Capítulo 9

Planejamento e Estimativas

Objetivos de Aprendizagem

Após estudar este capítulo, você deverá ser capaz de:

- Explicar a importância do planejamento.
- Estimar o tamanho e o custo para se criar um produto de software.
- Avaliar a importância das estimativas de atualização e de acompanhamento.
- Preparar um plano de gerenciamento de projeto que se conforme ao padrão IEEE.

Os desafios de criar um produto de software não possuem nenhuma solução fácil. Construir um produto de software grande leva tempo e consome recursos. O planejamento meticuloso no início do projeto talvez seja o único e mais importante fator que ditará seu sucesso ou fracasso. O planejamento inicial, porém, de forma alguma é o suficiente. O planejamento, assim como os testes, deve continuar ao longo do processo do desenvolvimento de software e de manutenção. Não obstante a necessidade de planejamento contínuo, essas atividades atingem um pico após as especificações terem sido formuladas, mas antes de as atividades de projeto começarem. Nesse estágio do processo, são feitas estimativas de custo e duração razoáveis, e é produzido um plano detalhado para completar o projeto.

Neste capítulo, distinguimos esses dois tipos de **planejamento** — o planejamento que prossegue ao longo do projeto e o intenso planejamento que deve ser realizado assim que as especificações estiverem finalizadas.

Observação: o material deste capítulo pode ser ministrado em paralelo com a Parte 2. O material do Capítulo 9 é necessário para o plano de gerenciamento de projeto de software para o estudo de caso da MSG Foundation (Seção 11.15; Exercícios 11.20 e 11.21), e para o estudo do projeto da Osric's Office Appliances and Decor (Exercício 11.16).

9.1 Planejamento e o Processo de Software

De forma ideal, gostaríamos de planejar todo o projeto de software logo no início do processo, e depois seguir com esse plano até o software desejado ter sido finalmente entregue ao cliente. Entretanto, isso é simplesmente impossível, pois faltam informações suficientes durante os fluxos de trabalho iniciais para sermos capazes de elaborar um plano razoável para o projeto completo. Por exemplo, durante o fluxo de trabalho de levantamento de necessidades (que não seja o próprio fluxo de trabalho de levantamento de necessidades), qualquer tipo de planejamento é inútil.

Há uma diferença enorme entre as informações disponíveis para os desenvolvedores no final do fluxo de trabalho de levantamento de necessidades e aquelas no final do fluxo de trabalho de análise, que é análoga à diferença entre um esboço preliminar e um desenho detalhado. No final do fluxo de trabalho de levantamento de necessidades, os desenvolvedores, na melhor das hipóteses, terão um entendimento do que o cliente precisa. De forma contrastante, no final do fluxo de trabalho de análise, em cuja oportunidade o cliente assina um documento declarando exatamente o que será construído, os desenvolvedores têm um entendimento detalhado da maior parte (mas normalmente não de todos) dos aspectos do produto-alvo. Esse é o primeiro ponto no processo em que podem ser determinadas as estimativas precisas de custo e duração do projeto.

Não obstante, em certas situações, pode-se solicitar a uma empresa que produza estimativas de custo e duração antes de as especificações poderem ser redigidas. Na pior hipótese, um cliente poderia insistir em ter uma proposta em um prazo de uma ou duas horas antes da discussão preliminar. A Figura 9.1 mostra o quão problemático isso pode ser. Baseado em um modelo proposto em (Boehm et al., 2000), ela representa o intervalo relativo de estimativas de custo para os diversos fluxos de trabalho do ciclo de vida do produto. Por exemplo, suponhamos que quando um produto passa pelo teste de aceitação no final do fluxo de trabalho de implementação e é entregue ao cliente, determina-se que seu custo é de $ 1 milhão. Se uma estimativa de custo foi feita em um ponto intermediário do fluxo de trabalho de levantamento de necessidades, é provável que essa esteja dentro do intervalo ($ 0,25 milhão – $ 4 milhões), como é mostrado na Figura 9.2. De modo similar, se a estimativa de custo tivesse sido feita no final do fluxo de trabalho de levantamento de necessidades, o intervalo de

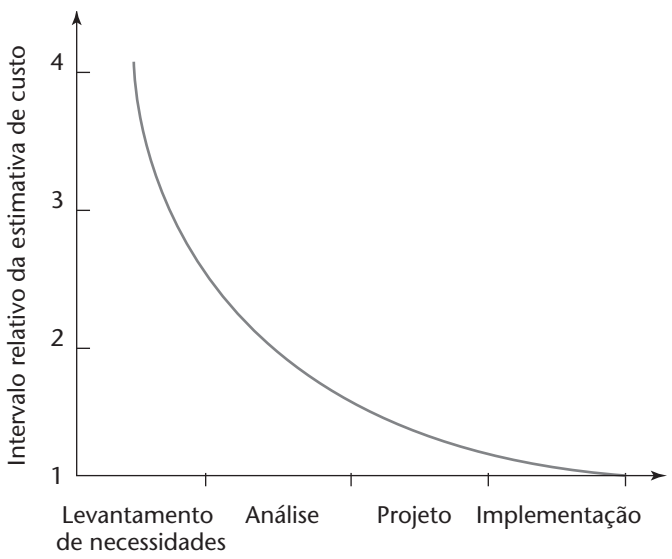

FIGURA 9.1
Um modelo para estimar o intervalo relativo de uma estimativa de custo para cada fluxo de trabalho do ciclo de vida do produto.

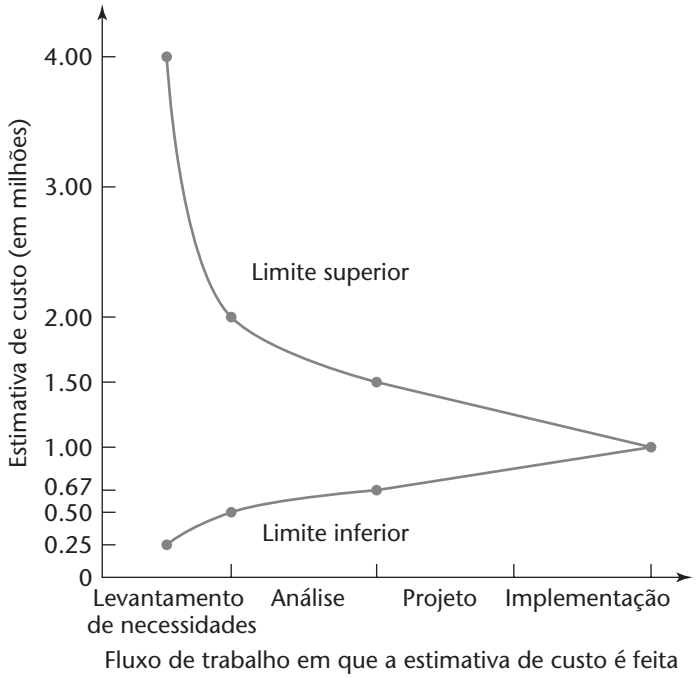

FIGURA 9.2
Intervalo das estimativas de custo para um produto de software que custa $ 1 milhão para ser feito.

estimativas prováveis teria sido reduzido para ($ 0,5 milhão – $ 2 milhões). Além disso, se a estimativa de custo tivesse sido feita no final do fluxo de trabalho de análise, isto é, no momento apropriado, o resultado provavelmente ainda teria caído no intervalo relativamente amplo de ($ 0,67 milhão – $ 1,5 milhão). Todos os quatro pontos são assinalados nas curvas de limite superior e inferior da Figura 9.2, que tem uma escala logarítmica no eixo vertical. Esse modelo é denominado **cone de incerteza**. Fica evidente nas Figuras 9.1 e 9.2 que a estimativa de custo não é uma ciência exata. As razões para isso são apresentadas na Seção 9.2.

Os dados sobre os quais o modelo de cone de incerteza se baseia são antigos, incluindo cinco propostas submetidas à aprovação da Divisão de Sistemas Eletrônicos da Força Aérea Americana (Devenny, 1976), sendo que as técnicas de estimativa foram aperfeiçoadas desde então. Apesar disso, a forma geral da curva da Figura 9.1 provavelmente não mudou tanto. Conseqüentemente, uma estimativa de custo ou duração prematura, isto é, uma estimativa feita antes de as especificações terem sido assinadas pelo cliente, provavelmente será consideravelmente menos acurada do que uma feita quando dados suficientes tiverem sido acumulados.

Examinaremos agora técnicas para estimar a duração e o custo de um projeto. Partiremos do pressuposto, para o restante do capítulo, de que o fluxo de trabalho de análise foi completado, ou seja, é possível realizar estimativas e planejamentos razoáveis.

9.2 Estimativa de Custo e Duração

O orçamento é uma parte fundamental de qualquer plano de gerenciamento de projeto de software. Antes de o projeto ser iniciado, o cliente precisa saber quanto ele terá de pagar pelo produto. Se a equipe de desenvolvimento calcular para menos o custo real, a empresa de desenvolvimento de software poderá perder dinheiro no projeto. De outro lado, se a equipe

de desenvolvimento estimar para mais, então o cliente pode decidir que, em vista de análise de custo–benefício ou de retorno sobre o investimento, não se justifica criar o produto. Ou o cliente pode passar a tarefa para outra empresa de desenvolvimento de software, cujo orçamento seja mais razoável. De ambas as formas, fica claro que realizar estimativas de custo precisas é crítico.

Na realidade, existem dois tipos de custos associados ao desenvolvimento de software. O primeiro deles é o **custo interno**, o **custo** para os desenvolvedores; o segundo é o **custo externo**, o **preço** que o cliente pagará. O custo interno abrange os salários das equipes de desenvolvimento, dos gerentes e do pessoal de apoio envolvido no projeto, o custo de hardware e software para desenvolver o produto e os gastos indiretos como aluguel, energia elétrica, água e salários da alta gerência. Embora o preço geralmente se baseie no custo mais uma margem de lucro, em alguns casos, fatores econômicos e psicológicos são importantes. Por exemplo, os desenvolvedores que precisam desesperadamente do trabalho talvez estejam dispostos a cobrar um preço mais baixo do cliente. Surge uma situação diferente quando um contrato é concedido por intermédio de propostas. Talvez o cliente rejeite uma proposta mais barata em razão de supor que a qualidade do produto resultante seja menor. Uma equipe de desenvolvimento deve, portanto, tentar chegar a uma proposta que seja ligeiramente, mas não significativamente, mais baixa do que ela acredita que sejam as propostas dos concorrentes.

Outra parte importante de qualquer plano é estimar a duração do projeto. Certamente, o cliente vai querer saber quando o produto acabado será entregue. Se as empresas de desenvolvimento de software não conseguirem cumprir o cronograma, então, na melhor das hipóteses, elas perderão credibilidade e, no pior caso, o cliente lançará mão de cláusulas de multa contratual. Em todos os casos, os gerentes responsáveis pelo plano de gerenciamento de projeto de software terão muito o que explicar. Ao contrário, se a empresa de desenvolvimento de software estimar a mais o tempo necessário para entregar o produto, porém, há grandes chances de esse cliente procurar outro fornecedor.

Infelizmente, de forma alguma é fácil estimar custo e duração de forma acurada. Há diversas variáveis envolvidas para sermos capazes de obter uma referência precisa tanto em termos de custo quanto de duração. Uma das grandes dificuldades é o fator humano. Há mais de 35 anos, Sackman e seus colaboradores observaram diferenças de até 28 para 1 entre pares de programadores (Sackman, Erikson e Grant, 1968). É fácil tentar rejeitar seus resultados dizendo que programadores experientes sempre superam os iniciantes, porém Sackman e seus colaboradores compararam pares de programadores equiparados. Eles observaram, por exemplo, dois programadores com dez anos de experiência em tipos de projetos similares, e mediram o tempo que eles levavam para realizar tarefas como codificação e depuração. Em seguida, observaram dois principiantes que já estavam na profissão pelo mesmo curto período de tempo e que tinham currículos acadêmicos semelhantes. Comparando os melhores e os piores desempenhos, eles observaram diferenças de 6 para1 no tamanho do produto, de 8 para1 no tempo de execução do produto, de 9 para1 no tempo de desenvolvimento, de 18 para1 no tempo de codificação e de 28 para 1 no tempo de depuração. Uma observação particularmente alarmante é que os melhores e piores desempenhos em um produto eram de dois programadores, cada um dos quais com 11 anos de experiência. Mesmo quando o melhor e o pior casos eram eliminados da amostra de Sackman, as diferenças observadas ainda estavam na ordem de 5 para1. Com base nesses resultados, fica claro que não podemos ter esperança de estimar custo e duração de projetos de software com qualquer grau de precisão (a menos que tenhamos informações detalhadas referentes a todas as habilidades de todos os funcionários, o que, na maioria das vezes, é incomum). Foi alegado que, em um grande projeto, as diferenças entre os indivíduos tendem a se anular; porém isso talvez seja pura ilusão; a presença de um ou de dois membros de equipe muito bons (ou muito ruins) pode provocar desvios acentuados em relação aos cronogramas e afetar de modo significativo o orçamento.

