Parte 1

Introdução à Engenharia de Software

Os nove primeiros capítulos deste livro desempenham um duplo papel. Eles apresentam ao leitor a engenharia de software e fornecem a base para o material da segunda metade do livro, na qual são descritos os fluxos de trabalho (atividades) do desenvolvimento de software.

O processo de software é a maneira pela qual um software é produzido. Ele começa com a exploração de conceitos e termina quando o produto é finalmente retirado de ação. Durante esse período, o produto passa por uma série de etapas como levantamento de necessidades e requisitos, análise (especificação), projeto, implementação, integração, manutenção pós-entrega e, finalmente, retirada do produto. O processo de software inclui as ferramentas e técnicas que são usadas para desenvolver e manter o software bem como os profissionais envolvidos.

O Capítulo 1, "O Escopo da Engenharia de Software", destaca quais técnicas são eficazes para produção de software em termos de custo e para promover a interação construtiva entre os membros da equipe de produção de software. A importância dos objetos é enfatizada ao longo do livro, iniciando por este capítulo.

Uma série de modelos de ciclo de vida diferentes é discutida detalhadamente no Capítulo 2, "Modelos de Ciclo de Vida de Software". Entre esses há o modelo de árvore de evolução, o modelo cascata, o modelo de prototipagem rápida, o modelo sincronizar e estabilizar, o modelo com software aberto, o modelo de processos ágeis, o modelo espiral e, acima de tudo, o modelo de ciclo de vida iterativo e incremental. Para possibilitar que o leitor decida sobre um modelo de ciclo de vida apropriado para um determinado projeto, os vários modelos de ciclo de vida são comparados e contrastados.

"O Processo de Software" é o título do Capítulo 3. A ênfase nesse capítulo é no Processo Unificado, atualmente a forma mais promissora de se desenvolver software. Os processos ágeis, uma abordagem alternativa ao desenvolvimento de software e que está crescendo em termos de popularidade, também são apresentados em detalhe. O capítulo termina abordando o aperfeiçoamento do processo de software.

O Capítulo 4 é intitulado "Equipes". Os projetos de hoje são muito grandes para serem completados por um único indivíduo dentro das limitações dadas. Em vez disso, uma equipe de profissionais de software colabora no projeto. O principal tópico desse capítulo é a questão de como as equipes deveriam ser organizadas de modo que os membros da equipe

trabalhem juntos de modo produtivo. Várias formas de organizar equipes são discutidas, entre as quais as equipes democráticas, equipes com programador-chefe, equipes sincronizar e estabilizar, equipes para código-fonte aberto e equipes para processos ágeis.

Um engenheiro de software precisa ser capaz de usar uma série de ferramentas diferentes, tanto analíticas quanto práticas. No Capítulo 5, "As Ferramentas de Trabalho", o leitor é apresentado a uma série de ferramentas de engenharia de software. Uma dessas é o refinamento gradual, uma técnica para decompor um grande problema em uma seqüência de problemas menores e mais fáceis de serem tratados. Outra ferramenta é a análise de custo–benefício, uma técnica para determinar se um projeto de software é viável financeiramente. Em seguida, são descritas as ferramentas CASE (Computer-Aided Software Engineering, engenharia de software com o auxílio do computador). CASE é um produto que ajuda os engenheiros de software a desenvolverem e manterem seu software. Finalmente, para gerenciar o processo de software se faz necessário medir várias quantidades para determinar se o projeto tem bom andamento. Tais medidas (métricas) são críticas para o sucesso de um projeto.

Os dois últimos tópicos do Capítulo 5, CASE e métrica, são tratados detalhadamente nos Capítulos de 10 a 15, que descrevem os fluxos de trabalho específicos do ciclo de vida de software. Há uma discussão das ferramentas CASE que suportam cada fluxo de trabalho, bem como uma descrição da métrica necessária para gerenciar esses fluxos de trabalho adequadamente.

O Capítulo 6, "Testes", discute os conceitos que estão por trás dos testes. A consideração de técnicas específicas de teste a cada fluxo de trabalho do ciclo de vida do software é postergada para os Capítulos de 10 a 15.

O Capítulo 7, "De Módulos a Objetos", explica, de forma detalhada, as classes e os objetos, e por que o paradigma de orientação a objetos está se provando ser mais bem-sucedido que o paradigma clássico. Os conceitos desse capítulo são utilizados no restante do livro, particularmente no Capítulo 10 ("Levantamento de Necessidades"), no Capítulo 12 ("Análise Orientada a Objetos") e no Capítulo 13 ("Projeto"), no qual é apresentado o projeto orientado a objetos.

As idéias do Capítulo 7 são estendidas ao Capítulo 8, "Reusabilidade e Portabilidade". É importante ser capaz de escrever software reutilizável e que pode ser portado para uma ampla gama de equipamentos diferentes. A primeira parte do capítulo é dedicada à reutilização; entre os tópicos, há uma série de estudos de caso de reutilização e também estratégias como estruturas e padrões orientados a objetos. Portabilidade é o segundo tópico importante; estratégias de portabilidade são apresentadas com certa profundidade. Um tema recorrente do capítulo é o papel dos objetos para se alcançar a reusabilidade e a portabilidade.

O último capítulo da Parte 1 é o Capítulo 9, "Planejamento e Estimativas". Antes de iniciar um projeto de software, é essencial planejar a operação toda em detalhe. Assim que o projeto é iniciado, a gerência deve monitorar de perto seu progresso, percebendo desvios em relação ao plano original e tomando medidas corretivas quando necessário. Da mesma forma, é vital que sejam fornecidas ao cliente estimativas precisas de quanto tempo o projeto vai durar e quanto ele custará. São mostradas diferentes técnicas de estimativa, inclusive pontos de função COCOMO II. É apresentada uma descrição detalhada de um plano de gerenciamento de projeto de software. O material desse capítulo é utilizado nos Capítulos 11 e 12. Quando o paradigma clássico é usado, as principais atividades de planejamento e de estimativas ocorrem no final da fase de análise clássica, conforme explicado no Capítulo 11. Quando o software é desenvolvido usando-se o paradigma de orientação a objetos, esse planejamento ocorre no final do fluxo de trabalho da análise orientada a objetos (Capítulo 12).

Capítulo 1

O Escopo da Engenharia de Software

Objetivos de Aprendizagem

Após estudar este capítulo, você deverá ser capaz de:

- Definir o que se entende por engenharia de software.
- Descrever o modelo clássico de ciclo de vida da engenharia de software.
- Explicar por que o paradigma de orientação a objetos é agora tão amplamente aceito.
- Discutir as implicações dos vários aspectos da engenharia de software.
- Distinguir entre a visão clássica e a visão moderna da manutenção.
- Discutir a importância de fazer planejamento, testes e documentação continuamente.
- Avaliar a importância de aderir a um código de ética.

Uma história bem conhecida fala de um executivo que recebeu uma conta gerada por computador no valor de $0,00. Após uma boa gargalhada com os amigos por causa dos "computadores idiotas", o executivo jogou a conta fora. Um mês depois, chegou outra conta similar, dessa vez informando que se passaram 30 dias do prazo. Depois, chegou a terceira conta. A quarta conta chegou um mês depois, acompanhada de uma mensagem que insinuava que haveria a possibilidade de ele ser acionado judicialmente caso a conta de $0,00 não fosse paga.

A quinta conta, que indicava haver se passado 120 dias do prazo, não insinuava nada — o aviso era grosseiro e direto e fazia ameaças com todo tipo de ações judiciais, caso a conta não fosse paga imediatamente. Temeroso de ter a classificação do crédito de sua organização nas mãos de uma máquina maníaca, o executivo ligou para um conhecido, que era engenheiro de software, e contou toda a triste história para ele. Tentando não rir, o engenheiro de software disse para o executivo enviar pelo correio um cheque no valor de $0,00. A ação teve o efeito desejado, pois, poucos dias depois, chegou um recibo no valor de $0,00. O executivo arquivou meticulosamente o recibo, para o caso de, no futuro, o computador alegar que a conta no valor de $0,00 ainda fosse devida.

Essa história bem conhecida teve uma conseqüência não tão bem conhecida. Poucos dias depois, o executivo foi intimado pelo gerente de seu banco que, com um cheque na mão, perguntou-lhe: "Por acaso esse cheque é seu?".

O executivo respondeu que sim.

"Você se importaria em me dizer por que preencheu um cheque no valor de $0,00?", perguntou o gerente.

Então, toda a longa história foi recontada. Quando o executivo terminou de falar, o gerente do banco voltou-se para ele e lhe perguntou de forma tranqüila: "Você tem noção do que o seu cheque de $0,00 causou ao *nosso* sistema de computadores?".

Um profissional de informática poderia rir dessa história, se bem que com certo nervosismo. Afinal de contas, cada um de nós já desenvolveu ou implementou um produto que, em sua forma original, teria resultado em algo equivalente a remeter contas estúpidas que cobram $0,00. Até agora, sempre detectamos esse tipo de falha durante a fase de testes. Mas nossas gargalhadas soam falsas, pois, lá no fundo de nossos pensamentos, existe o medo de que algum dia não detectemos a falha antes de o produto ser entregue ao cliente.

Uma falha de software certamente menos divertida foi detectada em 9 de novembro de 1979. O Comando Estratégico da Força Aérea norte-americana entrou caoticamente em estado de alerta quando a rede de computadores do sistema mundial de controle e comando militar (WWMCCS, sigla para worldwide military command and control system) informou que a União Soviética havia lançado mísseis contra os Estados Unidos (Neumann, 1980)0. O que realmente aconteceu foi que um ataque simulado foi interpretado como real, da mesma forma que ocorreria no filme *Jogos de Guerra,* cerca de cinco anos depois. Embora, compreensivelmente, o Ministério de Defesa dos Estados Unidos não tenha dado detalhes sobre qual foi precisamente o mecanismo por meio do qual as informações sobre o teste foram interpretadas como dados reais, parece razoável atribuir o problema a uma falha de software. O sistema como um todo não foi projetado para diferenciar situações reais daquelas simuladas, ou então a interface com o usuário não possuía as verificações necessárias para garantir que os usuários finais do sistema fossem capazes de estabelecer a diferença entre realidade e ficção. Em outras palavras, uma falha de software – se realmente o problema foi causado pelo software – poderia ter levado a civilização, como a conhecemos hoje, a um abrupto e trágico fim. (Veja a Seção *Vale a pena saber*, no Quadro 1.1, para mais informações sobre desastres causados por outras falhas de software.)

Independentemente de estarmos tratando de cobrança de contas ou de defesa aérea, grande parte de nosso software é entregue depois do prazo, acima do orçamento previsto e com falhas residuais, e não atende às necessidades dos clientes. A engenharia de software é uma tentativa de solucionar esses problemas. Ou seja, a **engenharia de software** é uma disciplina cujo objetivo é produzir software isento de falhas, entregue dentro de prazo e orçamento previstos, e que atenda às necessidades do cliente. Além disso, o software deve ser fácil de ser modificado quando as necessidades do usuário mudarem.

O campo de ação da engenharia de software é extremamente amplo. Alguns aspectos da engenharia de software podem ser classificados como matemáticos ou de ciência da computação; outros aspectos caem nas áreas da economia, administração ou psicologia. Para mostrar o abrangente alcance da engenharia de software, examinaremos agora cinco aspectos distintos.

1.1 Aspectos Históricos

É fato que os geradores de energia elétrica falham, porém com uma freqüência muito menor do que a dos programas de folha de pagamento. As pontes de vez em quando caem, mas com freqüência consideravelmente menor do que a dos sistemas operacionais. Acreditando que o projeto, a implementação e a manutenção de software possam ser colocados no mesmo pa-

No caso da rede WWMCCS, a catástrofe foi evitada no último momento. Entretanto, as conseqüências de outras falhas de software foram fatais. Por exemplo, entre 1985 e 1987, pelo menos dois pacientes morreram por conseqüência de graves *overdoses* de radiação produzidas pelo acelerador linear Therac-25, de uso médico (Leveson e Turner, 1993). A causa foi uma falha no software de controle.

Da mesma forma, na Guerra do Golfo, em 1991, um míssil Scud passou pelo escudo antimíssil Patriot e atingiu um acampamento militar próximo de Dhahran, na Arábia Saudita. Ao todo, morreram 28 soldados americanos e 98 ficaram feridos. O software para o míssil Patriot continha uma falha de contagem de tempo acumulativa. O Patriot havia sido projetado para operar apenas por algumas horas por vez, após o que o contador de tempo era reiniciado. Como resultado, a falha jamais havia tido um efeito significativo e, conseqüentemente, não fora detectada. Na Guerra do Golfo, entretanto, a bateria de mísseis Patriot em Dhahran ficou operando por mais de 100 horas ininterruptas. Isso fez com que a discrepância de tempo acumulado se tornasse suficientemente grande para fazer com que o sistema se tornasse impreciso.

Durante a Guerra do Golfo, os Estados Unidos enviaram mísseis Patriot a Israel para proteção contra mísseis Scud. As forças israelenses detectaram o problema da contagem de tempo apenas após 8 horas e relataram o problema imediatamente para o fabricante nos Estados Unidos. O fabricante corrigiu a falha o mais rápido possível, porém, tragicamente, o novo software chegou somente um dia depois de o acampamento ter sido atingido pelo Scud (Mellor, 1994).

Felizmente, são extremamente raras as mortes ou os danos graves produzidos por uma falha de software. Entretanto, uma falha pode causar problemas sérios para milhares e milhares de pessoas. Por exemplo, em fevereiro de 2003, uma falha de software resultou na remessa de 50 mil cheques para pagamento de aposentadoria, por parte do Ministério da Fazenda dos Estados Unidos, que foram impressos sem o nome do beneficiário, de modo que os cheques não podiam ser depositados nem sacados *St. Petersburg Times Online*, 2003. Em abril de 2003, muitos financiados foram informados pela SLM Corp. (comumente conhecida como Sallie Mae) que os juros sobre os empréstimos para estudantes haviam sido calculados de maneira errada como conseqüência de uma falha de software, falha essa que ocorria desde 1992, mas que fora detectada apenas no final de 2002. Cerca de um milhão de financiados foram informados de que teriam de pagar mais, fosse na forma de mensalidades mais altas ou na forma de pagamentos de juros adicionais sobre empréstimos, que se estenderiam além do prazo original de dez anos GJSentinel.com, 2003. Ambas as falhas foram corrigidas rapidamente, porém, juntas, elas provocaram perdas financeiras significativas para cerca de um milhão de pessoas.

tamar que as disciplinas de engenharia tradicionais, um grupo de estudos da OTAN cunhou, em 1967, o termo *engenharia de software*. A alegação de que criar software é similar a outras tarefas da engenharia foi endossada pela Conferência de Engenharia de Software da OTAN, realizada em Garmisch, na Alemanha (Naur, Randell e Buxton, 1976). Esse endosso não é nenhuma surpresa; o próprio nome da conferência reflete a crença de que a produção de software deve ser uma atividade semelhante à engenharia (entretanto, veja o Quadro 1.2). Uma conclusão a que chegaram os conferencistas foi de que a engenharia de software deveria usar as filosofias e os paradigmas das disciplinas de engenharia já estabelecidas para resolver o que eles denominaram **crises de software**, ou seja, que a qualidade do software geralmente era inaceitavelmente baixa, e que os prazos e orçamentos previstos não estavam sendo cumpridos.

