de combustível. O motorista pode abastecer com o combustível necessário. Quando o abastecimento estiver concluído e a mangueira da bomba for devolvida ao suporte, a conta do cartão de crédito do motorista é debitada com o custo do combustível. O cartão de crédito é devolvido após ser debitado. Se o cartão for inválido, a bomba o devolve antes de liberar a saída do combustível.

7.7 Desenhe um diagrama de sequência mostrando as interações dos objetos em um sistema de diário de grupo quando um grupo de pessoas está marcando uma reunião.

7.8 Desenhe um diagrama de estado da UML mostrando as possíveis mudanças de estado no diário de grupo ou no sistema do posto de abastecimento de combustível.

7.9 Usando exemplos, explique por que o gerenciamento de configuração é importante quando um time está desenvolvendo um produto de software.

7.10 Uma pequena empresa desenvolveu um produto de software especializado que ela configura especialmente para cada cliente. Os clientes novos normalmente têm necessidades específicas a serem incorporadas ao seu sistema e eles pagam para que sejam desenvolvidas e integradas ao produto. A empresa de software tem uma oportunidade para disputar um novo contrato, o que mais do que dobraria a sua base de clientes. O novo cliente deseja ter algum envolvimento na configuração do sistema. Explique porque, nessas circunstâncias, poderia ser uma boa ideia a empresa dona do software transformá-lo em código aberto.

REFERÊNCIAS

ABBOTT, R. Program design by informal english descriptions. *Comm. ACM*, v. 26, n. 11, 1983. p. 882--894. doi:10.1145/182.358441.

ALEXANDER, C. *A Timeless way of building*. Oxford, UK: Oxford University Press. 1979.

BAYERSDORFER, M. Managing a project with open source components. *ACM Interactions*, v. 14, n. 6, 2007. p. 33-34. doi: 10.1145/1300655.1300677.

BECK, K.; CUNNINGHAM, W. A laboratory for teaching object-oriented thinking. In: *Proc. OOPSLA'89 (Conference on Object-Oriented Programming, Systems, Languages and Applications)*, 1-6. ACM Press. 1989. doi:10.1145/74878.74879.

BUSCHMANN, F.; HENNEY, K.; SCHMIDT, D. C. *Pattern-oriented software architecture volume 4*: a pattern language for distributed computing. New York: John Wiley & Sons. 2007a.

_____. *Pattern-oriented software architecture volume 5*: on patterns and pattern languages. New York: John Wiley & Sons. 2007b.

BUSCHMANN, F.; MEUNIER, R.; ROHNERT, H.; SOMMERLAD, P. *Pattern-oriented software architecture*: a system of patterns, v. 1. New York: John Wiley & Sons, 1996.

CHAPMAN, C. A short guide to open-source and similar licences. *Smashing Magazine*, 24 mar. 2010. Disponível em: <http://www.smashingmagazine.com/2010/03/24/a-short-guide-to-open-source-and-similar-licenses/>. Acesso em: 22 abr. 2018.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. *Design patterns*: elements of reusable object-oriented software. Reading, MA.: Addison-Wesley, 1995.

KIRCHER, M.; JAIN, P. *Pattern-oriented software architecture volume 3*: patterns for resource management. New York: John Wiley & Sons, 2004.

LOELIGER, J.; MCCULLOUGH, M. *Version control with Git*: powerful tools and techniques for collaborative software development. Sebastopol, CA: O'Reilly & Associates, 2012.

PILATO, C.; COLLINS-SUSSMAN, B.; FITZPATRICK, B. *Version control with Subversion*. Sebastopol, CA: O'Reilly & Associates, 2008.

RAYMOND, E. S. *The cathedral and the bazaar*: musings on Linux and open source by an accidental revolutionary. Sebastopol. CA: O'Reilly & Associates, 2001.

SCHMIDT, D.; STAL, M.; ROHNERT, H.; BUSCHMANN, F. *Pattern-oriented software architecture volume 2*: patterns for concurrent and networked objects. New York: John Wiley & Sons, 2000.

ST. LAURENT, A. *Understanding open source and free software licensing*. Sebastopol, CA: O'Reilly & Associates, 2004.

VOGEL, L. *Eclipse IDE*: a tutorial. Hamburg, Germany: Vogella Gmbh, 2013.

WIRFS-BROCK, R.; WILKERSON, B.; WEINER, L. *Designing object-oriented software*. Englewood Cliffs, NJ: Prentice-Hall, 1990.

8 Teste de software

OBJETIVOS

O objetivo deste capítulo é introduzir o teste de software e seus processos. Ao ler este capítulo, você:

- compreenderá os estágios de teste, desde o desenvolvimento até a aceitação pelos clientes do sistema;
- será apresentado às técnicas que ajudam a escolher os casos de teste concebidos para descobrir defeitos de programação;
- compreenderá o desenvolvimento com testes *a priori* (*test-first*), em que os testes são projetados antes da escrita do código e executados automaticamente;
- conhecerá três tipos de testes diferentes — teste de componentes, teste de sistemas e teste de lançamento (*release*);
- compreenderá as diferenças entre teste de desenvolvimento e teste de usuário.

CONTEÚDO

8.1 Teste de desenvolvimento
8.2 Desenvolvimento dirigido por testes
8.3 Teste de lançamento
8.4 Teste de usuário

Os testes pretendem mostrar que um programa faz o que foi destinado a fazer e descobrir defeitos antes que ele seja colocado em uso. Em um teste de software, um programa é executado com uso de dados artificiais, e os resultados são conferidos em busca de erros, anomalias ou informações sobre os atributos não funcionais do programa.

Quem testa um software, tenta fazer duas coisas:

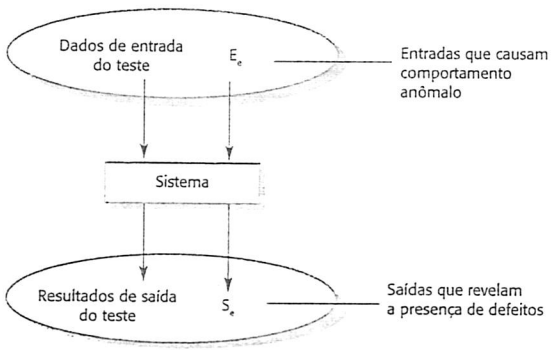
1. Demonstrar ao desenvolvedor e ao cliente que o software atende aos seus requisitos. No caso de software customizado, isso significa que deve haver pelo menos um teste para cada requisito no documento de requisitos. No caso de produtos de software genéricos, isso significa que deve haver testes para todas as características do sistema que serão incluídas no lançamento do produto. Também podem ser testadas combinações de características para averiguar a existência de interações indesejadas entre elas.

2. Encontrar entradas ou sequências de entradas nas quais o software se comporta de modo incorreto, indesejável ou fora da conformidade de suas especificações. Esses fatores são causados por defeitos (*bugs*), e o teste de software busca encontrá-los a fim de erradicar comportamentos indesejáveis, como falhas de sistema, interações indesejadas com outros sistemas, processamentos incorretos e dados corrompidos.

O primeiro caso é o teste de validação, no qual se espera que o sistema funcione corretamente por meio do uso de um conjunto de casos de teste que reflita o uso esperado do sistema. O segundo é o teste de defeitos, no qual os casos de teste são concebidos para expor defeitos. Nesse teste, os casos podem ser pouco claros de forma deliberada, e não precisam refletir a maneira como o sistema normalmente é utilizado. Naturalmente, não há um limite exato entre essas duas abordagens. Durante o teste de validação, serão encontrados defeitos no sistema; durante o teste de defeitos, alguns deles mostrarão que o programa atende aos seus requisitos.

A Figura 8.1 mostra as diferenças entre o teste de validação e o teste de defeitos. O sistema testado deve ser pensado como uma caixa-preta. Ele aceita entradas de algum conjunto de entradas E e gera saídas em um conjunto de saídas S. Algumas saídas estarão erradas – as saídas no conjunto S_e – e serão geradas pelo sistema em resposta às entradas no conjunto E_e. A prioridade no teste de defeitos é encontrar essas entradas no conjunto E_e, pois elas revelam problemas com o sistema. O teste de validação envolve testar com entradas corretas que estão fora do conjunto E_e e que estimulam o sistema a gerar as saídas corretas esperadas.

FIGURA 8.1 Modelo de entrada e saída de teste de programa.



Os testes não conseguem demonstrar que o software está livre de defeitos ou que vai se comportar de acordo com a sua especificação em qualquer circunstância. Sempre é possível que um teste negligenciado descubra mais problemas com o sistema. Como declarou de forma eloquente Edsger Dijkstra (1972), um dos primeiros colaboradores no desenvolvimento da engenharia de software:

> *"O teste só consegue mostrar a presença de erros, não a sua ausência."*[1]

1 DIJKSTRA, E. W. "The humble programmer." *Comm. ACM*, v. 15, n. 10, 1972, p. 859-66. doi:10.1145/355604.361591.

Os testes fazem parte de um processo mais amplo de verificação e validação de software (V & V). A verificação e a validação não são a mesma coisa, embora sejam frequentemente confundidas. Barry Boehm, um pioneiro da engenharia de software, expressou de forma sucinta a diferença entre elas (BOEHM, 1979):

- **Validação**: Estamos construindo o produto certo?
- **Verificação**: Estamos construindo o produto corretamente?

Os processos de verificação e validação estão preocupados em conferir se o software que está sendo desenvolvido cumpre sua especificação e fornece a funcionalidade esperada pelas pessoas que estão pagando por ele. Esses processos de conferência começam logo que os requisitos são disponibilizados e continuam por todos os estágios do processo de desenvolvimento.

A verificação de software é o processo de conferir se o software cumpre seus requisitos funcionais e não funcionais declarados. A validação de software é um processo mais geral, cujo objetivo é assegurar que o software atenda às expectativas do cliente, e vai além da conferência da conformidade com a especificação, para demonstrar que o software faz o que se espera dele; ela é essencial porque, conforme discutimos no Capítulo 4, as declarações de requisitos nem sempre refletem os desejos ou necessidades reais dos clientes e usuários do sistema.