Outro fator humano que pode afetar as estimativas é que, em um país livre, não há nenhuma forma de garantir que um membro de fundamental importância na equipe não saia da empresa antes de o projeto terminar. Gasta-se, depois, tempo e dinheiro tentando preencher essa vaga e na integração do substituto na equipe ou na reorganização dos membros da equipe remanescentes para compensar a saída do funcionário em questão. De qualquer forma, os cronogramas não são mais cumpridos, e as estimativas fracassam.

Há outra questão subjacente ao problema da estimativa de custo: como se deve medir o tamanho de um produto?

9.2.1 Métrica para o Tamanho de um Produto

A métrica mais comum para o tamanho de um produto é o número de linhas de código. Duas unidades comumente usadas são: **linhas de código** (LOC, em inglês) e **milhares de instruções-fonte entregues** (KSDI, em inglês). Existem muitos problemas associados ao uso de linhas de código (Van der Poel e Schach, 1983).

- A criação de código-fonte é apenas uma pequena parte do esforço total do desenvolvimento de software. Parece pouco convincente que o tempo necessário para os fluxos de trabalho de levantamento de necessidades, análise, projeto, implementação e testes (que incluem atividades de planejamento e de documentação) possa ser expresso única e exclusivamente em função do número de linhas de código do produto final.
- A implementação do mesmo produto em duas linguagens distintas resulta em versões com números de linhas diferentes. Da mesma forma, com linguagens como a Lisp ou várias linguagens não-procedurais de 4ª geração (Seção 14.2), o conceito de linha de código não existe.
- Normalmente, não é exatamente claro como proceder à contagem de linhas de código. Deve-se contar apenas as linhas de código executável ou as definições de dados também? E os comentários, devem ser contados? Em caso negativo, corre-se o risco de o programador relutar em gastar tempo no que ele acredita ser comentários "improdutivos"; porém, se os comentários forem contados, o perigo contrário passa a ser de o programador redigir páginas e páginas de comentários em uma tentativa de aumentar sua aparente produtividade. Da mesma forma, o que dizer da contagem de instruções de linguagem de controle de tarefas? Outro problema é como linhas alteradas ou eliminadas devam ser contadas: durante o aprimoramento do produto para aumentar seu desempenho, algumas vezes o número de linhas é reduzido. A reutilização de código (Seção 8.1) também complica a contagem de linhas: se o código reutilizado for modificado, como ele deve ser contado? E o que acontece se o código for herdado de uma classe-parental (Seção 7.8)? Em suma, a aparentemente objetiva métrica de linhas de código não é nem um pouco direta para se contar.
- Nem todo código produzido é entregue ao cliente. Não é raro que metade do código consista de ferramentas necessárias para dar suporte à atividade de desenvolvimento.
- Suponha que um desenvolvedor de software use um gerador de código como, por exemplo, um gerador de relatórios, um gerador de telas ou um gerador de interfaces gráficas com o usuário (GUI). Após alguns minutos de atividade de projeto por parte do desenvolvedor, a ferramenta poderia gerar milhares de linhas de código.
- O número de linhas de código no produto final pode ser determinado apenas quando o produto estiver completamente pronto. Portanto, tomar como base o número de linhas de código-fonte para fazer estimativas de custo é extremamente perigoso. Para iniciar o processo de estimativas, o número de linhas de código no produto finalizado deve ser estimado. Em seguida, essa estimativa é usada para calcular o custo do produto. Não apenas existe incerteza em qualquer técnica de estimativa como também, se o valor do próprio

estimador de custo for incerto (isto é, o número de linhas de código de um produto que ainda não foi construído), a confiabilidade da estimativa de custo resultante dificilmente será muito grande.

Pelo fato de o número de linhas de código ser tão pouco fidedigno, devemos considerar outras métricas. Outra forma de estimar o tamanho de um produto é o uso de métricas baseadas em quantidades mensuráveis que possam ser determinadas logo no início do processo de software. Por exemplo, Van der Poel e Schach (1983) propuseram a **métrica FFP** para estimar o custo de produtos de processamento de dados de médio porte. Os três elementos estruturais básicos de um produto de processamento de dados são seus arquivos (*files*), fluxos (*flows*) e processos (*processes*); o nome FFP é um acrônimo formado das letras iniciais desses elementos. Um *arquivo* é definido com um conjunto de registros lógica ou fisicamente relacionados, permanentemente residentes no produto; arquivos temporários e de transação são excluídos. *Processo* é uma manipulação lógica ou aritmética de dados funcionalmente definida; alguns exemplos seriam classificação, validação ou atualização de dados. Dado um número de arquivos, *Fi*, de fluxos, *Fl*, e de processos, *Pr*, em um produto, seu tamanho *S* e seu custo *C* são dados por:

$$S = Fi + Fl + Pr \qquad (9.1)$$

$$C = d \times S \qquad (9.2)$$

em que *d* é uma constante que varia de empresa para empresa. A constante *d* é uma medida da **eficiência** (**produtividade**) do processo de software em uma empresa. O tamanho de um produto é simplesmente a soma do número de arquivos, fluxos e processos, uma quantidade que pode ser determinada assim que o projeto de arquitetura estiver pronto. O custo será proporcional ao tamanho, sendo que a constante de proporcionalidade *d* é determinada por um método dos mínimos quadrados aplicado a dados de custos relacionados a produtos desenvolvidos anteriormente por essa empresa. Diferentemente das métricas baseadas no número de linhas de código, o custo pode ser estimado antes de ser iniciado o processo de codificação.

A validade e a confiabilidade da métrica FFP foram demonstradas usando-se uma amostra intencional que cobria um intervalo de aplicações de processamento de dados de médio porte. Infelizmente, a métrica jamais foi estendida de modo a incluir bancos de dados, um componente essencial de vários produtos de processamento de dados.

Uma métrica similar, porém desenvolvida de forma independente, para o tamanho de um produto, foi criada por Albrecht (1979), e baseia-se em pontos de função. A métrica de Albrecht baseia-se no número de itens de entrada, *Inp*, itens de saída, *Out*, consultas, *Inq*, arquivos-mestre, *Maf*, e de interfaces, *Inf*. Em sua forma mais simples, o número de **pontos de função**, *FP*, é dado pela seguinte equação:

$$FP = 4 \times Inp + 5 \times Out + 4 \times Inq + 10 \times Maf + 7 \times Inf \qquad (9.3)$$

Como se trata de uma medida do tamanho do produto, ela pode ser usada para estimativas de custo e de produtividade.

A Equação (9.3) é uma simplificação exagerada de um cálculo feito em três etapas. Primeiramente, são calculados os pontos de função não ajustados:

1. Cada um dos componentes de um produto — *Inp, Out, Inq, Maf* e *Inf* — deve ser classificado como simples, médio ou complexo (veja a Figura 9.3).

2. Atribui-se a cada componente uma série de pontos de função, dependendo de seu nível. Por exemplo, são atribuídos quatro pontos de função a uma entrada média, conforme está representado na Equação (9.3), porém, atribui-se apenas três pontos de função a uma entrada

FIGURA 9.3
Tabela de valores de pontos de função.

	Nível de Compatibilidade		
Componente	**Amostra**	**Média**	**Complexo**
Item de entrada	3	4	6
Item de saída	4	5	7
Consulta	3	4	6
Arquivo-mestre	7	10	15
Interface	5	7	10

FIGURA 9.4
Fatores técnicos para o cálculo de pontos de função.

1. Comunicação de dados
2. Processamento de dados distribuído
3. Critérios de desempenho
4. Hardware utilizado de forma intensiva
5. Taxas de transação elevadas
6. Entrada de dados on-line
7. Eficiência de usuário final
8. Atualização on-line
9. Cálculos complexos
10. Reusabilidade
11. Facilidade de instalação
12. Facilidade de operação
13. Portabilidade
14. Facilidade de manutenção

simples, ao passo que, para uma entrada complexa, são atribuídos seis pontos de função. Os dados necessários para essa etapa são apresentados na Figura 9.3.

3. Os pontos de função atribuídos a cada componente são somados, resultando nos **pontos de função não ajustados** (*UFP*, em inglês).

Em segundo lugar é calculado o **fator de complexidade técnica** (*TCF*, em inglês). Trata-se de uma medida do efeito de 14 fatores técnicos, como taxas de transação elevadas, critérios de desempenho (por exemplo, *throughput* e tempo de resposta) e a atualização on-line; o conjunto completo de fatores é mostrado na Figura 9.4. Atribui-se um valor de 0 ("não está presente ou nenhuma influência") a 5 ("forte influência em tudo") a cada um desses 14 fatores. Os 14 números resultantes são somados, obtendo-se o grau de influência total (*DI*). O *TCF* é dado então por

$$TCF = 0{,}65 + 0{,}01 \times DI \qquad \textbf{(9.4)}$$

Como *DI* pode variar de 0 a 70, o *TCF* varia de 0,65 a 1,35.

	Versão Assembler	Versão Ada
Tamanho do código-fonte	70 KDSI	25 KDSI
Custos de desenvolvimento	$ 1.043.000	$ 590.000
KDSI por homem-mês	0,335	0,211
Custo por instrução-fonte	$ 14,90	$ 23,60
Pontos de função por homem-mês	1,65	2,92
Custo por ponto de função	$ 3,023	$ 1,170

FIGURA 9.5
Comparação de produtos Ada e assembler.
Fonte: (Jones, 1987) (© 1987 IEEE)

Em terceiro lugar, *FP*, o número de pontos de função, é dado por

$$FP = UFP \times TCF \qquad (9.5)$$

Experimentos para medir as taxas de produtividade de software mostraram uma melhor adequação usando-se pontos de função do que usando-se KDSI. Por exemplo, Jones (1987) afirmou ter observado erros acima de 800% na contagem KDSI, mas de *apenas* (grifo nosso) 200% na contagem de pontos de função, um comentário bastante revelador.

Para mostrar a superioridade dos pontos de função em relação ao número de linhas de código, Jones (1987) cita o exemplo mostrado na Figura 9.5. O mesmo produto foi codificado tanto em assembler quanto em Ada, e os resultados foram parecidos. Consideremos, primeiramente, o KDSI por homem-mês. Essa métrica diz-nos que codificar em assembler aparentemente é 60% mais eficiente do que codificar em Ada, o que é, evidentemente, falso. Linguagens de terceira geração, como Ada, suplantaram o assembler simplesmente porque é muito mais eficiente codificar em uma linguagem de terceira geração. Consideremos agora a segunda métrica — custo por instrução-fonte. Note que uma instrução em Ada nesse produto equivale a 2,8 instruções em assembler. O uso de custo por instrução-fonte como medida de eficiência implica novamente ser mais eficiente codificar em assembler do que em Ada. Entretanto, quando se adota pontos de função por homem-mês como métrica de eficiência de programação, a superioridade de Ada em relação ao assembler reflete-se claramente.

De outro lado, tanto os pontos de função quanto a métrica FFP das equações (9.1) e (9.2) sofrem da mesma fraqueza. Freqüentemente, a manutenção do produto é medida de modo inexato. Quando se faz a manutenção de um produto, podem ser feitas mudanças importantes sem alterar o número de arquivos, fluxos e processos ou o número de entradas, saídas, consultas, arquivos-mestre e interfaces. O número de linhas de código não é melhor nesse sentido. Para pegar um caso extremo, é possível substituir cada linha de um produto por uma linha completamente diferente sem alterar o número total de linhas de código.
Foram propostas pelo menos 40 variantes e extensões dos pontos de função de Albrecht (Maxwell e Forselius, 2000). Os pontos de função Mk II foram propostos por Symons (1991) para fornecer uma maneira mais precisa para o cálculo dos pontos de função não ajustados (*UFP*). O software é decomposto em um conjunto de transações de componentes, cada uma formada por uma entrada, um processo e uma saída. O valor de *UFP* é calculado a partir dessas entradas, dos processos e das saídas. Os pontos de função Mk II são amplamente usados ao redor do mundo (Boehm, 1997).

