificuldades significam que o reuso sistemático não acontece por acaso, mas tem de ser planejado e introduzido por meio de um programa de reuso empregado por toda a organização. Há muitos anos, esse princípio já foi reconhecido no Japão (Matsumoto, 1984), onde o reuso é parte integrante da abordagem de fábrica japonesa para o desenvolvimento de software (Cusamano, 1989). Empresas como a Hewlett-Packard têm sido também muito bem-sucedidas em seus programas de reuso (Griss e Wosser, 1995) e sua experiência foi incorporada em um livro geral, escrito por Jacobson *et al.* (1997).

Uma alternativa para visão do reuso orientado a componentes é o ponto de vista do gerador. Nessa abordagem do reuso, o conhecimento reutilizável é obtido em um sistema gerador de programas, que pode ser programado em uma linguagem orientada a domínios. A descrição da aplicação especifica, de maneira abstrata, quais componentes reutilizáveis deverão ser

Problema	Explicação
Aumento nos custos de manutenção	Se o código-fonte do componente não estiver disponível, então os custos de manutenção poderão aumentar, uma vez que os elementos reutilizados no sistema podem ser tornar crescentemente incompatíveis com as mudanças do sistema.
Falta de ferramentas de apoio	Os conjuntos de ferramentas CASE não apóiam o desenvolvimento com reuso. Pode ser difícil ou impossível integrar essas ferramentas com um sistema de biblioteca de componentes. O processo de software assumido por essas ferramentas pode não levar em conta o reuso.
Síndrome do 'não-foi-inventado-aqui'	Alguns engenheiros de software às vezes preferem reescrever componentes, porque acreditam que podem fazer melhor que o componente reutilizável. Isso tem a ver em parte com confiança e em parte com o fato de que escrever um software original é visto como mais desafiador do que reutilizar o software de outras pessoas.
Manutenção de uma biblioteca de componentes	Implementar uma biblioteca de componentes e assegurar que os desenvolvedores de software utilizem essa biblioteca pode ser dispendioso. Nossas técnicas atuais de classificação, catalogação e recuperação de componentes de software são imaturas.
Encontrar e adaptar componentes reutilizáveis	Os componentes de software devem ser encontrados em uma biblioteca, compreendidos e, algumas vezes, adaptados, a fim de trabalharem em um novo ambiente. Os engenheiros precisam ter uma razoável certeza de poder encontrar um componente em uma biblioteca, antes de incluírem a rotina de busca de componente como parte de seu processo normal de desenvolvimento.

Figura 14.2 ▲
Problemas com o reuso.

utilizados, como eles devem ser combinados e qual é sua parametrização. Com essas informações, pode ser gerado um sistema de software operacional (Figura 14.3).

O reuso baseado em geradores só será possível quando as abstrações de domínio e seu mapeamento em código executável puderem ser definidos. Uma linguagem de domínio específico (como uma 4GL) é, então, utilizada para compor e controlar as abstrações de domínio. Entre as áreas em que essa abordagem tem sido bem-sucedida estão:

1. *Geradores de aplicação para processamento de dados corporativos* A entrada, nesse caso, pode ser uma 4GL, ou o esquema pode ser completamente interativo, no qual o usuário define telas e ações de processamento. A saída é um programa em uma linguagem como COBOL ou SQL.
2. *Geradores de analisador gramatical para processamento de linguagem* A entrada do gerador é um conjunto de regras gramaticais que descreve a linguagem a ser analisada e a saída é um analisador gramatical de linguagem.
3. *Geradores de código em ferramentas CASE* A entrada para esses geradores é um projeto de software e a saída é um programa que implementa o sistema projetado.

O reuso com base em geradores tem uma boa relação custo-benefício, mas depende de identificar abstrações estereotípicas de domínio. Isso tem sido possível nas áreas mencionadas anteriormente e, em menor extensão, em áreas como sistemas de comando e controle (O'Connor *et al.*, 1994). Sua vantagem primordial é que é mais fácil para os usuários finais desenvolver programas utilizando geradores, em comparação com outras abordagens com base em componentes para o reuso. Contudo, a necessidade de uma compreensão profunda dos conceitos de domínio da aplicação e dos modelos de domínio tem limitado a aplicabilidade dessa técnica.

Figura 14.3 ▶
Reuso com base em geradores.

14.1 Desenvolvimento baseado em componentes

O desenvolvimento baseado em componentes, ou a engenharia de software baseada em componentes, emergiu no final da década de 90 como uma abordagem baseada no reuso para o desenvolvimento de sistemas de software. Sua motivação foi a frustração de que o desenvolvimento orientado a objetos não tinha conduzido a um extensivo reuso, como originalmente foi sugerido. As classes de objetos individuais eram muito detalhadas e específicas e tinham de estar associadas a uma aplicação em tempo de compilação ou quando o sistema estivesse conectado. O conhecimento detalhado das classes era necessário para sua utilização, e isso, geralmente, significava que o código-fonte precisava estar disponível, apresentando problemas difíceis para a comercialização de componentes. Apesar das primeiras previsões otimistas, nenhum mercado significativo para os objetos individuais foi desenvolvido.

Os componentes são mais abstratos do que as classes de objetos e podem ser considerados provedores de serviços *stand-alone*. Quando um sistema precisa de algum serviço, ele chama um componente para fornecer esse serviço, sem se preocupar a respeito de onde esse componente está sendo executado ou com a linguagem de programação utilizada para desenvolver o componente. Por exemplo, um componente muito simples pode ser uma função matemática individual que calcula a raiz quadrada de um número. Quando um programa requer um cálculo de raiz quadrada, chama o componente que fornece esse cálculo. No outro extremo da escala, um sistema que precisa realizar algum cálculo aritmético pode chamar um componente de planilha de cálculo, que fornece um serviço de cálculo.

Visualizar um componente como um provedor de serviços enfatiza duas importantes características de um componente reutilizável:

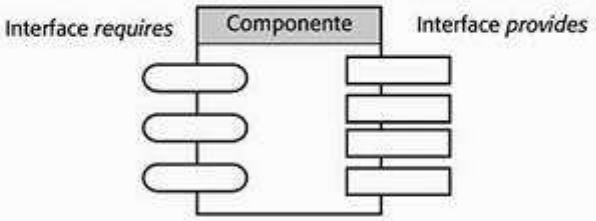
1. O componente é uma entidade executável independente. O código-fonte não está disponível, de modo que o componente não é compilado com outros componentes do sistema.
2. Os componentes publicam sua interface e todas as interações são feitas por meio dessa interface. A interface do componente é expressa em termos de operações parametrizadas e seu estado interno nunca é exposto.

Os componentes são definidos por suas interfaces e, nos casos mais genéricos, podem ser imaginados como duas interfaces relacionadas, como mostra a Figura 14.4:

- Uma interface denominada *provides* ('fornece'), que define os serviços fornecidos pelo componente.
- Uma interface denominada *requires* ('requer'), que especifica que serviços devem estar disponíveis a partir do sistema que está utilizando o componente. Se essas interfaces não forem fornecidas, o componente não funcionará.

Por exemplo, considere um componente que forneça serviços de impressão, como ilustra a Figura 14.5. Nesse caso, os serviços fornecidos se destinam a imprimir um documento, descobrir o estado da fila associada com determinada impressora, registrar e cancelar o regis-

Figura 14.4 ▶
Interfaces de componentes.



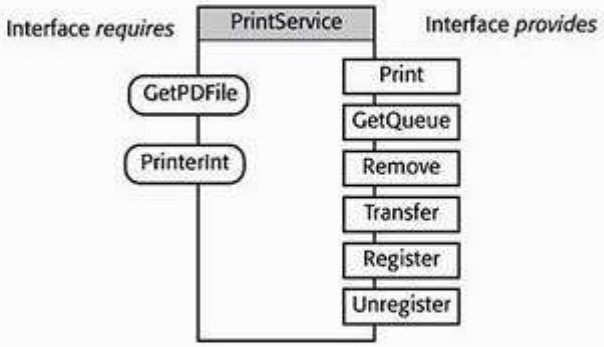


Figura 14.5 ▶
Um componente de serviços de impressão.

tro de uma impressora com o componente de serviços de impressão, mover um serviço de impressão de uma impressora para outra e remover um serviço de uma fila de impressão. Entre os requisitos para esse componente estão a plataforma subjacente, que deve fornecer um serviço chamado GetPDFile para recuperar o arquivo de descrição da impressora para um tipo de impressora, e um serviço chamado PrinterInt, que transfere comandos para uma impressora especificada.

Os componentes podem existir em diferentes níveis de abstração, desde uma simples subrotina de biblioteca até uma aplicação completa, como o Excel, da Microsoft. Meyer (1999) identifica cinco diferentes níveis de abstração:

1. *Abstração funcional* O componente implementa uma única função, como uma função matemática. Essencialmente, a interface *provides* é a própria função.
2. *Agrupamentos casuais* O componente é uma coleção de entidades inadequadamente relacionadas, que podem ser declarações de dados, funções, entre outras. A interface *provides* consiste em nomes de todas as entidades do agrupamento.
3. *Abstrações de dados* O componente representa uma abstração de dados ou classe em uma linguagem orientada a objetos. A interface *provides* consiste em operações para criar, modificar e acessar a abstração de dados.
4. *Abstrações em clusters* O componente é um grupo de classes relacionadas que trabalham em conjunto. Elas são chamadas, às vezes, de *framework*. A interface *provides* é a composição de todas as interfaces *provides* dos objetos que constituem o *framework*. Será discutido sobre o *framework* na Seção 14.1.1.
5. *Abstração de sistema* O componente é um sistema inteiramente autocontido. Reutilizar abstrações de nível de sistema é, às vezes, chamado de reuso de produtos COTS. A interface *provides* é chamada de API (*application programming interface* – interface de programação de aplicações), que é definida para permitir que os programas acessem os comandos e as operações do sistema. O reuso de produtos COTS será discutido na Seção 14.1.2.

O desenvolvimento orientado a componentes pode ser integrado em um processo de desenvolvimento de sistemas, incorporando uma atividade específica de reuso, como mostra a Figura 14.6. O projetista de sistemas completa um projeto de alto nível e as especificações dos componentes desse projeto. Essas especificações são utilizadas para encontrar componentes para o reuso. Elas podem ser incorporadas no nível de arquitetura ou em níveis mais detalhados do projeto.

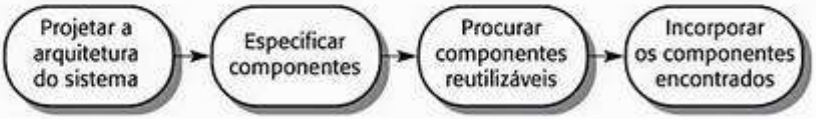


Figura 14.6 ▶
Um processo de reuso 'oportunista'.

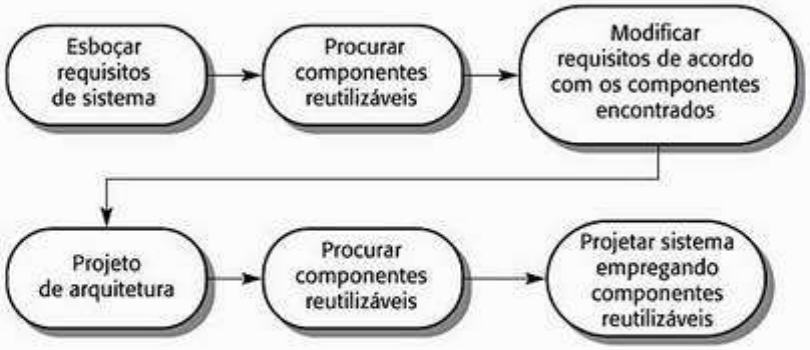


Figura 14.7 Desenvolvimento com reuso.

Embora essa abordagem possa resultar no reuso significativo, ela contrasta com a abordagem adotada em outras disciplinas da engenharia em que o reuso orienta o processo de projeto. Em vez de projetar e, depois, procurar componentes reutilizáveis, os engenheiros, primeiramente, procuram componentes reutilizáveis. Eles baseiam seu projeto nos componentes que estão disponíveis (Figura 14.7).

No desenvolvimento orientado ao reuso, os requisitos de sistema são modificados de acordo com os componentes reutilizáveis disponíveis. O projeto também se baseia nos componentes existentes. Naturalmente, isso significa que é possível que haja uma conciliação dos requisitos. O projeto pode ser menos eficiente do que um projeto de propósito especial. Contudo, os menores custos de desenvolvimento, a entrega mais rápida do sistema e o aumento da confiabilidade do sistema devem compensar esse aspecto.

O processo de implementação do sistema com o uso de componentes é, em geral, um processo de prototipação ou um processo de desenvolvimento incremental. Uma linguagem de programação-padrão, como Java, pode ser utilizada com componentes em uma biblioteca referenciada pelo programa. Como alternativa (e mais comumente), uma linguagem de *scripting*, que é especificamente projetada para integrar componentes reutilizáveis, é utilizada para possibilitar o rápido desenvolvimento do programa.

A primeira linguagem de *scripting* proposta para a integração de componentes reutilizáveis foi a shell do Unix (Bourne, 1978). Desde então, uma série de linguagens de *scripting*, como Visual Basic e TCL/TK, foi desenvolvida. Ousterhout (1998) discute os benefícios da linguagem de *scripting*, inclusive sua natureza sem tipo, e o fato de que elas são interpretadas, em vez de compiladas.

Como foi discutido no Capítulo 8 ("Prototipação de software"), os usuários de Visual Basic constroem aplicações visualmente, definindo a interface com a seleção de tipos de componentes de interface, como por exemplo menus, campos de entrada de texto e caixas de verificação, e posicionando-os na tela. Componentes individuais ou seqüências de componentes que realizam algum cálculo podem então ser associados às entidades da interface.

Talvez a principal dificuldade do desenvolvimento com base em componentes seja o problema da manutenção e da evolução. De modo geral, o código-fonte dos componentes não está disponível e, à medida que os requisitos da aplicação se modificam, pode ser impossível alterar os componentes para refletir esses requisitos. A opção de modificar os requisitos nesse estágio, para adequá-los aos componentes disponíveis, em geral, não é possível, uma vez que a aplicação já está em uso. Portanto, é necessário maior trabalho para reutilizar os componentes e, ao longo do tempo, isso leva ao aumento dos custos de manutenção. Contudo, considerando que o desenvolvimento baseado em componentes permite a entrega mais rápida do software, as organizações podem preferir aceitar esses custos em longo prazo.

14.1.1 *Frameworks* de aplicações

Os primeiros proponentes do desenvolvimento orientado a objetos sugeriam que os objetos eram a mais adequada abstração para o reuso. Contudo, como foi explicado na seção anterior, os objetos são, freqüentemente, de alta granularidade e são específicos para uma

aplicação em particular. Em vez disso, tornou-se claro que o reuso é mais compatível em um processo de desenvolvimento orientado a objetos, por meio de abstrações de maior granularidade, chamadas de *frameworks*.

Os *frameworks* (ou *frameworks* de aplicações) são um projeto de subsistema constituído de um conjunto de classes abstratas e concretas e da interface entre elas (Wirfs-Brock e Johnson, 1990). Detalhes específicos do subsistema de aplicações são implementados com o acréscimo de componentes e o fornecimento da implementação concreta das classes abstratas nos *frameworks*. Os *frameworks* raramente são aplicações propriamente ditas. As aplicações, normalmente, são construídas pela integração de diversos *frameworks*.

Fayad e Schmidt (1997) identificaram três classes de *frameworks*:

1. Frameworks *de infra-estrutura de sistema* Esses *frameworks* são compatíveis com o desenvolvimento das infra-estruturas de sistemas, como comunicações, interfaces com o usuário e compiladores (Schmidt, 1997).

2. Frameworks *de integração com middleware* Consistem em um conjunto de classes de objetos-padrão e associadas, que aceitam a comunicação de componentes e a troca de informações. Entre os exemplos desse tipo de *framework* estão CORBA, COM e DCOM, da Microsoft, e Java Beans (Orfali e Harkey, 1998). Esse tipo de *framework* já foi descrito no Capítulo 11, em que foram abordadas as arquiteturas de objetos distribuídos.

3. Frameworks *de aplicações corporativos* Eles se ocupam de domínios específicos de aplicações, como telecomunicações ou sistemas financeiros (Baumer *et al.*, 1997). Eles incluem o conhecimento de domínio de aplicações e são compatíveis com o desenvolvimento de aplicações para o usuário final. Eles são relacionados a famílias de aplicações, cuja estrutura foi discutida na Seção 14.2. Contudo, são normalmente mais abstratos e, assim, possibilitam a criação de uma gama mais ampla de aplicações.

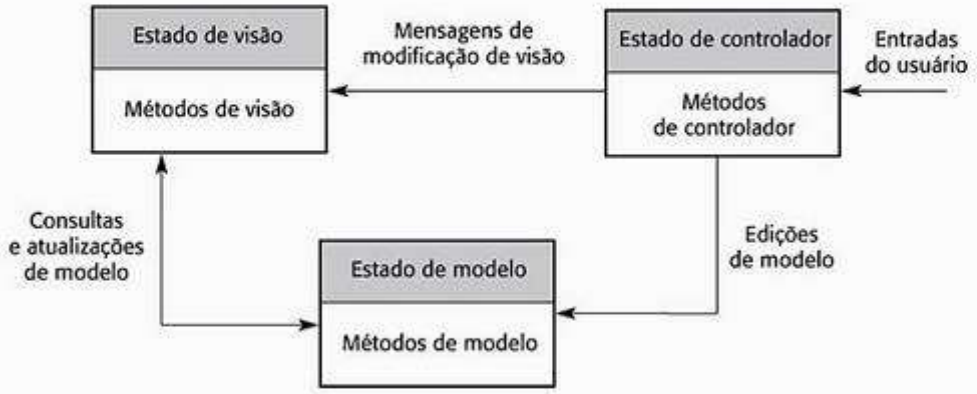
Como o nome sugere, um *framework* é uma estrutura genérica que pode ser ampliada para criar um subsistema ou uma aplicação mais específica. Ampliar o *framework* pode envolver o acréscimo de classes concretas, que herdam operações de classes abstratas no *framework*. Além disso, os retornos de chamada podem ser definidos. Esses são métodos chamados em resposta a eventos reconhecidos pelo *framework*.

No momento em que este livro foi escrito, os *frameworks* mais bem desenvolvidos eram os *frameworks* de infra-estrutura de sistemas, especialmente aqueles dedicados a interfaces gráficas com o usuário. Os *frameworks* de aplicações corporativos estão começando a surgir (Codenie *et al.*, 1997), mas ainda estamos tentando compreender as estruturas mais eficazes e as organizações para esses gerenciamentos.

Um dos *frameworks* mais conhecidos e amplamente utilizados para projeto GUI é o *framework* 'Modelo-Visão-Controlador' (MVC) (Figura 14.8). Ele foi originalmente proposto na década de 80, como uma abordagem ao projeto da GUI, que permitia múltiplas apresentações de um objeto e estilos separados de interação com cada uma dessas apresentações. O *framework* MVC

Figura 14.8 ▶

O *framework* 'Modelo-Visão-Controlador'.



aceita a apresentação de dados de diferentes maneiras (veja a Figura 14.13) e a interação separada com cada uma dessas apresentações. Quando os dados são modificados por meio de uma das apresentações, todas as outras são atualizadas.

Os *frameworks* são, freqüentemente, ocorrências de uma série de padrões, como será discutido na Seção 14.2. Por exemplo, o *framework* MVC inclui o padrão *Observer*, que é descrito na Figura 14.12, o padrão *Strategy*, que se ocupa de atualizar o modelo, o padrão *Composite* e uma série de outros padrões que são discutidos por Gamma *et al.* (1995).

As aplicações que são construídas com o uso de *frameworks* podem ser a base para outras oportunidades de reuso, por meio do conceito de famílias de aplicações, como será discutido na Seção 14.3. Por eles serem construídos com o uso de um *framework*, modificar os membros da família para criar novos membros é uma tarefa simplificada.

O problema fundamental com os *frameworks* é sua inerente complexidade e o tempo que leva para se aprender a utilizá-los. Vários meses podem ser necessários para compreender inteiramente um *framework*, e assim é provável que, em grandes organizações, alguns engenheiros de software tenham de se especializar em *frameworks*. Não há dúvida nenhuma de que essa é uma abordagem eficaz do reuso, mas o alto custo de introdução limita seu uso generalizado.

14.1.2 Reuso de produtos COTS

O termo COTS (produtos de prateleira) pode, em princípio, se aplicar a qualquer componente oferecido por um terceiro (fabricante). Contudo, ele é normalmente mais utilizado para se referir a produtos de software de sistema. Por isso, refiro-me aos sistemas COTS ou, algumas vezes, apenas COTS. Em geral, dá-se preferência a sistemas COTS. Como a funcionalidade oferecida por esses sistemas é muito mais ampla do que a de componentes mais especializados, o potencial de retorno pelo reuso aumenta.

Naturalmente, alguns tipos de sistemas COTS têm sido utilizados por muitos anos. Os sistemas de bancos de dados talvez sejam o melhor exemplo disso. Poucos desenvolvedores considerariam implementar seu próprio sistema de gerenciamento de bancos de dados. Contudo, até bem recentemente, havia apenas uns poucos sistemas de grande porte, como os sistemas de gerenciamento de bancos de dados e monitores de teleprocessamento, que eram reutilizados de maneira rotineira.

Novas abordagens para o projeto de sistemas, que permitem o acesso a programas de funções de sistema, atualmente significam que criar grandes sistemas, como sistemas de comércio eletrônico (*e-commerce*) pela integração de uma gama de sistemas COTS, deve sempre ser considerado uma séria opção de projeto. Devido à funcionalidade que esses sistemas oferecem, é possível reduzir muito os custos e os prazos de entrega, em comparação com o desenvolvimento de um software novo. Além disso, os riscos podem ser reduzidos, uma vez que o produto já está disponível e que os gerentes podem verificar se eles atendem a seus requisitos.

Em princípio, utilizar um sistema COTS em larga escala é o mesmo que utilizar qualquer outro componente mais específico. É preciso compreender as interfaces do sistema e utilizá-las exclusivamente para se comunicar com o componente. É necessário um compromisso entre requisitos específicos em relação ao desenvolvimento rápido e ao reuso; deve-se projetar uma arquitetura de sistemas que permita aos sistemas COTS operarem juntos.

Contudo, o fato de que esses produtos são usualmente sistemas grandes em si e que são freqüentemente vendidos como *stand-alone* separados apresenta alguns problemas adicionais. Boehm (1999) discute quatro problemas relacionados à integração de sistemas COTS:

1. *Falta de controle sobre a funcionalidade e o desempenho* Embora a interface publicada de um produto possa aparecer para oferecer os recursos requeridos, esses podem não estar adequadamente implementados ou podem operar de maneira inadequada. O produto pode ter operações ocultas que interferem em sua operação. Resolver esses problemas pode ser

uma prioridade para o integrador de produtos COTS, mas pode não ser uma preocupação real para o fabricante do produto. Os usuários podem simplesmente precisar encontrar um meio de contornar os problemas, se desejarem reutilizar o produto COTS.

2. *Problemas com a interoperabilidade de sistema COTS* Às vezes, é difícil conseguir que os produtos COTS trabalhem juntos, porque cada produto inclui diferentes suposições sobre como ele será utilizado. Garlan *et al.* (1995) relataram sua tentativa de integrar quatro produtos COTS e descobriram que três desses produtos eram baseados em eventos, mas cada um deles utilizava um modelo diferente de eventos e considerava que tinha acesso exclusivo à fila de eventos. Como conseqüência, o projeto exigiu um trabalho cinco vezes maior do que o originalmente previsto e a duração do cronograma foi, na verdade, de dois anos, em vez dos seis meses previstos.

3. *Nenhum controle sobre a evolução de sistema* Os fabricantes de produtos COTS tomam suas próprias decisões sobre as mudanças nos sistemas em resposta às pressões de mercado. Para os PCs, em particular, novas versões são sempre produzidas muito freqüentemente e podem não ser compatíveis com todas as versões anteriores. As novas versões podem ter funcionalidade adicional indesejável e as anteriores podem se tornar indisponíveis e deixar de ter suporte técnico.

4. *Suporte técnico dos fabricantes de COTS* O nível de suporte técnico disponível a partir dos fabricantes de COTS varia muito. Por serem sistemas de prateleira, o suporte técnico do fabricante é particularmente importante quando surgem problemas, porque os desenvolvedores não têm acesso ao código-fonte e à documentação detalhada do sistema. Embora os fabricantes possam se comprometer em fornecer suporte técnico, as mudanças de mercado e as circunstâncias econômicas podem tornar difícil que eles cumpram com seu compromisso. Por exemplo, um fabricante de sistemas COTS pode decidir descontinuar um produto por causa de sua demanda limitada ou ser absorvido por outra empresa, que não queira proporcionar suporte a todos os seus atuais produtos.

Naturalmente, é improvável que todos esses problemas surjam em todos os casos, mas na minha opinião pelo menos um deles deve ser esperado na maioria dos projetos de integração relacionados a COTS. Como conseqüência, a relação custo-benefício e os prazos de reuso de COTS talvez sejam menores do que possam parecer, a partir de uma análise otimista inicial.

Além disso, Boehm (1999) reconhece que, em muitos casos, o custo de manutenção do sistema e de sua evolução pode ser maior quando os produtos COTS são utilizados. Todas essas dificuldades mencionadas anteriormente são problemas do ciclo de vida, e eles não afetam somente o desenvolvimento inicial do sistema. À medida que o pessoal envolvido na manutenção do sistema se torna mais distante dos desenvolvedores originais do sistema, fica mais provável que dificuldades reais possam surgir com os produtos COTS integrados.

Apesar desses problemas, os benefícios do reuso dos produtos COTS são muito significativos, porque esses sistemas oferecem muita funcionalidade para o usuário. Meses e, algumas vezes, anos de esforço de implementação poderão ser economizados, se um sistema existente for reutilizado e os prazos de desenvolvimento do sistema puderem ser drasticamente reduzidos. Uma vez que a entrega rápida do sistema é o principal requisito de muitos sistemas, essa forma de reuso pode provavelmente se tornar cada vez mais praticada.

14.1.3 Desenvolvimento de componente para reuso

O processo ideal de desenvolvimento de componentes deve ter como base a experiência, em que componentes reutilizáveis são construídos a partir de componentes existentes, que já tenham sido reutilizados de maneira 'oportunista'. Utilizando o conhecimento sobre os problemas de reuso e as adaptações dos componentes necessários para possibilitar o reuso, pode ser criada uma versão de componentes mais genérica e, portanto, mais reutilizável.

Existem várias características de componentes que levam à possibilidade de reuso:

1. O componente deve refletir abstrações estáveis de domínio, que são conceitos fundamentais no domínio de aplicações, as quais se modificam aos poucos. Por exemplo, em um sistema bancário, as abstrações de domínio podem ser contas, correntistas, extratos de contas e assim por diante. Em um sistema de gerenciamento de hospital, as abstrações de domínio podem ser pacientes, tratamentos, enfermeiras, entre outras.

2. O componente deve ocultar a maneira como seu estado é representado e deve fornecer operações que permitam que o estado seja acessado e atualizado. Por exemplo, em um componente que representa uma conta bancária, deve haver operações de consulta ao saldo da conta, alterações no saldo, registro de transações na conta, entre outros.

3. O componente deve ser tão independente quanto possível. Idealmente, um componente deve se manter isolado, de maneira que não precise de qualquer outro componente para entrar em operação. Na prática, isso só é possível para os componentes muito simples; os componentes mais complexos, inevitavelmente, terão algumas dependências de outros componentes. O melhor será minimizar essa dependência, especialmente se forem dependências de componentes, como funções de sistemas operacionais, que podem se modificar.

4. Todas as exceções devem ser parte da interface do componente. Os componentes não devem manipular as próprias exceções, uma vez que diferentes aplicações terão diferentes requisitos para a manipulação de exceções. Em vez disso, o componente deve definir que exceções podem surgir e deve publicá-las como parte da interface. Por exemplo, um componente simples que implementa uma estrutura de dados em pilha deve detectar e publicar as exceções de overflow (transbordo de pilha) e underflow (pilha vazia) da pilha.

Em muitos sistemas existentes, há grandes segmentos de código que implementam abstrações de domínio, mas não podem ser utilizados diretamente como componentes. A razão disso é que eles não são empacotados de acordo com o modelo da Figura 14.4, com as interfaces *requires* e *provides* claramente definidas. Para tornar esses componentes reutilizáveis, de modo geral, é necessário construir uma capa. A capa oculta a complexidade do código subjacente e fornece uma interface para que os componentes externos acessem os serviços fornecidos.

Existe uma inevitável conciliação entre a facilidade de reuso e a de uso de um componente. Tornar um componente reutilizável implica fornecer uma interface muito genérica, com operações que apresentam diferentes maneiras pelas quais o componente pode ser utilizado. Tornar o componente reutilizável implica fornecer uma interface mínima e simples, que seja fácil de compreender. A facilidade do reuso aumenta a complexidade e, desse modo, reduz a facilidade de compreensão do componente. Portanto, é mais difícil para os engenheiros decidir quando e como reutilizar esse componente. Os projetistas de componentes reutilizáveis devem, conseqüentemente, encontrar um equilíbrio entre a generalidade e a facilidade de compreensão.

14.2 Famílias de aplicações

Uma das abordagens mais eficazes do reuso se baseia na noção das famílias de aplicações. Uma família de aplicações, ou linha de produtos, é um conjunto de aplicações relacionadas que têm uma arquitetura de domínio específico em comum, como foi discutido no Capítulo 10. Contudo, cada aplicação específica é especializada, de alguma maneira. O núcleo em comum da família de aplicações é reutilizado cada vez que uma nova aplicação é requerida. O novo desenvolvimento pode envolver a necessidade de escrever alguns componentes adicionais e de adaptar alguns componentes da aplicação, a fim de atender a novos requisitos.

Existem vários tipos de especialização de uma família de aplicações que podem ser desenvolvidos:

1. *Especialização de plataforma* quando diferentes versões da aplicação são desenvolvidas para diferentes plataformas. Por exemplo, as versões da aplicação podem existir para as plataformas Windows NT, Solaris e Linux. Nesse caso, a funcionalidade da aplicação permanece normalmente inalterada; somente são modificados os componentes que fazem interface com o hardware e o sistema operacional.
2. *Especialização da configuração* quando diferentes versões da aplicação são criadas para lidar com diferentes dispositivos periféricos. Por exemplo, um sistema para os serviços de emergência pode existir em diferentes versões, dependendo do tipo de sistema de rádio utilizado. Nesse caso, a funcionalidade pode variar para refletir a funcionalidade dos periféricos e os componentes que fazem interface com os periféricos devem ser modificados.
3. *Especialização funcional* quando diferentes versões da aplicação são criadas para clientes com diferentes requisitos. Por exemplo, um sistema de automação de bibliotecas pode ser modificado, dependendo de onde ele será utilizado: em uma biblioteca pública, em uma biblioteca de referência ou em uma biblioteca de universidade. Nesse caso, os componentes que implementam a funcionalidade podem ser modificados e novos componentes podem ser adicionados ao sistema.

Para ilustrar essa abordagem do reuso, considere a Figura 14.9, que mostra a arquitetura de um sistema para gerenciamento de estoque. Os sistemas de gerenciamento de estoque são utilizados pelas organizações para manter o controle de seus ativos. Portanto, uma companhia de fornecimento de energia pode ter um sistema de gerenciamento de estoque que controla todas as instalações fixas e os equipamentos instalados nesses locais. Uma universidade pode ter um sistema de gerenciamento de estoque para manter o controle do equipamento utilizado em seus laboratórios de ensino.

Os sistemas de gerenciamento de estoque, obviamente, variam, dependendo do tipo de recursos que estão sendo geridos e das informações que são exigidas para cada recurso. Por exemplo, o sistema da companhia de fornecimento de energia não precisará ter um recurso que permita que as localizações dos recursos sejam alteradas – a natureza desse sistema é que essas localizações são fixas. O sistema de controle de estoque de uma universidade, contudo, deve ser capaz de modificar a localização dos equipamentos, à medida que eles sejam transferidos de um laboratório para outro.

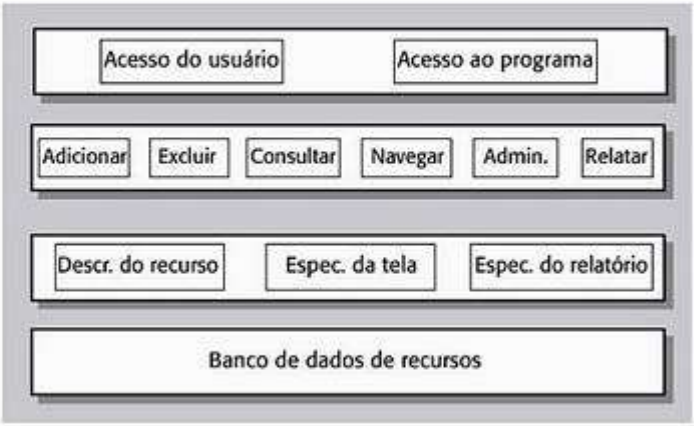
Contudo, esses sistemas devem fornecer um núcleo de recursos, como a capacidade de acrescentar um recurso ao estoque, a capacidade de excluir um recurso do estoque, a possibilidade de fazer consultas e navegar pelo estoque e a habilidade de criar relatórios. Por conseguinte, é possível projetar a arquitetura de sistemas de gerenciamento de estoque de maneira que ela se torne uma família de aplicações, com cada instância do sistema oferecendo compatibilidade com diferentes tipos de recursos.

Para permitir um reuso significativo nessas situações, a arquitetura de sistema deve ser projetada para separar os recursos essenciais do sistema das informações detalhadas sobre os recursos a serem gerenciados e o acesso do usuário a essas informações. Na Figura 14.9, isso é obtido pelo uso de uma arquitetura em camadas, em que uma camada específica inclui as descrições dos recursos, as telas a serem exibidas e os relatórios a serem produzidos. As camadas superiores do sistema utilizam essas descrições em suas operações e não incluem informações fixas sobre os recursos. Diferentes aplicações de gerenciamento de estoque podem ser produzidos, modificando essa camada de descrição.

Naturalmente, esse tipo de adequação pode ser conseguido em sistemas orientados a objetos, especificando um objeto de recurso abstrato e, então, utilizando os recursos de herança para especialização, dependendo do tipo de recurso que estiver sendo gerenciado. Provavelmente, isso conduziria a uma arquitetura bem diferente daquela mostrada na Figura 14.9. Contudo,

Figura 14.9

Um sistema genérico de gerenciamento de recursos.



para esse tipo de sistema, o desenvolvimento orientado a objetos pode não ser apropriado. Quando as aplicações dependem de grandes bancos de dados com milhões de registros, mas com relativamente poucos tipos de entidades lógicas, um sistema orientado a objetos pode ser muito menos eficiente do que um sistema construído com base em um banco de dados relacional. No momento em que este livro foi escrito, os bancos de dados comerciais orientados a objetos ainda eram relativamente lentos e não adequados para lidar com centenas de transações por segundo.

Além de desenvolver novos membros da família de aplicações, ao desenvolver descrições de novos recursos também é possível acrescentar novas funcionalidades ao sistema pela inclusão de novos módulos nas camadas do sistema. Para ilustrar essa alternativa, foi adaptado o sistema de gerenciamento de recursos, da Figura 14.9, para criar um sistema de biblioteca (Figura 14.10). Isso implica adicionar novos recursos para emitir e retornar recursos e permitir que os usuários sejam registrados no sistema. Isso está exemplificado à direita, na Figura 14.10. Como o acesso ao programa não é exigido, nesse caso o nível superior do sistema precisa apenas aceitar o acesso dos usuários aos recursos.

Em geral, desenvolver aplicações pela adaptação de uma versão genérica da aplicação significa que uma proporção muito alta do código da aplicação é reutilizada. Além disso, a experiência da aplicação é freqüentemente transferível de um sistema para outro, de modo que, quando os engenheiros de software se reúnem em uma equipe de desenvolvimento, seu processo de aprendizado é reduzido. O teste é simplificado, porque os testes para grandes partes da aplicação podem também ser reutilizados e o tempo total de desenvolvimento para a aplicação diminui.

A Figura 14.11 mostra as etapas envolvidas em adaptar uma família de aplicações para criar uma nova aplicação. Os detalhes do processo provavelmente poderão diferir de maneira radi-

Figura 14.10

Um sistema de biblioteca.

Figura 14.11 ▶

Desenvolvimento de um membro da família.

cal, dependendo do domínio da aplicação e do ambiente organizacional do sistema. As etapas envolvidas no processo genérico são:

1. *Levantar os requisitos dos* stakeholders Isso pode ter como base um processo normal de engenharia de requisitos. Contudo, como um sistema já existe, isso normalmente envolverá demonstrá-lo e testá-lo e expressar os requisitos à medida que as modificações forem requeridas.
2. *Escolher um membro da família mais adequado* Os requisitos são analisados e o membro da família mais adequado a esses requisitos é escolhido para modificação. Esse não precisa ser o sistema que foi demonstrado.
3. *Renegociar os requisitos* À medida que surgirem mais detalhes sobre as mudanças requeridas no sistema já existente e que o projeto for planejado, pode haver alguma renegociação dos requisitos, a fim de minimizar as mudanças necessárias.
4. *Adaptar o sistema existente* Novos módulos são desenvolvidos para o sistema existente, e os que existem são adaptados para atender aos novos requisitos.
5. *Entregar o novo membro da família* O novo membro da família de aplicações é entregue ao cliente. Nesse estágio, é preciso documentar as principais características, de modo que a aplicação possa ser utilizada como base para o desenvolvimento de outros sistemas, no futuro.

Quando se cria um novo membro de uma família de aplicações, talvez seja necessário chegar a uma conciliação entre o maior reuso possível da aplicação genérica e a satisfação dos requisitos detalhados dos *stakeholders*. Quanto mais detalhados forem os requisitos do sistema, menor a probabilidade de os componentes existentes atenderem a esses requisitos. Contudo, se os *stakeholders* se dispuserem a ser flexíveis e a limitar modificações requeridas para o sistema, este, em geral, poderá ser entregue mais rapidamente e por um custo mais baixo.

Com algumas exceções, as famílias de aplicações de modo geral derivam das aplicações existentes. Ou seja, uma organização desenvolve uma aplicação e, então, quando uma nova aplicação é requerida, ela utiliza a primeira aplicação como base para a nova. Outras exigências quanto a novas aplicações fazem com que o processo continue. Contudo, como as modificações tendem a corromper a estrutura da aplicação, em certo momento, é necessário tomar uma decisão específica de projetar uma família genérica de aplicações. Esse projeto tem como base a reutilização do conhecimento que foi adquirido desenvolvendo-se o conjunto inicial de aplicações.

14.3 Padrões de projeto

Quando se tenta reutilizar componentes executáveis, existe, inevitavelmente, a restrição às decisões do projeto detalhado, que foram tomadas pelos implementadores desses componentes. Essas decisões vão desde os algoritmos específicos utilizados para implementar os

componentes até os objetos e os tipos nas interfaces dos componentes. Se essas decisões de projeto estiverem em conflito com seus requisitos específicos, então a reutilização dos componentes será impossível ou introduzirá significativas ineficiências em seu sistema.

Uma maneira de contornar essa situação é reutilizar projetos mais abstratos, que não incluam detalhes de implementação. Esses são, então, implementados especificamente, para que se adequem aos requisitos de sua aplicação. As primeiras instâncias dessa abordagem do reuso se originaram da documentação e publicação de algoritmos fundamentais (Knuth, 1971) e, posteriormente, da documentação de tipos abstratos de dados, como pilhas, árvores e listas (Booch, 1987). Mais recentemente, essa abordagem de reuso tem sido incorporada à noção de padrões de projeto.

Os padrões de projeto (Gamma *et al.*, 1995) derivaram de idéias propostas por Christopher Alexander (Alexander *et al.*, 1977), que sugeriu que havia certos padrões para o projeto de edifícios que eram comuns e inerentemente satisfatórios e eficazes. O 'padrão' é uma descrição do problema e a essência de sua solução, de modo que a solução possa ser reutilizada em diferentes casos. O padrão não é uma especificação detalhada. Em vez disso, você pode pensar nele como uma descrição de conhecimento e experiência acumulados. É uma solução bem testada para um problema comum. A esse respeito, os padrões podem ser utilizados durante a análise, a fim de desenvolver modelos de sistemas, do mesmo modo que durante o processo de projeto.

No projeto de software, os padrões de projeto têm estado inevitavelmente associados com o projeto orientado a objetos. Eles dependem freqüentemente das características do objeto, como herança e polimorfismo, para oferecer generalidade. Contudo, o princípio geral é que essa abordagem deve ser igualmente aplicável a todas as abordagens de projeto de software.

Gamma *et al.* (1995) definem os quatro elementos essenciais dos padrões de projeto:

1. Um nome que seja uma referência significativa ao padrão.
2. Uma descrição da área de problema que explique quando o padrão pode ser aplicado.
3. Uma descrição da solução que explique as diferentes partes da solução de projeto, suas relações e suas responsabilidades. Não se trata de uma descrição de projeto concreta, é um *template* para uma solução de projeto, que pode servir de exemplo, de diversas manei-

Figura 14.12 ▼
Descrição do padrão *Observer*.

Nome do padrão: Observer

Descrição: Separa o display do estado de um objeto do próprio objeto e permite que sejam fornecidos displays alternativos. Quando o estado do objeto se modifica, todos os displays são automaticamente notificados e atualizados para refletir a mudança.

Descrição de problema: Em muitas situações, é necessário fornecer múltiplos displays de alguma informação de estado, como um display gráfico e um display tabular. Nem todas precisam ser conhecidas quando a informação é especificada. Todas as apresentações alternativas podem aceitar interação e, quando o estado é modificado, todos os displays devem ser atualizados.

Esse padrão pode ser utilizado em todas as situações em que mais de um formato de display para informações de estado pode ser requerido e em que não seja necessário para o objeto que mantém as informações de estado saber sobre os específicos formatos de display utilizados.

Descrição de solução: A estrutura do padrão é mostrada na Figura 14.14, que define dois objetos abstratos, Subject e Observer, e dois objetos concretos, ConcreteSubject e ConcreteObject, que herdam os atributos dos objetos abstratos relacionados. O estado a ser exibido é mantido em ConcreteSubject, que também herda operações de Subject, permitindo acrescentar e remover Observers e editar uma notificação, quando o estado se modifica.

O ConcreteObserver mantém uma cópia do estado de ConcreteSubject e implementa a interface Update () do Observer, que permite que essas cópias sejam mantidas atualizadas. O ConcreteObserver automaticamente exibe seu estado – esta não é normalmente uma operação de interface.

Conseqüências: O subject somente conhece o Observer abstrato e não conhece detalhes da classe concreta. Portanto, existe uma associação mínima entre esses objetos. Devido a essa falta de conhecimento, otimizações que aprimoram o desempenho de exibição são impraticáveis. Mudanças em subject podem fazer com que um conjunto de atualizações ligadas aos observers sejam geradas, algumas das quais podem não ser necessárias.

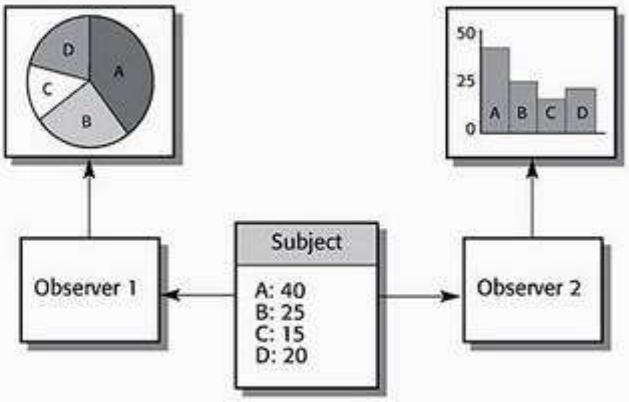


Figura 14.13 ▶
Displays múltiplos.

ras. De modo geral, isso é expresso graficamente e mostra as relações entre os objetos e as classes de objeto na solução.

4. Uma declaração das conseqüências – os resultados e as conciliações – da aplicação do padrão. Isso é utilizado para ajudar os projetistas a compreender se um padrão pode ou não ser aplicado em determinada solução.

Esses elementos essenciais da descrição de um padrão podem ser mais decompostos. Por exemplo, Gamma *et al.* (1995) dividem a descrição de um problema em *motivação* – uma descrição de por que o padrão é útil – e *aplicabilidade* – uma descrição de situações em que o padrão pode ser utilizado. Na descrição da solução, eles descrevem a estrutura do padrão, os participantes, as colaborações e a implementação.

Para ilustrar a descrição do padrão, foi descrito um dos padrões mais comumente utilizados, sugerido por Gamma *et al.*, ou seja, o padrão *Observer* (Figura 14.12). Esse padrão é utilizado quando são requeridas diferentes apresentações de um estado de objeto. Ele separa o objeto que deve ser exibido e as diferentes maneiras de apresentação. Esse exemplo está ilustrado na Figura 14.13, que mostra duas apresentações gráficas do mesmo conjunto de dados. Na descrição apresentada, são utilizados os quatro elementos essenciais de descrição e complemento com uma breve declaração sobre o que o padrão pode fazer.

As representações gráficas normalmente são utilizadas para ilustrar as classes de objeto que são empregadas em padrões e em seus relacionamentos. A Figura 14.14 é a representação em UML do padrão *Observer*.

O uso de padrões é uma maneira muito eficaz de reuso, mas eles têm um custo alto de introdução em processos de projeto e somente podem ser utilizados efetivamente por programadores experientes. A razão do alto custo de introdução é que os padrões são complexos. Para utilizá-los, um projetista deve conhecer e compreender muitos padrões em detalhes. Isso é diferente de um componente executável, em que somente as interfaces precisam ser compreendidas. Obviamente, desenvolver essa compreensão requer algum tempo.

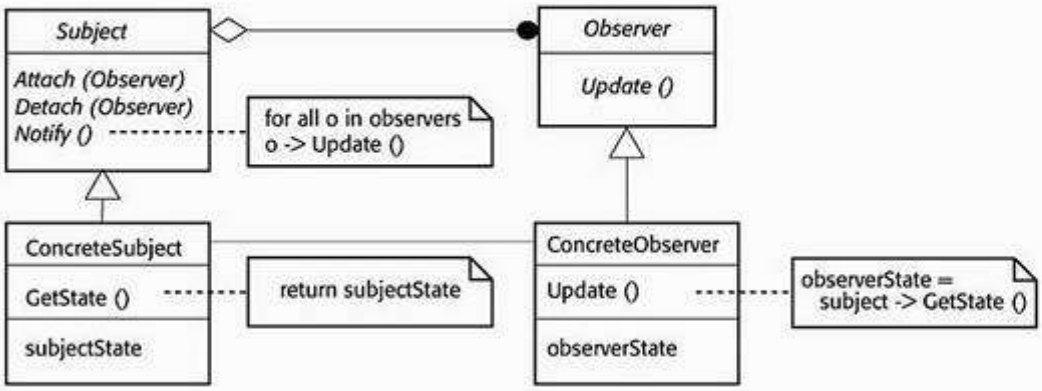


Figura 14.14 ▶
O padrão *Observer*.

Os padrões são úteis somente para o uso de programadores experientes, porque eles podem reconhecer as situações genéricas em que um padrão pode ser aplicado. Os programadores inexperientes, mesmo que tenham lido os livros sobre padrões, sempre terão dificuldade em decidir se um padrão deve ser reutilizado ou se é necessária uma solução de propósito especial.

PONTOS-CHAVE

- O projeto com reuso envolve projetar o software com base nos exemplos existentes de um bom projeto, fazendo uso de componentes de software, quando eles estiverem disponíveis.
- As vantagens do reuso de software são os custos mais baixos, o desenvolvimento mais rápido de software e menores riscos. A confiabilidade do sistema aumenta e os especialistas podem ser utilizados mais eficazmente, concentrando sua habilidade no projeto de componentes reutilizáveis.
- O desenvolvimento com base em componentes leva em conta a utilização de componentes de 'caixa preta', com interfaces claramente definidas, como as interfaces *requires* e *provides*. Entre os tipos de componentes que podem ser reutilizados estão funções, abstrações de dados, *frameworks* e completos sistemas de aplicação.
- O reuso de produtos COTS se ocupa do reuso de 'sistemas de prateleira' em grande escala. Esses produtos oferecem bastante funcionalidade e seu reuso pode reduzir radicalmente os custos e o tempo de desenvolvimento.
- Os componentes de software que tenham sido projetados para reuso devem ser independentes, refletir abstrações estáveis de domínio, fornecer acesso a estado por meio de operações de interface e não devem manipular exceções.
- As famílias de aplicações são aplicações relacionadas, que são desenvolvidas a partir de uma ou mais aplicações básicas. Um sistema genérico é adaptado e especializado para atender aos requisitos específicos de funcionalidade, plataforma-alvo ou configuração operacional.
- Os padrões de projeto são abstrações de alto nível, que documentam soluções bem-sucedidas de projeto. Eles são fundamentais para o reuso de projetos no desenvolvimento orientado a objetos. Uma descrição-padrão deve incluir um nome de padrão, uma descrição de problema e de solução, uma declaração dos resultados e os componentes de uso do padrão.

LEITURAS SUGERIDAS

Component software: beyond object-oriented programming. No momento em que este livro foi escrito, esse era o único livro disponível sobre o desenvolvimento baseado em componentes; no entanto, espero que outros apareçam em um futuro próximo. É uma descrição muito boa da abordagem, mas não é realmente acessível, a menos que o leitor tenha algum background no desenvolvimento orientado a objetos. (C. Szyperski, 1998, Addison Wesley.)

Software reuse: architecture, process and organization for business success. Uma discussão abrangente do reuso, com ênfase específica em questões organizacionais e na integração do reuso com um processo de desenvolvimento orientado a objetos. (I. Jacobson, M. Griss e P. Jonsson, 1997, Addison Wesley.)

Design patterns: elements of reusable object-oriented software. Esse é o manual original dos padrões de software, que introduziu a noção de padrões de software em uma ampla comunidade. (E. Gamma, R. Helm, R. Johnson e J. Vlissides, 1995, Addison Wesley.)

EXERCÍCIOS

14.1 Quais são os principais fatores técnicos e não técnicos que prejudicam o reuso de software? A partir de sua própria experiência, você reutiliza muito produtos de software? Em caso negativo, por que não?

14.2 Explique por que as economias de custo do reuso de software não são simplesmente proporcionais ao tamanho dos componentes reutilizados.

14.3 Dê quatro circunstâncias em que você não recomendaria o reuso de software.

14.4 Sugira possíveis interfaces *requires* e *provides* para os seguintes componentes:

- Um componente que implementa uma conta bancária.
- Um componente que implementa um teclado independente de linguagem. Teclados em países diferentes têm diferentes organizações de teclas e diferentes conjuntos de caracteres.
- Um componente que implementa recursos de gerenciamento de versões, conforme será discutido no Capítulo 29.

14.5 Qual é a diferença entre um *framework* de aplicações e um produto COTS, no que diz respeito ao reuso? Por que, às vezes, é mais fácil reutilizar um produto COTS do que um *framework* de aplicações?

14.6 Utilizando o exemplo do sistema da estação meteorológica, descrito no Capítulo 12, sugira uma arquitetura para uma família de aplicações que se ocupem do monitoramento remoto e da coleta de dados.

14.7 Utilizando o exemplo de uma família de gerenciamento de estoque, mostrado na Figura 14.9, sugira operações que tenham de ser acrescentadas ou modificadas para aceitar a reordenação de itens de estoque, quando o nível cair abaixo de um número especificado.

14.8 Por que os padrões são uma maneira eficiente de reuso de projeto? Quais são as desvantagens dessa abordagem para o reuso?

14.9 O reuso de software levanta uma série de questões de copyright e de propriedade intelectual. Se um cliente paga um fornecedor de software para desenvolver algum sistema, quem tem o direito de reutilizar o código desenvolvido? O fornecedor do software tem o direito de utilizar esse código como base para um componente genérico? Que mecanismos de pagamento poderiam ser utilizados para reembolsar os provedores de componentes reutilizáveis? Discuta essas e outras questões éticas associadas ao reuso de software.

15 Projeto de interface com o usuário

Objetivos

O objetivo deste capítulo é apresentar alguns aspectos do projeto de interface com o usuário que são importantes para os engenheiros de software. Depois de ler este capítulo, você poderá:

- compreender os princípios gerais de projeto que devem ser seguidos pelos engenheiros responsáveis pelo projeto de interface com o usuário;
- conhecer cinco estilos deferentes de interação com um sistema de software;
- conhecer diferentes estilos de apresentação de informações e saber quando a apresentação gráfica de informações é apropriada;
- compreender alguns fundamentos do projeto de apoio ao usuário, que é integrado no software, e
- compreender os atributos da facilidade de uso e as abordagens simples para a avaliação de sistemas.

Sumário

15.1 Princípios de projeto de interface com o usuário
15.2 Interação dos usuários
15.3 Apresentação das informações
15.4 Apoio ao usuário
15.5 Avaliação da interface

O projeto de sistemas de computador abrange várias atividades, desde o projeto de hardware até o projeto de interface com o usuário. Embora especialistas sejam freqüentemente empregados para o projeto de hardware, poucas organizações empregam projetistas especializados em interfaces. Portanto, os engenheiros de software geralmente têm de assumir a responsabilidade pelo projeto de interface com o usuário, bem como pelo projeto do software para implementar essa interface. Especialistas em fatores humanos podem prestar assistência nesse processo, em grandes organizações; em empresas menores, esses especialistas raramente são utilizados.

O bom projeto de interface com o usuário é fundamental para o sucesso de um sistema. Uma interface que é difícil de ser utilizada, na melhor das hipóteses, resultará em um alto nível de erros do usuário. Pior ainda, os usuários simplesmente se recusarão a utilizar o sistema de software, independentemente de sua funcionalidade. Se as informações forem apresentadas de maneira confusa ou enganosa, os usuários poderão se confundir com o significado dessas informações. Eles podem iniciar uma seqüência de ações que venham a corromper os dados ou mesmo a causar falhas catastróficas no sistema.

Quando a primeira edição deste livro foi publicada, em 1982, o dispositivo-padrão de interação era um terminal alfanumérico 'burro', com caracteres verdes ou azuis exibidos sobre um fundo preto. As interfaces com o usuário tinham de ser baseadas em textos ou em formulários. Quase todos os usuários de computadores, atualmente, têm um computador pessoal. Esses equipamentos fornecem uma GUI (*graphical user interface* – interface gráfica com o usuário) que aceita um display colorido de alta resolução e interações com o uso de um mouse e de um teclado.

Embora as interfaces baseadas em texto ainda sejam amplamente utilizadas, em especial em sistemas legados, os usuários de computadores atualmente esperam que os sistemas de aplicações tenham algum tipo de interface gráfica com o usuário. A Figura 15.1 mostra as características principais desse tipo de interface.

As vantagens das GUIs são:

1. Elas são relativamente fáceis de aprender e utilizar. Usuários sem nenhuma experiência em computação podem aprender a utilizar a interface depois de uma rápida sessão de treinamento.
2. O usuário tem várias telas (janelas) para a interação com o sistema. É possível alternar de uma tarefa para outra, sem perder de vista as informações geradas durante a primeira tarefa.
3. É possível a interação rápida de tela inteira, com acesso imediato a qualquer ponto da tela.

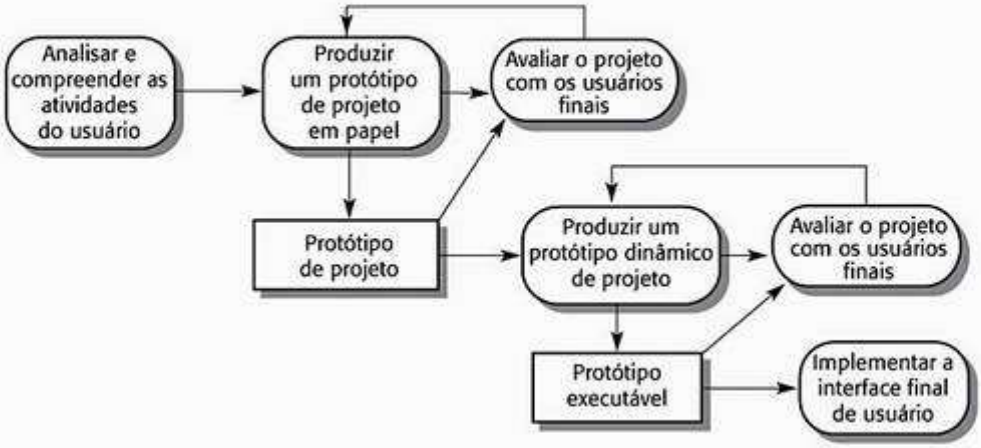
Figura 15.1 ▶

Características das interfaces gráficas com o usuário.

Característica	Descrição
Janelas	Várias janelas permitem que informações diferentes sejam exibidas simultaneamente na tela do usuário.
Ícones	Os ícones representam diferentes tipos de informações. Em alguns sistemas, eles representam arquivos; em outros, representam processos.
Menus	Os comandos são selecionados a partir de um menu, em vez de serem digitados em uma linguagem de comando.
Apontamento	Um dispositivo de apontamento, como um mouse, é utilizado para selecionar as opções em um menu ou indicar os itens de interesse em uma janela.
Gráficos	Elementos gráficos podem ser misturados com texto no mesmo monitor.

Figura 15.2

Processo de projeto de interface com o usuário.



O objetivo deste capítulo é sensibilizar os engenheiros de software sobre algumas questões básicas importantes para o projeto de interface com o usuário. Os projetistas de software e programadores são, freqüentemente, competentes usuários da tecnologia, como as classes Swing, de Java (Eckstein *et al.*, 1998), ou HTML (Musciano e Kennedy, 1998), que são fundamentais para a implementação da interface com o usuário. Contudo, muito freqüentemente, eles não utilizam essa tecnologia de maneira apropriada e criam interfaces que são deselegantes, inadequadas e difíceis de utilizar.

Portanto, procura-se dar alguma orientação sobre o projeto de recursos para os usuários finais, e não sobre o projeto de software que implementa esses recursos. Devido às limitações de espaço, são consideradas somente as interfaces gráficas. Não são abordadas interfaces que requerem displays especiais (talvez, muito simples), como celulares, aparelhos de vídeo, televisões, aparelhos de fax e copiadoras. Naturalmente, é possível apenas fazer uma introdução desse tópico neste texto. Caso necessário, consulte os textos de Shneiderman (1998), Dix *et al.* (1998) e Preece *et al.* (1994) para mais informações sobre esse assunto.

A Figura 15.2 ilustra o processo iterativo do projeto de interface com o usuário. Como foi discutido no Capítulo 8, o desenvolvimento exploratório é a abordagem mais eficaz no projeto de interfaces. Esse processo de prototipação pode começar com modelos de interface esboçados em papel, antes de se começar a desenvolver projetos em tela, que simulam a interação com o usuário. Deve ser utilizada uma abordagem centrada no usuário (Norman e Draper, 1986), com os usuários finais do sistema tomando parte ativa no processo de projeto. Em alguns casos, o papel do usuário é de avaliação; em outros, eles participam como membros ativos da equipe de projeto (Kyng, 1988; Greenbaum e Kyng, 1991).

Uma atividade fundamental do projeto de UI (*user interface* – interface com o usuário) é a análise das atividades do usuário que deverão ser compatíveis com o sistema de computador. Sem uma compreensão do que o usuário deseja fazer com o sistema de computador não há nenhuma perspectiva realista de projetar uma interface com o usuário que seja eficaz. Para desenvolver essa compreensão, podem ser utilizadas determinadas técnicas, como a análise de tarefas (Diaper, 1989), os estudos etnográficos, como discutido no Capítulo 6 (Suchman, 1983; Hughes *et al.*, 1997), as entrevistas com usuários e as observações ou, comumente, uma mistura de todas elas.

15.1 Princípios de projeto de interface com o usuário

Os projetistas de interface com o usuário têm de levar em conta a capacidade física e mental das pessoas que utilizam software. As pessoas têm uma memória limitada, de curto

prazo (veja o Capítulo 22), e cometem erros, especialmente quando precisam lidar com muitas informações ou estão sob estresse. Elas têm ampla variedade de capacidades físicas. Tudo isso deve ser levado em conta quando são projetadas interfaces com o usuário.

As capacidades humanas são a base para os princípios de projeto, mostrados na Figura 15.3. Esses são princípios gerais, aplicáveis a todos os projetos de interface com o usuário, e normalmente devem ser exemplificados com mais diretrizes de projeto detalhadas para determinadas organizações ou tipos de sistema específicos. Uma lista mais longa de diretrizes mais específicas para o projeto de interface com o usuário é fornecida por Shneiderman (1998).

O princípio da familiaridade com o usuário sugere que os usuários não devem ser forçados a se adaptar a uma interface porque sua implementação é conveniente. A interface deve utilizar termos familiares ao usuário e os objetos manipulados pelo sistema devem estar diretamente relacionados ao ambiente do usuário. Por exemplo, se um sistema for projetado para ser utilizado por controladores de tráfego aéreo, os objetos manipulados deverão ser aeronaves, rotas de vôo, bóias luminosas etc. As operações associadas podem ser: aumentar ou reduzir a velocidade de uma aeronave, ajustar o 'teto' e mudar a altitude. A implementação básica da interface, em termos de arquivos e estruturas de dados, deve estar oculta do usuário final.

O princípio de consistência da interface com o usuário significa que os comandos e os menus do sistema devem ter o mesmo formato, os parâmetros devem ser fornecidos a todos os comandos da mesma maneira e a pontuação utilizada nos comandos deve ser similar. Interfaces consistentes reduzem o tempo de aprendizagem do usuário. O conhecimento apreendido em um comando ou uma aplicação vale para outras partes do sistema.

A consistência da interface com os subsistemas também é importante. Tanto quanto for possível, os comandos com os mesmos significados em subsistemas diferentes devem ser expressos da mesma maneira. Freqüentemente, ocorrem erros quando os mesmos comandos de teclado, como 'Control-k' ou 'Control-b', significam coisas diferentes em diferentes sistemas. Por exemplo, no processador de textos que foi utilizado para escrever este livro, o comando 'Control-b' significa marcar um trecho em negrito, mas no programa de gráficos que utilizo para criar diagramas significa mover o objeto selecionado para trás de outro objeto. Erros são cometidos quando eles são utilizados em conjunto e, muitas vezes, procura-se marcar um texto em negrito em um diagrama utilizando-se a combinação de teclas. Isso resulta no desaparecimento do texto e pode causar muita confusão. Os padrões de sistemas com janelas geralmente definem atalhos de teclas de comando e devem ser seguidos para ajudar a evitar esse tipo de erro.

Figura 15.3 ▶

Princípios do projeto de interface com o usuário.

Princípios	Descrição
Familiaridade com o usuário	A interface deve utilizar termos e conceitos que tenham como base a experiência das pessoas que mais vão utilizar o sistema.
Consistência	A interface deve ser consistente, no sentido de que, sempre que possível, operações semelhantes devem ser ativadas da mesma maneira.
Mínimo de surpresa	Os usuários nunca devem ser surpreendidos com o comportamento de um sistema.
Facilidade de recuperação	A interface deve incluir mecanismos para permitir aos usuários a recuperação a partir de erros.
Orientação do usuário	A interface deve fornecer feedback significativo, quando ocorrerem erros, e oferecer recursos sensíveis ao contexto de ajuda ao usuário.
Diversidade de usuários	A interface deve fornecer recursos de interação apropriados a diferentes tipos de usuários de sistema.

Esse nível de consistência é um nível básico. Os projetistas de interfaces devem sempre tentar atingir esse nível em uma interface com o usuário. Um nível superior de consistência, às vezes, também é desejável. Por exemplo, é possível que seja apropriado oferecer a compatibilidade entre as mesmas operações (como impressão e cópia) em todos os tipos de entidades do sistema. Contudo, Grudin (1989) ressalta que a consistência completa não é possível ou desejável. Pode ser sensato implementar a operação de exclusão, em um PC, arrastando entidades para dentro de uma lixeira. Seria estranho apagar um texto em um processador de textos dessa maneira.

O princípio do mínimo de surpresa é apropriado, porque os usuários ficam muito irritados quando um sistema se comporta de maneira inesperada. À medida que um sistema é utilizado, os usuários constroem um modelo mental de como o sistema trabalha. Se uma ação em um contexto causar um tipo particular de mudança, é razoável esperar que a mesma ação em um contexto diferente cause uma mudança semelhante. Se algo completamente diferente acontecer, o usuário ficará surpreso e confuso. Os projetistas de interface devem, portanto, se assegurar de que ações semelhantes tenham efeitos equivalentes.

O princípio da facilidade de recuperação é importante, porque os usuários inevitavelmente cometem erros quando utilizam os sistemas. O projeto de interface pode minimizar esses erros (isto é, utilizar menus significa que os erros de digitação são evitados), mas eles nunca serão completamente eliminados. As interfaces com o usuário devem conter recursos que permitam aos usuários a recuperação a partir dos erros cometidos. Esse processo pode ser de dois tipos:

1. *Confirmação de ações destrutivas* Se os usuários especificam uma ação que seja potencialmente destrutiva, é necessário pedir que eles confirmem se aquilo é o que realmente desejam fazer, antes que qualquer informação seja destruída.

2. *Disponibilidade de um recurso do tipo desfazer* A opção desfazer restaura o sistema a um estado anterior à ocorrência da ação. Vários níveis de desfazer são úteis, uma vez que os usuários nem sempre reconhecem imediatamente que cometeram um erro.

Um princípio correlato é o princípio da assistência ao usuário. As interfaces devem ter opções de assistência ou recursos de ajuda ao usuário inseridos. Esses recursos devem estar integrados ao sistema e proporcionar diferentes níveis de ajuda e orientação. Os níveis devem abranger desde informações básicas para iniciar o sistema até uma descrição completa dos recursos do sistema. Os recursos de ajuda, como será discutido na Seção 15.4, devem ser estruturados e os usuários não devem ser sobrecarregados com informações quando pedem ajuda.

O princípio da diversidade de usuários reconhece que, para muitos sistemas interativos, pode haver diferentes tipos de usuários. Alguns usuários serão usuários casuais, que interagem ocasionalmente com o sistema; enquanto outros podem ser usuários freqüentes, que utilizam o sistema por diversas horas, todos os dias. Os usuários casuais precisam de interfaces que forneçam orientação, enquanto os usuários freqüentes requerem atalhos que lhes permitam interagir o mais rápido possível. Além disso, os usuários podem apresentar diferentes tipos de problemas físicos e, se possível, a interface deve ser adaptável para atendê-los. Portanto, pode ser necessário fornecer recursos para exibir texto ampliado, substituir som por texto, produzir botões grandes, e assim por diante.

O princípio de reconhecimento da diversidade de usuários pode ser conflitante com outros princípios do projeto de interface, porque alguns tipos de usuários podem preferir uma interação muito rápida a, por exemplo, dispor da consistência da interface com o usuário. Da mesma maneira, o nível de orientação requerida para o usuário pode ser radicalmente diferente para diferentes tipos de usuários e pode ser impossível desenvolver compatibilidade que seja adequada a todos eles. O projetista de interface deve, inevitavelmente, fazer algumas conciliações, dependendo dos usuários reais do sistema.

15.2 Interação dos usuários

O projetista de uma interface com o usuário para um computador se defronta com duas importantes questões. Como as informações podem, a partir do usuário, ser fornecidas ao sistema de computador e como as informações do computador podem ser apresentadas ao usuário? Uma interface com o usuário, que seja coerente, deve integrar a interação do usuário e a apresentação de informações.

A interação dos usuários é o foco desta seção, com a apresentação de informações abordada na Seção 15.3. A interação do usuário significa emitir comandos e dados associados ao sistema de computador. Nos primeiros computadores, a única maneira de fazer isso era por meio de uma interface de linha de comando, em que uma linguagem de propósito especial era utilizada para a comunicação com a máquina. Contudo, essa abordagem somente era utilizável por especialistas, e uma série de outras abordagens evoluíram, o que tornou sua utilização mais fácil. Shneiderman (1998) classificou esses diferentes modos de interação em cinco estilos principais:

1. *Manipulação direta* na qual o usuário interage diretamente com objetos na tela. Por exemplo, para excluir um arquivo, o usuário pode arrastá-lo para uma lixeira.
2. *Seleção de menu* na qual o usuário seleciona um comando a partir de uma lista de possibilidades (um menu). Acontece, freqüentemente, que outro objeto na tela é selecionado ao mesmo tempo e o comando opera nesse objeto. Nessa abordagem, para excluir um arquivo, o usuário o seleciona e, então, seleciona o comando para excluir.
3. *Preenchimento de formulário* na qual o usuário preenche os campos de um formulário. Alguns campos podem ter menus associados e o formulário pode ter 'botões' que, quando pressionados, iniciam alguma ação. Seria artificial excluir um arquivo utilizando uma interface baseada em formulários. Isso envolveria preencher o nome do arquivo e, então, 'pressionar' o botão para excluir.
4. *Linguagem de comando* na qual o usuário emite um comando especial e parâmetros associados para instruir o sistema sobre o que fazer. Para excluir um arquivo, o usuário emite um comando para excluir, com o nome do arquivo como parâmetro.
5. *Linguagem natural* na qual o usuário emite um comando em linguagem natural. Para excluir um arquivo, o usuário pode, portanto, digitar 'deletar o arquivo denominado xxx'.

Cada um desses diferentes estilos de interação tem suas vantagens e desvantagens e são mais adequados para diferentes tipos de aplicações e usuários (Shneiderman, 1998). A Figura 15.4 mostra as principais vantagens e desvantagens desses estilos e sugere tipos de aplicações em que eles podem ser utilizados.

Naturalmente, esses estilos de interação podem ser misturados e vários estilos diferentes são utilizados na mesma aplicação. Por exemplo, o Windows, da Microsoft, aceita a manipulação direta da representação de ícones de arquivos e diretórios, a seleção de comandos baseada em menus e, para alguns comandos, como os comandos de configuração, o usuário precisa preencher um formulário de propósito especial, que lhe é apresentado.

As interfaces com o usuário na Web se baseiam na compatibilidade fornecida pela HTML (a linguagem de descrição de páginas, utilizada para páginas Web), juntamente com linguagens como Java, que podem associar programas com componentes em uma página. Como essas interfaces baseadas na Web são, freqüentemente, projetadas para usuários casuais, elas utilizam, na maioria das vezes, interfaces baseadas em formulários. É possível construir interfaces de manipulação direta na Web, mas, no momento em que este livro foi escrito, esta ainda era uma tarefa de programação complexa.

Em princípio, seria possível separar o estilo de interação das entidades subjacentes, que são manipuladas por meio da interface com o usuário. Essa foi a base do modelo de Seeheim

Estilo de interação	Principais vantagens	Principais desvantagens	Exemplos de aplicações
Manipulação direta	Interação rápida e intuitiva. Fácil de aprender.	Pode ser difícil de implementar. Adequada somente quando houver uma metáfora visual para tarefas e objetos.	Videogames. Sistemas de CAD.
Seleção de menu	Evita erros de usuários. Exige pouca digitação.	Lento para os usuários experientes. Poderá se tornar complexo, se houver muitas opções de menu.	Sistemas de propósito mais genéricos.
Preenchimento de formulário	Entrada de dados simples. Fácil de aprender.	Toma muito espaço em tela.	Controle de estoque. Processamento de empréstimo pessoal.
Linguagem de comando	Flexível e com grande capacidade.	Difícil de aprender. Gerenciamento de erros inadequado.	Sistemas operacionais. Sistemas de recuperação de informações de bibliotecas.
Linguagem natural	Acessível a usuários casuais. Facilmente ampliado.	Requer mais digitação. Os sistemas de compreensão de linguagem natural não são confiáveis.	Sistemas de tabelas de tempo. Sistemas de recuperação de informações na Web.

Figura 15.4 ▲

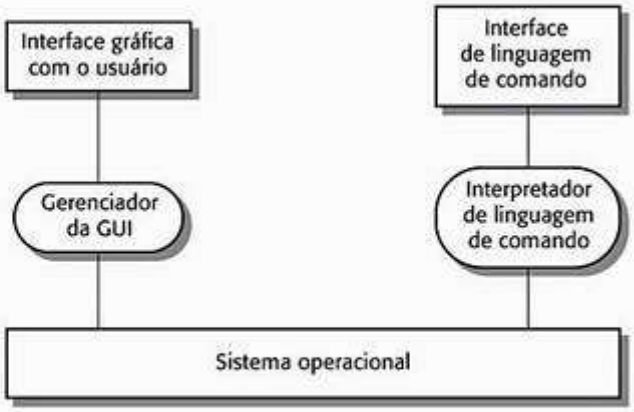
Vantagens e desvantagens dos estilos de interação.

(Pfaff e ten Hagen, 1985) de gerenciamento de interface com o usuário. Nesse modelo, a apresentação das informações, o gerenciamento de diálogo e a aplicação são separados. Na realidade, esse é um modelo ideal e não um modelo prático, mas é certamente possível ter interfaces separadas para diferentes classes de usuários (usuários casuais e usuários experientes, por exemplo), que interagem com o mesmo sistema básico. Isso está ilustrado na Figura 15.5, que mostra uma interface de linguagem de comando e uma interface gráfica para um sistema operacional subjacente, como Linux.

Essa separação da apresentação, da interação e das entidades que estão envolvidas na interface com o usuário também é fundamental para a abordagem do tipo Modelo-Visão-Controlador (MVC), que será discutida na próxima seção. Esse modelo é comparável ao modelo de Seeheim, mas é utilizado para implementar a interface com o usuário para objetos, em vez de aplicações completas.

Figura 15.5 ▶

Interfaces de múltiplos usuários.



15.3 Apresentação das informações

Todos os sistemas interativos precisam fornecer algum meio de apresentar as informações aos usuários. A apresentação de informações pode ser simplesmente uma representação direta das informações de entrada (por exemplo, o texto em um processador de textos) ou pode apresentar as informações graficamente. Uma boa prática de projeto de sistema é manter o software requerido para apresentação de informações separado das próprias informações. Até certo ponto, isso contradiz a filosofia orientada a objetos, que sugere que as operações que envolvem dados devem ser definidas com os próprios dados. Contudo, isso pressupõe que o projetista dos objetos sempre sabe a melhor maneira de apresentar as informações; definitivamente, nem sempre isso é verdadeiro. Muitas vezes, é difícil saber a melhor maneira de apresentar os dados, quando eles estão sendo definidos, e as estruturas de objetos não devem fixar as operações de apresentação.

Separando o sistema de apresentação e os dados, a representação na tela do usuário pode ser modificada, sem precisar modificar o sistema computacional subjacente. Essa opção está ilustrada na Figura 15.6.

A abordagem MVC (Figura 15.7), que primeiro esteve amplamente disponível em Smalltalk (Goldberg e Robson, 1983), é um meio eficaz de trabalhar com várias apresentações de dados. Os usuários podem interagir com cada apresentação, utilizando o estilo que for apropriado. Os dados a serem expostos estão encapsulados em um modelo de objeto. Cada modelo de objeto pode ter uma série de objetos de visão separados, associados a ele, em que cada visão é uma representação diferente de display do modelo. Já foi ilustrado esse conceito, no capítulo anterior, quando se discutiu a abordagem MVC como uma estrutura orientada a objetos.

Cada visão tem um objeto controlador associado, que manipula a entrada dos usuários e a interação de dispositivos. Portanto, um modelo que representa dados numéricos pode ter uma visão que represente os dados como um histograma e uma visão que apresente os dados como uma tabela. O modelo pode ser editado, modificando-se os valores na tabela, aumentando ou reduzindo a extensão das barras no histograma. Essa opção foi ilustrada no Capítulo 14 (Figura 14.14), quando foi discutido o uso do padrão *Observer* na implementação do *framework* MVC.

Encontrar a melhor apresentação das informações exige o conhecimento da experiência dos usuários das informações e da maneira pela qual eles utilizam o sistema. Ao decidir como apresentar as informações, o projetista tem de levar uma série de fatores em conta:

1. O usuário está interessado em informações precisas ou nas relações entre diferentes valores de dados?
2. Com que rapidez os valores das informações são modificados? A mudança em um valor deve ser indicada imediatamente ao usuário?
3. O usuário deve tomar alguma iniciativa em resposta a uma mudança nas informações?
4. O usuário precisa interagir com as informações exibidas por meio de uma interface de manipulação direta?
5. As informações a serem exibidas são textuais ou numéricas? Os valores relativos dos itens das informações são importantes?

Figura 15.6 ▶

Apresentação das informações.

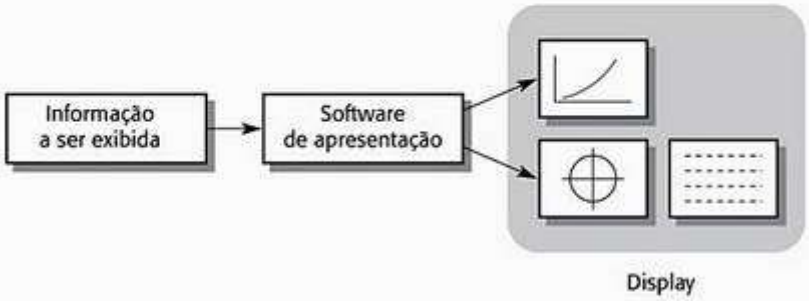
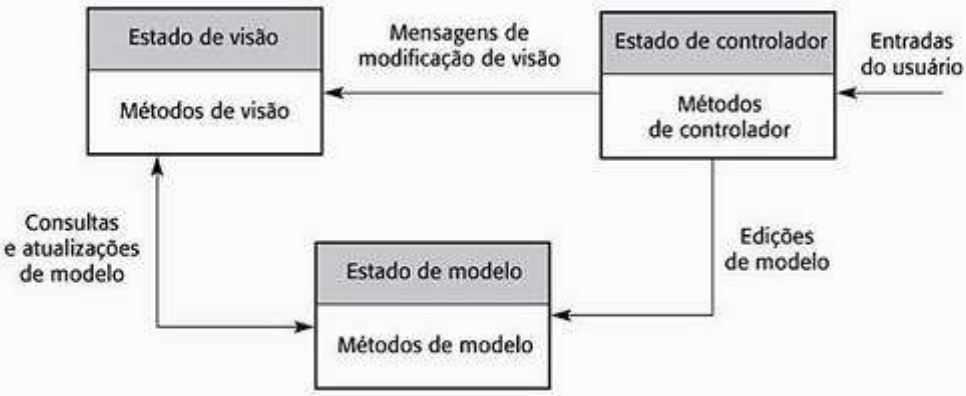


Figura 15.7

Modelo MVC de interação do usuário.



As informações que não se modificam durante uma sessão podem ser apresentadas graficamente ou como texto, dependendo da aplicação. A apresentação textual ocupa menos espaço em tela, mas não pode ser lida 'de imediato'. As informações que não se modificam devem ser diferenciadas das informações dinâmicas, utilizando-se um estilo de apresentação diferente. Por exemplo, toda informação estática pode ser apresentada com a utilização de determinada fonte ou pode ser destacada com o uso de uma cor específica ou, ainda, ter sempre um ícone associado a ela.

As informações devem ser representadas como texto quando informações numéricas precisas forem necessárias e as informações se modificarem relativamente devagar. Se os dados se modificam rapidamente ou se as relações entre dados são significativos, a apresentação gráfica deve sempre ser utilizada.

Por exemplo, considere um sistema que registre e resuma os números referentes às vendas mensais em uma empresa. A Figura 15.8 ilustra como a mesma informação pode ser apresentada como texto ou como um formulário gráfico.

De modo geral, os gerentes que estudam os números relacionados a vendas estão mais interessados em tendências ou em números anômalos do que em números precisos. A apresentação gráfica dessas informações como um histograma faz com que os números anômalos referentes a março e maio se sobressaiam dentre os demais. A Figura 15.8 também ilustra como a apresentação em texto toma menos espaço do que uma apresentação gráfica das mesmas informações.

As informações numéricas com variação dinâmica em geral são mais bem apresentadas graficamente, utilizando-se uma representação análoga. Os displays digitais que se modifi-

Figura 15.8

Apresentações alternativas de informação.

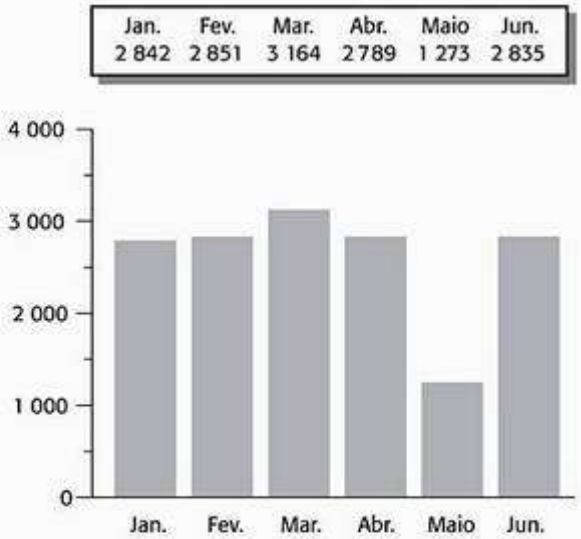
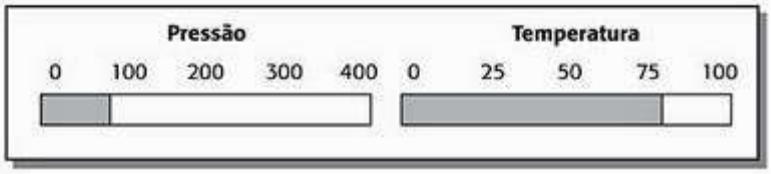


Figura 15.9

Métodos de apresentação de informações numéricas que variam dinamicamente.

Figura 15.10

Display de informações gráficas que mostra valores relativos.



cam constantemente são confusos, uma vez que as informações precisas são de difícil assimilação rápida. O display gráfico pode ser complementado, se necessário, por um display digital preciso. Diferentes meios de apresentar as informações numéricas dinâmicas são mostrados na Figura 15.9.

Displays analógicos contínuos dão ao observador algum senso de valor relativo. Na Figura 15.10, os valores de temperatura e pressão são aproximadamente os mesmos. Contudo, o display gráfico mostra que a temperatura está perto de seu valor máximo, ao passo que a pressão não atingiu 25 por cento de seu valor máximo. Com apenas um valor digital, o observador precisa conhecer os valores máximos. Ele precisa calcular mentalmente o estado relativo da leitura. O tempo extra necessário para pensar pode levar a erros humanos, em situações de estresse, quando ocorrerem problemas e os displays dos operadores estiverem mostrando leituras anormais.

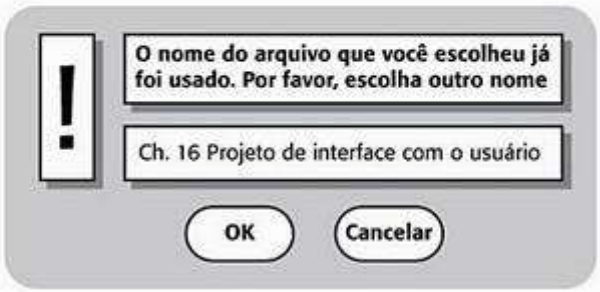
Quando informações alfanuméricas precisam ser apresentadas, os gráficos podem ser utilizados para obter as informações a partir de seu background. Em vez de apresentar uma linha de informações, elas podem ser exibidas em uma caixa ou indicadas com o uso de um ícone (Figura 15.11). A caixa que exibe a mensagem se destaca na atual exibição de tela. A atenção do usuário é imediatamente atraída para ela.

O destaque gráfico também pode ser utilizado para chamar a atenção do usuário quanto a modificações de partes do display. Contudo, se essas mudanças ocorrerem rapidamente, o destaque gráfico não deverá ser utilizado, uma vez que modificações rápidas podem fazer com que o display pareça piscar, e isso perturba e irrita os usuários.

Quando grandes quantidades de informações precisam ser apresentadas, é possível utilizar visualizações abstratas, que vinculam itens de dados relacionados. Isso pode expor relações que não são óbvias a partir dos dados brutos. Os projetistas de interface com o usuário

Figura 15.11

Visão de texto de informações alfanuméricas.



devem estar cientes das possibilidades de visualização, especialmente quando a interface com o usuário do sistema deve representar entidades físicas. Exemplos de visualizações de dados são:

1. Informações meteorológicas, recolhidas a partir de uma série de fontes, são mostradas como um mapa meteorológico com linhas isobáricas, correntes e frentes de ar etc.
2. O estado de uma rede de telefonia é exibido graficamente como um conjunto de nós vinculados em um centro de gerenciamento de rede.
3. O estado de uma indústria química é visualizado ao mostrar as pressões e as temperaturas, em um conjunto interligado de tanques e tubulações.
4. O modelo de uma molécula é exibido e manipulado em três dimensões, por meio de um sistema de realidade virtual.
5. Um conjunto de páginas Web é mostrado como uma árvore hiperbólica (Lamping *et al.*, 1995).

Shneiderman (1998) apresenta uma boa visão geral de diferentes abordagens de visualização e identifica diferentes classes de visualização que podem ser utilizadas. Entre essas classes estão a visualização de dados que utilizam apresentações em 2-D ou 3-D e como árvores ou redes. A maioria se ocupa com o display de grandes quantidades de informações gerenciadas em um computador. Contudo, o uso mais comum de visualização em interfaces com o usuário é representar alguma estrutura física, tal como a estrutura molecular de uma nova medicação, as conexões em uma rede de telecomunicações etc. As apresentações tridimensionais que podem utilizar equipamento especial de realidade virtual são particularmente eficazes na visualização de produtos. A manipulação direta dessas visualizações é uma maneira muito eficaz de interagir com os dados.

15.3.1 Cores nos projetos de interface

Todos os sistemas interativos, exceto os especializados – os sistemas de tela pequena, displays coloridos e interfaces com o usuário –, utilizam cores de diferentes maneiras. Em alguns sistemas (como os processadores de textos) a cor é utilizada simplesmente para dar destaque; em outros (como os sistemas de CAD), ela é utilizada para mostrar diferentes camadas em um projeto.

As cores podem aperfeiçoar as interfaces com o usuário, auxiliando os usuários a compreender e gerenciar a complexidade. Contudo, é fácil utilizar cores inadequadamente e criar interfaces com o usuário que não sejam atrativas e que induzam a erros. Como princípio geral, os projetistas de interfaces com o usuário devem ser conservadores no uso de cores nos displays dos usuários. Shneiderman (1998) apresenta 14 importantes diretrizes para o uso eficaz de cores em interfaces com o usuário. Dessas, as mais importantes são:

1. *Limite o número de cores utilizadas e seja conservador no modo de utilizá-las* Não se deve utilizar mais do que quatro ou cinco cores diferentes em uma janela, e não mais do que sete em uma interface de sistema. A cor deve ser utilizada seletivamente e de maneira consistente. Não se deve simplesmente utilizar cores para 'alegrar' a interface.

2. *Utilize a mudança de cores para mostrar uma modificação no status do sistema* Se um display muda de cor, isso deve significar que um evento importante ocorreu. Portanto, em um marcador de combustível, uma mudança de cor pode indicar que a reserva de combustível está baixa. O destaque de cores é particularmente importante em displays complexos, em que centenas de entidades distintas podem ser exibidas.

3. *Utilize código de cores para apoiar a tarefa que os usuários estão tentando realizar* Se eles precisarem identificar casos anômalos, destaque essas instâncias; se similaridades também devem ser descobertas, destaque essa necessidade utilizando cores diferentes.

4. *Utilize código de cores de maneira cuidadosa e consistente* Se uma parte de um sistema exibe mensagens de erro em vermelho (digamos), todas as outras partes devem fazer o mesmo; o vermelho não deve ser utilizado para nada mais. Se isso ocorrer, o usuário poderá interpretar o display vermelho como uma mensagem de erro. É preciso levar em conta que algumas classes de usuários podem fazer suposições sobre o significado de determinadas cores.

5. *Seja cuidadoso quanto a pares de cores* Por causa da fisiologia do olho humano, as pessoas não podem focalizar o vermelho e o azul simultaneamente. O estresse visual é uma provável conseqüência do vermelho em um display azul. Outras combinações de cores podem também ser visualmente perturbadoras ou difíceis de ler.

Os dois erros mais comuns cometidos pelos projetistas, quando eles incorporam cores em uma interface com o usuário, são a associação de significados com determinadas cores e a utilização de cores demais em um display.

As cores não devem ser utilizadas para representar significados, por duas razões. Cerca de 10 por cento dos homens são cegos para as cores e podem se enganar com o significado em questão. As percepções humanas de cores são diferentes e existem diferentes convenções, em diferentes profissões, sobre o significado de diferentes cores. Os usuários com experiências diferentes podem, inconscientemente, interpretar a mesma cor de maneiras diferentes. Por exemplo, para um motorista, vermelho geralmente significa *perigo*. Contudo, para um químico, vermelho significa *quente*.

Se as cores forem utilizadas em demasia ou forem muito intensas, o display poderá parecer confuso. A massa de cores pode perturbar o usuário (da mesma maneira que algumas pinturas abstratas não podem ser vistas por muito tempo, sem gerar algum desconforto) e causar cansaço visual. A confusão do usuário também será possível se as cores forem utilizadas de modo inconsistente.

15.4 Apoio ao usuário

Um princípio de projeto sugerido na primeira seção deste capítulo foi que as interfaces com o usuário devem sempre fornecer algum tipo de sistema de ajuda on-line. Os sistemas de ajuda são uma faceta de uma parte geral de projeto de interface com o usuário, ou seja, o fornecimento de orientação aos usuários, o que abrange três áreas:

- as mensagens produzidas pelo sistema, em resposta às ações dos usuários;
- o sistema de ajuda on-line, e
- a documentação fornecida com o sistema.

O projeto de informações úteis e esclarecedoras para os usuários deve ser levado seriamente em conta e deve estar sujeito ao mesmo processo de qualidade que os projetos ou os programas. Os gerentes devem dedicar tempo e esforço suficiente para o projeto de mensagens, podendo ser necessário envolver redatores profissionais e artistas gráficos nesse processo. Ao projetar mensagens de erro ou texto de ajuda, os fatores mostrados na Figura 15.12 devem ser considerados.

15.4.1 Mensagens de erro

A primeira impressão que os usuários podem ter de um sistema de software é sobre as mensagens de erro do sistema. Usuários inexperientes podem começar o trabalho e cometer

Figura 15.12 ▶

Fatores de projeto na redação de mensagens.

Fator	Descrição
Contexto	O sistema de orientação ao usuário deve estar ciente do que o usuário está fazendo e deve ajustar a mensagem emitida ao contexto atual.
Experiência	À medida que os usuários se familiarizam com um sistema, eles ficam irritados com mensagens longas e complexas. Contudo, os iniciantes acham difícil compreender declarações curtas e concisas sobre o problema. O sistema de orientação ao usuário deve fornecer ambos os tipos de mensagens e permitir que o usuário controle a sua concisão.
Nível de habilidade	As mensagens devem ser ajustadas às habilidades do usuário, como também a sua experiência. As mensagens para diferentes classes de usuários podem ser expressas de diferentes maneiras, dependendo da terminologia que seja familiar ao leitor.
Estilo	As mensagens devem ser positivas, e não negativas. Elas devem utilizar o modo ativo, em vez do modo passivo de expressão. Elas nunca deverão ser ofensivas ou engraçadas.
Cultura	Sempre que possível, o projetista de mensagens deve estar familiarizado com a cultura do país em que o sistema é vendido. Existem distintas diferenças de cultura entre a Europa, a Ásia e a América. Uma mensagem adequada para uma cultura pode ser inaceitável em outra.

um erro inicial e precisarão compreender imediatamente a mensagem de erro resultante. Isso pode ser difícil o suficiente para os engenheiros de software habilitados. Freqüentemente, isso é impossível para os usuários de sistema inexperientes ou casuais.

O conhecimento e a experiência dos usuários devem ser previstos, quando se projetam mensagens de erro. Por exemplo, digamos que o usuário do sistema seja uma enfermeira, em uma unidade de tratamento intensivo de um hospital. O monitoramento do paciente está sendo realizado por um sistema de computador. Para visualizar o estado atual do paciente (batimento cardíaco, temperatura etc.), o usuário do sistema seleciona 'display' em um menu e fornece o nome do paciente, como mostra a Figura 15.13.

Nesse caso, digamos que o nome do paciente fosse Pates, em vez de Bates, de modo que a inserção do nome pela enfermeira não pudesse ser reconhecida. O sistema gera uma mensagem de erro. As mensagens de erro devem sempre ser educadas, concisas, consistentes e construtivas. Elas não podem ser abusivas e não devem estar associadas a 'beeps' ou outros ruídos que possam constranger o usuário. Sempre que possível, a mensagem deve sugerir como o erro pode ser corrigido. A mensagem de erro deve estar associada a um sistema de ajuda on-line sensível ao contexto.

A Figura 15.14 mostra exemplos de mensagens de erro adequadas e inadequadas. A mensagem da esquerda está mal formulada; ela é negativa (acusa o usuário de cometer um erro), não está ajustada ao nível de habilidade e experiência do usuário e não leva em conta as informações do contexto. Ela não sugere como a situação pode ser solucionada. Ela utiliza termos específicos do sistema (ID do paciente), em vez de linguagem orientada ao usuário. A mensagem do lado direito é melhor. Ela é positiva, indica que o problema é do sistema, e não do usuário. Ela identifica o problema nos termos utilizados por enfermeiras e oferece um meio fácil de corrigir o erro, pressionando um simples botão. O sistema de ajuda está disponível, caso seja necessário.

Figura 15.13 ▶

Entrada do nome de um paciente por uma enfermeira.

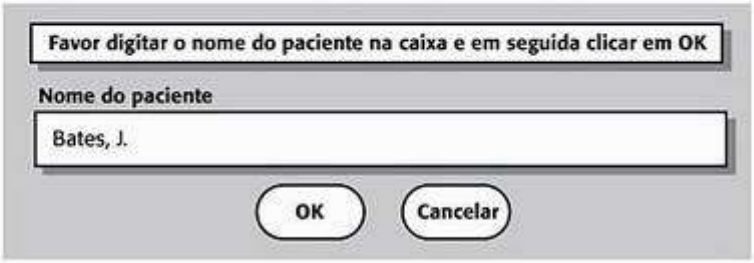
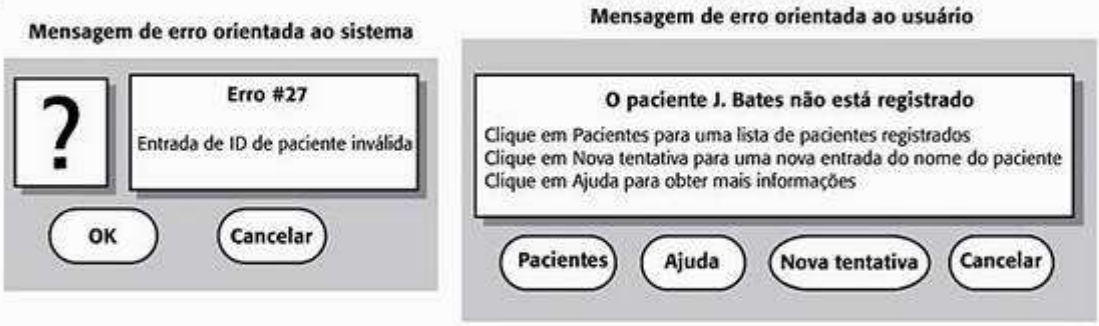


Figura 15.14 ▶

Mensagens orientadas ao usuário e ao sistema.



15.4.2 Projeto de sistema de ajuda

Quando se apresenta aos usuários uma mensagem de erro, eles não compreendem e procuram informações no sistema de ajuda. Esse é um exemplo de *ajuda!*, que significa 'ajude-me, estou com dificuldades'. Outro meio de fazer um pedido de ajuda é *ajuda?*, que significa 'ajude-me, preciso de informações'. Diferentes recursos de sistema e estruturas de mensagens podem ser necessários para fornecer esses diferentes tipos de ajuda.

Os sistemas de ajuda devem fornecer uma série de diferentes pontos de entrada de usuário (Figura 15.15). Esses pontos devem permitir que o usuário entre no sistema de ajuda, na parte superior da hierarquia de mensagens, e busque a informação. Como alternativa, os usuários podem entrar no sistema de ajuda para obter a explicação de uma mensagem de erro ou podem pedir a explicação de determinado comando da aplicação.

Todos os sistemas de ajuda abrangentes têm uma estrutura de rede complexa, em que cada esquema de informação de ajuda pode se referir a diversos outros esquemas de informação. De modo geral, a estrutura dessa rede é hierárquica, com ligações cruzadas, como mostra a Figura 15.15. As informações genéricas ficam no topo da hierarquia e as informações detalhadas, na parte inferior.

Podem surgir problemas com os sistemas de ajuda quando os usuários entram na rede depois de cometer um engano e, então, navegam na rede. Em pouco tempo, eles podem ficar

Figura 15.15 ▶

Pontos de entrada em um sistema de ajuda.

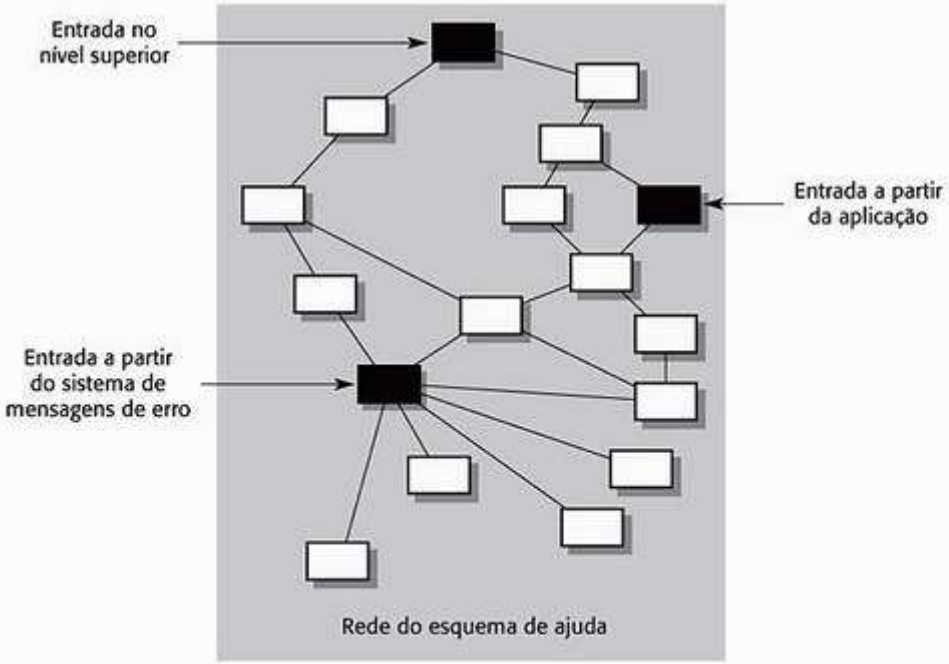
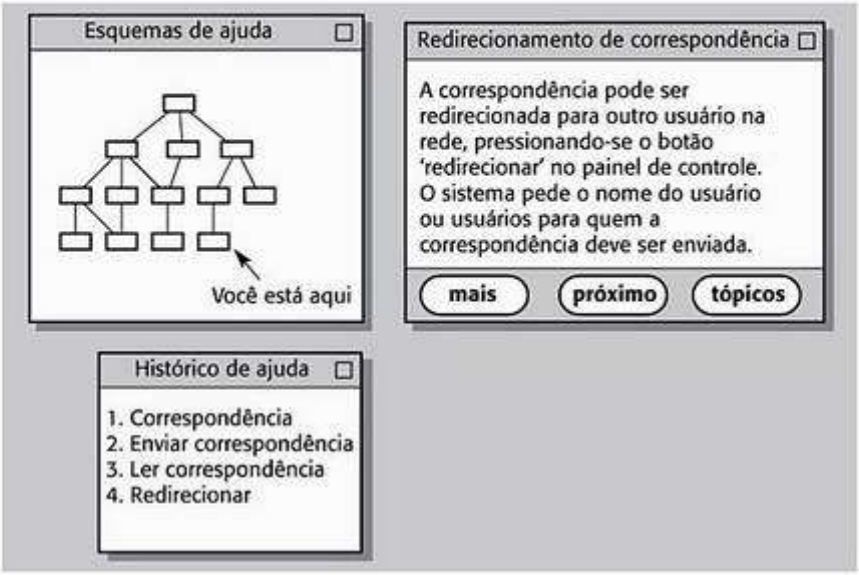


Figura 15.16

Janelas do sistema de ajuda.



totalmente perdidos, precisando abandonar a sessão e começar de novo em algum ponto conhecido da rede.

Exibir informações de ajuda em várias janelas pode ajudar a aliviar essa situação. A Figura 15.16 mostra um display de tela em que há três janelas de ajuda. No entanto, o espaço da tela é sempre limitado e o projetista tem de estar ciente de que exibir telas extras pode ocultar outras informações, as quais são importantes.

O texto em um sistema de ajuda deve ser preparado com a ajuda de especialistas em aplicações. O esquema de ajuda não deve simplesmente ser uma reprodução do manual do usuário, uma vez que as pessoas lêem textos em papel e em telas de maneiras diferentes. O texto propriamente dito, seu layout e estilo, têm de ser cuidadosamente elaborados, a fim de garantir que ele seja legível em uma janela relativamente pequena. Na Figura 15.16, o esquema de ajuda ('Redirecionamento de correspondência') é relativamente curto. Não se deve sobrecarregar o usuário com informações em nenhum dos esquemas. Três botões são fornecidos no esquema de ajuda para pedir mais informações, para passar para o próximo esquema de ajuda e para ativar uma lista de tópicos em que a ajuda está disponível.

A janela de 'histórico' mostra os esquemas que já foram consultados. Deve ser possível retornar a esses esquemas selecionando um item dessa lista. A janela de navegação é um 'mapa' gráfico da rede do sistema de ajuda. A posição atual nesse mapa deve ser destacada pelo uso de cores, sombreado ou, nesse caso, por uma anotação.

Os usuários devem ser capazes de se transferir para outro esquema, selecionando a partir do esquema que estão lendo, de um esquema na 'janela de histórico' para ler novamente ou refazer suas etapas ou de um nó no 'mapa' da rede para mover para aquele nó.

Os sistemas de ajuda podem ser implementados como um conjunto de páginas vinculadas na Web ou utilizar um sistema genérico de hipertexto, que possa ser integrado com aplicações. A hierarquia pode ser facilmente percorrida por meio da seleção de partes de uma mensagem que estejam destacadas como links. Os sistemas da Web têm a vantagem de ser de fácil implementação e não exigir nenhum software especial. Contudo, pode ser difícil vinculá-los à aplicação, a fim de fornecer ajuda sensível ao contexto.

15.4.3 Documentação do usuário

A documentação do usuário não é estritamente parte do projeto de interface com o usuário, mas é uma boa prática projetar o apoio da ajuda on-line em conjunto com a documen-

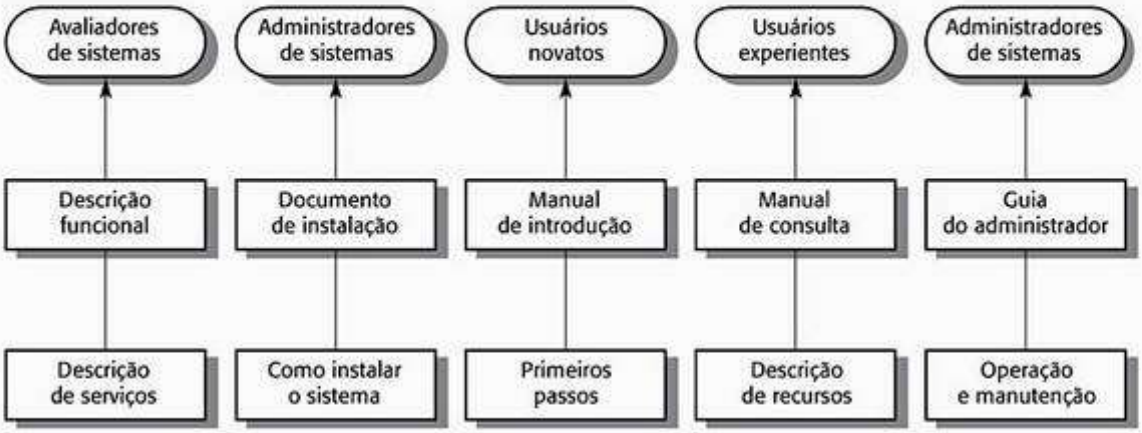


Figura 15.17 ▲

Tipos de documentos produzidos para dar apoio aos usuários.

tação em papel. Os manuais de sistemas devem fornecer informações mais detalhadas do que a ajuda on-line e devem ser projetados para serem utilizados por diferentes classes de usuários finais do sistema.

Para satisfazer a essas diferentes classes de usuários e diferentes níveis de especialidade dos usuários, existem, pelo menos, cinco documentos (ou, talvez, capítulos em um único documento), que devem ser entregues com um sistema de software (Figura 15.17).

Esses documentos são:

1. *Uma descrição funcional,* que deve descrever, muito brevemente, os serviços que o sistema fornece. Os usuários devem poder ler esse documento com um manual de introdução e decidir se o sistema é o que eles necessitam.

2. *Um documento de instalação,* que deve fornecer detalhes sobre como instalar o sistema. Ele deve descrever os discos nos quais o sistema é fornecido, os arquivos nesses discos e a configuração mínima de hardware exigida. Deve trazer instruções de instalação e detalhes de como configurar arquivos dependentes de configuração.

3. *Uma manual de introdução,* que deve apresentar uma introdução informal sobre o sistema, descrevendo sua utilização 'normal'. Ele deve descrever como iniciar o uso do sistema e como os usuários finais podem utilizar os recursos comuns do sistema. Deve ser bem ilustrado com exemplos e incluir informações sobre como fazer a recuperação a partir de erros e reiniciar o trabalho. Carroll (1992) sugere que o enfoque na recuperação a partir de erros e a minimização das informações que os usuários precisam ler, eliminando as redundâncias, são maneiras eficazes de organizar os manuais de introdução.

4. *Um manual de consulta,* que deve descrever as facilidades do sistema e sua utilização, fornecer uma lista de mensagens de erro e suas causas possíveis e descobrir como fazer a recuperação a partir dos erros detectados.

5. *Um guia do administrador* pode ser fornecido em alguns tipos de sistemas. Ele deve descrever as mensagens geradas quando o sistema interage com outros sistemas e explicar como reagir a essas mensagens. Se o hardware do sistema estiver envolvido, ele deverá explicar como reconhecer e reparar problemas relacionados ao hardware, como conectar novos periféricos, e assim por diante.

Além de manuais, outros documentos de fácil utilização podem ser fornecidos. Um cartão de consulta rápida, que relacione os recursos disponíveis do sistema e como utilizá-los, é particularmente conveniente para os usuários experientes.

15.5 Avaliação da interface

A avaliação da interface é o processo de analisar a facilidade de uso dessa interface e verificar se ela cumpre os requisitos do usuário. Portanto, ela deve ser parte do processo normal de verificação e validação para sistemas de software. Neilsen (1993), em seu livro sobre engenharia da facilidade de utilização, traz um bom capítulo sobre esse tópico.

Idealmente, uma avaliação deve ser conduzida em relação a uma especificação de facilidade de uso, com base em atributos dessa facilidade, como mostra a Figura 15.18. Podem ser criadas medidas para esses atributos de facilidade de uso. Por exemplo, uma medida de facilidade de aprendizado pode estabelecer que um operador familiarizado com o trabalho deve ser capaz de utilizar 80 por cento da funcionalidade do sistema, depois de três horas de treinamento. Contudo, é mais comum especificar a facilidade de uso (se ela realmente for especificada) qualitativamente, em vez de utilizar medidas. Os projetistas de interfaces, portanto, devem fazer uso de seu julgamento e de sua experiência na avaliação da interface.

A avaliação sistemática de um projeto de interface com o usuário pode ser um processo dispendioso, que envolve cientistas bem-informados e projetistas gráficos. Ela pode envolver o projeto e a realização de uma série estatisticamente grande de experimentos, com usuários típicos em laboratórios especialmente construídos, preparados com equipamentos de monitoramento. Uma avaliação da interface com o usuário, dessa maneira, é economicamente irrealista para os sistemas desenvolvidos por pequenas organizações com recursos limitados.

Existe uma série de técnicas mais simples e menos dispendiosas de avaliação da interface com o usuário, que podem identificar determinadas deficiências do projeto de interface com o usuário:

1. questionários, que coletam informações sobre o que os usuários pensam a respeito da interface;
2. observação dos usuários que trabalham com o sistema e 'pensam em voz alta' sobre como eles estão tentando utilizar o sistema para realizar algumas tarefas;
3. registros de uso típico de sistema em núcleo, e
4. inclusão de software que coleta informações sobre os recursos mais utilizados e os erros mais comuns.

Pesquisar os usuários com o uso de um questionário é uma maneira relativamente barata de avaliar uma interface. As questões devem ser precisas e não genéricas. Não adianta fazer

Figura 15.18 ▶

Atributos da facilidade de uso.

Atributo	Descrição
Facilidade de aprendizado	Quanto tempo leva um novo usuário para se tornar produtivo com o sistema?
Velocidade de operação	Em que grau a resposta do sistema combina com a prática de trabalho do usuário?
Robustez	Qual é o nível de tolerância do sistema aos erros dos usuários?
Facilidade de recuperação	Com que eficiência o sistema se recupera a partir de erros cometidos pelos usuários?
Facilidade de adaptação	Até que ponto o sistema está integrado a um único modelo de trabalho?

perguntas como "Por favor, comente sobre a facilidade de uso da interface", uma vez que as respostas provavelmente variarão tanto que nenhuma tendência em comum emergirá. Em vez disso, faça perguntas específicas, como: "Por favor, dê sua nota sobre a facilidade de compreensão das mensagens de erro, em uma escala de 1 a 5. Nessa escala, a nota 1 significa que as mensagens são muito claras e 5 significa que elas são incompreensíveis". Não só essas perguntas são mais fáceis de ser respondidas, como também elas provavelmente fornecerão informações úteis para melhorar a interface.

Deve-se pedir que os usuários 'dêem nota' a sua própria experiência e a seu conhecimento, ao preencher o questionário. Isso permite ao projetista verificar se usuários com qualquer nível específico de conhecimento têm problemas com a interface. Os questionários poderão ser utilizados até mesmo antes que qualquer sistema executável esteja disponível, se um modelo em papel da interface for elaborado e avaliado.

A avaliação com base na observação simplesmente envolve observar os usuários quando eles utilizam um sistema, notando os recursos utilizados, os erros cometidos etc. Essa opção pode ser complementada por sessões de 'pensar em voz alta', em que os usuários falam sobre o que estão tentando fazer, como eles compreendem o sistema e como estão tentando utilizar o sistema para realizar seus objetivos.

Um equipamento de vídeo de custo relativamente baixo significa que a observação direta pode ser mantida pela gravação de sessões com o usuário, para análise posterior. Uma análise completa de vídeo é dispendiosa e requer um conjunto de avaliação especialmente equipado, com diversas câmeras focalizadas sobre o usuário e na tela. Contudo, o registro em vídeo de operações selecionadas do usuário pode ser útil para detectar problemas. Outros métodos de avaliação devem ser utilizados para descobrir que operações causam dificuldades ao usuário.

A análise de gravações permite que o projetista constate se a interface requer muito movimento das mãos (um problema com alguns sistemas é que os usuários devem movimentar as mãos do teclado para o mouse) e verificar se são necessários movimentos não naturais dos olhos. Uma interface que requer muitas mudanças de foco pode significar que o usuário cometerá mais erros e perderá de vista partes do display.

Instrumentalizar o código para coletar estatísticas de utilização permite a melhoria da interface, de uma série de maneiras. As operações mais comuns podem ser detectadas. A interface pode ser reorganizada, de modo que seja mais rápido fazer a seleção. Por exemplo, se são utilizados menus instantâneos ou menus em cascata, as operações mais freqüentes devem estar no topo do menu e as operações destrutivas, mais abaixo. O código instrumentalizado também permite que os comandos com tendência a causar erros sejam detectados e modificados.

Por fim, uma maneira fácil de responder aos usuários pode ser fornecida, ao equipar cada programa com um comando rápido, que o usuário pode utilizar para fornecer mensagens à pessoa responsável pela manutenção da ferramenta. Isso faz com que os usuários sintam que suas opiniões estão sendo consideradas. O projetista da interface e outros engenheiros podem obter feedback rápido sobre problemas individuais.

Nenhuma dessas abordagens relativamente simples para a avaliação da interface com o usuário é totalmente garantida e é provável que elas não detectem todos os problemas de interface com o usuário. Contudo, as técnicas podem ser utilizadas com um grupo de voluntários, sem gastar muitos recursos, antes do lançamento do sistema. Muitos dos piores problemas de projeto de interface com o usuário podem ser descobertos e corrigidos.

PONTOS-CHAVE

- O processo de projeto de interface deve ser centrado no usuário. Uma interface deve interagir com os usuários em seus termos, ser lógica e consistente e incluir recursos para ajudar os usuários com o sistema e para que eles possam conseguir a recuperação a partir de erros.
- Entre os estilos de interação com um sistema de software estão a manipulação direta, os sistemas de menus, o preenchimento de formulários, a linguagem de comando e a linguagem natural.
- O display de informações gráficas deve ser utilizado quando se pretende apresentar tendências e valores aproximados. O display digital deve ser utilizado somente quando a precisão é necessária.
- As cores devem ser utilizadas com equilíbrio e consistentemente nas interfaces com o usuário. Os projetistas devem levar em conta o fato de que um grande número de pessoas não consegue identificar cores.
- Os sistemas de ajuda ao usuário devem fornecer dois tipos de ajuda: 'Ajuda!', que significa "ajude-me, estou com dificuldades", e 'Ajuda?', que quer dizer "ajude-me, preciso de informações".
- As mensagens de erro não devem sugerir que o usuário é o culpado. Elas devem oferecer sugestões sobre como reparar o erro e fornecer uma conexão ao sistema de ajuda.
- A documentação do usuário deve incluir manuais para principiantes e manuais de consulta. Devem ser fornecidos documentos separados para os administradores de sistemas.
- A especificação de sistemas deve incluir, sempre que possível, valores quantitativos para os atributos de facilidade de uso e o processo de avaliação deve verificar o sistema quanto a esses requisitos.

LEITURAS SUGERIDAS

Human–computer interaction, 2. ed. Um bom texto geral, cujos pontos fortes são o enfoque nas questões de projeto e o trabalho cooperativo. (A. Dix, J. Finlay, G. Abowd e R. Beale, 1998, Prentice Hall.)

Designing the user interface. Essa é a última revisão do que foi, provavelmente, o primeiro livro nessa área. Um livro excelente, bastante abrangente. (B. Shneiderman, 1998, Addison Wesley.)

Developing user interfaces. Esse livro focaliza a implementação, em vez do projeto de interfaces com o usuário. É, portanto, um bom complemento para outros textos que se concentram no projeto de interfaces. (D. Olsen, 1998, Morgan Kaufmann.)

EXERCÍCIOS

15.1 Foi sugerido, na Seção 15.1, que os objetos manipulados por usuários devem ser obtidos a partir de seu domínio, em vez do domínio de computador. Sugira objetos apropriados para os seguintes tipos de usuários e sistemas:

- um assistente de almoxarifado que utiliza um catálogo automatizado de peças;
- um piloto de linha aérea que utiliza um sistema de monitoramento de segurança de aeronaves;
- um gerente que manipula um banco de dados financeiro, e
- um policial que utiliza um sistema de controle de viatura.

15.2 Sugira situações em que seja insensato ou impossível fornecer uma interface com o usuário que seja consistente.

15.3 Que fatores devem ser levados em conta ao se projetar uma interface com base em menus para sistemas *walk-up*, como máquinas ATM de bancos? Faça um comentário crítico por escrito sobre a interface de um ATM que você utiliza.

15.4 Sugira modos pelos quais a interface com o usuário para um sistema de comércio eletrônico (*e-commerce*), como uma livraria ou uma loja de música on-line, possa ser adaptada para usuários com algum tipo de dificuldade visual ou problemas de controle muscular.

15.5 Discuta as vantagens do display de informações gráficas e sugira quatro aplicações em que seja mais apropriado utilizar o display gráfico, em vez do digital, para exibir informações numéricas.

15.6 Quais são as diretrizes a serem seguidas quando são utilizadas cores em uma interface com o usuário? Sugira como as cores podem ser utilizadas de maneira mais eficaz na interface de um sistema de aplicação que você utilize.

15.7 Escreva um breve conjunto de diretrizes para os projetistas de sistemas de orientação aos usuários.

15.8 Considere as mensagens de erro produzidas pelo MS-Windows, Unix, MacOS ou algum outro sistema operacional. Sugira como elas podem ser melhoradas.

15.9 Projete um questionário para coletar informações sobre a interface com o usuário de alguma ferramenta (como um processador de textos) com a qual você esteja familiarizado. Se possível, distribua esse questionário para certo número de usuários e tente avaliar os resultados. O que esses resultados informam sobre o projeto de interface com o usuário?

15.10 Discuta se é ético instrumentalizar um software sem dizer aos usuários finais que o trabalho deles está sendo monitorado.

15.11 Quais questões éticas os projetistas de interface com o usuário poderão enfrentar se tentarem conciliar as necessidades dos usuários finais de um sistema com as necessidades da organização que está pagando para que o sistema seja desenvolvido?

PARTE QUATRO

Sistemas críticos

PÁGINA EM BRANCO

16 Confiança

Objetivos

O objetivo deste capítulo é introduzir a noção de confiança e sua importância para os sistemas críticos. Depois de ler este capítulo, você poderá:

- compreender as quatro dimensões da confiança: disponibilidade, confiabilidade, segurança e proteção;
- ter noção de sistemas críticos, em que uma falha de sistema pode causar graves conseqüências humanas e econômicas, e
- compreender que, para alcançar a confiança, é preciso evitar enganos durante o desenvolvimento de um sistema, detectar e eliminar erros quando o sistema estiver em uso e limitar os danos causados por falhas operacionais.

Sumário

16.1 Sistemas críticos
16.2 Disponibilidade e confiabilidade
16.3 Segurança
16.4 Proteção

Todos nós estamos familiarizados com os problemas das falhas de sistemas de computador. Por razões que não são óbvias, algumas vezes os sistemas computacionais entram em colapso e falham em disponibilizar os serviços que foram pedidos. Os programas que correm nesses computadores podem não operar como esperado e, ocasionalmente, corromper os dados gerenciados pelo sistema. Aprendemos a conviver com essas falhas e poucos de nós confiam totalmente nos microcomputadores que costumamos utilizar.

A confiança em um sistema de computador é uma propriedade do sistema, que equivale à sua integridade. A integridade significa, essencialmente, o grau de confiança dos usuários de que o sistema operará como eles esperam e não 'falhará' em uso normal. Essa propriedade não pode ser expressa numericamente, mas utilizamos termos relativos, como 'não confiável', 'muito confiável' e 'ultraconfiável', para refletir os diferentes graus de confiança que podemos ter em um sistema.

Integridade e utilidade não são, naturalmente, a mesma coisa. O processador de textos que foi utilizado para escrever este livro não é, na minha opinião, um sistema muito confiável, mas é muito útil. Contudo, para refletir essa falta de confiança no sistema, muitas vezes meu trabalho é salvo e várias cópias de backup são mantidas. Portanto, a falta de confiança no sistema é compensada com ações que limitam os danos que possam resultar de alguma falha do sistema.

Existem quatro dimensões principais de confiança, como mostra a Figura 16.1:

1. *Disponibilidade* Informalmente, a disponibilidade de um sistema é a probabilidade de que ele estará rodando e em condições de disponibilizar serviços úteis a qualquer momento.
2. *Confiabilidade* Informalmente, a confiabilidade de um sistema é a probabilidade, por determinado período de tempo, de que o sistema disponibilizará serviços de maneira correta, conforme o usuário espera.
3. *Segurança* Informalmente, a segurança de um sistema é um julgamento da probabilidade de o sistema causar danos às pessoas ou a seu ambiente.
4. *Proteção* Informalmente, a proteção de um sistema é um julgamento da probabilidade de o sistema poder resistir à invasão acidental ou deliberada.

Como será discutido no Capítulo 17, disponibilidade e confiança são, em essência, probabilidades e, portanto, podem ser expressas quantitativamente. Segurança e proteção são julgamentos feitos com base em evidências sobre o sistema. Eles raramente são expressos como valores numéricos, mas podem ser expressos em termos de níveis de integridade. Portanto, a segurança de um sistema de nível 1 é menor do que a segurança de um sistema de nível 2, que é menor do que a segurança de um sistema de nível 3, e assim por diante.

Por causa das despesas adicionais de projeto, implementação e validação, aumentar a confiabilidade de um sistema pode elevar drasticamente os custos de desenvolvimento. A Figura

Figura 16.1 ▶
Dimensões da confiança.

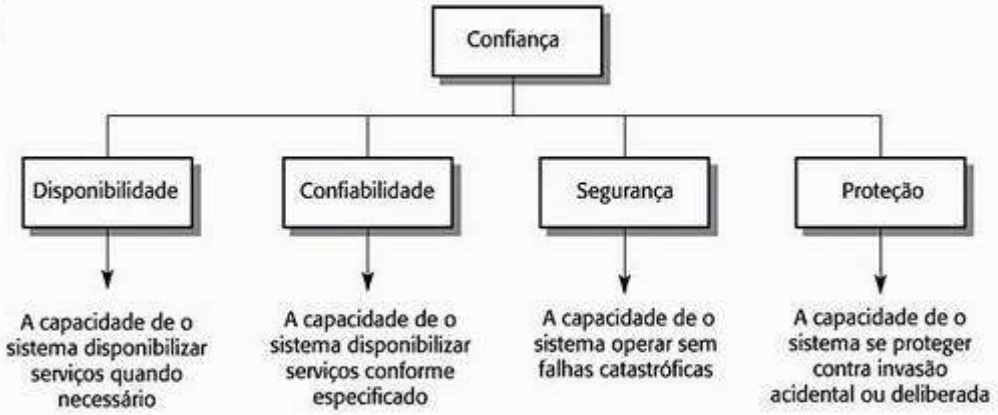
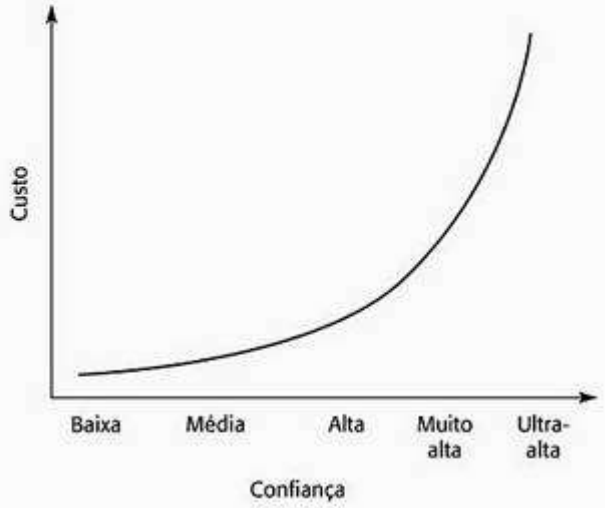


Figura 16.2 ▶

Curva de custo/confiança.



