CAPÍTULO

19 TESTANDO APLICAÇÕES ORIENTADAS A OBJETO

CONCEITOS-CHAVE

No Capítulo 18, vimos que o objetivo do teste é encontrar o maior número possível de erros com um trabalho razoável durante um intervalo de tempo real. Embora esse objetivo fundamental permaneça inalterado, para software orientado a objeto (*object-oriented*, OO), a natureza dos programas orientados a objeto muda tanto a estratégia de teste quanto suas táticas.

Poderíamos argumentar que, como o tamanho das bibliotecas de classe reutilizáveis aumenta, uma maior reutilização dispensará a necessidade de um teste mais pesado dos sistemas orientados a objeto. Exatamente o oposto é verdadeiro. Binder [Bin94b] discute isso quando diz:

> Cada reutilização é um novo contexto de uso e é prudente o reteste. Parece provável que mais testes sejam necessários e não menos, para se atingir uma alta confiabilidade em sistemas orientados a objeto.

Para testar adequadamente os sistemas orientados a objeto, três coisas precisam ser feitas: (1) a definição do teste deve ser ampliada para incluir as técnicas de descoberta de erro aplicadas à análise orientada a objeto e modelos de projeto, (2) a estratégia para teste de unidade e de integração deve mudar significativamente e (3) o projeto de casos de teste deve levar em conta as características especiais do software orientado a objeto.

PANORAMA

O que é? A arquitetura do software orientado a objeto (OO) resulta em uma série de subsistemas em camadas que encapsulam classes colaboradoras. Cada um desses elementos de sistema (subsistemas e classes) executa funções que ajudam a satisfazer os requisitos do sistema. É necessário testar um sistema orientado a objeto em diversos níveis em um esforço para descobrir erros que podem ocorrer à medida que as classes colaboram umas com as outras e os subsistemas se comunicam através das camadas da arquitetura.

Quem realiza? O teste orientado a objeto é executado por engenheiros de software e especialistas em teste.

Porque é importante? É necessário executar o programa antes que seja entregue ao cliente para remover todos os erros, de maneira que o usuário não passe por nenhuma frustração com um produto de má qualidade. Para encontrar o maior número possível de erros, devem ser feitos testes sistematicamente e projetados casos de teste usando técnicas bem definidas.

Quais são as etapas envolvidas? O teste orientado a objeto é estrategicamente análogo ao de sistemas convencionais, mas é taticamente diferente. Em virtude de os modelos de análise e projeto orientados a objeto serem similares em estrutura e conteúdo ao programa orientado a objeto resultante, o "teste" é iniciado com a revisão dos modelos. Uma vez gerado o código, começa o teste orientado a objeto "inicialmente" com o teste de classes. É projetada uma série de testes que exercitam as operações de classe e examinam se existem erros enquanto uma classe colabora com outras. À medida que as classes são integradas para formar um subsistema, aplicam-se testes baseados em sequências de execução (*thread*), baseados no uso e o teste de conjunto juntamente com abordagens baseadas em falhas para usar completamente as classes colaboradoras. Por fim, são usados casos de uso (desenvolvidos como parte do modelo de requisitos) para descobrir erros em nível de validação de software.

Qual é o artefato? É projetado e documentado um conjunto de casos de teste, desenvolvidos para usar as classes, suas colaborações e comportamentos; são definidos os resultados esperados e registrados os obtidos.

Como garantir que o trabalho foi realizado corretamente? Ao iniciar o teste, mude o seu ponto de vista. Tente por todos os meios "quebrar" o software! Projete casos de teste de maneira disciplinada e reveja os casos que você criou para que sejam completos.

19.1 AMPLIANDO A VERSÃO DO TESTE

A construção de software orientado a objeto inicia-se com a criação dos requisitos (análises) e projetos de modelo.[1] Devido à natureza evolucionária do paradigma da engenharia de software orientado a objeto, esses modelos começam como representações relativamente informais de requisitos de sistema e evoluem para modelos detalhados de classes, relações entre classes, projeto e alocação de sistema e projeto de objeto (incorporando um modelo da conectividade de objeto via mensagem). Em cada estágio, os modelos podem ser "testados" para descobrir erros antes de sua propagação para a próxima iteração.

Pode-se argumentar que a revisão da análise orientada a objeto e dos modelos de projeto é especialmente útil porque a mesma construção semântica (por exemplo, classes, atributos, operações, mensagens) aparece nos níveis de análise, projeto e código. A detecção de um problema na definição de atributos de classe durante a análise evitará efeitos colaterais que poderiam ocorrer se o problema não fosse descoberto antes do projeto ou codificação (ou mesmo a próxima iteração da análise).

AVISO

Embora a revisão da análise orientada a objeto e dos modelos de projeto seja parte integral do "teste" de uma aplicação orientada a objeto, lembre-se de que isso não é suficiente. Devem-se realizar testes executáveis também.

Por exemplo, considere uma classe na qual um conjunto de atributos é definido durante a primeira iteração da análise. Um atributo estranho é acrescentado à classe (decorrente da falta de entendimento do domínio do problema). Duas operações são então especificadas para manipular o atributo. É feita uma revisão e um especialista em domínios aponta o problema. Eliminando o atributo estranho nesse estágio, problemas e trabalho desnecessário podem ser evitados durante a análise:

1. Poderiam ter sido geradas subclasses especiais para acomodar o atributo desnecessário ou suas exceções. O trabalho envolvido na criação de subclasses desnecessárias foi evitado.
2. Uma interpretação incorreta da definição de classe pode levar a relações de classe incorretas ou estranhas.
3. O comportamento do sistema ou suas classes pode ser caracterizado inadequadamente para acomodar o atributo estranho.

Se o problema não fosse descoberto (por meio de revisão antecipada) durante a análise e se propagasse, poderiam ocorrer as seguintes situações durante o projeto:

1. Pode ocorrer alocação imprópria da classe para o subsistema e/ou tarefas durante o projeto do sistema.
2. Pode ser despendido trabalho de projeto desnecessário para criar o projeto procedural para as operações que lidam com o atributo estranho.
3. O modelo de mensagens estará incorreto (porque devem ser designadas mensagens para as operações estranhas).

> "As ferramentas que utilizamos têm uma profunda (e definitiva!) influência sobre nossos hábitos de pensar, e, portanto, em nossas habilidades de pensar."
>
> **Edsger Dijkstra**

Se o problema não for detectado durante o projeto e passar para a atividade de codificação, será despendido um considerável esforço na geração de código que implementa um atributo desnecessário, duas operações desnecessárias, mensagens que controlam comunicação entre objetos e muitos outros problemas relacionados. Além disso, o teste da classe absorverá mais tempo do que o necessário. Uma vez descoberto o problema, deve ser feita a modificação do sistema, levando-se sempre em conta a alta possibilidade de efeitos colaterais causados pela alteração.

