5 Teste do Módulo (Unidade)

Até este ponto, ignoramos amplamente a mecânica de testes e o tamanho do programa que está sendo testado. No entanto, como grandes programas (digamos, de 500 instruções ou mais de 50 classes) requerem tratamento de teste especial, neste capítulo, consideramos um passo inicial na estruturação do teste de um grande programa: teste de módulo. Os capítulos 6 e 7 enumeram as etapas restantes.

Teste de módulo (ou teste de unidade) é um processo de teste de subprogramas, sub-rotinas, classes ou procedimentos individuais em um programa. Mais especificamente, em vez de testar inicialmente o programa como um todo, testar é primeiro focado nos blocos de construção menores do programa. As motivações para fazer isso são três. Primeiro, o teste de módulo é uma maneira de gerenciar o elementos combinados de teste, uma vez que a atenção está focada inicialmente em unidades do programa. Em segundo lugar, o teste de módulo facilita a tarefa de depuração (o processo de identificar e corrigir um erro descoberto), uma vez que, quando um erro for encontrado, ele é conhecido por existir em um módulo específico. Finalmente, o teste de módulo introduz o paralelismo no processo de teste de programa, apresentando-nos a oportunidade de testar vários módulos simultaneamente.

O objetivo do teste de módulo é comparar a função de um módulo com alguma especificação funcional ou de interface que define o módulo. Para enfatizar novamente o objetivo de todos os processos de teste, o objetivo aqui não é mostrar que o módulo atende sua especificação, mas que o módulo contradiz a especificação. Neste capítulo, abordamos o teste de módulo de três pontos de vista:

1. A maneira como os casos de teste são projetados.
2. A ordem em que os módulos devem ser testados e integrados.
3. Orientação sobre a realização dos testes.

Projeto de Caso de Teste

Você precisa de dois tipos de informações ao projetar casos de teste para um teste de módulo: uma especificação para o módulo e o código-fonte do módulo. A especificação normalmente define os parâmetros de entrada e saída do módulo e sua função.

O teste de módulo é amplamente orientado para a caixa branca. Uma razão é que, à medida que você testa entidades maiores, como programas inteiros (que será o caso de processos de teste subsequentes), o teste de caixa branca se torna menos viável. Uma segunda razão é que os processos de teste subsequentes são orientados para encontrar diferentes tipos de erros (por exemplo, erros não necessariamente associados à lógica do programa, como o programa não atender aos requisitos de seus usuários). Portanto, o procedimento de projeto de caso de teste para um teste de módulo é o seguinte:

Analise a lógica do módulo usando um ou mais métodos de caixa branca e, em seguida, complemente esses casos de teste aplicando métodos de caixa preta à especificação do módulo.

Os métodos de projeto de casos de teste que usaremos foram definidos no Capítulo 4; nós irá ilustrar seu uso em um módulo de teste aqui através de um exemplo.

Suponha que desejamos testar um módulo chamado BÔNUS e sua função é adicionar $ 2.000 ao salário de todos os funcionários do departamento ou departamentos com a maior receita de vendas. No entanto, se o salário atual de um funcionário qualificado for de US$ 150.000 ou mais, ou se o funcionário for um gerente, o salário deverá ser aumentado em apenas US$ 1.000.

As entradas do módulo são mostradas nas tabelas da Figura 5.1. Se o módulo executar sua função corretamente, ele retornará um código de erro de 0. Se o funcionário ou a tabela de departamentos não contiverem entradas, ele retornará um código de erro de 1. Se não encontrar nenhum funcionário em um departamento elegível, retornará um erro código de 2.

O código-fonte do módulo é mostrado na Figura 5.2. Os parâmetros de entrada ESIZE e DSIZE contêm o número de entradas nas tabelas de funcionários e departamentos. Observe que, embora o módulo seja escrito em PL/1, a discussão a seguir é amplamente independente da linguagem; as técnicas são aplicáveis a programas codificados em outras linguagens. Além disso, como a lógica PL/1 no módulo é bastante simples, praticamente qualquer leitor, mesmo aqueles não familiarizados com PL/1, deve ser capaz de entendê-la.

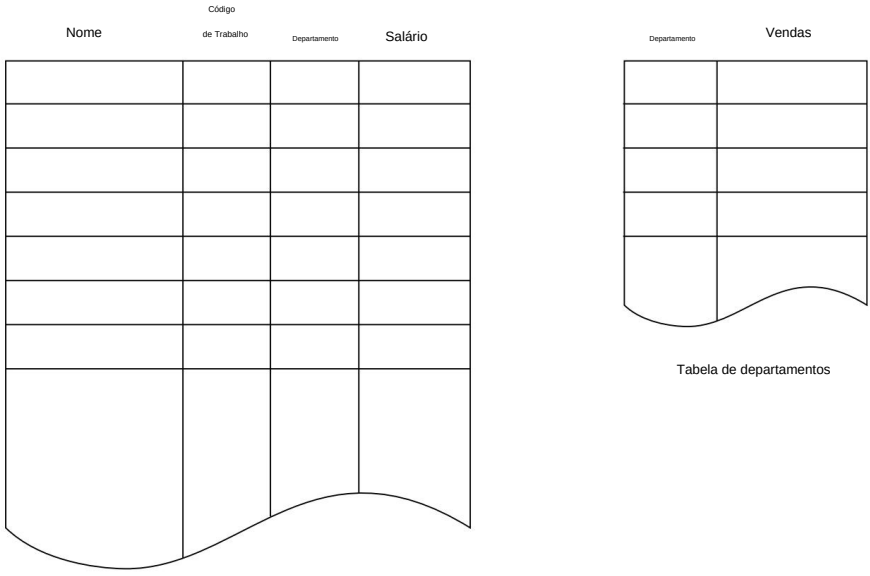

FIGURA 5.1 Tabelas de Entrada para o Módulo BÔNUS.

```
BÔNUS: PROCEDIMENTO (EMPTAB,DEPTTAB,ESIZE,DSIZE,ERRCODE);
DECLARAR 1 EMPTAB (*),
                2 NOME CARACTERÍSTICA(6),
                2 CARACTERÍSTICAS DE CÓDIGO(1),
                2 DEP CHAR(3),
                2 DECIMAL FIXO DE SALÁRIO(7,2);
DECLARAR 1 DEPTTAB (*),
                2 DEP CHAR(3),
                2 VENDAS DECIMAL FIXO(8,2);
DECLARE (ESIZE,DSIZE) BINÁRIO FIXO;
DECLARE ERRCODE DECIMAL FIXO(1);
DECLARE MAXSALES FIXED DECIMAL(8,2) INIT(0); /*MAX. VENDAS EM DEPTTAB*/ DECLARAR
(I,J,K) BINÁRIO FIXO; /*CONTADORES*/ DECLARE ENCONTRADO BIT(1); /*CHAVE DE PESQUISA
SINC DECIMAL FIXO(7,2) INIT(200,00); /*INCREMENTO PADRÃO*/ DECLARE LINC FIXED
DECIMAL(7,2) INIT(100.00); /*INCREMENTO INFERIOR*/ DECLARAR O SALÁRIO FIXO
DECIMAL(7,2) INIT(15000.00); /*LIMITE DE SALÁRIO*/

DECLARE MGR CHAR(1) INIT('M');                                    (contínuo)
```

FIGURA 5.2 Módulo BÔNUS.

```
1 ERRCODE=0;
2 SE(TAMANHO<=0)|(TAMANHO<=0)
3       ENTÃO CÓDIGO DE ERRO=1;                         /*EMPTAB OU DEPTTAB ESTÃO VAZIOS*/
4       MAIS FAÇA;
5           FAÇO I = 1 PARA DIMENSIONAR;                /* ENCONTRAR MAXSALES E MAXDEPTS*/
6               IF(VENDAS(I)>=MAXVENDAS) THEN MAXVENDAS=VENDAS(I);
7           FIM;
8           DO J = 1 PARA DIMENSIONAR;
9               IF(VENDAS(J)=MAXVENDAS)                 /*DEPARTAMENTO ELEGÍVEL*/
10                  ENTÃO FAÇA;
11                      ENCONTRADO='0'B;
12                      FAÇA K = 1 PARA TAMANHO;
13                          IF(EMPTAB.DEPT(K)=DEPTTAB.DEPT(J))
14                              ENTÃO FAÇA;
15                                  ENCONTRADO='1'B;
16                                  SE(SALÁRIO(K)>=SALÁRIO)|CÓDIGO(K)=MGR)
17                                      ENTÃO SALÁRIO(K)=SALÁRIO(K)+LINC;
18                                      OUTRO SALÁRIO(K)=SALÁRIO(K)+SINC;
19                                  FIM;
20                      FIM;
21                      SE(-ENCONTRADO) ENTÃO CÓDIGO DE ERRO=2;
22                  FIM;
23          FIM;
24      FIM;
25 FIM;
```

FIGURA 5.2 (continuação)

Barra lateral 5.1: Plano de fundo PL/1

Leitores novos no desenvolvimento de software podem não estar familiarizados com PL/1 e pensar que é uma linguagem "morta". É verdade, provavelmente há muito pouco novo desenvolvimento usando PL/1, mas manutenção de sistemas existentes continua, e as construções PL/1 ainda são uma boa maneira de aprender sobre os procedimentos de programação.