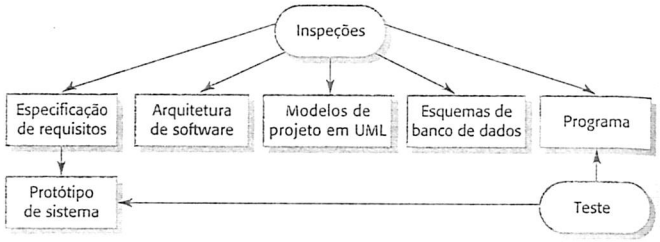
O objetivo dos processos de verificação e validação é estabelecer a confiança de que o sistema de software é 'adequado para a finalidade'. Isso significa que o sistema deve ser bom o bastante para o uso que se pretende fazer dele. O nível de confiança necessária depende da finalidade do sistema, das expectativas dos usuários e do ambiente de mercado real:

1. *Finalidade do software.* Quanto mais crítico o software, mais importante é a sua confiabilidade. Por exemplo, o nível de confiança necessário de um software utilizado para controlar um sistema crítico em segurança é muito maior do que o de um sistema de demonstração, que age como um protótipo das ideias de novos produtos.
2. *Expectativas de usuários.* Devido às experiências anteriores com software defeituoso e pouco confiável, alguns usuários têm baixas expectativas quanto à qualidade do produto adquirido e não se surpreendem com uma falha. Quando um novo sistema é instalado, os usuários podem tolerar falhas porque os benefícios do seu uso superam os custos de recuperação das falhas. No entanto, à medida que o produto de software se estabelece, os usuários esperam que ele se torne mais confiável. Consequentemente, pode ser necessário um teste mais completo das últimas versões do sistema.
3. *Ambiente de mercado.* Quando uma empresa de software coloca um sistema no mercado, ela deve levar em conta os produtos concorrentes, o preço que os clientes estão dispostos a pagar por ele e o cronograma necessário para entregá-lo. Em um ambiente competitivo, a empresa pode resolver lançar um programa antes de ele ter sido totalmente testado e depurado, pois deseja ser a primeira no mercado. Se um produto de software ou aplicativo for barato, os usuários podem se dispor a tolerar um nível mais baixo de confiabilidade.

Assim como envolvem o teste de software, os processos de verificação e validação podem envolver inspeções e revisões de software, a fim de analisar e conferir os requisitos do sistema, os modelos de projeto, o código-fonte do programa e até mesmo os testes

de sistema propostos. Essas são técnicas estáticas de V & V, nas quais não é necessário executar o software para verificá-lo. A Figura 8.2 mostra que as inspeções e o teste de software apoiam a V & V em diferentes estágios do processo de software. As setas indicam os estágios do processo em que as técnicas podem ser utilizadas.

FIGURA 8.2 Inspeções e teste.



As inspeções se concentram principalmente no código-fonte do sistema, mas qualquer representação legível do software, como seus requisitos ou um modelo de projeto, pode ser inspecionada. Em uma inspeção de sistema para descobrir erros, utiliza-se o conhecimento do sistema, seu domínio de aplicação e a linguagem de programação ou de modelagem.

A inspeção de software tem três vantagens em relação ao teste:

1. Durante o teste, alguns erros podem mascarar (ocultar) outros. Quando um erro leva a saídas inesperadas, não é possível saber se as anomalias de saídas posteriores se devem a um novo erro ou a efeitos colaterais do erro original. Como a inspeção não envolve a execução do sistema, não é preciso se preocupar com as interações entre os erros. Consequentemente, uma única sessão de inspeção pode descobrir muitos erros em um sistema.
2. Versões incompletas de um sistema podem ser inspecionadas sem custos adicionais. Se um programa estiver incompleto, então é preciso desenvolver trechos de testes especializados para as partes disponíveis. Obviamente, isso aumenta os custos de desenvolvimento do sistema.
3. Assim como a busca por defeitos de programação, uma inspeção também pode considerar atributos mais amplos da qualidade de um programa, como a conformidade com padrões, a portabilidade e a manutenibilidade. É possível procurar por ineficiências, algoritmos inadequados e mau estilo de programação que poderiam tornar o sistema difícil de manter e atualizar.

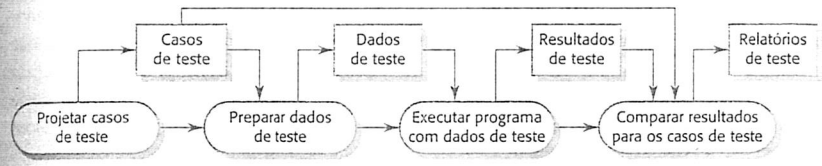
As inspeções de programa são uma ideia antiga, e vários estudos e experimentos mostraram que as inspeções são mais eficazes na descoberta de defeitos do que o teste de programa. Fagan (1976) relatou que mais de 60% dos erros em um programa podem ser detectados usando inspeções informais do programa. No processo *Cleanroom* (PROWELL *et al.*, 1999), reivindica-se que mais de 90% dos defeitos podem ser descobertos nas inspeções dos programas.

No entanto, essas inspeções não podem substituir o teste de software, uma vez que elas não são boas para descobrir defeitos provenientes de interações inesperadas entre as diferentes partes de um programa, problemas de temporização ou problemas com o desempenho do sistema. Nas empresas pequenas ou nos grupos de desenvolvimento, pode ser difícil e caro reunir uma equipe de inspeção à parte, já que todos os possíveis membros também podem ser desenvolvedores do software.

Discutirei as revisões e inspeções com mais profundidade no Capítulo 24 (Gerenciamento da qualidade). A análise estática, em que o código-fonte de um programa é analisado automaticamente para descobrir anomalias, é explicada no Capítulo 12. Neste capítulo, me concentrarei no teste e nos processos de teste.

A Figura 8.3 é um modelo abstrato do processo de teste tradicional, conforme utilizado no desenvolvimento dirigido por plano. Os casos de teste são especificações das entradas para o teste e das saídas esperadas do sistema (os resultados do teste), além de uma declaração do que está sendo testado. Os dados de teste são as entradas criadas para testar um sistema. Às vezes, eles podem ser criados automaticamente, mas a geração automática dos casos de teste é impossível. As pessoas que compreendem o que o sistema se destina a fazer devem se envolver na especificação dos resultados esperados do teste. No entanto, a execução do teste pode ser automatizada. Os resultados são comparados automaticamente com os resultados previstos, então não é necessário que uma pessoa procure erros e anomalias durante a execução do teste.

FIGURA 8.3 Modelo do processo de teste de software.



Geralmente, um sistema de software comercial passa por três estágios de teste:

1. *Teste de desenvolvimento*, em que o sistema é testado durante o desenvolvimento para descobrir defeitos. Os projetistas de sistema e programadores tendem a se envolver no processo de teste.
2. *Teste de lançamento (*release*)*, em que uma equipe de testes separada testa uma versão completa do sistema antes de ela ser entregue aos usuários. O objetivo do teste de lançamento é conferir se o sistema cumpre os requisitos dos *stakeholders*.
3. *Teste de usuário*, em que os usuários ou potenciais usuários de um sistema testam-no em seu próprio ambiente. Nos produtos de software, o 'usuário' pode ser um grupo de marketing interno que irá decidir se o software pode ser anunciado, lançado e vendido. O teste de aceitação é um tipo de teste de usuário, em que o cliente testa formalmente um sistema para decidir se deveria aceitá-lo do fornecedor ou se mais desenvolvimento é necessário.

Planejamento do teste

O planejamento do teste se preocupa com o cronograma e a definição dos recursos para todas as atividades no processo de teste, levando em conta desde a definição do processo até as pessoas envolvidas e o tempo disponíveis. Normalmente, é criado um plano de teste, definindo o que deve ser testado, o cronograma de teste previsto e como os testes serão registrados. Nos sistemas críticos, o plano de teste também pode incluir detalhes dos testes a serem realizados no software.

Na prática, o processo de teste geralmente envolve uma mistura de testes manuais e automatizados. No teste manual, um testador executa o programa com alguns dados de teste e compara os resultados com as suas expectativas. Ele anota e informa as discrepâncias para os programadores. Os testes automatizados são codificados em um programa executado cada vez que o sistema em desenvolvimento tiver que ser testado. Isso é mais rápido do que o teste manual, especialmente quando envolve testes de regressão — reexecutar testes anteriores para verificar se as mudanças no programa não introduziram novos defeitos.

Infelizmente, o teste nunca pode ser completamente automatizado, já que os testes automatizados só conseguem checar se um programa faz o que se propõe. É praticamente impossível usar testes automatizados para sistemas que dependem de como as coisas se parecem (por exemplo, uma interface gráfica com o usuário) ou para testar se um programa não tem efeitos colaterais imprevistos.

8.1 TESTE DE DESENVOLVIMENTO

O teste de desenvolvimento inclui todas as atividades de teste executadas pelo time responsável pelo sistema. O testador do software normalmente é o programador que o desenvolveu. Alguns processos de desenvolvimento usam pares programador-testador (CUSUMANO; SELBY, 1998): cada programador tem um testador associado que desenvolve os testes e o auxilia nesse processo. Em sistemas críticos, um processo mais formal pode ser utilizado, com um grupo de teste separado dentro do time de desenvolvimento, responsável por desenvolver os testes e manter registros detalhados dos seus resultados.

Existem três estágios do teste de desenvolvimento:

1. *Teste de unidade*, em que são testadas unidades de programa ou classes individuais. Esse tipo de teste deve se concentrar em testar a funcionalidade dos objetos e seus métodos.
2. *Teste de componentes*, em que várias unidades são integradas, criando componentes compostos. Esse teste deve se concentrar em testar as interfaces dos componentes que promovem acesso às suas funções.
3. *Teste de sistema*, em que alguns ou todos os componentes em um sistema são integrados e o sistema é testado como um todo. O teste de sistema deve se concentrar em testar as interações dos componentes.

O teste de desenvolvimento é basicamente um processo de teste dos defeitos, cujo objetivo é descobrir *bugs* do software. Portanto, normalmente ele é intercalado com a depuração — processo de localizar problemas no código e alterar o programa para consertá-los.

Depuração

Depuração é o processo de consertar erros e problemas que foram descobertos por meio de testes. Usando as informações dos testes de programa, os depuradores aplicam seu conhecimento de linguagem de programação e o resultado esperado do teste para localizar e consertar o erro do programa. Normalmente, a depuração de um programa usa ferramentas interativas que fornecem informações extras sobre a execução do programa.