9.2.2 Técnicas de Estimativa de Custo

Não obstante as dificuldades em se estimar o tamanho, é essencial que os desenvolvedores de software simplesmente façam o máximo possível para obter estimativas precisas tanto da duração quanto do custo do projeto, levando em conta, ao mesmo tempo, o maior número possível de fatores que possam afetar suas estimativas. Entre estes há os níveis de capaci-

tação de pessoal, a complexidade do projeto, o tamanho do projeto (o custo aumenta com o tamanho, mas muito do mais que linearmente), a familiaridade da equipe de desenvolvimento com o campo de aplicação, o hardware em que o produto será usado e a disponibilidade de ferramentas CASE. Outro fator é o efeito do prazo de entrega. Se um projeto tem de ser cumprido dentro de certo prazo, o esforço em homem-mês é maior do que se nenhuma restrição fosse imposta sobre o tempo de finalização; portanto, maior o custo. Isso mostra que duração e custo não são independentes: quanto mais curto for o prazo, maior o esforço e, portanto, maior o custo.

Da lista anterior, que de forma alguma pretende ser completa, fica claro que fazer estimativas é um problema difícil. Têm sido usadas várias abordagens, com maior ou menor sucesso.

1. Opinião de Especialistas por Analogia

Na técnica de **opinião de especialistas por analogia**, são consultados vários especialistas. Um especialista chega a uma estimativa comparando o produto desejado a outros finalizados com os quais ele esteve ativamente envolvido e percebendo semelhanças e diferenças. Por exemplo, um especialista poderia comparar o produto desejado a um produto similar desenvolvido há dois anos para os quais os dados foram introduzidos no modo batch, ao passo que o produto desejado deve ter captura de dados on-line. Como a empresa está familiarizada com o tipo do produto a ser desenvolvido, o especialista reduz o tempo de desenvolvimento e o esforço em 15%. Entretanto, a interface gráfica com o usuário é ligeiramente complexa; isso aumenta o tempo e o esforço em 25%. Finalmente, o produto desejado tem de ser desenvolvido em uma linguagem com a qual a maioria da equipe de desenvolvimento não está familiarizada e, portanto, aumentando em 15% o tempo de desenvolvimento e em 20% o esforço. Combinando esses três números, o especialista decide que o produto desejado levará 25% a mais de tempo e 30% a mais de esforço que o anterior. Pelo fato de o produto anterior ter levado 12 meses para ser concluído, e exigido cem homens-mês, estima-se que o produto atual levará 15 meses para ser finalizado e consumirá 130 homens-mês.

Outros dois especialistas da empresa comparam os mesmos dois produtos. Um conclui que o produto desejado levará 13,5 meses para ficar pronto e consumirá 140 homens-mês. O outro chega aos seguintes números: 16 meses e 95 homens-mês. Como as previsões desses três especialistas podem ser conciliadas? Um método é a **técnica Delphi**: ela permite aos especialistas chegarem a um consenso sem ter reuniões de grupo, que pode ter o efeito colateral indesejado de um especialista mais persuasivo influenciar os demais participantes do grupo. Nessa técnica, os especialistas trabalham de modo independente. Cada um deles produz uma estimativa e desenvolve um raciocínio para essa estimativa. Essas estimativas e os raciocínios são distribuídos a todos os especialistas, que então produzem uma segunda estimativa. Esse processo de estimativa e distribuição continua até que os especialistas consigam chegar a um acordo dentro de uma tolerância aceitável. Não acontecem reuniões de grupo durante o processo interativo.

A avaliação de imóveis normalmente é feita usando-se a opinião de especialistas por analogia. Um avaliador chega a um valor comparando um imóvel a imóveis similares que foram vendidos recentemente. Suponha que a casa A seja avaliada, a casa B vizinha acaba de ser vendida por $ 205.000, e a casa C na próxima rua tenha sido vendida três meses antes por $ 218.000. O avaliador poderia raciocinar da seguinte forma: a casa A tem um banheiro a mais que a casa B e o jardim é 464 m^2 maior. A casa C tem aproximadamente o mesmo tamanho da casa A, mas o telhado está em péssimas condições. De outro lado, a casa C tem uma banheira de hidromassagem. Após profunda reflexão, o avaliador poderia chegar a um valor de $ 215.000 para a casa A.

No caso de produtos de software, a opinião de especialistas por analogia é menos precisa do que na avaliação de imóveis. Lembre-se de que nosso primeiro especialista de software alegou que usar uma linguagem com a qual os programadores não estivessem familiarizados poderia aumentar o tempo de desenvolvimento em 15% e o esforço em 20%. A menos que o especialista tenha alguns dados válidos a partir dos quais o efeito de cada diferença possa ser determinado (uma possibilidade muito remota), erros induzidos por aquilo que pode ser descrito como mera adivinhação resultarão em estimativas de custo totalmente incorretas. Além disso, a menos que os especialistas sejam abençoados com uma excelente memória (ou tenham mantido registros detalhados), suas vagas lembranças de produtos finalizados talvez sejam suficientemente imprecisas a ponto de invalidar suas previsões. Por último, os especialistas também são seres humanos e, portanto, podem ter tendências que afetem suas previsões. Ao mesmo tempo, os resultados de estimativas feitas por um grupo devem refletir sua experiência coletiva; se esta for suficientemente ampla, o resultado pode muito bem ser preciso.

2. Abordagem Bottom-up

Uma maneira de tentar reduzir os erros resultantes da avaliação de um produto como um todo é dividi-lo produto em componentes menores. Estimativas de custo e duração são feitas para cada componente separadamente e combinadas para fornecer um número global. Essa **abordagem bottom-up** apresenta a vantagem de que estimar custos para vários componentes geralmente é mais rápido e preciso do que para um único grande componente. Além disso, o processo de estimativa provavelmente é mais detalhado do que um único e grande produto monolítico. O ponto fraco dessa abordagem é que um produto é, na verdade, mais que a simples soma de seus componentes.

Com o paradigma de orientação a objetos, a independência de várias classes ajuda a abordagem bottom-up. Entretanto, interações entre os vários objetos do produto complicam o processo de estimativa.

3. Modelos Algorítmicos de Estimativas de Custo

Nessa abordagem é usada uma métrica como os pontos de função ou a métrica FFP, como entrada para um modelo para determinar o custo do produto. O estimador calcula o valor da métrica; em seguida, estimativas de custo e duração podem ser calculadas usando-se o modelo. Superficialmente, um **modelo algorítmico de estimativas de custo** é superior à opinião de especialistas, pois um ser humano, como foi dito anteriormente, está sujeito a tendências e poderia deixar passar despercebidos certos aspectos tanto do produto finalizado quanto daquele alvo da avaliação. De modo contrastante, um modelo algorítmico de estimativas de custo é imparcial; todos os produtos são tratados da mesma forma. O perigo de um modelo desses é que suas estimativas são boas apenas se suas hipóteses subjacentes o forem. Por exemplo, subjacente ao modelo de pontos de função está a hipótese de que cada aspecto de um produto está incorporado nas cinco quantidades do lado direito da Equação (9.3) e nos 14 fatores técnicos. Outro problema é que, normalmente, é necessário um nível significativo de julgamento subjetivo para decidir quais valores atribuir aos parâmetros do modelo. Por exemplo, freqüentemente não fica claro se um determinado fator técnico do modelo de pontos de função deve ser classificado como 3 ou 4.

Foram propostos diversos modelos de custo algorítmicos. Alguns deles se baseiam em teorias matemáticas sobre como o software é desenvolvido. Outros fundamentam-se em estatísticas; é estudado um grande número de projetos e são determinadas regras empíricas a partir desses dados. Modelos híbridos incorporam equações matemáticas, modelagem estatística e opinião de especialistas por analogia. O híbrido mais importante é o COCOMO de Boehm, que é descrito de forma detalhada na Seção 9.2.3. (Veja o Quadro 9.1 para uma discussão sobre o acrônimo COCOMO.)

Vale a pena saber — Quadro 9.1

COCOMO é um acrônimo formado pelas duas primeiras letras de cada palavra existente em COnstructive COst MOdel. Qualquer conexão com a cidade de Kokomo, Indiana, nos Estados Unidos, é mera coincidência.
MO, em COCOMO, significa "modelo" e, portanto, a expressão modelo COCOMO não deve ser usada. A expressão cai na mesma categoria de "ATM machine" (em inglês, Automatic Teller Machine) e "PIN number" (em inglês, Private Identification Number), ambos idealizados pelo departamento "Departamento de Informações Redundantes".

9.2.3 COCOMO Intermediário

COCOMO é, na verdade, uma série de três modelos que vai desde um modelo de macroestimativa, que trata o produto como um todo, até um modelo de microestimativa, que trata o produto de forma detalhada. Nesta seção, é dada uma descrição do COCOMO intermediário, que possui um nível intermediário de complexidade e detalhe. Ele é descrito detalhadamente em (Boehm, 1981); uma visão geral é apresentada em (Boehm, 1984).

Calcular o tempo de desenvolvimento usando COCOMO intermediário é feito em dois estágios. Primeiramente, fornece-se uma estimativa grosseira do esforço de desenvolvimento. Devem ser estimados dois parâmetros: o comprimento do produto em KDSI e o modo de desenvolvimento do produto, uma medida do nível intrínseco de dificuldade para desenvolver esse produto. Existem três modos: *orgânico* (pequeno e objetivo), *semi-objetivo* (de tamanho médio) e *embutido* (complexo).

A partir desses dois parâmetros, pode ser calculado o **esforço nominal**. Por exemplo, se o projeto for avaliado de forma essencialmente objetiva (orgânico), então o esforço nominal (em homens-mês) será dado pela equação

$$\text{Esforço nominal} = 3{,}2 \times (\text{KDSI})^{1{,}05} \text{ homens-mês} \qquad (9.6)$$

As constantes 3,2 e 1,05 são os valores que melhor se adaptam aos dados nos produtos de modo orgânico usados por Boehm para desenvolver o COCOMO intermediário.

Por exemplo, se o produto a ser construído é orgânico e estima-se que terá 12 mil instruções-fonte entregues (12 KDSI), o esforço nominal será

$$3{,}2 \times (12)^{1{,}05} = 43 \text{ homens-mês}$$

(leia, porém, o Quadro 9.2 para um comentário sobre esse valor).

Em seguida, esse valor nominal deve ser multiplicado por 15 **multiplicadores de esforço de desenvolvimento de software**. Esses multiplicadores e seus valores são mostrados na Figura 9.6. Cada multiplicador pode ter até seis valores. Por exemplo, atribuem-se os valores 0,70, 0,85, 1,00, 1,15, 1,30 ou 1,65 ao multiplicador de complexidade do produto, de acordo com a complexidade de projeto avaliada pelos desenvolvedores, a saber: muito pequena, pequena, nominal (média), grande, muito grande ou extragrande. Como pode ser observado na Figura 9.6, todos os 15 multiplicadores assumem o valor 1,00 quando o parâmetro correspondente for nominal.

Boehm fornece diretrizes para ajudar o desenvolvedor a determinar se o parâmetro deve, de fato, ser classificado como nominal, ou se a classificação deve ser menor ou maior. Por exemplo, consideremos novamente o multiplicador de complexidade de módulo. Se as operações de controle do módulo forem formadas, basicamente, por uma seqüência de construtos de programação estruturada (como **if-then-else**, **do-while**, **case**), a complexidade será considerada *muito pequena*. Se esses operadores forem aninhados, a classificação é *pequena*. Acrescentar controle entre módulos e tabelas de decisão aumenta a classificação para *nominal*. Se os operadores forem altamente aninhados, com predicados

Vale a pena saber — Quadro 9.2

Uma reação ao valor do esforço nominal poderia ser: "Se são necessários 43 homens-mês de esforço para produzir 12 mil instruções-fonte entregues, então, em média, cada programador produz menos de 300 linhas de código por mês. Eu escrevi muito mais que isso em apenas uma noite!".
Um produto de 300 linhas normalmente é simplesmente isso: 300 linhas de código. Ao contrário, para ser mantido, um produto de 12 mil linhas tem de passar por todos os fluxos de trabalho do ciclo de vida. Ou seja, o esforço total de 43 pessoas-mês é dividido entre várias atividades, dentre as quais a codificação.