Durante estágios posteriores de seu desenvolvimento, os modelos de análise orientada a objeto (*object-oriented analysis*, OOA) e projeto orientado a objeto (*object-oriented design*, OOD) proporcionam informações substanciais sobre a estrutura e comportamento do sistema. Por essa razão, esses modelos deverão ser submetidos a rigorosa revisão antes da geração do código.

---

1 As técnicas de análise e modelagem de projeto são apresentadas na Parte 2 deste livro. Os conceitos orientados a objeto básicos são apresentados no Apêndice 2.

Todos os modelos orientados a objeto deverão ser testados (nesse contexto, o termo *teste* incorpora revisões técnicas) quanto à sua exatidão, integralidade e consistência com o contexto de sintaxe do modelo, semânticas e pragmatismo [Lin94a].

19.2 Testando Modelos de Análise Orientada a Objeto (OOA) e Projeto Orientado a Objeto (OOD)

Análises e modelos de projeto não podem ser testados no sentido convencional, pois não podem ser executados. No entanto, podem ser usadas revisões técnicas (Capítulo 15) para examinar sua exatidão e consistência.

19.2.1 Exatidão dos modelos de OOA e OODs

A notação e a sintaxe usadas para representar modelos de análise e projeto estará vinculada a métodos específicos de análise e projeto escolhidos para o projeto. Consequentemente a exatidão sintática é julgada com base no uso apropriado da simbologia; cada modelo é revisado para garantir que sejam mantidas as convenções apropriadas de modelagem.

Durante a análise e o projeto, pode-se verificar a exatidão semântica de acordo com o modelo no domínio do problema do mundo real. Se o modelo reflete com precisão o mundo real, (com um nível de detalhes apropriado para o estágio de desenvolvimento no qual o modelo é revisado), ele é semanticamente correto. Para determinar se de fato o modelo reflete os requisitos do mundo real, deve ser apresentado aos especialistas em domínio de problema que examinarão as definições de classe e a hierarquia quanto a omissões e ambiguidades. Relações de classe (conexões de instância) são avaliadas para determinar se refletem precisamente as conexões do objeto no mundo real.[2]

19.2.2 Consistência dos modelos orientados a objeto

A consistência de modelos orientados a objeto pode ser avaliada "considerando-se a relação entre entidades no modelo. Um modelo de análise ou projeto inconsistente tem representações em uma parte que não são refletidas corretamente em outras partes do modelo" [McG94].

Para avaliarmos a consistência, devemos examinar cada classe e suas conexões com as outras classes. O modelo classe-responsabilidade-colaboração (CRC) ou um diagrama de relacionamentos entre objetos pode ser usado para facilitar tal atividade. Conforme vimos no Capítulo 6, o modelo CRC é composto de cartões de índice CRC. Cada cartão lista o nome da classe, suas responsabilidades (operações) e colaboradores (outras classes às quais envia mensagens e das quais depende para a execução de suas responsabilidades). As colaborações implicam uma série de relacionamentos (conexões) entre classes do sistema orientado a objeto. O modelo de relacionamento de objeto fornece uma representação gráfica das conexões entre classes. Todas essas informações podem ser obtidas do modelo de análise (Capítulos 6 e 7).

Para avaliar o modelo de classe, recomendam-se os seguintes passos [McG94]:

1. **Rever o modelo CRC e o modelo de relacionamento de objeto.** Faça uma verificação comparativa para garantir que todas as colaborações relacionadas ao modelo de requisitos estão corretamente refletidas em ambos.
2. **Inspecionar a descrição de cada cartão de índice CRC para determinar se uma responsabilidade delegada faz parte da definição do colaborador.** Por exemplo, considere uma classe definida para um sistema de controle de ponto de venda que se chama **CréditoDeVenda**. Essa classe tem um cartão de índice CRC conforme mostra a Figura 19.1.

2 Casos de uso podem ser valiosos para rastrear modelos de análise e projeto em um cenário de uso no mundo real para o sistema orientado a objeto.

FIGURA 19.1
Um exemplo de um cartão de índice CRC usado para revisão

nome da classe: créditoDeVenda	
tipo da classe: eventoDeTransação	
característica da classe: nãoTangível, atômica, sequencial, permanente, protegida	
responsabilidades:	colaboradores:
lerCartãoCrédito	cartãoDeCrédito
pegarAutorização	autoridadeDeCédito
solicitarValorDaCompra	tíqueteDoProduto
	registroDoVendedor
	arquivoDeBalanço
gerarConta	conta

Para essa coleção de classes e colaborações, pergunte se uma responsabilidade (por exemplo, *lerCartãoCrédito*) é atendida se for delegada ao colaborador citado (**CartãoDeCrédito**). Isto é, a classe **CartãoDeCrédito** tem uma operação que a habilita a ser lida? Nesse caso, a resposta é "sim". A relação-objeto é percorrida para garantir que todas essas conexões são válidas.

3. **Inverta a conexão para garantir que cada colaborador ao qual é solicitado serviço esteja recebendo solicitações de uma origem razoável.** Por exemplo, se a classe **CartãoDeCrédito** receber uma solicitação para **purchase amount (valor da compra)** da classe **CréditoDeVenda**, haverá um problema. **CartãoDeCrédito** não conhece o valor da compra.
4. Usando as conexões invertidas examinadas na etapa 3, determine se podem ser necessárias outras classes ou se as responsabilidades estão corretamente agrupadas entre as classes.
5. **Determine se responsabilidades solicitadas mais amplas podem ser combinadas em uma única responsabilidade.** Por exemplo, *lerCartãoCrédito* e *pegaAutorização* ocorrem em todas as situações. Elas podem ser combinadas em uma responsabilidade de *validação de solicitação de crédito*, que incorpora a obtenção do número do cartão de crédito e obtenção da autorização.

Você deverá aplicar os passos de 1 a 5 iterativamente a cada classe e através de cada evolução do modelo de requisitos.

Uma vez criado o modelo de projeto (Capítulos 9 a 11), você deverá realizar também revisões do projeto do sistema e projeto do objeto. O projeto do sistema representa a arquitetura geral do produto, os subsistemas que compõem o produto, a maneira pela qual os subsistemas são alocados a processadores, a alocação de classes para subsistemas e o projeto da interface do usuário. O modelo de objeto apresenta os detalhes de cada classe e as atividades de mensagens necessárias para implementar colaborações entre classes.

O projeto do sistema é revisado examinando-se o modelo objeto-comportamento desenvolvido durante a análise orientada a objeto e mapeando o comportamento requerido do sistema em relação aos subsistemas projetados para atender a esse comportamento. A concorrência e a alocação de tarefas também é revista segundo o contexto de comportamento do sistema. Os estados comportamentais do sistema são avaliados para determinar quais existem concorrentemente. São usados casos de uso para exercitar o projeto da interface do usuário.

