18 Testando Aplicativos Convencionais

Conceitos-Chave

O teste apresenta uma anomalia interessante para os engenheiros de software, que são, por natureza, pessoas construtivas. O teste requer que o desenvolvedor descarte noções preconcebidas da "corretividade" do software recém-desenvolvido e passe a trabalhar arduamente projetando casos de teste para "quebrar" o software. Beizer [Bei90] descreve essa situação eficazmente quando declara:

> Há um mito de que, se fôssemos realmente bons em programação, não precisaríamos caçar erros. Se pudéssemos realmente nos concentrar, se todos usassem programação estruturada, projeto com detalhamento progressivo... Então não haveria erros. E assim continua o mito. Existem erros, diz o mito, porque somos ruins no que fazemos; e se somos ruins no que fazemos, devemos nos sentir culpados por isso. Portanto, o teste e o planejamento de casos de teste é um reconhecimento de falha, que sugere uma boa dose de culpa. E o tédio do teste é exatamente a punição pelos nossos erros. Punição por quê? Por sermos humanos? Culpados de quê? De não conseguir atingir a perfeição desumana? De não distinguir entre o que um outro programador pensa e o que ele diz? Por não conseguir ser telepático? Por não resolver problemas de comunicação humana que já existem... Há quarenta séculos?

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

Panorama

O que é? Uma vez gerado o código-fonte, o software deve ser testado para descobrir (e corrigir) tantos erros quanto possível antes de fornecê-lo ao seu cliente. Sua meta é projetar um conjunto de casos de teste que tenha a mais alta probabilidade de encontrar erros — mas como? É aqui que entram em cena as técnicas de teste de software. Essas técnicas fornecem diretrizes sistemáticas para projetar testes que (1) exercitam a lógica interna e as interfaces de todos os componentes do software e (2) exercitam os domínios de entrada e saída do programa para descobrir erros no funcionamento, comportamento e desempenho do programa.

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

Por que é importante? Revisões e outras ações SQA podem descobrir e realmente descobrem erros, mas não são suficientes. Toda vez que o programa é executado, o cliente testa-o! Portanto, você tem de executar o programa antes que ele chegue ao cliente com o objetivo específico de encontrar e remover todos os erros. Para encontrar o maior número possível de erros, devem ser executados testes sistematicamente, e os casos de teste devem ser projetados usando técnicas disciplinadas.

Quais são as etapas envolvidas? Para aplicações convencionais, o software é testado a partir de duas perspectivas diferentes: (1) a lógica interna do programa é exercitada usando técnicas de projeto de caso de teste "caixa branca" e (2) os requisitos de software são exercitados usando técnicas de projeto de casos de teste "caixa preta". Casos de teste de uso ajudam no projeto de testes para descobrir erros no nível de validação de software. Em todos os casos, a intenção é encontrar o número máximo de erros com o mínimo de esforço e tempo.

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

Como garantir que o trabalho foi feito corretamente? Quando você começar o teste, mude o seu ponto de vista. Tente "quebrar" o software! Projete casos de teste de forma disciplinada e reveja os casos de teste que você criou, quanto à perfeição. Além disso, você pode avaliar a abrangência do teste e monitorar as atividades de detecção de erros.

teste de estrutura de controle 437
teste do caminho básico. 485
teste de matriz ortogonal 442

Neste capítulo, discutimos técnicas para projetar casos de teste de software para aplicações convencionais. O projeto de casos de teste focaliza um conjunto de técnicas para a criação de casos de teste, que satisfazem os objetivos globais e as estratégias de teste discutidas no Capítulo 17.

18.1 Fundamentos do Teste de Software

"Todo programa faz alguma coisa certa, só que pode não ser aquilo que queremos que ele faça."

Autor desconhecido

O objetivo do teste é encontrar erros, e um bom teste é aquele que tem alta probabilidade de encontrar um erro. Portanto, um engenheiro de software deve projetar e implementar um sistema ou produto baseado em computador tendo em mente a "testabilidade". Ao mesmo tempo, os próprios testes devem ter uma série de características que permitam atingir o objetivo de encontrar o maior número de erros com o mínimo de esforço.

Testabilidade. James Bach[1] dá a seguinte definição para testabilidade: "*Testabilidade de software* é simplesmente a facilidade com que um programa de computador pode ser testado". As seguintes características levam a um software testável:

Operabilidade. "Quanto melhor funcionar, mais eficientemente pode ser testado." Se um sistema for projetado e implementado tendo em mente a qualidade, haverá poucos defeitos bloqueando a execução dos testes, permitindo que o teste ocorra sem sobressaltos.

? Quais são as características da testabilidade?

Observabilidade. "O que você vê é o que você testa." Entradas fornecidas como parte do teste produzem saídas distintas. Estados e variáveis do sistema são visíveis ou podem ser consultados durante a execução. Saída incorreta é facilmente identificada. Erros internos são automaticamente detectados e relatados. O código-fonte é acessível.

Controlabilidade. "Quanto melhor pudermos controlar o software, mais o teste pode ser automatizado e otimizado." Todas as possíveis saídas podem ser geradas por meio de alguma combinação de entrada, e os formatos de entrada e saída são consistentes e estruturados. Todo o código é executável através de alguma combinação de entrada. Estados e variáveis de software e hardware podem ser controlados diretamente pelo engenheiro de teste. Os testes podem ser convenientemente especificados, automatizados e reproduzidos.

Decomponibilidade. "Controlando o escopo do teste, podemos isolar problemas mais rapidamente e executar um reteste mais racionalmente." O sistema de software é construído a partir de módulos independentes que podem ser testados de forma independente.

"Erros são mais comuns, mais disseminados e mais problemáticos no software do que em outras tecnologias."

David Parnas

Simplicidade. "Quanto menos tiver que testar, mais rapidamente podemos testá-lo." O programa deverá ter *simplicidade funcional* (por exemplo, o conjunto de características é o mínimo necessário para satisfazer os requisitos); *simplicidade estrutural* (por exemplo, a arquitetura é modularizada para limitar a propagação de falhas), e *simplicidade de código* (por exemplo, é adotado um padrão de codificação para facilitar a inspeção e a manutenção).

Estabilidade. "Quanto menos alterações, menos interrupções no teste." As alterações no software são pouco frequentes, controladas quando elas ocorrem e não invalidam os testes existentes. O software recupera-se bem das falhas.

Compreensibilidade. "Quanto mais informações tivermos, mais inteligente será o teste." O projeto arquitetural e as dependências entre componentes internos, externos e compartilhados são bem compreendidas. A documentação técnica é instantaneamente acessível, bem organizada, específica, detalhada e precisa. Alterações no projeto são comunicadas aos testadores.

1 Os parágrafos seguintes são usados com permissão de James Bach (copyright 1994) e foram adaptados a partir de material que originalmente apareceu em uma postagem no newsgroup comp.software-eng.

Os atributos sugeridos por Bach podem ser utilizados por um engenheiro de software para desenvolver uma configuração de software (isto é, programas, dados e documentos) que é sensível ao teste.

Características do teste. E quanto aos próprios testes? Kaner, Falk e Nguyen [Kan93] sugerem os seguintes atributos para um "bom" teste:

O que é um "bom teste"?

Um bom teste tem alta probabilidade de encontrar um erro. Para atingir esse objetivo, o testador deve entender o software e tentar desenvolver uma imagem mental de como o software pode falhar. O ideal é que as classes de falhas sejam investigadas. Por exemplo, uma classe de falha em potencial, em uma interface gráfica com o usuário é uma falha em reconhecer a posição correta do mouse. Seria preparado um conjunto de testes para exercitar o mouse na tentativa de demonstrar erro no reconhecimento da posição do mouse.

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

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

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

CasaSegura

Projetando testes únicos

Cena: Sala de Vinod.

Participantes: Vinod e Ed — membros da equipe de engenharia de software do CasaSegura.

Conversa:

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

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

Vinod: Vejamos... Você observou que a senha correta será 8080, certo?

Ed: Hã, hã.

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

Ed: Certo, e eu também testo senhas que são semelhantes à senha correta, veja... 8081 e 8180.

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

Ed: Bem, são valores diferentes.

Vinod: É verdade, mas se 1234 não descobrir um erro... Em outras palavras, se a operação validacaoDeSenha percebe que é uma senha inválida, não é provável que 6789 nos mostre algo novo.

Ed: Entendo o que você quer dizer.

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

Ed: Sem problemas... Vou pensar um pouco mais nisso.

18.2 Visões Interna e Externa do Teste

Qualquer produto de engenharia (e muitas outras coisas) pode ser testado por uma de duas maneiras: (1) Conhecendo a função especificada para o qual um produto foi projetado para realizar, podem ser feitos testes que demonstram que cada uma das funções é totalmente operacional, embora ao mesmo tempo procurem erros em cada função. (2) Conhecendo o funciona-

"Há apenas uma regra no projeto de casos de teste: abranger todas as características; mas não faça muitos casos de teste."

Tsuneo Yamaura

mento interno de um produto, podem ser realizados testes para garantir que "tudo se encaixa", isto é, que as operações internas foram realizadas de acordo com as especificações e que todos os componentes internos foram adequadamente exercitados. A primeira abordagem de teste usa uma visão externa e é chamada de teste caixa-preta. A segunda abordagem requer uma visão interna e é chamada de teste caixa-branca.[2]

O *teste caixa-preta* faz referência a testes realizados na interface do software. Um teste caixa-preta examina alguns aspectos fundamentais de um sistema, com pouca preocupação em relação à estrutura lógica interna do software. O *teste caixa-branca* fundamenta-se em um exame rigoroso do detalhe procedimental. Os caminhos lógicos do software e as colaborações entre componentes são testados exercitando conjuntos específicos de condições e/ou ciclos.