FIGURA 9.6
Multiplicadores de esforço de desenvolvimento de software para COCOMO intermediário.
Fonte: (Boehm, 1994). (© 1984 IEEE.)

	Classificação					
Alavancadores de custo	**Muito pequena**	**Pequena**	**Nominal**	**Grande**	**Muito grande**	**Extra grande**
Atributos do produto						
Confiabilidade de software exigida	0,75	0,88	1,00	1,15	1,40	
Tamanho do banco de dados		0,94	1,00	1,08	1,16	
Complexidade do produto	0,70	0,85	1,00	1,15	1,30	1,65
Atributos computacionais						
Restrição de tempo de execução			1,00	1,11	1,30	1,66
Restrição de armazenamento na memória principal			1,00	1,06	1,21	1,56
Volatilidade da máquina virtual*		0,87	1,00	1,15	1,30	
Tempo de retorno		0,87	1,00	1,07	1,15	
Atributos pessoais						
Habilidades do analista	1,46	1,19	1,00	0,86	0,71	
Experiência aplicada	1,29	1,13	1,00	0,91	0,82	
Capacidade do programador	1,42	1,17	1,00	0,86	0,70	
Experiência em máquina virtual*	1,21	1,10	1,00	0,90		
Experiência em linguagens de programação	1,14	1,07	1,00	0,95		
Atributos do projeto						
Uso de práticas de programação modernas	1,24	1,10	1,00	0,91	0,82	
Uso de ferramentas de software	1,24	1,10	1,00	0,91	0,83	
Cronograma de desenvolvimento exigido	1,23	1,08	1,00	1,04	1,10	

*Para um dado produto de software, a máquina virtual subjacente é o complexo de hardware e software (sistema operacional, sistema de gerenciamento de bancos de dados) que ele requisita para cumprir sua tarefa.

compostos, filas e pilhas, a classificação passa a ser *alta*. A presença de código reentrante e recursivo, e o tratamento de interrupções de prioridade fixa elevam a classificação para *muito grande*. Finalmente, o cronograma de recursos múltiplos, com prioridades que mudam dinamicamente, e o controle no nível de microcódigo garantem que a classificação seja *extragrande*. Essas classificações aplicam-se às operações de controle. Um módulo também tem de ser avaliado do ponto de vista de operações computacionais, operações dependentes de dispositivos e operações de gerenciamento de dados. Para detalhes sobre os critérios para cálculo de cada um dos 15 multiplicadores, consulte (Boehm, 1981).

Para ver como isso funciona, Boehm (1984) usa como exemplo software de processamento de comunicação baseado em microprocessadores para uma nova e altamente confiável

rede de transferência eletrônica de fundos, com as exigências de interface, desempenho e cronograma de desenvolvimento. Esse produto concorda com a descrição de modo embutido e estima-se que ele tenha 10 mil instruções-fonte entregues (10 KDSI); portanto, o esforço de desenvolvimento nominal é dado por

$$\text{Esforço nominal} = 2{,}8 \times (\text{KDSI})^{1{,}20} \qquad \textbf{(9.7)}$$

(Enfatizando, as constantes 2,8 e 1,20 são os valores que melhor se adaptam aos dados em produtos embutidos.) Pelo fato de estimar-se que o projeto tem 10 KDSI, o esforço nominal é

$$2{,}8 \times (10)^{1{,}20} = 44 \text{ homens-mês}$$

O esforço de desenvolvimento estimado é obtido multiplicando-se o esforço nominal pelos 15 fatores de esforço de desenvolvimento de software. As classificações desses multiplicadores e seus valores são dados na Figura 9.7. Usando esses valores, o produto dos multiplicadores é estimado em 1,35, portanto, o esforço nominal do projeto é

$$1{,}35 \times 44 = 59 \text{ homens-mês}$$

Esse número é usado em outras fórmulas para determinar custos em moeda, cronogramas de desenvolvimento, distribuições de atividades e fases, custos computacionais, custos de manutenção anuais e outros itens relativos; para maiores detalhes, veja (Boehm, 1981). O COCOMO intermediário é um modelo algorítmico de estimativas de custo, que oferece ao usuário praticamente toda a assistência concebível para o planejamento de projetos.

FIGURA 9.7
Classificações de multiplicadores de esforço para o COCOMO intermediário no caso de software de comunicação baseado em microprocessadores.
Fonte: (Boehm, 1984). (© 1984 IEEE.)

Alavancadores de custo	Situação	Classificação	Multiplicador de esforço
Confiabilidade de software exigida	Graves conseqüências financeiras por causa de falha de software	Grande	1,15
Tamanho do banco de dados	20 mil bytes	Pequeno	0,94
Complexidade do produto	Processamento de comunicação	Muito grande	1,30
Restrição do tempo de execução	Usará 70% do tempo disponível	Grande	1,11
Restrição de armazenamento na memória principal	45K dos 64K para armazenamento (70%)	Grande	1,06
Volatilidade da máquina virtual*	Baseia-se em hardware microprocessado comercial	Nominal	1,00
Tempo de retorno	Tempo de retorno médio de duas horas	Nominal	1,00
Habilidades do analista	Excelentes analistas sênior	Grande	0,86
Experiência em aplicações	Três anos	Nominal	1,00
Capacidade do programador	Excelentes programadores sênior	Grande	0,86
Experiência em máquina virtual*	Seis meses	Pequena	1,10
Experiência em linguagens de programação	Doze meses	Nominal	1,00
Uso de práticas de programação modernas	A maioria das técnicas em uso há mais de um ano	Grande	0,91
Uso de ferramentas de software	No nível de ferramentas para microcomputadores básicas	Pequena	1,10
Cronograma de desenvolvimento exigido	Nove meses	Nominal	1,00

O COCOMO intermediário foi validado em relação a uma ampla gama de 63 projetos que envolvem uma grande variedade de áreas de aplicação. Os resultados da aplicação do COCOMO intermediário nessa ampla amostra são que os valores reais caíram num intervalo de mais ou menos 20% dos valores previstos em 68% das vezes. Tentativas de melhorar essa precisão não fizeram muito sentido, pois, na maioria das empresas, os dados de entrada para o intermediário geralmente são precisos apenas no intervalo de aproximadamente 20%. Apesar disso, a precisão obtida por avaliadores experientes classificou-o como o que havia de mais avançado em termos de pesquisa de estimativas de custo na década de 1980; nenhuma outra técnica foi acurada de forma consistente.

O principal problema com este modelo é que sua entrada mais importante é o número de linhas de código no produto-alvo. Se essa estimativa estiver incorreta, então todas as previsões, sem exceção, do modelo podem estar incorretas. Por causa da possibilidade de as previsões do COCOMO intermediário ou de qualquer outra técnica de estimativa poderem ser imprecisas, a gerência deve monitorar todas as previsões ao longo do processo de desenvolvimento do software.

9.2.4 COCOMO II

O COCOMO foi postulado em 1981. Naquela época, o único modelo de ciclo de vida em uso era o cascata. A maioria dos programas rodava em mainframes. Tecnologias como cliente/servidor e orientação a objetos eram, basicamente, desconhecidas. Conseqüentemente, o COCOMO não incorpora nenhum desses fatores. Porém, à medida que tecnologias mais novas começaram a se tornar aceitas como práticas da engenharia de software, ele começou a se tornar menos preciso.

O COCOMO II (Boehm et al., 2000) é uma revisão importante do COCOMO de 1981. O COCOMO II é capaz de lidar com uma ampla variedade de técnicas de engenharia de software modernas, entre as quais a orientação a objetos, os vários modelos de ciclo de vida descritos no Capítulo 2, a prototipagem rápida (Seção 10.13), as linguagens de quarta geração (Seção 14.2), a reutilização (Seção 8.1) e o software de prateleira (Seção 1.11). O Ele é, ao mesmo tempo, flexível e sofisticado. Infelizmente, para atingir esse nível, o COCOMO II é consideravelmente mais complexo que o COCOMO original. Conseqüentemente, o leitor que quiser utilizá-lo deve estudar detalhadamente (Boehm et al., 2000); daremos aqui apenas uma visão geral das principais diferenças entre o COCOMO II e o COCOMO intermediário.

Primeiramente, o COCOMO intermediário consiste de um modelo geral baseado em linhas de código (KDSI). Já, o COCOMO II é formado por três modelos diferentes. O **modelo de composição de aplicação**, baseado em pontos de objeto (similares aos pontos de função), é aplicado nos primeiros fluxos de trabalho, quando existe um conhecimento mínimo sobre o produto a ser construído. Depois, à medida que mais conhecimento torna-se disponível, é usado o modelo de projeto inicial; que se baseia em pontos de função. Finalmente, quando os desenvolvedores tiverem o máximo de informações, é usado o **modelo pós-arquitetura**, que utiliza pontos de função ou linhas de código (KDSI). A saída do COCOMO intermediário é uma estimativa de custo e duração. Conseqüentemente, se a estimativa mais provável do esforço for E, então o modelo de composição de aplicação retorna o intervalo (0,50E, 2,0E), e o modelo pós-arquitetura retorna o intervalo (0,80E, 1,25E). Isso reflete a precisão crescente da progressão dos modelos do COCOMO II.

Uma segunda diferença reside no modelo de esforço por trás do COCOMO:

$$\text{Esforço} = a \times (\text{tamanho})^b \qquad \textbf{(9.8)}$$

em que a e b são constantes. No COCOMO intermediário, o expoente b assume três valores diferentes, dependendo de o modo do produto a ser construído ser orgânico, ($b = 1{,}05$), semi-

objetivo ($b = 1{,}12$) ou embutido ($b = 1{,}20$). No COCOMO II, o valor de b varia entre 1,01 e 1,26, dependendo de uma série de parâmetros do modelo. Isso engloba familiaridade com os produtos desse tipo, nível de maturidade de processo (Seção 3.13), extensão da solução de riscos (Seção 2.7) e o grau de cooperação em equipe (Seção 4.1).

Uma terceira diferença é a hipótese referente à reutilização. O COCOMO intermediário pressupõe que a economia resultante da reutilização é diretamente proporcional ao nível de reutilização. O COCOMO II leva em consideração que pequenas mudanças em software reutilizado incorrem em custos desproporcionalmente elevados (pois o código tem de ser entendido em detalhes mesmo para uma mínima alteração, e o custo de testar um módulo modificado é relativamente grande).

Em quarto lugar, agora existem 17 alavancadores de custo, em vez dos 15 do COCOMO intermediário. Sete dos alavancadores de custo são novos, por exemplo, a reusabilidade exigida em produtos futuros, a rotatividade anual de mão-de-obra e a possibilidade de o produto ser desenvolvido em vários locais.

O COCOMO II foi calibrado usando-se 83 projetos de uma série de domínios diferentes. O modelo ainda é bastante novo para existir muitos resultados referentes à sua precisão e, particularmente, não se sabe até que ponto ele é um avanço em relação ao seu predecessor, o COCOMO original (de 1981).

9.2.5 Acompanhamento das Estimativas de Custo e Duração

Enquanto o produto está sendo desenvolvido, o esforço de desenvolvimento real deve ser constantemente comparado com as previsões. Por exemplo, suponha que a métrica de estimativa usada pelos desenvolvedores de software previu que a **duração** do fluxo de trabalho de análise seria de três meses e exigiria sete homens-mês de esforço. Entretanto, já se passaram quatro meses e foram empregados dez homens-mês de esforço, muito embora as especificações não estejam completas. Desvios desse tipo podem servir como um primeiro alerta de que algo deu errado, e uma ação corretiva deve ser tomada imediatamente. O problema poderia ser que o tamanho do produto foi seriamente estimado para menos ou que a equipe de desenvolvimento não é tão competente como se imaginava. Seja qual for a razão, o custo e a duração estarão, perigosamente, acima do previsto, e a gerência deve tomar medidas apropriadas para minimizar seus efeitos.

Deve-se fazer um acompanhamento cuidadoso das previsões ao longo de todo o processo de desenvolvimento, independentemente das técnicas usadas para se fazer tais previsões. Os desvios podem se dar pelo uso de métricas que fornecem previsões inadequadas, desenvolvimento de software ineficiente, uma combinação de ambos os fatores ou alguma outra razão. O importante é detectar desvios logo no início e tomar medidas corretivas imediatamente. Além disso, é essencial atualizar continuamente as previsões em vista de informações adicionais, à medida que elas forem se tornando disponíveis.

Agora que as métricas para estimativas de custo e duração foram discutidas, descreveremos os componentes do plano de gerenciamento de projeto de software.

