

Engenharia de Software

Ficha T. Prática nº 9

Fonte: Eng. De Software, Coleção Schaum

Objectivo: Testes de software

1. Introdução

Teste de software é a execução do software com os dados de testes actuais. Algumas vezes é chamado teste dinâmico de software para distinguir de análise estática, como é chamado o teste estático. Análise estática envolve analisar o código fonte para identificar problemas. Embora outras técnicas sejam usadas na validação do software, a execução com dados de teste é essencial.

2. Fundamentos de teste de software

Teste exaustivo é a execução de todos os casos possíveis de teste e raramente podemos fazê-lo. Todo sistema tem muitos casos de teste. Por exemplo, um programa com dois inteiros de entrada numa máquina com palavras de 32 bits teria em 2^{64} possibilidades de casos de teste. Assim, o teste é sempre executado numa percentagem muito pequena dos possíveis casos de teste.

Existem 2 preocupações básicas no teste de software: (1) quais casos de teste usar e (2) quantos casos de teste são necessários. A selecção de casos de teste pode ser baseada nas especificações, na estrutura do código, no fluxo de dados ou na selecção aleatória de casos de teste. Pode ser vista como uma tentativa de espalhar os casos de teste através do espaço de entrada. Algumas áreas no domínio podem ser especialmente sujeitas a erros e podem necessitar de atenção extra. O critério de parada pode ser baseado no critério de cobertura, tal como executar n casos de teste em cada subdomínio, ou baseado no comportamento do critério, tal como testar até que a média de erros seja menor do que um x definido.

Um programa pode ser pensado como um mapeamento dum espaço de domínio para um espaço de resposta ou uma escala. Dada uma entrada, que é um ponto num espaço de domínio, o programa gera uma saída, que é um ponto na escala. Da mesma forma, a especificação dum programa é um mapa dum espaço de domínio para um espaço de resposta.

Uma especificação é essencial para o teste de software. O grau de correcção do software é definido como o mapeamento do programa, sendo o mesmo do mapeamento da especificação. Um bom ditado para lembrar é “um programa sem especificação está sempre correcto”. Um programa sem especificação não pode ser testado pela especificação e o programa faz o que faz e não viola a especificação.

Um caso de teste deverá sempre incluir o resultado esperado. É muito fácil examinar uma saída do computador e pensar que ela está correcta. Se a saída esperada é diferente da saída actual, deverá decidir-se qual está correcta.

3. Critério de cobertura de teste

Um critério de cobertura de teste é uma regra sobre como seleccionar testes e quando parar a testagem. Uma questão básica na pesquisa de teste é como comparar a eficiência de diferentes critérios de cobertura de teste. A abordagem padrão é usar o relacionamento de inclusão.

Inclusão: um critério de teste A inclui o critério de cobertura dum teste B se qualquer conjunto que satisfaça um critério A também satisfaz o critério B. Isso significa que o critério de cobertura de A dalguma forma inclui o critério B. Por exemplo, se um critério de cobertura de teste requer que todo comando seja executado e outro critério requer que todo comando seja executado e testes adicionais sejam feitos, então o segundo critério irá incluir o primeiro.

Pesquisadores têm identificado relacionamentos de inclusão entre a maioria dos critérios convencionais. Entretanto, embora a inclusão seja uma característica usada para comparar critérios de teste, ela não mede a efectividade relativa de dois critérios. Isso acontece porque a maioria dos critérios especifica como um conjunto de casos de teste será escolhido. Escolher um conjunto mínimo de casos de teste para satisfazer um critério não é tão eficiente quanto escolher bons casos de teste até que o critério seja encontrado. Entretanto, um bom conjunto de casos de teste que satisfaça um critério “fraco” pode ser melhor que uma escolha pobre dum conjunto que satisfaça um critério “forte”.

4. Testes

Teste funcional: no teste funcional, a especificação de software é utilizada para identificar subdomínios que deveriam ser testados. Um dos primeiro passos é gerar um caso de teste para cada tipo distinto de saída do programa. Por exemplo, cada mensagem de erro deveria ser gerada. A seguir, todos os casos especiais deveriam ter casos de teste. Situações de excepção deveriam ser testadas, bem como erros comuns e enganoso. O resultado deve ser um conjunto de casos de teste ue irá testar totalmente o programa quando implementado. Esse conjunto de casos de teste deve também ajudar a identificar para o desenvolvedor alguns comportamentos esperados para o software proposto.