PONTO-CHAVE

Testes caixa-branca só podem ser projetados depois que o projeto no nível de componente (ou código-fonte) existir. Os detalhes lógicos do programa devem estar disponíveis.

À primeira vista poderia parecer que um teste caixa-branca realizado de forma rigorosa resultaria em "programas 100% corretos". Tudo o que seria preciso fazer seria definir todos os caminhos lógicos, desenvolver casos de teste para exercitá-los e avaliar os resultados, ou seja, gerar casos de teste para exercitar a lógica do programa de forma exaustiva. Infelizmente, o teste exaustivo apresenta certos problemas logísticos. Mesmo para pequenos programas, o número de caminhos lógicos possíveis pode ser muito grande. No entanto, o teste caixa-branca não deve ser descartado como impraticável. Um número limitado de caminhos lógicos importantes pode ser selecionado e exercitado. Estruturas de dados importantes podem ser investigadas quando à validade.

INFORMAÇÕES

Teste Exaustivo

Considere um programa de 100 linhas em linguagem C. Após algumas declarações básicas de dados, o programa contém dois laços aninhados que executam de 1 a 20 vezes cada um, dependendo das condições especificadas na entrada. Dentro do ciclo, são necessárias 4 construções if-then-else. Há aproximadamente 10^{14} caminhos possíveis que podem ser executados nesse programa!

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

Portanto, é razoável afirmar que o teste exaustivo é impossível para grandes sistemas de software.

18.3 Teste Caixa-Branca

"Os defeitos ficam à espreita nas esquinas e se reúnem nas fronteiras."

Boris Beizer

O *teste caixa-branca, também* chamado de *teste da caixa-de-vidro,* é uma filosofia de projeto de casos de teste que usa a estrutura de controle descrita como parte do projeto no nível de componentes para derivar casos de teste. Usando métodos de teste caixa-branca, o engenheiro de software pode criar casos de teste que (1) garantam que todos os caminhos independentes de um módulo foram exercitados pelo menos uma vez, (2) exercitam todas as decisões lógicas nos seus estados verdadeiro e falso, (3) executam todos os ciclos em seus limites e dentro de suas fronteiras operacionais, e (4) exercitam estruturas de dados internas para assegurar a sua validade.

18.4 Teste do Caminho Básico

O *teste de caminho básico* é uma técnica de teste caixa-branca proposta por Tom McCabe [McC76]. O teste de caminho básico permite ao projetista de casos de teste derivar uma medida da complexidade lógica de um projeto procedimental e usar essa medida como guia para

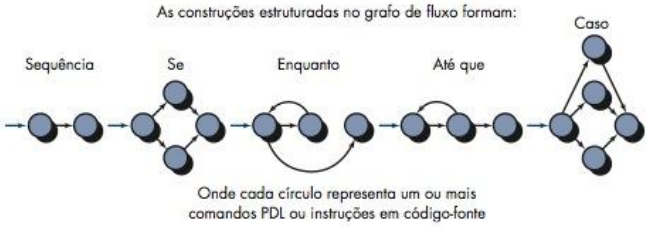
2 Os termos *teste funcional* e *teste estrutural* são às vezes usados em lugar de teste caixa-preta e teste caixa-branca, respectivamente.

definir um conjunto base de caminhos de execução. Casos de teste criados para exercitar o conjunto básico executam com certeza todas as instruções de um programa pelo menos uma vez durante o teste.

18.4.1 Notação de grafo de fluxo

Antes de considerarmos o teste de caminho básico, deve ser introduzida uma notação simples para a representação do fluxo de controle, chamada de *grafo de fluxo* (ou *grafo de programa*).[3] O grafo de fluxo representa o fluxo de controle lógico usando a notação ilustrada na Figura 18.1. Cada construção estruturada (Capítulo 10) tem um símbolo correspondente no grafo de fluxo.

FIGURA 18.1
Notação de grafo de fluxo

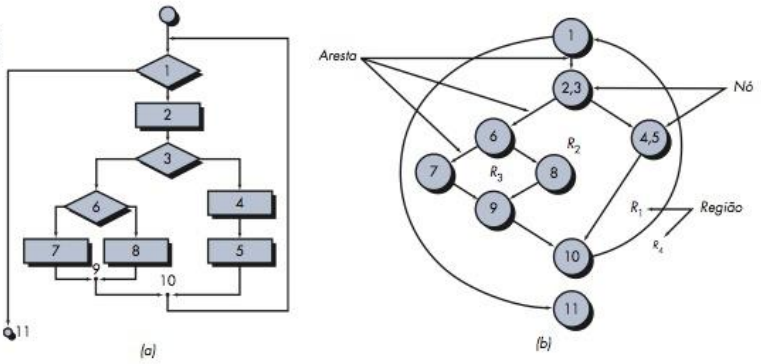


AVISO

Um grafo de fluxo somente deve ser desenhado quando a estrutura lógica de um componente for complexa. O grafo de fluxo permite seguir os caminhos de programa mais facilmente.

Para ilustrar o uso de um grafo de fluxo, considere a representação do projeto procedimental na Figura 18.2a. É usado um fluxograma para mostrar a estrutura de controle do programa. A Figura 18.2b mapeia o fluxograma em um grafo de fluxo correspondente (considerando que os losangos de decisão do fluxograma não contêm nenhuma condição composta). Na Figura 18.2b, cada círculo, chamado de *nó do grafo de fluxo,* representa um ou mais comandos procedurais. Uma sequência de retângulos de processamento e um losango de decisão podem ser mapeados em um único nó. As setas no grafo de fluxo, chamadas de *arestas* ou *ligações,* representam fluxo de controle e são análogas às setas do fluxograma. Uma aresta deve terminar em um nó, mesmo que esse nó não represente qualquer comando procedural (por exemplo, veja o símbolo do diagrama de fluxo para a construção se-então-senão - if-then-else). As áreas limitadas por arestas e nós são chamadas de *regiões.* Ao contarmos as regiões, incluímos a área fora do grafo como uma região.[4]

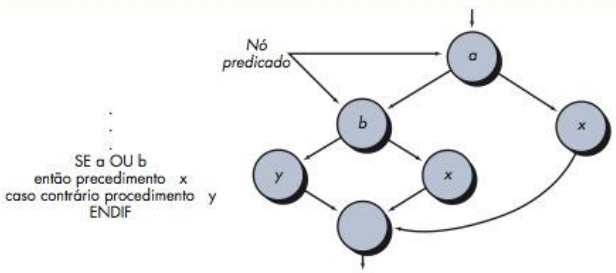
FIGURA 18.2
(a) Fluxograma e (b) grafo de fluxo



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

4 Uma discussão mais detalhada sobre grafos e seus usos é apresentada na Seção 18.6.1.

FIGURA 18.3
Lógica composta



Quando condições compostas são encontradas em um projeto procedimental, a geração de um grafo de fluxo torna-se ligeiramente mais complicada. Uma condição composta ocorre quando um ou mais operadores booleanos (OR, AND, NAND, NOR lógicos) estão presentes em um comando condicional. De acordo com a Figura 18.3, o trecho de PDL (*program design language*) é traduzido no grafo de fluxo mostrado. Note que é criado um nó separado para cada uma das condições *a* e *b* no comando SE *a* OU *b*. Cada nó contendo uma condição é chamado de *nó predicado* (*predicate node*) e é caracterizado por duas ou mais arestas saindo dele.

18.4.2 Caminhos de programa independentes

Um *caminho independente* é qualquer caminho através do programa que introduz pelo menos um novo conjunto de comandos de processamento ou uma nova condição. Quando definido em termos de um grafo de fluxo, um caminho independente deve incluir pelo menos uma aresta que não tenha sido atravessada antes de o caminho ser definido. Por exemplo, um conjunto de caminhos independentes para o grafo de fluxo ilustrado na Figura 18.2b é

Caminho 1: 1-11

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

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

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

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

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

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

A complexidade ciclomática é uma métrica útil para previsão dos módulos que têm a tendência de apresentar erros. Ela pode ser usada tanto para o planejamento de teste quanto para projeto de casos de teste.

Os caminhos de 1 a 4 constituem um *conjunto base* para o grafo de fluxo da Figura 18.2b. Isto é, se testes podem ser projetados para forçar a execução desses caminhos (conjunto base), cada comando do programa terá sido executado com certeza pelo menos uma vez e cada condição terá sido executada em seus lados verdadeiro e falso. Deve-se notar que o conjunto base não é único. De fato, vários conjuntos base diferentes podem ser derivados para um dado projeto procedimental.

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

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

Como se calcula a complexidade ciclomática?

1. O número de regiões do grafo de fluxo corresponde à complexidade ciclomática.
2. A complexidade ciclomática $V(G)$ para um grafo de fluxo G é definida como
   $V(G) = E - N + 2$
   em que E é o número de arestas do grafo de fluxo e N é o número de nós do grafo de fluxo.
3. A complexidade ciclomática $V(G)$ para um grafo de fluxo G também é definida como
   $V(G) = P + 1$
   em que P é o número de nós predicados contidos no grafo de fluxo G.

PONTO-CHAVE

A complexidade ciclomática fornece o limite superior no número de casos de teste que precisam ser executados para garantir que cada comando do programa tenha sido executado pelo menos uma vez.

Examinando mais uma vez o diagrama de fluxo da Figura 18.2b, a complexidade ciclomática pode ser calculada usando cada um dos algoritmos citados anteriormente:

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

Portanto, a complexidade ciclomática para o grafo de fluxo da Figura 18.2b é 4.

