Ian
SOMMERVILLE

engenharia de SOFTWARE

9ª edição

NING
PEARSON

CAPÍTULO

1 2 3 4 5 6 7 **8** 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26

Testes de software

Objetivos

O objetivo deste capítulo é introduzir testes de software e processos de testes de software. Com a leitura deste capítulo, você:

- compreenderá os estágios de teste durante o desenvolvimento para os testes de aceitação por parte dos usuários de sistema;
- terá sido apresentado a técnicas que ajudam a escolher casos de teste orientados para a descoberta de defeitos de programa;
- compreenderá o desenvolvimento *test-first*, em que você projeta testes antes de escrever o código e os executa automaticamente;
- conhecerá as diferenças importantes entre teste de componentes, de sistemas e de release, e estará ciente dos processos e técnicas de teste de usuário.

Conteúdo

O teste é destinado a mostrar que um programa faz o que é proposto a fazer e para descobrir os defeitos do programa antes do uso. Quando se testa o software, o programa é executado usando dados fictícios. Os resultados do teste são verificados à procura de erros, anomalias ou informações sobre os atributos não funcionais do programa.

O processo de teste tem dois objetivos distintos:

1. Demonstrar ao desenvolvedor e ao cliente que o software atende a seus requisitos. Para softwares customizados, isso significa que deve haver pelo menos um teste para cada requisito do documento de requisitos. Para softwares genéricos, isso significa que deve haver testes para todas as características do sistema, além de suas combinações, que serão incorporadas ao release do produto.
2. Descobrir situações em que o software se comporta de maneira incorreta, indesejável ou de forma diferente das especificações. Essas são consequências de defeitos de software. O teste de defeitos preocupa-se com a eliminação de comportamentos indesejáveis do sistema, tais como panes, interações indesejáveis com outros sistemas, processamentos incorretos e corrupção de dados.

O primeiro objetivo leva a testes de validação, nos quais você espera que o sistema execute corretamente usando determinado conjunto de casos de teste que refletem o uso esperado do sistema. O segundo objetivo leva a testes de defeitos, nos quais os casos de teste são projetados para expor os defeitos. Os casos de teste na busca por defeitos podem ser

...ferentemente obscuros e não precisam refletir com precisão a maneira como o sistema costuma ser usado. Claro que ...existem limites definidos entre essas duas abordagens de teste. Durante os testes de validação, você vai encontrar de... no sistema; durante o teste de defeitos, alguns dos testes mostrarão que o programa corresponde a seus requisitos.

...diagrama da Figura 8.1 pode ajudar a explicar as diferenças entre os testes de validação e o teste de defeitos. Pense ...sistema sendo testado como uma caixa-preta. O sistema aceita entradas a partir de algum conjunto de entradas I e gera ...em um conjunto de saídas O. Algumas das saídas estarão erradas. Estas são as saídas no conjunto O_e, geradas pelo ...ema em resposta a entradas definidas no conjunto I_e. A prioridade nos testes de defeitos é encontrar essas entradas ...definidas no conjunto I_e, pois elas revelam problemas com o sistema. Testes de validação envolvem os testes com entradas ...corretas que estão fora do I_e. Estes estimulam o sistema a gerar corretamente as saídas.

...testes não podem demonstrar se o software é livre de defeitos ou se ele se comportará conforme especificado em ...qualquer situação. É sempre possível que um teste que você tenha esquecido seja aquele que poderia descobrir mais ...problemas no sistema. Como eloquentemente afirmou Edsger Dijkstra, um dos primeiros colaboradores para o desenvol... da engenharia de software (DIJKSTRA et al. 1972):

...*testes podem mostrar apenas a presença de erros, e não sua ausência.*

...teste é parte de um amplo processo de verificação e validação (V&V). Verificação e validação não são a mesma coisa, ...embora sejam frequentemente confundidas.

...Barry Boehm, pioneiro da engenharia de software, expressou sucintamente a diferença entre validação e verificação ...(BOEHM, 1979):

- 'Validação: estamos construindo o produto certo?'
- 'Verificação: estamos construindo o produto da maneira certa?'

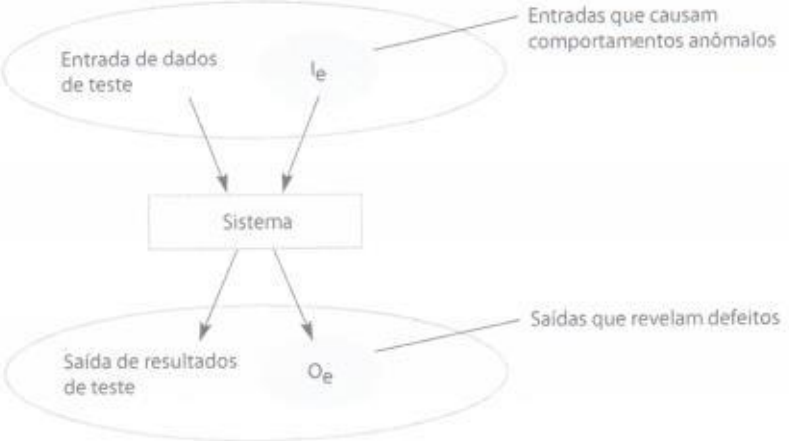
...processos de verificação e validação objetivam verificar se o software em desenvolvimento satisfaz suas especifica... e oferece a funcionalidade esperada pelas pessoas que estão pagando pelo software. Esses processos de verificação ...começam assim que os requisitos estão disponíveis e continuam em todas as fases do processo de desenvolvimento.

...objetivo da verificação é checar se o software atende a seus requisitos funcionais e não funcionais. Validação, no ...entanto, é um processo mais geral. O objetivo da validação é garantir que o software atenda às expectativas do cliente. Ele ...vai além da simples verificação de conformidade com as especificações, pois tenta demonstrar que o software faz o que o ...cliente espera que ele faça. A validação é essencial porque, como já discutido no Capítulo 4, especificações de requisitos ...nem sempre refletem os desejos ou necessidades dos clientes e usuários do sistema.

...objetivo final dos processos de verificação e validação é estabelecer a confiança de que o software está 'pronto para ...propósito'. Isso significa que o sistema deve ser bom o suficiente para seu intuito. O nível de confiança exigido depende ...propósito do sistema, das expectativas dos usuários do sistema e do atual ambiente de marketing:

1. *Finalidade do software.* O mais importante quando se fala sobre software é que ele seja confiável. Por exemplo, ...nível de confiança necessário para um software ser usado para controlar um sistema crítico de segurança é ...muito maior do que o necessário para um protótipo que foi desenvolvido para demonstrar as ideias de novos ...produtos.

Figura 8.1 Um modelo de entrada-saída de teste de programa



2. *Expectativas de usuários.* Devido a suas experiências com softwares defeituosos e não confiáveis, muitos usuários têm baixas expectativas acerca da qualidade de software. Eles não se surpreendem quando o software falha. Quando um novo sistema é instalado, os usuários podem tolerar falhas, pois os benefícios do uso compensam os custos de recuperação de falhas. Nessas situações, você pode não precisar se dedicar muito a testar o software. No entanto, com o amadurecimento do software, os usuários esperam que ele se torne mais confiável e, assim, testes mais completos das próximas versões podem ser necessários.
3. *Ambiente de marketing.* Quando um sistema é comercializado, os vendedores devem levar em conta os produtos concorrentes, o preço que os clientes estão dispostos a pagar por um sistema e os prazos necessários para a entrega desse sistema. Em um ambiente competitivo, uma empresa de software pode decidir lançar um produto antes que ele tenha sido totalmente testado e depurado, pois quer ser a primeira no mercado. Se um software é muito barato, os usuários podem tolerar um baixo nível de confiabilidade.

Assim como testes de software, o processo V&V pode incluir inspeções e revisões. Eles analisam e verificam os requisitos de sistema, modelos de projeto, o código-fonte de programa e até mesmo os testes de sistema propostos. Essas são chamadas técnicas 'estáticas' de V&V, em que você não precisa executar o software para verificá-lo. A Figura 8.2 mostra as inspeções e testes de software que apoiam o V&V em diferentes estágios do processo de software. As setas indicam os estágios do processo em que as técnicas podem ser usadas.

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

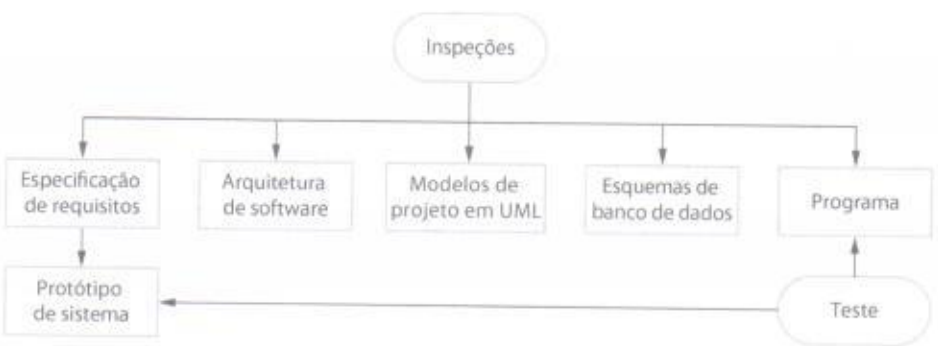
Existem três vantagens da inspeção de software sobre os testes:

1. Durante o teste, erros podem mascarar (esconder) outros erros. Quando um erro conduz saídas inesperadas, você nunca tem certeza se as anomalias seguintes são devidas a um novo erro ou efeitos colaterais do erro original. Como a inspeção é um processo estático, você não precisa se preocupar com as interações entre os erros. Consequentemente, uma sessão única de inspeção pode descobrir muitos erros no sistema.
2. Versões incompletas de um sistema podem ser inspecionadas sem custos adicionais. Se um programa é incompleto, você precisa desenvolver dispositivos de teste especializados para testar as partes disponíveis. Isso, obviamente, aumenta os custos de desenvolvimento do sistema.
3. Bem como a procura por defeitos de programa, uma inspeção pode considerar outros atributos de qualidade de um programa, como a conformidade com os padrões, portabilidade e manutenibilidade. Você pode procurar ineficiências, algoritmos inadequados e um estilo pobre de programação que poderiam tornar o sistema de difícil manutenção e atualização.