No seu livro *The Art of Software Testing*, Glenford Myers coloca o seguinte problema de teste funcional: desenvolver um bom conjunto de casos de teste para um programa que aceita três números a, b, e c, interpretar esses números como lados dum triângulo e saídas do tipo do triângulo. Myers relata que na sua experiência a maioria dos desenvolvedores não responderá com um bom conjunto de testes.

Exemplo: para o exemplo clássico do triângulo, podemos dividir o espaço do domínio em 3 subdomínios, um para cada tipo diferente de triângulo que será considerado; escaleno (nenhum lado igual), isósceles (dois lados iguais) e equilátero (todos os lados iguais). Podemos identificar duas situações de erro: um subdomínio com entradas inválidas e um subdomínoi em que os lados não formam um triângulo. Além disso, já que a ordem dos lados não é especificada, todas as combinações devem ser testadas. Finalmente, cada caso de teste deve especificar o valor de saída.

Subdomínio	Exemplo de caso de teste
Escaleno	
Tamanho crescente	3,4,5 escaleno

Tamanho decrescente 5,4,3 escaleno
 Maior como segundo 4,5,3 escaleno

Isósceles:

A= B e outro lado maior 5,5,8 isósceles
 A = C e outro lado maior 5,8,5 isósceles
 B= C e outro lado maior 8,5,5 isósceles
 A= B e outro lado menor 8,8,5 isósceles
 A = C e outro lado menor 8,5,8 isósceles
 B= C e outro lado maior 5,8,8 isósceles

Equilátero

Todos os lados iguais 5, 5, 5 equilátero

Não é triângulo

Maior primeiro 6,4,2 não é triângulo
 Maior segundo 4,6,2 não é triângulo
 Maior terceiro 1,2,3 não é triângulo

Entradas inválidas:

Uma entrada inválida (-1,2,4) entrada inválida
 Duas entradas inválidas (3,-2,-5) entradas inválidas
 Três entradas inválidas (0,0,0) entradas inválidas

Matrizes de teste : uma maneira de formalizar a identificação de subdomínios é construir uma matriz que poderíamos identificar da especificação e então, sistematicamente, identificar todas as combinações dessas condições como sendo verdadeiras ou falsas.

Exemplo: as condições do problema do triângulo podem ser (1) $a = b$ ou $a = c$ ou $b = c$ (2) $a = b = c$ (3) $a < b + c$ ou $b < a + c$ ou $c < a + b$, e (4) $a, b, c > 0$. Estas 4 condições podem ser colocadas nas linhas da matriz. As colunas da matriz serão cada uma num subdomínio. Para cada subdomínio um V será colocada em cada coluna cuja condição for verdadeira, e um F quando a condição for falso. Todas as combinações válidas de V e F serão usadas. Se existem três condições, existirão $2^3 = 8$ subdomínios (colunas). Linhas adicionais serão usadas para valores de a, b e c e para as saídas esperadas para cada subdomínio.

Condições	1	2	3	4	5	6	7	8
$a = b$ ou $a = c$ ou $b = c$	V	V	V	V	V	F	F	F
$a = b = c$	V	V	F	F	F	F	F	F
$a < b + c$ ou $b < a + c$ ou $c < a + b$	V	F	V	V	F	V	V	F
$a, b, c > 0$	V	F	V	F	F	V	F	F
Exemplo de Caso teste	0,0,0	3,3,3	0,4,0	3,8,3	5,8,5	0,5,6	3,4,8	3,4,5
Saída esperada	inválido	equilátero	inválido	Não é triângulo	isósceles	inválido	Não é triângulo	escaleno

Teste estrutural. Baseado na estrutura do código fonte. O critério de teste estrutural mais simples é o de cobertura de todo comando, também chamado de cobertura C0. O teste de cobertura de todo comando diz que todo comando de código fonte deve ser executado por algum caso de teste.