16.2 mostra a relação entre os custos e as melhorias adicionais da confiança. Esse gráfico vale para todos os aspectos da confiança – confiabilidade, disponibilidade, segurança e proteção. Devido à natureza exponencial dessa curva de custo/confiança, não é possível provar que um sistema seja 100 por cento confiável, uma vez que os custos de garantia de confiança seriam, então, infinitos.

De modo geral, é verdadeiro afirmar que altos níveis de confiança somente podem ser alcançados à custa do desempenho do sistema. Um software confiável inclui código extra, freqüentemente redundante, para realizar a verificação necessária dos estados de sistema excepcionais e para possibilitar a recuperação a partir de falhas do sistema. Isso reduz o desempenho do sistema e aumenta a quantidade de armazenamento requerida pelo software. Contudo, existe uma série de razões pelas quais a confiança é, em geral, um atributo mais importante do que o desempenho:

1. *De maneira geral, os sistemas que não são confiáveis, não apresentam segurança ou são inseguros não são utilizados* Se um sistema não for confiável para seus usuários, eles se recusarão a utilizá-lo. Além disso, eles podem também se recusar a utilizar outros produtos da mesma empresa que produziu esse sistema, uma vez que esses produtos também podem não ser confiáveis.

2. *Os custos da falha de um sistema podem ser enormes* Para algumas aplicações, como um sistema de controle de reator ou um sistema de navegação de aeronave, o custo da falha do sistema é maior, em termos de ordens de grandeza, do que o custo do sistema de controle.

3. *É difícil readequar a confiança* De modo geral, é possível ajustar um sistema ineficiente, porque a maior parte do tempo de execução é gasta em pequenas seções do programa. Um sistema que não é confiável é mais difícil de passar por melhorias, uma vez que a falta de integridade tende a estar distribuída por todo o sistema.

4. *Freqüentemente, é possível compensar a falta de desempenho do sistema* Às vezes, os usuários podem adequar seu trabalho para atender a um sistema com desempenho inadequado. Por outro lado, uma falha na confiança geralmente surpreende o usuário. O software não confiável pode violar o sistema e os dados do usuário, sem aviso, e pode falhar, trazendo conseqüências sérias, que somente se manifestarão posteriormente.

5. *Sistemas não confiáveis podem causar a perda de informações* É muito dispendioso coletar e manter dados; algumas vezes, eles podem valer mais do que o sistema computacional no qual são processados. É preciso empregar muito esforço e dinheiro para duplicar dados valiosos e preservá-los, para que não sejam corrompidos.

A confiabilidade do produto de software é influenciada pelo processo de software utilizado para desenvolver o produto. Um processo repetitivo, orientado no sentido de evitar defeitos, poderá provavelmente desenvolver um sistema confiável. Contudo, não existe uma relação simples entre a qualidade do processo e a qualidade do produto. A adequação a determinado processo não permite que seja feita qualquer avaliação quantitativa da qualidade do produto. As relações entre qualidade de processo e de produto serão discutidas nos capítulos 24 e 25, que abordam o gerenciamento de qualidade e a melhoria de processo.

16.1 Sistemas críticos

A falha de muitos sistemas controlados por software causa inconveniência, mas não danos sérios e prolongados. Contudo, há certos sistemas em que as falhas podem resultar em perdas econômicas significativas, danos físicos ou ameaças à vida humana. Esses sistemas, em geral, são chamados de sistemas críticos. A confiança é um atributo essencial dos sistemas críticos e todos os aspectos da confiança (disponibilidade, confiabilidade, segurança e proteção) podem ser importantes. Atingir um alto nível de confiança é, normalmente, o requisito mais importante para os sistemas críticos.

Existem três tipos principais de sistemas críticos:

1. *Sistema crítico de segurança* Um sistema cuja falha pode resultar em ferimentos, na perda de uma vida ou em grande dano ambiental. Um exemplo de sistema de segurança crítica é o sistema de controle para uma fábrica de produtos químicos.
2. *Sistema crítico de missão* Um sistema cuja falha pode resultar na falha de alguma atividade orientada a metas. Um exemplo de sistema de missão crítica é o sistema de navegação de aeronaves.
3. *Sistema crítico de negócios* Um sistema cuja falha pode resultar no fracasso dos negócios que utilizam o sistema. Um exemplo de um sistema de negócios críticos é o sistema de contas de clientes em um banco.

Os custos da falha de um sistema crítico são, freqüentemente, muito altos. Entre esses custos estão os custos diretos da falha, que podem requerer a substituição do sistema, e os custos indiretos, como os custos de litígio e as perdas de negócios, nos quais podem resultar se o sistema ficar indisponível. Os altos custos em potencial, relacionados a falhas, significam que a integridade das técnicas e dos processos de desenvolvimento, em geral, é mais importante do que os custos relativos à aplicação dessas técnicas.

Conseqüentemente, os sistemas críticos são em geral desenvolvidos com a utilização de técnicas bem testadas, em vez de técnicas novas, que não tenham sido submetidas à experimentação prática extensiva. Por exemplo, apenas recentemente as técnicas orientadas a objeto foram utilizadas para o desenvolvimento de sistemas críticos e muitos projetos de desenvolvimento desses sistemas ainda são baseados no projeto orientado a funções.

Contudo, as técnicas de engenharia de software que não são normalmente viáveis, em termos de custos, podem ser utilizadas para o desenvolvimento de sistemas críticos. Por exemplo, o uso de especificação e verificação formal de um programa em relação a suas especificações. Uma razão pela qual esse método é utilizado é o fato de que ele ajuda a reduzir a quantidade de testes requeridos. Para os sistemas críticos, os custos de verificação e de validação são, em geral, muito altos e podem consumir mais de 50 por cento do total dos custos de desenvolvimento.

Embora este livro esteja centrado na engenharia de software e os capítulos se concentrem em questões de software, a confiança é realmente um conceito de sistema. Quando se con-

sidera a confiança em sistemas críticos, existem três tipos de 'componentes de sistema' que estão sujeitos a falhas:

1. hardware de sistema, que pode falhar devido a enganos em seu projeto, porque os componentes falham como resultado de erros de fabricação ou porque os componentes de hardware chegaram ao final de sua vida útil;
2. software de sistema, que pode falhar devido a enganos em sua especificação, seu projeto ou sua implementação, e
3. operadores humanos do sistema, que podem falhar ao operar o sistema corretamente.

Portanto, se o objetivo é melhorar a confiança em um sistema, é preciso considerar todos esses aspectos e suas interações. Esse aspecto é ilustrado em alguns exemplos desenvolvidos em outros capítulos, nesta parte do livro.

16.1.1 Um simples sistema crítico de segurança

Para ilustrar os capítulos sobre os sistemas críticos, é utilizado o exemplo de um sistema médico simples. É um sistema de fornecimento de insulina para o controle de diabete. Espera-se que a maioria dos leitores tenha algum conhecimento desse problema e de seu tratamento. Esse sistema já foi apresentado no Capítulo 9 e, formalmente, parte de sua funcionalidade foi especificada.

A diabete é uma condição relativamente comum, em que o corpo humano é incapaz de produzir quantidades suficientes de um hormônio chamado insulina. A insulina metaboliza a glicose no sangue. O tratamento convencional da diabete envolve injeções regulares de insulina geneticamente produzida.

O problema com esse tratamento é que o nível de insulina no sangue não depende do nível de glicose no sangue, mas é uma função do momento em que a injeção da insulina foi aplicada. Isso pode levar a níveis muito baixos de glicose no sangue (se houver muita insulina) ou a níveis muito altos de açúcar no sangue (se houver muito pouca insulina). Pouco açúcar no sangue é, em curto prazo, uma condição mais séria, uma vez que pode resultar em disfunção temporária do cérebro e, por último, em inconsciência e morte. Em longo prazo, altos níveis contínuos de açúcar no sangue podem levar a danos na vista e no fígado e problemas cardíacos.

Avanços atuais no desenvolvimento de sensores miniaturizados significam que, atualmente, é possível desenvolver sistemas automatizados de administração de insulina. Esses sistemas monitoram os níveis de açúcar no sangue e administram uma dose apropriada de insulina, quando necessário. Já existem sistemas que administram a insulina dessa maneira para tratamento de pacientes em hospitais. No futuro, poderá ser possível para muitos diabéticos ter esses sistemas permanentemente conectados ao corpo.

Figura 16.3 ▶

Estrutura da bomba de insulina.

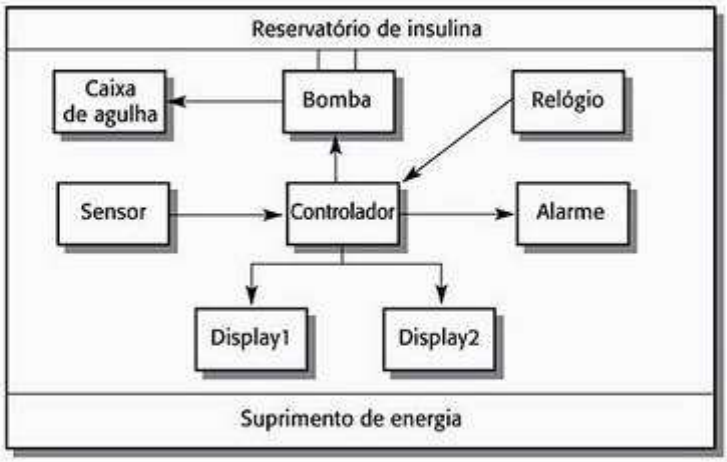


Figura 16.4

Modelo de fluxo de dados da bomba de insulina.

Um sistema de administração de insulina pode funcionar com o uso de um microssensor inserido no paciente, a fim de medir alguns parâmetros sanguíneos, proporcionais ao nível de açúcar. Essas informações são, então, remetidas para o controlador da bomba, que calcula o nível do sangue, julga a quantidade de insulina necessária e envia sinais para uma bomba em miniatura, que ministra a insulina por meio de uma agulha permanentemente conectada. Os sistemas de administração de insulina são geralmente controlados por software. A Figura 16.3 mostra os componentes e a organização da bomba de insulina. A Figura 16.4 é um modelo de fluxo de dados que ilustra como uma entrada de nível de açúcar no sangue é transformada em uma seqüência de comandos de controle da bomba.

Três dimensões de confiança são aplicadas a esse sistema de administração de insulina:

1. *Disponibilidade* É importante que o sistema esteja disponível para fornecer insulina quando necessário.
2. *Confiabilidade* É importante que o sistema opere de modo confiável e que forneça a quantidade correta de insulina para contrabalançar o nível atual de açúcar no sangue.
3. *Segurança* Uma falha no sistema pode, em princípio, fazer com que doses excessivas de insulina sejam ministradas, o que ameaçaria a vida do paciente. É importante que esse tipo de falha de sistema não ocorra.

Será explicado nos capítulos posteriores como essas dimensões de confiança podem ser especificadas e validadas.

16.2 Disponibilidade e confiabilidade

Nesta seção, são discutidas duas dimensões da confiança que estão estreitamente relacionadas: a disponibilidade e a confiabilidade. A disponibilidade de um sistema é a probabilidade de que ele poderá fornecer serviços a seus usuários, quando isso for solicitado. A confiabilidade, por sua vez, é a probabilidade de que os serviços do sistema sejam prestados conforme especificado. É claro que a confiabilidade implica a disponibilidade, porque, se um serviço especificado não for prestado, o sistema não estará se comportando de acordo com sua especificação.

No entanto, é útil fazer a distinção entre essas características, porque os requisitos de disponibilidade e confiabilidade podem ser diferentes. Por exemplo, alguns sistemas podem tolerar falhas até certo ponto freqüentes, desde que possam se recuperar com rapidez a partir dessas falhas. Eles, portanto, têm requisitos de confiabilidade relativamente pequenos. Os mes-

mos sistemas, contudo, podem ter altos níveis de requisitos de disponibilidade, por causa das expectativas dos usuários de serviços contínuos.

Um bom exemplo desse tipo de sistema é uma chave de comutação de telefones. Os usuários esperam um sinal de discar quando pegam o telefone; assim, o sistema tem altos requisitos de disponibilidade. Entretanto, se um defeito no sistema causar uma falha de conexão, isso será freqüentemente recuperável. As chaves de comutação geralmente incluem recursos de reparo que podem reiniciar o sistema e refazer a tentativa de conexão. Isso pode ser feito rapidamente e o usuário do telefone pode nem mesmo notar que ocorreu uma falha. Portanto, a disponibilidade, em vez da confiabilidade, é o principal requisito de confiança nessa situação.

Outra distinção entre essas características é que a disponibilidade não depende simplesmente do sistema em si, mas também do tempo necessário para reparar os defeitos que tornam o sistema indisponível. Portanto, se o sistema A falha uma vez por ano e o sistema B falha uma vez por mês, então A é mais confiável do que B. Contudo, se o sistema A leva três dias para reiniciar e o sistema B leva dez minutos para reiniciar, então a disponibilidade de B durante o ano é consideravelmente maior do que a de A. Os usuários prefeririam o sistema B ao sistema A.

A disponibilidade e a confiabilidade do sistema podem ser definidas mais precisamente como se segue:

1. *Confiabilidade* É a probabilidade de operação livre de falhas durante um tempo especificado, em um dado ambiente, para um propósito específico.
2. *Disponibilidade* É a probabilidade de um sistema, em determinado instante, ser operacional e capaz de fornecer os serviços requeridos.

Essas são definições cuidadosas desses termos e um dos problemas práticos no desenvolvimento de sistemas confiáveis é que nossas noções intuitivas de confiabilidade e disponibilidade são, na maioria das vezes, mais amplas do que essas definições limitadas. A definição de confiabilidade afirma que o ambiente em que o sistema é utilizado e o propósito para o qual é utilizado devem ser considerados, quando se pensa em sua confiabilidade. Portanto, as medições de confiabilidade em um ambiente não são necessariamente transferíveis para outro ambiente, em que o sistema é utilizado de outra maneira.

Por exemplo, a confiabilidade em um sistema de software pode ser medida em um ambiente de escritório, em que a maioria dos usuários não se interessa pela operação do software. Eles seguem as instruções para usá-lo e não tentam fazer experiências com o sistema. Em um ambiente universitário, a confiabilidade pode ser inteiramente diferente. Nesse caso, os estudantes exploram os limites do sistema e podem utilizá-lo de maneiras inesperadas. Isso pode resultar em falhas do sistema, que não ocorreram no ambiente mais restrito do escritório.

As percepções e os padrões humanos de uso também são importantes. Por exemplo, considere uma situação em que um carro apresenta um defeito em seu sistema de limpeza de pára-brisas, que resulta na falha intermitente dos limpadores de operar corretamente sob chuva pesada. A confiabilidade desse sistema, do ponto de vista de um motorista, depende de onde ele mora e de como utiliza o carro. Um motorista localizado em Seattle (região de clima úmido) será provavelmente mais afetado por essa falha do que um motorista em Las Vegas (que tem clima seco). A percepção do motorista de Seattle será de que o sistema não é confiável, enquanto o motorista de Las Vegas talvez nunca note o problema.

Outra dificuldade com essas definições é que elas não levam em conta a gravidade da falha ou as conseqüências da indisponibilidade. As pessoas, naturalmente, estão mais preocupadas com falhas de sistema que tenham conseqüências sérias e sua percepção da confiabilidade do sistema é influenciada por essas conseqüências. Por exemplo, é irritante quando um motor de automóvel falha logo depois que a partida é dada, mas inevitavelmente ele opera de maneira correta depois que é dada a partida outra vez. Contudo, isso não interfere na operação normal do carro e muitos motoristas não pensariam no fato de que ele não é

confiável. Por outro lado, a maioria dos motoristas pensará que um motor que falha uma vez por mês, por exemplo, quando em alta velocidade, não é confiável.

Uma definição estrita da confiabilidade relaciona-se com a implementação do sistema conforme sua especificação. Em outras palavras, o sistema se comportará de modo confiável, se seu comportamento for consistente com aquele definido na especificação. Contudo, uma causa comum da não-confiabilidade percebida é que a especificação do sistema não combina com as expectativas dos usuários do sistema. Infelizmente, muitas especificações são incompletas ou incorretas e fica por conta dos engenheiros de software interpretar como o sistema deveria se comportar. Como eles não são especialistas no domínio, eles podem, portanto, não implementar o comportamento que os usuários esperam.

Confiabilidade e disponibilidade são geralmente consideradas as dimensões mais importantes da confiança. Se um sistema não for confiável, será difícil garantir segurança ou proteção, uma vez que esses fatores podem estar comprometidos com a ocorrência de falhas de sistema. Se um sistema não está disponível, as perdas econômicas conseqüentes podem ser muito grandes. Softwares não confiáveis resultam em altos custos para os usuários finais. Os desenvolvedores de sistemas não confiáveis podem adquirir má reputação, devido à qualidade, e perder futuras oportunidades de negócios.

A confiabilidade é comprometida com a ocorrência de falhas do sistema. Pode se tratar de uma falha em fornecer um serviço, uma falha em realizar um serviço conforme especificado ou a realização de um serviço de maneira que ele não apresente segurança e proteção. Algumas dessas falhas são conseqüências de erros de especificação ou falhas em sistemas associados, como um sistema de telecomunicações. Contudo, muitas falhas são conseqüências de um comportamento errôneo do sistema, que deriva de defeitos no sistema. Quando se discute a confiabilidade, é útil distinguir entre os termos defeito, erro e falha. A definição desses termos encontra-se na Figura 16.5.

Defeitos de sistema não resultam necessariamente em erros de sistema, uma vez que o estado defeituoso pode ser transitório e corrigido antes da ocorrência de um comportamento errôneo. Os erros de sistema não resultam necessariamente em falhas do sistema, uma vez que o comportamento também pode ser transitório e não ter nenhum efeito observável, ou o sistema pode incluir uma proteção que assegure que o comportamento errôneo seja descoberto e corrigido antes que os serviços do sistema sejam afetados.

Essa distinção entre os diferentes termos mostrada na Figura 16.5 é útil porque ajuda a identificar três abordagens complementares, que podem ser utilizadas para melhorar a confiabilidade de um sistema:

1. *Evitar defeitos* São utilizadas técnicas de desenvolvimento que minimizam a possibilidade de erros e/ou os captam antes que eles resultem na introdução de defeitos no sistema. Entre os exemplos dessas técnicas estão evitar construções de linguagem de programação propensas a erros, como ponteiros, e o emprego da análise estática para detectar anomalias no programa, como será discutido no Capítulo 19.

2. *Detecção e exclusão de defeitos* O uso de técnicas de verificação e validação, que aumentam as chances de que defeitos sejam detectados e removidos antes que o sistema seja utilizado. Testes e depurações sistemáticos do sistema são um exemplo de técnica de detecção de defeitos.

3. *Tolerância a defeitos* O uso de técnicas que assegurem que os defeitos em um sistema não resultem em erros do sistema ou que assegurem que os erros do sistema não resultem em falhas. A incorporação de recursos de autoverificação em um sistema e a utilização de módulos redundantes de sistema são exemplos de técnicas de tolerância a falhas.

O desenvolvimento de sistemas tolerantes a defeitos será abordado no Capítulo 18, em que serão discutidas, também brevemente, algumas técnicas para evitar defeitos. As abordagens com base em processos para evitar defeitos serão analisadas no Capítulo 24 e a detecção de defeitos será discutida nos capítulos 19 e 20.

Figura 16.5 ▶

Terminologia da confiabilidade.

Termo	Descrição
Falha de sistema	Um evento que ocorre em algum momento, quando o sistema não fornece o serviço como é esperado por seus usuários.
Erro de sistema	Comportamento errôneo do sistema, quando o comportamento do sistema não atende à sua especificação.
Defeito de sistema	Um estado incorreto do sistema, ou seja, um estado do sistema que é inesperado pelos seus projetistas.
Erro ou engano humano	Comportamento humano que resulta na introdução de defeitos no sistema.

Os defeitos de software causam falhas de software quando o código com defeito é executado com um conjunto de entradas que expõem o defeito. O código funciona adequadamente para a maioria das entradas. A Figura 16.6, extraída de Littlewood (1990), mostra um sistema de software como um mapeamento de um conjunto de entradas para um conjunto de saídas. Um programa tem muitas entradas possíveis (por questões de simplicidade, combinações e seqüências de entradas são consideradas uma única entrada). O programa responde a essas entradas produzindo uma saída ou um conjunto de saídas.

Algumas dessas entradas (mostradas na elipse sombreada da Figura 16.6) provocam falhas de sistema, quando saídas errôneas são geradas pelo programa. A confiabilidade do software está relacionada à probabilidade de que, em uma execução do programa em particular, a entrada do sistema seja um membro do conjunto de entradas que provoquem uma saída errônea.

Existe uma relação complexa entre a confiabilidade observada no sistema e o número de defeitos de software latentes. Mills *et al.* (1987) mostraram que nem todos os defeitos de software apresentam igual probabilidade de provocar uma falha de software. De modo geral, existe uma série de membros de I_e que são mais prováveis de serem selecionados do que outros. Se essas entradas não fizerem com que partes defeituosas do software sejam executadas, então não haverá falhas. Portanto, a confiabilidade do programa depende principalmente do número de entradas que provocam saídas errôneas durante o uso normal do sistema. Defeitos que somente ocorrem em situações excepcionais têm pequeno efeito na confiabilidade do sistema.

A confiabilidade está relacionada à probabilidade de um erro ocorrer durante o uso operacional. Remover defeitos de software das partes do sistema que são raramente utilizadas faz pouca diferença real para a confiabilidade verificada. Mills *et al.* descobriram que, em

Figura 16.6 ▶

Um sistema como um mapeamento de entrada/saída.

Figura 16.7 ▶

Padrões de utilização de software.

seu software, a remoção de 60 por cento dos defeitos do produto levou a apenas 3 por cento de melhoria da confiabilidade. Isso foi confirmado em um estudo separado de erros em produtos de software da IBM. Adams (1984) notou que muitos defeitos nos produtos somente tinham probabilidade de provocar falhas depois de centenas ou milhares de meses de sua utilização.

Portanto, um programa pode conter defeitos conhecidos, mas pode, ainda, ser considerado confiável pelos usuários. Talvez eles nunca venham a selecionar uma entrada errônea e, assim, as falhas de programa nunca surgirão. Além disso, os usuários experientes com freqüência 'contornam' os defeitos de software que já são conhecidos como causadores de falhas. Eles deliberadamente evitam utilizar recursos do sistema que sabem que podem causar problemas. Reparar os defeitos desses recursos pode não fazer nenhuma diferença prática para a confiabilidade, do ponto de vista desses usuários.

Cada usuário de um sistema utiliza-o de diferentes maneiras. Os defeitos que afetam a confiabilidade do sistema para um usuário podem nunca ser revelados sob um modo diferente de trabalhar (Figura 16.7). Na Figura 16.7, o conjunto de entradas errôneas é correspondente à elipse sombreada na Figura 16.6. O conjunto de entradas produzido pelo Usuário 2 cruza com esse conjunto de entradas errôneas. Portanto, o Usuário 2 experimentará algumas falhas de sistema. O Usuário 1 e o Usuário 3, contudo, nunca utilizam entradas do conjunto errôneo. Para eles, o software será sempre confiável.

16.3 Segurança

A segurança de um sistema é um atributo que reflete a capacidade do sistema de operar, normal e anormalmente, sem ameaçar as pessoas ou o ambiente. Quando a segurança é um atributo essencial de um sistema crítico, esse sistema é um 'sistema de segurança crítica'. Entre os exemplos de sistemas críticos de segurança estão os sistemas de controle e monitoramento de aeronaves, os sistemas de controle de processo em indústrias químicas e farmacêuticas e os sistemas críticos de controle de automóveis.

O controle de hardware de sistemas críticos de segurança é mais simples de implementar e analisar do que o controle de software. Contudo, estamos atualmente construindo sistemas com tal complexidade, que não podem ser controlados somente por hardware. Algum controle de software é essencial, devido à necessidade de gerenciar grandes números de sensores e atuadores com complexas leis de controle. Um exemplo dessa complexidade se encon-

tra nas aeronaves militares avançadas, que são aerodinamicamente instáveis. Elas requerem o contínuo ajuste controlado por software de suas superfícies de vôo, para assegurar que não caiam.

Os softwares críticos de segurança são de duas classes:

1. *Software crítico de segurança primária* Esse é um software que é embutido como um controlador em um sistema. A disfunção desse software pode causar uma disfunção de hardware, o que resulta em ferimentos em pessoas ou em dano ambiental. Abordo esse tipo de software.

2. *Software crítico de segurança secundária* Esse é um software que pode indiretamente resultar em ferimentos. Exemplos desses sistemas são os sistemas de projeto de engenharia auxiliados por computador, cuja disfunção pode resultar em um erro de projeto no objeto que está sendo projetado. Esse defeito poderá pôr em risco as pessoas, se o sistema projetado operar inadequadamente. Outro exemplo de um sistema crítico de segurança secundária é o banco de dados de um médico, que contém detalhes de remédios ministrados em pacientes. Os erros nesse sistema podem resultar em uma dosagem incorreta de remédio que está sendo ministrado.

A confiabilidade e a segurança do sistema estão relacionadas, mas são atributos de confiança distintos. Naturalmente, um sistema crítico de segurança deve ser confiável no que se refere a atender suas especificações e operar sem falhas. Ele pode incorporar características de tolerância a defeitos, de modo que possa fornecer um serviço contínuo, mesmo que ocorram defeitos. Contudo, os sistemas que apresentam tolerância a defeitos não são necessariamente seguros. O software pode, ainda assim, funcionar mal e provocar um comportamento de sistema que resulte em um acidente.

À parte o fato de que nunca podemos ter 100 por cento de certeza de que um sistema de software é livre de defeitos e tolerante a eles, existem diversas outras razões pelas quais os sistemas de software que são confiáveis não são necessariamente seguros:

1. A especificação pode estar incompleta, pelo fato de não descrever o comportamento exigido do sistema em algumas situações críticas. Uma alta porcentagem de disfunções do sistema (Boehm *et al.*, 1975; Endres, 1975; Nakajo e Kume, 1991; Lutz, 1993) é resultado da especificação, e não de erros de projeto. Em um estudo sobre erros em sistemas embutidos, Lutz conclui:

   > (...) dificuldades com requisitos são a causa básica dos erros de software relacionados à segurança, erros que persistiram até a integração e os testes de sistema.

2. As disfunções de hardware podem fazer com que o sistema se comporte de maneira imprevisível e apresentar o software com um ambiente não previsto. Quando componentes estão prestes a falhar, eles podem se comportar irregularmente e gerar sinais que estão fora do alcance para serem manipulados pelo software.

3. O operador do sistema pode gerar entradas que não são individualmente incorretas, mas que, em determinadas situações, podem levar a uma disfunção do sistema. Um exemplo desse caso ocorreu quando um mecânico instruiu o software de gerenciamento de recursos de uma aeronave a levantar o trem de aterrissagem. O software cumpriu a instrução do mecânico, apesar de o avião estar no solo.

Um vocabulário especializado evoluiu para discutir os sistemas críticos de segurança e é importante compreender os termos específicos utilizados. Na Figura 16.8, são mostradas algumas definições adaptadas de termos inicialmente definidos por Leveson (1985).

Figura 16.8 ▶

Terminologia de segurança.

Termo	Definição
Acidente (ou desastre)	Um evento ou uma seqüência de eventos não planejados, que resulte em morte ou ferimento, danos à propriedade ou ao ambiente. Uma máquina controlada por computador, que provoque um ferimento em seu operador, é um exemplo de acidente.
Perigo	Uma condição com o potencial de provocar ou contribuir para um acidente. Uma falha do sensor que detecta um obstáculo em frente a uma máquina é um exemplo de perigo.
Dano	Uma medida da perda resultante de um desastre. O dano pode abranger a morte de muitas pessoas, como resultado de um acidente, ferimentos superficiais ou danos a propriedades.
Severidade do perigo	Uma avaliação do pior dano possível que poderia resultar de um perigo em particular. A severidade do perigo pode abranger desde a catástrofe, quando muitas pessoas morrem, até casos menores, em que somente resultem danos leves.
Probabilidade de perigo	A probabilidade de ocorrerem eventos que criem perigo. Os valores da probabilidade tendem a ser arbitrários, mas vão de provável (a chance de 1 para 100 de um perigo ocorrer) a não plausível (nenhuma situação concebível em que o perigo possa ocorrer é provável).
Risco	Uma média da probabilidade de que o sistema provocará um acidente. O risco é avaliado, considerando a probabilidade do perigo, a sua gravidade e a probabilidade de que ele resulte em um acidente.

A chave para garantir a segurança é assegurar que esses acidentes não ocorram ou que as conseqüências de um acidente sejam mínimas. Isso pode ser obtido de três maneiras complementares:

1. *Evitar o perigo* O sistema é projetado de modo que os perigos sejam evitados. Por exemplo, um sistema de corte, que exige que o operador pressione dois botões separados ao mesmo tempo para operar a máquina, evita o perigo de as mãos do operador estarem no curso de passagem da lâmina.
2. *Detectar e eliminar o perigo* O sistema é projetado de modo que os perigos sejam detectados e eliminados antes que resultem em acidentes. Por exemplo, o sistema de uma indústria química pode detectar pressão excessiva e abrir uma válvula de alívio para reduzir essa pressão, antes que uma explosão ocorra.
3. *Limitar o prejuízo* O sistema pode incluir características de proteção que minimizem os danos que podem resultar de um acidente. Por exemplo, um motor de aeronave normalmente inclui extintores de incêndio automáticos. Se ocorrer um incêndio, ele pode ser controlado antes que se torne uma ameaça aos passageiros ou à tripulação.

Os acidentes geralmente ocorrem quando várias coisas acontecem de errado ao mesmo tempo. Uma análise de acidentes sérios (Perrow, 1984) sugeriu que eles eram quase todos devidos a uma combinação de disfunções, e não a falhas isoladas. A combinação não prevista levou a interações, que resultaram na falha do sistema. Perrow também sugere que é impossível prever todas as combinações possíveis de disfunções do sistema e que os acidentes são uma parte inevitável do uso de sistemas complexos. O software tende a aumentar a complexidade do sistema; assim, utilizar o controle por software *pode* aumentar a probabilidade de acidentes no sistema.

Isso não significa que o controle por software invariavelmente aumente o risco associado a um sistema. O controle por software e seu monitoramento podem melhorar a segurança

de sistemas. Os sistemas controlados por software podem monitorar uma gama maior de condições do que os sistemas eletromecânicos. Eles podem ser adaptados com relativa facilidade e envolvem o uso de hardware de computador, que tem uma confiabilidade inerente muito alta e é fisicamente pequeno e leve. Os sistemas controlados por software podem fornecer intertravamentos de segurança sofisticados e aceitar estratégias de controle que reduzem o tempo em que as pessoas precisam estar em ambientes que apresentem periculosidade.

16.4 Proteção

A proteção de um sistema é uma avaliação do ponto em que o sistema protege a si mesmo de ataques externos, que podem ser acidentais ou deliberados. Os ataques podem ser, por exemplo, por vírus, uso não autorizado de serviços do sistema, modificação não autorizada do sistema ou de seus dados, entre outros. A proteção é importante para todos os sistemas críticos. Sem um nível razoável de proteção, a disponibilidade, a confiabilidade e a segurança do sistema poderão ser comprometidas, se ataques externos provocarem algum dano ao sistema.

A razão disso é que todos os métodos, para assegurar disponibilidade, confiabilidade e proteção, se baseiam no fato de que o sistema operacional é o mesmo daquele que foi originalmente instalado. Se esse sistema instalado foi comprometido de alguma maneira (por exemplo, se o software foi modificado para incluir um vírus), então os argumentos para confiabilidade e segurança originalmente efetuados não podem mais ser sustentados. O software de sistema pode ser corrompido e se comportar de maneira imprevisível.

Por outro lado, erros no desenvolvimento de um sistema podem levar a falhas na proteção. Se um sistema não responde a entradas inesperadas ou se seus limites não são verificados, então os atacantes podem explorar essas fraquezas, a fim de obter acesso ao sistema. Um importante incidente relacionado à proteção (o worm da Internet) (Spafford, 1989) aproveitou-se do fato de que programas em linguagem C não incluíam a verificação de limites para sobrescrever parte da memória com código que permitisse o acesso não autorizado ao sistema.

Naturalmente, existem alguns tipos de sistemas críticos em que a proteção é a dimensão mais importante da confiança do sistema. Os sistemas militares, os sistemas de comércio eletrônico e os sistemas que envolvam o processamento e o intercâmbio de informações confidenciais precisam ser projetados de maneira que atinjam um alto nível de proteção. Se um sistema de reservas de passagens aéreas (por exemplo) não estiver disponível, isso provocará inconveniência e alguns atrasos na emissão de passagens. Contudo, se o sistema for inseguro e puder aceitar reservas falsas, então a empresa aérea que possui o sistema poderá perder muito dinheiro, como resultado desse problema.

Há três tipos de danos que podem ser provocados por ataques externos:

1. *Interrupção de serviço* O sistema pode ser forçado a um estado em que seus serviços normais se tornem indisponíveis. Isso, obviamente, afeta a disponibilidade do sistema.

2. *Corrupção de programas ou dados* Os componentes de software de sistemas podem ser alterados sem autorização. Isso pode afetar o comportamento do sistema e, portanto, sua confiabilidade e segurança. Se o dano for grave, a disponibilidade do sistema poderá ser afetada.

3. *Revelação de informações confidenciais* As informações gerenciadas pelo sistema podem ser confidenciais e o ataque externo pode expô-las a pessoas não autorizadas. Dependendo do tipo de dados, isso poderá afetar a segurança do sistema e permitir ataques posteriores que afetem a disponibilidade do sistema ou sua confiabilidade.

Como acontece com os outros aspectos da confiança, existe uma terminologia especializada associada à proteção. Alguns termos importantes, como foi discutido por Pfleeger (1997), estão definidos na Figura 16.9.

Nesse caso, existe uma clara analogia com algumas palavras da terminologia de segurança, de maneira que uma exposição é análoga a um acidente e uma vulnerabilidade é análoga a um perigo. Portanto, há abordagens semelhantes, que podem ser utilizadas para assegurar a proteção de um sistema:

1. *Evitar a vulnerabilidade* O sistema é projetado de modo que vulnerabilidades não ocorram. Por exemplo, se um sistema não estiver conectado a uma rede pública externa, não haverá nenhuma possibilidade de ataques externos.
2. *Detectar e neutralizar ataques* O sistema é projetado para detectar vulnerabilidades e removê-las antes que resultem em uma exposição. Um exemplo de detecção e remoção de vulnerabilidade é o uso de um verificador de vírus que analisa os arquivos recebidos, à procura de vírus, e modifica esses arquivos, a fim de remover os possíveis vírus.
3. *Limitar a exposição* As conseqüências de um ataque bem-sucedido são minimizadas. Entre os exemplos da limitação de exposição estão os backups regulares do sistema e uma política de gerenciamento de configuração que permita recriar softwares danificados.

A proteção tem se tornado cada vez mais importante, à medida que mais e mais sistemas são conectados à Internet. As conexões da Internet proporcionam maior funcionalidade do sistema (por exemplo, um cliente pode ser capaz de acessar sua conta bancária diretamente), mas isso também significa que o sistema pode ser atacado por pessoas com intenções hostis. A Internet também significa que os detalhes de vulnerabilidade específicos do sistema podem ser facilmente disseminados, de modo que mais pessoas podem ser capazes de atacar o sistema.

Um atributo relacionado muito importante para os sistemas baseados na Internet é a sobrevivência (Ellison *et al.*, 1999a). Sobrevivência é a capacidade de um sistema continuar a prestar serviços enquanto está sob ataque e, potencialmente, parte do sistema está incapacitada. A sobrevivência está de maneira clara relacionada à proteção e à disponibilidade. Possibilitar a sobrevivência envolve identificar importantes componentes do sistema e assegurar que eles possam prestar um mínimo de serviço (Ellison *et al.*, 1999b). Três estratégias são utilizadas para ressaltar a sobrevivência: resistência a ataques, reconhecimento de um ataque e recuperação a partir dos danos provocados por um ataque. Não há espaço para abordar esse tópico aqui, mas são apresentados alguns links nas páginas Web do livro, para fornecer mais informações sobre pesquisa de sobrevivência.

Figura 16.9 ▶ Terminologia de proteção.

Termo	Definição
Exposição	Possível perda ou dano em um sistema de computação.
Vulnerabilidade	Fraqueza em um sistema baseado em computador, que pode ser explorada para causar perdas ou danos.
Ataque	Exploração da vulnerabilidade de um sistema.
Ameaças	Circunstâncias que têm o potencial de causar perda ou dano.
Controle	Medida de proteção que reduz a vulnerabilidade do sistema.

PONTOS-CHAVE

- A confiança em um sistema de computador é uma propriedade que reflete o grau de confiança do usuário no sistema. As mais importantes dimensões da confiança são a disponibilidade, a confiabilidade, a segurança e a proteção.
- Um sistema crítico é um sistema em que as falhas podem resultar em perdas econômicas significativas ou ameaças à vida humana. As três importantes classes de sistemas críticos são: os sistemas críticos de segurança, os sistemas críticos de missão e os sistemas críticos de negócios.
- A disponibilidade de um sistema é a probabilidade de que ele será capaz de prestar serviços a seus usuários quando for necessário e a confiabilidade é a probabilidade de que os serviços do sistema sejam prestados conforme especificado.
- A confiabilidade e a disponibilidade são geralmente consideradas as mais importantes dimensões da confiança. Se um sistema não for confiável, será difícil assegurar sua segurança ou proteção, uma vez que elas podem ser comprometidas com a ocorrência de falhas do sistema.
- A confiabilidade está relacionada à probabilidade de um erro ocorrer durante o uso operacional. Um programa pode conter defeitos conhecidos, mas ainda pode ser visto como confiável pelos seus usuários. Eles podem nunca utilizar recursos do sistema que sejam afetados por esses defeitos.
- A segurança de um sistema é um de seus atributos, que reflete a capacidade de o sistema operar, normal ou anormalmente, sem ameaçar as pessoas ou o ambiente. Quando a segurança for um atributo essencial de um sistema crítico, este será um sistema crítico de segurança.
- A proteção é importante para todos os sistemas críticos. Sem um nível de proteção razoável, a disponibilidade, a confiabilidade e a segurança do sistema podem ser comprometidas, caso ataques externos provoquem algum dano ao sistema.

LEITURAS SUGERIDAS

"Survivability: protecting your critical systems". Uma introdução acessível ao tópico de sobrevivência e por que ela é importante. (R. Ellison *et al.*, *IEEE Internet Computing*, nov./dez. 1999.)

Computer security. Esse livro é uma excelente introdução à área de proteção de computadores. Ele aborda princípios básicos, práticas, proteção de sistemas distribuídos e teoria de proteção. (D. Gollmann, 1999, John Wiley and Sons.)

Handbook of software reliability engineering. Esse livro apresenta uma discussão detalhada da confiabilidade do software a partir de dois pontos de vista, o prático e o teórico. (M.R. Lyu, 1996, McGraw-Hill.)

Computer-related risks. Trata-se de uma coleção extraída do Fórum de Riscos na Internet sobre incidentes que ocorreram em sistemas automatizados. Ela mostra o que pode realmente dar errado em sistemas relacionados com a segurança. (P.G. Neumann, 1995, Addison Wesley.)

EXERCÍCIOS

16.1 Quais são as dimensões mais importantes da confiança no sistema? Por que o custo de assegurar a confiança é exponencial?

16.2 Sugira seis razões pelas quais a confiança é importante em sistemas críticos.

16.3 Utilizando um exemplo, explique as dificuldades de descrever o que significa a confiabilidade de software.

16.4 Avalie a confiabilidade de um sistema de software que você utilize regularmente, mantendo uma tabela de acompanhamento de falhas de sistema e defeitos observados. Elabore um manual de usuário que descreva como utilizar, de maneira eficaz, esse sistema, na presença desses defeitos.

16.5 Identifique seis produtos para consumidores que possam conter, agora ou no futuro, sistemas de software de segurança crítica.

16.6 Explique por que assegurar a confiabilidade do sistema não é uma garantia de segurança do sistema.

16.7 Em um sistema médico que é projetado para emitir radiação para o tratamento de tumores, sugira um perigo que pode surgir e proponha um recurso de software que pode ser utilizado para assegurar que o perigo identificado não resulte em um acidente.

16.8 Explique por que há uma estreita relação entre a disponibilidade do sistema e a proteção do sistema.

16.9 Em termos de proteção de computadores, explique as diferenças entre um ataque e uma ameaça.

16.10 É ético que um engenheiro concorde em fornecer um sistema de software com defeitos conhecidos para um cliente? Faz alguma diferença se o cliente for avisado da existência desses defeitos com antecedência? Seria razoável fazer declarações sobre a confiabilidade do software, nessas circunstâncias?

16.11 Suponha que você faça parte de uma equipe que desenvolveu um software para uma fábrica de produtos químicos, o qual falhou e ocasionou um sério incidente de poluição. Seu patrão é entrevistado na televisão e declara que não há defeitos no software e que os problemas devem ser oriundos de procedimentos operacionais. Você é procurado pelo repórter do jornal para dar sua opinião. Discuta como deveria conduzir essa entrevista.

17 Especificação de sistemas críticos

Objetivos

O objetivo deste capítulo é explicar como especificar os requisitos funcionais e não funcionais de confiança. Depois de ler este capítulo, você poderá:

- conhecer uma série de métricas para a especificação da confiabilidade e compreender como elas podem ser utilizadas para especificar os requisitos de confiabilidade;
- compreender como os requisitos de segurança para os sistemas críticos podem ser derivados de uma análise de perigos e riscos, e
- compreender as similaridades entre os processos de especificação de requisitos de segurança e de proteção.

Sumário

17.1 Especificação de confiabilidade de software
17.2 Especificação de segurança
17.3 Especificação de proteção

Foram discutidos os processos de requisitos e as técnicas para o desenvolvimento da especificação de sistemas nos capítulos 5 a 9. Este capítulo complementa esses capítulos anteriores com uma discussão sobre questões específicas, que surgem na especificação de sistemas críticos. Devido aos altos custos em potencial das falhas de sistema, é importante assegurar que a especificação para os sistemas críticos seja de alta qualidade e reflita precisamente as reais necessidades dos usuários do sistema.

A necessidade de confiar em sistemas críticos gera requisitos de sistema funcionais e não funcionais:

1. Os requisitos funcionais de sistemas podem ser gerados para definir a verificação de erros e a recuperação de recursos que forneçam proteção contra falhas do sistema.
2. Os requisitos não funcionais podem ser gerados para definir a necessária confiabilidade e disponibilidade do sistema.

Além desses requisitos, as considerações de segurança e de proteção podem gerar um outro tipo de requisito, difícil de ser classificado como um requisito funcional ou não funcional. Esses requisitos talvez sejam mais bem descritos como requisitos 'não permitidos'. Em comparação com os requisitos funcionais normais, que definem o que o sistema fará, os requisitos 'não permitidos' definem o comportamento de sistema que é inaceitável. Entre os exemplos de requisitos 'não permitidos' estão:

O sistema não poderá permitir que os usuários modifiquem as permissões de acesso em quaisquer arquivos que eles não tenham criado (proteção).

O sistema não poderá permitir a seleção do modo reversão de impulso quando a aeronave estiver em vôo (segurança).

O sistema não poderá permitir a ativação simultânea de mais de três sinais de alarme (segurança).

Esses requisitos 'não permitidos' são, algumas vezes, decompostos em requisitos funcionais de software mais específicos. Como alternativa, as decisões de implementação podem ser adiadas até que o sistema seja projetado.

Os requisitos dos usuários para sistemas críticos sempre serão especificados utilizando-se a linguagem natural e os modelos de sistema. Contudo, como foi discutido no Capítulo 9, a especificação formal e a verificação associada são, muito provavelmente, eficazes, em termos de custos, no desenvolvimento de sistemas críticos (Hall, 1996; Wordsworth, 1996). As especificações formais não são apenas uma base para a verificação do projeto e da implementação. Elas são a maneira mais precisa de especificar sistemas e, assim, reduzir a possibilidade de equívocos. Além disso, construir uma especificação formal força uma análise detalhada dos requisitos e essa é uma maneira eficaz de descobrir problemas na especificação.

17.1 Especificação de confiabilidade de software

A confiabilidade é um conceito complexo, que deve sempre ser considerado no sistema, e não no nível individual dos componentes. Como os componentes em um sistema são interdependentes, uma falha em um único componente pode se propagar pelo sistema e afetar a operação de outros componentes. Em um sistema baseado em computadores, deve-se considerar três dimensões, quando se especifica a confiabilidade geral do sistema:

1. *Confiabilidade de hardware* Qual é a probabilidade de ocorrer uma falha em um componente de hardware e quanto tempo leva para reparar esse componente?
2. *Confiabilidade de software* Qual é a probabilidade de um componente de software produzir uma saída incorreta? As falhas de software são diferentes das falhas de hardware, porque o software não se desgasta; ele pode continuar operando corretamente depois de um resultado incorreto ter sido produzido.
3. *Confiabilidade de operador* Qual é a probabilidade de o operador de um sistema cometer um erro?

Tudo isso está estreitamente ligado. Uma falha de hardware pode gerar sinais falsos, que estejam fora do alcance das entradas esperadas pelo software. O software pode se comportar de modo imprevisível. O comportamento inesperado do sistema pode confundir o operador, resultando em estresse para ele. Erros por parte do operador são muito prováveis em condições de estresse. O operador pode, então, agir de maneira incorreta e fornecer entradas que sejam impróprias para a atual condição de falha. Essas entradas confundem ainda mais o sistema e mais erros podem ser gerados. Portanto, pode ocorrer uma situação em que uma única falha de subsistema, que seja recuperável, rapidamente se torne um sério problema, exigindo a interrupção completa do sistema.

A engenharia de confiabilidade de sistema (Lyu, 1996) é uma subdisciplina especializada da engenharia de sistemas, que se ocupa de fazer julgamentos gerais sobre a confiabilidade do sistema. Ela leva em conta as probabilidades de falhas de diferentes componentes em um sistema e como esses componentes são combinados para afetar a confiabilidade geral de um sistema. De maneira simples, se um sistema depende do componente A e do componente B, com as probabilidades de falha de P_A e P_B, então a probabilidade geral de falha de sistema P_S é:

$$P_S = P_A + P_B$$

À medida que o número de componentes dependentes aumenta, cresce a probabilidade geral de falhas do sistema. Se houver muitos e muitos componentes críticos em um sistema, então esses componentes individuais precisarão ser muito confiáveis para assegurar que P_S seja relativamente baixo. Para aumentar a confiabilidade, os componentes podem ser duplicados, como será discutido no Capítulo 18. Uma série de componentes idênticos trabalham em conjunto, e o grupo de componentes é operacional, desde que qualquer componente funcione corretamente. Isso significa (outra vez, de maneira simples) que, se a probabilidade de falha de um componente for P_A e se todas as falhas forem independentes, a probabilidade geral de falha P_S de um grupo com n componentes replicados idênticos será calculada multiplicando-se a probabilidade de falha de cada componente:

$$P_S = P_A^n$$

A confiabilidade do sistema deve ser especificada como um requisito não funcional, que é expresso quantitativamente utilizando-se uma das métricas discutidas na seção seguinte. Para cumprir os requisitos de confiabilidade não funcionais, pode ser necessário especificar mais requisitos funcionais e de projeto sobre o sistema que apontam como as falhas podem ser evitadas ou toleradas. Entre os exemplos desses requisitos de confiabilidade estão:

1. Uma extensão predefinida para todos os valores que são inseridos pelo operador será especificada e o sistema verificará se todas as entradas feitas pelo operador estão dentro dessa extensão predefinida.
2. Como parte do processo de inicialização, o sistema verificará todos os discos, procurando por blocos ruins.
3. A programação de N-versões será utilizada para implementar o sistema de controle de freios.

4. O sistema deve ser implementado em um subconjunto seguro de Ada e verificado utilizando-se a análise estática (veja o Capítulo 19).

Não existem regras simples que podem ser utilizadas para derivar requisitos de confiabilidade funcionais. Em organizações que desenvolvem sistemas críticos, de modo geral, existe o conhecimento organizacional sobre possíveis requisitos de confiabilidade e como eles afetam a confiabilidade real do sistema. Essas organizações podem se especializar em tipos específicos de sistemas, como sistemas de controle de estradas de ferro, e, assim, os requisitos de confiabilidade, uma vez derivados, podem ser reutilizados em vários sistemas.

17.1.1 Métricas de confiabilidade

As métricas de confiabilidade foram, primeiramente, delineadas para os componentes de hardware. A falha de componentes de hardware é inevitável, devido a fatores físicos, como abrasão mecânica e aquecimento elétrico. Os componentes têm algum tempo médio de vida útil, e isso se reflete na métrica de confiabilidade de hardware mais amplamente utilizada, o MTTF (*mean time to failure* – tempo médio para ocorrência de uma falha). O MTTF é o tempo médio pelo qual se espera que um componente se mantenha operacional. Quando o componente de hardware falhar, então a falha geralmente será permanente; por isso, o MTTR (*mean time to repair* – tempo médio para reparos), que reflete o tempo para reparar ou substituir um componente, também é importante.

Contudo, essas métricas de hardware nem sempre são aplicáveis à especificação de confiabilidade do software, devido à natureza diversa das falhas de software e hardware. As falhas de componentes de software são, muitas vezes, transitórias, e não permanentes. Elas somente se manifestam com alguma entrada. Se os dados não são danificados, o sistema pode continuar em operação depois da ocorrência de uma falha.

As métricas que foram utilizadas para especificar a confiabilidade e a disponibilidade são mostradas na Figura 17.1. A escolha de qual métrica deve ser utilizada depende do tipo de sistema ao qual ela se aplica e dos requisitos do domínio da aplicação. Entre alguns exemplos dos tipos de sistemas em que essas diferentes métricas podem ser utilizadas estão:

1. *Probabilidade de falha sob demanda* Essa métrica é muito apropriada para os sistemas em que os serviços são solicitados em intervalos imprevisíveis ou em intervalos relativamente longos e quando ocorrem sérias conseqüências, se o serviço não for prestado. Ela pode ser utilizada para especificar sistemas de proteção, como a confiabilidade de um sistema de alívio de pressão em uma indústria química ou um sistema de parada de emergência em uma usina de geração de energia.

2. *Taxa de ocorrência de falhas* Essa métrica deve ser utilizada quando as demandas regulares são feitas sobre os serviços do sistema e quando é importante que esses serviços sejam corretamente prestados. Ela pode ser utilizada na especificação de um sistema de caixa eletrônico, que processa as transações dos clientes, ou em um sistema de reserva de hotel.

3. *Tempo médio para a ocorrência de falhas* Essa métrica deve ser utilizada em sistemas em que ocorrem longas transações, por exemplo, quando as pessoas utilizam o sistema por longos períodos. O tempo médio para a ocorrência de falhas deve ser maior do que a duração média da transação. Entre os exemplos de sistemas em que essa medida pode ser utilizada estão os sistemas de processadores de texto ou os sistemas de CAD.

4. *Disponibilidade* Essa métrica deve ser utilizada em sistemas non-stop, quando os usuários esperam que o sistema preste um serviço contínuo. Entre os exemplos desses sistemas estão os sistemas de comutação telefônica e os sistemas de sinalização em estradas de ferro.

Métrica	Explicação
POFOD* Probabilidade de falha sob demanda	A probabilidade de o sistema falhar quando um pedido de serviço é feito. Uma POFOD de 0,001 significa que um entre mil pedidos de serviços pode resultar em falha.
ROCOF** Taxa de ocorrência de falha	A freqüência com que um comportamento inesperado pode ocorrer. Uma ROCOF de 2/100 significa que duas falhas têm possibilidade de ocorrer em cada cem unidades operacionais de tempo. Essa medida é, algumas vezes, chamada de intensidade de falhas.
MTTF*** Tempo médio para ocorrência de falha	O tempo médio entre as falhas observadas no sistema. Um MTTF de 500 significa que uma falha pode ser esperada a cada 500 unidades de tempo.
AVAIL**** Disponibilidade	A probabilidade de que o sistema esteja disponível para uso em um tempo específico. A disponibilidade de 0,998 significa que, em cada mil unidades de tempo, o sistema tem a probabilidade de estar disponível em 998 dessas unidades.

* Probability of Failure on Demand
** Rate of Failure Occurrence
*** Mean Time to Failure
**** Availability

Figura 17.1 ▲
Métricas da confiabilidade.

Existem três tipos de medições que podem ser feitas quando se avalia a confiabilidade de um sistema:

1. O número de falhas do sistema, considerando um número de pedidos de serviços do sistema. Esse método é utilizado para medir a POFOD.
2. O tempo (ou o número de transações) entre falhas do sistema. Esse método é utilizado para medir a ROCOF e o MTTF.
3. O tempo gasto para reparar ou reiniciar o sistema, quando ocorre uma falha. Considerando que o sistema precisa estar continuamente disponível, esse método é utilizado para medir a AVAIL.

As unidades de tempo que podem ser utilizadas nessas métricas são o tempo registrado no calendário, o tempo no processador ou alguma unidade diferente, como um número de transações. Em sistemas que empregam muito de seu tempo aguardando para responder a um pedido de serviço, como os sistemas de comutação telefônica, a unidade de tempo que deve ser utilizada é o tempo do processador. Basear a confiabilidade no tempo registrado em calendário resultaria em um número otimista.

O tempo do calendário é uma unidade de tempo apropriada para se utilizar em sistemas que estão em operação contínua. Por exemplo, os sistemas de monitoramento, como sistemas de alarme e outros tipos de sistemas de controle de processo, estão nessa categoria. Os sistemas que processam transações, como os caixas eletrônicos ou os sistemas de reserva de passagens aéreas, têm cargas variáveis sobre eles, dependendo da hora do dia. Nesses casos, a unidade de 'tempo' utilizada deve ser o número de transações, ou seja, a ROCOF seria o número de transações que falharam por N milhares de transações.

17.1.2 Requisitos de confiabilidade não funcionais

Em muitos documentos de requisitos de sistema, os requisitos de confiabilidade não são cuidadosamente especificados. As especificações de confiabilidade são subjetivas e não são mensuráveis. Por exemplo, declarações como "O software será confiável sob condições normais de uso" não têm significado. Declarações quase quantitativas, como "O software não deverá exi-

bir mais do que N defeitos/1 000 linhas de código", são igualmente inúteis. É impossível medir o número de defeitos/1 000 linhas de código, uma vez que não se pode saber quando todos os defeitos foram descobertos. Além disso, a declaração não significa nada em termos do comportamento dinâmico do sistema. São as falhas de software, e não seus defeitos, que afetam a confiabilidade de um sistema.

Os tipos de falhas que podem ocorrer são específicas do sistema, e as conseqüências de uma falha de sistema dependem de sua natureza. Quando se escreve uma especificação de confiabilidade, o especificador deve identificar diferentes tipos de falhas e considerar se essas falhas devem ser tratadas diferentemente na especificação. Entre os exemplos de diferentes tipos de falhas estão os mostrados na Figura 17.2. Com certeza, podem ocorrer combinações dessas falhas, como uma falha que seja transitória, recuperável e que pode corromper o sistema.

A maioria dos grandes sistemas é composta de diversos subsistemas, com diferentes requisitos de confiabilidade. Pelo fato de o software com grande confiabilidade ser dispendioso, de modo geral, é sensato avaliar os requisitos de confiabilidade de cada subsistema separadamente, em vez de impor os mesmos requisitos de confiabilidade a todos os subsistemas. Isso evita fazer grandes exigências de confiabilidade nos subsistemas em que isso é desnecessário.

As etapas envolvidas no estabelecimento de uma especificação de confiabilidade são:

1. Para cada subsistema identificado, definir os diferentes tipos de falhas do sistema que podem ocorrer e analisar as conseqüências dessas falhas.
2. A partir da análise das falhas do sistema, dividi-las em classes apropriadas. Um ponto de partida razoável é utilizar os tipos de falhas mostrados na Figura 17.2.
3. Para cada classe de falhas identificada, definir os requisitos de confiabilidade, utilizando uma métrica apropriada de confiabilidade. Não é necessário utilizar a mesma métrica para diferentes classes de falhas. Se uma falha exigir alguma intervenção para a recuperação, a probabilidade de essa falha ocorrer imediatamente deverá ser a métrica mais apropriada. Quando a recuperação automática for possível e o efeito da falha for inconveniência para o usuário, a ROCOF poderá ser a mais apropriada.
4. Quando for apropriado, identificar os requisitos funcionais de confiabilidade que definam a funcionalidade do sistema, a fim de reduzir a probabilidade de falhas críticas.

Como um exemplo da especificação de confiabilidade, considere os requisitos de confiabilidade para um caixa eletrônico de banco. Suponha que cada máquina na rede seja utilizada cerca de 300 vezes por dia. O tempo de vida útil do sistema de hardware é de oito anos, e a atualização do software ocorre normalmente a cada dois anos. Portanto, durante o tempo de vida útil de uma versão de software, cada máquina deverá manipular cerca de 200 mil transações. Um banco tem mil máquinas em sua rede; isso significa que são realizadas 300 mil transações no banco de dados central por dia (digamos, cem milhões por ano).

Figura 17.2 ▶ Classificação de falhas.

Classe de falha	Descrição
Transiente	Ocorre somente com algumas entradas.
Permanente	Ocorre com todas as entradas.
Recuperável	O sistema pode se recuperar sem a intervenção do operador.
Irrecuperável	É necessária a intervenção do operador para a recuperação a partir da falha.
Não corruptível	A falha não corrompe o estado do sistema ou seus dados
Corruptível	A falha corrompe o estado do sistema ou seus dados

Figura 17.3 ▶

Especificação de confiabilidade para um caixa eletrônico.

Classe de falha	Exemplo	Métrica de confiabilidade
Permanente, não corruptível	O sistema falha ao operar com qualquer cartão que seja inserido. O software deve ser reiniciado para corrigir a falha.	ROCOF 1 ocorrência/1 000 dias
Transiente, não corruptível	Os dados da fita magnética não podem ser lidos em um cartão sem defeito que tenha sido inserido.	ROCOF 1 em 1 000 transações
Transiente, corruptível	Um padrão de transações na rede causa a corrupção do banco de dados.	Não quantificável! Nunca deve acontecer no tempo de vida útil do sistema

As falhas são classificadas em duas amplas categorias: as falhas que afetam uma única máquina na rede e as que afetam o banco de dados e, por conseguinte, todos os caixas eletrônicos na rede. Obviamente, o último tipo de falha é menos aceitável do que as falhas locais em um caixa eletrônico.

A Figura 17.3 mostra possíveis classes de falhas e possíveis especificações de confiabilidade para diferentes tipos de falhas de sistema. Os requisitos de confiabilidade definem que é aceitável que uma falha permanente ocorra em uma máquina cerca de uma vez a cada três anos. Isso significa que, em média, uma máquina na rede bancária pode ser afetada a cada dia. Por outro lado, as falhas que simplesmente significam que uma transação deve ser abortada e que o usuário precisa reiniciar a operação podem ocorrer com relativa freqüência. Seu único efeito é causar uma pequena inconveniência ao usuário.

Idealmente, os defeitos que corrompem o banco de dados não devem ocorrer no tempo de vida útil do software. Portanto, o requisito de confiabilidade que pode ser definido, nesse caso, é que a probabilidade de ocorrer uma falha capaz de corromper o banco de dados do sistema, quando uma demanda é feita, é menor do que 1 em 200 milhões de transações. Isto é, no tempo de vida útil de uma versão de software de caixa eletrônico, nunca deveria haver um erro que provocasse a corrupção de um banco de dados.

Contudo, um requisito de confiabilidade como esse não pode, na verdade, ser testado. Digamos que cada transação dure um segundo do tempo da máquina e que um simulador possa ser construído para a rede ATM. A simulação das transações que ocorrem em um único dia na rede tomará 300 mil segundos, isto é, aproximadamente 3,5 dias. Claramente, esse período poderia ser diminuído pela redução do tempo de transação e pela utilização de vários simuladores, mas ainda é muito difícil testar o sistema para validar a especificação de confiabilidade.

É impossível validar requisitos quantitativos que exijam um nível muito alto de confiabilidade. Por exemplo, digamos que um sistema se destine ao uso em uma aplicação crítica de segurança e, portanto, nunca deva falhar durante todo o tempo de vida útil do sistema. Suponha que mil cópias do sistema devam ser instaladas e que o sistema seja 'executado' mil vezes por segundo. O tempo de vida útil projetado para o sistema é de dez anos. O número total estimado de execuções do sistema é, portanto, aproximadamente $3 * 10^{14}$. Não há razão para especificar que a taxa de ocorrência de falhas deve ser de $1/10^{15}$ execuções (isso permite algum fator de segurança), uma vez que não se pode testar o sistema por tempo suficiente para validar esse nível de confiabilidade.

Como outro exemplo de especificação de confiabilidade, considere o sistema de bomba de insulina que foi apresentado no Capítulo 16. Esse sistema aplica insulina uma série de vezes por dia, e a glicose no sangue do usuário é monitorada várias vezes por hora. Devido ao fato de o uso do sistema ser intermitente e de as conseqüências de uma falha serem sérias, a métrica de confiabilidade mais apropriada é a POFOD (probabilidade de falha sob demanda).

Como será discutido na próxima seção, a falha em fornecer insulina não tem implicações imediatas de segurança e, assim, fatores comerciais, e não fatores de segurança, definem o nível de confiabilidade requerida. Os custos de serviço são altos, porque os usuários necessitam de um serviço de reparo muito rápido e de um serviço de substituição. É de interesse do fabricante limitar o número de falhas permanentes que exijam reparo.

Novamente, dois tipos de falhas podem ser identificados:

1. Falhas transientes, que podem ser reparadas pela ação dos usuários, como reiniciar ou reajustar a máquina. Para esses tipos de falhas, um valor relativamente baixo da POFOD (digamos, 0,002) pode ser aceitável. Isso significa que uma falha pode ocorrer a cada 500 demandas feitas à máquina, ou seja, aproximadamente uma vez a cada 3,5 dias.
2. Falhas permanentes, que exigem que a máquina seja reparada pelo fabricante. A probabilidade desse tipo de falhas deve ser muito menor. Aproximadamente uma vez por ano é o número mínimo, portanto, a POFOD não deve ser maior do que 0,00002.

O custo de desenvolvimento e validação de uma especificação de confiabilidade do sistema pode ser muito alto. As organizações precisam ser realistas para definir se esses custos valem a pena. Eles são claramente justificáveis para sistemas em que a operação confiável seja fundamental, como os sistemas de comutação telefônica ou os sistemas em que uma falha pode resultar em grandes perdas econômicas. Eles provavelmente não são justificáveis para muitos tipos de sistemas de negócios ou sistemas científicos. De modo geral, esses sistemas têm requisitos de confiabilidade relativamente modestos, uma vez que os custos decorrentes de uma falha são apenas atrasos de processamento e sua recuperação é uma tarefa direta e de certa maneira não dispendiosa.

17.2 Especificação de segurança

A operação segura é a característica necessária de um sistema de software relacionado à segurança. Isso significa que é preciso considerar, durante o processo de engenharia de requisitos, os perigos em potencial que podem ocorrer. Cada perigo deve ser avaliado pelo risco que causa, e a especificação pode descrever como o software deve se comportar para minimizar esse risco ou exigir que esse perigo nunca aconteça.

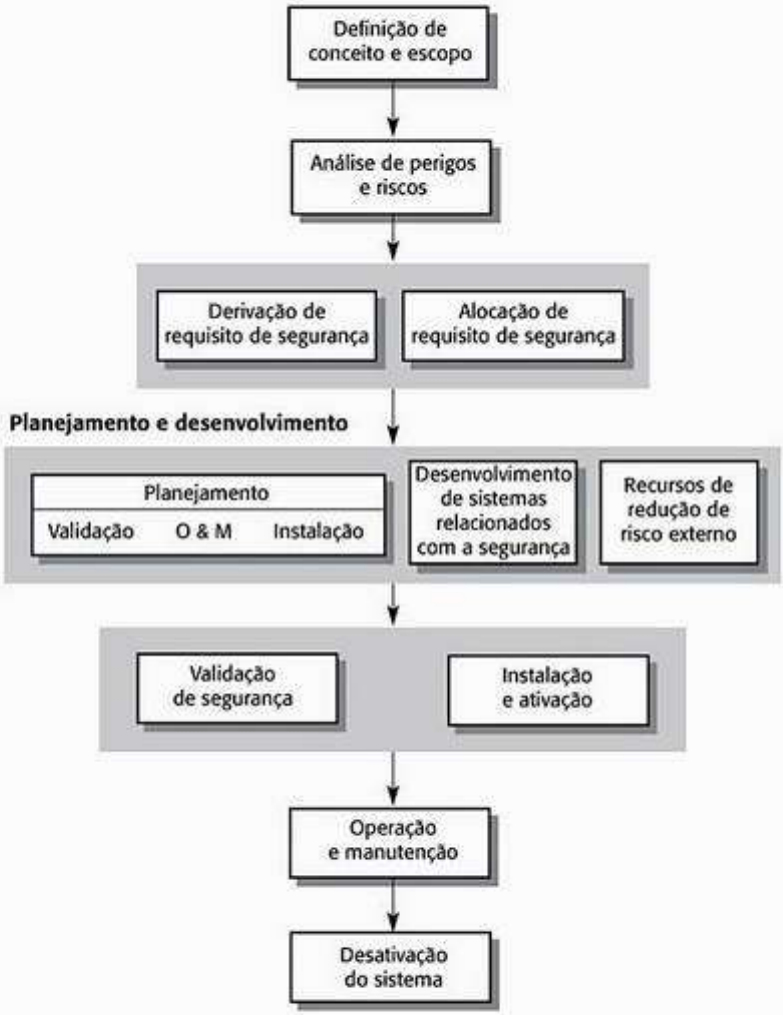
O processo de especificação e garantia de segurança é parte de um 'ciclo de vida de segurança' geral, ilustrado na Figura 17.4. A Figura 17.4 é uma maneira simplificada da apresentação de Redmill do ciclo de vida de segurança (Redmill, 1998), como foi proposto em um padrão internacional para o gerenciamento de segurança IEC 61508 (IEC, 1998). Como se pode ver pela Figura 17.4, esse padrão aborda todos os aspectos do gerenciamento de segurança, desde a definição inicial do enfoque, passando pelo planejamento e desenvolvimento do sistema, até a desativação do sistema.

Os primeiros estágios do ciclo de vida de segurança do padrão IEC 61508 definem o enfoque do sistema, avaliam os perigos em potencial do sistema e estimam os riscos que ele apresenta. Isso é seguido por especificações de requisitos do sistema e pela alocação desses requisitos de segurança em diferentes subsistemas. A atividade de desenvolvimento envolve o planejamento e a implementação. O próprio sistema crítico de segurança é projetado e implementado do mesmo modo que os sistemas externos relacionados, que podem fornecer maior proteção. Paralelamente a isso, são planejadas a validação de segurança e a operação e manutenção do sistema.

O gerenciamento de segurança não termina com a entrega do sistema. Depois da entrega, o sistema precisa ser instalado como foi planejado, de modo que a análise de perigos permaneça válida. A validação de segurança é, então, realizada antes que o sistema seja colocado em uso. A segurança deve também ser gerenciada durante a operação e (particularmente) a manutenção do sistema. Muitos problemas com sistemas, relacionados à segurança, surgem em razão de um processo de manutenção inadequado, de modo que é particularmente importante que o sistema seja projetado para ser de fácil manutenção. Por fim, as considerações de

Figura 17.4 ▶

Ciclo de vida de segurança do IEC 61508.



segurança que podem ser aplicadas durante a desativação (por exemplo, disposição de material perigoso em caixas fechadas) devem também ser levadas em conta.

17.2.1 Análise de perigos e riscos

A análise de perigos e riscos envolve estudar o sistema e seu ambiente operacional. O objetivo é descobrir perigos em potencial que podem surgir nesse ambiente, as principais causas desses perigos e os riscos associados a eles. Esse é um processo difícil e complexo, que requer a avaliação adicional e a colaboração de diferentes fontes de experiência. Ele deve ser empreendido por engenheiros experientes, em conjunto com especialistas no domínio e consultores de segurança profissionais. Técnicas de trabalho em grupo, como o brainstorming (troca intensiva de idéias), podem ser utilizadas para identificar os perigos, que também podem ser identificados porque um dos analistas envolvidos tem experiência direta com algum incidente prévio que resultou em um perigo.

A Figura 17.5 mostra o processo iterativo de análise de perigos e riscos:

1. *Identificação do perigo* Os perigos em potencial que podem surgir são identificados. Esses perigos dependem do ambiente em que o sistema deverá ser utilizado.
2. *Análise de risco e classificação do perigo* Os perigos são considerados separadamente. Aqueles que são potencialmente sérios, e não implausíveis, são selecionados para aná-

Figura 17.5 ▲
Análise de perigos e riscos.

lise posterior. Nesse estágio, alguns perigos podem ser eliminados, simplesmente porque é muito improvável que eles aconteçam (por exemplo, um choque elétrico e um terremoto simultâneos).

3. *Decomposição do perigo* Cada perigo é analisado individualmente para que se descubram suas causas em potencial. Técnicas como a análise da árvore de defeitos (discutida posteriormente nesta seção) podem ser utilizadas.

4. *Avaliação da redução de riscos* São feitas propostas de como os riscos identificados podem ser reduzidos ou eliminados. Essas propostas são a entrada para uma atividade de especificação de requisitos de segurança mais detalhada, como mostra o modelo de ciclo de vida de segurança (Figura 17.4).

Em grandes sistemas, a análise de perigos e riscos é geralmente estruturada em uma série de fases (Leveson, 1986). Essas fases são:

- a análise preliminar de perigos, em que os riscos principais são identificados;
- a análise de perigos mais detalhada do sistema e do subsistema;
- a análise de perigos relacionados ao software, quando os riscos de falhas de software são considerados, e
- a análise de perigo operacional, que se ocupa da interface com o usuário do sistema e dos riscos que surgem por erros do operador.

Essas análises identificam perigos e associam um risco a cada um deles. Isso envolve estimar a probabilidade de ocorrência de perigos, estimar a probabilidade de o perigo provocar um incidente e estimar a provável gravidade desse incidente. O julgamento de engenharia é utilizado para fazer a avaliação desses riscos.

O processo de análise de perigos geralmente envolve considerar diferentes classes de perigos, como os perigos físicos, elétricos, biológicos, de radiação (quando apropriado), perigos devidos a falhas de serviços e assim por diante. Cada uma dessas classes é, então, analisada em detalhes, a fim de descobrir os perigos associados.

Para o sistema de aplicação de insulina, apresentado anteriormente, os perigos e suas classes associadas são:

1. dose excessiva de insulina (falha de serviço);
2. dose insuficiente de insulina (falha de serviço);
3. falha de energia devido à bateria gasta (falha elétrica);
4. a máquina interfere eletricamente em outro equipamento médico, como um marcapasso cardíaco (falha elétrica);
5. contato inadequado do sensor e do atuador, causado pela conexão incorreta (falha física);
6. desarranjo de partes da máquina no corpo do paciente (falha física);
7. infecção provocada pela introdução da máquina (perigo biológico), e
8. reação alérgica aos materiais ou à insulina utilizados na máquina (perigo biológico).

Os riscos associados com esses perigos são abordados na Seção 17.2.3. Em virtude de esse sistema ser pequeno, não são necessárias várias fases de análise de risco. Muitos sistemas críticos de segurança, contudo, são muito grandes (por exemplo, uma fábrica de produtos químicos), e o processo de análise é longo, complexo e dispendioso.

17.2.2 Análise da árvore de defeitos

Para cada perigo identificado, uma análise detalhada deve ser realizada, a fim de descobrir as condições que podem causar esse perigo. As técnicas de análise de perigos podem ser dedutivas ou indutivas. As técnicas dedutivas, que tendem a ser mais fáceis de utilizar, se iniciam com o perigo e operam a partir dele para identificar a possível falha do sistema. As técnicas indutivas se iniciam com uma falha proposta do sistema e são identificados quais perigos podem surgir. Sempre que possível, as técnicas dedutivas e indutivas devem ser utilizadas para a análise de perigos.

Existem várias técnicas que foram propostas como abordagens possíveis para essas análises. Elas incluem revisões e checklists, além de técnicas mais formais, como a análise de rede de Petri (Peterson, 1981), a lógica formal (Jahanian e Mok, 1986) e a análise da árvore de defeitos (Leveson e Harvey, 1983).

Nesta seção, é abordada a análise da árvore de defeitos. Essa é uma técnica de análise de perigos amplamente utilizada, que é relativamente fácil de compreender, sem a necessidade de conhecimento especializado. A análise da árvore de defeitos envolve identificar o evento indesejável e trabalhar retroativamente a partir desse evento, a fim de descobrir as possíveis causas do perigo. O perigo está na raiz da árvore, e as folhas representam as causas do perigo em potencial.

Os perigos referentes a software normalmente se relacionam com falhas na prestação de um serviço do sistema ou com a falha de sistemas de monitoramento. Os sistemas de monitoramento podem detectar condições potencialmente perigosas, como falhas de energia.

A Figura 17.6 é a árvore de defeitos, que pode ser identificada para os possíveis perigos relacionados a software no sistema de fornecimento de insulina. A dose insuficiente e a excessiva de insulina realmente representam um único perigo, ou seja, 'dose incorreta de insulina administrada', e uma única árvore de defeitos pode ser traçada. Naturalmente, ao especificar como o software deve reagir aos perigos, a distinção entre uma dose insuficiente de insulina e uma dose excessiva deve ser levada em conta.

A árvore de defeitos, na Figura 17.6, está incompleta. Somente os defeitos de software em potencial foram inteiramente decompostos. Os defeitos de hardware, como carga de bateria baixa, que provoca a falha dos sensores, não foram mostrados. Nesse nível, não é possível fazer uma análise mais profunda. Contudo, à medida que um projeto e a sua implementação são desenvolvidos, uma análise mais detalhada da árvore de defeitos deve ser realizada. Leveson e Harvey (1983) e Leveson (1985) mostram como as árvores de defeitos podem ser desenvolvidas no projeto de software até o nível individual da declaração de linguagem de programação.

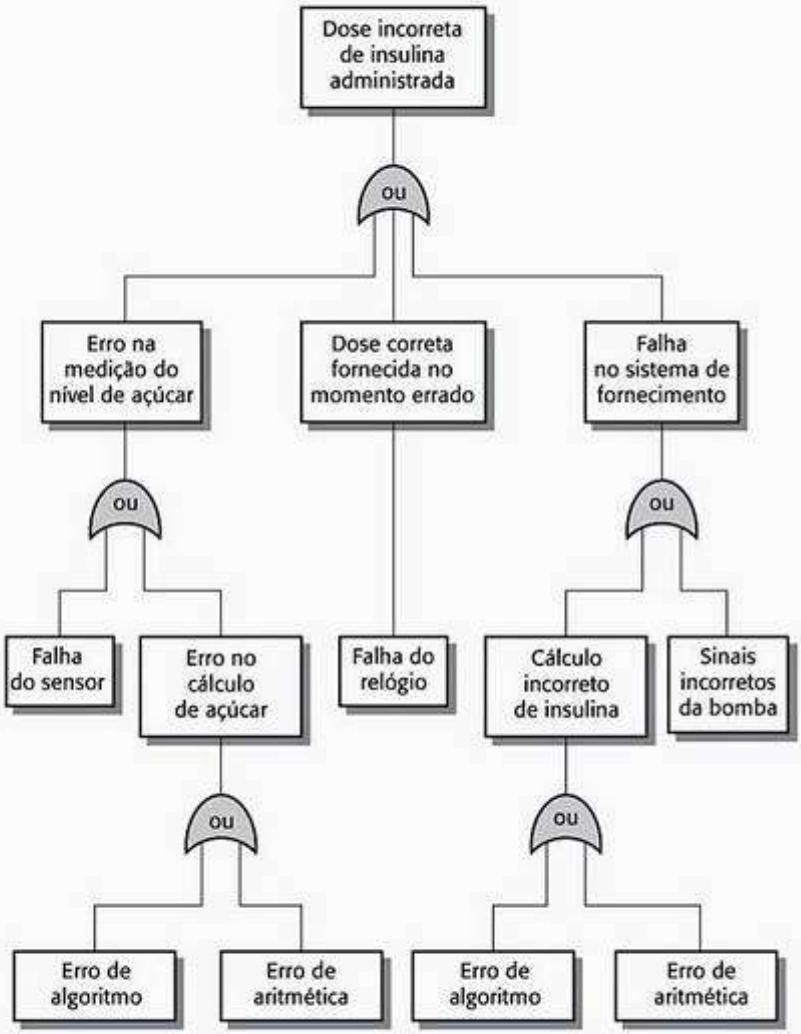
17.2.3 Avaliação de riscos

O processo de avaliação de riscos começa depois que todos os perigos foram identificados. A avaliação de riscos considera a gravidade de cada perigo, a probabilidade de que ele venha a surgir e a probabilidade de que um incidente ocorra a partir de um perigo. Para cada perigo, o resultado do processo de avaliação de riscos é uma declaração de aceitabilidade, que classifica o nível de aceitação do perigo, como se segue:

1. *Intolerável* O sistema deve ser projetado de maneira que o perigo não possa surgir ou, se surgir, não resulte em um acidente.

Figura 17.6 ▶

Árvore de defeitos para o sistema de fornecimento de insulina.



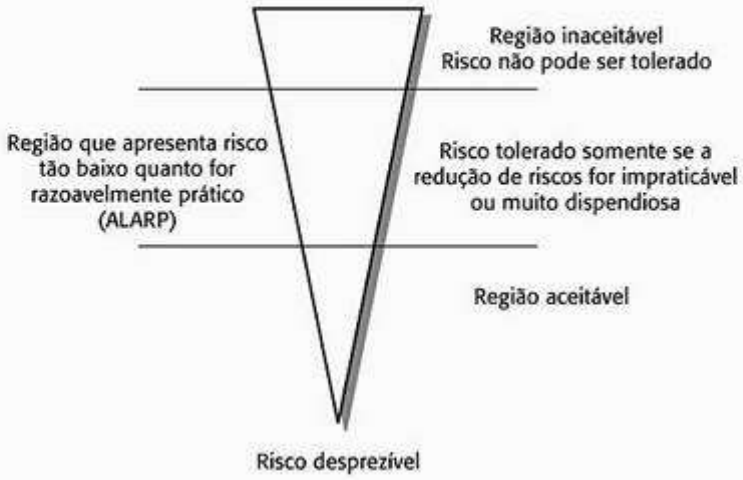
2. *ALARP (as low as reasonably practical – tão baixo quanto for razoavelmente prático)* O sistema precisa ser projetado de modo que a probabilidade de um acidente ocorrer por causa do perigo seja minimizada, estando sujeita a outras considerações, como custo e entrega.

3. *Aceitável* Enquanto os projetistas de sistema devem realizar todas as etapas para reduzir a probabilidade de esse perigo surgir, essas etapas não devem aumentar os custos, o tempo necessário para a entrega e outros atributos não funcionais do sistema.

A Figura 17.7 (Brazendale e Bell, 1994) mostra essas três regiões. O formato do diagrama reflete os custos necessários para assegurar que os perigos não resultem em acidentes. O custo do projeto de sistema para cuidar do perigo é uma função da largura do triângulo. Os custos mais altos estão relacionados a perigos representados no topo do diagrama e os custos mais baixos, a perigos no vértice inferior do triângulo.

Os limites entre as regiões na Figura 17.7 não são fixos para qualquer classe específica do sistema. Em vez disso, eles tendem a ser empurrados para baixo, com o passar do tempo, devido às expectativas públicas de segurança e a considerações políticas. Embora os custos financeiros da aceitação de perigos e do pagamento por quaisquer acidentes resultantes possam ser menores do que os custos da prevenção de acidentes, a opinião pública pode considerar que os custos adicionais devem ser aceitos. Por exemplo, pode ser mais barato para uma empresa limpar a poluição na rara ocasião em que ela ocorra, em vez de instalar sis-

Figura 17.7 ▶
Níveis de risco.



temas de prevenção de poluição. Isso podia ser aceitável nas décadas de 60 e 70, mas, hoje em dia, não é mais pública ou politicamente aceitável. O limite entre a região intolerável e a região ALARP desceu, de modo que os perigos que podem ter sido aceitos no passado sejam, atualmente, intoleráveis.

O processo de avaliação de riscos envolve estimar a probabilidade e a gravidade do perigo. De modo geral, isso é muito difícil de ser feito de maneira exata e, normalmente, depende de julgamentos de engenharia. As probabilidades e as gravidades são atribuídas utilizando-se termos relativos, como 'provável', 'improvável' e 'raro' ou 'alto', 'médio' e 'baixo'. Experiências anteriores do sistema podem permitir que algum valor numérico seja associado a esses termos. Contudo, como os acidentes são relativamente incomuns, é em geral muito difícil validar a exatidão desse valor.

A Figura 17.8 mostra uma classificação de riscos dos perigos identificados na seção precedente para o sistema de fornecimento de insulina. Como não se trata de um livro de medicina, as estimativas nessa tabela são apenas para efeito de ilustração. Observe que uma dose excessiva de insulina é potencialmente mais séria do que uma dose insuficiente de insulina em curto prazo.

Os perigos de 3 a 8 não estão relacionados com o software, de modo que não serão discutidos nesta seção. Para compensar esses perigos, a máquina deve ter um software nativo de autoconferência, que possa monitorar o estado do sistema e avisar sobre alguns desses perigos. A advertência poderá permitir que o perigo seja detectado antes de ocasionar um acidente. Entre os exemplos de perigos que podem ser detectados estão as falhas de energia e a colocação incorreta da máquina. O software de monitoramento é, naturalmente, relacionado à segurança, uma vez que a falha em detectar um perigo poderia resultar em um acidente.

17.2.4 Redução de riscos

Uma vez identificados os perigos em potencial e suas causas, a especificação do sistema deve ser formulada, de modo que seja improvável que esses perigos resultem em um acidente. No Capítulo 16, foram identificadas três possíveis estratégias que podem ser utilizadas:

1. *Evitar o perigo* O sistema é projetado de modo que o perigo não possa ocorrer.
2. *Detectar e eliminar o perigo* O sistema é projetado de modo que os perigos sejam detectados e neutralizados antes que resultem em um acidente.
3. *Limitar os danos* O sistema é projetado de modo que as conseqüências de um acidente sejam minimizadas.

Perigo identificado	Probabilidade do perigo	Severidade do perigo	Risco estimulado	Nível de aceitação
1. Dose excessiva de insulina	Média	Alta	Alto	Intolerável
2. Dose insuficiente de insulina	Média	Baixa	Baixo	Aceitável
3. Falha de energia elétrica	Alta	Baixa	Baixo	Aceitável
4. Máquina incorretamente equipada	Alta	Alta	Alto	Intolerável
5. Máquina quebra quando utilizada no paciente	Baixa	Alta	Médio	ALARP
6. Máquina provoca infecção	Média	Média	Médio	ALARP
7. Interferência elétrica	Baixa	Alta	Médio	ALARP
8. Reação alérgica	Baixa	Baixa	Baixo	Aceitável

Figura 17.8 ▲
Análise de risco de perigos identificados.

Normalmente, os projetistas de sistemas críticos de segurança utilizam uma combinação dessas três abordagens. Por exemplo, os perigos intoleráveis podem ser manejados reduzindo-se sua probabilidade tanto quanto for possível e acrescentando-se um sistema de proteção, para o caso de o perigo ocorrer.

Em um sistema de fornecimento de insulina, um 'estado seguro' é um estado de interrupção, quando nenhuma insulina é injetada. Por um curto período, isso não será uma ameaça para a saúde do diabético. Se os problemas em potencial identificados na Figura 17.6 forem considerados, as seguintes 'soluções' poderão ser desenvolvidas:

1. *Erro de aritmética* Ocorre quando algum cálculo aritmético provoca uma falha de representação. A especificação deve identificar todos os possíveis erros aritméticos que podem ocorrer. Esses erros dependem do algoritmo utilizado. A especificação pode definir que um manipulador de exceção deve ser incluído para cada erro aritmético identificado. A especificação deverá estabelecer a ação a ser tomada para cada um desses erros, se eles ocorrerem. Uma ação segura é interromper o sistema de fornecimento e ativar um alarme de advertência.

2. *Erro de algoritmo* Essa é uma situação mais difícil, uma vez que nenhuma situação anômala definida pode ser detectada. Ela pode ser detectada comparando-se a dose calculada de insulina necessária com a dosagem previamente fornecida. Se for muito maior, isso poderá significar que a quantidade foi calculada erroneamente. O sistema pode também manter o acompanhamento da seqüência de dosagens. Depois de uma série de dosagens acima da média ter sido aplicada, um aviso pode ser emitido e as demais dosagens serão limitadas.

As árvores de defeitos também são utilizadas para identificar problemas de hardware em potencial. Elas podem fornecer critérios de requisitos para o software detectar e, possivelmente, corrigir esses problemas. Por exemplo, as doses de insulina não são administradas muito freqüentemente, ou seja, não mais do que duas ou três vezes por hora; algumas vezes, menos do que isso. Portanto, existe a capacidade disponível no processador do sistema para executar programas de diagnóstico e programas de autoverificação. Os erros de hardware, como erros de sensores, de bomba e de relógio, podem ser descobertos e emitidos avisos antes que eles tenham um efeito sério sobre o paciente.

Alguns requisitos de segurança resultantes para o sistema de bomba de insulina são mostrados na Figura 17.9. São requisitos do usuário e, naturalmente, seriam expressos com mais detalhes em uma especificação final do sistema. Nesses requisitos, as referências às tabelas 3 e 4 se relacionam a tabelas que seriam incluídas no documento de requisitos.

Figura 17.9 ▶

Exemplos de requisitos de segurança para uma bomba de insulina.

RS1: O sistema não fornecerá uma única dose de insulina que seja maior do que uma dose máxima especificada por um usuário do sistema.
RS2: O sistema não fornecerá uma dose diária cumulativa de insulina que seja maior do que o máximo especificado por um usuário do sistema.
RS3: O sistema incluirá um recurso de diagnóstico de hardware, que será executado pelo menos quatro vezes por hora.
RS4: O sistema incluirá um manipulador de exceções para todas as exceções que são identificadas na Tabela 3.
RS5: O alarme sonoro será emitido quando qualquer anomalia de hardware for descoberta, e uma mensagem de diagnóstico, como a definida na Tabela 4, deverá ser exibida.

17.3 Especificação de proteção

A especificação de requisitos de proteção para sistemas tem algo em comum com os requisitos de segurança. É impraticável especificá-los quantitativamente, e os requisitos de proteção são, com freqüência, requisitos 'não permitidos', que definem comportamentos inaceitáveis do sistema, em vez da funcionalidade exigida para o sistema. Contudo, existem importantes diferenças entre esses tipos de requisitos:

1. A noção de um ciclo de vida de segurança que abrange todos os aspectos do gerenciamento de segurança é bem desenvolvida. A área de especificação e gerenciamento de proteção é imatura e não existe nenhum equivalente aceito para um ciclo de vida de proteção.
2. O conjunto de ameaças à proteção, enfrentadas pelos sistemas, é bastante genérico. Todos os sistemas precisam se proteger contra a invação, a interrupção de serviços etc. Por outro lado, os perigos em sistemas críticos de segurança são normalmente específicos do domínio.
3. Técnicas e tecnologias de proteção, como os dispositivos de criptografia e autenticação, são bastante aperfeiçoadas. Contudo, muitas tecnologias de proteção foram desenvolvidas para sistemas especializados (como sistemas militares e financeiros) e existem problemas na transferência para o uso mais geral. Técnicas de segurança de software ainda são objeto de pesquisa.

A abordagem convencional (não computadorizada) da análise de proteção tem como base os ativos a serem protegidos e seu valor para uma organização. Portanto, um banco fornecerá muita proteção a uma área em que grandes quantias de dinheiro estão armazenadas, em comparação com outras áreas públicas (digamos), em que as perdas em potencial são limitadas. A mesma abordagem pode ser utilizada para especificar a proteção para sistemas baseados em computadores. Um possível processo de especificação de proteção é mostrado na Figura 17.10.

Os estágios nesse processo são:

1. *Identificação e avaliação de ativos* Os ativos (dados e programas) e seu requerido grau de proteção são identificados. Observe que o grau de proteção necessário depende do valor do ativo, de modo que uma senha de arquivo (digamos) é normalmente mais valiosa do que um conjunto de páginas públicas na Web, uma vez que um ataque bem-sucedido ao arquivo de senhas traz sérias conseqüências para todo o sistema.
2. *Análise de ameaças e avaliação de riscos* As possíveis ameaças contra a proteção são identificadas e são estimados os riscos associados a cada uma dessas ameaças.
3. *Atribuição de ameaças* As ameaças identificadas são relacionadas aos ativos, de modo que, para cada ativo identificado, haja uma lista de ameaças associadas.

Figura 17.10 ▲ Especificação de proteção.

4. *Análise de tecnologia* São avaliadas as tecnologias disponíveis para a proteção e sua aplicabilidade contra as ameaças identificadas.
5. *Especificação dos requisitos de proteção* Os requisitos de proteção são especificados. Quando apropriados, eles identificarão explicitamente as tecnologias de proteção que podem ser utilizadas contra diferentes ameaças ao sistema.

Atualmente, a especificação e o gerenciamento da proteção são áreas de pesquisa importantes. Os padrões do gerenciamento de proteção estão em desenvolvimento (ISO/IEC, 1998), e é provável que sejam estabelecidos nos próximos anos.

PONTOS-CHAVE

- Os requisitos de confiabilidade devem ser definidos quantitativamente na especificação de requisitos do sistema.
- Existem várias métricas de confiabilidade diferentes, como a probabilidade de falhas sob demanda, a taxa de ocorrência de falhas, o tempo médio para a ocorrência de falhas e a disponibilidade. A métrica mais apropriada para um sistema específico depende do tipo de sistema e do domínio da aplicação. Diferentes métricas podem ser utilizadas para diferentes subsistemas.
- As especificações não funcionais da confiabilidade podem levar a requisitos funcionais do sistema que definam recursos do sistema cuja função seja reduzir seu número de falhas e, portanto, aumentar sua confiabilidade.
- A análise de perigos é uma atividade fundamental no processo de especificação de segurança. Ela envolve identificar condições perigosas que possam comprometer a segurança do sistema. Os requisitos do sistema são, então, gerados para assegurar que esses perigos não surjam ou que, se ocorrerem, não resultem em acidentes.
- A análise de riscos é o processo de avaliação da probabilidade de que um perigo possa resultar em um acidente. Ela identifica perigos importantes, que devem ser evitados no sistema, e classifica os riscos de acordo com sua gravidade.
- Para especificar os requisitos de proteção, é preciso identificar os ativos que devem ser protegidos e definir como as técnicas e a terminologia de proteção devem ser utilizadas para proteger esses ativos.

LEITURAS SUGERIDAS

"Requirements definition for survivable network systems". Discute os problemas da definição de requisitos para sistemas de sobrevivência, quando a sobrevivência se relaciona à disponibilidade e à proteção. (R.C. Linger, N.R. Mead e H.F. Lipson, *Proc. ICRE'98,* IEEE Press.)

Requirements engineering: a good practice guide. Esse livro traz uma seção sobre a especificação de sistemas críticos e uma discussão sobre o uso de métodos formais na especificação de sistemas críticos. (I. Sommerville e P. Sawyer, 1997, John Wiley and Sons.)

Safeware: system safety and computers. Essa é uma discussão completa de todos os aspectos dos sistemas críticos de segurança. É particularmente específica em sua descrição da análise de perigos e da derivação de requisitos a partir disso. (N. Leveson, 1995, Addison Wesley.)

EXERCÍCIOS

17.1 Por que, algumas vezes, não é apropriado utilizar medidas de confiabilidade de hardware para estimar a confiabilidade de sistemas de software? Justifique sua resposta com um exemplo.

17.2 Sugira métricas de confiabilidade apropriadas para as seguintes classes de sistemas de software e dê razões para sua escolha. Preveja o uso desses sistemas e sugira valores apropriados para as métricas de confiabilidade:

- um sistema que monitora pacientes em uma unidade de terapia intensiva;
- um processador de textos;
- um sistema de controle de máquina de venda automática;
- um sistema de controle de freios em um carro;
- um sistema para controlar uma unidade de refrigeração, e
- um gerador de relatórios de gerenciamento.

17.3 Você é responsável por escrever a especificação para um sistema de software que controla uma rede de terminais de EPOS (*electronic point of sale* — ponto-de-venda eletrônico) em uma loja. O sistema aceita as informações do código de barras de um terminal, consulta um banco de dados de produtos e retorna o nome do item e seu preço ao terminal, para exibição. O sistema tem de estar continuamente disponível durante o tempo de operação da loja.

Justificando sua escolha, defina métricas apropriadas de confiabilidade para especificar a confiabilidade de um sistema como esse e escreva uma especificação de confiabilidade plausível, que leve em conta o fato de que alguns defeitos são mais sérios do que outros.

17.4 Sugira quatro requisitos funcionais que podem ser gerados para esse sistema de loja, a fim de aumentar a confiabilidade do sistema.

17.5 Explique por que os limites do triângulo de risco, mostrado na Figura 17.7, estão sujeitos a mudanças com o passar do tempo e conforme as atitudes sociais se modificam.

17.6 No sistema da bomba de insulina, o usuário precisa mudar a agulha e o fornecimento de insulina em intervalos regulares e pode também modificar a dose única máxima e a dose diária máxima que podem ser ministradas. Sugira três erros de usuário que podem ocorrer e proponha requisitos de segurança que evitariam que esses erros resultassem em um acidente.

17.7 Um sistema de software crítico de segurança para o tratamento de pacientes com câncer tem dois componentes principais:

- Um equipamento de terapia por radiação, que aplica doses controladas de radiação nos locais onde existe um tumor. Esse equipamento é controlado por um sistema de software integrado.

- Um banco de dados de tratamento, que inclui os detalhes do tratamento dado a cada paciente. Os requisitos de tratamento são inseridos nesse banco de dados e são automaticamente fornecidos para o equipamento de terapia por radiação.

Identifique três perigos que podem surgir no sistema. Para cada perigo, sugira um requisito defensivo que reduzirá a probabilidade de que esses perigos resultem em acidentes. Explique por que a defesa proposta pode reduzir o risco associado com o perigo.

17.8 Sugira como a análise da árvore de defeitos poderia ser modificada para o uso na especificação de proteção. As ameaças em um sistema crítico de segurança são análogas aos perigos em um sistema crítico de segurança.

17.9 Os engenheiros de software que trabalham na especificação e no desenvolvimento de sistemas relacionados com a segurança devem ser certificados de alguma maneira?

18 Desenvolvimento de sistemas críticos

Objetivos

O objetivo deste capítulo é introduzir técnicas de implementação que são utilizadas no desenvolvimento de sistemas críticos. Depois de ler este capítulo, você poderá:

- compreender as técnicas de desenvolvimento de software que ajudam a evitar a introdução de defeitos em um sistema de software;
- compreender o conceito de tolerância a defeitos em software e como a programação defensiva pode ser utilizada para suportar a tolerância a defeitos;
- compreender como os recursos de manipulação de exceções em uma linguagem de programação podem ser utilizados na implementação de sistemas tolerantes a defeitos, e
- conhecer algo sobre a programação de N-versões e blocos de recuperação — duas diferentes abordagens para implementar arquiteturas tolerantes a defeitos.

Sumário

18.1 Minimização de defeitos
18.2 Tolerância a defeitos
18.3 Arquiteturas tolerantes a defeitos
18.4 Projeto de sistema seguro

Técnicas aperfeiçoadas de engenharia de software, melhores linguagens de programação e melhor gerenciamento da qualidade levaram a melhorias significativas em termos de confiança, para a maioria dos softwares. Contudo, para os sistemas críticos, como aqueles que controlam máquinas não assistidas, comutadores de telecomunicações ou motores de aeronaves, é necessário um cuidado adicional para se poder alcançar altos níveis de confiança. Nesses casos, técnicas especiais de programação podem ser utilizadas para assegurar que o sistema seja seguro, protegido e confiável.

Existem duas abordagens complementares que podem ser utilizadas quando o objetivo é desenvolver software confiável:

1. *Evitar defeitos* O processo de projeto e de implementação do sistema deve utilizar abordagens para o desenvolvimento de sistema que minimizem os erros humanos e que ajudem a descobrir defeitos no sistema, antes que ele seja colocado em uso.

2. *Tolerância a defeitos* O sistema deve ser projetado de tal maneira que defeitos ou comportamentos inesperados do sistema, durante sua execução, sejam detectados e gerenciados, para que a falha do sistema não ocorra.

Algumas vezes, uma terceira abordagem, a detecção de defeitos, é sugerida; mas na minha opinião ela está contida nas abordagens de evitar defeitos e tolerância a defeitos. Defeitos que são detectados antes da entrega são evitados no sistema operacional ao passo que defeitos que são detectados durante a execução fazem parte da tolerância a defeitos.

Evitar defeitos significa impedir que eles ocorram nos sistemas que são fornecidos aos clientes. Podemos fazer isso de duas maneiras: pelo uso de técnicas de programação, que minimizam o número de defeitos introduzidos em sistemas (minimização de defeitos), e pelo uso de técnicas de validação estáticas e dinâmicas, que descobrem esses defeitos e permitem que eles sejam corrigidos antes da entrega do sistema (detecção de defeitos). Neste capítulo, são estudadas a minimização de defeitos e a tolerância a defeitos, e uma série de técnicas de verificação e validação orientadas à detecção de defeitos será abordada do Capítulo 19 ao 21.

18.1 Minimização de defeitos

Um bom processo de software deve ter o objetivo de desenvolver softwares *livres de defeitos*. O software livre de defeito é aquele que atende exatamente a suas especificações. Contudo, isso não significa necessariamente que o software sempre se comportará como esperado pelos seus usuários. Pode haver erros nas especificações, que se refletirão no software, ou então os usuários podem cometer enganos ou utilizar mal o sistema de software. O software que é livre de defeitos não é, portanto, necessariamente livre de falhas. Contudo, minimizar os defeitos de software tem um impacto significativo no número de falhas do sistema, e isso deve sempre ser uma meta do desenvolvimento de sistemas críticos.

Existe uma série de requisitos para o desenvolvimento de software livre de defeitos:

1. Deve haver uma especificação precisa do sistema (de preferência, formal), que defina o sistema a ser implementado.

2. A organização que desenvolve o sistema precisa ter uma cultura de qualidade organizacional, segundo a qual a qualidade é o elemento direcionador do processo de software. Em geral, os programadores querem escrever programas livres de bugs.

3. Uma abordagem para o projeto e a implementação de software com base na ocultação e no encapsulamento de informações deve ser utilizada. As linguagens orientadas a objetos, como Java, obviamente satisfazem essa condição. O desenvolvimento de programas que são projetados para a facilidade de leitura e de compreensão deve ser incentivado.
4. Uma linguagem de programação que é fortemente baseada em tipos, como Java ou Ada, deve ser utilizada para o desenvolvimento. Em uma linguagem assim, muitos erros de programação podem ser detectados pelo compilador de linguagem.
5. O uso de algumas construções de programação que estão potencialmente sujeitas a erros deve ser evitado sempre que possível. Essas construções serão discutidas na próxima seção.
6. O processo de desenvolvimento deve ser definido e os desenvolvedores precisam ser treinados para a aplicação desse processo. Os gerentes de qualidade devem verificar a conformidade com o processo.

Conseguir um software livre de defeitos é muito difícil, caso linguagens de programação de baixo nível, com limitada verificação de tipos, como a linguagem C, sejam utilizadas para o desenvolvimento de programas. Existem duas razões para isso:

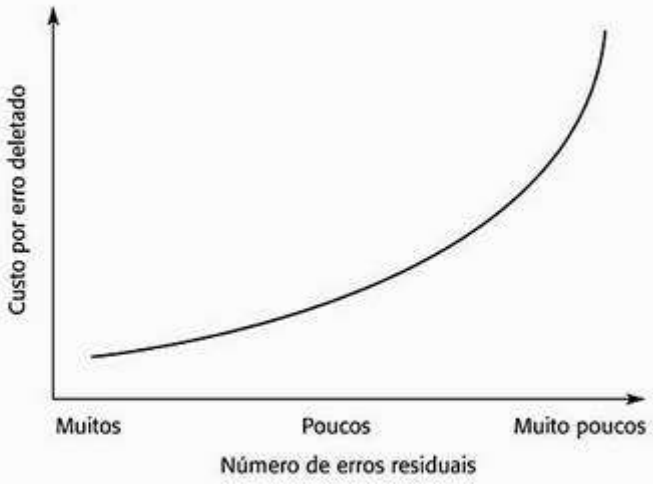
1. Essas linguagens incluem construções, como ponteiros, que sabemos, por experiência, que estão sujeitas a erros. Apesar de todo o cuidado que um programador pode tomar, ainda é possível a introdução de defeitos no programa, que são muito difíceis de ser detectados.
2. A natureza dessas linguagens as leva a um estilo de programação rígida. Isso faz com que o programa seja mais difícil de ler e de compreender; portanto, é menos provável que os leitores do programa possam localizar os erros que tenham ocorrido.

Naturalmente, a vantagem de utilizar essas linguagens de baixo nível é que suas construções são menos abstratas e, assim, é possível escrever programas muito eficientes. Em alguns casos, o alto desempenho é essencial e não pode ser alcançado de outra maneira. Nessas circunstâncias, é preciso dedicar maior esforço à realização de testes e à detecção de erros, se um alto nível de confiança for necessário.

Acredito que nossas técnicas de engenharia de software permitam produzir um software livre de defeitos, mas que será economicamente impraticável. É extremamente difícil e dispendioso alcançar esse objetivo. O custo de descobrir e eliminar defeitos remanescentes tende a crescer exponencialmente (Figura 18.1) durante a validação. À medida que o software se torna mais confiável, mais e mais testes são necessários para encontrar cada vez menos defeitos.

Figura 18.1 ▶

Crescentes custos da remoção de defeitos residuais.



Em conseqüência, as organizações de desenvolvimento de software, explícita ou implicitamente, aceitam que seu software contenha alguns defeitos residuais. O nível de defeitos depende do tipo de sistema. Os produtos 'de prateleira' têm um nível relativamente alto de defeitos (embora sejam muito melhores do que eram há dez anos), ao passo que os sistemas críticos em geral têm um nível de defeitos muito menor.

A razão para aceitar defeitos é que, se e quando o sistema falhar, é mais barato pagar pelas conseqüências da falha do que descobrir e eliminar os defeitos antes da entrega do sistema. Essa é uma prática bastante comum entre fabricantes de produtos de software para computadores pessoais. Contudo, como foi discutido no Capítulo 16, a decisão de lançar softwares defeituosos não é simplesmente uma decisão econômica. A aceitação social e política das falhas de sistemas também deve ser levada em consideração.

18.1.1 Evitando erros

Os defeitos em programas e, portanto, as muitas falhas em programas são, freqüentemente, conseqüências de erros humanos. Os programadores cometem erros porque perdem o controle de todas as relações entre as variáveis de estado. Eles escrevem declarações de programa que resultam em um comportamento inesperado e em mudanças de estado do sistema. As pessoas sempre cometerão enganos, mas ficou claro, no final da década de 60, que algumas abordagens de programação eram mais sujeitas a erros do que outras.

Dijkstra (1968a) reconheceu que a declaração 'goto' era uma construção de programação inerentemente sujeita a erros, o que tornava difícil localizar mudanças de estado. Essa observação levou ao desenvolvimento da chamada *programação estruturada*. Programação estruturada significa programar sem utilizar declarações 'goto', programar utilizando somente loops 'while' e declarações 'if' como construções de controle e projetar utilizando uma abordagem top-down. A adoção da programação estruturada foi um marco importante no desenvolvimento da engenharia de software, porque foi o primeiro passo para evitar uma abordagem indisciplinada do desenvolvimento de software.

Junto das construções incondicionais (declarações 'goto'), existem diversas outras construções de linguagem de programação e técnicas de programação que são inerentemente sujeitas a erros. É menos provável que defeitos sejam introduzidos em programas, se o uso dessas construções for minimizado. Elas incluem:

1. *Números em ponto flutuante* Os números em ponto flutuante são inerentemente imprecisos. Eles apresentam um problema específico quando são comparados, porque a imprecisão da representação pode levar a comparações inválidas. Por exemplo, o número 3,00000000 pode, algumas vezes, ser representado como 2,99999999 e, algumas vezes, como 3,00000001. Uma comparação mostraria que esses valores são desiguais. Os números com ponto fixo, quando um número é representado com um determinado número de casas decimais, são mais seguros, uma vez que é possível fazer comparações exatas.

2. *Ponteiros* Ponteiros são construções de baixo nível, que mantêm endereços que se referem diretamente às áreas da memória da máquina. Eles são perigosos porque os erros em sua utilização podem ser devastadores, já que permitem construções do tipo 'alias' (nomes alternativos), como é discutido a seguir, e porque fazem a verificação associada de vetores e outras estruturas mais difíceis de ser implementadas. O overflow de buffer, quando um programador deliberadamente constrói um programa para escrever memória além do fim de um buffer, é uma conhecida vulnerabilidade de proteção.

3. *Alocação dinâmica de memória* A memória do programa pode ser alocada em *run-time*, em vez de em tempo de compilação. O perigo, nesse caso, é que a memória pode não ser liberada, de maneira que o sistema pode ficar sem memória disponível. Esse pode ser um tipo de erro muito sutil para se detectar, uma vez que o sistema pode ser executado com sucesso por muito tempo, até que o problema ocorra.

4. *Paralelismo* O paralelismo é perigoso devido às dificuldades em se prever os efeitos sutis das interações de sincronia entre processos paralelos. De modo geral, os problemas de timing não podem ser detectados pela inspeção do programa e a combinação peculiar de circunstâncias que causam um problema de timing pode não ocorrer durante os testes do sistema. O paralelismo pode ser inevitável, mas seu uso deve ser cuidadosamente controlado, a fim de minimizar a dependência entre processos. Recursos de linguagem de programação, como as tarefas Ada ou os *threads* Java, ajudam a gerenciar o paralelismo para que alguns erros de programação sejam evitados.

5. *Recursão* A recursão é a situação em que um procedimento ou método chama a si mesmo ou chama outro procedimento que, então, chama o procedimento original que o chamou. Seu uso pode resultar em programas concisos, mas pode ser difícil acompanhar a lógica de programas recursivos. Os erros de programação são, por conseguinte, mais difíceis de detectar. Os erros no uso da recursão podem resultar na alocação de toda a memória do sistema, uma vez que são criadas variáveis de pilhas temporárias.

6. *Interrupções* As interrupções são um meio de forçar o controle a se transferir para uma seção de código, sem considerar o código atualmente em execução. Os perigos disso são óbvios, uma vez que a interrupção pode causar o término de uma operação crítica.

7. *Herança* A herança, em linguagens de programação orientadas a objetos, aceita o reuso e a decomposição de problemas, mas significa que o código associado a um objeto não está todo em um único local. Isso torna mais difícil compreender o comportamento do objeto. Por isso, é mais provável que erros de programação passem desapercebidos. Além disso, a herança, quando combinada com ligação dinâmica, pode causar problemas de timing em *run-time*. Em diferentes momentos, diferentes instâncias de um método específico podem ser chamadas e diferentes períodos serão gastos à procura pela instância correta do método.

8. *Construções do tipo 'alias' (nomes alternativos)* Ocorre quando diferentes nomes são utilizados para se referir à mesma entidade em um programa. É fácil para os leitores de programa não notar declarações que mudam a entidade, quando há vários nomes a serem considerados.

9. *Processamento de entradas default* Alguns sistemas fornecem um default para o processamento de entradas, sem considerar a entrada apresentada ao sistema. Esse é um defeito de proteção, uma vez que um programador pode explorar isso apresentando o programa com entradas inesperadas, que não são rejeitadas pelo sistema.

Alguns padrões para o desenvolvimento de sistemas críticos de segurança proíbem inteiramente o uso dessas construções. Contudo, essa posição extremada não é uma prática normal. Todas essas construções e técnicas são úteis, mas devem ser utilizadas com cuidado. Sempre que possível, seus efeitos potencialmente perigosos devem ser controlados, utilizando-os dentro de tipos abstratos de dados ou objetos. Elas atuarão como firewalls, limitando os danos causados, se ocorrerem erros.

Os projetistas de Java reconheceram alguns problemas das construções sujeitas a erros. A linguagem não tem recursos para transferências arbitrárias de controle (uma declaração 'goto'); ela tem coleta de lixo integrada, não havendo necessidade de alocar memória dinâmica, e não aceita o uso de ponteiros. Contudo, a representação numérica em Java não permite que o overflow seja detectado pelo sistema *run-time*, e as falhas devidas a erros de ponto flutuante ainda podem ocorrer.

18.1.2 Ocultação de informações

Um princípio de proteção adotado por organizações militares é o princípio do 'é preciso saber'. Somente os indivíduos que precisam saber de uma determinada informação, a fim de

cumprir seus deveres, recebem essa informação. A informação que não for diretamente relevante para seu trabalho será sonegada.

Ao realizar a programação, um princípio análogo deve ser adotado para controlar o acesso aos dados do sistema. Os componentes do programa devem ter permissão de acesso somente aos dados de que precisam para sua implementação. O acesso a outros dados deve ser negado, mediante ocultação, utilizando-se as regras de aplicação da linguagem de programação. Se a ocultação de informações for utilizada, a informação oculta não poderá ser corrompida por componentes do programa que não devam utilizá-la. Se a interface permanecer a mesma, a representação dos dados poderá ser modificada sem afetar outros componentes no sistema.

A ocultação de informações é muito mais simples em Java do que nas linguagens de programação mais antigas, como C ou Pascal. Essas linguagens não têm construções de encapsulamento, como classes de objetos, e assim os detalhes da implementação de estruturas de dados não podem ser protegidos. Outras partes do programa podem acessar a estrutura diretamente, o que pode conduzir a inesperados efeitos colaterais, quando são feitas mudanças.

Em geral, é uma boa prática, quando se programa em uma linguagem orientada a objetos, fornecer métodos que acessem e atualizem valores de atributos, em vez de permitir que outros objetos acessem esses atributos diretamente. Isso significa que a representação do atributo pode ser modificada sem fazer referência aos outros objetos que utilizam o atributo. É particularmente importante que essa abordagem seja utilizada para estruturas de dados e outros atributos complexos.

A construção da definição de interface em Java significa que é possível utilizar essa abordagem e declarar a interface para um objeto sem fazer referência à sua implementação. Isso está ilustrado na Figura 18.2. Os usuários de objetos do tipo Queue (fila) podem pôr objetos na fila, removê-los dela e consultar seu tamanho. Contudo, na classe que implementa essa interface, a implementação real da fila deve ser oculta, declarando que os atributos e métodos são privativos daquela classe de objeto.

Um tipo relacionado à ocultação de informações é ilustrado na Figura 18.3. Nas situações em que um limitado conjunto de valores pode ser atribuído a alguma variável, esses valores devem ser declarados como constantes. Linguagens como C++ aceitam tipos enumerados, mas em Java isso deve ser implementado associando-se essas restrições com a declaração de classes. Por exemplo, considere um sistema de sinalização que trabalhe com as luzes vermelha, amarela e verde e que esteja sendo implementado em Java. Um tipo de Signal deve ser definido e incluir declarações de constantes que reflitam essas cores. Portanto, é possível se referir a Signal.red, Signal.green e assim por diante. Isso evita a atribuição acidental de valores incorretos a variáveis do tipo Signal.

Figura 18.2 ▶

Uma especificação de fila que utiliza uma declaração de interface Java.

```
interface Queue {
   public void put (Object o) ;
   public void remove (Object o) ;
   public int size () ;
} //Queue
```

Figura 18.3 ▶

Uma declaração de classe Signal em Java.

```
class Signal {
   static public final int red = 1 ;
   static public final int amber = 2 ;
   static public final int green = 3 ;
   public int sigState ;
}
```

18.1.3 Processos de software confiáveis

Para desenvolver software com um número mínimo de defeitos, é essencial ter um processo de desenvolvimento de software que seja bem definido, repetível e que inclua uma gama de atividades de verificação e validação. Um processo bem definido é aquele que tenha sido padronizado e documentado. Um processo repetível é aquele que não depende de interpretação e julgamento individual. Melhor do que isso, independentemente do pessoal envolvido no processo, a organização pode estar certa de que o processo será bem-sucedido. A importância dos processos, em geral, e as melhorias de processo serão discutidas no Capítulo 25.

O processo deve incluir testes bem planejados e abrangentes, como também outras atividades, cujo objetivo é a detecção de defeitos. Entre as atividades de validação que são dirigidas à minimização de defeitos estão:

1. *Inspeções de requisitos* Como foi discutido no Capítulo 6, essas inspeções se destinam a descobrir problemas com a especificação de sistemas. Uma grande proporção de defeitos em softwares entregues resulta em erros nas especificações. Se eles puderem ser descobertos e eliminados a partir da especificação, então essa classe de defeitos será minimizada.

2. *Gerenciamento de requisitos* O gerenciamento de requisitos, discutido no Capítulo 6, se ocupa de manter o acompanhamento das mudanças nos requisitos e em dar continuidade ao projeto e à implementação. Muitos erros em sistemas entregues são resultados de falhas em assegurar que uma mudança de requisito tenha realmente sido incluída no projeto e na implementação do sistema.

3. *Verificação de modelo* A verificação de modelo envolve a análise automática de modelos de sistema por ferramentas CASE, que verificam se esses modelos têm consistência interna e externa. A consistência interna significa que um único modelo é consistente e a consistência externa, que diferentes modelos do sistema (por exemplo, um modelo de estado e um modelo de objeto) são consistentes.

4. *Inspeções de projeto e de código* Como será discutido no Capítulo 19, as inspeções de projeto e de código são, freqüentemente, baseadas em checklists de defeitos comuns e se destinam a descobrir e eliminar esses defeitos antes dos testes de sistema.

5. *Análise estática* A análise estática é uma técnica automatizada de análise de programas, em que o programa é analisado em detalhes para encontrar condições potencialmente errôneas. Isso será discutido no Capítulo 19.

6. *Planejamento e gerenciamento de testes* Um abrangente conjunto de testes para o sistema deve ser projetado e o próprio processo de testes deve ser cuidadosamente gerenciado a fim de assegurar a cobertura completa deles e a facilidade de acompanhamento entre os testes de sistemas e os requisitos de projeto do sistema. Esses testes serão discutidos no Capítulo 20.

O gerenciamento efetivo da configuração é essencial para toda a documentação associada a um sistema crítico. Como será discutido no Capítulo 29, o gerenciamento de configuração se ocupa de manter o acompanhamento das diferentes versões de um sistema e de seus componentes. Os erros em sistemas, algumas vezes, resultam da integração do componente errado ou da versão errada de um componente.

18.2 Tolerância a defeitos

Um sistema tolerante a defeitos é aquele que pode continuar em operação depois que alguns defeitos tenham se manifestado. O objetivo da tolerância a defeitos é assegurar que os defei-

tos do sistema não resultem na falha do sistema. A tolerância a defeitos é necessária nas situações em que a falha do sistema pode causar algum acidente catastrófico ou quando uma perda de operação do sistema causar grandes prejuízos econômicos. Por exemplo, os computadores em uma aeronave devem continuar a trabalhar até que a aeronave tenha aterrissado e os computadores de um sistema de controle de tráfego aéreo devem estar continuamente disponíveis, enquanto algum avião estiver no ar.

Seria possível pensar que os recursos da tolerância a defeitos não precisam ser incluídos em um sistema que tenha sido desenvolvido utilizando-se técnicas que minimizam defeitos. Se não houver defeitos no sistema, parecerá não haver qualquer chance de falhas no sistema. Contudo, estar 'livre de defeitos' não significa estar 'livre de falhas'. Isso somente pode significar que o programa corresponde a sua especificação. A especificação pode conter erros ou omissões e pode ter como base suposições incorretas sobre o ambiente do sistema. E, é claro, não podemos nunca demonstrar, de maneira conclusiva, que um sistema é inteiramente livre de defeitos. Nos sistemas que apresentam os mais altos requisitos de confiabilidade e disponibilidade, a aceitação explícita da tolerância a defeitos pode, portanto, ser necessária.

Existem quatro aspectos da tolerância a defeitos:

1. *Detecção de defeitos* O sistema deve detectar que uma determinada combinação de estados ocorreu, o que pode levar a uma falha do sistema.
2. *Avaliação de danos* As partes do sistema afetadas pelo defeito devem ser detectadas.
3. *Recuperação de defeitos* O sistema precisa ser restaurado a um estado 'seguro' conhecido. Isso pode ser alcançado corrigindo-se o estado afetado (recuperação de erro para frente) ou restaurando-se o sistema a um estado 'seguro' conhecido (recuperação de erro para trás).
4. *Reparo de defeitos* Isso envolve modificar o sistema, de modo que o defeito não se repita. Em muitos casos, os defeitos de software se manifestam como estados transientes. Eles se devem a uma peculiar combinação de entradas do sistema. Nenhum reparo é necessário, uma vez que o processamento normal pode reassumir imediatamente depois da recuperação do defeito. Essa é uma importante distinção entre defeitos de hardware e de software.

Existem duas abordagens complementares que podem ser utilizadas para implementar tolerância a defeitos em software:

1. A *programação defensiva* é uma abordagem do desenvolvimento de programas em que os programadores supõem que pode haver defeitos não detectados ou inconsistências em seus programas. Um código redundante é incorporado, a fim de verificar o estado do sistema, após modificações, e de assegurar que a mudança de estado seja consistente. Se inconsistências forem detectadas, a mudança de estado é redefinida ou o estado é restaurado para um estado correto conhecido.
2. *Arquiteturas tolerantes a defeitos* são arquiteturas de sistema de hardware e de software que fornecem a compatibilidade explícita com a tolerância a defeitos. Elas incluem redundância de hardware e de software e incorporam um controlador de tolerância a defeitos que detecta problemas e apóia a recuperação a partir de defeitos. Essa abordagem de tolerância a defeitos será discutida na Seção 18.3.

A programação defensiva é uma técnica que pode ser utilizada em qualquer sistema. Essencialmente, é preciso acrescentar à programação recursos de verificação extra e de recuperação de erros, mesmo em situações em que parece que a ocorrência de erros é improvável. Contudo, antes de prosseguir com a discussão sobre essa técnica, será descrita a manipulação de exceções, um recurso essencial para apoiar a tolerância a defeitos.

18.2.1 Manipulação de exceções

Quando um erro de algum tipo ou um evento inesperado ocorre durante a execução de um programa, isso é chamado de *exceção*. Exemplos de exceções podem ser uma falha em um sistema de energia, uma tentativa de acessar um item de dados não existente, overflow e underflow numérico, entre outros. As exceções podem ser provocadas por condições de hardware ou de software. Quando uma exceção ocorre, ela deve ser gerenciada pelo sistema. Isso pode ser feito dentro do próprio programa ou pode envolver a transferência do controle para um mecanismo de manipulação de exceções do sistema.

Em linguagens de programação como C, as declarações 'if' devem ser utilizadas para detectar a exceção e para transferir o controle para o código de manipulação de exceções. Existem dois problemas com essa abordagem:

1. Diferentes exceções podem ocorrer em pontos diferentes do programa, e a mesma exceção pode ocorrer em diferentes lugares. Isso significa que um grande número de verificações explícitas de exceções deve ser incluído no programa. Isso aumenta o tamanho e a complexidade do programa e o torna mais difícil de compreender. Há uma probabilidade maior de os programadores cometerem erros e de os leitores do programa não conseguirem localizar esses erros, quando estiverem verificando o programa.

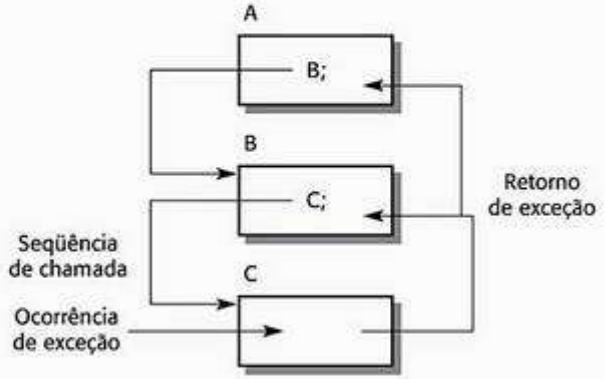
2. Quando uma exceção ocorre em uma seqüência de funções aninhadas ou chamadas de procedimento, não existe uma maneira fácil de transmitir essa ocorrência de uma função para outra. O controle é passado para baixo, por meio de uma seqüência de procedimentos. Quando uma exceção ocorre, essa estrutura de controle precisa ser desdobrada. Considere a situação da Figura 18.4, quando a função A chama a função B, que chama a função C. Se uma exceção ocorrer durante a execução de C, isso poderá ser tão sério que a execução de B não poderá continuar. A função B deverá retornar imediatamente para a função A, que também deverá ser informada de que B terminou de maneira anormal e de que uma exceção ocorreu. Novamente, uma quantidade de código extra poderá ser adicionada ao programa, para gerenciar as exceções.

Se a linguagem de programação incluir construções que aceitem a manipulação de exceções, então não serão necessárias declarações condicionais extras para verificar as exceções. Em vez disso, a linguagem de programação aceita um tipo *built-in* especial (freqüentemente, chamado de **Exception**) e diferentes exceções podem ser declaradas como sendo desse tipo. Quando uma situação excepcional ocorre, a exceção é assinalada e o sistema *run-time* da linguagem de programação transfere o controle para um manipulador de exceções. Essa é uma seção de código que declara nomes de exceção e ações apropriadas para manipular cada exceção.