As inspeções de programa são uma ideia antiga, e vários estudos e experimentos demonstraram que as inspeções são mais eficazes na descoberta de defeitos do que os testes de programa. Fagan (1986) relatou que mais de 60% dos erros em um programa podem ser detectados por meio de inspeções informais de programa. No processo Cleanroom (PROWELL et al., 1999), afirma-se que mais de 90% dos defeitos podem ser descobertos em inspeções de programas.

No entanto, as inspeções não podem substituir os testes de software. As inspeções não são boas para descobrir defeitos que surgem devido a interações inesperadas entre diferentes partes de um programa, problemas de *timing* ou com o desempenho do sistema. Além disso, pode ser difícil e caro montar uma equipe de inspeção, especialmente em pequenas empresas ou grupos de desenvolvimento, já que todos os membros da equipe também podem ser desenvolvedores de

Figura 8.2 Testes de inspeção



software. No Capítulo 24 (Gerenciamento de qualidade), discuto revisões e inspeções com mais detalhes. A análise estática automatizada, em que o texto-fonte de um programa é automaticamente analisado para descobrir anomalias, é explicada no Capítulo 15. Neste capítulo, o foco está nos testes e processos de testes.

A Figura 8.3 é um modelo abstrato do processo 'tradicional' de testes, como usado no desenvolvimento dirigido a planos. Os casos de teste são especificações das entradas para o teste e da saída esperada 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, os dados de teste podem ser gerados automaticamente, mas a geração automática de casos de teste é impossível, pois as pessoas que entendem o propósito do sistema devem ser envolvidas para especificar os resultados esperados. No entanto, a execução do teste pode ser automatizada. Os resultados esperados são automaticamente comparados aos resultados previstos, por isso não há necessidade de uma pessoa para procurar erros e anomalias na execução dos testes.

Geralmente, o sistema de software comercial tem de passar por três estágios de teste:

1. Testes em desenvolvimento, em que o sistema é testado durante o desenvolvimento para descobrir *bugs* e defeitos. Projetistas de sistemas e programadores podem estar envolvidos no processo de teste.
2. Testes de release, em que uma equipe de teste independente testa uma versão completa do sistema antes que ele seja liberado para os usuários. O objetivo dos testes de release é verificar se o sistema atende aos requisitos dos *stakeholders* de sistema.
3. Testes de usuário, em que os usuários ou potenciais usuários de um sistema testam o sistema em seu próprio ambiente. Para produtos de software, o 'usuário' pode ser um grupo de marketing interno, que decidirá se o software pode ser comercializado, liberado e vendido. Os testes de aceitação são um tipo de teste de usuário no qual o cliente testa formalmente o sistema para decidir se ele deve ser aceito por parte do fornecedor do sistema ou se é necessário um desenvolvimento adicional.

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 suas expectativas; ele anota e reporta as discrepâncias aos desenvolvedores do programa. Em testes automatizados, os testes são codificados em um programa que é executado cada vez que o sistema em desenvolvimento é testado. Essa forma é geralmente mais rápida que o teste manual, especialmente quando envolve testes de regressão — reexecução de testes anteriores para verificar se as alterações no programa não introduziram novos *bugs*.

O uso de testes automatizados tem aumentado consideravelmente nos últimos anos. Entretanto, os testes nunca poderão ser totalmente automatizados, já que testes automáticos só podem verificar se um programa faz aquilo a que é proposto. É praticamente impossível usar testes automatizados para testar os sistemas que dependem de como as coisas estão (por exemplo, uma interface gráfica de usuário), ou para testar se um programa não tem efeitos colaterais indesejados.

8.1 Testes de desenvolvimento

Testes de desenvolvimento incluem todas as atividades de testes que são realizadas pela equipe de desenvolvimento do sistema. O testador do software geralmente é o programador que o desenvolveu, embora nem sempre seja assim. Alguns processos de desenvolvimento usam programadores/testadores em pares (CUSAMANO e SELBY, 1998), nos quais cada programador tem um testador associado para desenvolver os testes e ajudar no processo. Para sistemas críticos, um processo mais formal pode ser usado por um grupo de testes independente

Figura 8.3 Um modelo do processo de teste de software

dentro da equipe de desenvolvimento. Eles são responsáveis pelo desenvolvimento de testes e pela manutenção de registros detalhados de seus resultados.

Durante o desenvolvimento, o teste pode ocorrer em três níveis de granularidade:

1. Teste unitário, em que as unidades individuais de programa ou classes de objetos são testadas individualmente. Testes unitários devem centrar-se em testar a funcionalidade dos objetos ou métodos.
2. Teste de componentes, em que várias unidades individuais são integradas para criar componentes compostos. Testes de componentes devem centrar-se em testar as interfaces dos componentes.
3. Teste de sistema, em que alguns ou todos os componentes de um sistema estão integrados e o sistema é testado como um todo. O teste de sistema deve centrar-se em testar as interações entre os componentes.

Testes de desenvolvimento são essencialmente um processo de teste de defeitos, em que o objetivo do teste é descobrir *bugs* no software. Normalmente, são intercalados com a depuração — o processo de localizar problemas com o código e alterar o programa para corrigir esses problemas.

8.1.1 Teste unitário

O teste unitário é o processo de testar os componentes de programa, como métodos ou classes de objeto. As funções individuais ou métodos são o tipo mais simples de componente. Seus testes devem ser chamadas para essas rotinas com parâmetros diferentes de entrada. Você pode usar as abordagens para projeto de casos de teste (discutidas na Seção 8.1.2), para projetar testes de funções ou métodos.

Quando você está testando as classes de objeto, deve projetar os testes para fornecer uma cobertura de todas as características do objeto. Isso significa que você deve:

- Testar todas as operações associadas ao objeto;
- Definir e verificar o valor de todos os atributos associados ao objeto;
- Colocar o objeto em todos os estados possíveis, o que significa simular todos os eventos que causam mudanças de estado.

Considere, por exemplo, o objeto EstaçãoMeteorológica do exemplo discutido no Capítulo 7. A interface desse objeto é mostrada na Figura 8.4. Ela tem um único atributo, que é seu identificador. Este é uma constante, definida quando a estação meteorológica é instalada. Portanto, você só precisa de um teste que verifique se ele foi configurado corretamente. Você precisa definir casos de teste para todos os métodos associados ao objeto, como relatarClima, relatarStatus etc. Preferencialmente, você deve testar métodos de forma isolada, mas, em alguns casos, algumas sequências de teste são necessárias. Por exemplo, para testar o método que desliga os instrumentos da estação meteorológica (desligar), você precisa ter executado o método de reiniciar.

A generalização ou herança faz testes de classes de objeto mais complicados. Você não pode simplesmente testar uma operação na classe em que ela está definida e assumir que funcionará corretamente nas subclasses que herdam a operação. A operação que é herdada pode fazer suposições sobre outras operações e atributos. Essas operações podem não ser válidas em algumas subclasses que herdam a operação. Portanto, é necessário testar a operação herdada em todos os contextos de uso.

Para testar os estados da estação meteorológica, usamos um modelo de estado, como o mostrado no capítulo anterior, na Figura 7.7. Usando esse modelo, é possível identificar as sequências de transições de estado que pre-

Figura 8.4 A interface do objeto EstaçãoMeteorológica

cisam ser testadas e definir as sequências de eventos para forçar essas transições. Em princípio, você deve testar cada sequência de transição de estado possível, embora na prática isso possa ser muito custoso. Exemplos de sequências de estado que devem ser testados na estação meteorológica incluem:

Desligar → Executar → Desligar

Configurar → Executar → Testar → Transmitir → Executar

Executar → Coletar → Executar → Resumir → Transmitir → Executar

Sempre que possível, você deve automatizar os testes unitários. Em testes unitários automatizados, pode-se usar um *framework* de automação de teste (como JUnit) para escrever e executar testes de seu programa. *Frameworks* de testes unitários fornecem classes de teste genéricas que você pode estender para criar casos de teste específicos. Eles podem, então, executar todos os testes que você implementou e informar, muitas vezes por meio de alguma interface gráfica, sobre o sucesso ou o fracasso dos testes. Um conjunto inteiro de testes frequentemente pode ser executado em poucos segundos; assim, é possível executar todos os testes cada vez que é feita uma alteração no programa.

Um teste automatizado tem três partes:

1. Uma parte de configuração, em que você inicia o sistema com o caso de teste, ou seja, as entradas e saídas esperadas.
2. Uma parte de chamada, quando você chama o objeto ou método a ser testado.
3. Uma parte de afirmação, em que você compara o resultado da chamada com o resultado esperado. Se a afirmação avaliada for verdadeira, o teste foi bem-sucedido; se for falsa, ele falhou.