Exemplo: o seguinte pseudocódigo implementa o problema do triângulo. A matriz mostra quais linhas são executadas por quais casos de teste. Observe que os três primeiros comandos (A,B, e C) são partes do mesmo nó. Pelo quarto caso de teste, todos os comandos já foram executados. Esse conjunto de casos de teste não é o menor conjunto que cobriria todos os comandos. Entretanto, encontrar o menor conjunto de testes nem sempre significa encontrar um bom conjunto de teste

Nó	Linha fonte	3,4,5	3,5,3	0,1,0	4,4,4
A	read a,b,c	*	*	*	*
B	type= "escaleno"	*	*	*	*
C	If(a==b b==c a==c)	*	*	*	*
D	type= "isosceles"		*	*	*
E	If (a==b&&b==c)	*	*	*	*
F	type= "equilátero"				*
G	If(a>=b+c b>=a+c c>=a+b)	*	*	*	*
H	type= "não é triângulo"			*	
I	If(a<=0 b<=0 c<=0)	*	*	*	*
J	type= "entradas inválidas"			*	
K	print type	*	*	*	*

Um critério de teste mais completo é o teste de todos os ramos, geralmente chamado teste de cobertura C1. Nesse critério, objetivo é passar por ambos os caminhos e as decisões.

Teste de todos os caminhos. Mais completo ainda é no entanto, o critério teste de todos os caminhos. Um caminho é uma sequência única de nós programas que são executados por um caso de teste. Na matriz de teste acima há 8 subdomínios. Cada um deles é um caminho. Nesse exemplo, existem 16 combinações diferentes de V e F. Entretanto, oito dessas combinações são inatingíveis. Pode ser extremamente difícil determinar se um caminho é inatingível ou se é apenas difícil encontrar um caso de teste que o execute. A maioria dos programas com ciclos terá um número infinito de caminhos. Em geral, o teste de todos os caminhos não é atingido.

Exemplo: a tabela mostra os 8 caminhos atingidos no pseudo-código do triângulo

Caminho	V/F	Caso de Teste	Saída
ABCEGIK	FFFF	3,4,5	escaleno
ABCEGHIK	FFVF	3,4,8	não é triângulo
ABCEGHIJK	FFVV	0,5,6	entrada inválida
ABCDEGIK	VFFF	5,8,5	Isósceles
ABCDEGHIK	VFVF	3,8,3	Não é triângulo
ABCDEGHIJK	VFVV	0,4,0	Entrada inválida
ABCDEFGIK	VVFF	3,3,3	Equilátero
ABCDEFGHIJK	VVVV	0,0,0	Entrada inválida

Cobertura de condição múltipla: O critério de teste de condição múltipla requer que cada condição de relação primitiva seja avaliada falsa e verdadeira. Além disso, todas as combinações de V/F para as relações primitivas numa condição devem ser testadas. Observe que uma avaliação displicente das expressões irá eliminar algumas

combinações. Por exemplo, num “e” de duas relações primitivas, a segunda não será avaliada se a primeira for falsa.

Exemplo. No pseudo-código do triângulo, existem condições múltiplas em cada comando de decisão. Primitivas não são executadas por causa da avaliação displicente são mostradas com um X.

if (a==b||b==c||a==c)

combinação	Possível caso de teste	Ramo
VXX	3,3,4	ABC-D
FVX	4,3,3	ABC-D
FFV	3,4,3	ABC-D
FFF	3,4,5	ABC-E

if (a==b&& b==c)

combinação	Possível caso de teste	Ramo
VV	3,3,3	E-F
VF	3,3,4	E-G
FX	4,3,3	E-G

if(a>=b+c||b>=a+c||c>=a+b)

combinação	Possível caso de teste	Ramo
VXX	8,4,3	G-H
FVX	4,8,3	G-H
FFV	4,3,8	G-H
FFF	3,3,3	G-I

if(a<=0||b<=0||c<=0)

combinação	Possível caso de teste	Ramo
VXX	0,4,5	I-J
FVX	4,-2,-2	I-J
FFV	5,4,-3	I-J
FFF	3,3,3	I-K