9.3 Componentes de um Plano de Gerenciamento de Projeto de Software

Um plano de gerenciamento de projeto de software possui três componentes principais: o trabalho a ser realizado, o recurso com o qual se realizará o trabalho e o dinheiro a ser gasto nele. Nesta seção, esses três ingredientes do plano são discutidos. A terminologia é extraída de (IEEE 1058, 1998), que será discutido de forma mais detalhada na Seção 9.4.

O desenvolvimento de software requer *recursos*. Os principais **recursos** necessários são as pessoas que desenvolverão o software, o hardware no qual o software rodará e um

software de apoio como sistemas operacionais, editores de texto e software de controle de versões (Seção 5.7).

O emprego de recursos como pessoal varia com o tempo. Norden (1958) demonstrou que, para grandes projetos, a **distribuição de Rayleigh** é uma boa aproximação, da maneira que o consumo de recursos, R_c, varia com o tempo, t, isto é,

$$R_c = \frac{t}{k^2} e^{-t^2/2k^2} \qquad 0 \leq t < \infty \qquad (9.9)$$

O parâmetro k é uma constante, o tempo em que o consumo está em seu pico, e $e = 2{,}71828...$, a base (natural) dos logaritmos neperianos. O consumo de recursos começa pequeno, sobe rapidamente até um pico e depois diminui a uma taxa mais lenta. Putnam (1978)investigou a aplicabilidade dos resultados de Norden ao desenvolvimento de software e descobriu que o consumo de pessoal e de outros recursos era modelado com certo grau de precisão pela distribuição de Rayleigh.

Portanto, é insuficiente, em um plano de software, meramente afirmar que são necessários três programadores-seniores com pelo menos cinco anos de experiência. Precisa-se de algo como o seguinte:

> São necessários três programadores-seniores com pelo menos cinco anos de experiência em programação em tempo real, dois para iniciar três meses após o projeto começar, o terceiro para começar seis meses depois disso. Dois serão afastados quando os testes de produtos começarem, e o terceiro, quando a manutenção pós-entrega for iniciada.

O fato de as necessidades de recurso dependerem do tempo aplica-se não apenas ao pessoal, mas também ao tempo de processamento, software de apoio, hardware computacional, instalações de escritório e até mesmo às viagens. Conseqüentemente, o plano de gerenciamento de projeto de software é uma função do tempo.

O trabalho a ser realizado cai em duas categorias. A primeira delas é o trabalho que continua ao longo do projeto e não se relaciona com nenhum fluxo de trabalho específico do desenvolvimento de software. Tal trabalho é denominado **função de projeto**. Exemplos são

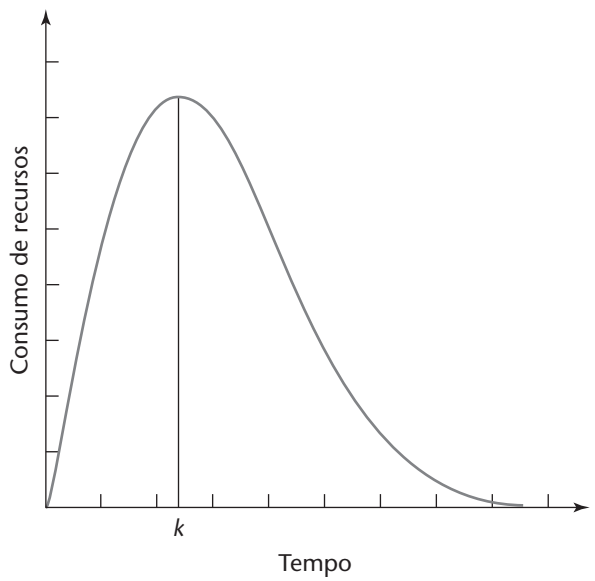

FIGURA 9.8
Curva de Rayleigh que mostra como o consumo de recursos varia com o tempo.

o gerenciamento de projeto e o controle de qualidade. Em segundo lugar, há o trabalho que depende de um determinado fluxo de trabalho no desenvolvimento do produto, por exemplo, denominado *atividade* ou *tarefa*. **Atividade** é uma unidade de trabalho principal que tem datas precisas de início e fim; consome recursos, como tempo de processamento ou homens-dia, e resulta em **produtos de trabalho**, um orçamento, documentos de projeto, cronogramas, código-fonte ou um manual de usuário. Por sua vez, uma atividade compreende um conjunto de tarefas, sendo que **tarefa** é a menor unidade de trabalho sujeita à responsabilidade da gerência. Existem, portanto, três tipos de trabalho em um plano de gerenciamento de projeto de software: funções de projeto realizadas ao longo do projeto, atividades (unidades de trabalho principais) e tarefas (unidades de trabalho secundárias).

Um aspecto crítico do plano se refere ao término dos produtos de trabalho. A data na qual um produto de trabalho é considerado terminado é denominada **marco**. Para determinar se um produto de trabalho atingiu de fato um marco, ele deve primeiro passar por uma série de **revisões** realizadas por colegas da equipe, pela gerência ou pelo cliente. Um marco típico é a data na qual o projeto é completado e passa pela revisão. Assim que um produto de trabalho tiver sido revisado e tiver obtido aprovação, ele se torna uma **referência** e pode ser alterado apenas por meio de procedimentos formais, conforme os descritos na Seção 5.8.2.

Na realidade, há mais coisas envolvidas em um produto de trabalho que meramente o produto em si. Um **pacote de trabalho** define não apenas o produto de trabalho, mas também as necessidades em termos de pessoal, duração, recursos, nome do responsável e critérios de aceitação. **Dinheiro**, obviamente, é um componente vital do plano. Deve-se elaborar um orçamento detalhado e o dinheiro deve ser alocado, como uma função de tempo, para as funções e atividades do projeto.

A questão de como elaborar um plano para produção de software é vista a seguir.

9.4 Estrutura do Plano de Gerenciamento de Projeto de Software

Existem várias formas de elaborar um plano de gerenciamento de projeto. Uma das melhores é o IEEE Standard 1058 (1998). Os componentes do plano são mostrados na Figura 9.9.

- O padrão foi elaborado por representantes de um grande número das principais organizações envolvidas em desenvolvimento de software. Foram pesquisadas tanto empresas quanto universidades, e os membros do grupo de trabalho e das equipes de revisão tinham vários anos de experiência na elaboração de planos de gerenciamento de projeto. O padrão incorpora essa experiência.
- O plano de gerenciamento de projeto do IEEE foi desenvolvido para uso com todos os tipos de produtos de software. Ele não impõe um modelo de ciclo de vida específico ou prescreve uma metodologia específica. O plano é, essencialmente, um arcabouço, cujos conteúdos são adaptados, em cada organização, para um domínio, uma equipe ou uma técnica específicos.
- O plano de gerenciamento de projeto do IEEE suporta aperfeiçoamento de processos. Por exemplo, várias das seções da estrutura refletem áreas de processos fundamentais do CMM (Seção 3.13) como a métrica e o gerenciamento de configurações.
- O plano de gerenciamento de projeto do IEEE é ideal para o Processo Unificado. Por exemplo, uma seção do plano é dedicada ao controle de reparos e outra, à gestão de riscos, ambos aspectos fundamentais do Processo Unificado.

De outro lado, embora se alegue no IEEE Standard 1058 (1998) que o plano de gerenciamento de projeto do IEEE aplica-se a projetos de software de todos os tamanhos, algumas das seções não são relevantes para software de pequeno porte. Por exemplo, a Seção 7.7 da

FIGURA 9.9
Estrutura do plano de gerenciamento de projeto do IEEE.

1 Visão Geral
- 1.1 Resumo do projeto
  - 1.1.1 Propósito, escopo e objetivos
  - 1.1.2 Hipóteses e restrições
  - 1.1.3 Entregáveis do projeto
  - 1.1.4 Resumo do orçamento e cronograma
- 1.2 Evolução do plano de gerenciamento de projeto

2 Materiais de referência

3 Definições e acrônimos

4 Organização do projeto
- 4.1 Interfaces externas
- 4.2 Estrutura interna
- 4.3 Funções e responsabilidades

5 Planos de processos gerenciais
- 5.1 Plano inicial
  - 5.1.1 Plano de estimativas
  - 5.1.2 Plano de alocação de pessoal
  - 5.1.3 Plano de aquisição de recursos
  - 5.1.4 Plano de treinamento do pessoal de projeto
- 5.2 Plano de trabalho
  - 5.2.1 Atividades de trabalho
  - 5.2.2 Alocação de cronograma
  - 5.2.3 Alocação de recursos
  - 5.2.4 Alocação orçamentária
- 5.3 Plano de controle
  - 5.3.1 Plano de controle de necessidades
  - 5.3.2 Plano de controle de cronograma
  - 5.3.3 Plano de controle orçamentário
  - 5.3.4 Plano de controle de qualidade
  - 5.3.5 Plano de geração de relatórios
  - 5.3.6 Plano de reunião de métricas
- 5.4 Plano de gerenciamento de riscos
- 5.5 Plano de fechamento de projeto

6 Planos de processos técnicos
- 6.1 Modelo de processo
- 6.2 Métodos, ferramentas e técnicas
- 6.3 Plano de infra-estrutura
- 6.4 Plano de aceitação do produto

7 Planos de processos de apoio
- 7.1 Plano de gerenciamento de configurações
- 7.2 Plano de realização de testes
- 7.3 Plano de documentação
- 7.4 Plano de garantia de qualidade
- 7.5 Plano de revisões e auditorias
- 7.6 Plano de resolução de problemas
- 7.7 Plano de gerenciamento de empresas terceirizadas
- 7.8 Plano de aperfeiçoamento de processos

8 Planos adicionais

estrutura do plano é intitulada "Plano de Gerenciamento de Empresas Terceirizadas", mas é insólito que empresas terceirizadas sejam usadas em projetos de pequeno porte.

Conseqüentemente, apresentaremos a seguir a estrutura do plano de duas formas distintas. Primeiramente, a estrutura completa é descrita na Seção 9.5. Em segundo lugar, uma versão ligeiramente abreviada da estrutura é usada no Apêndice F para um plano de gerenciamento de um projeto de pequeno porte, o estudo de caso da MSG Foundation.

9.5 O Plano de Gerenciamento de Projeto de Software da IEEE

Descrevemos a seguir a estrutura do **plano de gerenciamento de projeto de software** (SPMP) **da IEEE** de forma detalhada. Os números e títulos no texto correspondem àqueles da Figura 9.9. Os vários termos usados foram definidos na Seção 9.3.

1 Visão Geral.

1.1 Resumo do projeto.

1.1.1 Propósito, escopo e objetivos. É fornecida uma breve descrição do propósito e do escopo do produto de software a ser entregue bem como dos objetivos do projeto. As necessidades do negócio são incluídas nessa subseção.

1.1.2 Hipóteses e restrições. Quaisquer hipóteses implícitas no projeto são declaradas aqui, com restrições como data de entrega, orçamento, recursos e artefatos a serem reutilizados.

1.1.3 Entregáveis do projeto. Todos os itens a serem entregues ao cliente são listados aqui, com às datas de entrega.

1.1.4 Resumo do orçamento e cronograma. O cronograma geral é apresentado aqui, junto com o orçamento global.

1.2 Evolução do plano de gerenciamento de projeto. Nenhum plano pode ser moldado em concreto. O plano de gerenciamento de projeto, assim como qualquer outro plano, requer atualização contínua em vista da experiência e das mudanças tanto na empresa do cliente quanto da desenvolvedora de software. Nessa seção são descritos os procedimentos formais e os mecanismos para modificar o plano, inclusive o mecanismo para que o próprio plano de gerenciamento de projeto passe a ser controlado pelo controle de configurações.

2 Materiais de referência. Todos os documentos referidos no plano de gerenciamento de projeto são listados aqui.

3 Definições e acrônimos. Essas informações garantem que o plano de gerenciamento de projeto seja compreendido da mesma maneira por todos.

4 Organização do projeto.

4.1 Interfaces externas. Nenhum projeto é construído no vácuo. Os membros do projeto têm de interagir com o cliente e com outros membros de sua própria organização. Além disso, talvez existam empresas terceirizadas envolvidas em um projeto de grande porte. Os limites administrativos e gerenciais entre o projeto e essas outras entidades devem ser traçados.