PL/1, que significa Programming Language One, foi desenvolvido na década de 1960 pela IBM para fornecer um ambiente de desenvolvimento semelhante ao inglês para suas máquinas de classe mainframe, começando com o IBM System/360. Nessa época da história do computador, muitos programadores estavam migrando para linguagens especializadas, como COBOL, projetada para desenvolvimento de aplicativos de negócios, e Fortran, projetada para aplicações científicas. (Consulte a Barra Lateral 3.1 no Capítulo 3 para obter um pouco de conhecimento sobre esses idiomas.)

Um dos principais objetivos dos projetistas de PL/1 era uma linguagem de desenvolvimento que pudesse competir com sucesso com COBOL e Fortran, proporcionando um ambiente de desenvolvimento que fosse mais fácil de aprender com uma linguagem mais natural. Todos os objetivos iniciais do PL/1 provavelmente nunca foram alcançados, mas esses primeiros designers obviamente fizeram sua lição de casa, porque o PL/1 foi refinado e atualizado ao longo dos anos e ainda está em uso em alguns ambientes hoje.

Em meados da década de 1990, o PL/1 foi estendido a outras plataformas de computador, incluindo OS/2, Linux, UNIX e Windows. O novo suporte ao sistema operacional trouxe extensões de idioma para fornecer mais flexibilidade e funcionalidade.

Independentemente de qual técnica de cobertura lógica você usa, a primeira etapa é listar as decisões condicionais no programa. Os candidatos neste programa são todos os comandos IF e DO . Ao inspecionar o programa, podemos ver que todas as instruções DO são iterações simples, e cada limite de iteração será igual ou maior que o valor inicial (o que significa que cada corpo do loop sempre será executado pelo menos uma vez); e a única maneira de sair de cada loop é através da instrução DO . Assim, as instruções DO neste programa não precisam de atenção especial, pois qualquer caso de teste que faça com que uma instrução DO seja executada acabará fazendo com que ela ramifique em ambas as direções (ou seja, entrar no corpo do loop e pular o corpo do loop). Portanto, as afirmações que devem ser analisadas são:

```
2 SE (TAMANHO<¼O) j (TAMANHO<¼0)
6 SE (VENDAS(I)>¼ VENDAS MÁXIMAS)
9 SE (VENDAS(J)¼ VENDAS MÁXIMAS)
13 SE (EMPTAB.DEPT(K)¼ DEPTTAB.DEPT(J))
16 SE (SALÁRIO(K)>¼ SALÁRIO) j (CÓDIGO(K)¼MGR)
21 SE (-ENCONTRADO) ENTÃO CÓDIGO DE ERRO¼ 2
```

TABELA 5.1 Situações Correspondentes aos Resultados da Decisão

Decisão	Verdadeiro Resultado	Resultado Falso
2	ESIZE ou DSIZE 0	ESIZE e DSIZE>0
6	Sempre ocorrerá pelo menos uma vez.	Encomende o DEPTTAB para que um departamento com vendas mais baixas ocorra depois de um departamento com vendas mais altas.
9	Sempre ocorrerá pelo menos uma vez.	Todos os departamentos não têm as mesmas vendas.
13	Há um funcionário em um departamento elegível.	Há um funcionário que não está em um departamento elegível.
16	Um funcionário elegível é um gerente ou recebe LSALARY ou mais.	Um funcionário elegível não é gerente e ganha menos de SALÁRIO.
21	Todos os departamentos elegíveis não contêm funcionários.	Um departamento elegível contém pelo menos um funcionário.

Dado o pequeno número de decisões, provavelmente deveríamos optar pela cobertura multicondicional, mas examinaremos todos os critérios de cobertura lógica (exceto cobertura de instrução, que sempre é muito limitada para ser útil) para ver seus efeitos.

Para satisfazer o critério de cobertura de decisão, precisamos de casos de teste suficientes para invocar ambos os resultados de cada uma das seis decisões. As situações de entrada necessárias para invocar todos os resultados de decisão estão listadas na Tabela 5.1. Desde dois de os resultados sempre ocorrerão, existem 10 situações que precisam ser forçadas por casos de teste. Observe que para construir a Tabela 5.1, as circunstâncias do resultado da decisão tiveram que ser rastreadas através da lógica do programa para determinar as circunstâncias de entrada correspondentes apropriadas. Por exemplo, a decisão 16 não é invocada por nenhum funcionário que cumpra as condições; o funcionário deve estar em um departamento elegível.

As 10 situações de interesse na Tabela 5.1 podem ser invocadas pelos dois casos de teste mostrados na Figura 5.3. Observe que cada caso de teste inclui uma definição da saída esperada, de acordo com os princípios discutidos no Capítulo 2.

Embora esses dois casos de teste atendam ao critério de cobertura de decisão, deve ser óbvio que pode haver muitos tipos de erros no módulo que não são detectados por esses dois casos de teste. Por exemplo, os casos de teste não exploram as circunstâncias em que o código de erro é 0, um funcionário é um gerente ou a tabela de departamentos está vazia (DSIZE<¼0).

<table>
<tr><th>Caso de teste</th><th>Entrada</th><th>Saída esperada</th></tr>
<tr><td>1</td><td>TAMANHO = 0
Todas as outras entradas são irrelevantes</td><td>CÓDIGO DE ERRO = 1
ESIZE, DSIZE, EMPTAB e DEPTTAB
são inalterados</td></tr>
<tr><td>2</td><td>ESIZE = DSIZE = 3
EMPTAB
<table><tr><td>JONES</td><td>E</td><td>D42</td><td>21.000,00</td></tr><tr><td>SMITH</td><td>E</td><td>D32</td><td>14.000,00</td></tr><tr><td>LORIN</td><td>E</td><td>D42</td><td>10.000,00</td></tr></table>DEPTTAB
<table><tr><td>D42</td><td>10.000,00</td></tr><tr><td>D32</td><td>8.000,00</td></tr><tr><td>D95</td><td>10.000,00</td></tr></table></td><td>CÓDIGO DE ERRO = 2
ESIZE, DSIZE e DEPTTAB permanecem inalterados
EMPTAB
<table><tr><td>JONES</td><td>E</td><td>D42</td><td>21.100,00</td></tr><tr><td>SMITH</td><td>E</td><td>D32</td><td>14.000,00</td></tr><tr><td>LORIN</td><td>E</td><td>D42</td><td>10.200,00</td></tr></table></td></tr>
</table>

FIGURA 5.3 Casos de Teste para Satisfazer o Critério de Cobertura de Decisão.

Um teste mais satisfatório pode ser obtido usando o critério de cobertura de condição. Aqui precisamos de casos de teste suficientes para invocar ambos os resultados de cada condição nas decisões. As condições e situações de entrada necessárias para invocar todos os resultados estão listadas na Tabela 5.2. Como dois dos resultados sempre ocorrerão, existem 14 situações que devem ser forçadas pelos casos de teste. Novamente, essas situações podem ser invocadas por apenas dois casos de teste, conforme mostrado na Figura 5.4.