Recursos explícitos de manipulação de exceção são fornecidos em Ada, C++ e Java. Em Java, novos tipos de exceções podem ser declarados, ampliando a classe **Exception**. As exce-

Figura 18.4 ▶

Retorno de exceção em uma chamada de função embutida.



ções são assinaladas em Java utilizando-se uma declaração throw. O manipulador de uma exceção é indicado pela palavra-chave catch, que é seguida por um bloco de códigos que pode manipular a exceção.

A Figura 18.5 ilustra o uso de exceções em Java. Esse exemplo é parte do software para a bomba de insulina, apresentada no Capítulo 16. Trata-se de um controlador de sensores, que lê um valor de glicose no sangue a partir de um sensor. A primeira declaração na Figura 18.5 mostra como as exceções em Java são declaradas. A classe de objetos chamada Exception é estendida e o método construtor define o código a ser implementado quando a exceção é lançada. Nesse caso, um alarme é ativado.

A classe Sensor fornece um único método, chamado readVal , que inclui uma declaração throw em sua declaração. Isso significa que uma declaração SensorFailureException pode ser lançada de dentro do método, mas espera-se que o método chamado forneça um manipulador para SensorFailureException. A palavra-chave try indica que uma exceção pode ser lançada no bloco seguinte de código. A exceção SensorFailureException será lançada, se um valor menor do que zero for retornado quando o sensor for verificado. DeviceIO.readInteger pode lançar uma exceção chamada deviceIOException, e assim um manipulador específico precisa ser incluído em seguida à palavra-chave catch. Nesse caso, o manipulador simplesmente lança uma exceção de falha de sensor, para indicar que o objeto que chama deve manipular a exceção.

Os recursos de manipulação de exceções em uma linguagem de programação não precisam ser utilizados somente para o gerenciamento de defeitos do sistema. Eles podem também ser utilizados para manipular condições de erro que possam ser previstas, embora, de maneira normal, devam ocorrer apenas raramente. Essa condição foi mostrada na Figura 18.6. Essa classe Java é uma implementação de um controle de temperatura em um freezer para alimentos. A temperatura requerida pode ser ajustada entre −18 e −40 graus Celsius.

Os alimentos congelados podem começar a descongelar e as bactérias tornam-se ativas em temperaturas acima de −18 graus. O sistema de controle mantém essa temperatura, ligando e desligando uma bomba, dependendo do valor de um sensor de temperatura. Se a temperatura requerida não puder ser mantida, o controlador acionará um alarme.

Na implementação Java, a temperatura do freezer é descoberta interrogando-se um objeto chamado tempSensor e a temperatura requerida, inspecionando-se um objeto chamado tempDial. Um objeto-bomba (Pump) responde a sinais para mudar seu estado. Uma vez ligada a bomba, o sistema espera algum tempo (chamando Thread.sleep) para a temperatura cair. Se a temperatura não tiver caído o suficiente, uma exceção chamada FreezerTooHotException será lançada.

Figura 18.5 ▶

Exceções em Java.

```
class SensorFailureException extends Exception {
  SensorFailureException (String msg) {
      super (msg) ;
      Alarm.activate (msg) ;
  }
} // SensorFailureException
class Sensor {
  int readVal () throws SensorFailureException {
  try {
      int theValue = DeviceIO.readInteger () ;
      if (theValue < 0)
          throw new SensorFailureException ("Sensor failure") ;
      return theValue ;
   }
  catch (deviceIOException e)
      {
          throw new SensorFailureException (" Sensor read error ") ;
      }
  }   // readVal
} // Sensor
```

Figura 18.6

Exceções em um controlador de temperatura de freezer.

```
class FreezerController {

  Sensor tempSensor = new Sensor () ;
  Dial tempDial = new Dial () ;
  float freezerTemp = tempSensor.readVal () ;
  final float dangerTemp = (float) -18.0 ;
  final long coolingTime = (long) 200000.0 ;

  public void run ( ) throws InterruptedException {
  try {
      Pump.switchIt (Pump.on) ;
      do {
          if (freezerTemp > tempDial.setting ())
              if (Pump.status == Pump.off)
              {
                  Pump.switchIt (Pump.on) ;
                  Thread.sleep (coolingTime) ;
              }
          else
              if (Pump.status == Pump.on)
                  Pump.switchIt (Pump.off) ;

          if (freezerTemp > dangerTemp)
              throw new FreezerTooHotException () ;
          freezerTemp = tempSensor.readVal () ;

      } while (true) ;

  } // try block
  catch (FreezerTooHotException f)
  {   Alarm.activate ( ) ; }
  catch (InterruptedException e)
  {
      System.out.println ("Thread exception") ;
      throw new InterruptedException ( ) ;
  }

  } //run
} // FreezerController
```

O manipulador de exceções (localizado no fim do código) capta essa exceção e ativa o objeto **Alarm**. Um manipulador é também incluído para a exceção **InterruptedException**, que pode ser lançada por **Thread.sleep**. Isso registra a exceção e então a relança para manipulação pelo método principal.

18.2.2 Detecção de defeitos

As linguagens de programação, como Java e Ada, têm um sistema rígido. Isso permite que muitos erros que causam corrupção de estado e falhas de sistema sejam detectados em tempo de compilação. O compilador pode detectar esses problemas, que abrem uma 'brecha' nas regras estritas. A verificação do compilador é, com certeza, limitada a valores estáticos, mas ele pode também gerar código automaticamente, para realizar verificações em *run-time* sobre, por exemplo, atribuições a vetores. Isso assegura que o índice de vetores não esteja fora da extensão do vetor e que a atribuição seja feita para um membro legal dele.

O primeiro estágio da tolerância a defeitos é detectar que um defeito (um estado errôneo do sistema) ocorreu, ou ocorrerá, a menos que alguma ação seja tomada de imediato. Normalmente, uma exceção é, então, lançada para uma seção de código no programa que possa gerenciar o defeito detectado.

Existem dois tipos de detecção de defeitos:

1. *Detecção preventiva de defeitos* Nesse caso, o mecanismo de detecção de defeitos é iniciado antes que uma mudança de estado seja cometida. Se um estado potencialmente errôneo for detectado, então a mudança de estado não será efetuada. De modo geral, o sistema de detecção de defeitos lança uma exceção, que inclui o tipo do defeito descoberto.

2. *Detecção de defeito retrospectiva* Nesse caso, o mecanismo de detecção de defeitos é iniciado depois que o estado do sistema foi modificado, a fim de verificar se um defeito ocorreu. Se um defeito for descoberto, uma exceção será assinalada e um mecanismo de reparo será utilizado para a recuperação a partir do defeito.

A detecção preventiva de defeitos é, freqüentemente, implementada definindo-se restrições que se aplicam ao estado do sistema e verificando-se essas restrições, quando um novo estado é computado. Esse método é simplificado quando o estado é dividido em um conjunto de objetos. As restrições que se aplicam a objetos individuais podem ser definidas e verificadas automaticamente, quando métodos de objeto que podem modificar o estado são iniciados. Essa condição está ilustrada na Figura 18.7, que é uma classe Java que implementa um tipo de número par positivo.

Nesse caso, a restrição de estado é que somente números positivos e pares podem ser atribuídos. Se for feita uma tentativa de atribuir um valor que viole essa restrição, uma exceção (NumericException) será lançada e assinalada para o método que chamou. Observe que a exceção não é manipulada dentro da classe PositiveEvenInteger. A manipulação da exceção deve sempre ser de responsabilidade do método que chamou, uma vez que ele conhece a ação de exceção mais apropriada para tomar.

A detecção preventiva de defeitos evita, de maneira geral, os problemas com o reparo de danos, uma vez que o estado do sistema será sempre válido. Contudo, ela envolve muito

Figura 18.7 ▶

Classe de número PositiveEven em Java.

```
class PositiveEvenInteger {
  int val = 0 ;
  PositiveEvenInteger (int n) throws NumericException
  {
      if (n < 0 | n%2 == 1)
          throw new NumericException () ;
      else
          val = n ;
  }// PositiveEvenInteger
  public void assign (int n) throws NumericException
  {
      if (n < 0 | n%2 == 1)
          throw new NumericException () ;
      else
          val = n ;
  } // assign
  int toInteger ()
  {
      return val ;
  } //to Integer
  boolean equals (PositiveEvenInteger n)
  {
      return (val == n.val) ;
  } // equals
} //PositiveEven
```

tempo extra, pois cada operação que modifica o estado precisa verificar a restrição antes que a mudança de estado seja efetuada. Em algumas classes de sistemas, em que o desempenho é importante ou em que é relativamente direto reparar um estado danificado, a detecção retrospectiva de defeitos pode ser utilizada. Além disso, se a restrição de correção de estado se aplicar às relações entre objetos, isso não poderá ser implementado verificando-se a restrição em um objeto individual.

A detecção retrospectiva de defeitos sempre envolve verificar uma restrição de estado, mas, nesse caso, o estado como um todo é examinado e uma função de verificação é aplicada a variáveis individuais de estado ou a diversas variáveis de estado. As funções de verificação individuais podem ser associadas em Java, utilizando-se a seguinte interface:

```
interface CheckableObject {
   public boolean check () ;
}
```

Os objetos a serem verificados são instanciações de uma classe de objetos que implementa essa interface; assim, cada objeto tem uma função de verificação associada. Cada classe de objeto implementa sua própria função de verificação, que define as restrições particulares que se aplicam a objetos dessa classe. Quando o estado como um todo é verificado, a vinculação dinâmica é utilizada para assegurar que a função de verificação, que é apropriada para a classe de objeto que está sendo verificada, seja aplicada. Podemos ver um exemplo disso na Figura 18.8, em que a função de verificação confirma se os elementos de um vetor satisfazem algumas restrições.

A detecção retrospectiva de defeitos aplicada a mais de uma variável de estado está ilustrada na Figura 18.9. Nesse exemplo, a verificação de detecção de defeitos é aplicada a elementos consecutivos de um vetor e verifica se o vetor é solicitado.

Figura 18.8 ▶

Uma classe de vetores com avaliação de danos.

```
class RobustArray {
   // Checks that all the objects in an array of objects
   // conform to some defined constraint

   boolean [] checkState ;
   CheckableObject [] theRobustArray ;

   RobustArray (CheckableObject [] theArray)
   {
        checkState = new boolean [theArray.length] ;
        theRobustArray = theArray ;
   } //RobustArray

   public void assessDamage () throws ArrayDamagedException
   {
        boolean hasBeenDamaged = false ;

        for (int i = 0 ; i < this.theRobustArray.length ; i ++)
        {
             if (! theRobustArray [i].check ())
                  {
                       checkState [i] = true ;
                       hasBeenDamaged = true ;
                  }
             else
                  checkState [i] = false ;
        }
        if (hasBeenDamaged)
             throw new ArrayDamagedException () ;
   } //assessDamage

} // RobustArray
```

Figura 18.9 ▶

Procedimento de classificação segura, com recuperação de erros para trás.

```
class SafeSort {
  static void sort ( int [] intarray, int order ) throws SortError
  {
      int [] copy = new int [intarray.length];
      // copy the input array
      for (int i = 0 ; i < intarray.length ; i++)
          copy [i] = intarray [i] ;
      try {
          Sort.bubblesort (intarray, intarray.length, order) ;
          if (order == Sort.ascending)
              for (int i = 0 ; i <= intarray.length-2 ; i++)
                  if (intarray [i] > intarray [i+1])
                      throw new SortError () ;
          else
              for (int i = 0 ; i <= intarray.length-2 ; i++)
                  if (intarray [i+1] > intarray [i])
                      throw new SortError () ;
      } // try block
      catch (SortError e )
      {
          for (int i = 0 ; i < intarray.length ; i++)
              intarray [i] = copy [i] ;
          throw new SortError ("Array not sorted") ;
      } //catch
  } // sort
} // SafeSort
```

18.2.3 Avaliação de danos

A avaliação de danos envolve a análise do estado do sistema, a fim de medir a extensão da corrupção do estado. Em muitos casos, isso pode ser evitado, verificando-se a ocorrência de defeitos antes que finalmente ocorra uma mudança de estado. Se um defeito for detectado, a mudança de estado não será aceita, de modo que nenhum dano será causado. Contudo, a avaliação de danos pode ser necessária quando um comprometimento do estado não puder ser evitado ou quando um defeito surgir porque uma seqüência inválida de estado, individualmente correto, modifica resultados em um estado incorreto do sistema.

A função dos procedimentos de avaliação de danos não é possibilitar a recuperação a partir do defeito, mas avaliar que partes do espaço de estado foram afetadas pelo defeito. O dano somente pode ser avaliado se for possível aplicar alguma 'função de validade', que avalie se o sistema é consistente. Se forem encontradas inconsistências, elas serão destacadas ou assinaladas de alguma maneira.

Na Figura 18.8, é apresentada a maneira como a avaliação de danos pode ser implementada em Java. Suponha-se que a estrutura de dados chamada RobustArray seja uma coleção de objetos do tipo CheckableObject. A classe que implementa o tipo CheckableObject precisa incluir um método chamado check, que pode testar se o valor do objeto satisfaz alguma restrição. Tem sentido que esse método de verificação seja associado a esse objeto e não ao objeto RobustArray, uma vez que os detalhes da verificação dependem do uso do tipo CheckableObject.

O método assessDamage, na classe RobustArray, examina cada elemento do vetor e verifica se seu estado está correto. Se um ou mais elementos do vetor não atenderem às restrições de estado definidas na função Check, então os elementos que estão danificados serão registrados no vetor checkState. Uma exceção chamada ArrayDamagedException é, então, lançada. Um manipulador para essa exceção, que gerencia o dano, deve ser incluído no método que chama. Ele pode utilizar a informação em checkState para decidir o que fazer.

Outras técnicas que podem ser utilizadas para detecção de defeitos e avaliação de danos dependem da representação do estado do sistema e da aplicação. Dentre os possíveis métodos estão:

1. O uso de checksums no intercâmbio de dados e dígitos de verificação em dados numéricos.
2. O uso de links redundantes em estruturas de dados que contenham ponteiros.
3. O uso de timers de watchdog em sistemas concorrentes.

As verificações de codificação (Fujiwara e Pradhan, 1990) podem ser utilizadas quando existe o intercâmbio de dados no qual um checksum é associado a dados numéricos. Um checksum é um valor que é calculado pela aplicação de alguma função matemática aos dados. A função utilizada deve apresentar um único valor para o pacote de dados intercambiado. Esse checksum é computado pelo remetente que aplica a função de checksum aos dados e anexa esse valor de função a eles. O receptor aplica a mesma função aos dados e compara os valores do checksum. Se forem diferentes, os dados foram corrompidos de algum modo. Isso também pode ser utilizado para detectar invasões na proteção e a corrupção deliberada de dados. Os dados corrompidos terão um checksum diferente e, assim, o fato de que a corrupção ocorreu poderá ser detectado.

Quando estruturas de dados ligadas são utilizadas, a representação pode se tornar redundante pela inclusão de referências retroativas. Ou seja, para cada referência de A para B, deverá existir uma referência equivalente de B para A. Também é possível manter a contagem do número de elementos na estrutura. A verificação pode determinar se as referências para frente e para trás são consistentes (elas devem se referir umas às outras, entre si) e se o tamanho armazenado e o tamanho da estrutura computada são os mesmos.

Quando os processos precisam reagir dentro de um período de tempo específico, um timer de watchdog pode ser instalado. Esse timer precisa ser reiniciado pelo processo em execução, depois que sua ação for concluída. Ele é iniciado juntamente com um processo e marca o tempo de execução desse processo. Ele pode ser interrogado por um controlador em intervalos regulares. Se, por alguma razão, o processo falhar e não terminar, o timer não será reiniciado. O controlador pode, portanto, detectar que surgiu um problema e tomar iniciativas para forçar o término do processo.

18.2.4 Recuperação de defeitos

A recuperação de defeitos é o processo de modificação do espaço de estado do sistema, de modo que os efeitos do defeito sejam minimizados. O sistema pode continuar em operação, talvez, de alguma maneira degradada. A recuperação para frente envolve tentar corrigir o estado danificado do sistema. A recuperação para trás restaura o estado do sistema para um estado conhecido e 'correto'.

Às vezes, a recuperação de erros para frente é de aplicação específica, com conhecimento do domínio, utilizado para computar possíveis correções de estado. Contudo, quando as informações de estado incluem redundância, as estratégias de recuperação de erros para frente podem, algumas vezes, ser utilizadas. Existem duas situações genéricas em que a recuperação de erros para frente pode ser aplicada:

1. *Quando os dados codificados são corrompidos* O uso de técnicas de codificação que acrescentam redundância aos dados permite que os erros sejam corrigidos e também detectados.
2. *Quando estruturas ligadas são corrompidas* Se os ponteiros para trás e para frente estiverem incluídos na estrutura de dados, ela poderá ser recriada, caso ponteiros suficientes permaneçam intactos. Essa técnica é freqüentemente utilizada para reparo de sistemas de arquivo e de banco de dados.

A recuperação de erros para trás é uma técnica mais simples, que faz a restauração do sistema para um estado seguro conhecido depois que um erro foi detectado. A maioria dos sistemas de bancos de dados inclui a recuperação de erros para trás. Quando um usuário inicia uma computação de um banco de dados, uma *transação* é iniciada. As modificações feitas durante essa transação não são imediatamente incorporadas ao banco de dados. O banco de dados somente é atualizado depois que a transação termina e nenhum problema é detectado. Se a transação falhar, o banco de dados não será atualizado.

As transações permitem a recuperação de erros, porque elas não fazem alterações no banco de dados até que estejam terminadas. Contudo, elas não permitem a recuperação a partir de mudanças de estado, que são válidas, mas incorretas. Checkpointing é uma técnica que pode permitir a recuperação a partir dessa situação. O estado do sistema é duplicado periodicamente. Quando um problema é descoberto, um estado correto pode ser restaurado a partir de uma dessas cópias.

Como exemplo de como a recuperação para trás pode ser implementada utilizando-se exceções, considere a classe Java SafeSort, mostrada na Figura 18.9, que inclui código para a detecção de erros e recuperação para trás.

O método copia o vetor antes da operação de sort (classificação). Nesse exemplo, é utilizada uma bubblesort, por questões de simplicidade, mas com certeza qualquer algoritmo de classificação pode ser utilizado. Se houver um erro no algoritmo de classificação e o vetor não estiver adequadamente classificado, isso será detectado por verificações explícitas na ordem dos elementos do vetor. Se o vetor não estiver adequadamente classificado, uma exceção SortError será lançada. O manipulador de exceção não tenta resolver o problema, mas restaura o valor original do vetor e lança novamente a SortError para indicar ao método que chamou que a classificação não foi bem-sucedida. A recuperação a partir do erro é de responsabilidade do método que chamou.

18.3 Arquiteturas tolerantes a defeitos

A programação defensiva é uma técnica eficaz para implementar a tolerância a defeitos. Ela é relativamente simples e, normalmente, não envolve muita complexidade adicionada ao sistema. Contudo, ela não pode lidar efetivamente com os defeitos de sistema que surgem a partir de interações entre o hardware e o software. Além disso, equívocos nos requisitos podem significar que o código do sistema e a defesa associada estão incorretos. Para sistemas mais críticos, principalmente aqueles com requisitos rigorosos de disponibilidade, uma arquitetura específica de sistema, projetada para aceitar a tolerância a defeitos, pode ser necessária. Entre os exemplos de sistemas que empregam essa abordagem de tolerância a defeitos estão os sistemas de aeronaves, que precisam estar em operação durante toda a permanência em vôo, os sistemas de telecomunicação e os sistemas críticos de comando e controle.

Há muitos anos havia a necessidade de construir hardware com tolerância a defeitos. A técnica mais comumente utilizada de tolerância a defeitos de hardware tem como base a noção de redundância modular tripla (*triple-modular redundancy* – TMR). A unidade de hardware é replicada três vezes (ou até mais). A saída de cada unidade é comparada. Se uma das unidades falhar e não produzir a mesma saída que as outras unidades, sua saída será ignorada. Um gerenciador de defeitos pode tentar reparar a unidade defeituosa de maneira automática, mas, se isso for impossível, o sistema será automaticamente reconfigurado, a fim de desativar a unidade. O sistema, então, continua a funcionar com duas unidades operando (Figura 18.10).

Essa abordagem de tolerância a defeitos se baseia no fato de que a maioria das falhas de hardware é resultante de falhas de componentes, e não de defeitos de projeto. Portanto, os componentes são propensos a falhar independentemente. Isso pressupõe que, quando totalmente operacionais, todas as unidades de hardware operam conforme foi especificado. Existe, portanto, uma pequena probabilidade de falha simultânea de componentes em todas as unidades de hardware.

Figura 18.10

Redundância modular tripla para resolver defeitos de hardware.

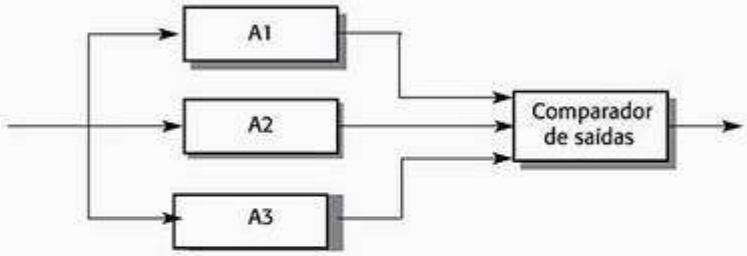
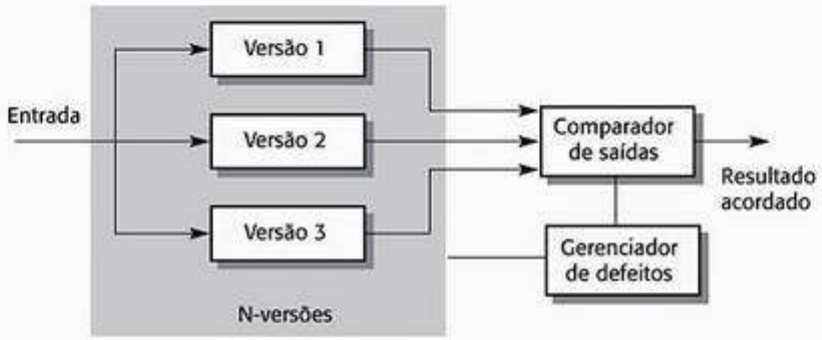


Figura 18.11

Programação de N-versões.



Naturalmente, os componentes podem ter um defeito em comum de projeto e, assim, produzir a mesma resposta (errada). A chance de que isso ocorra pode ser reduzida com a utilização de unidades de hardware que tenham uma especificação em comum, mas que sejam projetadas e construídas por diferentes fabricantes. Supõe-se que seja pequena a probabilidade de diferentes equipes cometerem o mesmo erro de projeto ou de fabricação.

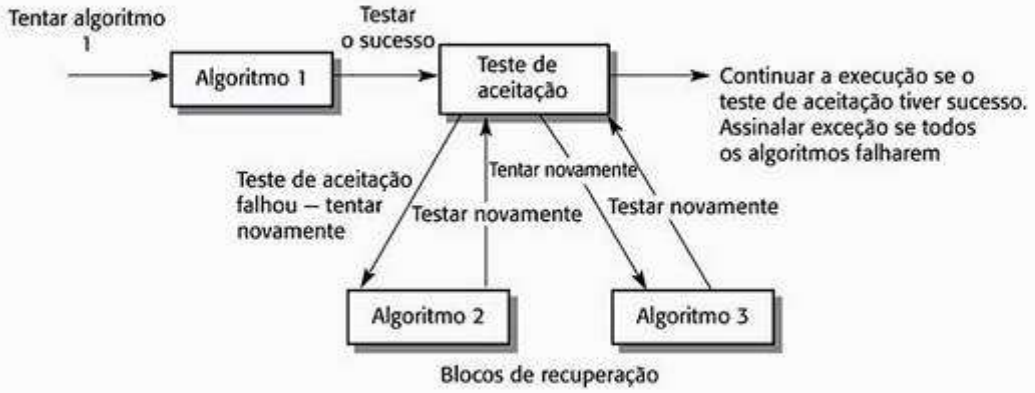
Se os requisitos de disponibilidade ou de confiabilidade para um sistema exigirem que o hardware seja tolerante a defeitos, então a tolerância a defeitos do software também será necessária. Existem duas abordagens semelhantes para possibilitar a tolerância a defeitos em software (veja as figuras 18.11 e 18.12). Ambas foram derivadas do modelo de hardware, em que componentes redundantes (ou, talvez, sistemas redundantes) são incluídos e componentes defeituosos podem ser desativados.

Essas duas abordagens para a tolerância a defeitos de software são:

1. *Programação de N-versões* Utilizando-se uma especificação comum, o sistema de software é implementado em uma série de diferentes versões, por diferentes equipes. Essas versões são executadas em paralelo, em computadores separados. Suas saídas são comparadas, de acordo com a utilização de um sistema de votação, e saídas inconsistentes, ou saídas que não sejam produzidas a tempo, são rejeitadas. Pelo menos três versões do sistema devem estar disponíveis, de maneira que duas versões devem ser consistentes, caso ocorra uma única falha. Essa é a abordagem mais comumente utilizada na tolerância a defeitos de software. Ela tem sido utilizada em sistemas de sinalização de estradas de ferro, em sistemas de aeronaves e em sistemas de proteção de reatores. Avizienis (1985, 1995) descreve essa abordagem.

2. *Blocos de recuperação* Nessa abordagem, cada componente de programa inclui um teste para verificar se o componente foi executado com sucesso. Isso inclui também um código alternativo, que permite que o sistema faça uma cópia e repita a computação, se o teste detectar uma falha. As implementações são, deliberadamente, diferentes interpretações da mesma especificação. Elas são executadas em seqüência, e não em paralelo; assim, o hardware replicado não é necessário. Em programação de N-versões, a implementação pode ser diferente, mas não é incomum que diferentes equipes escolham os mesmos algoritmos para implementar a especificação. Randell (1975) e Randell e Xu (1995) descrevem o método de blocos de recuperação.

Figura 18.12 ▶
Blocos de recuperação.



Essas abordagens são baseadas na noção de diversidade de projeto e implementação. Quando abordagens diversas são utilizadas para implementar a mesma especificação, é uma suposição razoável que diferentes versões do software não incluam os mesmos defeitos e, desse modo, é improvável que ocorram falhas comuns. A diversidade pode ser obtida de várias maneiras:

1. Incluindo requisitos para que diferentes abordagens de projeto sejam utilizadas. Por exemplo, uma equipe pode ser necessária para produzir um projeto orientado a objetos e outra equipe pode produzir um projeto orientado a funções.
2. Incluindo requisitos para que as implementações sejam escritas em diferentes linguagens de programação. Por exemplo, em um sistema com três versões, as linguagens Ada, C++ e Java podem ser utilizadas para escrever as versões de software.
3. Exigindo o uso de diferentes ferramentas e ambientes de desenvolvimento para o sistema.
4. Exigindo explicitamente o uso de diferentes algoritmos em algumas partes da implementação. Contudo, isso limita a liberdade da equipe de projeto e pode ser difícil a reconciliação com os requisitos de desempenho do sistema.

Cada equipe deve trabalhar com uma especificação separada (a V-spec), derivada da especificação de requisitos do sistema (Avizienis, 1995). Além de especificar a funcionalidade do sistema, a V-spec deve definir quando gerar saídas de sistema para comparação. As equipes de desenvolvimento de cada versão devem trabalhar isoladas para reduzir a possibilidade de desenvolver erros comuns sobre o sistema.

A diversidade de projetos certamente aumenta a confiabilidade geral do sistema. Contudo, uma série de experiências sugeriu que a suposição de que diferentes equipes de projeto não cometem os mesmos erros pode não ser sempre válida (Knight e Leveson, 1986; Brilliant *et al.*, 1990; Leveson, 1995). Diferentes equipes de desenvolvimento podem cometer os mesmos erros, devido a uma má interpretação comum da especificação ou porque, independentemente, chegam aos mesmos algoritmos para resolver o problema. Os blocos de recuperação reduzem a probabilidade de erros comuns porque diferentes algoritmos são utilizados para cada bloco de recuperação.

A inadequação de ambas as abordagens para tolerância a defeitos é que elas se baseiam na suposição de que a especificação é correta. Elas não aceitam erros de especificação. Em muitos casos, contudo, a especificação é incorreta ou está incompleta e, assim, o sistema se comporta de maneira inesperada. Um meio de reduzir a possibilidade de erros comuns de especificação é desenvolver a V-spec para o sistema independentemente e defini-la em diferentes linguagens. Portanto, uma equipe de desenvolvimento pode trabalhar a partir de uma especificação formal; outra, a partir de um modelo de sistema baseado em estado, e a terceira, a partir de uma especificação em linguagem natural. Isso ajuda a evitar alguns erros de interpretação de especificação, mas não contorna o problema de erros de especificação.

Possibilitar a tolerância a defeitos de software requer que ele seja executado sob o controle de um controlador tolerante a defeitos, que assegurará que as etapas envolvidas na tolerância a defeitos sejam executadas. Esse controlador examina as saídas e as compara. Se forem diferentes, algumas ações de recuperação poderão ser iniciadas. Laprie *et al.* (1995) descrevem arquiteturas de sistemas tolerantes a defeitos. Os leitores interessados devem consultar a seção "Leituras sugeridas" para obter mais informações sobre esse tópico.

18.4 Projeto de sistema seguro

Como regra, o projeto para software crítico de segurança deve ser feito com base na ocultação de informações e na simplicidade do software. As partes do sistema que são críticas de segurança devem ser isoladas de outras partes do sistema. Isso pode ser conseguido pelo uso da abstração de dados e controle ou pela separação física. O software crítico de segurança pode ser executado em um computador separado, com mínimos links de comunicação com outras partes do sistema.

Os softwares críticos de segurança devem ser tão simples quanto possível. As características de linguagem potencialmente sujeitas a erros, discutidas neste capítulo, devem ser evitadas sempre que possível. Elas podem, até mesmo, ser proibidas por padrões de desenvolvimento de sistemas críticos de segurança. Em alguns casos, pode ser um requisito que o sistema seja desenvolvido em um subconjunto de linguagem que tenha excluído características inseguras. Esses subconjuntos foram desenvolvidos para Modula-2, Ada e Pascal, e é provável que um subconjunto seguro em Java venha a ser desenvolvido. Esses subconjuntos excluem características de linguagem que não sejam adequadamente definidas e características que sejam inerentemente inseguras, como números reais e indicadores.

O software tolerante a defeitos deve somente ser utilizado em sistemas críticos de segurança, quando não houver nenhum estado seguro e quando a segurança do sistema depender de sua disponibilidade. Essas técnicas aumentam a complexidade, tornando o software mais difícil de ser validado. Como foi discutido na seção anterior, a pesquisa demonstrou que os argumentos a favor da confiabilidade por meio da diversidade (programação de N-versões) nem sempre são válidos. Quando se desenvolveu software a partir da mesma especificação, diferentes equipes cometeram os mesmos erros. A redundância de software não proporcionou o aumento teoricamente previsto na confiabilidade do sistema. Além disso, se a especificação for incorreta, todas as versões incluirão os erros comuns de especificação.

Isso não significa que a programação de N-versões seja inútil. Ela pode reduzir o número absoluto de falhas no sistema. Embora se admita que defeitos comuns estavam presentes nas diferentes versões do sistema, Bishop *et al.* (1986) constataram que o número de falhas do sistema foi reduzido. A programação de N-versões proporciona maior confiança, mas não confiança absoluta no sistema.

Em vez de utilizar a programação de N-versões, que aumenta a complexidade do sistema, uma alternativa seria manter o sistema o mais simples possível e dedicar muitos recursos para sua validação. Manter a simplicidade do software reduz a probabilidade de que erros sejam introduzidos. Isso também significa que os custos muito altos de validação de segurança serão reduzidos. Somente uma quantidade relativamente pequena de software está relacionada à segurança.

Parnas *et al.* (1990) apóiam essa abordagem. Eles sugerem que a segurança pode ser mais bem garantida minimizando e isolando componentes de código críticos de segurança e utilizando as técnicas o mais simples possível para escrever seções de código críticas de segurança. A validação deve ter como base uma combinação de testes completos, revisões com base em especificações matemáticas e um processo de desenvolvimento certificado.

PONTOS-CHAVE

- A confiança em um programa pode ser alcançada evitando-se a introdução de defeitos e incluindo recursos de tolerância a defeitos que permitam que o sistema permaneça operacional depois que o defeito tenha provocado uma falha no sistema.
- Alguns princípios e técnicas de programação, como declarações 'goto', ponteiros, recursão, herança e números em ponto flutuante, são inerentemente sujeitos a erros. Eles não devem ser utilizados no desenvolvimento de sistemas confiáveis.
- O uso de um processo bem-definido e repetível será essencial, se os defeitos em um sistema tiverem de ser minimizados. O processo deve incluir atividades de verificação e validação em todos os estágios, desde a definição de requisitos até a implementação do sistema.
- O software tolerante a defeitos pode continuar em execução apesar da ocorrência de defeitos que provocam falhas no sistema.
- Existem quatro aspectos da tolerância a defeitos em programas: a detecção de defeitos, a avaliação de danos, a recuperação de defeitos e o reparo de defeitos.
- A programação defensiva é uma técnica de programação que envolve incorporar no programa verificações de defeitos e código de recuperação de defeitos. Os defeitos são detectados antes que provoquem uma falha no sistema.
- Os recursos de manipulação de exceções, fornecidos em linguagens como Java e C++, são utilizados para aceitar a programação defensiva. Eles evitam a duplicação de códigos e localizam a manipulação de erros em um programa.
- A programação de N-versões e os blocos de recuperação são abordagens alternativas para arquiteturas de tolerância a defeitos, em que cópias redundantes de hardware e de software são mantidas. Ambas se apóiam na diversidade de projeto e no uso de um controlador de tolerância a defeitos para coordenar a execução de unidades de programa redundantes.

LEITURAS SUGERIDAS

Software fault tolerance. Essa coleção traz diversos artigos que discutem os blocos de recuperação e a programação de N-versões. Ela traz também um bom artigo sobre arquiteturas de sistemas de tolerância a defeitos. (M.R. Lyu (ed.), 1995, John Wiley and Sons.)

EXERCÍCIOS

18.1 Explique por que a herança é uma construção potencialmente sujeita a erros e por que seu uso deve ser minimizado quando se desenvolvem sistemas críticos em uma linguagem orientada a objetos.

18.2 Considerando que a recursão é uma construção inerentemente sujeita a erros, projete uma classe de objetos para implementar árvores binárias que não utilizem a recursão em sua implementação.

18.3 Descreva três técnicas de programação defensiva que podem ser utilizadas para reduzir a probabilidade de que um defeito de software leve a uma falha de sistema.

18.4 Descreva brevemente as estratégias de recuperação para a frente e para trás de defeitos. Por que a recuperação de defeitos para trás é utilizada com mais freqüência que a recuperação para a frente? Dê dois exemplos de classes de sistema em que a recuperação para trás de defeitos pode ser utilizada.

18.5 Que pré-condições devem prevalecer antes que a recuperação de erros para a frente possa ser implementada em um sistema tolerante a defeitos? A recuperação de erros para a frente é possível em sistemas interativos?

18.6 Projete um tipo abstrato de dados ou classe de objeto, chamado RobustList, que implemente a recuperação de erros para a frente em uma lista ligada. Inclua operações para verificar a lista quanto à possibilidade de corrupção e para reconstruí-la, caso alguma corrupção tenha ocorrido. Suponha que seja possível verificar a corrupção mantendo referências para a frente e para trás e a partir de membros adjacentes da lista.

18.7 Sugira circunstâncias em que seja apropriado utilizar uma arquitetura tolerante a defeitos, quando se implementa um sistema de controle baseado em software, e explique por que essa abordagem é necessária.

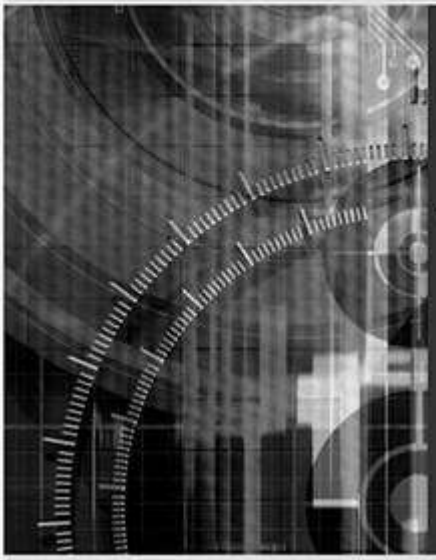
18.8 Foi sugerido que o software de controle para uma máquina de terapia de radiação (utilizada para o tratamento de pacientes com câncer) fosse implementado com a utilização da programação de N-versões. Comente se você acha que essa é uma boa sugestão ou não.

18.9 Dê duas razões pelas quais todas as diferentes versões de sistema, em um sistema de N-versões, podem falhar de maneira similar.

18.10 Discuta os problemas de desenvolvimento e de manutenção de sistemas non-stop, como o software de comutação telefônica. Como poderiam ser utilizadas exceções no desenvolvimento desses sistemas?

18.11 A utilização das técnicas discutidas neste capítulo para produzir software seguro obviamente envolve consideráveis custos extras. Que custos extras podem ser justificados, se cem vidas forem salvas durante os 15 anos do tempo de duração de um sistema? Seriam os mesmos custos justificáveis, se dez vidas fossem salvas? Quanto vale uma vida? A capacidade de ganho das pessoas afetadas faz diferença nesse julgamento?

PÁGINA EM BRANCO



PARTE CINCO

Verificação e Validação

PÁGINA EM BRANCO

19 Verificação e validação

Objetivos

O objetivo deste capítulo é introduzir a verificação e a validação de software, com uma abordagem particular das técnicas estáticas de verificação. Depois de ler este capítulo, você poderá:

- compreender as diferenças entre verificação e validação de software;
- conhecer as inspeções de programa como um método para descobrir defeitos em programas;
- compreender por que a análise estática de programas é uma importante técnica de verificação, e
- compreender o método Cleanroom para o desenvolvimento de programas e por que ele pode ser eficaz.

Sumário

19.1 Planejamento de verificação e validação
19.2 Inspeções de software
19.3 Análise estática automatizada
19.4 Desenvolvimento de software Cleanroom

Verificação e validação (V & V) é o nome dado aos processos de verificação e análise que asseguram que o software cumpra com suas especificações e atenda às necessidades dos clientes que estão pagando por ele. A verificação e validação constituem um processo de ciclo de vida completo, começando com as revisões dos requisitos e continuando com as revisões de projeto e as inspeções de código até chegar aos testes de produto. Deve haver atividades de V & V em cada estágio do processo de software. Essas atividades verificam se os resultados das atividades de processo estão conforme o especificado.

A verificação e validação não são a mesma coisa, embora sejam facilmente confundidas. A diferença entre elas é sucintamente expressa por Boehm (1979):

- 'Validação: estamos construindo o produto certo?'
- 'Verificação: estamos construindo certo o produto?'

Essas definições nos dizem que o papel da verificação envolve checar se o software cumpre com suas especificações. É preciso verificar se o sistema cumpre com seus requisitos funcionais e não funcionais especificados. A validação é um processo mais genérico. É necessário assegurar que o software atenda às expectativas do cliente, e isso vai além de checar a conformidade do sistema com suas especificações, a fim de mostrar que o software faz o que o cliente espera que faça, exatamente como foi especificado.

Como foi discutido no Capítulo 6, é muito importante a validação, com antecedência, dos requisitos do sistema. É fácil cometer erros e omissões nos requisitos do sistema, e nesses casos o software final provavelmente não atenderá às expectativas do cliente. Contudo, na realidade, é provável que a validação de requisitos não descubra todos os problemas com os requisitos. Alguns defeitos e deficiências nos requisitos podem, às vezes, ser descobertos somente quando a implementação do sistema está completa.

Dentro do processo de V & V, duas técnicas de checagem e análise de sistemas podem ser utilizadas:

1. As *inspeções de software*, que analisam e verificam as representações do sistema, como o documento de requisitos, os diagramas de projeto e o código-fonte do programa. As inspeções podem ser aplicadas em todos os estágios do processo e ser complementadas por alguma análise automática do texto de origem de um sistema ou dos documentos associados. As inspeções de software e as análises automatizadas são técnicas estáticas de V & V, uma vez que não requerem que o sistema seja executado.

2. Os *testes de software*, que envolvem executar uma implementação do software com os dados de teste e examinar as saídas dele e seu comportamento operacional, a fim de verificar se ele está sendo executado conforme o esperado. Os testes são uma técnica dinâmica de verificação e validação porque trabalham com uma representação executável do sistema.

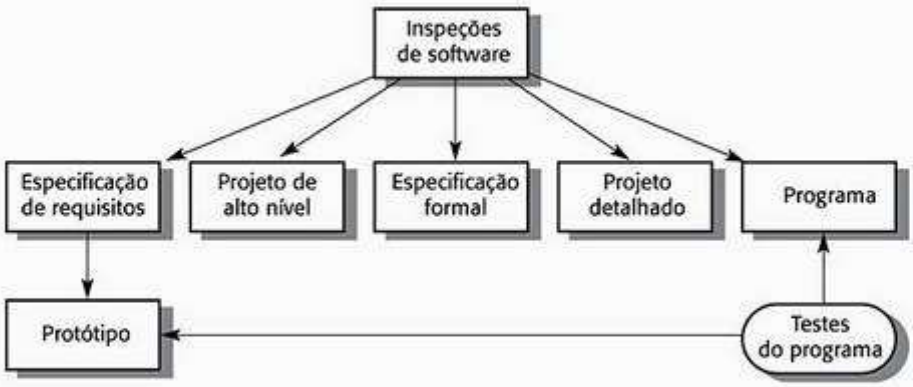
A Figura 19.1 mostra a posição das inspeções e dos testes no processo de software. As setas indicam os estágios no processo em que as técnicas podem ser utilizadas. Portanto, as inspeções de software podem ser empregadas em todos os estágios do processo de software. Os testes, contudo, somente podem ser utilizados quando um protótipo ou um programa executável estiver disponível.

As técnicas de inspeção incluem inspeções de programa, análises de código-fonte automatizadas e verificação formal. Contudo, as técnicas estáticas somente podem verificar a correspondência entre um programa e sua especificação (verificação); elas não podem demonstrar que o software é operacionalmente útil nem checar suas características não funcionais, como seu desempenho e sua confiabilidade. Portanto, alguns testes são sempre necessários para validar um sistema de software.

Embora as inspeções de software sejam, hoje em dia, amplamente utilizadas, os testes de programação são ainda a técnica predominante de verificação e validação. Esses testes envolvem executar o programa com a utilização de dados, como os dados reais processados por ele. A existência de defeitos ou inadequações no programa é inferida pelo exame das saídas do programa e pela busca de anomalias. Os testes podem ser realizados durante a fase de imple-

Figura 19.1 ▶

Verificação e validação estática e dinâmica.



mentação, a fim de verificar se o software se comporta como pretendido pelo seu projetista, e depois que a implementação estiver terminada.

Existem dois tipos distintos de testes que podem ser utilizados em diferentes estágios no processo de software:

1. Os *testes de defeitos*, que se destinam a encontrar inconsistências entre o programa e sua especificação. Essas inconsistências são, geralmente, devidas a defeitos no programa. Os testes são projetados para revelar a presença de defeitos no sistema em vez de simular seu uso operacional. Esse tipo de testes será abordado no Capítulo 20.

2. Os *testes estatísticos*, que são utilizados para testar o desempenho e a confiabilidade do programa e para checar como ele trabalha sob condições operacionais. Os testes são projetados para refletir a freqüência das entradas do usuário. Depois de realizar os testes, uma estimativa da confiabilidade operacional do sistema pode ser feita pela contagem do número de falhas observadas no sistema. O desempenho do programa pode ser julgado medindo-se o tempo de execução e o tempo de resposta do sistema quando ele processa os dados estatísticos de teste. Serão discutidos os testes estatísticos e a estimativa de confiabilidade no Capítulo 21.

Naturalmente, não existe uma delimitação nítida e rápida entre essas abordagens dos testes. Durante os testes para a verificação de defeitos, os responsáveis pelos testes terão uma percepção intuitiva da confiabilidade do software; durante os testes estatísticos, é provável que alguns defeitos sejam descobertos.

A meta definitiva do processo de verificação e validação é estabelecer a confiança de que o sistema de software é 'adequado a seu propósito'. Isso não significa que o programa tenha de ser inteiramente livre de defeitos. Em vez disso, significa que o sistema deve ser suficientemente bom para o uso pretendido. O nível de confiança necessário depende do propósito do sistema, das expectativas dos usuários e do atual ambiente de mercado para o sistema:

1. *Função do software* O nível de confiança depende do quanto o software é importante para uma organização. Por exemplo, o nível de confiança exigido para o software utilizado para controlar um sistema de segurança crítica é muito mais alto do que o requerido para um sistema de software empregado como protótipo, que tenha sido desenvolvido para demonstrar algumas novas idéias.

2. *Expectativas do usuário* É uma triste consideração, na indústria de software, o fato de que muitos usuários tenham poucas expectativas em relação a seu software e que não se surpreendam quando ele falha durante o uso. Eles estão dispostos a aceitar essas falhas de sistema quando os benefícios do uso sobrepujam as desvantagens. Contudo, a tolerância do usuário às falhas de sistema tem decrescido desde a década de 90. Atualmente, é menos aceitável a entrega de sistemas não confiáveis; assim, as empresas de software devem dedicar maior esforço à verificação e validação.

3. *Ambiente de mercado* Quando um sistema é comercializado, seus vendedores precisam levar em conta os programas dos concorrentes, o preço que os clientes estão dispostos a pagar por ele e o cronograma para a entrega desse sistema. Quando uma companhia tem poucos concorrentes, ela pode decidir entregar um programa antes de ele estar inteiramente testado e depurado, porque quer ser a primeira a apresentar o produto no mercado. Quando os clientes não estão dispostos a pagar altos preços por um software, eles podem estar dispostos a tolerar mais defeitos de software. Todos esses fatores devem ser considerados quando se decide quanto esforço deve ser dedicado ao processo de V & V.

Durante a verificação e validação de software, normalmente os defeitos no programa são descobertos e o programa tem de ser então modificado, a fim de corrigir esses defeitos. Esse processo de depuração é, freqüentemente, integrado com outras atividades de verificação e validação. Contudo, os testes (ou, mais geralmente, a verificação e validação) e a depuração são processos diferentes, que não devem ser integrados:

1. A verificação e validação são um processo que estabelece a existência de defeitos em um processo de software.
2. A depuração é um processo (Figura 19.2) que localiza e corrige esses defeitos.

Não existe um método simples para a depuração de um programa. Os depuradores hábeis procuram por padrões nas saídas dos testes em que o defeito é exibido e utilizam seu conhecimento do tipo de defeito, o padrão de saída, a linguagem de programação e o processo de programação para localizar o defeito. O conhecimento do processo é importante. Os depuradores conhecem os erros comuns dos programadores (como deixar de adicionar um contador) e comparam esses erros em relação aos padrões observados. Os erros característicos da linguagem de programação, como os erros de direcionamento de ponteiros em C, podem também ser considerados.

Localizar os defeitos em um programa nem sempre é um processo simples, uma vez que o defeito não precisa necessariamente estar perto do ponto em que o programa falhou. Para localizar um defeito em um programa, o programador responsável pela depuração pode ter de projetar programas de teste adicionais, que repitam o defeito original e ajudem a descobrir a origem dele no programa. O rastreamento manual do programa e a execução simulada podem ser necessários. Em alguns casos, ferramentas de depuração que coletam informações sobre a execução do programa podem ser úteis.

As ferramentas interativas de depuração são, geralmente, parte de um conjunto de ferramentas de suporte a linguagem, que são integradas como um sistema de compilação. Elas fornecem um ambiente especializado de *run-time* para o programa, que permite o acesso à tabela de símbolos do compilador e, a partir daí, aos valores das variáveis do programa. Freqüentemente, os usuários podem controlar a execução, ao longo do programa, considerando declaração por declaração. Depois de cada declaração ter sido executada, os valores das variáveis podem ser examinados e os erros em potencial são descobertos.

Depois de descoberto um defeito no programa, ele tem de ser corrigido e o sistema deve ser revalidado. Isso pode envolver re-inspecionar o programa ou repetir os testes anteriores

Figura 19.2 ▶
Processo de depuração.

(testes de regressão). Os testes de regressão são utilizados para checar se as mudanças feitas em um programa não introduziram novos erros no sistema. A experiência tem mostrado que uma proporção relativamente alta de 'reparos de defeitos' são reparos incompletos ou introduzem novos defeitos no programa.

Em princípio, durante os testes de regressão, todos os testes devem ser repetidos depois de cada reparo de um defeito; na prática, isso é muito dispendioso. Como parte do plano de testes, as dependências entre partes do sistema e os testes associados com cada parte devem ser identificados. Isto é, deve haver facilidade de rastreamento a partir dos casos de teste até as características do programa que são testadas. Se essa facilidade de rastreamento estiver documentada, então é possível executar um subconjunto de todos os dados de testes, a fim de checar o componente modificado e seus dependentes.

19.1 Planejamento de verificação e validação

O processo de verificação e validação é dispendioso. Para alguns sistemas grandes, como os sistemas de tempo real, com complexas restrições não funcionais, metade do orçamento do desenvolvimento do sistema pode ser gasto em V & V. Um planejamento cuidadoso é necessário para obter o melhor das inspeções e dos testes e para controlar os custos do processo de verificação e validação.

O planejamento de verificação e validação de um sistema de software deve se iniciar no começo do processo de desenvolvimento. O modelo de desenvolvimento de software indicado na Figura 19.3 mostra como os planos de testes devem ser derivados da especificação e do projeto de sistema. Algumas vezes, isso é chamado de modelo em V (vire a Figura 19.3 para ver o V). Esse diagrama também mostra como a atividade de verificação e validação é dividida em uma série de estágios, com cada fase dirigida por testes que foram definidos para checar a conformidade do programa com seu projeto e sua especificação.

O processo de planejamento de V & V deve decidir sobre o equilíbrio entre as abordagens estáticas e dinâmicas para a verificação e validação, especificar padrões e procedimentos para as inspeções e os testes de software, estabelecer checklists para orientar as inspeções de programa (veja a Seção 19.2) e definir o plano de teste de software. O esforço relativo devotado a inspeções e testes depende do tipo de sistema que está sendo desenvolvido e da habilidade organizacional. Quanto maior for a importância de um sistema, mais esforço deve ser dedicado às técnicas estáticas de verificação.

O planejamento de testes se ocupa em estabelecer padrões para o processo de testes, em vez de descrever testes de produtos. Os planos de testes não são apenas documentos de gerenciamento. Eles se destinam também aos engenheiros de software envolvidos em projetar e

Figura 19.3 ▼

Planos de teste como ligação entre o desenvolvimento e os testes.

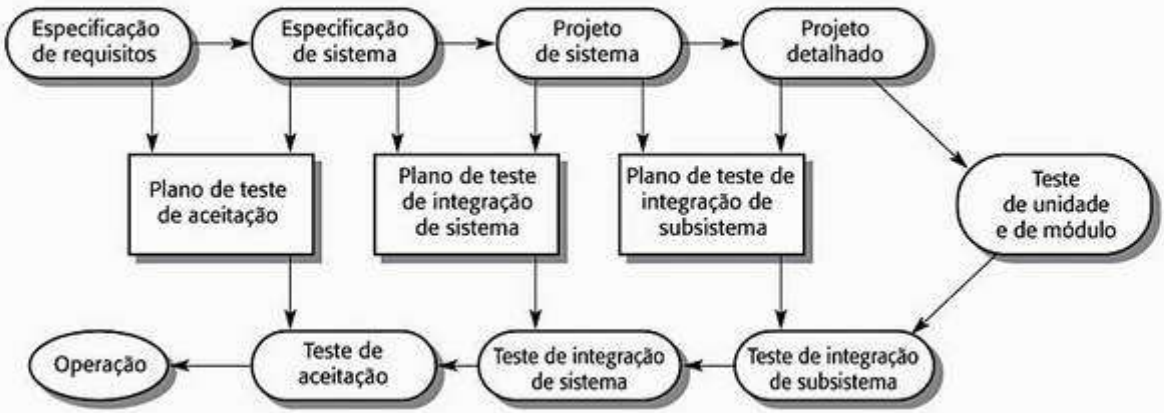


Figura 19.4

Estrutura de um plano de teste de software.

O processo de teste
Uma descrição das principais fases do processo de teste, que podem ser como descrito anteriormente neste capítulo.

Facilidade de rastreamento dos requisitos
Os usuários são os grandes interessados em que o sistema cumpra com seus requisitos e os testes devem ser planejados de modo que todos os requisitos sejam individualmente testados.

Itens a serem testados
É preciso especificar os produtos do processo de software que devem ser testados.

Cronograma de testes
Um cronograma geral dos testes e a alocação de recursos para esse cronograma. Isso, obviamente, está vinculado ao cronograma mais geral de desenvolvimento de projeto.

Procedimentos de registro de testes
Não é suficiente simplesmente executar testes. Os resultados dos testes devem ser sistematicamente registrados. Deve ser possível fazer a auditoria do processo de testes, a fim de checar se eles foram realizados corretamente.

Requisitos de hardware e software
Essa seção deve estabelecer as ferramentas de software requeridas e a estimativa de utilização de hardware.

Restrições
As restrições que afetam o processo de testes, como falta de pessoal, devem ser previstas nessa seção.

realizar os testes de sistemas e ajudam o pessoal técnico a ter uma visão geral dos testes de sistema e a situar seu trabalho nesse contexto. Os planos de testes também fornecem informações ao pessoal responsável por assegurar que os recursos apropriados de hardware e software estejam disponíveis para o grupo de testes.

Os componentes principais de um plano de testes são mostrados na Figura 19.4. Esse plano deve incluir muitos dos possíveis acontecimentos inesperados, para prever e contornar possíveis atrasos de projeto e de implementação e para que o pessoal alocado para os testes possa ser distribuído para outras atividades. Uma boa descrição dos planos de testes e de sua relação com planos de qualidade mais genéricos é fornecida por Frewin e Hatton (1986).

Como outros planos, o plano de testes não é um documento estático. Ele deve ser revisado regularmente, uma vez que os testes são uma atividade dependente da conclusão da implementação. Se uma parte do sistema estiver incompleta, ele não poderá ser entregue para os testes de integração, como será discutido no capítulo seguinte. Portanto, o plano deve ser revisado, a fim de redistribuir os responsáveis pelos testes para outras atividades.

19.2 Inspeções de software

Os testes sistemáticos de programa exigem grande número de testes a serem desenvolvidos, executados e examinados. Isso significa que o processo é demorado e dispendioso. Cada teste executado tende a descobrir um único defeito no programa ou, quando muito, apenas alguns defeitos. A razão disso é que falhas resultantes de defeitos de sistema freqüentemente acarretam a deterioração de dados do sistema. Conseqüentemente, pode ficar difícil dizer se anomalias de saída são resultado de um novo defeito ou se são efeitos colaterais de um defeito existente.

As inspeções de software não requerem que o programa seja executado; assim, podem ser utilizadas como uma técnica de verificação antes que os programas sejam implementados. Durante uma inspeção, examina-se a representação original de um sistema, que pode ser um modelo de sistema, uma especificação ou um código de linguagem de alto nível. Utiliza-se

o conhecimento do sistema a ser desenvolvido e as semânticas da representação original para descobrir erros. Cada erro pode ser considerado isoladamente, sem a preocupação sobre como ele afetará o comportamento do sistema. As interações de erros não são significativas, e um componente inteiro pode ser verificado em uma única sessão.

As inspeções provaram ser uma técnica eficaz de detecção de erros. Os erros podem ser encontrados de um modo mais barato, por meio da inspeção, do que com extensivos programas de testes. Isso foi demonstrado em um experimento feito por Basili e Selby (1987), que empiricamente compararam a eficácia das inspeções e dos testes. Eles constataram que a revisão estática de código era mais eficaz e menos dispendiosa que os testes de defeitos para descobrir defeitos em um programa. Gilb e Graham (1993) também constataram que isso é verdadeiro.

Fagan (1986) relatou que mais de 60 por cento dos erros em um programa podem ser detectados utilizando-se inspeções informais do programa. Mills *et al.* (1987) sugerem que uma abordagem mais formal, que utilize verificação matemática, pode detectar mais de 90 por cento dos erros em um programa. Essa técnica é empregada no processo Cleanroom, que será descrita na Seção 19.3. Esse processo de inspeção pode também considerar outros atributos da qualidade, como a conformidade com padrões, a portabilidade e a facilidade de manutenção. Esses atributos da qualidade serão discutidos no Capítulo 24.

Há duas razões pelas quais as revisões e as inspeções são, geralmente, mais eficazes do que os testes para descobrir defeitos em componentes e subsistemas:

1. Muitos defeitos diferentes podem ser detectados em uma única sessão de inspeção. O problema com os testes é que, freqüentemente, eles podem detectar somente um erro por teste, porque os defeitos podem fazer com que o programa seja interrompido ou podem interferir com os sintomas de outros defeitos do programa.
2. Elas reutilizam o conhecimento de domínio e de linguagem de programação. Em essência, os revisores provavelmente já viram os tipos de erros que ocorrem com freqüência em linguagens de programação específicas e em determinados tipos de aplicação. Eles podem, portanto, se concentrar nesses tipos de erros durante a análise.

Isso não significa que as inspeções devam substituir inteiramente os testes de sistema. Em vez disso, elas devem ser utilizadas como um processo inicial de verificação, para levantar a maioria dos defeitos do programa. As inspeções de software podem verificar a conformidade com uma especificação, mas não podem validar o comportamento dinâmico. Além disso, é quase sempre impraticável inspecionar um sistema inteiro, que é integrado por uma série de diferentes subsistemas. Os testes são a única técnica de V & V possível no nível do sistema. Os testes são também necessários para a avaliação da confiabilidade, a análise do desempenho, a validação da interface com o usuário e para checar se os requisitos de software são o que o usuário realmente quer.

Revisões e testes não são técnicas concorrentes de V & V. Elas têm suas próprias vantagens e desvantagens e devem ser utilizadas em conjunto no processo de verificação e validação. Na verdade, Gilb e Graham (1993) sugerem que um dos mais eficazes usos das revisões é a revisão dos casos de teste para um sistema. As revisões podem descobrir problemas com esses testes e podem ajudar a projetar maneiras mais eficazes de testar o sistema.

Às vezes, é difícil introduzir inspeções em organizações tradicionais de desenvolvimento de software. Os engenheiros de software com experiência em testes de programas podem relutar em aceitar que essas técnicas possam ser mais eficazes do que os testes para a detecção de defeitos. Os gerentes podem ter dúvidas sobre essas técnicas, porque elas requerem custos adicionais durante o projeto e o desenvolvimento, e podem não querer assumir o risco de que não haverá uma economia correspondente durante os testes de programa. As inspeções inevitavelmente sobrecarregam os custos iniciais de V & V de software e somente resultam em economia depois que os grupos de desenvolvimento se tornam experientes em seu uso.

Serão discutidas as inspeções de software neste capítulo, com descrição das inspeções de programa em que o código-fonte é inspecionado à procura de defeitos. Contudo, as técnicas

de inspeção de software podem ser aplicadas a quaisquer documentos produzidos durante o processo de software. Técnicas comparáveis de inspeção podem ser aplicadas a especificações de requisitos, definições detalhadas de projeto, projetos de estrutura de dados, planos de testes e documentação do usuário.

19.2.1 Inspeções de programa

As inspeções de programa são revisões cujo objetivo é a detecção de defeitos no programa. A noção de processo formalizado de inspeção foi primeiramente desenvolvida na IBM, na década de 70, e é descrita por Fagan (1976, 1986). Atualmente, esse é um método amplamente utilizado de verificação de programa. A partir do método original de Fagan, uma série de abordagens alternativas de inspeção foram desenvolvidas (Gilb e Graham, 1993). Contudo, todas elas têm como base a noção original de Fagan, de que uma equipe com membros que apresentem diferentes experiências deve fazer uma revisão cuidadosa, linha por linha, do código-fonte do programa.

A diferença fundamental entre as inspeções de programa e outros tipos de revisão de qualidade é que a principal meta das inspeções é detectar defeitos, em vez de considerar questões mais amplas de projeto. Os defeitos podem ser erros lógicos, anomalias no código que podem indicar uma condição errônea ou a não-conformidade com padrões organizacionais ou de projeto. Por outro lado, outros tipos de revisão podem se ocupar mais com os cronogramas, os custos, o progresso em comparação com metas predefinidas ou a avaliação quanto a se o software cumprirá ou não as metas organizacionais.

O processo de inspeção é um processo formal, realizado por uma pequena equipe de pelo menos quatro pessoas. Os membros da equipe analisam sistematicamente o código e apontam possíveis defeitos. Nas propostas originais de Fagan, ele sugeriu funções a serem assumidas, como autor, leitor, testador e moderador. O leitor lê o código em voz alta para a equipe de inspeção, o testador inspeciona o código a partir de uma perspectiva de teste e o moderador organiza o processo.

À medida que as organizações ganharam experiência com a inspeção, emergiram outras propostas para os papéis na equipe. Em uma discussão sobre como a inspeção foi bem-sucedida no processo de desenvolvimento da Hewlett-Packard, Grady e Van Slack (1994) sugeriram seis papéis, como mostra a Figura 19.5. Diferentes papéis podem ser adotados pela mesma pessoa, de maneira que o tamanho da equipe pode variar de uma inspeção para outra.

Grady e Van Slack relataram que nem sempre existe a necessidade do papel de um leitor. A esse respeito, eles modificaram o processo a partir do que foi originalmente proposto

Figura 19.5 ▶ Papéis no processo de inspeção.

Papel	Descrição
Autor ou proprietário	O programador ou o projetista responsável por produzir o programa ou o documento. Responsável pela correção de defeitos descobertos durante o processo de inspeção.
Inspetor	Encontra erros, omissões e inconsistências em programas e documentos. Pode também identificar questões mais amplas que estejam fora da abordagem da equipe de inspeção.
Leitor	Lê o código ou o documento em uma reunião de inspeção.
Relator	Registra os resultados da reunião de inspeção.
Presidente ou moderador	Gerencia o processo e facilita a inspeção. Relata os resultados de processo ao moderador-chefe.
Moderador-chefe	Responsável pelas melhorias do processo de inspeção, pela atualização de checklists, pelo desenvolvimento de padrões etc.

por Fagan, em que uma parte integral do processo envolvia ler o programa em voz alta. Gilb e Graham (1993) também não acham que um leitor seja necessário. Eles sugerem que os inspetores devem ser selecionados para refletir diferentes pontos de vista, como o dos testes, do usuário final, do gerenciamento de qualidade etc.

Antes de um programa de inspeções começar, é essencial que:

1. Haja uma especificação precisa do código a ser inspecionado. É impossível inspecionar um componente com o nível de detalhes necessário para detectar defeitos sem ter uma especificação completa.
2. Os membros da equipe de inspeção estejam familiarizados com os padrões organizacionais.
3. Exista uma versão do código disponível que esteja atualizada e sintaticamente correta. Não há razão para inspecionar código que esteja 'quase completo', mesmo que um atraso provoque uma interrupção no cronograma.

Um processo de inspeção muito genérico é mostrado na Figura 19.6. Esse processo é adaptado conforme necessário pelas organizações que utilizam inspeções de programa.

O moderador é responsável pelo planejamento da inspeção. Isso envolve selecionar uma equipe de inspeção, organizar uma sala de reuniões e garantir que o material a ser inspecionado e suas especificações estejam completos. O programa a ser inspecionado é apresentado à equipe de inspeção durante o estágio de introdução, em que o autor do código descreve o que o programa se destina a fazer. Em seguida, há um período de preparo individual. Cada membro da equipe de inspeção estuda a especificação e o programa e procura defeitos no código.

A inspeção deve ser relativamente curta (com duração de não mais de duas horas) e deve se ocupar, com exclusividade, em identificar defeitos, anomalias e não-conformidades com padrões. A equipe de inspeção não deve sugerir como esses defeitos devem ser corrigidos nem recomendar modificações em outros componentes.

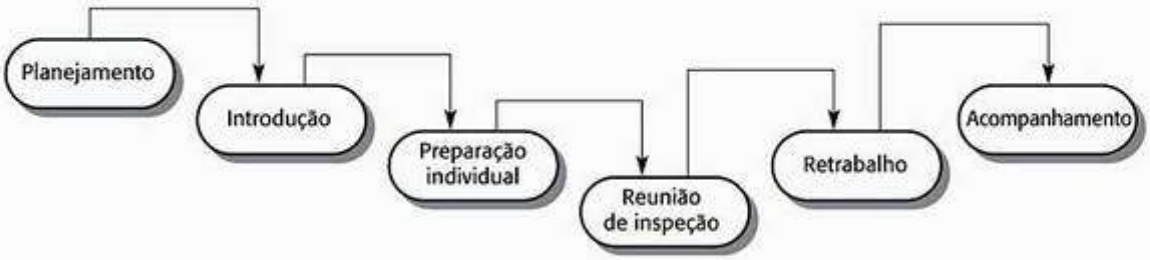
Em seguida à inspeção, o programa é modificado pelo seu autor, a fim de corrigir os problemas identificados. No estágio de acompanhamento, o moderador deve decidir se uma nova inspeção do código é necessária. Como alternativa, ele pode decidir que uma nova inspeção completa não é necessária e que os defeitos foram corrigidos com sucesso. Em seguida o documento é aprovado pelo moderador para liberação.

O processo de inspeção deve sempre ser dirigido por uma checklist de erros comuns dos programadores. Ela deve ser estabelecida pela discussão com o pessoal experiente e precisa ser regularmente atualizada, à medida que maior experiência seja adquirida com o processo de inspeção. Diferentes checklists devem ser preparadas para diferentes linguagens de programação.

Essa checklist varia de acordo com a linguagem de programação, devido aos diferentes níveis de checagem fornecidos pelo compilador de linguagem. Por exemplo, um compilador Ada confere se as funções têm o número correto de parâmetros; um compilador C não faz isso. As possíveis checagens que podem ser feitas durante o processo de inspeção são mostradas na Figura 19.7. Gilb e Graham (1993) enfatizam que cada organização deve desenvolver sua própria checklist, que deve ter como base padrões e práticas locais e que precisa ser atualizada quando são descobertos novos tipos de defeitos.

Figura 19.6 ▼

Processo de inspeção.



Classe de defeitos	Checagem de inspeção
Defeitos nos dados	Todas as variáveis de programa são iniciadas antes de seus valores serem utilizados? Todas as constantes foram denominadas? O limite superior de vetores deve ser igual ao tamanho do vetor ou ao tamanho −1? Se as *strings* de caracteres são utilizadas, um delimitador é explicitamente designado? Existe alguma possibilidade de overflow de buffer?
Defeitos de controle	Para cada declaração condicional, a condição está correta? Cada loop está certo para terminar? As declarações compostas estão corretamente entre parênteses? Em declarações 'case', todos os casos são levados em conta? Se um identificador break é requerido após cada caso em declarações 'case', esse identificador foi incluído?
Defeitos de entrada/saída	Todas as variáveis de entrada são utilizadas? Todas as variáveis de saída têm um valor designado antes de saírem? Entradas inesperadas podem fazer com que os dados sejam corrompidos?
Defeitos de interface	Todas as chamadas de funções e métodos têm o número correto de parâmetros? Os tipos formais e reais de parâmetros combinam? Os parâmetros estão na ordem certa? Se componentes acessam memória compartilhada, eles têm o mesmo modelo da estrutura de memória compartilhada?
Defeitos de gerenciamento de armazenamento	Se uma estrutura ligada é modificada, todos os links foram corretamente redesignados? Se o armazenamento dinâmico é utilizado, o espaço foi alocado corretamente? O espaço é explicitamente liberado, depois que não é mais necessário?
Defeitos de gerenciamento de exceções	Todas as possíveis condições de erros foram levadas em conta?

Figura 19.7 ▲

Checagens de inspeção.

À medida que uma organização adquire experiência no processo de inspeção, ela pode utilizar os resultados desse processo como meio de aperfeiçoá-lo. Pode ser feita uma análise dos defeitos encontrados durante o processo de inspeção. A equipe de inspeção e os autores do código que foi inspecionado podem sugerir razões pelas quais esses defeitos ocorreram. Sempre que possível, o processo deve ser modificado para eliminar as razões dos defeitos, de modo que não ocorram de novo em sistemas futuros.

Gilb e Graham relatam que uma série de organizações abandonou o teste de unidades em favor das inspeções. Eles constataram que as inspeções de programa são tão eficazes em detectar erros que os custos do teste de unidade não são justificáveis. Como será discutido mais adiante neste capítulo, as inspeções de software substituíram os testes de unidade no processo Cleanroom do desenvolvimento de software.

A quantidade de código que pode ser inspecionada em um determinado tempo depende da experiência da equipe de inspeção, da linguagem de programação e do domínio da aplicação. Quando o processo de inspeção foi medido na IBM, Fagan fez as seguintes observações:

1. Cerca de 500 declarações de código-fonte por hora podem ser consideradas durante o estágio de revisão geral.
2. Durante a preparação individual, cerca de 125 declarações de código-fonte por hora podem ser examinadas.
3. De 90 a 125 declarações por hora podem ser inspecionadas durante a reunião.

Esses números foram confirmados por dados coletados pela AT & T (Barnard e Price, 1994), quando medições do processo de inspeção evidenciaram valores comparáveis.

Fagan sugere que o tempo máximo gasto em uma inspeção deve ser de cerca de duas horas, uma vez que a eficiência do processo de detecção de defeitos diminui depois desse período. A inspeção, portanto, deve ser um processo freqüente, realizado em componentes de software relativamente pequenos, durante o desenvolvimento do programa.

Com quatro pessoas envolvidas em uma equipe de inspeção, o custo de inspecionar cem linhas de código é aproximadamente equivalente ao esforço de uma pessoa/dia. Isso supõe que a inspeção propriamente dita leva cerca de uma hora e que cada membro da equipe gasta de uma a duas horas para se preparar para a inspeção. Os custos de testes são muito variáveis e dependem do número de defeitos no programa. Contudo, o esforço requerido para a inspeção de programa é possivelmente menor do que a metade do esforço que seria exigido para os testes de defeitos equivalentes.

A introdução da avaliação tem implicações para o gerenciamento de projetos, e o gerenciamento sensível é importante, se a inspeção for aceita pelas equipes de desenvolvimento de software. As inspeções são um processo público de detecção de erros, comparadas com o processo mais privativo de testes e, inevitavelmente, os enganos cometidos por indivíduos são revelados para toda a equipe de programação.

Os gerentes devem se assegurar de que existe uma clara separação entre as inspeções e as avaliações do pessoal. Os resultados de inspeção que revelarem erros nunca deverão ser utilizados na avaliação de carreira de engenheiros. Os líderes de equipes de inspeção devem ser cuidadosamente treinados para gerenciar o processo e para desenvolver uma cultura que forneça apoio quando forem constatados erros e na qual nunca exista a noção de culpa associada a esses erros.

19.3 Análise estática automatizada

Os analisadores estáticos de programa são ferramentas de software que analisam o código-fonte de um programa e detectam possíveis defeitos e anomalias. Eles não requerem que o programa seja executado. Em vez disso, percorrem o texto do programa e reconhecem os diferentes tipos de declarações. Eles podem, então, detectar se as declarações são bem elaboradas ou não, fazer inferências sobre o fluxo de controle do programa e, em muitos casos, calcular o conjunto de todos os possíveis valores para os dados do programa. Eles complementam os recursos de detecção de erros fornecidos pelo compilador de linguagem.

A intenção da análise estática automatizada é chamar a atenção do verificador para anomalias no programa, tais como variáveis que são utilizadas sem iniciação, variáveis que não são utilizadas, dados cujos valores podem exceder os limites etc. Algumas das checagens que podem ser detectadas pela análise estática são mostradas na Figura 19.8. Embora essas condições não sejam necessariamente errôneas, acontece de muitas dessas anomalias serem o resultado de erros de programação e de omissões. A análise estática automatizada é mais bem utilizada com as inspeções de software. Elas fornecem mais informações para a equipe de inspeção.

Os estágios envolvidos na análise estática incluem:

1. *Análise do fluxo de controle* Esse estágio identifica e destaca loops com múltiplos pontos de saída ou de entrada e código inacessível. O código inacessível é o código cercado por declarações incondicionais 'goto' ou que está em uma ramificação de declaração condicional, em que a condição de guarda nunca pode ser verdadeira.

2. *Análise da utilização de dados* Esse estágio ressalta como as variáveis no programa são utilizadas. Ele detecta variáveis que são utilizadas sem prévia iniciação, variáveis que são escritas duas vezes sem uma tarefa de impedimento e variáveis que são declaradas,

Figura 19.8 ▶

Checagem da análise estática automatizada.

Classe de defeitos	Checagem da análise estática
Defeitos em dados	Variáveis utilizadas antes da inicialização Variáveis declaradas, mas nunca utilizadas Variáveis atribuídas duas vezes, mas nunca utilizadas entre atribuições Possíveis violações de limites de vetor Variáveis não declaradas
Defeitos de controle	Código inacessível Ramificações incondicionais em loops
Defeitos de entrada/saída	Variáveis de saída sem tarefa de impedimento
Defeitos de interface	Desacordo quanto ao tipo de parâmetro Desacordo quanto ao número de parâmetros Não-utilização dos resultados de funções Funções e procedimentos não chamados
Defeitos de gerenciamento de armazenamento	Ponteiros não designados Aritmética de ponteiro

mas nunca são utilizadas. A análise do uso de dados também descobre testes ineficientes, em que a condição de testes é redundante. As condições redundantes são aquelas cujo valor nunca muda – são sempre verdadeiras ou sempre falsas.

3. *Análise de interface* Essa análise verifica a consistência das declarações de rotinas e procedimentos e seu uso. Ela é desnecessária se uma linguagem claramente tipificada, como Java, for utilizada para a implementação, uma vez que o compilador realiza essas checagens. A análise de interface pode detectar erros de tipo em linguagens inadequadamente tipificadas, como FORTRAN e C. A análise de interface pode também detectar funções e procedimentos que são declarados e nunca chamados, ou resultados de funções que nunca são utilizados.

4. *Análise do fluxo de informações* Essa fase da análise identifica as dependências entre as variáveis de entrada e as variáveis de saída. Embora ela não detecte anomalias, a derivação dos valores utilizados no programa é explicitamente relacionada. As derivações errôneas devem, portanto, ser mais fáceis de ser detectadas durante uma inspeção de código ou revisão. A análise de fluxo de informações pode também mostrar as condições que afetam um valor da variável.

5. *Análise de caminho* Essa fase da análise semântica identifica todos os caminhos possíveis no programa e exibe as declarações executadas nesse caminho. Essencialmente, ela elucida o controle do programa e permite que cada possível predicado seja analisado individualmente.

A análise do fluxo de informações e a análise de caminho geram uma imensa quantidade de informações. Essas informações não destacam condições anômalas, mas simplesmente apresentam o programa sob um ponto de vista diferente. Por causa da grande quantidade de informações gerada, essas fases da análise estática são, algumas vezes, deixadas de fora do processo. Somente as fases iniciais, que detectam condições anômalas diretamente, são utilizadas.

Os analisadores estáticos são particularmente valiosos quando uma linguagem de programação, como C, é utilizada. A linguagem C não tem regras estritas de tipo e a checagem que o compilador C pode fazer é limitada. Portanto, existe uma grande abrangência para os erros de programadores que podem ser automaticamente descobertos pela ferramenta de análise. Isso é particularmente importante quando C (e, em menor extensão, C++) é utilizada para o

Figura 19.9 ▶

Análise estática do LINT.

```
138% more lint_ex.c
#include <stdio.h>
printarray (Anarray)
  int Anarray;
{
  printf("%d",Anarray) ;
}
main ( )
{
  int Anarray[5] ; int i ; char c ;
  printarray (Anarray, i, c) ;
  printarray (Anarray) ;
}
139% cc lint_ex.c
140% lint lint_ex.c
lint_ex.c(10): warning: c may be used before set
lint_ex.c(10): warning: i may be used before set
printarray: variable # of args. lint_ex.c(4) :: lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) :: lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) :: lint_ex.c(11)
printf returns value which is always ignored
```

desenvolvimento de sistemas críticos. Nesse caso, a análise estática pode reduzir significativamente os custos de teste.

Os sistemas Unix e Linux incluem um analisador estático chamado LINT, para programas C. O LINT oferece checagem estática equivalente à verificação proporcionada pelo compilador em uma linguagem claramente tipificada, como Java. Um exemplo de saída produzida por LINT é mostrado na Figura 19.9. Nessa transcrição de uma sessão de terminal Unix, os comandos são mostrados em itálico. O primeiro comando apresenta o (despropositado) programa. Ele define uma função com um parâmetro chamado printarray e, então, chama essa função com três parâmetros. As variáveis i e c são declaradas, mas não lhes são atribuídos valores. O valor retornado pela função nunca é utilizado.

A linha numerada 139 mostra a compilação C deste programa com nenhum erro relatado pelo compilador C e, em seguida, ocorre uma chamada do analisador estático LINT, que detecta e relata os erros de programa.

O analisador estático mostra que as variáveis escalares c e i foram utilizadas, mas não iniciadas, e que printarray foi chamada com um número de argumentos diferente do declarado. Ele identifica também o uso inconsistente do primeiro argumento em printarray e o fato de que o valor da função nunca é utilizado.

A análise com base em ferramentas não pode substituir as inspeções, uma vez que existem alguns tipos de erros que os analisadores estáticos não podem detectar. Por exemplo, eles podem detectar variáveis não iniciadas, mas não podem detectar as iniciações que estiverem incorretas. Em linguagens inadequadamente tipificadas, como C, os analisadores estáticos podem detectar funções que tenham os números e os tipos de argumentos errados, mas não podem detectar situações em que um argumento incorreto do tipo correto tenha sido fornecido para uma função.

Não há dúvida de que, para linguagens como C, a análise estática é uma técnica eficaz para descobrir erros de programa. Contudo, com as modernas linguagens de programação, como Java, os projetistas de linguagem removeram algumas características de linguagem sujeitas a erros. Todas as variáveis precisam ser inicializadas, não existem declarações 'goto', e assim é menos provável que códigos inacessíveis sejam criados acidentalmente e o gerenciamento de armazenamento seja automático. Essa abordagem para evitar erros, em vez da detecção de erros, é mais eficaz para melhorar a confiabilidade de um programa. Portanto, a análise estática automática pode não ser eficaz em termos de custos para os programas em Java.

19.4 Desenvolvimento de software Cleanroom

O desenvolvimento de software Cleanroom (Mills *et al.*, 1987; Cobb e Mills, 1990; Linger, 1994; Prowell *et al.*, 1999) é uma filosofia de desenvolvimento de software que tem como base evitar defeitos de software pelo uso de um rigoroso processo de inspeção. O objetivo dessa abordagem do desenvolvimento de software é conseguir um software sem nenhum defeito. O nome *Cleanroom* foi derivado da analogia com as unidades de fabricação de semicondutores. Nessas unidades (*cleanrooms*), os defeitos são evitados pela fabricação em uma atmosfera ultralimpa. Esse tópico é discutido neste capítulo porque esse método substituiu o teste de unidade de componentes de sistema por inspeções para verificar a consistência desses componentes com suas especificações.

Um modelo do processo Cleanroom, adaptado da descrição dada por Linger (1994), é mostrado na Figura 19.10.

A abordagem Cleanroom para o desenvolvimento de software baseia-se em cinco características principais:

1. *Especificação formal* O software a ser desenvolvido é formalmente especificado. Um modelo de transição de estado, que mostra as respostas do sistema a estímulos, é utilizado para expressar a especificação.
2. *Desenvolvimento incremental* O software é decomposto em incrementos, que são desenvolvidos e validados separadamente, utilizando-se o processo Cleanroom. Esses incrementos são especificados, com entrada do cliente, em um estágio inicial do processo.
3. *Programação estruturada* É utilizado somente um número limitado de construções abstratas de controle e de dados. O processo de desenvolvimento do programa é um processo de refinamento da especificação, feito passo a passo. Um número limitado de construções é utilizado e a meta é aplicar as transformações com preservação de correção (discutidas no Capítulo 3) à especificação, a fim de criar o código do programa.

Figura 19.10 ▼

Processo de desenvolvimento pelo método Cleanroom.

4. *Verificação estática* O software desenvolvido é verificado estaticamente utilizando-se rigorosas inspeções de software. Não existe nenhum processo de teste de unidade ou de módulo para os componentes de código.

5. *Teste estatístico do sistema* O incremento de software integrado é testado estatisticamente, como será discutido no Capítulo 21, para determinar sua confiabilidade. Esses testes estatísticos baseiam-se em um perfil operacional, que é desenvolvido em paralelo com a especificação do sistema, como mostra a Figura 19.10.

O desenvolvimento incremental, mostrado na Figura 19.11 e discutido logo no início no Capítulo 3, envolve a produção e a entrega do software em incrementos discretos. Um incremento pode ser executado por comandos de usuário e é um sistema útil, limitado, por si. Os usuários reportam seus feedbacks do sistema e propõem mudanças que sejam requeridas. O desenvolvimento incremental é importante porque minimiza o transtorno causado no processo de desenvolvimento pelas mudanças de requisitos requeridas pelo cliente.

Quando uma especificação é definida como uma unidade única, as mudanças nos requisitos solicitadas pelo cliente (que são inevitáveis) afetam a especificação e o processo de desenvolvimento. A especificação e o projeto precisam ser continuamente refeitos. Com o desenvolvimento incremental, a especificação para o incremento é congelada, embora os pedidos de modificações para o restante do sistema sejam aceitos. O incremento do software é entregue em sua conclusão.

A abordagem de desenvolvimento incremental no processo Cleanroom consiste em fornecer a importante funcionalidade para o cliente nos incrementos iniciais. As funções de sistema menos importantes são incluídas em incrementos posteriores. Portanto, o cliente tem a oportunidade de experimentar esses principais incrementos antes que todo o sistema seja entregue. Se forem descobertos problemas nos requisitos, o cliente fornece essas informações como feedback para a equipe de desenvolvimento e solicita nova versão do incremento.

Isso significa que as funções mais importantes para o cliente recebem a maior validação. À medida que novos incrementos são desenvolvidos, estes são combinados com os incrementos já existentes, e o sistema integrado é testado. Portanto, os incrementos existentes são testados novamente com novos casos de teste sempre que novos incrementos do sistema são adicionados.

O processo Cleanroom é projetado para aceitar uma rigorosa inspeção do programa. Um modelo de sistema baseado em estados é produzido para servir como uma especificação de sistema, e esse modelo é refinado por meio de uma série de diferentes modelos de sistema para um programa executável. A abordagem utilizada para o desenvolvimento tem como base transformações bem definidas, que tentam preservar a exatidão em cada transformação, para uma representação mais detalhada. Essa abordagem geral do desenvolvimento foi discutida no Capítulo 3. As inspeções de programa são suplementadas por argumentos matemáticos rigorosos, que demonstram que a saída da transformação é consistente com sua entrada.

Os argumentos matemáticos utilizados no processo Cleanroom são mais ineficientes do que as provas matemáticas formais. As provas matemáticas formais de que um programa

Figura 19.11 ▶

Processo de desenvolvimento incremental.

está correto com respeito a sua especificação são muito dispendiosas de se desenvolver. Elas dependem do uso do conhecimento da semântica formal da linguagem de programação para construir teorias que relacionem o programa com sua especificação formal. Essas teorias precisam, então, ser provadas matematicamente, muitas vezes com a assistência de programas grandes e complexos para provar teoremas. Em razão de seu alto custo e das habilidades especializadas necessárias, as provas somente são realizadas para aplicações muito especiais de segurança e de proteção fundamentais.

Existem três equipes envolvidas, quando o processo Cleanroom é utilizado para o desenvolvimento de grandes sistemas:

1. *Equipe de especificação* Essa equipe é responsável pelo desenvolvimento e pela manutenção da especificação do sistema. As especificações orientadas ao cliente (definição de requisitos) e as especificações matemáticas para verificação são produzidas por essa equipe. Em alguns casos, quando a especificação está completa, a equipe de especificação assumirá também a responsabilidade pelo desenvolvimento.

2. *Equipe de desenvolvimento* Essa equipe tem a responsabilidade de desenvolver e verificar o software. O software não é executado durante o processo de desenvolvimento. É utilizada uma abordagem estruturada e formal para a verificação, baseada na inspeção de código, suplementada com argumentos relativos à precisão.

3. *Equipe de certificação* Essa equipe é responsável pelo desenvolvimento de um conjunto de testes estatísticos para testar o software, depois que ele foi desenvolvido. Esses testes são baseados em especificação formal. O desenvolvimento de casos de teste é realizado em paralelo com o desenvolvimento do software. Os casos de teste são utilizados para certificar a confiabilidade do software. Os modelos de aumento de confiabilidade, discutidos no Capítulo 21, podem ser utilizados para decidir quando interromper os testes.

O uso da abordagem Cleanroom tem resultado em softwares com muito poucos erros, e ela não parece ser mais dispendiosa do que o desenvolvimento convencional. Cobb e Mills (1990) discutem diversos projetos de desenvolvimento bem-sucedidos com o método Cleanroom, que tiveram uma taxa de falhas uniformemente baixa nos sistemas entregues. Os custos desses projetos foram comparáveis com outros projetos que utilizaram as técnicas de desenvolvimento convencionais.

A verificação estática é eficaz em termos de custos no processo Cleanroom. A grande maioria dos defeitos é descoberta antes da execução, e estes não são introduzidos no software desenvolvido. Linger (1994) relata que, em média, somente 2,3 defeitos por mil linhas de código-fonte foram descobertos durante os testes para projetos Cleanroom. Os custos totais de desenvolvimento não aumentam, porque menos esforço é requerido para testar e reparar o software desenvolvido.

Selby *et al.* (1987), utilizando estudantes como desenvolvedores, comparou o desenvolvimento pelo método Cleanroom com as técnicas convencionais. Eles constataram que a maioria das equipes podia utilizar o método Cleanroom com sucesso. Os programas produzidos tinham maior qualidade do que aqueles produzidos com a utilização de técnicas tradicionais – o código-fonte tinha mais comentários e uma estrutura mais simples. A maioria das equipes que utilizaram o método Cleanroom cumpriu os prazos programados.

O desenvolvimento Cleanroom funciona quando praticado por engenheiros habilitados e empenhados. Contudo, o uso desse método tem sido restrito a organizações tecnologicamente avançadas, e sua adoção tem sido relativamente pequena. Os relatos de sucesso da abordagem Cleanroom na indústria têm vindo, em sua maioria, embora não exclusivamente, de seus desenvolvedores. Não sabemos se esse processo pode ser transferido para outros tipos de organização de desenvolvimento de software. Essas organizações, geralmente, têm um número menor de engenheiros, e estes são menos comprometidos e menos habilitados. Transferir a abordagem Cleanroom para essas organizações ainda permanece um desafio.

PONTOS-CHAVE

- A verificação e a validação não são a mesma coisa. A verificação destina-se a mostrar que um programa cumpre com sua especificação. A validação destina-se a demonstrar que o programa faz o que o usuário requer.
- Os planos de teste devem incluir uma descrição dos itens a serem testados, o cronograma dos testes, os procedimentos para gerenciar o processo de testes, os requisitos de hardware e software, e quaisquer problemas que possam ocorrer nos testes.
- As técnicas de verificação estática envolvem o exame e a análise do código-fonte do programa, a fim de detectar erros. Elas devem ser utilizadas com os testes de programa, como parte do processo de V & V.
- As inspeções de programa são eficazes em identificar erros de programa. O objetivo de uma inspeção é localizar defeitos. O processo de inspeção deve ser dirigido por uma checklist de defeitos.
- O código do programa é sistematicamente verificado por uma pequena equipe. Os membros da equipe incluem um líder ou moderador, o autor do código, um leitor, que apresenta o código durante a inspeção, e um testador, que considera o código a partir de uma perspectiva de testes.
- Os analisadores estáticos são ferramentas de software que processam o código-fonte do programa e chamam a atenção para anomalias, como seções de código não utilizadas e variáveis não inicializadas. Essas anomalias podem ser o resultado de defeitos no código.
- O desenvolvimento de software Cleanroom é uma abordagem do desenvolvimento de software que se apóia em técnicas estáticas para a verificação do programa e testes estatísticos para a certificação da confiabilidade do sistema. Ele tem sido bem-sucedido na produção de sistemas com alto nível de confiabilidade.

LEITURAS SUGERIDAS

Cleanroom software engineering: technology and process. Um novo livro sobre a abordagem Cleanroom, que tem seções sobre os fundamentos da técnica e do processo e um estudo prático de caso. (S.J. Prowell, C.J. Trammell, R.C. Linger e J.H. Poore, 1999, Addison Wesley Longman.)

Software inspection. Um livro bastante completo, que aborda em detalhes as inspeções de programa como uma técnica de detecção de defeitos. Ele tende a ser, às vezes, um tanto exagerado no uso de palavras, mas os estudos de caso são particularmente úteis para ilustrar a prática da inspeção. (T. Gilb e D. Graham, 1993, Addison Wesley.)

"Using inspections to investigate program correctness". Esse artigo discute como um processo de inspeção, com base em matemática, mas não inteiramente formal, mostra-se eficaz em descobrir erros de programa. (R.N. Britcher, *IEEE Computer*, 21(11), nov. 1988.)

EXERCÍCIOS

19.1 Discuta as diferenças entre verificação e validação e explique por que a validação é um processo particularmente difícil.

19.2 Explique por que não é necessário que um programa seja inteiramente livre de defeitos antes de ser entregue ao cliente. Em que extensão os testes podem ser utilizados para validar que o programa é adequado a seu propósito?

19.3 Explique por que as inspeções de programa são uma técnica eficaz para detectar erros em um programa. Que tipos de erros podem ser descobertos por meio das inspeções?

19.4 Explique por que uma organização com uma cultura competitiva e elitista provavelmente acharia difícil introduzir inspeções de programa como uma técnica de V & V.

19.5 Utilizando seu conhecimento de Java, C++, C ou alguma outra linguagem de programação, extraia uma checklist de erros comuns (não erros de sintaxe), que não podem ser detectados por um compilador, mas podem ser detectados em uma inspeção de programa.

19.6 Produza uma lista de condições que podem ser detectadas por um analisador estático para Java, Ada ou C++. Comente essa lista em comparação com a lista mostrada na Figura 19.7.

19.7 Leia os artigos publicados sobre o desenvolvimento pelo método *Cleanroom* e escreva um relatório de gerenciamento ressaltando as vantagens, os custos e os riscos de adotar essa abordagem para o desenvolvimento de software.

19.8 Um gerente decide utilizar os relatórios de inspeções de programa como uma entrada para o processo de avaliação de pessoal. Esses relatórios mostram quem cometeu e quem descobriu os erros do programa. Esse seria um comportamento ético por parte do gerente? Seria ético que a equipe fosse informada previamente de que isso aconteceria? Que diferença isso poderia fazer para o processo de inspeção?

19.9 Uma abordagem comumente adotada para os testes de sistema é testar o sistema até terminar o orçamento destinado aos testes e, então, entregá-lo aos clientes. Discuta a ética dessa abordagem.

20 Testes de software

Objetivos

O objetivo deste capítulo é introduzir as técnicas que podem ser utilizadas para testar programas, a fim de descobrir defeitos nesses programas. Depois de ler este capítulo, você poderá:

- compreender uma série de técnicas de testes, que são utilizadas para descobrir defeitos em programas;
- conhecer as diretrizes que apóiam os testes de interfaces de componentes;
- compreender abordagens específicas dos testes de componentes e dos testes de integração para sistemas orientados a objetos, e
- compreender os princípios da operação de apoio de ferramentas CASE para testes.

Sumário

20.1 Testes para a detecção de defeitos
20.2 Testes de integração
20.3 Testes orientados a objetos
20.4 *Workbenches* de testes

No Capítulo 3, foi discutido um processo geral de testes que começava com o teste de unidades individuais de programas, como funções ou objetos. Esse processo era integrado em subsistemas e sistemas e as interações dessas unidades eram testadas. Finalmente, depois de completar o sistema, o cliente pode realizar uma série de testes de aceitação para verificar se o sistema é executado conforme o especificado.

Uma visão mais abstrata desse processo de testes é mostrada na Figura 20.1. A fase de teste de componentes se ocupa de testar o funcionamento de componentes claramente identificáveis. Esses componentes podem ser funções ou grupos de métodos coletados juntos em um módulo ou em objetos. Durante os testes de integração, esses componentes são integrados para formar subsistemas ou o sistema completo. Nessa fase, os testes devem focalizar as interações entre os componentes e a funcionalidade e o desempenho do sistema como um todo. Inevitavelmente, contudo, defeitos em componentes, que tenham sido omitidos durante os primeiros testes, são descobertos durante os testes de integração.

Como parte do processo de planejamento de V & V (verificação e validação), os gerentes precisam tomar decisões sobre quem deve ser responsável pelos diferentes estágios de teste. Para a maioria dos sistemas, os programadores assumem a responsabilidade pelo teste de seu próprio código (módulos ou objetos). Uma vez concluído, o trabalho é passado para uma equipe de integração, que integra os módulos de diferentes desenvolvedores, constrói o software e testa o sistema como um todo. Contudo, no caso de sistemas críticos, um processo mais formal pode ser utilizado, quando testadores independentes são responsáveis por todas as fases do processo de testes. Os testes são desenvolvidos separadamente e são mantidos registros detalhados.

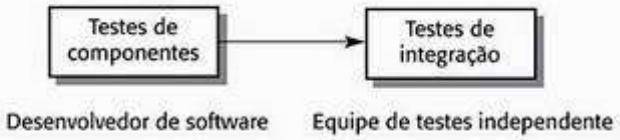
Ao testar sistemas críticos, uma especificação detalhada de cada componente de software é utilizada pela equipe independente, a fim de derivar os testes para o sistema. Contudo, em muitos outros casos, os testes constituem um processo mais intuitivo, uma vez que não há tempo para escrever especificações detalhadas de cada uma das partes de um sistema de software. Em vez disso, as interfaces dos principais componentes de sistema são especificadas, e programadores individuais e as equipes de programação assumem a responsabilidade pelo projeto, pelo desenvolvimento e pelos testes desses componentes. Geralmente, os testes têm como base uma compreensão intuitiva de como esses componentes deveriam operar.

Os testes de integração, contudo, devem se basear em uma especificação escrita do sistema, que pode ser uma especificação detalhada dos requisitos de sistema, como foi discutido no Capítulo 5, ou pode ser uma especificação, orientada pelo usuário, das características que devem ser implementadas no sistema. Uma equipe separada é sempre responsável pelos testes de integração. Como foi discutido no Capítulo 3, eles utilizam os documentos de usuário e os documentos de requisitos do sistema para desenvolver planos detalhados dos testes de integração (veja a Figura 3.12).

Livros sobre testes, como o de Beizer (1990), Kit (1995) e Perry (1995), baseiam-se na experiência prática, quando os sistemas são desenvolvidos utilizando-se um modelo funcional. Eles não tratam especificamente de testes de sistemas orientados a objetos. Sob a perspectiva dos testes, os sistemas orientados a objetos diferem dos sistemas orientados a funções, de duas maneiras fundamentais:

1. Nos sistemas orientados a funções, existe uma distinção bastante nítida entre as unidades básicas de programas (funções) e os conjuntos dessas unidades de programas (módulos). Em sistemas orientados a objetos, não há tal distinção. Os objetos podem ser simples entidades, como uma lista, ou entidades complexas, como o objeto estação meteorológica, descrito no Capítulo 12, que inclui uma série de outros objetos.

Figura 20.1 ▶
Fases de testes.



2. Freqüentemente, não existe uma hierarquia de objetos bem definida, como é comum nos sistemas orientados a funções. As estratégias de integração, como a integração top-down e bottom-up, que são discutidas na Seção 20.2, são muitas vezes inadequadas.

Essas diferenças significam que o limite entre os testes de componentes e os testes de integração não é exato, no caso de sistemas orientados a objetos. É uma continuação do processo ininterrupto de desenvolvimento orientado a objetos, quando os objetos são a estrutura básica utilizada em todas as fases do processo de software. Embora muitas técnicas de testes sejam independentes do projeto de sistemas, algumas técnicas foram desenvolvidas especificamente dirigidas para os testes de sistemas orientados a objetos. Serão discutidas as abordagens de testes orientados a objetos na Seção 20.3.

20.1 Testes para a detecção de defeitos

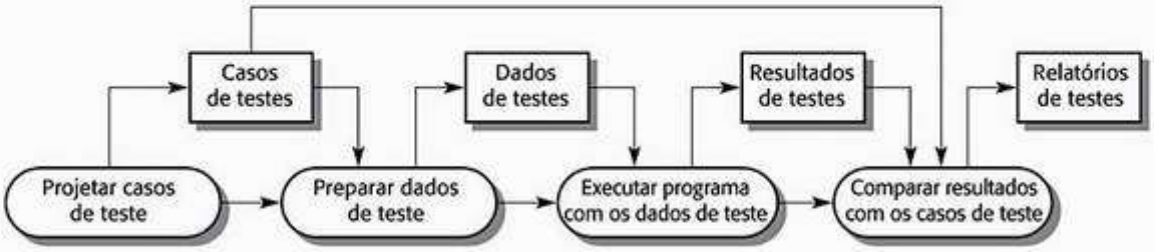
A finalidade dos testes para a detecção de defeitos é expor defeitos latentes em um sistema de software antes de o sistema ser entregue. Isso contrasta com os testes de validação, que se destinam a demonstrar que um sistema cumpre com suas especificações. Os testes de validação exigem que o sistema opere corretamente, utilizando os casos de teste de aceitação que foram apresentados. Um teste bem-sucedido para a detecção de defeitos é aquele que faz com que o sistema opere *incorretamente* e, como conseqüência, expõe um defeito existente. Isso enfatiza um fato importante sobre os testes e demonstra *a presença*, e não a ausência, de defeitos de programa.

Um modelo geral do processo de teste para a detecção de defeitos é mostrado na Figura 20.2. Os casos de testes são especificações das entradas para o teste e da saída esperada do sistema, mais uma declaração do que está sendo testado. Os dados de testes são as entradas que foram criadas para testar o sistema. Os dados de testes podem, algumas vezes, ser gerados automaticamente. A geração automática de casos de teste é impossível, porque a saída de teste não pode ser prevista.

São impraticáveis os testes exaustivos, quando cada seqüência possível de execução do programa é testada. Os testes, portanto, têm de ser baseados em um subconjunto de possíveis casos de testes. As organizações devem desenvolver políticas para escolher esse subconjunto, em vez de deixar essa tarefa por conta da equipe de desenvolvimento. Essas políticas podem ter como base as políticas gerais de testes, como uma política que define que todas as declarações de programa devem ser executadas pelo menos uma vez. Como alternativa, as políticas de testes podem ter como base a experiência de utilização do sistema e podem focalizar o teste nas características do sistema operacional. Por exemplo:

1. Todas as funções de sistema que são acessadas por meio de menus devem ser testadas.
2. As combinações de funções (por exemplo, formatação de texto) que são acessadas pelo mesmo menu devem ser testadas.

Figura 20.2 ▼
Processo de teste para a detecção de defeitos.



3. Quando a entrada do usuário é fornecida, todas as funções devem ser testadas com entradas corretas e incorretas.

Fica claro, a partir da experiência com os principais produtos de software, como processadores de textos ou planilhas de cálculo, que diretrizes comparáveis têm sido utilizadas durante os testes de produtos. Combinações não usuais de funcionalidade podem resultar em erros, mas as funções mais utilizadas, em geral, trabalham corretamente.

20.1.1 Testes de caixa preta

Os testes funcionais, ou testes de caixa preta, são uma abordagem na qual os testes são derivados da especificação de programa ou de componente. O sistema é uma 'caixa preta', cujo comportamento somente pode ser determinado estudando-se suas entradas e saídas relacionadas. A esse método também se dá o nome de *testes funcionais*, porque o testador está preocupado somente com a funcionalidade, e não com a implementação do software.

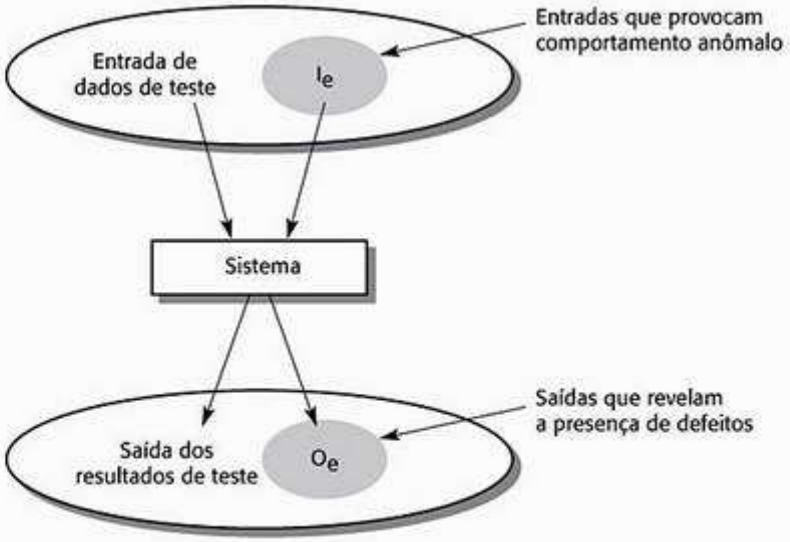
A Figura 20.3 ilustra o modelo de um sistema empregado em testes de caixa preta. Essa abordagem é igualmente aplicável a sistemas que são organizados como funções ou como objetos. O testador apresenta as entradas ao componente ou ao sistema e examina as saídas correspondentes. Se as saídas não são aquelas previstas, então o teste detectou *com sucesso* um problema com o software.

O principal problema para o responsável pelos testes de defeitos é selecionar entradas que tenham grande probabilidade de ser membros do conjunto I_e. Muitas vezes, a seleção desses casos de teste tem como base a experiência prévia dos engenheiros de teste. Eles utilizam o conhecimento de domínio para identificar casos de teste que possam revelar defeitos. Contudo, a abordagem sistemática da seleção de dados de teste, discutida na seção seguinte, pode também ser utilizada para complementar esse conhecimento heurístico.

20.1.2 Partição de equivalência

De modo geral, os dados de entrada para um programa se dividem em uma série de diferentes classes, que têm características comuns, por exemplo, números positivos, números negativos, *strings* sem 'brancos' etc. Os programas normalmente se comportam de maneira comparável para todos os membros de uma classe. Devido a esse comportamento equiva-

Figura 20.3 ▶
Teste de caixa preta.



lente, essas classes são, algumas vezes, chamadas de partições de equivalência ou domínios (Beizer, 1990). Uma abordagem sistemática dos testes para a detecção de defeitos baseia-se em identificar todas as partições de equivalência que tenham de ser manuseadas por um programa. Os casos de teste são projetados de modo que as entradas e as saídas fiquem dentro dessas partições.

Na Figura 20.4, cada partição de equivalência é mostrada como uma elipse. As partições de equivalência de entradas são conjuntos de dados em que todos os membros do conjunto devem ser processados de maneira equivalente. As partições de equivalência de saída são saídas de programa que têm características comuns e, assim, podem ser consideradas uma classe distinta. Também se identificam partições em que as entradas estão fora das outras partições escolhidas. Estas, por sua vez, testam se o programa manuseia corretamente a entrada inválida. As entradas válidas e inválidas também formam partições de equivalência.

Uma vez identificado um conjunto de partições, escolhem-se então casos de teste a partir de cada uma dessas partições. Uma boa diretriz a seguir para a seleção de casos de teste é escolher casos de teste nos limites das partições e também casos próximos ao ponto médio da partição. O ponto em questão para essa diretriz é que os projetistas e os programadores tendem a considerar valores típicos de entradas, quando desenvolvem um sistema. Esses valores são testados escolhendo-se o ponto médio da partição. Os valores-limite são sempre atípicos (por exemplo, o zero pode se comportar diferentemente de outros números que não são negativos) e, assim, ignorados pelos desenvolvedores. Os erros de programa, freqüentemente, ocorrem quando se processam esses valores atípicos.

As partições de equivalência podem ser identificadas utilizando-se a especificação de programa ou a documentação de usuário e pelo testador, utilizando sua própria experiência para prever que classes de valores de entrada podem detectar erros. Por exemplo, digamos que uma especificação de programa declare que ele aceita de 4 a 10 entradas, que são números inteiros de cinco dígitos, maiores do que 10 mil. A Figura 20.5 mostra as partições de equivalência identificadas para essa situação e os possíveis valores de dados dos testes de entrada.

Para ilustrar a derivação dos casos de teste, é utilizada a especificação de um exemplo simplificado (uma rotina de busca), que procura por uma seqüência de elementos para um determinado elemento (*key* – chave). Ela retorna a posição desse elemento na seqüência. A

Figura 20.4 ▶

Partição de equivalência.

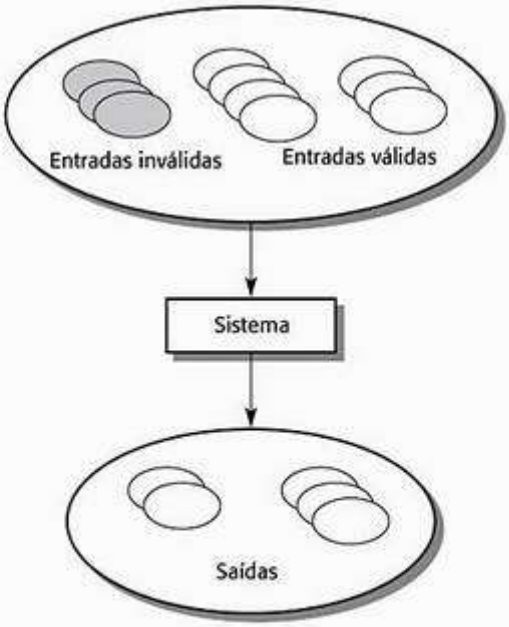
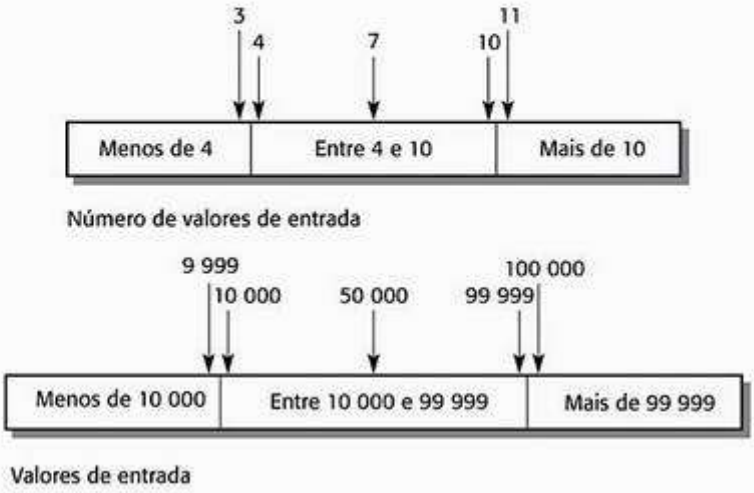


Figura 20.5

Partições de equivalência.



especificação dessa rotina, utilizando pré e pós-condições, é mostrada na Figura 20.6. A precondição declara que a rotina de busca não foi projetada para trabalhar com seqüências vazias. A pós-condição declara que a variável Found será estabelecida, se o elemento-chave estiver na seqüência. A posição do elemento-chave é indicada pelo índice L. O índice será indefinido, se o elemento não estiver na seqüência.

A partir dessa especificação, duas partições de equivalência óbvias podem ser identificadas:

- entradas em que o elemento-chave é um membro da seqüência (Found = true) e
- entradas em que o elemento-chave não é um membro de seqüência (Found = false).

Além de derivar partições de equivalência a partir da especificação de sistema, há várias diretrizes de testes que também podem ser utilizadas. Alguns exemplos dessas diretrizes de testes, que se aplicam a seqüências, são:

1. Teste o software com seqüências que tenham somente um único valor. Os programadores, naturalmente, pensam nas seqüências como constituídas de diversos valores e, algumas vezes, incluem essa suposição em seus programas. Como conseqüência, o programa pode não trabalhar adequadamente quando apresentado com uma seqüência de valor único.
2. Utilize diferentes seqüências, de diferentes tamanhos, em diferentes testes. Isso diminui as chances de que um programa com defeito produza acidentalmente uma saída correta, devido a algumas características acidentais de entrada.

Figura 20.6

Especificação de uma rotina de busca.

```
procedure Search (Key : ELEM ; T: SEQ of ELEM ;
 Found : in out BOOLEAN; L: in out ELEM_INDEX) ;
Pre-condition
  -- a seqüência tem pelo menos um elemento
  T'FIRST <= T'LAST
Post-condition
  -- o elemento é encontrado e é referenciado por L
  ( Found and T (L) = Key)
or
  -- o elemento não está na seqüência
  ( not Found and
  not (exists i, T'FIRST >= i <= T'LAST, T (i) = Key ))
```

3. Derive testes de maneira que o primeiro, o médio e o último elemento da seqüência sejam acessados. Essa abordagem revela problemas nos limites da partição.

Utilizando essas diretrizes, duas outras partições de equivalência do vetor de entrada podem ser identificadas:

- A seqüência de entrada tem um único valor.
- O número de elementos na seqüência de entrada é maior do que 1.

Essas partições devem ser combinadas com as partições de equivalência previamente identificadas, considerando as partições de equivalência resumidas na Figura 20.7.

Um conjunto de possíveis casos de teste baseados nessas partições também é mostrado na Figura 20.7. Se o elemento-chave não estiver na seqüência, o valor de L será indefinido ('??'). A diretriz de que diferentes seqüências de diferentes tamanhos devem ser utilizadas foi aplicada nesses casos de teste.

O conjunto de valores de entrada utilizado para testar a rotina de busca não é exaustivo. A rotina pode falhar, se a seqüência de entrada incluir os elementos 1, 2, 3 e 4. Contudo, é razoável supor que se o teste falhar em detectar defeitos, quando um membro de uma classe for processado, nenhum outro membro daquela classe identificará defeitos. Naturalmente, podem ainda existir defeitos. Algumas partições de equivalência podem não ter sido identificadas, podem ter sido cometidos erros na identificação da partição de equivalência ou os dados de teste podem ter sido preparados incorretamente.

Deliberadamente, deixei de fora os testes projetados para apresentar o sistema com parâmetros na ordem errada, do tipo errado etc. Esse tipo de erro é detectado mais facilmente, utilizando-se a inspeção de programa ou a análise estática automatizada. Da mesma maneira, os testes não verificam a corrupção inesperada de dados fora do componente. Não tem sentido para os testes de caixa preta verificar essa corrupção. A inspeção de código, como foi discutido no Capítulo 19, pode revelar se é provável que esse tipo de problema ocorra.

Figura 20.7 ▶

Partições de equivalência para rotina de busca.

Vetor	Elemento
Valor único	Em seqüência
Valor único	Não em seqüência
Mais que 1 valor	Primeiro elemento na seqüência
Mais que 1 valor	Último elemento na seqüência
Mais que 1 valor	Elemento médio na seqüência
Mais que 1 valor	Não em seqüência

Seqüência de entradas (T)	Chave (Key)	Saídas (Found, L)
17	17	verdadeiro, 1
17	0	falso, ??
17, 29, 21, 23	17	verdadeiro, 1
41, 18, 9, 31, 30, 16, 45	45	verdadeiro, 7
17, 18, 21, 23, 29, 41, 38	23	verdadeiro, 4
21, 23, 29, 33, 38	25	falso, ??

20.1.3 Testes de estrutura

Os testes de estrutura (Figura 20.8) consistem em uma abordagem de testes, em que estes são derivados do conhecimento da estrutura e da implementação do software. Essa abordagem é, às vezes, chamada de testes de 'caixa branca', testes de 'caixa de vidro' ou testes de 'caixa clara', para distingui-los dos testes de caixa preta.

Os testes de estrutura são, em geral, aplicados a unidades de programa relativamente pequenas, como sub-rotinas, ou às operações associadas com um objeto. Como o nome sugere, o testador pode analisar o código e utilizar conhecimento sobre a estrutura de um componente, a fim de derivar os dados para o teste. A análise do código pode ser utilizada para descobrir quantos casos de teste são necessários para garantir que todas as declarações no programa ou componente sejam executadas pelo menos uma vez durante o processo de teste.

O conhecimento do algoritmo utilizado para implementar alguma função pode ser utilizado para identificar outras partições de equivalência. Para ilustrar essa situação, foi considerada a especificação de rotina de busca (Figura 20.6) como uma rotina de busca binária (Figura 20.9). Naturalmente, isso tem pré-condições mais estritas. A seqüência é implementada como um vetor. Esse vetor tem de ser solicitado, e o valor do limite inferior do vetor deve ser menor do que o valor do limite superior.

Examinando o código da rotina de busca, podemos ver que a busca binária envolve dividir o espaço de busca em três partes. Cada uma dessas partes constitui uma partição de equivalência (Figura 20.10). Os casos de teste em que a chave está no limite de cada uma dessas partições devem ser escolhidos para executar o código.

Os casos de teste mostrados na Figura 20.7 devem, portanto, ser modificados, de modo que o vetor de entrada seja organizado em ordem crescente. Outros casos, com base no conhecimento do algoritmo utilizado, devem ser acrescentados ao conjunto de teste. Esses são elementos adjacentes ao ponto médio do vetor. A Figura 20.11 mostra um conjunto de casos de teste para a rotina de busca binária.

20.1.4 Testes de caminho

Os testes de caminho são uma estratégia de teste de estrutura, cujo objetivo é exercitar cada caminho de execução independente, por meio de um componente ou programa. Se cada caminho independente for executado, então todas as declarações no componente devem ter sido executadas pelo menos uma vez. Além do mais, todas as declarações condicionais são testadas para os casos verdadeiros e os falsos. Em um processo de desenvolvimento orientado a objetos, os testes de caminho podem ser utilizados quando são testados métodos associados a objetos.

De modo geral, o número de caminhos em um programa é proporcional a seu tamanho. Como os módulos são integrados em sistemas, torna-se inviável utilizar as técnicas de testes de estrutura. As técnicas de teste de caminho são, por conseguinte, utilizadas principalmente nos estágios de testes de unidade e testes de módulo do processo de teste.

Figura 20.8 ▶
Testes de estrutura.

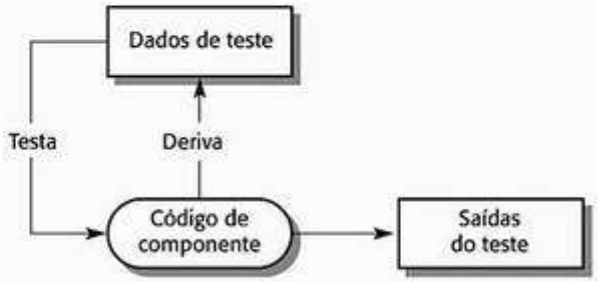


Figura 20.9

Implementação em Java de uma rotina de busca binária.

```
Class BinSeach {
// Esse é um encapsulamento de uma função de busca
// binária que considera um vetor de objetos ordenados e uma chave
// e retorna um objeto com 2 atributos, chamados
// index – o valor do índice de vetor
// found – um booleano que indica se uma chave está ou não no vetor
// Um objeto é retornado pois não é possível em Java passar
// tipos básicos por referência para uma função e
// retornar dois valores
// A chave será –1 se o elemento não for encontrado
    public static void search ( int key, int [ ] elemArray, Result r )
    {
        int bottom = 0 ;
        int top = elemArray.length – 1 ;
        int mid ;
        r.found = false ; r.index = –1 ;
        while ( bottom <= top )
        {
            mid = (top + bottom) / 2 ;
            if (elemArray [mid] = = key)
            {
                r.index = mid ;
                r.found = true ;
                return ;
            } // if part
            else
            {
                if (elemArray [mid] < key)
                    bottom = mid + 1 ;
                else
                    top = mid – 1 ;
            }
        } //while loop
    } // search
} //BinSearch
```

Os testes de caminho não testam todas as possíveis combinações de todos os caminhos no programa. Para quaisquer componentes, à parte aqueles muito triviais, sem loops, esse é um objetivo impossível. Existe um número infinito de possíveis combinações de caminhos em programas com loops. Os defeitos que se manifestam quando surgem combinações particulares de caminhos podem ainda estar presentes, muito embora todas as combinações de programas tenham sido executadas pelo menos uma vez.

O ponto de partida dos testes de caminho é um grafo de fluxo de programa. Esse é um modelo em esqueleto de todos os caminhos do programa. Um diagrama de fluxo consiste em nós que representam decisões e em ramos que mostram o fluxo de controle. O grafo de fluxo é construído substituindo-se declarações de controle de programa por diagramas equivalentes. Se não houver nenhuma declaração 'goto' em um programa, a derivação de seu diagrama de fluxo é um processo simples. As declarações seqüenciais (atribuições, chamadas de procedimento e declarações de E/S) podem ser ignoradas na construção do diagrama de fluxo. Cada ramo em uma declaração condicional (*if-then-else* ou *case*) é mostrado como um caminho separado, e os loops são indicados por uma seta fazendo o loop de volta para o nó de condição de loop. Os loops e os ramos condicionais são ilustrados no grafo de fluxo para a rotina de busca binária (Figura 20.12).

O objetivo dos testes de estrutura é assegurar que cada caminho de programa independente seja executado pelo menos uma vez. Um caminho de programa independente é aquele que atravessa pelo menos um novo ramo no grafo de fluxo. Em termos de programa, isso significa exercitar uma ou mais novas condições. Os ramos verdadeiros e falsos de todas as condições têm de ser executados pelo menos uma vez.

Figura 20.10

Classes de equivalência de busca binária.

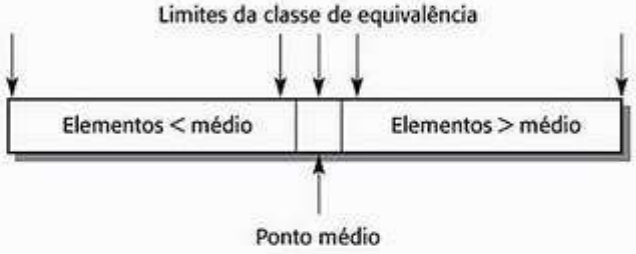


Figura 20.11

Casos de teste para a rotina de busca.

Vetor de entrada (T)	Chave (Key)	Saída (Found, L)
17	17	verdadeiro, 1
17	0	falso, ??
17, 21, 23, 29	17	verdadeiro, 1
9, 16, 18, 30, 31, 41, 45	45	verdadeiro, 7
17, 18, 21, 23, 29, 38, 41	23	verdadeiro, 4
7, 18, 21, 23, 29, 33, 38	21	verdadeiro, 3
12, 18, 21, 23, 32	23	verdadeiro, 4
21, 23, 29, 33, 38	25	falso, ??

O grafo de fluxo para o procedimento de busca binária é mostrado na Figura 20.12. Portanto, acompanhando o fluxo, vemos que os caminhos independentes no grafo de fluxo de busca binária são:

1, 2, 3, 8, 9
1, 2, 3, 4, 6, 7, 2
1, 2, 3, 4, 5, 7, 2
1, 2, 3, 4, 6, 7, 2, 8, 9

Se todos esses caminhos forem executados, poderemos estar certos de que:

1. todas as declarações no método foram executadas pelo menos uma vez e
2. todos os ramos foram exercitados para as condições verdadeiras e falsas.

O número de caminhos independentes em um programa pode ser descoberto calculando-se a complexidade ciclomática (McCabe, 1976) do grafo de fluxo do programa. A complexidade ciclomática, CC, de qualquer grafo G conectado pode ser calculada de acordo com a seguinte fórmula:

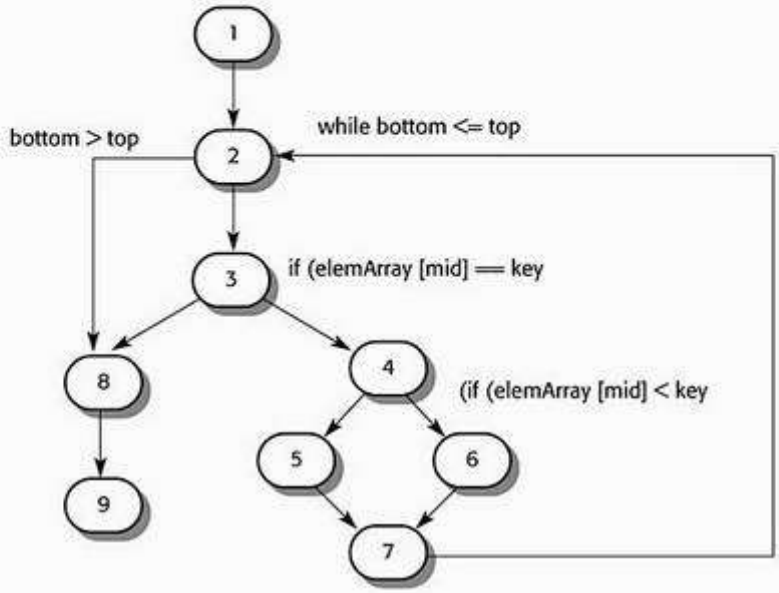
CC (G) = Número (ramos) – Número (nós) + 2

Para os programas sem declarações 'goto', o valor da complexidade ciclomática é um a mais que o número de condições no programa. Em uma condição composta de mais de um teste, deve-se contar cada teste. Assim, se houver seis declarações 'if' e um 'loop while', com todas as expressões condicionais simples, a complexidade ciclomática será 8. Se uma expressão condicional for uma expressão composta de dois operadores lógicos ('and' ou 'or'), a complexidade ciclomática será 10. A complexidade ciclomática da rotina de busca binária (Figura 20.10) será 4.

Depois de descobrir o número de caminhos independentes no código pelo cálculo da complexidade ciclomática, o próximo passo será projetar casos de teste para executar cada um

Figura 20.12 ▶

Grafo de fluxo para uma rotina de busca binária.



desses caminhos. O número mínimo de casos de teste requerido para testar todos os caminhos do programa é igual à complexidade ciclomática.