Apesar dos vários casos de sucesso, uma quantidade inaceitavelmente grande de produtos de software ainda está sendo entregue com atraso, acima do orçamento e com falhas residuais. Por exemplo, a Standish Group é uma empresa de pesquisa que analisa projetos de desenvolvimento de software. A Figura 1.1 mostra a síntese de um estudo realizado por essa empresa sobre 9.236 projetos de desenvolvimento concluídos em 2004 (Hayes, 2004). Apenas 29% dos projetos foram finalizados com sucesso, ao passo que 18% foram cancelados

Vale a pena saber — Quadro 1.2

Conforme afirmado no item 1.1, o objetivo da conferência de Garmisch foi fazer com que o desenvolvimento de software fosse tão bem-sucedido quanto a engenharia tradicional. Mas, de forma alguma, nem todos os projetos da engenharia tradicional são bem-sucedidos. Consideremos, por exemplo, a construção de uma ponte.

Em julho de 1940, foi finalizada a construção de uma ponte suspensa sobre o estreito de Tacoma, no estado de Washington, nos Estados Unidos. Pouco depois, foi descoberto que a ponte oscilava e curvava-se perigosamente quando ocorriam ventanias. Carros que circulavam pela ponte desapareciam em vales e, em seguida, reapareciam quando esse trecho da ponte subia novamente. Por causa desse comportamento, a ponte recebeu o apelido de "Cavalinho galopante" (em inglês: *Galloping Gertie*). Finalmente, em 7 de novembro de 1940, a ponte ruiu quando ocorreram ventos de 67 quilômetros por hora; felizmente, a ponte havia sido fechada para o tráfego algumas horas antes. Os 15 minutos finais de sua vida foram filmados e estão guardados no U.S. National Film Registry.

Uma falha, um pouco mais engraçada, na construção de uma ponte foi observada em janeiro de 2004. Estava sendo construída uma ponte sobre o Alto Reno, próximo à cidade alemã de Laufenberg, para interligar a Alemanha à Suíça. A parte alemã da ponte foi projetada e construída por uma equipe de engenheiros alemães, ao passo que o trecho suíço, por uma equipe suíça. Quando as duas partes foram interligadas, à primeira vista notou-se que o trecho do lado alemão era, aproximadamente, 21 polegadas (54 centímetros) mais alto que o trecho suíço. Foram necessárias intervenções importantes para reconstruir e corrigir o problema, causado por diferentes interpretações sobre o "nível do mar", que, para os engenheiros suíços, era o nível médio do Mar Mediterrâneo, enquanto, para os engenheiros alemães, a base era o Mar do Norte. Para compensar tal diferença nos níveis do mar, o lado suíço deveria ter sido elevado em 27 centímetros. Em vez disso, ele foi rebaixado 27 centímetros, resultando na diferença de 54 centímetros (Spiegel Online, 2004).

antes do término ou jamais foram implementados. Os 53% restantes foram terminados e instalados nos computadores dos clientes. Entretanto, esses projetos estavam fora do orçamento previsto, atrasados ou tinham um número menor de recursos e de funcionalidades em relação à especificação inicial. Em outras palavras, em 2004, menos de um terço dos projetos de desenvolvimento de software foi bem-sucedido; mais da metade dos projetos apresentaram um ou mais sintomas da crise de software.

As implicações financeiras da crise de software são terríveis. Em uma pesquisa realizada pelo Cutter Consortium (2002), foi constatado o seguinte:

- Uma estarrecedora parcela de 78% das empresas de TI envolveram-se em disputas que acabaram em ações judiciais.
- Em 67% desses casos, a funcionalidade ou o desempenho dos produtos de software como foram entregues não atendiam à funcionalidade alegada pelos desenvolvedores de software.

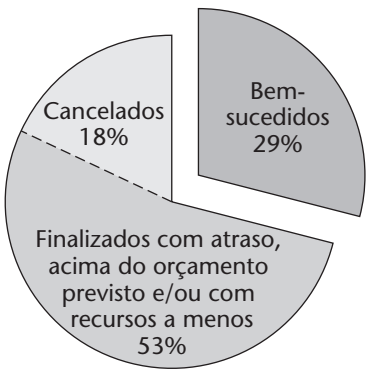


FIGURA 1.1
Resultados de mais de 9 mil projetos de desenvolvimento de software concluídos em 2004.

Fonte: (Hayes, 2004)

- Em 56% desses casos, a data de entrega prometida não foi cumprida várias vezes.
- Em 45% desses casos, as falhas foram tão graves que o produto de software não era utilizável.

Fica claro que uma parcela incrivelmente baixa dos sistemas de software é entregue dentro do prazo e do orçamento previstos, sem falhas e atendendo às necessidades do cliente. Para atingir essas metas, um engenheiro de software tem de adquirir uma ampla gama de habilidades, tanto técnicas quanto gerenciais. Essas habilidades devem ser aplicadas não apenas à programação, mas em cada etapa da produção de software, desde a fase de levantamento de necessidades até a de manutenção pós-entrega do sistema.

O fato de, cerca de 40 anos depois, a crise de software existir entre nós revela duas coisas. Primeiro, que o processo de produção de software, embora se assemelhe em muitos aspectos à engenharia tradicional, possui características e problemas particulares. Em segundo lugar, o termo crise de software talvez tenha de ser rebatizado para **depressão de software**, dada a sua longa duração e seus prognósticos ruins.

Consideraremos agora os aspectos econômicos da engenharia de software.

1.2 Aspectos Econômicos

Uma empresa de software que usa atualmente a técnica de codificação TC_{antiga} descobre que a utilização de uma nova técnica de codificação TC_{nova} poderia resultar na produção de código em apenas nove décimos do tempo utilizado pela técnica TC_{antiga}, e, conseqüentemente, com nove décimos do custo. O bom senso parece indicar que a TC_{nova} seria a técnica apropriada a ser adotada. Na realidade, embora o bom senso certamente dite que a técnica mais rápida é a técnica a ser escolhida, a economia da engenharia de software pode implicar o contrário.

- Uma razão para isso seria o custo de introduzir uma tecnologia nova em uma organização. O fato de a codificação ser 10% mais rápida quando a técnica TC_{nova} é empregada talvez seja menos importante que os custos implicados pela introdução da técnica TC_{nova} na organização. Talvez seja necessário completar dois ou três projetos antes de recuperar o investimento no treinamento. Da mesma forma, enquanto estiver em treinamento nessa nova técnica, o pessoal de software não poderá dedicar-se a um trabalho produtivo. Mesmo após retornarem, uma curva de aprendizagem pronunciada pode estar envolvida nesse processo, e talvez exija meses de prática na TC_{nova} antes de os profissionais da área de software atingirem o mesmo nível de competência com a TC_{nova} que já possuem com a TC_{antiga}. Portanto, os primeiros projetos que utilizarem a TC_{nova} podem levar muito mais tempo para ser finalizados do que se a empresa continuasse a usar a TC_{antiga}. Todos esses custos devem ser levados em consideração no momento de decidir se é adequado ou não mudar para a TC_{nova}.
- Uma segunda razão para a economia da engenharia de software poder eventualmente indicar que a TC_{antiga} deva ser mantida é a questão da manutenção. A técnica de codificação da TC_{nova} pode ser, de fato, até 10% mais rápida do que a da TC_{antiga}, bem como o código resultante pode ter qualidade comparável em termos de atendimento às necessidades atuais do cliente. Porém, o uso da técnica TC_{nova} pode resultar em código de difícil manutenção, tornando o custo da TC_{nova} mais alto ao longo do ciclo de vida do produto. Obviamente, se o desenvolvedor de software não for responsável por qualquer manutenção pós-entrega do produto, então, sob o ponto de vista apenas daquele desenvolvedor, a TC_{nova} é uma proposta mais interessante. Afinal de contas, o uso da TC_{nova} custaria 10% menos. O cliente poderia insistir em manter a TC_{antiga} e pagar os custos iniciais mais elevados na expectativa de que o custo ao longo de todo o ciclo de vida do software seja

menor. Infelizmente, é usual que o único objetivo tanto do cliente quanto do fornecedor de software seja produzir código o mais rápido possível. Os efeitos no longo prazo da adoção de uma determinada técnica geralmente são ignorados diante dos ganhos no curto prazo. Aplicar princípios econômicos à engenharia de software requer que o cliente opte pelas técnicas que reduzam os custos no longo prazo.

Esse exemplo trata de codificação, que constitui menos de 10% das atividades envolvidas no desenvolvimento de software. Entretanto, os princípios econômicos se aplicam também a todos os demais aspectos da produção de software.

Veremos agora a importância da manutenção.

1.3 Aspectos da Manutenção

Nesta seção, descreveremos a manutenção no contexto do ciclo de vida do software. **Um modelo de ciclo de vida** é uma descrição das etapas que deveriam ser seguidas quando se cria um produto de software. Foram propostos diversos modelos diferentes de ciclo de vida; vários deles são descritos no Capítulo 2. Pelo fato de quase sempre ser mais fácil realizar uma seqüência de tarefas menores que uma única grande tarefa, o modelo de ciclo de vida completo é subdividido em uma série de etapas menores, denominadas **fases**. O número de fases varia de modelo para modelo – desde poucas, como quatro, até muitas, como oito. Contrastando com um modelo de ciclo de vida, que é uma descrição teórica daquilo que deve ser feito, a série de etapas realizadas em um determinado produto de software, desde o estudo do conceito até a retirada final do produto, é denominada **ciclo de vida** daquele produto. Na prática, as fases do ciclo de vida de um produto de software talvez não sejam percorridas exatamente como especificadas no modelo de ciclo de vida, particularmente quando custos e prazos forem maiores que os previstos. Tem-se alegado que mais projetos de software deram errado em conseqüência da falta de tempo do que por causa de quaisquer outras razões combinadas (Brooks, 1975).

Até o final da década de 1970, a maioria das organizações produzia software usando como modelo de ciclo de vida o que hoje em dia é denominado **modelo cascata**. Existem muitas variações desse modelo, porém, em geral, um produto desenvolvido que usa esse modelo clássico de ciclo de vida passa pelas seis fases mostradas na Figura 1.2. Provavelmente, essas fases não correspondem exatamente às fases de qualquer empresa em particular, porém elas são suficientemente próximas da maioria das aplicações para os fins deste livro. De forma similar, o nome exato de cada fase varia de empresa para empresa. Os nomes aqui adotados para as diversas fases foram escolhidos por serem os mais genéricos possíveis, na esperança de que o leitor possa sentir-se à vontade com eles.

1. *Fase de levantamento de necessidades*. Durante a **fase de levantamento de necessidades**, o conceito é explorado e refinado, e as necessidades do cliente são levantadas.
2. *Fase de análise (especificação)*. As necessidades do cliente são analisadas e apresentadas na forma de um **documento de especificações**, "aquilo que se espera que o produto

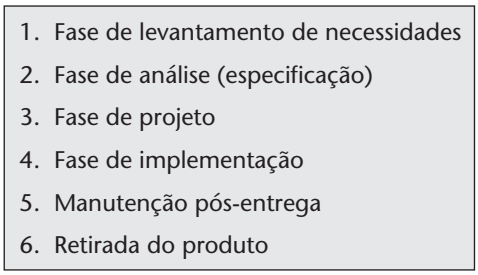


FIGURA 1.2
As seis fases do modelo clássico de ciclo de vida.

faça". A **fase de análise** é chamada, algumas vezes, de **fase de especificação**. No final dessa fase é feito um plano, o **plano de gerenciamento de projeto de software**, que descreve o desenvolvimento do software proposto de forma detalhada.

3. *Fase de projeto*. As especificações são submetidas a dois processos de projeto consecutivos durante a **fase de projeto**. Em primeiro lugar, há o **projeto de arquitetura**, no qual o produto como um todo é subdividido em componentes, denominados **módulos**. Em seguida, cada módulo é projetado; esse processo é denominado **projeto detalhado**. Os dois **documentos de projeto** descrevem "como o produto realiza aquilo que se deseja".

4. *Fase de implementação*. Os vários componentes passam, separadamente, por **codificação** e testes (**testes de unidades**). Em seguida, os componentes do produto são combinados e testados como um todo, e isso é denominado **integração**. Quando os desenvolvedores estão satisfeitos em termos do funcionamento do produto, ele é testado pelo cliente (**teste de aceitação**). A **fase de implementação** termina quando o produto é aceito pelo cliente e instalado nos equipamentos dele. (Veremos, no Capítulo 14, que a codificação e a integração devem ser realizadas paralelamente.)

5. *Manutenção pós-entrega*. O produto é usado para executar as tarefas para as quais foi desenvolvido. Durante esse período, ele sofre manutenção. A **manutenção pós-entrega** inclui todas as modificações feitas no produto assim que ele tiver sido entregue e instalado nos equipamentos do cliente e passar pelo teste de aceitação. A manutenção pós-entrega inclui a **manutenção corretiva** (ou **reparo de software**), que consiste na eliminação de falhas residuais, mantendo inalteradas as especificações, bem como o **aperfeiçoamento** (ou atualização de software), que consiste em mudanças nas especificações e na implementação dessas mudanças. Por outro lado, existem dois tipos de aperfeiçoamento. O primeiro é a **manutenção de aperfeiçoamento**, que são mudanças que o cliente acredita que irão melhorar a eficiência do produto como, por exemplo, uma funcionalidade adicional ou a diminuição no tempo de resposta. O segundo é a **manutenção adaptativa**, que são mudanças feitas em resposta a alterações no ambiente no qual o produto opera como, por exemplo, um novo sistema operacional ou hardware, ou então novas regulamentações governamentais. (Para uma visão geral dos três tipos de manutenção pós-entrega, veja o Quadro 1.3).

6. *Retirada do produto*. A **retirada do produto** (ou **descontinuação do produto**) ocorre quando o produto é retirado de serviço. Isso acontece no momento em que a funcionalidade proposta pelo produto não atende mais às expectativas do cliente.

Examinaremos agora a definição de *manutenção* de forma mais detalhada.

1.3.1 Visões Clássica e Moderna da Manutenção

Na década de 1970, considerava-se que a produção de software era constituída por duas atividades distintas, realizadas em seqüência: *desenvolvimento* seguido por *manutenção*. Partindo-se da estaca zero, o produto de software era desenvolvido e, então, instalado nos equipamentos do cliente. Qualquer mudança no software após a instalação nos equipamentos do cliente e sua respectiva aceitação, seja para solucionar uma falha residual, seja para estender a funcionalidade, constituíam a manutenção clássica (IEEE 610.12, 1990). Portanto, a maneira pela qual o software era desenvolvido classicamente pode ser descrita como um **modelo desenvolvimento-depois-manutenção**.

Essa é uma **definição temporal**, isto é, uma atividade será classificada como manutenção ou desenvolvimento dependendo de quando ela for realizada. Suponhamos que seja detectada uma falha no software e que ela seja corrigida no dia seguinte, após o software

Um dos resultados mais amplamente citados na engenharia de software é que 17,4% das atividades de manutenção pós-entrega são, por natureza, corretivas; 18,2% são adaptativas; 60,3% são de aperfeiçoamento; e 4,1% podem ser classificadas como "outras". Esse resultado foi extraído de um artigo publicado em 1978 (Lientz, Swanson e Tompkins, 1978).