E o mais importante, o valor para $V(G)$ fornece um limite superior para o número de caminhos independentes que podem formar o conjunto base e, como consequência, um limite superior sobre o número de testes que devem ser projetados e executados para garantir a abrangência de todos os comandos do programa.

CasaSegura

Usando a complexidade ciclomática

Cena: Sala da Shakira.

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

Conversa:

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

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

Shakira: O prazo final para o primeiro incremento está se esgotando e eu gostaria de... Oh, estou preocupada.

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

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

Vinod: V de G.

Shakira: Hã?

Vinod: Complexidade ciclomática — V de G. Basta calcular V(G) para cada uma das operações dentro de cada um dos componentes e ver quais têm os maiores valores para V(G). São essas que têm maior tendência a apresentar erro.

Shakira: E como eu calculo V de G?

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

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

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

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

"O foguete Ariane 5 explodiu no lançamento simplesmente por um defeito de software (uma falha) envolvendo a conversão de um valor em ponto flutuante de 64 bits em um inteiro de 16 bits. O foguete e seus quatro satélites não estavam segurados e valiam $500 milhões. Testes de caminho [que exercitam o caminho de conversão] teriam descoberto o defeito, mas foram vetados por razões de orçamento."

Notícia de um jornal

18.4.3 Derivação de casos de teste

O método de teste de caminho base pode ser aplicado a um projeto procedimental ou ao código-fonte. Nesta seção, apresento o teste de caminho básico como uma série de passos. O procedimento *média* (*average*), mostrado em PDL na Figura 18.4, será usado como exemplo para ilustrar cada passo no método de projeto de casos de teste. Note que *média*, embora sendo um algoritmo extremamente simples, contém condições compostas e ciclos. Os seguintes passos podem ser aplicados para derivar o conjunto base:

1. **Usando o projeto ou o código como base, desenhe o grafo de fluxo correspondente.** Um grafo de fluxo é criado usando os símbolos e regras de construção apresentados na Seção 18.4.1. De acordo com o PDL para *média* na Figura 18.4, é criado um grafo de fluxo enumerando-se os comandos PDL que serão mapeados por nós correspondentes do grafo de fluxo. O grafo de fluxo correspondente é mostrado na Figura 18.5.
2. **Determine a complexidade ciclomática do diagrama de fluxo resultante. A complexidade ciclomática** $V(G)$ é determinada aplicando-se os algoritmos descritos na Seção 18.4.2. Deve-se notar que $V(G)$ pode ser determinado sem desenvolver um grafo de fluxo contando todos os comandos condicionais no PDL (para o procedimento *média*, o total de condições compostas é igual a dois) e somando 1. De acordo com a Figura 18.5,

   $V(G)$ = 6 regiões

   $V(G)$ = 17 arestas − 13 nós + 2 = 6

   $V(G)$ = 5 nós predicados + 1 = 6
3. **Determine um conjunto base de caminhos linearmente independentes.** O valor de $V(G)$ fornece o limite superior no número de caminhos linearmente independentes através da estrutura de controle do programa. No caso do procedimento *média*, esperamos especificar seis caminhos:

   Caminho 1: 1-2-10-11-13

   Caminho 2: 1-2-10-12-13

   Caminho 3: 1-2-3-10-11-13

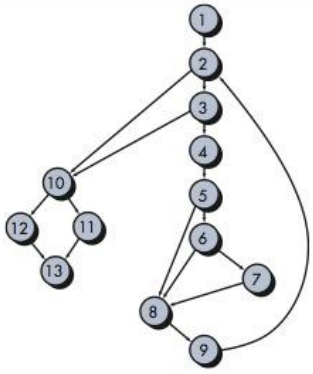
FIGURA 18.4
PDL com nós identificados

```
PROCEDURE average;
    *  Este procedimento calcula a média de 100 ou menos números
       situados entre valores limites; também calcula a soma e o
       total de números válidos.

INTERFACE RETURNS average, total.input, total.valid;
INTERFACE ACCEPTS value, minimum, maximum;

TYPE value[1:100] IS SCALAR ARRAY;
TYPE average, total.input, total.valid;
   minimum, maximum, sum IS SCALAR;
TYPE i IS INTEGER;
1  i = 1;
   total.input = total.valid = 0;
   sum = 0;
2  DO WHILE value[i] <> -999 AND total.input < 100   3
   4  increment total.input by 1;
      IF value[i] > = minimum AND value[i] < = maximum   6
5        THEN increment total.valid by 1;
7               sum = s sum + value[i]
         ELSE skip
      ENDIF
8     increment i by 1;
9  ENDDO
   IF total.valid > 0   10
   11 THEN average = sum / total.valid;
12    ELSE average = -999;
   13 ENDIF
END average
```

FIGURA 18.5
Grafo de fluxo para o procedimento *médio*



Caminho 4: 1-2-3-4-5-8-9-2-...

Caminho 5: 1-2-3-4-5-6-8-9-2-...

Caminho 6: 1-2-3-4-5-6-7-8-9-2-...

A reticência (...) após os caminhos 4, 5 e 6 indica que qualquer caminho através do resto da estrutura de controle é aceitável. Muitas vezes compensa identificar nós predicados como um auxílio na dedução de casos de teste. Nesse caso, os nós 2, 3, 5, 6 e 10 são nós predicados.

4. **Prepare casos de teste que vão forçar a execução de cada caminho do conjunto base.** Os dados devem ser escolhidos de forma que as condições nos nós predicados sejam definidas de forma apropriada à medida que cada caminho é testado. Cada caso de teste é executado e comparado com os resultados esperados. Depois que todos os casos de teste tiverem sido completados, o testador pode ter a certeza de que todos os comandos do programa foram executados pelo menos uma vez.

É importante notar que alguns caminhos independentes (por exemplo, caminho 1 em nosso exemplo) não podem ser testados de forma individual. Isso significa que a combinação de dados necessária para percorrer o caminho não pode ser conseguida no fluxo normal do programa. Nesses casos, esses caminhos são testados como parte de outro teste de caminho.

18.4.4 Matrizes de grafos

O procedimento para derivar o grafo de fluxo e até determinar um conjunto de caminhos base é passível de mecanização. Uma estrutura de dados, chamada de *matriz de grafos,* pode ser muito útil para o desenvolvimento de uma ferramenta de software que ajuda no teste do caminho base.

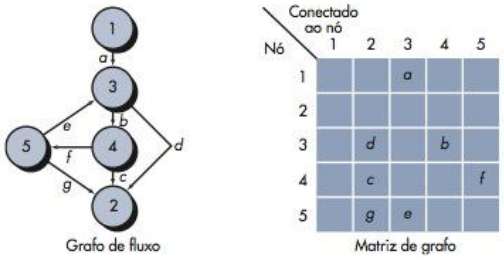
Uma matriz de grafo é uma matriz quadrada cujo tamanho (número de linhas e colunas) é igual ao número de nós no grafo de fluxo. Cada linha e coluna corresponde a um nó identificado, e as entradas da matriz correspondem a conexões (arestas) entre nós. Um exemplo simples de um grafo de fluxo e sua correspondente matriz de grafo [Bei90] é mostrado na Figura 18.6.

O que é uma matriz de grafo e como posso ampliá-la para uso em teste?

Observando a figura, cada nó do grafo de fluxo é identificado por números, enquanto cada aresta é identificada por letras. Uma letra na matriz corresponde à conexão entre dois nós. Por exemplo, o nó 3 está conectado ao nó 4 pela aresta *b*.

Até aqui, a matriz de grafo é nada mais do que uma representação tabular de um grafo de fluxo. No entanto, acrescentando um *peso de ligação* a cada entrada da matriz, a matriz de grafo pode se tornar uma poderosa ferramenta para avaliar a estrutura de controle de programa

FIGURA 18.6
Matriz de grafo



durante o teste. O peso da ligação fornece informações adicionais sobre o fluxo de controle. Em sua forma mais simples, o peso da ligação é 1 (existe uma conexão) ou 0 (não existe uma conexão). Mas pesos de ligação podem ser atribuídos de acordo com outras propriedades mais interessantes:

- A probabilidade de que uma ligação (aresta) será executada.
- O tempo de processamento gasto para percorrer uma ligação
- A quantidade de memória necessária para percorrer uma ligação
- Os recursos necessários para percorrer uma ligação.

Beizer [Bei90] apresenta um tratamento completo de outros algoritmos matemáticos que podem ser aplicados às matrizes de grafos. Usando essas técnicas, a análise necessária para projeto de casos de teste pode ser parcial ou totalmente automatizada.

18.5 TESTE DE ESTRUTURA DE CONTROLE

"Dar mais atenção à execução dos testes do que ao seu projeto é um erro clássico."

Brian Marick

A técnica de teste de caminho base descrita na Seção 18.4 é uma dentre várias técnicas para teste de estrutura de controle. Embora o teste de caminho base seja simples e altamente eficaz, ele sozinho não é suficiente. Nesta seção discutimos outras variações do teste de estrutura de controle. Elas ampliam a abrangência do teste e melhoram a qualidade do teste caixa-branca.

