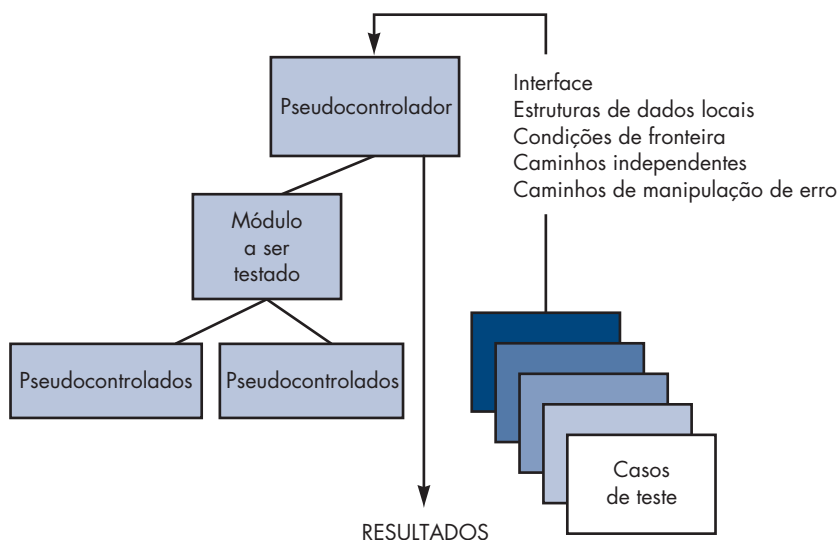


FIGURA 17.4**Ambiente de teste de unidade**

Há algumas situações nas quais você não terá recursos para fazer um teste de unidade simples. Selecione os módulos críticos ou complexos e faça o teste de unidade apenas neles.

do caso de teste, passa esses dados para o componente (a ser testado), e imprime resultados relevantes. Os *pseudocontroladores* servem para substituir módulos que são subordinados (chamados pelo) componente a ser testado. Um pseudocontrolado, ou “pseudosubprograma”, usa a interface dos módulos subordinados, pode fazer uma manipulação de dados mínima, fornece uma verificação de entrada e retorna o controle para o módulo que está sendo testado.

Pseudocontroladores e pseudocontrolados representam despesas indiretas. Isto é, ambos são softwares que devem ser escritos (projeto formal normalmente não é aplicado), mas que não são fornecidos com o produto de software final. Se os pseudocontroladores e pseudocontrolados são mantidos simples, as despesas reais indiretas são relativamente baixas. Infelizmente, muitos componentes não podem ser adequadamente testados no nível de unidade de modo adequado com software adicional simples. Em tais casos, o teste completo pode ser adiado até a etapa de integração (em que os pseudocontroladores e pseudocontrolados são também usados).

O teste de unidade é simplificado quando um componente com alta coesão é projetado. Quando somente uma função é implementada por um componente, o número de casos de teste é reduzido, e os erros podem ser mais facilmente previstos e descobertos.

17.3.2 Teste de integração

Um novato no mundo do software pode levantar uma questão aparentemente legítima quando todos os módulos tiverem passado pelo teste de unidade: “Se todos funcionam individualmente, porque você duvida que funcionem quando estiverem juntos?”. O problema, naturalmente, é “colocá-los todos juntos” — interfaces. Dados podem ser perdidos através de uma interface; um componente pode ter um efeito inesperado ou adverso sobre outro, subfunções, quando combinadas, podem não produzir a função principal desejada; imprecisão aceitável individualmente pode ser amplificada em níveis não aceitáveis; estruturas de dados globais podem apresentar problemas. Infelizmente, a lista não acaba.

O teste de integração é uma técnica sistemática para construir a arquitetura de software ao mesmo tempo que conduz testes para descobrir erros associados com as interfaces. O objetivo é construir uma estrutura de programa determinada pelo projeto a partir de componentes testados em unidade.