Entretanto, o resultado apresentado nesse artigo não resultou de *medições* nos dados de manutenção. Os autores, em vez disso, realizaram uma pesquisa com gerentes de manutenção que deveriam *estimar* quanto tempo era dedicado a cada uma dessas categorias dentro de suas empresas como um todo, e informar o grau de confiança que eles tinham em suas estimativas. Mais especificamente, foi perguntado aos gerentes de manutenção de software participantes dessa pesquisa se suas respostas se baseavam em dados razoavelmente precisos, em dados mínimos ou em dado algum; 49,3% afirmaram que suas respostas se baseavam em dados razoavelmente precisos; 37,7%, em dados mínimos; e 8,7%, em nenhum dado.

Na realidade, deveria ser questionado se qualquer um dos participantes da pesquisa tinha "dados razoavelmente precisos" referentes à porcentagem de tempo dedicada às categorias de manutenção incluídas na pesquisa; a maioria deles, provavelmente, não tinha nem "dados mínimos". Nessa pesquisa, foi solicitado aos participantes que indicassem qual porcentagem da manutenção era constituída de itens como "reparos de emergência" ou "depuração de rotina"; a partir desses dados brutos, foram deduzidas as porcentagens de manutenção adaptativa, corretiva e perfectiva. A engenharia de software estava apenas começando como disciplina no ano de 1978, e era uma exceção para os gerentes de manutenção de software coletarem informações detalhadas necessárias para responder a uma pesquisa dessas. De fato, na terminologia moderna, em 1978 praticamente toda a empresa ainda se encontrava no nível 1 da escala CMM (veja a Seção 3.13).

Portanto, temos bases concretas para questionar se a real distribuição das atividades de manutenção nos idos de 1978 guardava qualquer semelhança com as estimativas dos gerentes de manutenção que participaram da pesquisa. A distribuição das atividades de manutenção certamente não acontece como nos dias de hoje. Por exemplo, os resultados sobre dados de manutenção reais para o Linux kernel (Schach et al., 2002) e para o compilador gcc (Schach et al, 2003b) mostram que pelo menos 50% da manutenção pós-entrega é corretiva, contrastando com os 17,4% obtidos na pesquisa em questão.

ter sido instalado. Por definição, isso se constitui em uma manutenção clássica. Porém, se essa mesma falha for detectada e corrigida um dia antes de o software ser instalado, em termos da definição, ela se constitui em desenvolvimento clássico. Suponhamos agora que um produto de software tenha acabado de ser instalado, porém o cliente deseja aumentar a funcionalidade do produto de software. Classicamente, isso seria descrito como "manutenção de aperfeiçoamento". Entretanto, se o cliente quiser que essa mesma alteração seja feita imediatamente antes de o software ser instalado, isso seria desenvolvimento clássico. Outra vez, não há nenhuma diferença na natureza das duas atividades, porém, classicamente, uma delas é considerada desenvolvimento, ao passo que a outra é considerada manutenção de aperfeiçoamento.

Além dessas inconsistências, duas outras razões explicam por que o modelo desenvolvimento-depois-manutenção não é mais viável hoje em dia:

1. Hoje em dia, certamente não é raro que a construção de um produto leve um ano ou mais. Durante esse período, as necessidades do cliente podem muito bem mudar. Por exemplo, o cliente poderia insistir que o produto fosse implementado em um microprocessador mais rápido que acaba de ser lançado. Alternativamente, a empresa-cliente poderia ter se expandido para a Bélgica enquanto o desenvolvimento estava em andamento, e o produto agora tem de ser modificado para poder também abranger as vendas na Bélgica. Para ver

como uma mudança nas necessidades do cliente pode afetar o ciclo de vida do software, suponha que as necessidades do cliente mudem enquanto o projeto está sendo desenvolvido. Ou seja, os desenvolvedores devem iniciar a atividade de "manutenção" muito antes de o produto ser instalado.

2. Um segundo problema com o modelo clássico de desenvolvimento-depois-manutenção surgiu como resultado da maneira pela qual criamos software. Na engenharia de software clássica, uma característica de desenvolvimento era que a equipe de desenvolvimento criava o produto-alvo partindo da estaca zero. Em contraste, como conseqüência do alto custo da produção de software hoje em dia, sempre que possível os desenvolvedores tentam reutilizar partes de produtos de software existentes no produto a ser criado (a reutilização é discutida detalhadamente no Capítulo 8). Portanto, o modelo de desenvolvimento-depois-manutenção é inadequado para os dias de hoje porque a reutilização é amplamente difundida.

Uma maneira mais realista de considerar a manutenção é aquela dada no padrão para processos de ciclo de vida publicado pela International Organization for Standardization (ISO) e pela International Electrotechnical Commission (IEC). Isto é, a manutenção é o processo que ocorre quando o "software sofre modificações no código e na respectiva documentação por causa de um problema ou pela necessidade de melhoria ou adaptação" (ISO/IEC 12207, 1995). Em termos dessa **definição operacional**, a manutenção ocorre sempre que uma falha for corrigida ou quando as necessidades mudarem, independentemente de isso ocorrer antes ou depois da instalação do produto. O Institute for Electrical and Electronics Engineers (IEEE) e a Electronic Industries Alliance (EIA) adotaram, subseqüentemente, essa definição (IEEE/EIA 12207.0-1996, 1998) quando os padrões IEEE foram modificados para atender à ISO/IEC 12207. (Veja o Quadro 1.4 para mais informações sobre a ISO.)

Neste livro, o termo *manutenção pós-entrega* refere-se à definição de manutenção da IEEE de 1990 como qualquer mudança no software após ele ter sido entregue e instalado nos equipamentos do cliente, e *manutenção moderna*, ou simplesmente **manutenção**, refere-se à definição da ISO/IEC de 1995 para atividades corretivas, de aperfeiçoamento ou adaptativas realizadas a qualquer momento. A manutenção pós-entrega é, portanto, um subconjunto da manutenção (moderna).

1.3.2 A Importância da Manutenção Pós-entrega

Diz-se, certas vezes, que apenas produtos de software ruins passam por manutenção pós-entrega. De fato, o oposto é verdadeiro: produtos ruins são descartados, ao passo que bons produtos são reparados e aperfeiçoados durante 10, 15 ou até mesmo 20 anos. Além disso, um produto de software é um modelo do mundo real, e o mundo real está mudando permanentemente. Como conseqüência disso, um software tem de sofrer manutenção constantemente para permanecer um reflexo fiel do mundo real.

Por exemplo, se a alíquota do imposto sobre vendas mudar de 6% para 7%, praticamente todos os produtos de software relativos a compra e venda terão de ser modificados. Suponha que o produto contenha a instrução em C++:

```
const float salesTax = 6.0;
```

ou a instrução equivalente em Java:

```
public static final float salesTax = (float) 6.0;
```

Vale a pena saber — Quadro 1.4

A International Organization for Standardization (ISO) é uma rede de institutos nacionais de padronização de 147 países, com sede central em Genebra, Suíça. A ISO publicou mais de 13.500 padrões aceitos internacionalmente, que vão desde padrões para velocidade de exposição de filmes fotográficos ("número ISO") até muitos dos padrões apresentados neste livro. Por exemplo, a ISO 9000 é discutida no Capítulo 3.

ISO não é apenas um acrônimo. Ele é derivado da palavra grega ἴσος, que significa "igual", a mesma raiz do prefixo *iso* que é encontrada em palavras como *isótopo*, *isóbaro* e *isósceles*. A International Organization for Standardization escolheu ISO como forma reduzida para seu nome para evitar o uso de vários acrônimos resultantes da tradução do nome "International Organization for Standardization" para as diversas línguas dos diferentes países-membros. Em vez disso, para haver uma padronização internacional, foi escolhida uma forma reduzida e universal de seu nome.

declarando que *salesTax* (imposto sobre vendas) é uma constante de ponto flutuante inicializada para o valor 6.0. Nesse caso, a manutenção é relativamente simples. Com o auxílio de um editor de textos, o valor 6.0 é substituído por 7.0, e o código é recompilado e "lincado" novamente. Entretanto, se em vez de usar o nome salesTax, o próprio valor 6.0 tivesse sido usado no produto em todos os pontos em que o valor da taxa de vendas fosse solicitado, então um produto desses seria extremamente difícil de ser modificado. Por exemplo, poderiam existir ocorrências para o valor 6.0 no código-fonte que deveriam ser modificadas para 7.0, mas que passaram despercebidas, ou ocorrências para 6.0 que não se referem ao imposto sobre vendas, mas que foram incorretamente alteradas para 7.0. Encontrar essas falhas quase sempre é difícil e consome muito tempo. Na realidade, com algum software, poderia ser menos dispendioso no longo prazo jogar fora o produto e recodificá-lo, em vez de tentar determinar qual das muitas constantes precisam ser alteradas e como fazer tais modificações.

O verdadeiro mundo em tempo real também está em constante mudança. Os mísseis carregados em um jato de combate podem ser substituídos por um novo modelo, exigindo uma mudança no componente de controle das armas do sistema de vôo associado. Um motor de seis cilindros está para ser oferecido como opção em um carro popular de quatro cilindros; isso implica mudar os computadores de bordo que controlam o sistema de injeção de combustível, de controle de tempo, e assim por diante.

Mas exatamente quanto tempo (= dinheiro) é dedicado à manutenção pós-entrega? O gráfico da Figura 1.3 (a) mostra que, cerca de 30 anos atrás, aproximadamente dois terços dos custos de software eram canalizados para a manutenção pós-entrega; os dados foram obtidos com a média de várias fontes de informação, entre as quais (Elshoff, 1976; Daly, 1977; Zelkowitz, Shraw e Gannon, 1979; e Boehm, 1981). Dados mais recentes mostram que uma proporção ainda maior é dedicada à manutenção pós-entrega. Muitas empresas dedicam 70% a 80% ou mais do orçamento para software em manutenção pós-entrega (Yourdon, 1992; Hatton, 1998), conforme mostrado na Figura 1.3 (b).

Surpreendentemente, as porcentagens do custo médio das fases de desenvolvimento clássicas mudaram radicalmente. Isso é mostrado na Figura 1.4, que compara os dados usados para obter a Figura 1.3 (a) com dados mais recentes de 132 projetos da Hewlett-Packard (Grady, 1994).

Consideremos agora a empresa de software que usa atualmente a técnica de codificação TC_{antiga} e que fica sabendo que a TC_{nova} pode reduzir o tempo de codificação em 10%. Mesmo que a TC_{nova} não apresente nenhum efeito adverso sobre a manutenção, um gerente de software perspicaz pensaria duas vezes antes de mudar as práticas de codificação. Todo o pessoal precisaria ser retreinado, teriam de ser adquiridas novas ferramentas de desenvolvimento de software e, quem sabe, teria de ser contratado pessoal novo com experiência na nova técnica. Todas essas despesas e os transtornos têm de ser tolerados para uma diminuição

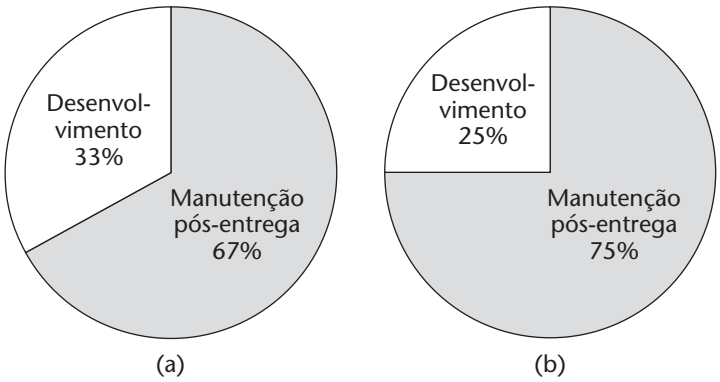


FIGURA 1.3
Porcentagens aproximadas de custo médio e de manutenção pós-entrega
(a) entre 1976 e 1981 e
(b) entre 1992 e 1998.

FIGURA 1.4 Uma comparação entre as porcentagens aproximadas de custo médio das fases de desenvolvimento clássico para vários projetos entre 1976 e 1981 e para os 132 projetos mais recentes da Hewlett-Packard.

	Vários projetos entre 1976 e 1981	**Projetos mais recentes da Hewlett-Packard**
Fases de levantamento de necessidades e de análise (especificação)	21%	18%
Fase de projeto	18%	19%
Fase de implementação		
Codificação (inclusive teste de unidades)	36%	34%
Integração	24%	29%

de, no máximo, 0,85% nos custos de software, pois, conforme é mostrado nas Figuras 1.3 (b) e 1.4, a codificação juntamente aos testes de unidades constitui, em média, apenas 34% dos 25%, ou seja, 8,5% do custo total com software.

Suponhamos agora que seja desenvolvida uma nova técnica que reduza em 10% os custos com manutenção pós-entrega. Essa deveria, provavelmente, ser introduzida de uma só vez, pois, em média, irá reduzir os custos em 7,5%. Os gastos indiretos envolvidos na mudança para essa técnica têm um preço pequeno a ser pago, dada a grande economia que será gerada como um todo.

Pelo fato de a manutenção pós-entrega ser tão importante, um aspecto fundamental da engenharia de software consiste naquelas técnicas, ferramentas e práticas que levam a uma redução nos custos de manutenção pós-entrega.

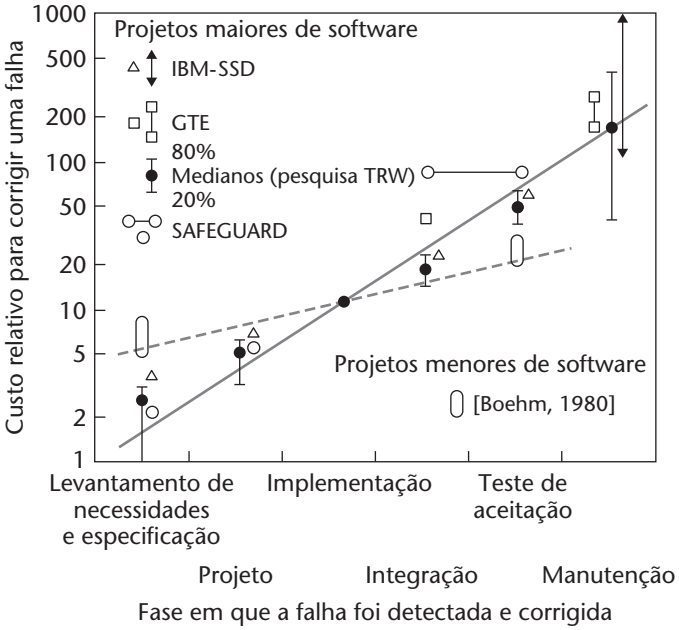
1.4 Aspectos Sobre Levantamento de Necessidades, Análise e Projeto

Os profissionais de software são seres humanos, e, portanto, algumas vezes, podem cometer erros ao desenvolver um produto. Conseqüentemente, existirá uma falha no software. Se esse erro for cometido na fase de levantamento de necessidades, a falha resultante provavelmente também irá aparecer nas especificações, no projeto e no código. Fica claro que, quanto antes corrigirmos uma falha, melhor será.