Os casos de teste na Figura 5.4 foram projetados para ilustrar um problema. Como eles invocam todos os resultados da Tabela 5.2, eles satisfazem o critério de cobertura de condição, mas provavelmente são um conjunto de casos de teste mais pobre do que os da Figura 5.3 em termos de satisfação do critério de cobertura de decisão.
A razão é que eles não executam todas as instruções. Por exemplo, a instrução 18 nunca é executada. Além disso, eles não realizam muito mais do que os casos de teste na Figura 5.3. Eles não causam a situação de saída ERRORCODE¼0. Se a instrução 2 tiver configurado erroneamente ESIZE¼0 e DSIZE¼0, esse erro não será detectado. É claro que um conjunto alternativo de casos de teste pode resolver esses problemas, mas o fato é que os dois casos de teste na Figura 5.4 satisfazem o critério de cobertura de condição.

Usar o critério de cobertura de decisão/condição eliminaria a principal fraqueza nos casos de teste na Figura 5.4. Aqui, forneceríamos casos de teste suficientes para que todos os resultados de todas as condições e decisões fossem invocados pelo menos uma vez. Tornar Jones um gerente e Lorin um não gerente poderia conseguir isso. Isso teria o resultado de gerar ambos os resultados da decisão 16, fazendo com que executemos a instrução 18.

TABELA 5.2 Situações Correspondentes aos Resultados da Condição

Condição de Decisão		Resultado real	Resultado Falso
2	TAMANHO 0	TAMANHO 0	TAMANHO>0
2	TAMANHO 0	TAMANHO 0	DSIZE>0
6	VENDAS(I) VENDAS MÁXIMAS	Sempre ocorrerá pelo menos uma vez.	Encomende o DEPTTAB para que um departamento com vendas mais baixas ocorra depois de um departamento com vendas mais altas.
9	VENDAS(J)¼ VENDAS MÁXIMAS	Sempre ocorrerá pelo menos uma vez.	Todos os departamentos não têm as mesmas vendas.
13	EMPATAB.DEPT (K)¼ DEPTTAB. DEPT(J)	Há um funcionário elegível não está em um departamento	Há um funcionário que em um departamento elegível. departamento.
16	SALÁRIO(K) SALÁRIO	Um funcionário elegível ganha LSALÁRIO ou mais.	Um funcionário elegível ganha menos que LSALARY.
16	CÓDIGO(K)¼MGR	Um funcionário elegível é um gerente.	Um funcionário elegível não é um gerente.
21	-ENCONTRADO	Um departamento elegível não contém funcionários.	Um departamento elegível contém pelo menos um funcionário.

Caso de teste	Entrada	Saída esperada
1	ESIZE = DSIZE = 0 Todas as outras entradas são irrelevantes	CÓDIGO DE ERRO = 1 ESIZE, DSIZE, EMPTAB e DEPTTAB permanecem inalterados
2	ESIZE = DSIZE = 3 EMPATAB JONES E D42 21.000,00 SMITH E D32 14.000,00 LORIN M D42 10.000,00 DEPTTAB D42 10.000,00 D32 8.000,00 D95 10.000,00	CÓDIGO DE ERRO = 2 ESIZE, DSIZE e DEPTTAB permanecem inalterados EMPATAB JONES E D42 21.000,00 SMITH E D32 14.000,00 LORIN M D42 10.100,00

FIGURA 5.4 Casos de Teste para Satisfazer o Critério de Cobertura da Condição.

Um problema com isso, no entanto, é que essencialmente não é melhor do que os casos de teste na Figura 5.3. Se o compilador que está sendo usado parar de avaliar uma expressão ou assim que determinar que um operando é verdadeiro, essa modificação resultaria na expressão CODE(K)¼MGR na instrução 16 nunca tendo um resultado verdadeiro. Portanto, se essa expressão fosse codificada incorretamente, os casos de teste não detectariam o erro.

O último critério a ser explorado é a cobertura multicondicional. Este critério requer casos de teste suficientes para que todas as combinações possíveis de condições em cada decisão sejam invocadas pelo menos uma vez. Isso pode ser feito trabalhando a partir da Tabela 5.2. As decisões 6, 9, 13 e 21 têm duas combinações cada uma; as decisões 2 e 16 têm quatro combinações cada. A metodologia para projetar os casos de teste é selecionar um que cubra o maior número possível de combinações, selecionar outro que cubra o maior número possível de combinações restantes e assim por diante. Um conjunto de casos de teste que satisfazem o critério de cobertura multicondicional é mostrado na Figura 5.5. O conjun

<table>
<tr><th>Teste caso</th><th>Entrada</th><th>Saída esperada</th></tr>
<tr><td>1</td><td>TAMANHO = 0 TAMANHO = 0
Todas as outras entradas são irrelevantes</td><td>CÓDIGO DE ERRO = 1
ESIZE, DSIZE, EMPTAB e
DEPTTAB permanecem inalterados</td></tr>
<tr><td>2</td><td>TAMANHO = 0 TAMANHO > 0
Todas as outras entradas são irrelevantes</td><td>O mesmo que acima</td></tr>
<tr><td>3</td><td>TAMANHO > 0 TAMANHO = 0
Todas as outras entradas são irrelevantes</td><td>O mesmo que acima</td></tr>
<tr><td>4</td><td>TAMANHO = 5 TAMANHO = 4
EMPATAB
<table>
<tr><td>JONES</td><td>M</td><td>D42</td><td>21.000,00</td></tr>
<tr><td>AVISO</td><td>M</td><td>D95</td><td>12.000,00</td></tr>
<tr><td>LORIN</td><td>E</td><td>D42</td><td>10.000,00</td></tr>
<tr><td>TOY</td><td>D95</td><td>E</td><td>16.000,00</td></tr>
<tr><td>SMITH</td><td>D32</td><td>E</td><td>14.000,00</td></tr>
</table>
DEPTTAB
<table>
<tr><td>D42</td><td>10.000,00</td></tr>
<tr><td>D32</td><td>8.000,00</td></tr>
<tr><td>D95</td><td>10.000,00</td></tr>
<tr><td>D44</td><td>10.000,00</td></tr>
</table></td><td>CÓDIGO DE ERRO = 2
ESIZE, DSIZE e DEPTTAB permanecem inalterados
EMPATAB
<table>
<tr><td>JONES</td><td>M</td><td>D42</td><td>21.100,00</td></tr>
<tr><td>AVISO</td><td>M</td><td>D95</td><td>12.100,00</td></tr>
<tr><td>LORIN</td><td>E</td><td>D42</td><td>10.200,00</td></tr>
<tr><td>BRINQUEDO</td><td>E</td><td>D95</td><td>16.100,00</td></tr>
<tr><td>SMITH</td><td>E</td><td>D32</td><td>14.000,00</td></tr>
</table></td></tr>
</table>

FIGURA 5.5 Casos de Teste para Atender o Critério de Cobertura Multicondicional.

abrangente do que os conjuntos anteriores de casos de teste, o que implica que deveríamos ter selecionado esse critério no início.

É importante perceber que o módulo BONUS pode ter um número tão grande de erros que mesmo os testes que satisfaçam o critério de cobertura multicondicional não detectariam todos eles. Por exemplo, nenhum caso de teste gera a situação em que ERRORCODE é retornado com o valor 0; assim, se a instrução 1 estivesse faltando, o erro não seria detectado. Se LSALARY fosse inicializado erroneamente em $ 150.000,01, o erro passaria despercebido. Se a declaração 16 indicasse SALÁRIO(K)>SALÁRIO em vez de SALÁRIO(K) >¼SALÁRIO, este erro não seria encontrado. Além disso, se uma variedade de erros off-by-one (como não manipular a última entrada em DEPTTAB ou EMPTAB corretamente) seria detectada dependeria em grande parte do acaso.