Muitas vezes há uma tendência de tentar integração não incremental; isto é, construir o programa usando uma abordagem big bang. Todos os componentes são combinados com antecedência. O programa inteiro é testado como um todo. E, usualmente, o resultado é o caos!



Adotar a abordagem “big bang” é uma estratégia ineficaz, destinada a fracassar. Integre incrementalmente, testando enquanto trabalha.



Ao desenvolver um cronograma de projeto, você terá de considerar a maneira pela qual a integração ocorrerá, de forma que os componentes estejam disponíveis quando forem necessários.

São encontrados muitos erros. A correção é difícil porque o isolamento das causas é complicado pela vasta expansão do programa inteiro. Uma vez corrigidos esses erros, novos erros aparecem e o processo parece não ter fim.

A integração incremental é o oposto da abordagem big bang. O programa é construído e testado em pequenos incrementos, em que os erros são mais fáceis de isolar e corrigir; as interfaces têm maior probabilidade de ser testadas completamente; e uma abordagem sistemática de teste pode ser aplicada. Nos próximos parágrafos, algumas estratégias de integração incremental diferentes serão discutidas.

Integração descendente. *Teste de integração descendente (top-down)* é uma abordagem incremental para a construção da arquitetura de software. Os módulos são integrados deslocando-se para baixo através da hierarquia de controle, começando com o módulo de controle principal (programa principal). Módulos subordinados ao módulo de controle principal são incorporados à estrutura de uma maneira: primeiro-em-profundidade ou primeiro-em-largura (*depth-first* ou *breadth-first*).

Na Figura 17.5, *integração primeiro-em-profundidade* integra todos os componentes em um caminho de controle principal da estrutura do programa. A seleção de um caminho principal é de certa forma arbitrária e depende das características específicas da aplicação. Por exemplo, selecionando o caminho da esquerda, os componentes M_1 , M_2 , M_5 seriam integrados primeiro. Em seguida, M_8 ou (se necessário para o funcionamento apropriado de M_2) seria integrado M_6 . Depois, são criados os caminhos de controle central e da direita. A *integração primeiro-em-largura* incorpora todos os componentes diretamente subordinados a cada nível, movendo-se através da estrutura horizontalmente. Pela figura, os componentes M_2 , M_3 e M_4 seriam integrados primeiro. Em seguida vem o próximo nível de controle, M_5 , M_6 e assim por diante. O processo de integração é executado em uma série de cinco passos:

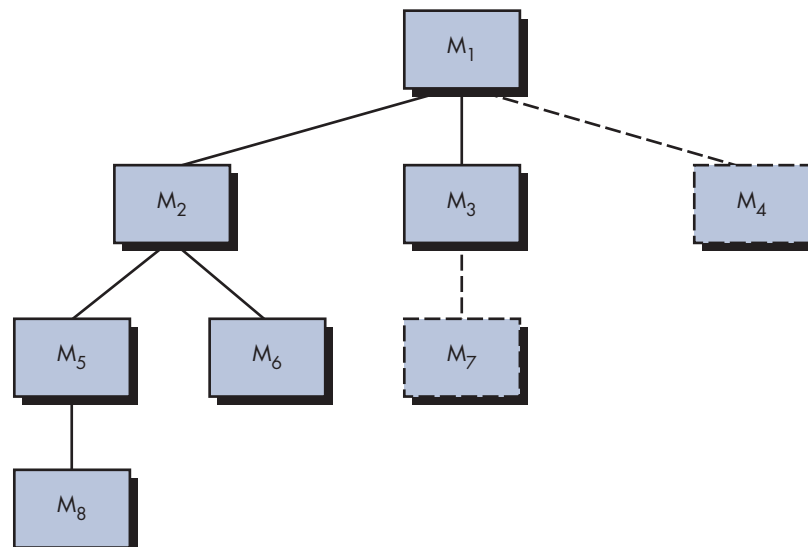


Quais são os passos para a integração descendente?