Os custos relativos a corrigir uma falha nas várias fases do modelo clássico de ciclo de vida são mostrados na Figura 1.5 (Boehm, 1981). A figura reflete dados da IBM (Fagan,

FIGURA 1.5 Custo relativo para corrigir uma falha em cada uma das fases do ciclo de vida clássico do software. A linha cheia é a que melhor reflete os dados relacionados a projetos maiores de software, e a linha tracejada é a mais adequada a dados relativos a projetos menores de software.

Fonte: Barry Boehm, Software Engineering Economics, © 1981, p. 40. Adaptado com a permissão da Prentice Hall, Englewood Cliffs, NJ.



1974), GTE (Daly, 1977), o projeto Safeguard (Stephenson, 1976) e alguns projetos menores da TRW (Boehm, 1980). A linha cheia na Figura 1.5 é a melhor representação para os dados sobre projetos maiores, ao passo que a linha tracejada reflete melhor os projetos menores. Para cada uma das fases do modelo clássico de ciclo de vida, o custo relativo correspondente para detectar e corrigir a falha é representado na Figura 1.6. Cada etapa na linha cheia da Figura 1.6 é construída pegando-se o ponto correspondente na linha reta cheia da Figura 1.5 e representando-se os dados graficamente em uma escala linear.

Suponha que custe $40 para se detectar e corrigir uma determinada falha durante a fase de projeto. A partir da linha cheia da Figura 1.6 (projetos entre 1974 e 1980), a mesma falha iria custar apenas cerca de $30 para ser corrigida durante a fase de análise. Porém, durante a manutenção pós-entrega, essa mesma falha custaria cerca de $2.000 para ser detectada e corrigida. Dados mais recentes mostram que hoje em dia é ainda mais importante detectar falhas quanto antes. A linha tracejada da Figura 1.6 mostra o custo para detectar e corrigir uma falha durante o desenvolvimento de software do sistema para o IBM AS/400 (Kan et al., 1994). Em média, a mesma falha teria custado $3.680 para ser corrigida durante a manutenção pós-entrega do software para o AS/400.

A razão para o custo de corrigir uma falha aumentar tão vertiginosamente está relacionada ao que deve ser feito para corrigir a falha. Logo no início do ciclo de vida de desenvolvimento, o produto existe, basicamente, apenas no papel, e corrigir uma falha pode significar simplesmente fazer uma modificação em um documento. O outro extremo é o produto já entregue para o cliente. No mínimo, corrigir uma falha nessa etapa significaria editar o código, recompilá-lo e relincá-lo e, então, testá-lo cuidadosamente para confirmar que o problema

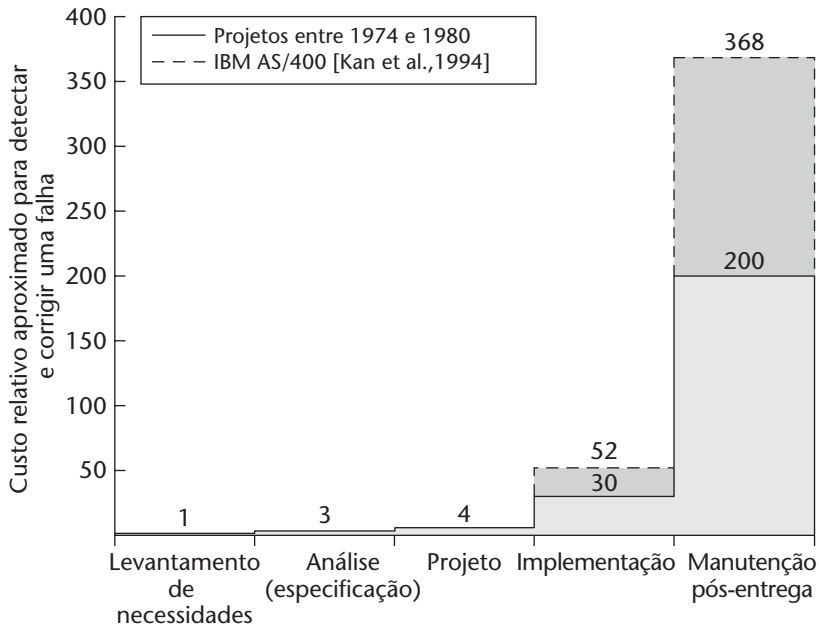


FIGURA 1.6
A linha cheia representa os pontos na linha cheia da Figura 1.5 plotados em uma escala linear. A linha tracejada representa dados mais novos.

foi solucionado. Em seguida, é fundamental verificar se a implementação da mudança não acaba criando um novo problema em algum outro ponto do produto. Toda a documentação relevante, inclusive manuais, precisa ser atualizada. Finalmente, o produto corrigido deve ser entregue e reinstalado. Moral da história: temos de encontrar as falhas logo cedo; caso contrário, isso custará dinheiro. Precisamos, portanto, empregar técnicas para detectar falhas durante as fases de levantamento de necessidades e de análise (especificação).

Há uma necessidade adicional para tais técnicas. Estudos mostraram (Boehm, 1979) que 60% a 70% de todas as falhas detectadas em grandes projetos acontecem nas fases de levantamento de necessidades, análise ou de projeto. Resultados mais recentes foram obtidos a partir de inspeções (inspeção é um exame meticuloso de um documento por uma equipe, conforme descrito na Seção 6.2.3). Durante 203 inspeções do software Jet Propulsion Laboratory, para o programa espacial interplanetário não tripulado da NASA, em média foram detectadas cerca de 1,9 falhas por página de um documento de especificações, 0,9 falhas por página de um projeto, mas apenas 0,3 falhas por página de código (Kelly, Sherif e Hops, 1992).

Conseqüentemente, é importante que melhoremos nossas técnicas de levantamento de necessidades, de análise e de projeto, não somente para que as falhas possam ser encontradas quanto antes como também porque as falhas de levantamento de necessidades, de análise e de projeto constituem uma proporção muito grande de todas as falhas. Assim como o exemplo da Seção 1.3 mostrou que reduzir os custos de manutenção pós-entrega em 10% significa reduzir o custo geral em cerca de 7,5%, reduzir as falhas nas fases de levantamento de necessidades, de análise e de projeto em 10% corresponde a reduzir o número total de falhas em 6% a 7%.

O fato de tantas falhas serem introduzidas muito cedo no ciclo de vida do software realça outro importante aspecto das técnicas de engenharia de software: técnicas que levam a melhores levantamentos de necessidades, especificações e projetos.

A maioria dos programas é produzida por uma equipe de engenheiros de software, e não por um único indivíduo responsável por cada aspecto do ciclo de vida de desenvolvimento e manutenção. Consideremos agora as implicações disso.

1.5 Aspectos da Equipe de Desenvolvimento

O custo de hardware continua a decrescer rapidamente. Um mainframe da década de 1950, que custava mais de 1 milhão de dólares antes da inflação, era consideravelmente menos potente em todos os aspectos que um laptop de hoje, que custa menos de mil dólares. Como resultado, as empresas têm poder aquisitivo para adquirir hardware capaz de rodar produtos pesados, isto é, produtos muito grandes para serem escritos por apenas uma pessoa dentro das restrições de tempo impostas. Se, por exemplo, um produto tiver de ser entregue em 18 meses, mas um único profissional da área de software levaria 15 anos para terminá-lo, então esse produto tem de ser desenvolvido por uma equipe. Entretanto, o desenvolvimento em equipe leva a problemas de compatibilidade entre os componentes do código e problemas de comunicação entre os membros da equipe.

Consideremos, por exemplo, os módulos de código p e q, de Jeff e Juliet, respectivamente, sendo que o módulo p chama o módulo q. Quando Jeff codifica p, ele escreve uma chamada para o q com cinco argumentos na lista de argumentos. Juliet codifica q com cinco argumentos, porém em uma ordem diferente daquela usada por Jeff. Algumas ferramentas de software, como o interpretador e carregador Java ou o *lint* para C (Seção 8.7.4), detectam esses tipos de violação, mas somente se os argumentos trocados forem de tipos diferentes; se eles forem do mesmo tipo, então o problema talvez não seja detectado por um longo período. É possível argumentar que esse é um problema de projeto e que, se os módulos tivessem sido projetados de forma mais cuidadosa, não teria surgido. Isso pode até ser verdade, porém, na prática, um projeto muitas vezes é modificado após a codificação ter sido iniciada; mas a notificação de uma alteração talvez não seja distribuída a todos os membros da equipe de desenvolvimento. Conseqüentemente, quando um projeto que afeta dois ou mais programadores é alterado, uma comunicação falha pode levar aos problemas de compatibilidade vividos por Jeff e Juliet. Esse tipo de problema é menos susceptível de acontecer quando apenas um indivíduo é responsável por cada aspecto do produto, como era o caso antes do surgimento de computadores mais poderosos e capazes de rodar produtos imensos, com preços acessíveis.

Mas problemas de compatibilidade são meramente a ponta do *iceberg* quando se trata do que pode surgir quando o software é desenvolvido por equipes. A menos que a equipe esteja organizada apropriadamente, pode se perder um tempo excessivamente grande em conversas entre os membros da equipe. Suponha que um produto leve um ano para ser finalizado por apenas um programador. Se a mesma tarefa for atribuída a uma equipe de seis programadores, o tempo para completar a mesma tarefa, muitas vezes, é próximo a um ano, em vez dos dois meses esperados, e a qualidade do código resultante pode muito bem ser pior do que se a tarefa toda tivesse sido atribuída a um único indivíduo (veja a Seção 4.1). Como uma parcela considerável do software de hoje é desenvolvida e mantida por equipes, o raio de ação da engenharia de software deve incluir técnicas para garantir que as equipes sejam adequadamente organizadas e gerenciadas.

Como foi mostrado nas seções anteriores, o escopo da engenharia de software é extremamente amplo. Ele inclui cada etapa do ciclo de vida do software, desde o levantamento de necessidades até a retirada do produto no final de sua vida. Também inclui aspectos humanos, como organização de equipe, e aspectos econômicos e jurídicos, como direitos autorais. Todos esses aspectos são incorporados implicitamente na definição de engenharia de software apresentada no início deste capítulo, a qual diz que a engenharia de software é uma disciplina cujo objetivo é a produção de software isento de falhas, entregue dentro do prazo e orçamento previstos, que atenda às necessidades do cliente.

Retornemos às fases clássicas da Figura 1.2 para questionar por que não existem as fases de planejamento, testes ou de documentação.

1.6 Por Que Não Existe Fase de Planejamento?

É evidente que é impossível desenvolver um produto de software sem um plano. Por conseguinte, se depreende ser essencial haver uma **fase de planejamento** logo no início do projeto.

O ponto-chave é que, até saber exatamente o que será desenvolvido, não há nenhuma maneira de formular um plano detalhado e preciso. Conseqüentemente, ocorrem três tipos de atividades de planejamento quando um produto de software é desenvolvido usando-se o paradigma clássico.

1. No início do projeto, ocorre um planejamento preliminar para as fases de levantamento de necessidades e de análise.
2. Assim que é conhecido precisamente aquilo que será desenvolvido, é elaborado o *plano de gerenciamento de projeto de software,* ou SPMP (sigla de Software Project Management Plan). Isso inclui o orçamento, as necessidades de pessoal e um cronograma detalhado. O primeiro momento em que temos condições de elaborar um plano de gerenciamento de projeto é aquele em que o documento de especificações foi aprovado pelo cliente, isto é, no final da fase de análise. Até esse momento, o planejamento tem de ser preliminar e parcial.
3. Ao longo de todo o projeto, o gerenciamento precisa monitorar o SPMP e estar atento a qualquer desvio em relação ao plano original.

Suponhamos, por exemplo, que o SPMP para um projeto específico afirme que o projeto como um todo levará 16 meses e que a fase de projeto consumirá quatro desses 16 meses. Após um ano, o gerenciamento percebe que o projeto como um todo parece estar evoluindo de forma consideravelmente mais lenta do que a prevista. Uma investigação detalhada mostra que, até então, foram dedicados oito meses à fase de projeto, que ainda se encontra longe de ser completada. O projeto provavelmente terá de ser abandonado, e os investimentos feitos até então serão desperdiçados. Em vez disso, o gerenciamento deveria ter acompanhado o progresso por fase e percebido, após no máximo dois meses, que há um problema sério na fase de projeto. Naquela oportunidade, deveria ter sido tomada uma decisão a respeito da melhor forma para prosseguir. O passo inicial usual, em uma situação dessas, é convocar um consultor para determinar se o projeto é factível e se a equipe de desenvolvimento é competente para realizar tal tarefa, ou se o risco de prosseguir é muito grande. Baseando-se no relatório do consultor, são consideradas então várias alternativas, inclusive reduzir o escopo do produto pretendido, e então projetar e implementar um menos ambicioso. Apenas se todas as demais alternativas forem consideradas impraticáveis, o projeto deve ser cancelado. No caso desse projeto específico, esse cancelamento teria de ter acontecido cerca de seis meses antes, caso o gerenciamento houvesse monitorado de perto o plano, poupando uma soma considerável.

Para concluir, não existe uma fase de planejamento separada. Em vez disso, as atividades de planejamento são executadas ao longo de todo o ciclo de vida do produto. Entretanto, há ocasiões em que as atividades de planejamento predominam, entre as quais está o início do projeto (planejamento preliminar) e logo depois de o documento de especificações ter sido aprovado pelo cliente (o plano de gerenciamento de projeto de software).

1.7 Por Que Não Existe Fase de Testes?

É essencial verificar meticulosamente um produto de software após ele ter sido desenvolvido. Conseqüentemente, é razoável perguntar-se por que não existe fase de testes após o produto ter sido implementado.

Infelizmente, verificar um produto de software apenas quando ele está pronto para ser entregue ao cliente já é muito tarde. Por exemplo, se existir uma falha no documento de espe-

cificações, essa falha terá se propagado para o projeto e para a implementação. Há vezes, no processo, em que os testes são realizados quase que totalmente à parte das demais atividades. Isso ocorre próximo ao final de cada fase (**verificação**) e, especialmente, antes de o produto ser repassado para o cliente (**validação**). Embora existam vezes em que os testes predominem, jamais deve haver ocasiões em que não está sendo realizado nenhum teste. Se os testes forem tratados como uma **fase de testes** separada, então existe um perigo real de que os testes não sejam realizados constantemente ao longo de cada fase do processo de desenvolvimento e de manutenção do produto.

Entretanto, nem mesmo isso é suficiente. O que é necessário é a verificação contínua de um produto de software. A verificação meticulosa deve acompanhar automaticamente cada uma das atividades de desenvolvimento e de manutenção. Do contrário, uma fase de testes separada é incompatível com a meta de garantir que um produto de software esteja, na medida do possível, livre de defeitos a todo instante.

Toda empresa de desenvolvimento de software deveria ter um grupo independente cuja responsabilidade primária fosse a de garantir que o produto entregue seja, de fato, aquilo que o cliente precisa e que o produto tenha sido elaborado corretamente em todos os aspectos. Esse grupo é denominado grupo de *garantia da qualidade de software* (SQA – sigla de Software Quality Assurance). A **qualidade** do software corresponde ao grau em que ele atende às especificações. Qualidade e garantia da qualidade de software são pontos descritos com mais detalhes no Capítulo 6, já que é papel da SQA estabelecer e fazer cumprir padrões.