O modelo orientado a objeto deverá ser testado em relação à rede objeto-relacionamento para assegurar que todos os objetos de projeto contenham os atributos e operações necessários para implementar as colaborações definidas para cada cartão de Índice CRC. Além disso, é revisada a especificação dos detalhes de operação (isto é, os) algoritmos que implementam as operações).

19.3 Estratégias de Teste Orientado a Objeto

Conforme mencionado no Capítulo 18, a estratégia de teste de software clássica começa com o "teste no pequeno" e se amplia para o "teste no grande". De acordo com o jargão do teste de software (Capítulo 18), você começa com *teste de unidade*, depois passa para o *teste de integração* e termina com a *validação e teste do sistema*. Em aplicações convencionais, o teste de unidade focaliza a menor unidade de programa compilável — o subprograma (por exemplo, componente, módulo, sub-rotina, procedimento). Depois que cada uma dessas unidades é testada individualmente, elas são integradas em uma estrutura de programa enquanto é executada uma série de testes de regressão para descobrir erros devidos ao interfaceamento de módulos e os efeitos colaterais causados pela adição de novas unidades. Por fim, o sistema como um todo é testado para assegurar que sejam descobertos os erros em requisitos.

19.3.1 Teste de unidade em contexto orientado a objeto

PONTO-CHAVE

A menor "unidade" que pode ser testada em software orientado a objeto é a classe. O teste de classe é controlado pelas operações encapsuladas pela classe e o comportamento de estado da classe.

Quando se considera o software orientado a objeto, o conceito de unidade muda. O encapsulamento controla a definição de classes e objetos. Cada classe e cada instância de uma classe (objeto) empacotam (pacotes) atributos (dados) e as operações (também conhecidas como métodos ou serviços) que manipulam esses dados. Em vez de testar um módulo individual, a menor unidade testável é a classe encapsulada. Pelo fato de uma classe poder conter muitas operações diferentes e uma operação em particular poder existir como parte de um conjunto de classes diferentes, o significado do teste de unidade muda significativamente.

Já não podemos mais testar uma única operação isoladamente (a visão convencional do teste de unidade), mas como parte de uma classe. Considere uma hierarquia de classe na qual uma operação *X()* é definida para a superclasse e é herdada por um conjunto de subclasses. Cada subclasse usa a operação *X()*, mas ela é aplicada de acordo com o contexto de atributos privados e operações definidas para cada subclasse. O contexto no qual a operação *X()* é usada varia de maneira sutil, desse modo, é necessário testar a operação *X()* no contexto de cada uma das subclasses. Isso significa que testar a operação *X()* em um vácuo (a abordagem tradicional de teste de unidade) é ineficaz no contexto orientado a objeto.

O teste de classe para software orientado a objeto equivalente ao teste de unidade para software convencional.[3] Ao contrário do teste de unidade para 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 orientado a objeto é controlado pelas operações encapsuladas pela classe e o comportamento de estado da classe.

19.3.2 Teste de integração em contexto orientado a objeto

Como o software orientado a objeto não tem uma estrutura de controle hierárquico, as estratégias de integração convencionais de cima para baixo e de baixo para cima têm pouco significado. Além disso, integrar as operações uma de cada vez em uma classe (a abordagem convencional incremental de integração) frequentemente é impossível devido às "interações direta e indireta dos componentes que formam a classe" [Ber93].

Há duas estratégias diferentes para teste de integração de sistemas orientados a objeto [Bin94a]. A primeira, *teste baseado em sequências de execução* (*thread-based testing*), integra o conjunto de classes necessárias para responder a uma entrada ou evento para o sistema. Cada

3 Os métodos de projeto de casos de teste para classes orientadas a objeto são discutidos nas Seções 19.4 a 19.6.

PONTO-CHAVE

O teste de integração para software orientado a objeto testa um conjunto de classes necessárias para responder a determinado evento.

sequência de execução (*thread*) é integrada e testada individualmente. O teste de regressão é aplicado para garantir que não ocorram efeitos colaterais. A segunda abordagem de integração, *teste baseado em uso*, começa a construção do sistema testando aquelas classes (chamadas *classes independentes*) que usam bem poucas (se usar alguma) classes servidoras. Depois que as classes independentes são testadas, testa-se a próxima camada de classes, chamadas de *classes dependentes*, que usam as classes independentes. Essa sequência de teste de camadas de classes dependentes continua até que o sistema inteiro seja construído. Diferentemente da integração convencional, o uso de pseudocontrolador e pseudocontrolados (Capítulo 18), como operações substitutas, deve ser evitado quando possível.

Teste de conjunto [McG94] é uma etapa do teste de integração de software orientado a objeto. Aqui, um *conjunto* de classes colaboradoras (determinado examinando-se o CRC e o modelo de relacionamento de objeto) é exercitado projetando-se casos de teste que tentam descobrir erros nas colaborações.

19.3.3 Teste de validação em contexto orientado a objeto

Em nível de validação ou de sistema, os detalhes das conexões de classes desaparecem. Assim como a validação convencional, a validação de software orientado a objeto focaliza as ações visíveis pelo usuário e as saídas do sistema reconhecíveis pelo usuário. Para ajudar na criação de testes de validação, o testador deverá fundamentar-se nos casos de uso (Capítulos 5 e 6) que fazem parte do modelo de requisitos. O caso de uso proporciona um cenário com grande possibilidade de detectar erros em requisitos de interação de usuário.

Os métodos convencionais de teste caixa-preta (Capítulo 18) podem ser usados para controlar testes de validação. Além disso, pode-se optar por criar casos de teste com base no modelo de comportamento de objeto e em um diagrama de fluxo de evento criado como parte da análise orientada a objeto.

19.4 Métodos de Teste Orientados a Objeto

"Encaro os testadores como os guarda-costas do projeto. Nós defendemos o flanco de nossos desenvolvedores contra as falhas, enquanto eles concentram-se na efetivação do sucesso."

James Bach

A arquitetura de software orientado a objeto resulta em uma série de subsistemas em camada que encapsulam as classes de colaboração. Cada um desses elementos de sistema (subsistemas e classes) executa funções que ajudam a satisfazer os requisitos do sistema. É necessário testar um sistema orientado a objeto em uma variedade de níveis diferentes para descobrir erros que podem ocorrer à medida que as classes colaboram umas com as outras e os subsistemas se comunicam através de camadas da arquitetura.