Dois pontos devem ficar aparentes agora: primeiro, o critério multicondicional é superior aos outros critérios e, segundo, qualquer critério de cobertura lógica não é bom o suficiente para servir como o único meio de derivar testes de módulo. Portanto, o próximo passo é complementar os testes da Figura 5.5 com um conjunto de testes de caixa preta. Para isso, as especificações da interface do BONUS são mostradas a seguir:

BONUS, um módulo PL/1, recebe cinco parâmetros, simbolicamente referidos aqui como EMPTAB, DEPTTAB, ESIZE, DSIZE e ERRORCODE. Os atributos desses parâmetros são:

```
DECLARE 1 EMPTAB(*), /*ENTRADA E SAÍDA*/
        2 NOME DE PERSONAGEM(6),
        2 CARACTERES DE CÓDIGO(1),
        2 PERSONAGEM DE DEPARTAMENTO(3),
        2 DECIMAL FIXO DE SALÁRIO(7,2);
DECLARE 1 DEPTTAB(*), /*INPUT*/
        2 PERSONAGEM DE DEPARTAMENTO(3),
        2 VENDAS DECIMAL FIXO(8,2);
DECLARE (ESIZE, DSIZE) BINÁRIO FIXO; /*ENTRADA*/
DECLARE ERRCODE DECIMAL FIXO(1); /*RESULTADO*/
```

O módulo assume que os argumentos transmitidos possuem esses atributos. ESIZE e DSIZE indicam o número de entradas em EMPTAB e DEPTTAB, respectivamente. Nenhuma suposição deve ser feita sobre a ordem das entradas em EMPTAB e DEPTTAB. A função do módulo é incrementar o salário (EMPTAB.SALARY) dos funcionários do departamento ou departamentos com maior valor de vendas (DEPTTAB.SALES). Se um elegível

o salário atual do funcionário é de $ 150.000 ou mais, ou se o empregado for um gerente (EMPTAB.CODE¼'M'), o incremento é de $1.000; caso contrário, o incremento para o funcionário elegível é de $ 2.000. O módulo assume que o salário incrementado caberá no campo EMPTAB.SALARY. Se ESIZE e DSIZE não forem maiores que 0, ERRCODE será definido como 1 e nenhuma ação adicional será tomada. Em todos os outros casos, a função é completamente executada. No entanto, se for constatado que um departamento de vendas máximas não tem funcionário, o processamento continua, mas ERRCODE terá o valor 2; caso contrário, é definido como 0.

Esta especificação não é adequada para gráficos de causa e efeito (não há um conjunto discernível de condições de entrada cujas combinações devam ser exploradas); assim, será utilizada a análise de valor de contorno. Os limites de entrada identificados são os seguintes:

1. EMPTAB tem 1 entrada.
2. EMPTAB tem o número máximo de entradas (65.535).
3. EMPTAB tem 0 entradas.
4. DEPTTAB tem 1 entrada.
5. O DEPTTAB possui 65.535 entradas.
6. DEPTTAB tem 0 entradas.
7. Um departamento de vendas máximas tem 1 funcionário.
8. Um departamento de vendas máximas tem 65.535 funcionários.
9. Um departamento de vendas máximas não tem funcionários.
10. Todos os departamentos do DEPTTAB têm as mesmas vendas.
11. O departamento de vendas máximas é a primeira entrada no DEPTTAB.
12. O departamento de vendas máximas é a última entrada no DEPTTAB.
13. Um funcionário elegível é a primeira entrada no EMPTAB.
14. Um funcionário elegível é a última entrada no EMPTAB.
15. Um funcionário elegível é um gerente.
16. Um funcionário elegível não é um gerente.
17. Um funcionário elegível que não seja gerente tem um salário de $ 149.999,99.
18. Um funcionário elegível que não seja gerente tem um salário de $ 150.000.
19. Um funcionário elegível que não seja gerente tem um salário de $ 150.000,01.

Os limites de saída são os seguintes:

20. ERRCODE¼0
21. ERRCODE¼1
22. ERRCODE¼2
23. O aumento salarial de um funcionário elegível é de $ 299.999,99.

Uma outra condição de teste baseada na técnica de adivinhação de erros é a seguinte:

24. Um departamento de vendas máximas sem funcionários é seguido no DEPTTAB com outro departamento de vendas máximo com funcionários.

Isso é usado para determinar se o módulo termina erroneamente processamento da entrada quando encontra uma situação ERRCODE¼2 .

Revendo essas 24 condições, os números 2, 5 e 8 parecem casos de teste impraticáveis. Como eles também representam condições que nunca ocorrerão (geralmente uma suposição perigosa a ser feita ao testar, mas aparentemente segura aqui), nós os excluímos. O próximo passo é comparar as 21 condições restantes com o conjunto atual de casos de teste (Figura 5.5) para determinar quais condições de contorno ainda não foram cobertas. Fazendo isso, vemos que as condições 1, 4, 7, 10, 14, 17, 18, 19, 20, 23 e 24 requerem casos de teste além daqueles da Figura 5.5.

O próximo passo é projetar casos de teste adicionais para cobrir as 11 condições de contorno. Uma abordagem é mesclar essas condições nos casos de teste existentes (ou seja, modificando o caso de teste 4 na Figura 5.5), mas isso não é recomendado porque isso pode inadvertidamente prejudicar a cobertura completa de multicondições dos casos de teste existentes. Portanto, a abordagem mais segura é adicionar casos de teste aos da Figura 5.5. Ao fazer isso, o objetivo é projetar o menor número de casos de teste necessário para cobrir as condições de contorno. Os três casos de teste na Figura 5.6 realizam isso. O caso de teste 5 cobre as condições 7, 10, 14, 17, 18, 19 e 20; o caso de teste 6 cobre as condições 1, 4 e 23; e o caso de teste 7 cobre a condição 24.

A premissa aqui é que a cobertura lógica, ou caixa branca, casos de teste em A Figura 5.6 forma um teste de módulo razoável para o procedimento BONUS.

Testes Incrementais

Ao realizar o processo de teste de módulo, há duas considerações principais: o projeto de um conjunto eficaz de casos de teste, que foi discutido na seção anterior, e a maneira pela qual os módulos são combinados para formar um programa de trabalho. A segunda consideração é importante porque tem as seguintes implicações:

A forma na qual os casos de teste do módulo são escritos

Os tipos de ferramentas de teste que podem ser usadas

Caso de teste	Entrada	Saída esperada
5	TAMANHO = 3 TAMANHO = 2 EMPATAB ALIADO E D36 14.999,99 MELHOR E D33 15.000,00 CELTO E D33 15.000,01 DEPTTAB D33 55.400,01 D36 55.400,01	CÓDIGO DE ERRO = 0 ESIZE, DSIZE e DEPTTAB são inalterado EMPATAB ALIADO E D36 15.199,99 MELHOR E D33 15.100,00 CELTO E D33 15.100,01
6	TAMANHO = 1 TAMANHO = 1 EMPATAB CHEFE M D99 99.899,99 DEPTTAB D99 99.000,00	CÓDIGO DE ERRO = 0 ESIZE, DSIZE e DEPTTAB são inalterado EMPATAB CHEFE M D99 99.999,99
7	TAMANHO = 2 TAMANHO = 2 EMPATAB DOLE E D67 10.000,00 FORD E D22 33.333,33 DEPTTAB D66 20.000,00 D67 20.000,00	CÓDIGO DE ERRO = 2 ESIZE, DSIZE e DEPTTAB são inalterado EMPATAB DOLE E D67 10.000,00 FORD E D22 33.333,33

FIGURA 5.6 Casos de Teste de Análise de Valor de Limite Complementar para BÔNUS.

A ordem em que os módulos são codificados e testados

O custo de gerar casos de teste

O custo de depuração (localização e reparação de erros detectados)

Em suma, então, é uma consideração de importância substancial. Nisso seção, discutimos duas abordagens, testes incrementais e não incrementais; a seguir, exploramos duas abordagens incrementais, de cima para baixo e desenvolvimento ou teste de baixo para cima.

A questão aqui ponderada é a seguinte: Você deve testar um programa testando cada módulo independentemente e, em seguida, combinando os módulos para

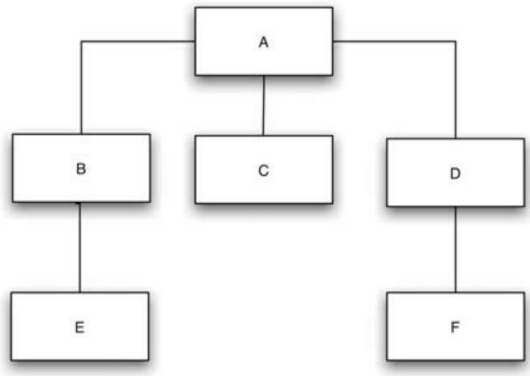

FIGURA 5.7 Exemplo de Programa de Seis Módulos.

forma o programa, ou você deve combinar o próximo módulo a ser testado com o conjunto de módulos testados anteriormente antes de ser testado? A primeira abordagem é chamada não-incremental, ou “big-bang”, teste ou integração; a segunda abordagem é conhecida como teste incremental ou integração.