O projeto do caso de teste é direto, no caso da rotina de busca binária. Contudo, quando os programas têm uma estrutura complexa de ramificação, pode ser difícil prever como qualquer caso de teste em particular será processado. Nesses casos, um analisador dinâmico de programa pode ser utilizado para descobrir o perfil de execução do programa.

Os analisadores de programa dinâmicos são ferramentas de teste que trabalham em conjunto com compiladores. Durante a compilação, instruções adicionais de código-objeto são acrescentadas ao código gerado. Estas contam o número de vezes que cada declaração de programa foi executada. Depois que o programa foi executado, um perfil de execução pode ser impresso, que mostra quais partes do programa foram ou não executadas, utilizando casos de teste particulares. Portanto, esse perfil de execução revela seções do programa que não foram testadas.

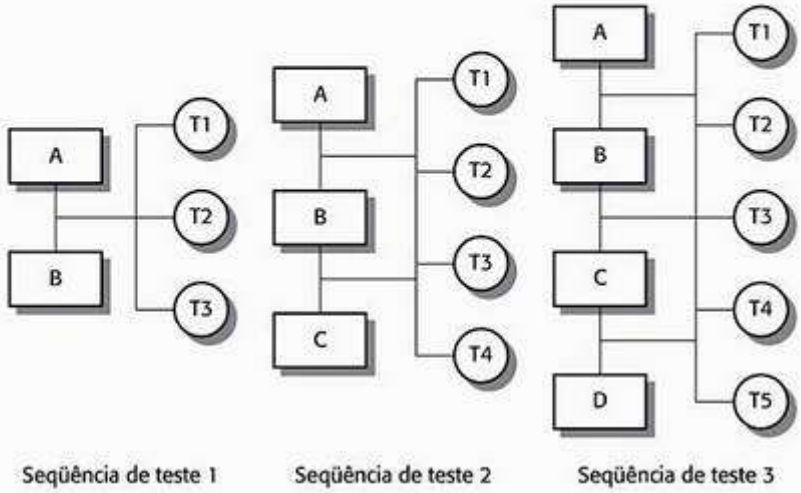
20.2 Testes de integração

Uma vez testados os componentes individuais de programa, eles devem ser integrados para criar um sistema parcial ou completo. Esse processo de integração envolve construir o sistema (veja o Capítulo 29) e testar o sistema resultante quanto a problemas que surjam a partir das interações de componentes. Os testes de integração devem ser desenvolvidos a partir da especificação do sistema e começar assim que as versões de alguns dos componentes do sistema estejam disponíveis.

A principal dificuldade que surge nos testes de integração é localizar erros descobertos durante o processo. Há complexas interações entre os componentes de sistema e, quando uma saída anômala é descoberta, pode ser difícil encontrar a origem do erro. Para facilitar a localização de erros, deve-se sempre utilizar uma abordagem adicional para a integração e os testes do sistema. Inicialmente, é necessário integrar uma configuração mínima do sistema e testar esse sistema. Em seguida, são adicionados componentes a essa configuração mínima, que é testada depois de cada componente adicionado.

Figura 20.13

Testes de integração incremental.



No exemplo mostrado na Figura 20.13, as seqüências de teste T1, T2 e T3 são, primeiramente, executadas em um sistema composto pelos módulo A e o módulo B (o sistema mínimo). Se eles revelarem defeitos, serão corrigidos. O módulo C é integrado, e os testes T1, T2 e T3 são repetidos para garantir que não tenha havido interações inesperadas com A e B. Se surgirem problemas nesses testes, isso significa que eles são devidos a interações com o novo módulo. A origem do problema é localizada, simplificando, assim, a localização do defeito e seu reparo. A seqüência de teste T4 é também executada no sistema. Finalmente, o módulo D é integrado e testado utilizando-se os testes existentes e os novos (T5).

É claro: a realidade raramente é tão simples quanto esse modelo sugere. A implementação de recursos de sistema pode ser distribuída em uma série de componentes. Testar um novo recurso pode, portanto, exigir a integração de diversos componentes diferentes. O teste pode revelar erros nas interações entre esses componentes individuais e em outras partes do sistema. Reparar erros pode ser difícil, porque isso afeta todo o grupo de componentes que implementam o recurso do sistema. Além disso, quando um novo componente é integrado e testado, isso pode modificar o padrão de interações prévias de componentes, já testadas. Podem ser revelados erros que não haviam sido detectados nos testes de configuração mais simples.

20.2.1 Testes top-down e bottom-up

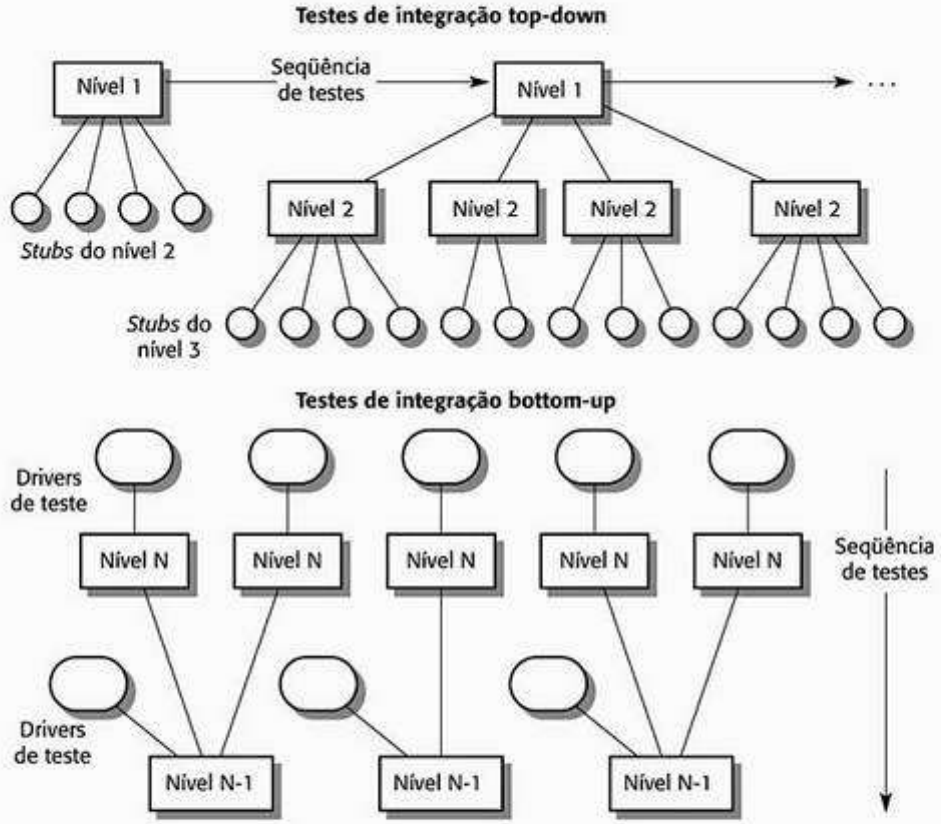
As estratégias de teste top-down e bottom-up (Figura 20.14) refletem abordagens diferentes da integração de sistema. Na integração top-down, os componentes de alto nível de um sistema são integrados e testados antes que seus projetos e sua implementação tenham sido completados. Na integração bottom-up, os componentes de nível inferior são integrados e testados antes que os componentes de nível superior tenham sido desenvolvidos.

Os testes top-down são parte integrante do processo de desenvolvimento top-down, o qual se inicia com os componentes de alto nível e vai descendo na hierarquia de componentes. O programa é representado como um componente abstrato único com subcomponentes representados por *stubs*. Os *stubs* têm a mesma interface que o componente, mas apresentam uma funcionalidade muito limitada. Depois que o componente de nível superior foi programado e testado, seus subcomponentes são implementados e testados da mesma maneira. Esse processo continua até que os componentes de nível inferior sejam implementados. E, então, todo o sistema foi inteiramente testado.

Por outro lado, os testes bottom-up envolvem integrar e testar os módulos de nível inferior na hierarquia e, então, subir na hierarquia de módulos, até que o módulo final seja testado. Essa abordagem não requer que o projeto de arquitetura de sistema esteja completo,

Figura 20.14 ▶

Testes de integração top-down e bottom-up.



de modo que pode começar em um estágio inicial do processo de desenvolvimento. Ele pode ser utilizado quando o sistema reutiliza e modifica componentes de outros programas.

Os testes de integração top-down e bottom-up podem ser comparados de acordo com quatro títulos:

1. *Validação de arquitetura* Os testes top-down oferecem maior probabilidade de descobrir erros na arquitetura de sistema e no projeto de alto nível, em um estágio inicial do processo de desenvolvimento. Como esses erros são geralmente estruturais, a detecção no início significa que eles podem ser corrigidos sem gerar custos exagerados. Com o teste bottom-up, o projeto de alto nível não é validado até estágios avançados do processo.

2. *Demonstração de sistema* Com o desenvolvimento top-down, um sistema de trabalho limitado está disponível em uma fase inicial do desenvolvimento. Isso é um incentivo psicológico importante para aqueles envolvidos no desenvolvimento do sistema, pois demonstra a viabilidade do sistema à gerência. A validação pode começar cedo no processo de teste, quando um sistema demonstrável pode ficar disponível para os usuários. Contudo, se o sistema for construído a partir de componentes reutilizáveis, também pode ser possível produzir algum tipo de demonstração utilizando-se a abordagem bottom-up.

3. *Implementação de teste* Os testes top-down estritos são difíceis de implementar porque os *stubs* de programa que simulam níveis inferiores do sistema precisam ser produzidos. Esses *stubs* de programa podem ser uma versão simplificada do componente requerido ou podem pedir que o testador forneça um valor apropriado ou simule a ação do componente. Quando são utilizados os testes bottom-up, é preciso escrever drivers de teste para exercitar os componentes de nível inferior. Esses drivers de teste simulam o ambiente dos componentes e chamam o componente que deve ser testado.

4. *Observação de teste* Ambos os tipos de testes, o bottom-up e o top-down, podem ter problemas com a observação do teste. Em muitos sistemas, os níveis superiores do sis-

tema, que são implementados primeiramente em um processo top-down, não geram saídas, mas, para testar esses níveis, eles devem ser forçados a fazer isso. O testador tem de criar um ambiente artificial para gerar os resultados de teste. Com os testes bottom-up, pode também ser necessário criar um ambiente artificial (os drivers de teste), de modo que a execução dos componentes de nível inferior possa ser observada.

Na realidade, os sistemas geralmente são desenvolvidos e testados utilizando-se uma mistura das abordagens top-down e bottom-up. Diferentes cronogramas de desenvolvimento para diferentes partes do sistema significam que a equipe de integração e de testes têm de trabalhar com quaisquer componentes que estejam disponíveis. Portanto, uma combinação de *stubs* e drivers de teste deve, inevitavelmente, ser desenvolvida durante o processo de teste de integração.

20.2.2 Testes de interface

Os testes de interface ocorrem quando módulos ou subsistemas são integrados para criar sistemas maiores. Cada módulo ou subsistema tem uma interface definida, que é chamada por outros componentes do programa. O objetivo dos testes de interface é detectar erros que possam ter sido introduzidos no sistema, em razão de erros ou suposições inválidas sobre as interfaces.

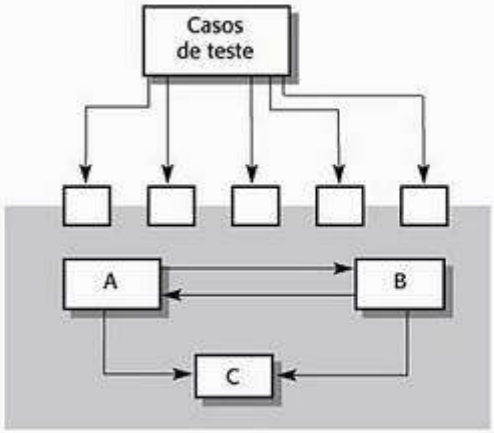
A Figura 20.15 ilustra os testes de interface. As setas para delimitação da caixa significam que os casos de teste não são aplicados aos componentes individuais, mas aos subsistemas criados pela combinação desses componentes.

Essa maneira de testar é, em particular, importante para o desenvolvimento orientado a objetos, principalmente quando os objetos e as classes de objetos são reutilizados. Os objetos são, essencialmente, definidos por suas interfaces e podem ser reutilizados em combinação com diferentes objetos em diferentes sistemas. Os erros de interface não podem ser detectados testando-se os objetos individuais. Os erros são um resultado da interação entre objetos e não do comportamento isolado de um único objeto.

Há diferentes tipos de interfaces entre os componentes de programa e, conseqüentemente, diferentes tipos de erros de interface podem ocorrer:

1. *Interfaces de parâmetros* Essas são interfaces em que dados ou, algumas vezes, referências de funções são passados de um componente para outro.
2. *Interfaces de memória compartilhada* Essas são interfaces em que um bloco de memória é compartilhado entre subsistemas. Os dados são colocados na memória por um subsistema e são recuperados a partir daí por outros subsistemas.
3. *Interfaces de procedimento* Essas são interfaces em que um subsistema encapsula um conjunto de procedimentos que podem ser chamados por outros subsistemas. Os objetos e os tipos abstratos de dados têm essa forma de interface.

Figura 20.15 ▶
Testes de interface.



4. *Interfaces de passagem de mensagem* Essas são interfaces em que um subsistema pede um serviço de outro subsistema, passando-lhe uma mensagem. Uma mensagem de retorno inclui os resultados da execução do serviço. Alguns sistemas orientados a objetos têm esse tipo de interface, como acontece com os sistemas cliente-servidor.

Os erros de interface são um dos mais comuns em sistemas complexos (Lutz, 1993). Eles se dividem em três classes:

- *Mau uso da interface* Um componente chama outro componente e comete um erro no uso de sua interface. Esse tipo de erro é particularmente comum com interfaces de parâmetros, em que os parâmetros podem ser do tipo errado, podem ser passados na ordem errada ou o número errado de parâmetros pode ser passado.
- *Mau entendimento da interface* Um componente que chama interpreta mal a especificação da interface do componente chamado e faz suposições sobre seu comportamento. O componente chamado não se comporta como esperado, e isso ocasiona o comportamento inesperado no componente que chama. Por exemplo, uma rotina de busca binária pode ser chamada com um vetor não ordenado para ser processada. A busca então falhará.
- *Erros de timing* Eles ocorrem em sistemas em tempo real, que utilizam uma memória compartilhada ou uma interface de passagem de mensagens. O produtor e o consumidor de dados podem operar em diferentes velocidades. A menos que cuidados específicos sejam tomados no projeto da interface, o consumidor pode acessar informações desatualizadas, porque o produtor delas não atualizou as informações da interface compartilhada.

Testes para detecção de defeitos de interface são difíceis, porque alguns erros de interface podem apenas se manifestar sob condições incomuns. Por exemplo, digamos que um objeto implementa uma fila como uma estrutura de dados de comprimento fixo. Um objeto que chama pode supor que a fila seja implementada como uma estrutura de dados infinita e pode não verificar o transbordo da fila na entrada de um item. Essa condição somente pode ser detectada durante os testes, projetando-se casos de teste que forcem a fila a transbordar, provocando a corrupção do comportamento do objeto, de maneira detectável.

Um outro problema pode surgir devido às interações entre defeitos em diferentes módulos ou objetos. Os defeitos em um objeto podem ser detectados somente quando algum outro objeto se comporta de maneira inesperada. Por exemplo, um objeto pode chamar outro objeto para receber algum serviço e pode presumir que a resposta esteja correta. Se houver má interpretação sobre o valor calculado, o valor retornado pode ser válido, mas incorreto. Isso somente se manifestará quando algum cálculo posterior der errado.

Algumas diretrizes gerais para os testes de interface são:

1. Examine o código a ser testado e relacione explicitamente cada chamada para um componente externo. Projete um conjunto de testes em que o valor dos parâmetros para os componentes externos esteja nos limites extremos. Esses valores extremos podem revelar inconsistências da interface.
2. Quando ponteiros são passados por meio de uma interface, sempre teste a interface com parâmetros de ponteiros nulos.
3. Quando um componente é chamado por meio de uma interface de procedimento, projete testes que devam provocar a falha do componente. Suposições diferentes de falhas é um dos equívocos mais comuns de especificação.
4. Utilize testes de estresse, como será discutido na seção seguinte, em sistemas de passagem de mensagens. Projete testes que gerem muito mais mensagens do que é provável ocorrer na prática. Problemas de timing podem ser revelados dessa maneira.

5. Quando vários componentes interagem por meio de memória compartilhada, projete testes que variem a ordem em que esses componentes são ativados. Esses testes podem revelar suposições implícitas, feitas pelo programador, sobre a ordem em que os dados compartilhados são produzidos e consumidos.

As técnicas estáticas, freqüentemente, são mais eficazes em termos de custos do que a realização de testes para descobrir erros de interface. Uma linguagem bem tipificada, como Java, permite que muitos erros de interface sejam detectados pelo compilador. Quando uma linguagem menos tipificada, como C, é utilizada, um analisador estático, como o LINT (veja o Capítulo 19), pode detectar erros de interface. Os programas de inspeção podem se concentrar em interfaces de componentes e é possível surgirem questões sobre o suposto comportamento da interface, durante o processo de inspeção.

20.2.3 Testes de estresse

Uma vez que um sistema tenha sido completamente integrado, é possível testá-lo quanto a propriedades emergentes (veja o Capítulo 2), como desempenho e confiabilidade. Os testes de desempenho precisam ser projetados para assegurar que o sistema possa processar sua carga pretendida. Isso envolve planejar uma série de testes em que a carga é constantemente aumentada, até que o desempenho do sistema se torne inaceitável.

Algumas classes de sistema são projetadas para manipular uma carga especificada. Por exemplo, um sistema de processamento de transações pode ser projetado para processar até 100 transações por segundo; um sistema operacional pode ser projetado para lidar com até 200 terminais separados. Os testes de estresse continuam além da carga máxima de projeto do sistema, até que o sistema falhe. Esse tipo de teste tem duas funções:

1. Testa o comportamento de falha do sistema. Podem surgir determinadas circunstâncias por meio de uma combinação inesperada de eventos, quando a carga colocada no sistema excede a carga máxima prevista. Nessas circunstâncias, é importante que a falha do sistema não cause a corrupção de dados ou a perda inesperada de serviços do usuário. Os testes de estresse verificam se, sobrecarregando o sistema, ele 'apresenta uma falha mais leve', em vez de entrar em colapso sob essa carga.
2. Causa estresse no sistema e pode acarretar a detecção de defeitos que, normalmente, não se manifestariam. Embora seja possível argumentar que esses defeitos provavelmente não causariam falhas sob uso normal, podem ocorrer combinações incomuns de circunstâncias normais a que os testes de estresse respondem.

Os testes de estresse são particularmente relevantes para sistemas distribuídos, com base em uma rede de processadores. Esses sistemas freqüentemente exibem uma degradação severa quando são sobrecarregados. A rede torna-se inundada com dados de coordenação, que os diferentes processos devem trocar, e, assim, os processos tornam-se cada vez mais lentos, à medida que eles esperam pelos dados solicitados a partir de outros processos.

20.3 Testes orientados a objetos

Na Figura 20.1, sugeri que havia duas atividades fundamentais no processo de testes. Esses são os testes de componentes, em que os componentes de sistema são testados individualmente, e os testes de integração, em que coleções de componentes são integradas em subsis-

temas e no sistema final para testes. Essas atividades são igualmente aplicáveis a sistemas orientados a objetos. Contudo, existem importantes diferenças entre os sistemas orientados a objetos e os sistemas desenvolvidos utilizando-se um modelo funcional:

1. Objetos como componentes individuais são, muitas vezes, maiores do que funções isoladas.
2. Os objetos que são integrados em subsistemas são, em geral, indevidamente acoplados, e não há um nível superior óbvio para o sistema.
3. Se objetos forem reutilizados, os testadores podem não ter nenhum acesso ao código-fonte do componente para análise.

Essas diferenças significam que as abordagens dos testes de caixa branca, com base na análise de código, devem ser ampliadas para cobrir objetos de maior granularidade e que abordagens alternativas de testes de integração podem ser adotadas. Contudo, uma vez integrado um sistema, o fato de que ele foi desenvolvido como um sistema orientado a objetos não deve ser aparente aos usuários do sistema.

Em um sistema orientado a objetos, quatro níveis de testes podem ser identificados:

1. *Testar as operações individuais associadas com objetos* Essas são funções ou procedimentos, e as abordagens de caixa preta e de caixa branca, discutidas anteriormente, podem ser utilizadas.
2. *Testar classes de objetos individuais* O princípio dos testes de caixa preta permanece inalterável, mas a noção de uma classe equivalente tem de ser estendida para cobrir seqüências de operação relacionadas. Da mesma maneira, os testes de estrutura requerem um tipo diferente de análise. Isso será discutido na Seção 20.3.1.
3. *Testar agrupamentos de objetos* A integração estrita top-down ou bottom-up é inadequada para criar grupos de objetos relacionados. Outras abordagens, tais como testes com base em cenários, devem ser utilizadas. Isso será discutido na Seção 20.3.2.
4. *Testar o sistema orientado a objetos* A verificação e validação em relação à especificação de requisitos de sistema são realizadas exatamente da mesma maneira que para qualquer outro tipo de sistema.

As técnicas de teste para sistemas orientados a objetos amadureceram rapidamente, e atualmente, existe uma boa quantidade de informações disponíveis sobre as técnicas de testes orientadas a objetos (Binder, 1999). As seções seguintes dão uma visão geral de uma abordagem básica para os testes de sistemas orientados a objetos.

20.3.1 Testes de classes de objeto

A abordagem para a cobertura de testes, discutida na Seção 20.1.3, se ocupava de assegurar que todas as declarações em um programa fossem executadas pelo menos uma vez e que todos os caminhos do programa fossem executados. Quando objetos são testados, a cobertura completa de testes deve incluir:

1. o teste isolado de todas as operações associadas com o objeto;
2. o estabelecimento e a interrogação de todos os atributos associados com o objeto, e
3. o exercício do objeto em todos os estados possíveis. Isso significa que todos os eventos que provoquem uma mudança de estado no objeto devem ser simulados.

Considere, por exemplo, a estação meteorológica do Capítulo 12, cuja interface é mostrada na Figura 20.16. Ela tem apenas um atributo, que é seu identificador. Essa é uma cons-

Figura 20.16
Interface do objeto estação meteorológica

tante, estabelecida quando a estação meteorológica é instalada. Portanto, somente é necessário um teste que verifique se ela foi estabelecida. É preciso definir casos de teste para **relatarClima**, **calibrar**, **testar**, **iniciar** e **desativar**. Como ideal, devem ser testados métodos isolados, mas, em alguns casos, algumas seqüências de testes são necessárias. Por exemplo, para testar **desativar**, é necessário ter executado o método **iniciar**.

Para testar os estados da estação meteorológica, utiliza-se um modelo de estado, como foi mostrado na Figura 12.14. Com esse modelo, podem ser identificadas seqüências de transições de estado que precisam ser testadas e podem ser definidas seqüências de eventos para forçar essas transições. Em princípio, é necessário testar cada possível seqüência de transição de estado, embora, na prática, isso possa ser muito dispendioso. Exemplos de seqüências de estado que devem ser testadas na estação meteorológica incluem:

Desativado → *Aguardando* → *Desativado*

Aguardando → *Calibrando* → *Testando* → *Transmitindo* → *Aguardando*

Aguardando → *Coletando* → *Aguardando* → *Resumindo* → *Transmitindo* → *Aguardando*

Se a herança é utilizada, isso torna mais difícil projetar testes de classe de objetos. Quando uma superclasse fornece operações que são herdadas por uma série de subclasses, todas essas subclasses devem ser testadas com todas as operações herdadas. A razão disso é que a operação herdada pode fazer suposições sobre outras operações e outros atributos, e estes podem ter sido modificados quando herdados. Igualmente, quando uma operação de superclasse é carregada, a operação de sobrescrever deve ser testada.

A noção de classes equivalentes pode também ser aplicada a classes de objetos. Os testes classificados nas mesmas classes de equivalência podem ser os que utilizam os mesmos atributos dos objetos. Portanto, as classes de equivalência devem ser identificadas, as quais inicializam, acessam e atualizam todos os atributos da classe de objeto.

20.3.2 Integração de objetos

Quando os sistemas orientados a objetos são desenvolvidos, os níveis de integração são menos distintos. Claramente, operações e dados são integrados para formar objetos e classes de objetos. O teste dessas classes de objetos corresponde ao teste de unidade. Não existe nenhuma equivalência direta com os testes de módulos em sistemas orientados a objetos. Contudo, Murphy *et al.* (1994) sugerem que grupos de classes que agem em combinação para fornecer um conjunto de serviços devem ser testados juntos. Eles chamam a isso de *teste de clusters*.

Nem a integração top-down nem a integração bottom-up são, na realidade, apropriadas para sistemas orientados a objetos. Nesses sistemas, não há um 'limite superior' óbvio que forneça uma meta para a integração, nem existe uma hierarquia definida de objetos que possam ser criados. Os clusters, portanto, devem ser criados utilizando-se o conhecimento de suas operações e as características do sistema implementadas por esses clusters. Existem três possíveis abordagens para os testes de integração, que podem ser utilizadas:

1. *Testes de use-case ou testes baseados em cenários* Os use-cases ou cenários (veja o Capítulo 6) descrevem um modo de uso do sistema. Os testes podem ser baseados nessas descrições de cenário e nos clusters de objetos criados para lidar com os use-cases que se relacionam com aquele modo de uso.

2. *Testes de* threads Os testes de *threads* baseiam-se no teste da resposta do sistema a uma entrada particular ou a um conjunto de eventos de entrada. Os sistemas orientados a objetos são, com freqüência, orientados a eventos; portanto, essa é uma maneira particularmente adequada de realizar testes. Para utilizar essa abordagem, é preciso identificar como o processamento de eventos faz seu caminho no sistema.

3. *Testes de interação de objetos* Uma abordagem relacionada aos grupos de testes de objetos interativos é proposta por Jorgensen e Erickson (1994). Eles sugerem que um nível intermediário de testes de integração pode ter como base a identificação de caminhos de 'método-mensagem' (MM). Essas são trilhas em uma seqüência de interações de objetos, que são interrompidas quando uma operação de objetos não demanda os serviços de nenhum outro objeto. Eles identificam também uma construção relacionada, a que chamam de ASF (*atomic system function* – função atômica de sistema). Uma ASF consiste em algum evento de entrada, seguido por uma seqüência de rotas de MM, que é encerrada por um evento de saída. Isso é similar à noção de *threads* em um sistema de tempo real.

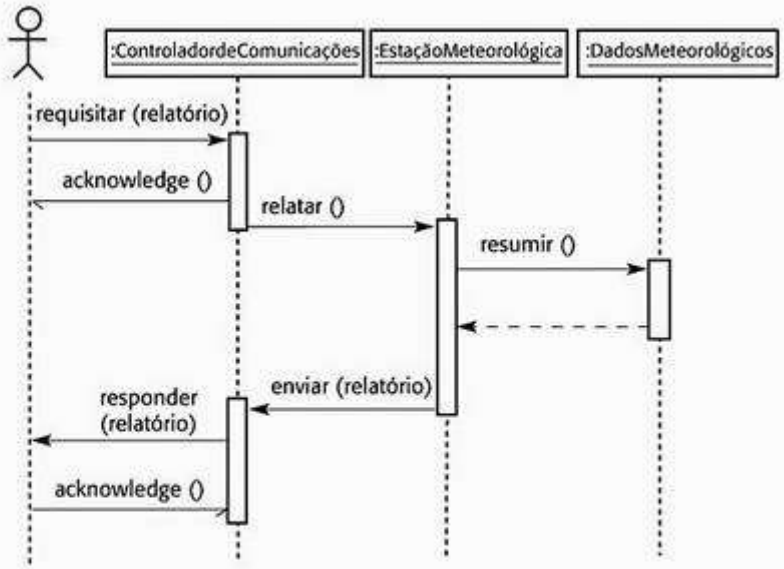
Os testes com base em cenários são, freqüentemente, os mais eficazes entre as abordagens, uma vez que podem ser organizados de maneira que os cenários mais prováveis sejam testados primeiramente e os cenários incomuns ou excepcionais sejam considerados posteriormente no processo de testes. Isso satisfaz um princípio fundamental de teste, ou seja, que a maior parte do esforço de realização de testes deve ser dedicada às partes do sistema que são mais utilizadas.

Para ilustrar os testes com base em cenários, novamente é utilizado o exemplo do sistema da estação meteorológica. Os cenários podem ser identificados a partir dos use-cases que podem ter sido desenvolvidos, mas eles podem não incluir informações suficientes para o propósito de realizar testes. Para complementar esses use-cases no momento do planejamento dos testes, podem ser utilizados os diagramas de interação, que mostram com mais detalhes como um cenário é implementado pelos objetos em um sistema.

Considere a Figura 20.17 (retirada do Capítulo 12), que mostra a seqüência de operações na estação meteorológica quando ela responde a um pedido para coletar dados para o sistema de mapeamento. É possível utilizar esse diagrama para identificar as operações que serão testadas e para ajudar a projetar os casos de teste para executar os testes.

Figura 20.17 ▶

Diagrama de seqüência de coleta de dados climáticos.



Portanto, pode-se ver que emitir um pedido de relatório resultará no seguinte *thread* de métodos a serem executados:

ControladordeComunicações: requisitar → EstaçãoMeteorológica: relatar → DadosMeteorológicos: resumir

Quando se escolhem cenários para imaginar testes de sistema, é importante assegurar que cada método seja executado em cada classe pelo menos uma vez. Portanto, deve-se criar uma checklist de classes e métodos de objeto e, quando um cenário for escolhido, assinalar os métodos que foram executados. Naturalmente, não é possível executar todas as combinações de métodos, mas isso, pelo menos, assegura que todos os métodos individuais foram testados como parte de uma seqüência.

O diagrama de seqüência pode também ser utilizado para identificar entradas e saídas que tenham de ser criadas para o teste:

1. A entrada de um pedido de relatório deve ter uma confirmação associada, e um relatório deve, por último, ser retornado a partir do pedido. Durante o teste, devem ser criados dados resumidos, que possam ser utilizados para checar se o relatório está corretamente organizado.
2. Um pedido de entrada para um relatório para EstaçãoMeteorológica resulta na geração de um relatório resumido. É possível testar isso isoladamente, criando dados brutos correspondentes ao resumo que se preparou para o teste de ControladordeComunicações e verificando se o objeto EstaçãoMeteorológica produz corretamente esse resumo.
3. Esses dados brutos também são utilizados para testar o objeto DadosMeteorológicos.

Naturalmente, o diagrama de seqüência foi simplificado na Figura 20.17, de maneira que não mostrasse as exceções. Um teste completo de cenário deve também levar em conta essas exceções e assegurar que sejam corretamente manuseadas pelos objetos.

20.4 *Workbenches* de testes

Os testes constituem uma fase dispendiosa e trabalhosa do processo de software. Como resultado, as ferramentas de testes foram as primeiras ferramentas de software a serem desenvolvidas. Atualmente, essas ferramentas oferecem uma gama de recursos e seu uso reduz significativamente o custo do processo de testes. Diferentes ferramentas de testes podem ser integradas em bancadas de testes.

A Figura 20.18 mostra algumas ferramentas que podem ser incluídas em *workbenches* de testes e as interações entre essas ferramentas. As ferramentas incluídas são:

1. *Gerenciador de testes* Gerencia a execução dos testes de programa. O gerenciador de testes mantém o acompanhamento dos dados de teste, dos resultados esperados e dos recursos de programa testados.
2. *Gerador de dados de teste* Gera dados de teste para o programa a ser testado. Isso pode ser realizado selecionando-se dados a partir de um banco de dados ou utilizando-se padrões para gerar dados aleatórios de modo correto.
3. *Oráculo* Gera previsões dos resultados esperados para o teste. Os oráculos podem ser versões prévias de programas ou sistemas protótipos. Os testes envolvem executar o oráculo e o programa a ser testado em paralelo. As diferenças em suas saídas são evidenciadas.
4. *Comparador de arquivos* Compara os resultados dos testes de programa com os resultados de testes precedentes e relata as diferenças entre eles. Os comparadores são essen-

ciais nos testes de regressão, quando são comparados os resultados de execução de versões antigas e novas. As diferenças nesses resultados indicam problemas em potencial com a nova versão do sistema.

5. *Gerador de relatórios* Fornece uma definição de relatório e recursos de geração para os resultados de testes.
6. *Analisador dinâmico* Adiciona código a um programa para contar o número de vezes que cada declaração foi executada. Depois que os testes foram executados, é gerado um perfil de execução que mostra a freqüência com que cada declaração de programa foi executada.
7. *Simulador* Diferentes tipos de simuladores podem ser fornecidos. Os simuladores-alvo simulam a máquina em que o programa deve ser executado. Os simuladores de interface com o usuário são programas orientados por scripts, que simulam múltiplas interações simultâneas com o usuário. A utilização de simuladores de E/S significa que o timing de seqüências de transações pode ser repetido.

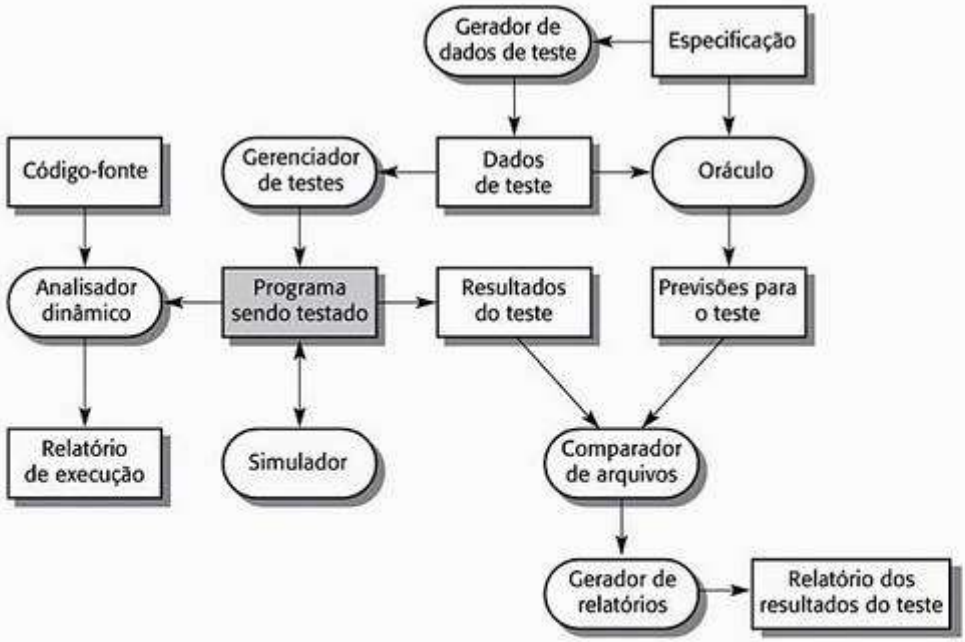
Os requisitos de testes para grandes sistemas dependem da especificação que está sendo desenvolvida. Como conseqüência, os *workbenches* de testes invariavelmente precisam ser adaptadas para se adequarem ao plano de teste de cada sistema. Exemplos de adaptações são:

1. Novas ferramentas podem precisar ser acrescentadas para testar características específicas da aplicação. Algumas ferramentas de teste existentes podem não ser requeridas.
2. Scripts podem ter de ser escritos para simuladores de interface com o usuário e padrões definidos para geradores de dados de teste. Formatos de relatórios também podem precisar ser definidos.
3. Conjuntos de resultados esperados dos testes podem ser preparados manualmente, se nenhuma versão prévia do programa estiver disponível para servir como oráculo.
4. Comparadores de arquivo de propósito especial podem ser escritos, incluindo o conhecimento da estrutura dos resultados de teste em arquivos.

De modo geral, é preciso muito tempo e esforço para criar um *workbench* de testes abrangente. *Workbenches* de teste completos, como mostra a Figura 20.18, são, portanto, utilizadas somente quando grandes sistemas estão sendo desenvolvidos. Para esses sistemas, os custos totais dos testes podem ser de até 50 por cento dos custos totais de desenvolvimento; por isso, é eficaz em termos de custos investir no apoio a ferramentas CASE de alta qualidade para os testes.

Figura 20.18 ▶

Um *workbench* de testes.



PONTOS-CHAVE

- É mais importante testar as partes do sistema comumente utilizadas do que as partes que apenas são exercitadas raramente.
- A partição de equivalência é uma maneira de derivar casos de teste. Ela depende de encontrar partições nos conjuntos de dados de entrada e saída e de executar o programa com valores, a partir dessas partições. Freqüentemente, o valor com maior probabilidade de levar a um teste bem-sucedido é um valor nos limites de uma partição.
- Os testes de caixa preta não necessitam acessar o código-fonte. Os casos de teste são derivados da especificação do programa.
- Os testes de estrutura baseiam-se na análise de um programa para determinar caminhos nele e na utilização dessa análise para ajudar na seleção de casos de teste.
- Os testes de integração devem se concentrar no teste das interações entre os componentes de um sistema e das interfaces com os componentes.
- Os defeitos de interface podem surgir devido a erros cometidos na leitura da especificação, da má compreensão, de erros nas especificações ou de suposições inválidas de timing. Os testes de interface procuram descobrir defeitos nas interfaces ou nos módulos.
- Ao testar as classes de objetos, devem ser projetados testes que executem todas as operações associadas com uma classe, designados e avaliados todos os atributos de objetos e testado o objeto em todos os estados possíveis.
- Os sistemas orientados a objetos devem ser integrados em torno de *clusters* naturais de objetos, como aqueles associados com um use-case particular ou um conjunto de use-cases ou threads de objetos.

LEITURAS SUGERIDAS

Testing object-oriented systems: models, patterns and tools. Esse é um livro imenso (com mais de mil páginas), que fornece uma cobertura completa dos testes orientados a objetos. Sua complexidade significa que esse não deve ser o primeiro livro que você deve ler sobre testes orientados a objetos (a maioria dos livros sobre desenvolvimento orientado a objetos tem um capítulo sobre testes), mas no desenvolvimento orientado a objetos ele é o equivalente ao livro de Beizer. (R.V. Binder, 1999, Addison Wesley Longman.)

"How to design practical test cases". Um artigo do tipo 'como fazer', sobre o projeto de casos de teste, escrito por um autor de uma empresa japonesa, que tem uma reputação muito boa no que se refere a fornecer software com muito poucos defeitos. (T. Yamaura, *IEEE Software*, 15(6), nov. 1998.)

"Regression testing in an industrial environment". Essa publicação discute a importância dos testes de regressão e oferece orientação prática sobre o processo. (A.K. Onoma, W-T Tsai, M. Poonawala e H. Suganuma, *Comm ACM*, 41(5), maio 1998.)

Software testing techniques. Esse é o livro definitivo sobre testes para a detecção de defeitos em programas, que são desenvolvidos utilizando-se uma abordagem orientada a funções. Ele é incrivelmente detalhado e completo. A abordagem de Beizer é prática, e não teórica, e ele dá orientações detalhadas para os testadores de sistemas. (B. Beizer, 1990, Van Nostrand Rheinhold.)

EXERCÍCIOS

20.1 Discuta as diferenças entre os testes de caixa preta e os testes de estrutura e sugira como eles podem ser utilizados em conjunto no processo de teste de defeitos.

20.2 Que problemas referentes a testes podem surgir em rotinas numéricas projetadas para manusear números muito grandes e números muito pequenos?

20.3 Derive um conjunto de casos de teste para os seguintes componentes:

- Uma rotina de classificação que classifica vetores de números inteiros.
- Uma rotina que considera uma linha de texto como entrada e conta o número de caracteres não vazios nessa linha.
- Uma rotina que examina uma linha de texto e substitui seqüências de caracteres vazios por um único caracter vazio.
- Um objeto que implementa *strings* de caracteres de comprimento variável. As operações devem incluir concatenação, comprimento (para dar o comprimento de uma *string*) e seleção de um subconjunto de *strings*.
- Um objeto que representa uma tabela de chaves, em que as entradas são feitas e recuperadas utilizando-se alguma chave alfabética.

20.4 Programe as três primeiras dessas rotinas utilizando uma linguagem de sua escolha e, para cada rotina, calcule sua complexidade ciclomática.

20.5 Mostre, utilizando uma pequena amostra, por que é praticamente impossível testar de maneira exaustiva um programa.

20.6 Examinando o código das rotinas que você escreveu, derive outros casos de teste além daqueles que você já considerou. A análise de código revelou omissões em seu conjunto inicial de casos de teste?

20.7 Implemente (utilizando Java ou C++) uma classe de objetos chamada SYMBOL_TABLE, que poderia ser utilizada como parte de um sistema de compilação. Inclua operações para adicionar um nome e informações de tipo associadas à tabela, para excluir um nome, modificar a informação associada a um nome e realizar buscas na tabela. Mantenha informações sobre onde o objeto é declarado. Organize uma inspeção de programa desse objeto (veja o Capítulo 19) e mantenha uma contagem cuidadosa dos erros descobertos. Teste o objeto utilizando a abordagem de caixa preta e compare os erros que forem revelados pelos testes com aqueles descobertos pela inspeção.

20.8 Explique por que os testes de interface são necessários, considerando que as unidades individuais foram extensivamente validadas pelos testes de unidade e pelas inspeções de programas.

20.9 A respeito das estratégias de testes top-down e bottom-up, discuta por que essas estratégias podem ser inadequadas para os sistemas orientados a objetos.

20.10 Derive casos de teste para testar os estados do forno de microondas, cujo modelo de estado é definido na Figura 7.5.

21 Validação de sistemas críticos

Objetivos

O objetivo deste capítulo é discutir as técnicas de verificação e validação que são utilizadas no desenvolvimento de sistemas críticos. Depois de ler este capítulo, você poderá:

- ficar ciente dos argumentos favoráveis e contrários ao uso de métodos formais na verificação de sistemas críticos;
- compreender como a confiabilidade de um sistema de software pode ser medida e como os modelos de aumento de confiabilidade são utilizados para prever a confiabilidade;
- compreender como a garantia de segurança depende do sistema de validação e da garantia de processo, e
- compreender a idéia de provas de segurança e como elas podem ser utilizadas para demonstrar que perigos não podem surgir em um sistema.

Sumário

21.1 Métodos formais e sistemas críticos
21.2 Validação de confiabilidade
21.3 Garantia de segurança
21.4 Avaliação de proteção

A verificação e validação de um sistema crítico têm, obviamente, muito em comum com a validação de qualquer outro sistema. Os processos de V & V devem mostrar que o sistema cumpre com sua especificação e que os serviços e o comportamento do sistema são compatíveis com os requisitos do cliente. Contudo, a natureza dos sistemas críticos é tal que geralmente é necessário aumentar a análise e os testes normais com processos adicionais, que são projetados para produzir evidências de que o sistema merece confiança. Há duas razões pelas quais isso é necessário:

1. *Custos das falhas* Os custos e as conseqüências das falhas de sistemas críticos são potencialmente muito maiores do que os de sistemas que não são críticos. Conseqüentemente, em relação aos custos, é melhor gastar mais em verificação e validação, de modo que os erros possam ser detectados e removidos antes de o sistema ser entregue.
2. *Validação dos atributos de confiança* Os clientes dos sistemas críticos precisam ser convencidos de que os atributos de confiança especificados (disponibilidade, confiabilidade, segurança e proteção) foram cumpridos pelo sistema. A avaliação desses atributos requer atividades específicas de verificação e validação, que serão discutidas posteriormente neste capítulo.

Por essas razões, os custos de V & V para os sistemas críticos são, geralmente, muito mais altos do que para outras classes de sistemas. Não é incomum que a verificação e validação consumam mais de 50 por cento do total dos custos de desenvolvimento para sistemas de software críticos. Naturalmente, esse custo será justificável, se uma dispendiosa falha de sistema for evitada. Por exemplo, em 1996 um sistema de software crítico de missão, no foguete Ariane 5, falhou e diversos satélites foram destruídos. A conseqüente perda foi de centenas de milhões de dólares.

Embora o processo de validação de sistemas críticos deva principalmente enfocar a validação do sistema, existem atividades relacionadas que verificam se processos definidos de desenvolvimento de sistema foram utilizados. Como discutido nos capítulos 18 e 24, a qualidade do sistema é afetada pela qualidade dos processos utilizados para desenvolvê-lo. Em suma, bons processos levam a bons sistemas. Portanto, para produzir sistemas que tenham altos requisitos de confiança, é preciso ter certeza de que foi empregado um bom processo de desenvolvimento.

Essa garantia de processo é uma parte inerente aos padrões da ISO 9000 para o gerenciamento de qualidade, que será discutido brevemente no Capítulo 24. Esses padrões requerem a documentação dos processos utilizados e das atividades associadas, que assegure que esses processos foram seguidos. Isso, normalmente, requer a geração de evidências tangíveis, como formulários assinados, que certifiquem o término de atividades de processo e as verificações da qualidade do produto. Os padrões ISO 9000 especificam quais saídas tangíveis de processo devem ser produzidas e quem é o responsável por produzir e checar esses formulários. Na Seção 21.3.3, ilustro isso utilizando um formulário-padrão que registra o processo de análise de perigo.

21.1 Métodos formais e sistemas críticos

Existe um debate contínuo na comunidade de sistemas críticos sobre o papel dos métodos formais no processo de desenvolvimento de software crítico de segurança e de proteção. O uso de especificação matemática formal e da verificação associada é obrigatório nos padrões de defesa do Reino Unido para software crítico de segurança (MOD, 1995). Contudo, muitos desenvolvedores de sistemas críticos não estão convencidos de que os métodos formais sejam eficazes em termos de custos e argumentam que eles podem até mesmo reduzir, em vez de aumentar, a confiança do sistema.

Os métodos formais podem ser utilizados em dois níveis, no desenvolvimento de sistemas críticos:

1. Uma especificação formal do sistema pode ser desenvolvida e matematicamente analisada quanto a inconsistências. Essa técnica é eficaz para descobrir erros e omissões de especificação, como foi discutido no Capítulo 9. Nesse capítulo, ilustrei o uso de uma técnica formal para especificar parte do software da bomba de insulina, descrita no Capítulo 16.
2. Pode ser desenvolvida uma verificação formal de que o código de um sistema de software é consistente com a especificação. Isso requer uma especificação formal e é eficaz em descobrir erros de programação e alguns erros de projeto. Um processo de desenvolvimento transformacional (Capítulo 3) ou um processo Cleanroom (Capítulo 19) pode ser utilizado para apoiar o processo de verificação formal.

O argumento para o uso de especificação formal e de verificação associada de programa é que a especificação formal força uma análise detalhada da especificação. Ela pode revelar inconsistências ou omissões em potencial que, de outro modo, permaneceriam sem ser descobertas até que o sistema estivesse operacional. A verificação formal demonstra que o programa desenvolvido cumpre com sua especificação, de modo que os erros de implementação não comprometam a confiança.

O argumento contra o uso da especificação formal é que ela requer notações especializadas. Essas notações só podem ser utilizadas por pessoal especialmente treinado e não podem ser compreendidas por especialistas em domínio. Portanto, problemas com os requisitos podem ficar ocultos pela formalidade. Os engenheiros de software não podem reconhecer dificuldades em potencial com os requisitos porque não compreendem o domínio; os especialistas em domínio não podem encontrar esses problemas porque não compreendem a especificação. Embora a especificação possa ser matematicamente consistente, ela não pode especificar as propriedades do sistema realmente requeridas.

A verificação de um sistema de software não trivial toma bastante tempo e requer ferramentas especializadas, como provadores de teoremas e perícia matemática. Em conseqüência, esse é um processo extremamente dispendioso e os custos não são lineares. À medida que o tamanho do sistema aumenta, os custos de verificação formal crescem desproporcionalmente. Portanto, muitas pessoas pensam que a verificação formal não é eficaz em termos de custo. O mesmo nível de segurança ou de proteção pode ser alcançado, a custos menores, pelo uso de outras técnicas de validação, como as inspeções e os testes de sistemas. Atualmente, é impossível confirmar ou refutar essa asserção, uma vez que poucos sistemas foram desenvolvidos utilizando-se métodos formais.

Algumas vezes, foi afirmado que o uso de métodos formais para o desenvolvimento de sistemas conduz a sistemas mais confiáveis e mais seguros. Não há dúvida de que uma especificação formal de sistema apresenta menor probabilidade de conter anomalias que tenham de ser resolvidas pelo projetista do sistema. Contudo, a especificação formal e a prova não garantem que o software seja confiável no uso prático. As razões disso são:

1. *A especificação pode não refletir os reais requisitos dos usuários do sistema* Lutz (1993) constatou que muitas falhas experimentadas pelos usuários eram conseqüências de erros e omissões de especificação, que não poderiam ser detectadas pela especificação formal do sistema. Além do mais, raras vezes os usuários compreendem as notações formais, de modo que não podem ler a especificação formal diretamente para encontrar erros e omissões.
2. *A prova pode conter erros* As provas de programa são grandes e complexas e, assim como os programas grandes e complexos, elas geralmente contêm erros.
3. *A prova pode supor um padrão de utilização incorreto* Se o sistema não for utilizado como previsto, a prova poderá ser inválida.

A despeito de suas desvantagens, minha opinião é de que os métodos formais têm um importante papel a desempenhar no desenvolvimento de softwares relacionados com a segurança e com a proteção. As especificações formais são muito eficazes para detectar problemas de especificação, que são as causas mais comuns das falhas de sistema. A verificação formal aumenta a confiança nos componentes mais importantes desses sistemas. O uso de abordagens formais está aumentando à medida que os compradores fazem disso uma exigência e mais e mais engenheiros se familiarizam com suas técnicas. Contudo, levará muitos anos antes que seu uso se torne universal para o desenvolvimento de sistemas críticos.

21.2 Validação de confiabilidade

Como foi explicado no Capítulo 17, uma série de diferentes medidas foram desenvolvidas, as quais podem ser utilizadas para especificar os requisitos de confiabilidade de um sistema. Para validar se o sistema cumpre com esses requisitos, é necessário medir a confiabilidade desse sistema como ela é percebida por um típico usuário de sistema.

O processo de medição de confiabilidade de um sistema é ilustrado na Figura 21.1. Esse processo envolve quatro estágios:

1. Os sistemas existentes do mesmo tipo são estudados para estabelecer um perfil operacional. Um perfil operacional define diferentes classes de entradas do sistema e a probabilidade dessas entradas em uso normal. Será discutido isso nas seções seguintes.
2. Um conjunto de dados de teste é construído (algumas vezes, com a ajuda de geradores de dados de teste), refletindo o perfil operacional.
3. O sistema é testado com esses dados e o número de falhas é observado. Os tempos dessas falhas também são registrados. Como foi discutido no Capítulo 17, as unidades de tempo escolhidas devem ser apropriadas para a medida de confiabilidade utilizada.
4. Depois que um número estatisticamente significativo de falhas foi observado, a confiabilidade do software pode ser calculada. É possível, então, calcular o valor apropriado da medida de confiabilidade.

Essa abordagem é, por vezes, chamada de testes estatísticos. O objetivo dos testes estatísticos é avaliar a confiabilidade do sistema. Isso contrasta com os testes para a detecção de defeitos, cujo objetivo é descobrir defeitos no sistema. Prowell *et al.* (1999) discutem os testes estatísticos em seu livro sobre engenharia de software Cleanroom.

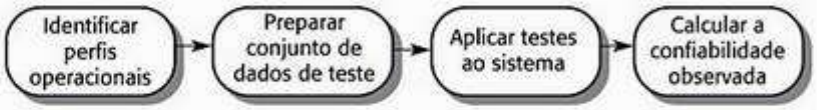
Essa abordagem conceitualmente atrativa para a medição de confiabilidade não é fácil de ser aplicada na prática. As principais dificuldades que surgem são devidas à:

1. *Incerteza do perfil operacional* Os perfis operacionais podem não ser uma reflexão precisa do uso real do sistema.
2. *Altos custos de geração de dados de teste* A definição de uma grande quantidade de dados de teste toma muito tempo, se não for possível gerar esses dados automaticamente.
3. *Incerteza estatística quando uma alta confiabilidade é especificada* É importante gerar um número estatisticamente significativo de falhas, a fim de permitir medições acuradas da confiabilidade.

O desenvolvimento de um perfil operacional acurado certamente é possível para alguns tipos de sistemas que tenham um modelo padronizado de uso. Para outros sistemas, contudo, exis-

Figura 21.1 ▶

Processo de medição de confiabilidade.



tem muitos usuários diferentes, e cada um deles tem sua maneira própria de utilizar o sistema. Como foi discutido no Capítulo 16, diferentes usuários podem ter impressões bem diferentes da confiabilidade porque utilizam diferentes serviços e recursos do sistema.

Sem dúvida alguma, a melhor maneira de gerar o grande conjunto de dados que é exigido para as medições de confiabilidade é utilizar alguma forma de gerador de dados de teste que possa ser estabelecida para gerar automaticamente entradas que combinam com o perfil operacional. Contudo, em geral não é possível automatizar a produção de todos os dados de teste para sistemas interativos. Os conjuntos de dados para esses sistemas têm de ser gerados manualmente, com custos correspondentemente maiores.

A incerteza estatística é um problema geral, quando tentamos medir a confiabilidade de um sistema. Para fazer uma previsão acurada da confiabilidade, é preciso mais do que simplesmente provocar uma única falha no sistema. É necessário gerar um número razoavelmente grande de falhas para ter a confiança de que a medição é precisa. Então, o problema é que, quanto mais se minimiza o número de defeitos em um sistema, mais difícil se torna medir a eficácia das técnicas de minimização de defeitos. Se forem especificados níveis muito altos de confiabilidade, é impraticável gerar falhas de sistema suficientes para verificar essas especificações.

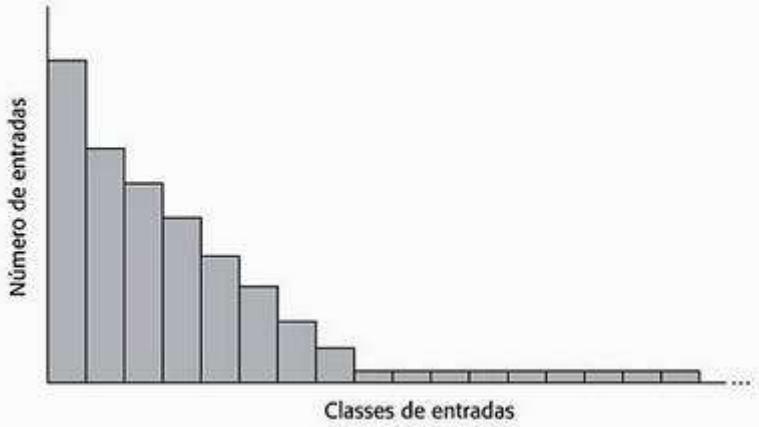
21.2.1 Perfis operacionais

O perfil operacional do software reflete como ele será utilizado na prática. Ele consiste em uma especificação de classes de entrada e da probabilidade de sua ocorrência. Quando um novo sistema de software substitui um sistema manual ou automatizado já existente, é razoavelmente fácil avaliar o padrão provável de uso do novo software. Ele deve corresponder aproximadamente ao uso existente, com alguma concessão feita à nova funcionalidade, que é (presumivelmente) incluída no novo software. Por exemplo, um perfil operacional pode ser especificado para os sistemas de comutação de telecomunicações, porque as empresas de telecomunicação conhecem os padrões de chamadas que esses sistemas têm de manipular.

Tipicamente, o perfil operacional é tal que as entradas que têm a maior probabilidade de ser geradas se classificam em um pequeno número de classes, como é mostrado à esquerda da Figura 21.2. Existe um número extremamente grande de classes em que entradas são altamente improváveis, mas não impossíveis. Essas classes são mostradas à direita da Figura 21.2. As reticências (...) significam que há muito mais dessas entradas incomuns que não são mostradas.

Figura 21.2 ▶

Um perfil operacional.



Musa (1993, 1998) sugere diretrizes para o desenvolvimento de perfis operacionais. No domínio de aplicação em que ele trabalhava (sistemas de telecomunicações), existe uma longa história de coleta de dados de utilização e, assim, o processo de desenvolvimento do perfil operacional é relativamente direto. Para um sistema que precisa de cerca de 15 pessoas/ano de desenvolvimento, um perfil operacional de cerca de uma pessoa/mês foi desenvolvido. Em outros casos, a geração do perfil operacional tomou mais tempo (2–3 pessoas/ano), mas o custo, naturalmente, é distribuído em uma série de versões do sistema. Musa reconhece que sua empresa (uma empresa de telecomunicações) teve, pelo menos, um retorno de dez vezes o investimento requerido para desenvolver um perfil operacional.

Contudo, quando um sistema de software é novo e inovador, é mais difícil prever como ele será utilizado. Os sistemas são utilizados por uma gama de usuários com diferentes expectativas, conhecimentos e experiência, e não existe nenhum banco de dados com o histórico de utilização. Os usuários de computadores, freqüentemente, fazem uso de sistemas de maneiras não previstas por seus desenvolvedores. O problema é ainda mais complexo pelo fato de que o perfil operacional pode se modificar, à medida que o sistema é utilizado. Quando as habilidades e a confiança dos usuários se modificam, à medida que eles ganham experiência com o sistema, eles podem utilizá-lo de maneira mais sofisticada. Devido a essas dificuldades, Hamlet (1992) sugere que, muitas vezes, é impossível desenvolver um perfil operacional confiável. Portanto, é difícil estimar o grau de incerteza nas medições de confiabilidade.

21.2.2 Predição de confiabilidade

Durante a validação de software, os gerentes precisam se dedicar aos testes de sistema. Como os testes são muito dispendiosos, é importante interrompê-los assim que possível, e não testar o sistema além do necessário. Os testes podem cessar quando o nível exigido de confiabilidade do sistema for atingido. Algumas vezes, naturalmente, as predições de confiabilidade podem revelar que o nível requerido de confiabilidade nunca será atingido. Nesse caso, o gerente tem de tomar decisões difíceis sobre reescrever partes do software ou renegociar o contrato do sistema.

Um modelo de aumento de confiabilidade é um modelo de como a confiabilidade do sistema se modifica ao longo do tempo, durante o processo de realização de testes. À medida que os defeitos de sistema são descobertos, os defeitos subjacentes, causadores dessas falhas, são reparados, de maneira que a confiabilidade do sistema deve melhorar durante o teste e a depuração do sistema. Para prever a confiabilidade, o modelo conceitual de aumento de confiabilidade precisa então ser traduzido em um modelo matemático. Aqui, não vou entrar nesse nível de detalhes, mas pretendo simplesmente discutir a noção conceitual do aumento da confiabilidade.

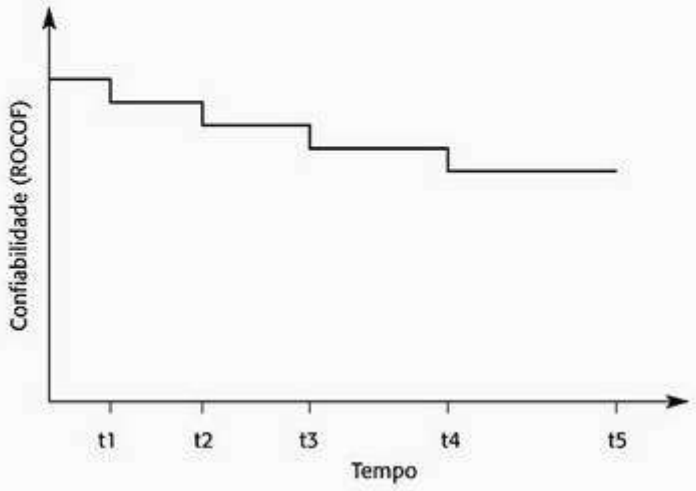
Há vários modelos que foram derivados de experimentos referentes à confiabilidade, em uma série de diferentes domínios de aplicação. O modelo mais simples de aumento de confiabilidade é um modelo de função por etapas (Jelinski e Moranda, 1972), em que a confiabilidade aumenta por um incremento constante toda vez que um defeito é descoberto e reparado (Figura 21.3). Esse modelo assume que os reparos de software são sempre corretamente implementados, de modo que o número de defeitos de software e das falhas associadas diminui com o tempo. À medida que os reparos são feitos, a ROCOF deve, por conseguinte, diminuir, como mostra a Figura 21.3. Observe que os períodos de tempo no eixo horizontal refletem o tempo entre as versões do sistema para teste e, assim, normalmente, têm tempos de duração desiguais.

Na prática, contudo, a depuração algumas vezes falha em corrigir defeitos de software e a alteração no código pode introduzir novos defeitos no sistema. A probabilidade de ocorrência desses defeitos pode ser maior do que a probabilidade de ocorrência do defeito que foi reparado. Portanto, a confiabilidade do sistema pode piorar, em vez de melhorar.

O modelo simples de aumento de confiabilidade por etapas iguais também presume que todos os defeitos contribuem igualmente para a confiabilidade e que cada reparo de defeito

Figura 21.3

Modelo de aumento de confiabilidade de função por etapas iguais.



contribui com a mesma quantidade de aumento de confiabilidade. Contudo, nem todos os defeitos são igualmente prováveis. O reparo dos defeitos mais comuns contribui mais para o aumento da confiabilidade do que o reparo de defeitos que somente ocorrem ocasionalmente. Portanto, o aumento da confiabilidade não é constante em cada período de tempo adicional.

Modelos posteriores, como o sugerido por Littlewood e Verrall (1973), levam esses problemas em consideração, introduzindo um elemento aleatório na melhoria do aumento da confiabilidade efetuado por um reparo de software. Assim, cada reparo não resulta em uma igual melhoria da confiabilidade, mas varia dependendo da perturbação aleatória (Figura 21.4).

O modelo de Littlewood e Verrall permite o aumento negativo da confiabilidade, quando um reparo de software introduz outros erros. Ele também modela o fato de que, à medida que os defeitos são reparados, a melhoria média na confiabilidade por reparo diminui. A razão disso é que os defeitos mais prováveis têm maior propensão a ser descobertos antes no processo de teste. O reparo, nesses casos, contribui para a maior parte do aumento da confiabilidade.

Os modelos mostrados anteriormente são modelos discretos, que refletem o aumento incremental da confiabilidade. Quando uma nova versão de software com defeitos reparados é entregue para testes, ela deve ter uma ROCOF mais baixa do que a versão precedente. Contudo, para prever a confiabilidade que será atingida depois de uma determinada quantidade de testes, os modelos matemáticos contínuos são necessários. Muitos modelos diferentes, derivados de diferentes domínios de aplicação, foram propostos e comparados (Musa *et al.*, 1987; Abdel-Ghaly *et al.*, 1986).

Figura 21.4

Modelo de aumento de confiabilidade de função por etapas aleatórias.

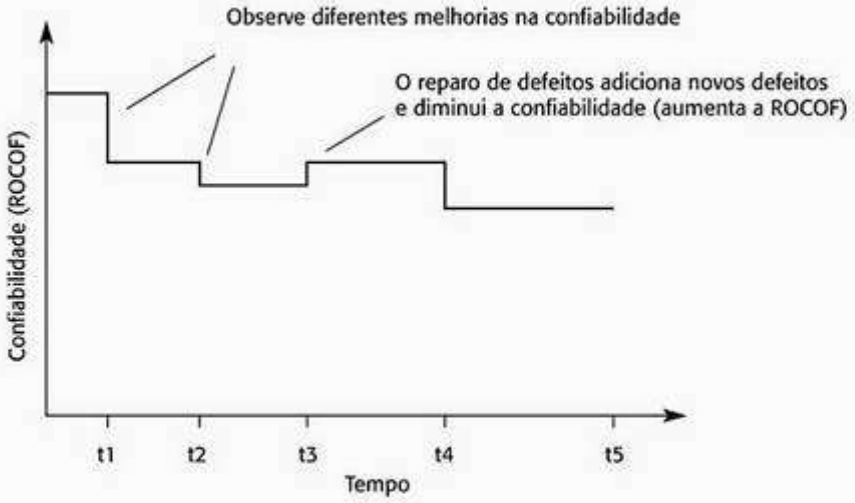
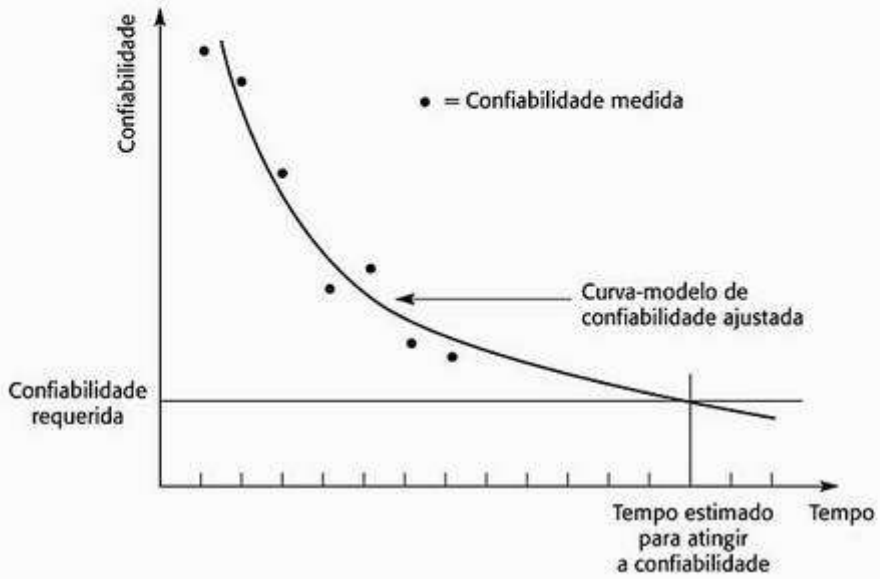


Figura 21.5 ▶

Previsão da confiabilidade.



De maneira simples, a confiabilidade pode ser prevista combinando-se os dados da confiabilidade medida com um modelo de confiabilidade conhecido. Esse modelo é então extrapolado para o nível de confiabilidade exigida. O tempo em que isso será atingido pode então ser verificado no gráfico (Figura 21.5). Portanto, os testes e a depuração devem continuar até esse momento.

Prever a confiabilidade do sistema a partir de um modelo de confiabilidade tem dois benefícios principais:

1. *Planejamento de testes* Considerando a atual programação de testes, pode ser previsto o momento em que eles estarão completos. Se isso ocorrer depois do prazo de entrega programado para o sistema, então recursos adicionais de testes poderão ser empregados para acelerar a taxa de aumento da confiabilidade.

2. *Negociações com o cliente* Algumas vezes, o modelo de confiabilidade mostra que o aumento da confiabilidade é muito lento e que uma quantidade desproporcional de esforço relativo a testes é exigida, proporcionando um benefício relativamente pequeno. Pode valer a pena renegociar os requisitos de confiabilidade com o cliente. Como alternativa, pode acontecer que o modelo preveja que a confiabilidade exigida é improvável de ser atingida. Nesse caso, a renegociação dos requisitos é essencial.

Os leitores que estiverem interessados em mais detalhes sobre o aumento de confiabilidade e os modelos de aumento de confiabilidade devem consultar o livro de Musa e os artigos de Littlewood mencionados aqui. Vários autores descreveram sua experiência prática com o uso de modelos de aumento de confiabilidade (Schneidewind e Keller, 1992; Sheldon *et al.*, 1992; Ehrlich *et al.*, 1993).

21.3 Garantia de segurança

A garantia de segurança e a validação de confiabilidade são processos bem diferentes. É possível especificar a confiabilidade quantitativamente utilizando-se algumas métricas e, então, medir a confiabilidade do sistema completo. A segurança não pode ser significativamente

especificada de maneira quantitativa e, portanto, não pode ser medida quando um sistema é testado.

Desse modo, a validação da segurança se refere ao estabelecimento de um nível de confiança no sistema que pode variar desde 'muito baixo' até 'muito alto'. Esse é um assunto para avaliação profissional. Em muitos casos, essa confiança é, em parte, baseada na experiência da organização que desenvolve o sistema. Se a empresa desenvolveu anteriormente uma série de sistemas de controle que operaram com segurança, então é razoável supor que ela continuará a desenvolver sistemas seguros desse tipo. Contudo, essa avaliação precisa ser sustentada por evidências tangíveis a partir do projeto de sistema, dos resultados da verificação e validação de sistema e dos processos de desenvolvimento do sistema que foram utilizados.

21.3.1 Verificação e validação

A verificação e validação de sistemas críticos de segurança têm muito em comum com os testes de quaisquer outros sistemas com altos níveis de requisitos de confiabilidade. Deve haver testes extensivos para descobrir tantos defeitos quantos forem possíveis, e os testes estatísticos devem ser utilizados para avaliar a confiabilidade do sistema. Contudo, devido às taxas de falhas extremamente baixas, que são requeridas em muitos sistemas críticos de segurança, os testes estatísticos nem sempre podem fornecer uma estimativa quantitativa da confiabilidade do sistema, em razão do elevadíssimo número de testes exigidos. Os testes simplesmente dão evidências que são utilizadas com outras evidências, como os resultados das revisões e da verificação estatística (veja o Capítulo 19) para julgar a segurança do sistema.

As revisões extensivas são essenciais durante um processo de desenvolvimento orientado à segurança. Parnas *et al.*, (1990) sugerem cinco tipos de revisão, que devem ser obrigatórias para os sistemas críticos de segurança:

1. revisão da função correta pretendida;
2. revisão de estruturas compreensíveis, de fácil manutenção;
3. revisão para verificar se o algoritmo e o projeto de estrutura de dados são consistentes com o comportamento especificado;
4. revisão da consistência do código e do algoritmo e do projeto de estrutura de dados, e
5. revisão de adequação dos casos de teste do sistema.

Uma suposição que fundamenta o trabalho de segurança de sistemas é de que o número de defeitos de sistema que pode levar a perigos é significativamente menor do que o número total de defeitos que podem existir no sistema. A garantia de segurança pode se concentrar nesses defeitos com perigo em potencial. Se puder ser demonstrado que esses defeitos não podem ocorrer ou que, se eles ocorrerem, o perigo associado não resultará em acidentes, então o sistema é seguro. Essa é a base dos argumentos de segurança, que são discutidos na próxima seção.

21.3.2 Argumentos de segurança

As provas de correção de programas têm sido propostas como uma técnica de verificação de software há mais de 25 anos. Entretanto, elas têm sido pouco utilizadas, exceto em laboratórios de pesquisa. Os problemas práticos da construção de uma prova de correção são tão grandes que poucas organizações as consideraram eficazes, em termos de custos, no desenvolvimento normal de sistemas. Contudo, como foi discutido anteriormente neste capítulo, para algumas aplicações críticas, pode ser econômico desenvolver provas de correção para aumentar a confiança de que o sistema cumpre com seus requisitos de segurança ou de proteção.

Embora talvez não seja eficaz, em termos de custos, desenvolver provas de correção para a maioria dos sistemas, algumas vezes, é possível desenvolver um argumento de segurança que demonstre que o programa cumpre com suas obrigações de segurança. Não é necessá-

rio provar que o programa cumpre com sua especificação; é preciso apenas demonstrar que a execução do programa não pode resultar em um estado inseguro.

A técnica mais eficaz para demonstrar a segurança de um sistema é a prova por contradição. Isso significa supor que o estado inseguro (identificado pela análise de perigo) pode ser alcançado com a execução do programa. Então, sistematicamente, o código é analisado, e demonstra-se que as pré-condições para esse estado de perigo são contraditórias às pós-condições de todos os caminhos que conduzem àquele estado. Se esse for o caso, a suposição inicial de um estado inseguro é incorreta. Se isso se repetir para todos os perigos identificados, então o software é seguro.

Como exemplo, considere o código na Figura 21.6, que pode ser parte da implementação do sistema de fornecimento de insulina. Alguns comentários foram acrescentados a esse código, a fim de relacioná-lo à árvore de defeitos mostrada na Figura 17.6.

O desenvolvimento de um argumento de segurança para esse código envolve demonstrar que a dosagem de insulina administrada nunca é maior do que um nível máximo, que é estabelecido para cada indivíduo diabético. Portanto, não é necessário provar que o sistema aplica a dose 'correta', mas apenas que ele nunca aplica uma dose excessiva no paciente.

Para construir o argumento de segurança, identifica-se a pré-condição para o estado inseguro que, nesse caso, é que currentDose > maxDose. Demonstra-se então que todos os caminhos do programa levam a uma contradição da asserção de insegurança. Se esse for o caso, a condição de insegurança não poderá ser verdadeira. Portanto, o sistema é seguro.

Os argumentos de segurança, como aqueles mostrados na Figura 21.7, são muito mais curtos do que as verificações formais de sistema. Primeiramente, são identificados todos os possíveis caminhos que conduzem ao estado potencialmente inseguro. Trabalha-se retroativamente, a partir desse estado inseguro, e se considera a última designação de todas as variáveis de estado em cada caminho que leva a ele. Cálculos prévios (como a declaração if 1, na Figura 21.7) não têm de ser considerados no argumento de segurança. Nesse exemplo, basta prestar atenção ao conjunto de possíveis valores de currentDose, imediatamente antes de o método administerInsulin ser executado.

No argumento de segurança mostrado na Figura 21.7, há três possíveis caminhos de programa que levam à chamada do método administerInsulin:

1. Nenhum ramo da declaração if 2 é executado. Isso somente pode acontecer se currentDose for maior que ou igual a minimumDose e menor que ou igual a maxDose.

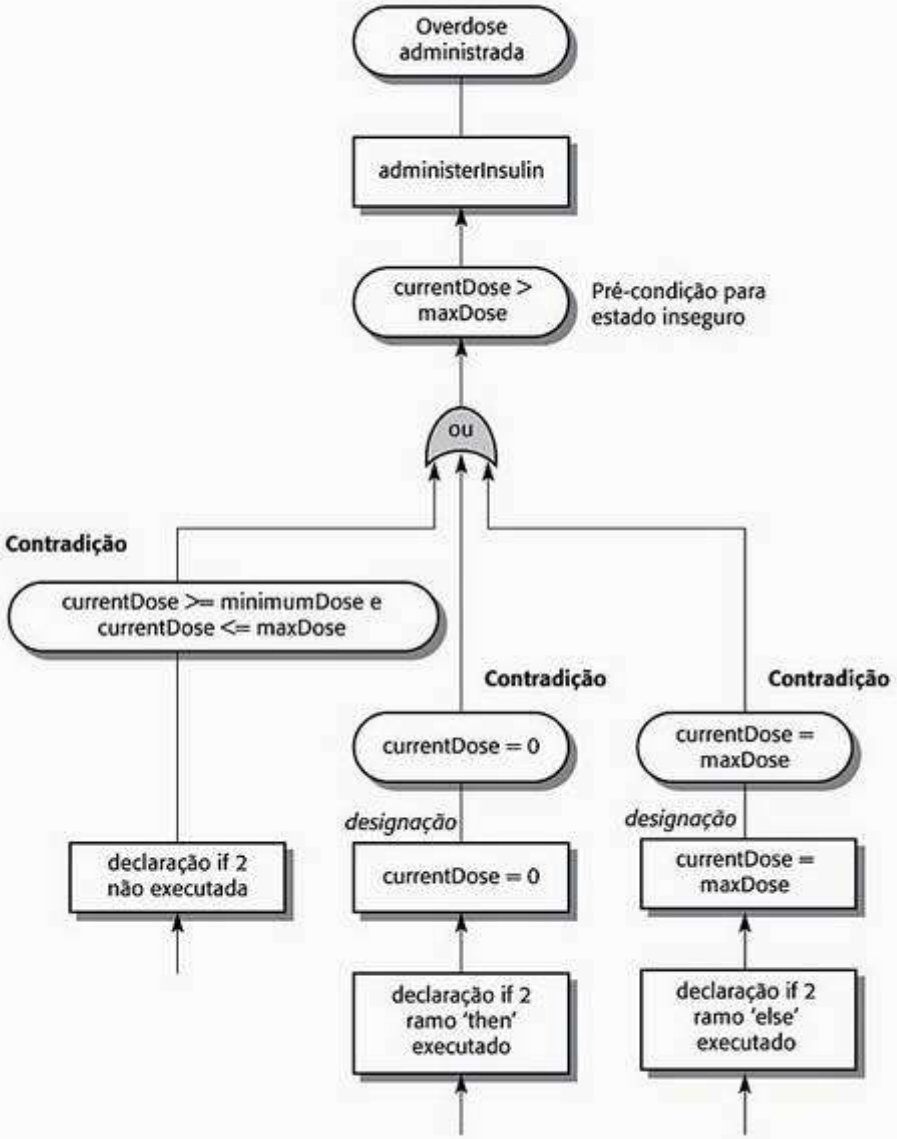
Figura 21.6 ▶

Código de fornecimento de insulina.

```
-- A dose de insulina a ser fornecida é uma função do
-- nível de açúcar no sangue, da dose anterior fornecida e
-- do tempo decorrido a partir do fornecimento da dose anterior

    currentDose = computeInsulin ( ) ;

    // Checar segurança - ajustar currentDose se necessário

    // declaração if 1

    if (previousDose = = 0)
    {
        if (currentDose > 16)
            currentDose = 16 ;
    }
    else
        if (currentDose > (previousDose * 2) )
            currentDose = previousDose * 2 ;

    // declaração if 2

    if ( currentDose < minimumDose )
            currentDose = 0 ;
    else if ( currentDose > maxDose )
            currentDose = maxDose ;
    administerInsulin (currentDose) ;
```

Figura 21.7

Argumento de segurança informal baseada nas demonstrações de contradições.



2. O ramo 'then' da declaração if 2 é executado. Nesse caso, a designação de currentDose para zero é executada. Portanto, sua pós-condição é currentDose = 0.
3. O ramo 'if' de 'else' da declaração if 2 é executado. Nesse caso, a designação de currentDose para maxDose é executada. Portanto, sua pós-condição é currentDose = maxDose.

Em todos os três casos, as pós-condições contradizem as pré-condições de insegurança e, assim, o sistema é seguro.

21.3.3 Garantia de processo

Já foi discutida a importância de garantir a qualidade do processo de desenvolvimento do sistema na introdução deste capítulo. Isso é importante para todos os sistemas críticos, mas é particularmente importante para os sistemas críticos de segurança. Existem duas razões para isso:

1. Acidentes são eventos raros em sistemas críticos, e pode ser particularmente impossível simulá-los durante os testes de um sistema.

2. Os requisitos de segurança, como foi discutido no Capítulo 16, algumas vezes, são requisitos do tipo 'não deve', que excluem um comportamento inseguro do sistema. É impossível demonstrar de maneira conclusiva, por meio de testes e outras atividades de validação, que esses requisitos foram cumpridos.

O modelo de ciclo de vida para o desenvolvimento de sistemas críticos de segurança torna claro que deve ser dada atenção explícita à segurança durante todos os estágios do processo de software. Isso significa que atividades específicas de garantia de segurança devem ser incluídas no processo. Dentre essas atividades destacam-se:

1. A criação de um registro de perigo e de um sistema de monitoramento que rastreie perigos a partir de análise preliminar de perigo até os testes e a validação do sistema.
2. A nomeação de engenheiros de segurança de projeto que tenham responsabilidades explícitas pelos aspectos de segurança do sistema.
3. O uso extensivo de revisões de segurança durante o processo de desenvolvimento.
4. A criação de um sistema de certificação de segurança mediante o qual componentes críticos de segurança sejam formalmente certificados quanto a sua segurança avaliada.
5. O uso de um sistema de gerenciamento de configuração muito detalhado (veja o Capítulo 29), que seja empregado para acompanhar toda a documentação relacionada à segurança e para mantê-la atualizada com a documentação técnica associada. Há pouca razão para se ter procedimentos rigorosos de validação, se uma falha de gerenciamento de configuração significar que o sistema errado será fornecido ao cliente.

Para ilustrar o que está envolvido na validação do processo, é utilizado o processo de análise de perigo, que é parte essencial do desenvolvimento de sistemas críticos de segurança. Relembre do Capítulo 17, em que a análise de perigo se ocupa da identificação e da análise de perigos. Esse é um processo contínuo durante o desenvolvimento do sistema. É importante ser capaz de demonstrar que os perigos foram adequadamente considerados em todos os estágios do processo.

Se o processo de desenvolvimento inclui nítida facilidade de rastreamento, desde a identificação do perigo pelo próprio sistema, então um argumento pode ser feito sobre por que esses perigos não resultarão em acidentes. Isso pode ser complementado por argumentos de segurança, como foi discutido na Seção 21.3.2. Quando a certificação externa é requerida antes de um sistema ser utilizado (por exemplo, em uma aeronave), é uma condição comum de certificação que essa facilidade de rastreamento possa ser demonstrada.

O principal documento de segurança é o registro de perigos, em que os perigos identificados durante o processo de especificação são documentados e rastreados. Esse registro de perigos é então utilizado em cada estágio do processo de desenvolvimento de software, a fim de avaliar como esse estágio de desenvolvimento levou aos perigos em consideração. Um exemplo simplificado de entradas de registro de perigos para o sistema de fornecimento de insulina é mostrado na Figura 21.8, que documenta o processo de análise de perigos e mostra os requisitos específicos gerados durante esse processo.

Como mostra a Figura 21.8, os indivíduos que têm algumas responsabilidades quanto à segurança devem ser explicitamente identificados. É importante nomear um engenheiro de segurança de projeto, que não deverá se envolver no desenvolvimento do sistema. A responsabilidade desse engenheiro é garantir que as verificações de segurança apropriadas tenham sido feitas e documentadas. O comprador do sistema pode também exigir um assessor de segurança independente, a ser indicado a partir de uma organização externa, para prestar contas diretamente ao cliente, sobre os assuntos de segurança.

Em muitos tipos de aplicação, os engenheiros de sistema com responsabilidades de segurança devem ser engenheiros certificados. No Reino Unido, isso significa que eles têm de ser

Figura 21.8

Página simplificada de registro de perigo.

Registro de perigo — **Página 4: impressa em 20.02.99**

Sistema: Sistema de Bomba de Insulina	*Arquivo: InsulinPump/Segurança/Registro de perigo*
Engenheiro de Segurança: James Brown	*Versão de registro: 1/3*
Perigo Identificado	Dose excessiva de insulina fornecida ao cliente
Identificado por	Jane Williams
Classe do nível crítico	1
Risco identificado	Alto
Árvore de defeitos identificada	SIM *Data* 24.01.99 *Localização:* Registro de perigo, Página 5
Criadores da árvore de defeitos	Jane Williams e Bill Smith
Árvore de defeitos checada	SIM *Data* 28.01.99 *Conferente:* James Brown

Requisitos de projeto de segurança de sistema

1. O sistema incluirá um software de autoteste, que testará o sistema de sensores, o relógio e o sistema de fornecimento de insulina.
2. O software de autoteste será executado uma vez por minuto.
3. No evento de o software de autoteste constatar um defeito em qualquer componente do sistema, um aviso sonoro será emitido e o display da bomba deverá indicar o nome do componente em que o defeito foi descoberto. O fornecimento de insulina deverá ser suspenso.
4. O sistema incorporará um sistema de sobreposição, que permite que seu usuário modifique a dosagem calculada de insulina a ser aplicada pelo sistema.
5. A quantidade de sobreposição deve ser limitada a não mais do que um valor predeterminado, que é definido quando o sistema é configurado pela equipe médica.

aceitos como membros de um dos institutos de engenharia (civil, elétrico, mecânico etc.) e que precisam ser engenheiros diplomados. Engenheiros inexperientes, com baixa qualificação, não podem assumir responsabilidades quanto à segurança. Isso não se aplica atualmente a engenheiros de software. Contudo, futuros padrões de processo para o desenvolvimento de software crítico de segurança podem exigir que os engenheiros de segurança de projeto sejam formalmente certificados com um nível mínimo de treinamento definido.

21.3.4 Verificação de segurança em *run-time*

A noção de programação defensiva foi introduzida no Capítulo 18, em que declarações redundantes foram adicionadas a programas, a fim de verificar possíveis defeitos no sistema. Uma técnica similar pode ser utilizada nos sistemas críticos de segurança. A verificação de código pode ser acrescentada para checar uma restrição de segurança e para lançar uma exceção, se essa restrição for violada. As restrições de segurança, que sempre devem ser mantidas em certos pontos de um programa, podem ser expressas como asserções. As asserções são predicados que descrevem condições que devem ser mantidas, antes que a próxima declaração seja executada. Em sistemas críticos de segurança, as asserções devem ser geradas a partir da especificação de segurança. Elas se destinam a garantir um comportamento seguro, em vez de um comportamento que atenda à especificação.

As asserções podem ser particularmente valiosas para garantir a segurança das comunicações entre os componentes do sistema. Por exemplo, no sistema de fornecimento de insulina, a dose de insulina ministrada envolve gerar sinais para a bomba de insulina, a fim de fornecer uma série de doses de insulina especificadas (Figura 21.9). O número de doses de insulina associado com a dose máxima permitida pode ser pré-calculada e incluída como uma asserção no sistema.

Portanto, se tiver ocorrido um erro de cálculo para currentDose – a variável de estado que especifica a quantidade de insulina a ser ministrada – ou se esse valor tiver sido corrompido de algum modo, então isso será detectado nesse estágio. Uma dose excessiva de insu-

Figura 21.9

Administração de insulina.

```
static void administerInsulin ( ) throws SafetyException {
        int maxIncrements = InsulinPump.maxDose / 8 ;
        int increments = InsulinPump.currentDose / 8 ;
        // assert currentDose <= InsulinPump.maxDose
        if (InsulinPump.currentDose > InsulinPump.maxDose)
            throw new SafetyException (Pump.doseHigh) ;
        else
            for (int i = 1 ; i <= increments; i++)
            {
                generateSignal () ;
                if (i > maxIncrements)
                    throw new SafetyException (Pump.incorrectIncrements) ;
            } // for loop
    } //administerInsulin
```

lina não será aplicada, uma vez que a verificação no método garante que a bomba não ministrará mais do que a maxDose.

A partir das asserções incluídas como comentários do programa, o código para verificar essas asserções pode ser gerado, como ilustra a Figura 21.9. Em princípio, grande parte dessa geração de código pode ser automatizada, utilizando-se um pré-processador de asserções. Contudo, raras vezes esses dispositivos estão disponíveis e o código de asserção é normalmente gerado de maneira manual.

21.4 Avaliação de proteção

A avaliação de sistemas de proteção está se tornando cada vez mais importante, à medida que mais e mais sistemas estão sendo conectados à Internet. Como foi discutido no Capítulo 16, os requisitos de proteção são semelhantes aos requisitos de segurança em alguns aspectos, no sentido de que elas são freqüentemente chamadas de requisitos do tipo 'não deve'. Isto é, elas especificam o comportamento do sistema que não é permitido, em vez do comportamento que é esperado. Todavia, nem sempre é possível definir esse comportamento como simples restrições que podem ser verificadas pelo sistema.

Uma diferença fundamental entre segurança e proteção é que os problemas de segurança são geralmente acidentais, enquanto os ataques a um sistema são propositais. Mesmo em sistemas que tenham estado em uso durante muitos anos, um atacante engenhoso pode descobrir um novo modo de ataque e pode penetrar no que era julgado um sistema seguro. Por exemplo, o algoritmo RSA para criptografar dados, que era considerado seguro, foi violado, em 1999.

Existem quatro abordagens complementares para verificar a proteção:

1. *Validação baseada na experiência* Nesse caso, o sistema é analisado em comparação com tipos de ataques conhecidos pela equipe de validação. Esse tipo de validação é geralmente realizado em conjunto com a validação baseada em ferramentas.

2. *Validação baseada em ferramentas* Nesse caso, várias ferramentas de segurança, como os verificadores de senhas, são utilizadas para analisar o sistema. Isso é realmente uma extensão da validação baseada na experiência, em que a experiência está incorporada nas ferramentas utilizadas.

3. *'Equipes de tigres'* Nesse caso, é constituída uma equipe que tem o objetivo de invadir o sistema de proteção. Ela simula ataques ao sistema e utiliza sua engenhosidade para descobrir novas maneiras de comprometer a proteção do sistema.

4. *Verificação formal* Um sistema pode ser verificado em comparação com uma especificação formal de segurança. Contudo, como em outras áreas, a verificação formal para proteção não é amplamente utilizada.

É muito difícil para os usuários finais de um sistema verificar sua proteção. Conseqüentemente, como foi discutido por Gollmann (1999), entidades na América do Norte e na Europa estabeleceram conjuntos de critérios de avaliação de proteção que podem ser conferidos por avaliadores especializados. Fornecedores de produtos de software podem submeter seus produtos à avaliação e à certificação em relação a esses critérios.

Portanto, se houver um requisito para um nível particular de proteção, é possível escolher um produto que tenha sido validado para esse nível. Contudo, muitos produtos não são certificados quanto à proteção, e a certificação se aplica a produtos individuais. Quando o sistema certificado é utilizado em conjunto com outros sistemas não certificados, como softwares desenvolvidos localmente, então o nível de proteção do sistema total não pode ser avaliado.

PONTOS-CHAVE

- Em razão dos altos custos das falhas em sistemas críticos, as técnicas de verificação e validação, como a especificação formal e a prova, que são normalmente muito dispendiosas, podem ser eficazes, em termos de custo, para esses sistemas.
- Os testes estatísticos são utilizados para estimar a confiabilidade do software. Eles se concentram em testar o sistema com um conjunto de dados de teste que reflete o perfil operacional do software. Os dados de teste podem ser gerados automaticamente.
- Os modelos de aumento de confiabilidade modelam as modificações na confiabilidade, à medida que os defeitos são removidos do software durante o processo de realização de testes. Os modelos de confiabilidade podem ser utilizados para prever quando a confiabilidade exigida será atingida.
- As provas de segurança são uma técnica eficaz da garantia de segurança do produto. Elas mostram que uma condição perigosa identificada nunca poderá ocorrer. Geralmente, elas são mais simples do que provar que um programa cumpre com suas especificações.
- É importante ter um processo bem definido, certificado para o desenvolvimento de sistemas críticos de segurança. O processo tem de incluir a identificação e o monitoramento de perigos em potencial.
- A validação da proteção pode ser realizada utilizando-se a análise baseada na experiência, a análise baseada em ferramentas ou então 'equipes de tigres' para simular ataques ao sistema.

LEITURAS SUGERIDAS

Software reliability engineering: more reliable software, faster development and testing. Esse é, provavelmente, o livro definitivo sobre o uso de perfis operacionais e modelos de confiabilidade para a avaliação de confiabilidade. Ele traz detalhes sobre experiências com testes estatísticos. (J.D. Musa, 1998, McGraw-Hill.)

Safety-critical computer systems. Esse excelente livro inclui um capítulo particularmente bom, em vez de métodos formais, sobre o desenvolvimento de sistemas de segurança crítica. (N. Storey, 1996, Addison Wesley.)

Safeware: system safety and computers. Apresenta um bom capítulo sobre a validação de sistemas críticos de segurança, com mais detalhes do que apresentei aqui sobre o uso de argumentos de segurança baseados em árvores de defeitos. (N. Leveson, 1995, Addison Wesley.)

EXERCÍCIOS

21.1 Descreva como você faria para validar a especificação de confiabilidade para um sistema de loja especificado no Exercício 17.3. Sua resposta deve incluir uma descrição de quaisquer ferramentas de validação que possam ser utilizadas.

21.2 Explique por que é praticamente impossível validar especificações de confiabilidade quando elas são expressas em termos de um número muito pequeno de falhas em relação ao tempo de duração total de um sistema.

21.3 Utilizando a literatura como informação de apoio, escreva um relatório para a gerência (que não tem nenhuma experiência prévia nessa área) sobre o uso de modelos de aumento de confiabilidade.

21.4 É ético um engenheiro concordar em entregar um sistema de software com defeitos conhecidos para um cliente? Faz alguma diferença se o cliente for avisado sobre esses defeitos antecipadamente? Seria razoável fazer afirmações sobre a confiabilidade do software nessas circunstâncias?

21.5 Explique por que garantir a confiabilidade do sistema não é uma garantia da segurança do sistema.

21.6 O mecanismo de controle de fechamento da porta em uma área de depósito de lixo nuclear é projetado para proporcionar uma operação segura. Dessa maneira, garante-se que a entrada para o depósito somente seja permitida quando os anteparos de radiação estiverem no lugar ou quando o nível de radiação na sala estiver abaixo de determinado valor (dangerLevel). Isto é,

(i) se os anteparos de radiação controlados remotamente estiverem ajustados dentro de uma sala, a porta pode ser aberta por um operador autorizado;
(ii) se o nível de radiação em uma sala for menor do que um valor especificado, a porta pode ser aberta por um operador autorizado, e
(iii) um operador autorizado é identificado por meio de um código de entrada também autorizado.

O código em Java mostrado na Figura 21.10 é utilizado para controlar o mecanismo de fechamento da porta. Observe que o estado seguro indica que a entrada não deve ser permitida.

Desenvolva um argumento de segurança que mostre que esse código é potencialmente inseguro. Modifique o código para torná-lo seguro.

21.7 Utilizando a especificação para o cálculo de dosagem, apresentada no Capítulo 9 (Figura 9.11), escreva em Java o método computeInsulin, como utilizado na Figura 21.6. Construa um argumento de segurança informal de que esse código é seguro.

21.8 Como você faria para validar um sistema de proteção de senhas para uma aplicação que desenvolveu? Explique a função de quaisquer ferramentas que você acredite que possam ser úteis.

21.9 Suponha que você faça parte de uma equipe que desenvolveu um software para uma usina química, que falhou e causou um sério incidente de poluição. Sua chefe é entrevistada na televisão e declara que o processo de validação é abrangente e que não há defeitos no software. Ela assevera que os problemas devem ser devidos a procedimentos operacionais inadequados. Os operadores do sistema rejeitam essa afirmação e propõem fazer uma greve. Como você lidaria com essa situação?

Figura 21.10 ▶

Controlador de fechadura de porta.

```
entryCode = lock.getEntryCode () ;
if (entryCode == lock.authorisedCode)
{
    shieldStatus = Shield.getStatus () ;
    radiationLevel = RadSensor.get () ;
    if (radiationLevel < dangerLevel)
        state = safe ;
    else
        state = unsafe ;
    if (shieldStatus == Shield.inPlace() )
        state = safe ;
    if (state == safe)
        {
            Door.locked = false ;
            Door.unlock () ;
        }
    else
    {
        Door.lock () ;
        Door.locked := true ;
    }
}
```

PARTE SEIS

Gerenciamento

PÁGINA EM BRANCO

22 Gerenciamento de pessoal

Objetivos

O objetivo deste capítulo é discutir a importância das pessoas no processo de engenharia de software. Depois de ler este capítulo, você poderá:

- compreender modelos simples de memória humana, resolução de problemas e motivação e algumas implicações disso para os gerentes de software;
- compreender as principais questões do trabalho em equipe: a composição, a coesão, a comunicação e a organização da equipe;
- compreender algumas das questões envolvidas na seleção e na retenção de pessoal em uma organização de desenvolvimento de software, e
- conhecer o P-CMM – um modelo que é um *framework* para aprimorar as capacidades dos desenvolvedores de software em uma organização.

Sumário

22.1 Limites para o pensamento
22.2 Trabalho em grupo
22.3 Seleção e manutenção de pessoal
22.4 Modelo de Maturidade de Capacitação de Pessoal

As pessoas que trabalham em uma organização de software são seu maior patrimônio. Elas representam o capital intelectual, e é responsabilidade dos gerentes de software garantir que a organização obtenha o melhor retorno de seu investimento em pessoas. Em empresas e economias bem-sucedidas, isso é alcançado quando as pessoas são respeitadas pela organização. Elas devem ter um nível de responsabilidade e compreensão que seja proporcional às suas habilidades.

O gerenciamento eficaz diz respeito, portanto, ao gerenciamento de pessoas em uma organização. Os gerentes de projeto têm de solucionar problemas técnicos e não técnicos, utilizando a capacidade das pessoas de sua equipe da maneira mais eficaz possível. Eles precisam motivar o pessoal, planejar e organizar seu trabalho e assegurar que o trabalho seja feito adequadamente. O gerenciamento inadequado de pessoas é uma das mais significativas contribuições para o fracasso de projetos.

Baseio minha discussão sobre o gerenciamento em fatores cognitivos e sociais, e não em qualquer teoria de gerenciamento atualmente em moda. A engenharia de software é uma atividade cognitiva e social; assim, esses fatores são importantes para desenvolver uma compreensão de como as pessoas escrevem software. Se os gerentes têm alguma compreensão desses fundamentos, eles podem fazer um trabalho mais profícuo para obter o melhor das pessoas com quem trabalham.

22.1 Limites para o pensamento

Existe uma grande diversidade de habilidades individuais, que refletem diferenças quanto à inteligência, educação e experiência, mas a maioria parece estar sujeita a restrições básicas em nosso pensamento. Essas restrições são uma conseqüência da maneira como as informações são armazenadas e modeladas em nossos cérebros. Não é preciso compreender detalhadamente o processamento das informações cognitivas. Contudo, acho que é importante definir uma percepção dos limites da maneira como pensamos. Isso ajuda a explicar por que algumas técnicas de engenharia de software são eficazes e proporcionam a percepção das interações entre pessoas em uma equipe de desenvolvimento de software.

22.1.1 Organização da memória

Os sistemas de software são entidades abstratas, e os engenheiros precisam se lembrar de suas características durante o processo de desenvolvimento. Por exemplo, os programadores devem compreender e se lembrar das relações entre uma listagem de código-fonte e o comportamento dinâmico do programa. Eles aplicam esse conhecimento armazenado em futuros desenvolvimentos de programa.

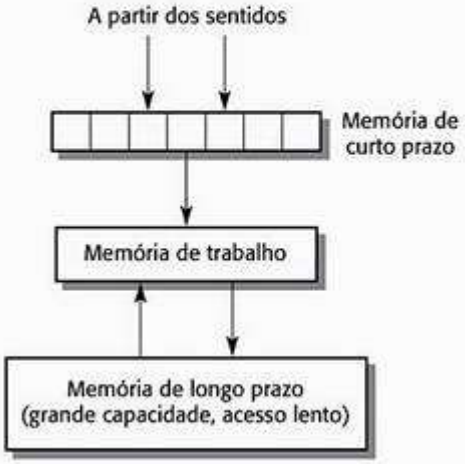
A organização da memória humana parece ser hierárquica, com três áreas distintas e conectadas (Figura 22.1):

1. *Uma memória de capacidade limitada, acesso rápido e de curto prazo* A entrada dos sentidos é recebida aqui para processamento inicial. Essa memória é comparável aos registros em um computador; ela é utilizada para o processamento de informações, e não para seu armazenamento.

2. *Uma área de memória de trabalho, com maior capacidade* Essa área da memória pode ser acessada por mais tempo do que a memória de curto prazo. Ela é utilizada para o processamento de informações, mas pode retê-las por períodos mais longos do que a memória

Figura 22.1 ▶

Organização da memória humana.



de curto prazo. Não é utilizada para a retenção de informações de longo prazo. Numa analogia com o computador, ela é como a RAM (*randomic access memory* – memória de acesso aleatório), em que as informações são mantidas enquanto durar uma computação.

3. *Memória de longo prazo* Essa memória tem grande capacidade, tempo de acesso relativamente lento e mecanismos de recuperação que não são confiáveis (nós nos esquecemos das coisas). A memória de longo prazo é utilizada para o armazenamento permanente das informações. Para continuar com a analogia, a memória de longo prazo é como a memória de disco em um computador.

A informação sobre um problema é colocada na memória de curto prazo, originada a partir da leitura de documentos e de conversas com pessoas. Ela é integrada, juntamente com outras informações relevantes da memória de longo prazo, na memória de trabalho. O resultado dessa integração forma a base para as soluções de problemas. Essas soluções são armazenadas na memória de longo prazo, para uso futuro. Naturalmente, a solução pode ser incorreta. À medida que mais informações se tornam disponíveis, a memória de longo prazo precisa ser modificada. Contudo, as informações incorretas não são inteiramente descartadas, mas retidas de alguma maneira. Sabemos disso porque aprendemos com os nossos erros.

O tamanho limitado da memória de curto prazo restringe nossos processos cognitivos. Em uma experiência clássica, Miller (1957) descobriu que a memória de curto prazo pode armazenar cerca de sete 'quanta' de informação. Um 'quantum' de informação não é um tamanho definido, mas uma entidade de informação coerente. Pode ser um número de telefone, o propósito de um objeto ou um nome de rua. Miller também descreve o processo de 'aglomeração', em que os 'quanta' de informação são reunidos em aglomerados.

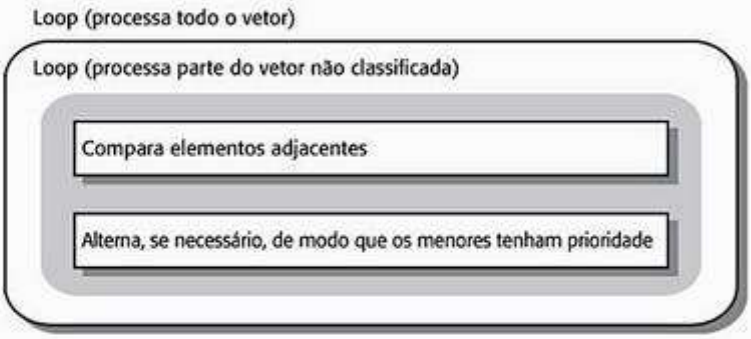
Se um problema envolve a entrada de mais informações do que a memória de curto prazo pode lidar, deve haver o processamento de informações e a transferência durante o processo de entrada. Isso pode resultar na perda de informações. Os erros surgem porque esse processamento de informações não pode se manter com a entrada de memória.

Shneiderman (1980) sugere que um processo de aglomeração de informações é utilizado na compreensão de problemas. Os leitores de programas abstraem as informações no programa em aglomerados, que são integrados em uma estrutura semântica interna. Os programas não são compreendidos a cada declaração, a menos que uma declaração represente um aglomerado lógico. A Figura 22.2 mostra como um simples programa de classificação pode ser 'aglomerado' por alguém que tenta compreendê-lo.

Uma vez estabelecido o programa que representa a estrutura semântica interna, esse conhecimento é transferido para a memória de longo prazo. Se for regularmente utilizada, em geral ela não será esquecida. Pode ser representada em diferentes notações, sem muitas dificuldades. Portanto, parece que aprendemos em termos de abstrações de alto nível, e não em termos de detalhes de baixo nível.

Figura 22.2

Aglomerados cognitivos em um programa de classificação.



O conhecimento adquirido durante o desenvolvimento de software e armazenado na memória de longo prazo se divide em duas classes:

1. *Conhecimento semântico* Esse é o conhecimento de conceitos, como a operação de uma declaração de atribuição, a noção de uma classe de objeto, o modo como uma técnica de busca hash opera e como as organizações são estruturadas. Esse conhecimento é adquirido por meio da experiência e do aprendizado e é retido em um modo de representação independente.

2. *Conhecimento sintático* Esse é o conhecimento da representação detalhada, como escrever uma descrição de objeto em UML, que funções-padrão estão disponíveis em uma linguagem de programação, se uma atribuição é escrita com a utilização de um sinal de '=' ou um ':=' etc. Esse conhecimento parece ser retido de uma maneira não processada.

Essa organização de conhecimento está ilustrada na Figura 22.3, adaptada do livro de Shneiderman (1992), sobre o projeto de interface com o usuário. Esse diagrama sugere que o conhecimento semântico e de tarefas é organizado e estruturado. Os relacionamentos entre diferentes fragmentos de conhecimento foram construídos. Por outro lado, o conhecimento sintático é arbitrário e desorganizado. Portanto, é mais fácil ocorrerem esquecimentos e erros quando se utiliza esse tipo de conhecimento.

O conhecimento semântico é adquirido por experiência e por meio do aprendizado ativo. Informações novas são conscientemente integradas com as estruturas semânticas existentes. O conhecimento sintático parece ser adquirido por memorização. Os novos conhecimentos sintáticos não são imediatamente integrados com o conhecimento já existente, mas podem interferir com ele. Eles são mais facilmente esquecidos do que o conhecimento semântico profundo.

Os diferentes modos de aquisição do conhecimento semântico e sintático ajudam a explicar como programadores experientes aprendem uma nova linguagem de programação. Eles não têm dificuldade em compreender conceitos de linguagem, como atribuições, loops, declarações condicionais etc. A sintaxe da linguagem, contudo, tende a se misturar com a sintaxe de linguagens familiares. Portanto, um programador em Ada que aprende Java pode escrever o operador de atribuição como ':=' em vez de '='.

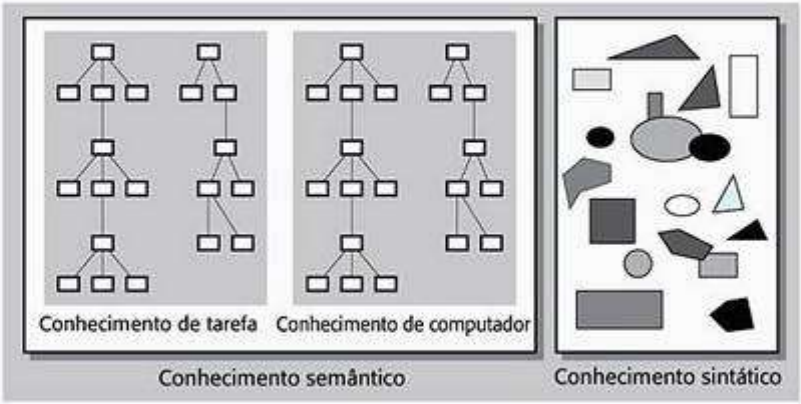
À medida que a compreensão de um conceito se desenvolve, ela fica armazenada na memória como um conhecimento semântico. O conhecimento semântico parece ser armazenado de maneira conceitual abstrata. Os detalhes conceituais podem ser novamente gerados em uma série de diferentes representações concretas (Soloway *et al.*, 1982; Card *et al.*, 1983).

Por exemplo, considere o algoritmo de busca binária, em que uma coleção ordenada é pesquisada para um item em particular. Isso envolve o exame do ponto médio da coleção e o uso do conhecimento do relacionamento de ordenação, a fim de verificar se o item principal está na parte superior ou inferior da coleção. Um programador que compreenda esse algoritmo pode facilmente produzir uma versão em Java, em Ada ou outra linguagem de programação.

Esse modelo explica por que, para muitas pessoas, aprender a programar é uma habilidade que parece chegar de repente, depois de um período de dificuldades. As habilidades de pro-

Figura 22.3

Conhecimento semântico e sintático.



gramação requerem uma compreensão dos conceitos semânticos e uma separação dos conceitos semânticos e sintáticos. Os professores, às vezes, têm dificuldades em compreender os problemas dos estudantes. Eles compreenderam e processaram as informações semânticas com sucesso; assim, só têm de considerar as informações sintáticas. Desse modo, eles podem achar difícil explicar os conceitos semânticos de uma maneira que os programadores novatos compreendam.

22.1.2 Resolução de problemas

Imaginar e escrever um programa é um processo de resolução de problemas. Para desenvolver um sistema de software, é preciso compreender o problema, formular uma estratégia de solução e, depois, traduzir tudo isso em um programa. O primeiro estágio envolve a inserção da declaração do problema na memória de trabalho a partir da memória de curto prazo. Ele é integrado com o conhecimento existente na memória de longo prazo e analisado para elaborar uma solução geral. Finalmente, a solução geral é aperfeiçoada em um programa executável (Figura 22.4).

A resolução de problemas requer geralmente que os conceitos de tarefa e de computador sejam integrados. Fatores organizacionais, como a necessidade de completar uma solução dentro do orçamento, são importantes. Assim, um usuário pode ser um especialista em conceitos de tarefas, um projetista de software pode ser um especialista em conceitos de computador e um gerente pode ser um especialista em processos organizacionais. Durante o processo de engenharia de software, todas essas habilidades podem ter de ser utilizadas e integradas.

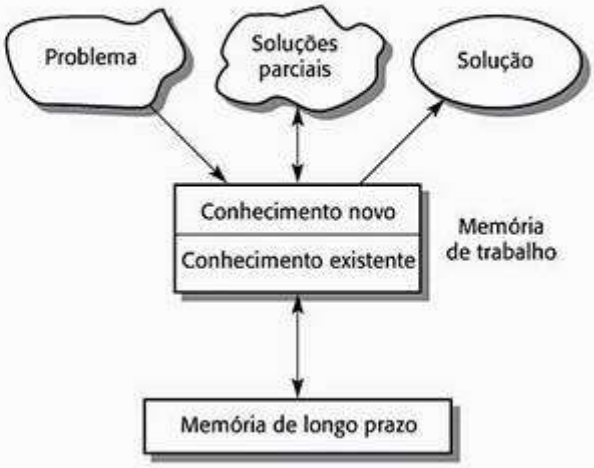
O desenvolvimento da solução (o programa) envolve a construção de um modelo semântico interno do problema e de um modelo correspondente da solução. Quando esse modelo estiver construído, ele pode ser representado em qualquer notação sintática apropriada. O processo de projeto de programa é, por conseguinte, um processo iterativo que envolve diversas etapas:

1. Integrar o conhecimento existente de computador e de tarefa, a fim de criar novo conhecimento e, a partir daí, compreender o problema. Curtis (Curtis *et al.*, 1988) sugere que a experiência em aplicações é particularmente importante nesse estágio.
2. Construir um modelo semântico da solução. Testar isso em relação ao problema e aperfeiçoá-lo até estar satisfatório.
3. Representar o modelo em alguma linguagem de programação ou notação de projeto.

Quando os gerentes precisam tomar decisões sobre quem deve estar envolvido em um projeto de longo prazo, eles devem considerar a habilidade de solucionar problemas em geral e a experiência de domínio, em vez de habilidades específicas de linguagem de programação.

Figura 22.4

Resolução de problema.



Uma vez compreendido um problema, os programadores experientes terão aproximadamente o mesmo grau de dificuldade em escrever um programa, independentemente da linguagem de programação utilizada. As habilidades de linguagem são necessárias e tomam tempo para serem desenvolvidas (particularmente, no caso de linguagens complexas, como C++), mas, considerando minha experiência, é muito mais fácil aprender uma linguagem específica de programação do que desenvolver habilidades de resolução de problemas.

A tradução de um modelo semântico para programa é, mais provavelmente, livre de erros, se a linguagem de programação incluir construções que combinem com as estruturas semânticas de nível inferior, que o programador compreende. Isso varia de indivíduo para indivíduo, mas, provavelmente, corresponde a conceitos, como, atribuições, loops, declarações condicionais, ocultação de informações, objetos, herança etc. Quanto mais próximo o ajuste entre a linguagem de programação e esses conceitos, mais fácil se torna escrever o programa.

Os programas escritos em linguagens de alto nível, como Java, devem, portanto, conter menos erros do que aqueles escritos em código assembly, porque os conceitos semânticos de nível inferior podem ser codificados diretamente como declarações de linguagem. Contudo, poderão surgir problemas, se conceitos funcionais e orientados a objetos forem misturados. Isso pode ser um problema, se uma empresa padronizou um tipo de método para análise (por exemplo, SADT), mas utiliza um processo de projeto e programação orientado a objetos.

O modelo também explica por que a programação estruturada (veja o Capítulo 18) é a melhor estratégia para organizar o controle de programa. Ela tem como base conceitos semânticos, como loops e declarações condicionais. A memória de curto prazo do programador não é sobrecarregada; assim, ele provavelmente cometerá menos erros. Os programas estruturados são mais fáceis de ser compreendidos, porque podem ser lidos de cima para baixo. As abstrações envolvidas em formar aglomerados podem ser feitas seqüencialmente, sem considerar outras partes do programa. A memória de curto prazo pode se dedicar a uma única seção de código. A informação da memória de trabalho sobre outras partes do programa, que interfiram com aquela seção, não precisa ser recuperada.

22.1.3 Motivação

Um dos papéis dos gerentes de projeto é motivar as pessoas que trabalham para eles. Maslow (1954) sugeriu que as pessoas são motivadas pela satisfação de suas necessidades e que suas necessidades são organizadas em uma série de níveis, como mostra a Figura 22.5. O nível inferior dessa hierarquia representa as necessidades de alimentação, sono etc. e a necessidade de elas se sentirem seguras em um ambiente. As necessidades sociais dizem respeito à necessidade de se sentir parte de um grupo social. As necessidades de estima são aquelas que se referem a se sentir respeitado pelos outros. As necessidades de auto-realização se relacionam ao desen-

volvimento pessoal. As prioridades humanas são satisfazer às necessidades de nível mais baixo, como a fome, antes das necessidades mais abstratas de nível mais alto.

As pessoas que trabalham em organizações de desenvolvimento de software em geral não estão famintas ou sedentas e não se sentem fisicamente ameaçadas pelo seu ambiente. Portanto, assegurar as necessidades sociais, de estima e de auto-realização é muito importante do ponto de vista de gerenciamento:

1. Satisfazer às necessidades sociais significa conceder às pessoas tempo para se encontrarem com seus colegas de trabalho e proporcionar os lugares para eles se encontrarem. Os canais de comunicação informais, de fácil uso, como o correio eletrônico, são muito importantes.
2. Para satisfazer às necessidades de estima, é preciso mostrar às pessoas que elas são valorizadas pela organização. O reconhecimento público de realizações é um modo simples, porém eficaz, de se conseguir isso. Obviamente, as pessoas precisam também sentir que elas são remuneradas de acordo com um nível que reflita suas habilidades e sua experiência.
3. Finalmente, para satisfazer às necessidades de auto-realização, é necessário dar às pessoas a responsabilidade pelo seu próprio trabalho, atribuindo-lhes tarefas difíceis (mas não impossíveis) e proporcionando-lhes programas de treinamento em que possam desenvolver suas habilidades.

Em um estudo psicológico sobre motivação, Bass e Dunteman (1963) classificaram os profissionais em três tipos:

1. *Orientados a tarefas*, que são motivados pelo trabalho que fazem. Na engenharia de software, eles são os técnicos motivados pelo desafio intelectual do desenvolvimento de software.
2. *Auto-orientados*, que são principalmente motivados pelo sucesso pessoal e pelo reconhecimento. Eles estão interessados no desenvolvimento de software como um meio de atingir seus próprios objetivos.
3. *Orientados a interações*, que são motivados pela presença e pelas ações dos colegas de trabalho. À medida que o desenvolvimento de software se torna mais centrado no usuário, os indivíduos orientados a interações estão se tornando cada vez mais envolvidos na engenharia de software.

As personalidades orientadas a interações, em geral, apreciam trabalhar como parte de um grupo, enquanto as pessoas orientadas a tarefas e as pessoas auto-orientadas preferem trabalhar sozinhas. As mulheres são mais propensas a serem orientadas a interações do que os homens. Elas são, quase sempre, agentes comunicadores mais eficazes.

A motivação de cada indivíduo é constituída de elementos de cada classe, mas um tipo de motivação é, geralmente, dominante em qualquer momento. Contudo, as personalidades não são estáticas e os indivíduos podem mudar. Por exemplo, as pessoas técnicas que sintam que não estão sendo adequadamente recompensadas podem se tornar auto-orientadas e colocar seus interesses pessoais antes das preocupações técnicas.

O problema com o modelo de motivação de Maslow é que ele assume um ponto de vista exclusivamente pessoal sobre a motivação, não levando em consideração, de maneira adequada, o fato de que as pessoas se sentem, elas mesmas, como parte de uma organização, de um grupo profissional e de alguma cultura. As pessoas não são motivadas apenas por necessidades pessoais, mas também pelos objetivos desses grupos mais amplos. Ser membro de um grupo coeso é altamente motivador para a maioria das pessoas. As pessoas com emprego satisfatório com freqüência gostam de ir para o trabalho porque são motivadas pelo pessoal com que trabalham e pelo trabalho que realizam.

Figura 22.5

Hierarquia das necessidades humanas.

22.2 Trabalho em grupo

A maior parte dos softwares profissionais é desenvolvida por equipes de projeto cujo tamanho varia de duas a várias centenas de pessoas. Contudo, como é impossível que todos, em uma grande equipe, trabalhem juntos de modo eficaz, em um único problema, essas grandes equipes geralmente são divididas em uma série de grupos. Cada grupo é responsável por um subprojeto, que desenvolve algum subsistema. Como regra geral, os grupos de projeto de engenharia de software não devem ter mais do que oito membros. Quando pequenos grupos são utilizados, os problemas de comunicação diminuem. Todo o grupo pode se reunir em torno de uma mesa e os profissionais podem conversar em seus próprios escritórios. Complexas estruturas de comunicação não são necessárias.

Reunir um grupo que trabalhe eficazmente é, por conseguinte, uma tarefa crucial do gerenciamento. Obviamente, é importante que o grupo tenha o equilíbrio certo entre as habilidades técnicas, a experiência e as personalidades individuais. Contudo, grupos bem-sucedidos são mais do que simplesmente um conjunto de indivíduos com o equilíbrio certo de habilidades. Um bom grupo tem um espírito de equipe, de modo que as pessoas envolvidas são motivadas pelo sucesso do grupo, assim como pelos seus próprios objetivos pessoais. Portanto, os gerentes devem promover atividades explícitas de *team building* para ajudar a estabelecer esse espírito essencial de lealdade ao grupo.

Existe uma série de fatores que influenciam o trabalho em grupo:

1. *Composição de grupo* Existe o equilíbrio certo de habilidades, experiência e personalidades na equipe?
2. *Coesão de grupo* O grupo pensa em si como uma equipe, e não como um conjunto de indivíduos que estão trabalhando juntos?
3. *Comunicações de grupo* Os membros do grupo se comunicam eficazmente uns com os outros?
4. *Organização de grupo* A equipe está organizada de tal maneira que todos se sintam valorizados e satisfeitos com seu papel no grupo?

22.2.1 Composição de grupo

Muitos engenheiros de software são motivados principalmente pelo seu trabalho. Os grupos de desenvolvimento de software, portanto, são freqüentemente compostos de pessoas que têm suas próprias idéias sobre como os problemas técnicos devem ser resolvidos. Isso é confirmado por problemas regularmente relatados referentes a padrões de interface ignorados, sistemas que são reprojetados à medida que são codificados, colocação de recursos desne-

cessários em sistemas etc. A seleção dos membros certos para um grupo de projeto é essencial, se esses tipos de problemas tiverem de ser evitados.

Um grupo que tenha personalidades complementares pode trabalhar melhor do que um grupo que tenha sido selecionado considerando-se exclusivamente as habilidades técnicas. Pessoas motivadas pelo trabalho são, muitas vezes, as mais capazes tecnicamente. As pessoas auto-orientadas serão, possivelmente, as melhores no sentido de impulsionar o trabalho adiante, até sua conclusão. As pessoas que são orientadas a interações ajudam na comunicação dentro do grupo. Acho que é particularmente importante ter pessoas orientadas a interações em um grupo, pois elas gostam de conversar e podem detectar possíveis tensões e discordâncias em um estágio inicial. Elas podem ajudar a resolver problemas e diferenças de personalidade, antes que isso tenha um impacto grave sobre o grupo.

Algumas vezes, é impossível selecionar um grupo com personalidades complementares. Nesse caso, o gerente de projeto precisa controlar o grupo, de modo que seus membros não permitam que os objetivos pessoais transcendam os objetivos organizacionais e do grupo. Esse controle é mais fácil de ser realizado se todos os membros do grupo participarem de cada estágio do projeto. As iniciativas individuais são muito prováveis quando os membros do grupo recebem instruções sem estarem cientes do papel que sua tarefa desempenha no projeto geral.

Por exemplo, digamos que a um engenheiro é dado um projeto de programa para codificar e ele observa possíveis melhorias de projeto. Se essas melhorias forem implementadas sem a compreensão do ponto de vista racional do projeto original, elas podem ter implicações adversas em outras partes do sistema. Se todo o grupo estiver envolvido no projeto desde o princípio, eles compreenderão por que as decisões de projeto foram tomadas. Eles podem se identificar com essas decisões, em vez de se opor a elas.

Dentro de um grupo, o líder tem um importante papel. Ele pode ser responsável por fornecer diretriz técnica e administração do projeto. Os líderes de grupos devem manter o acompanhamento do trabalho cotidiano de seu grupo, precisam se certificar de que as pessoas estão trabalhando de modo eficaz e devem trabalhar estreitamente com os gerentes de projeto no planejamento de projeto.

Normalmente, os líderes são nomeados e se reportam ao gerente geral de projeto. Contudo, o líder nomeado pode não ser o líder real do grupo, em se tratando de assuntos técnicos. Os membros do grupo podem procurar outro membro para a liderança, e esse membro pode ser o engenheiro mais competente tecnicamente ou um melhor motivador do que o líder de grupo que foi nomeado.

Algumas vezes, é eficaz separar a liderança técnica da administração de projeto. As pessoas tecnicamente competentes nem sempre são os melhores administradores. Quando elas precisam assumir um papel administrativo, isso reduz seu valor geral para o grupo. É melhor que eles tenham o apoio de administradores que os aliviem das tarefas de administração diária.

Se um líder indesejado for imposto a um grupo, isso, provavelmente, causará tensões. Os membros não respeitarão o líder e podem rejeitar a lealdade ao grupo, em favor de objetivos pessoais. Esse é um problema específico em uma área caracterizada por rápidas mudanças, como a engenharia de software, em que novos membros podem estar mais atualizados e mais bem orientados do que líderes de grupo experientes. Algumas pessoas com experiência podem se ressentir com a imposição de um líder jovem, que traz novas idéias.

22.2.2 Coesão de grupo

Os membros de um grupo coeso pensam nele como mais importante do que seus indivíduos. Os membros de um grupo bem liderado, coeso, são leais ao grupo. Eles se identificam com os objetivos do grupo e com os outros membros e tentam proteger o grupo como uma entidade, em relação à interferência externa. Isso torna o grupo consolidado e capaz de lidar com problemas e situações inesperados. O grupo pode lidar com mudanças, fornecendo apoio e ajuda mutuamente.

As vantagens de um grupo coeso são:

1. *É possível desenvolver um padrão de qualidade para o grupo* Em razão desse padrão ser estabelecido por consenso, existe maior probabilidade de ele ser seguido do que os padrões externos impostos ao grupo.
2. *Os membros do grupo trabalham estreitamente em conjunto* As pessoas do grupo aprendem umas com as outras. As inibições provocadas pela falta de conhecimento são minimizadas, uma vez que o aprendizado mútuo é encorajado.
3. *Os membros do grupo podem conhecer o trabalho uns dos outros* A continuidade pode ser mantida, se um membro deixar o grupo.
4. *A programação sem egoísmos pode ser praticada* Os programas são considerados propriedade do grupo, e não uma propriedade pessoal.