Os métodos de projeto de casos de teste para software orientado a objeto continuam a evoluir. No entanto, uma abordagem geral para projeto de casos de teste orientado a objeto foi sugerida por Berard [Ber93]:

1. Cada caso de teste deverá ser identificado de forma única e associado explicitamente com a classe a ser testada.
2. Deverá ser definida a finalidade do teste.
3. Deverá ser feita uma lista das etapas de teste para cada teste e ela deverá conter:
   a. Uma lista dos estados especificados para a classe a ser testada
   b. Uma lista das mensagens e operações que serão executadas em consequência do teste
   c. Uma lista das exceções que podem ocorrer enquanto a classe é testada
   d. Uma lista de condições externas (alterações no ambiente externo ao software que deve existir para fazer corretamente o teste)
   e. Informações suplementares que ajudarão a entender ou implementar o teste

Diferentemente do projeto convencional de casos de teste, controlado por uma visão entrada-processo-saída do software ou do detalhe algorítmico dos módulos individuais, o teste

orientado a objeto concentra-se no planejamento de sequências apropriadas de operação para executar os estados de uma classe.

19.4.1 As implicações no projeto de casos de teste dos conceitos orientados a objeto

WebRef
Uma excelente coleção de publicações e recursos sobre teste orientado a objeto pode ser encontrada em **www.rbsc.com.**

À medida que uma classe evolui através dos modelos de requisitos e de projeto, torna-se alvo para o projeto de casos de teste. Como os atributos e operações são encapsulados, as operações de teste fora da classe em geral são improdutivas. Embora o encapsulamento seja um conceito de projeto essencial para software orientado a objeto, ele pode criar um pequeno obstáculo na realização do teste. Conforme afirma Binder [Bin94a], "O teste requer um relato sobre o estado concreto e abstrato de um objeto". Ainda assim, o encapsulamento pode dificultar a obtenção dessa informação. A menos que sejam inseridas operações internas para relatar os valores dos atributos da classe, um resumo do estado de um objeto pode ser difícil de obter.

A herança também pode apresentar desafios adicionais durante o projeto de casos de teste. Observamos que a cada novo contexto de uso é requerido um reteste, mesmo que tenha sido possível a reutilização. Além disso, a herança múltipla[4] complica ainda mais o teste, aumentando o número de contextos para os quais é necessário o teste [Bin94a]. Se subclasses instanciadas a partir de uma superclasse forem usadas dentro do mesmo domínio do problema, é provável que o conjunto de casos de teste derivado para a superclasse possa ser usado para testar a subclasse. No entanto, se a subclasse for usada em um contexto inteiramente diferente, os casos de teste da superclasse terão pouca aplicabilidade, e um novo conjunto de testes precisa ser projetado.

19.4.2 Aplicabilidade dos métodos convencionais de projeto de casos de teste

Os métodos de teste caixa-branca descritos no Capítulo 18 podem ser aplicados às operações definidas para uma classe. Técnicas de caminho-base, teste de ciclos ou fluxo de dados podem ajudar a garantir que cada instrução em uma operação tenha sido testada. No entanto, a estrutura concisa de muitas operações de classe sugere argumentos de que o esforço aplicado no teste caixa-branca poderia ser mais bem redirecionado para testes no nível de classe.

Os métodos de teste caixa-preta são apropriados para sistemas orientados a objeto, assim como para sistemas desenvolvidos por meio de métodos convencionais de engenharia de software. Conforme mencionado no Capítulo 18, casos de uso podem proporcionar informações úteis no projeto de testes caixa-preta e baseados em estado.

19.4.3 Teste baseado em falhas[5]

PONTO-CHAVE
A estratégia para o teste baseado em falhas é formular uma hipótese de uma série de falhas possíveis e criar testes para provar cada uma das hipóteses.

O objetivo do *teste baseado em falhas* em um sistema orientado a objeto é projetar testes que tenham grande probabilidade de descobrir falhas plausíveis. Como o produto ou sistema deve satisfazer os requisitos do cliente, o planejamento preliminar necessário para realizar o teste baseado em falhas começa com o modelo de análise. O testador procura por falhas plausíveis (aspectos da implementação do sistema que podem resultar em defeitos). Para determinar se essas falhas existem, projetam-se casos de teste para exercitar o projeto ou código.

Naturalmente, a eficácia dessas técnicas depende de como os testadores consideram uma falha plausível. Se falhas reais em um sistema orientado a objeto são vistas como não plausíveis, essa abordagem não é melhor do que qualquer técnica de teste aleatório. No entanto, se os modelos de análise e projeto puderem proporcionar conhecimento aprofundado sobre o que provavelmente pode sair errado, o teste baseado em falhas pode encontrar quantidade significativa de erros com esforço relativamente pequeno.

---

4 Um conceito orientado a objeto que deverá ser usado com extremo cuidado.

5 As Seções 19.4.3 a 19.4.6 foram adaptadas de um artigo de Brian Marick postado na Internet no newsgroup comp.testing. Essa adaptação foi incluída com permissão do autor. Para mais informações sobre esses tópicos, veja [Mar94]. Deve-se notar que as técnicas discutidas nas Seções 19.4.3 a 19.4.6 são também aplicáveis para software convencional.

Que tipos de falhas são encontrados em chamadas de operação e nas conexões de mensagens?

O teste de integração procura por falhas plausíveis em chamadas de operação ou conexões de mensagem. Três tipos de falhas encontram-se nesse contexto: resultado inesperado, uso de operação/mensagem errada e invocação incorreta. Para determinar as falhas plausíveis quando funções (operações) são invocadas, o comportamento da operação deve ser examinado.

O teste de integração se aplica tanto a atributos quanto a operações. Os "comportamentos" de um objeto são definidos pelos valores atribuídos a seus atributos. O teste deve exercitar os atributos para determinar se ocorrem os valores apropriados para tipos distintos de comportamento de objeto.

É importante notar que o teste de integração tenta encontrar erros no objeto-cliente, não no servidor. Em termos convencionais, o foco do teste de integração é determinar se existem erros no código chamador, não no código chamado. A chamada da operação é usada como um indício, uma maneira de encontrar os requisitos de teste que usam o código chamador.

19.4.4 Casos de teste e a hierarquia de classe

A herança não torna óbvia a necessidade de um teste completo de todas as classes derivadas. De fato, ela pode realmente complicar o processo de teste. Considere a seguinte situação. Uma classe **Base** contém operações *herdada()* e *redefinida()*. Uma classe **Derivada** redefine *redefinida()* para servir em um contexto local. Há pouca dúvida de que **Derivada::redefinida()** tem de ser testada porque representa um novo projeto e um novo código. Mas **Derivada::herdada()** tem de ser retestada?

PONTO-CHAVE

Mesmo que uma classe base tenha sido completamente testada, será preciso testar todas as classes derivadas dela.

Se **Derivada::herdada()** chama *redefinida()* e o comportamento de *redefinida()* foi alterado, **Derivada::herdada()** pode processar de maneira errada o novo comportamento. Portanto, precisa de novos testes apesar de o projeto e código não terem mudado. É importante notar que apenas um subconjunto de todos os testes para **Derivada::herdada()** pode ter de ser realizado. Se parte do projeto e código para *herdada()* não depende de *redefinida()* (não o chama nem chama qualquer código que o chame indiretamente), aquele código não precisa ser retestado na classe derivada.