O programa da Figura 5.7 é usado como exemplo. Os retângulos representam os seis módulos (sub-rotinas ou procedimentos) do programa. As linhas conectando os módulos representam a hierarquia de controle do programa; ou seja, o módulo A chama os módulos B, C e D; o módulo B chama o módulo E; e assim por diante. O teste não incremental, a abordagem tradicional, é realizado da seguinte maneira. Primeiro, um teste de módulo é realizado em cada um dos seis módulos, testando cada módulo como uma entidade independente. Os módulos podem ser testados ao mesmo tempo ou em sucessão, dependendo do ambiente (por exemplo, instalações de computação interativa versus processamento em lote) e do número de pessoas envolvidas. Finalmente, os módulos são combinados ou integrados (por exemplo, "link editado") para formar o programa.

O teste de cada módulo requer um módulo de driver especial e um ou mais módulos stub. Por exemplo, para testar o módulo B, os casos de teste são primeiro projetados e, em seguida, alimentados ao módulo B passando-lhe argumentos de entrada de um módulo driver, um pequeno módulo que deve ser codificado para "conduzir" ou transmitir casos de teste através do módulo em teste. (Alternativamente, uma ferramenta de teste pode ser usada.) O módulo driver também deve exibir, para o testador, os resultados produzidos por B. Além disso, como o módulo B chama o módulo E, algo deve estar presente para receber o controle quando B chamar E. Um módulo stub, um módulo especial com o nome "E " que deve ser codificado para simular a função do módulo E, faz isso.

Quando o teste do módulo de todos os seis módulos estiver concluído, o módulos são combinados para formar o programa.

A abordagem alternativa é o teste incremental. Em vez de testar cada módulo isoladamente, o próximo módulo a ser testado é combinado primeiro com o conjunto de módulos que já foram testados.

É prematuro fornecer um procedimento para testar incrementalmente o programa na Figura 5.7, porque há um grande número de abordagens incrementais possíveis. Uma questão chave é se devemos começar no topo ou no fundo do programa. No entanto, já que discutiremos essa questão na próxima seção, vamos supor, por enquanto, que estamos começando de baixo.

O primeiro passo é testar os módulos E, C e F, em paralelo (por três pessoas) ou em série. Observe que devemos preparar um driver para cada módulo, mas não um stub. O próximo passo é testar B e D; mas em vez de testá-los isoladamente, eles são combinados com os módulos E e F, respectivamente. Em outras palavras, para testar o módulo B, um driver é escrito, incorporando os casos de teste, e o par BE é testado. O processo incremental, adicionando o próximo módulo ao conjunto ou subconjunto de módulos testados anteriormente, continua até que o último módulo (módulo A neste caso) seja testado. Observe que esse procedimento poderia ter progredido alternativamente de cima para baixo.

Várias observações devem ser evidentes neste momento:

1. Testes não incrementais requerem mais trabalho. Para o programa da Figura 5.7, cinco drivers e cinco stubs devem ser preparados (assumindo que não precisamos de um módulo de driver para o módulo superior). O teste incremental de baixo para cima exigiria cinco pilotos, mas sem canhotos. Um teste incremental de cima para baixo exigiria cinco canhotos, mas nenhum motorista. Menos trabalho é necessário porque os módulos testados anteriormente são usados em vez dos módulos de driver (se você começar de cima) ou módulos stub (se você começar de baixo) necessários na abordagem não incremental.
2. Erros de programação relacionados a interfaces incompatíveis ou suposições incorretas entre os módulos serão detectados mais cedo quando o teste incremental for usado. A razão é que as combinações de módulos são testadas juntas em um momento inicial. No entanto, quando testes não incrementais são usados, os módulos não “vêem uns aos outros” até o final do processo.

3. Como resultado, a depuração deve ser mais fácil se o teste incremental for usado. Se assumirmos que existem erros relacionados a interfaces e suposições intermódulos (uma boa suposição, por experiência), então, se

teste não incremental foi usado, os erros não aparecerão até que todo o programa tenha sido combinado. Neste momento, podemos ter dificuldade em identificar o erro, pois ele pode estar em qualquer lugar dentro do programa. Por outro lado, se o teste incremental for usado, um erro desse tipo deve ser mais fácil de identificar, porque é provável que o erro esteja associado ao módulo adicionado mais recentemente.

4. Testes incrementais podem resultar em testes mais completos. Se você estiver testando o módulo B, o módulo E ou A (dependendo se você começou de baixo ou de cima) é executado como resultado. Embora E ou A devesse ter sido exaustivamente testado anteriormente, talvez executá-lo como resultado do teste do módulo de B invoque uma nova condição, talvez uma que represente uma deficiência no teste original de E ou A.
   Por outro lado, se o teste não incremental for usado, o teste de B afetará apenas o módulo B. Em outras palavras, o teste incremental substitui os módulos testados anteriormente pelos stubs ou drivers necessários no teste não incremental. Como resultado, os módulos reais recebem mais exposição pela conclusão do último teste do módulo.
5. A abordagem não incremental parece usar menos tempo de máquina. Se o módulo A da Figura 5.7 estiver sendo testado usando a abordagem de baixo para cima, os módulos B, C, D, E e F provavelmente serão executados durante a execução de A. Em um teste não incremental de A, apenas stubs para B, C e E são executados. O mesmo vale para um teste incremental de cima para baixo. Se o módulo F está sendo testado, os módulos A, B, C, D e E podem ser executados durante o teste de F ; no teste não incremental de F, apenas o driver para F, mais o próprio F , exe cute. Assim, o número de instruções de máquina executadas durante uma execução de teste usando a abordagem incremental é aparentemente maior do que para a abordagem não incremental. Contrabalançando isso está o fato de que o teste não incremental requer mais drivers e stubs do que o teste incremental; tempo de máquina é necessário para desenvolver os drivers e stubs.
6. No início da fase de teste do módulo, há mais oportunidades para atividades paralelas quando o teste não incremental é usado (ou seja, todos os módulos podem ser testados simultaneamente). Isso pode ser significativo em um projeto grande (muitos módulos e pessoas), já que o número de funcionários de um projeto geralmente atinge seu pico no início da fase de teste do módulo.

Em resumo, as observações 1 a 4 são vantagens do teste incremental, enquanto as observações 5 e 6 são desvantagens. Dadas as tendências atuais

na indústria de computação (os custos de hardware têm diminuído e parecem
destinados a continuar a fazê-lo, enquanto a capacidade de hardware aumenta, e os custos de
mão de obra e as consequências de erros de software estão aumentando), e
dado o fato de que quanto mais cedo um erro for encontrado, menor será o custo de repará-lo,
você pode ver que as observações de 1 a 4 estão crescendo em importância, enquanto a
observação 5 está se tornando menos importante. Observação 6
parece ser uma desvantagem fraca, se houver. Isso leva à conclusão
que o teste incremental é superior.

Teste de cima para baixo versus teste de baixo para cima

Dada a conclusão da seção anterior - que o teste incremental é
superior aos testes não incrementais - a seguir exploramos dois testes incrementais
estratégias: testes de cima para baixo e de baixo para cima. Antes de entrar neles, no entanto,
devemos esclarecer vários equívocos. Primeiro, os termos de cima para baixo
testes, desenvolvimento de cima para baixo e design de cima para baixo geralmente são usados
como sinônimos. Teste de cima para baixo e desenvolvimento de cima para baixo são sinônimos (eles
representam uma estratégia de ordenar a codificação e teste de módulos), mas
design de cima para baixo é algo bem diferente e independente. Um programa
que foi projetado de cima para baixo pode ser testado incrementalmente em
uma moda de cima para baixo ou de baixo para cima.

