19

Teste de *software* – Nível de componentes

Conceitos-chave

teste do caminho básico
teste caixa-preta
análise de valor limite
teste de conjunto
teste de estrutura de controle
complexidade ciclomática
depuração
particionamento de equivalência
grupo independente de teste
teste de integração
teste de interface
teste orientado a objetos
scaffolding
teste de sistema
métodos de teste
estratégias de teste
teste de unidade
teste de validação
verificação
teste caixa-branca

O teste de componentes de *software* incorpora uma estratégia que descreve os passos a serem executados como parte do teste, define quando esses passos são planejados e então executados e quanto trabalho, tempo e recursos serão necessários. Dentro dessa estratégia, o teste de componentes de *software* implementa um conjunto de táticas de teste de componentes que envolve planejamento dos testes, projeto de casos de teste, execução dos testes e coleta e avaliação dos dados resultantes. Este capítulo considera ambas as estratégias e as táticas de teste de componentes.

Panorama

O que é? O *software* é testado para revelar erros cometidos inadvertidamente quando ele foi projetado e construído. Uma estratégia de teste de componentes de *software* considera o teste de componentes individuais e a sua integração a um sistema em funcionamento.

Quem realiza? Uma estratégia de teste de componentes de *software* é desenvolvida pelo gerente de projeto, pelos engenheiros de *software* e pelos especialistas em testes.

Por que é importante? O teste muitas vezes exige mais trabalho de projeto do que qualquer outra ação da engenharia de *software*. Se for feito casualmente, perde-se tempo, fazem-se esforços desnecessários, e, ainda pior, erros passam sem ser detectados.

Quais são as etapas envolvidas? O teste começa pelo "pequeno" e passa para o "grande". Ou seja, os testes

iniciais focam em um único componente ou em um pequeno grupo de componentes relacionados e descobrem erros nos dados e na lógica de processamento que foram encapsulados pelo(s) componente(s). Depois de testados, os componentes devem ser integrados até que o sistema completo esteja pronto.

Qual é o artefato? A especificação do teste documenta a abordagem da equipe de *software* para o teste, definindo um plano que descreve uma estratégia global e um procedimento e designando etapas específicas de teste e os tipos de casos de teste que serão feitos.

Como garantir que o trabalho foi realizado corretamente? Um plano e um procedimento de teste eficazes levarão a uma construção ordenada do *software* e à descoberta de erros em cada estágio do processo de construção.

Para ser eficaz, uma estratégia de teste de componentes deve ser flexível o bastante para promover uma estratégia de teste personalizada, mas rígida o bastante para estimular um planejamento razoável e o acompanhamento à medida que o projeto progride. O teste de componentes ainda é responsabilidade dos engenheiros de *software* individuais. Quem realiza o teste, de que forma os engenheiros comunicam os seus resultados uns para os outros e quando os testes são realizados são fatores determinados pela abordagem de integração de *software* e pela filosofia de projeto adotadas pela equipe de desenvolvimento.

Essas "abordagens e filosofias" são chamadas de *estratégia e tática*, assuntos que serão apresentados neste capítulo. No Capítulo 20, discutimos as técnicas de teste de integração que acabam definindo a estratégia de desenvolvimento da equipe.

19.1 Uma abordagem estratégica do teste de *software*

Teste é um conjunto de atividades que podem ser planejadas com antecedência e executadas sistematicamente. Por essa razão, deverá ser definido, para o processo de *software*, um modelo para o teste – um conjunto de etapas no qual podem ser empregadas técnicas específicas de projeto de caso de teste e métodos de teste.

Muitas estratégias de teste de *software* já foram propostas na literatura [Jan16] [Dak14] [Gut15]. Todas elas fornecem um modelo para o teste e todas têm as seguintes características genéricas:

- Para executar um teste eficaz, faça revisões técnicas eficazes (Capítulo 16). Fazendo isso, muitos erros serão eliminados antes do começo do teste.
- O teste começa no nível de componente e progride em direção à integração do sistema computacional como um todo.
- Diferentes técnicas de teste são apropriadas para diferentes abordagens de engenharia de *software* e em diferentes pontos no tempo.
- O teste é realizado pelo desenvolvedor do *software* e (para grandes projetos) por um grupo de teste independente.
- O teste e a depuração são atividades diferentes, mas a depuração deve ser associada a alguma estratégia de teste.

Uma estratégia de teste de *software* deve acomodar testes de baixo nível, necessários para verificar se um pequeno segmento de código fonte foi implementado corretamente, bem como testes de alto nível, que validam as funções principais do sistema de acordo com os requisitos do cliente. Uma estratégia deve fornecer diretrizes para o profissional e uma série de metas para o gerente. Como os passos da estratégia de teste ocorrem no instante em que as pressões pelo prazo começam a aumentar, deve ser possível medir o progresso no desenvolvimento, e os problemas devem ser revelados o mais cedo possível.

19.1.1 Verificação e validação

O teste de *software* é um elemento de um tema mais amplo, muitas vezes conhecido como verificação e validação (V&V). *Verificação* refere-se ao conjunto de tarefas que garantem que o *software* implemente corretamente uma função específica. *Validação* refere-se ao conjunto de tarefas que asseguram que o *software* foi criado e pode ser rastreado segundo os requisitos do cliente. Boehm [Boe81] define de outra maneira:

> Verificação: "Estamos criando o produto corretamente?"
>
> Validação: "Estamos criando o produto certo?"

A definição de V&V abrange muitas atividades de garantia da qualidade do *software* (Capítulo 19).[1]

A verificação e a validação incluem uma ampla gama de atividades de garantia de qualidade de *software* (SQA, do inglês *software quality assurance*): revisões técnicas, auditorias de qualidade e configuração, monitoramento de desempenho, simulação, estudo de viabilidade, revisão de documentação, revisão de base de dados, análise de algoritmo, teste de desenvolvimento, teste de usabilidade, teste de qualificação, teste de aceitação e teste de instalação. Embora a aplicação de teste tenha um papel extremamente importante em V&V, muitas outras atividades também são necessárias.

O teste proporciona o último elemento a partir do qual a qualidade pode ser estimada e, mais pragmaticamente, os erros podem ser descobertos. Mas o teste não deve ser visto como uma rede de segurança. Como se costuma dizer, "Você não pode testar qualidade. Se a qualidade não está lá antes de um teste, ela não estará lá quando o teste terminar". A qualidade é incorporada ao *software* por meio do processo de engenharia de *software*, e os testes não podem ser simplesmente aplicados no final do processo para consertar tudo. A aplicação correta de métodos e ferramentas, de

revisões técnicas eficazes e de um sólido gerenciamento e avaliação conduzem todos à qualidade que é confirmada durante o teste.

19.1.2 Organizando o teste de *software*

Para todo projeto de *software*, há um conflito de interesses inerente que ocorre logo que o teste começa. As pessoas que criaram o *software* são agora convocadas para testá-lo. Isso parece essencialmente inofensivo; afinal, quem conhece melhor o programa do que os seus próprios desenvolvedores? Infelizmente, esses mesmos desenvolvedores têm interesse em demonstrar que o programa é isento de erros e funciona de acordo com os requisitos do cliente – e que será concluído dentro do prazo e do orçamento previstos. Cada um desses interesses vai contra o teste completo.

Do ponto de vista psicológico, a análise e o projeto de *software* (juntamente com a sua codificação) são tarefas construtivas. O engenheiro de *software* analisa, modela e então cria um programa de computador e sua documentação. Como qualquer outro construtor, o engenheiro de *software* tem orgulho do edifício que construiu e encara com desconfiança qualquer um que tente estragar sua obra. Quando o teste começa, há uma tentativa sutil, embora definida, de "quebrar" aquela coisa que o engenheiro de *software* construiu. Do ponto de vista do construtor, o teste pode ser considerado (psicologicamente) destrutivo. Assim, o construtor vai, calmamente, projetando e executando testes que demonstram que o programa funciona, em vez de descobrir os erros. Infelizmente, no entanto, os erros estão presentes. E, se o engenheiro de *software* não os encontrar, o cliente encontrará!

Frequentemente, há muitas noções incorretas que podem ser inferidas a partir da discussão apresentada: (1) que o desenvolvedor de *software* não deve fazer nenhum teste; (2) que o *software* deve ser "atirado aos leões", ou seja, entregue a estranhos que realizarão testes implacáveis; e (3) que os testadores se envolvem no projeto somente no início das etapas do teste. Todas essas declarações são incorretas.

O desenvolvedor do *software* é sempre responsável pelo teste das unidades individuais (componentes) do programa, garantindo que cada uma execute a função ou apresente o comportamento para o qual foi projetada. Em muitos casos, o desenvolvedor também faz o *teste de integração* – uma etapa de teste que leva à construção (e ao teste) da arquitetura completa do *software*. Somente depois que a arquitetura do *software* está concluída é que o grupo de teste independente se envolve.

O papel de um *grupo independente de teste* (ITG, do inglês *independent test group)* é remover problemas inerentes associados ao fato de deixar o criador testar aquilo que ele mesmo criou. O teste independente remove o conflito de interesses que, de outra forma, poderia estar presente. Afinal, o pessoal do ITG é pago para encontrar erros.

No entanto, você não entrega simplesmente o programa para o pessoal do ITG e vai embora. O desenvolvedor e o pessoal do ITG trabalham juntos durante todo o projeto de *software* para garantir que testes completos sejam realizados. Enquanto o teste está sendo realizado, o desenvolvedor deve estar disponível para corrigir os erros encontrados.

O ITG faz parte da equipe de desenvolvimento de *software*, pois se envolve durante a análise e o projeto e permanece envolvido (planejando e especificando procedimentos de teste) durante o projeto inteiro. No entanto, em muitos casos, o ITG se reporta à organização de garantia de qualidade do *software*, adquirindo, assim, um grau de independência que poderia não ser possível se fizesse parte da equipe de engenharia de *software*.

19.1.3 Visão global

O processo de *software* pode ser visto como a espiral ilustrada na Figura 19.1. Inicialmente, a engenharia de sistemas define o papel do *software* e passa à análise dos requisitos de *software*, na qual são estabelecidos o domínio da informação, a função, o comportamento, o desempenho, as restrições e os critérios de

validação para o *software*. Deslocando-se para o interior da espiral, chega-se ao projeto e, por fim, à codificação. Para desenvolver *software* de computador, percorre-se a espiral para o interior ao longo de linhas que indicam a diminuição do nível de abstração a cada volta.

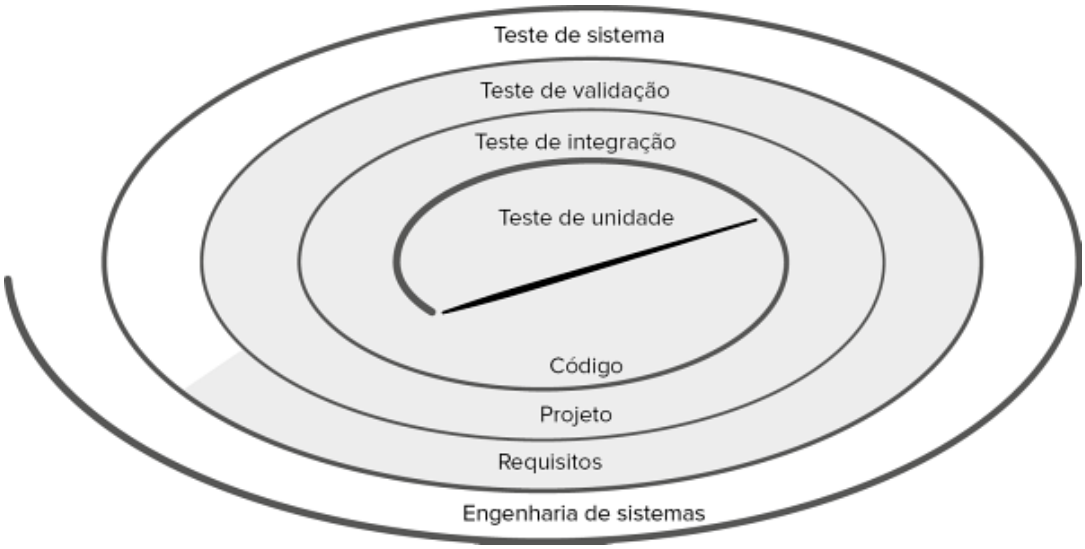

Figura 19.1
Estratégia de teste.

Uma estratégia para teste de *software* pode também ser vista no conceito da espiral (Figura 19.1). O *teste de unidade* começa no centro da espiral e se concentra em cada unidade (p. ex., componente, classe ou objeto de conteúdo de WebApp) do *software*, conforme implementado no código-fonte. O teste prossegue movendo-se em direção ao exterior da espiral, passando pelo *teste de integração*, em que o foco está no projeto e na construção da arquitetura de *software*. Continuando na mesma direção da espiral, encontramos o *teste de validação*, em que requisitos estabelecidos como parte dos requisitos de modelagem são validados em relação ao *software* criado. Por fim, chegamos ao *teste do sistema*, no qual o *software* e outros elementos são testados como um todo. Para testar

um *software* de computador, percorre-se a espiral em direção ao seu exterior, ao longo de linhas que indicam o escopo do teste a cada volta.

Considerando o processo de um ponto de vista procedimental, o teste dentro do contexto de engenharia de *software* é, na realidade, uma série de quatro etapas implementadas sequencialmente. As etapas estão ilustradas na Figura 19.2. Inicialmente, os testes focalizam cada componente individualmente, garantindo que ele funcione adequadamente como uma unidade. Daí o nome *teste de unidade*. O teste de unidade usa intensamente técnicas de teste, com caminhos específicos na estrutura de controle de um componente para garantir a cobertura completa e a máxima detecção de erros. Em seguida, o componente deve ser montado ou integrado para formar o pacote de *software* completo. O teste de integração cuida de problemas associados a aspectos duais de verificação e construção de programa. Técnicas de projeto de casos de teste que focalizam entradas e saídas são mais predominantes durante a integração, embora técnicas que usam caminhos específicos de programa possam ser utilizadas para segurança dos principais caminhos de controle. Depois que o *software* foi integrado (construído), é executada uma série de *testes de ordem superior*. Os critérios de validação (estabelecidos durante a análise de requisitos) devem ser avaliados. O teste de validação proporciona a garantia final de que o *software* satisfaz a todos os requisitos funcionais, comportamentais e de desempenho.