1. O módulo de controle principal é utilizado como um testador (*test driver*) e todos os componentes diretamente subordinados ao módulo de controle principal são substituídos por pseudocontroladores.
2. Dependendo da abordagem de integração selecionada (isto é, primeiro-em-profundidade ou primeiro-em-largura), pseudocontroladores subordinados são substituídos, um de cada vez, pelos componentes reais.

FIGURA 17.5

Integração descendente



3. Os testes são feitos à medida que cada componente é integrado.
4. Ao fim de cada conjunto de testes, outro pseudocontrolador substitui o componente real.
5. O teste de regressão (discutido mais adiante nesta seção) pode ser executado para garantir que não tenham sido introduzidos novos erros.

O processo continua a partir do passo 2 até que toda a estrutura do programa esteja concluída.

A estratégia de integração descendente verifica os principais pontos de controle ou decisão antecipada no processo de teste. Em uma estrutura de programa bem construída, a tomada de decisão ocorre nos níveis superiores na hierarquia e, portanto, é encontrada primeiro. Se existirem problemas de controle principal, um reconhecimento prévio é essencial. Se for selecionada a integração em profundidade, uma função completa do software pode ser implementada e demonstrada. A demonstração antecipada da capacidade funcional é um gerador de confiança para todos os interessados no programa.

A estratégia descendente parece relativamente descomplicada, mas na prática, podem surgir problemas logísticos. O mais comum desses problemas ocorre quando o processamento em baixos níveis na hierarquia é necessário para testar adequadamente níveis superiores. Pseudocontroladores substituem módulos de baixo nível no início do teste de cima para baixo; portanto, nenhum dado significativo pode fluir ascendentemente na estrutura do programa. Você, como testador, tem três escolhas: (1) adiar muitos testes até que os pseudocontroladores sejam substituídos pelos módulos reais, (2) desenvolver pseudocontroladores com funções limitadas que simulam o módulo real, ou (3) integrar o software de baixo para cima (ascendente).

A primeira abordagem (adiar os testes até que os pseudocontroladores sejam substituídos por módulos reais) pode fazer você perder algum controle sobre a correspondência entre testes específicos e incorporação de módulos específicos. Isso pode resultar em dificuldades para determinar a causa de erros e tende a violar a natureza altamente restrita da abordagem descendente. A segunda abordagem é prática, mas pode levar a uma complicação significativa, à medida que os pseudocontroladores se tornam mais e mais complexos. A terceira abordagem, chamada de integração ascendente, é discutida nos próximos parágrafos.

Integração ascendente. O teste de integração ascendente (*bottom-up*), como o nome diz, começa a construção e o teste com *módulos atômicos* (isto é, componentes nos níveis mais baixos na estrutura do programa). Devido aos componentes serem integrados de baixo para cima, a funcionalidade proporcionada por componentes subordinados a um dado nível está sempre disponível e a necessidade de pseudocontroladores é eliminada. Uma estratégia de integração ascendente pode ser implementada com os seguintes passos:

1. Componentes de baixo nível são combinados em agregados (*clusters*, também chamados de *builds*, construções) que executam uma subfunção específica de software.
2. Um pseudocontrolador (um programa de controle para teste) é escrito para coordenar entrada e saída do caso de teste.
3. O agregado é testado.
4. Os pseudocontroladores (*drivers*) são removidos, e os agregados são combinados movendo-se para cima na estrutura do programa.