Base::redefinida() e **Derivada::redefinida()** são duas operações diferentes com diferentes especificações e implementações. Cada uma teria um conjunto de requisitos de teste derivados da especificação e implementação. Estes investigam falhas plausíveis: falhas na integração, falhas de condição, falhas de limites e assim por diante. Mas as operações provavelmente serão similares. Seus conjuntos de requisitos de teste vão se sobrepor. Quanto melhor for o projeto orientado a objeto, maior é a sobreposição. Precisam ser obtidos novos testes somente para aqueles requisitos **Derivada::redefinida()** que não são satisfeitos pelos testes **Base::redefinida()**.

Para resumir, os testes **Base::redefinida()** são aplicados a objetos da classe **Derivada**. Entradas de teste podem ser apropriadas para as classes base e derivada, mas os resultados esperados podem ser diferentes na classe derivada.

19.4.5 Projeto de teste baseado em cenário

O teste baseado em falhas omite dois tipos principais de erro: (1) especificações incorretas e (2) interações entre subsistemas. Quando ocorrem erros associados a uma especificação incorreta, o produto não realiza o que o cliente deseja. Pode fazer alguma coisa errada ou omitir uma funcionalidade importante. Mas em qualquer circunstância, a qualidade (conformidade com os requisitos) é prejudicada. Erros associados com interação de subsistema ocorrem quando o comportamento de um subsistema cria circunstâncias (por exemplo, eventos, fluxo de dados) que faz um outro subsistema falhar.

PONTO-CHAVE

Testes baseados em cenário indicarão erros que ocorrem quando qualquer ator interage com o software.

O teste baseado em cenário concentra-se naquilo que o usuário faz, não no que o produto faz. Isso significa detectar as tarefas (por meio de casos de uso) que o usuário tem de executar e aplicá-las, bem como suas variantes como testes.

Cenários descobrem erros de interação. Mas, para tanto, os casos de teste devem ser mais complexos e realistas do que os testes baseados em falhas. Os testes baseados em cenário tendem a usar múltiplos subsistemas em um único teste (os usuários não se limitam ao uso de um subsistema de cada vez).

Como exemplo, considere o projeto de testes baseados em cenário para um editor de texto. Os casos de uso são apresentados a seguir:

Caso de uso: corrigir o esboço final

Background: é comum imprimir o esboço "final", ler e descobrir alguns erros evidentes que não estavam óbvios na tela. Esse caso de uso descreve a sequência de eventos que ocorre quando isso acontece.

1. Imprima o documento inteiro.
2. Reveja o documento alterando algumas páginas.
3. À medida que cada página é alterada, ela é impressa.
4. Algumas vezes uma série de páginas é impressa.

"Se você quer e espera que um programa funcione, provavelmente verá um programa funcionando — não verá as falhas."

Cem Kaner et al.

Esse cenário descreve duas coisas: o teste e necessidades específicas de usuário. As necessidades do usuário são óbvias: (1) um método para imprimir páginas separadas e (2) um método para imprimir um intervalo de páginas. Do ponto de vista do teste, há necessidade de testar a edição após a impressão (e vice-versa). Portanto, o testador trabalha no projeto de testes que descobrirão erros na função de edição causados pela função de impressão; erros que indicarão que as duas funções do software não são propriamente independentes.

Caso de uso: imprimir uma nova cópia

Background: alguém solicita ao usuário uma nova cópia do documento. Ela deve ser impressa.

1. Abra o documento.
2. Imprima-o.
3. Feche o documento.

Novamente, a abordagem do teste é relativamente óbvia. Exceto que esse documento não veio de lugar nenhum. Foi criado em uma tarefa anterior. Essa tarefa afeta a presente?

Em muitos editores modernos, os documentos "lembram-se" de como foram impressos da última vez. Por padrão, são impressos da mesma maneira na próxima vez. Após o cenário **Corrigir o esboço final**, basta selecionar "Imprimir" no menu e clicar o botão Imprimir na caixa de diálogo e será impressa novamente a última página corrigida. De acordo com o editor, o cenário correto deverá se parecer com este:

Embora o teste baseado em cenário tenha seus méritos, você terá um melhor retorno do tempo investido revisando casos de uso quando são desenvolvidos como parte do modelo de análise.

Caso de uso: imprimir uma nova cópia

1. Abra o documento.
2. Selecione "Imprimir" no menu.
3. Verifique se está imprimindo um intervalo de páginas; se estiver, clique para imprimir o documento inteiro.
4. Clique o botão Imprimir.
5. Feche o documento.

Esse cenário indica um erro potencial de especificação. O editor não faz aquilo que o usuário espera que ele faça. Os clientes muitas vezes não observam a verificação da etapa 3. Ficarão irritados quando encontrarem uma página na impressora quando esperavam encontrar 100. Clientes irritados indicam problemas de especificação.

Um projetista de casos de teste poderia não perceber essa dependência quando projeta os testes, mas é possível que o problema apareça durante o teste. O testador teria de argumentar com a provável resposta, "Deve ser dessa maneira que funciona!".

19.4.6 Teste da estrutura superficial e estrutura profunda

Estrutura superficial refere-se à estrutura observável externamente de um programa orientado a objeto; a estrutura imediatamente óbvia para um usuário final. Em vez de executar funções, os usuários de muitos sistemas orientados a objeto podem receber objetos para manipular de alguma forma. Qualquer que seja a interface, os testes ainda são baseados nas tarefas do usuário. A captura dessas tarefas envolve o entendimento, observação e diálogo com os usuários representativos (e os usuários não representativos que puderem ser encontrados).

PONTO-CHAVE

O teste da estrutura superficial é análogo ao teste caixa-preta. O teste de estrutura profunda é similar ao teste caixa-branca.

Certamente haverá alguma diferença no detalhe. Por exemplo, em um sistema convencional com uma interface orientada a comandos, o usuário pode utilizar a lista de todos os comandos como uma verificação do teste. Se não existiam cenários de teste para simular um comando, provavelmente o teste não considerou algumas tarefas do usuário (ou a interface tem comandos inúteis). Em uma interface baseada em objeto, o testador pode usar a lista de todos os objetos como uma verificação de teste.

Os melhores testes são criados quando o projetista encara o sistema de maneira nova ou não convencional. Por exemplo, se o sistema ou produto tem uma interface baseada em comandos, serão desenvolvidos testes mais completos se o projetista do caso de teste simular as operações independentes dos objetos. Faça perguntas do tipo, "Será que o usuário vai querer utilizar essa operação — que se aplica somente ao objeto **Scanner** — enquanto estiver trabalhando com a impressora?". Qualquer que seja o estilo da interface, o projeto do caso de teste que simula a estrutura superficial deve usar tanto objetos quanto as operações como indícios que levam a tarefas não consideradas.