A programação sem egoísmos (Weinberg, 1971) é um estilo de trabalho em grupo, em que projetos, programas e outros documentos são considerados propriedade comum do grupo, e não propriedade do indivíduo que os desenvolveu. Se os engenheiros pensam sobre seu trabalho dessa maneira, tendem a apresentá-lo mais facilmente para inspeção por outros membros do grupo, a aceitar críticas e a trabalhar com o grupo para aperfeiçoar o programa. A coesão do grupo é melhorada, porque todos os membros sentem que têm uma responsabilidade compartilhada em relação ao software.

Além de aprimorar a qualidade de projetos, programas e documentos, a programação sem interferência do ego também melhora a comunicação dentro do grupo, encorajando discussões desinibidas, sem considerações quanto a status, experiência ou gênero. Os membros individuais cooperam ativamente com outros membros do grupo, ao longo do curso do projeto. Isso tudo serve para aproximar os membros do grupo e fazê-los se sentirem parte dele.

A coesão do grupo depende de muitos fatores, inclusive da cultura organizacional e da personalidade dos membros do grupo. Os gerentes podem encorajar a coesão de diversas maneiras. Podem organizar eventos sociais para os membros do grupo e suas famílias. Podem tentar estabelecer um sentimento de identidade de grupo, dando um nome a ele e estabelecendo uma identidade e um território para ele. Podem se envolver em atividades explícitas de formação do grupo, como esportes e jogos.

Contudo, segundo minha experiência, uma das maneiras mais eficazes de promover a coesão é garantir que os membros do grupo sejam tratados como pessoas responsáveis e de confiança, dando-lhes acesso às informações. Freqüentemente, os gerentes sentem que não podem revelar certas informações para todo o grupo. Isso, inevitavelmente, cria um ambiente de desconfiança. A simples troca de informações é uma maneira barata e eficiente de fazer as pessoas se sentirem como parte de um grupo.

Todavia, grupos consolidados e coesos podem, às vezes, sofrer dois tipos de problemas:

1. *Resistência irracional a uma mudança de liderança* Se o líder de um grupo coeso tiver de ser substituído por alguém de fora do grupo, seus membros podem se unir contrariamente ao novo líder. Os membros do grupo podem perder tempo resistindo a mudanças propostas pelo novo líder, com uma conseqüente diminuição da produtividade. Sempre que possível, os novos líderes deverão ser, por conseguinte, indicados pelos membros dos grupos.
2. *Pensamento em grupo* O pensamento em grupo (*groupthink*) (Janis, 1972) é o nome dado a uma situação em que as principais habilidades de membros do grupo são prejudicadas pela lealdade ao grupo. A consideração de possíveis alternativas é substituída pela lealdade às normas e às decisões do grupo. Qualquer proposta aceita pela maioria do grupo pode ser adotada, sem a consideração adequada de possíveis alternativas.

Para evitar o pensamento em grupo, podem ser organizadas sessões formais, em que os membros do grupo são encorajados a criticar decisões. Especialistas externos podem partici-

par para analisar as decisões do grupo. Pessoas que são naturalmente briguentas, questionadoras e desrespeitosas em relação ao *status quo* podem ser indicadas como membros do grupo. Elas atuam como 'advogado do diabo', questionando constantemente as decisões do grupo, forçando assim os outros membros a pensar sobre o assunto e a avaliar suas atividades.

22.2.3 Comunicações de grupo

É essencial que haja uma boa comunicação entre os membros do grupo de desenvolvimento de software. Os membros do grupo devem trocar informações sobre o status de seus trabalhos, as decisões de projeto que foram tomadas e as mudanças que foram necessárias em relação a decisões prévias. A boa comunicação também reforça a coesão do grupo, uma vez que seus membros chegam a compreender as motivações e os pontos positivos e negativos das outras pessoas no grupo.

Alguns fatores importantes que influenciam a eficácia da comunicação são:

1. *Tamanho do grupo* À medida que o grupo aumenta em tamanho, torna-se mais difícil garantir que todos os membros se comuniquem eficazmente uns com os outros. O número de elos de comunicação em uma direção é $n * (n - 1)$, em que n é o tamanho do grupo; portanto, pode-se ver que, com um grupo de sete ou oito pessoas, é bem possível que algumas delas raramente se comuniquem entre si. A diferença de status entre os membros do grupo significa que as comunicações são feitas freqüentemente em uma única direção. Os membros com status superior tendem a dominar a comunicação com os membros de menor status, que, às vezes, relutam em dar início a uma conversa ou em fazer observações críticas.

2. *Estrutura do grupo* As pessoas em grupos informalmente estruturados se comunicam mais efetivamente do que em grupos com uma estrutura formal, hierárquica. Em grupos hierárquicos, as comunicações tendem a fluir para cima e para baixo na hierarquia. As pessoas de mesmo nível podem não conversar entre si, o que é um problema particular em um projeto grande, com vários grupos de desenvolvimento. Se as pessoas que trabalham em diferentes subsistemas somente se comunicarem por meio de seus gerentes, acabarão ocorrendo atrasos e desentendimentos.

3. *Composição do grupo* Se houver pessoas demais no grupo, com os mesmos tipos de personalidade, elas poderão se confrontar, e a comunicação pode ficar inibida. Geralmente, a comunicação é melhor em grupos compostos de homens e mulheres (Marshall e Heslin, 1975) do que em grupos compostos somente de pessoas do mesmo sexo. As mulheres tendem a ser mais orientadas à interação do que os homens, e os membros femininos do grupo podem atuar como controladores e facilitadores da interação no grupo.

4. *O ambiente físico de trabalho do grupo* A organização do local de trabalho é um fator importante para facilitar ou inibir a comunicação. Esse tópico será discutido na Seção 22.3.1.

22.2.4 Organização de grupo

Os pequenos grupos de programação são, geralmente, organizados de maneira bastante informal. O líder do grupo se envolve no desenvolvimento do software com os outros membros. Um líder técnico pode surgir, controlando efetivamente a produção de software. Em um grupo informal, o trabalho a ser realizado é discutido pelo grupo como um todo e as tarefas são distribuídas de acordo com a habilidade e a experiência. O projeto de sistema de alto nível é realizado pelos membros seniores do grupo, mas o projeto de nível inferior é de responsabilidade do membro que é designado para uma determinada tarefa.

Os grupos informais podem ser muito bem-sucedidos, particularmente quando a maioria dos membros do grupo é experiente e competente. O grupo funciona como uma unidade

democrática, tomando decisões por consenso. Psicologicamente, isso melhora o espírito do grupo, com um resultante aumento da coesão e do desempenho. Se um grupo for composto principalmente por membros inexperientes e incompetentes, a informalidade pode ser um obstáculo. Nenhuma autoridade definida existe para orientar o trabalho, provocando uma falta de coordenação entre os membros do grupo e, possivelmente, eventuais falhas no projeto.

Uma variante organizacional interessante da organização democrática do grupo é descrita por Beck, em seu livro sobre *extreme programming* (programação extrema) (Beck, 2000). Nessa abordagem, muitas decisões que são usualmente vistas como decisões de gerenciamento, como as decisões sobre prazos, são repassadas para os membros do grupo. Os programadores trabalham juntos, em pares, para desenvolver o código e assumem a responsabilidade pelos programas que são desenvolvidos. Essa abordagem foi bem-sucedida, segundo relatos, mas, da mesma maneira que o desenvolvimento Cleanroom, discutido no Capítulo 19, suspeito que ela requer pessoal altamente qualificado e motivado, para que seja bem-sucedida.

Como será discutido no Capítulo 23, a habilidade individual tem influência mais significativa na produtividade do programador. Para fazer uso mais eficaz dos programadores altamente habilitados, Baker (1972) e outros (Aron, 1974; Brooks, 1975) sugeriram que as equipes devem ser constituídas em torno de um programador-chefe altamente qualificado. O princípio básico da equipe de um programador-chefe é que um pessoal habilitado e experiente deve ser responsável por todo o desenvolvimento de software. Eles não devem se preocupar com assuntos de rotina e devem ter bom suporte técnico e administrativo para seu trabalho. Sua comunicação com pessoas externas ao grupo deve ser limitada (Figura 22.6).

Os membros-chave de uma equipe de programador-chefe são:

1. Um programador-chefe, que assume toda a responsabilidade pelo projeto, pela programação, pelos testes e pela instalação do sistema.
2. Um programador-reserva experiente, cujo trabalho é dar apoio ao programador-chefe e assumir a responsabilidade pela validação do software;
3. Um bibliotecário, cujo papel é assumir todas as funções administrativas associadas com um projeto, tais como o gerenciamento de configuração, a finalização da documentação etc.

Dependendo do tamanho e do tipo da aplicação, outros especialistas podem ser transferidos de um pool de especialistas para trabalhar temporária ou permanentemente com uma equipe. Podem ser administradores, especialistas em sistemas operacionais ou em linguagens, engenheiros de testes etc.

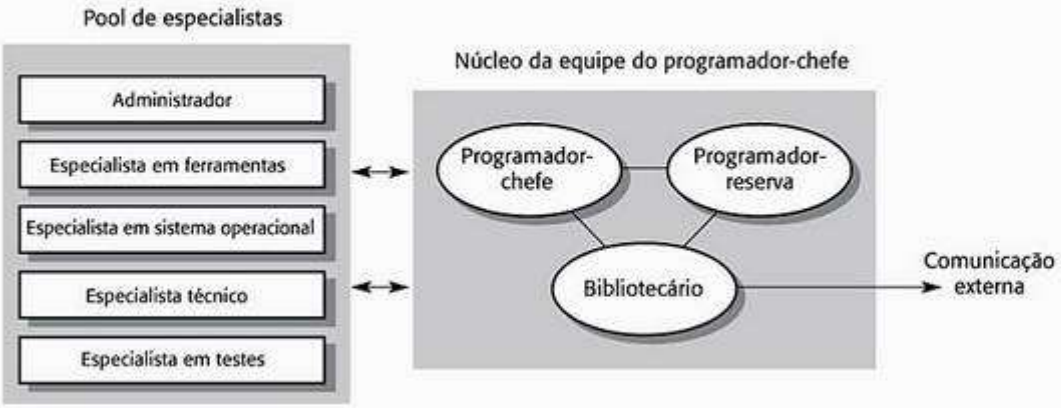
A razão para essa abordagem se origina do fato de que existem imensas diferenças de habilidade de programação entre os engenheiros de software. Os melhores programadores podem ser até cerca de 25 vezes mais produtivos do que os piores programadores. Portanto, tem sentido utilizar os melhores profissionais, do modo mais efetivo, e proporcionar-lhes o melhor apoio possível. Portanto, embora as propostas para uma equipe com programadores-chefe existam há mais de 25 anos, ainda hoje, essa é uma maneira eficaz de organizar um pequeno grupo de desenvolvimento.

Se o pessoal certo estiver disponível, a utilização dessa organização de grupo bem estruturada pode ser bastante eficaz. Contudo, existe uma série de problemas:

1. Não é comum encontrar projetistas e programadores talentosos. A abordagem depende de um excelente programador-chefe e de seu programador-reserva. Se eles cometerem erros, não haverá ninguém para questionar suas decisões. Em um grupo democrático, qualquer um pode questionar decisões e, portanto, é possível descobrir problemas.
2. O programador-chefe assume toda a responsabilidade e pode reivindicar todo o crédito pelo sucesso. Contudo, os membros do grupo podem se ressentir, se seu papel no projeto não for reconhecido. A necessidade de estima desses membros poderá não ser satisfeita, se todo o crédito pelo sucesso for dado ao programador-chefe.

Figura 22.6 ▶

Uma equipe de programador-chefe.



3. Os projetos poderão ser prejudicados se o programador-chefe e seu assistente adoecerem ou deixarem a organização. Os gerentes podem não querer aceitar esse risco.
4. As estruturas organizacionais podem não ser capazes de acomodar a introdução desse tipo de grupo. As grandes empresas podem ter uma estrutura de hierarquia bem desenvolvida. Nomear programadores-chefe externos a essa estrutura pode ser muito difícil. Nas pequenas empresas, pode ser impossível que um deles se dedique inteiramente a uma tarefa.

As equipes de programadores-chefe podem, por conseguinte, ser uma estratégia arriscada para as organizações adotarem. Contudo, podemos aprender com as experiências desse tipo de grupo. Faz sentido dar apoio a pessoas tecnicamente talentosas, com bibliotecários, administradores etc. Suas habilidades podem, portanto, ser utilizadas da melhor maneira. Tornar especialistas disponíveis durante pequenos prazos pode ser mais produtivo do que utilizar programadores com experiências gerais por um período mais longo.

22.3 Seleção e manutenção de pessoal

Uma das funções de um gerente de projeto é escolher o pessoal para trabalhar no projeto. Em casos excepcionais, os gerentes de projeto podem indicar o pessoal mais bem qualificado para o trabalho, independentemente de suas outras responsabilidades ou de considerações quanto a orçamento. Contudo, normalmente, os gerentes de projeto não têm livre escolha de pessoal. Eles podem precisar utilizar quem estiver disponível em uma organização, podem ter de encontrar profissionais muito rapidamente e podem ter um orçamento limitado. Esse orçamento limitado pode restringir o número de engenheiros experientes (dispendiosos) que podem trabalhar no projeto.

Se um gerente tiver alguma opção de pessoal para trabalhar no projeto, os fatores que podem influenciar em sua decisão são mostrados na Figura 22.7. Não há uma maneira de avaliar esses fatores em termos de importância, pois esse aspecto varia, dependendo do domínio da aplicação, do tipo de projeto e das habilidades de outros membros da equipe de projeto.

Geralmente, a decisão sobre quem indicar para um projeto é tomada por meio de três tipos de informações:

1. a informação fornecida pelos candidatos sobre a sua formação e experiência (pelo seu *curriculum vitae*);
2. a informação obtida pela entrevista de candidatos, e
3. as recomendações de outras pessoas que tenham trabalhado com os candidatos.

Figura 22.7 ▶

Fatores considerados na seleção de pessoal.

Fator	Explicação
Experiência no domínio da aplicação	Para um projeto desenvolver um sistema bem-sucedido, os desenvolvedores têm de compreender o domínio da aplicação.
Experiência na plataforma	Isso pode ser importante, se uma programação de baixo nível estiver envolvida. Do contrário, geralmente esse não é um atributo importante.
Experiência na linguagem de programação	Normalmente, isso é importante somente para projetos de curta duração, quando há pouco tempo para aprender uma nova linguagem.
Formação educacional	Esse aspecto pode fornecer um indicador dos fundamentos básicos que o candidato deve saber e de sua capacidade de aprender. Esse fator se torna cada vez mais irrelevante, à medida que os engenheiros ganham experiência em diversos projetos.
Capacidade de comunicação	Isso é importante, devido à necessidade dos membros da equipe de se comunicarem verbalmente e por escrito com outros engenheiros, gerentes e clientes.
Capacidade de adaptação	A capacidade de adaptação pode ser julgada pela observação dos diferentes tipos de experiência que os candidatos tiveram. Esse é um importante atributo, pois indica capacidade de aprender.
Atitude	O pessoal de projeto deve ter uma atitude positiva a respeito de seu trabalho e deve ter boa disposição em aprender novas técnicas. Esse é um importante atributo, mas muitas vezes de difícil avaliação.
Personalidade	Esse é um importante atributo, mas de difícil avaliação. Os candidatos devem ser razoavelmente compatíveis com os outros membros da equipe. Nenhum tipo particular de personalidade é mais ou menos adequado à engenharia de software.

Algumas empresas utilizam vários tipos de testes para avaliar candidatos, incluindo testes de aptidão em programação e testes psicométricos. Os testes psicométricos pretendem traçar um perfil psicológico do candidato, indicando atitudes e sua adequação a certos tipos de tarefas. Alguns gerentes consideram inúteis esses testes; outros acham que eles fornecem informações úteis para a seleção de pessoal. Como foi discutido anteriormente, a habilidade de resolver problemas parece se relacionar à construção de modelos semânticos, o que é um processo de longo prazo. Os testes de aptidão e psicométricos normalmente se baseiam no término rápido das questões. Ainda não foi demonstrado de maneira convincente que existe uma ligação entre a habilidade de resolver problemas e os testes de aptidão.

Os gerentes de projeto, algumas vezes, se defrontam com dificuldades em encontrar pessoas com as habilidades e a experiência adequadas. As equipes têm de ser constituídas de engenheiros relativamente inexperientes. Isso pode levar a problemas, em razão da falta de compreensão do domínio da aplicação ou da tecnologia utilizada no projeto.

Uma das razões disso é que, em algumas organizações, o pessoal tecnicamente capacitado atinge logo uma posição elevada em sua carreira. Para progredir mais, eles devem assumir responsabilidades administrativas. A promoção desse pessoal para a posição gerencial significa que se perdem pessoas com capacitações técnicas úteis. Para evitar essa perda, algumas empresas desenvolveram estruturas paralelas de carreira administrativa e técnica, de iguais valores. O pessoal técnico experiente é recompensado no mesmo nível que os gerentes o são. À medida que a carreira de um engenheiro se desenvolve, ele pode se especializar em atividades técnicas ou administrativas e se movimentar entre elas, sem perda de status ou de salário.

22.3.1 Ambientes de trabalho

O local de trabalho tem efeitos importantes sobre o desempenho das pessoas e em sua satisfação no trabalho. Experimentos psicológicos mostraram que o comportamento é afetado pelo tamanho da sala, pela mobília, pelo equipamento, pela temperatura, umidade, claridade e qualidade de iluminação, pelo nível de ruído e pelo grau de privacidade possível. O comportamento do grupo é afetado pela organização da arquitetura e pelos recursos de telecomunicação. A comunicação dentro de um grupo é afetada pela arquitetura do edifício e pela organização do local de trabalho.

Existe um custo real e significativo na falha em proporcionar boas condições de trabalho. Quando as pessoas estão infelizes quanto a suas condições de trabalho, a rotatividade aumenta. Portanto, é necessário ter maiores gastos com recrutamento e treinamento. Os projetos de software podem sofrer atrasos, devido à falta de pessoal qualificado (DeMarco e Lister, 1999).

Os profissionais de desenvolvimento de software, geralmente, trabalham em amplas áreas abertas de escritórios, algumas vezes, em cubículos, e somente os gerentes seniores têm escritórios individuais. McCue (1978) realizou um estudo que mostrou que a arquitetura de espaço aberto, favorecida por muitas organizações, não era nem popular nem produtiva. Os fatores ambientais mais importantes, identificados nesse estudo, foram:

1. *Privacidade* Os programadores exigem uma área na qual possam se concentrar e trabalhar, sem interrupções.
2. *Percepção do exterior* As pessoas preferem trabalhar com luz natural e com uma vista do ambiente externo.
3. *Personalização* Os indivíduos adotam diferentes práticas de trabalho e têm diferentes opiniões sobre decoração. É importante a capacidade de reorganizar o local de trabalho para adequá-lo às práticas de trabalho e personalizar esse ambiente.

Em suma, as pessoas gostam de escritórios individuais, que possam organizar como desejarem. Nesse caso, existem menos perturbações e interrupções, do que em espaços abertos. Em escritórios abertos, a privacidade e um ambiente de trabalho calmo e silencioso são aspectos negados às pessoas. Esses escritórios são limitados quanto ao modo como as pessoas podem personalizar seu local de trabalho. A concentração pode ser difícil e o desempenho é prejudicado.

Possibilitar escritórios individuais para o pessoal de engenharia de software pode fazer uma diferença significativa na produtividade. DeMarco e Lister (1985) compararam a produtividade de programadores em diferentes tipos de locais de trabalho. Eles constataram que alguns fatores, como um local de trabalho reservado e a capacidade de evitar interrupções, tinham um efeito importante. Os programadores que tinham boas condições de trabalho eram duas vezes mais produtivos do que os programadores igualmente habilitados que tinham de trabalhar em condições mais precárias.

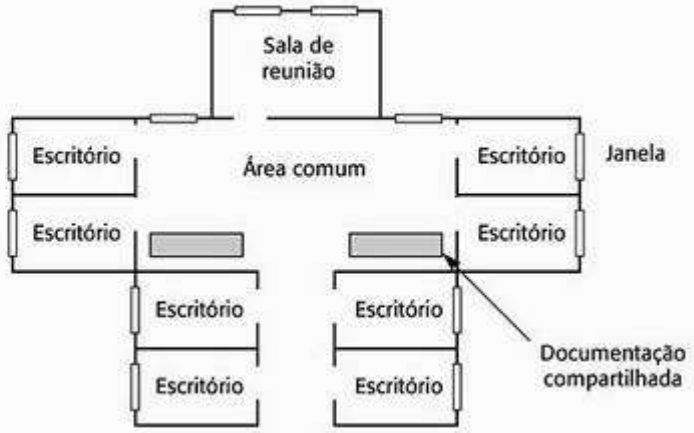
Os grupos de desenvolvimento necessitam de áreas em que todos os membros do grupo possam se reunir e discutir seu projeto, tanto formal quanto informalmente. As salas de reunião devem poder acomodar todo o grupo, mantendo a privacidade. Aparentemente, os requisitos de privacidade individual e de comunicação em grupo são objetivos exclusivos. McCue sugeriu que essas necessidades conflitantes poderiam ser acomodadas pelo agrupamento dos escritórios individuais em torno de salas de reunião maiores (Figura 22.8).

Um modelo semelhante é sugerido por Beck (2000), em sua descrição de um ambiente para *extreme programming*. Contudo, ele sugere manter uma área aberta com todas as atividades de programação ocorrendo na área comum, e cubículos individuais para os membros do grupo, quando estes quiserem trabalhar sozinhos. Obviamente, um requisito importante é fornecer o espaço individual e o espaço comum, de modo que as pessoas possam trabalhar sozinhas ou em grupo, quando for necessário.

Esse tipo de comunicação ajuda as pessoas a resolverem seus problemas e a trocarem informações de maneira informal, mas eficaz. Weinberg (1971) faz um relato interessante de como

Figura 22.8

Organização dos escritórios e da sala de reunião.



uma organização desejava impedir que os programadores perdessem tempo conversando uns com os outros, em volta de uma máquina de café. A máquina foi removida e, em seguida, imediatamente houve um aumento expressivo nos pedidos de assistência de programação formal. Além de conversarem em volta da máquina, as pessoas resolviam os problemas umas das outras. Isso ilustra como as empresas precisam de locais informais de encontro, além das salas formais de reunião.

22.4 Modelo de Maturidade de Capacitação de Pessoal

O Instituto de Engenharia de Software (SEI), nos Estados Unidos, está engajado em um programa de longa duração para a melhoria de processo de software. Parte desse programa é o Modelo de Maturidade de Capacitação (*Capability Maturity Model* – CMM) para processos de software, o qual é discutido no Capítulo 25. Ele aborda as melhores práticas de engenharia de software. Para apoiar esse modelo, eles também propuseram um Modelo de Maturidade de Capacitação de Pessoal (*People Capability Maturity Model* – P-CMM) (Curtis *et al.*, 1995). Isso pode ser utilizado como uma estrutura para melhorar a maneira pela qual uma organização gerencia seu patrimônio humano.

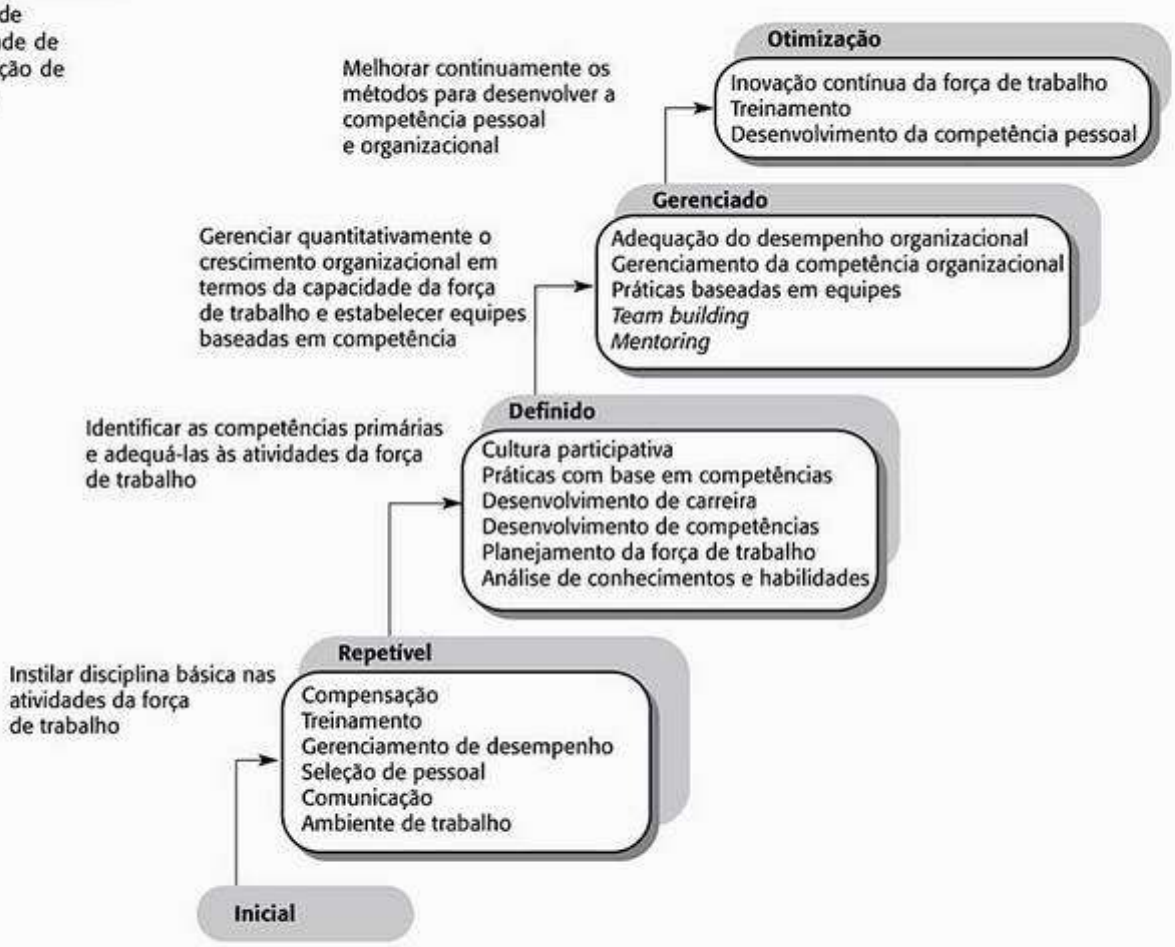
Assim como o CMM, o P-CMM é um modelo de cinco níveis, como mostra a Figura 22.9. Os cinco níveis são:

1. *Inicial* Práticas informais de gestão de pessoal são utilizadas, quando necessário.
2. *Repetível* Estabelecimento de políticas para o desenvolvimento de capacitação de pessoal.
3. *Definido* Padronização das melhores práticas de gerenciamento de pessoal na organização.
4. *Gerenciado* Metas quantitativas para o gerenciamento de pessoal são estabelecidas e introduzidas.
5. *Otimização* Enfoca a melhoria contínua da competência individual e a motivação da força de trabalho.

Curtis *et al.* (1995) definem que os objetivos estratégicos do P-CMM são:

Figura 22.9

Modelo de Maturidade de Capacitação de Pessoal.



1. melhorar a capacidade das organizações de software pelo aumento da capacidade de sua força de trabalho;
2. assegurar que a capacidade de desenvolvimento de software seja um atributo da organização e não apenas de alguns indivíduos;
3. adequar a motivação dos indivíduos à da organização, e
4. reter o patrimônio humano (por exemplo, pessoas com conhecimento e habilidades fundamentais) na organização.

O P-CMM é uma ferramenta prática para a melhoria do gerenciamento de pessoal em uma organização porque fornece uma estrutura para a motivação, o reconhecimento, a padronização e a melhoria das boas práticas. Ele reforça a necessidade de reconhecer a importância das pessoas como indivíduos e de desenvolver suas capacidades. É claro: a aplicação completa desse modelo é muito dispendiosa e, provavelmente, desnecessária para muitas organizações. Contudo, acredito que esse modelo seja um guia útil, que pode conduzir a significativas melhorias na capacitação das empresas para produzir software de alta qualidade.

PONTOS-CHAVE

- O gerenciamento de software se ocupa principalmente de gerenciar pessoal. Os gerentes devem, portanto, ter alguma compreensão dos fatores humanos, de modo que não façam exigências irrealistas de si mesmos ou de seu pessoal.
- As pessoas têm uma memória rápida, de curto prazo, uma memória de trabalho e uma memória de longo prazo. O conhecimento pode ser sintático e arbitrário ou semântico, mais profundo. Resolver problemas envolve integrar o conhecimento semântico, a partir da memória de longo prazo, com as novas informações da memória de curto prazo.
- Os fatores que podem ser utilizados para selecionar pessoal incluem a experiência no domínio da aplicação, a facilidade de adaptação e a personalidade.
- Os grupos de desenvolvimento de software devem ser pequenos e coesos. Os líderes de grupos devem ser tecnicamente competentes e devem ter o apoio administrativo e técnico.
- As comunicações dentro de um grupo são influenciadas por fatores como o status dos membros do grupo, o tamanho dele, sua composição sexual, as personalidades e os canais de comunicação disponíveis.
- O fornecimento de um ambiente de trabalho privativo, com recursos apropriados de computação e comunicação, pode melhorar a produtividade e a satisfação dos programadores.
- O Modelo de Maturidade de Capacitação de Pessoal (P-CMM) oferece um *framework* e recomendação associada para a melhoria da capacidade do pessoal em uma organização e para a melhoria de capacitação da organização, no que se refere a obter benefícios de seu patrimônio humano.

LEITURAS SUGERIDAS

Peopleware: productive projects and teams. 2. ed. Trata-se de uma nova edição do livro clássico sobre a importância de tratar as pessoas adequadamente, quando se gerencia projetos de software. É uma leitura fácil, e esse é um dos poucos livros que reconhecem a importância do local em que as pessoas trabalham. Bastante recomendado. (T. DeMarco e T. Lister, 1999, Dorset House.)

Overview of the people capability maturity model. Apresenta um sumário do modelo do SEI. (B. Curtis, W.E. Hefley, S. Miller, Software Engineering Institute, relatório CMU/SEI-95-MM-01.)

Software management. 4. ed. Esse é um texto tutorial do IEEE, com diversos artigos sobre gerenciamento e motivação de pessoal. (D.J. Reifer, 1993, IEEE Press.)

Working with computers: theory versus outcome. Esse livro é uma coleção de artigos sobre fatores humanos. Esses artigos são escritos principalmente por cientistas conceituados, com interesse no desenvolvimento de software. (G. van der Veer, T.R. Green, J.-M. Hoc e D.M. Murray (eds.), 1988, Academic Press.)

EXERCÍCIOS

22.1 Descreva brevemente a hierarquia da memória humana e explique por que a organização dessa hierarquia sugere que os sistemas orientados a objetos são mais fáceis de compreender do que os sistemas baseados na decomposição funcional.

22.2 Qual é a diferença entre conhecimento sintático e semântico? Pela sua própria experiência, sugira uma série de exemplos de cada um desses tipos de conhecimento.

22.3 Como gerente de treinamento, você é o responsável pelo treinamento inicial da linguagem de programação para as novas admissões de pessoal graduado em sua empresa, cujo ramo é o desenvolvimento de sistemas de defesa aeroespacial. A linguagem de programação principal utilizada é Ada, que foi projetada para a programação de sistemas de defesa. Os estagiários podem ser graduados em ciência de computação, engenheiros e cientistas especializados em física. Alguns, mas não todos os estagiários, têm experiência prévia com programação; nenhum deles tem experiência prévia em Ada. Explique como você estruturaria o treinamento em programação para esse grupo de graduados.

22.4 Que fatores devem ser levados em conta quando se seleciona pessoal para trabalhar em um projeto de desenvolvimento de software?

22.5 Explique por que manter todos os membros de um grupo informados sobre o progresso e as decisões técnicas em um projeto pode melhorar a coesão do grupo.

22.6 Explique o que você entende por pensamento em grupo (*groupthink*). Descreva os perigos desse fenômeno e explique como eles podem ser evitados.

22.7 Você é um gerente de programação a quem foi dada a tarefa de resgatar um projeto, que é fundamental para o sucesso de sua empresa. A gerência sênior lhe deu um orçamento aberto, sem limites, e você pode escolher uma equipe de projeto de até cinco pessoas de quaisquer outros projetos em andamento na empresa. Contudo, uma empresa rival, que trabalha na mesma área, está ativamente recrutando pessoal, e diversas pessoas que trabalhavam para sua empresa se demitiram para se juntar a ela.

Descreva dois modelos de organização de equipes de programação que possam ser utilizados nessa situação e escolha um deles. Apresente as razões de sua escolha e explique por que rejeitou o modelo alternativo.

22.8 Por que os escritórios abertos e comuns são, algumas vezes, menos adequados para o desenvolvimento de software do que os escritórios individuais? Sob que condições você acredita que os ambientes abertos podem ser melhores?

22.9 Por que razão o P-CMM é uma estrutura eficaz para melhorar o gerenciamento de pessoal em uma organização? Sugira como isso pode ser modificado para ser utilizado em pequenas empresas.

22.10 Os gerentes devem ser amigáveis e sociáveis com os membros juniores de seu grupo?

22.11 É ético fornecer respostas que você ache que o interlocutor deseja ouvir, em vez de dizer o que realmente pensa, ao se submeter a testes psicológicos?

23 Estimativa de custo de software

Objetivos

O objetivo deste capítulo é apresentar técnicas de estimativa do custo e do esforço requeridos para a produção de software. Depois de ler este capítulo, você poderá:

- compreender os fundamentos dos custos e dos preços de software e a complexa relação entre eles;
- conhecer as três métricas utilizadas para a avaliação de produtividade de software;
- perceber que uma série de diferentes técnicas devem ser utilizadas para estimar os custos e os prazos de software, e
- compreender os princípios do modelo COCOMO 2 para a estimativa algorítmica de custo.

Sumário

23.1 Produtividade
23.2 Técnicas de estimativa
23.3 Modelagem algorítmica de custos
23.4 Duração de projeto e pessoal

No Capítulo 4, foi apresentado o processo de planejamento de projeto. Durante esse processo, um projeto é dividido em uma série de atividades, que são executadas em paralelo ou em seqüência. A discussão inicial do planejamento de projeto se concentrou nas maneiras de representar essas atividades, suas dependências e a alocação de pessoal para realizar essas tarefas.

Neste capítulo, dedico-me à questão de associar estimativas de esforço e tempo com as atividades de projeto identificadas. Os gerentes de projeto têm de estimar as respostas às seguintes questões:

1. Quanto esforço é requerido para completar uma atividade?
2. Quantos dias são necessários para completar uma atividade?
3. Qual é o custo total de uma atividade?

A estimativa de projeto e o cronograma para sua execução são realizados juntos. Contudo, pode ser necessária alguma estimativa de custo em um estágio prévio do projeto, antes que cronogramas detalhados sejam preparados. Essas estimativas podem ser necessárias para estabelecer um orçamento para o projeto ou para definir um preço do software para o cliente.

Quando o projeto já estiver em andamento, as estimativas devem ser atualizadas regularmente. Isso ajuda no processo de planejamento e permite o uso eficaz de recursos. Se o gasto real for significativamente maior do que as estimativas, então o gerente de projeto tem de tomar alguma iniciativa. Isso pode envolver reivindicar recursos adicionais para o projeto ou modificar o trabalho a ser feito.

Existem três parâmetros envolvidos no cálculo do custo total de um projeto de desenvolvimento de software:

- custos de hardware e software, inclusive manutenção;
- custos de viagens e treinamento, e
- custos relativos ao esforço empregado (o custo do pagamento dos engenheiros de software).

Para a maioria dos projetos, o custo dominante é o que se refere ao esforço empregado. Os computadores que são suficientemente poderosos para o desenvolvimento de software são relativamente baratos. Embora os custos com viagens possam ser significativos quando um projeto é desenvolvido em diferentes locais, eles são relativamente baixos, para a maior parte dos projetos. Além disso, o uso de correio eletrônico, fax e teleconferências pode reduzir as viagens necessárias.

Os custos relativos ao esforço não são simplesmente os custos dos salários dos engenheiros de software envolvidos no projeto. As organizações calculam os custos de esforço em termos dos custos gerais, considerando o custo total de operação da organização, dividindo-o pelo número de pessoas produtivas. Portanto, os custos que se seguem fazem parte do custo total de esforço:

1. custos relativos ao fornecimento de energia;
2. custos com o pessoal de apoio, como contadores, secretárias, técnicos e pessoal de limpeza;
3. custos com operações em rede e comunicações;
4. custos com as instalações centrais, como biblioteca, instalações recreativas etc., e
5. custos com previdência social e benefícios aos funcionários, como pensões e seguro-saúde.

De modo geral, esse fator relacionado às despesas gerais é cerca de duas vezes o salário dos engenheiros de software, dependendo do tamanho da organização e de suas despesas gerais associadas. Portanto, se um engenheiro receber um total de 90 mil dólares por ano, o custo total para a organização será de 180 mil dólares por ano ou 15 mil por mês.

O cálculo dos custos de software deve ser realizado objetivamente, com a intenção de prever com precisão qual será o custo de desenvolvimento do software para o fornecedor. Se o

Figura 23.1 ▶

Fatores que afetam a estimativa do preço do software.

Fator	Descrição
Oportunidade de mercado	Uma organização de desenvolvimento pode cotar um preço baixo porque deseja iniciar em um novo segmento do mercado de software. Aceitando um lucro baixo em um projeto, é possível ter a oportunidade de obter melhores lucros, posteriormente. A experiência obtida pode permitir o desenvolvimento de novos projetos.
Incerteza da estimativa de custo	Se uma organização estiver insegura sobre sua estimativa de custos, ela pode aumentar seu preço bem acima de seu lucro normal, justificando alguma despesa eventual.
Condições contratuais	Um cliente pode querer permitir que o desenvolvedor mantenha a propriedade sobre o código-fonte e o reutilize em outros projetos. O preço cobrado pode então ser menor do que se o código-fonte fosse entregue ao cliente.
Volatilidade dos requisitos	Se existe a probabilidade de que os requisitos sejam modificados, uma organização pode baixar seu preço para ganhar um contrato. Depois que o contrato for efetivado, altos preços podem ser cobrados por mudanças nos requisitos.
Saúde financeira	Desenvolvedores em dificuldades financeiras podem reduzir seu preço para conseguir um contrato. É melhor obter um lucro menor do que o normal ou mesmo apenas equilibrar o lucro e a despesa do que perder o contrato.

custo do projeto tiver sido calculado como parte de uma proposta para um cliente, então será preciso tomar uma decisão sobre o preço cotado para o cliente. Classicamente, o preço não é nada mais do que o custo mais o lucro. Contudo, a relação entre o custo do projeto e o preço para o cliente, geralmente, não é tão simples.

A estimativa de preço do software precisa levar em conta considerações mais amplas sobre questões organizacionais, econômicas, políticas e de negócios. Os fatores que podem ser levados em conta são mostrados na Figura 23.1. Portanto, pode não haver uma relação simples entre o preço para o cliente do software e os custos de desenvolvimento. Em razão das considerações organizacionais envolvidas, a estimativa de preço geralmente envolve a gerência sênior da organização, além dos gerentes de projeto de software.

23.1 Produtividade

A produtividade em um sistema de manufatura pode ser medida pela contagem do número de unidades produzidas, dividindo-se o resultado pelo número de pessoa-hora requerido para produzir essas unidades. Contudo, para qualquer problema de software, existem muitas soluções diferentes, as quais têm atributos diferentes. Uma solução pode ser executada mais eficientemente, enquanto outra pode ser mais legível e de mais fácil manutenção. Quando soluções com diferentes atributos são produzidas, comparar suas taxas de produção não é realmente importante.

Contudo, a produtividade de engenheiros no processo de desenvolvimento de software pode ser estimada pelos gerentes. Essas estimativas podem ser necessárias para a estimativa do projeto e para avaliar se as melhorias de processo ou de tecnologia são eficazes. Essas estimativas de produtividade geralmente se baseiam em medir alguns atributos do software e dividir o resultado pelo esforço total exigido para o desenvolvimento. Há dois tipos de medidas que têm sido utilizadas:

1. *Medidas relacionadas a tamanho* Essas medidas são relacionadas ao tamanho de alguma saída de uma atividade. Dessas, a mais comum é o número de linhas de código-fonte

fornecidas. Outras medidas que podem ser utilizadas são o número de instruções de código-objeto entregues ou o número de páginas de documentação do sistema.

2. *Medidas relacionadas a funções* Essas medidas são relacionadas à funcionalidade geral do software entregue. A produtividade é expressa em termos da funcionalidade útil produzida em um tempo determinado. Os pontos de função e os pontos de objeto são as melhores medidas conhecidas desse tipo.

O número de linhas de código-fonte por programador-mês é uma medida amplamente utilizada na medição de produtividade. Esse número é calculado contando-se o número total de linhas de código-fonte entregues. A contagem é dividida pelo tempo total, em termos de programadores/mês, necessário para completar o projeto. Esse tempo, portanto, inclui o tempo requerido para análise e projeto, codificação, testes e documentação.

Essa abordagem foi inicialmente desenvolvida quando a maioria das programações eram feitas em FORTRAN, linguagem assembly ou COBOL. Nessa época, os programas eram registrados em cartões, com uma declaração em cada cartão. O número de linhas de código era fácil de ser calculado; ele correspondia ao número de cartões na caixa de programa. Contudo, programas em linguagens como Java ou C++ consistem em declarações, comandos executáveis e comentários. Eles podem incluir instruções do tipo macro, que se expandem em diversas linhas de código. É possível que haja mais de uma declaração por linha. Portanto, não existe uma relação simples entre as declarações de programa e as linhas em uma listagem.

Algumas técnicas de contagem de linhas consideram somente as declarações executáveis; outras contam os comandos executáveis e as declarações de dados; algumas contam cada linha escrita no programa, independentemente do que esteja naquela linha. Foram propostos padrões para a contagem de linhas em diferentes linguagens (Park, 1992), mas eles não são amplamente conhecidos. As comparações de produtividade entre as organizações são impossíveis, a menos que cada organização utilize o mesmo método para a contagem de linhas de código.

A comparação de produtividade por meio de diferentes linguagens de programação pode também fornecer impressões enganosas da produtividade do programador. Quanto mais expressiva for a linguagem de programação, menor será a produtividade aparente. Essa anomalia surge porque todas as atividades de desenvolvimento de software são consideradas em conjunto quando se calcula a produtividade, mas a medida utilizada somente se aplica ao processo de programação.

Por exemplo, considere um sistema que pode ser codificado em 5 000 linhas de código assembly ou 1 500 linhas de código de linguagem de alto nível. O tempo de desenvolvimento para as várias fases é mostrado na Figura 23.2. O programador em assembly tem uma produtividade de 714 linhas/mês, enquanto o programador de uma linguagem de alto nível tem menos do que a metade dessa produtividade: 300 linhas/mês. Mesmo assim, os custos de desenvolvimento para o sistema de linguagem de alto nível são mais baixos e ele é produzido em menos tempo.

Uma alternativa para utilizar o tamanho do código como o atributo do produto estimado é utilizar alguma medida da funcionalidade do código. Isso evita a anomalia mostrada anteriormente, uma vez que a funcionalidade é independente da linguagem de implementação. MacDonell (1994) descreve brevemente e compara várias medidas diferentes baseadas em função.

A mais conhecida dessas medidas é a contagem de pontos de função. Esse método foi proposto por Albrecht (1979) e aperfeiçoado por Albrecht e Gaffney (1983). Os pontos de função são independentes de linguagem; assim, a produtividade em diferentes linguagens de programação pode ser comparada. A produtividade é expressa em pontos de função produzidos por pessoa-mês. Os pontos de função são mais apropriados aos sistemas de processamento de dados dominados por operações de entrada e saída. Um ponto de função não é uma característica única, mas uma combinação de características de programa. O número

Figura 23.2 ▶

Tempo de desenvolvimento de sistema.

	Análise	Projeto	Codificação	Testes	Documentação
Código assembly	3 semanas	5 semanas	8 semanas	10 semanas	2 semanas
Linguagem de alto nível	3 semanas	5 semanas	8 semanas	6 semanas	2 semanas

	Tamanho	Esforço	Produtividade
Código assembly	5 000 linhas	28 semanas	714 linhas/mês
Linguagem de alto nível	1 500 linhas	20 semanas	300 linhas/mês

total de pontos de função em um programa é calculado medindo-se ou estimando-se as seguintes características do programa:

- entradas e saídas externas;
- interações com os usuários;
- interfaces externas, e
- arquivos utilizados pelo sistema.

Cada uma dessas características é individualmente avaliada em termos de complexidade e recebe um valor ponderado, que varia de 3, para entradas externas simples, a 15, para arquivos internos complexos. Podem ser utilizados os valores ponderados propostos por Albrecht ou os valores baseados na experiência local.

A contagem de pontos de função não ajustada (*unadjusted function-point count* – UFC) é computada multiplicando-se cada contagem bruta pelo peso estimado e somando-se todos os valores.

$$UFC = \sum (\text{número de elementos de determinado tipo}) \times (\text{peso})$$

A contagem inicial de pontos de função é, em seguida, modificada por fatores cujo valor tem como base a complexidade geral do projeto. Isso leva em conta o grau de processamento distribuído, a quantidade de reuso, o desempenho etc. A contagem de pontos de função não ajustada é multiplicada pelos fatores de complexidade do projeto, a fim de produzir uma contagem final de pontos de função.

Symons (1988) observa que a natureza subjetiva das estimativas de complexidade significa que a contagem dos pontos de função em um programa depende de quem faz a estimativa. Diferentes pessoas têm diferentes noções de complexidade. Existem grandes variações na contagem de pontos de função, dependendo do julgamento de quem faz a estimativa. Por essa razão, há diferentes opiniões sobre o valor de pontos de função (Furey e Kitchenham, 1997). Contudo, os usuários argumentam que, a despeito de suas imperfeições, elas são eficazes em situações práticas (Kemerer, 1993).

Os pontos de objeto (Banker *et al.*, 1992) são uma alternativa aos pontos de função, quando as linguagens 4GL ou linguagens comparáveis são utilizadas para o desenvolvimento de software. Os pontos de objeto são utilizados no modelo de estimativa COCOMO 2, discutido posteriormente neste capítulo. Os pontos de objeto não são classes de objeto que podem ser produzidas quando uma abordagem orientada a objetos é adotada para o desenvolvimento de software. Em vez disso, o número de pontos de objeto em um programa é uma estimativa ponderada dos seguintes fatores:

1. O número de telas separadas que são exibidas. As telas simples contam como 1 ponto de objeto, as telas moderadamente complexas, como 2, e as telas muito complexas, como 3 pontos de objeto.

2. O número de relatórios que são produzidos. Para relatórios simples contam-se 2 pontos de objeto, para relatórios moderadamente complexos contam-se 5 e para relatórios que são provavelmente de difícil produção contam-se 8 pontos de objeto.
3. O número de módulos em 3GL, que devem ser desenvolvidos para suplementar o código 4GL. Cada módulo 3GL conta como 10 pontos de objeto.

A vantagem de utilizar os pontos de objeto, em vez dos pontos de função, é que eles são mais fáceis de estimar a partir de uma especificação de software de alto nível. Os pontos de objeto são apenas relativos a telas, relatórios e módulos 3GL.

Se pontos de função ou pontos de objeto são utilizados, eles podem ser estimados em um estágio inicial do processo de desenvolvimento. As estimativas desses parâmetros podem ser feitas assim que as interações externas do sistema tenham sido projetadas. Nesse estágio, é muito difícil produzir uma estimativa precisa do tamanho de um programa em linhas de código-fonte. Na verdade, a linguagem de programação a ser utilizada talvez ainda nem mesmo tenha sido decidida. As estimativas iniciais são essenciais quando se utilizam os modelos de estimativa algorítmica de custo, que serão discutidos posteriormente neste capítulo.

As contagens de pontos de função podem ser utilizadas em conjunto com as de estimativas de linhas de código. O número de pontos de função é utilizado para estimar o tamanho final do código. Por meio da análise histórica de dados, pode ser estimado o número médio de linhas de código, AVC, em uma linguagem particular, exigida para implementar um ponto de função. O tamanho estimado do código para uma nova aplicação é calculado como se segue:

Tamanho do código = AVC × Número de pontos de função.

Os valores de AVC variam de 200–300 LOC/FP em linguagem assembly a 2–4 LOC/FP para uma linguagem de quarta geração.

A produtividade de engenheiros individuais que trabalham em uma organização é afetada por uma série de fatores. Alguns dos fatores mais importantes estão resumidos na Figura 23.3. Contudo, as diferenças individuais quanto à habilidade são mais significativas do que qualquer um desses fatores. Em uma avaliação preliminar de produtividade, Sackman *et al.* (1968) constataram que alguns programadores eram, pelo menos, 10 vezes mais produtivos do que outros. Acredito que essa situação ainda seja verdadeira. As grandes equipes, provavelmente, apresentam uma mistura de capacitações e, assim, têm uma produtividade 'média'. Nas pequenas equipes, contudo, a produtividade geral depende, em grande parte, das aptidões e habilidades individuais.

Não existe algo como um valor 'médio' para a produtividade que se aplique aos domínios de aplicação e às organizações. Para os sistemas integrados grandes e complexos, a produtividade pode ser tão pequena quanto 30 linhas/programador-mês. Para os sistemas de aplicação bem compreendidos e diretos, ela pode ser tão alta quanto 900 linhas/mês. Quando se faz a medição em termos de pontos de objeto, Boehm *et al.* (1995) sugerem que a produtividade varia de 4 pontos de objeto por mês a 50 pontos, dependendo do suporte técnico a ferramentas e da capacitação do desenvolvedor.

O problema com as medidas expressas em volume/tempo é que elas não levam em conta as características não funcionais do software, como confiabilidade, facilidade de manutenção etc. Elas induzem a pensar que mais sempre significa melhor. Beck (2000) levanta uma excelente observação. Para ele, se temos uma abordagem baseada em melhoria contínua e na simplificação de código, então a contagem de linhas de código não significa muito.

Essas medidas também não levam em conta a possibilidade de reutilizar o software produzido. O que realmente desejamos estimar é o custo de derivar um determinado sistema com particular funcionalidade, qualidade, desempenho, facilidade de manutenção etc. Isso está relacionado apenas indiretamente com medidas tangíveis, como o tamanho do sistema.

Figura 23.3 ▶

Fatores que afetam a produtividade da engenharia de software.

Fator	Descrição
Experiência no domínio da aplicação	O conhecimento do domínio da aplicação é essencial para o desenvolvimento efetivo de software. Os engenheiros que já compreendem um domínio são, provavelmente, os mais produtivos.
Qualidade do processo	O processo de desenvolvimento utilizado pode ter um efeito significativo na produtividade. Esse tópico será abordado no Capítulo 25.
Tamanho do projeto	Quanto maior o projeto, mais tempo é necessário para a comunicação da equipe. Existe menos tempo disponível para o desenvolvimento e, assim, a produtividade individual é reduzida.
Suporte à tecnologia	Um bom suporte à tecnologia, como as ferramentas CASE, os sistemas de gerenciamento de configuração etc., pode melhorar a produtividade.
Ambiente de trabalho	Como foi discutido no Capítulo 22, um ambiente de trabalho tranqüilo, com áreas de trabalho privativas, contribui para a melhoria da produtividade.

Isso se torna um problema em particular, se os gerentes utilizam as medidas de produtividade para julgar as capacitações do pessoal. Nessas situações, os engenheiros podem comprometer a qualidade para se tornarem 'mais produtivos'. Esse pode ser o caso em que o programador 'menos produtivo' produz códigos mais confiáveis, que sejam mais fáceis de compreender e mais baratos em termos de manutenção. As medidas de produtividade têm, portanto, de ser utilizadas apenas como um guia; elas não devem ser utilizadas sem uma análise cuidadosa.

23.2 Técnicas de estimativa

Não existe uma maneira simples de fazer uma estimativa precisa do esforço necessário para desenvolver um sistema de software. As estimativas iniciais talvez precisem ser feitas com base em uma definição de alto nível dos requisitos de usuário. É possível que o software tenha de ser executado em computadores que não sejam familiares ou que seja utilizada uma tecnologia de desenvolvimento nova. As pessoas envolvidas no projeto, e também suas habilidades, provavelmente não serão conhecidas. Todos esses fatores significam que é difícil produzir uma estimativa exata dos custos de desenvolvimento do sistema em um estágio muito inicial do projeto.

Além disso, existe uma dificuldade fundamental em avaliar a precisão de diferentes abordagens para as técnicas de estimativa de custos. As estimativas de custos de projeto são, freqüentemente, auto-suficientes. A estimativa é utilizada para definir o orçamento de projeto, e o produto é ajustado, de modo que o valor do orçamento seja cumprido. Não conheço relatórios de experimentos controlados com custos de projeto em que os custos estimados não tenham sido utilizados para influenciar o experimento. Um experimento controlado não revelaria a estimativa de custos para o gerente de projeto. Os custos reais seriam, então, comparados com os custos estimados de projeto.

Não obstante, as organizações precisam fazer estimativas de esforço e custo de software. Para fazer isso, uma ou mais das técnicas descritas na Figura 23.4 podem ser utilizadas (Boehm, 1981).

Figura 23.4 ▶

Técnicas de estimativa de custos.

Técnica	Descrição
Modelagem algorítmica de custo	É desenvolvido um modelo, utilizando-se as informações sobre o custo histórico, que relacione alguma métrica de software (geralmente, seu tamanho) ao custo do projeto. É feita uma estimativa dessa métrica, e o modelo prevê o esforço requerido.
Julgamento de um especialista	São consultados diversos especialistas nas técnicas de desenvolvimento do software proposto e no domínio da aplicação. Cada um deles estima o custo do projeto. Essas estimativas são comparadas e discutidas. O processo de estimativa prossegue iterativamente até que se obtenha uma estimativa em comum.
Estimativa por analogia	Essa técnica é aplicável quando outros projetos, no mesmo domínio de aplicação, tenham sido concluídos. O custo de um novo projeto é estimado por analogia com esses projetos terminados. Myers (1989) dá uma descrição muito clara dessa abordagem.
Lei de Parkinson	A Lei de Parkinson define que o trabalho se amplia para preencher o tempo disponível. O custo é determinado pelos recursos disponíveis, e não pela avaliação objetiva. Se o software tiver de ser entregue em 12 meses e cinco pessoas estiverem disponíveis, o esforço requerido é estimado em 60 homens-mês.
Preço definido para ganhar	O custo do software é estimado para ser o que o cliente tiver disponível para gastar no projeto. O esforço de estimativa depende do orçamento do cliente, e não da funcionalidade do software.

Hihn e Habib-agahi (1991) descrevem um experimento em que eles pediram a gerentes que estimassem o tamanho de um sistema de software a ser desenvolvido e o esforço necessário. Os gerentes utilizaram o julgamento e a estimativa de especialistas, por analogia. Constatou-se que eles eram razoavelmente precisos para estimar o esforço exigido, mas suas estimativas de tamanho de código foram muito menos exatas. Isso significa que as estimativas de custo baseadas em uma estimativa de tamanho de código eram também imprecisas.

Essas abordagens sobre a estimativa de custos podem ser manejadas, utilizando-se uma abordagem top-down ou bottom-up. Uma abordagem top-down se inicia no nível do sistema. O responsável pela estimativa começa examinando a funcionalidade geral do produto e como essa funcionalidade é fornecida por subfunções que interagem entre si. Os custos de atividades no nível de sistema, como integração, gerenciamento de configuração e documentação, são levados em conta.

A abordagem bottom-up, por outro lado, começa no nível do componente. O sistema é decomposto em componentes, e é calculado o esforço requerido para desenvolver cada um desses componentes. Em seguida, esses custos são adicionados, para definir o esforço requerido para o desenvolvimento de todo o sistema.

As desvantagens da abordagem top-down são as vantagens da abordagem bottom-up e vice-versa. A estimativa top-down pode subestimar os custos da resolução de problemas técnicos difíceis, associados com componentes específicos, como interfaces para hardware não padronizado. Não existe uma justificativa detalhada para a estimativa produzida. Por outro lado, a estimativa bottom-up produz essa justificativa e considera cada componente. Contudo, essa abordagem, provavelmente, poderá subestimar os custos das atividades do sistema, como a integração. A estimativa bottom-up é também mais dispendiosa. Deve haver um projeto inicial de sistema para identificar os componentes a serem custeados.

Cada técnica de estimativa tem seus próprios pontos positivos e negativos. No caso de grandes projetos, é necessário utilizar várias técnicas de estimativa de custos e comparar seus resultados. Se elas predizem custos radicalmente diferentes, isso sugere que não se tem informações suficientes para fazer a estimativa. É necessário obter mais informações e repetir o processo de estimativa de custos, até que as estimativas sejam convergentes.

Essas técnicas de estimativa são aplicáveis quando um documento de requisitos do sistema tiver sido produzido. Isso deve definir todos os usuários e os requisitos do sistema. Portanto, é possível fazer uma estimativa razoável da extensão da funcionalidade do sistema a ser desenvolvido. Em geral, os projetos de engenharia de grandes sistemas, normalmente, terão esse documento de requisitos.

Contudo, em muitos casos, os custos de muitos projetos devem ser estimados utilizando-se somente um esboço geral dos requisitos do usuário para o sistema. Isso significa que os responsáveis pelas estimativas têm muito poucas informações com que trabalhar. A análise e especificação de requisitos são tarefas dispendiosas, e os gerentes em uma empresa podem precisar de uma estimativa inicial de custos para o sistema, antes de terem um orçamento aprovado para esse processo.

Sob essas circunstâncias, o 'preço definido para ganhar' é uma estratégia comumente utilizada. A noção de 'preço definido para ganhar' pode parecer antiética e inadequada aos negócios. Contudo, essa abordagem tem algumas vantagens. Um custo de projeto é combinado com base em uma proposta geral. Em seguida, as negociações são desenvolvidas com o cliente, a fim de se estabelecer a especificação detalhada de projeto. Essa especificação é restrita pelo custo especificado. O comprador e o vendedor precisam entrar em acordo sobre qual é a funcionalidade aceitável para o sistema. O fator fixado, em muitos projetos, não são os requisitos de projeto, mas os custos relativos. Os requisitos podem ser modificados, de modo que o custo não pode ser excedido.

Ao estimar os custos de software, os gerentes precisam levar em conta que há muitas diferenças importantes entre os projetos anteriores e futuros. Uma série de novos métodos e novas técnicas de desenvolvimento foi introduzida nos últimos dez anos. Muitos gerentes têm pouca experiência nessas técnicas ou o conhecimento de como elas afetam os custos de projeto. Dentre os exemplos das mudanças que podem afetar as estimativas baseadas na experiência destacam-se:

- o desenvolvimento orientado a objetos, em vez do desenvolvimento orientado a funções;
- os sistemas cliente-servidor, em vez de sistemas baseados em mainframes;
- o uso de componentes de software 'de prateleira', em vez do desenvolvimento de componentes;
- o desenvolvimento para e com reuso, em vez de novo desenvolvimento de todas as partes de um sistema, e
- o uso de ferramentas CASE e geradores de programas, em vez do desenvolvimento de software sem suporte.

Todos esses fatores tornam mais difícil para os gerentes produzir estimativas precisas dos custos de produção de software. Suas experiências prévias podem não ser importantes para ajudá-los a estimar os custos de projetos de software.

23.3 Modelagem algorítmica de custos

A abordagem mais sistemática, embora não necessariamente a mais precisa, para a estimativa de software é a estimativa algorítmica de custos. Um modelo algorítmico de custos pode ser construído, analisando-se os custos e os atributos de projetos concluídos. Uma fórmula matemática é utilizada para prever os custos baseados em estimativas sobre o tamanho de projeto, o número de programadores e outros fatores relacionados ao processo e ao produto. Kitchenham (1990b) descreve 13 modelos algorítmicos de custos, que foram desenvolvidos a partir de observações empíricas.

A maioria dos modelos algorítmicos de estimativa tem um componente exponencial. Isso reflete o fato de que os custos, normalmente, não aumentam de maneira linear com o tama-

nho do projeto. À medida que o tamanho do projeto aumenta, custos extras são gerados, devido às despesas adicionais com a comunicação em equipes maiores, com o gerenciamento de uma configuração mais complexa, com a maior dificuldade de integração do sistema etc. Esses custos podem também incluir multiplicadores, refletindo os atributos do produto em desenvolvimento, a plataforma de desenvolvimento, o processo e os desenvolvedores de software.

Em sua forma mais geral, uma estimativa algorítmica de custos, para o custo de software, pode ser expressa como:

$$\text{Esforço} = A \times \text{Tamanho}^B \times M.$$

A é um fator constante, que depende de práticas organizacionais locais e do tipo de software desenvolvido. O Tamanho pode ser uma avaliação do tamanho do código do software ou uma estimativa de funcionalidade, expressa em pontos de função ou de objeto. O valor do expoente B, geralmente, fica entre 1 e 1,5. Ele reflete o esforço desproporcional exigido para grandes projetos. M é um multiplicador constituído pela combinação de diferentes atributos de processo, produto e desenvolvimento.

Todos os modelos algorítmicos sofrem as mesmas dificuldades básicas:

1. Freqüentemente, é difícil estimar o Tamanho no estágio inicial de um projeto, quando apenas uma especificação está disponível. Estimativas por pontos de função e pontos de objeto são mais fáceis de serem produzidas do que as estimativas de tamanho de código, mas podem ainda não ser exatas.
2. As estimativas dos fatores que contribuem para B e M são subjetivas. As estimativas variam de uma pessoa para outra, dependendo de seu conhecimento e de sua experiência.

O número de linhas de código-fonte no sistema concluído é a medida básica utilizada, na maioria dos modelos algorítmicos de custos. A estimativa de tamanho pode envolver a estimativa por analogia com outros projetos, a estimativa pela conversão de pontos de função para tamanho de código, a estimativa pela gradação dos tamanhos de componentes do sistema e utilizando-se um componente de referência conhecido, para estimar o tamanho do componente, ou pode simplesmente ser uma questão de julgamento da engenharia.

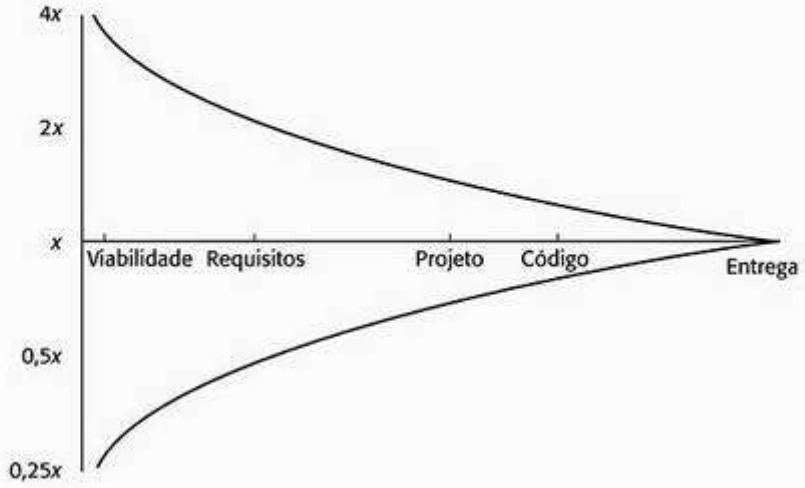
As decisões de projeto, que podem não ser conhecidas quando as estimativas iniciais são feitas, influenciam o tamanho final do sistema. Por exemplo, uma aplicação que exija o gerenciamento complexo de dados pode utilizar um banco de dados comercial ou implementar seu próprio gerenciador de dados. Se um banco de dados comercial for utilizado, o tamanho do código será menor. A linguagem de programação utilizada também é importante. Uma linguagem como Java pode significar que mais linhas de código são necessárias do que se (digamos) utilizarmos a linguagem C. Contudo, esse código extra permite maior verificação do tempo de compilação; assim, os custos de validação, possivelmente, serão reduzidos. Como se deve levar isso em consideração? Além disso, o extenso reuso no processo de desenvolvimento de software precisa ser estimado, e o tamanho estimado deve ser ajustado para levar isso em consideração.

Se modelos algorítmicos forem utilizados para os custos de projeto, eles deverão ser ajustados, utilizando-se dados para o tipo de projeto que está sendo estimado, e suas saídas deverão ser cuidadosamente interpretadas. O responsável pela estimativa deve desenvolver uma série de estimativas (a pior, a esperada e a melhor), e não uma só. A fórmula de custos deve ser aplicada a todas elas. Provavelmente, os erros em estimativas iniciais serão importantes. As estimativas têm melhor probabilidade de ser precisas quando o produto for bem compreendido, o modelo tiver sido ajustado para a organização que o utiliza e as opções de linguagem e hardware forem predefinidas.

A exatidão das estimativas produzidas por um modelo algorítmico depende das informações disponíveis sobre o sistema. À medida que o processo de software progride, mais informações se tornam disponíveis e, assim, as estimativas se tornam cada vez mais pre-

Figura 23.5

Incerteza da estimativa.



cisas. Se a estimativa inicial de esforço requerido for de *x* meses de esforço, essa variação pode ser de 0,25*x* a 4*x*, quando o sistema é inicialmente proposto. Isso se estreita durante o processo de desenvolvimento, como mostra a Figura 23.5. Essa figura, adaptada a partir de um trabalho de Boehm (Boehm *et al.*, 1995), reflete a experiência de um grande número de projetos de desenvolvimento de software.

23.3.1 O modelo COCOMO

Existe uma série de modelos algorítmicos que foram propostos como base para estimar o esforço, os prazos e os custos de um projeto de software. Eles são conceitualmente semelhantes, mas utilizam diferentes valores de parâmetros. O modelo específico discutido aqui é denominado COCOMO (*constructive cost model* – modelo de custo construtivo), que é um modelo empírico. Ele foi derivado pela coleta de dados a partir de um grande número de projetos de software e, em seguida, pela análise desses dados para descobrir fórmulas que fossem mais adequadas às observações. Escolhi o COCOMO pelas seguintes razões:

1. Ele é bem documentado, de domínio público, e é compatível com ferramentas comerciais e de domínio público.
2. Ele tem sido amplamente utilizado e avaliado.
3. Tem uma longa linhagem, desde sua primeira versão, lançada em 1981 (Boehm, 1981), seguindo um aperfeiçoamento ajustado ao desenvolvimento de software em Ada (Boehm e Royce, 1989), até sua mais recente versão, publicada em 1995 (Boehm *et al.*, 1995).

A primeira versão do modelo COCOMO (atualmente, conhecido como COCOMO 81) possuía três níveis, e cada nível refletia os detalhes da análise da estimativa de custos. O primeiro nível (básico) fornecia uma estimativa inicial, preliminar; o segundo nível modificava essa condição, utilizando uma série de multiplicadores de projeto e processo, e o nível mais detalhado produzia estimativas para diferentes fases do projeto. A Figura 23.6 mostra a fórmula básica do COCOMO para diferentes tipos de projetos. O multiplicador M é similar ao discutido a seguir para o COCOMO 2.

O COCOMO 81 supunha que o software seria desenvolvido de acordo com um processo em cascata e que a maioria dos softwares seria desenvolvida 'a partir do zero'. Contudo, ocorreram mudanças radicais no desenvolvimento de software desde que a versão inicial foi proposta. O software pode ser criado mediante a montagem de componentes reutilizáveis, vinculando-os por meio de alguma linguagem de *scripting*. A prototipação e o desenvolvimento incremental são

Complexidade do projeto	Fórmula	Descrição
Simples	PM* = 2,4 $(KDSI)^{1,05} \times M$	Aplicações bem compreendidas, desenvolvidas por pequenas equipes.
Moderada	PM = 3,0 $(KDSI)^{1,12} \times M$	Projetos mais complexos, em que os membros da equipe podem ter experiência limitada com os sistemas relacionados.
Embutida	PM = 3,6 $(KDSI)^{1,20} \times M$	Projetos complexos, em que o software é parte de um conjunto complexo de hardware, software, regulamentos e procedimentos operacionais.

* Person-month (pessoa-mês).

Figura 23.6 ▲
Modelo COCOMO 81 básico.

modelos de processo comumente utilizados. Em muitos casos, subsistemas de prateleira são utilizados. Os softwares já existentes passam pelo processo de reengenharia, para a criação de novos softwares. O suporte técnico às ferramentas CASE, para a maioria das atividades de processo de software, está disponível atualmente.

Para levar essas mudanças em consideração, o modelo COCOMO 2 reconhece diferentes abordagens de desenvolvimento de software, como a prototipação, o desenvolvimento pela composição de componentes e o uso de 4GLs. Os níveis de modelo não refletem simplesmente estimativas cada vez mais complexas e detalhadas. Os níveis estão associados com atividades no processo de software, de modo que as estimativas iniciais podem ser feitas previamente no processo, com as estimativas mais detalhadas sendo realizadas depois que a arquitetura do sistema foi definida. Os níveis identificados no COCOMO 2 são:

1. *Nível inicial de prototipação* As estimativas de tamanho são feitas com base em pontos de objeto, e uma fórmula simples de tamanho/produtividade é utilizada para estimar o esforço requerido.
2. *Nível inicial de projeto* Esse nível corresponde à conclusão dos requisitos do sistema com (talvez) algum projeto inicial. As estimativas são baseadas em pontos de função, que são, então, convertidas para o número de linhas de código-fonte. A fórmula segue o modo-padrão, discutido anteriormente, com um simples conjunto de multiplicadores associados a ela.
3. *Nível pós-arquitetura* Uma vez projetada a arquitetura do sistema, uma estimativa razoavelmente exata do tamanho do software pode ser feita. A estimativa desse nível utiliza um conjunto mais amplo de multiplicadores, refletindo a capacidade pessoal, as características de produto e de projeto.

O nível inicial de prototipação

O nível inicial de prototipação foi introduzido no COCOMO para apoiar a realização de estimativas de esforço necessário para projetos de prototipação e para projetos em que o software é desenvolvido pela composição dos componentes existentes. Ele é baseado em uma estimativa de pontos de objeto ponderados (discutida na Seção 23.1), que é dividida por um número-padrão para a produtividade estimada. A produtividade do programador depende de sua experiência e capacitação das ferramentas CASE utilizadas para apoiar o desenvolvimento. A Figura 23.7 mostra os diferentes níveis de produtividade sugeridos pelos desenvolvedores do modelo (Boehm *et al.*, 1995).

Nesse nível, o reuso é comum e, assim, o número de pontos de objeto utilizado na estimativa de prazos é ajustado para levar em conta a percentagem de reuso (% de reuso) que se espera. Portanto, a fórmula final para o cálculo do prazo é:

$$PM = (NOP \times (1 - \%\ reuso/100))/PROD.$$

PM *(person-month)* é o esforço medido em pessoa-mês, NOP *(number of object points)* é o número de pontos de objeto e PROD *(productivity)* é a produtividade, como mostra a Figura 23.7.

Experiência e capacitação do desenvolvedor	Muito baixa	Baixa	Nominal	Alta	Muito alta
Maturidade e capacitação de CASE	Muito baixa	Baixa	Nominal	Alta	Muito alta
PROD (NOP/mês)	4	7	13	25	50

Figura 23.7 ▲ Produtividade por pontos de objeto.

O nível inicial de projeto

As estimativas produzidas nesse estágio se baseiam na seguinte fórmula-padrão para os modelos algorítmicos:

$$\text{Esforço} = A \times \text{Tamanho}^B \times M.$$

Com base em seu amplo conjunto de dados, Boehm propõe que o coeficiente A deve ser 2,5 para as estimativas feitas nesse nível. O tamanho do sistema é expresso em KSLOC, ou seja, o número de milhares de linhas de código-fonte. Esse número é calculado estimando-se o número de pontos de função no software e convertendo-o para KSLOC, com a utilização de tabelas-padrão, que relacionam o tamanho do software a pontos de função, para diferentes linguagens de programação. Essa estimativa de tamanho se refere ao código que é implementado manualmente, em vez de ser gerado ou reutilizado.

O expoente B reflete o esforço aumentado, requerido à medida que o tamanho do projeto aumenta. Esse expoente não é fixo, como na primeira versão do COCOMO, mas pode variar de 1,1 a 1,24, dependendo da novidade do projeto, da flexibilidade de desenvolvimento, dos processos de resolução de risco utilizados, da coesão da equipe de desenvolvimento e do nível de maturidade de processo (veja o Capítulo 25) da organização. A maneira pela qual esse expoente é calculado será discutida nas seções seguintes.

O multiplicador M baseia-se em um conjunto simplificado de sete projetos e direcionadores de processo derivados, dentre os quais destacam-se: confiabilidade e complexidade do produto (*product reliability and complexity* – RCPX), reuso requerido (*reuse required* – RUSE), dificuldade de plataforma (*platform difficulty* – PDIF), capacitação pessoal (*personnel capability* – PERS), experiência pessoal (*personnel experience* – PREX), prazo (*schedule* – SCED) e recursos de suporte (*support facilities* – FCIL). Esses valores podem ser estimados diretamente em uma escala de seis pontos, em que 1 corresponde a valores muito baixos para esses multiplicadores e 6 corresponde a valores muito altos. Como alternativa, eles podem ser calculados combinando-se os valores dos multiplicadores mais detalhados utilizados no nível de pós-arquitetura.

Isso resulta em um cálculo de esforço, como se segue:

$$PM = A \times \text{Tamanho}^B \times M + PM_m, \text{ onde}$$

$$M = PERS \times RCPX \times RUSE \times PDIF \times PREX \times FCIL \times SCED.$$

O último termo na fórmula (PM_m) é um fator utilizado quando uma percentagem significativa do código é gerada automaticamente. Em geral, é necessária alguma entrada manual, a fim de ajustar esse código, mas o nível de produtividade é muito mais alto do que o código criado manualmente. Portanto, o esforço requerido (PM_m) é calculado de maneira separada, utilizando-se a seguinte fórmula, e, então, adicionado ao esforço calculado para o código desenvolvido manualmente.

$$PM_m = (ASLOC \times (AT/100))/ATPROD$$

ASLOC é o número de linhas de código-fonte geradas automaticamente e ATPROD é o nível de produtividade para esse tipo de produção de código. Contudo, algum esforço é exigido

para o código de interface gerado com o restante do sistema. Isso depende da percentagem do código total do sistema automaticamente gerado (AT). Portanto, a produtividade real depende do número de módulos gerados. Quanto menor a quantidade de código gerado, maior o custo geral envolvido em integrar esse código com outro código no sistema.

O nível pós-arquitetura

As estimativas produzidas no nível pós-arquitetura têm como base a mesma fórmula básica utilizada nas primeiras estimativas de projeto. Contudo, a estimativa de tamanho para o software deve ser mais precisa neste estágio, no processo de estimativa, e são utilizados 17 atributos, em vez de sete, para aperfeiçoar o cálculo do esforço inicial.

A estimativa do número total de linhas de código-fonte é ajustada para levar em conta dois importantes fatores de projeto:

- *A volatilidade dos requisitos* É feita uma estimativa do retrabalho que pode ser necessário para adequar as mudanças nos requisitos do sistema. Essa estimativa é expressa como o número de linhas de código-fonte que tem de ser modificado, e esse número é adicionado à estimativa inicial de tamanho.
- *A possível extensão do reuso* O amplo reuso significa que o número de linhas de código-fonte que realmente serão desenvolvidas precisa ser corrigido. Contudo, Selby (1988) constatou que os custos de reuso são não lineares, devido ao esforço inicial exigido para descobrir e selecionar componentes e do esforço requerido para compreender os componentes reutilizáveis e suas interfaces, se elas precisarem ser modificadas.

Os efeitos do reuso são levados em conta no COCOMO 2, ajustando-se a quantidade de esforço de acordo com a seguinte fórmula:

$$\text{ESLOC} = \text{ASLOC} \times (\text{AA} + \text{SU} + 0{,}4\text{DM} + 0{,}3\text{CM} + 0{,}3\text{IM})/100,$$

onde ESLOC é o número equivalente de linhas do novo código, ASLOC é o número de linhas de código reutilizável, que tem de ser modificado, DM é a percentagem do projeto modificado, CM é a percentagem do código que foi modificado e IM é a percentagem do esforço original de integração requerido para integrar o software reutilizado. SU é um fator que tem como base o custo de entendimento do software, que varia de 50, para os códigos não estruturados complexos, até 10, para os códigos orientados a objetos, escritos adequadamente. AA é um fator que reflete os custos da avaliação inicial para decidir se o software pode ser utilizado. Ele depende da quantidade de testes e da avaliação requerida. Seu valor varia de 0 a 8.

O expoente na fórmula de cálculo de esforço tinha três valores possíveis no COCOMO 1. Esses valores eram relacionados a diferentes níveis de complexidade de projeto. À medida que os projetos se tornam mais complexos, os efeitos do aumento de tamanho do sistema se tornam mais significativos. Contudo, práticas e procedimentos organizacionais podem controlar essa deseconomia de escala e isso é reconhecido no COCOMO 2. O expoente é estimado considerando cinco fatores em escala, como mostra a Figura 23.8. Esses fatores são classificados em uma escala de seis pontos, desde Muito baixo até Extra alto (5 a 0). As classificações resultantes são acrescentadas, divididas por 100, e o resultado é somado a 1,01 para fornecer o termo expoente.

Para ilustrar esse processo, imagine que uma organização está assumindo um projeto em um domínio em que tem pouca experiência prévia. O cliente não definiu o processo a ser utilizado e não houve tempo, conforme o cronograma do projeto, para uma análise de risco significativa. Uma nova equipe de desenvolvimento precisa ser constituída para implementar esse sistema. A organização recentemente implantou um programa de melhoria de processo e foi classificada como uma organização de Nível 2, de acordo com o modelo CMM

Figura 23.8

Fatores de escala utilizados no cálculo de expoente do COCOMO 2.

Fator de escala	Explicação
Precedência	Reflete a experiência prévia da organização com o tipo de projeto. Muito baixo significa nenhuma experiência prévia; Extra alto significa que a organização está inteiramente familiarizada com esse domínio da aplicação.
Flexibilidade de desenvolvimento	Reflete o grau de flexibilidade no processo de desenvolvimento. Muito baixo significa que um processo prescrito é utilizado; Extra alto significa que o cliente apenas estabelece objetivos gerais.
Resolução de arquitetura/risco	Reflete a extensão da análise de risco realizada. Muito baixo significa pouca análise; Extra alto significa uma análise de risco completa e abrangente.
Coesão da equipe	Reflete o quanto os membros da equipe de desenvolvimento se conhecem bem e trabalham juntos. Muito baixo significa interações muito difíceis; extra alto significa uma equipe integrada, e eficaz, sem problemas de comunicação.
Maturidade do processo	Reflete o processo de maturidade da organização. O cálculo desse valor depende do Questionário de Maturidade de CMM, mas uma estimativa pode ser feita subtraindo-se de 5 o nível de maturidade de processo de CMM.

(veja o Capítulo 25). Os possíveis valores para as classificações utilizadas no cálculo de expoente são:

1. *Precedência* Esse é um novo projeto para a organização – classificação: Baixo (4).
2. *Flexibilidade de desenvolvimento* Nenhum envolvimento do cliente – classificação: Muito alto (1).
3. *Resolução de arquitetura/risco* Nenhuma análise de risco realizada – classificação: Muito baixo (5).
4. *Coesão da equipe* Equipe nova, portanto nenhuma informação – classificação: Nominal (3)
5. *Maturidade do processo* Algum controle de processo implantado – classificação: Nominal (3).

A soma desses valores é 16; assim, o expoente resultante é 1,17.

Os atributos (Figura 23.9) utilizados para adequar as estimativas iniciais no modelo pós-arquitetura se dividem em quatro classes:

1. Os atributos de produto se referem às características requeridas do produto de software que está sendo desenvolvido.
2. Os atributos de computador são restrições impostas ao software pela plataforma de hardware.
3. Os atributos pessoais são multiplicadores, que levam em conta a experiência e a capacitação do pessoal que está trabalhando no projeto.
4. Os atributos de projeto se referem às características particulares do projeto de desenvolvimento de software.

A Figura 23.10 mostra um exemplo de como essas diretrizes de custo influenciam as estimativas de esforço. Considerei um valor para o expoente de 1,17, como foi discutido no exemplo mostrado anteriormente, e presumi que RELY, CPLX, STOR, TOOL e SCED são os principais custos no projeto. Todas as outras linhas de custos têm um valor nominal de 1, de modo que não afetam o cálculo do esforço.

No exemplo, atribuí valores máximos e mínimos aos principais direcionadores de custo, a fim de mostrar como eles influenciam a estimativa de esforço. Os valores utilizados são aqueles do manual de referência do COCOMO 2 (Boehm, 1997). Você pode ver que valores

Atributos de produto			
RELY	Confiabilidade exigida do sistema	DATA	Tamanho do banco de dados utilizado
CPLX	Complexidade dos módulos do sistema	RUSE	Percentagem requerida de componentes reusáveis
DOCU	Extensão da documentação requerida		
Atributos de computador			
TIME	Restrições de tempo de execução	STOR	Restrições de memória
PVOL	Volatilidade da plataforma de desenvolvimento		
Atributos de pessoal			
ACAP	Capacitação de analistas de projeto	PCAP	Capacitação do programador
PCON	Continuidade de pessoal	AEXP	Experiência do analista no domínio do projeto
PEXP	Experiência do programador no domínio do projeto	LTEX	Experiência na linguagem e na ferramenta
Atributos de projeto			
TOOL	Uso de ferramentas de software	SITE	Extensão do trabalho em vários locais e qualidade da comunicação entre locais
SCED	Redução do cronograma de desenvolvimento		

Figura 23.9 ▲

Direcionadores de custo de projeto.

altos para os direcionadores de custos levam a uma estimativa de esforço que é mais de três vezes maior do que a estimativa inicial, enquanto os valores baixos reduzem a estimativa para cerca de 1/3 do original. Isso destaca a ampla distinção entre os diferentes tipos de projeto e as dificuldades de transferir experiência de um domínio de aplicação para outro.

A fórmula proposta pelos desenvolvedores do modelo COCOMO 2 reflete sua experiência e os dados, mas, em meu ponto de vista, ela parece muito complexa para o uso prático. Existem atributos em demasia, e há muita incerteza para estimar seus valores. Em princípio, cada usuário do modelo deve ajustar o modelo e os valores dos atributos de acordo com os dados históricos próprios dos projetos, uma vez que isso reflete circunstâncias locais que afetam o modelo. Na prática, contudo, poucas organizações coletaram dados suficientes a partir de projetos passados, de uma maneira que possa apoiar o ajuste do modelo. O uso prático do COCOMO 2 normalmente começaria com os valores publicados para os parâmetros do modelo.

Figura 23.10 ▶

Efeito dos direcionadores de custo nas estimativas de esforço.

Valor do expoente	1,17
Tamanho do sistema (incluindo fatores de reuso e volatilidade de requisitos)	128 000 DSI
Estimativa inicial de COCOMO sem direcionadores de custo	**730 pessoas-mês**
Confiabilidade	Muito alta, multiplicador = 1,39
Complexidade	Muito alta, multiplicador = 1,3
Restrição de memória	Alta, multiplicador = 1,21
Uso de ferramenta	Baixo, multiplicador = 1,12
Cronograma	Acelerado, multiplicador = 1,29
Estimativa de COCOMO ajustada	**2 306 pessoas-mês**
Confiabilidade	Muito baixa, multiplicador = 0,75
Complexidade	Muito baixa, multiplicador = 0,75
Restrição de memória	Nenhuma, multiplicador = 1
Uso de ferramenta	Muito alto, multiplicador = 0,72
Cronograma	Normal, multiplicador = 1
Estimativa de COCOMO ajustada	**295 pessoas-mês**

23.3.2 Modelos algorítmicos de custo no planejamento de projeto

Os gerentes de projeto podem utilizar um modelo algorítmico de custos para comparar diferentes meios de investir dinheiro para reduzir os custos de projeto. Isso é particularmente importante quando é preciso haver uma conciliação do custo de hardware/software e quando houver a necessidade de recrutar uma nova equipe com habilidades específicas para o projeto.

Considere um sistema embutido para controlar um experimento que deverá ser lançado no espaço. Os experimentos espaciais precisam ser confiáveis e estão sujeitos a rigorosos limites de peso. O número de chips em uma placa de circuito integrado precisa ser reduzido. Em termos do modelo COCOMO, os multiplicadores baseados em restrições relativas a computadores e à confiabilidade são maiores do que 1.

Existem três componentes a serem levados em conta, quando se faz a estimativa do projeto:

1. O custo do hardware pretendido para executar o sistema.
2. O custo da plataforma (computador mais software) para desenvolver o sistema.
3. O custo do esforço necessário para desenvolver o software.

A Figura 23.11 mostra algumas das possíveis opções que podem ser consideradas. Elas incluem gastar mais no hardware pretendido, a fim de reduzir os custos de software ou investir em melhores ferramentas de desenvolvimento.

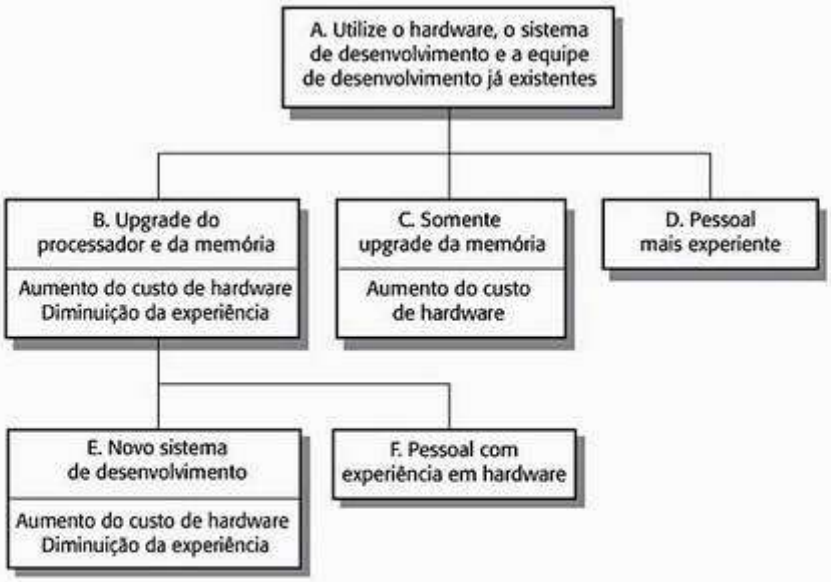
Os custos adicionais de hardware podem ser aceitáveis, nesse caso, porque o sistema é especializado e não precisa ser produzido em massa. Todavia, se o hardware for embutido em produtos de consumo, investir no hardware pretendido para reduzir os custos de software raramente é uma alternativa aceitável, porque isso aumenta o custo unitário do produto.

A Figura 23.12 mostra os custos de hardware, software e os custos totais para as opções A–F, mostradas na Figura 23.11. Aplicando o modelo COCOMO 2 sem direcionadores de custos, é possível prever um esforço de 45 pessoas-mês para desenvolver um sistema de software embutido para essa aplicação. O custo médio para uma pessoa-mês de esforço é de 15 mil dólares.

Os multiplicadores relevantes se baseiam em restrições de armazenamento e tempo de execução (TIME e STOR), disponibilidade de suporte à ferramenta (*cross-compilers* etc.) para o sistema em desenvolvimento (TOOL), experiência da equipe de desenvolvimento e experiência na plataforma (LTEX). Em todas as opções, o multiplicador de confiabilidade (RELY) é 1,39, o que indica que um esforço extra-significativo é necessário para desenvolver um sistema confiável.

Figura 23.11 ▶

Opções de gerenciamento.



Opção	RELY	STOR	TIME	TOOLS	LTEX	Esforço total	Custo de software	Custo de hardware	Custo total
A	1,39	1,06	1,11	0,86	1	63	949 393	100 000	1 049 393
B	1,39	1	1	1,12	1,22	88	1 313 550	120 000	1 402 025
C	1,39	1	1,11	0,86	1	60	895 653	105 000	1 000 653
D	*1,39*	*1,06*	*1,11*	*0,86*	*0,84*	*51*	*769 008*	*100 000*	*897 490*
E	1,39	1	1	0,72	1,22	56	844 425	220 000	1 044 159
F	1,39	1	1	1,12	0,84	57	851 180	120 000	1 002 706

Figura 23.12 ▲

Custos das opções de gerenciamento.

O custo de software (*software cost* – SC) é calculado como se segue:

SC = Estimativa de esforço × RELY × TIME × STOR × TOOL × EXP × US$ 15.000.

A opção A representa o custo de construir o sistema com o suporte técnico e o pessoal existente. Ela representa uma *baseline* de comparação. Todas as outras opções envolvem mais despesas de hardware ou o recrutamento (com os custos e riscos associados) de pessoal novo. A opção B mostra que fazer o upgrade do hardware não necessariamente reduz custos, porque o multiplicador de experiência é mais significativo. Na verdade, é mais eficaz em termos de custo fazer o upgrade da memória, em vez de toda a configuração do computador.

A opção D parece oferecer os custos mais baixos para todas as estimativas básicas. Nenhuma despesa adicional de hardware está envolvida, mas é preciso selecionar pessoal novo para o projeto. Se esse pessoal já estiver disponível na empresa, essa será provavelmente a melhor opção a escolher. Caso contrário, o pessoal precisa ser recrutado externamente, o que envolve custos e riscos significativos. Essa alternativa pode significar que as vantagens de custo dessa opção são muito menos significativas do que sugere a Figura 23.12. A opção C oferece uma economia de quase 50 mil dólares, com praticamente nenhum risco associado. Os gerentes de projeto conservadores, provavelmente, selecionariam essa opção, em vez da opção D, que apresenta maior risco.

As comparações mostram a importância da experiência da equipe como um multiplicador. Se o pessoal de boa qualidade, com a experiência certa, for recrutado, isso poderá reduzir bastante os custos de projeto. Essa condição é consistente com a discussão sobre os fatores de produtividade, na Seção 23.1, e revela também que o investimento em novo hardware e ferramentas pode não ser eficaz em termos de custos. Essas estratégias são promovidas com freqüência por desenvolvedores que gostam de trabalhar com novos sistemas. Contudo, a perda de experiência tem um efeito mais significativo sobre o custo do sistema do que as economias que provêm de um sistema de hardware novo.

23.4 Duração de projeto e pessoal

Além de estimar o esforço requerido para desenvolver um sistema de software e o custo total desse esforço, os gerentes de projeto também precisam estimar quanto tempo levará para desenvolvê-lo e quando o pessoal será necessário para trabalhar no projeto. O tempo de desenvolvimento para o projeto é chamado de prazo de projeto. Cada vez mais, as organizações estão pedindo prazos de desenvolvimento mais curtos, de maneira que seus produtos possam ser levados ao mercado antes dos produtos de seus concorrentes.

A relação entre o número de pessoas que trabalham em um projeto, o esforço total requerido e o tempo de desenvolvimento não é linear. À medida que o número de pessoas aumenta, mais esforço pode ser necessário; as pessoas podem gastar mais tempo se comunicando; mais tempo é exigido para definir as interfaces entre as partes do sistema. Duplicar o número de pessoas (por exemplo) não significa que a duração do projeto seja reduzida pela metade.

O modelo COCOMO inclui uma fórmula para estimar o tempo de calendário (TDEV) requerido para completar o projeto. A fórmula de cálculo de tempo é a mesma para todos os níveis do COCOMO:

$$\mathsf{TDEV} = 3 \times (\mathsf{PM})^{(0{,}33 + 0{,}2 * (B - 1{,}01))}.$$

PM é o cálculo do esforço e B é o expoente calculado, como foi discutido anteriormente (B é 1 para os modelos iniciais de prototipação). Esse cálculo prevê o prazo nominal para o projeto.

Contudo, o prazo previsto para o projeto e o prazo requerido pelo plano de projeto não são necessariamente o mesmo. O prazo planejado pode ser mais curto ou mais longo do que o prazo nominal previsto. Contudo, obviamente, existe um limite para a amplitude das mudanças de prazo, e esse limite é previsto pelo modelo COCOMO 2:

$$\mathsf{TDEV} = 3 \times (\mathsf{PM})^{(0{,}33 + 0{,}2 * (B - 1{,}01))} \times \mathit{SCEDPercentage/100}.$$

SCEDPercentage é o aumento ou a diminuição percentual do prazo nominal. Se o número previsto diferir muito do prazo planejado, isso sugere que existe alto risco de ocorrer problemas com a entrega do software conforme foi planejado.

Para ilustrar o cálculo do prazo de desenvolvimento no COCOMO, presuma que foi estimado um projeto de software que exigiria 60 meses de esforço de desenvolvimento (Opção C, na Figura 23.12). Suponha que o valor do expoente B seja 1,17.

Pela equação de prazo, o tempo requerido para concluir o projeto é:

$$\mathsf{TDEV} = 3\ (60)^{0{,}36} = 13 \text{ meses}.$$

Nesse caso, não há nenhuma redução ou expansão do prazo, de modo que o último termo na fórmula não tem nenhum efeito no cálculo.

Uma interessante implicação do modelo COCOMO é que o tempo requerido para realizar o projeto é uma função do esforço total necessário para o projeto. Isso não depende do número de engenheiros de software que trabalham no projeto e confirma a noção de que acrescentar mais pessoas a um projeto que está atrasado, provavelmente, não ajudará a recuperar o prazo. Myers (1989) discute os problemas da aceleração de um prazo. Ele sugere que, provavelmente, os projetos podem enfrentar grandes problemas, se houver a tentativa de desenvolver um software sem que haja tempo disponível suficiente.

Dividir o esforço requerido em um projeto pelo prazo de desenvolvimento não fornece uma informação útil do número de pessoas necessárias para a equipe de projeto. Em geral, o número de pessoas empregadas em um projeto de software aumenta a partir de um número relativamente pequeno, até chegar a um pico e, depois, declinar.

Somente um pequeno número de pessoas é necessário, no começo do projeto, para realizar o planejamento e a especificação. À medida que o projeto progride e mais trabalho detalhado é necessário, o número de pessoas sobe até atingir um limite. Depois da implementação e dos testes unitários estarem concluídos, o número de pessoas exigido começa a diminuir até atingir um ou dois profissionais, quando o produto for entregue. Um aumento muito rápido do pessoal necessário para o projeto, freqüentemente, se correlaciona a atrasos no projeto. Os gerentes devem, portanto, evitar acrescentar muita gente ao projeto na fase inicial de seu ciclo de vida.

O aumento do esforço pode ser modelado pelo que é chamado de curva de Rayleigh (Londeix, 1987), e o modelo de estimativa de Putnam (Putnam, 1978) incorpora um modelo

de provisão de pessoal de projeto com base nessa curva. O modelo de Putnam também inclui o tempo de desenvolvimento como um fator-chave. À medida que o tempo de desenvolvimento é reduzido, o esforço requerido para desenvolver o sistema aumenta exponencialmente.

PONTOS-CHAVE

- Os fatores que afetam a produtividade incluem a aptidão individual (o fator dominante), a experiência no domínio, o processo de desenvolvimento, o tamanho do projeto, o suporte a ferramentas e o ambiente de trabalho.
- Há várias técnicas de estimativa de custo de software. Ao preparar uma estimativa, diversas dessas técnicas devem ser utilizadas. Se as estimativas divergirem amplamente, isso significa que uma informação inadequada sobre estimativa está disponível.
- O software é, freqüentemente, avaliado com um preço definido para ganhar um contrato e a funcionalidade do sistema é, então, ajustada para se adequar ao preço estimado.
- A modelagem algorítmica de custos sofre da dificuldade fundamental de que ela se baseia em atributos do produto acabado para fazer a estimativa. Nos estágios iniciais do projeto, é impossível estimar esses atributos precisamente.
- O modelo COCOMO de custos é um modelo bem-desenvolvido, que leva em conta o projeto, o produto, o hardware e os atributos pessoais, quando se formula uma estimativa de custo. Ele inclui também uma maneira de estimar os prazos de desenvolvimento.
- Os modelos algorítmicos de custos são valiosos para o gerenciamento, uma vez que aceitam a análise quantitativa de opções. Eles permitem o cálculo do custo de várias opções e, mesmo com erros, as opções podem ser comparadas em uma base objetiva.
- O tempo necessário para realizar um projeto não é simplesmente proporcional ao número de pessoas que trabalham nele. Colocar mais pessoas em um projeto que está atrasado pode acarretar ainda mais atrasos.

LEITURAS SUGERIDAS

Software project management: readings and cases. Uma seleção de artigos e estudos de casos sobre o gerenciamento de projeto de software, que é particularmente importante na abordagem da modelagem algorítmica de custos. (C.F. Kemerer (ed.), 1997, Irwin.)

"Cost models for future life cycle processes: COCOMO 2". Uma apresentação abrangente do modelo de estimativa de custos COCOMO 2, que inclui o ponto de vista racional das fórmulas utilizadas. (B. Boehm *et al.*, *Annals of Software Engineering*, 1, Balzer Science Publishers, 1995.)

"Nine management guidelines for better cost estimating". Boa orientação prática sobre a estimativa de custos. (A. Lederer e J. Pasad, *Comm. ACM*, 35(2), fev. 1992).

EXERCÍCIOS

23.1 Descreva duas métricas que tenham sido utilizadas para medir a produtividade de um programador. Comente brevemente sobre as vantagens e as desvantagens dessas métricas.

23.2 No desenvolvimento de grandes sistemas embarcados de tempo real, sugira cinco fatores que, provavelmente, tenham efeito significativo na produtividade do desenvolvimento de software.

23.3 As estimativas de custo são arriscadas, independentemente da técnica de estimativa utilizada. Sugira quatro maneiras pelas quais o risco em uma estimativa de custos possa ser reduzido.

23.4 Um gerente de software está a cargo do desenvolvimento de um sistema de software crítico de segurança, que é projetado para controlar uma máquina de radioterapia destinada ao tratamento de pacientes com câncer. Esse sistema está embutido na máquina e deve ser executado em um processador de propósito especial, com uma quantidade fixa de memória (8 Mbytes). A máquina se comunica com um banco de dados do paciente para obter os detalhes necessários e, depois do tratamento, automaticamente registra a dose de radiação emitida e outros detalhes do tratamento no banco de dados.

O método COCOMO é utilizado para estimar o esforço requerido para desenvolver esse sistema, e uma estimativa de 26 pessoas-mês é calculada. Todos os multiplicadores de direcionadores de custo foram ajustados para 1 ao fazer a estimativa.

Explique por que essa estimativa deve ser ajustada para levar em conta fatores de projeto, pessoal, produto e fatores organizacionais. Sugira quatro fatores que possam ter efeitos significativos na estimativa inicial de COCOMO e proponha possíveis valores para esses fatores. Justifique o motivo da inclusão de cada fator.

23.5 Dê três razões pelas quais as estimativas algorítmicas de custos, preparadas em diferentes organizações, não são diretamente comparáveis.

23.6 Explique como a abordagem algorítmica para estimativa de custos pode ser utilizada pelos gerentes de projeto para a análise das opções. Sugira uma situação em que os gerentes possam escolher uma abordagem que não tenha como base o custo mais baixo de projeto.

23.7 Implemente o modelo COCOMO, utilizando uma planilha de cálculo, como o Excel, da Microsoft. Os detalhes do modelo podem ser transferidos por download a partir do site Web do COCOMO 2. Introduzi um link para este site nas páginas Web do livro.

23.8 Alguns projetos de software muito grandes envolvem escrever milhões de linhas de código. Sugira o grau de utilidade que os modelos de estimativa de custos podem, provavelmente, ter para esses sistemas. Por que as suposições em que se baseiam podem ser inválidas para os sistemas de software muito grandes?

23.9 É ético para uma empresa cotar um preço baixo para um contrato de software, sabendo que os requisitos são ambíguos e que eles podem cobrar altos preços pelas mudanças subseqüentes, solicitadas pelo cliente?

23.10 A produtividade medida deve ser utilizada pelos gerentes durante o processo de avaliação da equipe? Que cuidados são necessários para garantir que a qualidade não seja afetada por essa decisão?

24 Gerenciamento de qualidade

Objetivos

Os objetivos deste capítulo são apresentar o gerenciamento de qualidade de software e descrever atividades específicas de gerenciamento de qualidade. Depois de ler este capítulo, você poderá:

- compreender o processo de gerenciamento de qualidade e as principais atividades do processo de garantia, planejamento e controle de qualidade;
- compreender a importância dos padrões no processo de gerenciamento de qualidade;
- compreender a noção de uma métrica de software e as diferenças entre as métricas preditivas e as métricas de controle, e
- compreender como a medição pode ser útil para avaliar alguns atributos de qualidade e as atuais limitações da medição de software.

Sumário

24.1 Garantia e padrões de qualidade
24.2 Planejamento de qualidade
24.3 Controle de qualidade
24.4 Medição e métricas de software

Atingir um alto nível de qualidade de produto ou serviço é o objetivo da maioria das organizações. Atualmente, não é mais aceitável entregar produtos com baixa qualidade e reparar os problemas e as deficiências depois que os produtos foram entregues ao cliente. A esse respeito, o software é igual a qualquer outro produto manufaturado, como carros, televisões e computadores.

Contudo, a qualidade do software é um conceito complexo, que não pode ser definido de maneira simples. Classicamente, a noção de qualidade tem sido a de que o produto desenvolvido deve cumprir com sua especificação (Crosby, 1979). Em um mundo ideal, essa definição se aplicaria a todos os produtos, mas, para os sistemas de software, existem problemas:

1. A especificação deve ser orientada em direção às características do produto que o cliente deseja. Contudo, a organização de desenvolvimento pode também ter requisitos (como os requisitos de facilidade de manutenção) que não estejam incluídos na especificação.
2. Não sabemos como especificar certas características de qualidade (por exemplo, a facilidade de manutenção) de uma maneira que não seja ambígua.
3. Conforme foi discutido na Parte 1, que abordou a engenharia de requisitos, é muito difícil escrever uma especificação completa de software. Portanto, embora um produto de software possa atender à sua especificação, os usuários podem não considerá-lo um produto de alta qualidade.

Obviamente, é preciso algum esforço para melhorar as especificações, mas hoje em dia temos de aceitar que elas serão imperfeitas. Devemos reconhecer os problemas com as especificações existentes e implantar procedimentos para melhorar a qualidade dentro das restrições impostas por uma especificação imperfeita. Em particular, os atributos de software, como facilidade de manutenção, portabilidade ou eficiência, podem ser atributos de qualidade fundamentais, que não são especificados explicitamente, mas afetam a qualidade percebida do sistema. Esses atributos de qualidade serão discutidos na Seção 24.2, que trata do planejamento de qualidade.

A responsabilidade dos gerentes de projeto em uma organização é garantir que o nível exigido de qualidade do produto seja atingido. Em princípio, o gerenciamento de qualidade simplesmente envolve definir procedimentos e padrões que devem ser utilizados durante o desenvolvimento de software e verificar se eles estão sendo seguidos por todos os engenheiros. Na prática, contudo, há mais detalhes sobre o gerenciamento de qualidade do que isso.

Os bons gerentes de qualidade objetivam desenvolver uma 'cultura de qualidade', segundo a qual todos os responsáveis pelo desenvolvimento do produto se comprometam a atingir um alto nível de qualidade para ele. Eles encorajam as equipes a assumir a responsabilidade pela qualidade de seu trabalho e a desenvolver novas abordagens para a melhoria dessa qualidade. Embora os padrões e os procedimentos sejam a base do gerenciamento de qualidade, os gerentes de qualidade experientes reconhecem que existem aspectos intangíveis da qualidade de software (adequação, legibilidade etc.), que não podem ser incorporados em padrões. Eles apóiam as pessoas interessadas nesses aspectos intangíveis de qualidade e encorajam o comportamento profissional de todos os membros da equipe.

O gerenciamento de qualidade de software pode ser estruturado em três atividades principais:

1. *Garantia de qualidade* O estabelecimento de uma estrutura de procedimentos e de padrões organizacionais, que conduzam ao software de alta qualidade.
2. *Planejamento de qualidade* A seleção de procedimentos e de padrões adequados a partir dessa estrutura e a adaptação destes para um projeto específico de software.
3. *Controle de qualidade* A definição e a aprovação de processos que assegurem que os procedimentos e os padrões de qualidade de projeto sejam seguidos pela equipe de desenvolvimento de software.

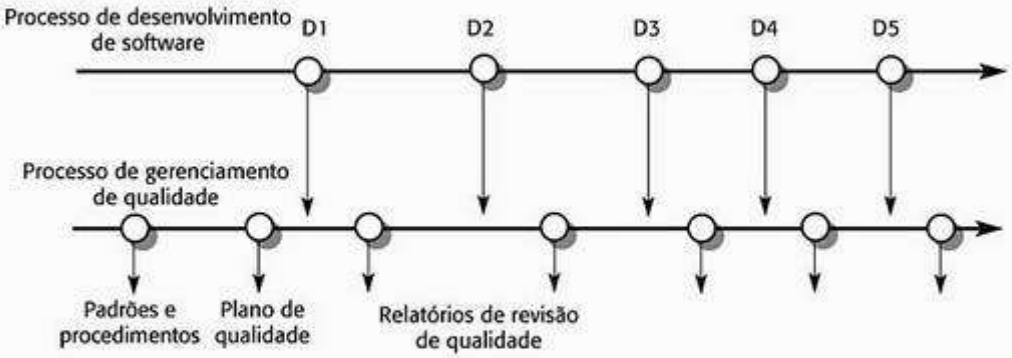


Figura 24.1 ▲

Gerenciamento de qualidade e desenvolvimento de software.

O gerenciamento de qualidade fornece uma verificação independente sobre o processo de desenvolvimento de software. Os produtos entregues a partir do processo de software são entradas para o processo de gerenciamento de qualidade e são verificados para assegurar que são consistentes com os padrões e os objetivos da organização (Figura 24.1). Como a equipe de garantia e controle de qualidade deve ser independente, ela pode ter uma visão objetiva do processo e pode relatar problemas e dificuldades para a gerência sênior na organização.

O gerenciamento de qualidade deve ser separado do gerenciamento de projeto, de modo que a qualidade não seja comprometida pelas responsabilidades de gerenciamento quanto ao orçamento e aos prazos de projeto. Uma equipe independente deve ser responsável pelo gerente de qualidade e deve se reportar à gerência superior ao nível de gerente de projeto. A equipe de gerenciamento de qualidade não deve ficar associada com qualquer grupo específico de desenvolvimento, mas deve assumir responsabilidade ampla pelo gerenciamento de qualidade na organização.

Um padrão internacional, que pode ser utilizado no desenvolvimento de um sistema de gerenciamento de qualidade em todas as indústrias, é chamado de ISO 9000. A ISO 9000 é um conjunto de padrões que pode ser aplicado a uma gama de organizações, desde a indústria de manufatura até as indústrias de serviços. A ISO 9001 é a mais geral desses padrões e se aplica a organizações que se preocupam com o processo de qualidade das empresas que projetam, desenvolvem e fazem manutenção de produtos. Um documento de apoio (ISO 9000-3) interpreta a ISO 9000 para o desenvolvimento de software. Diversos livros que descrevem o padrão ISO 9000 estão disponíveis (Johnson, 1993; Oskarsson e Glass, 1995; Peach, 1996).

A ISO 9001 é um modelo genérico de processo de qualidade que descreve vários aspectos do processo e define quais padrões e procedimentos devem existir dentro de uma organização. Como a ISO 9001 não é específica para uma única indústria, esses documentos não são definidos em detalhes. Dentro de cada organização específica, um conjunto apropriado de processos de qualidade deve ser definido e documentado em um manual de qualidade organizacional.

A Figura 24.2 mostra as áreas cobertas pela ISO 9001. Neste livro, não tenho espaço para discutir esse padrão em qualquer profundidade. Ince (1994) e Oskarsson e Glass (1995) fazem um relato mais detalhado de como o padrão pode ser utilizado para desenvolver processos de gerenciamento de qualidade de software.

Os procedimentos de garantia de qualidade em uma organização são documentados em um manual que define o processo de qualidade. Em alguns países, existem instituições que certificam que o processo de qualidade, segundo expresso no manual, está em conformidade com a ISO 9001. Cada vez mais, os clientes procuram a certificação da ISO 9000 em um fornecedor, como um indicador do nível de seriedade com que esse fornecedor considera a qualidade.

A relação entre a ISO 9000, o manual de qualidade e os planos individuais de qualidade de projeto é mostrada na Figura 24.3, extraída de um modelo fornecido pelo livro de Ince (1994).

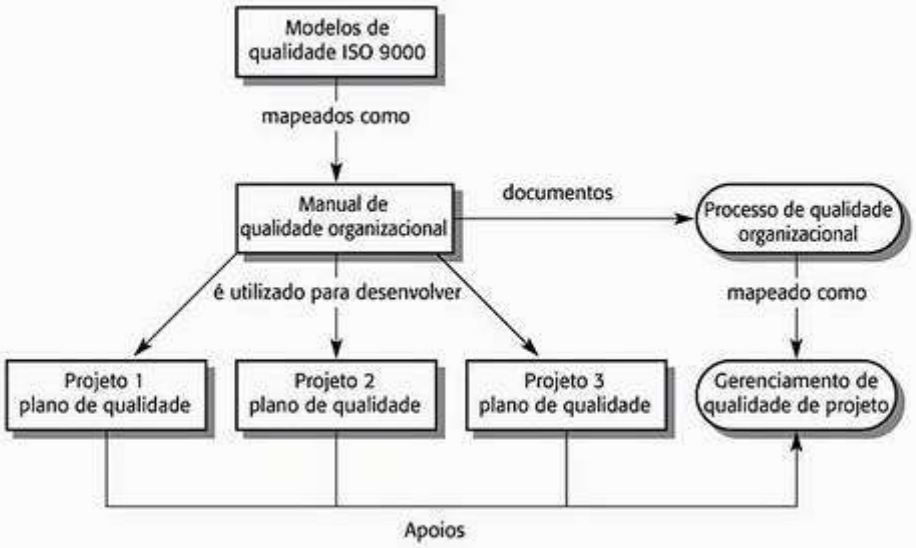
Figura 24.2

Áreas cobertas pelo modelo ISO 9001 para a garantia de qualidade.

Responsabilidade de gerenciamento	Sistema de qualidade
Controle de produtos que não estão em conformidade	Controle de projeto
Manuseio, armazenamento, embalagem e entrega	Compras
Produtos fornecidos para o comprador	Identificação e facilidade de rastreamento do produto
Controle de processo	Inspeção e teste
Equipamentos de inspeção e teste	Status de inspeção e teste
Revisão de contrato	Ação corretiva
Controle de documento	Registros de qualidade
Auditorias internas de qualidade	Treinamento
Prestação de serviço	Técnicas estatísticas

Figura 24.3

A ISO 9000 e o gerenciamento de qualidade.



24.1 Garantia e padrões de qualidade

As atividades de garantia de qualidade (*quality assurance* – QA) definem uma estrutura para atingir a qualidade de software. Esse processo de QA envolve definir ou selecionar os padrões que devem ser aplicados ao processo de desenvolvimento de software ou ao produto de software. Esses padrões podem ser integrados em procedimentos ou processos que são aplicados durante o desenvolvimento. Os processos podem ser apoiados pelo uso de ferramentas que integrem o conhecimento dos padrões de qualidade.

Existem dois tipos de padrões que podem ser estabelecidos como parte do processo de garantia de qualidade:

1. *Padrões de produto* São os padrões que se aplicam ao produto de software em desenvolvimento. Eles incluem padrões de documentos, como a estrutura do documento de

requisitos a ser produzido; padrões de documentação, como um cabeçalho-padrão de comentário para uma definição de classe de objeto, e padrões de codificação, que definem como uma linguagem de programação deve ser utilizada.

2. *Padrões de processo* São os padrões que definem os processos a serem seguidos durante o desenvolvimento de software. Eles podem incluir definições de especificação, processos de projeto e validação, e uma descrição dos documentos que devem ser gerados no curso desses processos.

Existe uma relação muito estreita entre os padrões de produto e de processo. Os padrões de produto aplicam-se às saídas do processo de software. E, em muitos casos, os padrões de processo incluem atividades específicas de processo que asseguram que os padrões de produto sejam seguidos. Será discutida a importante relação entre qualidade de produto e processo na Seção 24.1.2.

Há uma série de razões pelas quais os padrões de software são importantes:

1. Eles fornecem um encapsulamento das melhores práticas ou, pelo menos, das mais adequadas. Esse conhecimento é freqüentemente adquirido depois de um grande número de tentativas e erros. Constituir esse conhecimento dentro de um padrão evita a repetição de erros cometidos anteriormente. Os padrões registram sabedoria que tem valor para a organização.
2. Eles fornecem uma infra-estrutura em torno da qual o processo de garantia de qualidade pode ser implementado. Considerando que os padrões incluem as melhores práticas, o controle de qualidade simplesmente garante que os padrões foram seguidos de maneira adequada.
3. Eles ajudam em termos de continuidade, quando o trabalho realizado por uma pessoa é assumido e continuado por outra pessoa. Os padrões asseguram que todos os engenheiros dentro de uma organização adotam as mesmas práticas. Conseqüentemente, o esforço de aprendizado é reduzido, quando se inicia um novo trabalho.

O desenvolvimento de padrões de projeto de engenharia de software é um processo difícil e demorado. Organizações nacionais e internacionais, como US DoD (*United States Department of Defense* – Departamento de Defesa dos Estados Unidos), ANSI (*American National Standards Institute* – Instituto Nacional Norte-americano de Padrões), BSI (*British Standards Institution* – Instituto Britânico de Padrões), Otan (Organização do Tratado do Atlântico Norte) e IEEE (*Institute of Electrical and Electronic Engineers* – Instituto de Engenheiros Elétricos e Eletrônicos), estão ativas na produção de padrões. Esses são padrões gerais, que podem ser aplicados a uma série de projetos. Organizações como a Otan, assim como outras organizações de defesa, podem exigir que seus próprios padrões sejam seguidos em contratos de software.

Padrões nacionais e internacionais foram desenvolvidos com abrangência da terminologia de engenharia de software, linguagens de programação, como Ada e C++, notações, como símbolos gráficos, procedimentos para levantar e escrever requisitos de software, procedimentos de garantia de qualidade e processos de verificação e validação de software (IEEE, 1994).

As equipes de garantia de qualidade que estiverem desenvolvendo padrões, normalmente, deverão basear os padrões organizacionais nos padrões nacionais e internacionais. Com a utilização desses padrões como ponto de partida, a equipe de garantia de qualidade deve elaborar um manual de padrões, que deve definir aqueles apropriados para sua organização. Exemplos de padrões que podem ser incluídos nesse manual são mostrados na Figura 24.4.

Os engenheiros de software, algumas vezes, consideram os padrões como burocráticos e irrelevantes para a atividade técnica de desenvolvimento de software. Isso é particularmente provável quando os padrões exigem tediosos preenchimentos de formulários e registros de trabalho. Embora concordem com a necessidade geral de padrões, os engenheiros, muitas vezes, encontram boas razões pelas quais os padrões não são necessariamente apropriados a seu projeto particular.

Figura 24.4

Padrões de produto e de processo.

Padrões de produto	Padrões de processo
Formulário de revisão de projeto	Conduta de revisão de projeto
Estrutura do documento de requisitos	Submissão de documentos a CM (gerenciamento de configuração)
Modelo de cabeçalho de procedimento	Processo de liberação de versão
Estilo de programação em Java	Processo de aprovação do plano de projeto
Modelo do plano de projeto	Processo de controle da mudança
Formulário de pedido de mudança	Processo de registro de teste

Para evitar esses problemas, os engenheiros de qualidade que estabelecem os padrões precisam ser adequadamente habilitados e devem seguir as seguintes etapas:

1. Envolver os engenheiros de software no desenvolvimento de padrões para o produto. Eles devem compreender a motivação por detrás do desenvolvimento dos padrões e devem se comprometer com eles. O documento de padrões não deve simplesmente declarar um padrão a ser seguido, mas deve incluir as razões pelas quais foram tomadas decisões de padronização específicas.
2. Revisar e modificar os padrões regularmente, para que eles reflitam as constantes evoluções tecnológicas. Uma vez desenvolvidos os padrões, eles tendem a ser preservados em um manual de padrões da empresa e sempre existe relutância em modificá-los. Um manual de padrões é essencial, mas deve evoluir com as circunstâncias e tecnologias mutáveis.
3. Fornecer ferramentas de software para apoiar os padrões sempre que possível. Os padrões que não recebem apoio são motivos de muitas queixas, devido ao difícil trabalho envolvido em implementá-los. Se o apoio de ferramentas estiver disponível, não será necessário muito esforço adicional para o desenvolvimento de acordo com os padrões.

Os padrões de processo podem causar dificuldades, se um processo pouco prático for imposto à equipe de desenvolvimento. Esses padrões são, muitas vezes, simples diretrizes que devem ser interpretadas de modo compreensivo pelos gerentes de projetos individuais. Não haverá nenhuma razão para prescrever uma maneira específica de trabalhar, se ela não for apropriada para o projeto ou para a equipe de projeto. Cada gerente de projeto deve, portanto, ter autoridade para modificar os padrões de processo, de acordo com as circunstâncias individuais. Contudo, os padrões que se relacionam à qualidade do produto e ao processo após a liberação devem ser modificados somente depois de cuidadosa consideração.

O gerente de projeto e o gerente de qualidade podem evitar os problemas relacionados a padrões não apropriados mediante o cuidadoso planejamento da qualidade. Eles devem decidir quais dos padrões constantes no manual devem ser utilizados sem alterações, quais devem ser modificados e quais devem ser ignorados. Pode ser necessário criar novos padrões em resposta a determinado requisito de projeto. Por exemplo, poderão ser exigidos padrões para especificação formal, se eles não tiverem sido utilizados em projetos anteriores. Esses novos padrões devem evoluir durante o projeto.

24.1.1 Padrões de documentação

Os padrões de documentação em um projeto de software são particularmente importantes, uma vez que os documentos são a única maneira tangível de representar o software e o

processo de software. Os documentos padronizados têm aparência, estrutura e qualidade consistentes e devem, portanto, ser de fácil leitura e compreensão.

Há três tipos de padrões de documentação:

1. *Padrões de processo de documentação* Eles definem o processo a ser seguido na produção de documentos.
2. *Padrões de documento* Eles regem a estrutura e a apresentação de documentos.
3. *Padrões de intercâmbio de documentos* São padrões que asseguram que todas as cópias eletrônicas de documentos sejam compatíveis.

Os padrões de processo definem o processo utilizado para produzir documentos. Isso significa definir os procedimentos envolvidos no desenvolvimento de documentos e as ferramentas de software utilizadas para a produção de documentos. Devem também ser definidos os procedimentos de verificação e refinamento, que assegurem a produção de documentos de alta qualidade.

Os padrões de qualidade de processo de documentação têm de ser flexíveis e devem englobar todos os tipos de documentos. Para documentos ou memorandos relacionados a trabalho, não há necessidade da conferência explícita da qualidade. Contudo, quando se tratar de documentos formais, utilizados para desenvolvimento posterior, ou quando esses documentos forem entregues aos clientes, deve ser adotado um processo formal de qualidade. A Figura 24.5 é um modelo de um possível processo.

A elaboração do rascunho, da conferência, da revisão e de um novo rascunho é um processo iterativo, que deve continuar até que seja produzido um documento de qualidade aceitável. O nível de qualidade aceitável depende do tipo de documento e dos leitores desse documento em potencial.

Os padrões de documento devem ser aplicados a todos os documentos produzidos durante o desenvolvimento de software. Os documentos devem ter estilo e aparência consistentes, e documentos do mesmo tipo devem ter uma estrutura consistente. Embora os padrões de documentos devam ser adaptados às necessidades de um projeto específico, é uma boa prática utilizar o mesmo 'estilo' em todos os documentos produzidos por uma organização.

Exemplos de padrões de documentos que podem ser desenvolvidos são:

1. *Padrões de identificação de documentos* Como os grandes projetos de sistema podem produzir milhares de documentos, cada documento precisa ser identificado de maneira única. Para os documentos formais, esse identificador pode ser o identificador formal definido pelo gerente de configuração. Para os documentos informais, o estilo do identificador de documentos deve ser definido pelo gerente de projeto.

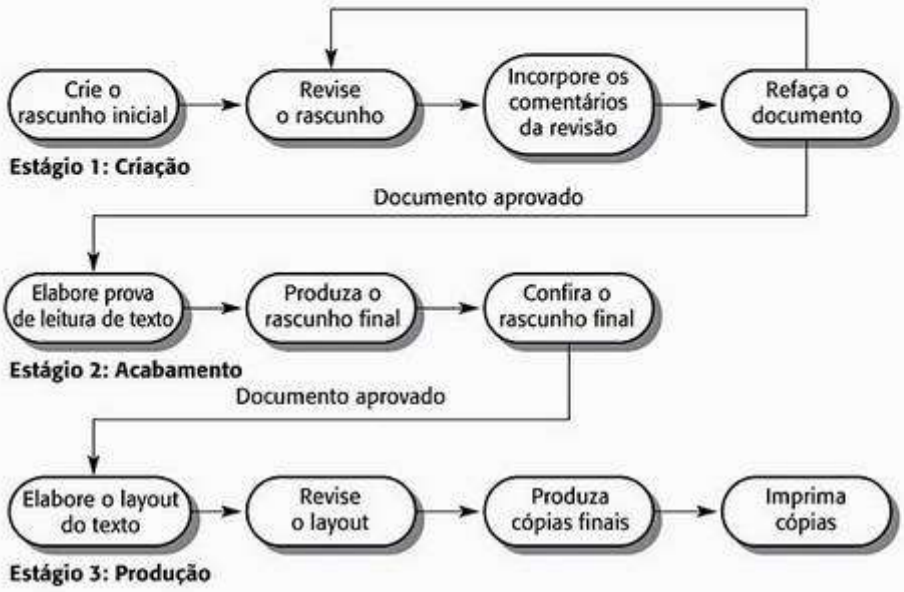
2. *Padrões de estrutura de documentos* Cada classe de documentos produzida durante um projeto de software deve seguir alguma estrutura-padrão. Os padrões de estrutura devem definir as seções a serem incluídas e especificar as convenções utilizadas para a numeração de páginas, as informações no cabeçalho e no rodapé da página e a numeração de seção e de subseção.

3. *Padrões de apresentação de documentos* Os padrões de apresentação de documentos definem um 'estilo' para os documentos e contribuem significativamente para sua consistência. Eles incluem a definição de fontes e estilos utilizados no documento, o uso de logotipos e nomes de empresas, o uso de cores para ressaltar a estrutura dos documentos, entre outros.