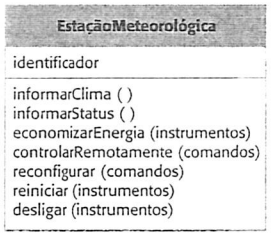
8.1.1 Teste de unidade

O teste de unidade é o processo de testar componentes de programa, como os métodos ou as classes. As funções individuais ou métodos são o tipo de componente mais simples. Seus testes devem consistir em chamadas para essas rotinas com diferentes parâmetros de entrada. É possível usar as abordagens para projeto de caso de teste, discutidas na Seção 8.1.2, para projetar os testes de função ou método.

Ao testar classes, deve-se projetar seus testes para proporcionar a cobertura de todos as características do objeto. Todas as operações associadas ao objeto devem ser testadas; os valores de todos os atributos associados ao objeto devem ser definidos e conferidos; e o objeto deve ser colocado em todos os estados possíveis. Isso quer dizer que todos os eventos que provocam uma mudança de estado devem ser simulados.

Considere, por exemplo, o objeto EstaçãoMeteorológica do exemplo que discuti no Capítulo 7. Os atributos e operações desse objeto estão exibidos na Figura 8.4.

FIGURA 8.4 Interface do objeto EstaçãoMeteorológica.



Ele tem um único atributo, o seu identificador, que é uma constante definida quando a estação meteorológica é instalada. Portanto, ele só precisa de um teste que verifique se a constante foi configurada corretamente. É necessário definir os casos de teste para todos os métodos associados ao objeto, como informarClima e informarStatus. Em condições ideais, os métodos devem ser testados isoladamente, mas, em alguns casos, a sequência de testes é necessária. Por exemplo, para testar o método que desliga os instrumentos da estação meteorológica (desligar), é necessário ter executado o método reiniciar.

A generalização ou herança complica o teste de classe. Não se pode simplesmente testar uma operação na classe em que ela é definida e supor que ela funcionará conforme o previsto em todas as subclasses que herdam a operação. A operação herdada pode levar a suposições a respeito de outras operações e atributos, que podem não ser válidas em algumas subclasses que herdam a operação. Portanto, deve-se testar a operação herdada em todos os lugares em que ela for utilizada.

Para testar os estados da estação meteorológica, dá para usar um modelo de máquina de estados, conforme discutido no Capítulo 7 (Figura 7.8). Com ele, é possível identificar sequências de transições de estado que precisam ser testadas e definir as sequências de eventos para forçar essas transições. A princípio, deve ser testada toda sequência possível de transição de estado, embora, na prática, isso possa ser caro demais. Os exemplos de sequências de estado que devem ser testadas na estação meteorológica incluem:

Desligado → Executando → Desligado
Configurando → Executando → Testando → Transmitindo → Executando
Executando → Coletando → Executando → Resumindo → Transmitindo → Executando

Sempre que possível, o teste de unidade deve ser automatizado; um *framework* de automação de teste, como o JUnit (TAHCHIEV *et al.*, 2010), pode ser usado para escrever e executar testes do programa. Os *frameworks* de teste de unidade fornecem classes de teste genéricas que podem ser estendidas para criar casos de teste específicos. Depois, eles podem executar todos os testes que foram implementados e informar, geralmente por meio de alguma interface gráfica com o usuário (GUI, do inglês *graphical user interface*), o sucesso ou não dos testes. Uma série inteira de testes pode ser executada em poucos segundos, então é possível executar todos os testes toda vez que você modificar o programa.

Um teste automatizado tem três partes:

1. *uma parte de configuração*, em que o sistema é iniciado com o caso de teste, ou seja, as entradas e as saídas esperadas;
2. *uma parte de chamada*, em que se chama o objeto ou o método a ser testado;
3. *uma parte de asserção*, em que o resultado da chamada é comparado com o resultado previsto. Caso o resultado da asserção seja verdadeiro, o teste foi bem-sucedido; se for falso, então ele fracassou.

Às vezes, o objeto que está sendo testado depende de outros, que podem não ter sido implementados ou cujo uso desacelera o processo de teste. Por exemplo, se um objeto chama um banco de dados, isso pode envolver um processo de configuração lento antes que o banco possa ser utilizado. Nesses casos, é possível usar *mock objects*.

Mock objects são objetos com a mesma interface dos objetos externos que estão sendo utilizados, simulando sua funcionalidade. Por exemplo, um objeto que simula um banco de dados pode ter apenas alguns itens de dados organizados em um vetor, que poderão ser acessados rapidamente, sem a sobrecarga de chamar um banco de dados e acessar os discos. De modo similar, *mock objects* podem ser utilizados para simular operações anormais ou eventos raros. Por exemplo, se um sistema foi feito para agir em determinados horários do dia, seu *mock object* pode simplesmente retornar esses horários, independentemente do horário real do relógio.

8.1.2 Escolhendo casos de teste de unidade

Testar é caro e demorado, então é importante escolher casos de teste de unidade eficazes. A eficácia, neste caso, significa duas coisas:

1. os casos de teste devem mostrar que, quando utilizado conforme o esperado, o componente que está sendo testado faz o que deveria fazer;
2. se houver defeitos no componente, eles devem ser revelados pelos casos de teste.

Portanto, devem ser projetados dois tipos de casos de teste. O primeiro deles deve refletir a operação normal de um programa e mostrar que o componente funciona. Por exemplo, ao testar um componente que cria e inicializa um novo registro de paciente, o caso de teste deve mostrar que o registro existe em um banco de dados e que seus campos foram definidos conforme especificado. O outro tipo de caso de

teste deve se basear na experiência de teste de onde surgem os problemas comuns. Ele deve usar entradas anormais para verificar se elas são corretamente processadas e não provocam a falha do componente.

Duas estratégias que podem ser eficazes para ajudar na escolha dos casos de teste são:

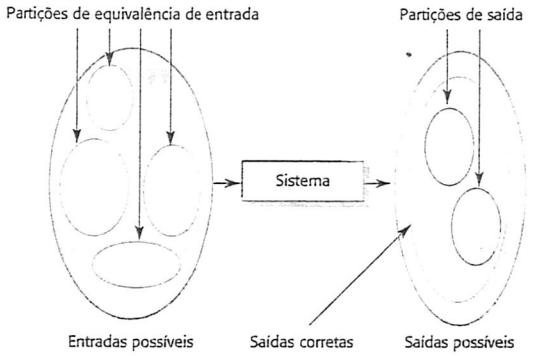
1. *Teste de partição.* São identificados grupos de entradas com características comuns que devem ser processadas da mesma maneira. Deve-se escolher testes de dentro de cada um desses grupos.
2. *Teste baseado em diretriz.* Nesse tipo de teste, diretrizes que refletem a experiência prévia com os tipos de erros que os programadores costumam cometer quando desenvolvem componentes são usadas para escolher os casos de teste.

Os dados de entrada e os resultados de saída de um programa podem ser encarados como membros de conjuntos com características comuns. Como exemplo desses conjuntos, temos os números positivos, números negativos e seleções de menu. Os programas normalmente se comportam de maneira comparável para todos os membros de um conjunto, ou seja, quando se testa um programa que faz um cálculo e requer dois números positivos, espera-se que o programa se comporte da mesma maneira com todos os números positivos.

Em virtude desse comportamento equivalente, às vezes essas classes são chamadas de partições ou domínios de equivalência (BEZIER, 1990). Uma abordagem sistemática para o projeto de caso de teste se baseia na identificação de todas as partições de entrada e saída de um sistema ou componente. Os casos de teste são projetados para que as entradas e saídas recaiam nessas partições. O teste de partição pode ser usado para conceber casos de teste tanto para sistemas quanto para componentes.

Na Figura 8.5, a grande elipse sombreada à esquerda representa o conjunto de todas as entradas possíveis para o programa que está sendo testado. As elipses menores não sombreadas representam partições de equivalência. Um programa sendo testado deve processar da mesma forma todos os membros de uma partição de equivalência de entrada.

FIGURA 8.5 Particionamento de equivalência.



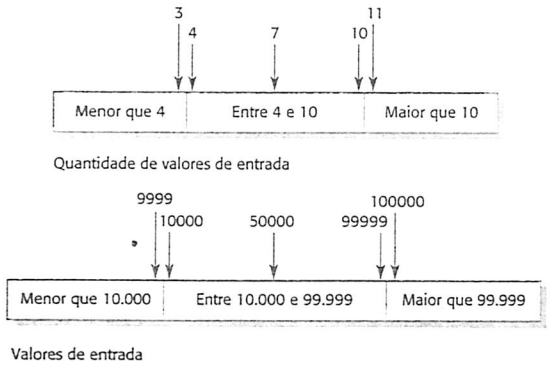
As partições de equivalência de saída são aquelas em que todas as saídas têm algo em comum. Às vezes há uma correspondência de 1:1 entre as partições de

equivalência de entrada e de saída. No entanto, nem sempre é isso que acontece: pode ser necessário definir uma partição de equivalência de entrada separada, em que a única característica comum é que as entradas geram saídas dentro de uma mesma partição de saída. A área sombreada na elipse da esquerda representa as entradas inválidas. A área sombreada na elipse da direita representa as exceções que podem ocorrer, ou seja, as respostas para as entradas inválidas.

Assim que for identificado um conjunto de partições, os casos de teste devem ser escolhidos a partir de cada uma delas. Uma boa regra prática para a escolha dos casos de teste é escolhê-los nos limites das partições, além de casos próximos do ponto médio da partição. A razão para isso é que os projetistas e programadores tendem a considerar valores típicos das entradas quando desenvolvem um sistema. É possível testar esses valores ao escolher o ponto médio da partição. Os valores limítrofes frequentemente são atípicos (por exemplo, zero pode se comportar de maneira diferente dos outros números não negativos) e, por isso, em alguns casos, são negligenciados pelos desenvolvedores. As falhas de programa ocorrem frequentemente quando esses valores atípicos são processados.

As partições podem ser identificadas usando a especificação do programa ou a documentação do usuário e a própria experiência, o que permite prever as classes de valores de entrada que tendem a detectar erros. Por exemplo, digamos que uma especificação diga que o programa aceita de quatro a dez entradas que consistem de inteiros de cinco dígitos maiores que 10.000. Essa informação é usada para identificar as partições de entrada e os possíveis valores de entrada do teste. Esses valores são exibidos na Figura 8.6.

FIGURA 8.6 Partições de equivalência.