Às vezes, o objeto que você está testando tem dependências em outros objetos que podem não ter sido escritos ou que atrasam o processo de teste quando são usados. Por exemplo, se o objeto chama um banco de dados, isso pode implicar um processo lento de instalação antes que ele possa ser usado. Nesses casos, você pode decidir usar um *mock object*. *Mock objects* são objetos com a mesma interface que os objetos externos usados para simular sua funcionalidade. Portanto, um *mock object* que simula um banco de dados pode ter apenas uns poucos itens organizados em um vetor. Eles podem ser acessados rapidamente, sem os *overheads* de chamar um banco de dados e acessar os discos. Da mesma forma, *mock objects* podem ser usados para simular operações anormais ou eventos raros. Por exemplo, se o sistema se destina a agir em determinados momentos do dia, seu *mock object* pode simplesmente retornar naqueles momentos, independentemente do horário real.

8.1.2 Escolha de casos de teste unitário

O teste é custoso e demorado, por isso é importante que você escolha casos efetivos de teste unitário. A efetividade, nesse caso, significa duas coisas:

1. Os casos de teste devem mostrar que, quando usado como esperado, o componente que você está testando faz o que ele é proposto a fazer.
2. Se houver defeitos nos componentes, estes devem ser revelados por casos de teste.

Você deve, portanto, escrever dois tipos de casos de teste. O primeiro deve refletir o funcionamento normal de um programa e deve mostrar que o componente funciona. Por exemplo, se você está testando um componente que cria e inicia um novo registro de paciente, seu caso de teste deve mostrar que o registro existe no banco de dados e que seus campos foram criados como especificados. O outro tipo de caso de teste deve ser baseado em testes de experiência, dos quais surgem os problemas mais comuns. Devem-se usar entradas anormais para verificar que estes são devidamente processados e que não fazem o componente falhar.

Discuto, aqui, duas estratégias que podem ser eficazes para ajudar você a escolher casos de teste. São elas:

1. Teste de partição, em que você identifica os grupos de entradas que possuem características comuns e devem ser tratados da mesma maneira. Você deve escolher os testes dentro de cada um desses grupos.
2. Testes baseados em diretrizes, em que você usa as diretrizes de testes para escolher casos de teste. Essas diretrizes refletem a experiência anterior dos tipos de erros que os programadores cometem frequentemente no desenvolvimento de componentes.

É comum os dados de entrada e os resultados de saída de um software caírem em diferentes classes com características comuns. Exemplos dessas classes são números positivos, números negativos e seleções no *menu*. Os programas geralmente se comportam de forma comparável para todos os membros de uma classe. Ou seja, se

você testar um programa que faz um cálculo e requer dois números positivos, você deve esperar que o programa se comporte da mesma forma para todos os números positivos.

Devido a esse comportamento equivalente, essas classes são também chamadas de partições ou domínios de equivalência (BEZIER, 1990). Uma abordagem sistemática para projetar casos de teste baseia-se na identificação de todas as partições de entrada e saída para um sistema ou componente. Os casos de teste são projetados para que as entradas ou saídas estejam dentro dessas partições. Um teste de partição pode ser usado para projetar casos de teste para ambos, sistemas e componentes.

Na Figura 8.5, a elipse maior 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 que está sendo testado deve processar todos os membros de uma partição de equivalência de entrada da mesma forma. As partições de equivalência de saída são partições dentro das quais todas as saídas têm algo em comum. Às vezes, não há um mapeamento 1:1 entre as partições de equivalência de entrada e de saída. No entanto, isso nem sempre é o caso. Você pode precisar definir uma partição de equivalência de entrada separada, na qual a única característica comum das entradas é que geram saídas dentro da mesma partição de saída. A área sombreada na elipse esquerda representa as entradas inválidas. A área sombreada na elipse da direita representa as exceções que podem ocorrer (ou seja, respostas às entradas inválidas).

Depois de ter identificado um conjunto de partições, você escolhe os casos de teste de cada uma. Uma boa prática para a seleção do caso de teste é escolher casos de teste sobre os limites das partições, além de casos perto do ponto médio da partição. A razão disso é que, no desenvolvimento de um sistema, os projetistas e programadores tendem a considerar os valores típicos de entrada. Você pode testá-los escolhendo o ponto médio da partição. Os valores-limite são frequentemente atípicos (por exemplo, zero pode comportar-se de maneira diferente de outros números não negativos) e por vezes são negligenciados pelos desenvolvedores. Falhas de programa ocorrem frequentemente ao se processarem esses valores atípicos.

As partições são identificadas usando-se a especificação de programa ou a documentação de usuário, bem como a partir da experiência com a qual você prevê as classes de valor de entrada prováveis de detectar erros. Por exemplo, digamos que uma especificação de programa defina que o programa aceita entradas de 4 a 8 que sejam valores inteiros de cinco dígitos superiores a 10.000. Você pode usar essas informações para identificar as partições de entrada e de possíveis valores de entrada de teste. Estes são mostrados na Figura 8.6.

Quando você usa a especificação de um sistema para identificar as partições de equivalência, isso é chamado 'teste de caixa-preta'. Nesse caso, você não precisa de nenhum conhecimento de como funciona o sistema. No entanto, pode ser útil para suplementar os testes de caixa-preta um 'teste de caixa-branca', em que você pode olhar o código do programa para encontrar outros testes possíveis. Por exemplo, seu código pode incluir exceções para lidar com entradas incorretas. Você pode usar esse conhecimento para identificar as 'partições de exceção' — diferentes intervalos, nos quais o mesmo tratamento de exceção deve ser aplicado.

Particionamento de equivalência é uma abordagem eficaz para testes, pois ajuda a explicar os erros que os programadores costumam cometer ao processar as entradas nos limites das partições. Você também pode usar

Figura 8.5 Particionamento de equivalência

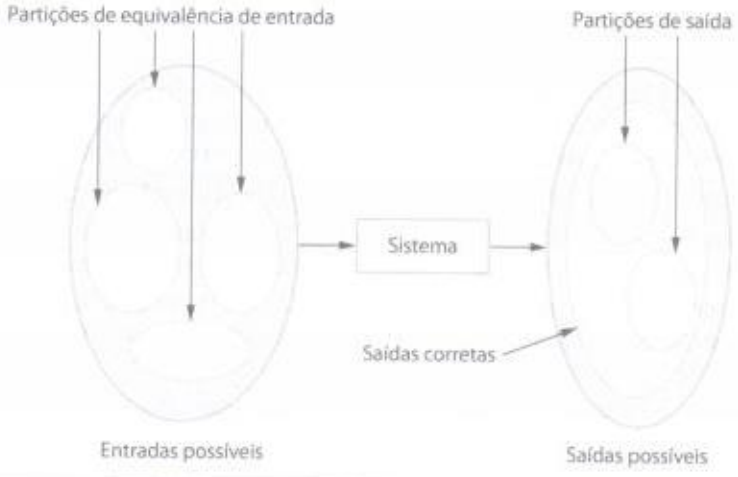


Figura 8.6 Partições de equivalência

diretrizes de teste para ajudar a escolher casos de teste. Diretrizes encapsulam o conhecimento de quais tipos de casos de teste são eficazes para descobrir erros. Por exemplo, quando você está testando programas com sequências, vetores ou listas, as diretrizes que podem ajudar a revelar defeitos incluem:

1. Testar software com as sequências que possuem apenas um valor. Programadores, em geral, pensam em sequências compostas de vários valores e, às vezes, incorporam essa hipótese em seus programas. Consequentemente, se apresentado com uma sequência de valor único, um programa pode não funcionar corretamente.
2. Usar sequências diferentes de tamanhos diferentes em testes diferentes. Isso diminui as chances de um programa com defeitos produzir acidentalmente uma saída correta, por causa de alguma característica acidental na entrada.
3. Derivar testes de modo que o primeiro elemento, o do meio e o último da sequência sejam acessados. Essa abordagem revela problemas nos limites de partição.

O livro de Whittaker (2002) inclui muitos exemplos de diretrizes que podem ser usadas no projeto de casos de teste. Algumas das diretrizes mais gerais que ele sugere são:

- Escolha entradas que forcem o sistema a gerar todas as mensagens de erro;
- Projete entradas que causem *overflow* de *buffers* de entrada;
- Repita a mesma entrada ou uma série de entradas inúmeras vezes;
- Obrigue a geração de saídas inválidas;
- Obrigue os resultados de cálculos a serem muito grandes ou muito pequenos.

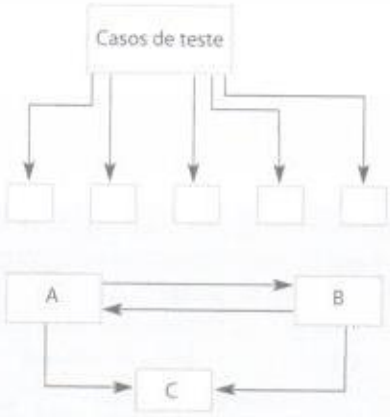
Conforme ganha experiência com o teste, você pode desenvolver suas próprias diretrizes de como escolher casos mais eficazes. Na próxima seção deste capítulo, dou mais exemplos de diretrizes de testes.