Em segundo lugar, os testes de baixo para cima (ou desenvolvimento de baixo para cima) são
muitas vezes equivocadamente equiparados a testes não incrementais. A razão é que de baixo para cima
o teste começa de maneira idêntica a um teste não incremental (ou seja,
quando os módulos inferiores, ou terminais, são testados), mas como vimos no
seção anterior, o teste de baixo para cima é uma estratégia incremental. Finalmente,
como ambas as estratégias são incrementais, não repetiremos aqui as vantagens
de testes incrementais; discutiremos apenas as diferenças entre os testes de cima para baixo e de
baixo para cima.

Teste de cima para baixo

A estratégia de cima para baixo começa com o módulo superior, ou inicial, no programa. Depois
disso, não há um único procedimento "certo" para selecionar o próximo
módulo a ser testado incrementalmente; a única regra é que para ser elegível para ser
o próximo módulo, pelo menos um dos módulos subordinados (chamando) do módulo
deve ter sido testado anteriormente.

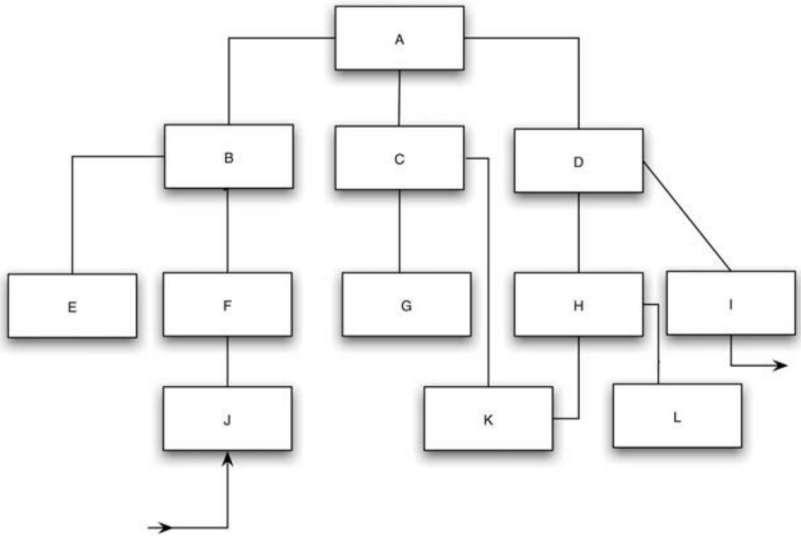

FIGURA 5.8 Exemplo de Programa de 12 Módulos.

A Figura 5.8 é usada para ilustrar essa estratégia. A a L são os 12 módulos do programa. Suponha que o módulo J contenha a leitura de E/S do programa operações e o módulo I contém as operações de gravação.

O primeiro passo é testar o módulo A. Para isso, os módulos stub que representam B, C e D devem ser escritos. Infelizmente, a produção de stub módulos é muitas vezes incompreendido; como evidência, muitas vezes você pode ver declarações como "um módulo stub precisa apenas escrever uma mensagem dizendo 'chegamos até aqui' "; e, "em muitos casos, o módulo fictício (stub) simplesmente sai - sem fazer qualquer trabalho." Na maioria das situações, essas declarações são falsas. Desde o módulo A chama o módulo B, A espera que B realize algum trabalho; este trabalho mais provavelmente algum resultado (argumentos de saída) foi retornado para A. Se o stub simplesmente retorna o controle ou escreve uma mensagem de erro sem retornar um resultado, o módulo A falhará, não por causa de um erro em A, mas por causa de uma falha do stub para simular o módulo correspondente. Além disso, devolver um A saída "com fio" de um módulo stub geralmente é insuficiente. Por exemplo, considere a tarefa de escrever um stub representando uma rotina de raiz quadrada, um banco de dados rotina de pesquisa de tabela, uma rotina "obter registro de arquivo mestre correspondente" ou semelhante. Se o stub retornar uma saída fixa com fio, mas não tiver o valor específico esperado pelo módulo chamador durante essa chamada, o módulo chamador poderá falhar ou produzir um resultado confuso. Assim, a produção de stubs não é uma tarefa trivial.

Outra consideração é a forma como os casos de teste são apresentados ao programa, uma consideração importante que nem é mencionada na maioria das discussões sobre testes de cima para baixo. Em nosso exemplo, a pergunta é: Como você alimenta os casos de teste para o módulo A? O módulo superior em programas típicos não recebe argumentos de entrada nem executa operações de entrada/saída, portanto a resposta não é imediatamente óbvia. A resposta é que os dados de teste são alimentados ao módulo (módulo A nesta situação) de um ou mais de seus stubs.
Para ilustrar, suponha que as funções de B, C e D sejam as seguintes:

B — Obter resumo do arquivo da transação.
C — Determina se o status semanal atende à cota.
D — Produzir relatório resumido semanal.

Um caso de teste para A, então, é um resumo de transação retornado do stub B.
O Stub D pode conter instruções para gravar seus dados de entrada em uma impressora, permitindo que os resultados de cada teste sejam examinados.

Neste programa, existe outro problema. Presumivelmente, o módulo A chama o módulo B apenas uma vez; portanto, o problema é como alimentar mais de um caso de teste para A. Uma solução é desenvolver várias versões do stub B, cada uma com um conjunto diferente de dados de teste conectados a serem retornados a A. Para executar os casos de teste, o programa é executado várias vezes, cada vez com uma versão diferente do stub B. Outra alternativa é colocar os dados de teste em arquivos externos e fazer com que o stub B leia os dados de teste e os retorne para A. Em ambos os casos, tendo em mente o anterior discussão, você deve ver que o desenvolvimento de módulos stub é mais difícil do que muitas vezes parece ser. Além disso, muitas vezes é necessário, devido às características do programa, representar um caso de teste em vários stubs abaixo do módulo em teste (ou seja, onde o módulo recebe dados para serem executados chamando vários módulos).

Depois que A foi testado, um módulo real substitui um dos stubs e os stubs exigidos por esse módulo são adicionados. Por exemplo, a Figura 5.9 pode representar a próxima versão do programa.

Depois de testar o módulo superior, várias sequências são possíveis. Por exemplo, se estivermos realizando todas as sequências de teste, quatro exemplos das muitas sequências possíveis de módulos são:

1. ABCDEFGHIJKL
2. ABEFJCGKDHLI
3. ADHIKLCGBFJE
4. ABFJDIECGKHL

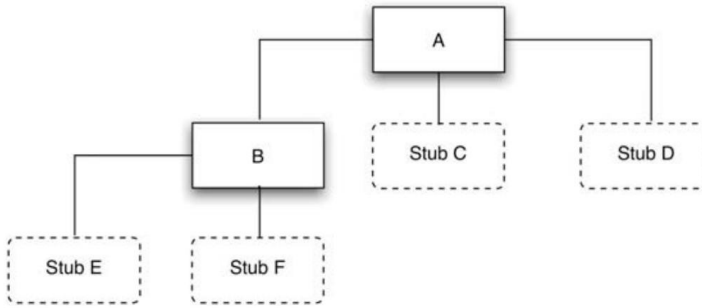

FIGURA 5.9 Segundo Passo no Teste Top-Down.

Se ocorrerem testes paralelos, outras alternativas são possíveis. Por exemplo, após o módulo A ter sido testado, um programador pode pegar o módulo A e teste a combinação AB; outro programador poderia testar AC; e um terceiro poderia testar AD. Em geral, não há melhor sequência, mas aqui estão duas orientações a considerar:

1. Se houver seções críticas do programa (talvez módulo G), projete a sequência de forma que essas seções sejam adicionadas o mais cedo possível. Uma "seção crítica" pode ser um módulo complexo, um módulo com um novo algoritmo, ou um módulo suspeito de ser propenso a erros.
2. Projete a sequência de forma que os módulos de E/S sejam adicionados o quanto antes possível.