Quando a especificação de um sistema é usada para identificar partições de equivalência, isso se chama teste caixa-preta. Nenhum conhecimento a respeito do funcionamento do sistema é necessário. Às vezes, é útil suplementar os testes caixa-preta com 'testes caixa-branca', nos quais o código do programa é examinado para encontrar outros testes possíveis. Por exemplo, o código pode incluir exceções para lidar com entradas incorretas. Esse conhecimento pode ser usado para identificar 'partições de exceção' — intervalos diferentes em que deve ser aplicado o mesmo tratamento de exceção.

O particionamento de equivalência é uma abordagem eficaz para os testes porque ajuda a levar em conta os erros que os programadores costumam cometer quando processam as entradas nos limites das partições. Também é possível usar diretrizes de teste para ajudar a escolher os casos de teste, as quais encapsulam o conhecimento que indica os tipos de casos de teste eficazes para descobrir erros. Ao testar programas com sequências, vetores ou listas, as diretrizes que poderiam ajudar a revelar defeitos incluem:

1. Testar o software com sequências que tenham um único valor. Programadores naturalmente pensam em sequências compostas de diversos valores e, às vezes, incorporam esse pressuposto em seus programas. Consequentemente, se uma sequência de valor único é apresentada, um programa pode não funcionar corretamente.
2. Usar sequências diferentes com tamanhos diferentes em testes diferentes. Isso diminui a chance de que um programa com defeitos venha a produzir acidentalmente uma saída correta em virtude de algumas características acidentais da entrada.
3. Derivar os testes para que os elementos do início, do meio e do fim da sequência sejam acessados. Essa abordagem revela problemas nos limites da partição.

Teste de caminho

Teste de caminho é uma estratégia que visa exercitar cada caminho de execução independente em um programa ou componente. Se todos os caminhos independentes forem executados, então todos os comandos no componente devem ter sido executados pelo menos uma vez. Todos os comandos condicionais são testados para verdadeiro ou falso. Em um processo de desenvolvimento orientado a objetos, o teste de caminho pode ser utilizado para testar os métodos associados aos objetos.

O livro de Whittaker (2009) inclui muitos exemplos de diretrizes que podem ser utilizadas no projeto de casos de teste. Algumas das diretrizes mais genéricas sugeridas por ele são:

- escolher entradas que forcem o sistema a gerar todas as mensagens de erro;
- criar entradas que provoquem *overflow* dos *buffers* de entrada;
- repetir várias vezes a mesma entrada ou série de entradas;
- forçar a geração de saídas inválidas;
- obrigar os resultados dos cálculos a serem grandes ou pequenos demais.

À medida que se ganha experiência com testes, é possível desenvolver as próprias diretrizes de como escolher casos de teste eficazes. Trago mais exemplos de diretrizes de teste nesta próxima seção.

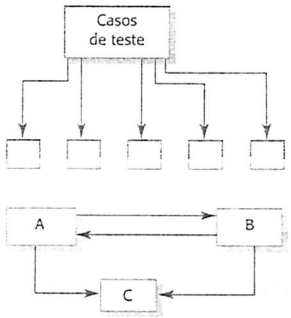
8.1.3 Teste de componentes

Os componentes de software consistem frequentemente em vários objetos que interagem entre si. Por exemplo, no sistema da estação meteorológica, o componente de reconfiguração inclui objetos que lidam com cada aspecto da reconfiguração. A funcionalidade desses objetos é acessada por meio das interfaces de componentes (ver Capítulo 7). Portanto, o teste de componentes compostos deve se concentrar

em mostrar que a interface (ou interfaces) do componente se comporta de acordo com a sua especificação. É possível supor que foram realizados testes de unidade em cada objeto dentro do componente.

A Figura 8.7 ilustra essa ideia de teste da interface do componente. Suponha que os componentes A, B e C foram integrados para criar um componente maior ou um subsistema. Os casos de teste não são aplicados aos componentes individuais, mas, sim, à interface do componente composto; erros de interface podem não ser detectáveis testando cada objeto porque esses erros resultam das interações entre os objetos no componente.

FIGURA 8.7 Teste de interface.



Existem diferentes tipos de interface entre os componentes do programa e, consequentemente, diferentes tipos de erro de interface que podem ocorrer:

1. *Interfaces de parâmetro.* São interfaces nas quais são passados dados ou, às vezes, referências a funções, de um componente para o outro. Os métodos em um objeto têm uma interface de parâmetro.
2. *Interfaces de memória compartilhada.* São interfaces nas quais um bloco de memória é compartilhado pelos componentes. Os dados são colocados na memória por um subsistema e recuperados por outros subsistemas. Esse tipo de interface é utilizado nos sistemas embarcados, nos quais sensores criam dados que são recuperados e processados por outros componentes do subsistema.
3. *Interfaces de procedimento.* São interfaces em que um componente encapsula um conjunto de procedimentos, que podem ser chamados por outros componentes. Objetos e componentes reusáveis têm essa forma de interface.
4. *Interfaces de passagem de mensagens.* São interfaces em que um componente solicita um serviço de outro componente passando uma mensagem para ele. Uma mensagem de retorno inclui os resultados da execução do serviço. Alguns sistemas orientados a objetos têm essa forma de interface, assim como os sistemas cliente-servidor.

Os erros de interface são uma das formas de erro mais comuns nos sistemas complexos (LUTZ, 1993). Eles são divididos em três classes:

- *Mau uso da interface.* Um componente chama outro componente e comete um erro no uso de sua interface. Esse tipo de erro é comum nas interfaces de parâmetro, cujos parâmetros podem ser do tipo errado ou ser passados na ordem ou na quantidade erradas.
- *Má compreensão da interface.* Um componente de chamada entende errado a especificação da interface do componente chamado e faz suposições a respeito do seu comportamento. O componente chamado não se comporta conforme o previsto, o que provoca um comportamento inesperado no componente de chamada. Por exemplo, um método de busca binária pode ser chamado com um parâmetro que é um vetor desordenado, o que faria a busca fracassar.
- *Erros de temporização.* Ocorrem nos sistemas de tempo real que usam uma interface de memória compartilhada ou de passagem de mensagem. O produtor e o consumidor dos dados podem trabalhar em velocidades diferentes. A menos que atenção especial seja tomada no projeto da interface, o consumidor pode acessar informações obsoletas porque o produtor da informação não atualizou a informação da interface compartilhada.

O teste de defeitos de interface é difícil porque algumas de suas falhas podem se manifestar apenas em condições incomuns. Por exemplo, digamos que um objeto implemente uma fila como uma estrutura de dados de tamanho fixo. Um objeto chamador pode supor que a fila é implementada como uma estrutura de dados infinita e, assim, não verificar o *overflow* da fila quando um item é inserido.

Essa condição só pode ser detectada se uma sequência de casos de teste que forcem o *overflow* da fila for projetada. Os testes devem averiguar como os objetos chamadores lidam com o *overflow*. No entanto, como essa é uma condição rara, os testadores podem achar que não vale a pena conferi-la enquanto escrevem o conjunto de testes para o objeto fila.

Outro problema pode surgir em virtude das interações entre os defeitos nos diferentes módulos ou objetos. Os defeitos em um objeto podem ser detectados apenas quando algum outro objeto se comporta de maneira inesperada. Digamos que um objeto chame outro para receber algum serviço e que o objeto chamador suponha que a resposta está correta. Se o serviço chamado for defeituoso de alguma maneira, o valor retornado pode ser válido, mas é incorreto. Portanto, o problema não é detectável imediatamente, mas só ficará óbvio quando algum cálculo posterior, usando o valor retornado, der errado.

Algumas diretrizes gerais para o teste de interface são:

1. Examinar o código a ser testado e identificar cada chamada a um componente externo. Projetar um conjunto de testes nos quais os valores dos parâmetros para os componentes externos estejam nas extremidades de seus intervalos, já que esses valores extremos são mais suscetíveis de revelar inconsistências na interface.
2. Testar sempre a interface com um ponteiro nulo como parâmetro nos casos em que ponteiros são passados para uma interface.
3. Nos casos em que um componente é chamado por meio de uma interface de procedimento, projetar testes que causem deliberadamente a falha do componente. Pressupostos de falha diferentes são um dos equívocos mais comuns da especificação.
4. Usar o teste de estresse nos sistemas de passagem de mensagens. Isso significa conceber testes que gerem muito mais mensagens do que costuma ocorrer na prática. Essa é uma maneira eficaz de revelar problemas de temporização.
5. Conceber testes que variem a ordem em que os componentes são ativados, nos casos em que vários componentes interagem por meio de memória

compartilhada. Esses testes podem revelar pressupostos implícitos feitos pelo programador a respeito da ordem em que os dados compartilhados são produzidos e consumidos.

Às vezes, é melhor usar inspeções e revisões em vez de testar em busca de erros de interface. As inspeções podem se concentrar nas interfaces de componentes e nas questões a respeito do comportamento presumido da interface durante o processo de inspeção.

8.1.4 Teste de sistema

O teste de sistema durante o desenvolvimento envolve a integração dos componentes para criar uma versão do sistema e depois testar o sistema integrado. Esse teste verifica se os componentes são compatíveis, se interagem corretamente e se transferem os dados certos no momento certo por meio de suas interfaces. Obviamente, ele se sobrepõe ao teste de componente, mas existem duas diferenças importantes:

1. Durante o teste de sistema, os componentes reusáveis que foram desenvolvidos separadamente e os sistemas de prateleira podem ser integrados com componentes recém-desenvolvidos. Depois, o sistema completo é testado.
2. Os componentes desenvolvidos por diferentes membros do time ou subtimes podem ser integrados nessa fase. O teste de sistema é um processo coletivo, e não individual. Em algumas empresas, ele pode envolver uma equipe de testes separada, sem participação de projetistas e programadores.

Todos os sistemas têm um comportamento emergente. Isso significa que algumas funcionalidades e características do sistema se tornam óbvias somente quando os componentes são unidos. Isso pode ser um comportamento emergente planejado, que precisa ser testado. Por exemplo, é possível integrar um componente de autenticação com um componente que atualize o banco de dados do sistema. Passa-se então a ter uma característica do sistema que restringe as atualizações de informação a usuários autorizados. No entanto, às vezes o comportamento emergente não é planejado, além de ser indesejado. É preciso desenvolver testes que verifiquem se o sistema está fazendo apenas o que deveria fazer.