"Não se envergonhe de seus erros e não os transforme em crimes."

Confúcio

A Estrutura profunda refere-se aos detalhes técnicos internos de um programa orientado a objeto, isto é, a estrutura entendida examinando-se o projeto e/ou código. O teste de estrutura profunda é projetado para simular dependências, comportamentos e mecanismos de comunicação estabelecidos como parte do modelo de projeto para software orientado a objeto.

Os requisitos e os modelos de teste são usados como base para o teste de estrutura profunda. Por exemplo, o diagrama de colaboração UML ou o modelo de distribuição representa colaborações entre objetos e subsistemas que podem não ser externamente visíveis. O projeto de casos de teste então pergunta: "Nós (como teste) capturamos alguma tarefa que usa a colaboração representada no diagrama de colaboração? Se não, por que não?".

19.5 MÉTODOS DE TESTE APLICÁVEIS NO NÍVEL DE CLASSE

AVISO

O número de permutações possíveis para teste aleatório pode crescer muito. Uma estratégia similar ao teste de matriz ortogonal pode ser usada para melhorar a eficiência de teste.

O teste "no pequeno" focaliza uma única classe e os métodos encapsulados pela classe. Teste aleatório e particionamento são métodos que podem ser usados para simular uma classe durante o teste orientado a objeto.

19.5.1 Teste aleatório para classes orientadas a objeto

Para apresentar breves ilustrações desses métodos, considere uma aplicação bancária na qual uma classe **Conta** tem as seguintes operações: *abrir()*, *estabelecer()*, *depositar()*, *retirar()*, *obterSaldo()*, *resumir()*, *limiteDeCrédito()* e *fechar()* [Kir94]. Cada uma dessas operações pode ser aplicada para **Conta,** mas certas restrições (por exemplo, primeiro a conta precisa ser aberta para que as outras operações possam ser aplicadas e fechada depois que todas as operações são completadas) são implícitas à natureza do problema. Mesmo com essas restrições, há muitas permutações das operações. O histórico de comportamento mínimo de uma instância de **Conta** inclui as seguintes operações:

abrir • estabelecer • depositar • retirar • fechar

Isso representa a sequência mínima de teste para **Conta**. No entanto, pode ocorrer uma ampla variedade de outros comportamentos nessa sequência:

abrir • estabelecer • depositar • [depositar | retirar | obterSaldo | resumir | limiteDeCrédito]" • retirar • fechar

Uma variedade de diferentes sequências de operações pode ser gerada aleatoriamente. Por exemplo:

Caso de teste r_1: **abrir•estabelecer•depositar•depositar• obterSaldo •resumir•retirar•fechar**
Caso de teste r_2: **abrir•estabelecer•depositar•retirar•depositar•obterSaldo•limiteDeCrédito •retirar•fechar**

Esses e outros testes de ordem aleatória podem ser usados para exercitar diferentes históricos de duração de instância de classe.

CasaSegura

Teste de classe

Cena: Escritório da Shakira.

Atores: Jamie e Shakira — membros da equipe de engenharia de software que estão trabalhando no projeto de um caso de teste para função de segurança do *CasaSegura*.

Conversa:

Shakira: Desenvolvi alguns testes para a classe Detector [Figura 10.4] — você sabe, aquela que permite acesso a todos os objetos **Sensor** para a função de segurança. Entendeu?

Jamie (rindo): Claro, é aquela que você usou para acrescentar o sensor "raiva de cachorro".

Shakira: Essa mesma. Bem, ela tem uma interface com quatro operações: ler(), habilitar(), desabilitar(), e testar(). Para que um sensor possa ser lido, tem de ser primeiro habilitado. Uma vez habilitado, pode ser lido e testado. Ele pode ser desabilitado a qualquer instante, exceto se uma condição de alarme estiver sendo processada. Assim, eu defini uma sequência simples de teste que irá simular sua história comportamental. [Mostra a Jamie a seguinte sequência.]

#1: habilitar•testar•ler•desabilitar

Jamie: Vai funcionar, mas voce terá que fazer mais testes do que isso

Shakira: Eu sei, eu sei, aqui estão algumas outras sequências que eu descobri. [Mostra a Jamie as seguintes sequências.]

#2: habilitar•testar*[ler]^n•testar•desabilitar

#3: [ler]^n

#4: habilitar*desabilitar•[testar | ler]

Jamie: Bem, deixe-me ver se entendi o objetivo dessas sequências.

#1 acontece de maneira trivial, assim como um uso convencional.

#2 repete a operação de leitura n vezes, e esse é um cenário provável.

#3 tenta ler o sensor antes de ele ser habilitado... Isso deve produzir uma mensagem de erro de algum tipo, certo?

#4 habilita e desabilita o sensor e então tenta lê-lo. Isso não é o mesmo que o teste #2?

Shakira: Na verdade, não. Em #4, o sensor foi habilitado. O que #4 realmente testa é se a operação de desabilitar funciona como deveria. Um ler() ou testar() após desabilitar() deverá gerar uma mensagem de erro. Se isso não acontecer, temos um erro na operação desabilitar.

Jamie: Certo. Lembre-se de que os quatro testes têm de ser aplicados a cada tipo de sensor, já que todas as operações podem ser sutilmente diferentes dependendo do tipo de sensor.

Shakira: Não se preocupe. Está planejado.

19.5.2 Teste de partição em nível de classe

Que opções de teste estão disponíveis em nível de classe?

O *teste de partição* reduz o número de casos de teste necessários para simular a classe de maneira muito semelhante ao particionamento de equivalência (Capítulo 18) para software tradicional. As entradas e saídas são classificadas e os casos de teste projetados para exercitar cada categoria. Mas como são criadas as categorias de particionamento?

O *particionamento baseado em estado* classifica as operações de classe com base na habilidade delas para mudar o estado da classe. Considerando novamente a classe **Conta**, as operações de estado incluem *depósito()* e *retirada()*, enquanto as operações de não estado incluem *saldo()*, *resumo()*, e *limiteDeCredito()*. Os testes são projetados de forma que simulam operações que mudam e que não mudam o estado, separadamente. Portanto,

Caso de teste p_1: **abrir•estabelecer•depósitar• depositar•retirar• retirar•fechar**
Caso de teste p_2: **abrir•estabelecer•depositar•resumir•limiteDeCrédito•retirar•fechar**

O caso de teste p_1 muda o estado, enquanto o caso de teste p_2 simula operações que não mudam o estado (exceto aquelas na sequência de teste mínima).