1.8 Por Que Não Existe Fase de Documentação?

Da mesma forma que jamais deve existir uma fase de planejamento ou de testes em separado, também jamais deve haver uma **fase de documentação** separada. Pelo contrário, a todo momento, a documentação de um produto de software deve ser completa, correta e atualizada. Por exemplo, durante a fase de análise, o documento de especificações deve refletir a versão atual das especificações e, de forma similar, isso deve ocorrer para as demais fases.

1. Uma razão para que seja essencial garantir que a documentação esteja sempre atualizada é o alto índice de rotatividade de pessoal no segmento de software. Suponha, por exemplo, que a documentação de projeto não tenha sido mantida atualizada e que o projetista-chefe tenha deixado a empresa em detrimento de um outro emprego. A partir disso será extremamente difícil atualizar o documento de projeto para que ele reflita todas as mudanças feitas enquanto o sistema estava sendo projetado.
2. É praticamente impossível realizar as etapas de uma determinada fase, a menos que a documentação da fase anterior esteja completa, correta e atualizada. Por exemplo, um documento de especificações incompleto vai, inevitavelmente, resultar em um projeto incompleto e, depois, em uma implementação incompleta.
4. É praticamente impossível testar se um produto de software está funcionando corretamente, a menos que estejam disponíveis documentos que atestem como o produto de software deveria se comportar.
5. A manutenção é praticamente impossível, a menos que exista uma documentação completa e correta que descreva precisamente o que a versão atual do produto faz.

Portanto, assim como não existe uma fase de planejamento ou de testes em separado, não há uma fase de documentação distinta. Em vez disso, o planejamento, os testes e a documentação devem ser atividades que acompanham todas as demais atividades, à medida que um produto de software é desenvolvido.

Agora examinaremos o paradigma de orientação a objetos.

1.9 O Paradigma de Orientação a Objetos

Antes de 1975, a maioria das empresas de software não usava nenhuma técnica específica; cada indivíduo trabalhava do seu próprio jeito. Grandes avanços foram feitos aproximadamente entre 1975 e 1985, com o desenvolvimento do assim chamado **paradigma clássico** ou **estruturado**. Entre as técnicas que formavam o paradigma clássico havia a análise de sistemas estruturada (Seção 11.3), a análise de fluxo de dados (Seção 13.3), a programação estruturada e os testes estruturados (Seção 14.13.2). Essas técnicas pareciam extremamente promissoras quando usadas no início. Entretanto, à medida que o tempo foi passando, constatou-se que elas ficaram aquém do esperado em dois aspectos:

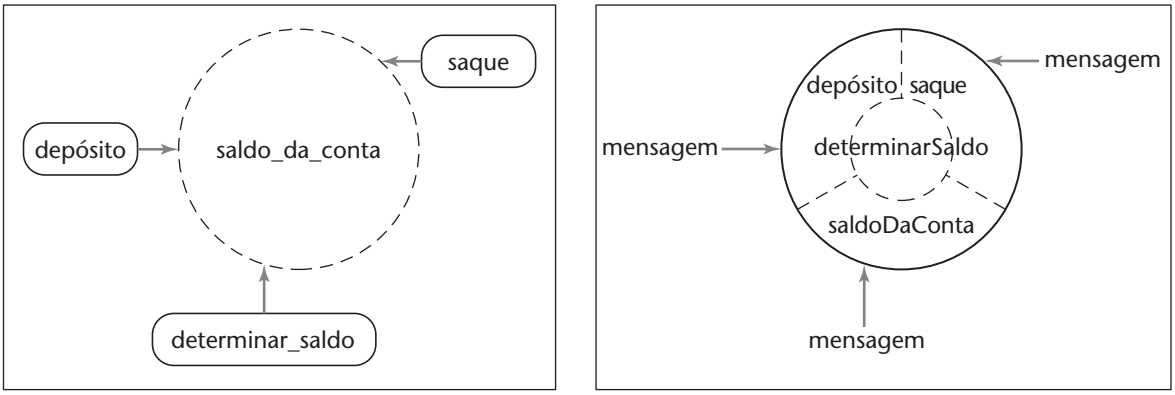
1. Algumas vezes, as técnicas eram incapazes de lidar com o tamanho cada vez maior dos produtos de software, isto é, as técnicas clássicas eram adequadas para elaborar produtos de escala pequena (tipicamente com 5 mil linhas de código) ou até mesmo produtos de escala média, com cerca de 50 mil linhas de código. Atualmente, entretanto, produtos de larga escala, com 500 mil linhas de código, são relativamente comuns; até mesmo produtos com 5 milhões ou mais de linhas de código não são considerados raros. Entretanto, as técnicas clássicas normalmente não conseguem ser ampliadas de forma a conseguir lidar com o desenvolvimento dos produtos atuais de maior porte.
2. O paradigma clássico não estava à altura das expectativas iniciais durante a manutenção pós-entrega. Um importante agente propulsor do paradigma clássico há 30 anos foi que, em média, dois terços do orçamento para software eram destinados à manutenção pós-entrega (veja a Figura 1.3). Infelizmente, o paradigma clássico não havia resolvido esse problema; conforme indicado na Seção 1.3.2, várias organizações ainda despendem de 70% a 80% ou mais de seu tempo e esforços em manutenção pós-entrega (Yourdon, 1992; Hatton, 1998).

Uma das principais razões para o sucesso limitado do paradigma clássico foi que as técnicas clássicas são orientadas a operações ou orientadas a atributos (dados), mas não a ambos ao mesmo tempo. Os componentes básicos de um produto de software são as operações do produto e os atributos sobre os quais essas operações agem. Por exemplo, determinar_altura_média[1] é uma operação que atua sobre um conjunto de alturas (atributos) e retorna a média dessas alturas (atributos). Algumas técnicas clássicas, quanto a análise de fluxo de dados (Seção 13.3), são orientadas a operações, ou seja, tais técnicas se concentram nas operações do produto; os atributos ficam em segundo plano. Diferentemente, técnicas como o desenvolvimento de sistemas Jackson (Seção 13.5) são orientadas a atributos. A ênfase, nesse caso, é nos atributos; as operações que atuam sobre os atributos são menos significativas.

Em contraste, o paradigma de orientação a objetos considera igualmente importantes tanto os atributos quanto as operações. Uma maneira simplista de compreender um objeto é vê-lo como um artefato de software unificado, que incorpora tanto os atributos quanto as operações realizadas sobre os atributos (um **artefato** é um componente de um produto de software como, por exemplo, um documento de especificações, um módulo de programa ou

[1] Neste livro, o nome de uma variável em um produto de software clássico é escrito usando-se a convenção clássica de separar as partes do nome de uma variável com sublinhas; por exemplo, essa_é_uma_variável_clássica. Uma variável em um produto de software orientado a objetos é escrita usando-se a convenção da programação orientada a objetos, ou seja, uma letra maiúscula para indicar o início de uma nova parte da variável; por exemplo, essaÉUmaVariávelOrientadaAObjetos.

FIGURA 1.7 Comparação da implementação de uma conta bancária usando (a) o paradigma clássico e (b) o paradigma de orientação a objetos. A linha cheia preta em volta do objeto indica que os detalhes de como o saldoDaConta é implementado não são conhecidos fora do objeto.



um manual). Essa definição de objeto é incompleta e será enriquecida posteriormente neste livro, assim que houver sido definido o conceito de *herança* (Seção 7.8). Não obstante, essa definição capta grande parte da essência de um objeto.

Uma conta bancária é um exemplo de objeto (veja a Figura 1.7). O componente de atributo do objeto é saldoDaConta. As operações que podem ser realizadas sobre esse saldo bancário incluem depositar dinheiro na conta, retirar dinheiro da conta e determinarSaldo. O objeto conta bancária combina um atributo com as três operações realizadas sobre esse atributo em um único artefato. Sob o ponto de vista do paradigma clássico, um produto que lida com bancos teria de incorporar um atributo, o saldo_da_conta, e três operações, depósito, saque e determinar_saldo.

Até agora, parece haver pouca diferença entre as duas abordagens. Entretanto, um ponto fundamental é a maneira pela qual um objeto é implementado. Mais especificamente, detalhes de como os atributos de um objeto são armazenados não são conhecidos externamente ao objeto. Trata-se de um exemplo de "ocultamento de informações", discutido com mais detalhes na Seção 7.6. No caso do objeto conta bancária mostrado na Figura 1.7 (b), o restante do produto de software está informado de que existe algo como um saldo dentro de um objeto conta bancária, mas não há a menor idéia quanto ao formato do saldoDaConta. Isso significa que não há informação externa ao objeto sobre o saldo bancário ser implementado como um número inteiro ou como um número de ponto flutuante, ou então como um campo (componente) de alguma estrutura maior. Essa barreira de informações que envolve o objeto é indicada pela linha cheia preta na Figura 1.7 (b), que representa uma implementação que usa o paradigma de orientação a objetos. Em contraste, uma linha tracejada envolve o saldo_da_conta na Figura 1.7 (a) porque todos os detalhes de saldo_da_conta são conhecidos pelos módulos na implementação que usa o paradigma clássico, e o valor de saldo_da_conta pode, portanto, ser alterado por qualquer um deles.

Retornando à Figura 1.7 (b), a implementação que usa orientação a objetos, se um cliente depositar $10 em uma conta, então uma **mensagem** será enviada para o método depósito do objeto relevante, informando-o para que incremente o atributo saldoDaConta em $10 (um **método** é uma implementação de uma operação). O método depósito está dentro do objeto conta bancária e sabe como o saldoDaConta é implementado; isso é indicado pela

linha tracejada circular dentro do objeto. Mas nenhuma entidade externa ao objeto precisa desse conhecimento. O fato de os três métodos na Figura 1.7 (b) isolarem saldoDaConta do restante do produto simboliza essa localização do conhecimento. O fato de que os detalhes da implementação são locais para um objeto conduz ao primeiro dos vários pontos fortes do paradigma de orientação a objetos:

1. Consideremos a manutenção pós-entrega. Suponha que o produto bancário tenha sido criado usando-se o paradigma clássico. Se o modo pelo qual é representado um saldo_bancário for alterado de, digamos, um (chamado) inteiro para um campo de uma estrutura, então cada parte desse produto que não possui relação com um saldo_da_conta terá de ser alterada, e essas alterações terão de ser feitas de maneira consistente. Em contraste, se for usado o paradigma de orientação a objetos, então as alterações precisarão ser feitas apenas dentro do objeto conta bancária propriamente. Nenhuma outra parte do produto tem conhecimento de como saldoDaConta é implementado, de modo que nenhuma outra parte pode ter acesso a saldoDaConta. Conseqüentemente, o paradigma de orientação a objetos torna a manutenção mais rápida e fácil, e a chance de introdução de uma **falha de regressão** (isto é, uma falha introduzida inadvertidamente em uma parte de um produto como conseqüência de uma alteração aparentemente não relacionada feita em outra parte do produto) é muito reduzida.

2. Além da manutenção, o paradigma de orientação a objetos também facilita o desenvolvimento. Em muitos casos, um objeto tem um equivalente físico. Por exemplo, um objeto conta bancária no produto banco corresponde a uma conta bancária real no banco para o qual esse produto está sendo desenvolvido. Como será mostrado na Parte 2, a modelagem desempenha um importante papel no paradigma de orientação a objetos. A estreita correspondência entre os objetos em um produto e seus equivalentes no mundo real deve levar a softwares de melhor qualidade.

3. Objetos bem projetados são unidades independentes. Conforme foi explicado, um objeto é formado tanto por atributos quanto pelas operações neles realizadas. Se todas as operações realizadas nos atributos de um objeto forem incluídas nesse objeto, então o objeto pode ser considerado uma entidade conceitualmente independente. Tudo no produto que se relaciona com a parte do mundo real modelada por aquele objeto pode ser encontrado no próprio objeto. Essa independência conceitual algumas vezes é denominada **encapsulamento** (Seção 7.4). Mas existe uma outra forma de independência, a independência física. Em um objeto bem projetado, o ocultamento de informações garante que os detalhes de implementação sejam ocultos para tudo o que estiver fora do objeto. A única forma de comunicação permitida é enviar uma mensagem para o objeto realizar uma determinada operação. A maneira pela qual a operação é realizada é de inteira responsabilidade do próprio objeto. Por essa razão, um projeto com orientação a objetos, algumas vezes, é denominado de **projeto dirigido por responsabilidade** (Wirfs-Brock, Wilkerson e Wiener, 1990) ou de **projeto por contrato** (Meyer, 1992). (Para outra visão sobre o projeto dirigido por responsabilidade, veja o Quadro 1.5, derivado de um exemplo encontrado em (Budd, 2002).)

4. Um produto criado com o uso do paradigma clássico é implementado como um conjunto de módulos, mas conceitualmente ele é basicamente uma unidade. Essa é uma das razões de o paradigma clássico ter tido menos sucesso quando aplicado a produtos maiores. Em contraste, quando o paradigma de orientação a objetos é usado corretamente, o produto resultante é formado por uma série de unidades menores e altamente independentes. O paradigma de orientação a objetos reduz o nível de complexidade de um produto de software e, portanto, simplifica tanto o desenvolvimento quanto a manutenção.

Vale a pena saber — Quadro 1.5

Suponha que você mora no Rio de Janeiro e quer enviar um buquê de flores pelo Dia das Mães para sua mãe que mora em São Paulo. Uma estratégia seria consultar na Internet qual floricultura está localizada perto da residência de sua mãe e fazer uma encomenda nessa floricultura. Uma forma mais conveniente seria fazer o pedido em um site especializado, deixando toda a responsabilidade pela entrega das flores para essa empresa. É irrelevante onde a empresa está localizada fisicamente ou para qual floricultura será repassada sua encomenda de flores. De qualquer forma, a empresa não divulga essas informações; é um exemplo de ocultamento de informações.

Exatamente da mesma forma, quando uma mensagem é enviada a um objeto, não apenas é completamente irrelevante o modo como o pedido é realizado como também a unidade que envia a mensagem não tem permissão nem mesmo de saber como é a estrutura interna do objeto. O próprio objeto é inteiramente responsável pelos detalhes do transporte da mensagem.

5. O paradigma de orientação a objetos permite reutilização; pelo fato de os objetos serem entidades independentes, eles podem ser utilizados em novos produtos futuros. Essa reutilização de objetos reduz o tempo e o custo, tanto em termos de desenvolvimento quanto de manutenção, conforme explicado no Capítulo 8.

Quando é utilizado o paradigma de orientação a objetos, o ciclo de vida clássico do software da Figura 1.2 tem de ser modificado. A Figura 1.8 compara o modelo de ciclo de vida do paradigma clássico com aquele do paradigma de orientação a objetos.

A primeira diferença parece ser puramente terminológica; a palavra *fase* é usada no paradigma clássico, ao passo que *fluxo de trabalho* é usado no paradigma de orientação a objetos. Na realidade, como será explicado em detalhes no Capítulo 2, não existe nenhuma correspondência entre fase e fluxo de trabalho. Pelo contrário, os dois termos são totalmente distintos, e essa distinção sintetiza as diferenças entre os modelos de ciclo de vida que formam a base dos dois paradigmas.