O teste de sistema deve se concentrar em testar as interações entre os componentes e objetos que compõem um sistema. Componentes ou sistemas reusáveis também podem ser testados para averiguar se funcionam conforme o previsto quando são integrados aos novos componentes. Esse teste de interação deve descobrir os defeitos do componente, que só são revelados quando ele é utilizado por outros no sistema. O teste de interação também ajuda a encontrar equívocos, cometidos pelos desenvolvedores do sistema, a respeito dos demais componentes.

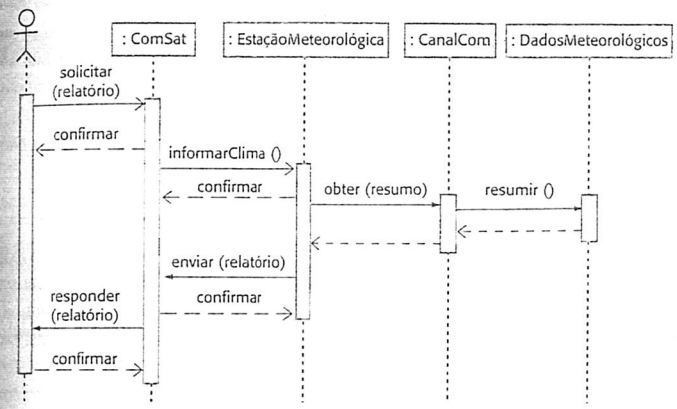
Em virtude de seu foco nas interações, o teste baseado em casos de uso é uma abordagem eficaz para o teste de sistema. Normalmente, cada caso de uso do sistema é implementado por vários componentes ou objetos. Testar o caso de uso força a ocorrência dessas interações. Caso seja desenvolvido um diagrama de sequência para modelar a implementação do caso de uso, será possível observar os objetos ou os componentes envolvidos na interação.

No exemplo da estação meteorológica na natureza, o software do sistema informa dados meteorológicos resumidos para um computador remoto, conforme descrito no

Capítulo 7 (Figura 7.3). A Figura 8.8 mostra a sequência de operações na estação meteorológica quando ela responde a uma solicitação para coletar dados para o sistema de mapeamento. Esse diagrama pode ser usado para identificar as operações que serão testadas e ajudar a projetar os casos de teste para executar os testes. Portanto, enviar uma solicitação de relatório vai resultar na execução da seguinte sequência de métodos:

ComSat:solicitar → EstaçãoMeteorológica:informarClima
→ CanalCom:obter (resumo) → DadosMeteorológicos:resumir

FIGURA 8.8 Diagrama de sequência da coleta de dados meteorológicos.



O diagrama de sequência ajuda a projetar casos de teste específicos que são necessários, pois mostram quais são as entradas exigidas e as saídas criadas:

1. Uma solicitação de entrada para um relatório deve ter uma confirmação associada. No final das contas, a solicitação deve retornar um relatório. Durante o teste, devem ser criados dados resumidos que possam ser utilizados para verificar se o relatório está organizado corretamente.
2. Uma solicitação de entrada para um relatório da **EstaçãoMeteorológica** resulta na geração de um relatório resumido. É possível testar isso isoladamente, criando dados brutos correspondentes ao resumo que foi preparado para o teste de **ComSat** e verificando se o objeto **EstaçãoMeteorológica** produz corretamente esse resumo. Esses dados brutos também são utilizados para testar o objeto **DadosMeteorológicos**.

É claro que simplifiquei o diagrama de sequência na Figura 8.8 para não mostrar as exceções. Um teste de caso de uso/cenário completo deve levar em conta essas exceções e assegurar que sejam tratadas corretamente.

Na maioria dos sistemas, é difícil saber o quanto de teste de sistema é essencial e quando se deve parar de testar. É impossível testar exaustivamente, ao ponto em que toda a sequência possível de execução do programa seja testada. Portanto, os testes devem se basear em subconjuntos possíveis de casos de teste. Em condições ideais, as empresas de software devem ter políticas para escolher

esse subconjunto, as quais poderiam se basear em políticas de teste genéricas, como uma determinação de que todas as linhas de código do programa sejam executadas pelo menos uma vez. Alternativamente, elas podem se basear na experiência de uso do sistema e se concentrar em testar as características do sistema pronto. Por exemplo:

1. Todas as funções do sistema que são acessadas por meio de menus devem ser testadas.
2. Combinações de funções (por exemplo, formatação de texto) que são acessadas por meio de um mesmo menu devem ser testadas.
3. No caso de entrada de dados do usuário, todas as funções devem ser testadas com entradas certas e erradas.

Pela experiência com importantes produtos de software, como processadores de texto ou planilhas eletrônicas, fica claro que, durante o teste do produto, normalmente são utilizadas diretrizes parecidas. Quando as características (*features*) do software são utilizadas isoladamente, normalmente elas funcionam. Como explica Whittaker (2009), os problemas surgem quando combinações de características menos utilizadas não foram testadas juntas. Ele dá o exemplo de um processador de texto muito utilizado, em que o uso de notas de rodapé com o leiaute de múltiplas colunas provoca uma disposição incorreta do texto.

O teste de sistema automatizado normalmente é mais difícil do que o teste automatizado de unidade ou de componente. O teste de unidade automatizado se baseia na previsão das saídas e na codificação posterior dessas previsões em um programa. Essa previsão é, então, comparada ao resultado. Entretanto, a razão de implementar um sistema pode ser a geração de saídas grandes ou que não podem ser previstas com facilidade. É possível examinar uma saída e verificar a sua credibilidade sem necessariamente ser capaz de criá-la antecipadamente.

Integração e teste incrementais

O teste de sistema envolve a integração de diferentes componentes e o teste do sistema integrado que foi criado. Uma abordagem incremental para integração e teste deve sempre ser usada: se um componente é integrado, o sistema é testado, se outro componente é integrado, o sistema é testado novamente, e assim por diante. Se ocorrerem problemas, eles provavelmente se deverão a interações com o componente integrado mais recentemente.

A integração e o teste incrementais são fundamentais para os métodos ágeis, cujos testes de regressão são executados toda vez que um novo incremento é integrado.

8.2 DESENVOLVIMENTO DIRIGIDO POR TESTES

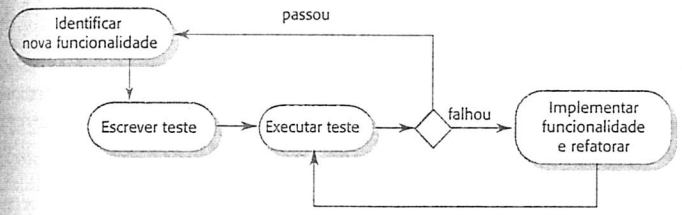
O desenvolvimento dirigido por testes (TDD, do inglês *test-driven development*) é uma abordagem para desenvolvimento de programas na qual se intercalam testes e desenvolvimento do código (BECK, 2002; JEFFRIES; MELNIK, 2007). Esse código é desenvolvido incrementalmente, junto a um conjunto de testes para cada incremento, e o próximo incremento não começa até que o código que está sendo desenvolvido seja aprovado em todos os testes. O desenvolvimento dirigido por testes foi introduzido como parte do método de desenvolvimento ágil Programação

Extrema (XP). Hoje, no entanto, ganhou aceitação geral e pode ser utilizado tanto em processos ágeis como em processos dirigidos por plano.

O processo fundamental de desenvolvimento dirigido por testes é exibido na Figura 8.9. As etapas no processo são:

1. Definir o incremento de funcionalidade necessário. Normalmente ele deve ser pequeno e implementável em poucas linhas de código.
2. Escrever um teste para a funcionalidade e implementá-lo como teste automatizado. Isso significa que o teste pode ser executado e informará se a funcionalidade for aprovada ou reprovada.
3. Executar o teste, junto a todos os outros testes que foram implementados. Inicialmente, a funcionalidade não terá sido implementada ainda, então o teste falhará. Isso é proposital, pois mostra que o teste acrescenta alguma coisa ao conjunto de testes.
4. Implementar a funcionalidade e executar novamente o teste. Isso pode envolver a refatoração do código existente para melhorá-lo e acrescentar novo código ao que já existe.
5. Depois que todos os testes são executados com sucesso, é possível passar para a implementação do próximo pedaço de funcionalidade.

FIGURA 8.9 Desenvolvimento dirigido por testes.



Um ambiente de teste automatizado — como o JUnit, que dá apoio ao teste de programas em Java (TAHCHIEV *et al.*, 2010) — é essencial para o TDD. Como o código é desenvolvido em incrementos muito pequenos, é necessário rodar cada teste a cada acréscimo de funcionalidade ou a cada refatoração do programa. Portanto, os testes são embutidos em um programa separado, que os executa e invoca o sistema que está sendo testado. Usando essa abordagem, é possível executar centenas de testes diferentes em poucos segundos.

O desenvolvimento dirigido por testes ajuda os programadores a esclarecerem suas ideias quanto ao que o segmento de código realmente deve fazer. Para escrever um teste, é preciso compreender o objetivo do que está sendo feito, já que essa compreensão facilita a escrita do código necessário. Naturalmente, o TDD não ajudará nos casos que se iniciam de um conhecimento ou de uma compreensão incompletos.

Caso não se conheça o suficiente para escrever os testes, então o código necessário não será desenvolvido. Por exemplo, se um cálculo envolve divisão, deve-se verificar se os números não estão sendo divididos por zero; porém, se o programador se esquecer de escrever um teste para isso, então o código que verifica isso nunca será incluído no programa.

Além de uma melhor compreensão do problema, outros benefícios do desenvolvimento dirigido por testes são:

1. *Cobertura de código.* A princípio, todo segmento de código que é escrito deve ter pelo menos um teste associado. Portanto, dá para confiar que todo o código no sistema foi realmente executado. O código é testado à medida que é escrito, então os defeitos são descobertos precocemente no processo de desenvolvimento.
2. *Teste de regressão.* Um conjunto de testes é desenvolvido incrementalmente à medida que o programa também é desenvolvido. Sempre é possível executar testes de regressão para conferir se as mudanças no programa não introduziram novos defeitos.
3. *Depuração simplificada.* Quando um teste falha, deve ficar óbvia a localização do problema. O código recém-escrito precisa ser verificado e modificado. Não é preciso usar ferramentas de depuração para localizar o problema. Relatos do uso de TDD sugerem que raramente é necessário usar um depurador automático no desenvolvimento dirigido por testes (MARTIN, 2007).
4. *Documentação de sistemas.* Os próprios testes agem como uma forma de documentação que descreve o que o código deveria estar fazendo. A leitura dos testes pode facilitar a compreensão do código.