4. *Padrões de atualização de documentos* Uma vez que um documento evolui para refletir as mudanças ocorridas no sistema, deve ser utilizado um meio consistente para indicar modificações nos documentos. É possível utilizar diferentes cores de capas, para uma nova versão de documento, e barras de mudança, na margem, para indicar parágrafos que foram adicionados ou modificados.

Figura 24.5

Processo de produção de documentos que inclui a verificação de qualidade.



Os padrões de intercâmbio de documentos são importantes, uma vez que existe o intercâmbio de cópias eletrônicas de documentos. O uso de padrões de intercâmbio permite que os documentos sejam transferidos eletronicamente e recriados em sua forma original.

Supondo-se que o uso de ferramentas-padrão seja obrigatório nos padrões de processo, os padrões de intercâmbio definem as convenções de uso dessas ferramentas. Dentre os exemplos de padrões de intercâmbio destacam-se o uso de um conjunto macro de padrões estabelecidos, caso um sistema de formatação de texto seja utilizado para a produção de documentos, ou o uso de uma folha de estilo-padrão, se um processador de texto for utilizado. Os padrões de intercâmbio podem também limitar as fontes e os estilos de texto utilizados, devido aos diferentes recursos das impressoras e dos displays.

24.1.2 Qualidade de produto e de processo

Uma suposição básica do gerenciamento de qualidade é que a qualidade do processo de desenvolvimento afeta diretamente a qualidade dos produtos fornecidos. Essa suposição é derivada dos sistemas de produção, em que a qualidade do produto está intimamente relacionada ao processo de produção. Na verdade, em sistemas automatizados de produção em massa, uma vez atingido um nível aceitável de qualidade de processo, a qualidade do produto decorre naturalmente. A Figura 24.6 ilustra essa abordagem de garantia de qualidade.

O processo de qualidade é particularmente importante no desenvolvimento de software. A razão disso é que é difícil medir os atributos de software, como a facilidade de manutenção, sem utilizar o software por um longo período. A melhoria da qualidade focaliza a identificação de produtos de boa qualidade, examinando os processos utilizados para desenvolver esses produtos e, então, generalizando esses processos, de modo que eles possam ser aplicados em uma série de projetos. Contudo, a relação entre a qualidade de processos de software e os produtos de software é complexa. A modificação do processo nem sempre conduz à melhoria da qualidade.

Existe uma nítida ligação entre a qualidade do processo e do produto em produção, porque o processo é relativamente fácil de padronizar e monitorar. Uma vez ajustados os sistemas de produção, eles podem ser executados novamente diversas vezes, a fim de produzir produtos de alta qualidade. O software não é manufaturado, mas é projetado. Como o desenvolvimento de software é um processo criativo, e não mecânico, é importante a influência das habilidades e das experiências individuais. Fatores externos, como a novidade de uma aplicação ou a pressão comercial para a liberação rápida de um produto, também afetam a qualidade do produto, independentemente do processo utilizado.

Figura 24.6 ▲
Qualidade baseada no processo.

Não obstante, a qualidade do processo tem uma influência significativa na qualidade do software. O gerenciamento de qualidade de processo envolve:

1. A definição de padrões de processo, como o modo pelo qual as revisões devem ser conduzidas, quando elas deverão ocorrer etc.
2. O monitoramento do processo de desenvolvimento, a fim de assegurar que os padrões sejam seguidos.
3. A elaboração de relatórios do processo de software para a gerência de projeto e para o comprador do software.

Um perigo relacionado à garantia de qualidade baseada no processo é que o processo prescrito pode ser inadequado para o tipo de software em desenvolvimento. Por exemplo, os padrões de qualidade de processo podem definir que uma especificação deve estar completa e aprovada antes que a implementação tenha início. Contudo, alguns sistemas podem exigir a prototipação, que envolve a implementação do programa. A equipe de qualidade pode sugerir que essa prototipação não deve ser realizada, porque sua qualidade não pode ser monitorada. Nessas situações, a gerência sênior tem de intervir para assegurar que o processo de qualidade apóie, e não prejudique, o desenvolvimento do produto.

No próximo capítulo, que trata da melhoria de processo, volto a abordar as relações entre a qualidade de processo e de produto.

24.2 Planejamento de qualidade

O planejamento de qualidade deve começar em um estágio inicial do processo de software. Um plano de qualidade deve estabelecer as qualidades desejadas para o produto. Ele deve definir como essas qualidades devem ser avaliadas; portanto, o plano define o que software de 'alta qualidade' realmente significa. Sem essa definição, diferentes engenheiros podem trabalhar de maneira contraditória para que os diferentes atributos do produto sejam otimizados. O resultado do processo de planejamento de qualidade é um plano de qualidade de projeto.

O plano de qualidade deve selecionar os padrões organizacionais que forem apropriados a um determinado produto e processo de desenvolvimento. Novos padrões podem ser definidos, se o projeto utilizar novos métodos e ferramentas. Humphrey (1989), em seu livro clássico sobre gerenciamento de software, sugere uma estrutura geral para um plano de qualidade, que inclui:

1. *Introdução sobre o produto* Uma descrição do produto, seu mercado pretendido e as respectivas expectativas quanto à qualidade.

Figura 24.7

Atributos de qualidade do software.

Segurança	Facilidade de compreensão	Portabilidade
Proteção	Testabilidade	Facilidade de uso
Confiabilidade	Facilidade de adaptação	Facilidade de reuso
Capacidade de recuperação	Modularidade	Eficiência
Robustez	Complexidade	Facilidade de aprendizado

2. *Planos para o produto* As datas importantes de liberação e as responsabilidades referentes ao produto, juntamente com os planos para a distribuição e a prestação de serviços relacionadas a ele.
3. *Descrições de processo* Os processos de desenvolvimento e de serviço que devem ser utilizados para o desenvolvimento e gerenciamento do produto.
4. *Metas de qualidade* As metas e os planos de qualidade para o produto, incluindo identificação e justificativa de importantes atributos da qualidade do produto.
5. *Riscos e gerenciamento de riscos* Os principais riscos que podem afetar a qualidade do produto e as ações para evitar esses riscos.

Ao escrever os planos de qualidade, é preciso tentar mantê-los tão breves quanto for possível. Se o documento for muito extenso, os engenheiros não o lerão, e isso anulará o propósito de produzir um plano de qualidade.

Existe uma ampla gama de atributos de qualidade de software em potencial (Figura 24.7) que devem ser considerados durante o processo de planejamento de qualidade. Em geral, não é possível para qualquer sistema ser otimizado em relação a todos esses atributos, e, assim, uma parte importante do planejamento de qualidade é selecionar os principais atributos de qualidade e planejar como eles podem ser obtidos.

O plano de qualidade deve definir os atributos de qualidade mais significativos para o produto a ser desenvolvido. Pode acontecer de a eficiência ser o mais importante e de os outros fatores terem de ser sacrificados para se obter essa eficiência. Se isso for estabelecido no plano, os engenheiros que trabalham no desenvolvimento poderão cooperar para obtê-lo. O plano deve também definir o processo de avaliação de qualidade. Essa deve ser uma forma-padrão de avaliar se alguma qualidade, como a facilidade de manutenção, está presente no produto.

24.3 Controle de qualidade

O controle de qualidade envolve supervisionar o processo de desenvolvimento de software, a fim de assegurar que os procedimentos e os padrões de garantia de qualidade sejam seguidos. Como foi discutido anteriormente neste capítulo (veja a Figura 24.1), produtos elaborados pelo processo de software são verificados em relação aos padrões definidos de projeto, no processo de controle de qualidade.

O processo de controle de qualidade tem seu próprio conjunto de procedimentos e relatórios, que deve ser utilizado durante o desenvolvimento de software. Esses procedimentos devem ser diretos e de fácil compreensão pelos engenheiros que estão desenvolvendo o software.

Há duas abordagens complementares para o controle de qualidade:

1. As revisões de qualidade, nas quais o software, sua documentação e os processos utilizados para produzi-lo são revisados por um grupo de pessoas. Elas são responsáveis por conferir se os padrões de projeto foram seguidos e se o software e os documentos estão

em conformidade com esses padrões. Os desvios em relação aos padrões são anotados e submetidos à atenção da gerência do projeto.

2. Avaliação automática de software, pela qual o software e os documentos produzidos são processados por algum programa e comparados com os padrões que se aplicam a esse projeto de desenvolvimento em particular. Essa avaliação automática pode envolver a medição quantitativa de alguns atributos de software. A medição e as métricas de software serão discultidas na Seção 24.4.

24.3.1 Revisões de qualidade

As revisões são métodos amplamente utilizados para a validação da qualidade de um processo ou produto. Elas envolvem um grupo de pessoas que examina parte ou todo o processo de software, o sistema ou sua documentação associada, a fim de descobrir possíveis problemas. As conclusões da revisão são formalmente registradas e passadas para o autor ou para quem for o responsável por corrigir os problemas constatados.

A Figura 24.8 descreve brevemente vários tipos diferentes de revisão. As inspeções de programa já foram abordadas no Capítulo 19. As revisões de progresso fazem parte do processo de gerenciamento, como foi discutido no Capítulo 4. Neste capítulo, concentro-me nas revisões como parte do processo de gerenciamento de qualidade. Os diferentes processos de revisão têm muito em comum, e já foi descrito o processo de definição de uma revisão no Capítulo 19.

O propósito da equipe de revisão é detectar erros e inconsistências e mostrá-los ao projetista ou ao autor do documento. As revisões são baseadas em documentos, mas não se limitam a especificações, projetos ou código. Documentos como modelos de processo, planos de teste, procedimentos de gerenciamento de configuração, padrões de processo e manuais de usuário também podem ser revisados.

A equipe de revisão deve incluir os membros de projeto que possam prestar uma contribuição eficaz. Por exemplo, se um projeto de subsistema está sendo revisado, projetistas de subsistemas devem ser incluídos na equipe de revisão. Eles podem fazer importantes comentários sobre as interfaces de subsistemas, que poderiam ser omitidos se o subsistema fosse considerado isoladamente.

A equipe de revisão deve ter um núcleo de 3 a 4 pessoas selecionadas como revisores principais. Um dos membros deve ser um projetista sênior, que possa assumir a responsabilidade de tomar decisões técnicas importantes. Os revisores principais podem convidar outros membros de projeto para contribuir na revisão. Eles não podem se envolver na revisão de todo o documento. Em vez disso, eles se concentram naquelas partes que afetam seu trabalho. Como alternativa, a equipe de revisão pode fazer circular o documento que está sendo revisado e pedir comentários por escrito de outros membros do projeto.

Os documentos a serem revisados devem ser distribuídos bem antes da revisão, para que os revisores tenham tempo de lê-los e compreendê-los. Embora um atraso possa perturbar o processo de desenvolvimento, a revisão não é eficaz se a respectiva equipe não tiver compreendido adequadamente os documentos antes de fazer a revisão.

A revisão em si deve ser relativamente breve (duas horas, no máximo). O autor do documento em revisão deve 'percorrer' o documento junto com a equipe de revisão. Um dos membros da equipe deve orientar a revisão e outro deve registrar formalmente todas as decisões sobre a revisão. Durante a revisão, o orientador é responsável por assegurar que todos os comentários escritos sejam considerados. Ao completar a revisão, as ações são anotadas e os formulários que registram os comentários e as ações devem ser assinados pelo projetista e pelo responsável pela revisão. Esses documentos são então arquivados como parte da documentação formal do projeto. Se apenas problemas menores forem constatados, uma outra revisão poderá ser desnecessária. O orientador é o responsável por assegurar que as mudanças requeridas sejam feitas. Se mudanças importantes forem necessárias, uma revisão de acompanhamento poderá ser programada.

Figura 24.8 ▶
Tipos de revisão.

Tipo de revisão	Propósito principal
Inspeções de projeto ou programa	Detectar erros detalhados nos requisitos, nos projetos ou no código. A revisão deve ser orientada por uma checklist de possíveis erros.
Revisões de progresso	Fornecer informações à gerência sobre o progresso geral do projeto. Essa é uma revisão de processo e de produto, e se preocupa com custos, planos e prazos.
Revisões de qualidade	Realizar uma análise técnica dos componentes ou da documentação do produto, a fim de encontrar inconsistências entre a especificação e o projeto, código ou documentação dos componentes e garantir que os padrões de qualidade definidos foram seguidos.

24.4 Medição e métricas de software

A medição de software se ocupa em obter um valor numérico para alguns atributos de um produto ou de um processo de software. Comparando esses valores uns com os outros e com os padrões que se aplicam em uma organização, é possível tirar conclusões sobre a qualidade do software ou dos processos de software. Por exemplo, digamos que uma organização planeje introduzir uma nova ferramenta de teste de software. Antes de introduzir a ferramenta, o número de defeitos de software descobertos em um determinado tempo pode ser registrado; depois de introduzi-la, esse processo é repetido. Se mais defeitos forem descobertos nos mesmos intervalos de tempo, depois de introduzir a ferramenta, então, aparentemente, ela oferece um suporte útil para o processo de validação de software.

Uma série de grandes empresas, como a Hewlett-Packard (Grady, 1993) e a AT&T (Barnard e Price, 1994), introduziram programas de métricas e estão utilizando métricas coletadas em seus processos de gerenciamento de qualidade. A maior parte da abordagem tem sido no sentido de coletar métricas relativas a defeitos de programas e processos de verificação e validação. Offen e Jeffrey (1997) e Hall e Fenton (1997) discutem a introdução desses programas na indústria. O manual denominado ami (Pulford *et al.*, 1996) fornece uma orientação detalhada sobre a medição e o uso da medição para a melhoria de processo.

Contudo, o uso da medição e de métricas sistemáticas de software ainda é relativamente incomum. Existe uma relutância em introduzir a medição, porque os benefícios não são bem definidos. Uma razão para isso é que, em muitas empresas, os processos de software utilizados ainda são organizados de maneira precária e não estão suficientemente aperfeiçoados para fazer uso de medições. Outra razão é que não há padrões para as métricas, e daí decorre o suporte limitado para a coleta e análise de dados. A maioria das empresas não estará preparada para introduzir medições até que esses padrões e essas ferramentas estejam disponíveis.

Uma métrica de software é qualquer tipo de medição que se refira a um sistema de software, processo ou documentação relacionada. Exemplos disso são as medidas do tamanho de um produto em linhas de código, o índice Fog (Gunning, 1962), que é uma medida do grau de facilidade de leitura de um trecho de texto escrito, o número de defeitos relatados em um produto de software fornecido e o número de pessoas-dia requerido para desenvolver um componente de sistema.

As métricas podem ser de controle ou preditivas. As métricas de controle são geralmente associadas a processos de software; as métricas preditivas são associadas a produtos de software. Exemplos de métricas de controle ou de processo são o esforço e o tempo médio requerido para reparar defeitos relatados. Exemplos de métricas preditivas são a complexidade ciclomática de um módulo, o comprimento médio de identificadores em um programa e o número de atributos e operações associadas com objetos em um projeto. Tanto a métrica de controle como a

preditiva podem influenciar na tomada de decisões de gerenciamento, como mostra a Figura 24.9. As métricas de processo e seu papel na melhoria do processo serão discutidos no Capítulo 25. A abordagem aqui é quanto à medição para prever a qualidade de um produto de software.

Com freqüência, é impossível medir os atributos da qualidade de software diretamente. Atributos como facilidade de manutenção, complexidade e facilidade de compreensão são afetados por muitos fatores diferentes, e não existem métricas diretas e simples para eles. Em vez disso, devemos medir algum atributo interno do software (como seu tamanho) e supor que exista uma relação entre o que podemos medir e o que desejamos saber. O ideal é que haja uma relação clara e validada entre os atributos de software internos e externos.

A Figura 24.10 mostra alguns atributos externos de qualidade, que podem ser de interesse, e atributos internos, que podem ser medidos e relacionados com atributos externos. O diagrama sugere que pode haver uma relação entre os atributos externos e internos, mas não diz qual relação é essa. Se a medida do atributo interno deve ser uma previsão útil da característica externa do software, três condições devem prevalecer (Kitchenham, 1990a):

1. O atributo interno deve ser medido precisamente.
2. Deve existir uma relação entre o que podemos medir e o atributo de comportamento externo.
3. Essa relação é compreendida, foi validada e pode ser expressa em uma fórmula ou um modelo.

A formulação de modelo envolve identificar o modo funcional do modelo (linear, exponencial etc.) pela análise dos dados coletados, identificando os parâmetros que devem ser incluídos no modelo e ajustando-os por meio dos dados existentes. Esse desenvolvimento de modelo,

Figura 24.9

Métricas preditivas e de controle.

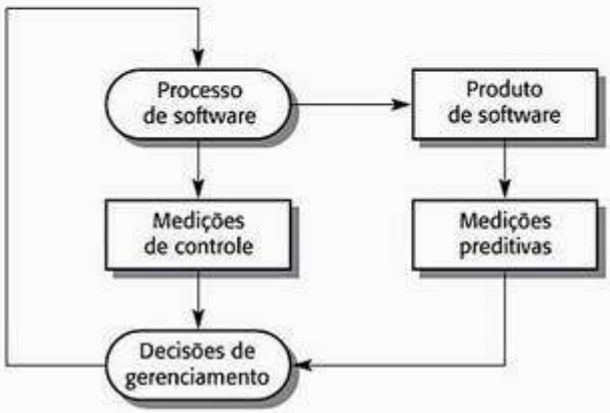
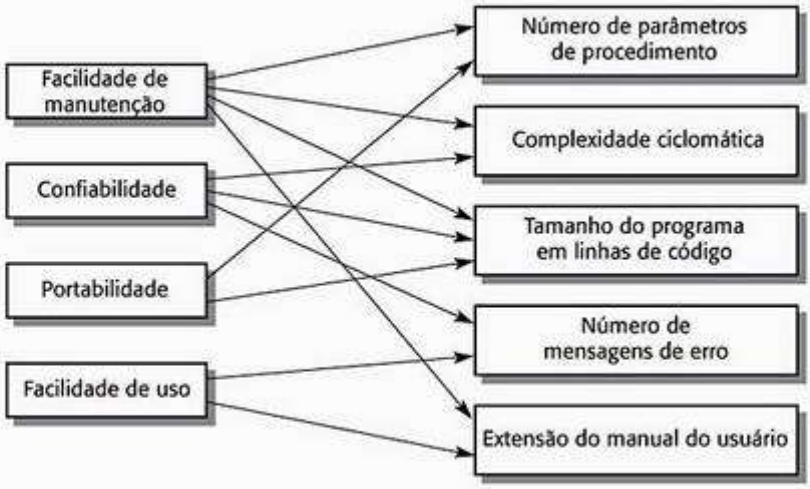


Figura 24.10

Relações entre os atributos internos e externos de software.



para ser confiável, exige experiência significativa nas técnicas estatísticas. Um especialista em estatística deve estar envolvido no processo.

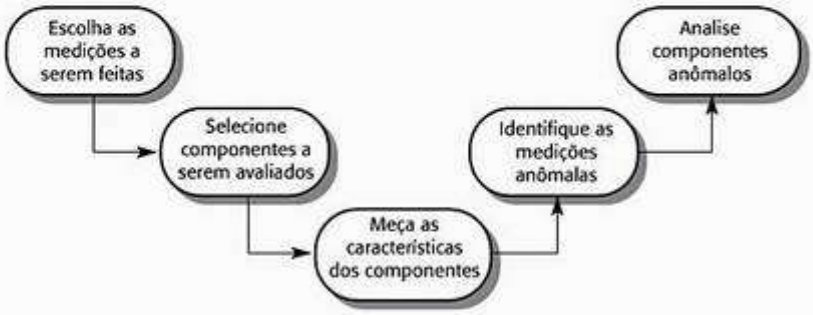
24.4.1 O processo de medição

Um processo de medição de software que pode ser parte de um processo de controle de qualidade é mostrado na Figura 24.11. Cada um dos componentes do sistema é analisado separadamente, e os diferentes valores da métrica devem ser comparados entre si e, talvez, com os dados históricos de medição coletados em projetos precedentes. Medições anômalas devem ser utilizadas para enfocar o esforço de garantia de qualidade nos componentes que possam apresentar problemas de qualidade.

Os principais estágios desse processo são:

1. *Escolha de medições a serem feitas* Devem ser formuladas as questões às quais as medições se destinam a responder, e as medições exigidas para responder a essas questões devem ser definidas. As medições que não forem diretamente relevantes para essas questões não precisam ser coletadas. O paradigma de GQM (*goal-question-metric* – objetivo-questão-métrica), de Basili (Basili e Rombach, 1988), discutido no capítulo seguinte, é uma boa abordagem, que deve ser utilizada quando se decide que dados devem ser coletados.
2. *Seleção de componentes a serem avaliados* Pode não ser necessário ou desejável avaliar valores de métricas para todos os componentes em um sistema de software. Em alguns casos, uma seleção representativa de componentes pode ser escolhida para a medição. Em outros casos, podem ser avaliados os componentes que são particularmente importantes, como os componentes centrais, que estão em uso quase constante.
3. *Medição de características dos componentes* Os componentes selecionados são medidos e os valores de métricas são computados. Isso envolve processar a representação dos componentes (projeto, código etc.) utilizando uma ferramenta automatizada de coleta de dados. Isso pode ser especialmente escrito ou já pode estar incorporado nas ferramentas CASE, que são utilizadas em uma organização.
4. *Identificação de medições anômalas* Uma vez feitas as medições de componentes, elas devem ser comparadas entre si e com as medições precedentes, que tenham sido registradas em um banco de dados de medições. É preciso procurar valores altos ou baixos incomuns para cada métrica, uma vez que isso sugere que pode haver problemas com o componente que apresenta esses valores.
5. *Análise de componentes anômalos* Uma vez identificados os componentes com valores anômalos em métricas particulares, esses componentes devem ser examinados para decidir se os valores anômalos significam que a qualidade do componente está comprometida ou não. Um valor anômalo para complexidade (digamos) não significa necessariamente um componente de baixa qualidade. É possível que haja alguma outra razão para o valor alto, e pode não significar que haja problemas com a qualidade do componente.

Figura 24.11 ▶
Processo de medição de produto.



Os dados coletados devem ser mantidos como um recurso organizacional, e os registros históricos de todos os projetos devem ser mantidos mesmo quando os dados tenham sido utilizados durante um determinado projeto. Uma vez estabelecido um banco de dados de medição suficientemente grande, as comparações entre projetos podem ser efetuadas e métricas específicas podem ser aprimoradas, de acordo com as necessidades organizacionais.

24.4.2 Métricas de produto

As métricas de produto se ocupam com as características do próprio software. Infelizmente, as características de software que podem ser medidas facilmente, como tamanho e complexidade ciclomática, não têm uma relação clara e universal com os atributos de qualidade, como facilidade de compreensão e manutenção. As relações variam, dependendo dos processos de desenvolvimento e da tecnologia e do tipo de sistema que está sendo desenvolvido. As organizações interessadas em medições precisam construir um banco de dados, que pode, então, ser utilizado para descobrir como os atributos do produto de software estão relacionados às qualidades de interesse para a organização.

As métricas de produtos se dividem em duas classes:

1. Métricas dinâmicas, que são coletadas por medições feitas de um programa em execução.
2. Métricas estáticas, que são coletadas por medições feitas das representações do sistema, como projeto, programa ou documentação.

Esses diferentes tipos de métricas estão relacionados com diferentes atributos de qualidade. As métricas dinâmicas ajudam a avaliar a eficiência e a confiabilidade de um programa, enquanto as métricas estáticas ajudam a medir a complexidade e a facilidade de compreensão e manutenção de um sistema de software.

As métricas dinâmicas em geral estão relacionadas bem estreitamente com os atributos de qualidade do software. É relativamente fácil medir o tempo de execução para determinadas funções e avaliar o tempo requerido para iniciar um sistema. Esses itens se relacionam diretamente com a eficiência do sistema. De maneira similar, o número de falhas de sistema e o tipo de falhas podem ser registrados em uma planilha e relacionados diretamente com a confiabilidade do software, como foi discutido no Capítulo 16.

As métricas estáticas, por outro lado, têm uma relação indireta com os atributos de qualidade. Um grande número dessas métricas foi proposto e experimentos foram realizados, a fim de tentar obter e validar as relações entre essas métricas e a complexidade, a facilidade de compreensão e a facilidade de manutenção do sistema. A Figura 24.12 descreve várias métricas estáticas utilizadas para avaliar atributos de qualidade. Naturalmente, a extensão do programa ou componente e a complexidade de controle parecem ser as previsões mais confiáveis quanto à facilidade de compreensão, a complexidade e a facilidade de manutenção do sistema.

Desde o início da década de 90, tem havido uma série de estudos de métricas orientadas a objetos. Algumas delas foram derivadas a partir das métricas mais antigas, mostradas na Figura 24.12, mas outras são exclusivas de sistemas orientados a objetos. Uma série de métricas orientadas a objetos está explicada na Figura 24.13. Essas métricas são menos aprimoradas do que as mostradas na Figura 24.12, que se destinam a projetos orientados a funções, e assim sua utilidade como métricas preditivas ainda está sendo estabelecida.

As métricas específicas que são relevantes dependem do projeto, das metas da equipe de gerenciamento de qualidade e do tipo de software que está sendo desenvolvido. Todas as métricas mostradas nas figuras 24.12 e 24.13 podem ser úteis em algumas situações. Igualmente, contudo, haverá outras situações em que elas não serão apropriadas. Ao introduzir a medição de software como parte do processo de gerenciamento de qualidade, as organizações devem procurar descobrir as métricas mais apropriadas para suas necessidades.

Figura 24.12 ▶

Métricas de produto de software.

Métricas de software	Descrição
Fan-in/Fan-out	*Fan-in* é uma medida do número de funções que chamam outra função (digamos X). *Fan-out* é o número de funções que são chamadas pela função X. Um valor alto para *fan-in* significa que X está estreitamente acoplado ao restante do projeto e que mudanças em X terão amplos efeitos de impacto. Um valor alto para *fan-out* sugere que a complexidade geral de X pode ser alta, devido à complexidade da lógica de controle, necessária para coordenar os componentes chamados.
Tamanho do código	Essa é uma medida do tamanho de um programa. Geralmente, quanto maior o tamanho do código de um componente de programa, mais complexo e sujeito a erros esse componente provavelmente será.
Complexidade ciclomática	Essa é uma medida da complexidade de controle de um programa. Essa complexidade de controle pode ser relacionada com a facilidade de compreensão do programa. O cálculo da complexidade ciclomática foi abordado no Capítulo 20.
Extensão dos identificadores	Essa é uma medida do comprimento médio de identificadores distintos em um programa. Quanto mais extensos os identificadores, mais provável é que eles tenham significado e, portanto, mais compreensível será o programa.
Profundidade de declarações condicionais aninhadas	Essa é uma medida da profundidade de declarações if aninhadas em um programa. As declarações if de muita profundidade são de difícil compreensão e estão potencialmente sujeitas a erros.
Índice Fog	Essa é uma medida do comprimento médio de palavras e sentenças em documentos. Quanto mais alto esse valor para o índice Fog, maior dificuldade o documento poderá apresentar em sua compreensão.

Figura 24.13 ▶

Métricas orientadas a objetos.

Métricas orientadas a objetos	Descrição
Profundidade da árvore de herança	Representa o número de níveis distintos na árvore de herança, em que as subclasses herdam atributos e operações (métodos) das superclasses. Quanto mais profunda a árvore de herança, mais complexo o projeto, uma vez que, potencialmente, muitas classes diferentes de objeto precisam ser compreendidas para que se compreendam as classes de objetos nas folhas da árvore.
Método de *fan-in/fan-out*	Está relacionado de maneira direta ao *fan-in* e ao *fan-out*, descritos acima, e significa, essencialmente, a mesma coisa. Contudo, pode ser apropriado fazer uma distinção entre as chamadas de outros métodos dentro do objeto e as chamadas de métodos externos.
Métodos ponderados por classe	É o número de métodos incluídos em uma classe ponderada pela complexidade de cada método. Por conseguinte, um único método pode ter uma complexidade de 1, e um método grande e complexo, um valor muito mais alto. Quanto maior o valor para essa métricas, mais complexa a classe de objeto. Objetos complexos provavelmente são mais difíceis de compreender. Podem não ser logicamente coesos e, assim, não ser reutilizados de maneira efetiva como superclasses em uma árvore de herança.
Número de operações sobrepostas	Esse é o número de operações em uma superclasse, que são sobrepostas em uma subclasse. Um alto valor para essa métrica indica que a superclasse utilizada pode não ser uma classe principal apropriada para a subclasse.

24.4.3 Análise de medições

Um dos problemas com a coleta de dados quantitativos sobre software e projetos de software está em compreender o que esses dados realmente significam. É fácil interpretar mal os dados e produzir inferências incorretas. As medições devem ser cuidadosamente analisadas para se compreender o que elas de fato significam.

Para ilustrar como os dados coletados podem ser interpretados de diferentes maneiras, considere a seguinte situação:

> Um gerente decide monitorar o número de pedidos de modificação submetidos pelos clientes, supondo que exista uma relação entre esses pedidos e a facilidade de uso e adequação do produto. Quanto mais alto o número de pedidos de mudança, menos o software atenderá às necessidades do cliente.
>
> Processar os pedidos de mudança e modificar o software é dispendioso. Portanto, a organização decide modificar seu processo, a fim de aumentar a satisfação do cliente e, ao mesmo tempo, reduzir os custos de mudanças. Pretende-se que as mudanças de processo resultem em melhores produtos e menos pedidos de modificações.
>
> As mudanças de processo são iniciadas, o que exige mais envolvimento do cliente no processo de projeto de software. Os testes beta de todos os produtos são introduzidos e as modificações solicitadas pelos clientes são incorporadas. São entregues novas versões de produtos, desenvolvidas com esse processo modificado. Em alguns casos, o número de pedidos de modificação é reduzido; em outros casos, ele é aumentado. O gerente fica sem entender, e não pode medir os efeitos das mudanças de processo sobre a qualidade do produto.

Para compreender por que isso pode acontecer, é preciso compreender por que os pedidos de mudança são feitos. Uma razão é que o software fornecido não faz o que os clientes desejam que ele faça. Outra possibilidade é que o software é muito bom e ampla e intensamente utilizado com as finalidades para as quais ele não foi originalmente projetado. Pelo fato de ter tanta gente fazendo uso dele, é natural que mais pedidos de modificações sejam gerados.

Uma terceira possibilidade é que a empresa que produz o software seja suscetível aos pedidos de mudança dos clientes. Os clientes ficam, portanto, satisfeitos com o serviço que recebem e geram muitos pedidos de mudança porque sabem que esses pedidos serão levados a sério. Suas sugestões serão provavelmente incorporadas em versões posteriores do software.

O número de pedidos de mudança pode diminuir porque as mudanças de processo foram eficazes e tornaram o software mais útil e adequado. Alternativamente, o número pode ter diminuído porque o produto perdeu participação no mercado para um produto rival. Conseqüentemente, existem menos usuários para o produto. O número de pedidos de mudança pode aumentar porque há mais usuários, porque o processo dos testes beta convenceu os usuários de que a empresa está disposta a fazer mudanças ou porque as configurações de testes beta não eram típicas para o maior uso do programa.

Para analisar os dados dos pedidos de mudança, não é preciso saber apenas o número de pedidos. Precisamos saber quem fez o pedido, como eles utilizam o software e por que o pedido foi feito. Precisamos saber se há fatores externos, como alterações no procedimento de um pedido de mudança ou mudanças no mercado, que possam ter algum efeito. Com essas informações, é possível descobrir se as mudanças de processo foram eficazes em aumentar a qualidade do produto.

Isso ilustra que a interpretação de dados quantitativos sobre um produto ou processo é um método incerto. Os processos e os produtos que estão sendo medidos não estão isolados de seus ambientes, e as mudanças nesse ambiente podem tornar inválidas as comparações de dados. Os dados quantitativos sobre as atividades humanas não podem ser sempre considerados pelo seu valor em si. As razões básicas que podem explicar o valor medido devem ser investigadas.

PONTOS-CHAVE

- O gerenciamento de qualidade de software se ocupa de garantir que o software tenha um pequeno número de defeitos e que atinja os padrões requeridos de facilidade de manutenção, confiabilidade, portabilidade etc. As atividades de gerenciamento de qualidade incluem a garantia de qualidade, que estabelece os padrões para o desenvolvimento de software, o planejamento de qualidade e o controle de qualidade, que verificam o software em relação aos padrões definidos.
- Um manual de qualidade organizacional deve documentar um conjunto de procedimentos de garantia de qualidade. Ele pode ser baseado no modelo genérico sugerido nos padrões ISO 9000.
- Os padrões de software são importantes para a garantia de qualidade, uma vez que representam uma identificação da 'melhor prática'. O processo de controle de qualidade se ocupa de verificar se o processo de software e o software em desenvolvimento estão em conformidade com esses padrões.
- As revisões dos produtos relacionados ao processo de software são a técnica mais amplamente utilizada para avaliar a qualidade.
- As medições de software podem ser utilizadas para coletar dados quantitativos sobre o software e seu processo. Os valores das métricas de software, que são coletados, podem ser utilizados para fazer inferências sobre a qualidade de produto e de processo.
- As métricas de qualidade de produto são particularmente valiosas para identificar componentes anômalos que possam apresentar problemas de qualidade. Esses componentes deverão ser analisados com mais detalhes.
- Não existem métricas de software padronizadas e universalmente aplicáveis. As organizações precisam selecionar suas métricas e analisar as medições feitas baseadas no conhecimento e em circunstâncias locais.

LEITURAS SUGERIDAS

"Making sense of measurement for small organizations". Esse é um artigo interessante sobre a aplicação prática de métricas. Aponta a questão de que todos os usos de métricas precisam levar em conta seu próprio contexto. (K. Kautz, *IEEE Software*, mar./abr. 1999.)

ISO 9000 and software quality assurance. Essa é uma introdução interessante à ISO 9000 e sua relevância para a garantia de qualidade de software. Sua estrutura baseia-se no padrão de qualidade, com um capítulo para cada tópico. (D. Ince, 1994, McGraw-Hill.)

IEEE software, mar. 1997. Essa é uma edição especial sobre a medição de software, que contém artigos particularmente interessantes sobre a introdução de um programa de medição.

A quantitative approach to software management: the ami handbook. É um excelente guia sobre 'como fazer', que discute como introduzir um programa de medição e utilizar os resultados para a melhoria de processo. (K. Pulford, A. Kuntzmann-Combelles e S. Shirlaw, 1996, Addison Wesley.)

EXERCÍCIOS

24.1 Explique por que um processo de software de alta qualidade pode levar a produtos de software de alta qualidade. Discuta os possíveis problemas com esse sistema de gerenciamento de qualidade.

24.2 Quais são os estágios envolvidos na revisão de um projeto de software?

24.3 Discuta a avaliação de qualidade de software de acordo com os atributos de qualidade mostrados na Figura 24.7. Você deve considerar um atributo por vez e explicar como ele pode ser avaliado.

24.4 Projete um formulário eletrônico que possa ser utilizado para registrar comentários de revisão e para remeter esses comentários eletronicamente aos revisores.

24.5 Descreva brevemente os padrões que podem ser utilizados para:

- o uso de construções de controle em C, C++ e Java;
- relatórios que possam ser submetidos a um projeto de período letivo em uma universidade;
- o processo de fazer e aprovar mudanças em um programa (veja o Capítulo 29), e
- o processo de comprar e instalar um novo computador.

24.6 Suponha que você trabalhe para uma organização que desenvolve produtos de banco de dados para sistemas de microcomputadores. A organização está interessada em quantificar seu desenvolvimento de software. Escreva um relatório propondo métricas apropriadas e sugira como os dados podem ser coletados.

24.7 Explique por que as métricas de projeto são, por si mesmas, um método inadequado para prever a qualidade de projeto.

24.8 Consulte a literatura e descubra outras métricas de qualidade de projeto que tenham sido sugeridas, além daquelas que foram discutidas aqui. Considere essas métricas em detalhes e avalie se elas podem ser de real valor.

24.9 Os padrões de software reprimem a inovação tecnológica?

24.10 Um colega, que é um programador muito bom, produz software com um pequeno número de defeitos, mas, consistentemente, ignora os padrões de qualidade organizacional. Como os gerentes da organização devem reagir a esse comportamento?

25 Melhoria de processo

Objetivos

O objetivo deste capítulo é explicar como os processos de software podem ser melhorados, a fim de produzirem software de melhor qualidade. Depois de ler este capítulo, você poderá:

- compreender os princípios da melhoria de processo de software e por que ela vale a pena;
- compreender como os fatores do processo de software influenciam na qualidade de software e na produtividade dos desenvolvedores de software;
- compreender o Modelo de Maturidade de Capacitação (*Capability Maturity Model* – CMM), processo do SEI, que pode ser utilizado para avaliar a qualidade do processo de software em grandes organizações, e
- compreender por que a melhoria baseada em CMM não é aplicável a todos os tipos de processo de software.

Sumário

25.1 Qualidade de processo e de produto
25.2 Modelagem e análise de processo
25.3 Medições de processo
25.4 Modelo de Maturidade de Capacitação do SEI
25.5 Classificação de processo

Durante os últimos anos, tem havido bastante interesse, por parte da comunidade de engenharia de software, pela melhoria de processo. A melhoria do processo significa compreender os processos existentes e modificá-los, a fim de melhorar a qualidade do produto e/ou reduzir os custos e o tempo de desenvolvimento. A maior parte da literatura relacionada a esse assunto tem se concentrado em aprimorar os processos para melhorar a qualidade do produto e, em particular, para reduzir o número de defeitos nos softwares fornecidos. Uma vez que esse objetivo é alcançado, a redução dos custos e do tempo pode se tornar a principal meta da melhoria.

Como foi discutido no Capítulo 24, existe uma forte relação entre a qualidade do produto de software desenvolvido e a qualidade do processo de software utilizado para criar esse produto. Melhorando o processo de software, espera-se que a qualidade do produto relacionado seja aperfeiçoada de modo correspondente. Os processos de software são inerentemente complexos e envolvem um número muito grande de atividades. Assim como os produtos, os processos também têm atributos ou características, como mostra a Figura 25.1.

Não é possível obter melhorias de processo que otimizem todos os atributos de processo simultaneamente. Por exemplo, se um rápido processo de desenvolvimento for exigido, então, pode ser necessário reduzir a visibilidade do processo. Tornar um processo visível significa produzir documentos em intervalos regulares, e isso retardará o processo.

A melhoria de processo não significa simplesmente adotar métodos ou ferramentas específicos ou algum modelo de processo que tenha sido utilizado em outro lugar. Embora as organizações que desenvolvem o mesmo tipo de software claramente tenham muito em comum, sempre há fatores locais organizacionais, procedimentos e padrões que influenciam o processo. A simples introdução de melhorias de processo divulgadas, provavelmente, não terá sucesso. As melhorias de processo devem ser sempre vistas como uma atividade específica de uma organização ou uma parte de uma organização maior.

Um modelo genérico da melhoria de processo está ilustrado na Figura 25.2. A melhoria de processo é um processo iterativo, de longo prazo. Cada uma das atividades mostradas na Figura 25.2 pode durar vários meses. O sucesso da melhoria de processo requer o comprometimento organizacional e a disponibilidade de recursos. Ela precisa ser sancionada pela gerência sênior e deve ter um orçamento associado, para apoiar as melhorias.

Figura 25.1 ▶
Características do processo.

Característica do processo	Descrição
Facilidade de compreensão	Até que ponto o processo está explicitamente definido e com que facilidade se pode compreender a definição do processo?
Visibilidade	As atividades de processo culminam em resultados nítidos, de modo que o progresso do processo seja externamente visível?
Facilidade de suporte	Até que ponto as atividades do processo podem ser apoiadas por ferramentas CASE?
Aceitabilidade	O processo definido é aceitável e utilizável pelos engenheiros responsáveis pela produção do produto de software?
Confiabilidade	O processo está projetado de tal maneira que seus erros sejam evitados ou identificados antes que resultem em erros no produto?
Robustez	O processo pode continuar, mesmo que surjam problemas inesperados?
Facilidade de manutenção	O processo pode evoluir para refletir os requisitos mutáveis da organização ou melhorias de processo identificadas?
Rapidez	Com que rapidez pode ser concluído o processo de entrega de um sistema, a partir de uma determinada especificação?

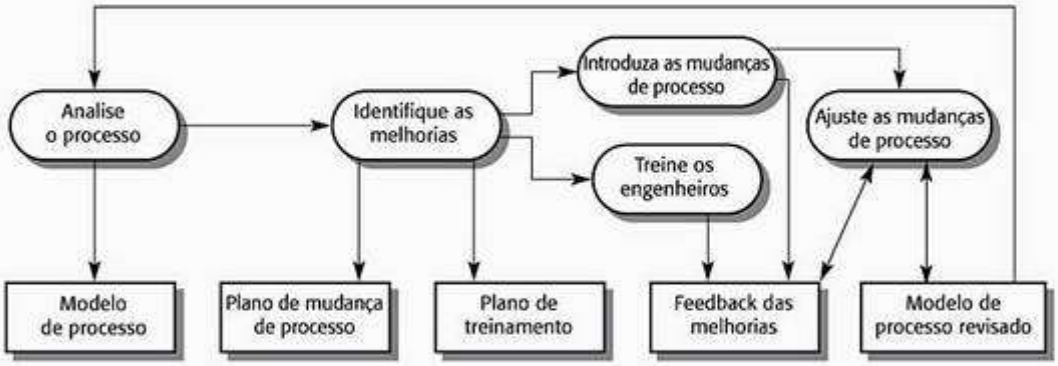


Figura 25.2 ▲
Modelo de melhoria de processo.

Há uma série de estágios importantes na melhoria de processo:

1. *Análise de processo* A análise de processo envolve examinar os processos existentes e produzir um modelo específico para documentar e compreender o processo. Em alguns casos, pode ser possível analisar o processo quantitativamente. As medições feitas durante a análise acrescentam informações extras ao modelo de processo. A análise quantitativa antes e depois de as mudanças terem sido introduzidas permite uma avaliação objetiva dos benefícios (ou dos problemas) da mudança de processo.

2. *Identificação de melhoria* Esse estágio se ocupa de utilizar os resultados da análise de processo para identificar gargalos relativos à qualidade, ao prazo e ao custo, em que os fatores de processo tenham adversamente influenciado a qualidade do produto. A melhoria de processo deve enfocar a eliminação desses gargalos, propondo novos procedimentos, métodos e ferramentas para corrigir os problemas.

3. *Introdução de mudança de processo* A introdução de uma mudança de processo significa implantar novos procedimentos, métodos e ferramentas, e integrá-los com outras atividades de processo. É importante dar tempo suficiente para introduzir as mudanças e garantir que elas sejam compatíveis com outras atividades de processo e com os procedimentos e padrões organizacionais.

4. *Treinamento em mudanças de processo* Sem treinamento, não é possível obter os plenos benefícios das mudanças de processo. Elas podem ser rejeitadas pelos gerentes e engenheiros responsáveis pelos projetos de desenvolvimento. É muito comum que as mudanças de processo sejam impostas sem treinamento adequado e que os efeitos dessas mudanças resultem na degradação e não na melhoria da qualidade do produto.

5. *Ajuste de mudanças* As mudanças de processo propostas nunca serão inteiramente eficazes assim que forem introduzidas. Há a necessidade de uma fase de ajuste, em que problemas menores sejam descobertos e modificações no processo sejam propostas e introduzidas. Essa fase de ajuste pode durar vários meses, até que os engenheiros de desenvolvimento estejam satisfeitos com o novo processo.

Uma vez introduzida uma mudança, o processo de melhoria pode iterar com outras análises utilizadas, a fim de identificar problemas de processo, propor melhorias etc. É impraticável introduzir muitas mudanças ao mesmo tempo. Além dos problemas de treinamento, a introdução de muitas mudanças torna impossível avaliar o efeito de cada uma delas no processo.

25.1 Qualidade de processo e de produto

Como foi discutido no Capítulo 24, a melhoria de processo se baseia na suposição de que o principal fator que influencia a qualidade do produto é o processo de desenvolvimento do produto. Essas idéias sobre o desenvolvimento de processo surgiram a partir do trabalho de um engenheiro norte-americano chamado W.E. Deming, que trabalhou na indústria japonesa, depois da Segunda Guerra Mundial, buscando a melhoria da qualidade. Há muitos anos, a indústria japonesa está comprometida com a iniciativa de melhoria contínua de processo (a palavra japonesa para isso é *kaizen*). Isso contribuiu amplamente para a alta qualidade geral dos produtos manufaturados japoneses.

Deming (e outros) introduziu a idéia do controle estatístico de qualidade, que se baseia em medir o número de defeitos de um produto e relacioná-los ao processo. O processo é aperfeiçoado com o objetivo de reduzir o número de defeitos do produto e até que ele seja repetível. Isto é, os resultados do processo são previsíveis e o número de defeitos é reduzido. Ocorre a padronização e começa outro ciclo de melhoria.

Quando a manufatura está envolvida, a relação processo/produto é óbvia. Melhorar um processo de maneira que os defeitos sejam evitados levará à produção de melhores produtos. Esse vínculo é menos óbvio quando o produto é intangível e dependente, até certo ponto, de processos intelectuais, que não podem ser automatizados. A qualidade do software não é dependente de um processo de manufatura, mas de um processo de projeto, em que as considerações individuais humanas são importantes. Em algumas classes de produtos, o processo utilizado é o determinante mais significativo da qualidade do produto. Contudo, para aplicações inovadoras, em particular, as pessoas envolvidas no processo podem ser mais importantes do que o processo utilizado.

Para os produtos de software (ou quaisquer outros produtos intelectuais, como livros, filmes etc., em que a qualidade depende principalmente do projeto), existem quatro fatores que podem afetar a sua qualidade. Eles são mostrados na Figura 25.3.

A influência de cada um desses fatores depende do tamanho e do tipo do projeto. Para sistemas muito grandes, compostos de subsistemas separados, desenvolvidos por diferentes equipes, o determinante principal da qualidade do produto é o processo de software. Os principais problemas com grandes projetos são referentes à integração, ao gerenciamento de projeto e à comunicação. Em geral, existe uma mistura de capacitações e experiências nos membros da equipe e, como o processo de desenvolvimento normalmente ocorre durante vários anos, a equipe de desenvolvimento é volátil. Ela pode mudar inteiramente durante o tempo de duração do projeto. Portanto, os indivíduos particularmente habilitados e talentosos em geral não exercem um efeito dominante ao longo da duração do projeto.

Para os pequenos projetos, contudo, quando há apenas poucos membros na equipe, a qualidade da equipe de desenvolvimento é mais importante do que o processo de desenvolvimento utilizado. Se a equipe tiver um alto nível de capacitação e experiência, a qualidade do produto provavelmente será alta. Se a equipe for inexperiente e menos capacitada, um bom processo poderá limitar o dano, mas, por si só, não produzirá um software de alta qualidade.

Quando as equipes são pequenas, a boa tecnologia de desenvolvimento é particularmente importante. A equipe não pode devotar muito tempo a procedimentos administrativos tediosos. Os engenheiros gastam a maior parte de seu tempo projetando e programando o sistema; assim, boas ferramentas podem afetar significativamente sua produtividade. Para os grandes projetos, um nível básico de tecnologia de desenvolvimento é essencial para o gerenciamento das informações. Paradoxalmente, contudo, as ferramentas sofisticadas são menos importantes. Os membros da equipe gastam uma pequena parte de seu tempo em atividades de desenvolvimento e gastam mais tempo se comunicando e compreendendo outras partes do sistema. Esse é o fator dominante que afeta sua produtividade. As ferramentas de desenvolvimento não fazem diferença quanto a isso.

Figura 25.3 ▶

Principais fatores da qualidade de produtos de software.

A base do retângulo, na Figura 25.3, é fundamental. Se um projeto, independentemente de seu tamanho, tiver um orçamento baixo ou o planejamento do cronograma de entrega não for realista, a qualidade do produto será afetada. Um bom processo requer recursos para sua implementação eficaz. Se esses recursos forem insuficientes, o processo não poderá operar efetivamente. Se os recursos forem inadequados, somente profissionais excelentes poderão salvar o projeto. Mesmo assim, se o déficit for muito grande, a qualidade do produto será prejudicada.

Freqüentemente, a causa real dos problemas de qualidade de software não é o gerenciamento ineficiente, processos inadequados ou um treinamento impróprio sobre a qualidade. Ao contrário, a causa é o fato de que as organizações precisam competir para sobreviver. Muitos projetos de software, deliberadamente, têm um orçamento pequeno para ganhar o contrato de desenvolvimento. O que se chama de 'preço definido para ganhar' (veja o Capítulo 23) é a conseqüência inevitável de um sistema competitivo. Não é de surpreender que a qualidade do produto, nesse sistema, seja de difícil controle.

25.2 Modelagem e análise de processo

A modelagem e a análise de processo envolvem o estudo dos processos existentes e o desenvolvimento de um modelo abstrato desses processos, que identifique suas principais características. Vários modelos genéricos de processo foram apresentados no Capítulo 1. Ao longo do livro, utilizei fragmentos de modelos de processo, a fim de discutir atividades específicas, como a engenharia de requisitos, o projeto de software etc. Huff (1996) apresenta uma boa visão do tópico geral da modelagem do processo de software.

A análise de processo se ocupa de estudar os processos existentes, a fim de compreender as relações entre as diferentes partes do processo. Os estágios iniciais da análise de processo são, inevitavelmente, qualitativos, em que o analista está tentando descobrir as principais características do modelo. Os estágios posteriores podem ser mais quantitativos. O processo é analisado em detalhes, com a utilização de várias métricas, que são coletadas automática ou manualmente. Depois da análise, o processo pode ser descrito, utilizando-se um modelo de processo.

O ponto de partida para a análise de processo deve ser qualquer modelo de processo 'formal' que seja utilizado. Muitas organizações têm esse modelo formal, que pode ser imposto a elas pelo cliente que solicitou o software. Esse padrão define as principais atividades e os produtos que devem ser gerados.

Os modelos formais podem servir como um ponto de partida útil para a análise de processo. Contudo, eles raramente incluem detalhes suficientes ou refletem as atividades reais do desenvolvimento de software. Os modelos formais de processo são bastante abstratos e somente definem as principais atividades de processo e os produtos a serem gerados. Em

geral, é necessário observar internamente o modelo para encontrar o processo real que está sendo aprovado. Além do mais, o processo real que é seguido freqüentemente difere de modo significativo do modelo formal, embora os produtos relacionados especificados, de modo geral, sejam produzidos.

As técnicas de análise de processo incluem:

1. *Questionários e entrevistas* Os engenheiros que trabalham em um projeto são questionados sobre o que realmente está em andamento. As respostas a um questionário formal são aprimoradas durante entrevistas pessoais com os envolvidos no processo.
2. *Estudos etnográficos* Como foi discutido no Capítulo 6, os estudos etnográficos podem ser utilizados para compreender a natureza do desenvolvimento de software como uma atividade humana. Essa análise revela sutilezas e complexidades que podem não ser descobertas utilizando-se outras técnicas.

Cada uma dessas abordagens tem vantagens e desvantagens. A análise com base em questionários pode ser realizada bem rapidamente, uma vez descobertas as perguntas certas. Contudo, se as perguntas forem mal formuladas ou inadequadas, pode-se terminar com um modelo incompleto ou inexato do processo. Além disso, a análise baseada em questionários se parece com um meio de avaliação. Os engenheiros que estão sendo questionados podem dar as respostas que eles acham que o avaliador queira ouvir, em vez de falar a verdade sobre o processo utilizado.

A análise etnográfica tem mais chance de descobrir o verdadeiro processo utilizado. Contudo, essa é uma atividade prolongada e dispendiosa, que pode durar vários meses. Ela depende da observação externa do processo. Uma análise completa deve continuar a partir dos estágios iniciais do projeto até a entrega do produto e sua manutenção. Provavelmente, isso é impraticável quando os projetos duram vários anos. A análise etnográfica, portanto, é muito útil quando é necessária uma compreensão profunda dos fragmentos do processo.

A saída da análise de processo é um modelo de processo que pode ser expresso com maior ou menor nível de detalhes. Os modelos de processo utilizados neste livro são abstratos, são modelos simplificados que apresentam uma visão genérica dos processos em questão. Nesse nível de abstração, esses processos são os mesmos em diferentes organizações. Contudo, esses modelos genéricos têm diferentes adaptações, dependendo do tipo de software que está sendo desenvolvido e do ambiente organizacional. Os modelos de processo detalhados geralmente não são transferíveis de uma organização para outra.

Os modelos genéricos de processo são uma base útil para se discutir sobre os processos. Contudo, eles não incluem informações suficientes para a análise e a melhoria do processo. A melhoria de processo precisa de informações sobre atividades, produtos gerados, pessoal, comunicações, cronogramas e outros processos organizacionais que afetam o processo de desenvolvimento de software. A Figura 25.4 explica o que pode ser incluído em um modelo detalhado de processo.

A sincronia e as dependências entre as atividades, os produtos gerados e as comunicações devem ser também registradas no modelo de processo. Atividades esporádicas podem ser conduzidas em paralelo ou, algumas vezes, em seqüência. Elas podem ser intercaladas, de modo que um mesmo engenheiro pode estar envolvido em várias atividades. Os produtos gerados podem ser dependentes uns dos outros ou de comunicações entre os engenheiros que trabalham no processo.

No exemplo de modelos de processo, apresentado neste livro, mostro a seqüência aproximada de atividades, da esquerda para a direita. As atividades que podem ser executadas em paralelo são alinhadas verticalmente, tanto quanto possível.

Os modelos detalhados de processos são extremamente complexos. É muito difícil construir um único modelo, que incorpore todos os elementos mostrados anteriormente. Para ilustrar a complexidade de um modelo, considere os fragmentos de processo a seguir. Eles descrevem o processo de teste de um único módulo em um grande sistema, que utiliza um processo de geren-

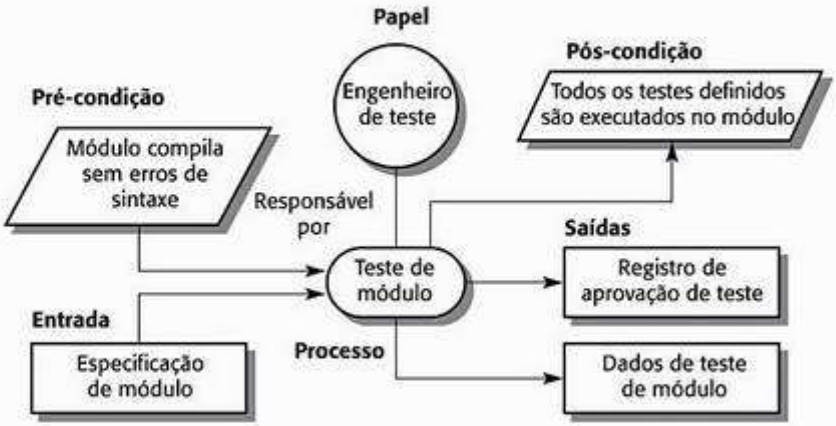
Figura 25.4 ▶

Elementos de um modelo de processo.

Elementos de um modelo de processo	Descrição
Atividade (representada por um retângulo com cantos arredondados, sem sombreado)	Uma atividade tem um objetivo e condições de entrada e saída claramente definidos. Exemplos de atividades são preparar um conjunto de dados de testes para testar um módulo, codificar uma função ou um módulo, conferir um documento etc. Geralmente, uma atividade é atômica, ou seja, é de responsabilidade de uma pessoa ou grupo. Não é decomposta em subatividades.
Processo (representado por um retângulo com cantos arredondados, com sombreado)	Um processo é um conjunto de atividades que têm alguma coerência, e cujo objetivo é geralmente estabelecido dentro de uma organização. Exemplos de processos são a análise de requisitos, o projeto de arquitetura, o planejamento de testes etc.
Produtos gerados (representados por um retângulo com sombreado)	Um produto relacionado é uma saída tangível de uma atividade, que é prevista em um plano de projeto.
Condição (representada por um paralelogramo)	Uma condição é uma precondição, que deve ser mantida antes que um processo ou uma atividade possa iniciar, ou uma pós-condição, que deve ser mantida depois que um processo ou uma atividade tenha terminado.
Papel (representado por um círculo com sombreado)	Um papel é uma área limitada de responsabilidade. Entre os exemplos de papéis estão o de gerente de configuração, engenheiro de testes e projetista de software. Uma pessoa pode assumir papéis diferentes, e um único papel pode ser associado a várias pessoas.
Exceção (não mostrada em exemplos aqui, mas pode ser representada por uma caixa com canto duplo)	Uma exceção é uma descrição de como modificar o processo, se algum evento, previsto ou não previsto, ocorrer. As exceções são com freqüência indefinidas, e fica por conta dos gerentes de projeto e dos engenheiros a manipulação das exceções.
Comunicação (representada por uma seta)	Uma troca de informações entre pessoas ou entre pessoas e sistemas de suporte à computação. A comunicação pode ser formal ou informal. A comunicação formal pode ser a aprovação de um produto gerado, feita por um gerente de projeto; a comunicação informal pode ser a troca de mensagens de correio eletrônico, a fim de resolver ambigüidades em um documento.

Figura 25.5 ▶

Processo de testes de módulos.



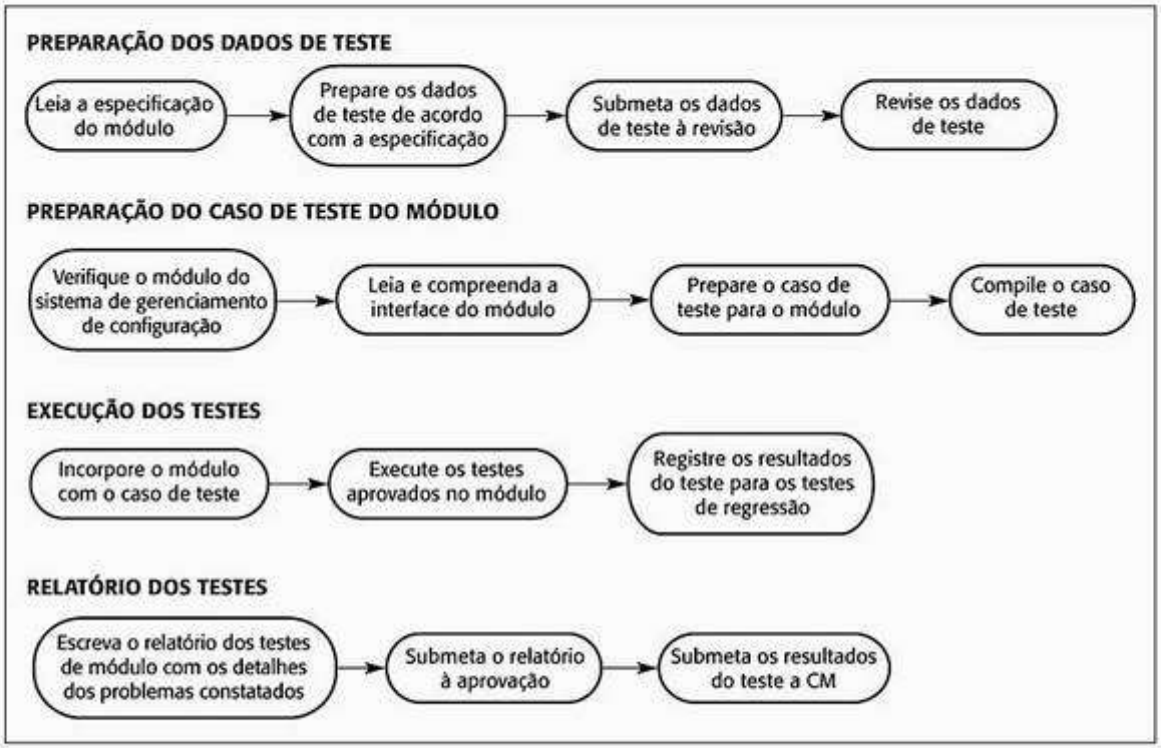


Figura 25.6 ▲
Atividades envolvidas nos testes de módulo.

ciamento de configuração estritamente controlado (veja o Capítulo 29). O software em teste e os dados de teste estão sob controle de configuração. A Figura 25.5 mostra quem é o responsável pelo processo de teste, pelas entradas e saídas do processo e pelas pré e pós-condições.

A Figura 25.6 decompõe o processo de 'teste de módulo' em uma série de atividades separadas. Esse fragmento mostra somente as atividades relativamente simples de testar o módulo. Há quatro seqüências de atividades empenhadas em preparar os dados de teste, escrever um caso de teste para o módulo, executar os testes e fazer os relatórios dos testes. As atividades de preparação seriam normalmente intercaladas. Obviamente, as atividades de preparação precedem as atividades de execução e de relatório.

Ignorei as informações sobre pré e pós-condições do processo e também as entradas e saídas de processo nesse diagrama. Essas informações tornariam o modelo complexo e de difícil compreensão. Em vez de tentar incluir todas as informações em um único modelo, talvez seja necessário fazer vários modelos diferentes, em níveis diversos de abstração. Esses modelos devem ser relacionados utilizando-se elementos comuns, como atividades ou produtos relacionados. Alguns modelos devem ser principalmente dedicados às atividades de processo outros, às informações de controle que dirigem a execução do processo.

25.2.1 Exceções do processo

Os processos de software são entidades muito complexas. Embora possa haver um modelo definido de processo em uma organização, isso pode apenas representar a situação ideal, na qual a equipe de desenvolvimento não precisa enfrentar problemas que não foram previstos. Na realidade, os problemas não previstos são um fato da vida cotidiana para os gerentes de projeto. O modelo 'ideal' de processo tem de ser modificado dinamicamente, à medida que são encontradas soluções para esses problemas. Exemplos dos tipos de exceções com que um gerente de projeto tem de lidar são:

1. várias pessoas-chave adoecem ao mesmo tempo, imediatamente antes da revisão de um projeto importante;

2. falha do processador de comunicações, o que significa que a correspondência eletrônica ficará inativa por diversos dias;
3. reorganização da empresa, o que significa que os gerentes terão de gastar muito de seu tempo com assuntos organizacionais, e não com o gerenciamento de projeto, e
4. é feito um pedido não previsto de novas propostas de projeto. É necessário desviar os esforços empregados no projeto, a fim de trabalhar em uma proposta.

Em geral, o efeito de uma exceção é que, de alguma maneira, os recursos, o orçamento ou os cronogramas de um projeto são modificados. É difícil prever todas as exceções com antecedência e incorporá-las em um modelo formal de processo. Muitas exceções são, por conseguinte, manejadas pelos gerentes dinamicamente com a mudança do processo 'padrão' para enfrentar essas circunstâncias inesperadas.

25.3 Medições de processo

As medições de processo são dados quantitativos sobre o processo de software. Humphrey (1989), em seu livro sobre melhoria de processos, argumenta que o conjunto de métricas de processo é essencial para a melhoria do processo. Ele também sugere que a medição tem um importante papel a ser desempenhado, em menor escala, na melhoria de processos de pessoal (Humphrey, 1995). As métricas de processo propriamente ditas podem ser utilizadas para avaliar se a eficiência de um processo foi melhorada ou não. Por exemplo, o esforço e o tempo devotados a testes podem ser monitorados. Melhorias efetivas no processo de testes devem reduzir o esforço ou o tempo necessário para os testes, ou ambos. Contudo, a medição de processo, por si só, não pode ser utilizada para determinar se a qualidade do produto melhorou. As métricas de produto (veja o Capítulo 24) devem também ser coletadas e relacionadas às atividades de processo.

Três classes de métricas de processo podem ser coletadas:

1. *O tempo gasto para um processo em particular ser concluído* Esse pode ser o tempo total dedicado ao processo, o 'tempo de calendário', o tempo gasto no processo por engenheiros individuais etc.
2. *Os recursos requeridos para um processo em particular* Os recursos podem ser o esforço total calculado em pessoa-dia, os custos de viagens, os recursos de computadores etc.
3. *O número de ocorrências de um evento em particular* Exemplos de eventos que podem ser monitorados são o número de defeitos descobertos durante a inspeção de código, o número de mudanças nos requisitos que foram solicitadas, o número médio de linhas de código modificadas em resposta a uma mudança nos requisitos etc.

Os primeiros dois tipos de medição podem ser utilizados para ajudar a descobrir se as mudanças de processo melhoraram a eficiência de um processo. Digamos que haja pontos fixos em um processo de desenvolvimento de software, como a aceitação de requisitos, a conclusão do projeto de arquitetura, o término da geração de dados de testes etc. Pode ser possível medir o tempo e o esforço requeridos para passar de um desses pontos fixos para outro. Os valores medidos podem ser utilizados para sugerir áreas em que o processo possa ser melhorado. Depois que as mudanças forem introduzidas, as medições de atributos do sistema podem mostrar se elas foram realmente benéficas.

As medições do número de eventos que ocorrem podem ter uma relação mais direta com a qualidade de software. Por exemplo, o aumento do número de defeitos descobertos pela mudança do processo de inspeção de programa provavelmente será refletido na qualidade

aperfeiçoada do produto. Contudo, isso tem de ser confirmado pelas medições subseqüentes do produto.

A dificuldade fundamental nas medições de processo é saber o que medir. Basili e Rombach (1988) propuseram o que chamam de paradigma de GQM (*goal-question-metric* – meta-questão-métrica). Esse paradigma é utilizado para ajudar a decidir que medições devem ser feitas e como elas devem ser utilizadas. Essa abordagem se baseia na identificação de:

1. *Metas* O que a organização está tentando alcançar. Exemplos de metas podem ser a produtividade melhorada dos programadores, um tempo mais curto de desenvolvimento de produtos, um aumento na confiabilidade do produto etc.
2. *Questões* São aperfeiçoamentos das metas, quando são identificadas áreas específicas de incerteza, relacionadas às metas. Normalmente, uma meta terá uma série de questões associadas, que precisam ser respondidas. Exemplos de questões relacionadas às metas mencionadas anteriormente são:
   - Como pode ser aumentado o número de linhas de código depuradas?
   - Como pode ser reduzido o tempo requerido para finalizar os requisitos do produto?
   - Como podem ser feitas avaliações de confiabilidade mais eficazes?
3. *Métricas* São as medições que precisam ser coletadas para ajudar a responder às questões e para confirmar se as melhorias de processo alcançaram a meta desejada ou não. Nos exemplos descritos anteriormente, dentre as medições que podem ser feitas destacam-se a produtividade individual dos programadores em linhas de código e em seu nível de experiência, o número de comunicações formais entre cliente e fornecedor para cada mudança de requisito e o número de testes necessários para provocar uma falha no produto.

A vantagem dessa abordagem aplicada à melhoria de processo é que ela separa os assuntos organizacionais (as metas) dos assuntos específicos de processo (as questões). Ela focaliza a coleta de dados e sugere que os dados coletados devem ser analisados de diferentes maneiras, dependendo da questão que deve ser respondida. Basili e Green (1993) descrevem como essa abordagem tem sido utilizada em um programa baseada em medições de longo prazo para a melhoria de processo.

A abordagem denominada GQM foi desenvolvida e combinada com o Modelo de Maturidade de Capacitação do SEI (discutido a seguir), no método ami (Pulford *et al.*, 1996) de melhoria de processo de software. Os desenvolvedores do método ami propõem uma abordagem por estágios para a melhoria de processo, em que a medição é introduzida quando uma organização tiver aplicado alguma disciplina em seus processos. Ela fornece diretrizes e orientação prática para a implementação da melhoria de processo baseada em medições.

25.4 Modelo de Maturidade de Capacitação do SEI

O Instituto de Engenharia de Software (*Software Engineering Institute* – SEI), da Carnegie-Mellon University, é um instituto fundado pelo DoD, Departamento de Defesa dos Estados Unidos, cuja missão é a transferência de tecnologia de software. Ele foi estabelecido para melhorar as capacitações da indústria de software dos Estados Unidos e, especialmente, as capacitações das organizações que recebem fundos do DoD para grandes projetos de defesa. Em meados da década de 80, o SEI iniciou um estudo dos meios para avaliar as capacitações dos fornecedores. Eles estavam particularmente interessados em fornecedores que apresentavam propostas para projetos de software financiados pelo Departamento de Defesa.

O resultado desse trabalho de avaliação de capacitação foi o Modelo de Maturidade de Capacitação de Software do SEI. Esse modelo tem influenciado muito no sentido de convencer a comunidade de engenharia de software em geral a considerar seriamente as melhorias de processo. O modelo do SEI classifica os processos de software em cinco diferentes níveis, como mostra a Figura 25.7.

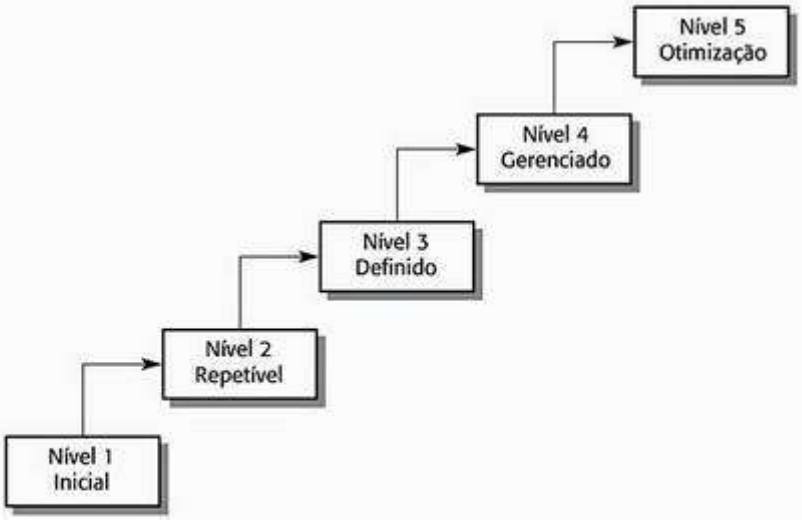
Os cinco níveis são definidos como se segue:

1. *Nível inicial* Nesse nível, uma organização não tem procedimentos eficazes de gerenciamento ou planos de projeto. Se existirem procedimentos formais para o controle de projeto, não existem os mecanismos organizacionais para garantir que eles são utilizados de modo consistente. A organização pode desenvolver software com sucesso, mas as características do software (qualidade etc.) e o processo de software (orçamento, cronograma etc.) serão imprevisíveis.
2. *Nível repetível* Nesse nível, uma organização tem procedimentos de gerenciamento formal, garantia de qualidade e controle de configuração já implantados. Ele é chamado de nível repetível porque a organização pode, com sucesso, repetir projetos do mesmo tipo. Contudo, não existe um modelo de processo formal. O sucesso do projeto depende de os gerentes individuais motivarem uma equipe e de a cultura organizacional atuar como uma descrição intuitiva do processo.
3. *Nível definido* Nesse nível, uma organização determinou seu processo e, assim, tem uma base para sua melhoria qualitativa de processo. Os procedimentos formais estão implantados, a fim de assegurar que o processo definido seja seguido em todos os projetos de software.
4. *Nível gerenciado* Uma organização nesse nível tem um processo definido e um programa formal de coleta de dados quantitativos. As métricas de processo e de produto são coletadas e fornecidas nas atividades de melhoria de processo.
5. *Nível de otimização* Nesse nível, uma organização está empenhada em obter melhorias contínuas de processo. A melhoria de processo é orçada e planejada, e é uma parte integrante do processo da organização.

Os níveis de maturidade, na versão inicial do modelo, foram criticados por serem muito imprecisos. Depois da experiência em utilizar o modelo para a avaliação de capacitação, como será discutido na seção seguinte, um modelo revisado foi adotado (Paulk *et al.*, 1993). Os cinco níveis foram mantidos, mas foram definidos mais especificamente em termos de áreas-chave

Figura 25.7 ▶

Modelo de Maturidade de Capacitação do SEI.



de processo (Figura 25.8). A melhoria de processo deve se ocupar de estabelecer esses processos-chave, e não simplesmente atingir algum nível arbitrário no modelo. Uma abordagem similar, com base em práticas-chave, tem sido utilizada para derivar um modelo de maturidade de processo de engenharia de requisitos (Sommerville e Sawyer, 1997).

O trabalho do SEI nesse modelo foi influenciado por métodos de controle estatístico de qualidade de produção. Humphrey (1988), na primeira descrição amplamente publicada do modelo, declara:

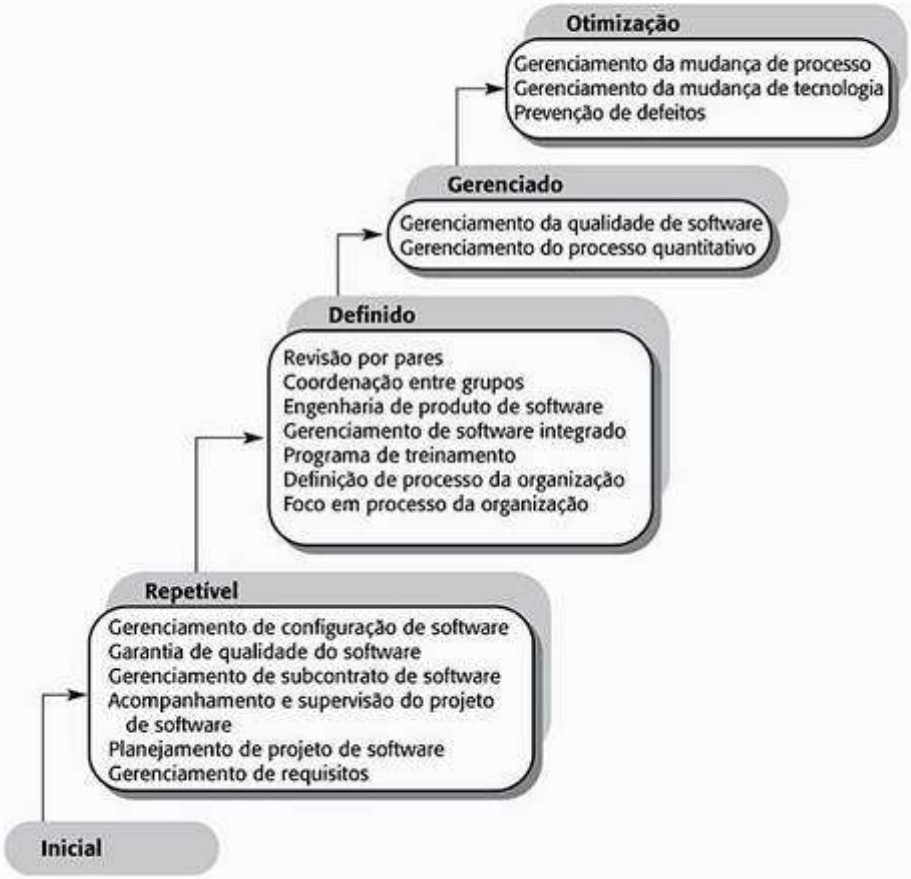
> W.E. Deming, em seu trabalho com a indústria japonesa depois da Segunda Guerra Mundial, aplicou os conceitos de controle de processo estatístico na indústria. Embora haja importantes diferenças, esses conceitos são tão aplicáveis a software quanto o são para automóveis, máquinas fotográficas, relógios de pulso e aço.

Embora certamente existam algumas semelhanças, não acho que os resultados da engenharia de produção possam ser transferidos diretamente para a engenharia de software. Como já foi discutido na Seção 25.1, fatores como a habilidade e a experiência dos engenheiros de desenvolvimento afetam a qualidade do produto. E isso é, freqüentemente, tão importante quanto os fatores relacionados ao processo.

O modelo de maturidade do SEI é uma importante contribuição, mas não deve ser tomado como um modelo definitivo de capacitação para todos os processos de software. Esse modelo foi desenvolvido para avaliar as capacitações das empresas que desenvolvem softwares de defesa. Esses são sistemas de software de grande porte, de longo tempo de duração, que têm complexas interfaces com o hardware e outros sistemas de software. Eles são desenvolvidos por grandes equipes de engenheiros e devem seguir os padrões e os procedimentos de desenvolvimento estabelecidos pelo Departamento de Defesa dos Estados Unidos.

Figura 25.8 ▶

Áreas-chave de processo (©1993 IEEE).



Os primeiros três níveis do modelo do SEI são relativamente simples de se compreender. As áreas-chave de processo incluem práticas atualmente utilizadas na indústria. Algumas organizações atingiram os níveis mais altos do modelo (Diaz e Sligo, 1997), mas os padrões e as práticas que são aplicáveis nesse nível não são amplamente compreendidos. Em alguns casos, a melhor prática pode divergir do modelo do SEI, devido a circunstâncias organizacionais locais.

Problemas nos níveis mais altos não negam a utilidade do modelo do SEI. A maioria das organizações está nos níveis mais baixos do processo de maturidade. Contudo, existem três outros sérios problemas com o modelo do SEI. Esses problemas podem significar que esse não é um bom fator de previsão da capacitação de uma organização para produzir software de alta qualidade.

Os problemas principais com o modelo de maturidade de capacitação são:

1. O modelo focaliza exclusivamente o gerenciamento de projeto, e não o desenvolvimento do produto. Ele não leva em conta o uso de tecnologia por uma organização, como a prototipação, os métodos formais ou estruturados, as ferramentas para análise estática etc.

2. Ele exclui a análise e a resolução de riscos como uma tecnologia fundamental do processo (Bollinger e McGowan, 1991). A importância da avaliação de riscos foi discutida no Capítulo 1. Sua vantagem é que ela descobre problemas antes que eles afetem seriamente o processo de desenvolvimento.

3. O domínio de aplicabilidade do modelo não é definido. Os autores do modelo claramente reconhecem que ele não é apropriado a todas as organizações. Contudo, eles não descrevem o tipo de organizações em que acreditam que o modelo deva ou não ser utilizado. A conseqüência é que esse modelo tem sido muito vendido como um meio de tentar resolver problemas de processo de software. Para organizações menores, em particular, o modelo é muito burocrático. Humphrey reconheceu isso e desenvolveu estratégias em menor escala das melhorias de processo. (Humphrey, 1995).

Os autores do modelo do SEI admitiram (Humphrey e Curtis, 1991) que a avaliação da tecnologia foi excluída porque não conseguiram encontrar nenhum modo-padrão para avaliar o uso da tecnologia. Além disso, eles sugeriram que os processos de gerenciamento definidos no modelo são essenciais, antes que a tecnologia possa ser efetivamente utilizada. Portanto, nos níveis inferiores do modelo, o uso de tecnologias específicas não é significativo, em se tratando da qualidade do produto.

Novamente, essa é uma simplificação exagerada. O uso eficaz de tecnologias, como protótipos e análise estática de programas, pode ter um efeito significativo na qualidade do produto. Isso não depende de como essas tecnologias são incorporadas no processo de software. O uso rotineiro de ferramentas e métodos por uma organização pode claramente separá-la de outras organizações no mesmo nível. Isso pode significar que a qualidade de seu produto é superior à qualidade dos produtos desenvolvidos por organizações em um nível de maturidade mais alto.

Existem diferenças significativas entre o desenvolvimento de produtos de software comerciais e de defesa. Os problemas em desenvolver sistemas de software muito grandes não são necessariamente compartilhados por todas as organizações envolvidas no desenvolvimento de software. Em particular, o desenvolvimento de software comercial pode responder mais rapidamente às mudanças tecnológicas. Isso é evidente no modo como o desenvolvimento de software tem se direcionado aos sistemas baseados na Internet e ao uso de arquiteturas distribuídas. O software de defesa, por outro lado, tem um longo processo de compra, um longo tempo de duração e, por boas razões, tende a ser mais conservador no que se refere a adotar novas técnicas. Algumas das idéias básicas do modelo do SEI são, em geral, relevantes, mas o modelo não é inteiramente aplicável fora do domínio para o qual foi projetado.

Paulk comparou o Modelo de Maturidade de Capacitação com o padrão ISO 9000 para o gerenciamento de qualidade, como foi discutido no Capítulo 24 (Paulk, 1995). Ele examinou

as áreas-chave de processo em cada nível do CMM e as comparou com os requisitos da ISO 9000, para os processos de gerenciamento de qualidade a serem definidos (veja a Figura 24.2). Para a grande maioria das áreas, há uma clara correlação entre os processos-chave e o padrão ISO 9000. O CMM é mais detalhado e prescritivo e inclui uma estrutura de melhoria de processo. Esses aspectos não são abordados no padrão ISO 9000. Em geral, as organizações cuja maturidade de processo é classificada no Nível 2 ou 3 provavelmente são compatíveis com a ISO 9000. Contudo, devido à definição abstrata da ISO 9000, algumas organizações de Nível 1 podem também satisfazer o padrão ISO 9000.

Com base na sua experiência com a maturidade de processo de software, o SEI desenvolveu uma série de outros modelos de maturidade de capacitação. O modelo de maturidade de capacitação de engenharia de sistemas avalia os processos de engenharia de sistema, como foi discutido no Capítulo 2; o modelo de maturidade de capacitação de aquisição de sistemas focaliza os processos de aquisição da empresa para software e hardware, e o modelo de maturidade de capacitação de pessoal (veja o Capítulo 22) se ocupa de processos organizacionais para a melhoria das capacitações e competências de pessoal. Detalhes de todos esses tópicos estão disponíveis no site Web do SEI, para o qual existe um link nas páginas Web deste livro.

25.4.1 Avaliação de capacitação

Em discussões sobre a maturidade de processo e o modelo SEI, algumas vezes, esquecemos que a intenção do modelo foi permitir ao Departamento de Defesa dos Estados Unidos avaliar as capacitações dos fornecedores de software. Na ocasião em que este livro foi escrito, não existia nenhuma exigência publicada para que os fornecedores atingissem um dado nível de maturidade. Contudo, na comunidade, não existe uma suposição de que organizações no nível mais alto de maturidade realmente tenham uma vantagem em suas propostas de serviços. No futuro, é provável que as organizações tenham de atingir certo nível de maturidade (provavelmente, o Nível 3) antes de poderem apresentar propostas de contratos de software para o DoD.

O modelo pretende representar as capacitações das organizações, e não a maturidade de projetos específicos. Isso tem sentido a partir do ponto de vista contratual, mas se uma organização for classificada no Nível 1 (digamos), isso não significa que todos os seus projetos estejam nesse nível. Dentro da organização, pode haver determinados projetos ou grupos que trabalhem em um nível de maturidade muito mais alto.

A avaliação de capacitação se baseia em um questionário-padrão projetado para identificar os principais processos na organização. Esse questionário é aplicado durante uma visita de avaliação, em que os gerentes de diferentes projetos são entrevistados. Depois de uma discussão sobre suas respostas ao questionário e do refinamento dessas respostas, é produzida uma pontuação de avaliação. Um modelo desse processo de avaliação é mostrado na Figura 25.9.

Acho que o problema principal com o modelo atual é sua estratificação em níveis e a associação de números com esses níveis. As diretrizes de avaliação requerem que uma organização tenha todas as práticas implantadas em um determinado nível antes de ser aceita nele. Assim, uma organização que tivesse 80 por cento de práticas do Nível 2 implantadas (digamos) e 70 por cento de práticas do Nível 3, receberia uma graduação de Nível 1. Uma abordagem melhor seria uma classificação de capacitação mais detalhada, em que fossem identificadas as práticas específicas padronizadas em uma organização.

A abordagem SPICE de avaliação de capacitação e melhoria de processo (Paulk e Konrad, 1994; El Amam *et al.*, 1997) é mais flexível do que o modelo do SEI. O SPICE foi proposto como um padrão ISO de estrutura de melhoria de processo. Ele inclui níveis de maturidade comparáveis com os níveis do SEI, mas também identifica áreas-chave de processo, como processos cliente-fornecedor, que podem passar por esse níveis. À medida que o nível de maturidade aumenta, o desempenho desses processos-chave deve melhorar.

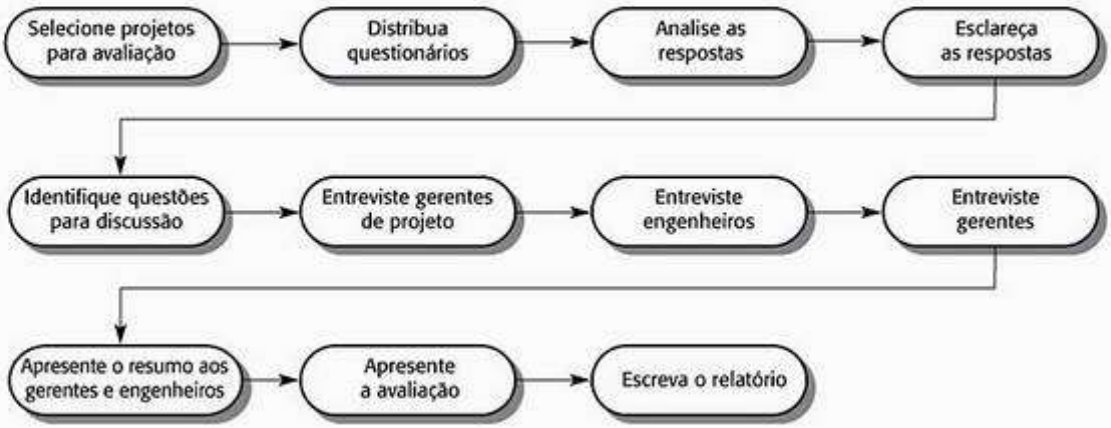


Figura 25.9 ▲

Processo de avaliação de capacitação.

O projeto de Bootstrap tinha o objetivo de ampliar e adaptar o modelo de maturidade do SEI para torná-lo aplicável a uma gama mais ampla de empresas. O modelo de Bootstrap (Haase *et al.*, 1994; Kuvaja *et al.*, 1994) utiliza os níveis de maturidade do SEI, mas também propõe:

- diretrizes para um sistema de qualidade para toda a empresa, a fim de apoiar a melhoria de processo;
- uma importante distinção entre organização, metodologia e tecnologia, e
- um modelo básico de processo (com base no modelo utilizado em Agências Espaciais Européias), que pode ser adotado.

A evolução do Modelo de Maturidade de Capacitação do SEI leva em conta essas abordagens alternativas. Durante o tempo de vida útil deste livro, prevejo que uma nova versão do CMM estará disponível, incorporando as melhores características das outras abordagens. Essa nova versão se tornará o padrão mundial para a avaliação e a melhoria do processo de software.

25.5 Classificação de processo

A classificação de maturidade de processo, proposta no modelo do SEI, é apropriada para grandes projetos de software, de longo tempo de duração, empreendidos por grandes organizações. Há muitos outros tipos de projetos e organizações de software em que essa visão de maturidade de processo não deve ser aplicada.

Em vez de tentar classificar processos em níveis, com delimitações bastante arbitrárias traçadas entre esses níveis, acredito que uma abordagem mais geral de classificação de processo possa ser aplicada a uma variedade maior de organizações e projetos. Diferentes tipos de processo podem ser identificados:

1. *Processos informais* São processos em que não há um modelo estritamente definido. O processo utilizado é escolhido pela equipe de desenvolvimento. Os processos informais podem utilizar procedimentos formais, como o gerenciamento de configuração, mas os procedimentos a serem utilizados e as relações entre eles não são predefinidos.

2. *Processos gerenciados* São processos em que há um modelo definido de processo implantado. Eles são utilizados para orientar o processo de desenvolvimento. O modelo de pro-

cesso define os procedimentos utilizados, os cronogramas e as relações entre os procedimentos.

3. *Processos metódicos* São processos em que se utiliza algum método ou alguns métodos de desenvolvimento definidos (como métodos sistemáticos para o projeto orientado a objetos). Esses processos se beneficiam do suporte de ferramentas CASE para processos de projeto e análise.

4. *Processos de melhoria* São processos que têm objetivos de melhoria inerentes. Há um orçamento específico para as melhorias de processo e os procedimentos em andamento para introduzir essas melhorias. Como parte dessas melhorias, medições quantitativas de processo podem ser introduzidas.

Essas classificações obviamente se sobrepõem, e um processo pode se dividir em diversas classes. Por exemplo, o processo pode ser informal, pelo fato de ser escolhido pela equipe de desenvolvimento. A equipe pode escolher um método de projeto em particular. Ela pode também ter uma capacitação de melhoria de processo. Nesse caso, o processo seria classificado como *informal*, *metódico* e *de melhoria*.

Essas classificações são úteis, porque servem de base para a melhoria de processo multidimensional. Elas ajudam as organizações a escolher um processo apropriado para diferentes tipos de desenvolvimento de produtos. A Figura 25.10 mostra diferentes tipos de produtos e o tipo de processo que pode ser utilizado para seu desenvolvimento.

As classes de sistema mostradas na Figura 25.10 podem se sobrepor. Portanto, podem ser desenvolvidos pequenos sistemas que deverão passar por reengenharia, utilizando-se um processo metódico. Os grandes sistemas sempre precisam de um processo gerenciado. Contudo, se o domínio não for bem compreendido, poderá ser difícil escolher um método de projeto apropriado. Os grandes sistemas podem, portanto, ser desenvolvidos utilizando-se um processo gerenciado que não seja baseado em nenhum método de projeto em particular.

A classificação do processo fornece uma base para escolher o processo certo a ser utilizado quando um determinado tipo de produto deve ser desenvolvido. Por exemplo, suponhamos que haja um programa que seja preciso para apoiar a transição de um tipo de sistema de computador para outro. Esse programa tem um tempo de duração relativamente curto. Seu desenvolvimento não requer os padrões e os procedimentos de gerenciamento apropriados aos softwares que serão utilizados durante muitos anos.

A classificação de processo reconhece que ele afeta a qualidade do produto, mas não supõe, contudo, que ele seja sempre o fator dominante. Ele fornece uma base para a melhoria de diferentes tipos de processo. Por exemplo, as melhorias em processos metódicos podem ser baseadas no melhor método de treinamento, na melhor integração de requisitos e de projeto, em ferramentas CASE aperfeiçoadas etc.

Figura 25.10 ▶

Aplicabilidade do processo.

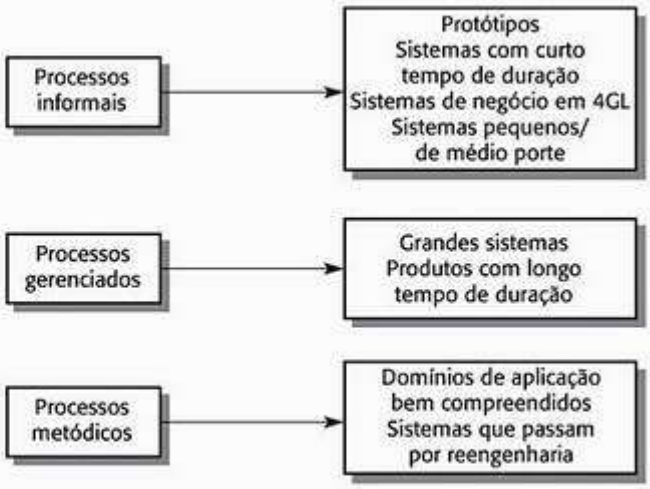
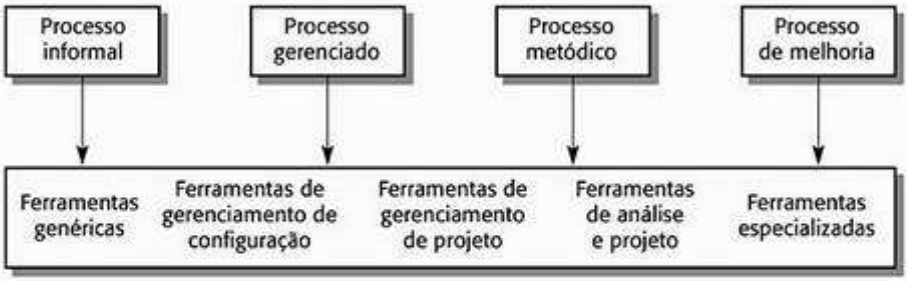


Figura 25.11

Apoio de ferramentas de processo.



A maioria dos processos de software, hoje em dia, tem algum apoio de ferramentas CASE, portanto são *processos apoiados*. Atualmente, os processos metódicos são apoiados, de modo usual, por ferramentas de análise e projeto. Contudo, os processos podem ter outros tipos de apoio de ferramentas (por exemplo, ferramentas de prototipação, ferramentas de teste), independentemente de um método de projeto estruturado estar em uso ou não.

O apoio de ferramentas que pode ser efetivo em apoiar os processos depende da classificação do processo. Por exemplo, os processos informais podem utilizar ferramentas genéricas, como linguagens de prototipação, compiladores, depuradores, processadores de texto etc. Raramente eles utilizarão ferramentas mais especializadas de um modo consistente. A Figura 25.11 mostra que diferentes ferramentas podem ser utilizadas no desenvolvimento de software. A eficácia dessas ferramentas depende do tipo de processo utilizado.

Ferramentas de análise e projeto são eficazes em termos de custo, provavelmente, somente quando um processo metódico estiver sendo seguido. Ferramentas especializadas são desenvolvidas como parte da atividade de melhoria de processo, a fim de fornecer apoio específico para melhorar certas atividades do processo.

PONTOS-CHAVE

- A melhoria de processo envolve análise de processo, padronização, medição e modificações. O treinamento é essencial para que a melhoria de processo seja eficaz.
- Os modelos de processo incluem descrições de atividades, subprocessos, papéis, exceções, comunicações, produtos gerados e outros processos.
- A medição deve ser utilizada para responder a questões específicas sobre o processo de software utilizado. Essas questões devem se basear nas metas de melhoria organizacional.
- Os três tipos de métricas de processo são: métricas de tempo, métricas de utilização de recursos e métricas de eventos.
- O Modelo de Maturidade do SEI classifica os processos de software em inicial, repetível, definido, gerenciado e de otimização. Ele identifica os processos-chave que devem ser utilizados em cada um desses níveis.
- O modelo do SEI é apropriado para grandes sistemas desenvolvidos por grandes equipes de engenheiros. Ele não deve ser aplicado, sem adaptação, em circunstâncias locais.
- Os processos podem ser classificados como informal, gerenciado, metódico e de melhoria. Essa classificação pode ser utilizada para identificar o apoio de ferramentas de processo.

LEITURAS SUGERIDAS

Trends in software: software process modelling and technology. Esse livro traz uma boa seleção de artigos de visão geral, que abordam diferentes aspectos dos processos de software, inclusive modelagem de processo, suporte a processos e o uso do CMM. (A. Fuggetta e A. Wolf (eds.), 1996, John Wiley and Sons.)

The capability maturity model for software. Trata-se de uma descrição abrangente da estrutura de melhoria e avaliação de processo do SEI. Inclui descrições de todos os principais processos e das razões para sua inclusão. (M. Paulk, C.V. Weber e B. Curtis, 1995, Addison Wesley.)

IEEE Software, 10(4), jul. 1993. Essa é uma edição especial do periódico dedicado a artigos sobre maturidade de processo. Inclui uma descrição da versão 1.1 do modelo de maturidade de capacitação e também artigos que discutem a melhoria e a modelagem de processo.

EXERCÍCIOS

25.1 Sugira modelos de processo para os seguintes processos:

- Acender fogo em madeira.
- Preparar uma refeição com três pratos (você escolhe o menu).
- Escrever um pequeno programa (com 50 linhas).

25.2 Em que circunstâncias a qualidade de produto pode ser determinada pela qualidade da equipe de desenvolvimento? Dê exemplos dos tipos de produtos de software que dependem particularmente do talento e das habilidade individuais.

25.3 Suponha que a meta da melhoria de processo em uma organização seja aumentar o número de componentes reutilizáveis produzidos durante o desenvolvimento. Sugira três questões, conforme o GQM, pelas quais isso pode ser alcançado.

25.4 Descreva três tipos de métricas de processo de software que podem ser coletadas como parte de uma melhoria de processo. Dê um exemplo de cada tipo de métrica.

25.5 Dê duas vantagens e duas desvantagens da abordagem de avaliação e melhoria de processo que estejam incorporadas no modelo de maturidade de processo.

25.6 Sugira dois domínios de aplicação em que o modelo de capacitação do SEI seja provavelmente inadequado. Dê as razões pelas quais isso ocorre.

25.7 Considere o tipo de processo de software utilizado em sua organização. Quantas áreas-chave de processo, identificadas no modelo do SEI, são utilizadas? Como esse modelo classificaria seu nível de maturidade de processo?

25.8 Sugira três ferramentas especializadas de software que podem ser desenvolvidas para apoiar um programa de melhoria de processo em uma organização.

25.9 Explique por que um processo metódico não é necessariamente um processo gerenciado, como foi definido na Seção 25.5.

25.10 Os programas de melhoria de processo, que envolvem medir o trabalho de pessoas no processo e na modificação dele, são inerentemente desumanos? Que resistência poderia surgir contra um programa de melhoria de processo?

PÁGINA EM BRANCO

PARTE SETE

Evolução

PÁGINA EM BRANCO

26 Sistemas legados

Objetivos

Os objetivos deste capítulo são apresentar os sistemas legados e descrever quantos desses sistemas já foram projetados. Depois de ler este capítulo, você poderá:

- compreender o que significa o termo 'sistemas legados' e por que esses sistemas são fundamentais para a operação de muitas empresas;
- conhecer as estruturas comuns dos sistemas legados;
- compreender os princípios do projeto orientado a funções — a estratégia de projeto mais comumente utilizada para os atuais sistemas legados, e
- compreender como os sistemas legados podem ser avaliados para decidir se eles devem ser descartados ou mantidos, se devem passar pelo processo de reengenharia ou ser substituídos.

Sumário

26.1 Estruturas dos sistemas legados
26.2 Projeto de sistemas legados
26.3 Avaliação dos sistemas legados

As empresas gastam muito dinheiro em sistemas de software, e para que elas obtenham um retorno desse investimento o software deve ser utilizado por vários anos. O tempo de duração de sistemas de software é muito variável, e muitos sistemas de grande porte permanecem em uso por mais de 10 anos. Algumas organizações ainda dependem de sistemas de software que têm mais de 20 anos de existência. Muitos desses antigos sistemas ainda são fundamentais para as empresas, isto é, as empresas dependem dos serviços fornecidos pelo software, e qualquer falha desses serviços teria um sério efeito em seu dia-a-dia. A esses antigos sistemas foi dado o nome de *sistemas legados*.

Os sistemas legados não são os sistemas que foram originalmente fornecidos. Fatores internos e externos, como o estado da economia nacional e internacional, as modificações nos mercados, as alterações nas leis, as mudanças de gestão e organização estrutural, significam que as empresas sofrem mudanças contínuas. Essas mudanças geram requisitos de software novos ou modificados; assim, todos os sistemas de software úteis inevitavelmente são modificados quando as empresas passam por mudanças. Portanto, os sistemas legados incorporam um grande número de alterações, que foram feitas durante muitos anos. Muitas pessoas diferentes se envolveram na realização dessas mudanças, e é incomum que qualquer pessoa tenha uma compreensão completa do sistema.

As empresas substituem regularmente seus equipamentos e suas máquinas por sistemas modernos. Contudo, descartar sistemas legados e substituí-los por software mais moderno envolve riscos significativos para as empresas. Como foi discutido no Capítulo 4, a maioria dos gerentes tenta minimizar os riscos e, por conseguinte, não quer enfrentar as dúvidas referentes a novos sistemas de software. Substituir um sistema legado é uma estratégia de negócio arriscada, por diversas razões:

1. Raramente existe uma especificação completa do sistema legado. A especificação original pode ter sido perdida. Se existir uma especificação, é pouco provável que ela incorpore todas as mudanças que foram feitas no sistema. Portanto, não existe uma maneira direta de especificar um novo sistema que seja funcionalmente idêntico ao sistema que está em uso.
2. Os processos corporativos e o modo como os sistemas legados operam estão sempre intrinsecamente entrelaçados. Esses processos foram projetados para tirar vantagem dos serviços de software e para evitar seus pontos negativos. Se o sistema for substituído, esses processos também terão de ser modificados, com custos e conseqüências potencialmente imprevisíveis.
3. Importantes regras corporativas podem estar inseridas no software e podem não estar documentadas em nenhum outro lugar. Uma regra corporativa é uma restrição que se aplica a alguma função da empresa, e romper com essa restrição pode acarretar conseqüências imprevisíveis para a empresa. Por exemplo, uma empresa de seguros pode ter inserido suas regras para avaliar o risco de um pedido de apólice em seu software. Se essas regras não forem mantidas, a empresa pode aceitar apólices de alto risco, que resultarão em dispendiosas reivindicações futuras.
4. O desenvolvimento de um software novo, por si só, é arriscado, uma vez que podem ocorrer problemas inesperados com um novo sistema. Ele pode não ser entregue em tempo nem pelo preço esperado.

Manter sistemas legados em uso evita os riscos de substituição, mas introduzir mudanças em um software existente, geralmente, é mais dispendioso, à medida que o sistema se torna mais velho. Os sistemas de software legados com poucos anos de duração são particularmente dispendiosos no que se refere a fazer alterações, por diversas razões:

1. Diferentes partes do sistema foram implementadas por diferentes equipes. Portanto, não há um estilo de programação consistente em todo o sistema.

2. Parte do sistema ou todo o sistema pode ser implementado com a utilização de uma linguagem de programação obsoleta. Pode ser difícil encontrar pessoal que tenha conhecimento dessas linguagens. Além disso, pode ser necessário um gasto dispendioso com a manutenção do sistema.

3. Freqüentemente, a documentação do sistema é inadequada e desatualizada. Em alguns casos, a única documentação é o código-fonte do sistema. Algumas vezes, o código-fonte foi perdido e somente a versão executável do sistema está disponível.

4. Em geral, os muitos anos de manutenção podem ter corrompido a estrutura do sistema, tornando-o cada vez mais difícil de ser compreendido. É possível que novos programas tenham sido acrescentados e que tenham sido criadas interfaces destes com outras partes do sistema, de acordo com a necessidade.

5. O sistema pode ter sido otimizado para melhorar a utilização de espaço ou a velocidade de execução, em vez de ter sido escrito para facilitar a compreensão. Isso acarreta dificuldades específicas para os programadores que aprenderam modernas técnicas de engenharia de software e que não foram expostos aos truques de programação utilizados.

6. Os dados processados pelo sistema podem estar armazenados em diferentes arquivos, que podem ter estruturas incompatíveis. É possível que alguns dados tenham sido duplicados ou estejam desatualizados, inexatos ou incompletos.

As empresas que têm um grande número de sistemas legados enfrentam um dilema fundamental. Se continuarem utilizando os sistemas legados e fazendo alterações conforme a necessidade, seus custos inevitavelmente aumentarão. Se decidirem substituir seus sistemas legados por novos sistemas, isso será dispendioso, e os novos sistemas podem não fornecer às empresas um apoio tão eficaz quanto o dos sistemas legados. Conseqüentemente, muitas empresas estão examinando técnicas de engenharia de software que ampliem o tempo de duração dos sistemas legados e que reduzam os custos de manter esses sistemas em uso. Serão discutidas algumas dessas técnicas no Capítulo 27, que aborda a evolução de produtos de software em geral, e no Capítulo 28, que aborda a reengenharia de software.

26.1 Estruturas dos sistemas legados

Os sistemas legados não são simplesmente sistemas de software antigos, embora os componentes de software desses sistemas sejam o principal enfoque deste capítulo. Os sistemas legados são sistemas sociotécnicos baseados em computadores (discutidos no Capítulo 2); assim, eles incluem software, hardware, dados e processos corporativos. As modificações em uma parte do sistema, inevitavelmente, envolvem mais modificações em outros componentes. As decisões sobre esses sistemas não são regidas sempre por critérios objetivos de engenharia, mas são afetadas por estratégias e políticas organizacionais mais amplas.

A Figura 26.1 ilustra as diferentes partes lógicas de um sistema legado e suas relações:

1. *Hardware de sistema* Em muitos casos, os sistemas legados foram escritos para hardware de mainframe, que não está mais disponível, tem uma manutenção dispendiosa e pode não ser compatível com as atuais políticas organizacionais de compras na área de TI (tecnologia da informação).

2. *Software de apoio* O sistema legado pode depender de diferentes produtos de software de apoio, desde o sistema operacional e utilitários fornecidos pelo fabricante de hardware até os compiladores utilizados para o desenvolvimento do sistema. Novamente, esses

Figura 26.1

Componentes de sistemas legados.

produtos de software podem estar obsoletos e não ter mais a assistência técnica de seus fornecedores originais.

3. *Software de aplicação* Como será discutido posteriormente, o sistema de aplicação que fornece os serviços de negócios é, em geral, composto de vários programas separados, desenvolvidos em épocas diferentes. Algumas vezes, o termo *sistema legado* se refere a esse software de aplicação, e não ao sistema como um todo.

4. *Dados de aplicação* Esses são os dados processados pelo sistema de aplicação. Em muitos sistemas legados, um imenso volume de dados se acumulou durante o tempo de duração do sistema. Esses dados podem ser inconsistentes e podem estar duplicados em diferentes arquivos.

5. *Processos de negócios* São os processos utilizados nas empresas a fim de atingir algum objetivo de negócios. Um exemplo de processo de negócios em uma empresa de seguros seria emitir uma apólice de seguro; em uma empresa manufatureira, um processo de negócios seria aceitar um pedido de produto e definir o processo de fabricação correspondente.

6. *Políticas e regras de negócios* São as definições de como a empresa deve ser conduzida e as restrições às quais ela deve se submeter. O uso de um sistema legado de aplicações pode estar inserido nessas políticas e regras.

Uma maneira alternativa de considerar esses diferentes componentes de um sistema legado é como uma série de camadas, como mostra a Figura 26.2. Cada camada depende da camada imediatamente inferior e faz interface com essa camada. Se as interfaces forem mantidas, será possível fazer alterações em uma camada sem afetar qualquer uma das camadas adjacentes.

Na prática, contudo, esse simples encapsulamento raramente funciona, e mudanças em uma camada do sistema podem requerer conseqüentes mudanças nas camadas que estão tanto acima como abaixo da camada alterada. As razões disso são:

1. A modificação de uma camada do sistema pode introduzir novos recursos, e as camadas superiores do sistema podem ser modificadas para se beneficiarem desses recursos. Por exemplo, um novo banco de dados, introduzido na camada de apoio do software, pode incluir recursos para acessar os dados por meio de um browser na Web, e os processos de negócios podem ser modificados para tirar vantagem desse recurso.

2. A modificação do software no sistema pode torná-lo mais lento, de modo que um novo hardware é necessário, a fim de melhorar o desempenho do sistema. O aumento do desempenho do novo hardware pode, então, significar que outras mudanças de software, antes impraticáveis, se tornem possíveis.

3. Muitas vezes, é impossível manter interfaces de hardware, especialmente se for proposta uma mudança radical para um novo tipo de hardware. Por exemplo, se uma empresa

Figura 26.2 ▶

Modelo em camadas de um sistema legado.

muda de hardware de mainframe para sistemas cliente-servidor (discutidos no Capítulo 11), geralmente eles têm diferentes sistemas operacionais. Desse modo, podem ser necessárias importantes mudanças no software de aplicação.

O software de aplicação em um sistema legado não é um único programa de aplicação, mas inclui geralmente uma série de diferentes programas. O sistema pode ter começado como um único programa, processando um ou dois arquivos de dados, mas, ao longo do tempo, podem ter sido implementadas alterações, com a adição de novos programas, que compartilham os dados e se comunicam com outros programas no sistema. Da mesma maneira, os arquivos iniciais de dados do sistema sofrem acréscimos, à medida que novas informações são exigidas. Esse exemplo está ilustrado na Figura 26.3. Diferentes programas compartilham arquivos de dados, de modo que as mudanças feitas em um programa, que possam afetar os dados, inevitavelmente resultam em alterações em outros programas.

Os diferentes programas no sistema legado de aplicação, em geral, foram escritos por pessoas diferentes e são, freqüentemente, escritos em diferentes linguagens de programação ou em diferentes versões de uma linguagem de programação. Por exemplo, o software original pode ter sido desenvolvido em COBOL-72, mas os programas posteriores podem ter sido implementados em uma nova versão da linguagem, como a COBOL-80. Os compiladores e o software de apoio para todas essas linguagens podem precisar ser mantidos. Isso tudo aumenta a complexidade do sistema e torna mais altos os custos para a realização de modificações.

Embora ainda existam sistemas legados que utilizam arquivos separados para manter seus dados, um grande número de sistemas corporativos centralizou seu gerenciamento de dados em um sistema de bancos de dados (Figura 26.4). A vantagem da adoção dessa estrutura é que os dados do sistema são descritos utilizando-se modelos de dados lógicos e físicos. É menos provável ocorrer a redundância e a duplicação de dados, e é mais fácil avaliar o impacto de mudanças no sistema que afetem seus dados. Os bancos de dados também fornecem a capacidade de processamento de transações, em que as modificações nos dados podem ser feitas de modo recuperável. Isso permite atualizações interativas dos dados.

Os pedidos de atualizações interativas dos dados se originam de diferentes terminais e em diferentes momentos. Por exemplo, um sistema bancário pode ter centenas de terminais, que são utilizados pelos caixas, nas agências, e pelo público, nos caixas automáticos. Essas transações individuais são feitas com base no banco de dados central das contas e têm de ser coletadas e organizadas de modo que não interfiram umas com as outras.

Um monitor de teleprocessamento, como o sistema CICS, da IBM, é um sistema de software que pode manipular e armazenar as entradas de muitas fontes diferentes. Em sistemas bancários, um monitor de teleprocessamento aceita transações de terminais de agências e ATMs e pode executar algum processamento local. Em seguida, ele armazena essas transações e as apresenta como uma lista em série para o banco de dados de contas, que atualiza a conta do cliente e confirma que a transação foi processada. Esse exemplo está ilustrado na Figura 26.5.

Figura 26.3

Sistemas legados de aplicação.

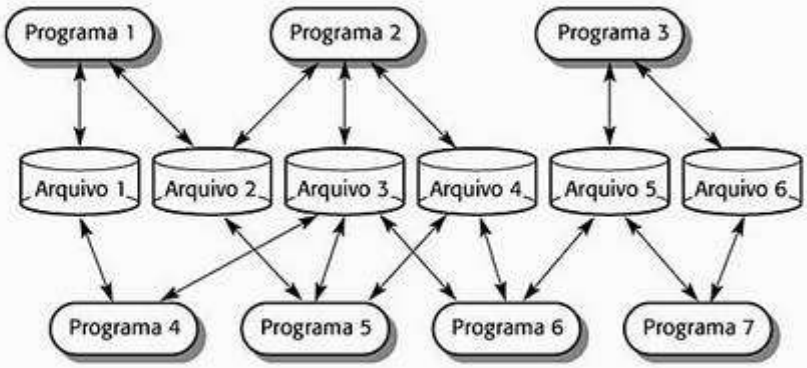


Figura 26.4

Sistemas centrados em bancos de dados.

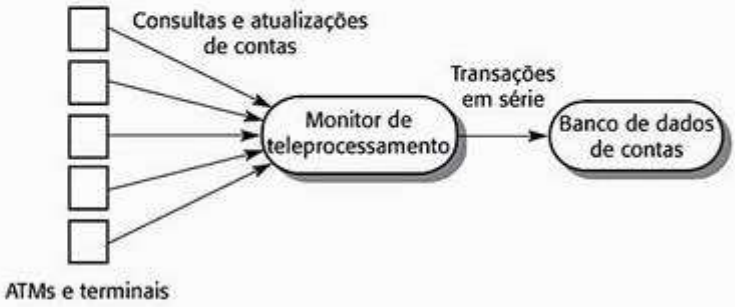
Existem duas questões importantes sobre legados em sistemas centrados em banco de dados:

1. O sistema de gerenciamento de banco de dados utilizado pode ser obsoleto e incompatível com os outros sistemas de gerenciamento de bancos de dados (*database management systems* – DBMS) utilizados por uma empresa. Os sistemas de gerenciamento de banco de dados relacionais são, atualmente, os mais eficazes para aplicações de negócios. Contudo, muitos sistemas legados dependem de sistemas de bancos de dados mais antigos, que têm como base modelos hierárquicos e de rede. Esses sistemas foram projetados para permitir a otimização do desempenho do sistema, em vez do simples gerenciamento de dados. Modernos equipamentos de hardware podem tornar essa otimização desnecessária, mas os custos de migrar para um modelo de dados relacionais é muito alto.

2. O monitor de teleprocessamento utilizado pode ter sido projetado para uso com um determinado sistema de banco de dados e para um hardware de mainframe. Portanto, pode não ser possível utilizar o mesmo monitor de teleprocessamento com um novo banco de dados. Essa parte do sistema pode também precisar ser substituída, e isso aumenta os custos e os riscos da modificação no sistema.

Figura 26.5

Processamento de transações que utilizam um monitor de teleprocessamento.



26.2 Projeto de sistemas legados

Praticamente todos os sistemas legados utilizados hoje em dia foram projetados antes de o desenvolvimento orientado a objetos ser amplamente utilizado na indústria. Em vez de serem organizados como um conjunto de objetos interativos, os programas, nesses sistemas, são geralmente estruturados como um conjunto de sub-rotinas ou funções. Cada sub-rotina fornece parte da funcionalidade do sistema e é chamada, conforme a necessidade, por outras sub-rotinas. Em algumas linguagens, as sub-rotinas têm seus próprios dados privativos, mas também acessam áreas de dados compartilhadas. Em outras linguagens, como as antigas versões de COBOL, os dados são compartilhados e acessíveis a todas as sub-rotinas.

Uma estratégia do projeto orientado a funções baseia-se em decompor um programa em um conjunto de funções interativas ou sub-rotinas, em um estado de sistema centralizado, compartilhado por essas funções (Figura 26.6). As informações de estado local nas funções são mantidas somente enquanto estão em execução. Essa estratégia de projeto está embutida como um 'projeto top-down' ou 'projeto estruturado', em uma série de métodos estruturados, que foram inventados na década de 70 e no início da década de 80 (Myers, 1975; Wirth, 1976; Constantine e Yourdon, 1979). Centenas de milhares de programas de aplicações foram desenvolvidos utilizando-se esses métodos e as ferramentas CASE associadas.

O projeto orientado a funções oculta os detalhes de um algoritmo em uma função, mas as informações de estado do sistema não são ocultadas. Isso pode causar problemas, porque uma função pode modificar o estado de uma maneira que as outras funções não esperam. As modificações em uma função e seu uso do estado do sistema podem causar alterações imprevistas no comportamento de outras funções. Isso é um problema específico dos sistemas legados, uma vez que os programas, geralmente, foram modificados por muitas pessoas diferentes. Não é comum uma pessoa compreender inteiramente como as diferentes partes de um programa interagem.

Uma abordagem funcional ao projeto é mais eficaz quando a quantidade de informações de estado do sistema é minimizada e o compartilhamento das informações é explícito. Os sistemas cujas respostas dependem de um único estímulo ou entrada e que não são afetados por históricos de entrada são naturalmente orientados a funções. Muitos sistemas de processamento de dados de negócios se ocupam de processar registros distintos. O processamento de um registro não depende dos resultados do processamento do registro precedente. Portanto, é bastante natural utilizar uma abordagem orientada a funções quando são projetados esses sistemas.

Os sistemas de processamento de dados de negócios são a maior classe de sistemas legados. Há dois tipos principais de sistemas de negócios:

1. *Sistemas de processamento em lotes* A entrada e a saída dos dados é em lotes (*batches*) provenientes de um arquivo, e não a partir de um terminal de usuário. Exemplos de sis-

Figura 26.6 ▶
Visão de projeto orientado a funções.

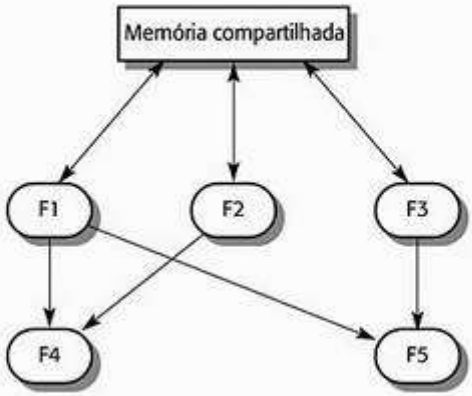
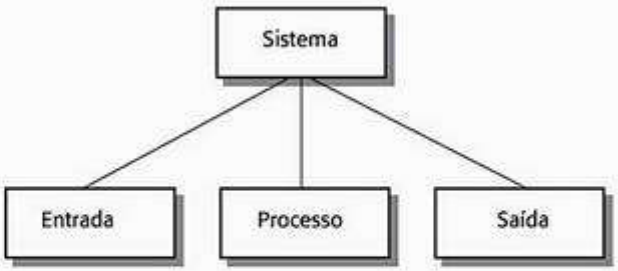


Figura 26.7 ▶
Modelo entrada-processo-saída



temas de processamento em lotes são os sistemas de folhas de pagamento, sistemas de cobrança etc.

2. *Sistemas de processamento de transações* As operações de entrada e saída de dados ocorrem como uma série de transações em um banco de dados, com a transação gerada a partir de um terminal de usuário.

Naturalmente, esses diferentes tipos de sistemas podem compartilhar dados. Portanto, um banco utiliza um sistema de processamento de transações para gerenciar as transações de contas, mas gera comandos bancários para os clientes que utilizam um sistema de processamento em lotes.

Ambos os sistemas de processamento, em lotes e por transação, geralmente acompanham um modelo 'entrada-processo-saída', como ilustra a Figura 26.7. Esses sistemas coletam entradas a partir de uma ou mais fontes, realizam algum processamento nessas entradas e, então, geram saídas, que são relacionadas, de alguma maneira, às entradas. Por exemplo, um sistema de cobrança telefônica considera as entradas, que são os registros de clientes e as leituras dos medidores telefônicos da chave comutadora, calcula os custos para cada cliente e, em seguida, gera as faturas de cobrança impressas, como saídas.

Os componentes de entrada, processamento e saída podem, eles mesmos, ser decompostos em uma estrutura entrada-processo-saída. Por exemplo:

1. Um componente de entrada pode ler alguns dados (entrada) a partir de um terminal de usuário, verificar a validade desses dados e corrigir alguns erros (processo) e, então, pôr em fila os dados válidos, a fim de realizar o processamento (saída).
2. Um componente de processamento pode iniciar uma transação a partir de uma fila (entrada), realizar alguns cálculos sobre os dados, criar um novo registro de dados anotando os resultados do cálculo (processo) e, então, pôr na fila esse novo registro, para realizar a impressão (saída).
3. Um componente de saída pode ler os registros a partir de uma fila (entrada), formatá-los de acordo com o formulário de saída (processo) e, então, remetê-los para uma impressora (saída).

O projeto de sistemas orientados a funções é freqüentemente modelado, utilizando-se diagramas de fluxo de dados. Apresentei os diagramas de fluxo de dados no Capítulo 7. Esses diagramas são uma representação funcional, em que cada retângulo com cantos arredondados, no fluxo de dados, representa uma função que implementa alguma transformação de dados, e cada seta representa um item de dado que é processado pela função. Os arquivos ou repositórios de dados são representados como retângulos na notação de fluxo de dados que utilizo para os diagramas de fluxo de dados. Esses diagramas mostram o processamento completo; por exemplo, todas as funções que atuam sobre os dados, à medida que eles passam pelos diferentes estágios do sistema.

Para ilustrar como o diagrama de fluxo de dados pode descrever um projeto orientado a funções, considere a Figura 26.8, que mostra o projeto de um sistema de folha de pagamento.

Figura 26.8 ▲

Diagrama de fluxo de dados de um sistema de folha de pagamento.

Esse projeto seria, normalmente, implementado como um sistema de processamento em lotes. Nesse sistema, as informações sobre os funcionários da organização são lidas para o sistema, o salário mensal e as deduções são calculados e os pagamentos são realizados. É possível ver como esse sistema segue a estrutura básica de entrada–processo–saída, discutida anteriormente:

1. As funções à esquerda do diagrama, 'Leia o registro de funcionários', 'Leia os dados de pagamento mensal' e 'Valide os dados do funcionário', fazem a entrada dos dados para cada funcionário e verificam esses dados.
2. A função 'Calcule o salário' calcula o salário bruto total para cada funcionário e as várias deduções feitas nesse salário. Em seguida, o salário mensal líquido é calculado.
3. As funções de saída, à direita do diagrama, registram uma série de arquivos que contém os detalhes das deduções feitas e o salário a ser pago. Esses arquivos são processados por outros programas, uma vez que os detalhes para todos os funcionários tenham sido computados. Um aviso de pagamento para o funcionário, que registra o pagamento líquido e a dedução feita, é impresso pelo sistema.

Um exemplo de um sistema de processamento de transações é o software que controla os caixas automáticos de bancos (ATMs). O serviço fornecido a um usuário é independente dos serviços anteriores que foram fornecidos; assim, podemos pensar nesse serviço como uma única transação. A Figura 26.9 ilustra um projeto funcional simplificado desse tipo de sistema. Utilizei uma linguagem de descrição de projeto orientada a funções, em vez de Java. Java é uma linguagem orientada a objetos, e não acho que seja natural utilizá-la para descrever projetos orientados a funções.

Nesse projeto, o sistema é implementado como um loop contínuo e as ações são ativadas quando um cartão é inserido. Funções como Dispense_cash, Get_account_number, Order_statement, Order_checkbook etc. podem ser identificadas e implementam as ações do sistema. O estado de sistema mantido pelo programa é mínimo. Os serviços de usuário operam independentemente e não interagem uns com os outros.

Cada ATM na rede do banco está gerando uma transação; assim, o sistema bancário deve ter algum meio de gerenciar transações potencialmente simultâneas e atualizar seu banco de

Figura 26.9

Descrição de projeto de um ATM.

```
ENTRADA

loop
    repeat
        Print_input_message (" Welcome - Please enter your card") ;
    until Card_input ;

    Account_number := Read_card ;
    Get_account_details (PIN, Account_balance, Cash_available) ;

PROCESSO

    if Invalid_card (PIN) then
        Retain_card ;
        Print ("Card retained - please contact your bank") ;
    else
    repeat
        Print_operation_select_message ;
        Button := Get_button ;
        case Get_button is
            when Cash_only =>
                Dispense_cash (Cash_available, Amount_dispensed) ;
            when Print_balance =>
                Print_customer_balance (Account_balance) ;
            when Statement =>
                Order_statement (Account_number) ;
            when Check_book =>
                Order_checkbook (Account_number) ;
        end case ;
        Print ("Press CONTINUE for more services or STOP to finish") ;
        Button := Get_button ;
    until Button = STOP ;

SAÍDA

    Eject_card ;
    Print ("Please take your card") ;
    Update_account_information (Account_number, Amount_dispensed) ;

end loop ;
```

dados de uma maneira controlada. Geralmente, essas ações são realizadas utilizando-se um monitor de teleprocessamento, como mostra a Figura 26.5.