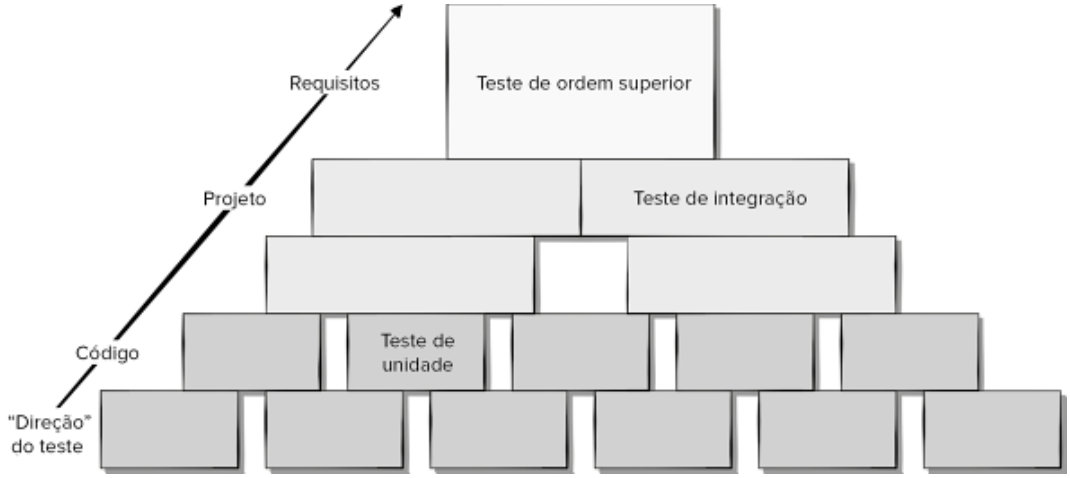

Figura 19.2
Etapas de teste de *software*.

A última etapa de teste de ordem superior extrapola os limites da engenharia de *software*, entrando em um contexto mais amplo de engenharia de sistemas de computadores (discutida no Capítulo 21). O *software*, uma vez validado, deve ser combinado com outros elementos do sistema (p. ex., *hardware*, pessoas, base de dados). O teste de sistema verifica se todos os elementos se combinam corretamente e se a função e o desempenho globais do sistema são obtidos.

Casa segura

Preparando-se para o teste

Cena: No escritório de Doug Miller, quando o projeto no nível de componente está em andamento e começa a construção de certos componentes.

Atores: Doug Miller, gerente de engenharia de *software*; Vinod, Jamie, Ed e Shakira – membros da equipe de engenharia de *software* do *CasaSegura*.

Conversa:

Doug: Parece-me que não dedicamos tempo suficiente para falar sobre o teste.

Vinod: É verdade, mas nós estávamos todos um tanto ocupados. Além disso, estivemos pensando sobre isso... Na verdade, mais do que pensando.

Doug (sorrindo): Eu sei... todos nós estamos sobrecarregados, mas ainda temos de pensar adiante.

Shakira: Eu gosto da ideia de projetar testes de unidades antes de começar a codificar qualquer um de meus componentes – e é o que estou tentando fazer. Tenho um arquivo grande de testes a serem executados logo que o código dos meus componentes estiver completo.

Doug: Esse é o conceito de Extreme Programming (um processo ágil de desenvolvimento de *software*; veja o Capítulo 3), não é mesmo?

Ed: É. Apesar de não estarmos usando Extreme Programming diretamente, decidimos que seria uma boa ideia projetar testes unitários antes de criar o componente – o projeto nos dá todas as informações de que precisamos.

Jamie: Eu já fiz a mesma coisa.

Vinod: E eu assumi o papel de integrador, de forma que, todas as vezes que um dos rapazes passar um componente para mim, vou integrá-lo e executar uma série de testes de regressão (ver Seção 20.3 para uma discussão sobre testes de regressão) no programa parcialmente integrado. Estive trabalhando para projetar uma série de testes apropriados para cada função do sistema.

Doug (para Vinod): Com que frequência você fará os testes?

Vinod: Todos os dias... até que o sistema esteja integrado... Bem, quero dizer, até que o incremento de *software* que pretendemos fornecer esteja integrado.

Doug: Vocês estão mais adiantados do que eu!

Vinod (rindo): Antecipação é tudo no negócio de *software*, chefe.

19.1.4 Critérios de "Pronto"

Uma questão clássica surge todas as vezes que se discute teste de *software*: "Quando podemos dizer que terminamos os testes – como podemos saber que já testamos o suficiente?". Infelizmente, não há uma resposta definitiva para essa pergunta, mas há algumas respostas pragmáticas e algumas tentativas iniciais e empíricas.

Uma resposta é: "O teste nunca termina; o encargo simplesmente passa do engenheiro de *software* para o usuário". Todas as vezes que o usuário executa o programa no computador, o programa está sendo testado. Esse fato destaca a importância de outras atividades de garantia da qualidade do *software*. Outra resposta (um tanto cínica, mas, ainda assim, exata) é: "O teste acaba quando o tempo ou o dinheiro acabam".

Embora alguns profissionais possam argumentar a respeito dessas respostas, o fato é que é necessário um critério mais rigoroso para determinar quando já foram executados testes em número suficiente. A abordagem de *estatística da garantia da qualidade* (Seção 17.6) sugere técnicas de uso estatísticas [Rya11] que executam uma série de testes derivados de uma amostragem estatística de todas as execuções possíveis do programa por todos os usuários em uma população escolhida. Coletando métricas durante o teste do *software* e utilizando modelos estatísticos existentes, é possível

desenvolver diretrizes significativas para responder à questão: "Quando terminamos o teste?".

19.2 Planejamento e manutenção de registros

Muitas estratégias podem ser utilizadas para testar um *software*. Em um dos extremos, pode-se aguardar até que o sistema esteja totalmente construído e então realizar os testes no sistema completo na esperança de encontrar erros. Essa abordagem, embora atraente, simplesmente não funciona. Ela resultará em um *software* cheio de erros que decepcionará todos os envolvidos. No outro extremo, você pode executar testes diariamente, sempre que uma parte do sistema for construída.

Uma estratégia de teste escolhida por muitas equipes de *software* (e que recomendamos) está entre os dois extremos. Ela assume uma visão incremental do teste, começando com o teste das unidades individuais de programa, passando para os testes destinados a facilitar a integração de unidades (às vezes diariamente) e culminando com testes que usam o sistema concluído à medida que evolui. O restante deste capítulo se concentrará no teste no nível de componentes e no projeto de casos de teste.

O teste de unidade focaliza o esforço de verificação na menor unidade de projeto do *software* – o componente ou módulo de *software*. Usando como guia a descrição de projeto no nível de componente, caminhos de controle importantes são testados para descobrir erros dentro dos limites do módulo. A complexidade relativa dos testes e os erros que os testes revelam são limitados pelo escopo restrito estabelecido para o teste de unidade. Esse teste enfoca a lógica interna de processamento e as estruturas de dados dentro dos limites de um componente. Esse tipo de teste pode ser conduzido em paralelo para diversos componentes.

Até a mesmo a melhor estratégia fracassará se não for resolvida uma série de problemas e obstáculos. Tom Gilb [Gil95] argumenta que uma estratégia de teste de *software* terá sucesso somente quando os testadores de *software*: (1) especificarem os requisitos do

produto de uma maneira quantificável muito antes de começar o teste; (2) definirem explicitamente os objetivos do teste; (3) entenderem os usuários do *software* e desenvolverem um perfil para cada categoria de usuário; (4) desenvolverem um plano de teste que enfatize o "teste do ciclo rápido";[2] (5) criarem *software* "robusto" que seja projetado para testar-se a si próprio (o conceito de antidefeito [*antibugging*] é discutido na Seção 9.3); (6) usarem revisões técnicas eficazes como filtro antes do teste; (7) realizarem revisões técnicas para avaliar a estratégia de teste e os próprios casos de teste; e (8) desenvolverem abordagem de melhoria contínua (Capítulo 28) para o processo de teste.

Esses princípios também se refletem no teste de *software* ágil. No desenvolvimento ágil, o plano de teste deve ser estabelecido antes da primeira reunião do *sprint* e deve ser revisado pelos envolvidos. O plano apenas descreve um cronograma aproximado, padrões e ferramentas a serem usados. Os casos de teste e as orientações para o seu uso são desenvolvidos e revisados pelos envolvidos enquanto o código necessário para implementar cada história de usuário é criado. Os resultados dos testes são compartilhados com todos os membros de equipe assim que isso é viável para permitir mudanças no desenvolvimento do código existente do código futuro. Por esse motivo, muitas equipes escolhem usar documentos *online* para a manutenção de registros dos seus testes.

A manutenção de registros de testes não precisa ser onerosa. Os casos de teste podem ser registrados[NT] em uma planilha do Google Docs que descreve rapidamente o caso de teste, contém um *link* para o requisito sendo testado, inclui o resultado esperado dos dados do caso de teste ou os critérios para o sucesso, permite que os testadores indiquem se o teste resultou em aprovação ou reprovação e registra a data em que o caso de teste foi executado, além de ter espaço para comentários sobre os motivos da reprovação para ajudar na depuração. Formulários *online* desse tipo podem ser consultados, quando necessário, para fins de análise, e são fáceis de resumir durante as reuniões da equipe. A Seção 19.3 discute questões relativas ao projeto de casos de teste.

19.2.1 O papel do *scaffolding*

O teste de componente normalmente é considerado um auxiliar para a etapa de codificação. O projeto dos testes de unidade pode ocorrer antes de a codificação começar ou depois que o código-fonte tiver sido gerado. Um exame das informações de projeto fornece instruções para estabelecer casos de teste que provavelmente revelarão os erros. Cada caso de teste deverá ser acoplado a um conjunto de resultados esperados.

Como um componente não é um programa independente, é necessário algum tipo de *scaffolding* (literalmente, "andaime" ou "estrutura temporária") para criar um *framework* de teste. Muitas vezes, como parte desse *framework*, deve ser desenvolvido um pseudocontrolador (*driver*) e/ou um pseudocontrolado (*stub*) para cada teste de unidade. O ambiente de teste de unidade está ilustrado na Figura 19.3. Em muitas aplicações, um *pseudocontrolador* nada mais é do que um "programa principal" que aceita dados do caso de teste, passa esses dados para o componente (a ser testado) e imprime resultados relevantes. Os *pseudocontrolados* servem para substituir módulos subordinados (chamados pelo) ao 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.

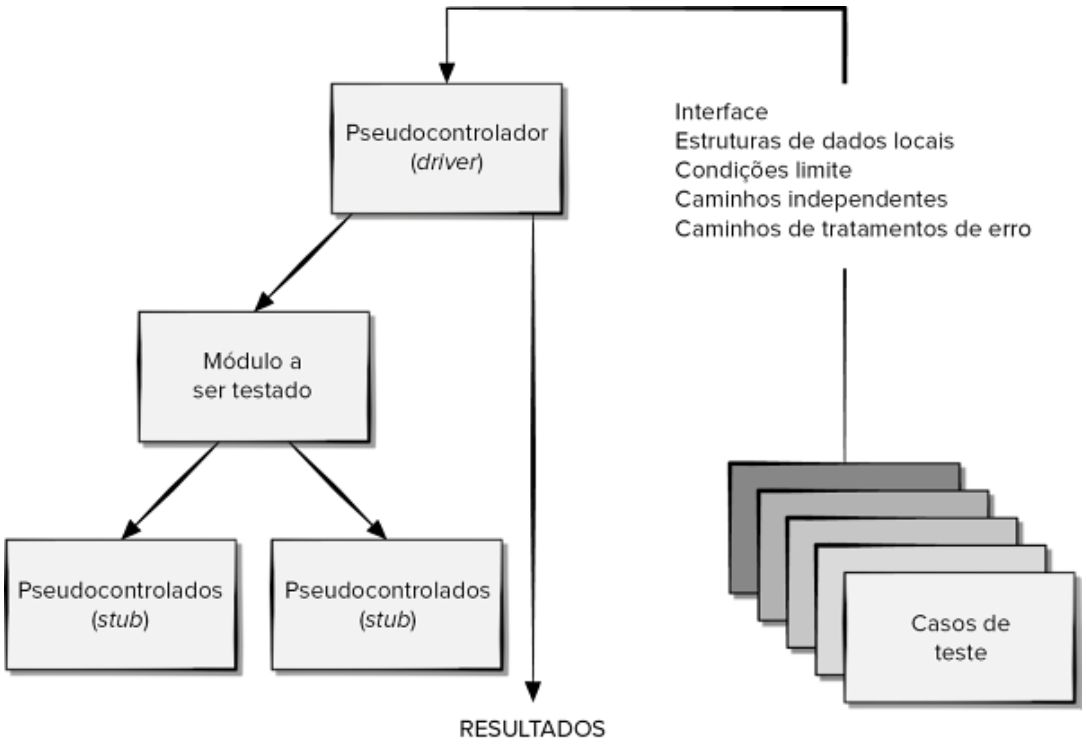

Figura 19.3
Ambiente de teste de unidade.

Pseudocontroladores e pseudocontrolados representam despesas indiretas.[NT] Isto é, ambos são *softwares* que devem ser codificados (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 também são usados).

19.2.2 Eficácia dos custos dos testes

Testes exaustivos exigem que todas as combinações possíveis de valores de entrada e ordens de casos de teste sejam processadas pelo componente sendo testado (p. ex., considere o *gerador de lances* em um jogo de xadrez para computador). Em alguns casos, isso exigiria a criação de um número quase infinito de conjuntos de dados. O retorno sobre testes exaustivos muitas vezes não vale a pena, pois os testes não são suficientes em si para provar que um componente foi implementado corretamente. Há situações em que não temos os recursos necessários para testes de unidades abrangentes. Nesses casos, os testadores devem selecionar módulos cruciais para o sucesso do projeto e aqueles para os quais suspeita-se haver maior probabilidade de erros (devido às suas métricas de complexidade) para que sejam o foco do seu teste de unidades. As Seções 19.4 a 19.6 discutem algumas das técnicas usadas para minimizar o número de casos de teste necessários para se fazer um bom trabalho nesse sentido.

Informações

Teste exaustivo

Considere um programa de 100 linhas em linguagem C. Após algumas declarações básicas de dados, o programa contém dois laços aninhados que executam de 1 a 20 vezes cada um, dependendo das condições especificadas na entrada. Dentro do ciclo, são necessárias 4 construções se-então-senão (if-then-else). Há aproximadamente 1014 caminhos possíveis que podem ser executados nesse programa!