Um dos benefícios mais importantes do TDD é que ele reduz os custos do teste de regressão, que envolve rodar um conjunto de testes que foram executados com sucesso após modificações terem sido feitas no sistema. O teste de regressão verifica se essas modificações não introduziram novos defeitos no sistema e se o novo código interage com o código existente conforme o esperado. Contudo, ele é caro e, às vezes, impraticável; nas vezes em que o sistema é testado manualmente, os custos de tempo e esforço são muito altos. Ao tentar escolher os testes mais relevantes para serem executados novamente, é fácil deixar passar testes importantes.

O teste automatizado reduz drasticamente os custos do teste de regressão. Os testes existentes podem ser reexecutados rapidamente e de forma barata. Depois de uma modificação em um sistema desenvolvido com testes *a priori* (*test-first*), todos os testes existentes devem ser executados com sucesso antes de se adicionar outra funcionalidade. Ao programar, pode-se ter confiança de que a nova funcionalidade adicionada não provocou ou revelou problemas com o código existente.

O desenvolvimento dirigido por testes é mais valioso no desenvolvimento de um novo software, em que a funcionalidade é implementada em um novo código ou por meio do uso de componentes de bibliotecas padrões. Caso estejam sendo reusados componentes com muito código ou sistemas legados, então é preciso escrever testes para esses sistemas como um todo. Não é possível decompô-los facilmente em elementos testáveis separadamente. O desenvolvimento incremental dirigido por testes é impraticável. Ele também pode ser ineficaz nos sistemas *multithreaded*, cujas diferentes *threads* podem ser intercaladas em momentos e execuções distintos do teste e, portanto, produzir resultados diferentes.

Se o TDD é empregado, ainda é preciso um processo de teste de sistema para validá-lo, ou seja, conferir se ele cumpre os requisitos de todos os *stakeholders* do sistema. O teste de sistema também averigua seu desempenho, sua confiabilidade e confere se o sistema faz coisas que não deveria fazer, como produzir saídas indesejadas. Andrea (2007) sugere como as ferramentas de teste podem ser estendidas para integrar alguns aspectos do teste de sistema com o TDD.

Hoje, o desenvolvimento dirigido por testes é amplamente utilizado, sendo a abordagem convencional para o teste de software. A maioria dos programadores que adotaram esse método está feliz com ele e o consideram uma maneira mais produtiva de desenvolver software. Também se reivindica que o uso do TDD incentiva a melhor estruturação de um programa e um código de melhor qualidade, mas os experimentos destinados a verificar essa reivindicação foram inconclusivos.

8.3 TESTE DE LANÇAMENTO

O teste de lançamento (*release*) é um processo de teste de um determinado lançamento de um sistema, destinado ao uso fora do time de desenvolvimento. Normalmente, o lançamento do sistema é voltado para clientes e usuários. Entretanto, em um projeto complexo, o lançamento poderia ser para outros times que estão desenvolvendo sistemas relacionados. Para produtos de software, o lançamento poderia ser para o gerenciamento do produto que depois o prepara para venda.

Existem duas distinções importantes entre o teste de lançamento e o teste de sistema durante o processo de desenvolvimento:

1. O time de desenvolvimento do sistema não deve ser responsável pelo teste de lançamento.
2. O teste de lançamento é o processo de conferir a validade para garantir que um sistema cumpra seus requisitos e seja bom o bastante para ser usado pelos clientes do sistema. O teste de sistema pelo time de desenvolvimento deve se concentrar em descobrir *bugs* nesse sistema (teste de defeitos).

O principal objetivo do teste de lançamento é convencer o fornecedor do sistema de que ele é suficientemente bom para uso. Se assim for, esse sistema pode ser lançado como um produto ou fornecido para o cliente. Portanto, o teste de lançamento tem de mostrar que o sistema fornece a funcionalidade, o desempenho e a dependabilidade especificados e que não falha durante o uso normal.

O teste de lançamento normalmente é um processo caixa-preta por meio do qual são derivados os testes a partir da especificação do sistema. O sistema é tratado como uma caixa-preta cujo comportamento só pode ser determinado pelo estudo de suas entradas e saídas relacionadas. Outro nome para isso é teste funcional, assim chamado porque o testador só se preocupa com a funcionalidade do software, e não com sua implementação.

8.3.1 Teste baseado em requisitos

Um princípio geral das práticas recomendadas em engenharia de requisitos é que requisitos devem ser testáveis. Em outras palavras, os requisitos devem ser escritos de modo que testes possam ser criados para eles, e um testador possa verificar se eles foram satisfeitos. Portanto, o teste baseado em requisitos é uma abordagem sistemática para o projeto de casos de teste, em que se considera cada requisito a fim de derivar deles um conjunto de testes. O teste baseado em requisitos é uma validação, não um teste de defeitos — o objetivo é demonstrar que o sistema implementou seus requisitos corretamente.

Por exemplo, considere os seguintes requisitos do sistema Mentcare, preocupados com a checagem de alergias de pacientes a certos medicamentos:

Se é sabido que um paciente é alérgico a qualquer medicamento específico, então a prescrição desse medicamento deve resultar em uma mensagem de alerta emitida para o usuário do sistema.

Se quem prescreve o medicamento opta por ignorar uma advertência de alergia, esse usuário deverá fornecer um motivo para isso.

Para conferir se esses requisitos foram satisfeitos, pode ser necessário desenvolver vários testes relacionados:

1. Criar um registro de paciente sem alergias conhecidas. Prescrever medicação para alergias conhecidas e checar se uma mensagem de advertência não é emitida pelo sistema.
2. Criar um registro de paciente com uma alergia conhecida. Prescrever medicação à qual o paciente é alérgico e verificar se a advertência é emitida pelo sistema.
3. Criar um registro de paciente com alergia a um ou mais medicamentos. Prescrever esses dois medicamentos separadamente e verificar se a advertência correta para cada medicamento é emitida.
4. Prescrever dois medicamentos aos quais o paciente é alérgico. Verificar se duas advertências são corretamente emitidas.
5. Prescrever um medicamento que emita uma advertência e ignorá-la. Verificar se o sistema exige que o usuário forneça informações explicando por que a advertência foi ignorada.

A partir dessa lista, é possível ver que testar um requisito não significa escrever apenas um único teste. Normalmente, vários testes devem ser escritos para garantir a cobertura de todo o requisito. Além disso, devem ser mantidos registros de rastreabilidade do teste baseado em requisitos, que vinculam os testes aos requisitos específicos que foram testados.

8.3.2 Teste de cenário

O teste de cenário é uma abordagem para o teste de lançamento por meio da qual são criados cenários de uso típicos, com o objetivo de empregá-los para desenvolver casos de teste para o sistema. O cenário é uma história que descreve uma maneira como o sistema poderia ser usado e que deve ser realista, para que os verdadeiros usuários sejam capazes de se relacionar com ele. Caso tenham sido utilizados cenários ou histórias de usuário como parte do processo de engenharia de requisitos (descrito no Capítulo 4), então será possível reutilizá-los como cenários de teste.

Em um curto artigo sobre teste de cenário, Kaner (2003) sugere que um teste de cenário deve ser uma história narrativa crível e razoavelmente complexa. Ele deve motivar os *stakeholders*: eles devem se relacionar com o cenário e acreditar que é importante que o sistema passe no teste. Kaner também sugere que o cenário deve ser fácil de avaliar. Se houver problemas com o sistema, então a equipe de teste de lançamento deve reconhecê-los.

Como um exemplo de possível cenário do sistema Mentcare, a Figura 8.10 descreve uma maneira de utilizar o sistema em uma visita domiciliar. Esse cenário testa uma série de características do sistema Mentcare:

1. Autenticação fazendo *login* no sistema.
2. Baixar e enviar registros de paciente especificados em um notebook.
3. Agendamento das visitas domiciliares.
4. Criptografar e descriptografar os registros dos pacientes em um dispositivo móvel.
5. Recuperação e modificação de registros.
6. Vínculos com o banco de dados de medicamentos que mantém informações sobre efeitos colaterais.
7. Sistema de solicitação de chamadas.

FIGURA 8.10 Uma história de usuário para o sistema Mentcare.

Jorge é um enfermeiro especializado em atendimento de pacientes de saúde mental. Uma de suas responsabilidades é visitar os pacientes em casa para averiguar se o seu tratamento é eficaz e se não estão sofrendo com efeitos colaterais da medicação. Em um dia de visitas domiciliares, Jorge entra no sistema Mentcare e o utiliza para imprimir a agenda de visitas do dia e um resumo das informações sobre os pacientes que visitará. Ele solicita que os registros dos pacientes sejam baixados para o seu notebook. O sistema pede a sua senha para criptografar os registros no notebook.

Um dos pacientes que ele visita é Tiago, que está sendo tratado com medicação para depressão. Tiago sente que a medicação o está ajudando, mas acredita que ela tem o efeito colateral de mantê-lo acordado à noite. Jorge procura o registro de Tiago e o sistema pede a sua senha para descriptografar o registro. Ele confere o medicamento prescrito e consulta seus efeitos colaterais. A insônia é um efeito colateral conhecido, então ele anota o problema no registro de Tiago e sugere que ele visite a clínica para mudar a medicação. Tiago concorda, e Jorge insere um aviso para ligar para ele quando voltar à clínica para marcar uma consulta com um médico. Jorge termina a consulta e o sistema criptografa novamente o registro de Tiago.

Após terminar suas consultas, Jorge volta para a clínica e envia os registros dos pacientes visitados ao banco de dados. O sistema gera para Jorge uma lista de chamada dos pacientes com quem ele teve contato para obter informações de acompanhamento e marcar consultas.

Ao testar um lançamento, é necessário percorrer esse cenário desempenhando o papel de Jorge e observando como o sistema se comporta em resposta às diferentes entradas. No papel de Jorge, é possível cometer erros deliberados, como digitar a senha errada para decodificar registros. Isso confere a resposta do sistema aos erros. Deve-se observar atentamente quaisquer problemas que apareçam, incluindo problemas de desempenho. Se um sistema for lento demais, isso vai mudar a maneira como é utilizado. Por exemplo, se levar tempo demais para criptografar um registro, então os usuários que disponham de pouco tempo podem pular esse estágio. Se eles perderem seu notebook, uma pessoa não autorizada poderia visualizar os registros dos pacientes.