Particionamento baseado em atributo classifica operações de classe com base nos atributos que elas usam. Para a classe **Conta**, os atributos **balance** e **limiteDeCrédito** podem ser usados para definir partições. As operações são divididas em três partições: (1) operações que usam **limiteDeCrédito,** (2) operações que modificam **limiteDeCrédito**, e (3) operações que não usam nem modificam **limiteDeCrédito.** São então projetadas sequências de teste para cada partição.

Particionamento baseado em categoria classifica operações de classe com base na função genérica que cada uma executa. Por exemplo, operações na classe **Conta** podem ser classificadas em operações de inicialização (*abrir, estabelecer*), operações computacionais (*depositar, retirar*), consultas (*obterSaldo, resumir, limiteDeCrédito*), e operações de terminação (*fechar*).

19.6 PROJETO DE CASO DE TESTE INTERCLASSE

"A fronteira que define o escopo do teste de unidade e teste de integração é diferente para o desenvolvimento orientado a objeto. Podem ser projetados e usados testes em muitos pontos no processo. Assim, 'projetar pequeno, codificar pequeno' torna-se 'projetar pequeno, codificar pequeno, testar pequeno'."

Robert Binder

O projeto de caso de teste torna-se mais complicado quando começa a integração do sistema orientado a objeto. É nesse estágio que o teste de colaborações entre classes deve começar. Para ilustrarmos a "geração de caso de teste interclasse" [Kir94], expandiremos o exemplo bancário introduzido na Seção 19.5 para incluir as classes e colaborações da Figura 19.2. As direções das setas na figura indicam a direção das mensagens, e os rótulos, as operações chamadas como consequência das colaborações sugeridas pelas mensagens.

Como o teste de classes individuais, o teste de colaboração entre classes pode ser feito aplicando-se métodos aleatórios e de particionamento, bem como teste baseado em cenário e teste comportamental.

19.6.1 Teste de múltiplas classes

Kirani e Tsai [Kir94] sugerem a seguinte sequência de passos para gerar casos de teste aleatórios de múltiplas classes:

1. Para cada classe cliente, use a lista de operações de classe para gerar uma série de sequências aleatórias de teste. As operações enviarão mensagens para outras classes servidoras.
2. Para cada mensagem gerada, determine a classe colaboradora e a operação correspondente no objeto servidor.
3. Para cada operação no objeto servidor (que foi chamado por mensagens enviadas pelo objeto cliente), determine as mensagens que ele transmite.
4. Para cada uma das mensagens, determine o próximo nível de operações chamadas e incorpore-as na sequência de teste.

Para ilustrar [Kir94], considere uma sequência de operações para a classe **Banco** relativas a uma classe **ATM** (Figura 19.2):

verificarConta • verificarPIN • [[verificarDiretrizes • requisiçãoDeRetirada] | requisiçãoDeDepósito | requisiçãoDeInformaçãoDeConta]^n

Um caso de teste aleatório para a classe **Banco** poderia ser

Caso de teste r_3 = **verificarConta • verificarPIN • requisiçãoDeDepósito**

Para as colaborações envolvidas nesse teste, são consideradas as mensagens associadas a cada uma das operações citadas no caso de teste r_3. **Banco** deve colaborar com **InformaçãoDeValidação** para executar a **verificarConta()** e **verificarPIN()**. **Banco** deve colaborar com **Conta** para executar **requisiçãoDeDepósito()**. Daí, um novo caso de teste que exercita essas colaborações é

Caso de teste r_4 = **verificarConta [Banco:contaVálidaInformaçãoDeValidação] • verificarPIN [Banco: PINVálidoInformaçãoDeValidação] • requisiçãoDeDepósito [Banco: depósitoConta]**

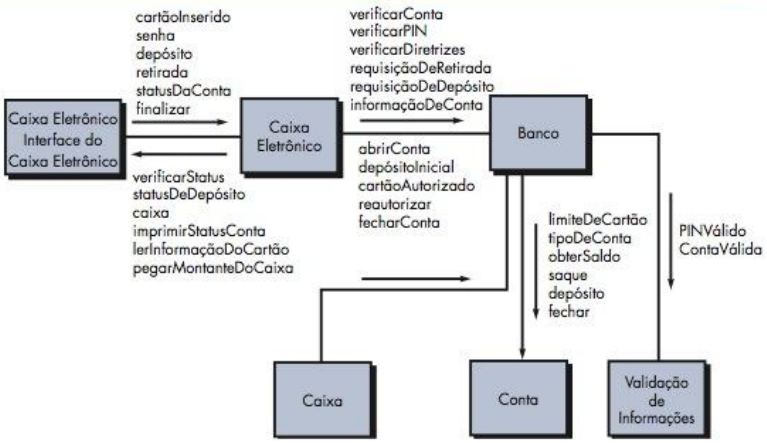

FIGURA 19.2
Diagrama de colaboração de classe para a aplicação bancária

***Fonte:* Adaptado de (Kir94)**

A abordagem para teste de partição de múltiplas classes é similar à usada para teste de partição de classes individuais. Uma classe simples é particionada conforme discutido na Seção 19.5.2. No entanto, a equivalência de teste é expandida para incluir operações chamadas via mensagens a classes colaboradoras. Uma abordagem alternativa particiona os testes com base nas interfaces para uma classe em particular. Conforme a Figura 19.2, a classe **Banco** recebe mensagens das classes **ATM** e **Caixa**. Os métodos da classe **Banco** podem, portanto, ser testados particionando-os naqueles que servem **ATM** e naqueles que servem **Caixa.** Pode ser usada a partição baseada em estado (Seção 19.5.2) para refinar ainda mais as partições.

19.6.2 Testes derivados de modelos comportamentais

O uso dos diagramas de estado como um modelo que representa o comportamento dinâmico de uma classe é discutido no Capítulo 7. O diagrama de estado para uma classe pode ser usado para ajudar a derivar uma sequência de testes que irão simular o comportamento dinâmico da classe (e as classes que colaboram com ela). A Figura 19.3 [Kir94] ilustra um diagrama de estado para a classe **Conta** discutida anteriormente. Observando a figura, as transações iniciais movem-se para os estados *conta vazia* e *conta estabelecida*. A maior parte do comportamento para instâncias da classe ocorre quando ainda no estado *conta ativa*. Uma retirada final e fechamento da conta fazem a classe conta transitar para os estados *conta inativa* e *conta "morta"*, respectivamente.