Para colocar esse número sob perspectiva, vamos supor que um processador de teste mágico (“mágico” porque não existe tal processador) tenha sido desenvolvido para teste exaustivo. O processador pode desenvolver um caso de teste, executá-lo e avaliar os resultados em um milissegundo. Trabalhando 24 horas por dia, 365 dias por ano, o processador gastaria 3.170 anos para testar o programa. Isso, sem dúvida, tumultuaria qualquer cronograma de desenvolvimento.

Portanto, pode-se afirmar que o teste exaustivo é impossível para grandes sistemas de *software*.

19.3 Projeto de caso de teste

Não é má ideia projetar casos de teste de unidade antes de desenvolver o código para um componente. Isso assegura que você desenvolverá um código que passará nos testes, ou pelo menos nos testes que já considerou.

Os testes de unidade estão ilustrados esquematicamente na Figura 19.4. A interface do módulo é testada para assegurar que as informações fluam corretamente para dentro e para fora da unidade de programa que está sendo testada (Seção 19.5.1). A estrutura de dados local é examinada para garantir que os dados armazenados temporariamente mantenham sua integridade durante todos os passos na execução de um algoritmo. Todos os caminhos independentes da estrutura de controle são usados para assegurar que todas as instruções em um módulo tenham sido executadas pelo menos uma vez (Seção 19.4.2). As condições limite são testadas para garantir que o módulo opere adequadamente nas fronteiras estabelecidas para limitar ou restringir o processamento (Seção 19.5.3). Por fim, são testados todos os caminhos de manipulação de erro.

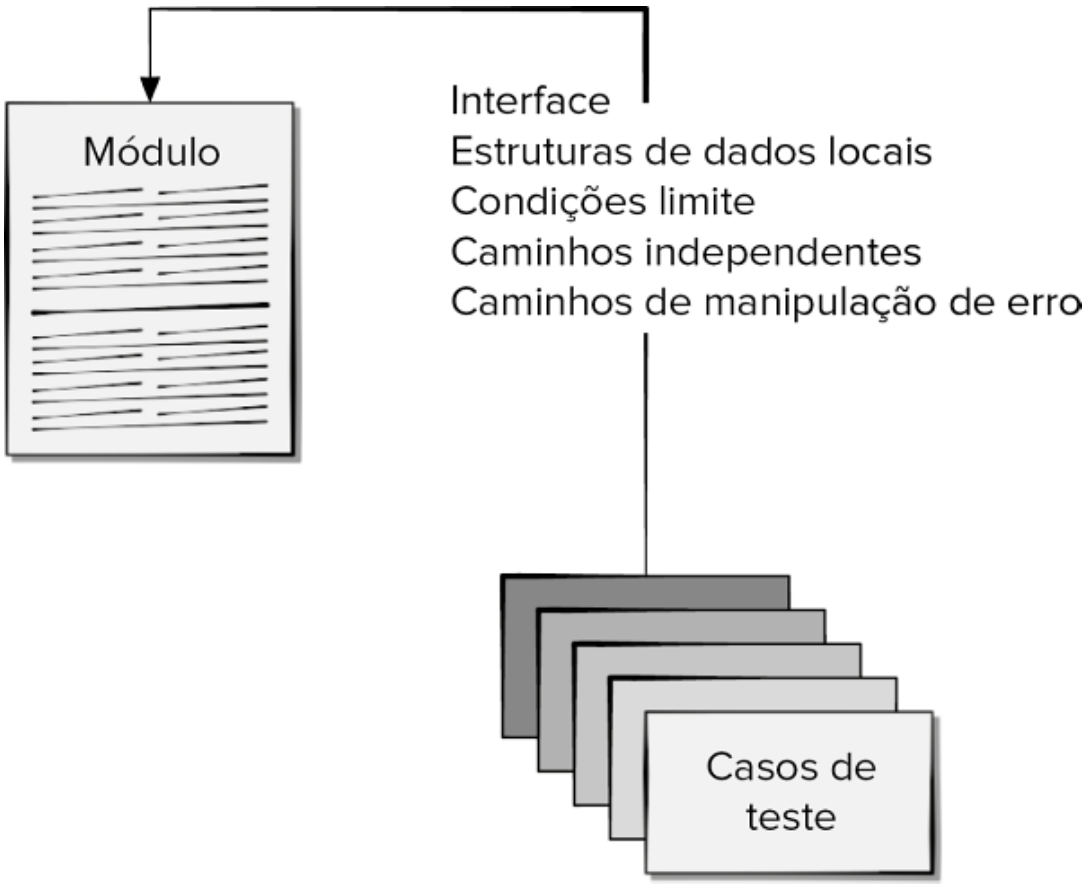

Figura 19.4
Teste de unidade.

O fluxo de dados por meio da interface de um componente é testado antes de iniciar qualquer outro teste. Se os dados não entram e saem corretamente, todos os outros testes são discutíveis. Além disso, estruturas de dados locais deverão ser ensaiadas, e o impacto local sobre dados globais deve ser apurado (se possível) durante o teste de unidade.

O teste seletivo de caminhos de execução é uma tarefa essencial durante o teste de unidade. Casos de teste devem ser projetados para descobrir erros devido a computações errôneas, comparações incorretas ou fluxo de controle inadequado.

O teste de fronteira é uma das tarefas mais importantes do teste de unidade. O *software* frequentemente falha nas suas fronteiras. Isto é, os erros frequentemente ocorrem quando o n -ésimo elemento de um conjunto n -dimensional é processado, quando a i -ésima repetição de um laço com i passadas é chamada, ou quando o valor máximo ou mínimo permitido é encontrado. Casos de teste que utilizam estrutura de dados, fluxo de controle e valores de dados logo abaixo, iguais ou logo acima dos máximos e mínimos têm grande possibilidade de descobrir erros.

Um bom projeto prevê condições de erro e estabelece caminhos de manipulação de erro para redirecionar ou encerrar ordenadamente o processamento quando ocorre um erro. Yourdon [You75] chama essa abordagem de *antidefeitos*. Infelizmente, há uma tendência de incorporar manipulação de erro no *software* e nunca testá-la. Projete testes para executar todos os caminhos de manipulação de erro. Se não fizer isso, o caminho pode falhar quando for solicitado, piorando uma situação já incerta.

Entre os erros em potencial que devem ser testados quando a manipulação de erro é avaliada, estão: (1) descrição confusa do erro; (2) o erro apontado não corresponde ao erro encontrado; (3) a condição do erro causa intervenção do sistema antes da manipulação do erro; (4) o processamento exceção-condição é incorreto; ou (5) a descrição do erro não fornece informações suficientes para ajudar na localização da causa do erro.

Casa segura

Projetando testes únicos

Cena: Sala de Vinod.

Atores: Vinod e Ed – membros da equipe de engenharia de *software* do *CasaSegura*.

Conversa:

Vinod: Então, esses são os casos de teste que você pretende usar para a operação *validaçãoDeSenha*.

Ed: Sim, eles devem abranger muito bem todas as possibilidades para os tipos de senhas que um usuário possa digitar.

Vinod: Então, vamos ver... você notou que a senha correta será 8080, certo?

Ed: Certo.

Vinod: E você especifica as senhas 1234 e 6789 para testar o erro no reconhecimento de senhas inválidas?

Ed: Certo, e testo senhas semelhantes à senha correta, veja... 8081 e 8180.

Vinod: Essas parecem OK, mas eu não vejo muito sentido em testar 1234 e 6789. Elas são redundantes... testam a mesma coisa, não é isso?

Ed: Bem, são valores diferentes.

Vinod: Verdade, mas se 1234 não revela um erro... em outras palavras... a operação *validacaoDeSenha* nota que essa é uma senha inválida, é improvável que 6789 nos mostre algo novo.

Ed: Entendo o que você quer dizer.

Vinod: Não estou tentando ser exigente... É que nós temos um tempo limitado para testar, portanto é uma boa ideia executar testes que tenham alta possibilidade de encontrar erros novos.

Ed: Sem problemas... Vou pensar um pouco mais sobre isso.

19.3.1 Requisitos e casos de uso

Na engenharia de requisitos (Capítulo 7), sugerimos iniciar o processo de coleta de requisitos trabalhando com os clientes para gerar histórias de usuário que os desenvolvedores poderiam refinar e transformar em casos de uso formais e modelos de análise. Esses casos de uso e modelos podem ser usados para orientar a criação sistemática de casos de teste que conseguem testar bem os requisitos funcionais de cada componente de *software* e oferecem um bom nível geral de cobertura do teste [Gut15].

Os artefatos da análise não revelam muito sobre a criação de casos de teste para requisitos não funcionais (p. ex., usabilidade ou confiabilidade). É aqui que os enunciados de aceitação do cliente, incluídos nas histórias de usuário, podem formar a base para a elaboração de casos de teste para os requisitos não funcionais associados com os componentes. Os desenvolvedores de casos de teste utilizam informações adicionais, com base na sua experiência profissional, para quantificar os critérios de aceitação e torná-los testáveis. O teste de requisitos não funcionais pode exigir o uso de métodos de testes de integração (Capítulo 20) ou outras técnicas de teste especializadas (Capítulo 21).

O propósito principal dos testes é ajudar os desenvolvedores a descobrir defeitos antes desconhecidos. Executar casos de teste que demonstram que o componente está rodando corretamente quase nunca é o suficiente. Como mencionamos anteriormente (Seção 19.3), é importante elaborar casos de teste que exercitam as capacidades de manipulação de erros do componente. Mas para descobrir novos defeitos, também é importante produzir casos de teste que testem que o componente não faz algo que não deveria fazer (p. ex., acessar fontes de dados privilegiadas sem as permissões apropriadas). Estes podem ser enunciados formalmente em *antirrequisitos*[3] e podem precisar de técnicas especializadas de teste de segurança (Seção 21.7) [Ale17]. Chamados de *casos de teste*

negativos, estes devem ser incluídos para garantir que os componentes se comportam de acordo com as expectativas do cliente.

19.3.2 Rastreabilidade

Para garantir que o processo de teste pode ser auditado, cada caso de teste precisa ser rastreado de volta aos requisitos ou antirrequisitos funcionais ou não funcionais. Muitas vezes, os requisitos não funcionais precisam remontar a requisitos de negócio ou de arquitetura específicos. Muitos desenvolvedores ágeis resistem ao conceito de rastreabilidade, considerado um ônus desnecessário para os desenvolvedores. Contudo, muitas falhas em processos de teste estão ligadas a ausências de caminhos de rastreabilidade, dados de teste inconsistentes ou cobertura de teste incompleta [Rem14]. Os testes de regressão (discutidos na Seção 20.3) exigem o reteste de componentes selecionados que podem ser afetados por alterações a outros componentes de *software* com os quais colaboram. Esta costuma ser considerada uma questão mais relevante nos testes de integração (Capítulo 20), mas garantir que os casos de teste podem ser ligados aos requisitos é um primeiro passo importante e é algo que precisa ser feito nos testes de componentes.

19.4 Teste caixa-branca

O teste caixa-branca, também chamado de teste da caixa-de-vidro ou teste estrutural, é uma filosofia de projeto de casos de teste que usa a estrutura de controle descrita como parte do projeto no nível de componentes para derivar casos de teste. Usando métodos de teste caixa-branca, o engenheiro de *software* pode criar casos de teste que (1) garantam que todos os caminhos independentes de um módulo foram exercitados pelo menos uma vez, (2) exercitem todas as decisões lógicas nos seus estados verdadeiro e falso, (3) executem todos os ciclos em seus limites e dentro de suas fronteiras

operacionais e (4) exercitem estruturas de dados internas para assegurar a sua validade.

19.4.1 Teste de caminho básico

O teste de caminho básico é uma técnica de teste caixa-branca proposta por Tom McCabe [McC76]. O teste de caminho básico permite ao projetista de casos de teste derivar uma medida da complexidade lógica de um projeto procedimental e usar essa medida como guia para definir um conjunto-base de caminhos de execução. Casos de teste criados para exercitar o conjunto-base executam com certeza todas as instruções de um programa pelo menos uma vez durante o teste.

Antes de apresentarmos o método do caminho básico, deve ser introduzida uma notação simples para a representação do fluxo de controle, chamada de grafo de fluxo (ou grafo de programa).[4] Um grafo de fluxo somente deve ser desenhado quando a estrutura lógica de um componente for complexa. O grafo de fluxo permite seguir mais facilmente os caminhos de um programa.

Para ilustrar o uso de um grafo de fluxo, considere a representação do projeto procedimental da Figura 19.5a. É usado um fluxograma para mostrar a estrutura de controle do programa. A Figura 19.5b mapeia o fluxograma em um grafo de fluxo correspondente (considerando que os losangos de decisão do fluxograma não contêm nenhuma condição composta). Na Figura 19.5b, cada círculo, chamado de *nó do grafo de fluxo*, representa um ou mais comandos procedurais. Uma sequência de retângulos de processamento e um losango de decisão podem ser mapeados em um único nó. As setas no grafo de fluxo, chamadas de *arestas* ou *ligações*, representam fluxo de controle e são análogas às setas do fluxograma. Uma aresta deve terminar em um nó, mesmo que esse nó não represente qualquer comando procedural (p. ex., veja o símbolo do diagrama de fluxo para a construção se-então-senão [*if-then-else*]). As áreas limitadas por arestas e nós são chamadas de

regiões. Ao contarmos as regiões, incluímos a área fora do grafo como uma região.

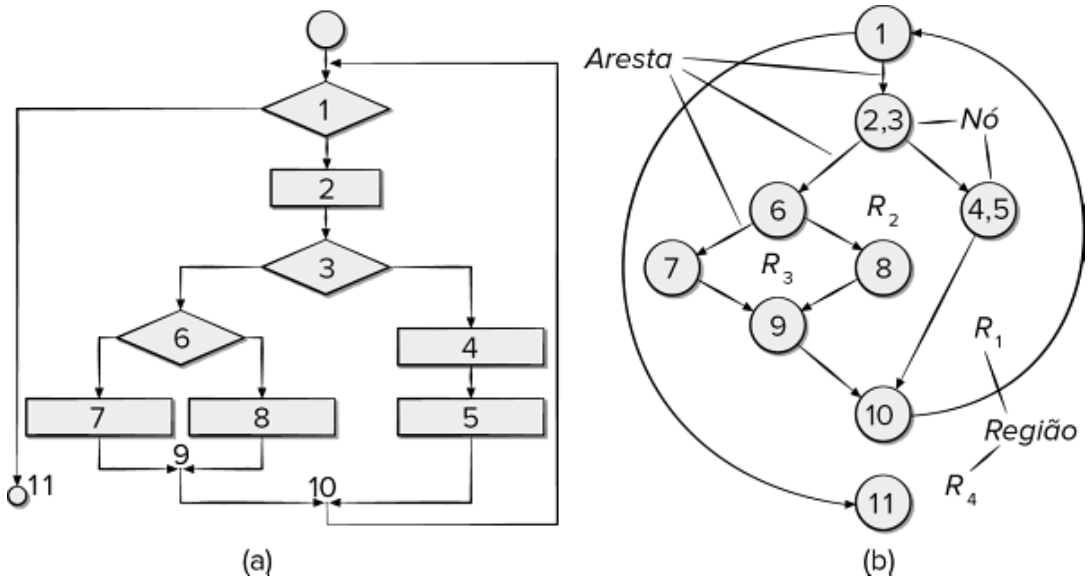

Figura 19.5
(a) Fluxograma e (b) grafo de fluxo.