4.2 Estrutura interna. Nessa seção é descrita a estrutura da própria empresa desenvolvedora de software. Por exemplo, muitas empresas desenvolvedoras de software são divididas em dois tipos de grupos: grupos de desenvolvimento, que trabalham em um único projeto, e grupos de suporte, que fornecem funções de apoio como gerenciamento de configurações

e garantia da qualidade para toda a empresa. Os limites administrativos e gerenciais entre o grupo de projeto e os grupos de suporte também devem ser definidos claramente.

4.3 Funções e responsabilidades. Deve-se identificar o responsável em cada função do projeto por exemplo, garantia da qualidade, e em cada atividade, como teste do produto.

5 Planos de processos gerenciais.

5.1 Plano inicial.

5.1.1 Plano de estimativas. As técnicas usadas para estimar o custo e a duração do projeto são listadas aqui, bem como a maneira pela qual essas estimativas são acompanhadas e, se necessário, modificadas, enquanto o projeto está em andamento.

5.1.2 Plano de alocação de pessoal. Quantidade e tipo de pessoal necessários são enumerados, juntamente das durações para as atividades em que eles são requisitados.

5.1.3 Plano de aquisição de recursos. A maneira pela qual se adquirem os recursos necessários, entre os quais hardware, software, contratos de serviço e serviços administrativos, é dada aqui.

5.1.4 Plano de treinamento do pessoal de projeto. Todo o treinamento necessário para se completar um projeto de forma bem-sucedida é apresentado nessa subseção.

5.2 Plano de trabalho.

5.2.1 Atividades de trabalho. Nessa subseção, as atividades de trabalho são especificadas até atingir o nível de tarefa, se apropriado.

5.2.2 Alocação de cronograma. Em geral, os pacotes de trabalho são interdependentes e mais dependentes ainda de eventos externos. Por exemplo, o fluxo de trabalho de implementação vem após o fluxo de trabalho de projeto e precede a realização de testes do produto. Nessa subseção, as dependências relevantes são especificadas.

5.2.3 Alocação de recursos. Os vários recursos listados anteriormente são alocados às funções, atividades e tarefas apropriadas do projeto.

5.2.4 Alocação orçamentária. Nessa seção, o orçamento global é subdividido em níveis de funções, atividades e tarefas de projeto.

5.3 Plano de controle.

5.3.1 Plano de controle de necessidades. Conforme será descrito na Parte 2 deste livro, enquanto um produto de software está sendo desenvolvido, as necessidades freqüentemente mudam. Os mecanismos usados para monitorar e controlar as mudanças de necessidades são dados nesta seção.

5.3.2 Plano de controle de cronograma. Nesta subseção, são listados mecanismos para medir o progresso, com uma descrição das medidas a serem tomadas se o estágio real do projeto estiver atrasado em relação àquele planejado.

5.3.3 Plano de controle orçamentário. É importante que os gastos não excedam aqueles previstos em orçamento. Mecanismos de controle para monitorar quando o custo real excede o custo orçado, bem como as medidas a serem tomadas se isso acontecer, são apresentados nesta subseção.

5.3.4 Plano de controle da qualidade. As formas pelas quais a qualidade é medida e controlada são descritas nesta subseção.

5.3.5 Plano de geração de relatórios. Para monitorar as necessidades, o cronograma, o orçamento e a qualidade, são necessários mecanismos para geração de relatórios. Tais mecanismos são descritos nesta subseção.

5.3.6 Plano de reunião de métricas. Conforme explicado na Seção 5.3, não é possível administrar o processo de desenvolvimento sem métricas relevantes. As métricas a serem reunidas são listadas nesta subseção.

5.4 Plano de gerenciamento de riscos. Os riscos devem ser identificados, priorizados, minimizados e monitorados. Todos os aspectos do gerenciamento de riscos são descritos nesta seção.

5.5 Plano de fechamento de projeto. As medidas a serem tomadas assim que o projeto for finalizado, inclusive a realocação de pessoal e o arquivamento de artefatos, são apresentadas aqui.

6 Planos de processos técnicos.

6.1 Modelo de processo. Nesta seção, é dada uma descrição detalhada do modelo de ciclo de vida a ser usado.

6.2 Métodos, ferramentas e técnicas. As metodologias de desenvolvimento e as linguagens de programação a serem usadas são descritas aqui.

6.3 Plano de infra-estrutura. Aspectos técnicos de hardware e software são descritos em detalhe nesta seção. Entre os itens a serem vistos, há os sistemas computacionais (hardware, sistemas operacionais, redes e software) que serão usados para desenvolver o produto bem como os sistemas de computação em que o produto de software será executado e as ferramentas CASE empregadas.

6.4 Plano de aceitação do produto. Para garantir que o produto de software finalizado passe pelo teste de aceitação devem ser elaborados critérios de aceitação, o cliente deve concordar, por escrito, com tais critérios e, em seguida, os desenvolvedores devem garantir que esses critérios serão, de fato, atendidos. A maneira pela qual esses três estágios do processo de aceitação serão executados é descrita nesta seção.

7 Planos de processos de apoio.

7.1 Plano de gerenciamento de configurações. Nesta seção é dada uma descrição detalhada dos meios pelos quais todos os artefatos passam a ser controlados pelo gerenciamento de configurações.

7.2 Plano de realização de testes. Testes, assim como todos os demais aspectos do processo de desenvolvimento de software, precisam de meticuloso planejamento.

7.3 Plano de documentação. Uma descrição de todo o tipo de documentação a ser entregue ou não ao cliente no final do projeto é incluída nesta seção.

7.4 Plano de garantia da qualidade. Todos os aspectos da garantia da qualidade, entre os quais testes, padrões e revisões, são abarcados por esta seção.

7.5 Plano de revisões e auditorias. Detalhes de como as revisões são conduzidas são apresentados nesta seção.

7.6 Plano de resolução de problemas. No curso do desenvolvimento de um produto de software certamente surgirão problemas. Por exemplo, uma revisão do projeto pode trazer à tona uma falha crítica no fluxo de trabalho de análise que requeira importantes modificações em praticamente todos os artefatos já completados. Nesta seção é descrita a maneira como se lida com tais problemas.

7.7 Plano de gerenciamento de empresas terceirizadas. Esta seção se aplica quando terceiros devem fornecer certos produtos do trabalho. A abordagem para selecionar e administrar o trabalho de empresas terceirizadas figura aqui.

7.8 Plano de aperfeiçoamento de processos. As estratégias de aperfeiçoamento de processos são abordadas nesta seção.

8 Planos adicionais. Para certos projetos, talvez seja necessária a presença de componentes adicionais. Em termos da estrutura formulada pelo IEEE, eles aparecem no final do plano. Entre esses componentes adicionais há planos de segurança, planos de proteção, planos de conversão de dados, planos de instalação e o plano de manutenção pós-entrega do projeto de software.

9.6 Planejamento de Testes

Um componente do SPMP que, muitas vezes, é menosprezado é o **planejamento de testes**. Como qualquer outra atividade do desenvolvimento de software, os testes precisam ser planejados. O SPMP deve incluir recursos para testes, e o cronograma detalhado deve indicar explicitamente os testes a serem realizados em cada fluxo de trabalho.

Sem um plano de testes, um projeto pode fracassar de vários modos. Por exemplo, durante a realização de testes do produto (Seção 3.7.4), o grupo de SQA deve verificar se todos os aspectos do documento de especificações, conforme assinado pelo cliente, foram implementados no produto final. Uma boa forma de ajudar o grupo de SQA nessa tarefa é exigir que o desenvolvimento seja rastreável (Seção 3.7), isto é, deve ser possível associar cada instrução contida no documento de especificações a uma parte do projeto, e cada parte do projeto deve estar refletida explicitamente no código. Uma técnica para se atingir isso é numerar cada instrução no documento de especificações e garantir que esses números se reflitam tanto no projeto quanto no código resultante. Entretanto, se o plano de testes não especificar que isso deve ser feito, é bastante improvável que a análise, o projeto e os artefatos de código sejam identificados apropriadamente. Conseqüentemente, quando finalmente os testes são realizados, será extremamente difícil para o grupo de SQA determinar se o produto é uma implementação completa das especificações. De fato, a rastreabilidade deveria iniciar com o levantamento das necessidades; cada instrução nos artefatos de levantamento de necessidades (ou cada parte do protótipo rápido) deve estar associada a uma parte dos artefatos de análise.

Um poderoso aspecto das inspeções é a lista detalhada das falhas detectadas durante uma inspeção. Suponha que uma equipe esteja inspecionando as especificações de um produto. Conforme explicamos na Seção 6.2.3, a lista de falhas é usada de duas maneiras. Em primeiro lugar, as estatísticas de falhas dessa inspeção devem ser comparadas às médias acumuladas das estatísticas de falhas de inspeções de especificações anteriores. Desvios de normas anteriores indicam problemas no projeto. Em segundo lugar, as estatísticas de falhas da inspeção de especificações atuais devem ser transferidas para as inspeções do projeto e do código do produto. Afinal, se existir um grande número de falhas de um determinado tipo, é possível que nem todas elas tenham sido detectadas durante a inspeção das especificações, e as inspeções do projeto e do código oferecem mais uma oportunidade para localização de quaisquer falhas remanescentes desse tipo. Entretanto, a menos que o plano de testes afirme que detalhes de todas as falhas devem ser cuidadosamente registrados, é improvável que essa tarefa seja feita.

Uma forma importante de testar módulos de código é o assim chamado "teste caixa preta" (Seção 14.11), no qual o código é executado com casos de ensaio baseados nas especificações. Membros do grupo de SQA lêem as especificações e elaboram casos de ensaio para verificar se o código obedece ou não ao documento de especificações. O melhor momento para elaborar casos de teste caixa preta é no final do fluxo de trabalho de análise, quando os detalhes do documento de especificações ainda estão frescos na memória dos membros do grupo de SQA que o inspecionaram. Porém, a menos que o plano de testes afirme explicitamente que os casos de teste caixa preta devam ser feitos nessa oportunidade, muito provavelmente apenas alguns poucos casos serão feitos juntos e apressadamente mais tarde; isto é, um número limitado de casos de ensaio será rapidamente agrupado apenas quando começar a existir pressão por parte da equipe de programação para que o grupo de SQA aprove seus módulos, a fim de que eles possam ser integrados ao produto totalmente. Como resultado, a qualidade do produto como um todo sofre as conseqüências.

Portanto, todo plano de testes tem de especificar qual teste deve ser realizado, quando e como ele deve ser realizado. Um plano de testes desses é uma parte essencial da Seção 7.2

do SPMP. Sem ele, a qualidade do produto como um todo indubitavelmente ficará comprometida.

9.7 Planejamento de Projetos Orientados a Objetos

Suponha que o paradigma clássico seja usado. De um ponto de vista conceitual, o produto resultante geralmente será uma única grande unidade, muito embora ele seja composto de módulos separados. Ao contrário, o emprego do paradigma de orientação a objetos resulta em um produto formado por uma série de componentes menores, relativamente independentes, ou seja, as classes. Isso torna o planejamento consideravelmente mais fácil, pois assim as estimativas de custo e duração podem ser calculadas de forma mais fácil e acuradas para unidades menores. Obviamente, as estimativas devem levar em conta que um produto é mais do que simplesmente a soma de suas partes. Os componentes separados não são totalmente independentes; eles podem chamar uns aos outros, e esses efeitos não devem ser desprezados.

Seriam as técnicas para estimativas de custo e duração descritas no presente capítulo aplicáveis ao paradigma de orientação a objetos? O COCOMO II (Seção 9.2.4) foi desenvolvido para lidar com tecnologia de software moderna, inclusive a orientação a objetos, mas o que dizer sobre métricas anteriores como pontos de função (Seção 9.2.1) e o COCOMO intermediário (Seção 9.2.3)? No caso do COCOMO intermediário, são necessárias pequenas alterações em alguns dos multiplicadores de custo (Pittman, 1993). Além disso, as ferramentas para estimativas do paradigma clássico parecem funcionar relativamente bem em projetos orientados a objetos, desde que não se adote o recurso da reutilização. Essa entra no paradigma de orientação a objetos de duas maneiras: a reutilização de componentes existentes durante o desenvolvimento e a produção deliberada (durante o projeto atual) de componentes para serem reutilizados em produtos futuros. Ambas as formas afetam o processo de estimativas. O emprego da reutilização durante o desenvolvimento reduz claramente o custo e a duração. Foram publicadas fórmulas mostrando a economia em função da reutilização (Schach, 1994), porém esses resultados estão relacionados ao paradigma clássico. Atualmente, não se dispõe de nenhuma informação de como o custo e a duração mudam quando se adota o recurso da reutilização no desenvolvimento de um produto orientado a objetos.