8.1.3 Teste de componente

Frequentemente, os componentes do software são compostos de diversos objetos que interagem. Por exemplo, no sistema de estação meteorológica, o componente de reconfiguração inclui objetos que lidam com cada aspecto da reconfiguração. Você pode acessar a funcionalidade desses objetos por meio da interface de componente definida. Testes de componentes compostos devem centrar-se em mostrar que a interface de componente se comporta de acordo com sua especificação. Você pode assumir que os testes unitários sobre os objetos individuais dentro do componente já foram concluídos.

A Figura 8.7 ilustra a ideia de teste de interface de componente. Suponha que os componentes A, B e C foram integrados para criar um componente maior ou subsistema. Os casos de teste não são aplicados aos componentes individuais, mas sim à interface de componente criada pela combinação desses componentes. Erros de interface no componente composto podem não ser detectáveis por meio de testes em objetos individuais, pois esses erros resultam de interações entre os objetos do componente.

Figura 8.7 Teste de interface



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

1. *Interfaces de parâmetro*. São as interfaces nas quais as referências de dados ou, às vezes, de função, são passadas de um componente para outro. Métodos de um objeto têm uma interface de parâmetro.
2. *Interfaces de memória compartilhada*. São as interfaces nas quais um bloco de memória é compartilhado entre os componentes. Os dados são colocados na memória por um subsistema e recuperados a partir daí por outros subsistemas. Esse tipo de interface é frequentemente usado em sistemas embutidos, em que os sensores criam dados que são recuperados e processados por outros componentes do sistema.
3. *Interfaces de procedimento*. São as interfaces nas quais um componente encapsula um conjunto de procedimentos que podem ser chamados por outros componentes. Objetos e componentes reusáveis têm esse tipo de interface.
4. *Interface de passagem de mensagem*. São as interfaces nas quais um componente solicita um serviço de outro componente, passando-lhe uma mensagem. Uma mensagem de retorno inclui os resultados da execução do serviço. Alguns sistemas orientados a objetos têm esse tipo de interface, como nos sistemas cliente-servidor.

Erros de interface são uma das formas mais comuns de erros em sistemas complexos (LUTZ, 1993). Esses erros são classificados em três classes:

- *Mau uso de interface*. Um componente chamador chama outro componente e comete um erro no uso de sua interface. Esse tipo de erro é comum com interfaces de parâmetro, em que os parâmetros podem ser de tipo errado ou ser passados na ordem errada, ou o número errado de parâmetros pode ser passado.
- *Mau-entendimento de interface*. Um componente chamador desconhece a especificação da interface do componente chamado e faz suposições sobre seu comportamento. O componente chamado não se comporta conforme o esperado, causando um comportamento inesperado no componente de chamada. Por exemplo, um método de busca binária pode ser chamado com um parâmetro que é um vetor não ordenado. A busca então falharia.
- *Erros de timing*. Eles ocorrem em sistemas em tempo real que usam uma memória compartilhada ou uma interface de passagem de mensagens. O produtor e o consumidor de dados podem operar em velocidades diferentes. A menos que se tome um cuidado especial no projeto da interface, o consumidor pode acessar uma informação desatualizada, porque o produtor da informação não atualizou as informações da interface compartilhada.

Testes para defeitos de interface são difíceis, pois alguns defeitos de interface só podem manifestar-se sob condições não usuais. Por exemplo, digamos que um objeto implementa uma fila como uma estrutura de dados de comprimento fixo. Um objeto chamador pode supor que a fila é implementada como uma estrutura de dados infinita e pode não verificar o *overflow* de fila quando um item for inserido. Essa condição só pode ser detectada durante os testes, projetando casos de teste que forçam a fila a transbordar e causar estouro, o que corrompe o comportamento do objeto de forma detectável.

Outro problema pode surgir por causa das interações entre defeitos em diferentes módulos ou objetos. Defeitos em um objeto só podem ser detectados quando outro objeto se comporta de maneira inesperada. Por exemplo, um objeto pode chamar outro para receber algum serviço e assumir que a resposta está correta. Se o serviço chamado está defeituoso de algum modo, o valor retornado pode ser válido, porém incorreto. Isso não é imediatamente detectado, mas só se torna evidente quando algum processamento posterior dá errado.

Algumas diretrizes gerais para os testes de interface são:

1. Examine o código a ser testado e liste, explicitamente, cada chamada para um componente externo. Projete um conjunto de testes em que os valores dos parâmetros para os componentes externos estão nos extremos de suas escalas. Esses valores extremos são mais suscetíveis a revelar inconsistências de interface.
2. Nos casos em que os ponteiros são passados por meio de uma interface, sempre teste a interface com os parâmetros de ponteiros nulos.
3. Nos casos em que um componente é chamado por meio de uma interface de procedimento, projete testes que deliberadamente causem uma falha no componente. Diferentes hipóteses de falhas são um dos equívocos mais comuns de especificação.
4. Use testes de estresse em sistemas de passagem de mensagem. Isso significa que você deve projetar testes que gerem muito mais mensagens do que seria provável ocorrer na prática. Essa é uma maneira eficaz de revelar problemas de *timing*.
5. Nos casos em que vários componentes interagem por meio de memória compartilhada, desenvolva testes que variam a ordem em que esses componentes são ativados. Esses testes podem revelar as suposições implícitas feitas pelo programador sobre a ordem na qual os dados compartilhados são produzidos e consumidos.

Inspeções e avaliações podem ser mais eficazes do que os testes para descobrir erros de interface. As inspeções podem concentrar-se em interfaces de componentes e questões sobre o comportamento da interface levantadas durante o processo de inspeção. Uma linguagem fortemente tipada, como Java, permite que o compilador apreenda muitos erros de interface. Analisadores estáticos (veja o Capítulo 15) podem detectar uma vasta gama de erros de interface.

8.1.4 Teste de sistema

O teste de sistema, durante o desenvolvimento, envolve a integração de componentes para criação de uma versão do sistema e, em seguida, o teste do sistema integrado. O teste de sistema verifica se os componentes são compatíveis, se interagem corretamente e transferem os dados certos no momento certo, por suas interfaces. Certamente, sobrepõem-se ao teste de componente, mas existem duas diferenças importantes:

1. Durante o teste de sistema, os componentes reusáveis que tenham sido desenvolvidos separadamente e os sistemas de prateleira podem ser integrados com componentes recém-desenvolvidos. Então, o sistema completo é testado.
2. Nesse estágio, componentes desenvolvidos por diferentes membros da equipe ou grupos podem ser integrados. O teste de sistema é um processo coletivo, não individual. Em algumas empresas, o teste de sistema pode envolver uma equipe independente, sem participação de projetistas e programadores.

Quando você integra componentes para criar um sistema, obtém um comportamento emergente. Isso significa que alguns elementos da funcionalidade do sistema só se tornam evidentes quando colocados juntos. Esse comportamento emergente pode ser planejado e precisa ser testado. Por exemplo, você pode integrar um componente de autenticação com um componente que atualiza as informações. Assim, você tem uma característica do sistema que restringe as informações de atualização para usuários autorizados. Contudo, às vezes, o comportamento emergente não é planejado ou desejado. É preciso desenvolver testes que verifiquem se o sistema está fazendo apenas o que ele supostamente deve fazer.

Portanto, o teste do sistema deve centrar-se em testar as interações entre os componentes e objetos que compõem um sistema. Você também pode testar componentes ou sistemas reusáveis para verificar se, quando integrados com novos componentes, eles funcionam como o esperado. Esse teste de interação deve descobrir *bugs* de componente que só são revelados quando um componente é usado por outros componentes do sistema. O teste de interação também ajuda a encontrar equívocos dos desenvolvedores de componentes sobre outros componentes do sistema.

Por causa de seu foco na interação, o teste baseado em caso de uso é uma abordagem eficaz para testes de sistema. Normalmente, cada caso de uso é implementado por vários componentes ou objetos do sistema. Testar os casos de uso força essas interações a ocorrerem. Se você desenvolveu um diagrama de sequência para modelar a implementação dos casos de uso, poderá ver os objetos ou componentes envolvidos na interação.

Para ilustrar isso, uso um exemplo do sistema da estação meteorológica no deserto, em que é solicitado à estação meteorológica o relatório resumido de dados meteorológicos para um computador remoto. Esse caso de uso está descrito no Quadro 7.1. A Figura 8.8 (que é uma cópia da Figura 7.6) mostra a sequência de operações na estação meteorológica quando esta responde a um pedido para coletar dados para o sistema de mapeamento. Você pode usar esse diagrama para identificar as operações que serão testadas e para ajudar a projetar os casos de teste para a execução. Portanto, a emissão de um pedido de um relatório resultará na execução do seguinte *thread* de métodos:

ComunicSat:solicitar → EstaçãoMeteorológica:relatarClima → LinkComunic:Obter(sumário) → DadosMeteorológicos:resumir