18.5.1 Teste de condição

Teste de condição [Tai89] é um método de projeto de caso de teste que exercita as condições lógicas contidas em um módulo de programa. Uma condição simples é uma variável booleana ou uma expressão relacional, possivelmente precedida por um operador NOT (¬). Uma expressão relacional toma a forma

E_1 <operador relacional > E_2

PONTO-CHAVE

Os erros são muito mais comuns nas proximidades de condições lógicas do que próximos dos comandos de processamento sequencial.

onde E_1 e E_2 são expressões aritméticas e <operador relacional> é um dos seguintes operadores: <, ≤, =, ≠ (não igual), >, ou ≥. Uma *condição composta* é formada por duas ou mais condições simples, operadores booleanos e parênteses. Assumimos que os operadores booleanos permitidos em uma condição composta incluem OR (|), AND (&) e NOT (¬). Uma condição sem expressões relacionais é conhecida como expressão booleana.

Se uma condição é incorreta, então pelo menos um componente da condição é incorreto. Portanto, os tipos de erros em uma condição incluem erros de operador booleano (operadores booleanos incorretos, faltando ou extra), erros de variável booleana, erros de parênteses booleanos, erros de operador relacional e erros de expressão aritmética. O método de teste de condição focaliza o teste de cada condição no programa para garantir que ele não contenha erros.

"Bons testadores são mestres na arte de encontrar 'alguma coisa esquisita' e agir sobre ela."

Brian Marick

18.5.2 Teste de fluxo de dados

O método de teste de fluxo de dados [Fra93] seleciona caminhos de teste de um programa de acordo com as localizações de definições e usos de variáveis no programa. Para ilustrar a abordagem de teste de fluxo de dados, suponha que a cada comando em um programa é atribuído um número de comando único e que cada função não modifique seus parâmetros ou variáveis globais. Para um comando com S como seu número de comando,

$$\text{DEF}(S) = \{X \mid \text{comando } S \text{ contém uma definição de } X\}$$
$$\text{USE}(S) = \{X \mid \text{comando } S \text{ contém um uso de } X\}$$

Se o comando S for um *comando if* ou *de ciclo*, seu conjunto DEF está vazio e seu conjunto USE é baseado na condição do comando S. Dizemos que a definição da variável X no comando S é *considerada viva* no comando S' se existe um caminho do comando S para o comando S' que não contenha outra definição de X.

AVISO

É irreal supor que o teste de fluxo de dados será usado extensivamente ao se testar um grande sistema. No entanto, ele pode ser usado especificamente em áreas do software que sejam suspeitas.

Uma *cadeia definição-uso* (*DU*) da variável X é da forma $[X, S, S']$, onde S e S' são números de instrução, X está em DEF(S) e USE(S'), e a definição de X na instrução S está viva no comando S'.

Uma estratégia simples de teste de fluxo de dados é requerer que toda cadeia DU seja coberta pelo menos uma vez. Chamamos essa estratégia de estratégia de teste DU. Tem sido demonstrado que o teste DU não garante a abrangência de todos os desvios de um programa. No entanto, não é possível garantir que um desvio seja coberto pelo teste DU somente em raras situações como, por exemplo, construções se-então-senão nas quais a *parte então* não tem definição de qualquer variável e a *parte senão* não existe. Nessa situação, o desvio senão do comando *se* não é necessariamente coberto pelo teste DU.

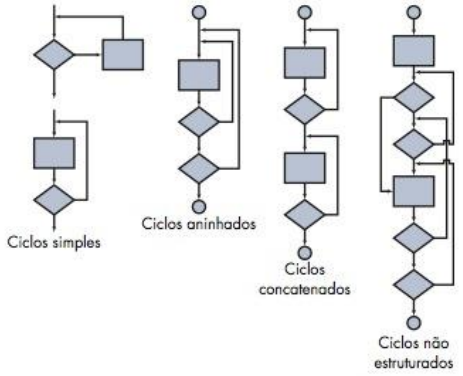
18.5.3 Teste de ciclo

Os ciclos (loop) são a pedra fundamental da grande maioria dos algoritmos implementados em software. E ainda, muitas vezes prestamos pouca atenção enquanto executamos testes de software.

O *teste de ciclo* é uma técnica de teste caixa-branca que focaliza exclusivamente a validade das construções de ciclo. Podem ser definidas quatro diferentes classes de ciclos [Bei90]: ciclos simples, ciclos concatenados, ciclos aninhados e ciclos não estruturados (Figura 18.7).

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

FIGURA 18.7
Classes de ciclos



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

Ciclos aninhados. Se fôssemos estender a abordagem de teste de ciclos simples para ciclos aninhados, o número de testes possíveis cresceria geometricamente à medida que o nível de aninhamento aumentasse. O resultado seria um número impossível de testes. Beizer [Bei90] sugere uma abordagem que ajudará a reduzir o número de testes:

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

Você não pode testar ciclos não estruturados eficientemente. Reprojete-os.

Ciclos concatenados. Ciclos concatenados podem ser testados usando a abordagem definida para ciclos simples, se cada um for independente do outro. No entanto, se dois ciclos forem concatenados e a contagem para o ciclo 1 for usada como valor individual para o ciclo 2, então os ciclos não são independentes. Quando os ciclos não são independentes, é recomendada a abordagem aplicada a ciclos aninhados.

Ciclos não estruturados. Sempre que possível, essa classe de ciclos deverá ser redesenhada para refletir o uso das construções de programação estruturada (Capítulo 10).

18.6 TESTE CAIXA-PRETA

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

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

"Errar é humano; encontrar um defeito é divino."

Robert Dunn

Diferentemente do teste caixa-branca, que é executado antecipadamente no processo de teste, o teste caixa-preta tende a ser aplicado durante estágios posteriores do teste (veja o Capítulo 17). Devido ao teste caixa-preta propositadamente desconsiderar a estrutura de controle, a atenção é focalizada no domínio das informações. Os testes são feitos para responder às seguintes questões:

Que perguntas são respondidas pelos testes caixa-preta?

- Como a validade funcional é testada?
- Como o comportamento e o desempenho do sistema é testado?
- Que classes de entrada farão bons casos de teste?
- O sistema é particularmente sensível a certos valores de entrada?
- Como as fronteiras de uma classe de dados é isolada?
- Que taxas e volumes de dados o sistema pode tolerar?
- Que efeito combinações específicas de dados terão sobre a operação do sistema?

PONTO-CHAVE

Um grafo representa a relação entre objetos dados e objetos programa, permitindo criar casos de teste que procuram por erros associados com essas relações.

Aplicando técnicas caixa-preta, você cria uma série de casos de teste que satisfaz aos seguintes critérios [Mye79]: (1) casos de teste que reduzem, de um valor maior do que um, o número de casos de teste adicionais que devem ser projetados para atingir um teste razoável; e (2) casos de teste que lhe dizem alguma coisa sobre a presença ou ausência de classes de erros, em vez de um erro associado somente com o teste específico que se está fazendo.

18.6.1 Métodos de teste com base em grafo

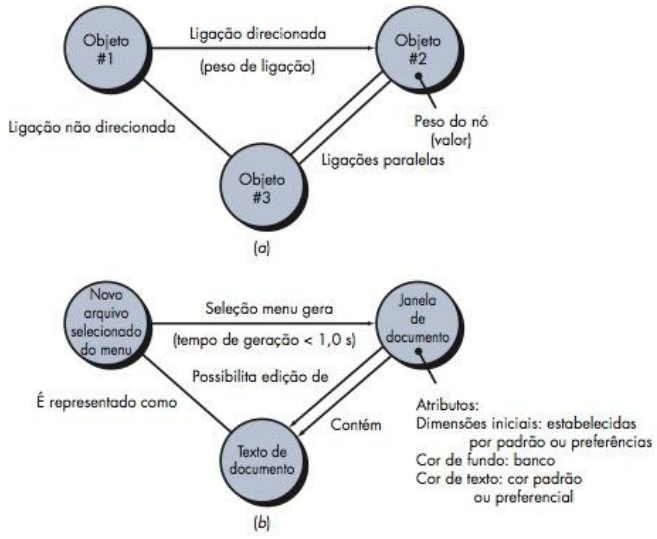
O primeiro passo no teste caixa-preta é entender os objetos[5] que são modelados no software e as relações que unem esses objetos. Uma vez conseguido isso, o próximo passo é definir uma série de testes que verificam que "todos os objetos têm a relação esperada uns com os outros" [Bei95]. Colocado de outra forma, o teste do software começa criando um grafo de objetos importantes e suas relações e então imaginando uma série de testes que abrangerá o grafo de forma que cada objeto e relação sejam exercitados e os erros sejam descobertos.

Para executar esses passos, você começa criando um *grafo* — uma coleção de *nós* que representam objetos, *ligações* que representam as relações entre objetos, *pesos de nó* que descrevem as propriedades de um nó (por exemplo, o valor específico de um dado ou comportamento de estado), e pesos de ligação (*link weights*) que descrevem alguma característica de uma ligação.

A representação simbólica de um grafo é mostrada na Figura 18.8a. Os nós são representados como círculos unidos por ligações que assumem várias formas diferentes. Uma *ligação direta* (representada por uma seta) indica que uma relação se move apenas em uma direção. Uma *ligação bidirecional,* também chamada de *ligação simétrica,* significa que a relação se aplica em ambas as direções. *Ligações paralelas* são usadas quando várias relações diferentes são estabelecidas entre os nós do grafo.