Passaremos agora para o objetivo de reutilizar partes do projeto atual. Pode ser que leve cerca de três vezes mais tempo projetar, implementar, testar e documentar um componente reutilizável do que um componente similar não reutilizável (Pittman, 1993). As estimativas de custo e duração devem ser modificadas para incorporar esse trabalho adicional, e o SPMP como um todo deve ser ajustado para incorporar o efeito do esforço de reutilização. Conseqüentemente, as duas atividades de reutilização funcionam em sentidos contrários. A reutilização de componentes existentes reduz o esforço geral no desenvolvimento de um produto orientado a objetos, ao passo que projetar componentes visando à reutilização em produtos futuros aumenta o esforço. Espera-se que, no longo prazo, a economia resultante da reutilização de classes seja maior do que os custos dos desenvolvimentos originais, e já existem algumas evidências que sustentam essa hipótese (Lim, 1994).

9.8 Necessidades de Treinamento

Quando se coloca em pauta o assunto **treinamento**, uma resposta comum do cliente é: "Não precisamos nos preocupar com treinamento até que o produto esteja pronto, quando, então, poderemos treinar nossos usuários". Trata-se de um comentário um tanto infeliz, implicando que apenas usuários precisam de treinamento. Na realidade, talvez seja necessário dar treinamento para os membros da equipe de desenvolvimento, partindo do treinamento em planejamento e estimativas. Quando são usadas novas técnicas de desenvolvimento de

software, como novas técnicas de projeto ou procedimentos de teste, deve-se fornecer treinamento a todos os membros da equipe que vai usar a nova técnica.

A introdução do paradigma de orientação a objetos tem conseqüências importantes no treinamento. A introdução de hardware ou de ferramentas de software, por exemplo, estações de trabalho ou um ambiente integrado (veja a Seção 14.23.2), também requer treinamento. Talvez os programadores precisem de treinamento no sistema operacional da máquina a ser usada para o desenvolvimento do produto bem como na linguagem de implementação. Treinamento na preparação de documentação normalmente é menosprezado, conforme é evidenciado pela qualidade inadequada de tal documentação. Os operadores de computadores certamente precisam de algum tipo de treinamento para serem capazes de executar o novo produto; talvez eles também precisem de treinamento adicional, se um novo hardware for utilizado.

O treinamento necessário pode ser obtido de vários modos. O mais fácil e menos prejudicial está no treinamento dentro da empresa, seja pelos colegas da própria empresa ou por consultores. Diversas companhias oferecem uma grande variedade de cursos de treinamento e, muitas vezes, os próprios colegas se dispõem a dar treinamento no final do expediente. Outra alternativa são os cursos pela Internet.

Assim que forem determinadas as necessidades de treinamento e o plano de treinamento for elaborado, este último deve ser incorporado ao SPMP.

9.9 Padrões de Documentação

O desenvolvimento de um produto de software é acompanhado de uma ampla gama de **documentação**. Jones descobriu que foram geradas 28 páginas de documentação para cada mil instruções (KDSI) em um produto comercial interno da IBM com cerca de 50 KDSI de tamanho e cerca de 66 páginas por KDSI para um produto de software comercial de mesmo porte. O sistema operacional IMS/360 Versão 2.3 tinha aproximadamente 166 KDSI de tamanho, e foram produzidas 157 páginas de documentação por KDSI. A documentação era de vários tipos, entre os quais planejamento, controle, financeiro e técnico (Jones, 1986a). Além desses tipos de documentação, o próprio código-fonte é uma forma de documentação; comentários dentro do código constituem uma documentação adicional.

Uma parte considerável do esforço de desenvolvimento de software é absorvida pela documentação. Uma pesquisa de 63 projetos de desenvolvimento e 25 projetos de manutenção pós-entrega demonstrou que, para cada 100 horas gastas em atividades relacionadas à codificação, 150 horas são gastas em atividades relacionadas à documentação (Boehm, 1981). Para produtos TRW grandes, a proporção de tempo dedicado a atividades relacionadas à documentação subiu para 200 horas para cada 100 horas relacionadas a código (Boehm et al., 1984).

É necessário haver padrões para cada tipo de documentação. Por exemplo, a uniformidade na documentação de projeto reduz interpretações errôneas entre os membros da equipe e ajuda o grupo de SQA. Embora funcionários novos tenham de ser treinados em padrões de documentação, não é necessário nenhum treinamento adicional quando funcionários existentes migram de um projeto a outro dentro de uma organização. Do ponto de vista da manutenção pós-entrega, padrões de codificação uniformes ajudam os programadores de manutenção a entenderem o código-fonte. A padronização é mais importante ainda para manuais de usuário, pois estes devem ser lidos por uma gama variada de indivíduos, poucos dos quais especialistas em computação. O IEEE desenvolveu um padrão para manuais de usuário (IEEE Standard 1063 for Software User Documentation).

Como parte do processo de planejamento, devem ser estabelecidos padrões para toda a documentação a ser produzida durante a produção de software. Esses padrões são incorporados ao SPMP.

Nos pontos em que se deve usar um padrão existente, como o ANSI/IEEE Standard for Software Test Documentation (ANSI/IEEE 829, 1991), o padrão é listado na Seção 2 do SPMP (materiais de referência). Se um padrão for elaborado especialmente para a atividade de desenvolvimento, ele figurará na Seção 6.2 (métodos, ferramentas e técnicas).

A documentação é um aspecto essencial da atividade de produção de software. Em termos práticos, o produto é a documentação, pois, sem documentação, o produto não pode ser mantido. Planejar a atividade de documentação em todos os seus aspectos, garantindo que há uma aderência ao plano, é um componente importante da produção de software bem-sucedida.

9.10 Ferramentas CASE para Planejamento e Estimativas

Existe um grande número de ferramentas disponíveis que automatizam o COCOMO intermediário e o COCOMO II. Para agilizar o cálculo quando o valor de um parâmetro é modificado, várias implementações do COCOMO intermediário foram escritas em linguagens de planilhas eletrônicas como Lotus 1-2-3 ou o Excel. Para desenvolver e atualizar o próprio plano, é fundamental um processador de texto.

Ferramentas de informações gerenciais também são úteis para o planejamento. Suponhamos, por exemplo, que uma grande empresa desenvolvedora de software tenha 150 programadores. Uma ferramenta para cronograma pode ajudar os planejadores a monitorar quais programadores já foram alocados para determinadas tarefas e quais estão disponíveis para o projeto atual.

São necessários também tipos de informações gerenciais mais genéricos. Podem ser usadas várias ferramentas gerenciais existentes no mercado, tanto para ajudar no processo de planejamento e estimativas quanto para monitorar o processo de desenvolvimento como um todo. Entre elas podemos citar MacProject e Microsoft Project.

9.11 Teste do Plano de Gerenciamento de Projeto de Software

Como indicamos no início deste capítulo, uma falha no plano de gerenciamento de projeto de software pode ter graves conseqüências financeiras para os desenvolvedores. É importante que a empresa desenvolvedora de software não estime, nem para mais nem para menos, o custo e a duração do projeto. Por essa razão, todo SPMP deve ser verificado pelo grupo de SQA antes de serem passadas estimativas ao cliente. A melhor maneira de testar o plano é por intermédio de uma inspeção do plano.

A equipe de inspeção do plano deve revisar detalhadamente o SPMP, prestando particular atenção nas estimativas de custo e duração. Para reduzir mais ainda os riscos, independentemente da métrica usada, as estimativas de custo e duração devem ser calculadas independentemente por um membro do grupo de SQA tão logo os membros da equipe de planejamento tenham determinado suas estimativas.

Revisão do Capítulo

O principal tema do capítulo é a importância do planejamento no processo de software (Seção 9.1). Um componente vital de qualquer plano de gerenciamento de projeto de software é estimar sua duração e custo (Seção 9.2). Foram propostas várias métricas para calcular o tamanho de um produto, entre as quais os pontos de função (Seção 9.2.1). Em seguida, são descritas várias métricas para estimativa de custo, particularmente o COCOMO intermediário (Seção 9.2.3) e o COCOMO II (Seção 9.2.4). Conforme descrito na Seção 9.2.5, é essencial monitorar todas as estimativas. Os três principais componentes de

um plano de gerenciamento de projeto de software — o trabalho a ser feito, os recursos com os quais este trabalho será feito e o dinheiro a ser gasto com ele são explicados na Seção 9.3. Um SPMP particular, o padrão IEEE, é descrito em linhas gerais na Seção 9.4 e, de forma detalhada, na Seção 9.5. Em seguida, há as seções sobre planejamento de testes (Seção 9.6), planejamento de projetos no paradigma da orientação a objetos (Seção 9.7), e necessidades de treinamento assim como padrões de documentação e suas implicações no processo de planejamento (Seções 9.8 e 9.9). As ferramentas CASE para planejamento e estimativas são descritas na Seção 9.10. O capítulo termina com material sobre como testar o plano de um gerenciamento de projeto de software (Seção 9.11).

Leitura Complementar

A obra em quatro volumes de Weinberg (Weinberg, 1992; 1993; 1994; 1997) fornece informações detalhadas sobre vários aspectos do gerenciamento de software, assim como (Bennatan, 2000) e (Reifer, 2000). Métricas para gerenciamento de projetos de software são discutidas em (Weller, 1994).

Para gerenciamento do paradigma da orientação a objetos, (Pittman, 1993) e (Nesi, 1998) devem ser consultados. Para mais informações sobre o IEEE Standard 1058 for Software Project Management Plans, o próprio padrão deve ser lido minuciosamente (IEEE 1058, 1998). A necessidade de planejamento meticuloso é descrita em (McConnell, 2001).

O trabalho clássico de Sackman é descrito em (Sackman, Erikson e Grant, 1968). Uma fonte mais detalhada é (Sackman, 1970).

Informações úteis sobre pontos de função podem ser encontradas em (Low e Jeffrey, 1990). Uma análise minuciosa sobre pontos de função bem como sugestões para seu aperfeiçoamento figuram em (Symons, 1991). Críticas aos pontos de função aparecem em (Kitchenham, 1997).

A confiabilidade dos pontos de função é discutida em (Kemerer e Porter, 1992) e em (Kemerer, 1993). Os pontos fracos e fortes dos pontos de função são apresentados em (Furey e Kitchenham, 1997). Uma fonte de informação abrangente sobre todos os aspectos dos pontos de função é (Boehm, 1997).

A justificativa teórica para o COCOMO intermediário, com detalhes completos para sua implementação, figura em (Boehm, 1981); uma versão mais reduzida é encontrada em (Boehm, 1984). O COCOMO II é descrito em (Boehm et al., 2000). Maneiras de melhorar as previsões obtidas com o COCOMO são apresentadas em (Smith, Hale e Parrish, 2001).

Briand e Wüst (2001) descrevem como estimar o esforço de desenvolvimento para produtos orientados a objetos. Como estimar o tamanho e os defeitos de produtos de software orientados a objetos é o tema de (Cartwright e Shepperd, 2000). Pontos de classe, uma extensão dos pontos de função para classes, são introduzidos em (Costagliola, Ferrucci, Tortora e Vitiello, 2005).

Erros na estimativa de esforço de software são analisados em (Jorgensen e Molokken-Ostvold, 2004). Dados sobre produtividade de software para uma série de produtos para processamento de dados comerciais são apresentados em (Maxwell e Forselius, 2000); a unidade de produtividade utilizada são pontos de função por hora. Outras medidas de produtividade são discutidas em (Kitchenham e Mendes, 2004). Estimativa do tamanho para software escrito com uma linguagem de quarta geração é apresentada em (Dolado, 2000). Vários artigos sobre estimativas podem ser encontrados na edição de novembro/dezembro de 2000 da *IEEE Software*.

Termos-chave

Exercícios

9.1 Por que você acha que algumas empresas de desenvolvimento de software chamam os *milestones* de *millstones*? (Dica: pesquise o sentido figurado de *millstone* em um dicionário de inglês.)

9.2 Você é um engenheiro de software na Bronkhorstspruit Software Developers. Há um ano, seu gerente anunciou que seu próximo produto compreenderia 9 arquivos, 49 fluxos e 92 processos.

(i) Usando a métrica FFP, determine seu tamanho.

(ii) Determinou-se que, para a Bronkhorstspruit Software Developers, a constante *d* na Equação (9.2) seria $ 1.003. Que estimativa de custo a métrica FFP previu?