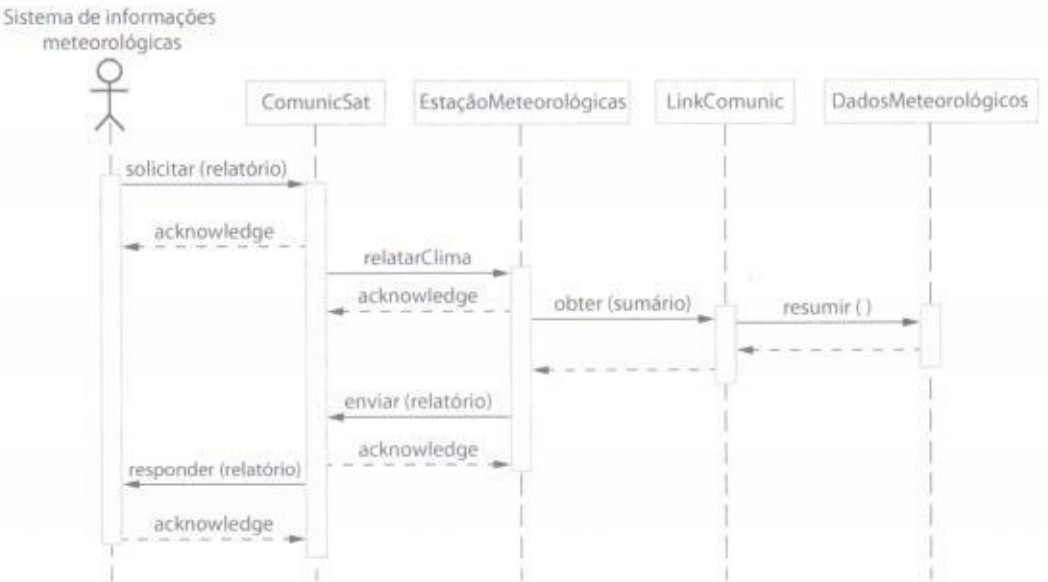
O diagrama de sequência ajuda a projetar os casos de teste específicos que você necessita, pois mostra quais são as entradas necessárias e que saídas são criadas:

1. Uma entrada de um pedido de relatório deve ter um *acknowledge* associado. Um relatório deve ser devolvido a partir da solicitação. Durante o teste, você deve criar dados resumidos que possam ser usados para verificar se o relatório está organizado corretamente.
2. Uma solicitação de entrada para um relatório na EstaçãoMeteorológica resulta em um relatório resumido que será gerado. Você pode fazer esse teste de forma isolada, criando dados brutos correspondentes ao resumo que você preparou para o teste de ComunicSat e verificar se o objeto EstaçãoMeteorológica produz essa síntese corretamente. Esses dados brutos também são usados para testar o objeto DadosMeteorológicos.

É claro que na Figura 8.8 eu simplifiquei o diagrama de sequência, para que ele não mostre as exceções. Um teste completo de caso de uso/cenário deve levar em consideração as exceções e garantir que os objetos as tratem corretamente.

Para a maioria dos sistemas, é difícil saber o quanto o teste de sistema é essencial e quando você deve parar de testar. É impossível fazer testes exaustivos, em que cada sequência possível de execução do programa seja testada. Assim, os testes precisam ser baseados em um subconjunto possível de casos de teste. Idealmente, as empresas de software deveriam ter políticas para a escolha desse grupo. Essas políticas podem ser baseadas em políticas de teste gerais, como uma política em que todas as declarações do programa devem ser executadas pelo menos uma vez. Como alternativa, podem basear-se na experiência de uso do sistema e centrar-se em testes de características do sistema operacional. Por exemplo:

Figura 8.8 Diagrama de sequência de coletar dados meteorológicos



1. Todas as funções do sistema acessadas por meio de *menus* devem ser testadas.
2. Combinações de funções (por exemplo, a formatação do texto) acessadas por meio de *menu* devem ser testadas.
3. Nos casos em que a entrada do usuário é fornecida, todas as funções devem ser testadas com entradas corretas e incorretas.

A partir de experiências com importantes produtos de software, como processadores de texto ou planilhas, fica evidente que durante o teste de produtos costumam ser usadas diretrizes semelhantes. Quando os recursos do software são usados de forma isolada, eles trabalham normalmente. Como explica Whittaker (2002), os problemas surgem quando as combinações das características menos usadas não são testadas em conjunto. Ele dá o exemplo de como, em um processador de texto comumente usado, usar as notas de rodapé com um *layout* de várias colunas causa erros no *layout* do texto.

Testes automatizados do sistema geralmente são mais difíceis do que testes automatizados de unidades ou de componentes. Testes automatizados de unidade baseiam-se em prever as saídas, e, em seguida, codificar essas previsões em um programa. A previsão é, então, comparada com o resultado. No entanto, o mais importante na aplicação de um sistema pode ser a geração de saídas que sejam grandes ou que não possam ser facilmente previstas. Você pode ser capaz de analisar uma saída e verificar sua credibilidade sem necessariamente ser capaz de criá-la com antecipação.

8.2 Desenvolvimento dirigido a testes

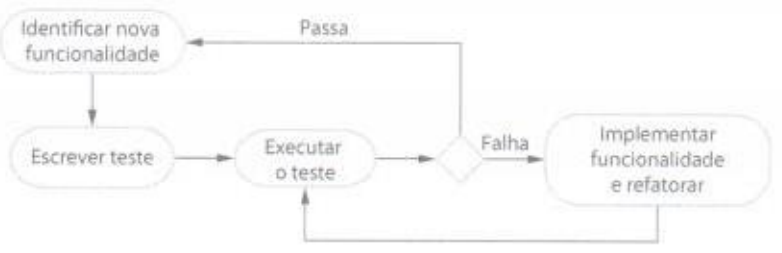
O desenvolvimento dirigido a testes (TDD, do inglês *Test-Driven Development*) é uma abordagem para o desenvolvimento de programas em que se intercalam testes e desenvolvimento de código (BECK, 2002; JEFFRIES e MELNIK, 2007). Essencialmente, você desenvolve um código de forma incremental, em conjunto com um teste para esse incremento. Você não caminha para o próximo incremento até que o código desenvolvido passe no teste. O desenvolvimento dirigido a testes foi apresentado como parte dos métodos ágeis, como o Extreme Programming. No entanto, ele também pode ser usado em processos de desenvolvimento dirigido a planos.

O processo fundamental de TDD é mostrado na Figura 8.9. As etapas do processo são:

1. Você começa identificando o incremento de funcionalidade necessário. Este, normalmente, deve ser pequeno e implementável em poucas linhas de código.
2. Você escreve um teste para essa funcionalidade e o implementa como um teste automatizado. Isso significa que o teste pode ser executado e relatará se passou ou falhou.
3. Você, então, executa o teste, junto com todos os outros testes implementados. Inicialmente, você não terá implementado a funcionalidade, logo, o novo teste falhará. Isso é proposital, pois mostra que o teste acrescenta algo ao conjunto de testes.
4. Você, então, implementa a funcionalidade e executa novamente o teste. Isso pode envolver a refatoração do código existente para melhorá-lo e adicionar um novo código sobre o que já está lá.
5. Depois que todos os testes forem executados com sucesso, você caminha para implementar a próxima parte da funcionalidade.

Um ambiente de testes automatizados, como o ambiente JUnit, que suporta o teste de programa Java (MASSOL e HUSTED, 2003), é essencial para o TDD. Como o código é desenvolvido em incrementos muito pequenos, você precisa ser capaz de executar todos os testes cada vez que adicionar funcionalidade ou refatorar o programa.

Figura 8.9 Desenvolvimento dirigido a testes



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 rodar centenas de testes separados em poucos segundos.

Um argumento forte a favor do desenvolvimento dirigido a testes é que ele ajuda os programadores a clarear suas ideias sobre o que um segmento de código supostamente deve fazer. Para escrever um teste, você precisa entender a que ele se destina, e como esse entendimento faz que seja mais fácil escrever o código necessário. Certamente, se você tem conhecimento ou compreensão incompleta, o desenvolvimento dirigido a testes não ajudará. Se você não sabe o suficiente para escrever os testes, não vai desenvolver o código necessário. Por exemplo, se seu cálculo envolve divisão, você deve verificar se não está dividindo o número por zero. Se você se esquecer de escrever um teste para isso, então o código para essa verificação nunca será incluído no programa.

Além de um melhor entendimento do problema, outros benefícios do desenvolvimento dirigido a testes são:

1. *Cobertura de código*. Em princípio, todo segmento de código que você escreve deve ter pelo menos um teste associado. Portanto, você pode ter certeza de que todo o código no sistema foi realmente executado. Cada código é testado enquanto está sendo escrito; assim, os defeitos são descobertos no início do processo de desenvolvimento.
2. *Teste de regressão*. Um conjunto de testes é desenvolvido de forma incremental enquanto um programa é desenvolvido. Você sempre pode executar testes de regressão para verificar se as mudanças no programa não introduziram novos *bugs*.
3. *Depuração simplificada*. Quando um teste falha, a localização do problema deve ser óbvia. O código recém-escrito precisa ser verificado e modificado. Você não precisa usar as ferramentas de depuração para localizar o problema. Alguns relatos de uso de desenvolvimento dirigido a testes sugerem que, em desenvolvimento dirigido a testes, quase nunca é necessário usar um sistema automatizado de depuração (MARTIN, 2007).
4. *Documentação de sistema*. Os testes em si mesmos agem como uma forma de documentação que descreve o que o código deve estar fazendo. Ler os testes pode tornar mais fácil a compreensão do código.

Um dos benefícios mais importantes de desenvolvimento dirigido a testes é que ele reduz os custos dos testes de regressão. O teste de regressão envolve a execução de conjuntos de testes que tenham sido executados com sucesso, após as alterações serem feitas em um sistema. O teste de regressão verifica se essas mudanças não introduziram novos *bugs* no sistema e se o novo código interage com o código existente conforme o esperado. O teste de regressão é muito caro e geralmente impraticável quando um sistema é testado manualmente, pois os custos com tempo e esforço são muito altos. Em tais situações, você precisa tentar escolher os testes mais relevantes para executar novamente, e é fácil perder testes importantes.