Neste capítulo, consideramos outra diferença entre os dois paradigmas: o papel desempenhado pelos módulos (no paradigma clássico) em comparação àquele desempenhado pelos objetos (no paradigma de orientação a objetos). Consideremos primeiramente a fase de projeto do paradigma clássico. Conforme foi dito na Seção 1.3, essa fase é dividida em duas subfases: projeto de arquitetura seguido pela fase de projeto detalhado. Na subfase de projeto de arquitetura, o produto é decomposto em componentes denominados *módulos*. Em seguida, durante a subfase de projeto detalhado, as estruturas de dados e os algoritmos de cada módulo são projetados alternadamente. Finalmente, durante a fase de implementação, esses módulos são implementados.

Se, em vez disso, for usado o paradigma de orientação a objetos, uma das etapas do fluxo de trabalho da análise da orientação a objetos é determinar as classes. Como uma classe é um tipo de módulo, o projeto de arquitetura é realizado durante o fluxo de trabalho da análise da

FIGURA 1.8
Comparação entre os modelos de ciclo de vida do paradigma clássico e do paradigma de orientação a objetos.

Paradigma clássico	Paradigma de orientação a objetos
1. Fase de levantamento de necessidades	1. Fluxo de trabalho do levantamento de necessidades
2. Fase de análise (especificação)	2′. Fluxo de trabalho da análise de orientação a objetos
3. Fase de projeto	3′. Fluxo de trabalho do projeto de orientação a objetos
4. Fase de implementação	4′. Fluxo de trabalho da implementação de orientação a objetos
5. Manutenção pós-entrega	5. Manutenção pós-entrega
6. Retirada do produto	6. Retirada do produto

FIGURA 1.9
Diferenças entre o paradigma clássico e o paradigma de orientação a objetos.

Paradigma clássico	Paradigma de orientação a objetos
2. Fase de análise (especificação) • Determinar o que o produto irá fazer	2′. Fluxo de trabalho da análise de orientação a objetos • Determinar o que o produto irá fazer • Extrair as classes
3. Fase de projeto • Projeto de arquitetura (extrair os módulos) • Projeto detalhado	3′. Fluxo de trabalho do projeto de orientação a objetos • Projeto detalhado
4. Fase de implementação • Codificar os módulos em uma linguagem de programação apropriada • Integrar	4′. Fluxo de trabalho da implementação de orientação a objetos • Codificar as classes em uma linguagem de programação orientada a objetos • Integrar

orientação a objetos. Conseqüentemente, a análise da orientação a objetos vai mais além do que a fase de análise (especificação) correspondente no paradigma clássico. Isso é mostrado na Figura 1.9.

Essa diferença entre os dois paradigmas tem conseqüências importantes. Quando o paradigma clássico é usado, quase sempre existe uma transição abrupta entre a fase de análise e a fase de projeto. Afinal de contas, o objetivo da fase de análise é determinar *aquilo* que o produto deverá fazer, ao passo que o propósito da fase de projeto é decidir *como* fazê-lo. Em contraste, quando é usada a análise na orientação a objetos, os objetos entram no ciclo de vida logo em seu início. Os objetos são extraídos no fluxo de trabalho da análise, projetados no fluxo de trabalho do projeto e codificados no fluxo de trabalho da implementação. O paradigma de orientação a objetos é, portanto, um método integrado; a transição de um fluxo de trabalho para outro fluxo de trabalho é muito mais suave do que no paradigma clássico, reduzindo o número de falhas ocorridas durante o desenvolvimento.

Conforme já foi mencionado, é inadequado definir um objeto meramente como um artefato de software que encapsula tanto atributos quanto operações e implementa o princípio de ocultamento de informações. Uma definição mais completa é apresentada no Capítulo 7, no qual os objetos são examinados em profundidade.

1.10 O Paradigma de Orientação a Objetos em Perspectiva

A Figura 1.1 é uma evidência dos vários pontos fracos do paradigma clássico (estruturado). Entretanto, o paradigma de orientação a objetos não é, de forma alguma, um remédio para todos os males:

- Como todos os métodos para a produção de software, o paradigma de orientação a objetos tem de ser usado corretamente; é tão fácil usar de forma incorreta o paradigma de orientação a objetos quanto qualquer outro paradigma.
- Quando corretamente aplicado, o paradigma de orientação a objetos é capaz de resolver alguns (mas não todos) os problemas do paradigma clássico.
- O paradigma de orientação a objetos apresenta alguns problemas próprios, conforme descrito na Seção 7.9.
- O paradigma de orientação a objetos é o melhor método disponível hoje em dia. Entretanto, como todas as tecnologias, certamente ele será superado por alguma tecnologia superior no futuro.

Neste livro, os prós e os contras tanto do paradigma clássico quanto do paradigma de orientação a objetos são indicados dentro do contexto do tópico específico em discussão.

Conseqüentemente, a comparação dos dois paradigmas não aparece apenas em um único lugar, mas espalhada ao longo de todo o livro.

Definiremos agora uma série de termos da engenharia de software.

1.11 Terminologia

O **cliente** é o indivíduo que gostaria que um produto fosse criado (desenvolvido). Os **desenvolvedores** são os membros de uma equipe responsável pela criação desse produto. Os desenvolvedores podem ser os responsáveis por todos os aspectos do processo, desde o levantamento das necessidades até as etapas subseqüentes, ou então ser responsáveis apenas pela implementação de um produto já concebido.

Tanto o cliente quanto os desenvolvedores podem fazer parte da mesma organização. Por exemplo, o cliente pode ser o chefe de uma companhia de seguros, e os desenvolvedores podem ser de uma equipe chefiada pelo diretor de desenvolvimento de software dessa mesma companhia. Isso é denominado **desenvolvimento de software interno**. Por outro lado, com **software por contrato**, o cliente e os desenvolvedores são membros de organizações totalmente independentes. Por exemplo, o cliente pode ser um funcionário de alto escalão do Ministério da Defesa, e os desenvolvedores podem ser empregados de uma importante empresa de defesa terceirizada, especializada em software para sistemas de armamentos. Em uma escala muito menor, o cliente pode ser um contador em um escritório composto por uma única pessoa, e o desenvolvedor, um estudante que obtém renda criando software em expedientes de meio período.

A terceira parte envolvida na produção de software é o **usuário**. Usuário é a pessoa ou as pessoas em cujo nome o cliente contratou o produto e aquele(s) que irá(ão) utilizar o software. No exemplo da companhia de seguros, os usuários serão os corretores de seguros que usarão o software para selecionar a apólice mais apropriada. Em alguns casos, o cliente e o usuário são a mesma pessoa (por exemplo, o contador mencionado anteriormente).

Em contraposição aos softwares personalizados de alto custo, desenvolvidos apenas para um cliente, várias cópias de software, como processadores de texto ou planilhas, são vendidas a preços muito mais baixos para um grande número de compradores. Isto é, os fabricantes de software desse tipo (como a Microsoft ou a Borland) recuperam o custo de desenvolvimento de um produto pela venda em grande volume. Esse tipo de software normalmente é chamado **software comercial de prateleira**, ou **COTS** (sigla para commercial off-the-shelf). No início, o termo para esse tipo de software era **software em caixa**, pois vinha em uma caixa que continha o CD ou os disquetes, os manuais e o contrato de licença, embalada com plástico retrátil. Hoje em dia, o software de prateleira, ou COTS, normalmente é obtido por download pela Internet, e não há nenhuma caixa embalada com plástico. Por essa razão, o software de prateleira hoje em dia, por vezes, é denominado **clickware** (acessível por um toque de dedo). Software de prateleira é desenvolvido para "o mercado", isto é, não existem clientes ou usuários específicos até que o software tenha sido desenvolvido e esteja disponível para compra.

O **software aberto** está se tornando muito popular. Um produto de software aberto é desenvolvido e mantido por uma equipe de voluntários e pode ser obtido por download na Internet e usado gratuitamente por qualquer pessoa. Entre os produtos abertos largamente usados há o sistema operacional Linux, o navegador Firefox e o servidor Web Apache. O termo *código-fonte aberto* refere-se à disponibilidade do código-fonte para todos, diferentemente da maioria dos produtos comerciais em que apenas a versão executável é vendida. Como qualquer usuário de um produto aberto pode examinar o código-fonte e relatar falhas aos desenvolvedores, muitos produtos de software aberto são de alta qualidade. A conseqüência esperada da natureza pública das falhas no software aberto foi formalizada por Raymond no

Cathedral and the Bazaar como *Lei de Linux*, nome dado em homenagem a Linus Torvalds, o criador do Linux (Raymond, 2000). A **Lei de Linux** afirma que "com um bom número de olhos, todos os bugs são superficiais". Em outras palavras, se um bom número de indivíduos examinar o código-fonte de um produto de software aberto, alguém será capaz de localizar uma determinda falha e sugerir como corrigi-la. Um princípio relacionado é: Lance o produto logo. Lance o produto freqüentemente. (Raymond, 2000). Ou seja, os desenvolvedores de sistemas com código-fonte aberto tendem a gastar menos tempo com testes que os desenvolvedores de código fechado, preferindo lançar uma nova versão de um produto praticamente assim que ele estiver pronto, deixando grande parte da responsabilidade pelos testes para os usuários.

Uma palavra usada em praticamente todas as páginas deste livro é **software**. O software consiste não apenas de um código em forma legível pelas máquinas como também de toda a documentação, que é parte integrante de qualquer projeto. O software inclui o documento de especificações, o documento de projeto, documentos legais e contábeis, assim como todos os tipos de manuais.

Desde a década de 1970, a diferença entre **programa** e **sistema** tornou-se pouco clara. Nos "bons e velhos tempos", a distinção era clara. Um programa era um trecho autônomo de código, geralmente na forma de uma pilha de cartões perfurados que poderiam ser executados. Um sistema era um conjunto de programas relacionados e poderia ser constituído pelos programas P, Q, R e S. A fita magnética F_1 era posicionada e então o programa P era executado. Isso fazia com que uma pilha de cartões de dados fosse lida e produzia-se como saída as fitas F_1 e F_3. A fita F_2 era então rebobinada e o programa Q era executado, produzindo a fita T_4 como saída. O programa R então juntava as fitas F_3 e F_4 na fita F_5; F_5 servia como entrada para o programa S, que imprimia uma série de relatórios.

Compare essa situação com um produto que é executado em uma máquina com um processador de comunicação front-end e um gerenciador de banco de dados back-end que controla em tempo real uma siderúrgica. O único bloco de software que controla a siderúrgica faz muito mais do que o sistema antigo, porém, em termos das definições clássicas de programa e de sistema, esse software é, sem dúvida nenhuma, um programa. Para criar ainda mais confusão, o termo *sistema* agora também é usado para representar uma combinação de hardware e software. Por exemplo, o sistema de controle de vôo em uma aeronave é formado tanto pelos computadores de bordo quanto pelo software que é executado neles. Dependendo de quem estiver usando o termo, o sistema de controle de vôo também pode incluir os controles, como o manche, que envia comandos para o computador e para as partes do avião, como os flaps das asas, controladas pelo computador. Além disso, dentro do contexto de desenvolvimento de software tradicional, o termo **análise de sistemas** se refere às duas primeiras fases (fase de levantamento de necessidades e fase de análise) e **projeto de sistemas** se refere à terceira fase (fase de projeto).

Para minimizar a confusão, este livro usa o termo **produto** para representar um bloco de software não trivial. Há duas razões para usarmos essa convenção. A primeira é simplesmente para eliminar a confusão entre programa e sistema, usando um terceiro termo. A segunda razão é mais importante. Este livro descreve o **processo** de produção de software, e o resultado final de um processo é denominado *produto*. Finalmente, o termo *sistema* é usado em seu sentido moderno, isto é, a combinação entre hardware e software, ou como parte de termos universalmente aceitos, como sistema operacional e sistema de informações gerenciais.

Dois termos amplamente usados no contexto da engenharia de software são *metodologia* e *paradigma*. Na década de 1970, o termo **metodologia** começou a ser usado no sentido de "uma maneira de desenvolver um produto de software"; o termo significa, realmente, a "ciência dos métodos". Depois, nos anos 1980, o termo **paradigma** tornou-se a palavra da moda do mundo dos negócios, como na frase: "Trata-se de um paradigma totalmente novo". O setor de software rapidamente começou a usar o termo *paradigma* em frases como **paradigma**

Vale a pena saber — Quadro 1.6

A primeira vez que se usou a palavra bug foi para indicar uma falha atribuída à falecida contra-almirante Grace Murray Hopper, uma das criadoras do COBOL. Em 9 de setembro de 1945, uma mariposa voou para dentro do computador Mark II, que Hopper e seus colegas usavam em Harvard, e se alojou entre as placas de contato de um relé. Portanto, existia realmente um *bug* (inseto) no sistema. Hopper grudou o inseto com um fita adesiva no livro de ocorrências do sistema e escreveu: "Primeiro caso encontrado de bug". O livro de ocorrências, com o inseto ainda lá grudado, encontra-se no museu do Naval Surface Weapons Center, em Dahlgren, Virgínia.

Apesar de essa ter sido, talvez, a primeira vez em que a palavra bug foi usada no contexto dos computadores, a palavra já era usada na gíria da engenharia no século XIX (Shapiro, 1994). Por exemplo, Thomas Alva Edison escreveu em 18 de novembro de 1878: "Essa coisa pára de funcionar, e então — 'Bugs' — como essas pequenas falhas e dificuldades são chamadas..." (Josephson, 1992). Uma definição de bug na edição de 1934 do *Webster's New English Dictionary* é: "Defeito em um aparelho ou em sua operação". Fica claro pelo comentário de Hopper que ela também já estava familiarizada com o emprego da palavra nesse contexto; caso contrário, ela teria explicado o seu significado.

de orientação a objetos e **paradigma clássico** (ou **tradicional**) para representar "um estilo de desenvolvimento de software". Essa foi mais uma escolha infeliz de terminologia porque paradigma é um modelo ou um padrão. Leitores eruditos que se sentirem ofendidos por essa corruptela do idioma são convidados calorosamente a contestar em nome do autor em termos de precisão lingüística; ele está cansado de ser um reformador quixotesco.

Metodologias ou paradigmas se aplicam ao processo como um todo, enquanto **técnicas** se aplicam apenas a uma parte do processo de software. Como exemplo, poderíamos citar as técnicas de codificação, de documentação e de planejamento.

Quando um programador comete um **engano**, a conseqüência desse engano é uma **imperfeição** no código. A execução do produto de software resulta então em uma **falha**, isto é, um comportamento incorreto observado no produto como conseqüência da incorreção. Um **erro** é o quanto um resultado é incorreto. Os termos *engano*, *imperfeição*, *falha* e *erro (mistake, fault, failure e error)* são definidos no padrão 610.12 da IEEE, *A Glossary of Software Engineering Terminology* (glossário de terminologia de engenharia de software) (IEEE 610.12, 1990), reconfirmado em 2002 (IEEE Standards, 2003). A palavra **defeito** é um termo genérico que pode se referir a uma falha, uma imperfeição ou um erro. Para maior precisão, neste livro minimizamos o emprego do termo abrangente *defeito*.