(iii) O produto foi recentemente concluído a um custo de $ 132.800. O que isso lhe diz em relação à produtividade de sua equipe de desenvolvimento?

9.3 Um produto-alvo possui sete entradas simples, duas entradas médias e dez entradas complexas. Existem 56 saídas médias, oito consultas simples, 12 arquivos-mestre médios e 17 interfaces complexas. Determine os pontos de função não ajustados (*UFP*).

9.4 Se o grau de influência global para o produto do Exercício 9.3 for 49, determine o número de pontos de função.

9.5 Apesar de seus inconvenientes, por que as linhas de código (LOC ou KDSI) são tão amplamente usadas como métrica para a determinação do tamanho do produto?

9.6 Você está incumbido de desenvolver um produto embutido de 67 KDSI que é nominal, exceto pelo fato de o tamanho do banco de dados ser classificado como muito grande e o uso de ferramentas de software ser classificado como pequeno. Usando o COCOMO intermediário, qual o esforço estimado em homens-mês?

9.7 Você está incumbido de desenvolver dois produtos de modo orgânico de 33 KDSI. Ambos são nominais em todos os aspectos, exceto pelo fato de o produto P1 ter complexidade muito grande e o produto P2 ter complexidade muito pequena. Para desenvolver o produto, você tem duas equipes à sua disposição. A equipe A recebe a classificação *muito grande* nos seguintes atributos: habilidades dos analistas, experiência em aplicações e capacidade dos programadores. A equipe A também é cotada como *muito grande* nos atributos experiência em máquina virtual e experiência em linguagens de programação. Já a equipe B recebe a classificação *muito pequena* nos cinco atributos.

(i) Qual o esforço total (em homens-mês) se a equipe A desenvolver o produto P1 e a equipe B desenvolver o produto P2?

(ii) Qual o esforço total (em homens-mês) se a equipe B desenvolver o produto P1 e a equipe A desenvolver o produto P2?

(iii) Qual das duas alocações de pessoal acima faz mais sentido? Sua intuição se baseia em previsões do COCOMO intermediário?

9.8 Você está incumbido de desenvolver um produto de modo orgânico de 49 KDSI que é nominal em todos os aspectos.

(i) Supondo-se um custo de $ 9.900 por homem-mês, qual a estimativa de custo do projeto?

(ii) Toda sua equipe de desenvolvimento pede demissão no início do projeto. Você é suficientemente afortunado para ser capaz de substituir a equipe anterior por uma equipe altamente

capaz e experiente, mas o custo por homem-mês subirá para $ 12.900. Quanto você espera ganhar (ou perder) como conseqüência da mudança de pessoal?

9.9 Você está incumbido de desenvolver o software para um produto que usa um conjunto de algoritmos recém-desenvolvidos para calcular caminhos ótimos em termos de custo para uma grande empresa transportadora. Usando o COCOMO intermediário, você determina que o custo do produto é de $ 470 mil. Entretanto, como verificação, você solicita a um membro de sua equipe que estime o esforço usando pontos de função. Ele lhe entrega um relatório indicando que a métrica de pontos de função prevê um custo de $ 985 mil, mais do que o dobro da sua previsão obtida pelo COCOMO. O que você faria?

9.10 Demonstre que a distribuição de Rayleigh (Equação (9.9) atinge seu valor máximo quando $t = k$. Determine o consumo de recursos correspondente.

9.11 Um plano de manutenção pós-entrega é considerado como um "componente adicional" para um plano de gerenciamento de projeto de software da IEEE. Tendo em mente que todo produto não trivial deve sofrer manutenção e que o custo da manutenção pós-entrega é, em média, duas ou três vezes maior do que o custo de desenvolvimento do produto, como isso pode ser justificado?

9.12 Por que os projetos de desenvolvimento de software geram tanta documentação?

9.13 Considere o projeto da Osric's Office Appliances and Decor des-crito no Apêndice A. Por que não é possível estimar o custo e a duração do projeto baseando-se simplesmente nas informações fornecidas no Apêndice A?

9.14 (Leituras na área de engenharia de software) A partir do texto (Costagiola, Ferrucci, Tortora e Vitiello, 2005), você ficaria convencido com a validação empírica dos pontos de classe?

Referências Bibliográficas

(Albrecht, 1979) ALBRECHT, A. J., Measuring Application Development Productivity, *Proceedings of the IBM SHARE/GUIDE Applications Development Symposium*, Monterey, CA, outubro 1979, p. 83-92.

(ANSI/IEEE 829, 1991) *Software Test Documentation*, ANSI/IEEE 829-1991, American National Standards Institute, Institute of Electrical and Electronic Engineers, Nova York, 1991.

(Bennatan, 2000) BENNATAN, E. M., *On Time Within Budget: Software Project Management Practices and Techniques,* 3ª ed., John Wiley and Sons, Nova York, 2000.

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

(Boehm, 1984) BOEHM, B. W., Software Engineering Economics, *IEEE Transactions on Software Engineering* SE-10 (janeiro 1984), p. 4-21.

(Boehm, 1997) BOEHM, R. (Ed.), Function Point FAQ, em ourworld.compuserve.com/homepages/softcomp/fpfaq.htm, 25 de junho de 1997.

(Boehm et al., 1984) BOEHM, B. W., PENEDO, M. H., STUCKLE, E. D., WILLIAMS, R. D. e PYSTER, A. B., A Software Development Environment for Improving Productivity, *IEEE Computer* **17** (junho 1984), p. 30-44.

(Boehm et al., 2000) BOEHM, B. W., ABTS, C., BROWN, A. W., CHULANI, S., CLARK, B. K., HOROWITZ, E., MADACHY, R., REIFER D. e STEECE B., *Software Cost Estimation with COCOMO II,* Prentice Hall, Upper Saddle River, NJ, 2000.

(Briand e Wüst, 2001) BRIAND, L. C. e WÜST, J., Modeling Development Effort in Object-Oriented Systems Using Design Properties, *IEEE Transactions on Software Engineering* **27** (novembro 2001), p. 963-86.

(Cartwright e Shepperd, 2000) CARTWRIGHT, M. e SHEPPERD, M., An Empirical Investigation of an Object-Oriented Software System, *IEEE Transactions on Software Engineering* **26** (agosto 2000), p. 786-95.

(Costagliola, Ferrucci, Tortora e Vitiello, 2005) COSTAGLIOLA, G., FERRUCCI, F., TORTORA, G. e VITIELLO, G., Class Point: An Approach for the Size Estimation of Object-Oriented Systems, *IEEE Transactions on Software Engineering* **31** (janeiro 2005), p. 52-74.

(Devenny, 1976) DEVENNY, T., An Exploratory Study of Software Cost Estimating at the Electronic Systems Division, tese nº GSM/SM/765–4, Air Force Institute of Technology, Dayton, OH, 1976.

(Dolado, 2000) DOLADO, J. J., A Validation of the Component-Based Method for Software Size Estimation, *IEEE Transactions on Software Engineering* **26** (outubro 2000), p. 1006-21.

(Furey e Kitchenham, 1997) FUREY, S. e KITCHENHAM, B., Function Points, *IEEE Software* **14** (março-abril 1997), p. 28-32.

(IEEE 1058, 1998) IEEE Standard for Software Project Management Plans, IEEE Std. 1058-1998, Institute of Electrical and Electronic Engineers, Nova York, 1998.

(Jones, 1986a) JONES, C., *Programming Productivity*, McGraw-Hill, Nova York, 1986.

(Jones, 1987) JONES, C., Letter to the Editor, *IEEE Computer* **20** (dezembro 1987), p. 4.

(Jorgensen e Molokken-Ostvold, 2004) JORGENSEN, M. e MOLOKKEN-OSTVOLD, K., Reasons for Software Effort Estimation Error: Impact of Respondent Role, Information Collection Approach e Data Analysis Method, *IEEE Transactions on Software Engineering* **30** (dezembro 2004), p. 993-1007.

(Kemerer, 1993) KEMERER, C. F., Reliability of Function Points Measurement: A Field Experiment, *Communications of the ACM* **36** (fevereiro 1993), p. 85-97.

(Kemerer e Porter, 1992) KEMERER, C. F. e PORTER, B. S., Improving the Reliability of Function Point Measurement: An Empirical Study, *IEEE Transactions on Software Engineering* **18** (novembro 1992), p. 1011-24.

(Kitchenham, 1997) KITCHENHAM, B., The Problem with Function Points, *IEEE Software* **14** (março -abril 1997), p. 29, 31.

(Kitchenham e Mendes, 2004) KITCHENHAM, B. e MENDES, E., Software Productivity Measurement Using Multiple Size Measures, *IEEE Transactions on Software Engineering* **30** (dezembro 2004), p. 1023-35.

(Lim, 1994) LIM, W. C., Effects of Reuse on Quality, Productivity e Economics, *IEEE Software* **11** (setembro 1994), p. 23-30.

(Low e Jeffrey, 1990) LOW, G. C. e JEFFREY, D. R., Function Points in the Estimation and Evaluation of the Software Process, *IEEE Transactions on Software Engineering* **16** (janeiro 1990), p. 64-71.

(Maxwell e Forselius, 2000) MAXWELL, K. D. e FORSELIUS, P., Benchmarking Software Development Productivity, *IEEE Software* **17** (janeiro-fevereiro 2000), p. 80-88.

(McConnell, 2001) MCCONNEL, S., The Nine Deadly Sins of Project Planning, *IEEE Software* **18** (novembro-dezembro 2001), p. 5-7.

(Nesi, 1998) NESI, P., Managing OO Projects Better, *IEEE Software* **15** (julho-agosto 1998), p. 50-60.

(Norden, 1958) NORDEN, P. V., Curve Fitting for a Model of Applied Research and Development Scheduling, *IBM Journal of Research and Development* **2** (julho 1958), p. 232-48.

(Pittman, 1993) PITTMAN, M., Lessons Learned in Managing, Object-Oriented Development, *IEEE Software* **10** (janeiro 1993), p. 43-53.

(Putnam, 1978) PUTNAM, L. H., A General Empirical Solution to the Macro Software Sizing and Estimating Problem, *IEEE Transactions on Software Engineering* **SE-4** (julho 1978), p. 345-61.

(Reifer, 2000) REIFER, D. J., Software Management: The Good, the Bad, and the Ugly, *IEEE Software* **17** (março-abril 2000), p. 73-75.

(Sackman, 1970) SACKMAN, H., *Man – Computer Problem Solving: Experimental Evaluation of Time-Sharing and Batch Processing,* Auerbach, Princeton, NJ, 1970.

(Sackman, Erikson e Grant, 1968) SACKMAN, H., ERIKSON, W. J. e GRANT, E. E., Exploratory Experimental Studies Comparing Online and Offline Programming Performance, *Communications of the ACM* **11** (janeiro 1968), p. 3-11.

(Schach, 1994) SCHACH, S. R., The Economic Impact of Software Reuse on Maintenance, *Journal of Software Maintenance: Research and Practice* **6** (julho/agosto 1994), p. 185-96.

(Smith, Hale e Parrish, 2001) SMITH, R. K., HALE, J. E. e PARRISH, A. S., An Empirical Study Using Task Assignment Patterns to Improve the Accuracy of Software Effort Estimation, *IEEE Transactions on Software Engineering* **27** (março 2001), p. 264-71.

(Symons, 1991) SYMONS, C. R., *Software Sizing and Estimating: Mk II FPA*, John Wiley and Sons, Chichester, UK, 1991.

(Van der Poel e Schach, 1983) VAN DER POEL, K. G. e SCHACH, S. R., A Software Metric for Cost Estimation and Efficiency Measurement in Data Processing System Development, *Journal of Systems and Software* **3** (setembro 1983), p. 187-91.

(Weinberg, 1992) WEINBERG, G. M., *Quality Software Management: Systems Thinking,* v. 1, Dorset House, Nova York, 1992.

(Weinberg, 1993) WEINBERG, G. M., *Quality Software Management: First-Order Measurement,* v. 2, Dorset House, Nova York, 1993.

(Weinberg, 1994) WEINBERG, G. M., *Quality Software Management: Congruent Action*,v. 3, Dorset House, Nova York, 1994.

(Weinberg, 1997) WEINBERG. M., *Quality Software Management: Anticipating Change*, v . 4, Dorset House, Nova York, 1997.

(Weller, 1994) WELLER, E. F., Using Metrics to Manage Software Projects, *IEEE Computer* **27** (setembro 1994), p. 27-34.