Um *caminho independente* é qualquer caminho através do programa que introduz pelo menos um novo conjunto de comandos de processamento ou uma nova condição. Quando definido em termos de um grafo de fluxo, um caminho independente deve incluir pelo menos uma aresta que não tenha sido atravessada antes de o caminho ser definido. Por exemplo, um conjunto de caminhos independentes para o grafo de fluxo ilustrado na Figura 19.5b é

Caminho 1: 1-11

Caminho 2: 1-2-3-4-5-10-1-11

Caminho 3: 1-2-3-6-8-9-10-1-11

Caminho 4: 1-2-3-6-7-9-10-1-11

Note que cada novo caminho introduz uma nova aresta. O caminho

1-2-3-4-5-10-1-2-3-6-8-9-10-1-11

não é considerado um caminho independente porque é simplesmente uma combinação dos caminhos já especificados e não atravessa nenhuma nova aresta.

Os caminhos de 1 a 4 constituem um *conjunto base* para o grafo de fluxo da Figura 19.5b. Isto é, se testes podem ser projetados para forçar a execução desses caminhos (conjunto base), cada comando do programa terá sido executado com certeza pelo menos uma vez, e cada condição terá sido executada em seus lados verdadeiro e falso. Deve-se notar que o conjunto-base não é único. De fato, vários conjuntos-base diferentes podem ser derivados para um dado projeto procedimental.

Como sabemos quantos caminhos procurar? O cálculo da complexidade ciclomática fornece a resposta. *Complexidade ciclomática* é uma métrica de *software* que fornece uma medida quantitativa da complexidade lógica de um programa. Quando usada no contexto do método de teste de caminho básico, o valor calculado para a complexidade ciclomática define o número de caminhos independentes no conjunto base de um programa, fornecendo um limite superior para a quantidade de testes que devem ser realizados para garantir que todos os comandos tenham sido executados pelo menos uma vez.

A complexidade ciclomática tem um fundamento na teoria dos grafos e fornece uma métrica de *software* extremamente útil. A complexidade é calculada por uma de três maneiras:

1. O número de regiões do grafo de fluxo corresponde à complexidade ciclomática.
2. A complexidade ciclomática $V(G)$ para um grafo de fluxo G é definida como

$V(G) = E - N + 2$

em que E é o número de arestas do grafo de fluxo e N é o número de nós do grafo de fluxo.

3. A complexidade ciclomática $V(G)$ para um grafo de fluxo G é definida como

   $V(G) = P + 1$

   em que P é o número de nós predicados contidos no grafo de fluxo G.

Examinando mais uma vez o diagrama de fluxo da Figura 19.5b, a complexidade ciclomática pode ser calculada usando cada um dos algoritmos citados anteriormente:

1. O grafo de fluxo tem quatro regiões.
2. $V(G)$ = 11 arestas – 9 nós + 2 = 4.
3. $V(G)$ = 3 nós predicados + 1 = 4.

Portanto, a complexidade ciclomática para o grafo de fluxo da Figura 19.5b é 4.

E o mais importante: o valor para $V(G)$ fornece um limite superior para o número de caminhos independentes que podem formar o conjunto-base e, como consequência, um limite superior sobre o número de testes que devem ser projetados e executados para garantir a abrangência de todos os comandos do programa. Assim, neste caso, precisaríamos definir no máximo quatro casos de teste para exercitar cada caminho lógico independente.

Casa segura

Usando a complexidade ciclomática

Cena: Sala da Shakira.

Atores: Vinod e Shakira – membros da equipe de engenharia de *software* do *CasaSegura* que estão trabalhando no planejamento de teste para as funções de segurança.

Conversa:

Shakira: Olha... sei que deveríamos fazer o teste de unidade em todos os componentes da função de segurança, mas eles são muitos, e, se você considerar o número de operações que precisam ser exercitadas, eu não sei... talvez devamos nos esquecer o teste caixa-branca, integrar tudo e começar a fazer os testes caixa-preta.

Vinod: Você acha que não temos tempo suficiente para fazer o teste dos componentes, realizar as operações e então integrar?

Shakira: O prazo final para o primeiro incremento está se esgotando mais rápido do que eu gostaria... sim, estou preocupada.

Vinod: Por que você não aplica testes caixa-branca pelo menos nas operações que têm maior probabilidade de apresentar erros?

Shakira (desesperada): E como posso saber exatamente quais são as que têm maior possibilidade de erro?

Vinod: V de G.

Shakira: Hein?

Vinod: Complexidade ciclomática – V de G. Basta calcular V(G) para cada uma das operações dentro de cada um dos

componentes e ver quais têm os maiores valores para V(G). São essas que têm maior tendência a apresentar erro.

Shakira: E como calculo V de G?

Vinod: É muito fácil. Aqui está um livro que descreve como fazer.

Shakira (folheando o livro): OK, não parece difícil. Vou tentar. As operações que tiverem os maiores V(G) serão as candidatas aos testes caixa-branca.

Vinod: Mas lembre-se de que não há garantia. Um componente com baixo valor V(G) pode ainda estar sujeito a erro.

Shakira: Tudo bem. Isso pelo menos me ajuda a limitar o número de componentes que precisam passar pelo teste caixa-branca.

19.4.2 Teste de estrutura de controle

A técnica de teste de caminho base descrita na Seção 19.4.1 é uma dentre várias técnicas para teste de estrutura de controle. Embora o teste de caminho base seja simples e altamente eficaz, ele sozinho não é suficiente. Nesta seção, são discutidas outras variações do teste de estrutura de controle. Elas ampliam a abrangência do teste e melhoram a qualidade do teste caixa-branca.

Teste de condição [Tai89] é um método de projeto de caso de teste que exercita as condições lógicas contidas em um módulo de programa. O *teste de fluxo de dados* [Fra93] seleciona caminhos de teste de um programa de acordo com a localização de definições e usos de variáveis no programa.

O *teste de ciclo* é uma técnica de teste caixa-branca que se concentra exclusivamente na validade das construções de ciclo. Podem ser definidas duas diferentes classes de ciclos [Bei90]: ciclos simples e ciclos aninhados (Figura 19.6).

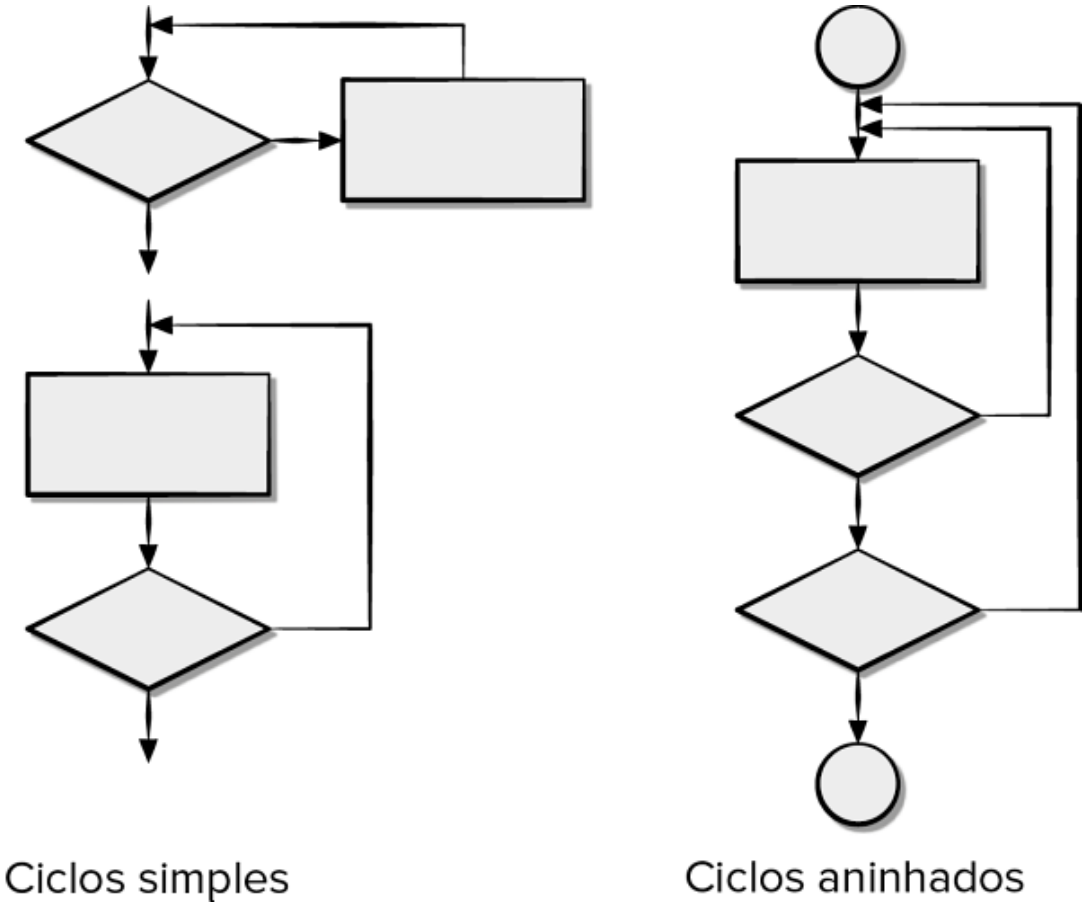

Figura 19.6
Classes de ciclos.

Ciclos simples. O seguinte conjunto de testes pode ser aplicado a ciclos simples, onde n é o número máximo de passadas permitidas através do ciclo.

1. Pular o ciclo inteiramente.
2. Somente uma passagem pelo ciclo.
3. Duas passagens pelo ciclo.
4. m passagens através do ciclo onde $m < n$.
5. $n - 1$, n, $n + 1$ passagens através do ciclo.

Ciclos aninhados. Se fôssemos estender a abordagem de teste de ciclos simples para ciclos aninhados, o número de testes possíveis

cresceria geometricamente à medida que o nível de aninhamento aumentasse. O resultado seria um número impossível de testes. Beizer [Bei90] sugere uma abordagem que ajudará a reduzir o número de testes:

1. Comece pelo ciclo mais interno. Coloque todos os outros ciclos nos seus valores mínimos.
2. Faça os testes de ciclo simples para o ciclo mais interno mantendo, ao mesmo tempo, os ciclos externos em seus parâmetros mínimos de iteração (p. ex., contador do ciclo). Acrescente outros testes para valores fora do intervalo ou excluídos.
3. Trabalhe para fora, fazendo testes para o próximo ciclo, mas mantendo todos os outros ciclos externos nos seus valores mínimos e outros ciclos aninhados com valores "típicos".
4. Continue até que todos os ciclos tenham sido testados.

19.5 Teste caixa-preta

Teste caixa-preta, também chamado de *teste comportamental* ou *teste funcional*, focaliza os requisitos funcionais do *software*. As técnicas de teste caixa-preta permitem derivar séries de condições de entrada que utilizarão completamente todos os requisitos funcionais para um programa. O teste caixa-preta não é uma alternativa às técnicas caixa-branca. Em vez disso, é uma abordagem complementar, com possibilidade de descobrir uma classe de erros diferente daquela obtida com métodos caixa-branca.

O teste caixa-preta tenta encontrar erros nas seguintes categorias: (1) funções incorretas ou ausentes; (2) erros de interface; (3) erros em estruturas de dados ou acesso a bases de dados externas; (4) erros de comportamento ou de desempenho; e (5) erros de inicialização e término.

Diferentemente do teste caixa-branca, que é executado antecipadamente no processo de teste, o teste caixa-preta tende a ser aplicado durante estágios posteriores do teste. Devido ao teste

caixa-preta propositadamente desconsiderar a estrutura de controle, a atenção é focalizada no domínio das informações. Os testes são feitos para responder às seguintes questões:

- Como a validade funcional é testada?
- Como o comportamento e o desempenho do sistema são testados?
- Quais classes de entrada farão bons casos de teste?
- O sistema é particularmente sensível a certos valores de entrada?
- Como as fronteiras de uma classe de dados são isoladas?
- Quais taxas e volumes de dados o sistema pode tolerar?
- Combinações específicas de dados terão quais efeitos sobre a operação do sistema?

Com a aplicação de técnicas de caixa-preta, é extraído um conjunto de casos de teste que satisfazem aos seguintes critérios [Mye79]: casos de teste que reduzem, de um valor maior que 1, o número de casos de teste adicionais que devem ser projetados para se obter um teste razoável; e casos de teste que dizem alguma coisa sobre a presença ou ausência de classes de erros, em vez de um erro associado somente ao teste específico que se está fazendo.

19.5.1 Teste de interface

O *teste de interface* é utilizado para verificar que o componente do programa aceita informações repassadas na ordem apropriada e nos tipos de dados apropriados e retorna informações na ordem e no formato de dados apropriados [Jan16]. O teste de interface costuma ser considerado parte do teste de integração. Como a maioria dos componentes não é independente, é importante se assegurar que, quando o componente é integrado ao programa em evolução, ele não estraga a versão. É aqui que o uso de pseudocontrolados (*stubs*) e pseudocontroladores (*drivers*) (Seção 19.2.1) se torna importante para quem testa componentes.

Às vezes, os pseudocontrolados e pseudocontroladores incorporam casos de teste a serem repassados para o componente ou acessados por ele. Em outros casos, o código de depuração pode precisar ser inserido no componente para verificar que os dados repassados foram recebidos corretamente (Seção 19.3). Em outros casos, ainda, o *framework* de teste deve conter código para verificar que os dados retornados pelo componente são recebidos corretamente. Alguns desenvolvedores ágeis preferem realizar o teste de interface usando uma cópia da versão de produção do programa em evolução tendo agregado parte desse código de depuração.