No entanto, testes automatizados, fundamentais para o desenvolvimento *test-first*, reduzem drasticamente os custos com testes de regressão. Os testes existentes podem ser executados novamente de forma rápida e barata. Após se fazer uma mudança para um sistema em desenvolvimento *test-first*, todos os testes existentes devem ser executados com êxito antes de qualquer funcionalidade ser adicionada. Como um programador, você precisa ter certeza de que a nova funcionalidade não tenha causado ou revelado problemas com o código existente.

O desenvolvimento dirigido a testes é de maior utilidade no desenvolvimento de softwares novos, em que a funcionalidade seja implementada no novo código ou usando bibliotecas-padrão já testadas. Se você estiver reusando componentes de código ou sistemas legados grandes, você precisa escrever testes para esses sistemas como um todo. O desenvolvimento dirigido a testes também pode ser ineficaz em sistemas multi-*threaded*. Os *threads* diferentes podem ser intercalados em tempos diferentes, em execuções diferentes, e isso pode produzir resultados diferentes.

Se você usa o desenvolvimento dirigido a testes, ainda precisa de um processo de teste de sistema para validar o sistema, isto é, para verificar se ele atende aos requisitos de todos os *stakeholders*. O teste de sistema também testa o desempenho, a confiabilidade, e verifica se o sistema não faz coisas que não deveria, como produzir resultados indesejados etc. Andrea (2007) sugere como ferramentas de teste podem ser estendidas para integrar alguns aspectos do teste de sistema com TDD.

O desenvolvimento dirigido a testes revelou-se uma abordagem de sucesso para projetos de pequenas e médias empresas. Geralmente, os programadores que adotaram essa abordagem estão satisfeitos com ela e acham que é uma maneira mais produtiva de desenvolver softwares (JEFFRIES e MELNIK, 2007). Em alguns experimentos, foi mostrado que essa abordagem gera melhorias na qualidade do código; em outros, os resultados foram inconclusivos; no entanto, não existem evidências de que o TDD leve à redução da qualidade de código.

8.3 Testes de release

Teste de release é o processo de testar um release particular de um sistema que se destina para uso fora da equipe de desenvolvimento. Geralmente, o release de sistema é para uso dos clientes e usuários. Em um projeto complexo, no entanto, ele pode ser para as outras equipes que estão desenvolvendo sistemas relacionados. Para produtos de software, o release pode ser para gerenciamento de produtos que, em seguida, o prepara para a venda.

Existem duas diferenças importantes entre o teste de release e o teste de sistema durante o processo de desenvolvimento:

1. Uma equipe separada, que não esteve envolvida no desenvolvimento do sistema, deve ser responsável pelo teste de release.
2. Testes de sistema feitos pela equipe de desenvolvimento devem centrar-se na descoberta de *bugs* no sistema (teste de defeitos). O objetivo do teste de release é verificar se o sistema atende a seus requisitos e é bom o suficiente para uso externo (teste de validação).

O objetivo principal do processo de teste de release é convencer o fornecedor do sistema de que esse sistema é bom o suficiente para uso. Se assim for, o sistema poderá ser lançado como um produto ou entregue aos clientes. Portanto, o teste de release precisa mostrar que o sistema oferece a funcionalidade, o desempenho e a confiança especificados e que não falhará durante o uso normal. Deve levar em conta todos os requisitos de sistema, e não apenas os requisitos de usuários finais do sistema.

Teste de release costuma ser um processo de teste de caixa-preta, no qual os testes são derivados da especificação de sistema. O sistema é tratado como uma caixa-preta cujo comportamento só pode ser determinado por meio do estudo das entradas e saídas relacionadas. Outro nome para isso é 'teste funcional', assim chamado porque o testador só está preocupado com a funcionalidade, e não com a implementação do software.

8.3.1 Testes baseados em requisitos

Um dos princípios gerais das boas práticas de engenharia de requisitos é que os requisitos devem ser testáveis, isto é, o requisito deve ser escrito de modo que um teste possa ser projetado para ele. Um testador pode, então, verificar se o requisito foi satisfeito. Testes baseados em requisitos, portanto, são uma abordagem sistemática para projeto de casos de teste em que você considera cada requisito e deriva um conjunto de testes para eles. Testes baseados em requisitos são mais uma validação do que um teste de defeitos — você está tentando demonstrar que o sistema implementou adequadamente seus requisitos.

Por exemplo, considere os requisitos relacionados ao MHC-PMS (apresentado no Capítulo 1), preocupados com a verificação de alergias a medicamentos:

> *Se é sabido que um paciente é alérgico a algum medicamento específico, uma prescrição para esse medicamento deve resultar em uma mensagem de aviso a ser emitida ao usuário do sistema.*
>
> *Se um médico opta por ignorar um aviso de alergia, ele deve fornecer uma razão pela qual esse aviso foi ignorado.*

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

1. Defina um registro de paciente sem alergias conhecidas. Prescreva medicação para alergias que são conhecidas. Verifique se uma mensagem de alerta não é emitida pelo sistema.
2. Defina um registro de paciente com alergia. Prescreva a medicação à qual o paciente é alérgico e verifique se o aviso é emitido pelo sistema.
3. Defina um registro de paciente no qual se registram alergias a dois ou mais medicamentos. Prescreva ambos os medicamentos separadamente e verifique se o aviso correto para cada um é emitido.
4. Prescreva dois medicamentos a que o paciente é alérgico. Verifique se os dois avisos são emitidos corretamente.
5. Prescreva um medicamento que emita um aviso e ignore esse aviso. Verifique se o sistema exige que o usuário forneça informações que expliquem por que o aviso foi ignorado.

A partir disso, você pode ver que testar um requisito não significa simplesmente escrever um único teste. Normalmente, você precisa escrever vários testes para garantir a cobertura dos requisitos. Você também deve manter registros de rastreabilidade de seus testes baseados em requisitos, que ligam os testes aos requisitos específicos que estão sendo testados.

8.3.2 Testes de cenário

Teste de cenário é uma abordagem de teste de release em que você imagina cenários típicos de uso e os usa para desenvolver casos de teste para o sistema. Um cenário é uma estória que descreve uma maneira de usar o sistema. Cenários devem ser realistas, e usuários reais do sistema devem ser capazes de se relacionar com eles. Se você já usou cenários como parte do processo de engenharia de requisitos (descritos no Capítulo 4), então você é capaz de reusá-los ao testar cenários.

Em um breve artigo sobre testes de cenário, Kaner (2003) sugere que um teste de cenário deve ser uma estória narrativa crível e bastante complexa. Deve motivar os *stakeholders*, ou seja, eles devem se relacionar com o cenário e acreditar que é importante que o sistema passe no teste. Kaner também sugere que devem ser de fácil avaliação. Se houver problemas com o sistema, a equipe de teste de release deve reconhecê-los. Como exemplo de um possível cenário a partir do MHC-PMS, o Quadro 8.1 descreve uma maneira pela qual o sistema pode ser usado em uma visita domiciliar.

O cenário testa uma série de características do MHC-PMS:

1. Autenticação por *logon* no sistema.
2. Download e *upload* de determinados registros do paciente em um computador portátil.
3. Programação de visitas domiciliares.
4. Criptografia e descriptografia de registros de pacientes em um dispositivo móvel.
5. Recuperação e modificação de registros.
6. Links com o banco de dados dos medicamentos, que mantém informações sobre os efeitos colaterais.
7. O sistema de aviso de chamada.

Se você é um testador de release, você percorre esse cenário no papel de Kate, observando como o sistema se comporta em resposta a entradas diferentes. Atuando no papel de Kate, você pode cometer erros deliberados, como introduzir a frase-chave errada para decodificar registros. Esse procedimento verifica a resposta do sistema aos erros. Você deve anotar cuidadosamente quaisquer problemas que possam surgir, incluindo problemas de desempenho. Se um sistema é muito lento, a maneira como ele é usado será diferente. Por exemplo, se leva muito tempo para criptografar um registro, então os usuários que têm pouco tempo podem pular essa etapa. Se eles perderem seu notebook, uma pessoa não autorizada poderia visualizar os registros de pacientes.

Ao usar uma abordagem baseada em cenários, você normalmente testa vários requisitos dentro do mesmo cenário. Assim, além de verificar os requisitos individuais, também está verificando quais combinações de requisitos não causam problemas.

Quadro 8.1 Um cenário de uso para o MHC-PMS

Kate é uma enfermeira especialista em saúde mental. Uma de suas responsabilidades é visitar os pacientes em casa para verificar a eficácia do tratamento e se os pacientes estão sofrendo com os efeitos colaterais da medicação.

Em um dia de visitas, Kate faz o *login* no MHC-PMS e usa-o para imprimir sua agenda de visitas domiciliares para aquele dia, juntamente com o resumo das informações sobre os pacientes a serem visitados. Ela pede que os registros desses pacientes sejam transferidos para seu notebook. É solicitada a palavra-chave para criptografar os registros no notebook.

Um dos pacientes que Kate visita é Jim, que está sendo tratado com medicação para depressão. Jim acha que a medicação está ajudando, mas acredita que tem o efeito colateral de mantê-lo acordado durante a noite. Kate vai consultar o registro de Jim; sua frase-chave é solicitada para decifrar o registro. Ela verifica o medicamento prescrito e consulta seus efeitos colaterais. Insônia é um efeito colateral conhecido. Ela faz uma observação sobre o problema no registro de Jim e sugere que ele visite a clínica para ter sua medicação alterada. Ele concorda. Assim, Kate faz um *prompt* de entrar em contato com ele quando ela voltar à clínica, para que faça uma consulta com um médico. Ela termina a consulta e o sistema criptografa novamente o registro de Jim.