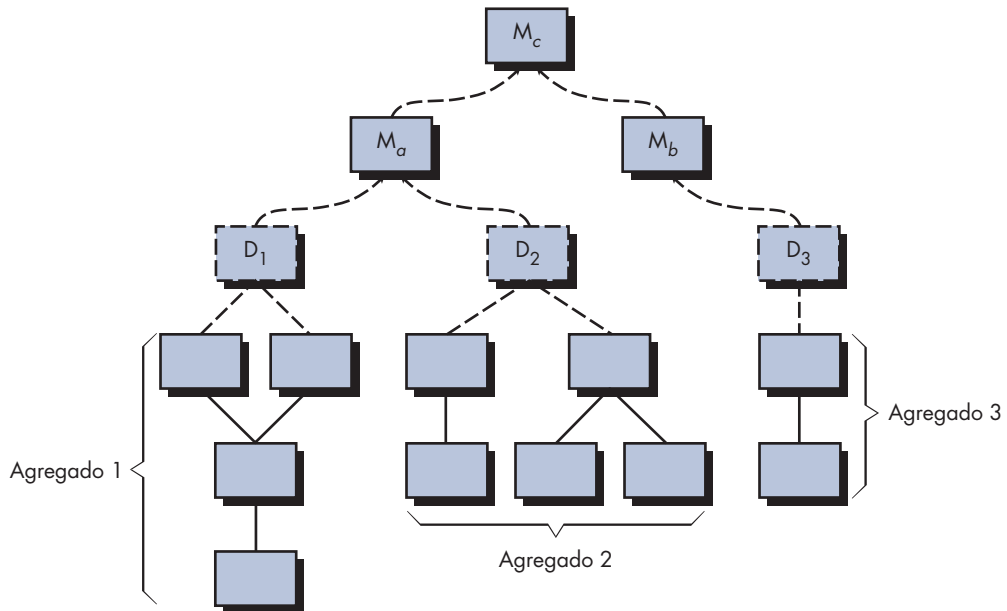
A integração segue o padrão ilustrado na Figura 17.6. Os componentes são combinados para formar os agregados (*clusters*) 1, 2 e 3. Cada um dos agregados é testado usando um pseudocontrolador (mostrado como um bloco tracejado). Componentes nos agregados 1 e 2 são subordinados a M_a . Os pseudocontroladores D_1 e D_2 são removidos e os agregados são interfaceados diretamente a M_a . De forma semelhante, o pseudocontrolador D_3 para o agregado 3 é removido antes da integração com o módulo M_b . M_a e M_b serão ambos integrados finalmente com o componente M_c , e assim por diante.

? Quais problemas podem ser encontrados quando é escolhida a integração descendente?

? Quais são os passos para a integração ascendente?

PONTO-CHAVE

A integração ascendente elimina a necessidade de pseudocontroladores complexos.

FIGURA 17.6**Integração
ascendente**

À medida que a integração se move para cima, a necessidade de pseudocontroladores de testes separados diminui. Na verdade, se os níveis descendentes da estrutura do programa forem integrados de cima para baixo, o número de pseudocontroladores pode ser bastante reduzido e a integração de agregados fica bastante simplificada.



O teste de regressão é uma estratégia importante para reduzir “efeitos colaterais”. Execute testes de regressão toda vez que for feita uma alteração grande no software (incluindo a integração de novos componentes).

Teste de regressão. Cada vez que um novo módulo é acrescentado como parte do teste de integração, o software muda. Novos caminhos de fluxo de dados são estabelecidos, podem ocorrer novas entradas e saídas e nova lógica de controle é chamada. Essas alterações podem causar problemas com funções que antes funcionavam corretamente. No contexto de uma estratégia de teste de integração, o teste de regressão é a reexecução do mesmo subconjunto de testes que já foram executados para assegurar que as alterações não tenham propagado efeitos colaterais indesejados.

Em um contexto mais amplo, testes bem-sucedidos (de qualquer tipo) resultam na descoberta de erros, e os erros devem ser corrigidos. Sempre que o software é corrigido, algum aspecto da configuração do software (o programa, sua documentação, ou os dados que o suportam) é alterado. O teste de regressão ajuda a garantir que as alterações (devido ao teste ou por outras razões) não introduzam comportamento indesejado ou erros adicionais.