19.5.2 Particionamento de equivalência

Particionamento de equivalência é um método de teste caixa-preta que divide o domínio de entrada de um programa em classes de dados a partir das quais podem ser criados casos de teste. Um caso de teste ideal descobre, sozinho, uma classe de erros (p. ex., processamento incorreto de todos os dados de caracteres) que poderia, de outro modo, exigir a execução de muitos casos de teste até que o erro geral aparecesse.

O projeto de casos de teste para particionamento de equivalência tem como base a avaliação das *classes de equivalência* para uma condição de entrada. Usando conceitos introduzidos na seção anterior, se um conjunto de objetos pode ser vinculado por relações simétricas, transitivas e reflexivas, uma classe de equivalência estará presente [Bei95]. Uma classe de equivalência representa um conjunto de estados válidos ou inválidos para condições de entrada. Tipicamente, uma condição de entrada é um valor numérico específico, um intervalo de valores, um conjunto de valores relacionados ou uma condição booleana. Classes de equivalência podem ser definidas de acordo com as seguintes regras:

1. Se uma condição de entrada especifica um intervalo, são definidas uma classe de equivalência válida e duas classes de equivalência inválidas.
2. Se uma condição de entrada exige um valor específico, são definidas uma classe de equivalência válida e duas classes de equivalência inválidas.
3. Se uma condição de entrada especifica um membro de um conjunto, são definidas uma classe de equivalência válida e uma classe de equivalência inválida.
4. Se uma condição de entrada for booleana, são definidas uma classe válida e uma inválida.

Aplicando as diretrizes para a derivação de classes de equivalência, podem ser desenvolvidos e executados casos de teste para o domínio de entrada de cada item de dado. Os casos de teste são selecionados de maneira que o máximo de atributos de uma classe de equivalência seja exercitado ao mesmo tempo.

19.5.3 Análise de valor limite

Um número maior de erros ocorre nas fronteiras do domínio de entrada e não no "centro". É por essa razão que foi desenvolvida a *análise do valor limite* (BVA, do inglês *boundary value analysis)* como uma técnica de teste. A análise de valor limite leva a uma seleção de casos de teste que utilizam valores limites.

A análise de valor limite é uma técnica de projeto de casos de teste que complementa o particionamento de equivalência. Em vez de selecionar qualquer elemento de uma classe de equivalência, a BVA conduz à seleção de casos de teste nas "bordas" da classe. Em vez de focalizar somente nas condições de entrada, a BVA obtém casos de teste também a partir do domínio de saída [Mye79].

As diretrizes para a BVA são similares, em muitos aspectos, àquelas fornecidas para o particionamento de equivalência:

1. Se uma condição de entrada especifica um intervalo limitado por valores *a* e *b*, deverão ser projetados casos de teste com valores *a* e *b* imediatamente acima e abaixo de *a* e *b.*
2. Se uma condição de entrada especifica um conjunto de valores, deverão ser desenvolvidos casos de teste que usam os números mínimo e máximo. São testados também valores imediatamente acima e abaixo dos valores mínimo e máximo.
3. Aplique as diretrizes 1 e 2 às condições de saída. Por exemplo, suponha que um programa de análise de engenharia precisa ter como saída uma tabela de temperatura *versus* pressão. Deverão ser projetados casos de teste para criar um relatório de saída que produza o número máximo (e mínimo) permitido de entradas da tabela.
4. Se as estruturas de dados internas do programa prescreveram fronteiras (p. ex., uma tabela tem um limite definido de 100 entradas), não se esqueça de criar um caso de teste para exercitar a estrutura de dados na fronteira.

Até certo ponto, a maioria dos engenheiros de *software* executa intuitivamente a BVA. Aplicando essas diretrizes, o teste de fronteira será mais completo, tendo, assim, uma possibilidade maior de detecção de erro.

19.6 Teste orientado a objetos

Quando consideramos o *software* orientado a objetos, 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 ao teste de unidade precisa ser modificada.

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 classes 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. 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 isoladamente (a abordagem de teste de unidade convencional) normalmente é ineficaz no contexto orientado a objetos.

19.6.1 Teste de conjunto

O teste de classe para *software* orientado a objetos é equivalente ao teste de unidade para *software* convencional. Ao contrário do teste de unidade para *software* convencional, que tende a focalizar o detalhe algorítmico de um módulo e os dados que fluem por meio da interface do módulo, o teste de classe para *software* orientado a objetos é controlado pelas operações encapsuladas pela classe e pelo comportamento de estado da classe.

Para uma breve ilustração desses métodos, considere uma aplicação bancária na qual uma classe **Conta** tem estas operações: *abrir(), estabelecer(), depositar(), retirar(), obterSaldo(), resumir(), limiteDeCrédito()* e *fechar()* [Kir94]. Cada operação pode ser aplicada para **Conta**, mas certas restrições (p. ex., primeiro a conta precisa ser aberta, para que as outras operações possam ser aplicadas, e fechada depois que todas as operações são concluídas) 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]n • retirar • fechar
```

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

Casos de teste r_1:

```
abrir • estabelecer • depositar • obterSaldo • resumir • retirar • fechar
```

Casos 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. O uso do particionamento de equivalência de teste (Seção 19.5.2) pode reduzir o número de casos de teste necessários.

Casa segura

Teste de classe

Cena: Sala 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 11.4) – você sabe, aquela que permite acesso a todos os objetos **Sensor** para a função de segurança. Conhece?

Jamie (rindo): Claro, é aquela que você usou para acrescentar o sensor "mal-estar 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, ele primeiro tem de ser 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, defini uma sequência simples de teste que vai simular sua história comportamental.

(Ela mostra a Jamie a seguinte sequência.)

`#1: habilitar • testar • ler • desabilitar`

Jamie: Vai funcionar, mas você terá de fazer mais testes do que isso.

Shakira: Eu sei, eu sei. Aqui estão algumas outras sequências que eu descobri. (Ela 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() deve 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.

19.6.2 Teste comportamental

O uso dos diagramas de estado como um modelo que representa o comportamento dinâmico de uma classe é discutido no Capítulo 8. O diagrama de estado de uma classe pode ser usado para ajudar a derivar uma sequência de testes que vão simular o comportamento dinâmico da classe (e as classes que colaboram com ela). A Figura 19.7 [Kir94] ilustra um diagrama de estado para a classe **Conta** discutida anteriormente. Na figura, as transações iniciais movem-se para os estados *conta vazia* e *conta estabelecida*. A maior parte do comportamento para instâncias da classe ocorre ainda no estado *conta ativa*. Uma retirada final e o fechamento da conta fazem a classe **Conta** transitar para os estados *conta inativa* e *conta "morta"*, respectivamente.

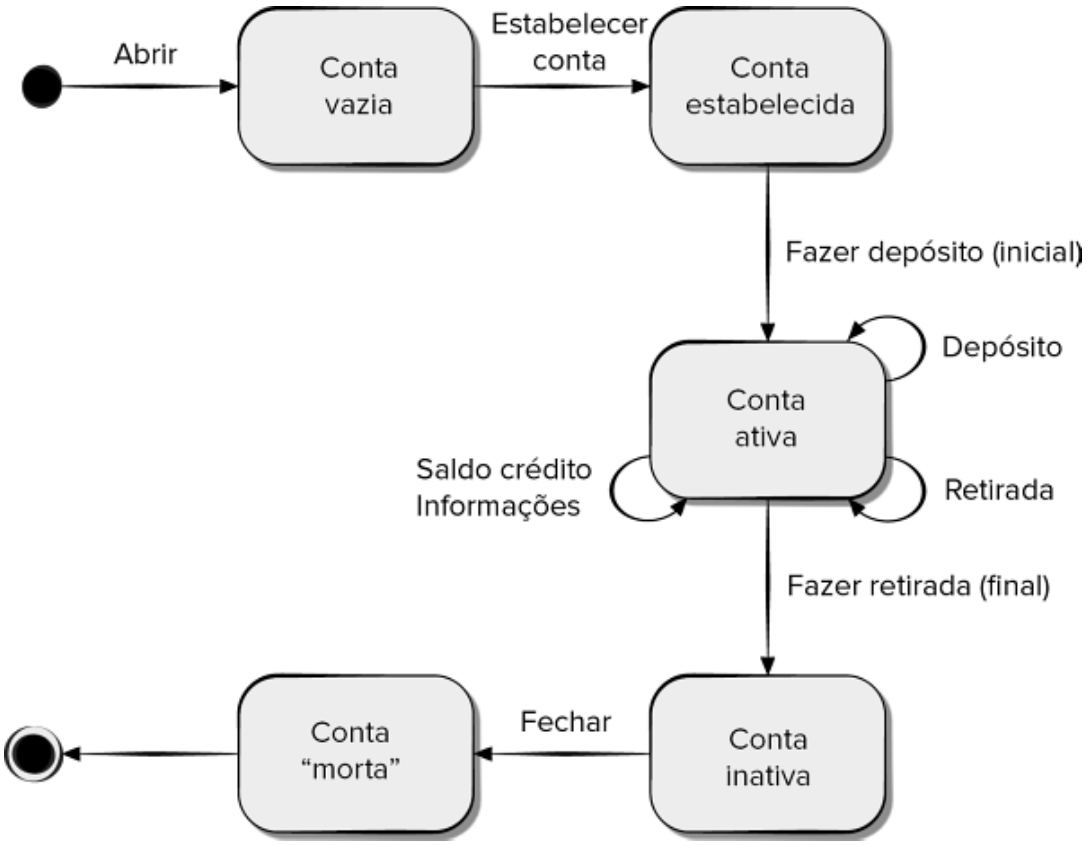

Figura 19.7
Diagrama de estados para a classe Conta.
Fonte: Kirani, Shekhar and Tsai, W. T., "Specification and Verification of Object-Oriented Programs," Technical Report TR 94-64, University of Minnesota, December 1994, 79.

Os testes a serem projetados deverão cobrir todos os estados. Isto é, as sequências de operação deverão fazer a classe **Conta** fazer a transição por todos os estados permitidos:

Caso de teste s_1:

`abrir • estabelecerConta • fazerDepósito (inicial) • fazerRetirada (final) • fechar`

Acrescentando as 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`

Mais casos de teste poderiam ser criados para garantir que todos os comportamentos da 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* (i.e., 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 `número do cartão` e `data de validade`, juntamente com identificadores específicos do banco, são definidos. O cartão de crédito é *submetido* (enviado) ao ser enviado para autorização e é *aprovado* quando a autorização é recebida. A transição de um estado para outro de **CartãoDeCrédito** 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 de transições que não foram testadas previamente e, portanto, violaria o critério primeiro-em-largura.

19.7 Resumo

O teste de *software* absorve a maior parte do esforço técnico em um processo de *software*. Independentemente do tipo de *software* criado, uma estratégia para planejamento sistemático de teste, execução e controle começa considerando pequenos elementos do *software* e se encaminha para fora no sentido de abranger o programa como um todo.

O objetivo do teste de *software* é descobrir erros. Para o *software* convencional, esse objetivo é atingido com uma série de etapas de teste. Testes de unidade e de integração (discutidos no Capítulo 20) concentram-se na verificação funcional de um componente e na incorporação dos componentes na arquitetura de *software*. A estratégia para teste de *software* orientado a objetos começa com testes que exercitam as operações dentro de uma classe e depois passa para o teste baseado em sequências de execução para integração (discutido na Seção 20.4.1). Sequências de execução são conjuntos de classes que respondem a uma entrada ou a um evento.

Os casos de teste devem remontar aos requisitos de *software*. Cada etapa de teste é realizada com uma série de técnicas sistemáticas que auxiliam no projeto dos casos de teste. Em cada etapa, o nível de abstração com o qual o *software* é considerado é ampliado. O principal objetivo do projeto de caso de teste é derivar uma série de testes que tenha a mais alta probabilidade de descobrir erros no *software*. Para conseguir esse objetivo, são usadas duas categorias de técnicas de projeto de caso de teste: teste caixa-branca e teste caixa-preta.

Os testes caixa-branca focam na estrutura de controle do programa. São criados casos de teste para assegurar que todas as instruções do programa foram executadas pelo menos uma vez durante o teste e que todas as condições lógicas foram exercitadas. O teste de caminho base, uma técnica caixa-branca, usa diagramas de programa (ou matrizes de grafo) para derivar o conjunto de testes linearmente independentes que garantirão a cobertura. O

teste de condições e de fluxo de dados exercita mais a lógica do programa, e o teste de ciclos complementa outras técnicas caixa-branca, fornecendo um procedimento para exercitar ciclos com vários graus de complexidade.

Os testes caixa-preta são projetados para validar requisitos funcionais sem levar em conta o funcionamento interno de um programa. As técnicas caixa-preta focam o domínio de informações do *software*, derivando casos de teste e particionando o domínio de entrada e saída de um programa de forma a proporcionar uma ampla cobertura do teste. O particionamento de equivalência divide o domínio de entrada em classes de dados que tendem a usar uma função específica do *software*. A análise de valor limite investiga a habilidade do programa para manipular dados nos limites do aceitável.

Diferentemente do teste (uma atividade sistemática, planejada), a depuração pode ser vista como uma arte. Começando com a indicação sintomática de um problema, a atividade de depuração deve rastrear a causa de um erro. Em alguns casos, os testes podem ajudar a identificar a causa fundamental do erro, mas, em geral, o recurso mais valioso é o conselho de outros membros da equipe de engenharia de *software*.

Problemas e pontos a ponderar

19.1 Usando as suas próprias palavras, descreva a diferença entre verificação e validação. Ambas usam métodos de projeto de caso de teste e estratégias de teste?

19.2 Liste alguns dos problemas que podem ser associados à criação de um grupo de teste independente. Um ITG e um grupo de SQA são formados pelas mesmas pessoas?

19.3 Por que um módulo altamente acoplado é difícil de testar em unidade?

19.4 O teste de unidade é possível ou até mesmo desejável em todas as circunstâncias? Dê exemplos para justificar a sua

resposta.

19.5 Você consegue pensar em objetivos de teste adicionais que não foram discutidos na Seção 19.1.1?

19.6 Selecione um componente de *software* que você tenha projetado e implementado recentemente. Projete um conjunto de casos de teste que garantirão que todos os comandos foram executados usando teste de caminho base.

19.7 Myers [Mye79] usa o seguinte programa como uma autoavaliação para a sua habilidade em especificar um teste adequado. Um programa lê três valores inteiros. Os três valores são interpretados como representantes dos comprimentos dos lados de um triângulo. O programa imprime uma mensagem que diz se o triângulo é escaleno, isósceles ou equilátero. Desenvolva um conjunto de casos de teste que você acha que testará adequadamente esse programa.

19.8 Projete e implemente o programa (com manipulação de erro onde for apropriado) especificado no Problema 19.7. Crie um grafo de fluxo para o programa e aplique teste de caminho base para desenvolver casos de teste que garantirão que todos os comandos no programa foram testados. Execute os casos e mostre seus resultados.

19.9 Dê pelo menos três exemplos nos quais o teste caixa-preta pode dar a impressão de que "está tudo OK", enquanto os testes caixa-branca podem descobrir um erro. Dê pelo menos três exemplos nos quais o teste caixa-branca pode dar a impressão de que "está tudo OK", enquanto os testes caixa-preta podem descobrir um erro.

19.10 Com suas próprias palavras, descreva por que a classe é a menor unidade adequada para teste em um sistema orientado a objetos.

[registrados] N. de RT.: Evoluindo para automação: o teste como código, como ativo de *software* e controlado por protocolos (p. ex, GIT) tem potencial.

[indiretas] N. de RT.: Uma maneira de amenizar o aspecto da despesa é "ativá-los". Fazem parte do *software* e são úteis em testes de regressão.

1 Deve-se observar que há uma forte divergência de opinião sobre quais tipos de testes constituem "validação". Algumas pessoas acreditam que *todo* teste é verificação e que a validação é realizada quando os requisitos são examinados e aprovados – e, mais tarde, pelo usuário, com o sistema já em operação. Outras pessoas consideram o teste de unidade e de integração (Capítulos 19 e 20) como verificação e o teste de ordem superior (Capítulo 21) como validação.

2 Gilb [Gil95] recomenda que uma equipe de *software* "aprenda a testar em ciclos rápidos (2% do trabalho de projeto) de incrementos de funcionalidade e/ou melhora da qualidade úteis ao cliente, ou pelo menos passíveis de experimentação no campo". O retorno gerado por esses testes de ciclo rápido pode ser usado para controlar os níveis de qualidade e as correspondentes estratégias de teste.

3 Ocasionalmente, os antirrequisitos são descritos durante a criação de casos de abuso que descrevem uma história de usuário da perspectiva de um usuário mal-intencionado e são parte da análise de ameaças (discutida no Capítulo 18).

4 Na realidade, o método do caminho básico pode ser executado sem o uso de grafos de fluxo. No entanto, eles servem como uma notação útil para entender o fluxo de controle e ilustrar a abordagem.

20

Teste de *software* – Nível de integração

Conceitos-chave

inteligência artificial

teste caixa-preta

integração ascendente

integração contínua

teste de classe

teste baseado em falhas

teste de integração

teste de partição de múltiplas classes

padrões

teste de regressão

teste baseado em cenários

teste fumaça

teste baseado em sequências de execução

integração descendente

teste de validação

teste caixa-branca

Um desenvolvedor isolado pode conseguir testar componentes de *software* sem envolver outros membros de equipe. Essa condição não vale para os testes de integração, nos quais um componente precisa interagir corretamente com componentes desenvolvidos

por outros membros. Os testes de integração revelam os muitos pontos fracos dos grupos de desenvolvimento de *software* que não formaram uma equipe coesa e consistente. O teste de integração apresenta um dilema interessante para os engenheiros de *software*, que são, por natureza, pessoas construtivas. Na verdade, ele requer que o desenvolvedor descarte noções preconcebidas da "corretividade" do *software* recém-desenvolvido e passe a trabalhar arduamente projetando casos de teste para "quebrar" o *software*. Isso significa que os membros de equipe precisam ser capazes de aceitar as sugestões dos colegas de que o seu código não está se comportando corretamente quando testado como parte do último incremento de *software*.

Panorama

O que é? O teste de integração monta os componentes de forma a permitir o teste de funções de *software* cada vez maiores, com a intenção de descobrir erros à medida que o *software* é montado.

Quem realiza? Durante os primeiros estágios do teste, um engenheiro de *software* executa todos os testes. Porém, à medida que o processo de teste avança, especialistas podem participar, além de outros envolvidos.

Por que é importante? É preciso utilizar técnicas disciplinadas para projetar os casos de teste de modo a garantir que os componentes foram integrados de forma correta em um produto de *software* completo.

Quais são as etapas envolvidas? A lógica interna do programa é exercitada usando técnicas de projeto de caso de teste "caixa-branca", e os requisitos de *software* são exercitados usando técnicas de projeto de caso de teste "caixa-preta".

Qual é o artefato? Um conjunto de casos de teste projetados para exercitar a lógica interna, interfaces, colaborações entre componentes e os requisitos externos é projetado e documentado, os resultados esperados são definidos, e os resultados obtidos são registrados.

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

Beizer [Bei90] descreve um "mito do *software*" enfrentado por todos os testadores quando escreve: "Há um mito de que, se fôssemos realmente bons em programação, não precisaríamos caçar erros (…) Existem erros, diz o mito, porque somos ruins no que fazemos; e se somos ruins no que fazemos, devemos nos sentir culpados por isso".

O teste deve realmente insinuar culpa? O teste é realmente destrutivo? A resposta a essas questões é "Não!".

No início deste livro, salientamos que o *software* é apenas um elemento de um grande sistema de computador. No final, o *software* é incorporado a outros elementos do sistema (p. ex., *hardware*, pessoas, informações), e são executados *testes de sistema* (uma série de testes de integração e validação). Esses testes estão fora do escopo do processo de *software* e não são executados somente por engenheiros de *software*. No entanto, as etapas executadas durante

o projeto de *software* e o teste podem aumentar muito a probabilidade de uma integração de *software* bem-sucedida em um sistema maior.

Neste capítulo, discutimos técnicas para estratégias de testes de integração de *software* aplicáveis à maioria das aplicações de *software*. O Capítulo 21 discute estratégias especializadas de teste de *software*.

20.1 Fundamentos do teste de *software*

O objetivo dos testes é encontrar erros, então um bom teste é aquele com alta probabilidade de achar um erro. Kaner, Falk e Nguyen [Kan93] sugerem os seguintes atributos para um "bom" teste:

Um bom teste tem alta probabilidade de encontrar um erro. Para atingir esse objetivo, o testador deve entender o *software* e tentar desenvolver uma imagem mental de como ele pode falhar.

Um bom teste não é redundante. O tempo e os recursos de teste são limitados. Não faz sentido realizar um teste que tenha a mesma finalidade de outro teste. Cada teste deve ter uma finalidade diferente (mesmo que seja sutilmente diferente).

Um bom teste deve ser "o melhor da raça" [Kan93]. Em um grupo de testes com finalidades similares, as limitações de tempo e recursos podem induzir à execução de apenas um subconjunto dos testes que tenha a maior probabilidade de revelar uma classe inteira de erros.

Um bom teste não deve ser nem muito simples nem muito complexo. Embora algumas vezes seja possível combinar uma série de testes em um caso de teste, os possíveis efeitos colaterais associados a essa abordagem podem mascarar erros. Em geral, cada teste deve ser executado separadamente.

Qualquer produto de engenharia (e muitas outras coisas) pode ser testado por uma de duas maneiras: (1) conhecendo-se a função especificada para a qual um produto foi projetado para realizar, podem ser feitos testes que demonstrem que cada uma das funções é totalmente operacional, embora ao mesmo tempo procurem erros em cada função; (2) conhecendo-se o funcionamento interno de um produto, podem ser realizados testes para garantir que "tudo se encaixa" – isto é, que as operações internas foram realizadas de acordo com as especificações e que todos os componentes internos foram adequadamente exercitados. A primeira abordagem de teste usa uma visão externa e é chamada de *teste caixa-preta*. A segunda estratégia exige uma visão interna e é chamada de *teste caixa-branca*.[1] Ambas são úteis nos testes de integração [Jan16].

20.1.1 Teste caixa-preta

O *teste caixa-preta* faz referência a testes de integração realizados pelo exercício das interfaces do componente com outros componentes e com outros sistemas. Um teste caixa-preta examina alguns aspectos fundamentais de um sistema, com pouca preocupação em relação à estrutura lógica interna do *software*. Em vez disso, o foco está em garantir que o componente execute corretamente dentro da versão de *software* maior quando os dados de entrada e o contexto de *software* especificado pelas suas precondições estão corretos e que se comporte de acordo com as maneiras especificadas pelas suas pós-condições. Obviamente, é importante garantir que o componente se comporte corretamente quando as suas pré-condições não são satisfeitas (p. ex., que consegue lidar com entradas incorretas sem entrar em pane).

O teste caixa-preta se baseia em requisitos especificados nas histórias de usuário (Capítulo 7). Os autores de casos de teste não precisam esperar que o código de implementação do componente seja escrito após a interface do componente ser definida. Diversos componentes que cooperam entre si podem precisar ser desenvolvidos para implementar a funcionalidade definida por

uma única história de usuário. O teste de validação (Seção 20.5) muitas vezes define casos de teste caixa-preta em termos de ações de entrada visíveis para o usuário e comportamentos de saída observáveis, sem nenhum conhecimento sobre como os componentes em si foram implementados.

20.1.2 Teste caixa-branca

