Resumo Capítulo 5

- Teste de módulo (ou teste de unidade) é um processo de teste de subprogramas, subrotinas, classes ou procedimentos individuais em um programa. Mais especificamente, em vez de testar inicialmente o programa como um todo, testar é focar nos blocos de construção menores do programa.

- Motivações:
    - É uma maneira de gerenciar os elementos combinados de teste, uma vez que a atenção está focada inicialmente em unidades de programa.
    - 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.
    - 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.

- 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).

    - O procedimento de projeto de caso de teste: 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.

Primeira etapa: é listar as decisões condicionais no programa.

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

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.

provavelmente deveríamos optar pela cobertura multicondicional

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.

<table>
<tr><th>Test case</th><th>Input</th><th>Expected output</th></tr>
<tr><td>1</td><td>ESIZE = 0
All other inputs are irrelevant</td><td>ERRCODE = 1
ESIZE, DSIZE, EMPTAB, and DEPTTAB are unchanged</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>ERRCODE = 2
ESIZE, DSIZE, and DEPTTAB are unchanged
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>

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.

Porque a cobertura de decisão seria falha:
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 ERRORCODE=0. Se a instrução 2 tiver configurado erroneamente ESIZE=0 e DSIZE=0, esse erro não será detectado.

<table>
<tr><th>Caso de teste</th><th>Entrada</th><th>Saída esperada</th></tr>
<tr><td>1</td><td>ESIZE = DSIZE = 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>ESIZE = DSIZE = 3
EMPATAB
<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>M</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
EMPATAB
<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>M</td><td>D42</td><td>10.100,00</td></tr></table></td></tr>
</table>

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.

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.

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.

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.

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.

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);

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 necessários para cobrir as condições de contorno.

Teste 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
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)

Você deve testar um programa testando cada módulo independentemente e, em seguida, combinando os módulos para 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 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.

Se você for fazer o teste de módulo individualmente vai requerer um módulo de driver especial e um ou mais módulos stub.

módulo driver é um pequeno módulo que deve ser codificado para "conduzir" ou transmitir casos de teste através do módulo em teste

Um módulo stub, um módulo especial com o nome "E " que deve ser codificado para simular a função do módulo E, faz isso.

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.

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.

Testes não incrementais são mais trabalhosos, pois irá ter que ter mais trabalho adicionando os módulos de drivers e stubs, se for feito de cima para baixo é com os drivers, se for feito de baixo para cima é com os stubs.

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.

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 , então, se teste não incremental foi usado, os erros não aparecerão até que todo o programa tenha sido combinado ele pode estar em qualquer lugar dentro do programa. 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.

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 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. (Desvantagem)

6No 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). (Desvantagem)

Teste de cima para baixo versus teste de baixo para cima

Teste de cima para baixo e desenvolvimento de cima para baixo são sinônimos (eles representam uma estratégia para 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 maneira de cima para baixo ou em uma maneira de baixo para cima.

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 na seção anterior, o teste de baixo para cima é uma estratégia incremental.

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.

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 escrever 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.

é a forma como os casos de teste são apresentados ao programa,

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.

, 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.

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).

Orientações com relação ao teste paralelo:
1Se 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.

2Projete a sequência de forma que os módulos de E/S sejam adicionados o quanto antes possível.

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.

Problemas com top down:
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.

devido à "distância" entre H e o ponto em que os dados de teste entram no programa,

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

As pessoas ocasionalmente sentem que a estratégia pode ser sobreposta à fase de projeto do programa. 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. Primeiro é preguiça de terminar a função começada e só fazer depois de outras já estarem prontas. Segundo, 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

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.

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.

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, as vantagens do programa esquelético inicial não estão presentes.

Uma comparação

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 os teste

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).

O uso de ferramentas de teste automatizadas pode minimizar parte do trabalho penoso do processo de teste.

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).

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.

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.

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.