Os testes a ser projetados deverão conseguir a cobertura de todos os estados. Isto é, as sequências de operação deverão fazer a classe **Conta** realizar transição através de todos os estados permitidos:

Caso de teste s_1: **abrir • estabelecerConta • fazerDepósito (inicial) • fazerRetirada (final) • fechar**

Deve-se notar que essa sequência é idêntica à sequência de teste mínimo discutida na Seção 19.5.2. Acrescentando sequências de teste adicionais à sequência mínima,

Caso de teste s_2: **abrir • estabelecerConta • fazerDepósito (inicial) • fazerDepósito • obterSaldo • obterLimiteDeCrédito • fazerRetirada (final) • fechar**

Caso de teste s_3: **abrir • estabelecerConta • fazerDepósito(inicial) • fazerDepósito • fazerRetirada • informaçãoDaConta • fazerRetirada (final) • fechar**

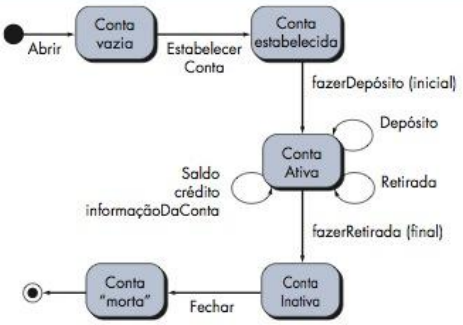

FIGURA 19.3
Diagrama de estado para a classe Conta
Fonte: **Adaptado de (Kir94)**

Mais casos de teste poderiam ainda ser criados para garantir que todos os comportamentos para a classe fossem adequadamente simulados. Em situações nas quais o comportamento da classe resulta em uma colaboração com uma ou mais classes, são usados diagramas de estados múltiplos para acompanhar o fluxo comportamental do sistema.

O modelo de estado pode ser percorrido de uma maneira primeiro-em-largura [McG94]. Nesse contexto, primeiro-em-largura implica que um caso de teste simula uma única transição e que, quando uma nova transição deve ser testada, somente as transições testadas anteriormente são usadas.

Considere um objeto **CartãoDeCrédito** que faz parte do sistema bancário. O estado inicial de **CartãoDeCrédito** é *indefinido* (não foi fornecido nenhum número de cartão de crédito). Após ler o cartão de crédito durante uma venda, o objeto assume um estado *definido*; isto é, os atributos **card number** e **expiration date,** juntamente com identificadores específicos do banco são definidos. O cartão de crédito é *submetido* (enviado) quando ele é enviado para autorização, e é aprovado quando a autorização é recebida. A transição de **CartãoDeCrédito** de um estado para outro pode ser testada derivando casos de teste que fazem a transição ocorrer. Uma abordagem primeiro-em-largura para esse tipo de teste não simularia *submetido* antes de simular *indefinido* e *definido*. Se o fizesse, faria uso das transições que não foram testadas previamente e, portanto, violaria o critério primeiro-em-largura.

19.7 RESUMO

O objetivo geral do teste orientado a objeto — encontrar o número máximo de erros com um mínimo de esforço — é idêntico ao objetivo do teste de software convencional. Mas as estratégias e as táticas para o teste orientado a objeto diferem significativamente. A visão do teste se amplia para incluir a revisão tanto dos requisitos quanto do modelo de projeto. Além disso, o foco do teste afasta-se do componente procedural (o módulo) e se aproxima da classe.

Em virtude dos requisitos e modelos de projeto orientados a objeto e o código fonte resultante serem semanticamente acoplados, o teste (na forma de revisões técnicas) começa durante a atividade de modelagem. Por essa razão, a revisão do CRC, da relação de objeto e dos modelos de comportamento pode ser vista como um teste de primeiro estágio.

Uma vez disponível o código, é aplicado o teste de unidade para cada classe. O projeto de testes para uma classe usa uma variedade de métodos: teste baseado em falha, teste aleatório e teste de partição. Cada um desses métodos simula as operações encapsuladas pela classe. Sequências de teste são projetadas para garantir que sejam exercitadas as operações relevantes. O estado da classe, representado pelos valores de seus atributos, é examinado para determinar se existem erros.

Teste de integração pode ser feito usando uma estratégia baseada em sequência de execução (*thread*) ou baseado em uso. O teste baseado em sequência de execução integra o conjunto de classes que colaboram para responder a uma entrada ou evento. O teste baseado em uso constrói o sistema em camadas, começando com aquelas classes que não fazem uso das classes servidoras. Métodos de projeto de caso de teste de integração podem também fazer uso de testes aleatórios e de partição. Além disso, testes baseados em cenário e derivados de modelos comportamentais podem ser usados para testar uma classe e seus colaboradores. Uma sequência de teste acompanha o fluxo de operações por meio das colaborações de classe.

Teste de validação orientado a objeto é teste orientado à caixa-preta e pode ser realizado aplicando-se os mesmos métodos caixa-preta discutidos para software convencional. No entanto, o teste baseado em cenário domina a validação de sistemas orientados a objeto, tornando o caso de uso um pseudocontrolador primário para teste de validação.

Problemas e Pontos a Ponderar

19.1. Com suas palavras, descreva por que a classe é a menor unidade razoável para teste em um sistema orientado a objeto.

19.2. Por que temos de retestar subclasses instanciadas de uma classe existente, se a classe existente já foi completamente testada? Podemos usar o projeto de caso de teste para a classe existente?

19.3. Por que o "teste" deveria começar com análise e projeto orientado a objeto?

19.4. Crie uma série de cartões de índice CRC para o *CasaSegura* e execute os passos descritos na Seção 19.2.2 para determinar se existem inconsistências.

19.5. Qual a diferença entre estratégias baseadas em sequências de execução (*thread*) e estratégias baseadas em uso para teste de integração? Como o teste de conjunto se encaixa?

19.6. Aplique teste aleatório e teste de particionamento a três classes definidas no projeto para o sistema *CasaSegura*. Produza casos de teste que indiquem as sequências de operação que serão chamadas.

19.7. Aplique teste de múltiplas classes e testes derivados de modelo comportamental para o projeto *CasaSegura*.

19.8. Crie quatro testes adicionais usando teste aleatório e métodos de particionamento, bem como teste de múltiplas classes e testes derivados do modelo comportamental para a aplicação bancária apresentada nas Seções 19.5 e 19.6.

Leituras e Fontes de Informação Complementares

Muitos livros sobre testes citados nas seções Leituras e Fontes de Informação Complementares dos Capítulos 17 e 18 discutem o teste de sistemas orientados a objeto até certo ponto. Schach (*Object-Oriented and Classical Software Engineering*, McGraw-Hill, 6th ed., 2004) considera o teste orientado a objeto dentro do contexto mais amplo da prática da engenharia de software. Sykes and McGregor (*Practical Guide to Testing Object-Oriented Software*, Addison-Wesley, 2001), Bashir and Goel (*Testing Object-Oriented Software*, Springer 2000), Binder (*Testing Object-Oriented Systems*, Addison-Wesley, 1999) e Kung e seus colegas (*Testing Object-Oriented Software*, Wiley-IEEE Computer Society Press, 1998) tratam o teste orientado a objeto em detalhes significativos.

Uma ampla gama de fontes de informação sobre métodos de teste orientado a objeto está disponível na Internet. Uma lista atualizada de referências na Web, relevante para as técnicas de teste, pode ser encontrada no site **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**.