O teste de regressão pode ser executado manualmente, reexecutando um subconjunto de todos os casos de teste ou usando ferramentas automáticas de captura/reexecução. Ferramentas de captura/reexecução permitem que o engenheiro de software capture casos de teste e resultados para reexecução e comparação subsequente. O conjunto de teste de regressão (o subconjunto de testes a ser executados) contém três classes diferentes de casos de teste:

- Uma amostra representativa dos testes que usam todas as funções do software.
- Testes adicionais que focalizam as funções de software que podem ser afetadas pela alteração.
- Testes que focalizam os componentes do software que foram alterados.

À medida que o teste de integração progride, o número de testes de regressão pode crescer muito. Portanto, o conjunto de testes de regressão deve ser projetado de forma a incluir somente aqueles testes que tratam de uma ou mais classes de erros em cada uma das funções principais do programa. É impraticável e ineficiente reexecutar todos os testes para todas as funções do programa quando ocorre uma alteração.

PONTO-CHAVE

O teste fumaça pode ser caracterizado como uma estratégia de integração rolante. O software é recriado (com novos componentes acrescentados) e o teste fumaça é realizado todos os dias.

Teste fumaça. *Teste fumaça* é uma abordagem de teste de integração usada frequentemente quando produtos de software são desenvolvidos. É projetado como um mecanismo de marcapasso para projetos com prazo crítico, permitindo que a equipe de software avalie o projeto frequentemente. Em essência, a abordagem teste fumaça abrange as seguintes atividades:

1. Componentes de software que foram traduzidos para um código são integrados em uma “construção” (*build*). Uma construção inclui todos os arquivos de dados, bibliotecas, módulos reutilizáveis e componentes necessários para implementar uma ou mais funções do produto.
2. Uma série de testes é criada para expor erros que impedem a construção de executar corretamente sua função. A finalidade deverá ser descobrir erros “bloqueadores” (*showstopper*) que apresentam a mais alta probabilidade de atrasar o cronograma do software.
3. A construção é integrada com outras construções, e o produto inteiro (em sua forma atual) passa diariamente pelo teste fumaça. A abordagem de integração pode ser descendente ou ascendente.

“Trate a construção diária (*daily build*) como o marcapasso do projeto. Se não houver marcapasso, o projeto está morto.”

Jim McCarthy

A frequência diária de teste do produto inteiro pode surpreender alguns leitores. No entanto, os testes frequentes dão, tanto aos gerentes quanto aos profissionais, uma ideia realística do progresso do teste de integração. McConnell [McC96] descreve o teste fumaça da seguinte maneira:

O teste fumaça deve usar o sistema inteiro de ponta a ponta. Ele não precisa ser exaustivo, mas deve ser capaz de expor os principais problemas. O teste fumaça deve ser bastante rigoroso de forma que, se a construção passar, você pode assumir que ele é estável o suficiente para ser testado mais rigorosamente.

O teste fumaça proporciona muitos benefícios quando aplicado a projetos de engenharia de software complexo e de prazo crítico:


- *O risco da integração é minimizado.* Devido aos teste fumaça serem feitos diariamente, as incompatibilidades e outros erros de bloqueio são descobertos logo, reduzindo assim a probabilidade de impacto sério no cronograma quando os erros são descobertos.
- *A qualidade do produto final é melhorada.* Devido ao fato de a abordagem ser orientada para a construção (integração), o teste fumaça pode descobrir erros funcionais, bem como erros de arquitetura e de projeto no nível de componente. Se esses erros forem corrigidos logo, resultará em melhor qualidade do produto.
- *O diagnóstico e a correção dos erros são simplificados.* Como todas as abordagens de teste de integração, os erros descobertos durante o teste fumaça provavelmente estarão associados com os “novos incrementos do software” — ou seja, o software que acaba de ser acrescentado à(s) construção(ões) é uma causa provável de um erro que acaba de ser descoberto.
- *É mais fácil avaliar o progresso.* A cada dia que passa, uma parte maior do software já está integrada e é demonstrado que funciona. Isso melhora o moral da equipe e dá aos gerentes uma boa indicação de que houve progressos.

Quais benefícios é possível obter do teste fumaça?