FIGURA 18.8

(a) Notação de grafo; (b) exemplo simples



5 Nesse contexto, considera-se o termo *objeto* no sentido mais amplo possível. Abrange os objetos de dados, componentes tradicionais (módulos) e elementos de software orientados a objeto

Como um exemplo simples, considere uma parte de um grafo para uma aplicação de processador de texto (Figura 18.8b) na qual

Objeto 1 = **arquivoNovo** (seleção de menu)
Objeto 2 = **janelaDeDocumento**
Objeto 3 = **textoDeDocumento**

De acordo com a figura, uma seleção de menu em **arquivoNovo** gera uma janela de documento. O peso do nó de **janelaDeDocumento** fornece uma lista dos atributos de janela que devem ser esperados quando a janela é gerada. O *peso da ligação* indica que a janela deve ser gerada em menos de 1,0 segundo. Uma *ligação indireta* estabelece uma relação simétrica entre a seleção de menu **arquivoNovo** e **textoDeDocumento,** e *ligações paralelas* indicam relações entre **janelaDeDocumento** e **textoDeDocumento.** Na realidade, teria de ser gerado um grafo muito mais detalhado como um precursor para o projeto do caso de teste. Você pode então criar casos de teste atravessando o grafo e percorrendo cada uma das relações mostradas. Esses casos de teste são projetados numa tentativa de encontrar erros em qualquer uma das relações. Beizer [Bei95] descreve uma série de métodos de teste comportamental que utilizam grafos:

Modelagem de fluxo de transação. Os nós representam passos em alguma transação (por exemplo, os passos necessários para fazer uma reserva em uma empresa aérea usando um serviço on-line), e as ligações representam a conexão lógica entre os passos (por exemplo, **flightInformationInput** [entrada de informação sobre o voo] é seguido por *validationAvailabilityProcessing* [processamento de validação da disponibilidade]). O grafo de fluxo de dados (Capítulo 7) pode ser usado para ajudar a criar grafos desse tipo.

Modelagem de estado finito. Os nós representam diferentes estados observáveis pelo usuário do software (por exemplo, cada uma das "telas" que aparecem quando um atendente recebe um pedido por telefone), e as ligações representam as transições que ocorrem para mover de um estado para outro (por exemplo, **orderInformation** [informação sobre o pedido] é verificada durante *inventoryAvailabilityLook-up* [consulta da disponibilidade no inventário] e é seguida pela entrada **customerBillingInformation** [informações para faturamento]). O grafo de estado (Capítulo 7) pode ser usado para ajudar a criar grafos desse tipo.

Modelagem de fluxo de dados. Os nós são objetos dados e as ligações são as transformações que ocorrem para traduzir um objeto dados em outro. Por exemplo, o nó imposto retido (FICA tax withheld – **FTW**) é calculado com base no salário bruto (Gross wages – **GW**) usando a relação, **FTW = 0,62 x GW.**

Modelagem no tempo. Os nós são objetos de programa, e as ligações são conexões sequenciais entre aqueles objetos. *Pesos de ligação* são usados para especificar os tempos de execução necessários enquanto o programa é executado.

Uma discussão detalhada de cada um desses métodos de teste com base em diagrama vai além do escopo deste livro. Se você estiver interessado, veja [Bei95] para uma descrição detalhada.

18.6.2 Particionamento de equivalência

AVISO

As classes de entrada são conhecidas relativamente no início da gestão de qualidade. Por essa razão, comece pensando sobre o particionamento de equivalência logo que o projeto for criado.

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

O projeto de casos de teste para particionamento de equivalência tem como base a avaliação das *classes de equivalência* para uma condição de entrada. Usando conceitos introduzidos na seção anterior, se um conjunto de objetos pode ser vinculado por relações simétricas, transitivas e reflexivas, uma classe de equivalência estará presente [Bei95]. Uma classe de equivalência representa um conjunto de estados válidos ou inválidos para condições de entrada. Tipicamente,

uma condição de entrada é um valor numérico específico, um intervalo de valores, um conjunto de valores relacionados ou uma condição booleana. Classes de equivalência podem ser definidas de acordo com as seguintes regras:

Como definir classes de equivalência para teste?

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

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

18.6.3 Análise de valor limite

"Uma maneira eficaz de testar código é exercitá-lo em suas fronteiras naturais."

Brian Kernighan

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

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

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

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

PONTO-CHAVE

A BVA estende o particionamento de equivalência focalizando os dados nas "fronteiras" de uma classe de equivalência.

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

18.6.4 Teste de matriz ortogonal

Há muitas aplicações nas quais o domínio de entrada é relativamente limitado. Isso significa que o número de parâmetros de entrada é pequeno e os valores que cada um dos parâmetros pode tomar estão claramente limitados. Quando esses números são muito pequenos (por exem-

plo, três parâmetros de entrada assumindo três valores discretos cada um), é possível considerar cada permutação de entrada e testar exaustivamente o domínio de entrada. No entanto, à medida que cresce o número de valores de entrada e o número de valores discretos para cada item de dado aumenta, o teste exaustivo torna-se impraticável ou impossível.

O *teste de matriz ortogonal* pode ser aplicado a problemas nos quais o domínio de entrada é relativamente pequeno, mas muito grande para acomodar o teste exaustivo. O método de teste de matriz ortogonal é particularmente útil para encontrar erros associados com falhas de regiões — uma categoria de erro associada com lógica defeituosa em um componente de software.

Para ilustrar a diferença entre teste de matriz ortogonal e abordagens mais convencionais do tipo "uma entrada de cada vez ", considere um sistema que tenha três itens de entrada, *X*, *Y* e *Z*. Cada um desses itens de entrada tem três valores discretos associados com ele. Existem $3^3 = 27$ casos de teste possíveis. Phadke [Pha97] sugere uma visualização geométrica dos casos de teste possíveis associados com X, Y e Z ilustrada na Figura 18.9. Olhando a figura, um item de entrada de cada vez pode ser variado em sequência ao longo de cada eixo de entrada. Isso resulta em uma abrangência relativamente limitada do domínio de entrada (representado pelo cubo da esquerda na figura).

Quando ocorre o teste de matriz ortogonal, é criada uma matriz ortogonal L9 de casos de teste. O conjunto ortogonal L9 tem uma "propriedade de balanceamento" [Pha97]. Ou seja, casos de teste (representados por pontos escuros na figura) são "dispersos uniformemente através do domínio do teste", conforme ilustra o cubo da direita na Figura 18.9. A cobertura de teste ao longo do domínio de entrada é mais completa.

PONTO-CHAVE

A BVA estende o particionamento de equivalência focalizando os dados nas "fronteiras" de uma classe de equivalência.

Para ilustrar o uso da matriz ortogonal L9, considere a função *envie* para uma aplicação de fax. São passados quatro parâmetros, P1, P2, P3, e P4, para a função *send*. Cada parâmetro assume três valores discretos. Por exemplo, P1 assume os valores:

P1 = 1, enviar agora
P1 = 2, enviar após 1 hora
P1 = 3, enviar depois da meia-noite

P2, P3, e P4 também assumiriam valores 1, 2 e 3, significando outras funções de envio.

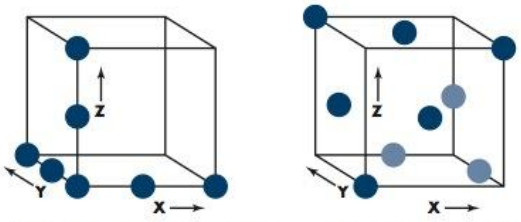
Se fosse escolhida a estratégia de teste "um item de entrada de cada vez ", seria especificada a seguinte sequência (P1, P2, P3, P4) de testes: (1, 1, 1, 1), (2, 1, 1, 1), (3, 1, 1, 1), (1, 2, 1, 1), (1, 3, 1, 1), (1, 1, 2, 1), (1, 1, 3, 1), (1, 1, 1, 2), e (1, 1, 1, 3). Phadke [Pha97] avalia esses casos de teste afirmando:

> Esses casos de teste somente são úteis quando se tem certeza de que esses parâmetros de teste não interagem. Eles podem detectar falhas de lógica onde um único valor de parâmetro faz o software falhar. Essas falhas são chamadas de *falhas de modo simples* (*single mode faults*). Esse método não pode detectar falhas lógicas que causam mau funcionamento quando dois ou mais parâmetros assumem simultaneamente certos valores; isto é, ele não pode detectar quaisquer interações. Portanto, sua habilidade em detectar falhas é limitada.

FIGURA 18.9

Uma visualização geométrica dos casos de teste

Fonte: **(Pha97)**



Um item de entrada de cada vez **Conjunto ortogonal L9**

Dado o número relativamente pequeno de parâmetros de entrada e valores discretos, o teste exaustivo é possível. O número de testes necessários é $3^4 = 81$, grande, porém controlável. Todas as falhas associadas com permutação de itens de dados seriam encontradas, mas o trabalho necessário é relativamente alto.

A abordagem de teste de matriz ortogonal permite obter boa abrangência de teste com bem menos casos de teste do que a estratégia exaustiva. Uma matriz ortogonal L9 para a função *envie* da aplicação de fax está ilustrada na Figura 18.10.

FIGURA 18.10
Uma matriz ortogonal L9

Casos de teste	Parâmetros de teste			
	P1	P2	P3	P4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

FERRAMENTAS DO SOFTWARE

Projeto de caso de teste

Objetivo: auxiliar a equipe de software no desenvolvimento de uma série completa de casos de teste, tanto para teste caixa-preta quanto para caixa-branca.

Mecanismos: essas ferramentas se classificam em duas categorias principais: ferramentas de teste estático e ferramentas de teste dinâmico. São usadas na indústria três tipos diferentes de ferramentas de teste estático: ferramentas de teste baseadas em código, linguagens especializadas de teste e ferramentas de teste baseadas em requisitos. As ferramentas de teste baseadas em código aceitam código-fonte como entrada e executam uma série de análises que resultam na geração de casos de teste. As linguagens de teste especializadas (por exemplo, ATLAS) permitem a um engenheiro de software escrever especificações detalhadas de teste que descrevem cada caso de teste e as logísticas para sua execução. Ferramentas de teste baseadas em requisitos isolam requisitos específicos de usuário e sugerem casos de teste (ou classes de testes) que exercitarão os requisitos. As ferramentas de teste dinâmico interagem com um programa em execução, verificando amplitude do caminho, testando asserções sobre o valor de variáveis específicas, e instrumentando o fluxo de execução do programa de qualquer modo.