O *teste caixa-branca*, também chamado de *teste da caixa-de-vidro* ou *teste estrutural*, é uma filosofia de teste de integração que usa o conhecimento sobre implementação da estrutura de controle descrita como parte do projeto no nível de componentes para derivar casos de teste. O teste caixa-branca fundamenta-se em um exame rigoroso do detalhe procedimental e dos detalhes da implementação da estrutura de dados. Testes caixa-branca só podem ser projetados depois que o projeto no nível de componente (ou código-fonte) existir. Os detalhes lógicos do programa devem estar disponíveis. Os caminhos lógicos do *software* e as colaborações entre componentes são o foco do teste de integração caixa-branca.

À primeira vista, poderia parecer que um teste caixa-branca realizado de forma rigorosa resultaria em "programas 100% corretos". Tudo o que seria preciso fazer seria definir todos os caminhos lógicos, desenvolver casos de teste para exercitá-los e avaliar os resultados, ou seja, gerar casos de teste para exercitar a lógica do programa de forma exaustiva. Infelizmente, o teste exaustivo apresenta certos problemas logísticos. Mesmo para programas pequenos, o número de caminhos lógicos possíveis pode ser muito grande. No entanto, o teste caixa-branca não deve ser descartado como impraticável. Os testadores devem selecionar um número limitado de caminhos lógicos a serem exercitados após a integração do componente. A validade de estruturas de dados importantes também deve ser testada após a integração do componente.

20.2 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, por que você duvida que funcionem quando estiverem juntos?". O problema, naturalmente, é "colocá-los todos juntos" – fazer 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 em que se realizam testes para descobrir erros associados às 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 a integração não incremental; isto é, construir o programa usando uma abordagem "*big bang*". Todos os componentes são combinados antecipadamente, e o programa inteiro é testado como um todo. E, normalmente, o resultado é o caos! Os erros são encontrados, mas a correção é difícil porque o isolamento das causas é complicado pela vasta extensão do programa inteiro. Adotar o método "*big bang*" é uma abordagem ineficaz, destinada a fracassar.

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 serem testadas completamente; e uma abordagem sistemática de teste pode ser aplicada. Integrar incrementalmente e ir testando pelo caminho é uma estratégia com melhor relação custo-benefício. Discutimos diversas estratégias comuns de testes de integração incremental no restante deste capítulo.

20.2.1 Integração descendente

Teste de integração descendente (*top-down*) é uma abordagem incremental para a construção da arquitetura de *software*. Os módulos (também chamados de "componentes" neste livro) são integrados deslocando-se para baixo por meio 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 20.1, a *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 (p. ex., os componentes que precisam ser implementados em um caso de uso). Por exemplo, selecionando o caminho da esquerda, os componentes M1, M2 e M5 seriam integrados primeiro. Em seguida, seria integrado o M8 – ou, se necessário para o funcionamento apropriado de M2, o M6. 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 M2, M3 e M4 seriam integrados primeiro. Em seguida, vem o próximo nível de controle, M5, M6 e assim por diante. O processo de integração é executado em uma série de cinco passos:

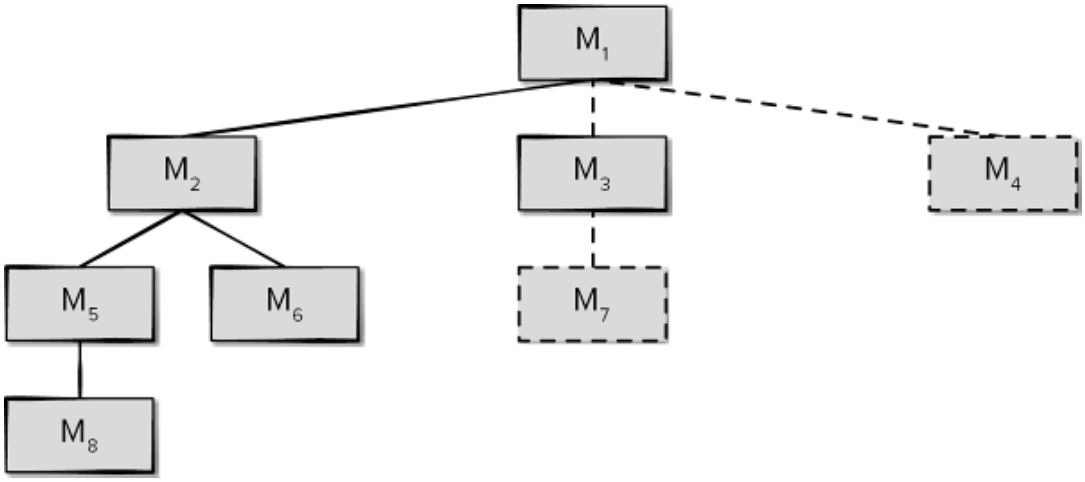

Figura 20.1
Integração descendente (*top-down*).

1. O módulo de controle principal é utilizado como um testador (*test driver*), e todos os componentes diretamente subordinados ao módulo de controle principal substituem os pseudocontrolados (*stubs*).
2. Dependendo da abordagem de integração selecionada (i.e., primeiro-em-profundidade ou primeiro-em-largura), componentes pseudocontrolados (*stubs*) subordinados são substituídos, um de cada vez, pelos componentes reais.
3. Os testes são feitos à medida que cada componente é integrado.
4. Ao fim de cada conjunto de testes, outro pseudocontrolado é substituído pelo 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 (*top-down*) 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 da hierarquia e, portanto, é encontrada primeiro. Se existirem problemas de controle principal, um reconhecimento prévio é essencial. Se for selecionada a integração primeiro-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 envolvidos.

20.2.2 Integração ascendente

O *teste de integração ascendente* (*bottom-up*), como o nome diz, começa a construção e o teste com *módulos atômicos* (i.e., componentes nos níveis mais baixos na estrutura do programa). A integração ascendente elimina a necessidade de pseudocontrolados complexos. Como os componentes são integrados de baixo para cima, a funcionalidade proporcionada por componentes subordinados a determinado nível está sempre disponível, e a necessidade de pseudocontrolados é eliminada. Uma estratégia de integração ascendente pode ser implementada com os seguintes passos:

1. Componentes de baixo nível são combinados em agregados (*cluster* – módulos) 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 módulo, como um conjunto de componentes, 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 20.2. Os componentes são combinados para formar os agregados (*módulos*) 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 fazem interface diretamente com 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 finalmente integrados ao componente M *c* e assim por diante.

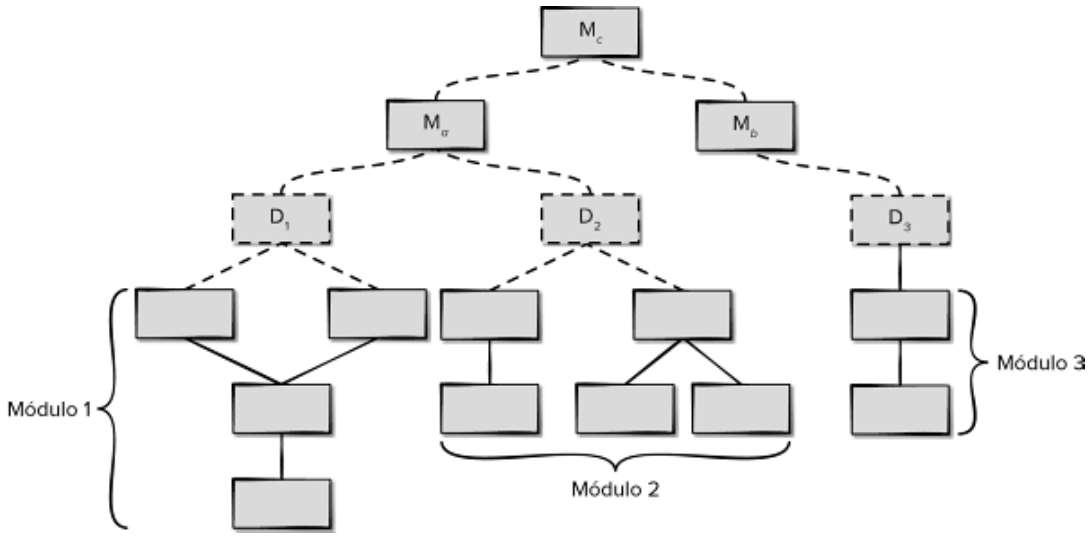

Figura 20.2
Integração ascendente (*bottom-up*).

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.

20.2.3 Integração contínua

Integração contínua é a prática de fundir componentes com o incremento de *software* em evolução uma ou mais vezes ao dia. É uma prática comum para equipes que seguem práticas de

desenvolvimento ágil, como XP (Seção 3.5.1) ou DevOps (Seção 3.5.2). O teste de integração deve ocorrer rápida e eficientemente se a equipe está tentando, como parte da entrega contínua, sempre ter um programa funcional implementado. Às vezes, é difícil executar a manutenção de sistemas com o uso de ferramentas de integração contínua [Ste18]. Os problemas de manutenção e integração contínua são discutidos em mais detalhes na Seção 22.4.

Teste fumaça é uma abordagem de teste de integração que pode ser usada quando produtos de *software* são desenvolvidos por uma equipe ágil usando tempos curtos para a construção de incrementos, e pode ser caracterizada como uma estratégia de integração contínua ou móvel. O *software* é recriado (com novos componentes acrescentados), e o teste fumaça é realizado todos os dias. É projetado como um mecanismo de marca-passo para projetos com prazo crítico, permitindo que a equipe de *software* avalie o projeto frequentemente. Em essência, a abordagem do 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 deve ser descobrir erros "bloqueadores" (*show-stoppers*) que apresentam a mais alta probabilidade de atrasar o cronograma do *software*.
3. A construção é integrada a outras construções, e o produto inteiro (em sua forma atual) passa diariamente pelo teste fumaça. A abordagem de integração pode ser descendente ou ascendente.

A frequência diária dos testes dá, 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 fim-a-fim. 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 supor 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* complexos e de prazo crítico:

- **O risco da integração é minimizado.** Como os testes fumaça são feitos diariamente, as incompatibilidades e outros erros bloqueadores são descobertos logo, reduzindo, assim, a probabilidade de impacto sério no cronograma quando os erros são descobertos.
- **A qualidade do produto final é melhorada.** Como a abordagem é 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, isso 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 aos "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 ele funciona. Isso melhora a moral da equipe e dá aos gerentes uma boa indicação de que houve progressos.

Em alguns aspectos, o teste fumaça lembra o teste de regressão (discutido na Seção 20.3), que ajuda a garantir que os componentes

recém-integrados não irão interferir com os comportamentos dos componentes existentes testados anteriormente. Para tanto, é uma boa ideia reexecutar um subconjunto de casos de teste que foram rodados com o componente de *software* existente antes dos novos componentes serem agregados. O esforço necessário para reexecutar os casos de teste não é trivial, e os testes automatizados podem ser utilizados para reduzir o tempo e o esforço necessários para executá-los novamente [Net18]. Uma discussão completa sobre testes automatizados iria além do escopo deste capítulo, mas *links* para ferramentas representativas estão disponíveis nas páginas Web que suplementam este livro.[2]

20.2.4 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 artefato 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, processamento do sensor, funções de comunicação e processamento do alarme.

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 criados incrementos de *software* para corresponder a cada fase.

Um cronograma para a integração, o desenvolvimento de *software* de *scaffolding* (Seção 19.2.1) e tópicos relacionados também são 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. Durante o desenvolvimento do cronograma do projeto, é preciso

considerar como a integração ocorre para que os componentes estejam disponíveis quando necessário. Uma breve descrição do *software* de *scaffolding* (pseudocontroladores e pseudocontrolados) concentra-se nas características que poderiam exigir dedicação especial. Por fim, 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 de teste detalhado necessário para realizar o plano de teste. São descritos a ordem de integração e os testes correspondentes em cada etapa de integração. É incluída também uma lista de todos os casos de teste (anotados para referência subsequente) e dos resultados esperados. No mundo ágil, esse nível de descrição dos casos de teste ocorre quando o código para implementar a história de usuário está sendo desenvolvido para que o código possa ser testado assim que estiver pronto para a integração.

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*. Em geral, a melhor forma de implementar o relatório de teste é usar um documento Web compartilhado para dar a todos os envolvidos acesso aos resultados mais recentes dos testes e ao estado atual do incremento de *software*. As informações contidas nesse documento *online* podem ser essenciais para os desenvolvedores durante a manutenção do *software* (Seção 4.9).

20.3 Inteligência artificial e testes 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 (I/O, do inglês *input / output*) e nova lógica de controle é chamada. Os efeitos colaterais associados a essas alterações podem causar

problemas em 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. Execute testes de regressão toda vez que for feita uma alteração grande no *software* (incluindo a integração de novos componentes). 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 subsequentes. O *conjunto de teste de regressão* (o subconjunto de testes a serem 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.

Yoo e Harman [Yoo13] escrevem sobre os usos em potencial da inteligência artificial (IA) na identificação de casos de teste para uso em conjuntos de teste de regressão. Uma ferramenta de *software* poderia analisar as dependências entre os componentes no

incremento de *software* após os novos componentes terem sido adicionados e gerar casos de teste automaticamente para serem utilizados nos testes de regressão. Outra possibilidade seria usar técnicas de aprendizado de máquina para selecionar conjuntos de casos de teste que otimizariam a descoberta de erros de colaboração entre componentes. O trabalho é promissor, mas ainda exige um nível significativo de interação humana para analisar os casos de teste e a sua ordem de execução recomendada.

Casa segura

Testes de regressão

Cena: Sala de Doug Miller, enquanto testes de integração estão sendo executados.

Atores: Doug Miller, gerente de engenharia de *software*; Vinod, Jamie, Ed e Shakira – membros da equipe de engenharia de *software* do *CasaSegura*.

Conversa:

Doug: Parece-me que não estamos dedicando tempo suficiente para retestar os componentes de *software* após integrarmos novos componentes.

Vinod: Acho que isso é verdade, mas não é suficiente que estejamos testando as interações dos novos componentes com aqueles com os quais devem colaborar?

Doug: Nem sempre. Alguns componentes causam alterações inesperadas aos dados utilizados por outros componentes. Sei que estamos ocupados, mas é importante descobrir esse tipo de problema o quanto antes.

Shakira: Até temos um repositório de casos de teste que temos utilizado. Talvez pudéssemos selecionar aleatoriamente vários casos de teste para serem rodados usando o nosso framework de teste automatizado.

Doug: É um bom começo, mas talvez devêssemos ser mais estratégicos na seleção dos nossos casos de teste.

Ed: Acho que poderíamos usar a nossa tabela de rastreabilidade de requisitos/casos de teste e usar nosso modelo de cartões CRC para confirmar.

Vinod: Tenho usado integração contínua, o que significa que integro cada componente assim que um dos desenvolvedores me passa ele. Tento executar uma série de testes de regressão no programa parcialmente integrado.

Jamie: Tenho tentado projetar um conjunto de testes apropriados para cada função do sistema. Talvez eu devesse identificar alguns dos mais importantes para que o Vinod os use no teste de regressão.

Doug (para Vinod): Com que frequência você executa os testes de regressão?

Vinod: Todos os dias em que integro um novo componente, uso os casos de teste de regressão... até decidirmos que o incremento de *software* está pronto.

Doug: Vamos experimentar usar os casos de teste de regressão do Jamie à medida que são criados e ver o que acontece.

20.4 Teste de integração em contexto orientado a objetos

Devido ao *software* orientado a objetos não ter uma estrutura óbvia de controle hierárquico, as estratégias tradicionais de integração descendente e ascendente (Seção 20.2) 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 orientados a objetos [Bin99]: teste baseado em sequências de execução e testes baseados em uso. 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 execução é integrada e testada individualmente. O teste de regressão é aplicado para garantir que não ocorram efeitos colaterais. Uma estratégia importante para integração de *software* orientado a objetos é o teste com base em sequência de execução (*thread)*. Sequências de execução são conjuntos de classes que respondem a uma entrada ou evento.

A segunda abordagem de integração, *teste baseado em uso* (*use-based testing*), inicia a construção do sistema testando as classes (chamadas de *classes independentes*) que usam poucas (ou nenhuma) 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 camadas de teste de classes dependentes continua até que todo o sistema seja construído. Testes baseados em uso focalizam classes que não colaboram intensamente com outras classes.

O uso de *software* de *scaffolding* também muda quando é executado o teste de integração de sistemas orientados a objetos. Pseudocontroladores podem ser utilizados para testar operações no nível mais baixo e para o teste de grupos inteiros de classes. Um pseudocontrolador também pode ser usado para substituir a interface de usuário, de maneira que os testes da funcionalidade do sistema podem ser conduzidos antes da implementação da interface. Pseudocontrolados podem ser usados em situações nas

quais é necessária a colaboração entre classes, mas uma (ou mais) das classes colaboradoras ainda não foi totalmente implementada.

O *teste de conjunto* (*cluster testing*) é uma etapa no teste de integração de *software* orientado a objetos. Nesse caso, um agregado de classes colaboradoras (determinado examinando-se os modelos CRC e o modelo objeto-relacionamento) é exercitado projetando-se casos de teste que tentam descobrir erros nas colaborações.

20.4.1 Projeto de caso de teste baseado em falhas[3]

O objetivo do *teste baseado em falhas* dentro de um sistema orientado a objetos 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. 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 testador procura por falhas plausíveis (i.e., 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.

Sem dúvida, a eficiência dessas técnicas depende de como os testadores consideram uma falha plausível. Se falhas reais em um sistema orientado a objetos são vistas como não plausíveis, essa abordagem não é melhor do que qualquer técnica de teste aleatório. Por outro lado, 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 uma quantidade significativa de erros com esforço relativamente pequeno.

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 observar 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.

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 19.6.1. No entanto, a equivalência de teste é expandida para incluir operações chamadas via mensagens para classes colaboradoras. Uma abordagem alternativa particiona os testes com base nas interfaces para uma classe em particular. Conforme a Figura 20.3, 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.**

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 ela 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** (caixa eletrônico) (Figura 20.3):