Manter os sistemas legados em operação é uma boa razão para o projeto orientado a funções continuar a ser utilizado por muitos anos. Contudo, o uso dessa abordagem não está restrito aos sistemas legados. Pode ser apropriado utilizar essa técnica em novos desenvolvimentos de sistemas:

1. Quando os sistemas de processamento de dados a serem implementados dependerem do processamento de transações e da atualização de bancos de dados. O programa que processa as transações não precisa manter informações sobre as transações precedentes, e, assim, os objetos que mantêm dados privativos são desnecessários; eles seguem o modelo entrada–processo–saída, discutido anteriormente.
2. Quando uma empresa tiver investido intensamente em métodos estruturados, ferramentas CASE associadas e treinamento de pessoal. Os riscos e os custos de mudar para uma abordagem orientada a objetos para o projeto de programa podem não ser justificáveis.

Embora muitos desenvolvedores de software considerem o projeto orientado a funções uma abordagem ultrapassada, o desenvolvimento orientado a objetos pode não oferecer vantagens significativas, nessas situações. Um desafio interessante que enfrentamos é assegurar que os sistemas orientados a funções e orientados a objetos possam trabalhar em conjunto.

26.3 Avaliação dos sistemas legados

As organizações que dependem de muitos sistemas legados e têm um orçamento limitado para a manutenção e a atualização desses sistemas precisam decidir como obter o melhor retorno de seu investimento. Isso significa que elas devem fazer uma avaliação realista de seus sistemas legados e, então, decidir sobre qual é a estratégia mais apropriada para que esses sistemas evoluam. Existem quatro opções estratégicas:

1. *Descartar completamente o sistema* Essa opção dever ser escolhida quando o sistema não estiver prestando uma contribuição efetiva aos processos de negócios. Isso ocorre quando os processos corporativos foram modificados, desde que o sistema foi instalado, e não são mais inteiramente dependentes do sistema. Essa situação é muito comum quando terminais de mainframe são substituídos por PCs e quando produtos de software de prateleira, nessas máquinas, são adaptados para fornecer o apoio computacional de que o processo de negócios precisa.

2. *Continuar a manter o sistema* Essa opção deve ser escolhida quando o sistema ainda é necessário, exceto quando ele for bastante estável e os usuários não pedirem um grande número de modificações no sistema.

3. *Transformar o sistema de alguma maneira para melhorar sua facilidade de manutenção* Essa opção deve ser escolhida quando a qualidade do sistema foi degradada por modificações regulares e quando essas mudanças regulares ainda forem exigidas. Esse tópico é abordado no Capítulo 28, que discute as técnicas de reengenharia de sistema.

4. *Substituir o sistema por um novo* Essa opção deve ser escolhida quando outros fatores, como hardware novo, significarem que o sistema antigo não pode continuar em operação ou quando sistemas de prateleira estiverem disponíveis e permitirem que o novo sistema seja desenvolvido por um custo razoável.

Naturalmente, essas opções não são exclusivas; assim, quando um sistema é composto por vários programas diferentes, diferentes opções podem ser aplicadas em diferentes partes do sistema.

Quando se está avaliando um sistema legado, deve-se considerá-lo sob duas perspectivas diferentes (Warren, 1998). A partir de uma perspectiva de negócios, deve-se fazer uma avaliação do valor desse sistema para a empresa. A partir de uma perspectiva de sistema, deve-se fazer uma avaliação da qualidade do software de aplicação e do software e do hardware de apoio do sistema. A combinação do valor de negócios com a qualidade do sistema é, então, utilizada para ajudar a informar a decisão sobre o que fazer com o sistema legado.

Para ilustrar essa avaliação, vamos supor que uma organização tenha dez sistemas legados. A qualidade e o valor de negócios de cada um desses sistemas são avaliados e comparados com outros sistemas, mediante um gráfico, que mostra o valor de negócios relativo e a qualidade do sistema. Esse gráfico está ilustrado na Figura 26.10.

Pela Figura 26.10, pode-se ver que existem quatro grupos de sistemas:

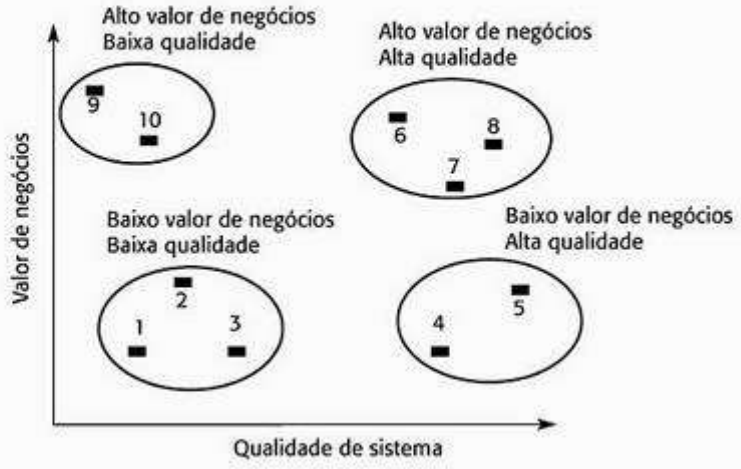
1. *Baixa qualidade, baixo valor de negócios* Manter esses sistemas em operação será dispendioso, e a taxa de retorno de investimento para os negócios será bastante pequena. Esses sistemas são candidatos a serem descartados.

2. *Baixa qualidade, alto valor de negócios* Esses sistemas estão prestando uma importante contribuição à empresa e, assim, não podem ser descartados. Contudo, sua baixa qualidade significa que os custos operacionais são altos, de modo que são candidatos à transformação ou à substituição do sistema, se um sistema adequado estiver disponível.

3. *Alta qualidade, baixo valor de negócios* São sistemas que não contribuem muito para os negócios, mas cuja manutenção pode não ser muito dispendiosa. Não vale o risco

Figura 26.10

Qualidade de sistema e valor de negócios.



substituir esses sistemas, de modo que a manutenção normal pode ser continuada ou eles podem ser descartados.

4. *Alta qualidade, alto valor de negócios* Esses sistemas devem ser mantidos em operação, mas sua alta qualidade significa que não é necessário investir na sua transformação ou substituição. Deve continuar a manutenção normal do sistema.

O ideal é que seja utilizada a avaliação objetiva para informar as decisões sobre o que fazer com um sistema legado. Contudo, em muitos casos, essas decisões não são realmente objetivas, mas se baseiam em considerações organizacionais ou políticas. Por exemplo, se duas empresas fazem uma fusão, os sistemas utilizados pela empresa com maior poder serão utilizados naturalmente pela empresa resultante e os outros sistemas poderão ser descartados. Se a gerência sênior em uma organização tomar a decisão de migrar para uma nova plataforma de hardware, então isso poderá exigir que aplicações sejam substituídas. Se não houver nenhum orçamento disponível para a transformação do sistema em um determinado ano, então a manutenção do sistema poderá continuar, embora isso venha a resultar em custos de longo prazo mais elevados.

26.3.1 Avaliação do valor de negócios

A avaliação do valor de negócios de um sistema é um julgamento subjetivo, e não existe nenhum método objetivo confiável que possa ser utilizado. Como ocorre com todos os processos subjetivos, se confiarmos simplesmente em uma opinião, é possível que obtenhamos um valor distorcido. Portanto, recomendo a adoção de uma abordagem orientada a pontos de vista (veja o Capítulo 6), na qual se identifica uma série de pontos de vista de negócios e se faz uma avaliação de valores, a partir de cada um desses pontos de vista. Os pontos de vista que devem ser considerados e as possíveis questões são:

1. *Usuários finais do sistema* Qual é a eficácia do sistema que apóia seus processos de negócios? Quanto da funcionalidade do sistema é utilizada?

2. *Clientes* O uso do sistema é transparente para os clientes ou suas interações são limitadas pelo sistema? Eles têm de esperar por causa do sistema? Os erros do sistema têm um impacto direto sobre os clientes?

3. *Gerentes de linha* Os gerentes pensam que o sistema é eficaz em contribuir para o sucesso de sua unidade? Os custos de manutenção do sistema em uso são justificados? Os dados manipulados pelo sistema são importantes para o funcionamento da unidade do gerente?

4. *Gerentes de TI* Existem dificuldades em encontrar pessoas para trabalhar no sistema? O sistema consome recursos que poderiam ser implementados mais efetivamente em outros sistemas?

5. *Gerentes seniores* O sistema e o processo de negócios associado prestam uma contribuição eficaz aos objetivos da empresa?

Uma vez identificados os pontos de vista, as pessoas relacionadas a cada um deles devem ser entrevistadas e suas respostas devem ser registradas. Isso deverá fornecer um quadro geral dos valores do sistema para a empresa e, em seguida, é possível fazer uma avaliação de seu valor de negócios.

26.3.2 Avaliação da qualidade do sistema

Vimos, na Figura 26.2, que um sistema legado não é apenas um sistema de software de aplicação, mas também inclui processos de negócios, o hardware e o ambiente de software de apoio do sistema. Para avaliar a qualidade do sistema, deve-se examinar todos os seus níveis. Uma vez coletadas informações sobre todos os aspectos do sistema, estamos em condições de fazer um julgamento sobre a qualidade dele e onde ele pode ser posicionado na Figura 26.10. Contudo, não existe nenhum método sistemático de fazer esse julgamento: ele depende dos sistemas individuais e de negócios que utilizam esse sistema.

Avaliação do processo de negócios

Avaliar a qualidade de um processo de negócios está estreitamente relacionado a avaliar o valor de negócios do sistema. Durante a avaliação do valor corporativo, será necessário fazer perguntas sobre o processo de negócios, e isso proporcionará alguma compreensão da eficácia desse processo, no que se refere a apoiar as metas de negócios.

Contudo, como o processo faz parte do sistema legado, talvez você queira descobrir informações mais detalhadas sobre ele, para que isso possa ajudá-lo a julgar se o processo é de alta qualidade ou não. Isso é importante, porque os processos de baixa qualidade exigem melhorias, o que leva a mudanças associadas ao sistema de software.

Para avaliar o processo, recomendo uma abordagem orientada a pontos de vista, comparável àquela utilizada para a avaliação do valor de negócios. Exemplos de perguntas que podem ser feitas são:

1. Existe um modelo definido do processo e existem procedimentos implantados para verificar se esse modelo é seguido?

2. As diferentes partes da organização têm os mesmos processos para as mesmas funções?

3. Como o pessoal envolvido no processo adaptou o modelo a fim de torná-lo mais adequado a seu trabalho?

4. As relações com outros processos de negócios são necessárias? Em caso positivo, elas são claras para o pessoal envolvido no processo?

5. O processo é compatível com o software legado de aplicação? Ele fornece as informações requeridas? O processo exige que os mesmos dados sejam fornecidos diversas vezes, em diferentes lugares?

Não fique surpreso se as respostas obtidas para essas perguntas, a partir de diferentes pessoas, forem totalmente diferentes. A gerência em uma organização pode pensar que ela tem processos de alta qualidade, mas os processos de fato utilizados podem ser totalmente diferentes daqueles supostos pelos gerentes. Quando se faz uma avaliação de processo, é preciso enfocar o verdadeiro processo utilizado e não se deve confiar somente na documentação de processo para fazer julgamentos sobre sua qualidade.

Figura 26.11 ▶

Fatores utilizados na avaliação do ambiente.

Fator	Questões
Estabilidade do fornecedor	O fornecedor ainda existe? O fornecedor é financeiramente estável e a continuação de sua existência é provável? Se o fornecedor não existe mais, alguma outra empresa faz sua manutenção?
Taxa de falhas	O hardware tem uma alta taxa de falhas relatadas? O software de apoio 'trava' e força novas inicializações do sistema?
Idade	Qual é a idade do hardware e do software? Quanto mais antigos forem o hardware e o software de apoio, mais obsoletos eles serão. Eles podem ainda funcionar corretamente, mas pode haver uma economia significativa e também benefícios de negócios em mudar para sistemas mais modernos.
Desempenho	O desempenho do sistema é adequado? Os problemas de desempenho têm efeito significativo sobre os usuários de sistema?
Requisitos de suporte	Qual suporte local é exigido pelo hardware e pelo software? Se houver altos custos associados com esse suporte, poderá valer a pena considerar a substituição do sistema.
Custos de manutenção	Quais são os custos de manutenção de hardware e licenças de software de apoio? Hardwares mais antigos podem ter custos de manutenção mais altos do que os sistemas modernos. O software de apoio pode ter altos custos anuais de licença.
Interoperabilidade	Existem problemas de interface do sistema com outros sistemas? Os compiladores podem ser utilizados com versões atuais do sistema operacional? A emulação de hardware é necessária?

Avaliação do ambiente

A partir da Figura 26.2, podemos ver que o ambiente de um sistema de software de aplicação inclui o software de apoio (sistema operacional, compiladores, utilitários etc.) utilizado pelo sistema e a plataforma de hardware na qual o sistema de aplicação é executado. É necessário avaliar esse ambiente, uma vez que os fatores ambientais são, muitas vezes, os direcionadores das alterações no software de aplicação.

Novamente, a avaliação do ambiente de sistema é um processo de julgamento, que é informado por medições do sistema e seus processos de manutenção. Entre os exemplos de dados que podem ser úteis estão os custos de manutenção do hardware de sistema e do software de apoio, o número de defeitos de hardware que ocorrem durante um certo período e a freqüência de *patches* e reparos aplicados ao software de apoio do sistema.

Os fatores que devem ser considerados durante a avaliação do ambiente são mostrados na Figura 26.11. Observe que esses fatores não são todos características técnicas do ambiente. Deve-se considerar também a confiabilidade dos fornecedores do hardware e do software de apoio. Se esses fornecedores não estiverem mais atuando no mercado, isso significará que pode não haver mais suporte de manutenção para seus sistemas.

Avaliação do software de aplicação

A avaliação da qualidade do software de aplicação existente é diferente das atividades de garantia de qualidade, que ocorrem durante o desenvolvimento de software (veja o Capítulo 24). O sistema legado pode ter sido desenvolvido com a utilização de técnicas e padrões que não estão mais em uso, a estrutura do sistema pode ter sido inevitavelmente corrompida pelas alterações e a documentação do sistema pode estar desatualizada.

Figura 26.12 ▶

Fatores utilizados na avaliação de aplicações.

Fator	Questões
Facilidade de compreensão	Qual é o nível de dificuldade para se entender o código-fonte do sistema atual? Qual é a complexidade das estruturas de controle que são utilizadas? As variáveis têm nomes significativos que refletem sua função?
Documentação	Que documentação de sistema está disponível? A documentação está completa, consistente e atualizada?
Dados	Existe um modelo de dados explícito para o sistema? Até que ponto os dados são duplicados em arquivos diferentes? Os dados utilizados pelo sistema estão atualizados e consistentes?
Desempenho	O desempenho da aplicação é adequado? Os problemas de desempenho têm um efeito significativo nos usuários de sistema?
Linguagem de programação	Modernos compiladores estão disponíveis para a linguagem de programação utilizada para desenvolver o sistema? A linguagem de programação ainda é utilizada para novos desenvolvimentos de sistema?
Gerenciamento de configuração	Todas as versões de todas as partes do sistema são gerenciadas por um sistema de gerenciamento de configuração? Há uma descrição explícita das versões de componentes utilizados no sistema corrente?
Dados de teste	Existem dados de teste para o sistema? Há um registro de testes de regressão realizados quando novas características são acrescentadas ao sistema?
Habilidades pessoais	Há pessoas disponíveis que tenham as habilidades para a manutenção da aplicação? Existe apenas um número limitado de pessoas que compreendem o sistema?

Alguns dos fatores que se pode utilizar para fazer um julgamento da qualidade do software de aplicação estão mostrados na Figura 26.12. Encontrar as respostas às perguntas associadas com esses fatores ajuda a avaliar a qualidade do sistema. Também é possível coletar dados quantitativos do sistema, que darão mais informações nas quais o julgamento pode se basear.

Exemplos de dados quantitativos que podem ser coletados são:

1. *O número de pedidos de modificações no sistema* As mudanças efetuadas no sistema tendem a corromper sua estrutura e a tornar outras mudanças mais difíceis. Quanto maior esse número, mais baixa será a qualidade do sistema.
2. *O número de diferentes interfaces com o usuário utilizadas pelo sistema* Esse é um fator importante nos sistemas baseados em formulários, quando cada formulário pode ser considerado como uma interface separada com o usuário. Quanto mais as interfaces forem diferentes, mais provável será a existência de inconsistências e redundâncias nessas interfaces.
3. *O volume de dados utilizados pelo sistema* Quanto maior o volume de dados (número de arquivos, tamanho de banco de dados etc.), mais complexo será o sistema.

Embora esses dados sejam utilizados freqüentemente, sua coleta pode ser muito dispendiosa. Isso pode significar que é impraticável coletá-los para a avaliação do sistema. Além disso, não existem valores absolutos que possam ser utilizados. A idade e o tamanho do sistema devem ser levados em conta quando são feitos julgamentos de qualidade baseados em medições.

PONTOS-CHAVE

- Um sistema legado é um sistema antigo que ainda fornece serviços de negócios essenciais.
- Os sistemas legados não são apenas sistemas de software de aplicações. Eles são sistemas sociotécnicos, baseados em computadores, portanto incluem processos de negócios, software de aplicação, software de apoio e hardware de sistema.
- A maior parte dos sistemas legados inclui uma série de diferentes programas e dados compartilhados associados a esses programas. Esses dados podem estar armazenados em arquivos ou em um sistema obsoleto de gerenciamento de banco de dados.
- A maioria dos sistemas legados foi projetada a partir de uma perspectiva funcional e é composta por conjuntos de funções interativas, que se comunicam por meio de parâmetros e áreas globais de dados compartilhados.
- No domínio de sistemas de negócios, a maioria dos sistemas legados é constituída de sistemas de processamento de lote ou sistemas de processamento de transação. Em ambos os casos, sua organização geral pode ser representada com a utilização de um modelo entrada–processo–saída.
- O valor de negócios de um sistema legado e a qualidade do software de aplicação, assim como seu ambiente, devem ser avaliados para ajudar a decidir entre substituir, transformar ou fazer a manutenção do sistema.
- O valor de negócios de um sistema é uma avaliação da eficácia do sistema em fornecer apoio aos objetivos do negócio.
- A qualidade do sistema depende da qualidade dos processos de negócios, da qualidade do software de aplicação em si e da qualidade do hardware e do software utilizados para apoiar o sistema.

LEITURAS SUGERIDAS

"Legacy information systems: issues and directions". Uma visão geral dos problemas dos sistemas legados, com ênfase nos problemas de dados legados. (J. Bisbal, D. Lawless, B. Wu e J. Grimson, *IEEE Software*, set./out. 1999.)

The renaissance of legacy systems. Esse livro se ocupa principalmente com um método para a evolução de sistemas legados. Contudo, ele inclui uma boa discussão geral sobre esses sistemas, estudos de caso que ilustram as estruturas dos sistemas legados e um capítulo sobre a avaliação de sistemas. (I. Warren, 1998, Springer.)

"Cash cow in the tar pit: reengineering a legacy system". Uma descrição agradável de experiências práticas com um projeto de evolução de sistema legado. (W.S. Adolph, *IEEE Software*, maio 1996.)

EXERCÍCIOS

26.1 Explique por que os sistemas legados podem ser fundamentais para a operação de uma empresa.

26.2 Sugira três razões pelas quais os sistemas de software se tornam mais difíceis de ser compreendidos, quando diferentes pessoas estão envolvidas na alteração desses sistemas.

26.3 Que dificuldades podem, provavelmente, surgir quando diferentes componentes de um sistema legado são implementados em diferentes linguagens de programação?

26.4 Por que é necessário o uso de um monitor de teleprocessamento quando os pedidos para atualizar dados em um sistema provêm de uma série de terminais diferentes? Como os modernos sistemas cliente–servidor reduzem a carga sobre o monitor de processamento?

26.5 A maioria dos sistemas legados utiliza uma abordagem orientada a funções, para o projeto. Explique por que essa abordagem de projeto pode ser mais apropriada para esses sistemas do que uma estratégia de projeto orientada a objetos.

26.6 Amplie a função 'Calcule o salário', na Figura 26.8, e desenhe um diagrama de fluxo de dados que mostre os cálculos realizados nessa função. Você precisa das seguintes informações para fazer isso:

- O registro de funcionários identifica a graduação de um funcionário, que pode ser utilizada para melhorar a tabela de pagamentos.
- Os funcionários em um nível inferior a uma determinada graduação podem receber o pagamento de horas extras com base no mesmo valor que o de suas horas normais. As horas extras a serem pagas são indicadas em seu registro do funcionário.
- Os impostos deduzidos dependem do código de imposto do funcionário (indicado no registro) e de seu salário anual. As deduções mensais para cada código e também um salário-padrão são indicados nas tabelas de impostos. Esses valores são escalonados para cima e para baixo, dependendo da relação entre o salário real e o salário-padrão utilizado.

26.7 Sob quais circunstâncias uma organização pode decidir descartar um sistema, quando a avaliação dele sugere que ele é de alta qualidade e alto valor de negócios?

26.8 Sugira dez perguntas que podem ser feitas aos usuários finais de um sistema quando se realiza uma avaliação de processo de negócios.

26.9 Explique por que os problemas com software de apoio podem significar que uma organização precisa substituir seus sistemas legados.

26.10 A gerência de uma organização pediu para você realizar uma avaliação de sistema e sugeriu que os resultados dessa avaliação mostrassem que o sistema está obsoleto e que deve ser substituído por um novo. Isso significará que uma série de mantenedores do sistema será redundante. Sua avaliação realmente mostra que o sistema tem boa manutenção e é de alta qualidade e alto valor de negócios. Como você relataria esses resultados à gerência?

27 Mudanças em software

Objetivos

Os objetivos deste capítulo são apresentar as mudanças realizadas em produtos de software e descrever várias maneiras de alterar um software. Depois de ler este capítulo, você poderá:

- compreender três diferentes estratégias para fazer alterações em sistemas de software, a saber: manutenção de software, evolução de arquitetura e reengenharia de software;
- compreender os princípios de manutenção de software e por que ele tem uma manutenção tão dispendiosa, e
- compreender como os sistemas legados podem ser transformados em sistemas cliente–servidor, a fim de prolongar sua duração e fazer uso efetivo de hardware moderno.

Sumário

27.1 Dinâmica da evolução de programas
27.2 Manutenção de software
27.3 Evolução de arquitetura

É impossível produzir sistemas de qualquer tamanho que não precisem ser modificados. Assim que o software é colocado em uso, novos requisitos emergem, e os requisitos existentes são modificados à medida que a empresa que executa esse software passa por modificações. Partes do software podem precisar ser modificadas para corrigir os erros encontrados na operação, melhorar seu desempenho ou outras características não funcionais. Tudo isso significa que, depois de serem entregues, os sistemas de software sempre evoluem, em resposta às exigências de mudanças.

As mudanças nos produtos de software são muito importantes porque hoje em dia as organizações dependem inteiramente de seus sistemas de software e investiram milhões de dólares neles. Seus sistemas são ativos importantes, e elas precisam investir em alterações nesses sistemas para manter o valor desses ativos. Como foi sugerido no Capítulo 1, um problema crucial para as organizações é implementar e gerenciar as mudanças em seus sistemas legados de modo que estes continuem a apoiar suas operações corporativas.

Há diversas estratégias para as mudanças em software (Warren, 1998):

1. *Manutenção de software* As mudanças no software são feitas em resposta a requisitos modificados, mas a estrutura fundamental do software permanece estável. Essa é a abordagem mais comum utilizada nas mudanças em sistemas. Será discutida a questão da manutenção posteriormente neste capítulo, na Seção 27.2.

2. *Transformação de arquitetura* Essa é uma abordagem mais radical das mudanças em software do que a manutenção, uma vez que envolve fazer alterações significativas na arquitetura do sistema de software. Mas, comumente, os sistemas evoluem a partir de uma arquitetura centralizada, centrada em dados, para uma arquitetura cliente-servidor. Será discutida a evolução de arquitetura na Seção 27.3.

3. *Reengenharia de software* Essa é diferente de outras estratégias, no sentido de que nenhuma funcionalidade nova é adicionada ao sistema. Em vez disso, o sistema é modificado, a fim de tornar mais fácil sua compreensão e alteração. A reengenharia de sistemas pode envolver algumas alterações estruturais, mas, geralmente, não envolve grandes mudanças de arquitetura. A reengenharia de software será abordada no Capítulo 28.

As estratégias que acabam de ser apresentadas não são mutuamente exclusivas. A reengenharia pode ser necessária para tornar o software mais fácil de ser compreendido, antes que sua arquitetura seja modificada ou que componentes sejam reutilizados. Algumas partes de um sistema podem ser substituídas por componentes de prateleira, enquanto outras partes, estáveis, são mantidas. Como foi discutido no Capítulo 26, a escolha da estratégia mais apropriada depende tanto da qualidade técnica do sistema quanto de seu valor de negócios.

Do mesmo modo, diferentes estratégias podem ser aplicadas a diferentes partes de um sistema ou a diferentes programas que constituem um sistema legado de informações. Um programa que é bem estruturado e que não necessita de manutenção freqüente pode ser mantido; outro programa, utilizado por muitas pessoas diferentes, em diferentes locais, pode ter sua arquitetura modificada, de modo que sua interface com o usuário seja executada em um computador cliente, e um terceiro programa no mesmo sistema pode ser substituído por uma alternativa de prateleira. Contudo, se os dados do sistema passam pelo processo de reengenharia, normalmente são exigidas mudanças em todos os programas do sistema.

Para alguns sistemas, nenhuma dessas estratégias de mudança é apropriada, e o sistema deve ser substituído. Como foi discutido no Capítulo 26, pode ser possível substituir parte ou todo o sistema por um sistema de prateleira (COTS). Contudo, pode não haver nenhuma alternativa de prateleira que seja adequada e, portanto, deve-se desenvolver um novo sistema personalizado. Nessa situação, pode ser possível reutilizar partes importantes do sistema original no desenvolvimento de seu substituto.

As mudanças em software inevitavelmente geram muitas versões diferentes de um sistema de software e seus componentes. É muito importante manter o acompanhamento dessas dife-

rentes versões e assegurar que as versões apropriadas dos componentes sejam utilizadas em cada versão do sistema. O gerenciamento de modificações em produtos de software é chamado de gerenciamento de configuração. Será discutido esse tópico no Capítulo 29.

27.1 Dinâmica da evolução de programas

A dinâmica de evolução de programa é o estudo da mudança no sistema. A maioria dos trabalhos nessa área tem sido realizada por Lehman e Belady (1985). A partir desses estudos, eles propuseram um conjunto de leis (Leis de Lehman) referentes a mudanças nos sistemas. Eles alegam que essas leis são invariáveis e amplamente aplicáveis. Lehman e Belady examinaram o crescimento e a evolução de uma série de grandes sistemas de software. As leis propostas foram derivadas dessas medições. Essas leis (hipóteses, na verdade) são mostradas na Figura 27.1.

A primeira lei define que a manutenção do sistema é um processo inevitável. À medida que o ambiente do sistema se modifica, surgem novos requisitos, e o sistema deve ser modificado. Quando o sistema modificado é novamente introduzido no ambiente, isso promove mais mudanças ambientais, de modo que o processo de evolução continua.

A segunda lei especifica que, à medida que um sistema é modificado, sua estrutura é degradada. Isso é observado em sistemas legados, como foi discutido no Capítulo 26. A única maneira de evitar que isso aconteça é investir na manutenção preventiva, quando se dedica tempo melhorando a estrutura do software, sem nada acrescentar a sua funcionalidade. Obviamente, isso significa custos adicionais, além daqueles de implementação das mudanças requeridas pelo sistema.

A terceira lei é, talvez, a mais interessante. Ela sugere que os sistemas de grande porte têm uma dinâmica própria, que é estabelecida em um estágio inicial do processo de desenvolvimento. Isso determina as tendências gerais do processo de manutenção do sistema e limita o número de possíveis mudanças no sistema. Lehman e Belady sugerem que essa lei é o resultado de fatores estruturais e organizacionais fundamentais.

Assim que o sistema excede um tamanho mínimo, ele age da mesma maneira que uma massa inerte. Seu tamanho inibe modificações maiores, porque as mudanças introduzem novos

Figura 27.1 ▶ Leis de Lehman.

Lei	Descrição
Mudança contínua	Um programa utilizado em um ambiente do mundo real necessariamente tem de ser modificado ou se tornará de maneira progressiva menos útil nesse ambiente.
Aumento da complexidade	À medida que um programa em evolução se modifica, sua estrutura tende a se tornar mais complexa. Recursos extras precisam ser dedicados a preservar e simplificar a estrutura.
Evolução de um programa de porte grande	A evolução do programa é um processo auto-regulador. Os atributos do sistema, como tamanho, tempo entre releases e número de erros relatados, são aproximadamente invariáveis para cada release do sistema.
Estabilidade organizacional	Durante o tempo de duração de um programa, sua taxa de desenvolvimento é aproximadamente constante e independente dos recursos dedicados ao desenvolvimento do sistema.
Conservação da familiaridade	Durante o tempo de duração de um sistema, as mudanças incrementais em cada release são aproximadamente constantes.

defeitos, que degradam a funcionalidade do sistema. Se um grande estágio de mudança for proposto, isso introduzirá muitos novos defeitos, que limitarão as modificações úteis fornecidas na nova versão do sistema.

Os sistemas de grande porte são, geralmente, produzidos por grandes organizações. Elas têm sua própria burocracia interna, que estabelece o orçamento das modificações para cada sistema e controla o processo de tomada de decisão. As maiores mudanças nos sistemas requerem a tomada de decisão organizacional e alterações no orçamento do projeto. Essas decisões tomam tempo. Durante esse tempo, outras mudanças, de maior prioridade, podem ser propostas. Pode ser necessário deixar as mudanças originais para uma data posterior. A taxa de modificações no sistema é, portanto, regida pelo processo de tomada de decisão, por parte da organização.

A quarta lei de Lehman sugere que a maioria dos grandes projetos de programação trabalha no que ele chama de um estado saturado, isto é, uma mudança nos recursos ou no pessoal tem efeitos imperceptíveis na evolução de longo prazo do sistema. Isso é também sugerido pela terceira lei, que propõe que a evolução do programa é amplamente independente das decisões de gerenciamento. Essa lei confirma que as grandes equipes de desenvolvimento de software são improdutivas, uma vez que as atividades indiretas de comunicação dominam o trabalho da equipe.

A quinta lei de Lehman diz respeito aos estágios de alterações em cada release de sistema. Acrescentar nova funcionalidade a um sistema inevitavelmente introduz novos defeitos. Quanto mais funcionalidade for acrescentada em cada release, mais defeitos ocorrerão. Portanto, um grande aumento de funcionalidade em um release do sistema significa que esse release terá de ser seguido por outro release, em que os defeitos do novo sistema sejam reparados. Relativamente pouca funcionalidade nova será incluída nessa versão. A lei sugere que não se deve orçar grandes aumentos de funcionalidade em cada versão sem levar em conta a necessidade de reparo de defeitos.

As observações de Lehman parecem ser, de modo geral, sensatas. Elas devem ser levadas em conta ao planejar o processo de manutenção. Pode ser que considerações de negócios exijam que essas observações sejam ignoradas, em qualquer momento. Por exemplo, por razões de marketing, pode ser necessário fazer diversas mudanças em sistemas maiores, em um único release. As conseqüências disso são que, provavelmente, serão necessários um ou mais releases dedicados ao reparo de erros.

Pode parecer que as diferenças radicais que são óbvias entre os releases de produtos de programa violam as leis de Lehman. Por exemplo, o Microsoft Word foi transformado de um simples processador de textos, que operava em 256K de memória, para um sistema gigante, repleto de recursos. Atualmente, ele precisa de muitos megabytes de memória e de um processador rápido, para que seja executado. Sua evolução parece contradizer a quarta e a quinta lei de Lehman. Contudo, suspeito que esses programas não sejam realmente uma seqüência de revisões. Em vez disso, o mesmo nome tem sido mantido, por razões de marketing, mas o programa em si tem sido amplamente reescrito entre releases.

27.2 Manutenção de software

A manutenção de software é o processo geral de modificação de um sistema depois que ele foi colocado em uso. As modificações podem ser simples, destinadas a corrigir erros de código, mais extensas, a fim de corrigir os erros de projeto, ou significativas, com a finalidade de corrigir erros de especificação ou acomodar novos requisitos. Como foi dito na introdução, a manutenção de software não envolve, normalmente, mudanças maiores na arquitetura do sistema. As mudanças são implementadas pela alteração dos componentes de sistema já existentes e, quando necessário, adicionando-se novos componentes ao sistema.

Existem três diferentes tipos de manutenção de software:

1. *Manutenção para reparar os defeitos no software* Em geral, a correção de erros de codificação é um processo relativamente barato; os erros de projeto são mais dispendiosos, uma vez que podem envolver a reprogramação de diversos componentes. Os erros de requisitos são os mais dispendiosos de corrigir, devido à extensiva atividade de reprojeto, que pode ser necessária.
2. *Manutenção para adaptar o software a um ambiente operacional diferente* Esse tipo de manutenção é necessário quando algum aspecto do ambiente de sistema é modificado, como o hardware, o sistema operacional da plataforma ou outras modificações no software de apoio. O sistema de aplicação deve ser modificado, para que seja adaptado, a fim de lidar com essas mudanças ambientais.
3. *Manutenção para fazer acréscimos à funcionalidade do sistema ou modificá-la* Esse tipo de manutenção é necessária quando os requisitos de sistema são modificados, em resposta a mudanças organizacionais ou de negócios. A escala das mudanças requerida para o software é, freqüentemente, muito maior do que para outros tipos de manutenção.

Na prática, não há uma distinção nítida entre esses diferentes tipos de manutenção. Os defeitos de software podem ser revelados porque um sistema foi utilizado de maneira imprevista, e a melhor maneira de reparar esses defeitos pode ser acrescentar nova funcionalidade, a fim de ajudar os usuários com o sistema. Ao adaptar o software a um novo ambiente, a funcionalidade pode ser acrescentada, a fim de tirar vantagem de novos recursos compatíveis com o ambiente. Pode ser necessário acrescentar nova funcionalidade a um sistema, porque os defeitos modificaram os padrões de utilização desse sistema, e um efeito colateral da nova funcionalidade é remover os defeitos do software.

Embora esses diferentes tipos de manutenção sejam geralmente reconhecidos, pessoas diferentes, algumas vezes, lhes dão nomes diferentes. A manutenção corretiva é universalmente utilizada para se referir à manutenção para o reparo de defeitos. Contudo, a manutenção adaptativa, algumas vezes, significa adaptação a um novo ambiente e, outras vezes, significa adaptar o software a novos requisitos. A manutenção evolutiva, às vezes, significa aperfeiçoar o software implementando novos requisitos e, em outros casos, se refere a manter a funcionalidade do sistema, melhorando sua estrutura e seu desempenho. Em razão dessa nomenclatura incerta, evitei utilizar todos esses termos neste capítulo.

É difícil encontrar números atualizados para o esforço relativo dedicado aos diferentes tipos de manutenção. Um levantamento um tanto antigo, feito por Lientz e Swanson (1980), constatou que cerca de 65 por cento da manutenção estava relacionada a implementar novos requisitos, 18 por cento, a modificar o sistema para adaptá-lo a um novo ambiente operacional, e 17 por cento, a corrigir defeitos do sistema (Figura 27.2). Números semelhantes foram

Figura 27.2 ▶

Distribuição de esforço de manutenção.

relatados por Nosek e Palvia (1990), dez anos mais tarde. Para os sistemas personalizados, acho que essa distribuição de custos ainda é aproximadamente correta.

A partir desses números, podemos observar que reparar defeitos em um sistema não é a atividade de manutenção mais dispendiosa. Em vez disso, a evolução do sistema para atender a novos ambientes e a requisitos novos ou modificados consome a maior parte do esforço de manutenção. A manutenção é, portanto, uma continuação do processo de desenvolvimento de sistema, com atividades associadas de especificação, projeto, implementação e testes. Um modelo em espiral, como o mostrado na Figura 27.3, é, por conseguinte, uma representação melhor do processo de software do que outras, como o modelo em cascata (veja a Figura 3.1), em que a manutenção é representada como uma atividade de processo separada.

Os custos de manutenção do sistema representam uma grande proporção do orçamento da maioria das organizações que utilizam sistemas de software. Na década de 80, Lientz e Swanson constataram que as grandes organizações dedicavam, pelo menos, 50 por cento do total de seu esforço de programação para a evolução dos sistemas já existentes. McKee (1984) constatou uma distribuição similar de esforço de manutenção em diferentes tipos de manutenção, mas sugeriu que a quantidade de esforço gasto em manutenção é de cerca de 65 a 75 por cento do esforço total disponível. Uma vez que as organizações substituíram os antigos sistemas por sistemas de prateleira, como os sistemas de planejamento de recursos corporativos, esse número pode não ter diminuído. Embora os detalhes possam ser incertos, sabemos que as alterações em software continuam sendo um custo importante para todas as organizações.

Os custos de manutenção, como uma proporção dos custos de desenvolvimento, variam de um domínio de aplicação para outro. Para os sistemas de aplicação de negócios, um estudo feito por Guimarães (1983) mostrou que os custos de manutenção eram amplamente comparáveis com os custos de desenvolvimento de sistemas. Para os sistemas embutidos de tempo real, os custos de manutenção podem ser até quatro vezes maiores do que os custos de desenvolvimento. Os elevados requisitos de confiabilidade e desempenho desses sistemas podem exigir que módulos sejam estreitamente vinculados e, como conseqüência, difíceis de ser modificados.

De modo geral, investir esforço ao projetar e implementar um sistema para reduzir os custos de manutenção é uma opção eficaz, em termos de custos. É mais dispendioso acrescentar funcionalidade depois da entrega, por causa da necessidade de compreender o sistema existente e analisar o impacto das mudanças no sistema. Portanto, qualquer trabalho feito durante o desenvolvimento para reduzir o custo dessa análise é útil para reduzir custos de manutenção. Boas técnicas de engenharia de software, como a especificação precisa, o uso do desenvolvimento orientado a objetos e do gerenciamento de configuração, contribuem para a redução dos custos de manutenção.

Figura 27.3 ▶

Modelo espiral de desenvolvimento.

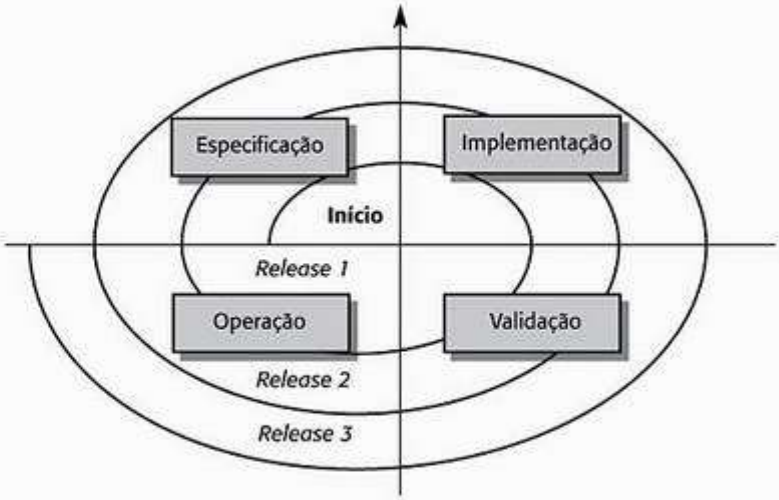
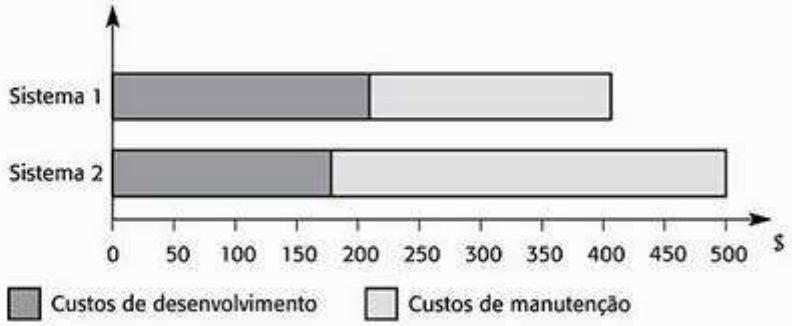


Figura 27.4

Custos de desenvolvimento e de manutenção.



A Figura 27.4 mostra como os custos gerais, ao longo do tempo de duração, podem diminuir, à medida que mais esforço é dedicado, durante o desenvolvimento do sistema, a fim de produzir um sistema com maior facilidade de manutenção. Em razão da redução em potencial dos custos de compreensão, análise e testes, existe um significativo efeito multiplicador, quando o sistema é desenvolvido visando a facilidade de manutenção. Para o Sistema 1, os custos extras de desenvolvimento, equivalentes a 25 mil dólares, são investidos para fazer o sistema ter uma manutenção mais fácil. Isso resulta em uma economia de cem mil dólares em custos de manutenção, durante o tempo de vida útil do sistema, supondo que um aumento percentual nos custos de desenvolvimento resulte em uma redução percentual comparável nos custos gerais do sistema.

Uma razão importante pela qual os custos de manutenção são altos é que é mais dispendioso acrescentar funcionalidade depois que um sistema estiver em operação do que implementar a mesma funcionalidade durante o desenvolvimento. Os principais fatores que distinguem o desenvolvimento e a manutenção e que levam a custos de manutenção mais altos são:

1. *Estabilidade da equipe* Depois de um sistema ter sido entregue, é normal a equipe de desenvolvimento se dispersar para trabalhar em novos projetos. A nova equipe ou os indivíduos responsáveis pela manutenção do sistema não compreendem o sistema ou o background das decisões de projeto do sistema. É necessário bastante esforço durante o processo de manutenção, a fim de compreender o sistema existente, antes de implementar mudanças nele.

2. *Responsabilidade contratual* O contrato para fazer a manutenção de um sistema, geralmente, é separado do contrato de desenvolvimento do sistema. O contrato de manutenção pode ser concedido a uma empresa diferente, no lugar do desenvolvedor original do sistema. Esse fator, juntamente com a falta de estabilidade da equipe, significa que não há nenhum incentivo para uma equipe de desenvolvimento escrever o software de maneira que seja fácil de ser modificado. Se uma equipe de desenvolvimento puder optar pela simplificação, a fim de economizar esforço durante o desenvolvimento, vale a pena que proceda dessa maneira, mesmo que isso signifique aumento dos custos de manutenção.

3. *Habilidade da equipe* O pessoal da manutenção, freqüentemente, tem pouca experiência e não está familiarizado com o domínio da aplicação. A manutenção tem uma imagem ruim entre os engenheiros de software. Ela é vista como um processo que requer menos habilidade do que o desenvolvimento do sistema e geralmente é designada para o pessoal mais novo. Além disso, os sistemas podem ter sido escritos em linguagens de programação obsoletas. O pessoal da manutenção pode não ter muita experiência no desenvolvimento com base nessas linguagens e precisa aprendê-las para fazer a manutenção do sistema.

4. *Idade e estrutura do programa* À medida que os programas envelhecem, suas estruturas tendem a se tornar mais difíceis de ser entendidas e modificadas. Além disso, muitos sistemas legados foram desenvolvidos sem as modernas técnicas de engenharia de software. Elas nunca foram bem estruturadas e eram freqüentemente otimizadas com vistas à eficiência, e não à facilidade de compreensão. A documentação dos sistemas antigos pode ter sido perdida ou pode estar inconsistente. Os sistemas antigos podem não ter passado pelo gerenciamento de configuração; assim, perde-se tempo procurando as versões certas dos componentes de sistema que devem ser modificados.

Os primeiros três problemas surgem do fato de que muitas organizações ainda fazem distinção entre o desenvolvimento e a manutenção de sistemas. A manutenção é vista como uma atividade de segunda classe, e não há nenhum incentivo em gastar dinheiro durante o desenvolvimento para reduzir os custos de modificações no sistema. A única solução no longo prazo para esse problema é aceitar que os sistemas raramente têm um tempo de vida útil definido, mas continuam em uso, de alguma maneira, por um período indefinido.

Em vez de desenvolver sistemas, mantê-los até que seja impossível realizar sua manutenção e, então, substituí-los, temos de adotar a noção de sistemas evolucionários. Os sistemas evolucionários são aqueles projetados para evoluírem e serem modificados em resposta a novas exigências. Eles podem ser criados a partir de sistemas legados já existentes, melhorando suas estruturas por meio da reengenharia (veja o Capítulo 28) e evoluindo a arquitetura desses sistemas, como será discutido na Seção 27.3.

A última questão nessa lista, ou seja, o problema com a estrutura de sistema degradada, é, de alguma maneira, o problema mais fácil de tratar. As técnicas de reengenharia podem ser aplicadas para melhorar a estrutura do sistema e sua compreensão. Se for apropriado, a transformação da arquitetura (discutida posteriormente neste capítulo) poderá adaptar o sistema ao novo hardware. O trabalho de manutenção preventiva (essencialmente, a reengenharia incremental) pode ser aceito para melhorar o sistema e torná-lo mais fácil de ser modificado.

27.2.1 O processo de manutenção

Os processos de manutenção variam consideravelmente, dependendo do tipo de software que está em manutenção, dos processos de desenvolvimento utilizados em uma organização e do pessoal envolvido no processo. Em algumas organizações, a manutenção pode ser um processo informal. A maioria dos pedidos de manutenção surge de conversações entre os usuários e os desenvolvedores do sistema. Em outras organizações, esse é um processo formalizado, com documentação estruturada produzida a cada estágio do processo. Contudo, em um nível abstrato, todos os processos de manutenção têm as mesmas atividades fundamentais de análise de modificações, planejamento de releases, implementação de sistemas e liberação de um sistema para os clientes.

O processo de manutenção é iniciado por um conjunto de pedidos de mudança por parte dos usuários do sistema, da gerência ou dos clientes. O custo e o impacto dessas mudanças são calculados para ver quanto do sistema é afetado pela mudança e quanto pode custar para implementar essa mudança. Se as mudanças propostas forem aceitas, um novo release do sistema será planejado. Durante esse planejamento, todas as mudanças propostas (reparo de defeitos, adaptação e novas funcionalidades) são consideradas e, então, é tomada uma decisão sobre quais mudanças devem ser implementadas na próxima versão do sistema. As mudanças são implementadas e validadas e uma nova versão do sistema é liberada. Depois disso, o processo se repete com um novo conjunto de mudanças propostas para a nova versão. A Figura 27.5, adaptada de Arthur (1988), mostra uma visão geral desse processo.

Em uma situação ideal, o estágio de implementação de mudanças desse processo deve modificar a especificação, o projeto e a implementação do sistema, a fim de refletir a modificação no sistema (Figura 27.6). Os novos requisitos que refletem as mudanças no sistema são propostos, analisados e validados. Os componentes do sistema são reprojetados e implementados,

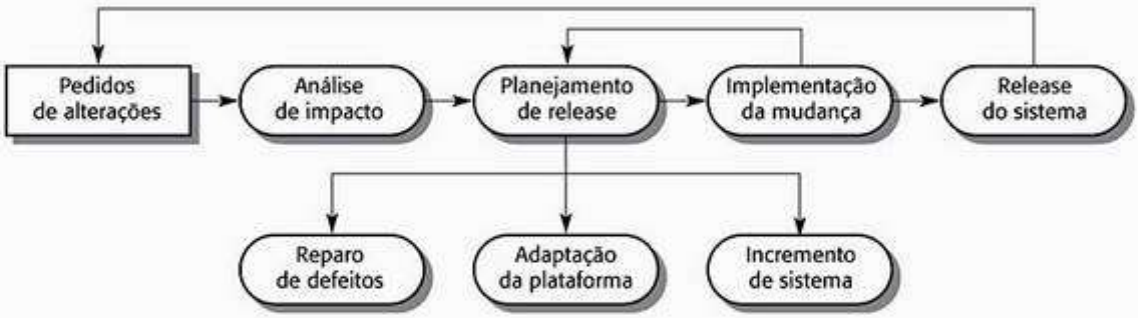


Figura 27.5 ▲
Visão geral do processo de manutenção.

e o sistema é testado novamente. Se for apropriado, a prototipação das mudanças propostas poderá ser realizada, como parte do processo de análise de mudanças.

Durante esse processo, os requisitos são analisados em detalhes e, freqüentemente, a partir dessas mudanças, surgem implicações que não eram aparentes no processo de análise inicial de modificação. Isso significa que as mudanças propostas podem ser modificadas e que pode ser necessário um maior entendimento com o cliente antes que essas mudanças sejam implementadas.

Os pedidos de modificações algumas vezes se relacionam com problemas no sistema que devem ser resolvidos com muita urgência. Essas mudanças urgentes podem surgir por três razões:

1. a emergência de um defeito de sistema que deve ser reparado para permitir a continuação da operação normal;
2. mudanças ambientais, que têm efeitos inesperados sobre o sistema, e
3. mudanças nos negócios que não foram previstas, que podem ser devidas ao surgimento de novos concorrentes ou de uma nova legislação.

Nesses casos, em geral, é mais importante fazer a mudança rapidamente do que assegurar que o processo formal de modificação seja seguido. Em vez de modificar os requisitos e o projeto, um reparo de emergência é feito no código do sistema, a fim de implementar a mudança (Figura 27.7). Contudo, o perigo dessa abordagem é que os requisitos, o projeto de software e o código, gradualmente, se tornam inconsistentes. É difícil evitar que isso aconteça, porque pode ser difícil implementar a mudança rapidamente. Os engenheiros de manutenção podem ser orientados no sentido de tratar de novos reparos de emergência no software, e a solução adequada é adiada. Se o engenheiro que fez a mudança no código deixar a equipe antes de o projeto ser atualizado, será difícil para o substituto readequar as mudanças aos requisitos e ao projeto.

Outro problema com os reparos de emergência em sistema é que eles precisam ser completados o quanto antes. Pode ser escolhida a solução possível, em vez de a melhor solução, no que diz respeito à estrutura do sistema. Isso acelera o processo de envelhecimento do software, de modo que as futuras mudanças se tornam progressivamente mais difíceis e os custos de manutenção aumentam.

Em uma situação ideal, quando são feitos reparos de emergência no código, o pedido de modificação deve permanecer em evidência, depois que os defeitos do código tiverem sido corrigidos. Eles podem então ser implementados novamente, com mais cuidado, depois que mais análises forem efetuadas. Naturalmente, o código do reparo pode ser reutilizado. Como alternativa, a melhor solução para o problema pode ser descoberta quando mais tempo for dedi-

Figura 27.6 ▶
Implementação de mudanças.

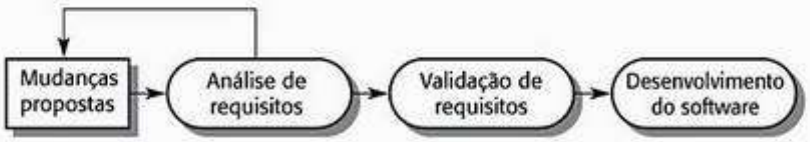
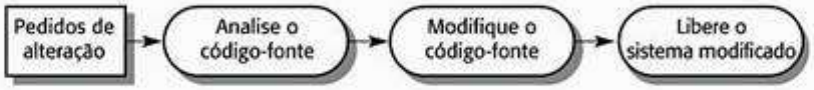


Figura 27.7 ▶
Processo de reparos de emergência.



cado à análise. Na prática, contudo, é quase inevitável que essas mudanças tenham uma baixa prioridade, e, após outras mudanças de sistema serem feitas, não mais é considerado realista refazer os reparos de emergência.

27.2.2 Previsão da manutenção

Os gerentes detestam surpresas, especialmente se elas resultam em custos altos inesperados. Portanto, tem sentido que eles procurem prever que mudanças no sistema possivelmente serão pedidas, que partes do sistema podem causar maiores dificuldades para o pessoal de manutenção e quais serão os custos gerais de manutenção para um sistema em um determinado período. A Figura 27.8 ilustra essas diferentes previsões e as questões associadas.

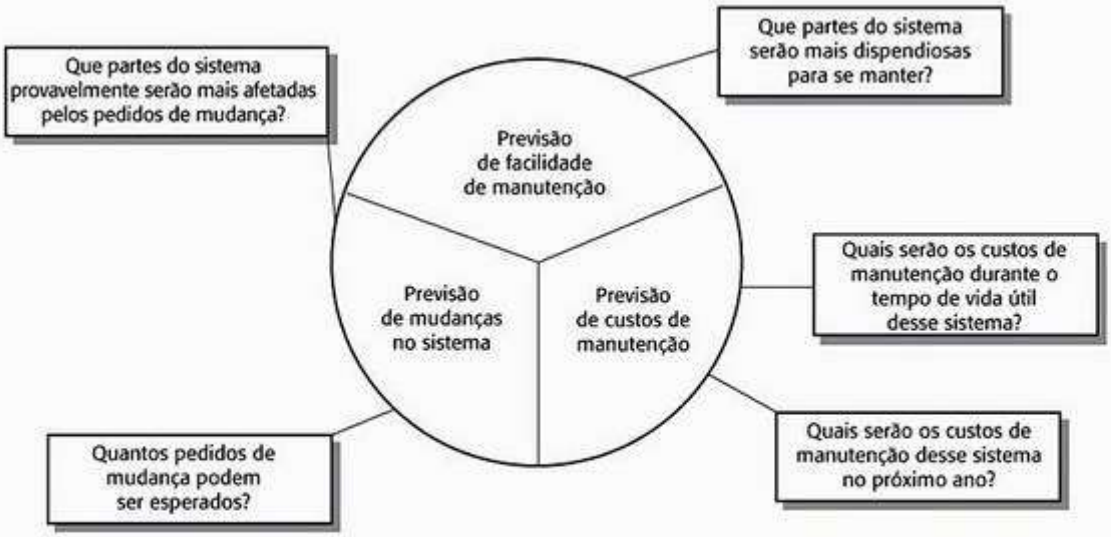
Essas diferentes previsões estão estreitamente relacionadas:

1. Se uma mudança no sistema deve ser aceita ou não, isso depende, até certo ponto, da facilidade de manutenção dos componentes de sistema afetados por essa mudança.
2. A implementação de mudanças no sistema tende a degradar a sua estrutura e, conseqüentemente, a reduzir sua facilidade de manutenção.
3. Os custos de manutenção dependem da quantidade de mudanças e os custos de implementação das mudanças dependem da facilidade de manutenção dos componentes do sistema.

A previsão do número de pedidos de alteração para um sistema requer uma compreensão da relação entre o sistema e seu ambiente externo. Alguns sistemas têm uma relação muito complexa com seu ambiente externo, e as mudanças nesse ambiente, inevitavelmente, resultam em mudanças no sistema. Para fazer um julgamento da relação entre um sistema e seu ambiente, deve-se avaliar:

Figura 27.8 ▼
Previsão da manutenção.

1. *O número e a complexidade das interfaces de sistema* Quanto maior o número de interfaces e quanto mais complexas elas forem, mais provável será que pedidos de modificações sejam efetuados.



2. *O número de requisitos de sistema inerentemente voláteis* Como foi discutido no Capítulo 6, os requisitos que refletem políticas e procedimentos organizacionais, provavelmente, serão mais voláteis do que os requisitos que se baseiam em características de domínio estáveis.

3. *Os processos de negócios em que o sistema é utilizado* À medida que os processos de negócios evoluem, eles geram pedidos de modificações no sistema. Quanto mais processos corporativos utilizarem um sistema, mais pedidos de mudança no sistema ocorrerão.

Para prever a facilidade de manutenção do sistema, é necessário compreender o número e os tipos de relações entre os diferentes componentes do sistema e também a inerente complexidade desses componentes. Foram realizados vários estudos dos diferentes tipos de complexidade em um sistema (McCabe, 1976; Halstead, 1977) e das relações entre a complexidade e a facilidade de manutenção (Kafura e Reddy, 1987; Banker *et al.*, 1993). Não é surpresa que esses estudos tenham constatado que, quanto mais complexo for um sistema ou componente, mais dispendiosa será sua manutenção.

Banker *et al.* (1993) realizaram um estudo utilizando uma série de programas comerciais escritos em COBOL. Eles avaliaram esses programas utilizando várias medidas complexas, incluindo o tamanho do procedimento, o tamanho do módulo e a densidade de ramificação, que é uma medida da complexidade de controle. Comparando a complexidade de diferentes partes do programa com os registros de manutenção real do programa, eles constataram que o uso de boas práticas de programação para reduzir a complexidade do sistema fazia reduzir os custos de manutenção.

Foi constatado que as medições de complexidade são particularmente úteis na identificação de componentes individuais de programa que devem ser de manutenção dispendiosa. No estudo de Kafura e Reddy (1987), eles examinaram uma série de componentes de sistema e constataram que o esforço de manutenção tendia a se concentrar em um pequeno número de componentes complexos. As implicações disso são que pode ser eficaz em termos de custo substituir componentes de sistema particularmente complexos por alternativas mais simples.

Depois que um sistema foi colocado em operação, é possível utilizar os dados de processo para ajudar a prever a facilidade de manutenção. Exemplos de métricas de processo que podem ser úteis para avaliar a facilidade de manutenção são:

1. *Número de pedidos de manutenção corretiva* Se o número de relatórios de falhas estiver aumentando, isso pode indicar que mais erros estão sendo introduzidos no programa do que os que estão sendo reparados durante o processo de manutenção. Isso pode indicar um declínio da facilidade de manutenção.

2. *Tempo médio requerido para a análise de impacto* Esse aspecto reflete o número de componentes de programa afetados pelo pedido de modificação. Se esse tempo aumentar, isso implica que cada vez mais componentes são afetados e que a facilidade de manutenção está diminuindo.

3. *Tempo médio gasto para implementar um pedido de modificação* Esse não é o mesmo que o tempo para a análise de impacto, embora possa se correlacionar com ele. As atividades envolvidas são proceder às mudanças no sistema e em sua documentação, em vez de simplesmente avaliar quais componentes são afetados. Esse tempo de modificação depende da dificuldade de programar, de modo que os requisitos não funcionais, como o desempenho, sejam cumpridos. Se aumentar o tempo para uma mudança, isso poderá indicar um declínio da facilidade de manutenção.

4. *Número de pedidos de modificação importantes* Se esse número aumentar com o tempo, isso poderá implicar declínio da facilidade de manutenção.

São utilizadas as informações previstas sobre os pedidos de modificação e as previsões sobre a facilidade de manutenção para fazer previsões dos custos de manutenção. A maioria dos gerentes combina essas informações com a intuição e a experiência, a fim de fazer

uma estimativa de custos. O modelo COCOMO 2 de estimativa de custos (Boehm *et al.*, 1995) sugere que uma estimativa do esforço de manutenção de software pode se basear no esforço para compreender o código existente e no esforço para desenvolver o novo código. Os leitores devem examinar, no artigo de Boehm, os detalhes dessa técnica de estimativa.

27.3 Evolução de arquitetura

Durante a manutenção do sistema, a maioria das mudanças individuais realizadas é localizada, e essas mudanças não afetam a arquitetura do sistema. Contudo, desde a década de 80, a economia de sistemas baseados em computadores mudou radicalmente, de maneira que os sistemas distribuídos, em vez de sistemas centralizados, são com freqüência a solução mais eficaz, em termos de custo, para os problemas de negócios. Portanto, muitas empresas enfrentam a necessidade de modificar seu sistema centralizado de mainframe para sistemas distribuídos cliente-servidor, como foi discutido no Capítulo 11.

Existe uma série de diferentes fatores que contribuem para essa mudança:

1. *Custos de hardware* Os custos de aquisição e manutenção de um sistema distribuído cliente-servidor são, geralmente, muito menores do que os custos de compra de um computador de mainframe com capacidade equivalente.
2. *Expectativas quanto à interface com o usuário* Muitos sistemas legados de mainframes fornecem interfaces baseadas em formulários. Contudo, a maioria dos usuários prefere interfaces gráficas com o usuário e interações mais fáceis com o sistema. Essas interfaces requerem muito mais computação local e somente podem ser fornecidas efetivamente em um sistema cliente-servidor.
3. *Acesso distribuído aos sistemas* As empresas estão, cada vez mais, distribuindo fisicamente sua organização, em vez de manter todos os recursos em um único local. Seus sistemas de computadores podem ser acessados a partir de diferentes localizações e por diferentes tipos de equipamentos. Os clientes e o pessoal podem acessar sistemas de sua casa, e isso tem de ser possibilitado.

Migrando para uma arquitetura distribuída, as organizações podem reduzir drasticamente os custos de hardware, desenvolver um sistema que tenha uma interface mais eficaz e um *look and feel* mais realista e apoiar os trabalhos distribuídos. No processo de migração, haverá, inevitavelmente, alguma conversão de sistema para um modelo orientado a objetos, e isso deve reduzir os custos da futura manutenção do sistema.

Contudo, modificar a arquitetura de um sistema legado é um desafio excepcional e um processo muito dispendioso. Os principais fatores que influenciam nessa decisão são mostrados na Figura 27.9. Antes de iniciar a migração da arquitetura, as organizações devem fazer uma cuidadosa avaliação de seus sistemas legados, para garantir que obterão um valor real nos negócios com a transformação da arquitetura.

Uma dificuldade fundamental de migrar muitos sistemas legados centralizados para uma arquitetura distribuída é que os sistemas não são estruturados de maneira que os componentes básicos de arquitetura possam ser identificados e separados de outros componentes. Como ideal, gostaríamos que os sistemas legados tivessem uma estrutura como a mostrada no diagrama à esquerda da Figura 27.10. Nesse caso, a interface com o usuário, os serviços fornecidos pelo sistema e pelo banco de dados estão claramente separados. Os serviços individuais estão bem definidos. Dentro da camada de serviço, é possível distinguir entre os diferentes serviços. Com esse tipo de estrutura, os elementos distribuídos podem ser identificados no sistema e podem ser reescritos para serem executados em computadores cliente.

Figura 27.9 ▶

Fatores que influenciam as decisões de distribuição de sistema.

Fator	Descrição
Importância para os negócios	Os retornos de investimento na distribuição de um sistema legado dependem de sua importância para os negócios e por quanto tempo ele continuará sendo importante. Se a distribuição fornecer apoio mais eficiente para os processos de negócios estáveis, então será mais provável que essa seja uma estratégia de evolução eficaz em termos de custos.
Idade do sistema	Quanto mais antigo for o sistema, mais difícil será modificar sua arquitetura, porque as modificações anteriores terão degradado a estrutura do sistema.
Estrutura do sistema	Quanto mais modular for o sistema, mais fácil será modificar sua arquitetura. Se a lógica da aplicação, o gerenciamento de dados e a interface com o usuário do sistema estiverem estreitamente acoplados, será difícil separar as funções para realizar a migração.
Políticas de suprimento de hardware	A distribuição da aplicação pode ser necessária, se houver uma política da empresa de substituir computadores mainframe dispendiosos por servidores mais baratos.

Na prática, a maioria dos sistemas legados é mais como o lado direito da Figura 27.10, em que os recursos da interface com o usuário, os serviços e o acesso aos dados estão misturados. Os serviços podem se sobrepor. Diferentes partes do serviço são implementadas em diferentes componentes do sistema. A interface com o usuário e o código do serviço estão integrados nos mesmos componentes, e pode não haver uma nítida distinção entre os serviços e o banco de dados do sistema. Nesses casos, pode não ser possível identificar as partes do sistema que podem ser distribuídas.

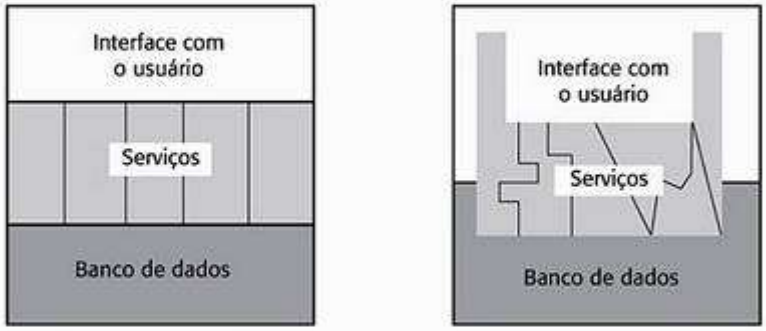
Em situações em que seja impraticável separar o sistema legado em componentes distribuíveis e implementar esses componentes em um sistema distribuído, uma abordagem alternativa pode ser utilizada. O sistema legado pode ser congelado e o sistema completo, empacotado (encapsulado) como um servidor. A interface com o usuário é implementada novamente no cliente e *middlewares* de propósito especial traduzem os pedidos a partir do cliente, para as interações com o sistema legado inalterado. Essa situação está ilustrada na Figura 27.11.

Embora a interface com o usuário e os serviços fornecidos por um sistema legado sejam geralmente integrados, quando se planeja uma estratégia de distribuição, pode ser útil considerá-los organizados em uma série de camadas lógicas (Figura 27.12). As camadas nesse diagrama representam candidatos em potencial para a distribuição.

1. A camada de apresentação se ocupa com o display e a organização das telas apresentadas aos usuários finais do sistema.

Figura 27.10 ▶

Estruturas de sistemas legados ideais e realistas.

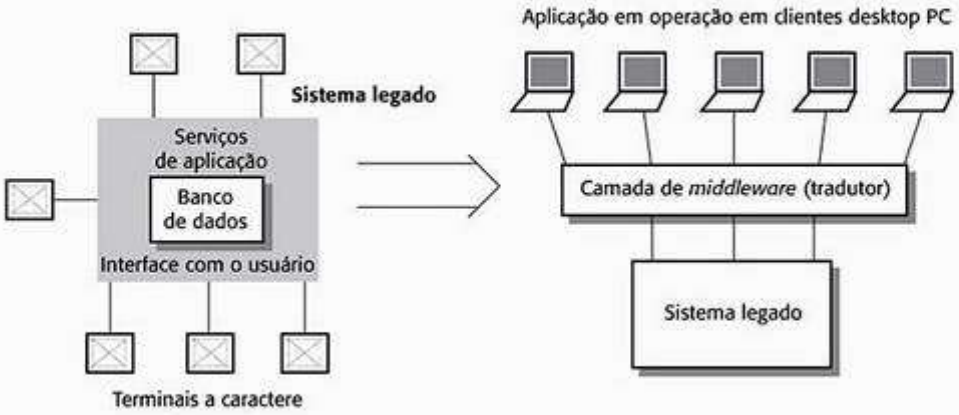


Modelo ideal para distribuição

Sistemas legados reais

Figura 27.11

Distribuição do sistema legado.



2. A camada de validação de dados se ocupa de verificar a entrada de dados feita pelo usuário final e a saída para o usuário final.
3. A camada de controle de interação se ocupa de gerenciar a seqüência de operações do usuário final e a seqüência de telas apresentadas ao usuário.
4. A camada dos serviços de aplicação se ocupa de fornecer os cálculos básicos oferecidos pela aplicação.
5. A camada de banco de dados fornece o gerenciamento e o armazenamento de dados da aplicação.

É impraticável distribuir o banco de dados para a maioria dos sistemas legados, mas existem várias opções alternativas de distribuição, como mostra a Figura 27.13. Na opção mais simples, o computador cliente se ocupa somente com a apresentação da interface com o usuário e todas as outras funções são mantidas no servidor. Na opção de distribuição mais radical, o servidor somente gerencia os dados do sistema, e todas as outras funções são distribuídas para o cliente. Naturalmente, essas opções não são exclusivas. É possível decidir começar com a distribuição da apresentação e distribuir as outras camadas lógicas quando tempo e recursos estiverem disponíveis. Além disso, como foi discutido no Capítulo 11, outras opções de distribuição que envolvem servidores de múltiplas camadas também são possíveis.

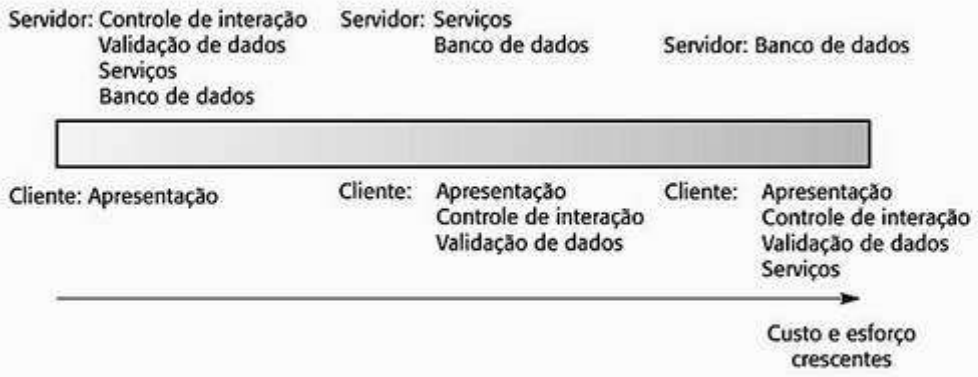
Quando um sistema legado é encapsulado e acessado por meio de uma camada de *middleware*, como na Figura 27.11, é possível implementar uma estratégia de distribuição que comece à esquerda da Figura 27.13 e, com o tempo, se mover cada vez mais para a direita. À medida que novos serviços são implementados, estes assumem as funções do sistema legado no servidor, transferindo, assim, cada vez mais serviços para o cliente. Essa distribuição gradual da funcionalidade pode significar que a maior parte do sistema legado inicial se torne inútil, e ele atue somente como um servidor de banco de dados para o sistema de distribuição.

Figura 27.12

Modelo de distribuição em camadas.

Figura 27.13

Espectro das opções de distribuição.



Uma vez atingido esse estágio, é necessário decidir se vale a pena manter o sistema legado ou se ele deve ser substituído por um sistema de gerenciamento de banco de dados. Entre os fatores a serem considerados estão o hardware de sistema, a capacitação do pessoal disponível, se um sistema de gerenciamento de banco de dados já está ou não em uso, se ele pode lidar com a quantidade de dados a serem gerenciados e os custos de reengenharia de dados.

27.3.1 Distribuição da interface com o usuário

Muitos sistemas legados foram projetados antes que interfaces gráficas com o usuário estivessem disponíveis. Eles utilizam interfaces com base em formulários, que são executadas em terminais especializados, que podem apenas exibir caracteres. Esses terminais têm limitado poder de processamento e de características de display, de modo que todas as funções de display e de computação associadas são manipuladas por um sistema central de mainframe. Mesmo quando esses terminais forem substituídos por PCs, a interface baseada em caractere pode ainda ser mantida por meio de um programa de emulação de terminal executado no PC.

A distribuição da interface com o usuário tira proveito do poder local de processamento disponível em desktop PCs, a fim de fornecer uma interface gráfica mais amigável aos usuários do sistema. Os serviços de interface com o usuário (apresentação, controle de interação e validação) são movidos para a máquina desktop, e a interface de formulários baseada em caracteres é substituída por uma GUI, em que o usuário pode apontar para menus, campos etc. Os dados que são processados, assim como a maior parte ou todos os serviços da aplicação, permanecem no sistema servidor.

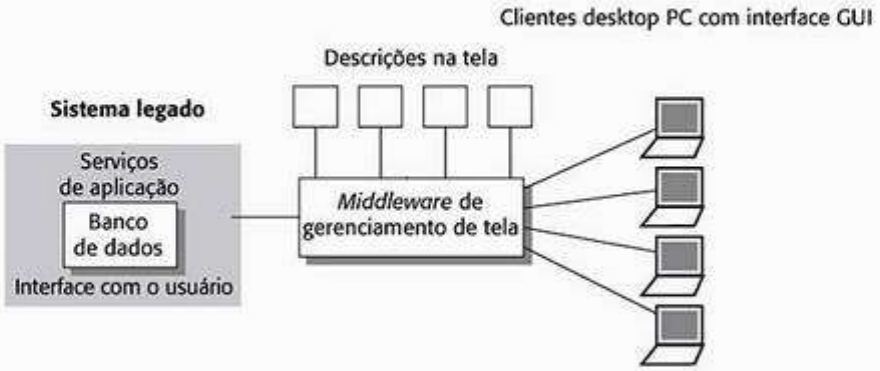
Se o sistema legado for estruturado de maneira que os serviços de interface com o usuário estejam claramente identificados, então o sistema legado poderá ser modificado para implementar a distribuição da interface com o usuário. Para essa distribuição, essas partes do sistema que lidam com a interação com o usuário são reimplementadas no computador cliente e se comunicam com a aplicação por intermédio da mesma interface que a UI (*user interface* – interface com o usuário) baseada em caracteres.

Em muitos casos, contudo, o código da interface com o usuário e a lógica de aplicação são estreitamente integrados, e é muito difícil separar o código da interface com usuário de outros códigos no sistema legado. Nesse caso, a interface distribuída com usuário pode ser implementada como ilustra a Figura 27.14.

O *middleware* de gerenciamento de tela, na Figura 27.14, se comunica com a aplicação e se comporta exatamente da mesma maneira que um terminal de usuário. O sistema de gerenciamento de tela utiliza uma descrição de cada tela para interpretar os dados no display. Em seguida, ele envia essa descrição para o cliente, onde o software de interface com o usuário a apresenta dentro de uma interface gráfica. Atualmente, esse recurso pode ser implemen-

Figura 27.14 ▶

Distribuição de interface do usuário.



tado com alguma facilidade, utilizando-se XML (St. Laurent e Cerami, 1999) para descrever a estrutura da interface. Não há, por conseguinte, nenhuma necessidade de modificar o sistema legado em si. Basta escrever o *middleware* de gerenciamento de tela e o software da interface com o usuário para o computador cliente.

Existem duas estratégias de implementação para a distribuição da interface com o usuário:

1. Implementar a interface, utilizando o sistema de gerenciamento de janelas, que é nativo no PC cliente, e implementar comunicações com o servidor.
2. Implementar a interface com o usuário, utilizando um Web browser.

No primeiro caso, a interface com o usuário é programada em uma linguagem convencional de programação, como Java, ou em uma linguagem de *scripting*, como Visual Basic. As chamadas são feitas para as funções no sistema operacional do cliente, a fim de implementar a interface com o usuário. No segundo caso, a implementação envolve utilizar os recursos em HTML e em Web browsers, a fim de construir uma interface com o usuário baseada em páginas Web. As computações que são requeridas podem ser implementadas no cliente, utilizando-se Java, ou no servidor Web, utilizando-se uma interface do tipo CGI (*common gateway interface* – interface de gateway comum) ou servlets. Cada uma dessas abordagens tem diferentes vantagens e desvantagens, como descrito na Figura 27.15.

Converter interfaces para interfaces baseadas na Web é uma opção atrativa, devido à independência da plataforma e da generalizada disponibilidade de Web browsers. Applets Java podem ser utilizados para fornecer computação local do lado do cliente, e assim essas interfaces podem ser comparáveis a interfaces desenvolvidas com a utilização do sistema de gerenciamento de janela. Contudo, alguns usuários podem ainda utilizar versões de browsers que são incapazes de executar Java ou as últimas versões de HTML, ou que não estão adequa-

Figura 27.15 ▶

Vantagens e desvantagens das estratégias de migração UI.

Estratégia	Vantagens	Desvantagens
Implementação com a utilização do sistema de gerenciamento de janelas	Acesso a todas as funções de UI; assim, não há nenhuma restrição ao projeto de UI. Melhor desempenho de UI.	Dependente de plataforma. Pode ser mais difícil atingir a consistência da interface.
Implementação com a utilização de um Web browser	Custos de treinamento mais baixos, devido à familiaridade do usuário com a Web. Mais fácil de atingir a consistência da interface.	Desempenho de UI potencialmente menor. O projeto de interface fica limitado pelos recursos fornecidos por Web browsers.

damente configuradas. Os projetistas de interfaces, muitas vezes, precisam elaborar o projeto para esses sistemas de capacitação inferior, e isso restringe a funcionalidade das interfaces baseadas na Web.

Uma dificuldade básica que surge ao converter as interfaces baseadas em formulários para interfaces gráficas com o usuário vem das diferentes abordagens que cada uma delas considera para o controle de interação e validação. Em um sistema baseado em formulários, o sistema computacional controla a interação e aplica as regras de validação de dados assim que eles são fornecidos. A maioria das aplicações requer que os campos de formulário sejam completados em uma seqüência específica e exibem os formulários somente quando outros formulários são preenchidos completa e corretamente.

Em uma GUI, o usuário está no controle e utiliza um modelo de interação em que ele aponta e clica para selecionar os campos. Não é natural que esse pedido seja controlado pela máquina. Se o controle for implementado, isso poderá resultar em tráfego de rede adicional entre o cliente e o servidor. A validação de dados somente pode ser possível depois que um formulário tiver sido inteiramente preenchido, ou o sistema pode se tornar lento, em virtude das freqüentes comunicações entre o cliente e o servidor.

PONTOS-CHAVE

- Entre as estratégias de modificação em software estão a manutenção de software, a evolução da arquitetura e a reengenharia de software.
- Parece haver uma série de relacionamentos invariantes (leis de Lehman) que afetam a evolução de um sistema de software. Eles foram derivados de observações empíricas e mostram que os custos de manutenção são inevitáveis. Eles fornecem diretrizes sobre como gerenciar o processo de manutenção.
- Existem três tipos principais de manutenção de software, que são a manutenção para reparar os defeitos no software, a manutenção para adaptar o software a um ambiente operacional diferente e a manutenção para adicionar ou modificar a funcionalidade do sistema.
- O custo da mudança de software geralmente excede os custos de desenvolvimento de software. Uma vez que as empresas gerenciam um número cada vez maior de sistemas legados, a maior parte de seu orçamento de software é absorvida na manutenção de sistemas legados.
- Os altos custos de manutenção são devidos à falta de estabilidade de pessoal, aos contratos de desenvolvimento que não encorajam a produção de códigos de fácil manutenção, à falta da capacidade requerida para manter um sistema e a uma estrutura de sistema que se degradou, devido à idade e às regulares modificações no sistema.
- A evolução da arquitetura envolve modificar a arquitetura de um sistema, a partir de uma arquitetura centralizada, centrada em dados, para uma arquitetura distribuída. Tanto a interface com o usuário quanto a funcionalidade do sistema podem ser distribuídas.
- Uma estratégia comum da evolução de arquitetura para sistemas legados é encapsular o sistema legado como um servidor e implementar uma interface com o usuário distribuída, que acessa a funcionalidade do sistema por meio de um *middleware* de propósito especial.

LEITURAS SUGERIDAS

The renaissance of legacy systems. Esse livro apresenta um capítulo que discute várias estratégias para migrar sistemas centralizados para sistemas cliente–servidor. (I. Warren, 1998, Springer.)

Software evolution and reuse. Discute uma série de estudos de caso industriais em que foram aplicadas técnicas de pesquisa (S. Hallsteinsen e M. Paci, 1997, Springer.)

EXERCÍCIOS

27.1 Explique por que um sistema de software que é implementado em uma organização em mudança é sempre sujeito a pedidos de alterações. Por que essas mudanças se tornam cada vez mais dispendiosas?

27.2 Explique a lógica inserida nas leis de Lehman. Sob quais condições essas leis podem falhar?

27.3 Explique as dificuldades de medir a facilidade de manutenção de programas. Sugira por que você deve utilizar várias métricas diferentes de complexidade ao tentar avaliar a facilidade de manutenção de um programa.

27.4 Como gerente de projeto de software, em uma empresa que se especializa no desenvolvimento de software para a indústria de óleo *offshore*, você recebeu a tarefa de descobrir os fatores que afetam a facilidade de manutenção de determinados sistemas desenvolvidos por sua empresa. Sugira como você pode estabelecer um programa para analisar o processo de manutenção para descobrir as métricas apropriadas de facilidade de manutenção para sua empresa.

27.5 Explique por que o encapsulamento de um sistema legado em mainframe e seu uso como um servidor deve ser considerado somente uma solução de curto prazo para os problemas de evolução de arquitetura.

27.6 Discuta as vantagens e as desvantagens de distribuir cada uma das camadas na Figura 27.12.

27.7 Dois grandes bancos internacionais, com diferentes bancos de dados de clientes, se unem e decidem que precisam ter acesso a todas as informações dos clientes, a partir de todas as agências. Dando razões para sua resposta, sugira a estratégia mais apropriada para possibilitar o acesso a esses sistemas e discuta brevemente como a solução pode ser implementada.

27.8 Os engenheiros de software têm responsabilidade profissional de produzir código de fácil manutenção, mesmo que isso não seja explicitamente solicitado por seus empregadores?

28 Reengenharia de software

Objetivos

O objetivo deste capítulo é explicar o processo de reengenharia de software, a fim de melhorar a facilidade de manutenção de um sistema de software. Depois de ler este capítulo, você poderá:

- compreender por que a reengenharia é, algumas vezes, uma opção eficaz, em termos de custos, para a evolução do sistema de software;
- compreender as atividades, como engenharia reversa e reestruturação de programa, que podem estar envolvidas no processo de reengenharia de software, e
- compreender as diferenças entre reengenharia de software e de dados e por que a reengenharia é um processo dispendioso e demorado.

Sumário

28.1 Tradução de código-fonte
28.2 Engenharia reversa
28.3 Melhoria de estrutura de programa
28.4 Modularização de programa
28.5 Reengenharia de dados

Nos capítulos 26 e 27, foram apresentados os sistemas legados e diferentes estratégias para a evolução de software. Os sistemas legados são antigos sistemas de software que são essenciais para o apoio ao processo de negócios. As empresas dependem desses sistemas e, assim, precisam mantê-los em operação. Entre as estratégias de evolução de software estão a manutenção, a substituição, a evolução de arquitetura e o tópico deste capítulo: reengenharia de software.

A reengenharia de software se ocupa de reimplementar sistemas legados, para que sua manutenção seja mais fácil. A reengenharia pode envolver redocumentar, organizar e reestruturar o sistema, traduzir o sistema para uma linguagem de programação mais moderna e modificar e atualizar a estrutura e os valores dos dados do sistema. A funcionalidade do software não é modificada e, normalmente, a arquitetura do sistema também permanece a mesma.

A partir de uma perspectiva técnica, a reengenharia de software pode parecer uma solução de segunda classe para o problema de evolução de sistema. A arquitetura de software não é atualizada; assim, distribuir sistemas centralizados é difícil. Em geral, não é possível mudar radicalmente a linguagem de programação do sistema, de modo que os antigos sistemas não podem ser convertidos para linguagens de programação orientadas a objetos, como Java ou C++. As limitações inerentes ao sistema são mantidas porque a funcionalidade do software permanece inalterada.

Contudo, do ponto de vista dos negócios, a reengenharia de software pode ser a única maneira viável de assegurar que os sistemas legados podem continuar em serviço. Pode ser muito dispendioso e muito arriscado adotar qualquer outra abordagem para a evolução do sistema. Para compreender as razões dessa situação, é preciso fazer uma primeira avaliação do problema do sistema legado.

A quantidade de código em sistemas legados é imensa. Em 1990, estimava-se (Ulrich, 1990) que havia 120 *bilhões* de linhas de código-fonte. A maioria desses sistemas tinha sido escrita em COBOL, uma linguagem de programação mais adequada para o processamento de dados de negócios, ou FORTRAN. FORTRAN é uma linguagem para programação científica ou matemática. Essas linguagens têm recursos de estruturação de programa limitados e, no caso do FORTRAN, um apoio muito limitado à estruturação de dados.

Embora muitos desses programas atualmente tenham sido substituídos, é provável que a maioria deles ainda esteja em operação. Ao mesmo tempo, desde 1990, tem havido um imenso aumento no uso de computadores para apoio ao processo de negócios. Portanto, suponho que atualmente devam existir cerca de 250 bilhões de linhas de código que estão sujeitas à manutenção. A maioria não está escrita em linguagens orientadas a objetos, e boa parte ainda é executada em computadores do tipo mainframe.

Existem tantos sistemas, que a completa substituição ou a reestruturação radical é financeiramente impensável para a maioria das organizações. A manutenção de velhos sistemas é cada vez mais dispendiosa, e a reengenharia desses sistemas prolonga seu tempo de vida útil. Como foi discutido no Capítulo 26, a reengenharia de um sistema é eficaz em termos de custo quando ele tem alto valor de negócios, mas é dispendioso para manter. A reengenharia melhora a estrutura do sistema, cria um nova documentação relacionada e faz com que ela seja de mais fácil compreensão.

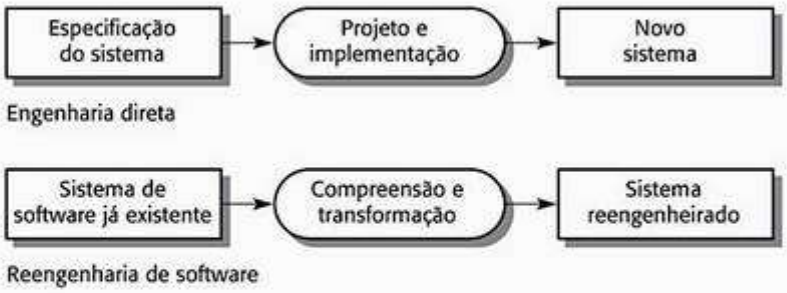
A reengenharia de software tem duas vantagens principais em relação a abordagens mais radicais para a evolução de sistemas:

1. *Riscos reduzidos* Existe um alto risco em redesenvolver um software que seja essencial para uma organização. Podem ser cometidos erros na especificação do sistema, ocorrer problemas de desenvolvimento etc.

2. *Custos reduzidos* O custo da reengenharia é significativamente menor do que os custos de desenvolvimento de um novo software. Ulrich (1990) cita o exemplo de um sistema comercial em que os custos de reimplementação foram estimados em 50 milhões de dólares. O sistema foi reengenheirado com sucesso por 12 milhões. Se considerarmos esses números como sendo típicos, o custo da reengenharia é quatro vezes menor do que o de reescrever um sistema.

Figura 28.1

Engenharia direta e reengenharia.



O termo reengenharia também está associado com a reengenharia de processo de negócios (Hammer, 1990). A reengenharia de processo de negócios se ocupa de reprojetar processos de negócios, a fim de reduzir o número de atividades redundantes e melhorar a eficiência do processo. Em geral, ela depende da introdução ou do aumento do apoio baseado em computadores para o processo. Muitas vezes, a reengenharia de processo é um fator direcionador da evolução de software, uma vez que os sistemas legados podem incorporar dependências implícitas dos processos existentes. Essas dependências devem ser descobertas e removidas, antes que a reengenharia de processo se torne possível. Portanto, a necessidade da reengenharia de software pode surgir em uma empresa quando se torna claro que a escala das mudanças requeridas pela reengenharia de processo de negócios não pode ser acomodada por meio da manutenção normal do programa.

A principal distinção entre a reengenharia e o novo desenvolvimento de software é o ponto de partida para o desenvolvimento. Em vez de iniciar com uma especificação escrita, o sistema antigo atua como uma especificação para o novo sistema. Chikofsky e Cross (1990) chamam o desenvolvimento convencional de *engenharia direta*, para distingui-la da reengenharia de software. A diferença está ilustrada na Figura 28.1. A engenharia direta inicia com uma especificação do sistema e envolve o projeto e a implementação de um novo sistema. A reengenharia começa com um sistema já existente, e o processo de desenvolvimento para a substituição tem como base compreender e transformar o sistema original.

A Figura 28.2 ilustra um possível processo de reengenharia. A entrada para o processo é um programa legado e a saída é uma versão estruturada e modularizada do mesmo programa. Ao mesmo tempo que ocorre a reengenharia do programa, os dados do sistema também podem passar por uma reengenharia. As atividades nesse processo de reengenharia são:

1. *Tradução de código-fonte* O programa é convertido de uma linguagem de programação antiga para uma versão mais moderna da mesma linguagem ou para uma linguagem diferente.
2. *Engenharia reversa* O programa é analisado e as informações são extraídas dele, a fim de ajudar a documentar sua organização e funcionalidade.
3. *Melhoria de estrutura do programa* A estrutura de controle do programa é analisada e modificada, a fim de torná-la mais fácil de ser lida e compreendida.
4. *Modularização de programa* As partes relacionadas do programa são agrupadas e, quando for apropriado, a redundância é removida. Em alguns casos, esse estágio pode envolver a transformação de arquitetura, como foi discutido no Capítulo 27.
5. *Reengenharia de dados* Os dados processados pelo programa são modificados, a fim de refletir as mudanças feitas nele.

A reengenharia de programa pode não exigir necessariamente todas as etapas mostradas na Figura 28.2. A tradução de código-fonte poderá não ser necessária, se a linguagem de programação utilizada para desenvolver o sistema ainda for aceita pelo fornecedor do compilador. Se a reengenharia depende completamente de ferramentas automatizadas, então a recuperação de

Figura 28.2 ▲

Processo de reengenharia.

documentação por meio da engenharia reversa pode ser desnecessária. A reengenharia de dados é necessária somente se as estruturas de dados do programa forem modificadas durante a reengenharia do sistema. Contudo, a reengenharia de software sempre envolve alguma reestruturação de programa.

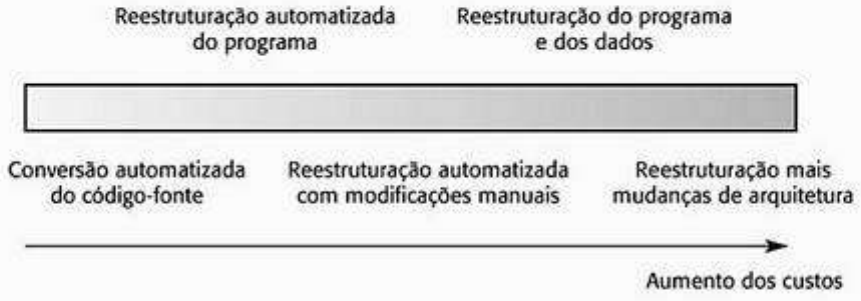
Os custos da reengenharia obviamente dependem da extensão do trabalho que é realizado. Existe uma grande variedade de abordagens possíveis para a reengenharia, como mostra a Figura 28.3. Os custos aumentam da esquerda para a direita, de modo que a tradução de código-fonte é a opção mais barata e a reengenharia como parte da migração de arquitetura é a mais dispendiosa.

Independentemente da extensão da reengenharia, os principais fatores que afetam seus custos são:

1. *A qualidade do software que deve passar pela reengenharia* Quanto menor a qualidade do software e de sua documentação associada (se existir), maiores serão os custos de reengenharia.
2. *O apoio às ferramentas disponíveis para a reengenharia* Normalmente, fazer a reengenharia de um sistema de software não é um processo eficaz em termos de custo, a menos que possam ser utilizadas as ferramentas CASE para automatizar a maior parte das modificações no programa.
3. *A extensão da conversão de dados requerida* Se a reengenharia exigir que grandes quantidades de dados sejam convertidas, isso aumentará significativamente os custos do processo.
4. *A disponibilidade de pessoal habilitado* Se o pessoal responsável pela manutenção do sistema não estiver envolvido no processo de reengenharia, isso aumentará seus custos. Os engenheiros responsáveis pela reengenharia de sistemas terão de gastar bastante tempo para compreender o sistema.

Figura 28.3 ▶

Abordagens da reengenharia.



A principal desvantagem da reengenharia de software é que há limites práticos referentes a quanto um sistema pode ser melhorado por ela. Não é possível, por exemplo, converter um sistema escrito, utilizando uma abordagem funcional para um sistema orientado a objetos. As grandes mudanças de arquitetura ou a reorganização radical do gerenciamento de dados do sistema não podem ser realizadas automaticamente e, assim, envolvem altos custos adicionais. Embora a reengenharia possa melhorar a facilidade de manutenção, o sistema resultante do processo de reengenharia, provavelmente, não terá uma manutenção tão fácil quanto um sistema novo, que foi desenvolvido com a utilização de métodos modernos de engenharia de software.

28.1 Tradução de código-fonte

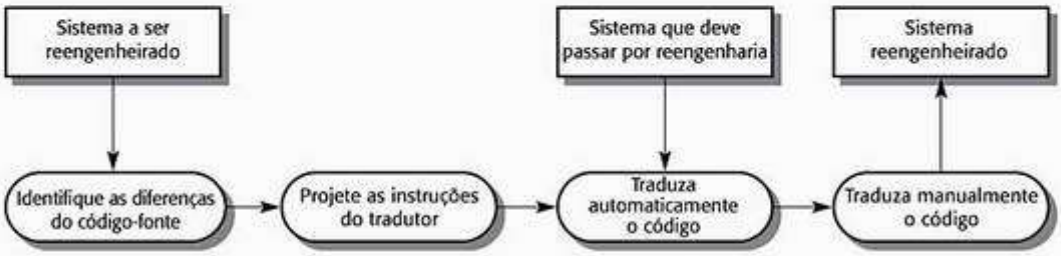
O meio mais simples de proceder à reengenharia de software é a tradução do programa, quando o código-fonte em uma linguagem de programação é automaticamente traduzido para o código-fonte em uma outra linguagem. A estrutura e a organização do programa em si permanecem inalteradas. A linguagem-alvo pode ser uma versão atualizada da linguagem original (por exemplo, de COBOL-74 para COBOL-85) ou pode ser uma tradução para uma linguagem completamente diferente (por exemplo, de FORTRAN para C).

A tradução no nível da linguagem original pode ser necessária pelas seguintes razões:

1. *Atualização da plataforma de hardware* A organização pode desejar mudar sua plataforma-padrão de hardware. É possível que não haja compiladores para a linguagem original disponíveis no novo hardware.
2. *Escassez de pessoal habilitado* Pode ser difícil encontrar pessoal treinado na linguagem original. Esse é um problema específico, quando os programas foram escritos em uma linguagem que não é padrão e que atualmente não é mais de uso geral.
3. *Mudanças na política organizacional* Uma organização pode decidir padronizar o uso de uma determinada linguagem, a fim de minimizar seus custos de apoio ao software. Manter muitas versões de antigos compiladores pode ser muito dispendioso.
4. *Falta de suporte ao software* Os fornecedores do compilador de linguagem podem ter abandonado suas atividades ou podem interromper o fornecimento de suporte técnico para seu produto.

A Figura 28.4 ilustra o processo de tradução de código-fonte. Pode não haver a necessidade de compreender detalhadamente a operação do software ou de modificar a arquitetura do sistema. A análise envolvida pode focalizar as considerações relacionadas à linguagem de programação, como a equivalência de construções de controle de programa.

Figura 28.4 ▼
Processo de tradução de programa.



A tradução de código-fonte só é economicamente realista se um tradutor automatizado estiver disponível para fazer a maior parte da tradução. Esse pode ser um programa especialmente escrito, pode ser uma ferramenta comprada pronta para converter de uma linguagem para outra ou um sistema de combinação de padrões. Nesse último caso, deve ser escrito um conjunto de instruções sobre como fazer a tradução de uma representação para outra. Os padrões parametrizados na linguagem fonte são definidos e associados com os padrões equivalentes na linguagem-alvo.

Em muitos casos, a tradução completamente automatizada é impossível. As construções na linguagem fonte podem não ter nenhum equivalente direto na linguagem-alvo. Pode haver instruções condicionais de compilação inseridas no código-fonte que não sejam compatíveis com a linguagem-alvo. Nessas circunstâncias, é preciso fazer as modificações manualmente, a fim de ajustar e melhorar o sistema gerado.

28.2 Engenharia reversa

A engenharia reversa é o processo de analisar o software com o objetivo de recuperar seu projeto e sua especificação. O programa em si permanece inalterado no processo de engenharia reversa. O código-fonte do software geralmente está disponível para a entrada do processo de engenharia reversa. Algumas vezes, contudo, se até o código-fonte foi perdido, a engenharia reversa precisa começar com o código executável.

A engenharia reversa não é o mesmo que a reengenharia. O objetivo da engenharia reversa é derivar o projeto ou a especificação de um sistema a partir de seu código-fonte. O objetivo da reengenharia é produzir um sistema novo, com manutenção mais fácil. Naturalmente, como podemos ver na Figura 28.2, a engenharia reversa para desenvolver uma melhor compreensão de um sistema é com freqüência parte do processo de reengenharia.

A engenharia reversa é utilizada durante o processo de reengenharia de software, a fim de recuperar o projeto do programa que os engenheiros utilizam para ajudá-los a compreender um programa antes de reorganizar sua estrutura. Contudo, a engenharia reversa não precisa ser sempre seguida da reengenharia:

1. O projeto e a especificação de um sistema existente podem passar por engenharia reversa, de modo que possam servir como uma entrada à especificação de requisitos, para a substituição desse programa.
2. Como alternativa, o projeto e a especificação podem passar por engenharia reversa, de modo que estejam disponíveis para ajudar na manutenção do programa. Com essas informações adicionais, pode não ser necessário fazer a reengenharia do código-fonte do sistema.

O processo de engenharia reversa está ilustrado na Figura 28.5. O processo inicia com uma fase de análise, durante a qual o sistema é analisado, utilizando-se ferramentas automatizadas, a fim de descobrir sua estrutura. Isso por si só não é suficiente para recriar o projeto do sistema. Os engenheiros, então, trabalham com o código-fonte do sistema e seu modelo estrutural. Eles adicionam as informações obtidas mediante a compreensão do sistema. Essas informações são mantidas como um grafo direcionado, que é vinculado ao código-fonte do programa.

Os browsers de armazenamento de informações são utilizados para comparar a estrutura do grafo e o código e para anotar informações extras no grafo. Documentos de vários tipos, como diagramas de programa e de estrutura de dados e matrizes de rastreamento, podem ser gerados a partir do grafo direcionado. As matrizes de rastreamento mostram onde as entidades são definidas e referenciadas no sistema. O processo de geração de documentos é itera-

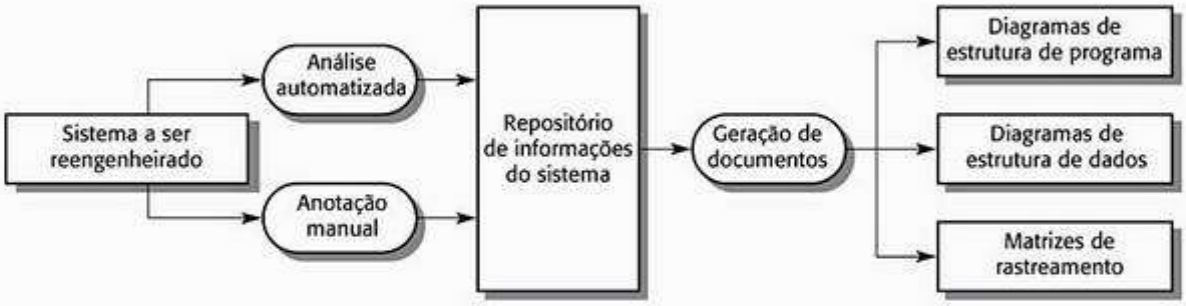


Figura 28.5 ▲
Processo de engenharia reversa.

tivo, uma vez que as informações de projeto são utilizadas para melhorar ainda mais as informações contidas no repositório do sistema.

As ferramentas para a compreensão do programa podem ser utilizadas para apoiar o processo de engenharia reversa. Geralmente, elas apresentam diferentes visões do sistema e possibilitam uma fácil navegação pelo código-fonte. Por exemplo, elas permitem aos usuários selecionar uma definição de dados e, então, prosseguir no código para onde o item de dados é utilizado. Exemplos desses browsers de programa são discutidos por Cleveland (1989), Oman e Cook (1990) e Ning *et al.* (1994).

Depois que a documentação de projeto do sistema foi gerada, outras informações podem ser adicionadas ao repositório de informações para ajudar a recriar a especificação do sistema. Isso, em geral, envolve mais anotações manuais da estrutura do sistema. A especificação não pode ser deduzida automaticamente a partir do modelo do sistema.

28.3 Melhoria de estrutura de programa

A necessidade de otimizar o uso de memória e a falta de entendimento da engenharia de software, por parte de muitos programadores, significa que muitos sistemas legados não são bem estruturados. Sua estrutura de controle é complexa, com muitas ramificações incondicionais, e a lógica de controle não é intuitiva. Essa estrutura pode também ter sido afetada por manutenções regulares. As mudanças no programa podem ter tornado alguns códigos inatingíveis, mas isso só pode ser descoberto depois de uma análise ampla. Os programadores de manutenção freqüentemente não ousam remover o código, no caso de ele poder ser acessado de maneira indireta.

A Figura 28.6 ilustra quanto a complexa lógica de controle pode dificultar o entendimento de um programa relativamente simples. O programa está escrito em uma notação similar ao FORTRAN, linguagem que foi freqüentemente utilizada para escrever esse tipo de programa. Contudo, não tornei o programa ainda mais difícil de ser compreendido utilizando nomes de variáveis secretas. O exemplo da Figura 28.6 é um controlador para um sistema de aquecimento. Um interruptor de painel pode ser colocado nas posições On, Off ou Controlled. Se o sistema estiver controlado, então ele será ligado e desligado, dependendo do ajuste de um timer e de um termostato. Se o aquecimento estiver ligado, Switch-heating o desligará e vice-versa.

Geralmente, os programas desenvolvem essa complexa estrutura lógica ao serem modificados durante a manutenção. Novas condições e ações associadas são acrescentadas, sem modificar a estrutura de controle existente. No curto prazo, essa é uma solução mais rápida e de menor risco, uma vez que reduz as chances de ser introduzidos defeitos no sistema. No longo prazo, contudo, isso resulta em código incompreensível. Complexas estruturas de código podem também surgir quando os programadores procuram evitar a duplicação de código. Algumas vezes, isso é necessário, quando os programas são restringidos por uma memória limitada.

Figura 28.6

Programa de controle com lógica 'espaguete'.

```
Start:   Get (Time-on, Time-off, Time, Setting, Temp, Switch)
         if Switch = off goto off
         if Switch = on goto on
         goto Cntrld
off:     if Heating-status = on goto Sw-off
         goto loop
on:      if Heating-status = off goto Sw-on
         goto loop
Cntrld:  if Time = Time-on goto on
         if Time = Time-off goto off
         if Time < Time-on goto Start
         if Time > Time-off goto Start
         if Temp > Setting then goto off
         if Temp < Setting then goto on
Sw-off:  Heating-status := off
         goto Switch
Sw-on:   Heating-status := on
Switch:  Switch-heating
loop:    goto Start
```

Figura 28.7

Programa estruturado de controle.

```
loop
  -- A declaração GET acha valores para as variáveis dadas a partir do ambiente do
sistema.
  Get (Time-on, Time-off, Time, Setting, Temp, Switch) ;
  case Switch of
      when On => if Heating-status = off then
                      Switch-heating ; Heating-status := on ;
                  end if ;
      when Off => if Heating-status = on then
                      Switch-heating ; Heating-status := off ;
                  end if ;
      when Controlled =>
          if Time >= Time-on and Time < = Time-off then
              if Temp > Setting and Heating-status = on then
                  Switch-heating; Heating-status = off ;
              elsif Temp < Setting and Heating-status = off then
                  Switch-heating; Heating-status := on ;
              end if ;
          end if ;
  end case ;
end loop ;
```

A Figura 28.7 mostra o mesmo sistema de controle, o qual escrevi utilizando declarações estruturadas de controle. O programa pode ser lido seqüencialmente, de cima para baixo, e assim é muito mais fácil de compreender. As três posições do interruptor, *on*, *off* e *controlled*, são claramente identificadas e vinculadas a seu código associado. Nesse caso, não utilizei Java, uma vez que o programa original não é orientado a objetos.

Assim como o controle não estruturado, condições complexas também podem ser simplificadas, como parte do processo de reestruturação de programa. A Figura 28.8 mostra como uma declaração condicional, incluindo a lógica 'not', pode se tornar mais compreensível.

Bohm e Jacopini (1966) provaram que qualquer programa poderia ser reescrito em termos de condicionais simples *if-then-else* e *loops while* e que declarações incondicionais *goto* não eram necessárias. Esse teorema é a base para a reestruturação automática do programa. A Figura 28.9 mostra os estágios da reestruturação automática de um programa. Primeiramente, o programa é convertido em um grafo direcionado, e então é gerado um programa estruturado equivalente, sem declarações 'goto'.

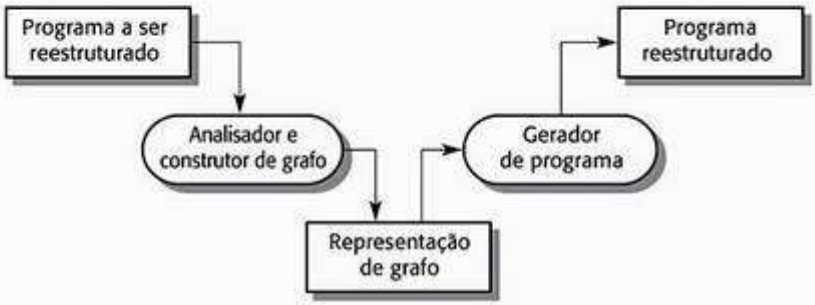
Figura 28.8

Simplificação de condição.

```
-- Condição complexa
if not (A > B and (C < D or not ( E > F) ) )...
-- Condição simplificada
if A <= B and (C>= D or E > F)...
```

Figura 28.9

Reestruturação automatizada de programa.



O grafo direcionado gerado é um grafo de fluxo de programa, que mostra como o controle se move ao longo do programa. Técnicas de simplificação e de transformação podem ser aplicadas a esse grafo, sem modificar sua semântica. Elas detectam e removem partes inatingíveis do código. Uma vez completada a simplificação, um novo programa é gerado. *Loops while* e declarações condicionais simples são substituídos por um controle com base em 'goto'. Esse programa pode estar na linguagem original ou em uma linguagem diferente (por exemplo, FORTRAN pode ser convertido em C).

Entre os problemas com a reestruturação automática de programa estão:

1. *Perda de comentários* Se o programa tiver comentários nas linhas, eles serão invariavelmente perdidos, como parte do processo de reestruturação.
2. *Perda de documentação* Do mesmo modo, a correspondência entre a documentação externa do programa e o programa também é perdida. Em muitos casos, no entanto, tanto os comentários quanto a documentação do programa estão desatualizados, e nesse caso esse não é um fator importante.
3. *Grandes exigências de computação* Os algoritmos embutidos em ferramentas de reestruturação são complexos. Mesmo com hardware moderno e rápido, pode levar um longo tempo para completar o processo de reestruturação de grandes programas.

Se o programa é dirigido por dados, com componentes estreitamente acoplados, por meio da estrutura de dados compartilhada, a reestruturação de código pode não levar a uma melhoria significativa na facilidade de compreensão. A modularização do programa, como aborda a seção seguinte, também pode ser necessária. Se o programa for escrito em um dialeto de linguagem que não seja um padrão, as ferramentas de reestruturação-padrão poderão não funcionar adequadamente e pode ser necessária uma significativa intervenção manual.

Em alguns casos, a opção de reestruturar todos os programas em um sistema pode não apresentar uma boa relação custo-benefício. Alguns programas podem ser de melhor qualidade do que outros e alguns podem não estar sujeitos a mudanças freqüentes. Arthur (1988) sugere que os dados devem ser coletados para ajudar a identificar os programas que podem se beneficiar mais com a reestruturação. Por exemplo, as seguintes métricas podem ser utilizadas, a fim de identificar os candidatos à reestruturação:

- índice de falhas;
- porcentagem de código-fonte modificada por ano, e
- complexidade do componente.

Outros fatores, como o grau em que os programas ou os componentes satisfazem aos atuais padrões, também podem ser levados em conta, quando se tomam decisões sobre reestruturação.

28.4 Modularização de programa

A modularização do programa é o processo de reorganizar um programa, de modo que as partes relacionadas sejam coletadas e consideradas um único módulo. Uma vez feito isso, fica mais fácil remover as redundâncias nesses componentes relacionados, otimizar suas interações e simplificar suas interfaces com o restante do programa. Por exemplo, em um programa que processa dados sismográficos, todas as operações associadas com a apresentação gráfica dos dados podem ser reunidas em um único módulo. Se o sistema tiver de ser distribuído, os módulos criados poderão ser encapsulados como objetos e acessados por meio de uma interface comum.

Vários tipos de módulos podem ser criados durante o processo de modularização do programa. Dentre eles destacam-se:

1. *Abstrações de dados* São os tipos de dados abstratos, criados a partir da associação de dados com componentes de processamento. Esse tópico será discutido na Seção 28.4.1.
2. *Módulos de hardware* Esses módulos estão estreitamente relacionados com as abstrações de dados e agrupam todas as funções utilizadas para controlar um determinado dispositivo de hardware.
3. *Módulos funcionais* São os módulos que agrupam as funções que realizam tarefas semelhantes ou estreitamente relacionadas. Por exemplo, todas as funções ocupadas com a entrada e a validação da entrada podem ser incorporadas em um único módulo. Esse tipo de modularização deve ser considerado quando for impraticável recuperar as abstrações de dados de programa.
4. *Módulos de apoio ao processo* São os módulos nos quais são agrupados todas as funções e todos os itens específicos de dados requeridos para apoiar um processo de negócios específico. Por exemplo, em um sistema de biblioteca, um módulo de apoio ao processo pode incluir toda a funcionalidade exigida para apoiar a emissão e o retorno de livros.

A modularização de programa em geral é realizada manualmente, com a inspeção e a edição do código. Para modularizar um programa, é preciso identificar as relações entre componentes e examinar o que eles fazem. Ferramentas de browsing e de visualização ajudam, mas é impossível automatizar esse processo completamente.

28.4.1 Recuperação de abstrações de dados

Para economizar espaço de memória, muitos sistemas legados se baseiam no uso de tabelas compartilhadas e em áreas de dados comuns. As informações nessas áreas são globalmente acessíveis e podem ser utilizadas por diferentes partes do sistema de diferentes maneiras. Fazer modificações nessas áreas de dados globais é uma tarefa dispendiosa, devido aos custos referentes à análise dos impactos das modificações por meio de todos os usos dos dados.

Para reduzir os custos das mudanças nessas áreas de dados compartilhados, o processo de modularização do programa pode se dedicar à identificação de abstrações de dados. As abstrações de dados ou os tipos de dados abstratos agrupam os dados e o processamento associado e são flexíveis em relação às mudanças. As abstrações de dados ocultam a representação deles e proporcionam funções de construção e de acesso, a fim de modificá-los e

inspecioná-los. Desde que a interface seja mantida, as modificações no tipo de dados não devem afetar outras partes do programa.

As etapas envolvidas na conversão de áreas globais de dados compartilhados para objetos ou tipos de dados abstratos são:

1. Analisar áreas comuns de dados para identificar suas abstrações lógicas. Freqüentemente, ocorrerá o caso em que várias abstrações estarão combinadas em uma única área de dados compartilhados. Elas deverão ser identificadas e reestruturadas logicamente.
2. Criar um objeto ou tipos de dados abstratos para cada uma dessas abstrações. Se a linguagem de programação não tiver recursos de ocultação de dados, simule um tipo de dado abstrato, fornecendo funções para atualizar e acessar todos os campos dos dados.
3. Utilizar um sistema de browsing, ou um gerador de referência cruzada, para encontrar todas as referências aos dados. Substituir essas referências por chamadas para funções apropriadas.

Esse processo parece ser demorado, mas é relativamente direto. Na prática, contudo, ele pode ser muito difícil, devido ao modo pelo qual as áreas de dados compartilhados são utilizadas. Nas versões anteriores de linguagem, como FORTRAN, que tem recursos limitados de estruturação de dados, os programadores podem ter projetado estratégias complexas de gerenciamento de dados, que foram implementadas com a utilização de vetores compartilhados. Portanto, o vetor pode, na verdade, ser utilizado como um tipo diferente de estrutura de dados. Outros problemas são causados pelo endereçamento indireto de estruturas de dados e endereços por offsets, a partir de alguma outra estrutura.

Se a máquina-alvo para o programa original tem memória limitada, isso causa outros problemas. Os programadores podem ter utilizado conhecimento sobre tempo de vida de dados e inserido isso no programa. Para evitar alocação de espaço extra, eles utilizam as mesmas áreas de dados para armazenar diferentes abstrações, em diferentes pontos do programa. Essas somente podem ser descobertas depois de uma análise estática e dinâmica detalhada do programa.

28.5 Reengenharia de dados

Até agora, a maior parte da discussão sobre a evolução de software focalizou os problemas com as modificações nos programas e nos sistemas de software. Contudo, em muitos casos, existem problemas associados à evolução dos dados. O armazenamento, a organização e a formatação dos dados processados por programas legados podem precisar evoluir, a fim de refletir as mudanças no software. O processo de análise e reorganização de estruturas de dados e, algumas vezes, os valores dos dados em um sistema, para torná-lo mais compreensível, são chamados de *reengenharia de dados*.

Em princípio, a reengenharia de dados não deverá ser necessária, se a funcionalidade do sistema permanecer inalterada. Na prática, contudo, há uma série de razões pelas quais é preciso modificar os dados, como também os programas, em um sistema legado:

1. *Degradação dos dados* Com o passar do tempo, a qualidade dos dados tende a diminuir. As modificações nos dados introduzem erros, valores duplicados podem ter sido criados e as mudanças no ambiente externo podem não estar refletidas nos dados. Isso é inevitável, porque, geralmente, o tempo de vida útil dos dados é muito longo. Por exemplo, os dados pessoais de um banco são criados quando uma conta é aberta e devem continuar existindo, pelo menos, ao longo da vida do cliente. À medida que as circunstâncias relacionadas ao cliente se modificam, essas mudanças podem não ser adequadamente incluídas nos dados do banco. A reengenharia de programa pode identificar os problemas de qualidade dos dados e, assim, ressaltar a necessidade da reengenharia de dados associados.

2. *Limites inerentes inseridos no sistema* Os desenvolvedores de muitos programas, no momento em que eles foram originalmente projetados, incluíram restrições quanto à quantidade de dados que poderia ser processada. Contudo, hoje em dia, é necessário que os programas processem muito mais dados do que o volume que foi originalmente imaginado pelos seus desenvolvedores. A reengenharia de dados pode ser exigida para eliminar essas limitações. Por exemplo, Rochester e Douglass (1993) descrevem um sistema de gerenciamento de fundos, que foi originalmente projetado para manipular até 99 fundos. A empresa que executava o sistema estava gerenciando mais de dois mil fundos e tinha de executar 23 cópias separadas do sistema. Eles decidiram, então, pela reengenharia do sistema e de seus dados associados.

3. *Evolução da arquitetura* Se um sistema centralizado migrar para uma arquitetura distribuída, é essencial que o núcleo dessa arquitetura seja um sistema de gerenciamento de dados, que pode ser acessado a partir de clientes remotos. Isso pode requerer um grande esforço relativo à reengenharia de dados, a fim de mover esses dados a partir de arquivos separados para dentro do sistema de gerenciamento de banco de dados do servidor. A iniciativa de migrar para uma arquitetura de programa distribuída pode ter início quando uma organização decide migrar do gerenciamento de dados baseado em arquivos para o sistema de gerenciamento de banco de dados.

Assim como acontece com a reengenharia de programa, existem várias abordagens para a reengenharia de dados, que refletem as razões pelas quais ela pode ser necessária. Essas abordagens são mostradas na Figura 28.10.

Rickets *et al.* (1993) descrevem alguns problemas que podem surgir com os dados em sistemas legados constituídos de diversos programas que cooperam entre si:

1. *Problemas com a denominação dos dados* Os nomes podem ser obscuros e de difícil compreensão. Nomes diferentes (sinônimos) podem ser dados à mesma entidade lógica, em diferentes programas do sistema. O mesmo nome pode ser utilizado em diferentes programas com significados diferentes.

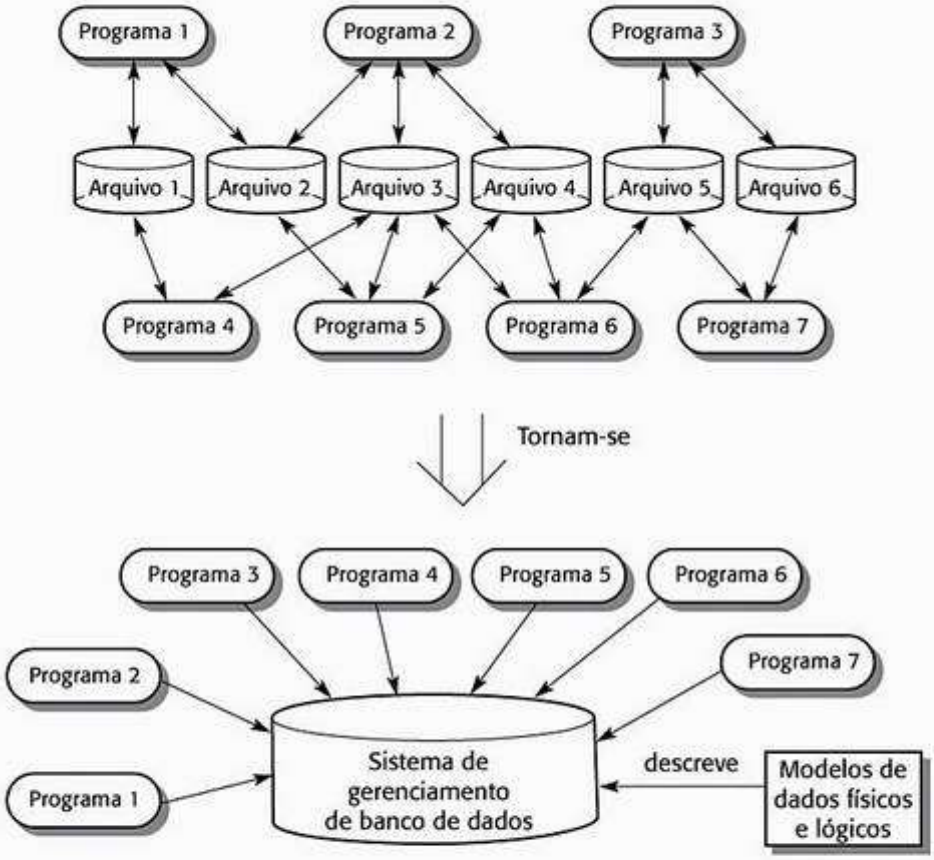
2. *Problemas com a extensão de campos* Isso acontece quando a extensão do campo em registros é explicitamente atribuída ao programa. Para o mesmo item, podem ser designadas diferentes extensões, em diferentes programas, ou a extensão do campo pode ser muito curta para representar os dados atuais. Para resolver esse problema, outros campos podem ser reutilizados, em alguns casos, de modo que a utilização de um campo de dados denominado nos programas de um sistema seja inconsistente.

Figura 28.10 ▶

Abordagens da reengenharia de dados.

Abordagem	Descrição
Limpeza de dados	Os registros e valores de dados são analisados, a fim de melhorar sua qualidade. As duplicações são removidas, as informações redundantes são excluídas e um formato consistente é aplicado a todos os registros. Normalmente, isso não deve requerer quaisquer mudanças nos programas associados.
Extensão de dados	Nesse caso, os dados e programas associados passam pelo processo de reengenharia, a fim de eliminar os limites no processamento de dados. Isso pode exigir mudanças nos programas para aumentar a extensão de campos, modificar limites superiores nas tabelas e assim por diante. Os dados em si podem precisar ser reescritos e limpos, para que reflitam as mudanças no programa.
Migração de dados	Nesse caso, ocorre a migração dos dados para o controle de um moderno sistema de gerenciamento de banco de dados. Os dados podem ser armazenados em arquivos separados ou ser gerenciados por um tipo de sistema de gerenciamento de banco de dados mais antigo. Essa situação é ilustrada na Figura 28.11.

Figura 28.11
Migração de dados.



3. *Problemas com a organização de registros* Os registros que representam a mesma entidade podem ser organizados diferentemente, em diferentes programas. Esse é um problema das linguagens como COBOL, quando a organização física dos registros é estabelecida pelo programador e refletida em arquivos. Em linguagens como C ++ ou Java, isso não é um problema, porque a organização física de um registro é de responsabilidade de um compilador.

4. *Literais hard-coded* Valores de literais (absolutos), como taxas de impostos, são incluídos diretamente no programa, em vez de serem referenciados com a utilização de um nome simbólico.

5. *Nenhum dicionário de dados* Pode não haver nenhum dicionário de dados que defina os nomes utilizados, sua representação e seu uso.

Assim como as definições de dados inconsistentes, os valores de dados também podem ser armazenados de modo inconsistente. Depois que as definições de dados passarem pelo processo de reengenharia, os valores de dados também deverão ser convertidos, a fim de se adequarem à nova estrutura. Rickets *et al.* também descrevem algumas inconsistências possíveis dos valores de dados. Essa situação é mostrada na Figura 28.12.

A análise detalhada dos programas que utilizam os dados é essencial, antes de ser feita a reengenharia de dados. Essa análise deve visar à descoberta da função dos identificadores no programa, pretender encontrar os valores de literais que devem ser substituídos por constantes denominadas e descobrir regras embutidas de validação de dados e conversões de representação de dados. Determinadas ferramentas, como os analisadores de referência cruzada e os comparadores de padrões, podem ser utilizadas para ajudar nessa análise. Deve ser criado um conjunto de tabelas que mostre onde os itens de dados são referenciados e as modificações a serem feitas em cada uma dessas referências.

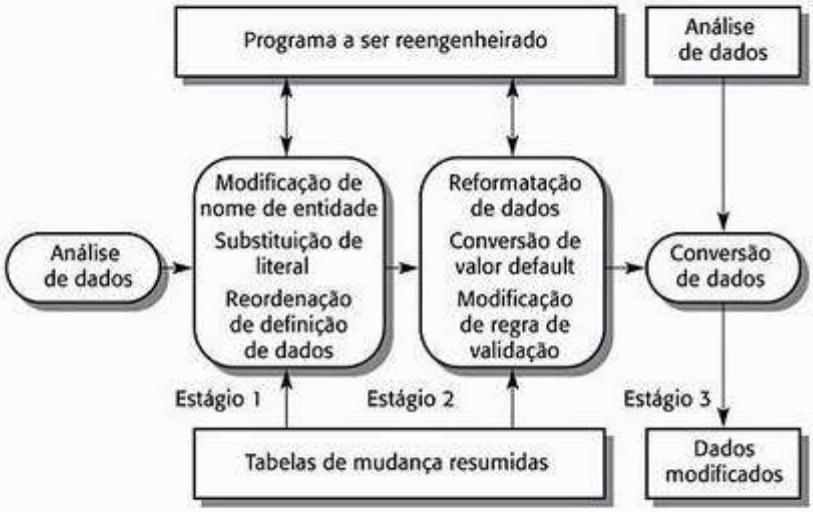
Figura 28.12 ▶

Inconsistência do valor dos dados.