Ferramentas representativas:[6]

McCabeTest, desenvolvida pela McCabe & Associates (**www.mccabe.com**), implementa uma variedade de técnicas de teste de caminho derivadas de uma avaliação da complexidade ciclomática e de outras métricas de software.

TestWorks, desenvolvida pela Software Research, Inc. (**www.soft.com/Products**), é uma série completa de ferramentas de teste automáticas que ajudam no projeto de casos de teste para software desenvolvido em C/C++ e Java e proporciona suporte para teste de regressão.

T-VEC Test Generation System, desenvolvida pela T-VEC Technologies (**www.t-vec.com**), é uma ferramenta que suporta teste de unidade, integração e validação ajudando no projeto de casos de teste usando informações contidas em uma especificação de requisitos Orientados a Objeto (OO).

e-TEST Suite, desenvolvida pela Empirix, Inc. (**www.empirix.com**), abrange uma série completa de ferramentas para testar WebApps, incluindo ferramentas que ajudam no projeto de casos de teste e planejamento de teste.

6 A citação de ferramentas aqui não representa um endosso, mas sim uma amostragem das ferramentas nessa categoria. Na maioria dos casos, os nomes das ferramentas são marcas registradas pelos seus respectivos desenvolvedores.

Phadke [Pha97] avalia o resultado dos testes usando a matriz ortogonal L9 da seguinte maneira:

> **Detecte e isole todas as falhas de modo simples.** Uma falha de modo simples é um problema consistente com qualquer nível de qualquer parâmetro isolado. Por exemplo, se todos os casos de teste de fator P1 = 1 causam uma condição de erro, trata-se de uma falha de modo simples. Nesse exemplo, os testes 1, 2 e 3 [Figura 18.10] mostrarão erros. Analisando as informações sobre quais testes mostram erros, pode-se identificar que valores de parâmetros causam a falha. Nesse exemplo, observando que os testes 1, 2 e 3 causam um erro, podemos isolar [o processamento lógico associado com "enviar agora" (P1 = 1)] como origem do erro. Esse isolamento da falha é importante para poder corrigi-la.
>
> **Detecte todas as falhas de modo duplo.** Se existe um problema consistente quando níveis específicos de dois parâmetros ocorrem juntos, chamamos isso de *falha de modo duplo* (*double mode fault*). Sem dúvida, uma falha de modo duplo é uma indicação de incompatibilidade do par ou interações danosas entre dois parâmetros de teste.
>
> **Falhas multimodo.** Matrizes ortogonais [do tipo mostrado] somente podem garantir a detecção de falhas de modo simples e duplo. No entanto, muitas falhas multimodo também são detectadas por esses testes.

Uma discussão detalhada do teste de matriz ortogonal pode ser encontrada em [Pha89].

18.7 TESTE BASEADO EM MODELOS

"É bastante difícil encontrar um erro no seu código quando você o está procurando; é ainda mais difícil quando você assume que o seu software é isento de erros."

Steve McConnell

Teste baseado em modelo (*Model-based testing* - MBT) é uma técnica de teste caixa-preta que usa informações contidas no modelo de requisitos como base para a geração de casos de teste. Em muitos casos, as técnicas baseadas em modelo usam diagramas de estado UML, um elemento do modelo comportamental (Capítulo 7), como base para o projeto de casos de teste.[7] A técnica MBT requer cinco passos:

1. **Analise um modelo comportamental existente para o software ou crie um.** Lembre-se de que o *modelo comportamental* indica como o software responderá a eventos ou estímulos externos. Para criar o modelo, você deverá executar os passos discutidos no Capítulo 7: (1) avaliar todos os casos de uso para entender completamente a sequência de interação dentro do sistema, (2) identificar eventos que controlam a sequência de interação e entender como esses eventos se relacionam com objetos específicos, (3) criar uma sequência para cada caso de uso, (4) criar um diagrama de estado UML para o sistema (por exemplo, veja a Figura 7.6), e (5) rever o modelo comportamental para verificar exatidão e consistência.
2. **Percorra o modelo comportamental e especifique as entradas que forçarão o software a fazer a transição de um estado para outro.** As entradas irão disparar eventos que farão a transição ocorrer.
3. **Reveja o modelo comportamental e observe as saídas esperadas à medida que o software faz a transição de um estado para outro.** Lembre-se de que cada transição de estado é disparada por um evento e que, em consequência da transição, alguma função é chamada e saídas são criadas. Para cada conjunto de entradas (casos de teste) que você especificou no passo 2, especifique as saídas esperadas como elas são caracterizadas no modelo comportamental. "Uma hipótese fundamental desse teste é que há algum mecanismo, uma *previsão de teste* (*test oracle*), que determinará se os resultados da execução de um teste são ou não corretos " [DAC03]. Essencialmente, uma previsão de teste estabelece a base para qualquer determinação da precisão da saída. Em muitos casos, a previsão é o modelo de requisitos, mas poderia ser também outro documento ou aplicação, dados gravados em algum lugar, ou mesmo um especialista humano.

7 O teste baseado em modelo também pode ser usado quando os requisitos de software são representados por tabelas de decisões, gramáticas, ou cadeias de Markov [DAC03].

4. **Execute os casos de teste.** Os testes podem ser executados manualmente ou pode ser criado um *script* de teste e executado usando uma ferramenta de teste.
5. **Compare os resultados real e esperado e tome a ação corretiva necessária**.

O MBT ajuda a descobrir erros no comportamento do software, e consequentemente, é extremamente útil ao testar aplicações acionadas por eventos.

18.8 TESTE PARA AMBIENTES, ARQUITETURAS E APLICAÇÕES ESPECIALIZADOS

Às vezes, é possível assegurar diretrizes e abordagens únicas quando são considerados ambientes, arquiteturas e aplicações especializados. Embora as técnicas de teste discutidas anteriormente neste capítulo e nos Capítulos 19 e 20 possam muitas vezes ser adaptadas a situações específicas, é importante considerar individualmente suas necessidades especiais.

18.8.1 Testando GUIs

As Interfaces Gráficas de Usuário (*Graphical user interfaces* - GUIs) apresentam desafios interessantes de teste. Devido aos componentes reutilizáveis serem agora parte comum do desenvolvimento de ambientes de GUI, a criação de interfaces de usuário tornou-se mais rápida e mais precisa (Capítulo 11). Mas, ao mesmo tempo, a complexidade das GUIs tem crescido, resultando em dificuldades maiores no projeto e execução de casos de teste.

Como muitas GUIs modernas têm a mesma aparência e comportamento, é possível criar uma série de testes padronizados. Grafos de modelagem de estado finito podem ser usados para criar uma série de testes que cuidam de objetos específicos de dados e programa relevantes para a GUI. Essa técnica de teste baseada em modelo foi discutida na Seção 18.7.

Devido ao grande número de permutações associadas com operações GUI, o teste de GUI deverá ser abordado usando ferramentas automatizadas. Uma ampla variedade de técnicas de teste GUI surgiu no mercado nos últimos anos.[8]

18.8.2 Teste de arquiteturas cliente-servidor

"O tópico de testes é uma área na qual existe uma boa dose de pontos comuns entre sistema tradicional e sistemas cliente/servidor."

Kelley Bourne

WebRef

Informações e recursos úteis para teste cliente-servidor podem ser encontrados em **www.csttechnologies.com**.

Que tipos de testes são executados para sistemas cliente-servidor?

A natureza distribuída de ambientes cliente-servidor, os aspectos de desempenho associados com processamento de transações, a presença potencial de muitas plataformas de hardware diferentes, a complexidade das comunicações em rede, a necessidade de servir a múltiplos clientes a partir de uma base de dados centralizada (ou em alguns casos, distribuída), e os requisitos de coordenação impostos sobre o servidor, tudo se combina para tornar o teste de arquiteturas cliente-servidor e o software que reside nelas consideravelmente mais difícil do que para aplicações individualizadas. De fato, estudos recentes indicam um aumento significativo no tempo e custo de teste quando são desenvolvidos ambientes cliente-servidor.

Em geral, o teste de software cliente-servidor ocorre em três níveis diferentes: (1) Aplicações cliente individuais são testadas em um modo "desconectado"; a operação do servidor e a rede subjacente não são consideradas. (2) O software cliente e as aplicações servidor associadas são testados em harmonia, mas as operações de rede não são exercitadas explicitamente. (3) É testada a arquitetura completa cliente-servidor, incluindo operação e desempenho de rede.

Embora sejam executados muitos tipos diferentes de testes em cada um desses níveis de detalhe, as abordagens de teste a seguir são muitas vezes encontradas para aplicações cliente-servidor:

- **Testes de função de aplicação.** A funcionalidade das aplicações cliente é testada usando os métodos discutidos anteriormente neste capítulo e nos Capítulos 19 e 20. Essencialmente, a aplicação é testada no modo individual (*stand-alone*), na tentativa de descobrir erros em sua operação.

8 Centenas, se não milhares de recursos de ferramentas de teste GUI podem ser avaliados na Internet. Um bom ponto de partida para ferramentas de código aberto (*open-source*) é **www.opensourcetesting.org/functional.php**.