Depois, terminadas suas consultas, Kate retorna à clínica e transfere os registros dos pacientes visitados para o banco de dados. O sistema gera para Kate uma lista de pacientes que ela precisa contatar para obter informações de acompanhamento e fazer agendamentos de consultas.

8.3.3 Testes de desempenho

Uma vez que o sistema tenha sido totalmente integrado, é possível testá-lo para propriedades emergentes, como desempenho e confiabilidade. Os testes de desempenho precisam ser projetados para assegurar que o sistema possa processar a carga a que se destina. Isso normalmente envolve a execução de uma série de testes em que você aumenta a carga até que o desempenho do sistema se torne inaceitável.

Tal como acontece com outros tipos de testes, testes de desempenho estão interessados tanto em demonstrar que o sistema atende seus requisitos quanto em descobrir problemas e defeitos do sistema. Para testar se os requisitos de desempenho estão sendo alcançados, você pode ter de construir um perfil operacional. Um perfil operacional (veja o Capítulo 15) é um conjunto de testes que refletem a mistura real de trabalho que será manipulado pelo sistema. Portanto, se 90% das transações no sistema são do tipo A, 5% são do tipo B e o restante é dos tipos C, D e E, você tem de projetar o perfil operacional de modo que a grande maioria dos testes seja do tipo A. Caso contrário, você não vai ter um teste preciso do desempenho operacional do sistema.

Essa não é necessariamente a melhor abordagem para testes de defeitos. A experiência tem mostrado que uma forma eficaz de descobrir os defeitos é projetar testes para os limites do sistema. Em testes de desempenho, isso significa estressar o sistema, fazendo demandas que estejam fora dos limites de projeto do software. Isso é conhecido como 'teste de estresse'. Por exemplo, digamos que você está testando um sistema de processamento de transações que é projetado para processar até 300 transações por segundo. Você começa a testar esse sistema com menos de 300 transações por segundo; então, aumenta gradualmente a carga no sistema para além de 300 transações por segundo, até que esteja bem além da carga máxima de projeto do sistema e o sistema falhe. Esse tipo de teste tem duas funções:

1. Testar o comportamento de falha do sistema. As circunstâncias podem surgir por meio de uma combinação inesperada de eventos em que a carga sobre o sistema excede a carga máxima prevista. Nessas circunstâncias, é importante que a falha do sistema não cause corrupção de dados ou perda inesperada de serviços de usuário. Testes de estresse que verificam a sobrecarga do sistema fazem com que ele caia de maneira suave em vez de entrar em colapso sob sua carga.
2. Estressar o sistema e trazer à luz defeitos que normalmente não são descobertos. Embora se possa argumentar que esses defeitos, em uso normal, não são suscetíveis a causarem falhas no sistema, pode haver combinações inusitadas de circunstâncias normais que o teste de estresse replique.

Os testes de estresse são particularmente relevantes para sistemas distribuídos baseados em uma rede de processadores. Esses sistemas frequentemente apresentam degradação severa quando estão muito carregados. A rede fica inundada com dados de coordenação que os diferentes processos devem trocar. Os processos tornam-se mais lentos à medida que aguardam os dados requisitados a outros processos. Os testes de estresse ajudam-no a descobrir quando a degradação começa, e, assim, você pode adicionar controles ao sistema para rejeitar operações além desse ponto.

8.4 Testes de usuário

Teste de usuário ou de cliente é um estágio no processo de teste em que os usuários ou clientes fornecem entradas e conselhos sobre o teste de sistema. Isso pode envolver o teste formal de um sistema que foi aprovado por um fornecedor externo ou processo informal em que os usuários experimentam um produto de software novo para ver se gostam e verificar se faz o que eles precisam. O teste de usuário é essencial, mesmo em sistemas abrangentes ou quando testes de release tenham sido realizados. A razão para isso é que as influências do ambiente de trabalho do usuário têm um efeito importante sobre a confiabilidade, o desempenho, a usabilidade e a robustez de um sistema.

É praticamente impossível para um desenvolvedor de sistemas replicar o ambiente de trabalho do sistema, pois os testes no ambiente do desenvolvedor são inevitavelmente artificiais. Por exemplo, um sistema que se destina a ser usado em um hospital é usado em um ambiente clínico em que outras coisas estão acontecendo, como emergências de pacientes, conversas com parentes etc. Isso tudo afeta o uso de um sistema, mas os desenvolvedores não podem incluí-los em seu ambiente de teste.

Na prática, existem três tipos de testes de usuário:

1. Teste alfa, em que os usuários do software trabalham com a equipe de desenvolvimento para testar o software no local do desenvolvedor.
2. Teste beta, em que um release do software é disponibilizado aos usuários para que possam experimentar e levantar os problemas que eles descobriram com os desenvolvedores do sistema.
3. Teste de aceitação, em que os clientes testam um sistema para decidir se está ou não pronto para ser aceito pelos desenvolvedores de sistemas e implantado no ambiente do cliente.

Em testes alfa, usuários e desenvolvedores trabalham em conjunto para testar um sistema que está sendo desenvolvido. Isso significa que os usuários podem identificar os problemas e as questões que não são aparentes para a equipe de testes de desenvolvimento. Os desenvolvedores só podem trabalhar a partir dos requisitos, mas, muitas vezes, estes não refletem outros fatores que afetam o uso prático do software. Os usuários podem, portanto, fornecer informações sobre as práticas que contribuem com projeto de testes mais realistas.

Os testes alfa são frequentemente usados no desenvolvimento de produtos de software que são vendidos como sistemas-pacote. Os usuários desses produtos podem estar dispostos a se envolver no processo de testes alfa, pois isso lhes antecipa informações sobre novas características do sistema, as quais eles poderão explorar. Também reduz o risco de que mudanças inesperadas no software tenham efeitos perturbadores sobre os negócios. No entanto, testes alfa também podem ser usados quando o software customizado está sendo desenvolvido. Os métodos ágeis, como XP, defendem o envolvimento do usuário no processo de desenvolvimento e alegam que os usuários devem desempenhar um papel fundamental no projeto de testes para o sistema.

O teste beta ocorre quando um release antecipado, por vezes inacabado, de um sistema de software é disponibilizado aos clientes e usuários para avaliação.

Testadores beta podem ser um grupo de clientes selecionados, os primeiros a adotarem o sistema. Como alternativa, o software pode ser disponibilizado para uso, para qualquer pessoa que esteja interessada nele. O teste beta é usado principalmente para produtos de software que são usados nos mais diversos ambientes (por oposição aos sistemas customizados, que são geralmente usados em um ambiente definido). É impossível para os desenvolvedores de produtos conhecerem e replicarem todos os ambientes em que o software será usado. O teste beta é essencial para descobrir justamente problemas de interação entre o software e as características do ambiente em que ele é usado. Também é uma forma de marketing — os clientes aprendem sobre seu sistema e o que ele pode fazer por eles.

O teste de aceitação é uma parte inerente ao desenvolvimento de sistemas customizados, que ocorre após o teste de release. Engloba o teste formal de um sistema pelo cliente para decidir se ele deve ou não ser aceito. A aceitação designa que o pagamento pelo sistema deve ser feito.

Existem seis estágios no processo de teste de aceitação, como mostra a Figura 8.10. São eles:

1. *Definir critérios de aceitação.* Esse estágio deve, idealmente, ocorrer no início do processo antes de o contrato do sistema ser assinado. Os critérios de aceitação devem ser parte do contrato do sistema e serem acordados entre o cliente e o desenvolvedor. Na prática, porém, pode ser difícil definir critérios para o início do processo. Os requisitos detalhados podem não estar disponíveis e podem haver mudanças significativas dos requisitos durante o processo de desenvolvimento.
2. *Planejar testes de aceitação.* Trata-se de decidir sobre os recursos, tempo e orçamento para os testes de aceitação e estabelecer um cronograma de testes. O plano de teste de aceitação também deve discutir a cobertura dos requisitos exigidos e a ordem em que as características do sistema são testadas. Deve definir os riscos para o processo de testes, como interrupção de sistema e desempenho inadequado, e discutir como esses riscos podem ser mitigados.

Figura 8.10 O processo de teste de aceitação

3. *Derivar testes de aceitação.* Uma vez que os testes de aceitação tenham sido estabelecidos, eles precisam ser projetados para verificar se existe ou não um sistema aceitável. Testes de aceitação devem ter como objetivo testar tanto as características funcionais quanto as não funcionais (por exemplo, desempenho) do sistema. Eles devem, idealmente, fornecer cobertura completa dos requisitos de sistema. Na prática, é difícil estabelecer critérios completamente objetivos de aceitação. Muitas vezes, fica margem para discussão sobre se o teste mostra ou não se um critério foi definitivamente cumprido.
4. *Executar testes de aceitação.* Os testes de aceitação acordados são executados no sistema. Idealmente, isso deve ocorrer no ambiente real em que o sistema será usado, mas isso pode ser interrompido e impraticável. Portanto, um ambiente de testes de usuário pode ter de ser configurado para executar esses testes. É difícil automatizar esse processo, pois parte dos testes de aceitação pode envolver testes de interações entre os usuários finais e o sistema. Pode ser necessário algum treinamento para os usuários finais.
5. *Negociar resultados de teste.* É muito improvável que todos os testes de aceitação definidos passem e que não ocorra qualquer problema com o sistema. Se esse for o caso, então o teste de aceitação está completo, e o sistema pode ser entregue. O mais comum, contudo, é que alguns problemas sejam descobertos. Nesses casos, o desenvolvedor e o cliente precisam negociar para decidir se o sistema é bom o suficiente para ser colocado em uso. Eles também devem concordar sobre a resposta do desenvolvedor para os problemas identificados.
6. *Rejeitar/aceitar sistema.* Esse estágio envolve uma reunião entre os desenvolvedores e o cliente para decidir se o sistema deve ser aceito. Se o sistema não for bom o suficiente para o uso, então será necessário um maior desenvolvimento para corrigir os problemas identificados. Depois de concluída, a fase de testes de aceitação é repetida.