Quando se usa uma abordagem baseada em cenário, normalmente se está testando vários requisitos dentro do mesmo cenário. Portanto, além de conferir os requisitos individuais, o testador também está conferindo se as combinações de requisitos não causam problemas.

8.3.3 Teste de desempenho

Depois que o sistema foi completamente integrado, é possível testar as propriedades emergentes, como o desempenho e a confiabilidade. Os testes de desempenho têm de ser projetados para garantir que o sistema possa processar sua carga

pretendida. Normalmente, isso envolve executar uma série de testes nos quais se aumenta a carga até o desempenho do sistema ficar inaceitável.

Assim como outros tipos de teste, o de desempenho se preocupa tanto em mostrar que o sistema cumpre os requisitos quanto em descobrir problemas e defeitos no sistema. Para testar se os requisitos de desempenho foram alcançados, pode ser necessário construir um perfil operacional, ou seja, um conjunto de testes que reflete a composição real do trabalho que será tratado pelo sistema (ver Capítulo 11). Portanto, se 90% das transações em um sistema forem do tipo A, 5% do tipo B e o restante dos tipos C, D e E, então deve ser projetado um perfil operacional para que a ampla maioria dos testes seja do tipo A, senão, não será possível obter um teste preciso do desempenho operacional do sistema.

Essa abordagem, é claro, não é necessariamente a melhor para testar defeitos. A experiência mostrou que uma maneira eficaz de descobri-los é projetar testes em torno de limites do sistema. No teste de desempenho, isso significa estressar o sistema com demandas que estão fora dos limites do projeto do software. Esse processo é conhecido como teste de estresse.

Imagine que um sistema de processamento de transações projetado para processar até 300 transações por segundo esteja sendo testado. De início, testa-se esse sistema com menos de 300 transações por segundo; depois, a carga é aumentada gradualmente no sistema, para além de 300 transações por segundo, até que fique bem acima da carga máxima de projeto e o sistema falhe. O teste de estresse ajuda a fazer duas coisas:

1. Testar o comportamento de falha do sistema. Podem surgir circunstâncias por meio de uma combinação inesperada de eventos, em que a carga depositada no sistema ultrapasse o máximo previsto. Nessas circunstâncias, a falha do sistema não deve causar corrupção de dados ou perda imprevista de serviços de usuário. O teste de estresse verifica se o fato de sobrecarregar o sistema faz com que ele falhe de forma 'suave' em vez de entrar em colapso em virtude da carga.
2. Revelar defeitos que só aparecem quando o sistema está plenamente carregado. Embora se possa argumentar que há pouca probabilidade de esses defeitos causarem falhas do sistema em uso normal, pode haver combinações incomuns de circunstâncias replicadas pelo teste de estresse.

O teste de estresse é particularmente relevante para sistemas distribuídos baseados em uma rede de processadores. Esses sistemas frequentemente exibem degradação severa quando estão intensamente carregados. A rede fica inundada com dados de coordenação que os diferentes processos precisam trocar. Esses processos ficam cada vez mais lentos enquanto esperam pelos dados necessários que estão com os outros processos. O teste de estresse ajuda a descobrir quando começa essa degradação, de maneira que se possa adicionar verificações no sistema para rejeitar transações além desse ponto.

8.4 TESTE DE USUÁRIO

O teste de usuário é um estágio no processo de teste no qual os usuários ou clientes fornecem entradas e conselhos sobre o teste de sistema. Isso pode envolver o teste formal de um sistema que foi contratado de um fornecedor externo. Por outro lado, pode ser um processo informal em que os usuários experimentam um novo produto

de software para ver se gostam e conferem se ele atende suas necessidades. O teste de usuário é essencial, mesmo em casos em que já tenham sido realizados testes abrangentes de sistema e de lançamento, pois as influências do ambiente de trabalho do usuário podem ter um efeito importante na confiabilidade, no desempenho, na usabilidade e na robustez de um sistema.

É praticamente impossível que um desenvolvedor de sistema replique o ambiente de trabalho desse sistema, pois os testes no ambiente de desenvolvimento são inevitavelmente artificiais. Por exemplo, um sistema destinado ao uso em um hospital é empregado em um ambiente clínico onde outras coisas estão acontecendo, como emergências de pacientes e conversas com parentes. Tudo isso afeta o uso de um sistema, mas os desenvolvedores não conseguem incluir esses fatos em seu ambiente de teste. Existem três tipos diferentes de teste de usuário:

1. *Teste-alfa*, no qual um grupo selecionado de usuários do software trabalha em estreita colaboração com o time de desenvolvimento para testar lançamentos iniciais do software.
2. *Teste-beta*, em que um lançamento do software é disponibilizado para um grupo maior de usuários, para que experimentem e informem os problemas descobertos para os desenvolvedores do sistema.
3. *Teste de aceitação*, em que os clientes testam o sistema para decidir se está pronto ou não para ser aceito junto dos desenvolvedores do sistema e instalado no ambiente do cliente.

No teste-alfa, usuários e desenvolvedores trabalham juntos para testar um sistema enquanto está sendo desenvolvido. Isso significa que os usuários podem identificar problemas e questões que não são imediatamente aparentes para a equipe de teste. Os desenvolvedores só podem trabalhar realmente a partir dos requisitos, mas frequentemente eles não refletem outros fatores que afetam o uso prático do software. Portanto, os usuários podem fornecer informações sobre a prática que ajudem no projeto de testes mais realistas.

O teste-alfa é frequentemente utilizado quando se desenvolvem produtos de software ou aplicativos. Os usuários experientes desses produtos podem estar dispostos a se envolver no processo de teste-alfa porque isso lhes dá informações iniciais a respeito das características do novo sistema que eles podem aproveitar. Também reduz o risco de que as mudanças imprevistas no software venham a ter efeitos perturbadores em seus negócios. No entanto, o teste-alfa também pode ser utilizado quando está sendo desenvolvido um software personalizado. Os métodos de desenvolvimento ágil defendem o envolvimento do usuário no processo de desenvolvimento e que os usuários devem desempenhar um papel fundamental na concepção dos testes do sistema.

O teste-beta acontece quando um lançamento inicial (às vezes inacabado) de um sistema de software é disponibilizado para um grupo maior de clientes e usuários para avaliação. Os aplicadores do teste-beta podem ser um grupo selecionado de clientes pioneiros na adoção do sistema. Como alternativa, o software pode ser disponibilizado publicamente para qualquer um que esteja interessado em experimentá-lo.

O teste-beta é utilizado principalmente em produtos de software aplicados a muitos contextos diferentes. Isso é importante, já que, ao contrário dos desenvolvedores de produtos personalizados, não há meio de o desenvolvedor do produto limitar o ambiente operacional do software. É impossível para os desenvolvedores de produto conhecerem e replicarem todos os contextos em que o produto de software será

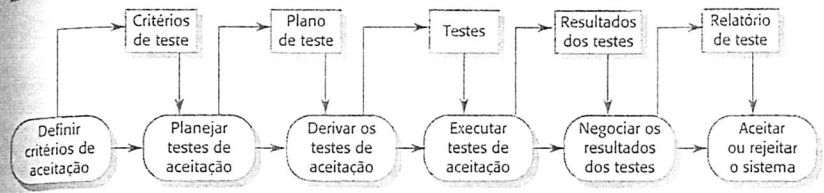
utilizado. Portanto, o teste-beta é utilizado para descobrir problemas de interação entre o software e as características de seu ambiente operacional. Ele também é uma forma de marketing, uma vez que os clientes aprendem sobre o sistema e o que ele pode fazer por eles.

O teste de aceitação é uma parte inerente do desenvolvimento de sistemas personalizados. Os clientes testam um sistema, usando seus próprios dados, e decidem se ele deve ser aceito do desenvolvedor do sistema. A aceitação implica que o pagamento final pelo software deve ser feito.

A Figura 8.11 mostra que existem seis estágios no processo de teste de aceitação:

1. *Definir critérios de aceitação.* Em condições ideais, essa etapa deve ocorrer no início do processo, antes de se assinar o contrato do sistema. Os critérios de aceitação devem fazer parte do contrato do sistema e ser aprovados pelo cliente e pelo desenvolvedor. Entretanto, na prática, pode ser difícil definir critérios tão no início do processo. Os requisitos detalhados podem não estar disponíveis, e os requisitos quase certamente mudarão durante o processo de desenvolvimento.
2. *Planejar testes de aceitação.* Essa etapa envolve decisões sobre recursos, tempo e orçamento tanto para o teste de aceitação quanto para o estabelecimento de um cronograma de testes. O plano do teste de aceitação também deve discutir a cobertura necessária dos requisitos e a ordem em que as características do sistema são testadas. Além disso, deve definir os riscos para o processo de teste, como as falhas do sistema e o desempenho inadequado, e discutir como esses riscos podem ser mitigados.
3. *Derivar os testes de aceitação.* Depois que os critérios de aceitação foram estabelecidos, podem ser criados testes para conferir se um sistema é aceitável ou não. Os testes de aceitação devem visar tanto as características funcionais quanto as não funcionais (por exemplo, desempenho) do sistema. Em condições ideais, eles devem proporcionar uma cobertura completa dos requisitos do sistema. Na prática, é difícil estabelecer critérios de aceitação completamente objetivos. Muitas vezes há escopo para argumentar sobre o teste mostrar ou não se um critério foi definitivamente cumprido.
4. *Executar testes de aceitação.* Os testes de aceitação acordados são executados no sistema. Em condições ideais, essa etapa deve ocorrer no ambiente real em que o sistema será utilizado, mas isso pode ser disruptivo e impraticável. Portanto, um ambiente de teste de usuário pode ter de ser configurado para executar esses testes. É difícil automatizar esse processo, já que parte dos testes de aceitação pode envolver o teste das interações entre usuários finais e o sistema. Pode ser necessário algum treinamento dos usuários finais.
5. *Negociar os resultados dos testes.* É muito improvável que todos os testes de aceitação definidos venham a ser aprovados e que não haverá problemas com o sistema. Se for o caso, então o teste de aceitação está completo e o sistema pode ser entregue. Na maioria das vezes, alguns problemas serão descobertos. Nesses casos, o desenvolvedor e o cliente devem negociar para decidir se o sistema é bom o bastante para ser usado. Eles também devem concordar sobre como o desenvolvedor vai consertar os problemas identificados.
6. *Aceitar ou rejeitar o sistema.* Esse estágio envolve uma reunião entre os desenvolvedores e o cliente para decidir se ou sistema deve ou não ser aceito. Se ele não for bom o bastante para ser utilizado, então é necessário mais desenvolvimento para consertar os problemas identificados. Feito isso, a fase de teste de aceitação é repetida.