Um termo que é evitado o máximo possível é **bug** (a história dessa palavra pode ser encontrada no Quadro 1.6). O termo *bug* hoje em dia é simplesmente um eufemismo para uma *imperfeição*. Embora geralmente não exista nenhum mal em usar eufemismos, a palavra *bug* tem conotações que não são propícias à produção de software de boa qualidade. Mais especificamente, em vez de dizer "Cometi um engano", um programador diria "Apareceu um bug no código" (não no *meu* código, mas *no* código), transferindo, portanto, a responsabilidade do erro do programador para o *bug*. Ninguém joga a culpa em um programador por comparecer ao trabalho com gripe, pois a gripe é causada pelo vírus da gripe (*flu bug*). Referir-se a um engano como um *bug* é uma maneira de se eximir da responsabilidade. Em contraste, o programador que diz "Cometi um engano" é um profissional de informática que assume a responsabilidade por suas ações.

Uma confusão considerável envolve a terminologia sobre o assunto orientação a objeto. Por exemplo, além do termo **atributo,** para indicar um componente de dados de um objeto, algumas vezes é usado o termo **variável de estado** na literatura sobre o assunto orientação a objeto. Em Java, o termo para isso é **variável de instância**. Em C++ é usado o termo **campo**, e no Visual Basic .NET o termo é **propriedade**. Em relação à implementação das operações de um objeto, normalmente, é usado o termo *método*; no C++, entretanto, o termo é **função-membro**. No C++, um membro de um objeto se refere tanto a um atributo

("campo") quanto a um método. Em Java, o termo *campo* é usado para indicar tanto um atributo ("variável de instância") quanto um método. Para evitar confusão, sempre que possível, os termos genéricos *atributo* e *método* são usados neste livro.

Felizmente, certa terminologia é amplamente aceita. Por exemplo, quando um método dentro de um objeto é requisitado, o termo usado quase que universalmente é **enviar uma mensagem** para o objeto.

1.12 Questões Éticas

Concluímos este capítulo com uma advertência. Os produtos de software são desenvolvidos e mantidos por seres humanos. Se esses indivíduos forem árduos trabalhadores, inteligentes, sensíveis, procurarem manter-se atualizados e, acima de tudo, forem *éticos*, então há grande chance de os produtos de software que eles desenvolvem e mantêm serem satisfatórios. Infelizmente, o oposto também é verdadeiro.

A maioria das associações de profissionais possui um código de **ética** ao qual todos os seus membros devem aderir. As duas principais associações de profissionais de informática, a Association for Computing Machinery (ACM) e a Computer Society of the Institute of Electrical and Electronic Engineers (IEEE-CS), aprovaram em conjunto um Código de Ética e Prática Profissional na área de Engenharia de Software como padrão para ensinar e exercer a profissão na área de engenharia de software (IEEE/ACM, 1999). Ele é bem extenso e, portanto, foi gerada uma versão sucinta de um preâmbulo e mais oito princípios. Segue abaixo a versão reduzida:

Código de Ética e Prática Profissional na área de Engenharia de Software[2] (versão 5.2)

conforme recomendado pelo grupo de trabalho formado pelo IEEE-CS/ACM sobre o Código de Ética e Prática Profissional na área de Engenharia de Software

Versão reduzida

Preâmbulo

A versão reduzida do código sintetiza as aspirações em um elevado nível de abstração; as cláusulas que fazem parte da versão integral fornecem exemplos e detalhes de como essas aspirações mudam a maneira pela qual atuamos como profissionais da área de engenharia de software. Sem essas aspirações, os detalhes podem tomar um aspecto meramente legal e enfadonho; sem os detalhes, as aspirações podem se tornar "pomposas" porém vazias; juntos, as aspirações e os detalhes formam um código coeso.

Os engenheiros de software devem se comprometer a transformar a análise, a especificação, o projeto, os testes e a manutenção do software em uma profissão benéfica e respeitada. De acordo com seu comprometimento com a saúde, a segurança e o bem-estar do público, os engenheiros de software devem aderir e cumprir os Oito Princípios abaixo:

1. *Público* — Os engenheiros de software deverão atuar de acordo com o interesse do público.
2. *Cliente e Empregador* — Os engenheiros de software deverão atuar de uma maneira que seja benéfica para o seu cliente e para o seu empregador, bem como de acordo com o interesse do público.
3. *Produto* — Os engenheiros de software deverão garantir que seus produtos e suas respectivas modificações atendam o máximo possível aos padrões profissionais mais elevados.
4. *Avaliações Profissionais* — Os engenheiros de software deverão manter os princípios de integridade e independência em suas avaliações profissionais.

[2] ©1999 pelo Institute of Electrical and Electronics Engineers, Inc. e pela Association for Computing Machinery, Inc.

5. *Gerenciamento* — Os gerentes e líderes de equipes de engenharia de software deverão concordar com e promover um enfoque ético no gerenciamento do desenvolvimento e da manutenção de software.
6. *Profissão* — Os engenheiros de software deverão promover a integridade e a reputação da profissão de acordo com o interesse do público.
7. *Colegas* — Os engenheiros de software deverão ser justos com seus colegas e apoiá-los.
8. *Em relação a si próprio* — Os engenheiros de software deverão adotar uma postura de aprendizagem por toda a vida referente à prática de sua profissão e promover um enfoque ético à prática da profissão.

Os códigos de ética de outras associações de profissionais de computação expressam sentimentos similares. É vital para o futuro de nossa profissão que observemos rigorosamente tais códigos de ética.

No Capítulo 2, examinaremos vários modelos de ciclo de vida para esclarecer ainda mais as diferenças entre o paradigma clássico e o paradigma de orientação a objetos.

Revisão do Capítulo

A engenharia de software é definida (Seção 1.1) como uma disciplina cujo objetivo é a produção de software isento de falhas, entregue dentro de prazo e orçamento previstos, e que atenda às necessidades do cliente. Para atingir tais objetivos devem ser usadas técnicas apropriadas durante a produção de software, inclusive ao realizar a análise (especificação), o projeto (Seção 1.4) e a manutenção pós-entrega (Seção 1.3). A engenharia de software abrange todas as etapas do ciclo de vida de um software e incorpora aspectos de várias outras áreas do conhecimento humano, como economia (Seção 1.2) e ciências sociais (Seção 1.5). Não existem fases de planejamento (Seção 1.6), de testes (Seção 1.7) e de documentação (Seção 1.8) separadas. Na Seção 1.9 é introduzido o conceito de objeto e é feita uma comparação entre os paradigmas clássico e paradigma de orientação a objetos. Em seguida, é avaliado o paradigma de orientação a objetos (Seção 1.10). Depois disso, na Seção 1.11, é explicada a terminologia usada no livro. Finalmente, são discutidas questões éticas na Seção 1.12.

Leitura Complementar

A primeira fonte de informação no âmbito da engenharia de software foi (Boehm, 1976). Para uma análise da abrangência pela qual a engenharia de software pode ser considerada como uma verdadeira disciplina da engenharia, veja (Wasserman, 1996) e (Ebert, Matsubara, Pezzé e Bertelsen, 1997). O futuro da engenharia de software é discutido em (Brereton et al., 1999; Kroeker et al., 1999 e Finkelstein, 2000). O estado atual da prática da engenharia de software é descrito em uma série de artigos na edição de novembro/dezembro de 2003 do *IEEE Software*.

Para ter uma visão sobre a importância da manutenção pós-entrega e de como planejá-la, veja (Parnas, 1994). A falta de confiabilidade do software e os riscos resultantes (especialmente em sistemas em que a segurança é crítica) são discutidos em (Mellor, 1994) e (Newmann, 1995). O desenvolvimento de software para produtos baseados em software de prateleira é o tema de (Brownsword, Oberndorf e Sledge, 2000). A aquisição de componentes de software de prateleira é descrita em (Ulkuniemi e Seppanen, 2004) e em (Keil e Tiwana, 2005).

Os riscos em sistemas empresariais são descritos em (Scott e Vessey, 2002), e os riscos em sistemas de informação em geral, em (Longstaff, Chittister, Pethia e Haimes, 2000). Uma visão moderna da crise de software aparece em (Glass, 1998). Zvegintzov (1998) explica a escassez de dados precisos sobre a prática de engenharia de software.

O fato de a matemática sustentar a engenharia de software é enfatizado em (Devlin, 2001). A importância da economia na engenharia de software é discutida em (Boehm, 1981; Baetjer, 1996 e Boehm e Huang, 2003). A edição de novembro/dezembro de 2002 do *IEEE Software* contém uma série de artigos sobre a economia da engenharia de software.

Dois clássicos sobre ciências sociais e engenharia de software são (Weinberg, 1971) e (Shneiderman, 1980). Nenhum dos dois livros requer conhecimento prévio de psicologia ou de ciência comportamental em geral. Um livro mais recente sobre o tema é (DeMarco e Lister, 1987).

A obra eterna de Brooks (1975), *The Mythical Man-Month*, é uma introdução altamente recomendada às realidades da engenharia de software. O livro inclui seções sobre todos os tópicos mencionados neste capítulo.

Uma excelente introdução sobre software aberto é o livro de (Raymond, 2000). Paulsen, Succi e Eberlein (2004) apresentam um estudo empírico comparando produtos de software abertos e fechados. A reutilização de componentes de software é descrita em (Madanmohan e De', 2004). Uma grande variedade de artigos sobre software aberto aparece na edição de janeiro/fevereiro de 2004 do *IEEE Software* e na edição nº 2 de 2005 do *IBM Systems Journal*.

Excelentes introduções ao paradigma de orientação a objeto são (Meyer, 1997) e (Budd, 2002). Uma perspectiva equilibrada desse paradigma é dada em (Radin, 1996). Khan, Al-A'ali e Girgis (1995) explicam as diferenças entre os paradigmas clássico e da orientação a objetos. Três projetos bem-sucedidos realizados com o emprego do paradigma de orientação a objetos são descritos em (Capper, Colgate, Hunter e James, 1994), com uma análise detalhada. Uma pesquisa sobre a visão de 150 desenvolvedores de software sobre o paradigma de orientação a objetos é relatada em (Johnson, 2000). Lições aprendidas com o desenvolvimento em larga escala de produtos orientados a objetos são apresentadas em (Maring, 1996) e (Fichman e Kemerer, 1997). Possíveis armadilhas do paradigma de orientação a objetos são descritas em (Webster, 1995).

Termos-chave

análise de sistemas, *25*
aperfeiçoamento, *9*
artefato, *19*
atributo, *26*
bug, *26*
campo, *26*
ciclo de vida, *8*
clickware, *24*
cliente, *24*
codificação, *9*
crises de software, *5*
defeito, *26*
definição operacional (de manutenção), *11*
definição temporal (de manutenção), *9*
depressão de software, *7*
descontinuação do produto, *9*
desenvolvedor, *24*
desenvolvimento de software interno, *24*
documento de especificações, *8*
documento de projeto, *9*
encapsulamento, *21*
engano, *26*
engenharia de software, *4*
enviar uma mensagem, *27*
erro, *26*
ética, *27*
falha de regressão, *21*
falha, *26*
fase de especificação, *9*
fase de análise, *9*
fase de documentação, *18*
fase de implementação, *9*
fase de levantamento de necessidades, *8*
fase de planejamento, *17*
fase de projeto, *9*
fase de testes, *18*
fases, *8*
função-membro, *26*
imperfeição, *26*
integração, *9*
Lei de Linux, *25*
manutenção adaptativa, *9*
manutenção corretiva, *9*
manutenção de aperfeiçoamento, *9*
manutenção pós-entrega, *9*
manutenção, *11*
mensagem, *20*
método, *20*
metodologia, *25*
modelo de ciclo de vida, *8*
modelo desenvolvimento-depois-manutenção, *9*
modelo cascata, *8*
módulo, *9*
paradigma clássico, *19*
paradigma de orientação a objetos, *25-26*
paradigma estruturado, *19*
paradigma tradicional, *26*
paradigma, *25*
plano de gerenciamento de projeto de software, *9*
processo, *25*
produto, *25*
programa, *25*
projeto de arquitetura, *9*
projeto de sistemas, *25*
projeto detalhado, *9*
projeto dirigido por responsabilidade, 21
projeto por contrato, *21*
propriedade, *26*
qualidade, *18*
reparo de software, *9*
sistema, *25*
software aberto, *24*

software comercial de prateleira (COTS), *24*
software em caixa, *24*
software por contrato, *24*
software, *25*
técnicas, *26*
teste de aceitação, *9*
teste de unidades, *9*
usuário, *24*
validação, *18*
variável de estado, *26*
variável de instância, *26*
verificação, *18*

Exercícios

1.1 Suponha que você seja encarregado da automação de uma grande padaria. O custo para desenvolvimento de software foi estimado em $425.000. Aproximadamente, quanto a mais de dinheiro será necessário para fazer a manutenção pós-entrega do software?

1.2 Existe uma maneira de conciliar a definição temporal clássica de manutenção e a definição operacional usada hoje em dia? Justifique sua resposta.

1.3 Você é um consultor em engenharia de software. O diretor de uma distribuidora regional de combustíveis quer que você desenvolva um produto de software que realize todas as funções contábeis da empresa e forneça informações on-line ao pessoal da matriz sobre pedidos e estoque dos vários tanques de armazenamento da empresa. São necessários computadores para 21 assistentes de contabilidade, para 15 auxiliares de pedidos e para 37 assistentes de tanques de depósito. Além disso, 14 gerentes precisam acessar esses dados. A empresa está disposta a pagar $30.000 pelo hardware e software juntos, e quer o produto de software concluído em quatro semanas. O que você diria a eles? Tenha em mente que a sua empresa quer esse cliente a qualquer custo, não importando quais sejam as suas solicitações.

1.4 Você é vice-almirante da Marinha de um determinado país. Foi decidido que seria chamada uma empresa desenvolvedora de software para produzir o software de controle de uma nova geração de mísseis navio a navio. Você está encarregado da supervisão do projeto. De modo a proteger o governo de seu país, que cláusulas você incluiria no contrato com os desenvolvedores de software?

1.5 Você é um engenheiro de software cuja função é supervisionar o desenvolvimento do software tratado no Exercício 1.4. Enumere maneiras pelas quais sua empresa poderia falhar no cumprimento das cláusulas contratuais do contrato com a Marinha. Quais são as possíveis causas para tais defeitos?

1.6 Sete meses após a entrega, foi detectada uma imperfeição no software de um produto que analisa DNA usando o reagente Stein-Röntgen. O custo para corrigir a imperfeição é de $16.700. A causa da imperfeição é uma frase ambígua no documento de especificações. Quanto teria custado aproximadamente para corrigir a imperfeição durante a fase de análise?

1.7 Suponha que a imperfeição do Exercício 1.6 tenha sido detectada durante a fase de implementação. Quanto teria custado, aproximadamente, para corrigir a imperfeição nessa fase?

1.8 Suponha que você seja o presidente de uma empresa que produz software em larga escala. Você mostra a Figura 1.6 a seus funcionários, estimulando-os a encontrarem imperfeições logo no início do ciclo de vida do software. Alguns argumentam que não é razoável esperar que alguém elimine imperfeições antes de haver introduzido o produto. Por exemplo, como alguém poderia eliminar uma imperfeição enquanto o projeto está sendo produzido, se a imperfeição na questão está no código? O que você responderia?

1.9 Descreva uma situação em que o cliente, o desenvolvedor e o usuário são a mesma pessoa.

1.10 Quais problemas podem surgir se o cliente, o desenvolvedor e o usuário forem a mesma pessoa? Como esses problemas podem ser resolvidos?