- **Testes de servidor.** São testadas as funções de coordenação e gerenciamento de dados do servidor. É considerado também o desempenho do servidor (tempo de resposta geral e taxa de saída de dados).
- **Testes de base de dados.** É testada a exatidão e a integridade dos dados armazenados pelo servidor. São examinadas as transações postadas por aplicações cliente para assegurar que os dados estejam corretamente armazenados, atualizados e acessados. É testado também o arquivamento.
- **Testes de transação.** É criada uma série de testes para garantir que cada classe de transações seja processada de acordo com os requisitos. Os testes focalizam a exatidão do processamento e também os problemas de desempenho (por exemplo, tempos de processamento de transação e volume de transação).
- **Testes de comunicação de rede.** Esses testes verificam se as comunicações entre os nós da rede ocorrem corretamente e se a passagem de mensagens, transações e tráfego relacionado com a rede ocorrem sem erro. Podem ser executados também testes de segurança de rede como parte desses testes.

Para completar essas abordagens de teste, Musa [Mus93] recomenda o desenvolvimento de *perfis operacionais* derivados de cenários de uso cliente-servidor.[9] Um perfil operacional indica como diferentes tipos de usuários interoperam com o sistema cliente-servidor. Isto é, os perfis fornecem um "padrão de uso" que pode ser aplicado quando os testes são projetados e executados. Por exemplo, para um tipo particular de usuário, que porcentagem das transações será de consultas? de atualizações? de pedidos?

Para desenvolver o perfil operacional, é necessário criar um conjunto de cenários que seja similar aos casos de uso (Capítulos 5 e 6). Cada cenário esclarece quem, onde, o que e por quê: quem é o usuário, onde (na arquitetura física cliente-servidor) ocorre a interação do sistema, qual é a transação e por que ela ocorreu. Cenários podem ser criados usando técnicas de extração de requisitos (Capítulo 5) ou por meio de discussões menos formais com os usuários finais. O resultado, no entanto, deverá ser o mesmo. Cada cenário deverá fornecer uma indicação das funções do sistema que serão necessárias para atender a um usuário em particular, a ordem na qual aquelas funções são necessárias, o tempo e resposta esperados e a frequência com que cada função é usada. Esses dados são então combinados (para todos os usuários) para criar o perfil operacional. Em geral, o trabalho de teste e o número de casos de teste a ser executado são alocados para cada cenário de usuário com base na frequência de uso e na criticidade das funções executadas.

18.8.3 Testando a documentação e os recursos de ajuda

O termo *teste de software* passa a imagem de grande quantidade de casos de teste preparados para exercitar programas de computador e os dados que manipulam. Recordando a definição de software apresentada no Capítulo 1, é importante notar que o teste também deve se estender ao terceiro elemento da configuração do software — a documentação.

Erros na documentação podem ser tão devastadores para a aceitação dos programas quanto os erros nos dados ou no código-fonte. Nada é mais frustrante do que seguir um guia do usuário ou um recurso de ajuda on-line exatamente e obter resultados ou comportamentos que não coincidem com aqueles previstos pela documentação. É por essa razão que o teste da documentação deve ser parte significativa de todo plano de teste de software.

O teste da documentação pode ser feito em duas fases. A primeira fase, a revisão técnica (Capítulo 15), examina o documento quanto à clareza de edição. A segunda fase, o teste ao vivo, usa a documentação em conjunto com o programa real.

---

9 Deve-se notar que os perfis operacionais podem ser usados no teste de todos os tipos de arquiteturas de sistema, não apenas na arquitetura cliente-servidor.

Surpreendentemente, um teste ao vivo para documentação pode ser feito usando técnicas análogas a muitos métodos de teste caixa-preta discutidos anteriormente. Teste baseado em diagrama pode ser usado para descrever o uso do programa; particionamento de equivalência e análise de valor limite podem ser usados para definir várias classes de entradas e interações associadas. MBT pode ser usado para garantir que o comportamento documentado e o comportamento real coincidam. O uso do programa é então acompanhado com base da utilização da documentação.

INFORMAÇÕES

Teste de Documentação

As seguintes questões deverão ser respondidas durante o teste de documentação e/ou recurso de ajuda:

- A documentação descreve com precisão como proceder em cada modo de utilização?
- A descrição de cada sequência de interação é precisa?
- Os exemplos são precisos?
- A terminologia, as descrições dos menus e as respostas do sistema são consistentes com o programa real?
- É relativamente fácil localizar diretrizes dentro da documentação?
- A solução de problemas pode ser obtida facilmente com a documentação?
- O sumário e o índice do documento são bons, exatos e completos?
- O estilo do documento (layout, fontes, recuos, gráficos) conduz ao entendimento e rápida assimilação das informações?
- Todas as mensagens de erro do software que aparecem para o usuário estão descritas em mais detalhes no documento? As ações a ser tomadas em consequência de uma mensagem de erro estão claramente delineadas?
- Se forem usadas ligações de hipertexto, elas são precisas e completas?
- Se for usado hipertexto, o estilo de navegação é apropriado para as informações requeridas?

A única maneira viável de responder a essas questões é através de uma consultoria independente (por exemplo, usuários escolhidos) para testar a documentação no contexto do uso do programa. Todas as discrepâncias são anotadas, e as áreas do documento que apresentam ambiguidade ou deficiências são marcadas para ser reescritas.

18.8.4 Teste para sistema em tempo real

A natureza assíncrona, dependente do tempo, de muitas aplicações em tempo real acrescenta um elemento novo e potencialmente difícil ao conjunto de testes — o tempo. O projetista do caso de teste tem de considerar não apenas os casos de teste convencionais mas também a manipulação de eventos (o processamento de interrupção), a temporização dos dados e o paralelismo das tarefas (processos) que manipulam os dados. Em muitas situações, dados de teste fornecidos quando o sistema em tempo real está em um estado resultarão em um processamento correto, enquanto os mesmos dados fornecidos quando o sistema está em um estado diferente podem resultar em erro.

Por exemplo, o software em tempo real que controla uma nova fotocopiadora aceita interrupções do operador (o operador da máquina pressiona teclas de controle como RESET ou ESCURECER) sem resultar em erro quando a máquina está fazendo cópias (no estado "copiando"). Essas mesmas interrupções do operador, se forem acionadas quando a máquina está no estado "atolada" (*jammed*), causam a perda do código de diagnóstico que indicava a localização do enrosco (um erro).

Além disso, a relação íntima que existe entre um software em tempo real e seu ambiente de hardware pode também causar problemas de teste. Testes de software devem levar em consideração o impacto das falhas do hardware sobre o processamento do software. Essas falhas podem ser extremamente difíceis de simular realisticamente.

Métodos de projeto de casos de teste abrangentes para sistemas em tempo real continuam evoluindo. No entanto, pode-se propor uma estratégia geral de quatro passos:

- **Teste de tarefa.** O primeiro passo no teste de software de tempo real é testar cada tarefa independentemente. Testes convencionais são projetados para cada tarefa e executados

Qual é uma estratégia eficaz para testar um sistema em tempo real?

independentemente durante esses testes. Testes de tarefa descobrem erros em lógica e funções, mas não em temporização ou comportamento.

- **Teste comportamental.** Usando modelos de sistema criados com ferramentas automáticas, é possível simular o comportamento de um sistema em tempo real e examinar seu comportamento em consequência de eventos externos. Essas atividades de análise podem servir como base para o projeto de casos de teste executados quando o software de tempo real é criado. Usando uma técnica que é similar ao particionamento de equivalência (Seção 18.6.2), eventos (por exemplo, interrupções, sinais de controle) são classificados para teste. Por exemplo, eventos para a fotocopiadora podem ser interrupções de usuário (por exemplo, zerar o contador), interrupções mecânicas (por exemplo, papel enroscado), interrupções de sistema (por exemplo, pouco toner), e modos de falha (por exemplo, rolo superaquecido). Cada um desses eventos é testado individualmente, e o comportamento do sistema executável é examinado para detectar erros que ocorrem em consequência do processamento associado com esses eventos. O comportamento do modelo de sistema (desenvolvido durante a atividade de análise) e o software executável podem ser comparados quanto ao desempenho. Uma vez testada cada classe de eventos, os eventos são apresentados ao sistema em ordem aleatória e com frequência aleatória. O comportamento do software é examinado para detectar erros de comportamento.
- **Teste intertarefas.** Uma vez isolados os erros em tarefas individuais e no comportamento do sistema, o teste passa para os erros relacionados com o tempo. Tarefas assíncronas que devem comunicar-se umas com as outras são testadas com diferentes taxas de dados e carga de processamento para determinar se ocorrerão erros de sincronização intertarefas. Além disso, tarefas que se comunicam via fila de mensagens ou armazenamento de dados são testadas para descobrir erros no dimensionamento dessas áreas de armazenamento.
- **Teste de sistema.** O software e o hardware são integrados, e é feita uma gama completa de testes do sistema na tentativa de descobrir erros na interface software-hardware. A maioria dos sistemas em tempo real processa interrupções. Portanto, é essencial testar a manipulação desses eventos booleanos. Usando o diagrama de estado (Capítulo 7), o testador desenvolve uma lista de todas as interrupções possíveis e o processamento que ocorre em consequência das interrupções. São planejados, então, testes para avaliar as seguintes características do sistema:
- São atribuídas prioridades corretas às interrupções e elas são manipuladas corretamente?
- O processamento de cada interrupção é manipulado corretamente?
- O desempenho (por exemplo, tempo de processamento) de cada procedimento de manipulação de interrupção está conforme os requisitos?
- A chegada de um alto volume de interrupções em instantes críticos cria problemas em função ou desempenho?