WebRef

Indicações para comentários sobre estratégias de teste podem ser encontradas em www.qalinks.com.

Opções estratégicas. Tem havido muita discussão (por exemplo, [Bei84]) sobre as vantagens e desvantagens relativas do teste de integração descendente *versus* ascendente. Em geral, as vantagens de uma estratégia tendem a resultar em desvantagens para outra. A maior desvantagem da abordagem descendente é a necessidade dos pseudocontroladores (*stubs*) e as dificuldades de teste que podem ser associadas com eles. Problemas associados com pseudocontroladores podem ser compensados pela vantagem de se testar antecipadamente as funções principais de controle. A maior desvantagem da integração ascendente é que “o programa como uma entidade não existe enquanto não for acrescentado o último módulo” [Mye79]. Essa desvantagem é compensada por um projeto de casos de teste mais simples e pela ausência de pseudocontroladores.

 O que é um “módulo crítico” e por que devemos identificá-lo?


A escolha de uma estratégia de integração depende das características do software e, algumas vezes, do cronograma do projeto. Em geral, uma abordagem combinada (também chamada de *teste sanduíche*), que usa testes descendentes para níveis superiores da estrutura do programa, acoplados com testes ascendentes para níveis subordinados, pode ser o melhor compromisso.

À medida que o teste de integração é conduzido, o testador deve identificar os módulos críticos. Um *módulo crítico* tem uma ou mais das seguintes características: (1) aborda vários requisitos de software, (2) tem um alto nível de controle (reside em posição relativamente alta na estrutura do programa), (3) é complexo ou sujeito a erros, ou (4) tem requisitos de desempenho bem definidos. Os módulos críticos devem ser testados o mais cedo possível. Além disso, os testes de regressão devem focalizar a função de módulo crítico.

Artefatos do teste de integração. Um plano global para integração do software e uma descrição dos testes específicos são documentados em uma *Especificação de Teste*. Esse produto incorpora um plano de teste e um documento de teste e torna-se parte da configuração do software. O teste é dividido em fases e construções que tratam de características funcionais e comportamentais específicas do software. Por exemplo, o teste de integração para o sistema de segurança *CasaSegura* pode ser dividido nas seguintes fases de teste:

- *Interação com o usuário* (entrada e saída de comandos, representação da tela, processamento e representação de erros)
- *Processamento do sensor* (aquisição da saída do sensor, determinação das condições do sensor, ações necessárias como consequência das condições)
- *Funções de comunicação* (habilidade para se comunicar com a estação de monitoramento central)
- *Processamento do alarme* (testes de ações do software que ocorrem quando um alarme é encontrado)

Cada uma dessas fases de teste de integração representa uma categoria funcional ampla dentro do software e geralmente pode ser relacionada a um domínio específico dentro da arquitetura do software. Portanto, são criadas construções de programa (grupos de módulos) para corresponder a cada fase. Os critérios e testes correspondentes a seguir são aplicados a todas as fases de teste:

 Qual critério deverá ser utilizado para projetar testes de integração?

Integridade da interface. As interfaces interna e externa são testadas à medida que cada módulo (ou agregado) é incorporado à estrutura.

Validade funcional. São executados testes destinados a descobrir erros funcionais.

Conteúdo de informação. São executados testes para descobrir erros associados com estruturas de dados locais ou globais.

Desempenho. São executados testes destinados a verificar os limites de desempenho estabelecidos durante o projeto do software.

Um cronograma para a integração, o desenvolvimento de software de uso geral e tópicos relacionados são também discutidos como parte do plano de teste. São estabelecidas as datas de início e fim para cada fase e são definidas “janelas de disponibilidade” para módulos submetidos a teste de unidade. Uma breve descrição do software de uso geral (pseudocontroladores e pseudocontrolados) concentra-se nas características que poderiam requerer dedicação especial. Finalmente, são descritos o ambiente e os recursos de teste. Configurações de hardware não usuais, simuladores exóticos e ferramentas ou técnicas especiais de teste são alguns dos muitos tópicos que também podem ser discutidos.