A motivação para o primeiro deve ser óbvia, mas a motivação para a segunda merece uma discussão mais aprofundada. Lembre-se de que um problema com stubs é que alguns deles devem conter os casos de teste, e outros devem escrever seus entrada para uma impressora ou monitor. No entanto, assim que o módulo aceitar o entrada do programa é adicionada, a representação dos casos de teste é consideravelmente simplificado; sua forma é idêntica à entrada aceita pelo programa final (por exemplo, de um arquivo de transação ou de um terminal). Da mesma forma, uma vez adicionado o módulo que executa a função de saída do programa, a colocação de código em módulos stub para escrever resultados de casos de teste pode não ser mais necessário. Assim, se os módulos J e I são os módulos de E/S, e se o módulo G desempenha alguma função crítica, a sequência incremental pode ser

ABFJDICGEKHL

e a forma do programa após o sexto incremento seria a mostrada na Figura 5.10.

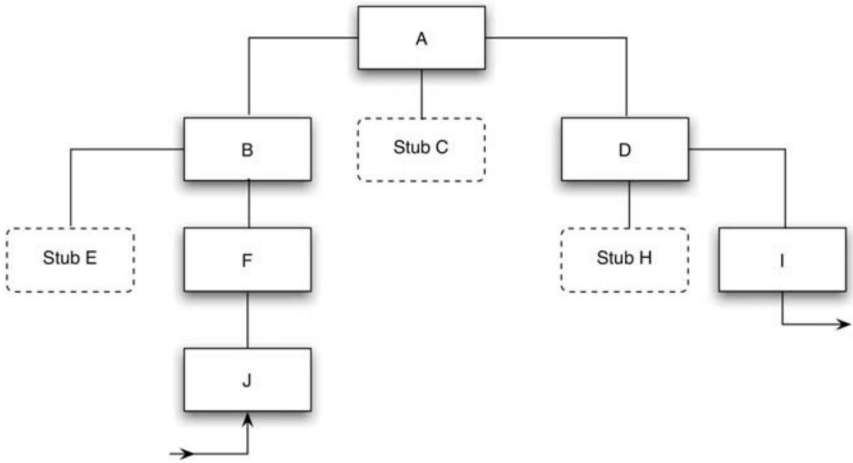

FIGURA 5.10 Estado Intermediário no Teste Top-Down.

Uma vez atingido o estado intermediário da Figura 5.10, a representação dos casos de teste e a inspeção dos resultados são simplificadas. Ele tem uma outra vantagem, pois você tem uma versão funcional do programa, ou seja, uma versão que executa operações reais de entrada e saída. No entanto, stubs ainda estão simulando alguns dos "interiores". Esta versão esquelética inicial:

Permite encontrar erros e problemas de fator humano.
Possibilita a demonstração do programa ao eventual usuário.
Serve como evidência de que o design geral do programa é sólido.
Serve como um impulsionador de moral.

Esses pontos representam a grande vantagem da estratégia top-down.

Por outro lado, a abordagem de cima para baixo tem algumas deficiências sérias. Suponha que nosso estado atual de teste seja o da Figura 5.10 e que nosso próximo passo seja substituir o stub H pelo módulo H. O que devemos fazer neste ponto (ou antes) é usar os métodos descritos anteriormente neste capítulo para projetar um conjunto de casos de teste para H. Observe, no entanto, que os casos de teste estão na forma de entradas de programa reais para o módulo J. Isso apresenta vários problemas. Primeiro, por causa dos módulos intermediários entre J e H (F, B, A e D), podemos achar impossível representar certos casos de teste para o módulo J que testa todas as situações predefinidas em H. Por exemplo, se H é o BONUS da Figura 5.2, pode ser impossível, devido à natureza do módulo D interveniente, criar alguns dos sete casos de teste das Figuras 5.5 e 5.6.

Em segundo lugar, devido à "distância" entre H e o ponto em que os dados de teste entram no programa, mesmo que fosse possível testar todas as situações, determinar quais dados alimentar J para testar essas situações em H é muitas vezes uma tarefa mental difícil.

Terceiro, como a saída exibida de um teste pode vir de um módulo que está a uma grande distância do módulo que está sendo testado, correlacionar a saída exibida com o que aconteceu no módulo pode ser difícil ou impossível. Considere adicionar o módulo E à Figura 5.10. Os resultados de cada caso de teste são determinados examinando a saída escrita pelo módulo I, mas por causa dos módulos intermediários, pode ser difícil deduzir a saída real de E (ou seja, os dados retornados para B).

A estratégia de cima para baixo, dependendo de como é abordada, pode ter mais dois problemas. As pessoas ocasionalmente sentem que a estratégia pode ser sobreposta à fase de projeto do programa. Por exemplo, se você estiver projetando o programa da Figura 5.8, você pode acreditar que, depois que os dois primeiros níveis forem projetados, os módulos A a D podem ser codificados e testados enquanto o projeto dos níveis inferiores progride. Como enfatizamos em outro lugar, essa geralmente é uma decisão imprudente. O design do programa é um processo iterativo, o que significa que, quando estamos projetando os níveis inferiores da estrutura de um programa, podemos descobrir mudanças ou melhorias desejáveis nos níveis superiores. Se os níveis superiores já foram codificados e testados, as melhorias desejáveis provavelmente serão descartadas, uma decisão imprudente a longo prazo.

Um problema final que geralmente surge na prática é não testar completamente um módulo antes de prosseguir para outro módulo. Isso ocorre por dois motivos: pela dificuldade de embutir dados de teste em módulos stub, e porque os níveis superiores de um programa geralmente fornecem recursos para os níveis inferiores. Na Figura 5.8, vimos que testar o módulo A pode exigir várias versões do stub para o módulo B. Na prática, há uma tendência de dizer: "Como isso representa muito trabalho, não executarei todos os casos de teste de A agora. Vou esperar até colocar o módulo J no programa, quando a representação dos casos de teste será mais fácil, e lembrar neste ponto de terminar o teste do módulo A." Claro, o problema aqui é que podemos esquecer de teste o restante do módulo A neste momento posterior. Além disso, como os níveis superiores geralmente fornecem recursos para uso dos níveis inferiores (por exemplo, abertura de arquivos), às vezes é difícil determinar se os recursos foram fornecidos corretamente (por exemplo, se um arquivo foi aberto com os atributos apropriados) até que o módulos inferiores que os utilizam são testados.

Teste de baixo para cima

O próximo passo é examinar a estratégia de teste incremental de baixo para cima. Na maioria das vezes, o teste de baixo para cima é o oposto do teste de cima para baixo; assim, as vantagens do teste de cima para baixo tornam-se as desvantagens do teste de baixo para cima, e as desvantagens do teste de cima para baixo tornam-se as vantagens do teste de baixo para cima. Por causa disso, a discussão sobre testes de baixo para cima é mais curta.

A estratégia bottom-up começa com os módulos terminais no programa (os módulos que não chamam outros módulos). Após esses módulos terem sido testados, novamente não há melhor procedimento para selecionar o próximo módulo a ser testado incrementalmente; a única regra é que para ser elegível para o próximo módulo, todos os módulos subordinados do módulo (os módulos que ele chama) devem ter sido testados previamente.

Voltando à Figura 5.8, o primeiro passo é testar alguns ou todos os módulos E, J, G, K, L e I, em série ou em paralelo. Para fazer isso, cada módulo precisa de um módulo de driver especial: um módulo que contém entradas de teste conectadas, chama o módulo que está sendo testado e exibe as saídas (ou compara as saídas reais com as saídas esperadas). Ao contrário da situação com stubs, várias versões de um driver não são necessárias, pois o módulo do driver pode chamar iterativamente o módulo que está sendo testado. Na maioria dos casos, os módulos de driver são mais fáceis de produzir do que os módulos stub.

Como foi o caso anterior, um fator que influencia a sequência de testes é a natureza crítica dos módulos. Se decidirmos que os módulos D e F são os mais crítico, um estado intermediário do teste incremental de baixo para cima pode ser o da Figura 5.11. Os próximos passos podem ser testar E e depois testar B, combinando B com os módulos E, F e J testados anteriormente.

Uma desvantagem da estratégia de baixo para cima é que não existe o conceito de um programa esquelético inicial. De fato, o programa de trabalho não existe até que o último módulo (módulo A) seja adicionado, e este programa de trabalho é o programa completo. Embora as funções de E/S possam ser testadas antes que todo o programa seja integrado (os módulos de E/S estão sendo usados na Figura 5.11), as vantagens do programa esquelético inicial não estão presentes.