1.11 Quais são as potenciais vantagens se o cliente, o desenvolvedor e o usuário forem a mesma pessoa?

1.12 Procure a palavra *sistema* no dicionário. Quantas definições diferentes existem lá? Anote as definições que são aplicáveis no contexto de engenharia de software.

1.13 Hoje é o seu primeiro dia em seu primeiro emprego. Seu gerente lhe passa a listagem de um programa e diz: “Veja se você consegue encontrar um bug”. O que você responderia?

1.14 Você é responsável pelo desenvolvimento do produto do exercício 1.1. Você usaria o paradigma clássico ou o paradigma de orientação a objeto? Justifique sua resposta.

1.15 Em vez de implementar o componente c9 de um produto de software, os desenvolvedores decidiram comprar um componente de software de prateleira com as mesmas especificações do componente c9. Quais são as vantagens e as desvantagens desse enfoque?

1.16 Em vez de implementar o componente c37 de um produto de software, seus desenvolvedores decidiram optar pelo uso de um componente aberto com as mesmas especificações do componente c37. Quais são os prós e os contras dessa abordagem?

1.17 Suponha que o produto para a empresa Osric's Office Appliances e para a Décor, do Apêndice A, tivesse sido implementado exatamente como descrito. Agora a Osric quer que o produto seja modificado de modo que a prioridade de um cliente na fila possa ser alterada manualmente. Em que pontos o produto existente terá de ser modificado? Seria melhor descartar tudo e começar de novo da estaca zero?

1.18 (Literatura sobre Engenharia de Software) A partir do texto Schach et al (2003b). Qual é sua opinião sobre os méritos relativos dos resultados baseados nas estimativas de gerentes comparados aos resultados computados a partir de dados reais?

Referências Bibliográficas

(Baetjer, 1996) Baetjer, H. *Software as Capital: An Economic Perspective on Software Engineering.* IEEE Computer Society Press, Los Alamitos, CA, 1996.

(Boehm, 1976) Boehm, B. W. Software Engineering. *IEEE Transactions on Computers* **C-25** (dezembro 1976), p. 1226–41.

(Boehm, 1979) Boehm, B. W. Software Engineering, R & D Trends and Defense Needs. In: Wegner, P. (Ed.) *Research Directions in Software Technology*. The MIT Press, Cambridge, MA, 1979.

(Boehm, 1980) Boehm, B. W. Developing Small-Scale Application Software Products: Some Experimental Results. *Proceedings of the Eighth IFIP World Computer Congress,* outubro 1980, p. 321-26.

(Boehm, 1981) Boehm, B. W. *Software Engineering Economics.* Prentice Hall, Englewood Cliffs, NJ, 1981.

(Boehm e Huang, 2003) Boehm, B. e Huang, L. G. Value-Based Software Engineering: A Case Study. *IEEE Computer* **36** (março 2003), p. 33-41.

(Brereton et al., 1999) Brereton, P., Budgen, D., Bennett, K., Munro, M., Layzell, P., Macaulay, L., Griffiths, D. e Stannett, C. The Future of Software. *Communications of the ACM* **42** (dezembro 1999), p. 78-84.

(Brooks, 1975) Brooks, Jr. F. P. *The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley, Reading, MA, 1975; Twentieth Anniversary Edition, Addison-Wesley, Reading, MA, 1995.

(Brownsword, Oberndorf e Sledge, 2000) Brownsword, L., Oberndorf, T. e Sledge, C. A.. Developing New Process for COTS-Based Systems. *IEEE Software* **17** (julho-agosto 2000), p. 40-47.

(Budd, 2002) Budd, T. A. *An Introduction to Object-Oriented Programming*. 3ª ed. Addison-Wesley, Reading, MA, 2002.

(Capper, Colgate, Hunter e James, 1994) Capper, N. P., Colgate, R. J., Hunter, J. C. e James, M. F. The Impact of Object-Oriented Technology on Software Quality: Three Case Histories. *IBM Systems Journal* **33** (nº. 1, 1994), p. 131-57.

(Cutter Consortium, 2002) Cutter Consortium. 78% of IT Organizations Have Litigated *The Cutter Edge,* <www.cutter.com/research/2002/edge020409.html>,[3] 9 de abril de 2002.

(Daly, 1977) Daly, E. B. Management of Software Development. *IEEE Transactions on Software Engineering* **SE-3** (maio 1977), p. 229-42.

(DeMarco e Lister, 1987) Demarco, T. e Lister, T. *Peopleware:* Productive Projects and Teams. Dorset House, Nova York, 1987.

(Devlin, 2001) Devlin, K. The Real Reason Why Software Engineers Need Math. *Communications of the ACM* **44** (outubro 2001), p. 21-22.

(Ebert, Matsubara, Pezzé e Bertelsen, 1997) Ebert, C., Matsubara, T., Pezzé, M. e Bertelsen, O. W. The Road to Maturity: Navigating between Craft and Science. *IEEE Software* **14** (novembro-dezembro 1997), p. 77-88.

(Elshoff, 1976) Elshoff, J. L. An Analysis of Some Commercial PL/I Programs. *IEEE Transactions on Software Engineering* **SE-2** (junho 1976), p. 113-20.

(Fagan, 1974) Fagan, M. E. Design and Code Inspections and Process Control in the Development of Programs. Technical Report IBM-SSD TR 21.572, IBM Corporation, dezembro de 1974.

(Fichman e Kemerer, 1997) Fichman, R. G. e Kemerer, C. F. Object Technology and Reuse: Lessons from Early Adopters. *IEEE Computer* **30** (julho 1997), p. 47-57.

(Finkelstein, 2000) Finkelstein, A. (Ed.). *The Future of Software Engineering.* IEEE Computer Society Press, Los Alamitos, CA, 2000.

(GJSentinel.com, 2003) Sallie Mae's Errors Double Some Bills. Disponível em: <www.gjsentinel.com/news/ content/coxnet/headlines/0522_salliemae.html>, 22 de Maio de, 2003.

(Glass, 1998) Glass, R. L. Is There Really a Software Crisis?. *IEEE Software* **15** (janeiro-fevereiro 1998), p. 104-5.

(Grady, 1994) Grady, R. B. Successfully Applying Software Metrics. *IEEE Computer* **27** (setembro 1994), p. 18-25.

(Hatton, 1998) Hatton, L. Does OO Sync with How We Think?. *IEEE Software* **15** (maio-junho 1998), p. 46-54.

(Hayes, 2004) Hayes, F. Chaos Is Back. *Computerworld*. Disponível em: <www.computerworld.com/managementtopics/management/project/story/0,10801,97283,00.html>, 8 de novembro de, 2004.

(IEEE 610.12, 1990) A Glossary of Software Engineering Terminology. IEEE 610.12-1990. Institute of Electrical and Electronic Engineers, Inc., 1990.

(IEEE Standards, 2003) Products and Projects Status Report. Disponível em: < standards.ieee.org/db/status/status.txt>, 3 de junho de, 2003.

(IEEE/ACM, 1999) Software Engineering Code of Ethics and Professional Practice, versão 5.2, como recomendado por IEEE-CS/ACM Joint Task Force on Software Engineering Ethics and Professional Practice. Disponível em: <www.computer.org/tab/seprof/code.htm>, 1999.

(IEEE/EIA 12207.0-1996, 1998) IEEE/EIA 12207.0-1996 Industry Implementation of International Standard ISO/IEC 12207:1995. Institute of Electrical and Electronic Engineers, Electronic Industries Alliance, Nova York, 1998.

(ISO/IEC 12207, 1995) ISO/IEC 12207:1995, Information Technology—Software Life-Cycle Processes. International Organization for Standardization, International Electrotechnical Commission, Geneva, 1995.

[3] Esse e todos os outros URL citados nesse livro estavam corretos no momento da preparação do livro. Entretanto, endereços Web podem ser alterados com freqüência e sem notificação prévia ou posterior. Se isso ocorrer, o leitor deve fazer uso de um sistema de busca para localizar um novo URL quando necessário.

(Johnson, 2000) Johnson, R. A. The Ups and Downs of Object-Oriented System Development. *Communications of the ACM* **43** (outubro 2000), p. 69-73.

(Josephson, 1992) Josephson, M. *Edison, A Biography.* John Wiley e Sons, Nova York, 1992.

(Kan et al., 1994) Kan, S. H., Dull, S. D., Amundson, D. N., Lindner R. J. e Hedger, R. J. AS/400 Software Quality Management. *IBM Systems Journal* **33** (nº. 1, 1994), p. 62-88.

(Keil e Tiwana, 2005) M. Keil e A. Tiwana. Beyond Cost: The Drivers of COTS Application Value. *IEEE Software* **22** (maio-junho 2005), p. 64-69.

(Kelly, Sherif e Hops, 1992) Kelly, J. C., Sherif, J. S. e Hops, J. An Analysis of Defect Densities Found during Software Inspections. *Journal of Systems and Software* **17** (janeiro 1992), p. 111-17.

(Khan, Al-A'ali e Girgis, 1995) Khan, E. H., Al-A'Ali, M. e Girgis, M. R. Object-Oriented Programming for Structured Procedural Programming. *IEEE Computer* **28** (outubro 1995), p. 48-57.

(Kroeker et al., 1999) Kroeker, K. K., Wall, L., Taylor, D. A., Horn, C., Bassett, P., Ousterhout, J. K., Griss, M. L., Soley, R. M., Waldo, J. e Simonyi, C. Software (R)evolution: A Roundtable. *IEEE Computer* **32** (maio 1999), p. 48-57.

(Leveson e Turner, 1993) Leveson, N. G. e Turner, C. S.. An Investigation of the Therac-25 Accidents. *IEEE Computer* **26** (julho 1993), p. 18-41.

(Lientz, Swanson e Tompkins, 1978) Lientz, B. P., Swanson, E. B. e Tompkins, G. E. Characteristics of Application Software Maintenance. *Communications of the ACM* **21** (junho 1978), p. 466-71.

(Longstaff, Chittister, Pethia e Haimes, 2000) Longstaff, T. A., Chittister, C., Pethia, R. e Haimes, Y. Y. Are We Forgetting the Risks of Information Technology?. *IEEE Computer* **33** (dezembro 2000), p. 43-51.

(Madanmohan e De', 2004) Madanmohan, T. R. e De', R. Open Source Reuse in Commercial Firms. *IEEE Software* **21** (novembro-dezembro 2004), p. 62-69.

(Maring, 1996) Maring, B. Object-Oriented Development of Large Applications. *IEEE Software* **13** (maio 1996), p. 33-40.

(Mellor, 1994) Mellor, P. CAD: Computer-Aided Disaster. Technical Report, Centre for Software Reliability, City University, London, julho de 1994.

(Meyer, 1992) Meyer, B. Applying 'Design by Contract'. *IEEE Computer* **25** (outubro 1992), p. 40-51.

(Meyer, 1997) Meyer, B. *Object-Oriented Software Construction*. 2ª ed., Prentice Hall, Upper Saddle River, NJ, 1997.

(Naur, Randell e Buxton, 1976) Naur, P., Randell, B. e Buxton, J. N. (Ed.). *Software Engineering: Concepts and Techniques: Proceedings of the NATO Conferences*. Petrocelli-Charter, Nova York, 1976.

(Neumann, 1980) Neumann, P. G. Letter from the Editor. *ACM SIGSOFT Software Engineering Notes* **5** (julho 1980), p. 2.

(Neumann, 1995) Neumann, P. G. *Computer-Related Risks*. Addison-Wesley, Reading, MA, 1995.

(Parnas, 1994) Parnas, D. L. Software Aging. *Proceedings of the 16th International Conference on Software Engineering*, Sorrento, Itália, maio de 1994, p. 279-87.

(Paulson, Succi e Eberlein, 2004) Paulson, J. W., Succi, G. e Eberlein, A. An Empirical Study of Open-Source and Closed-Source Software Products. *IEEE Transactions on Software Engineering* **30** (abril 2004), p. 246-56.

(Radin, 1996) Radin, G. Object Technology in Perspective. *IBM Systems Journal* **35** (nº. 2, 1996), p. 124-26.

(Raymond, 2000) Raymond, E. S. *The Cathedral and the Bazaar*: *Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly & Associates, Sebastopol, CA, 2000. Disponível em: <www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/>.

(Schach et al., 2002) Schach, S. R., Jin, B., Wright, D. R., Heller, G. Z. e Offutt, A. J. Maintainability of the Linux Kernel. *IEE Proceedings—Software* **149** (fevereiro 2002), p. 18-23.

(Schach et al., 2003b) Schach, S. R., Jin, B., Heller, G. Z., Yu, L. e Offutt, J. Determining the Distribution of Maintenance Categories: Survey versus Measurement. *Empirical Software Engineering* **8** (dezembro 2003), p. 351-66.

(Scott e Vessey, 2002) Scott, J. E. e Vessey, I. Managing Risks in Enterprise Systems Implementations. *Communications of the ACM* **45** (abril 2002), p. 74-81.

(Shapiro, 1994) Shapiro, F. R. The First Bug. *Byte* **19** (abril 1994), p. 308.

(Shneiderman, 1980) Shneiderman, B. *Software Psychology: Human Factors in Computer and Information Systems*. Winthrop Publishers, Cambridge, MA, 1980.

(Spiegel Online, 2004) Rheinbrücke mit Treppe—54 Zentimeter Höhenunterschied. Disponível em: <www.spiegel.de/panorama/0,1518,281837,00.html>.

(St. Petersburg Times Online, 2003) Thousands of Federal Checks Uncashable. Disponível em: <www.sptimes.com/2003/02/07/Worldandnation/Thousands_of_federal_.shtml>. 7 de fevereiro de 2003.

(Stephenson, 1976) Stephenson, W. E. An Analysis of the Resources Used in Safeguard System Software Development. Bell Laboratories, Draft Paper, agosto de 1976.

(Ulkuniemi e Seppanen, 2004) Ulkuniemi, P. e Seppanen, V. COTS Component Acquisition in an Emerging Market. *IEEE Software* **21** (novembro-dezembro 2004), p. 76-82.

(Wasserman, 1996) Wasserman, A. I. Toward a Discipline of Software Engineering. *IEEE Software* **13** (novembro-dezembro 1996), p. 23-31.

(Webster, 1995) Webster, B. F. *Pitfalls of Object-Oriented Development*. M&T Books, Nova York, 1995.

(Weinberg, 1971) Weinberg, G. M. *The Psychology of Computer Programming*. Van Nostrand Reinhold, Nova York, 1971.

(Wirfs-Brock, Wilkerson e Wiener, 1990) Wirfs-Brock, R., Wilkerson, B. e Wiener, L. *Designing Object-Oriented Software*, Prentice Hall, Englewood Cliffs, NJ, 1990.

(Yourdon, 1992) Yourdon, E. *The Decline and Fall of the American Programmer*. Yourdon Press, Upper Saddle River, NJ, 1992.

(Zelkowitz, Shaw e Gannon, 1979) Zelkowitz, M. V., Shaw, A. C. e Gannon, J. D. *Principles of Software Engineering and Design.* Prentice Hall, Englewood Cliffs, NJ, 1979.

(Zvegintzov, 1998) Zvegintzov, N. Frequently Begged Questions and How to Answer Them. *IEEE Software* **15** (janeiro-fevereiro 1998), p. 93-96.