Em seguida, é descrito o procedimento detalhado de teste que é necessário para realizar o plano de teste. São descritas a ordem de integração e os testes correspondentes em cada etapa de integração. É incluída também uma lista de todos os casos de testes (anotados para referência subsequente) e dos resultados esperados.

Um histórico dos resultados reais do teste, problemas ou peculiaridades é registrado em um *Relatório de Teste* que pode ser anexado à *Especificação de Teste*, caso desejado. Essas informações podem ser vitais durante a manutenção do software. São apresentados também referências e apêndices apropriados.

Assim como quaisquer outros elementos de uma configuração de software, o formato da especificação de teste pode ser adaptado às necessidades locais de uma organização de engenharia de software. Porém, é importante notar que uma estratégia de integração (contida em um plano de teste) e detalhes do teste (descritos em um procedimento de teste) são ingredientes essenciais e devem estar presentes.

17.4 ESTRATÉGIAS DE TESTE PARA SOFTWARE ORIENTADO A OBJETO³

De forma simplificada, o objetivo do teste é encontrar o maior número possível de erros com um esforço gerenciável durante um intervalo de tempo realístico. Embora esse objetivo fundamental permaneça inalterado para software orientado a objetos, a natureza do software orientado a objeto muda tanto a estratégia quanto a tática de teste (Capítulo 19).

17.4.1 Teste de unidade no contexto OO

Quando consideramos o software orientado a objeto, o conceito de unidades se modifica. O encapsulamento controla a definição de classes e objetos. Isso significa que cada classe e cada instância de uma classe (objeto) empacotam atributos (dados) e as operações que manipulam esses dados. Uma classe encapsulada é usualmente o foco do teste de unidade. No entanto, operações (métodos) dentro da classe são as menores unidades testáveis. Como uma classe pode conter um conjunto de diferentes operações, e uma operação em particular pode existir como parte de um conjunto de diferentes classes, a tática aplicada a teste de unidade precisa modificar-se.

Não podemos mais testar uma única operação isoladamente (a visão convencional do teste de unidade) mas sim como parte de uma classe. Para ilustrar, considere uma hierarquia de classe na qual uma operação *X* é definida para a superclasse e é herdada por várias subclasses. Cada subclasse usa uma operação *X*, mas é aplicada dentro do contexto dos atributos e operações privadas definidas para a subclasse. Como o contexto no qual a operação *X* é utilizada varia de maneira sutil, torna-se necessário testar a operação *X* no contexto de cada uma das subclasses. Isso significa que testar a operação *X* isoladamente (a abordagem de teste de unidade convencional) é usualmente ineficaz no contexto orientado a objeto.

O teste de classe para software OO é o equivalente ao teste de unidade para software convencional. Diferentemente do teste de unidade do software convencional, que tende a focalizar o detalhe algorítmico de um módulo e os dados que fluem através da interface do módulo, o teste de classe para software OO é controlado pelas operações encapsuladas na classe e pelo estado de comportamento da classe.

17.4.2 Teste de integração no contexto OO

Devido ao software orientado a objeto não ter uma estrutura óbvia de controle hierárquico, as estratégias tradicionais de integração descendente e ascendente (Seção 17.3.2) têm pouco significado. Além disso, integrar operações uma de cada vez em uma classe (a abordagem convencional de integração incremental) frequentemente é impossível devido “às interações diretas e indiretas dos componentes que formam a classe” [Ber93].

Há duas estratégias diferentes para teste de integração de sistemas OO [Bin94b]. A primeira, *teste baseado em sequência de execução (thread-based testing)*, integra o conjunto de classes necessárias para responder a uma entrada ou um evento do sistema. Cada sequência de

PONTO-CHAVE

O teste de classe para OO é análogo ao teste de módulo para software convencional. Não é aconselhável testar operações isoladamente.

³ Os conceitos básicos orientados a objeto são apresentados no Apêndice 2.