```
verificarConta • verificarPIN •
[[verificarDiretrizes • requisiçãoDeRetirada] |
requisiçãoDeDepósito | requisiçãoDe
InformaçãoDeConta]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: contaVálidaInformação
DeValidação] • verificarPIN [Banco: PINVálido
InformaçãoDeValidação] • requisiçãoDeDepósito
[Banco: depósitoConta]
```

20.4.2 Projeto de caso de teste baseado em cenários

O teste baseado em falhas omite dois tipos principais de erros: (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. Ele pode fazer alguma coisa

errada ou omitir uma funcionalidade importante. Mas, em qualquer circunstância, a qualidade (conformidade com os requisitos) é prejudicada. Erros associados à interação de subsistemas ocorrem quando o comportamento de um subsistema cria circunstâncias (p. ex., eventos, fluxo de dados) que fazem outro subsistema falhar.

Testes baseados em cenários indicarão erros que ocorrem quando qualquer ator interage com o *software*. O teste baseado em cenários 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.

Esses testes 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 cenários tendem a usar vários subsistemas em um único teste (os usuários não se limitam ao uso de um subsistema de cada vez).

O projeto de caso de teste torna-se mais complicado quando começa a integração do sistema orientado a objetos. É nesse estágio que o teste de colaborações entre classes deve começar. Para ilustrar a "geração de caso de teste entre classes" [Kir94], vamos expandir o exemplo bancário apresentado na Seção 19.6 para incluir as classes e colaborações da Figura 20.3. As direções das setas na figura indicam a direção das mensagens, e os rótulos indicam as operações chamadas como consequência das colaborações sugeridas pelas mensagens.

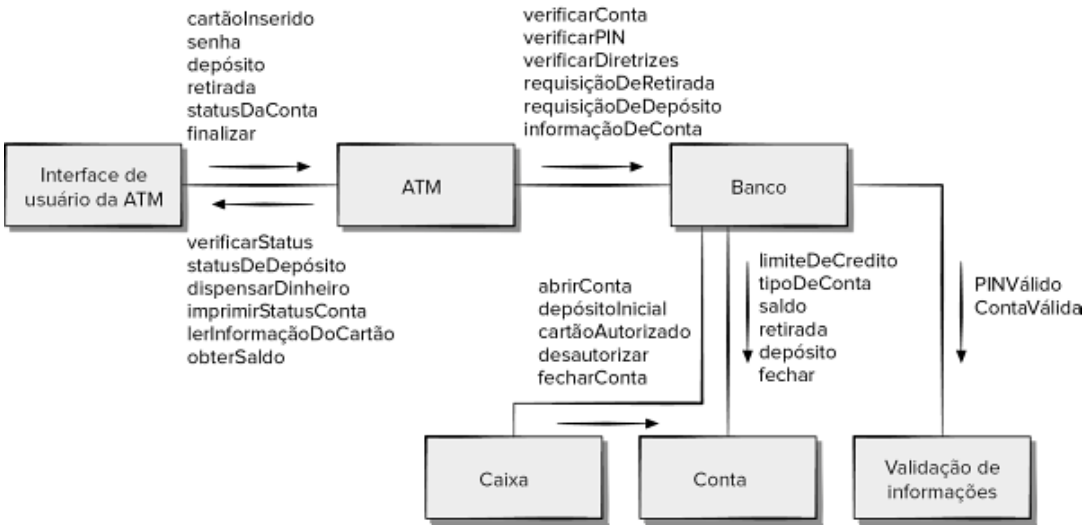

Figura 20.3
Diagrama de colaboração de classe para a aplicação bancária.
Fonte: Kirani, Shekhar and Tsai, W. T., "Specification and Verification of Object-Oriented Programs," Technical Report TR 94-64, University of Minnesota, December$_4$, 1994, 72.

Como o teste de classes individuais, o teste de colaboração entre classes pode ser feito com a aplicação de métodos aleatórios e de particionamento, bem como teste baseado em cenários e teste comportamental.

20.5 Teste de validação

Como todas as etapas de teste, a validação tenta descobrir erros, mas o foco está no nível de requisitos – em coisas que ficarão imediatamente aparentes para o usuário. O teste de validação começa quando termina o teste de integração, quando os componentes individuais já foram exercitados, o *software* está completamente montado como um pacote e os erros de interface já foram descobertos e corrigidos. No nível de validação ou de sistema, a distinção entre diferentes categorias de *software*

desaparece. O teste focaliza ações visíveis ao usuário e saídas do sistema reconhecíveis pelo usuário.

A validação pode ser definida de várias maneiras, mas uma definição simples (embora rigorosa) é que a validação tem sucesso quando o *software* funciona de uma maneira que pode ser razoavelmente esperada pelo cliente. Nesse ponto, um desenvolvedor de *software* veterano pode protestar: "Quem ou o que é o árbitro para decidir o que são expectativas razoáveis?". Se uma *especificação de requisitos de software* foi desenvolvida, ela descreve cada história de usuário, todos os atributos do *software* visíveis ao usuário e os critérios de aceitação do cliente para cada um. Os critérios de aceitação do usuário formam a base para uma abordagem de teste de validação.

A validação de *software* é conseguida por meio de uma série de testes que demonstram conformidade com os requisitos. Um plano de teste descreve as classes de testes a serem realizados, e um procedimento de teste define casos de teste específicos destinados a garantir que todos os requisitos funcionais sejam satisfeitos, todas as características comportamentais sejam obtidas, todo o conteúdo seja preciso e adequadamente apresentado, todos os requisitos de desempenho sejam atendidos, a documentação esteja correta e a usabilidade e outros requisitos sejam cumpridos (p. ex., transportabilidade, compatibilidade, recuperação de erro, facilidade de manutenção). Se for descoberto um desvio em relação à especificação, é criada uma *lista de deficiências*. Deve ser estabelecido um método para solucionar deficiências (aceitável para os envolvidos). Os métodos de teste especializados para esses requisitos não funcionais são discutidos no Capítulo 21.

Um elemento importante do processo de validação é a *revisão da configuração*. A finalidade da revisão é garantir que todos os elementos da configuração do *software* tenham sido adequadamente desenvolvidos, estejam catalogados e tenham os detalhes necessários para amparar as atividades de suporte. A revisão de configuração, também chamada de auditoria, é discutida em mais detalhes no Capítulo 22.

Casa segura

Preparando-se para a validação

Cena: Escritório de Doug Miller, onde continua o projeto no nível de componente e a criação de certos componentes.

Atores: Doug Miller, gerente de engenharia de *software*; Vinod, Jamie, Ed e Shakira – membros da equipe de engenharia de *software* do *CasaSegura*.

Conversa:

Doug: O primeiro incremento estará pronto para validação em... cerca de três semanas?

Vinod: Certo. A integração está indo bem. Estamos fazendo o teste fumaça diariamente, encontrando alguns erros, mas nada que não se possa resolver. Até agora, tudo bem.

Doug: Fale sobre a validação.

Shakira: Bem, usaremos como base para nosso projeto de teste todos os casos de uso. Ainda não comecei, mas desenvolveremos testes para todos os casos de uso pelos quais sou responsável.

Ed: A mesma coisa aqui.

Jamie: Eu também, mas temos de juntar as nossas ações para teste de aceitação e para testes alfa e beta, não?

Doug: Sim. Na verdade, estive pensando que poderíamos contratar alguém para nos ajudar na validação. Tenho verba

disponível no orçamento... e teríamos um novo ponto de vista.

Vinod: Eu acho que temos tudo sob controle.

Doug: Estou certo de que sim, mas um ITG nos dará uma visão independente sobre o *software*.

Jamie: Estamos com o tempo apertado aqui, Doug. Não temos tempo suficiente para pajear qualquer um que você trouxer aqui.

Doug: Eu sei, eu sei. Mas se um ITG trabalhar a partir dos requisitos e casos de uso, não será necessário muito acompanhamento.

Vinod: Eu ainda acho que temos tudo sob controle.

Doug: Eu sei, Vinod, mas vou insistir nisso. Vamos marcar uma reunião com o representante do ITG ainda esta semana. Vamos dar o pontapé inicial e ver até onde eles chegam.

Vinod: Ok, talvez eles possam aliviar a carga.

No nível de validação ou de sistema, os detalhes das conexões de classes desaparecem. A validação de *software* orientado a objetos enfoca as ações visíveis pelo usuário e as saídas do sistema reconhecíveis por ele. Para ajudar na criação de testes de validação, o testador deve fundamentar-se nos casos de uso (Capítulos 7 e 8) 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 19) 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 criado como parte da análise orientada a objetos.

20.6 Padrões de teste

O uso de padrões como um mecanismo para descrever soluções para problemas de projeto específicos foi discutido no Capítulo 15. Mas os padrões também podem ser usados para propor soluções para outras situações de engenharia de *software* – nesse caso, o teste de *software*. Os *padrões de teste* descrevem problemas comuns de teste e soluções que podem ajudar a lidar com eles.

Grande parte do teste de *software*, inclusive durante a década passada, tem sido uma atividade improvisada. Se os padrões de teste podem ajudar uma equipe de *software* a se comunicar sobre testes de forma mais eficaz, a entender as forças motivadoras que conduzem a uma abordagem específica para o teste e a encarar o projeto de testes como uma atividade evolucionária, na qual cada iteração resulta em um conjunto mais completo de casos de teste, então os padrões realmente dão uma grande contribuição.

Os padrões de teste são descritos de maneira muito semelhante aos padrões de projeto (Capítulo 15). Já foram propostas dezenas de padrões de teste na literatura (p. ex., [Mar02]). Os três padrões a seguir (apresentados apenas de uma forma superficial) fornecem exemplos representativos:

Nome do padrão: **testeEmDupla**

Resumo: Um padrão orientado a processo, o **teste em dupla** descreve uma técnica que é análoga à programação em pares (Capítulo 3), na qual dois testadores trabalham em conjunto para projetar e executar uma série de testes que podem ser aplicados a atividades de teste de unidade, integração ou validação.

Nome do padrão: **interfaceDeTesteSeparada**

Resumo: Há necessidade de testar todas as classes em um sistema orientado a objetos, incluindo "classes internas" (i.e., classes que não expõem qualquer interface fora do

componente que as utilizou). O padrão **interfaceDeTesteSeparada** descreve como criar "uma interface de teste que pode ser usada para descrever testes específicos em classes que são visíveis somente internamente a um componente" [Lan01].

Nome do padrão: **testeDeCenário**

Resumo: Uma vez feitos os testes de unidade e de integração, é necessário determinar se o *software* se comportará ou não de maneira que satisfaça aos usuários. O padrão **testeDeCenário** descreve uma técnica para exercitar o *software* do ponto de vista do usuário. Uma falha nesse nível indica que o *software* deixou de atender aos requisitos visíveis para o usuário [Kan01].

Uma discussão abrangente sobre os padrões de teste está fora dos objetivos deste livro. Se você estiver interessado em mais detalhes, consulte [Bin99], [Mar02], [Tho04], [Mac10] e [Gon17] para informações adicionais sobre esse importante assunto.

20.7 Resumo

O teste de integração constrói a arquitetura de *software* ao mesmo tempo em que se realizam testes para descobrir erros associados às interfaces. O objetivo é construir uma estrutura de programa determinada pelo projeto a partir de componentes testados em unidade.

Desenvolvedores de *software* experientes muitas vezes afirmam: "O teste nunca termina, ele apenas se transfere de você (o engenheiro de *software*) para o seu cliente. Toda vez que um cliente usa o programa, um teste está sendo realizado". Aplicando o projeto de caso de teste, você pode obter um teste mais completo e, assim, descobrir e corrigir o maior número de erros antes que comecem os "testes do cliente".

Hetzel [Het84] descreve o teste caixa-branca como "teste no pequeno" (para o que é particular). Sua implicação é que os testes caixa-branca que foram considerados neste capítulo são aplicados tipicamente a pequenos componentes de programas (p. ex., módulos ou pequenos grupos de módulos). O teste caixa-preta, por outro lado, amplia o seu foco e pode ser chamado de "teste no grande" (para o que é amplo).

O teste caixa-preta se baseia em requisitos especificados nas histórias de usuário ou em alguma outra representação de modelagem da análise. Os autores de casos de teste não precisam esperar que o código de implementação do componente seja lido, desde que entendam os requisitos da funcionalidade sendo testada. O teste de validação muitas vezes é realizado com casos de teste caixa-preta que produzem ações de entrada visíveis para o usuário e comportamentos de saída observáveis.

O teste caixa-branca exige um exame rigoroso do detalhe procedimental e dos detalhes da implementação da estrutura de dados para os componentes sendo testados. Testes caixa-branca só podem ser projetados depois que o projeto no nível de componente (ou código-fonte) existir. Os caminhos lógicos do *software* e as colaborações entre componentes são o foco do teste de integração caixa-branca.

O teste de integração do *software* orientado a objetos pode ser feito com uma estratégia baseada em sequências de execução ou baseada em uso. O teste baseado em sequências de execução integra o conjunto de classes que colaboram para responder a uma entrada ou um evento. O teste baseado em uso constrói o sistema em camadas, começando com as classes que não fazem uso de classes servidoras. Métodos de projeto de caso de teste de integração também podem 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 suas colaboradoras. Uma sequência de teste acompanha o fluxo de operações por meio das colaborações de classe.

Teste de validação orientado a objetos é um teste orientado a caixa-preta e pode ser realizado aplicando-se os mesmos métodos caixa-preta discutidos para *software* convencional. Entretanto, o teste baseado em cenários domina a validação de sistemas orientados a objeto, tornando o caso de uso um pseudocontrolador primário para teste de validação.

O teste de regressão é o processo de reexecutar um determinado caso de teste após qualquer alteração a um sistema de *software*. Os testes de regressão devem ser executados sempre que novos componentes ou alterações são adicionados a um incremento de *software*. O teste de regressão ajuda a garantir que as alterações não introduzam comportamento indesejado ou erros adicionais.

Os padrões de teste descrevem problemas comuns de teste e soluções que podem ajudar a lidar com eles. Se os padrões de teste podem ajudar uma equipe de *software* a se comunicar sobre testes de forma mais eficaz, a entender as forças motivadoras que conduzem a uma abordagem específica para o teste e a encarar o projeto de testes como uma atividade evolucionária, na qual cada iteração resulta em um conjunto mais completo de casos de teste, então os padrões realmente dão uma grande contribuição.

Problemas e pontos a ponderar

20.1 Como o cronograma de projeto pode afetar o teste de integração?

20.2 Quem deve executar o teste de validação – o desenvolvedor do *software* ou o usuário do *software*? Justifique a sua resposta.

20.3 Poderá o teste exaustivo (mesmo que seja possível para programas muito pequenos) garantir que um programa está 100% correto?

20.4 Por que o “teste” deve começar com análise e projeto de requisitos?

20.5 Os requisitos não funcionais (p. ex., segurança e desempenho) deveriam ser parte dos testes de integração?

20.6 Por que temos de testar novamente as subclasses instanciadas de uma classe existente se a classe existente já foi completamente testada?

20.7 Qual a diferença entre estratégias baseadas em sequências de execução e estratégias baseadas em uso para teste de integração?

20.8 Desenvolva uma estratégia de teste completa para o sistema *CasaSegura* discutido anteriormente neste livro. Documente-a em uma *especificação de teste.*

20.9 Escolha uma das histórias de usuário do sistema *CasaSegura* para embasar os testes baseados em cenários e construa um conjunto de casos de testes de integração necessários para realizar tais testes para a história de usuário escolhida.

20.10 Para os casos de teste elaborados no Problema 20.9, identifique um subconjunto de casos de teste que você usará para o teste de regressão de componentes de *software* adicionados ao programa.

---

1 Os termos *teste funcional* e *teste estrutural* às vezes são usados em lugar de teste caixa-preta e teste caixa-branca, respectivamente.

2 Consulte o *site* deste livro (em inglês).

3 As Seções 20.4.1 e 20.4.2 foram adaptadas de um artigo de Brian Marick originalmente postado na Internet no grupo 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 20.4.1 e 20.4.2 também são aplicáveis para *software* convencional.