Além disso, deverão ser testadas as áreas globais de dados usadas para transferir informações como parte do processamento de interrupção, para avaliar o potencial de geração de efeitos colaterais.

18.9 PADRÕES PARA TESTE DE SOFTWARE

WebRef

Um catálogo de padrões de teste de software pode ser encontrado em **www.rbsc.com/pages/TestPatternList.htm**.

O uso de padrões como um mecanismo para descrever soluções para problemas específicos de projeto foi discutido no Capítulo 12. Mas os padrões também podem ser usados para propor soluções para outras situações de engenharia de software — nesse caso, o teste de software. Os *padrões de teste* descrevem problemas comuns de teste e soluções que podem ajudar a lidar com eles.

Os padrões de teste fornecem não apenas diretrizes úteis no início das atividades de teste, mas também trazem três benefícios adicionais descritos por Marick [Mar02]:

PONTO-CHAVE

A BVA estende o particionamento de equivalência focalizando os dados nos "fronteiros" de uma classe de equivalência.

1. Eles [os padrões] fornecem um vocabulário para solucionadores de problemas. "Ei, sabe, deveríamos usar um Objeto nulo."
2. Eles focalizam a atenção nas forças que estão por trás de um problema. Isso permite que os projetistas [de casos de teste] entendam melhor quando e por que uma solução se aplica.
3. Eles estimulam o pensamento iterativo. Cada solução cria um novo contexto no qual novos problemas podem ser resolvidos.

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

WebRef

Padrões que descrevem organização de teste, eficiência, estratégia e solução de problemas podem ser encontrados em: **www.testing.com/test-patterns/patterns/**.

Os padrões de teste são descritos de maneira muito semelhante aos padrões de projeto (Capítulo 12). Já foram propostos dezenas de padrões de testes na literatura (por exemplo, [Mar02]). Os três padrões a seguir (apresentados apenas de uma forma superficial) fornecem exemplos representativos:

Nome do padrão: **Teste aos pares**

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

Nome do padrão: **Interface de teste separada**

Resumo: Há necessidade de testar todas as classes em um sistema orientado a objeto, incluindo "classes internas" (isto é, classes que não expõem qualquer interface fora do componente que as utilizou). O padrão **Interface de teste separada** descreve como criar "uma interface de teste que pode ser usada para descrever testes específicos em classes que são visíveis somente internamente a um componente" [Lan01].

Nome do padrão: **Teste de cenário**

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

Uma discussão clara dos padrões de teste está além do escopo deste livro. Se você estiver interessado em mais detalhes, veja em [Bin99] e [Mar02] informações adicionais sobre esse importante tópico.

18.10 RESUMO

O objetivo primário do projeto de caso de teste é criar uma série de testes que tenha a mais alta probabilidade de descobrir erros no software. Para conseguir esse objetivo, são usadas duas categorias diferentes de técnicas de projeto de caso de teste: teste caixa-branca e teste caixa-preta.

Os testes caixa-branca focalizam a estrutura de controle do programa. São criados casos de teste para assegurar que todas as instruções no programa foram executadas pelo menos uma vez durante o teste e que todas as condições lógicas foram exercitadas. O teste de caminho base, uma técnica caixa-branca, usa diagramas de programa (ou matrizes gráficas) para derivar o conjunto de testes linearmente independentes que garantirão abranger todas as instruções. O teste de condições e de fluxo de dados exercita mais a lógica do programa, e o teste de ciclos complementa outras técnicas caixa-branca fornecendo um procedimento para exercitar ciclos com vários graus de complexidade.

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

Os testes caixa-preta são projetados para validar requisitos funcionais sem levar em conta o funcionamento interno de um programa. As técnicas caixa-preta focalizam o domínio de informações do software, derivando casos de teste e particionando o domínio de entrada e saída de um programa de forma a proporcionar uma ampla cobertura do teste. O particionamento de equivalência divide o domínio de entrada em classes de dados que tendem a usar uma função específica do software. A análise de valor limite investiga a habilidade do programa para manipular dados nos limites do aceitável. O teste de matriz ortogonal proporciona um método eficiente e sistemático para testar sistemas com poucos parâmetros de entrada. Teste baseado em modelo usa elementos do modelo de requisitos para testar o comportamento de um aplicativo.

Métodos especializados de teste abrangem um grande conjunto de recursos de software e áreas de aplicação. O teste de interfaces gráficas de usuário, arquiteturas cliente-servidor, documentação e recursos de ajuda e sistemas em tempo real requerem, cada um deles, diretrizes e técnicas especializadas.

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

Problemas e Pontos a Ponderar

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

18.2. Projete e implemente o programa (com manipulação de erro onde for apropriado) especificado no Problema 18.1. Crie um grafo de fluxo para o programa e aplique teste de caminho base para desenvolver casos de teste que garantirão que todos os comandos no programa foram testados. Execute os casos e mostre os seus resultados.

18.3. Você consegue pensar em objetivos de teste adicionais que não foram discutidos na Seção 18.1.1?

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

18.5. Especifique, projete e implemente uma ferramenta de software que calculará a complexidade ciclomática para a linguagem de programação de sua escolha. Use uma matriz de grafo como estrutura de dados operativos em seu projeto.

18.6. Leia Beizer [Bei95] ou alguma outra fonte de informação relacionada com a Internet (por exemplo, **www.laynetworks.com/ Discrete%20Mathematics_1g.htm**) e determine como o programa que você desenvolveu no Problema 18.5 pode ser ampliado para acomodar vários pesos de ligação. Amplie a sua ferramenta para probabilidades de execução de processo ou tempos de processamento de ligação.

18.7. Projete uma ferramenta automatizada que reconheça ciclos e os classifique conforme indicado na Seção 18.5.3.

18.8. Amplie a ferramenta descrita no Problema 18.7 para gerar casos de teste para cada categoria de ciclo, quando encontrado. Será necessário executar essa função interativamente com o testador.

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

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

18.11. Teste um manual de usuário (ou um recurso de ajuda) para uma aplicação que você usa frequentemente. Encontre pelo menos um erro na documentação.

Leituras e Fontes de Informação Complementares

Praticamente todos os livros dedicados a teste de software consideram tanto estratégia quanto táticas. Portanto, as leituras complementares citadas no Capítulo 17 são igualmente aplicáveis para este capítulo. Everett e Raymond (*Software Testing,* Wiley-IEEE Computer Society Press, 2007), Black (*Pragmatic Software Testing,*Wiley, 2007), Spiller et al. (*Software Testing Process: Test Management,* Rocky Nook, 2007), Perry (*Effective Methods for Software Testing,* 3d ed., Wiley, 2005), Lewis (*Software Testing and Continuous Quality Improvement,* 2d ed., Auerbach, 2004), Loveland et al. (*Software Testing Techniques,* Charles River Media, 2004), Burnstein (*Practical Software Testing,* Springer, 2003), Dustin (*Effective Software Testing,* Addison-Wesley, 2002), Craig e Kaskiel (*Systematic Software Testing,* Artech House, 2002), Tamres (*Introducing Software Testing,* Addison-Wesley, 2002) e Whittaker (*How to Break Software,* Addison-Wesley, 2002) são apenas uma pequena amostra dos muitos livros que discutem princípios de teste, conceitos, estratégias e métodos.

Uma segunda edição do trabalho clássico de Myers [Mye79] foi produzida por Myers e seus colegas (*The Art of Software Testing,* 2d ed., Wiley, 2004) e abrange técnicas de projeto de caso de teste com detalhe considerável. Pezze e Young (*Software Testing and Analysis,* Wiley, 2007), Perry (*Effective Methods for Software Testing,* 3d ed., Wiley, 2006), Copeland (*A Practitioner's Guide to Software Test Design,* Artech, 2003), Hutcheson (*Software Testing Fundamentals,* Wiley, 2003), Jorgensen (*Software Testing: A Craftsman's Approach,* 2d ed., CRC Press, 2002) todos eles fornecem apresentações úteis de métodos e técnicas de projeto de casos de teste. A obra clássica de Beizer [Bei90] contém uma abrangência clara de técnicas caixa-branca, introduzindo um nível de rigor matemático que frequentemente falta em outros tratados sobre teste. Seu mais recente livro [Bei95] apresenta um tratamento conciso de importantes métodos.

Teste de software é uma atividade intensiva em termos de recursos. É por essa razão que muitas organizações automatizam partes do processo de teste. Livros de Li e Wu (*Effective Software Test Automation,* Sybex, 2004); Mosely e Posey (*Just Enough Software Test Automation,* Prentice-Hall, 2002); Dustin, Rashka, e Poston (*Automated Software Testing: Introduction, Management, and Performance,* Addison-Wesley, 1999); Graham et al. (*Software Test Automation,* Addison-Wesley, 1999); e Poston (*Automating Specification-Based Software Testing,* IEEE Computer Society, 1996) discutem ferramentas, estratégias e métodos para teste automático. Nquyen et al. (*Global Software Test Automation,* Happy About Press, 2006) apresentam uma visão geral executiva da automação de teste.

Thomas e seus colegas (*Java Testing Patterns,* Wiley, 2004) e Binder [Bin99] descrevem padrões de teste que abrangem métodos de teste, classes/clusters, subsistemas, componentes reutilizáveis, frameworks e sistemas, bem como automação de teste e testes especializados de bases de dados.

Há disponível na Internet uma ampla variedade de recursos de informação sobre métodos de projeto de casos de teste. Uma lista atualizada das referências relevantes para as técnicas de teste pode ser encontrada no site **www.mhhe.com/engcs/compsci/pressman/professional/ olc/ser.htm**.