Nos métodos ágeis, como XP, o teste de aceitação tem um significado bastante diferente. Em princípio, ele compartilha a ideia de que os usuários devem decidir quando o sistema é aceitável. No entanto, no XP, o usuário é parte da equipe de desenvolvimento (isto é, ele ou ela é um testador alfa) e fornece os requisitos de sistema em termos de estórias de usuários. Ele ou ela também é responsável pela definição dos testes, que decide se o software desenvolvido apoia ou não a estória de usuário. Os testes são automatizados, e o desenvolvimento não continua até os testes de aceitação de estória passarem. Portanto, não há uma atividade de teste de aceitação em separado.

Como já discutido no Capítulo 3, um problema com o envolvimento do usuário é garantir que o usuário que está integrado na equipe de desenvolvimento seja um usuário 'típico', com conhecimento geral de como o sistema será usado. Considerando que pode ser difícil encontrar tal usuário, os testes de aceitação não podem refletir verdadeiramente a prática. Além disso, o requisito por testes automatizados limita severamente a flexibilidade de testar sistemas interativos. Para estes sistemas, testes de aceitação podem exigir que grupos de usuários finais usem o sistema como se fosse parte de seu trabalho diário.

Você pode pensar que o teste de aceitação é uma questão clara de corte contratual. Se um sistema não passar em seus testes de aceitação, então não deve ser aceito, e o pagamento não deve ser feito. No entanto, a realidade é mais complexa. Os clientes querem usar o software assim que possível por causa dos benefícios de sua implantação imediata. Eles podem ter comprado novos equipamentos, treinado seu pessoal e alterado seus processos. Eles podem estar dispostos a aceitar o software, independentemente dos problemas, porque os custos do não uso do software são maiores do que os custos de se trabalhar nos problemas. Portanto, o resultado das negociações pode ser de aceitação condicional do sistema. O cliente pode aceitar o sistema para que a implantação possa começar. E o fornecedor do sistema se compromete a sanar os problemas urgentes e entregar uma nova versão para o cliente o mais rápido possível.

PONTOS IMPORTANTES

- Os testes só podem anunciar a presença de defeitos em um programa. Não podem demonstrar que não existem defeitos remanescentes.
- Testes de desenvolvimento são de responsabilidade da equipe de desenvolvimento de software. Outra equipe deve ser responsável por testar o sistema antes que ele seja liberado para os clientes. No processo de testes de usuário, clientes ou usuários do sistema fornecem dados de teste e verificam se os testes são bem-sucedidos.
- Testes de desenvolvimento incluem testes unitários, nos quais você testa objetos e métodos específicos; testes de componentes, em que você testa diversos grupos de objetos; e testes de sistema, nos quais você testa sistemas parciais ou completos.
- Ao testar o software, você deve tentar 'quebrar' o software usando sua experiência e diretrizes para escolher os tipos de casos de teste que têm sido eficazes na descoberta de defeitos em outros sistemas.

- Sempre que possível, você deve escrever testes automatizados. Os testes são incorporados em um programa que pode ser executado cada vez que uma alteração é feita para um sistema.
- O desenvolvimento *test-first* é uma abordagem de desenvolvimento na qual os testes são escritos antes do código que será testado. Pequenas alterações no código são feitas, e o código é refatorado até que todos os testes sejam executados com êxito.
- Testes de cenário são úteis porque replicam o uso prático do sistema. Trata-se de inventar um cenário típico de uso e usar isso para derivar casos de teste.
- Teste de aceitação é um processo de teste de usuário no qual o objetivo é decidir se o software é bom o suficiente para ser implantado e usado em seu ambiente operacional.

LEITURA COMPLEMENTAR

'How to design practical test cases'. Um artigo sobre como projetar casos de teste, escrito por um autor de uma empresa japonesa que tem uma reputação muito boa em entregar softwares com pouquíssimos defeitos. (YAMAURA, T. *IEEE Software*, v. 15, n. 6, nov. 1998. Disponível em: <http://dx.doi.org/10.1109/52.730835>.)

How to Break Software: A Practical Guide to Testing. Esse é um livro sobre testes de software, mais prático do que teórico, no qual o autor apresenta um conjunto de diretrizes baseadas em experiências em projetar testes suscetíveis de serem eficazes na descoberta de defeitos de sistema. (WHITTAKER, J. A. *How to Break Software: A Practical Guide to Testing*. Addison-Wesley, 2002.)

'Software Testing and Verification'. Essa edição especial do *IBM Systems Journal* abrange uma série de artigos sobre testes, incluindo uma boa visão geral, artigos sobre métricas e automação de teste. (*IBM Systems Journal*, v. 41, n. 1, jan. 2002.)

'Test-Driven Development'. Essa edição especial sobre desenvolvimento dirigido a testes inclui uma boa visão geral de TDD, bem como artigos de experiência em como TDD tem sido usado para diferentes tipos de software. (*IEEE Software*, v. 24, n. 3, mai./jun. 2007.)

EXERCÍCIOS

8.1 Explique por que um programa não precisa, necessariamente, ser completamente livre de defeitos antes de ser entregue a seus clientes.

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

8.3 Algumas pessoas argumentam que os desenvolvedores não devem ser envolvidos nos testes de seu próprio código, mas que todos os testes devem ser de responsabilidade de uma equipe independente. Dê argumentos a favor e contra a realização de testes pelos próprios desenvolvedores.

8.4 Você foi convidado para testar um método chamado 'catWhiteSpace' em um objeto 'Parágrafo'. Dentro do parágrafo, as sequências de caracteres em branco são substituídas por um único caractere em branco. Identifique partições para testar esse exemplo e derive um conjunto de testes para o método 'catWhiteSpace'.

8.5 O que é o teste de regressão? Explique como o uso de testes automatizados e um *framework* de teste, tal qual o JUnit, simplifica os testes de regressão.

8.6 O MHC-PMS é construído por meio da adaptação de um sistema de informações de prateleira. Quais são as diferenças entre esses testes de software e os testes de um sistema desenvolvido por meio de uma linguagem orientada a objetos como Java?

8.7 Escreva um cenário que poderia ser usado para ajudar no projeto de testes para o sistema da estação meteorológica no deserto.

8.8 O que você entende pelo termo 'testes de estresse'? Sugira como você pode estressar o teste MHC-PMS.

8.9 Quais são as vantagens de os usuários se envolverem nos testes de release em um estágio inicial do processo de teste? Existem desvantagens na participação do usuário?

8.10 Uma abordagem comum para o teste de sistema é testar o sistema até que o orçamento de teste esteja esgotado e, em seguida, entregar o sistema para os clientes. Discuta a ética dessa abordagem para os sistemas que são entregues aos clientes externos.

REFERÊNCIAS

ANDREA, J. Envisioning the Next Generation of Functional Testing Tools. *IEEE Software*, v. 24, n. 3, 2007, p. 58-65.

BECK, K. *Test Driven Development*: By Example. Boston: Addison-Wesley, 2002.

BEZIER, B. *Software Testing Techniques*. 2. ed. Nova York: Van Nostrand Rheinhold, 1990.

BOEHM, B. W. Software engineering; R & D Trends and defense needs. In: *Research Directions in Software Technology*. WEGNER, P. (Org.). Cambridge, Mass.: MIT Press, 1979, p. 1-9.

CUSAMANO, M.; SELBY, R. W. *Microsoft Secrets*. Nova York: Simon and Shuster, 1998.

DIJKSTRA, E. W.; DAHL, O. J.; HOARE, C. A. R. *Structured Programming*. Londres: Academic Press, 1972.

FAGAN, M. E. Advances in Software Inspections. *IEEE Trans. on Software Eng*, v. SE-12, n. 7, 1986, p. 744-751.

JEFFRIES, R.; MELNIK, G. TDD: The Art of Fearless Programming. *IEEE Software*, v. 24, 2007, p. 24-30.

KANER, C. The power of 'What If...' and nine ways to fuel your imagination: Cem Kaner on scenario testing. *Software Testing and Quality Engineering*, v. 5, n. 5, 2003, p. 16-22.

LUTZ, R. R. Analyzing Software Requirements Errors in Safety-Critical Embedded Systems. *RE'93*, San Diego, Calif.: IEEE, 1993.

MARTIN, R. C. Professionalism and Test-Driven Development. *IEEE Software*, v. 24, n. 3, 2007, p. 32-36.

MASSOL, V.; HUSTED, T. *JUnit in Action*. Greenwich, Conn.: Manning Publications Co., 2003.

PROWELL, S. J.; TRAMMELL, C. J.; Linger, R. C.; POORE, J. H. *Cleanroom Software Engineering*: Technology and Process. Reading, Mass.: Addison-Wesley, 1999.

WHITTAKER, J. W. *How to Break Software*: A Practical Guide to Testing. Boston: Addison-Wesley, 2002.