FIGURA 8.11 Processo de teste de aceitação.



Há quem ache que o teste de aceitação é uma questão contratual clara. Se um sistema não passar nos testes de aceitação, então ele deve ser rejeitado e o pagamento não deve ser realizado. No entanto, a realidade é mais complexa. Os clientes querem usar o software o quanto antes em virtude dos benefícios de sua instalação imediata. Eles podem ter comprado hardware novo, treinado pessoal e mudado seus processos. Podem estar dispostos a aceitar o software, apesar dos problemas, porque o custo de não usá-lo é maior do que o custo de contornar os problemas.

Portanto, o resultado das negociações pode ser a aceitação condicional do sistema. O cliente pode aceitar o sistema para que a instalação possa começar. O fornecedor do sistema concorda em consertar os problemas urgentes e entregar uma nova versão para o cliente o mais rápido possível.

Nos métodos ágeis, como a Programação Extrema, pode não haver uma atividade de teste de aceitação separada. O usuário final faz parte do time de desenvolvimento (ou seja, ele faz um teste-alfa) e fornece os requisitos do sistema em termos de histórias de usuário. Ele também é responsável por definir os testes, que decidem se o software desenvolvido atende ou não as suas histórias. Portanto, esses testes equivalem aos testes de aceitação. Esses testes são automatizados e o desenvolvimento não prossegue antes que os testes de aceitação da história tenham sido executados com êxito.

Quando os usuários estão incorporados a um time de desenvolvimento de software, eles devem ser usuários 'típicos' com conhecimento geral sobre como o sistema será utilizado. No entanto, pode ser difícil encontrar esses usuários e, portanto, os testes de aceitação podem não ser um reflexo verdadeiro de como um sistema é utilizado na prática. Além do mais, a necessidade de testes automatizados limita a flexibilidade dos sistemas interativos de teste. Nesse tipo de sistema, o teste de aceitação pode exigir grupos de usuários finais usando o sistema como se ele fizesse parte do trabalho diário. Sendo assim, embora um 'usuário incorporado' seja um conceito atraente a princípio, ele não necessariamente leva a testes de alta qualidade do sistema.

O problema do envolvimento do usuário nos times ágeis é uma razão para muitas empresas usarem uma mistura de testes ágeis e mais tradicionais. O sistema pode ser desenvolvido usando técnicas de desenvolvimento ágil, mas, após a conclusão de um grande lançamento, utiliza-se o teste de aceitação separado para decidir se o sistema deve ser aceito.

PONTOS-CHAVE

- O teste só consegue mostrar a presença de erros em um programa. Ele não consegue mostrar que não existem outros defeitos remanescentes.
- O teste de desenvolvimento é responsabilidade do time que desenvolve o software. Outra equipe deve ser responsável por testar um sistema antes que ele seja enviado aos usuários. No processo de teste de usuário, clientes ou usuários do sistema fornecem dados de teste e verificam se os testes são bem-sucedidos.
- O teste de desenvolvimento inclui o teste de unidade, no qual são testados objetos e métodos individuais; o teste de componentes, no qual são testados grupos de objetos relacionados; e o teste de sistema, no qual são testados sistemas parciais ou completos.
- Quando se testa um software, é necessário tentar 'quebrá-lo' usando experiência e diretrizes para escolher os tipos de casos de teste que foram eficazes na descoberta de defeitos em outros sistemas.
- Sempre que possível, testes automatizados devem ser escritos. Eles são incorporados em um programa, que pode ser executado toda vez que for feita uma mudança em um sistema.
- O desenvolvimento com testes *a priori* (*test-first*) é uma abordagem para o desenvolvimento por meio da qual os testes são escritos antes do código ser testado. São feitas pequenas alterações no código e ele é refatorado até que todos os testes sejam executados com êxito.
- O teste de cenário é útil porque replica o uso prático do sistema. Ele envolve inventar um cenário de uso típico e aproveitá-lo para derivar casos de teste.
- O teste de aceitação é um processo de teste de usuário no qual o objetivo é decidir se o software é bom o bastante para ser instalado e utilizado em seu ambiente operacional planejado.

LEITURAS ADICIONAIS

"How to design practical test cases." Um artigo didático sobre o projeto de casos de teste escrito por um autor de uma empresa japonesa com boa reputação por entregar software com muito poucos defeitos. YAMAURA, T. *IEEE Software*, v. 15, n. 6, nov. 1998. doi:10.1109/52.730835.

"Test-driven development." Essa edição especial sobre desenvolvimento dirigido por testes inclui uma boa visão geral do TDD e artigos sobre como ele tem sido utilizado em diferentes tipos de software. *IEEE Software*, v. 24, n. 3, mai./jun. 2007.

Exploratory software testing. Trata-se de um livro prático (e não teórico) sobre teste de software que desenvolve as ideias do livro anterior de Whittaker, *How to break software*. O autor apresenta um conjunto de diretrizes baseadas na experiência sobre teste de software. WHITTAKER, J. A. Addison-Wesley, 2009.

How Google tests software. Esse é um livro sobre teste de sistemas de larga escala baseados na nuvem e apresenta um novo conjunto de desafios comparados com as aplicações de software personalizadas. Embora eu não ache que a abordagem do Google possa ser utilizada diretamente, existem lições interessantes nesse livro para teste de sistema em larga escala. WHITTAKER, J.; ARBON, J.; CAROLLO, J. Addison-Wesley, 2012.

SITE[2]

Apresentações em PowerPoint para este capítulo disponíveis em: <http://software-engineering-book.com/slides/chap8/>.
Links para vídeos de apoio disponíveis em: <http://software-engineering-book.com/videos/implementation-and-evolution/>.

2 Todo o conteúdo disponibilizado na seção *Site* de todos os capítulos está em inglês.

EXERCÍCIOS

8.1 Explique por que não é necessário que um programa seja completamente isento de defeitos antes de ser entregue aos seus clientes.

8.2 Explique por que os testes só conseguem detectar a presença de erros, e não a sua ausência.

8.3 Algumas pessoas argumentam que os desenvolvedores não devem se envolver no teste do seu próprio código, mas que todo teste deve ser de responsabilidade de uma equipe diferente. Apresente argumentos contra e a favor do teste ser realizado pelos próprios desenvolvedores.

8.4 Imagine que pediram a você que testasse um método chamado `reduzirEspaçoEmBranco` em um objeto 'Parágrafo' que, dentro do parágrafo, substitui sequências de espaços em branco por um único espaço em branco. Identifique as partições de teste desse exemplo e derive um conjunto de testes para o método `reduzirEspaçoEmBranco`.

8.5 O que é teste de regressão? Explique como o uso dos testes automatizados e de um *framework* de teste como JUnit simplifica esse procedimento.

8.6 O sistema Mentcare é construído adaptando um sistema de prateleira. Quais são as diferenças entre testar um sistema como esse e testar um software desenvolvido com uma linguagem orientada a objetos como Java?

8.7 Escreva um cenário que poderia ser utilizado para ajudar a projetar testes para o sistema de estações meteorológicas na natureza.

8.8 O que você entende pelo termo 'teste de estresse'? Sugira como você poderia aplicá-lo ao sistema Mentcare.

8.9 Quais são os benefícios de envolver os usuários no teste de lançamento em um estágio inicial do processo de teste? Existem desvantagens no envolvimento do usuário?

8.10 Uma abordagem comum para o teste de sistema é testá-lo até consumir todo o orçamento de teste e depois entregar o sistema para os clientes. Discuta a ética dessa abordagem para os sistemas que são entregues para clientes externos.

REFERÊNCIAS

ANDREA, J. "Envisioning the next generation of functional testing tools." *IEEE Software*, v. 24, n. 3, 2007. p. 58-65. doi:10.1109/MS.2007.73.

BECK, K. *Test driven development*: by example. Boston: Addison-Wesley, 2002.

BEZIER, B. *Software testing techniques*. 2. ed. New York: Van Nostrand Reinhold, 1990.

BOEHM, B. W. "Software engineering; R & D trends and defense needs." In: WEGNER, P. (ed.). *Research directions in software technology*, 1-9. Cambridge, MA: MIT Press, 1979.

CUSUMANO, M.; SELBY, R. W. *Microsoft secrets*. New York: Simon & Schuster, 1998.

DIJKSTRA, E. W. "The humble programmer." *Comm. ACM*, v. 15, n. 10, 1972. p. 859-866. doi:10.1145/355604.361591.

FAGAN, M. E. "Design and code inspections to reduce errors in program development." *IBM Systems J.*, v. 15, n. 3, 1976. p. 182-211.

JEFFRIES, R.; MELNIK, G. "TDD: the art of fearless programming." *IEEE Software*, v. 24, 2007. p. 24-30. doi:10.1109/MS.2007.75.

KANER, C. "An introduction to scenario testing." *Software Testing and Quality Engineering*, out. 2003.

LUTZ, R. R. "Analysing software requirements errors in safety-critical embedded systems." *RE'93*. San Diego, CA: IEEE, 1993. p. 126-133. doi:10.1109/ISRE.1993.324825.

MARTIN, R. C. "Professionalism and test-driven development." *IEEE Software*, v. 24, n. 3, 2007. p. 32-6. doi:10.1109/MS.2007.85.

PROWELL, S. J.; TRAMMELL, C. J.; LINGER, R. C.; POORE, J. H. *Cleanroom software engineering*: technology and process. Reading, MA: Addison-Wesley, 1999.

TAHCHIEV, P.; LEME, F.; MASSOL, V.; GREGORY, G. *JUnit in Action*. 2. ed. Greenwich, CT: Manning Publications, 2010.

WHITTAKER, J. A. *Exploratory software testing*. Boston: Addison-Wesley, 2009.