Inconsistência dos dados	Descrição
Valores default inconsistentes	Diferentes programas atribuem diferentes valores default aos mesmos itens de dados lógicos. Isso causa problemas para outros programas que não sejam os que criaram os dados. Esse problema acontece quando aos valores que faltam é atribuído um valor default válido. Os dados que faltam não poderão ser descobertos.
Unidades inconsistentes	A mesma informação é representada em diferentes unidades, em diferentes programas. Por exemplo, nos Estados Unidos ou no Reino Unido, os dados referentes a peso podem ser representados em libras em programas mais antigos e em quilogramas em sistemas mais recentes. Um problema maior desse tipo surgiu na Europa, com a introdução da moeda única. Os sistemas legados foram escritos para lidar com unidades de moeda nacional, de modo que os dados precisam ser convertidos para o Euro.
Regras de validação inconsistentes	Diferentes programas aplicam diferentes regras de validação de dados. Os dados escritos para um programa podem ser rejeitados por outro. Esse é um problema específico para dados de arquivos que podem não ter sido atualizados de acordo com as mudanças nas regras de validação de dados.
Semânticas de representação inconsistentes	Os programas presumem algum significado no modo como os itens são representados. Por exemplo, alguns programas podem presumir que o texto com letra maiúscula representa um endereço. Eles podem utilizar diferentes convenções e, portanto, rejeitar dados que sejam semanticamente válidos.
Manuseio inconsistente de valores negativos	Alguns programas rejeitam valores negativos para entidades que devem ser sempre positivas. Outros, contudo, podem aceitá-las como valores negativos ou deixar de reconhecê-las como negativas e convertê-las para um valor positivo.

Figura 28.13 ▶

Processo de reengenharia de dados.



A Figura 28.13 ilustra o processo de reengenharia de dados, supondo que as definições de dados sejam modificadas, os valores literais sejam denominados, os formatos de dados sejam reorganizados e os valores de dados sejam convertidos. As tabelas de mudança resumidas contêm detalhes de todas as mudanças a serem efetuadas. Portanto, elas são utilizadas em todos os estágios do processo de reengenharia de dados.

No Estágio 1 desse processo, as definições de dados no programa são modificadas para melhorar a facilidade de compreensão. Os dados, por si mesmos, não são afetados por essas

modificações. É possível automatizar esse processo até certo ponto, utilizando sistemas de comparação de padrões, como *awk* (Aho *et al.*, 1988), para encontrar e substituir definições ou para desenvolver descrições dos dados em XML (St. Laurent e Cerami, 1999), e utilizar isso para derivar ferramentas de conversão de dados. Contudo, algum trabalho manual é quase sempre necessário para completar o processo. O processo de reengenharia de dados pode ser interrompido, nesse estágio, se o objetivo for simplesmente melhorar a facilidade de compreensão das definições de estrutura de dados em um programa. Todavia, se houver problemas com os valores dos dados, como foi discutido anteriormente, o Estágio 2 do processo poderá ser iniciado.

Se uma organização decidir prosseguir com o Estágio 2 do processo, ela fica, então, obrigada a ir para o Estágio 3, a conversão de dados. Geralmente, esse é um processo muito dispendioso. Devem ser escritos programas que incluam o conhecimento da antiga e da nova organização. Eles processam os dados antigos e geram informação convertida. Novamente, os sistemas de comparação de padrões podem ser utilizados para implementar essa conversão.

PONTOS-CHAVE

- O objetivo da reengenharia de sistemas é melhorar a estrutura do sistema e torná-la mais fácil de ser compreendida. O custo da manutenção futura do sistema deve, portanto, ser reduzido.
- O processo de reengenharia inclui tradução de código-fonte, engenharia reversa, melhoria de estrutura de programa, modularização de programa e reengenharia de dados.
- A tradução de código-fonte é a conversão automatizada de um programa escrito em uma linguagem de programação para outra linguagem. Ela pode ser necessária quando a linguagem de programação original for obsoleta.
- A engenharia reversa é o processo de derivar um projeto e a especificação de sistema a partir de seu código-fonte. Ferramentas como os browsers de programa podem ser utilizadas para ajudar no processo.
- A melhoria da estrutura de programa envolve substituir construções não estruturadas de controle, como declarações *goto*, por *loops while* e declarações condicionais. Esse processo pode ser automatizado.
- A modularização de programa envolve reorganizar o código-fonte de um programa, a fim de agrupar itens relacionados. Isso os torna mais fáceis de ser compreendidos e modificados.
- A reengenharia de dados pode ser necessária, devido ao gerenciamento inconsistente de dados pelos programas em um sistema legado. O objetivo da reengenharia de dados pode ser a reengenharia de todos os programas, a fim de utilizar um banco de dados comum.
- Os custos da reengenharia de dados serão aumentados significativamente se os dados existentes tiverem de ser convertidos para um novo formato.

LEITURAS SUGERIDAS

"Examining data quality". Essa seção especial traz uma série de artigos que discutem questões de qualidade de dados e o impacto da má qualidade dos dados. (G.K. Tayi e D.P. Ballou, *Comm. ACM*, 41(2), fev. 1998.)

Software re-engineering. Trata-se de um tutorial do IEEE, que apresenta a maior parte dos artigos importantes sobre reengenharia que foram publicados antes de 1992. Muitos dos artigos mencionados neste capítulo foram reimpressos nesse tutorial. (R.S. Arnold, 1994, IEEE Press.)

"DoD legacy sistems: reverse engineering data requirements". Apresenta uma boa descrição dos problemas práticos que surgem com os sistemas legados. O artigo focaliza a reengenharia de dados, em que sistemas que gerenciam dados similares, mas incompatíveis, foram combinados. Outros artigos dessa edição especial sobre engenharia reversa também são relevantes. (P. Aiken, A. Muntz e R. Richards, *Comm. ACM*, 37(5), maio 1994.)

EXERCÍCIOS

28.1 Sob quais circunstâncias você acredita que um software deve ser descartado e reescrito, em vez de passar por reengenharia?

28.2 Compare as construções de controle (loops e condicionais) em duas linguagens de programação que você conheça. Escreva uma breve descrição de como traduzir as construções de controle de uma linguagem para as construções correspondentes da outra.

28.3 Traduza a rotina não estruturada mostrada na Figura 28.14 para sua equivalente estruturada e mostre o que ela deve fazer.

28.4 Escreva um conjunto de diretrizes que pode ser utilizado para ajudar a encontrar módulos em um programa não estruturado.

28.5 Sugira nomes significativos para as variáveis utilizadas no programa, mostradas na Figura 28.14, e construa entradas de dicionário de dados para esses nomes.

28.6 Que problemas podem surgir quando se convertem dados de um tipo de sistema de gerenciamento de banco de dados para outro (por exemplo, de hierárquico para relacional ou de relacional para orientado a objetos)?

28.7 Explique por que é impossível recuperar uma especificação de sistema analisando automaticamente o código-fonte do sistema.

Figura 28.14 ▶

Programa não estruturado.

```
routine BS (K, T, S, L)
B:= 1
NXT:  if S >= B goto CON
L = −1
goto STP
CON:  L := INTEGER (B / S)
L := INTEGER ((B+S) / 2)
if T (L) = K then return
if T(L) > K then goto GRT
B := L+1
goto NXT
GRT:  S := L−1
goto NXT
STP: end
```

28.8 Utilizando exemplos, descreva os problemas com a degradação de dados que podem ser enfrentados durante o processo de limpeza de dados.

28.9 O problema 'bug do milênio', que se referia à questão de as datas serem representadas por dois dígitos nos computadores, tornou-se um importante problema de manutenção de programa, para muitas organizações. Quais foram as implicações desse problema para a reengenharia de dados?

28.10 Uma empresa estabelece, de maneira rotineira, condições contratuais para os programadores free lancers que trabalham na reengenharia de suas aplicações, o que os impede de aceitar contratos com empresas similares. A razão para isso é que a reengenharia, inevitavelmente, revela informações de negócios. Essa é uma atitude razoável por parte da empresa, considerando que eles não têm nenhuma obrigação com a contratante depois que seus contratos terminam?

29 Gerenciamento de configuração

Objetivos

O objetivo deste capítulo é apresentar o processo de gerenciamento de código e documentação de um sistema de software em desenvolvimento. Depois de ler este capítulo, você poderá:

- compreender por que o gerenciamento de configuração de software é importante;
- conhecer quatro atividades principais de gerenciamento de configuração — planejamento de gerenciamento de configuração, gerenciamento de mudanças, gerenciamento de versões e releases e construção de sistemas, e
- compreender como as ferramentas CASE para o gerenciamento de configuração podem ser utilizadas para apoiar processos de gerenciamento de configuração.

Sumário

29.1 Planejamento de gerenciamento de configuração
29.2 Gerenciamento de mudanças
29.3 Gerenciamento de versões e releases
29.4 Construção de sistemas
29.5 Ferramentas CASE para o gerenciamento de configuração

O gerenciamento de configuração (*configuration management* – CM) é o desenvolvimento e a aplicação de padrões e procedimentos para gerenciar um produto de sistema em desenvolvimento. É necessário gerenciar os sistemas em desenvolvimento porque, à medida que eles se desenvolvem, são criadas muitas versões diferentes de software. Essas versões incorporam propostas de mudanças, correções de defeitos e adaptações para diferentes hardwares e sistemas operacionais. É possível que haja várias versões em desenvolvimento e em uso ao mesmo tempo. É necessário manter o controle das mudanças que foram implementadas e de como essas mudanças foram incluídas no software.

Os procedimentos de gerenciamento de configuração definem como registrar e processar as mudanças propostas para o sistema, como relacioná-las aos componentes de sistema e aos métodos utilizados para identificar diferentes versões do sistema. As ferramentas de gerenciamento de configuração são utilizadas para armazenar versões de componentes de sistema, construir sistemas a partir desses componentes e acompanhar os releases das versões de sistema para os clientes.

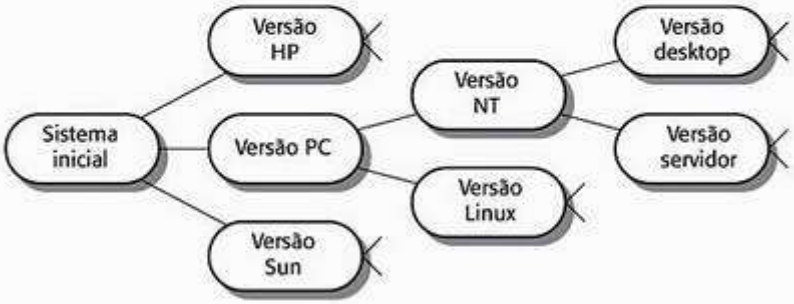
O gerenciamento de configuração é, às vezes, visto como parte de um processo mais geral de gerenciamento de qualidade de software, como foi discutido no Capítulo 24. O mesmo gerente pode compartilhar as responsabilidades pelo gerenciamento de qualidade e pelo gerenciamento de configuração. O software é liberado pelos desenvolvedores para uma equipe de controle de qualidade, que é responsável por conferir se o sistema é de qualidade aceitável. Esse software é, então, passado para a equipe de gerenciamento de configuração, que fica responsável por controlar as mudanças no software. Os sistemas controlados são, algumas vezes, chamados de baselines, quando são o ponto de partida para o desevolvimento controlado.

Existem muitas razões pelas quais os sistemas existem sob diferentes configurações. As configurações podem ser produzidas para diferentes computadores, para diferentes sistemas operacionais, incorporando funções específicas para o cliente, e assim por diante (Figura 29.1). Os gerentes de configuração são responsáveis por manter o controle das diferenças entre as versões de software, a fim de assegurar que as novas versões sejam derivadas de uma maneira controlada e por liberar novas versões para os clientes certos e no momento certo.

O processo de gerenciamento da configuração e da documentação associada deve ser baseado em padrões. Um exemplo desses padrões é o IEEE 828-1983, que define um padrão para planos de gerenciamento de configuração. Dentro de uma organização, esses padrões devem ser publicados em um manual de gerenciamento de configuração ou como parte de um manual de qualidade. Os padrões externos podem ser utilizados como base para padrões organizacionais mais detalhados, que são feitos de maneira adequada a um ambiente específico. Na verdade, não importa qual padrão seja considerado ponto de partida, uma vez que todos eles apresentam processos comparáveis. Tanto nos padrões de qualidade da ISO 9000 (Ince, 1994) quanto no Modelo de Maturidade de Capacitação do SEI (Paulk *et al.*, 1993), as organizações devem definir e seguir padrões formais de CM para a certificação de qualidade.

Em um processo tradicional de desenvolvimento de software, baseado no modelo em cascata (veja o Capítulo 3), o software é entregue para a equipe de gerenciamento de configuração depois que o desenvolvimento foi concluído e os componentes individuais de software foram

Figura 29.1 ▶
Famílias de sistemas.



testados. Essa equipe assume, então, a responsabilidade de construir o sistema completo e de gerenciar os testes de sistema. Os defeitos descobertos durante os testes de sistema são relatados para a equipe de desenvolvimento, para que sejam reparados. Depois que foram feitos esses reparos, uma nova versão do componente reparado é entregue para a equipe de CM.

Essa abordagem influenciou o desenvolvimento de padrões de gerenciamento de configuração, e esses padrões têm uma suposição embutida, de que o modelo em cascata do processo de software será utilizado para o desenvolvimento do sistema (Bersoff e Davis, 1991). Isso significa que eles não podem ser prontamente aplicados a abordagens alternativas de desenvolvimento de software, como a prototipação evolucionária ou o desenvolvimento incremental. Para possibilitar esse estilo de desenvolvimento, algumas organizações desenvolveram uma abordagem modificada do gerenciamento de configuração, que apóia o desenvolvimento e a realização de testes de sistema concorrentes. Essa abordagem se baseia em uma construção muito regular (às vezes, diariamente) de todo o sistema, a partir de seus componentes:

1. A organização de desenvolvimento estabelece um horário de entrega (digamos, às 14 horas) para os componentes de sistema. Se os desenvolvedores estiverem escrevendo novas versões dos componentes, eles deverão entregá-las dentro desse prazo. Os componentes podem estar incompletos, mas deverão fornecer alguma funcionalidade básica que possa ser testada.
2. Uma nova versão do sistema é construída a partir desses componentes, por compilação e ligação, para constituírem um sistema completo.
3. Em seguida, o sistema é entregue à equipe de testes, que realiza um conjunto de testes predefinidos para o sistema. Ao mesmo tempo, os desenvolvedores ainda estão trabalhando em seus componentes, para aumentar a funcionalidade e reparar os defeitos descobertos em testes anteriores.
4. Os defeitos descobertos durante os testes de sistema são documentados e relatados aos desenvolvedores do sistema. Eles reparar esses defeitos em uma versão subseqüente do componente.

A principal vantagem de utilizar construções diárias de software é que elas aumentam as chances de encontrar problemas que surgem a partir das interações de componentes, no início do processo. Além disso, a construção diária encoraja os testes de unidades de componentes. Psicologicamente, os desenvolvedores ficam sob a pressão de não prejudicar a construção, isto é, entregar versões de componentes que provoquem a falha de todo o sistema. Eles, portanto, relutam em entregar novas versões de componentes que não tenham sido adequadamente testadas. Um tempo menor para os testes de sistema é gasto na descoberta e resolução de defeitos de software que deveriam ter sido descobertos durante os testes de unidades.

O uso bem-sucedido de construções diárias requer um processo muito rigoroso de gerenciamento de mudanças, a fim de manter o controle dos problemas descobertos e reparados. Ele também leva a uma série muito grande de versões de sistemas e componentes, que precisam ser gerenciados. Portanto, um bom gerenciamento de configuração é essencial para que essa abordagem seja bem-sucedida.

29.1 Planejamento de gerenciamento de configuração

Um plano de gerenciamento de configuração descreve os padrões e os procedimentos que devem ser utilizados para o gerenciamento de configuração. O ponto de partida para

desenvolver o plano deve ser um conjunto de padrões gerais de gerenciamento de configuração para toda a empresa, os quais devem ser adaptados, conforme necessário, a cada projeto específico. O plano de CM deve ser organizado em uma série de capítulos e deve apresentar:

1. a definição de quais entidades serão gerenciadas e um esquema formal para identificar essas entidades;
2. uma declaração de quem assume a responsabilidade pelos procedimentos de gerenciamento de configuração e por submeter as entidades controladas à equipe de gerenciamento de configuração;
3. as políticas de gerenciamento de configuração que são utilizadas para o controle de qualidade e o gerenciamento de versão;
4. uma descrição dos registros do processo de gerenciamento de configuração que devem ser mantidos;
5. uma descrição das ferramentas a serem utilizadas pelo gerenciamento de configuração e do processo a ser aplicado quando são utilizadas essas ferramentas, e
6. uma definição do banco de dados de configuração, que será utilizado para registrar as informações sobre a configuração.

Outras informações, como o gerenciamento de software proveniente de fornecedores externos e os procedimentos de auditoria para o processo de CM, podem também ser incluídas no plano de CM.

Uma parte importante do plano de CM é a definição de responsabilidades. Ela deve definir quem é responsável pela entrega de cada documento ou componente de software para o gerenciamento de garantia de qualidade e configuração. Ela também pode definir os revisores de cada documento. A pessoa responsável pela entrega do documento não precisa ser a mesma que produz o documento. Para simplificar as interfaces, muitas vezes, é conveniente dar aos gerentes de projeto, ou ao líderes de equipe, a responsabilidade por todos os documentos produzidos pelas suas equipes.

29.1.1 Identificação de itens de configuração

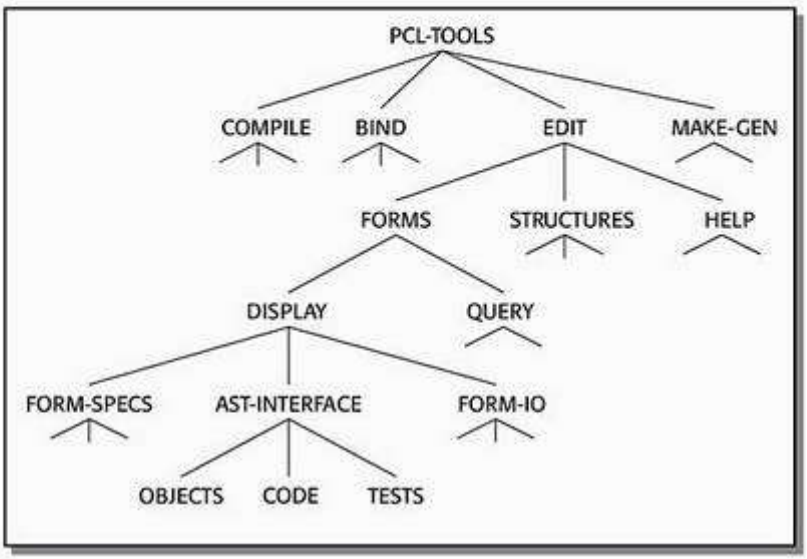
Ao longo do desenvolvimento de um sistema de software de grande porte, milhares de documentos são produzidos. Muitos deles são documentos técnicos de trabalho, que apresentam um retrato de idéias, para desenvolvimento posterior, e estão sujeitos a mudanças freqüentes e regulares. Outros são memorandos entre escritórios, minutas de reuniões de grupo, esboços de planos e propostas, entre outros. Esses documentos podem ser de interesse para alguém que queira manter um histórico do projeto. Contudo, eles não são necessários para a futura manutenção do sistema.

Durante o processo de planejamento de gerenciamento de configuração, decide-se exatamente que itens (ou classes de itens) devem ser controlados. Os documentos ou grupos de documentos relacionados, sob o controle de configuração, são documentos formais ou itens de configuração. Os planos de projeto, as especificações, os projetos, os programas e os conjuntos de dados de teste são, geralmente, mantidos como *itens de configuração*. Contudo, todos os documentos que podem ser úteis para a futura manutenção do sistema devem ser controlados.

O esquema de denominação de documentos deve atribuir um único nome a todos os documentos sob o controle de configuração. Sempre existem relações entre esses documentos. Por exemplo, os documentos de projeto serão associados a programas. Essas relações podem ser registradas implicitamente, mediante a organização do esquema de denominação, de modo

Figura 29.2

Hierarquia de configuração.



que os documentos relacionados tenham uma raiz comum em seu nome. Isso leva a um esquema de nomes hierarquizados, que podem ser:

PCL-TOOLS/EDIT/FORMS/DISPLAY/AST-INTERFACE/CODE
PCL-TOOLS/EDIT/HELP/QUERY/HELPFRAMES/FR-1

A parte inicial é o nome do projeto, PCL-TOOLS. Nesse projeto, há quatro ferramentas (TOOLS) separadas. O nome da ferramenta é utilizado como a próxima parte do nome. Cada ferramenta é constituída de módulos denominados de maneira diferente. Esse processo de decomposição continua até que os documentos formais do nível básico sejam referenciados (Figura 29.2). As folhas da hierarquia de documentação são os documentos formais de projeto. A Figura 29.2 mostra que três documentos formais são requeridos para cada entidade gerenciada. Eles são a descrição de objeto (OBJECTS), o código do componente (CODE) e o conjunto de testes para esse código (TESTS).

O problema com os esquemas de denominação desse tipo é que eles são baseados no projeto. Os identificadores associam os componentes com um determinado projeto. Isso pode reduzir as oportunidades de reuso. Cópias de componentes reutilizáveis devem normalmente ser retiradas desse esquema e renomeadas de acordo com seu domínio de aplicação. Outros problemas poderão surgir, se o esquema de denominação de documentos for utilizado como uma base direta para projetar uma estrutura de repositório para armazenar os componentes gerenciados. Os usuários dos documentos devem, então, saber seus nomes, para poder encontrá-los, e todos os documentos do mesmo tipo (por exemplo, os documentos de projeto) não são mantidos juntos em um mesmo lugar. É possível que também haja problemas em relacionar o esquema de denominação ao esquema de identificação utilizado por um sistema de gerenciamento de versões.

29.1.2 O banco de dados de configuração

O banco de dados de configuração é utilizado para registrar todas as informações relevantes relacionadas às configurações. Suas funções principais são ajudar na avaliação do impacto das mudanças ocorridas no sistema e fornecer informações gerenciais sobre o processo de CM. Além de definir o esquema de banco de dados de configuração, os procedimentos para registro e recuperação de informações de projeto devem ser definidos como parte do processo de planejamento de CM.

Um banco de dados de configuração tem de ser capaz de fornecer respostas a uma variedade de consultas sobre as configurações de sistema. Consultas típicas podem ser:

1. Que clientes receberam uma determinada versão do sistema?
2. Que hardware e que configuração de sistema operacional são necessários para executar uma versão específica do sistema?
3. Quantas versões de um sistema foram criadas e quais foram as datas de sua criação?
4. Que versões de um sistema podem ser afetadas, se um determinado componente for modificado?
5. Quantas solicitações de mudança podem ser feitas em uma versão específica?
6. Quantos defeitos relatados existem em uma determinada versão?

O ideal é que o banco de dados de configuração seja integrado com o sistema de gerenciamento de versão, que é utilizado para armazenar e gerenciar os documentos formais de projeto. Essa abordagem, apoiada por algumas ferramentas CASE integradas, torna possível vincular diretamente as mudanças aos documentos e aos componentes afetados por elas. As vinculações entre documentos, como documentos de projeto, e o código do programa podem ser mantidas de modo que é relativamente fácil encontrar tudo o que deve ser modificado, quando uma mudança for proposta.

Contudo, muitas empresas não utilizam ferramentas CASE integradas para o gerenciamento de configuração, mas mantêm o seu banco de dados de configuração como um sistema separado. Os itens de configuração podem ser armazenados em arquivos ou em uma versão do sistema de gerenciamento, como RCS (Tichy, 1985), uma versão bem conhecida de sistema de gerenciamento para Unix. Esse banco de dados de configuração armazena informações sobre os itens de configuração e referencia seus nomes de arquivo na versão de sistema de gerenciamento. Embora essa seja uma abordagem relativamente barata e flexível, sua desvantagem é que os itens de configuração podem ser modificados sem passar pelo banco de dados de configuração. Portanto, não se pode ter certeza de que o banco de dados de configuração é uma descrição atualizada do estado do sistema.

29.2 Gerenciamento de mudanças

As mudanças são um 'fato da vida' para os sistemas de software de grande porte. Como foi discutido em capítulos anteriores, as necessidades e os requisitos organizacionais modificam o tempo de vida útil de um sistema. Isso requer que mudanças correspondentes sejam feitas no software. Um processo definido de gerenciamento de mudanças e as ferramentas CASE associadas garantem que essas mudanças sejam registradas e aplicadas ao sistema de maneira econômica.

O processo de gerenciamento de mudanças (Figura 29.3) deve entrar em ação quando o software, ou a documentação associada, estiver sob o controle da equipe de gerenciamento de configuração. Ele pode ser iniciado durante os testes de sistemas ou depois que o software tenha sido entregue aos clientes. Os procedimentos de gerenciamento de mudanças devem ser concebidos para assegurar que os custos e os benefícios das mudanças sejam adequadamente analisados e as mudanças em um sistema sejam feitas de maneira controlada.

O primeiro estágio do processo de gerenciamento de mudanças é o preenchimento de um formulário de solicitação de mudança (*change request form* – CRF), em que o requisitante estabelece a mudança requerida no sistema. Além de registrar a solicitação de mudança, o CRF registra as recomendações relativas à mudança, os custos estimados relacionados e as datas de solicitação, aprovação, implementação e validação da mudança. Ele pode também incluir uma seção na qual o engenheiro de manutenção faz um esboço de como a mudança

Figura 29.3

Processo de gerenciamento de mudança.

```
Solicite a mudança preenchendo um formulário de solicitação de mudança (CRF)
Analise a solicitação de mudança
if a mudança for válida, then
    Avalie como a mudança pode ser implementada
    Avalie o custo da mudança
    Registre a solicitação de mudança no banco de dados
    Submeta a solicitação ao grupo de controle de mudança
    if a mudança for aceita, then
        repeat
            faça as mudanças no software
            registre as mudanças e a vinculação à solicitação de mudança associada
            submeta o software modificado à aprovação de qualidade
        until a qualidade do software ser adequada
        crie nova versão do sistema
    else
        recuse a solicitação de mudança
else
    recuse a solicitação de mudança
```

deverá ser implementada. As solicitações de mudança devem ser registradas no banco de dados de configuração. A equipe de CM pode, portanto, acompanhar o status das solicitações de mudança que são associadas com componentes de software específicos.

Um exemplo de formulário de solicitação de mudança parcialmente preenchido é mostrado na Figura 29.4. Esse formulário, em geral, é definido durante o processo de planejamento do CM. Contudo, para alguns contratos, particularmente contratos com o governo, o formulário de solicitação de mudança precisa estar de acordo com um padrão especificado pelo cliente.

Uma vez que um formulário de solicitação de mudança tenha sido submetido, ele é analisado, a fim de ser verificado se a mudança é válida. Algumas solicitações de mudança podem ocorrer em virtude de erros de compreensão, e não de defeitos do sistema; outras podem se referir a defeitos já conhecidos. Se a análise constatar que uma solicitação de mudança é inválida, duplicada ou que ele já tenha sido considerada, a mudança deve ser rejeitada. A razão para a rejeição deve ser encaminhada à pessoa que fez o pedido.

Figura 29.4

Formulário de solicitação de mudança (CRF) parcialmente preenchido.

Formulário de Solicitação de Mudança

Projeto: Proteus/PCL-TOOLS **Número:** 23/94
Requisitante da mudança: I. Sommerville **Data:** 1/12/98
Mudança solicitada: Quando um componente for selecionado a partir da estrutura, exibir o nome do arquivo onde ele está armazenado.

Analista da mudança: G. Dean **Data da análise:** 10/12/98
Componentes afetados: Display-Icon.Select, Display-Icon.Display

Componentes associados: FileTable

Avaliação da mudança: Relativamente simples de implementar quando uma tabela de nomes está disponível. Requer o projeto e a implementação de um campo de display. Nenhuma mudança é requerida nos componentes associados.

Prioridade da mudança: Baixa
Implementação da mudança:
Esforço estimado: 0,5 dia
Data para CCB: 15/12/98 **Data de decisão do CCB:** 1/2/99
Decisão do CCB: Mudança aceita. Mudança a ser implementada no Release 2.1.
Implementador da mudança: **Data da mudança:**
Data de submissão a QA: **Decisão QA:**
Data de submissão a CM:
Comentários

No caso das mudanças válidas, a etapa seguinte do processo é a avaliação da mudança e do custo relacionado. O impacto da mudança no restante do sistema deve ser verificado e deve ser realizada uma análise técnica de como implementar essa mudança. Em seguida, é estimado o custo de fazer essa mudança e possivelmente de modificar outros componentes do sistema, a fim de acomodá-la. Isso deve ser registrado no formulário de solicitação de mudança. Esse processo de avaliação pode utilizar o banco de dados de configuração, onde as inter-relações dos componentes estão registradas. O impacto da mudança em outros componentes pode então ser considerado.

A menos que a mudança simplesmente envolva corrigir erros de menor importância nos displays de tela ou em documentos, ela deverá ser submetida a um comitê de controle de mudança (*change control board* – CCB), que decidirá se a mudança deverá ser aceita ou não. O comitê de controle de mudança considera o impacto da mudança sob o ponto de vista estratégico e organizacional, e não sob o ponto de vista técnico. Ele decide se a mudança é economicamente justificável e se há boas razões organizacionais para aceitá-la.

O termo 'comitê de controle de mudança' pode soar muito formal. Trata-se de um grupo bem maior, que toma as decisões sobre as mudanças. Os comitês de controle de mudança formalmente estruturados, que incluem profissionais de nível sênior do cliente e do fornecedor, constituem uma exigência nos projetos militares. Para os projetos de pequeno e médio porte, contudo, o comitê de controle de mudança pode consistir simplesmente de um gerente de projeto e mais um ou dois engenheiros que não estejam diretamente envolvidos no desenvolvimento de software. Em alguns casos, pode haver apenas um único revisor da mudança, que orienta se as mudanças são justificáveis ou não.

Quando um conjunto de mudanças é aprovado, o software é encaminhado para a equipe de desenvolvimento ou manutenção, para implementação. Uma vez terminadas as mudanças, o software revisado deve ser revalidado, a fim de verificar se as mudanças foram corretamente implementadas. A equipe de CM, e não os desenvolvedores do sistema, constrói, então, uma nova versão ou release do software.

Quando novas versões do sistema são criadas por meio de construções diárias do sistema, é utilizado um processo mais simples de gerenciamento de mudança. Os problemas e as mudanças devem ser ainda registrados, mas as mudanças que afetam somente componentes e módulos individuais não precisam ser avaliadas de maneira independente. Elas são encaminhadas diretamente para o desenvolvedor do sistema, que as aceita ou produz uma justificativa explicando por que elas não são necessárias. As mudanças que afetam os módulos de sistema produzidos por diferentes equipes de desenvolvimento também devem ser avaliadas por algum tipo de autoridade de controle de mudança, que decide se elas devem ser implementadas.

À medida que os componentes de software são modificados, deve ser mantido um registro das mudanças feitas em cada um deles. Algumas vezes, isso é chamado de histórico de alteração de um componente. A melhor maneira de manter esse registro é em um prólogo de comentário padronizado, mantido no início do componente (veja a Figura 29.5). Ele deve se referir à solicitação de mudança associado à mudança de software. Ferramentas especializadas podem ser utilizadas para processar os históricos de alteração e produzir relatórios sobre as mudanças no componente.

29.3 Gerenciamento de versões e releases

O gerenciamento de versões e releases consiste no processo de identificar e acompanhar o desenvolvimento de diferentes versões e releases de um sistema. Os gerentes de versão preparam procedimentos, a fim de garantir que diferentes versões de um sistema possam ser acessadas quando necessário e que elas não sejam modificadas acidentalmente. Eles também podem trabalhar com o pessoal responsável pelo relacionamento com o cliente para planejar quando

Figura 29.5 ▶

Informação de cabeçalho de componente.

```
// Projeto PROTEUS (ESPRIT 6087)
//
// PCL-TOOLS/EDIT/FORMS/DISPLAY/AST-INTERFACE
//
// Objeto: PCL-TOOL-Desc
// Autor: G. Dean
// Data de criação: 10 de novenbro de 1998
//
// © Lancaster University 1998
//
// Histórico de alterações
// Versão   Modificador   Data        Mudança          Razão
// 1.0      J. Jones      1/12/1998   Adicionar cabeçalho  Submetido ao CM
// 1.1      G. Dean       9/4/1999    Novo campo       Ped. mudança R07/99
```

novos releases de um sistema devem ser distribuídos. As novas versões de um sistema devem ser sempre criadas pela equipe de CM, e não pelos desenvolvedores de sistema, mesmo quando não se destinem à liberação externa. Isso facilita manter a consistência da configuração de banco de dados, uma vez que somente a equipe de CM pode modificar as informações sobre a versão.

Uma *versão* de sistema é uma instância de um sistema que difere, de muitas maneiras, de outras instâncias. Novas versões do sistema podem ter diferentes funcionalidades e desempenhos ou podem resolver defeitos no sistema. Algumas versões podem ser funcionalmente equivalentes, mas projetadas para diferentes configurações de hardware ou software. Se houver apenas pequenas diferenças entre versões, às vezes uma delas será chamada de *variante* da outra.

Um *release* de sistema é uma versão que é distribuída para os clientes. Cada release de sistema deve incluir nova funcionalidade ou se destinar a uma diferente plataforma de hardware. Há sempre muito mais versões de um sistema do que releases, uma vez que as versões são criadas dentro de uma organização para o desenvolvimento interno ou testes, e nunca são liberadas para clientes.

Atualmente, o gerenciamento de versão é apoiado por ferramentas CASE, como será discutido na Seção 29.5. Essas ferramentas gerenciam o armazenamento de cada versão do sistema e controlam o acesso aos componentes do sistema. Elas devem ser retiradas do sistema para edição. Quando são inseridas novamente no sistema, uma nova versão é criada e denominada pelo sistema de gerenciamento de versões.

29.3.1 Identificação de versões

Dentro de um grande sistema de software, existem centenas de componentes de software, que podem existir em muitas versões diferentes. Os procedimentos para o gerenciamento de versão devem definir uma maneira não ambígua de identificar cada versão de componente. Versões específicas de componentes podem ser recuperadas, conforme seja necessário, para mudanças futuras.

Há três técnicas básicas que podem ser utilizadas para a identificação de componentes:

1. *Numeração de versões* É atribuído um número explícito e único de versão ao componente. Esse é o esquema de identificação mais comumente utilizado.

2. *Identificação baseada em atributos* Cada componente tem um nome (que não é único em todas as versões) e um conjunto de atributos associados, que difere para cada versão do componente (Estublier e Casallas, 1994). Os componentes são, por conseguinte, identificados pela combinação de nome e conjunto de atributos.

3. *Identificação orientada a mudanças* Cada sistema é denominado, como ocorre na identificação baseada em atributos, mas também é associado com uma ou mais solicitações de mudança (Munch *et al.*, 1993). A versão do sistema é identificada, associando o nome com as mudanças implementadas no componente.

Numeração de versões

Em um esquema simples de numeração de versão, o nome do componente ou do sistema é acrescido de um número de versão. Portanto, podemos falar sobre Solaris 2.6 (a versão 2.6 do sistema Solaris) e sobre a versão 1.4 do componente *getToken*. A primeira versão pode ser chamada de 1.0, as versões subseqüentes, de 1.1, 1.2, e assim por diante. Em algum estágio, um novo release é criado (release 2.0) e o processo começa de novo, na versão 2.1, 2.2 e assim por diante. O esquema é linear, com base na suposição de que as versões de sistema são criadas em seqüência. Muitas ferramentas de gerenciamento de versões (veja a Seção 29.5), como o RCS, apóiam essa abordagem de identificação de versão.

Essa abordagem e a derivação de uma série de diferentes versões do sistema, a partir de versões anteriores, estão ilustradas na Figura 29.6. As setas, nesse diagrama, apontam a partir da versão original para uma nova versão do sistema, que é criada a partir da origem. Observe que a derivação de versões não é necessariamente linear e as versões identificadas por números consecutivos podem ser produzidas a partir de diferentes baselines. Esse exemplo é mostrado na Figura 29.6, em que a versão 2.2 é criada a partir da versão 1.2, e não a partir da versão 2.1. Em princípio, qualquer versão existente pode ser utilizada como ponto de partida para uma nova versão do sistema.

Esse esquema é simples, mas requer um bom gerenciamento das informações associadas, a fim de manter o controle das diferenças entre as versões e as relações entre as mudanças propostas para o sistema e as versões do sistema. Portanto, poderá ser difícil encontrar sistemas e versões de componentes específicos quando são requeridos, especialmente se não houver nenhum vínculo integrado entre o banco de dados da configuração e as versões de sistema armazenadas.

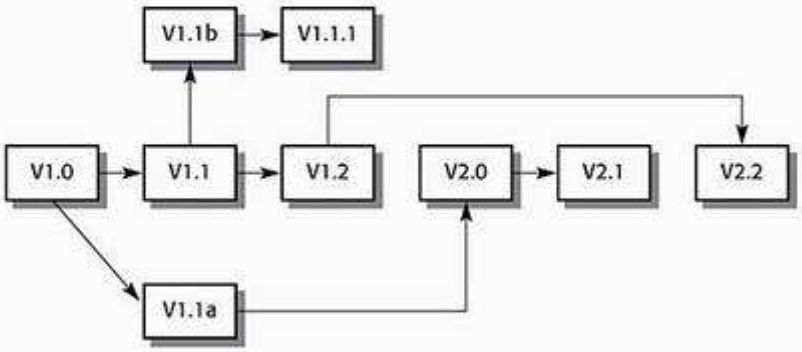
Identificação baseada em atributos

Um problema fundamental com os esquemas explícitos de denominação de versão é que eles não refletem os diferentes atributos que podem ser utilizados para identificar as versões. Exemplos desses atributos identificadores são:

- Cliente.
- Linguagem de desenvolvimento.
- Status de desenvolvimento.
- Plataforma de hardware.
- Data de criação.

Figura 29.6 ▶

Estrutura de derivação de versões.



Se cada versão for identificada por um único conjunto de atributos, será fácil acrescentar novas versões que sejam derivadas de qualquer uma das versões existentes. Elas são identificadas com a utilização de um único conjunto de valores de atributos e compartilham a maioria desses valores com sua versão original, de modo que são mantidas as relações entre as versões. As versões podem ser acessadas mediante a especificação dos valores de atributos requeridos. As funções relacionadas aos atributos apóiam consultas, como por exemplo 'a versão criada mais recentemente' e 'a versão criada entre datas específicas'. Por exemplo, a versão do sistema de software AC3D, desenvolvido em Java para Windows NT, em janeiro de 1999, seria identificada:

AC3D (linguagem = Java, plataforma = NT4, data = jan1999)

A identificação baseada em atributos pode ser implementada diretamente pelo sistema de gerenciamento de versões. Contudo, mais comumente, ela é implementada sobre um esquema de denominação de versão oculta e o banco de dados de configuração mantém os vínculos entre atributos identificadores, sistemas relacionados e versões de componentes.

Identificação orientada a mudanças

A identificação com base em atributos de versões de sistema elimina alguns dos problemas de recuperação de versão, que são encontrados em esquemas de numeração simples. Contudo, para recuperar uma versão, ainda é preciso conhecer seus atributos associados. Além disso, ainda é preciso utilizar um sistema separado de gerenciamento de mudanças, a fim de descobrir as relações entre versões e mudanças.

A identificação orientada a mudanças é utilizada para sistemas, e não para componentes, de modo que as versões de componentes individuais ficam ocultas dos usuários do sistema de CM. Cada mudança de sistema proposta tem um *conjunto de mudanças* associado, que descreve as mudanças feitas nos diferentes componentes de sistema, a fim de implementar aquela mudança. Os conjuntos de mudanças podem ser aplicados em seqüência, de modo que, pelo menos em princípio, pode ser criada uma versão do sistema que incorpore qualquer conjunto arbitrário de mudanças. Portanto, nenhuma identificação explícita de versão é requerida. A equipe de CM interage com o sistema de gerenciamento de versões indiretamente, por meio do sistema de gerenciamento de mudanças.

Na prática, naturalmente, não é possível aplicar conjuntos arbitrários de mudanças em um sistema. Diferentes conjuntos de mudanças podem ser incompatíveis, de maneira que a aplicação do conjunto de mudanças A, seguido pelo conjunto de mudanças D, pode criar um sistema inválido. Além disso, os conjuntos de dados podem ser conflitantes, quando diferentes mudanças afetam o mesmo código do sistema. Para tratar dessas dificuldades, as ferramentas de gerenciamento de versões que apóiam a identificação orientada a mudanças permitem que sejam especificadas regras de consistência de sistema que limitam o modo pelo qual os conjuntos de mudanças podem ser combinados.

29.3.2 Gerenciamento de releases

Um release de sistema é uma versão do sistema que é distribuído para os clientes. Os gerentes de releases de sistemas são responsáveis por decidir quando o sistema pode ser liberado para os clientes, gerenciando o processo de criação de release e o meio de distribuição e documentando o release, a fim de assegurar que ele possa ser recriado exatamente como foi distribuído, se isso for necessário.

Um release de sistema não é apenas o código executável do sistema. O release pode também incluir:

1. *arquivos de configuração* que definem como o release deve ser configurado para instalações específicas;

2. *arquivos de dados*, que são necessários para a operação bem-sucedida do sistema;
3. *um programa de instalação*, que é utilizado para ajudar a instalar o sistema no hardware-alvo;
4. *documentação eletrônica e em papel* que descreva o sistema, e
5. *embalagem e publicidade associada*, que tenham sido projetadas para esse release.

Os gerentes responsáveis pelos releases não podem supor que os clientes sempre vão instalar os novos releases de sistema. Alguns usuários do sistema podem estar satisfeitos com a versão atual ou achar que não vale a pena o custo de migrar para um novo release. Novos releases do sistema não podem, portanto, depender da existência de releases anteriores. Considere a seguinte situação:

1. O release 1 de um sistema é distribuído e colocado em uso.
2. O release 2 é produzido e requer a instalação de novos arquivos de dados, mas alguns clientes não precisam dos recursos do release 2 e, assim, continuam utilizando o release 1.
3. O release 3 requer os arquivos de dados instalados no release 2 e não tem arquivos de dados novos.

O distribuidor de software não pode supor que os arquivos requeridos para o release 3 já tenham sido instalados em todos os locais de instalação. Alguns desses locais podem passar diretamente do release 1 para o release 3, ignorando o release 2. Outros locais podem ter modificado os arquivos de dados associados com o release 2, a fim de refletir as respectivas circunstâncias. Portanto, os arquivos de dados devem ser distribuídos e instalados com o release 3 do sistema.

Tomada de decisão de releases

A preparação e a distribuição de um release de sistema é um processo dispendioso, particularmente para os produtos de software destinados ao mercado massivo. Se os releases forem muito freqüentes, os clientes não poderão fazer a atualização para novos releases; se não forem suficientemente freqüentes, a participação da empresa no mercado pode diminuir, uma vez que os clientes vão procurar sistemas alternativos. Naturalmente, isso não se aplica ao software personalizado, desenvolvido especialmente para uma organização. Contudo, para esse tipo de software, os releases pouco freqüentes podem significar uma divergência crescente entre o software e os processos de negócios para os quais ele foi projetado, a fim de oferecer o devido suporte.

As decisões sobre quando liberar uma nova versão do sistema devem ser regidas por considerações técnicas e organizacionais, como mostra a Figura 29.7.

Criação de releases

A criação de releases é o processo de criar um conjunto de arquivos e documentação que inclua todos os componentes do release do sistema. O código executável dos programas e todos os arquivos de dados associados precisam ser coletados e identificados. As descrições de configuração podem precisar ser escritas para diferentes hardwares e sistemas operacionais e as instruções talvez tenham de ser preparadas para os clientes que precisem configurar seus próprios sistemas. Se forem distribuídos manuais de leitura em computadores, as cópias eletrônicas deverão ser armazenadas com o software. Os scripts para o programa de instalação podem ter de ser escritos. Por fim, quando todas as informações estiverem disponíveis, um disco mestre do release é preparado e entregue para distribuição.

Atualmente, o meio de distribuição normal para o release do sistema são os discos de CD-ROM, que podem armazenar até 600 Mbytes de dados. Assim, muitos produtos de software são também liberados, tornando-se disponíveis na Internet e permitindo aos clientes fazer sua transferência por download. Contudo, muitas pessoas acham que as transferências por

Figura 29.7 ▶

Fatores que influenciam a estratégia de sistema de releases.

Fator	Descrição
Qualidade técnica do sistema	Se forem relatados sérios defeitos nos sistemas, que afetam o modo como muitos clientes utilizam esse sistema, pode ser necessário um release de correção de defeitos. Contudo, os defeitos menores no sistema podem ser corrigidos criando-se *patches* (muitas vezes, distribuídos pela Internet) que podem ser aplicados ao atual release do sistema.
Quinta lei de Lehman (veja o Capítulo 27)	Ela sugere que o aumento de funcionalidade, que é incluído em cada release, seja aproximadamente constante. Portanto, se houver um release de sistema com nova funcionalidade significativa, ele poderá ser seguido de um release de correção.
Competição	Um novo release do sistema pode ser necessário, porque um produto concorrente está disponível.
Requisitos de mercado	O departamento de marketing de uma organização pode ter assumido o compromisso de que os releases estarão disponíveis em uma determinada data.
Propostas de mudanças do cliente	Para os sistemas customizados, os clientes podem ter feito (e pago) um conjunto específico de propostas de mudanças no sistema, e eles esperam um release de sistema, assim que ele for implementado.

download são muito demoradas, quando os arquivos são grandes, e preferem a distribuição por CD-ROM.

Documentação de release

Quando um release de sistema é produzido, ele deve ser documentado, a fim de assegurar que poderá ser recriado com exatidão no futuro. Isso é particularmente importante para os sistemas embutidos personalizados, com longo tempo de vida útil. Os clientes podem utilizar um único release desses sistemas por muitos anos e exigir mudanças específicas para um determinado release de software, muito tempo depois da data de liberação original.

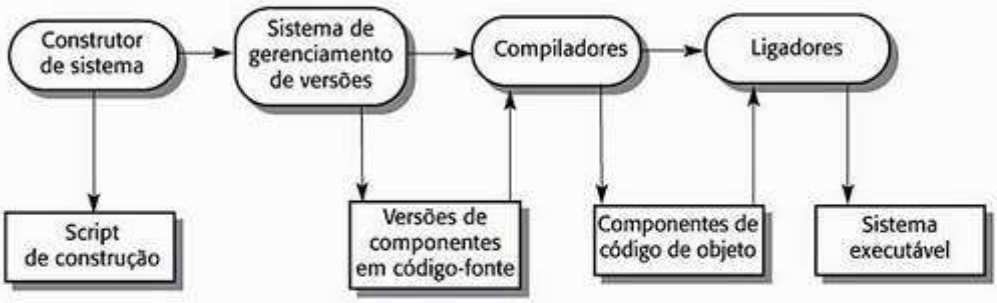
Para documentar um release, é preciso documentar as versões específicas dos componentes do código-fonte, que foram utilizados para criar o código executável. É necessário também manter cópias do código-fonte e do código executável, assim como de todos os dados e arquivos de configuração. Também devem ser registradas as versões do sistema operacional, bibliotecas, compiladores e outras ferramentas utilizadas para construir o software, que podem ser necessários para construir, com exatidão, o mesmo sistema, posteriormente. Nesses casos, as cópias do software de plataforma e das ferramentas podem também ser armazenadas no sistema de gerenciamento de versões.

29.4 Construção de sistemas

A construção do sistema é o processo de compilar e ligar componentes de software em um programa que é executado em uma configuração-alvo específica. Quando se está construindo um sistema a partir de seus componentes, é preciso pensar nas seguintes questões:

1. Todos os componentes que constituem um sistema foram incluídos nas instruções de construção?
2. A versão apropriada de cada componente requerido foi incluída nas instruções de construção?

Figura 29.8
Construção do sistema.



3. Todos os arquivos requeridos estão disponíveis?
4. Se os arquivos de dados são referenciados dentro de um componente, o nome utilizado é o mesmo que o do arquivo de dados na máquina-alvo?
5. A versão apropriada do compilador e outras ferramentas requeridas estão disponíveis? As versões atuais de ferramentas de software podem ser incompatíveis com as versões mais antigas utilizadas para desenvolver o sistema.

Hoje em dia, as ferramentas de CM são utilizadas para automatizar o processo de construção de sistemas. A equipe de CM escreve um script de construção que define as dependências entre os diferentes componentes do sistema. Ela especifica também as ferramentas utilizadas para compilar e ligar os componentes do sistema. A ferramenta de construção do sistema interpreta o script de construção e chama outros programas, conforme seja necessário, a fim de construir o sistema executável. Esse processo está ilustrado na Figura 29.8.

As dependências entre os componentes são especificadas no script de construção, de modo que o sistema de construção de sistemas possa decidir quando os componentes devem ser recompilados e quando os códigos existentes podem ser reutilizados. As dependências de script de construção são mais comumente especificadas como dependências entre os arquivos nos quais os componentes do código-fonte estão armazenados. Contudo, quando existem vários arquivos de código-fonte que representam diferentes versões de componentes, às vezes, não se sabe exatamente quais arquivos-fonte foram utilizados para derivar os componentes orientados a objetos. Essa confusão é particularmente provável quando a correspondência entre os arquivos de código-fonte e de objeto depende de ter o mesmo nome, mas diferentes sufixos (por exemplo, .c e .o).

Para evitar algumas das dificuldades das dependências físicas de arquivos, foram produzidos alguns sistemas experimentais, baseados em linguagens de descrição de módulo (Sommerville e Dean, 1996). Esses sistemas utilizam uma descrição da estrutura lógica do software e um mapeamento para uma estrutura de armazenamento, a fim de inferir as dependências entre os arquivos que contenham componentes de código-fonte. Essa abordagem reduz a possibilidade de erros e leva a descrições mais compreensíveis do processo de construção do sistema.

29.5 Ferramentas CASE para o gerenciamento de configuração

De modo geral, os processos de gerenciamento de configuração são padronizados e envolvem a aplicação de procedimentos predefinidos. Eles exigem o gerenciamento cuidadoso de quantidades de dados muito grandes, e a atenção aos detalhes é essencial. Quando se constrói um sistema a partir de versões de componentes, um único erro de gerenciamento de configuração pode significar que o software não funcionará adequadamente. Como conseqüência, a ferramenta CASE de apoio é essencial para o gerenciamento de configuração e, desde a década

de 70, tem sido produzido um grande número de diferentes ferramentas, que se destinam a diferentes áreas de gerenciamento de configuração.

São exemplos de ferramentas de CM de primeira geração: SCCS (Rochkind, 1975) e RCS (Tichy, 1985), para o controle de revisão, e *make* (Feldman, 1979), para a construção de sistemas. Essas são ferramentas independentes, que lidam com atividades específicas do processo de gerenciamento de configuração. As ferramentas de segunda geração, como Lifespan (Whitgift, 1991) e DSEE (Leblang e Chase, 1987), fornecem algum apoio ao processo de CM integrado, mas não dão apoio a todas as atividades de CM. No momento em que este livro foi escrito, conjuntos de ferramentas CASE integradas estavam disponíveis (Leblang, 1994) e apoiavam o planejamento de configuração, o gerenciamento de mudanças, o gerenciamento de versões e a construção de sistemas. Contudo, esses conjuntos integrados de CM são complexos e dispendiosos, e muitas organizações ainda utilizam o apoio de ferramentas de CM de primeira e segunda gerações.

29.5.1 Apoio ao gerenciamento de mudanças

Cada pessoa envolvida no processo de gerenciamento de mudanças é responsável por alguma atividade. Elas concluem essa atividade e, então, encaminham os formulários e itens associados à configuração para outra pessoa. A natureza de procedimento desse processo significa que um modelo de processo de mudança pode ser projetado e integrado com um sistema de gerenciamento de versões. Esse modelo pode, então, ser interpretado, de modo que os documentos certos sejam fornecidos para o pessoal certo, no momento certo.

As ferramentas de gerenciamento de mudanças podem, por conseguinte, fornecer os seguintes recursos para apoiar o processo:

1. *Um editor de formulários*, que permite a criação e o preenchimento de formulários de solicitação de mudanças.
2. *Um sistema de workflow*, que permite que a equipe de CM especifique as diferentes pessoas que devem processar o formulário de solicitação de mudança e a ordem desse processamento. Esse sistema encaminhará, também automaticamente, os formulários para as pessoas certas, no momento certo, e informará aos membros adequados da equipe sobre o progresso da mudança em questão. O correio eletrônico é utilizado para fornecer atualizações do progresso para as pessoas envolvidas no processo.
3. *Um banco de dados de mudanças*, que é utilizado para gerenciar todas as solicitações de mudança e que pode estar vinculado a um sistema de gerenciamento de versões. De modo geral, são fornecidos recursos de consulta, que permitem que a equipe de CM encontre propostas específicas de mudança.

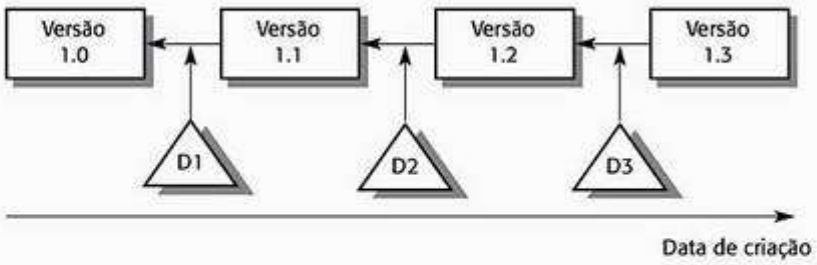
29.5.2 Apoio ao gerenciamento de versões

O gerenciamento de versões envolve gerenciar grandes quantidades de informação e assegurar que as mudanças no sistema sejam registradas e controladas. As ferramentas de gerenciamento de versões controlam um repositório de itens de configuração, cujo conteúdo é imutável (ou seja, ele não pode ser modificado). Para trabalhar em um item de configuração, ele deve ser movido do repositório para algum diretório de trabalho. Depois que o trabalho estiver concluído, ele será inserido novamente no repositório, e uma nova versão será criada de maneira automática.

Todos os sistemas de gerenciamento de versão fornecem um conjunto básico comparável de capacidades, embora alguns deles tenham recursos mais sofisticados do que outros. Exemplos dessas capacidades são:

1. *Identificação de versões e releases* Versões gerenciadas são designadas com identificadores quando são submetidas ao sistema. Diferentes sistemas apóiam diferentes tipos de identificação de versões, discutidos na Seção 29.3.1.

Figura 29.9
Elaboração de versão baseada em delta.



2. *Gerenciamento de armazenamento* Para reduzir o espaço de armazenamento necessário para diferentes versões, que são, em geral, as mesmas, os sistemas de gerenciamento de versões fornecem recursos de gerenciamento de armazenamento, de modo que as versões sejam descritas pelas suas diferenças em relação a uma versão mestre. As diferenças entre versões são representadas por um símbolo *delta*, que encapsula as instruções requeridas para recriar a versão de sistema associada. Esse exemplo está ilustrado na Figura 29.9, que mostra como os deltas retroativos podem ser aplicados à ultima versão de um sistema, a fim de recriar versões anteriores.

3. *Registro de histórico de mudanças* Todas as mudanças feitas no código de um sistema ou componente são registradas e relacionadas. Em alguns sistemas, essas mudanças podem ser utilizadas para selecionar uma determinada versão do sistema.

4. *Desenvolvimento independente* Diferentes versões de um sistema podem ser desenvolvidas em paralelo, e cada versão pode ser modificada independentemente. Por exemplo, o release 1 pode ser modificado depois que o desenvolvimento do release 2 estiver em progresso, pela adição de novos deltas de nível 1. O sistema de gerenciamento de versões mantém o controle dos componentes que foram retirados para edição e assegura que as mudanças feitas ao mesmo componente, por diferentes desenvolvedores, não interfiram. Alguns sistemas somente permitem que uma instância de um componente seja retirada para edição; outros solucionam conflitos em potencial, quando os componentes editados são novamente inseridos no sistema.

29.5.3 Apoio à construção de sistemas

A construção de sistemas é um processo que requer intenso uso de computação. Se todos os componentes de um grande sistema tiverem de ser compilados e ligados, isso poderá demorar várias horas. Pode haver centenas de arquivos envolvidos com a conseqüente possibilidade de erro humano, se eles forem compilados e ligados manualmente. As ferramentas de construção de sistema automatizam o processo de construção, a fim de reduzir a possibilidade de erros humanos e, quando possível, minimizam o tempo necessário para a construção de um sistema.

As ferramentas de construção de sistema podem ser ferramentas *stand-alone*, como as derivadas do utilitário *make*, do Unix (Feldman, 1979), ou podem ser integradas com as ferramentas de gerenciamento de versões. Os recursos oferecidos pelas ferramentas CASE de construção de sistemas podem incluir:

1. *Uma linguagem de especificação de dependência e um interpretador associado* As dependências de componentes podem ser descritas e a recompilação pode ser minimizada. Explico isso com mais detalhes posteriormente nesta seção.

2. *Seleção de ferramentas e apoio à instanciação* Podem ser especificados os compiladores e outras ferramentas de processamento, que são utilizados para processar os arquivos de código-fonte, e as instâncias podem ser produzidas, conforme seja necessário.

3. *Compilação distribuída* Alguns construtores de sistemas, especialmente aqueles que são parte de sistemas de CM integrados, apóiam a compilação distribuída, em rede. Em vez de todas as compilações serem realizadas em uma única máquina, os construtores de sistemas procuram por processadores ociosos na rede e acionam uma série de compilações paralelas. Isso reduz drasticamente o tempo necessário para construir um sistema.

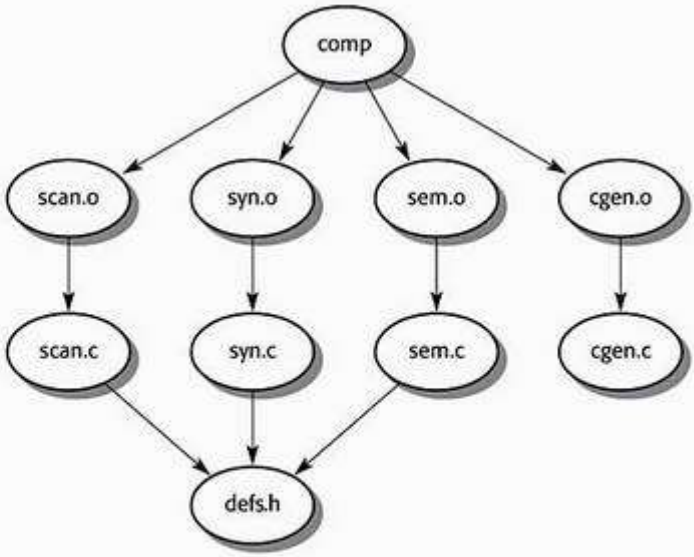
4. *Gerenciamento de objetos derivados* Objetos derivados são objetos criados a partir de outros objetos-fonte. O gerenciamento de objetos derivados vincula o código-fonte e os objetos derivados, e somente torna a derivar um objeto quando isso for solicitado por mudanças no código-fonte.

O gerenciamento de objetos derivados e a minimização da recompilação são mais bem explicados por meio de um exemplo simples. Considere uma situação em que um programa chamado comp é criado a partir dos módulos de objeto scan.o, syn.o, sem.o e cgen.o. Para cada módulo de objeto existe um código-fonte, chamado scan.c, syn.c, sem.c e cgen.c, respectivamente. Um arquivo de declarações, chamado defs.h, é compartilhado por scan.c, syn.c e sem.c (Figura 29.10). Na Figura 29.10, a seta significa 'depende de'. Portanto, comp depende de scan.o, syn.o e sem.o; cgen.o e scan.o dependem de scan.c, e assim por diante.

Se scan.c for modificado, a ferramenta de construção de sistemas poderá detectar que o objeto derivado scan.o deve ser recriado e chama o compilador apropriado para compilar scan.c, a fim de criar um novo objeto derivado, scan.o. Ele, então, utiliza a ligação de dependência entre comp e scan.o para detectar que comp deve ser recriado, ligando scan.o, syn.o, sem.o e cgen.o. O sistema pode detectar que os outros componentes de código de objeto estão inalterados e, assim, a recompilação do código-fonte associado é desnecessária.

Alguns construtores de sistema utilizam a data de modificação do arquivo como o atributo principal para decidir se a recompilação é necessária ou não. Se um arquivo de código-fonte for modificado depois de seu arquivo de código de objeto correspondente, então o arquivo do código de objeto será recriado. Contudo, isso normalmente significa que existe um objeto derivado somente para a mais recente versão do código-fonte. Se uma versão anterior do código-fonte tiver de ser recompilada, sua data de modificação deverá ser ajustada artificialmente, a fim de forçar o construtor de sistema a reconhecer a necessidade de recompilação. Outros sistemas utilizam uma abordagem mais sofisticada para derivar o gerenciamento de objetos. Eles marcam os objetos derivados com o identificador de versão do código-fonte e, dentro dos limites da capacidade de armazenamento, mantêm todos os objetos derivados. Portanto, geralmente, é possível recuperar o código de objeto de todas as versões de componentes de código-fonte, sem recompilação.

Figura 29.10 ▶
Dependências de componentes.



PONTOS-CHAVE

- O gerenciamento de configuração é o gerenciamento de mudanças no sistema. Quando é feita a manutenção de um sistema, o papel da equipe de CM é garantir que as mudanças sejam incorporadas de maneira controlada.
- Em um projeto de grande porte, um esquema formal de denominação de documentos deve ser estabelecido e utilizado como base para manter o controle das diferentes versões de todos os documentos do projeto.
- A equipe de CM deve ser apoiada por um banco de dados de configuração que registra as informações sobre as mudanças no sistema e as solicitações de mudança que sejam importantes. Os projetos devem ter um meio formal de solicitar mudanças no sistema.
- Ao ser definido um esquema de gerenciamento de configuração, um esquema consistente de identificação de versões deve ser estabelecido. As versões podem ser identificadas pelo número da versão, por um conjunto associado de atributos ou pelas mudanças propostas no sistema, que elas implementam.
- Os releases de sistema incluem código executável, arquivos de dados, arquivos de configuração e documentação. O gerenciamento de releases envolve tomar decisões sobre quando liberar um sistema, preparando toda a informação para distribuição e a documentação de cada release do sistema.
- A construção de sistemas é o processo de montagem de componentes do sistema em um programa executável que deverá ser executado em algum sistema computacional alvo.
- As ferramentas CASE estão disponíveis para apoiar todas as atividades de gerenciamento de configuração. Elas incluem ferramentas como RCS, para gerenciar as versões do sistema, ferramentas para apoiar o gerenciamento de mudanças e ferramentas de construção de sistemas.
- As ferramentas CASE para CM podem ser ferramentas *stand-alone*, que apóiam o gerenciamento de mudanças, o gerenciamento de versões e a construção de sistemas, ou podem ser sistemas integrados, que fornecem uma única interface para todo o apoio a CM.

LEITURAS SUGERIDAS

System configuration management. Trata-se do último registro de uma série de workshops sobre o gerenciamento de configuração de software. Ele resume as últimas pesquisas e práticas industriais nessa área. (J. Estublier, 1999, Springer.)

Trends in software: configuration management. Essa é uma coleção de artigos sobre os diferentes aspectos do gerenciamento de configuração, escritos por autores que são pesquisadores ativos e profissionais nessa área. É uma boa introdução para estudantes e profissionais que estejam interessados em uma leitura sobre tópicos avançados de CM. (W. Tichy (ed.), 1995, John Wiley and Sons.)

Implementing configuration management. Esse livro foi escrito sob a perspectiva de um grande fornecedor de sistemas. Discute os problemas de gerenciamento de configuração de sistemas, em que hardware software e *firmware* integrado devem ser controlados. (F.J. Buckley, 1993, IEEE Press.)

EXERCÍCIOS

29.1 Explique por que você não deve utilizar o título de um documento para identificá-lo, em um sistema de gerenciamento de configuração. Sugira um padrão para um esquema de identificação de documentos que possa ser utilizado para todos os projetos em uma organização.

29.2 Utilizando a abordagem de entidade relacional ou orientada a objetos (veja o Capítulo 7), projete um modelo de banco de dados de configuração que registra informações sobre componentes de sistema, versões, releases e mudanças. Alguns requisitos para o modelo de dados são:

- Deve ser possível recuperar todas as versões ou uma única versão identificada de um componente.
- Deve ser possível recuperar a 'mais recente' versão de um componente.
- Deve ser possível descobrir que as solicitações de mudança foram implementadas por uma determinada versão de um sistema.
- Deve ser possível descobrir que as versões de componentes estão incluídas em uma versão específica de um sistema.
- Deve ser possível recuperar um determinado release de um sistema, de acordo com a data de liberação ou o nome dos clientes para os quais o release foi entregue.

29.3 Utilizando um diagrama de fluxo de dados, descreva um procedimento de gerenciamento de mudanças que possa ser utilizado em uma grande organização dedicada a desenvolver software para clientes externos. Podem ser sugeridas mudanças a partir de fontes externas ou internas.

29.4 Descreva as dificuldades que podem ser encontradas na construção de um sistema. Sugira problemas específicos que podem surgir quando um sistema é construído em um computador hospedeiro para alguma máquina-alvo.

29.5 Com referência à construção de sistemas, explique por que, algumas vezes, pode ser necessário manter computadores obsoletos, nos quais grandes sistemas de software foram desenvolvidos.

29.6 Um problema comum com a construção de sistemas ocorre quando nomes de arquivos físicos são incorporados no código do sistema e a respectiva estrutura de arquivos desses nomes difere daquela da máquina-alvo. Escreva um conjunto de diretrizes para programadores que ajude a evitar esse e outros problemas de construção de sistema.

29.7 Descreva os cinco fatores que devem ser levados em consideração pelos engenheiros durante o processo de construção de um release de um grande sistema de software.

29.8 Descreva duas maneiras pelas quais as ferramentas de construção de sistemas podem otimizar o processo de construção de uma versão de um sistema, a partir de seus componentes.

PÁGINA EM BRANCO

Referências

Abbott, R. (1983). "Program design by informal English descriptions". *Comm. ACM*, **26**(11), p. 882–894. (Cap. 12)

Abdel-Ghaly, A.A., Chan, P.Y. *et al.* (1986). "Evaluation of competing software reliability predictions". *IEEE Trans. on Software Engineering*, **SE-12**(9), p. 950–967. (Cap. 21)

Ackroyd, S., Harper, R. *et al.* (1992). *Information technology and practical police work*. Milton Keynes: Open University Press. (Cap. 2)

Adams, E.N. (1984). "Optimizing preventative service of software products". *IBM J. Res & Dev.*, **28**(1), p. 2–14. (Cap. 16)

Aho, A.V., Kernighan, B.W. *et al.* (1988). *The Awk programming language*. Englewood Cliffs, NJ: Prentice Hall. (Caps. 8 e 28)

Albrecht, A.J. (1979). "Measuring application development productivity". *SHARE/GUIDE IBM Application Development Symposium*. (Cap. 23)

Albrecht, A.J. e Gaffney, J.E. (1983). "Software function, lines of code and development effort prediction: a software science validation". *IEEE Trans. on Software Engineering*, **SE-9**(6), p. 639–647. (Cap. 23)

Alexander, C., Ishikawa, S. *et al.* (1977). *A pattern language*. Oxford: Oxford University Press. (Cap. 14)

Alford, M.W. (1977). "A requirements engineering methodology for real time processing requirements". *IEEE Trans. on Software Engineering*, **SE-3**(1), p. 60–69. (Cap. 5)

Alford, M.W. (1985). "SREM at the age of eight: the distributed computing design system". *IEEE Computer*, **18**(4), p. 36–46. (Cap. 5)

Aron, J.D. (1974). *The program development process*. Reading, MA: Addison Wesley. (Cap. 22)

Aron, J.D. (1983). *The program development process: part 2 – the programming team*. Reading, MA: Addison Wesley. (Cap. 4)

Arthur, L.J. (1988). *Software evolution*. Nova York: John Wiley and Sons. (Caps. 27 e 28)

Avizienis, A. (1985). "The N-version approach to fault-tolerant software". *IEEE Trans. on Software Engineering*, **SE-11**(12), p. 1491–1501. (Cap. 18)

Avizienis, A.A. (1995). "A methodology of N-version programming". In: Lyu, M.R. (ed.). *Software fault tolerance*. Chichester: John Wiley and Sons, p. 23–46. (Cap. 18)

Baker, F.T. (1972). "Chief programmer team management of production programming". *IBM Systems J.*, **11**(1), p. 56–73. (Cap. 22)

Baker, T.P. e Scallon, G.M. (1986). "An architecture for real-time software systems". *IEEE Software*, **3**(3), p. 50–58. (Cap. 13)

Banker, R.D., Datar, S.M. *et al.* (1993). "Software complexity and maintenance costs". *Comm. ACM*, **36**(11), p. 81–94. (Cap. 27)

Banker, R., Kauffman, R. *et al.* (1992). "An empirical test of object-based output measurement metrics in a computer-aided software engineering (CASE) environment". *J. of Management Info. Sys.*, **8**(3), p. 127–150. (Cap. 23)

Bansler, J.P. and Bødker, K. (1993). "A reappraisal of structured analysis: design in an organizational context". *ACM Trans. on Information Systems*, **11**(2), p. 165–193. (Cap. 3)

Barker, R. (1989). *CASE* method: entity relationship modelling*. Wokingham: Addison Wesley. (Cap. 7)

Barnard, J. e Price, A. (1994). "Managing code inspection information". *IEEE Software*, **11**(2), p. 59–69. (Caps. 19 e 24)

Barnes, J. (1998). *Programming in Ada 95*. Harlow, Essex: Addison Wesley. (Cap. 13)

Barnes, J.G.P. (1994). *Programming in Ada*. 4. ed. Wokingham: Addison Wesley. (Cap. 13)

Basili, V. e Green, S. (1993). "Software process improvement at the SEL". *IEEE Software*, **11**(4), p. 58–66. (Cap. 25)

Basili, V.R. e Rombach, H.D. (1988). "The TAME project: towards improvementoriented software environments". *IEEE Trans. on Software Engineering*, **14**(6), p. 758–773. (Caps. 24 e 25)

Basili, V.R. e Selby, R.W. (1987). "Comparing the effectiveness of software testing strategies". *IEEE Trans. on Software Engineering*, **SE-13**(12), p. 1278–1296. (Cap. 19)

Bass, B.M. e Dunteman, G. (1963). "Behaviour in groups as a function of self, interaction and task orientation". *J. Abnorm. Soc. Psychology*, **66**(4), p. 19–28. (Cap. 22)

Bass, L., Clements, P. *et al.* (1998). *Software architecture in practice*. Reading, MA: Addison Wesley. (Cap. 10)

Baumer, D., Gryczan, G. *et al.* (1997). "Framework development for large systems". *Comm. ACM*, **40**(10), p. 52–159. (Cap. 14)

Beck, K. (1999). "Embracing change with extreme programming". *IEEE Computer*, **32**(10), p. 70–78. (Cap. 3)

Beck, K. (2000). *Extreme programming explained*. Reading, MA: Addison Wesley Longman. (Caps. 22 e 23)

Beck, K. e Cunningham, W. (1989). "A laboratory for teaching object-oriented thinking". *SIGPLAN Notices*, **24**(10). (Cap. 12)

Beizer, B. (1990). *Software testing techniques*. 2. ed. Nova York: Van Nostrand Rheinhold. (Cap. 20)

Bell, T.E., Bixler, D.C. *et al.* (1977). "An extendable approach to computer aided software requirements engineering". *IEEE Trans. on Software Engineering*, **SE-3**(1), p. 49–60. (Cap. 5)

Bentley, R., Rodden, T. *et al.* (1992). "Ethnographically-informed systems design for air traffic control". *Proc. CSCW'92*, Toronto, Canadá. (Cap. 6)

Bernstein, P.A. (1996). "Middleware: a model for distributed systems services". *Comm. ACM*, **39**(2), p. 86–97. (Cap. 11)

Bersoff, E.H. e Davis, A.M. (1991). "Impact of life cycle models on software configuration management". *Comm. ACM*, **34**(8), p. 104–118. (Cap. 29)

Binder, R.V. (1999). *Testing object-oriented systems: models, patterns and tools*. Reading, MA: Addison Wesley Longman. (Cap. 20)

Bishop, P., Esp, D. *et al.* (1986). "PODS – a project on diverse software". *IEEE Trans. on Software Engineering*, **SE-12**(9), p. 929–940. (Cap. 18)

Boehm, B. (1997). *COCOMO II Model Definition Manual*. Computer Science Dept., University of Southern California. (Cap. 23)

Boehm, B. (1999). "COTS integration: plug and pray?". *IEEE Computer*, **32**(1), p. 135–138. (Cap. 14)

Boehm, B. e Royce, W. (1989). "Ada COCOMO and the Ada Process Model". *Proc. 5th COCOMO Users' Group Meeting*, Pittsburgh, Software Engineering Institute. (Cap. 23)

Boehm, B., Clark, B. *et al.* (1995). "Cost models for future life cycle processes: COCOMO 2". *Annals of Software Engineering*, **1**, p. 57–94. (Caps. 23 e 27)

Boehm, B.W. (1979). "Software engineering: R & D trends and defense needs". In: Wegner, P. (ed.). *Research. Directions in software technology*. Cambridge, MA: MIT Press. (Cap. 19)

Boehm, B.W. (1981). *Software engineering economics*. Englewood Cliffs, NJ: Prentice Hall. (Cap. 23)

Boehm, B.W. (1988). "A spiral model of software development and enhancement". *IEEE Computer*, **21**(5), p. 61–72. (Caps. 3 e 4)

Boehm, B.W., Gray, T.E. *et al.* (1984). "Prototyping versus specifying: a multi-project experiment". *IEEE Transactions on Software Engineering*, **SE-10**(3), p. 290–303. (Cap. 8)

Boehm, B.W., McClean, R.L. *et al.* (1975). "Some experience with automated aids to the design of large-scale reliable software". *IEEE Trans. on Software Engineering*, **SE-1**(1), p. 125–133. (Cap. 16)

Bohm, C. e Jacopini, G. (1966). "Flow diagrams, turing machines and languages with only two formation rules". *Comm. ACM*, **9**(5), p. 366–371. (Cap. 28)

Bollinger, T. e McGowan, C. (1991). "A critical look at software capability evaluations". *IEEE Software*, **8**(4), p. 25–41. (Cap. 25)

Bolognesi, T. e Brinksma, E. (1987). "Introduction to the ISO specification language LOTOS". *Computer Networks*, **14**(1), p. 25–59. (Cap. 9)

Booch, G. (1987). *Software components with Ada: structures tools and subsystems*. Menlo Park, CA: Benjamin Cummings. (Cap. 14)

Booch, G. (1994). *Object-oriented analysis and design with applications*. Redwood City, CA: Benjamin Cummings. (Caps. 1, 3, 6, 7 e 12)

Booch, G., Rumbaugh, J. *et al.* (1999). *The unified modeling language user guide*. Reading, MA: Addison Wesley Longman. (Caps. 1, 3, 7 e 12)

Bourne, S.R. (1978). "The Unix shell". *Bell Sys. Tech. J.*, **57**(6), p. 1971–1990. (Caps. 8 e 14)

Brazendale, J. e Bell, R. (1994). "Safety-related control and protection systems: standards update". *IEE Computing and Control Engineering J.*, **5**(1), p. 6–12. (Cap. 17)

Brilliant, S.S., Knight, J.C. *et al.* (1990). "Analysis of faults in an N-version software experiment". *IEEE Trans. on Software Engineering*, **16**(2), p. 238–247. (Cap. 18)

Brinch-Hansen, P. (1973). *Operating system principles*. Englewood Cliffs, NJ: Prentice Hall. (Cap. 13)

Brooks, F.P. (1975). *The mythical man month*. Reading, MA: Addison Wesley. (Caps. 4 e 22)

Brown, A.W., Earl, A.N. *et al.* (1992). *Software engineering environments*. Londres: McGraw-Hill. (Cap. 10)

Burns, A. (1991). "Scheduling hard real-time systems: a review". *BCS/IEE Software Engineering J.*, **6**(3), p. 116–128. (Cap. 13)

Burns, A. e Wellings, A. (1990). *Real-time systems and their programming languages*. Wokingham: Addison Wesley. (Cap. 13)

Burns, A. e Wellings, A. (1997). *Real-time systems and programming languages*. 2. ed. Harlow, UK: Addison Wesley. (Cap. 13)

Button, G. e Sharrock, W. (1997). "The production of order and the order of production". *Proc. ECSCW'97*, Lancaster, UK, Kluwer. (Cap. 6)

Buxton, J. (1980). *Requirements for Ada programming support environments: stoneman*. Washington D.C: U.S. Department of Defense. (Cap. 10)

Card, S., Moran, T.P. *et al.* (1983). *The Psychology of human-computer interaction*. Hillsdale, NJ: Lawrence Erlbaum Associates. (Cap. 22)

Carroll, J.M. (1992). *The Nurnberg funnel: designing minimalist instruction for practical computer skill*. Boston MA: MIT Press. (Cap. 15)

Checkland, P. (1981). *Systems thinking, systems practice*. Chichester: John Wiley and Sons. (Cap. 2)

Checkland, P. e Scholes, J. (1990). *Soft systems methodology in action*. Chichester: John Wiley and Sons. (Cap. 2)

Chen, P. (1976). "The entity relationship model – towards a unified view of data". *ACM Trans. on Database Systems*, **1**(1), p. 9–36. (Cap. 7)

Chikofsky, E.J. e Cross, J.H. (1990). "Reverse engineering and design recovery: a taxonomy". *IEEE Software*, **7**(1), p. 13–17. (Cap. 28)

Cleveland, L. (1989). "A program understanding support environment". *IBM Sys. J.*, **28**(2), p. 324–344. (Cap. 28)

Coad, P. e Yourdon, E. (1990). *Object-oriented analysis*. Englewood Cliffs, NJ: Prentice Hall. (Caps. 7 e 12)

Cobb, R.H. e Mills, H.D. (1990). "Engineering software under statistical quality control". *IEEE Software*, **7**(6), p. 44–54. (Cap. 19)

Codd, E.F. (1979). "Extending the database relational model to capture more meaning". *ACM Trans. on Database Systems*, **4**(4), p. 397–434. (Cap. 7)

Codenie, W., De Hondt, K. *et al.* (1997). "From custom applications to domain-specific frameworks". *Comm. ACM*, **40**(10), p. 70–77. (Cap. 14)

Cohen, B., Harwood, W.T. *et al.* (1986). *The specification of complex systems*. Wokingham: Addison Wesley. (Cap. 9)

Constantine, L.L. e Yourdon, E. (1979). *Structured design*. Englewood Cliffs, NJ: Prentice Hall. (Caps. 3 e 26)

Cooling, J.E. (1991). *Software design for real-time systems*. Londres: Chapman and Hall. (Cap. 13)

Coulouris, G., Dollimore, J. *et al.* (1994). *Distributed systems: concepts and design*. Wokingham: Addison Wesley. (Cap. 11)

Crosby, P. (1979). *Quality is free*. Nova York: McGraw-Hill. (Cap. 24)

Curtis, B., Hefley, W.E. *et al.* (1995). *Overview of the people capability maturity model*, Software Engineering Institute. (Cap. 22)

Curtis, B., Krasner, H. *et al.* (1988). "A field study of the software design process for large systems". *Comm. ACM*, **31**(11), p. 1268–1287. (Cap. 22)

Cusamano, M. (1989). "The software factory: a historical interpretation". *IEEE Software*, **6**(2), p. 23–30. (Cap. 14)

Dasarthy, B. (1985). "Timing constraints for real-time systems: constructs for expressing them, methods of validating them". *IEEE Trans. on Software Engineering*, **SE-11**(1), p. 80–86. (Cap. 13)

Date, C.J. e Darwen, H. (1997). *A guide to the SQL standard*. 4. ed. Reading, MA: Addison Wesley. (Cap. 8)

Davis, A.M. (1990). *Software requirements: analysis and specification*. Englewood Cliffs, NJ: Prentice Hall. (Cap. 5)

Davis, A.M. (1993). *Software requirements: objects, functions and states*. Englewood Cliffs, NJ: Prentice Hall. (Cap. 5)

Dehbonei, B. e Mejia, F. (1995). "Formal development of safety-critical software systems in railway signalling". In: Hinchey, M. e Bowen, J.P. (eds.). *Applications of formal methods*. Londres, Prentice Hall, p. 227–252. (Cap. 9)

DeMarco, T. (1978). *Structured analysis and system specification*. Nova York: Yourdon Press. (Caps. 1 e 7)

DeMarco, T. e Lister, T. (1985). "Programmer performance and the effects of the workplace". *Proc. 8th Int. Conf. on Software Engineering*, Londres, IEEE Press. (Cap. 22)

DeMarco, T. e Lister, T. (1999). *Peopleware: productive projects and teams*. 2. ed. Nova York: Dorset House. (Cap. 22)

Diaper, D. (1989). *Task analysis for human-computer interaction*. Chichester: Ellis Horwood. (Cap. 15)

Diaz, M. and Sligo, J. (1997). "How software process improvement helped Motorola". *IEEE Software*, **14**(5), p. 75–82. (Cap. 25)

Dijkstra, E.W. (1968a). "Goto statement considered harmful". *Comm. ACM*, **11**(3), p. 147–148. (Cap. 18)

Dijkstra, E.W. (1968b). "Cooperating sequential processes". In: Genuys, F. (ed.). *Programming languages*. Londres: Academic Press, p. 43–112. (Cap. 13)

Diller, A. (1994). *Z: an introduction to formal methods*. 2. ed. Nova York: John Wiley and Sons. (Cap. 9)

Dix, A., Finlay, J. *et al.* (1998). *Human-computer interaction*. 2. ed. Londres: Prentice Hall. (Cap. 15)

Easterbrook, S. (1993). "Domain modelling with hierarchies of alternative viewpoints". *Proc. RE'93*, San Diego, USA. (Cap. 6)

Easterbrook, S. e Nuseibeh, B. (1996). "Using viewpoints for inconsistency management". *BCS/IEE Software Engineering J.*, **11**(1), p. 31–43. (Cap. 6)

Easterbrook, S., Lutz, R. *et al.* (1998). "Experiences using lightweight formal methods for requirements modeling". *IEEE Trans. on Software Engineering*, **24**(1), p. 4–14. (Cap. 9)

Eckstein, R., Loy, M. *et al.* (1998). *Java swing*. Sebastopol, CA: O'Reilly and Associates Inc. (Cap. 15)

ECMA (1991). *A reference model for frameworks of computer-assisted software engineering environments*. Tech. Rept. European Computer Manufacturers Association. (Cap. 10)

Ehrlich, W., Prasanna, B. *et al.* (1993). "Determining the cost of a stop-test decision". *IEEE Software*, **9**(4), p. 33–42. (Cap. 21)

El Amam, K., Drouin, J. *et al.* (1997). *SPICE: the theory and practice of software process improvement and capability determination*. Los Alamitos, CA: IEEE Computer Society Press. (Cap. 25)

Ellison, R.J., Fisher, D.A. *et al.* (1999a). "Survivability: protecting your critical systems". *IEEE Internet Computing* **3**(6), p. 55–63. (Cap. 16)

Ellison, R.J., Linger, R.C. *et al.* (1999b). "Survivable network system analysis: a case study". *IEEE Software*, **16**(4), p. 70–77. (Cap. 16)

Endres, A. (1975). "An analysis of errors and their causes in system programs". *IEEE Trans. on Software Engineering*, **SE-1**(2), p. 140–149. (Cap. 16)

Estublier, J. e Casallas, R. (1994). "The Adele configuration manager". In: Tichy, W. (ed.). *Configuration management*. Chichester: John Wiley and Sons, p. 99–134. (Cap. 29)

Fagan, M.E. (1976). "Design and code inspections to reduce errors in program development". *IBM Systems J.*, **15**(3), p. 182–211. (Cap. 19)

Fagan, M.E. (1986). "Advances in software inspections". *IEEE Trans. on Software Engineering*, **SE-12**(7), p. 744–751. (Cap. 19)

Fayad, M.E. e Schmidt, D.C. (1997). "Object-oriented application frameworks". *Comm. ACM*, **40**(10), p. 32–38. (Cap. 14)

Feldman, S.I. (1979). "MAKE – a program for maintaining computer programs". *Software-Practice and Experience*, **9**(4), p. 255–265. (Cap. 29)

Finkelstein, A., Kramer, J. *et al.* (1990). "Viewpoint oriented software development". *Proc. 3rd Int. Workshop on Software Engineering and its Applications*, Toulouse, France. (Cap. 6)

Forte, G. (1992). "Tools fair: out of the lab, onto the shelf". *IEEE Software*, **9**(3), p. 70–79. (Cap. 8)

Fowler, M. e Scott, K. (1997). *UML distilled: applying the standard object modelling language*. Reading, MA: Addison Wesley. (Caps. 1 e 6)

Frewin, G.D. e Hatton, B.J. (1986). "Quality management – procedures and practises". *IEE/BCS Software Engineering J.*, **1**(1), p. 29–38. (Cap. 19)

Fromme, B. e Walker, J. (1993). "An open architecture for tool and process integration". *Proc. 6th Conf. on Software Engineering Environments*, Reading, UK, IEEE Press. (Cap. 10)

Fuggetta, A. (1993). "A classification of CASE technology". *IEEE Computer*, **26**(12), p. 25–38. (Cap. 3)

Fujiwara, E. e Pradhan, D.K. (1990). "Error-control coding in computers". *IEEE Computer*, **23**(7), p. 63–72. (Cap. 18)

Furey, S. e Kitchenham, B. (1997). "Point/counterpoint: function points". *IEEE Software*, **14**(2), p. 28–31. (Cap. 23)

Futatsugi, K., Goguen, J.A. *et al.* (1985). "Principles of OBJ2". *12th ACM Symp. on Principles of Programming Languages*, Nova Orleans. (Cap. 9)

Gamma, E., Helm, R. *et al.* (1995). *Design patterns: elements of reusable object-oriented software*. Reading, MA: Addison Wesley. (Caps. 10 e 14)

Gane, C. e Sarson, T. (1979). *Structured systems analysis*. Englewood Cliffs, NJ: Prentice Hall. (Cap. 3)

Garlan, D. e Shaw, M. (1993). "An introduction to software architecture", *Advances in Software Engineering and Knowledge Engineering*, **1**, p. 1–26. (Cap. 10)

Garlan, D., Allen, R. *et al.* (1995). "Architectural mismatch: why reuse is so hard". *IEEE Software*, **12**(6), p. 17–26. (Cap. 14)

Garlan, D., Kaiser, G.E. *et al.* (1992). "Using tool abstraction to compose systems". *IEEE Computer*, **25**(6), p. 30–38. (Cap. 10)

Gilb, T. e Graham, D. (1993). *Software inspection*. Wokingham: Addison Wesley. (Cap. 19)

Goldberg, A. e Robson, D. (1983). *Smalltalk-80. The language and its implementation*. Reading, MA: Addison Wesley. (Cap. 15)

Gollmann, D. (1999). *Computer security*. Chichester: John Wiley and Sons. (Cap. 21)

Gomaa, H. (1993). *Software design methods for concurrent and real-time systems*. Reading, MA: Addison Wesley. (Cap. 13)

Gordon, V.S. e Bieman, J.M. (1995). "Rapid prototyping: lessons learned". *IEEE Software*, **12**(1), p. 85–95. (Cap. 8)

Gotterbarn, D., Miller, K. *et al.* (1999). "Software engineering code of ethics is approved". *Comm. ACM*, **42**(10), p. 102–107. (Cap. 1)

Grady, R.B. (1993). "Practical results from measuring software quality". *Comm. ACM*, **36**(11), p. 62–68. (Cap. 24)

Grady, R.B. e Van Slack, T. (1994). "Key lessons in achieving widespread inspection use". *IEEE Software*, **11**(4), p. 46–57. (Cap. 19)

Graham, I. (1994). *Object-oriented methods*. 2. ed. Wokingham: Addison Wesley. (Cap. 12)

Greenbaum, J. e Kyng, M. (1991). *Design at work: cooperative design of computer systems*. Hillsdale, NJ: Lawrence Erlbaum Associates. (Cap. 15)

Griss, M.L. and Wosser, M. (1995). "Making reuse work at Hewlett-Packard". *IEEE Software*, **12**(1), p. 105–107. (Cap. 14)

Grudin, J. (1989). "The case against user interface consistency". *Comm. ACM*, **32**(10), p. 1164–1173. (Cap. 15)

Guimarães, T. (1983). "Managing application program maintenance expenditures". *Comm. ACM*, **26**(10), p. 739–1746. (Cap. 27)

Gunning, R. (1962). *Techniques of clear writing*. Nova York: McGraw-Hill. (Cap. 24)

Guttag, J. (1977). "Abstract data types and the development of data structures". *Comm. ACM*, **20**(6), p. 396–405. (Cap. 9)

Guttag, J.V., Horning, J.J. *et al.* (1985). "The Larch family of specification languages". *IEEE Software*, **2**(5), p. 24–36. (Cap. 9)

Guttag, J., Horning, J. *et al.* (1993). *Larch: languages and tools for formal specification*. Heidelberg: Springer-Verlag. (Cap. 9)

Haase, V., Messnarz, R. *et al.* (1994). "Bootstrap: fine tuning process assessment". *IEEE Software*, **11**(4), p. 25–35. (Cap. 25)

Hall, A. (1990). "Seven myths of formal methods". *IEEE Software*, **7** (5), p. 11–20. (Cap. 9)

Hall, A. (1996). "Using formal methods to develop an ATC information system". *IEEE Software*, **13**(2), p. 66–76. (Caps. 9 e 17)

Hall, E. (1998). *Managing risk: methods for software systems development*. Reading, MA: Addison Wesley Longman. (Cap. 4)

Hall, T. e Fenton, N. (1997). "Implementing effective software metrics programs". *IEEE Software*, **14**(2), p. 55–64. (Cap. 24)

Halstead, M.H. (1977). *Elements of software science*. Amsterdã: North-Holland. (Cap. 27)

Hamlet, D. (1992). "Are we testing for true reliability?". *IEEE Software*, **9**(4), p. 21–27. (Cap. 21)

Hammer, M. (1990). "Reengineering work: don't automate, obliterate". *Harvard Business Review*, jul.–ago. 1990, p. 104–112. (Cap. 28)

Hammer, M. e McLeod, D. (1981). "Database descriptions with SDM: a semantic database model". *ACM Trans. on Database Systems*, **6**(3), p. 351–386. (Cap. 7)

Harel, D. (1987). "Statecharts: a visual formalism for complex systems". *Sci. Comput. Programming*, **8**(3), p. 231–274. (Caps. 7, 12 e 13)

Harel, D. (1988). "On visual formalisms". *Comm. ACM*, **31**(5), p. 514–1530. (Caps. 7 e 13)

Harker, S.D.P., Easton, K.D. *et al.* (1993). "The change and evolution of requirements as a challenge to the practice of software engineering". *Proc. RE'93*, San Diego, USA. (Cap. 6)

Hayes, I. (ed.). (1987). *Specification case studies*. Londres: Prentice Hall. (Cap. 9)

Heath, C. e Luff, P. (1991). "Collaborative activity and technological design: task coordination in the London Underground control room". *Proc. ECSCW'91*, Amsterdã, Kluwer. (Cap. 6)

Heninger, K.L. (1980). "Specifying software requirements for complex systems. New techniques and their applications". *IEEE Trans. on Software Engineering*, **SE-6**(1), p. 2–13. (Cap. 5)

Hihn, J. e Habib-agahi, H. (1991). "Cost estimation of software intensive projects: a survey of current practices". *Proc. 13th Int. Conf. on Software Engineering*, Austin TX, ACM Press. (Cap. 23)

Hoare, C.A.R. (1974). "Monitors: an operating system structuring concept". *Comm. ACM*, 21(8), p. 666–677. (Cap. 13)

Hoare, C.A.R. (1985). *Communicating sequential processes*. Londres: Prentice Hall. (Caps. 8 e 9)

Huff, C. e Martin, C.D. (1995). "Computing consequences: a framework for teaching ethical computing". *Comm. ACM*, **38**(12), p. 75–84. (Cap. 1)

Huff, C.C. (1992). "Elements of a realistic CASE tool adoption budget". *Comm. ACM* **35**(4), p. 45–54. (Cap. 3)

Huff, K.E. (1996). "Software process modeling". In: Fugetta, A. e Wolf, A. (eds.). *Trends in software: software process*. Nova York: John Wiley and Sons. (Cap. 25)

Hughes, J.A., O'Brien, J. *et al.* (1997). "Designing with ethnography: a presentation framework for design". *Proc. DIS'97*, Amsterdã, ACM Press. (Cap. 15)

Hughes, J.A., Randall, D. *et al.* (1992). "Faltering from ethnography to design". *Proc. CSCW'92*, Toronto, Canadá. (Cap. 6)

Hughes, J., Rodden, T. *et al.* (1994). "Moving out from the control room: ethnography in system design". *Proc. CSCW'94*, Greensborough, Carolina do Norte. (Cap. 6)

Hull, R. e King, R. (1987). "Semantic database modeling: survey, applications and research issues". *ACM Computing Surveys*, **19**(3), p. 201–260. (Cap. 7)

Humphrey, W. (1989). *Managing the software process*. Reading, MA: Addison Wesley. (Caps. 24 e 25)

Humphrey, W. e Curtis, B. (1991). "Comment on 'A critical look'". *IEEE Software*, **8**(4), p. 42–47. (Cap. 25)

Humphrey, W.S. (1988). "Characterizing the software process". *IEEE Software*, **5**(2), p. 73–79. (Cap. 25)

Humphrey, W.S. (1995). *A discipline for software engineering*. Reading, MA: Addison Wesley. (Cap. 25)

IEC (1998). *Draft Standard IEC 6150: Functional safety of electrical/electronic/programmable electronic safety-related systems*. International Electrotechnical Commission, Genebra. (Cap. 17)

IEEE (1993). "IEEE recommended practice for software requirements specifications". In: Thayer, R.H. e Dorfman, M. (eds.). *Software requirements engineering*. Los Alamitos, CA: IEEE Computer Society Press. (Cap. 5)

IEEE (1994). *Software Engineering Standards Collection*. Los Alamitos CA: IEEE Press. (Cap. 24)

Ince, D. (1994). *ISO 9000 and software quality assurance*. Londres: McGraw-Hill. (Caps. 24 e 29)

Ince, D.C. e Hekmatpour, S. (1987). "Software prototyping: progress and prospects". *Information and Software Technology*, **29**(1), p. 8–14. (Cap. 8)

ISO/IEC (1998). *Information technology: guidelines for the management of IT security*, ISO/IEC. (Cap. 17)

Jackson, M.A. (1983). *System development*. Londres: Prentice Hall. (Caps. 1 e 3)

Jackson, M.A. (1995). *Requirements and specifications*. Wokingham: Addison Wesley. (Cap. 5)

Jacky, J. (1995). "Specifying a safety-critical control system". *IEEE Trans. on Software Engineering*, **21**(2), p. 99–106. (Cap. 9)

Jacky, J. (1997). *The way of Z: practical programming with formal methods*. Cambridge: Cambridge University Press. (Cap. 9)

Jacky, J., Unger, J. *et al.* (1997). "Experience with Z: developing a control program for a radiation therapy machine". *Proc. ZUM'97*, Reading, Springer. (Cap. 9)

Jacobson, I., Christerson, M. *et al.* (1993). *Object-oriented software engineering.* Wokingham: Addison Wesley. (Caps. 5, 6 e 12)

Jacobson, I., Griss, M. *et al.* (1997). *Software reuse: architecture, process and organization for business success.* Reading, MA: Addison Wesley. (Cap. 14)

Jahanian, F. e Mok, A.K. (1986). "Safety analysis of timing properties in real-time systems". *IEEE Trans. on Software Engineering*, **SE-12**(9), p. 890–904. (Cap. 17)

Janis, I.L. (1972). *Victims of groupthink. A psychological study of foreign policy decisions and fiascos.* Boston: Houghton Mifflin. (Cap. 22)

Jelinski, Z. e Moranda, P.B. (1972). "Software reliability research". In: Frieberger, W. (ed.). *Statistical computer performance evaluation.* Nova York: Academic Press, p. 465–484. (Cap. 21)

Johnson, P.L. (1993). *ISO 9000: meeting the new international standards.* Nova York: McGraw-Hill. (Cap. 24)

Jones, C.B. (1980). *Software development: a rigorous approach.* Londres: Prentice Hall. (Cap. 9)

Jones, C.B. (1986). *Systematic software development using VDM.* Londres: Prentice Hall. (Cap. 9)

Jorgensen, P.C. e Erickson, C. (1994). "Object-oriented integration testing". *Comm. ACM*, **37**(9), p. 30–38. (Cap. 20)

Kafura, D. e Reddy, G.R. (1987). "The use of software complexity metrics in software maintenance". *IEEE Trans. on Software Engineering*, **SE-13**(3), p. 335–343. (Cap. 27)

Kemerer, C.F. (1993). "Reliability of function points measurement: a field experiment". *Comm. ACM*, **36**(2), p. 85–97. (Cap. 23)

Kit, E. (1995). *Software testing in the real world: improving the process.* Reading, MA: Addison Wesley. (Cap. 20)

Kitchenham, B. (1990a). "Measuring software development". In: Rook, P. (ed.). *Software reliability handbook.* Amsterdã: Elsevier, p. 303–331. (Cap. 24)

Kitchenham, B. (1990b). "Software development cost models". In: Rook, P. (ed.). *Software reliability handbook.* Amsterdã: Elsevier, p. 487–517. (Cap. 23)

Knight, J.C. and Leveson, N.G. (1986). "An experimental evaluation of the assumption of independence in multi-version programming". *IEEE Trans. on Software Engineering*, **SE-12**(1), p. 96–109. (Cap. 18)

Knuth, D.E. (1971). *The art of computer programming: fundamental algorithms.* Reading, MA: Addison Wesley. (Cap. 14)

Kotonya, G. e Sommerville, I. (1992). "Viewpoints for requirements definition". *BCS/IEE Software Engineering J.*, **7**(6), p. 375–387. (Cap. 6)

Kotonya, G. e Sommerville, I. (1996). "Requirements engineering with viewpoints". *BCS/IEE Software Engineering J.*, **11**(1), p. 5–18. (Cap. 6)

Kotonya, G. e Sommerville, I. (1998). *Requirements engineering: processes and techniques.* Chichester, UK: John Wiley and Sons. (Caps. 5 e 6)

Kramer, J. e Magee, J. (1985). "Dynamic configuration for distributed systems". *IEEE Trans. on Software Engineering*, **SE-11**(4), p. 424–435. (Cap. 13)

Kuvaja, P., Similä, J. *et al.* (1994). *Software process assessment and improvement: the BOOTSTRAP Approach.* Oxford: Blackwell Publishers. (Cap. 25)

Kyng, M. (1988). "Designing for a dollar a day". *Proc. CSCW'88*, Portland OR, ACM Press. (Cap. 15)

Lamping, J., Rao, R. *et al.* (1995). "A focus + context technique based on hyperbolic geometry for visualising large hierarchies". *Proc. CHI'95*, ACM Press. (Cap. 15)

Lapric, J.-C., Arlat, J. *et al.* (1995). "Architectural issues in software fault tolerance". In: Lyu, M.R. (ed.). *Software fault tolerance.* Chichester: John Wiley and Sons, p. 47–80. (Cap. 18)

Laudon, K. (1995). "Ethical concepts and information technology". *Comm. ACM*, **38**(12), p. 33–39. (Cap. 1)

Leblang, D. (1994). "The CM challenge: configuration management that works". In: Tichy, W. (ed.). *Configuration management.* Chichester: John Wiley and Sons, p. 1–38. (Cap. 29)

Leblang, D.B. e Chase, R.P. (1987). "Parallel software configuration management in a network environment". *IEEE Software*, **4**(6), p. 28–35. (Cap. 29)

Lehman, M.M. e Belady, L. (1985). *Program evolution: processes of software change.* Londres: Academic Press. (Cap. 27)

Leveson, N.G. (1985). "Software safety". In: Anderson, T. (ed.). *Resilient computing systems.* Londres: Collins, p. 123–143. (Caps. 16 e 17)

Leveson, N.G. (1986). "Software safety: why, what and how". *ACM Computing Surveys*, **18**(2), p. 125–163. (Cap. 17)

Leveson, N.G. (1995). *Safeware: system safety and computers.* Reading, MA: Addison Wesley. (Cap. 18)

Leveson, N.G. e Harvey, P.R. (1983). "Analysing software safety". *IEEE Trans. on Software Engineering*, **SE-9**(5), p. 569-579. (Cap. 17)

Lientz, B.P. e Swanson, E.B. (1980). *Software maintenance management*. Reading, MA: Addison Wesley. (Cap. 27)

Linger, R.C. (1994). "Cleanroom process model". *IEEE Software*, **11**(2), p. 50-58. (Caps. 3, 9 e 19)

Liskov, B. e Guttag, J. (1986). *Abstraction and specification in program development*. Cambridge, MA: MIT Press. (Cap. 9)

Littlewood, B. (1990). "Software reliability growth models". In: Rook, P. (ed.). *Software reliability handbook*. Amsterdã: Elsevier, p. 401-412. (Cap. 16)

Littlewood, B. e Verrall, J.L. (1973). "A Bayesian reliability growth model for computer software". *Applied Statistics*, **22**, p. 332-346. (Cap. 21)

Londeix, B. (1987). *Cost estimation for software development*. Wokingham: Addison Wesley. (Cap. 23)

Luqi (1992). "Computer-aided prototyping for a command and control system using CAPS". *IEEE Software*, **9**(1), p. 56-67. (Cap. 8)

Lutz, M. (1996). *Programming Python*. Sebastopol, CA: O'Reilly and Associates. (Cap. 8)

Lutz, R.R. (1993). "Analysing software requirements errors in safety-critical embedded systems". *Proc. RE'93*, San Diego, CA, IEEE. (Caps. 16, 20 e 21)

Lyu, M. (ed.). (1996). *Software reliability engineering*. Nova York: McGraw-Hill. (Cap. 17)

MacDonell, S.G. (1994). "Comparative review of functional complexity assessment methods for effort estimation". *BCS/IEE Software Engineering J.*, **9**(3), p. 107-117. (Cap. 23)

Marshall, J.E. e Heslin, R. (1975). "Boys and girls together: sexual composition and the effect of density on group size and cohesiveness". *J. of Personality and Social Psychology*, **35**(5), p. 952-961. (Cap. 22)

Maslow, A.A. (1954). *Motivation and personality*. Nova York: Harper and Row. (Cap. 22)

Matsumoto, Y. (1984). "Some experience in promoting reusable software: presentation in higher abstract levels". *IEEE. Trans. on Software Engineering*, **SE-10**(5), p. 502-512. (Cap. 14)

McCabe, T.J. (1976). "A complexity measure". *IEEE Trans. on Software Engineering*, **SE-2**(4), p. 308-320. (Caps. 20 e 27)

McCue, G.M. (1978). "IBM's Santa Teresa Laboratory: architectural design for program development". *IBM Systems J.*, **17**(1), p. 4-25. (Cap. 22)

McGuffin, R.W., Elliston, A.E. *et al.* (1979). "CADES: software engineering in practice". *Proc. 4th Int. Conf. on Software Engineering*, Munique, Alemanha. (Cap. 10)

McKee, J.R. (1984). "Maintenance as a function of design". *AFIPS National Computer Conf.*, Las Vegas. (Cap. 27)

Meyer, B. (1999). "On to components". *IEEE Computer*, **32**(1), p. 139-140. (Cap. 14)

Miller, G.A. (1957). "The magical number 7 plus or minus two: some limits on our capacity for processing information". *Psychological Review*, **63**, p. 81-97. (Cap. 22)

Millington, D. e Stapleton, J. (1995). "Special report: developing a RAD Standard". *IEEE Software*, **12**(5), p. 54-56. (Cap. 8)

Mills, H.D., Dyer, M. *et al.* (1987). "Cleanroom software engineering". *IEEE Software* **4**(5), p. 19-25. (Caps. 3, 9, 16 e 19)

Mills, H.D., O'Neill, D. *et al.* (1980). "The management of software engineering". *IBM Sys. J.*, **24**(2), p. 414-477. (Cap. 3)

MOD (1995). *The procurement of safety critical software*. Ed. revisada. UK Ministry of Defence, Procurement Executive. (Caps. 9 e 21)

Mullery, G. (1979). "CORE: a method for controlled requirements specification". *Proc. 4th Int. Conf. on Software Engineering*, Munique, IEEE Press. (Cap. 6)

Mumford, E. (1989). "User participation in a changing environment: why we need it". In: Knight, K. (ed.). *Participation in systems development*. Londres: Kogan Page. (Cap. 2)

Munch, B.P., Larsen, J.-L. *et al.* (1993). "Uniform versioning: the change-oriented model". *Proc. 4th Workshop on Software Configuration Management*, Baltimore, MD. (Cap. 29)

Murphy, G.C., Townsend, P. *et al.* (1994). "Experiences with cluster and class testing". *Comm. ACM*, **37**(9), p. 39-47. (Cap. 20)

Musa, J.D. (1993). "Operational profiles in software reliability engineering". *IEEE Software*, **10**(2), p. 14-32. (Cap. 21)

Musa, J.D. (1998). *Software reliability engineering: more reliable software, faster development and testing*. Nova York: McGraw-Hill. (Cap. 21)

Musa, J.D., Iannino, A. *et al.* (1987). *Software reliability: measurement, prediction, application*. Nova York: McGraw-Hill. (Cap. 21)

Musciano, C. e Kennedy, B. (1998). *HTML: the definitive guide*. Sebastopol, CA: O'Reilly and Associates Inc. (Cap. 15)

Myers, G.J. (1975). *Reliable software through composite design*. Nova York: Petrocelli/Charter. (Cap. 26)

Myers, W. (1989). "Allow plenty of time for large-scale software". *IEEE Software*, **6**(4), p. 92–99. (Cap. 23)

Nakajo, T. e Kume, H. (1991). "A case history analysis of software error-cause relationships". *IEEE Trans. on Software Engineering*, **18**(8), p. 830–838. (Cap. 16)

Neil, M., Ostrolenk, G. *et al.* (1998). "Lessons from using Z to specify a software tool". *IEEE Trans. on Software Engineering*, **24**(1), p. 15–23. (Cap. 9)

Neilsen, J. (1993). *Usability engineering*. Nova York: Academic Press. (Cap. 15)

Nii, H.P. (1986). "Blackboard systems, parts 1 and 2". *AI Magazine*, **7**(3/4), p. 38–53, 62–69. (Cap. 10)

Nilsen, K. (1998). "Adding real-time capabilities to Java". *Comm. ACM*, **41**(6), p. 49–56. (Cap. 13)

Ning, J.Q., Engberts, A. *et al.* (1994). "Automated support for legacy code understanding". *IEEE Software*, **37**(5), p. 50–57. (Cap. 28)

Norman, D.A. e Draper, S.W. (1986). *User-centered system design*. Hillsdale, NJ: Lawrence Erlbaum. (Caps. 8 e 15)

Nosek, J.T. e Palvia, P. (1990). "Software maintenance management: changes in the last decade". *Software Maintenance: Research and Practice*, **2**(3), p. 157–174. (Cap. 27)

Nuseibeh, B., Kramer, J. *et al.* (1994). "A framework for expressing the relationships between multiple views in requirements specifications". *IEEE Trans. on Software Engineering*, **20**(10), p. 760–773. (Cap. 6)

O'Connor, J., Mansour, C. *et al.* (1994). "Reuse in command and control systems". *IEEE Software*, **11**(4), p. 70–79. (Cap. 14)

Offen, R.J. e Jeffrey, R. (1997). "Establishing software measurement programs". *IEEE Software*, **14**(2), p. 45–54. (Cap. 24)

Oman, P.W. e Cook, C.R. (1990). "The book paradigm for improved maintenance". *IEEE Software*, **7**(1), p. 39–45. (Cap. 28)

Orfali, R. e Harkey, D. (1998). *Client/server programming with Java and CORBA*. Nova York: John Wiley and Sons. (Caps. 8, 11 e 14)

Oskarsson, O. e Glass, R.L. (1995). *An ISO 9000 approach to building quality software*. Englewood Cliffs, NJ: Prentice Hall. (Cap. 24)

Ould, M. (1999). *Managing software quality and business risk*. Chichester: John Wiley and Sons. (Cap. 4)

Ousterhout, J. (1994). *TCL and the TK toolkit*. Reading, MA: Addison Wesley. (Cap. 8)

Ousterhout, J. (1998). "Scripting: higher-level programming for the 21st century". *IEEE Computer*, **31**(3), p. 23–30. (Caps. 8 e 14)

Park, R. (1992). *Software size measurement: a framework for counting source statements*, Software Engineering Institute. (Cap. 23)

Parnas, D.L., Van Schouwen, J. *et al.* (1990). "Evaluation of safety-critical software". *Comm. ACM*, **33**(6), p. 636–651. (Caps. 18 e 21)

Paulk, M. (1995). "How ISO 9000 compares with the CMM". *IEEE Software*, **12**(1), p. 74–84. (Cap. 25)

Paulk, M.C. e Konrad, M. (1994). "An overview of ISO's SPICE project". *IEEE Computer*, **27**(4), p. 68–70. (Cap. 25)

Paulk, M.C., Curtis, B. *et al.* (1993). "Capability Maturity Model, version 1.1". *IEEE Software*, **10**(4), p. 18–27. (Caps. 25 e 29)

Peach, R.W. (1996). *The ISO 9000 handbook*. 3. ed. Nova York: Irwin Professional Publishers. (Cap. 24)

Perrow, C. (1984). *Normal accidents: Living with high-risk technology*. Nova York: Basic Books. (Cap. 16)

Perry, W. (1995). *Effective methods for software testing*. Nova York: John Wiley and Sons. (Cap. 20)

Peterson, J.L. (1981). *Petri net theory and the modeling of systems*. Nova York: McGraw-Hill. (Caps. 9 e 17)

Pfaff, G. e ten Hagen, P.J.W. (1985). *Seeheim workshop on user interface management systems*. Heidelberg: Springer-Verlag. (Cap. 15)

Pfleeger, C.P. (1997). *Security in computing*. 2. ed. Englewood Cliffs: Prentice Hall. (Cap. 16)

Pope, A. (1998). *CORBA*. Harlow, UK: Addison Wesley Longman. (Cap. 8)

Preece, J., Rogers, Y. *et al.* (1994). *Human-computer interaction*. Wokingham: Addison Wesley. (Cap. 15)

Pressman, R. S. (1997). *Software engineering: a practitioner's approach*. Nova York: McGraw Hill. (Cap. 1)

Prieto-Díaz, R. e Arango, G. (1991). *Domain analysis and software system modeling*. Los Alamitos CA: IEEE Press. (Cap. 6)

Prowell, S.J., Trammell, C.J. *et al.* (1999). *Cleanroom software engineering: technology and process*. Reading, MA: Addison Wesley Longman. (Caps. 3, 9, 19 e 21)

Pulford, K., Kuntzmann-Combelles, A. *et al.* (1996). *A quantitative approach to software management: the ami handbook*. Wokingham: Addison Wesley. (Caps. 24 e 25)

Putnam, L.H. (1978). "A general empirical solution to the macro software sizing and estimating problem". *IEEE Trans. on Software Engineering*, **SE-4**(3), p. 345–361. (Cap. 23)

Pycock, J. e Bowers, J. (1996). "Getting others to get it right: an ethnography of design work in the fashion industry". *Proc. CSCW'96*, Boston, ACM Press. (Cap. 6)

Randell, B. (1975). "System structure for software fault tolerance". *IEEE Trans. on Software Engineering*, **SE-1**(2), p. 220–232. (Cap. 18)

Randell, B. e Xu, J. (1995). "The evolution of the recovery block concept". In: Lya, M.R. (ed.). *Software fault tolerance*. Chichester: John Wiley and Sons, p. 1–22. (Cap. 18)

Redmill, F. (1998). "IEC 61508: Principles and use in the management of safety". *IEE Computing and Control Engineering J.*, **9**(10), p. 205–213. (Cap. 17)

Reiss, S.P. (1990). "Connecting tools using message passing in the field environment". *IEEE Software*, **7**(4), p. 57–66. (Cap. 10)

Rettig, M. (1994). "Practical programmer: prototyping for tiny fingers". *Comm. ACM*, **37**(4), p. 21–27. (Cap. 8)

Rickets, J.A., DelMonaco, J.C. *et al.* (1993). "Data reengineering for application systems". In: Arnold, R.S. (ed.). *Software reengineering*. Los Alamitos CA: IEEE Press, p. 288–293. (Cap. 28)

Rittel, H. e Webber, M. (1973). "Dilemmas in a general theory of planning". *Policy Sciences*, **4**, p. 155–169. (Cap. 2)

Robinson, P.J. (1992). *Hierarchical object-oriented design*. Englewood Cliffs, NJ: Prentice Hall. (Caps. 3 e 12)

Rochester, J.B. e Douglass, D.P. (1993). "Re-engineering existing systems". In: Arnold, R.S. (ed.). *Software reengineering*. Los Alamitos, CA: IEEE Press, p. 41–51. (Cap. 28)

Rochkind, M.J. (1975). "The source code control system". *IEEE Trans. on Software Engineering*, **SE-1**(4), p. 255–265. (Cap. 29)

Rockwell, R. e Gera, M.H. (1993). "The eureka software factory CoRe: a conceptual reference model for software factories". *Proc. 6th Conf. on Software Engineering Environments*, Reading, UK, IEEE Press. (Cap. 10)

Ross, D.T. (1977). "Structured analysis (SA). A language for communicating ideas". *IEEE Trans. on Software Engineering*, **SE-3**(1), p. 16–34. (Caps. 5 e 6)

Royce, W.W. (1970). "Managing the development of large software systems: concepts and techniques". *Proc. IEEE WESTCON*, Los Angeles, CA. (Cap. 3)

Rubin, K. e Goldberg, A. (1992). "Object behaviour analysis". *Comm. ACM*, **35**(9), p. 48–62. (Cap. 12)

Rumbaugh, J., Blaha, M. *et al.* (1991). *Object-oriented modeling and design*. Englewood Cliffs, NJ: Prentice Hall. (Caps. 1, 3, 6 e 7)

Rumbaugh, J., Jacobson, I. *et al.* (1999a). *The Unified Modeling Language reference manual*. Reading, MA: Addison Wesley. (Caps. 1, 3, 7, 11 e 13)

Rumbaugh, J., Jacobson, I. *et al.* (1999b). *The unified software development process*. Reading, MA: Addison Wesley. (Caps. 1, 3, 7 e 12)

Sackman, H., Erikson, W.J. *et al.* (1968). "Exploratory experimentation studies comparing on-line and off line programming performance". *Comm. ACM*, **11**(1), p. 3–11. (Cap. 23)

Schmidt, D.C. (1997). "Applying design patterns and frameworks to develop object-oriented communications software". In: Saulus, P. (ed.). *Handbook of programming languages, vol. 1*. Nova York: Macmillan Computer Publishing. (Cap. 14)

Schneidewind, N.F. e Keller, T.W. (1992). "Applying reliability models to the space shuttle". *IEEE Software*, **9**(4), p. 28–33. (Cap. 21)

Schoman, K. e Ross D.T. (1977). "Structured analysis for requirements definition". *IEEE Trans. on Software Engineering*, **SE-3**(1), p. 6–15. (Caps. 5 e 6)

Selby, R. (1988). "Empirically analyzing software reuse in a production environment". In: Tracz, W. (ed.). *Software reuse: emerging technology*. Los Alamitos, CA: IEEE Computer Society Press. (Cap. 23)

Selby, R.W., Basili, V.R. *et al.* (1987). "Cleanroom software development: an empirical evaluation". *IEEE Trans. on Software Engineering*, **SE-13**(9), p. 1027–1037. (Caps. 3 e 19)

Sessions, R. (1997). *COM and DCOM: Microsoft's vision for distributed objects*. Nova York: John Wiley and Sons. (Cap. 8)

Sheldon, F.T., Kavi, K.M. *et al.* (1992). "Reliability measurement: from theory to practice". *IEEE Software*, **9**(4), p. 13–20. (Cap. 21)

Shlaer, S. e Mellor, S. (1988). *Object-oriented systems analysis: modeling the world in data*. Englewood Cliffs, NJ: Yourdon Press. (Cap. 12)

Shneiderman, B. (1980). *Software psychology*. Cambridge, MA: Winthrop Publishers Inc. (Cap. 22)

Shneiderman, B. (1992). *Designing the user interface*. 1. ed. Reading, MA: Addison Wesley. (Cap. 22)

Shneiderman, B. (1998). *Designing the user interface*. 3. ed. Reading, MA: Addison Wesley. (Caps. 8 e 15)

Siegel, J. (1998). "OMG overview: CORBA and the OMA in enterprise computing". *Comm. ACM*, **41**(10), p. 37–43. (Cap. 11)

Silberschaltz, A. e Galvin, P. (1998). *Operating system concepts*. 5. ed. Reading, MA: Addison Wesley. (Cap. 13)

Soloway, E., Ehrlich, K. *et al.* (1982). "What do novices know about programming?". In: Badre, A. e Sheneiderman, B. (eds.). *Directions in human-computer interaction* . Norwood, NJ: Ablex Publishing Co. (Cap. 22)

Sommerville, I. e Dean, G. (1996). "PCL: a language for modelling evolving system architectures". *BCS/IEE Software Engineering J.*, **11**(2), p. 111–121. (Cap. 29)

Sommerville, I. e Sawyer, P. (1997). *Requirements engineering: a good practice guide*. Chichester: John Wiley and Sons. (Caps. 8 e 25)

Sommerville, I., Rodden, T. *et al.* (1993). "Integrating ethnography into the requirements engineering process". *Proc. RE'93*, San Diego, CA, IEEE Computer Society Press. (Cap. 6)

Spafford, E. (1989). "The Internet worm: crisis and aftermath". *Comm ACM*, **32**(6), p. 678–687. (Cap. 16)

Spivey, J.M. (1990). "Specifying a real-time kernel". *IEEE Software*, **7**(5), p. 21–28. (Cap. 9)

Spivey, J.M. (1992). *The Z notation: a reference manual*. 2. ed. Londres: Prentice Hall. (Cap. 9)

St. Laurent, S. e Cerami, E. (1999). *Building XML Applications*. Nova York: McGraw-Hill. (Caps. 27 e 28)

Stapleton, J. (1997). *DSDM Dynamic Systems Development Method*. Harlow, UK: Addison Wesley Longman. (Cap. 8)

Suchman, L. (1983). "Office procedures as practical action". *ACM Trans. on Office Information Systems*, **1**(3), p. 320–328. (Caps. 6 e 15)

Swartz, A.J. (1996). "Airport 95: automated baggage system?". *ACM Software Engineering Notes*, **21**(2), p. 79–83. (Cap. 2)

Symons, C.R. (1988). "Function-point analysis: difficulties and improvements". *IEEE Trans. on Software Engineering*, **14**(1), p. 2–11. (Cap. 23)

Szyperski, C. (1998). *Component software: beyond object-oriented programming*. Reading, MA: Addison Wesley. (Cap. 14)

Tanenbaum, A.S. e Woodhull, A.S. (1997). *Operating systems: design and implementation*. 2. ed. Englewood Cliffs, NJ: Prentice Hall. (Cap. 13)

Teichrow, D. e Hershey, E.A. (1977). "PSL/PSA: a computer aided technique for structured documentation and analysis of information processing systems". *IEEE Trans. on Software Engineering*, **SE-3**(1), p. 41–48. (Cap. 5)

Thayer, R.H. e Dorfman, M. (1997). *Software requirements engineering*. 2. ed. Los Alamitos, CA: IEEE Computer Society Press. (Cap. 5)

Tichy, W. (1985). "RCS: a system for version control". *Software Practice and Experience*, **15**(7), p. 637–654. (Cap. 29)

Ulrich, W.M. (1990). "The evolutionary growth of software reengineering and the decade ahead". *American Programmer*, **3**(10), p. 14–20. (Cap. 28)

Wall, L., Christiansen, T. *et al.* (1996). *Programming Perl*. Sebastopol, CA: O'Reilly and Associates. (Cap. 8)

Ward, P. e Mellor, S. (1985). *Structured development for real-time systems*. Englewood Cliffs, NJ: Prentice Hall. (Caps. 7 e 13)

Warren, I. (ed.) (1998). *The renaissance of legacy systems*. Londres: Springer. (Caps. 26 e 27)

Weinberg, G. (1971). *The psychology of computer programming*. Nova York: Van Nostrand. (Cap. 22)

White, S., Alford, M. *et al.* (1993). "Systems engineering of computer-based systems". *IEEE Computer*, **26**(11), p. 54–65. (Cap. 2)

Whitgift, D. (1991). *Software configuration management: methods and tools*. Chichester: John Wiley and Sons. (Cap. 29)

Wirfs-Brock, R.J. e Johnson, R.E. (1990). "Surveying current research in object-oriented design". *Comm. ACM*, **33**(9), p. 104–124. (Cap. 14)

Wirfs-Brock, R., Wilkerson, B. *et al.* (1990). *Designing object-oriented software*. Englewood Cliffs, NJ: Prentice Hall. (Cap. 12)

Wirth, N. (1976). *Systematic programming: an introduction*. Englewood Cliffs, NJ: Prentice Hall. (Cap. 26)

Wood, J. e Silver, D. (1995). *Joint application development*. 2 ed. Nova York: John Wiley and Sons. (Cap. 8)

Woodcock, J.C.P. e Davies, J. (1996). *Using Z: specification, proof, refinement*. Londres: Prentice Hall. (Cap. 9)

Wordsworth, J. (1996). *Software engineering with B*. Wokingham: Addison Wesley. (Caps. 3, 9 e 17)

Wordsworth, J.B. (1991). "The CICS application programming interface definition". *Proc. Z User Workshop*, Oxford, Berlin: Springer. (Cap. 9)

Zave, P. (1989). "A compositional approach to multiparadigm programming". *IEEE Software*, **6**(5), p. 15–27. (Cap. 8)

Zave, P. e Schell, W. (1986). "Salient features of an executable specification language and its environment". *IEEE Trans. on Software Engineering*, **SE-12**(2), p. 312–325. (Cap. 8)

Zimmermann, H. (1980). "OSI reference model: the ISO model of architecture for open systems interconnection". *IEEE Transactions on Communications*, **COM-28**(4), p. 425–432. (Cap. 10)