Os problemas associados à impossibilidade, ou dificuldade, de criar todas as situações de teste na abordagem top-down não existem aqui. Se você pensar em um módulo de driver como uma sonda de teste, a sonda está sendo colocada diretamente no módulo que está sendo testado; não há módulos intermediários para se preocupar. Examinando outros problemas associados à abordagem de cima para baixo, você

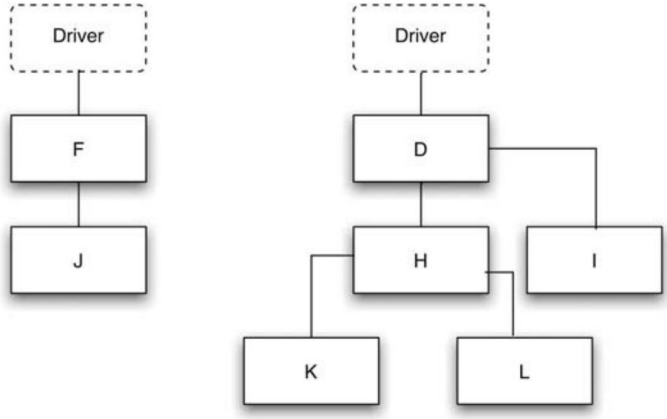

FIGURA 5.11 Estado Intermediário no Teste Bottom-Up.

não pode tomar a decisão imprudente de sobrepor projeto e teste, já que o
o teste de baixo para cima não pode começar até que a parte inferior do programa tenha sido
projetada. Além disso, o problema de não completar o teste de um módulo antes
iniciar outro, devido à dificuldade de codificação de dados de teste em versões
de um stub, não existe ao usar o teste de baixo para cima.

Uma comparação

Seria conveniente se a questão de cima para baixo versus de baixo para cima fosse tão
clara como a questão incremental versus não incremental, mas infelizmente não é. A Tabela 5.3
resume as vantagens e desvantagens relativas das duas abordagens (excluindo o discutido
anteriormente
vantagens compartilhadas por ambos - as do teste incremental). A primeira vantagem de cada
abordagem pode parecer o fator decisivo, mas há
nenhuma evidência mostrando que grandes falhas ocorrem com mais frequência nos níveis
superior ou inferior do programa típico. A maneira mais segura de tomar uma decisão é
pesar os fatores na Tabela 5.3 em relação ao programa específico que está sendo
testado. Na falta de tal programa aqui, as graves consequências da
quarta desvantagem - de teste de cima para baixo e a disponibilidade de ferramentas de teste
que eliminam a necessidade de drivers, mas não de canhotos - parece dar a
estratégia de baixo para cima a borda.

Além disso, pode ser evidente que os testes de cima para baixo e de baixo para cima
não são as únicas estratégias incrementais possíveis.

TABELA 5.3 Comparação de testes de cima para baixo e de baixo para cima

Teste de cima para baixo	
Vantagens	Desvantagens
1. Vantajosa quando grandes falhas ocorrem na parte superior do programa. 2. Uma vez que as funções de E/S são adicionadas, a representação de casos é mais fácil. 3. O programa esquelético precoce permite manifestações e eleva o moral.	1. Os módulos stub devem ser produzidos. 2. Os módulos de stub geralmente são mais complicados do que parecem à primeira vista. 3. Antes que as funções de E/S sejam adicionadas, a representação de casos de teste em stubs pode ser difícil. 4. As condições de teste podem ser impossíveis ou muito difíceis de criar. 5. A observação do resultado do teste é mais difícil. 6. Leva à conclusão de que projeto e teste podem ser sobrepostos. 7. Adia a conclusão dos testes determinados módulos.

Desvantagens do Teste	
Vantagens 1.	Bottom-Up 1. Os
Vantajoso quando grandes falhas ocorrem na parte inferior do programa. 2. As condições de teste são mais fáceis de criar. 3. A observação dos resultados dos testes é mais fácil.	módulos de driver devem ser produzidos. 2. O programa como uma entidade não existe até que o último módulo seja adicionado.

Executando o Teste

A parte restante do teste do módulo é o ato de realmente realizar o teste. Um conjunto de dicas e diretrizes para fazer isso está incluído aqui.

Quando um caso de teste produz uma situação em que os resultados reais do módulo não correspondem aos resultados esperados, há duas explicações possíveis: o módulo contém um erro ou os resultados esperados estão incorretos (o caso de teste está incorreto). Para minimizar essa confusão, o conjunto de casos de teste

devem ser revisados ou inspecionados antes que o teste seja executado (ou seja, os casos de teste devem ser testados).

O uso de ferramentas de teste automatizadas pode minimizar parte do trabalho penoso do processo de teste. Por exemplo, existem ferramentas de teste que eliminam a necessidade de módulos de driver. As ferramentas de análise de fluxo enumeram os caminhos através de um programa, encontram instruções que nunca podem ser executadas (código "inalcançável") e identificam instâncias em que uma variável é usada antes de receber um valor.

Como foi a prática anterior neste capítulo, lembre-se de que uma definição do resultado esperado é uma parte necessária de um caso de teste. Ao executar um teste, lembre-se de procurar por efeitos colaterais (instâncias em que um módulo faz algo que não deveria fazer). Em geral, essas situações são difíceis de detectar, mas algumas delas podem ser encontradas verificando, após a execução do caso de teste, as entradas do módulo que não devem ser alteradas. Por exemplo, o caso de teste 7 na Figura 5.6 afirma que, como parte do resultado esperado, ESIZE, DSIZE e DEPTTAB devem permanecer inalterados. Ao executar este caso de teste, não apenas a saída deve ser examinada para obter o resultado correto, mas ESIZE, DSIZE e DEPTTAB devem ser examinados para determinar se foram alterados erroneamente.

Os problemas psicológicos associados a uma pessoa que tenta testar seus próprios programas também se aplicam aos testes de módulo. Em vez de testar seus próprios módulos, os programadores podem trocá-los; mais especificamente, o programador do módulo chamador é sempre um bom candidato para testar o módulo chamado. Observe que isso se aplica apenas a testes; a depuração de um módulo sempre deve ser realizada pelo programador original.

Evite casos de teste descartáveis; representá-los de tal forma que possam ser reutilizados no futuro. Lembre-se do fenômeno contra-intuitivo na Figura 2.2. Se um número anormalmente alto de erros for encontrado em um subconjunto dos módulos, é provável que esses módulos contenham ainda mais erros, ainda não detectados. Esses módulos devem ser submetidos a mais testes de módulo e, possivelmente, a um passo a passo ou inspeção de código adicional. Por fim, lembre-se de que o objetivo de um teste de módulo não é demonstrar que o módulo funciona corretamente, mas demonstrar a presença de erros no módulo.

Resumo

Neste capítulo, apresentamos alguns dos mecanismos de teste, especialmente no que se refere a programas grandes. Este é um processo de teste de componentes individuais do programa – sub-rotinas, subprogramas, classes e

procedimentos. No teste de módulo, você compara a funcionalidade do software com a especificação que define sua função pretendida. O teste de módulo ou unidade pode ser uma parte importante da caixa de ferramentas de um desenvolvedor para ajudar a obter um aplicativo confiável, especialmente com linguagens orientadas a objetos, como Java e C#. O objetivo no teste de módulo é o mesmo de qualquer outro tipo de teste de software: tentar mostrar como o programa contradiz a especificação. Além da especificação do software, você precisará do código-fonte de cada módulo para efetuar um teste de módulo.

O teste de módulo é basicamente um teste de caixa branca. (Consulte o Capítulo 4 para obter mais informações sobre procedimentos de caixa branca e projeto de casos de teste para teste.) Um projeto de teste de módulo completo incluirá estratégias incrementais, como técnicas de cima para baixo e de baixo para cima.

É útil, ao se preparar para um teste de módulo, revisar os princípios psicológicos e econômicos apresentados no Capítulo 2.

Mais um ponto: o software de teste de módulo é apenas o começo de um procedimento de teste exaustivo. Você precisará passar para testes de ordem superior, que abordamos no Capítulo 6, e testes de usuários, abordados no Capítulo 7.