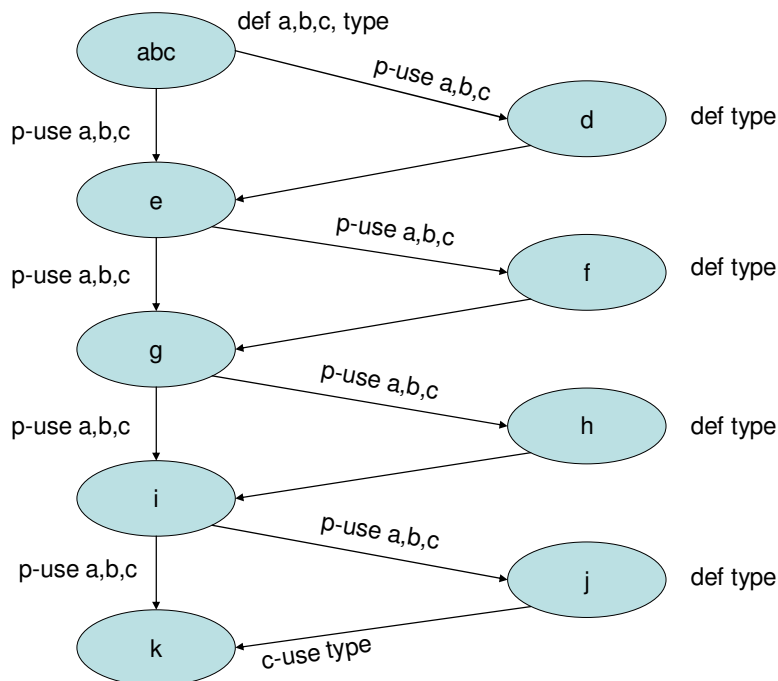
Teste de subdomínio: Teste de subdomínio é a ideia de particionar o domínio da entrada em subdomínios mutuamente exclusivos e requerendo um número igual de casos de teste de cada subdomínio. Essa era basicamente a ideia por trás da matriz de teste. O teste de subdomínio é mais geral, já que não restringe como os subdomínios são seleccionados. Geralmente, se existe uma boa razão para seleccionar os subdomínios, então eles podem ser úteis para o teste. Além disso, os subdomínios de outras abordagens podem ser divididos em subdomínios menores. Trabalhos teóricos têm demonstrado que subdividir subdomínios é efectivo apenas se tende a isolar erros potenciais dentro dos subdomínios individuais.

A cobertura de todo comando e todo ramo não são testes de subdomínios. Não existem subdomínios mutuamente exclusivos relacionados à execução de diferentes comandos ou ramos. A cobertura de todo caminho é uma cobertura de subdomínio, já que os casos de teste dos subdomínios que executam um caminho particular por um programa são mutuamente exclusivos do domínio para qualquer outro caminho.

Exemplo: Para o exemplo do triângulo, devemos iniciar com um subdomínio para cada saída. Posteriormente, isso poderá ser subdividido em novos subdomínios, baseados na maior ou do pior elemento na primeira posição, segunda posição ou terceira posição (quando apropriado)

Subdomínio	Possível caso de teste
Equilátero	3,3,3
Isós-primeiro	8,5,5
Isós-segundo	5,8,5
Isós-terceiro	5,5,8
Escaleno-primeiro	5,4,3
Escaleno-segundo	4,5,3
Escaleno terceiro	3,4,5
Não é triângulo-primeiro	8,3,3
Não é triângulo-segundo	3,8,4
Não é triângulo-terceiro	4,3,8
Entrada inválida-primeiro	0,3,4
Entrada inválida-segundo	3,0,4
Entrada inválida-terceiro	3,4,0

Teste de fluxo de dados: é baseado no fluxo de dados através do programa. Os dados fluem de onde são definidos para onde são usados. A definição dos dados ou *def* ocorre quando o valor é atribuído à variável. Dois tipos diferentes de uso podem ser identificados. O uso computacional, ou *c-use*, ocorre quando a variável aparece do lado direito do comando de atribuição. Um *c-use* ocorre no comando de atribuição. O uso predicado ou *p-use*, ocorre quando a variável é usada na condição do comando de decisão. Uma *p-use* é designada para ambos os ramos do comando de decisão. Um caminho de decisão livre, ou *def-free*, é um caminho da definição duma variável para o uso dessa variável, não incluindo outra definição da variável.



Existem muitos critérios para o teste de fluxo de dados. O critério básico inclui *dcu*, que requer um caminho *def-free* das definições para um *c-use*, *dpu*, que requer um caminho *def-free* da definição para um *p-use*, e *du* que requer um caminho *def-free* de cada definição para os usos possíveis. O critério mais extensivo is *all-du-paths*, que requer todos os caminhos *def-free* de cada definição para cada uso possível.

Exemplo:

dcu- o único c-use é para a variável type, no nodo k

1. De def type no nó abc até o nó k caminho: abc, e g, i, k
2. De def type no nó d até o nó k caminho: d, e, g, i, k
3. De def type no nó f até o nó k caminho: f, g, i, k
4. De def type no nó h até o nó k caminho: h, i, k
5. De def type no nó j até o nó k caminho: j, k

dpu- o único p-use é para as variáveis a,b,c e o único nó def para elas é o nó abc

1. Do nó abc até o arco abc-d
2. Do nó abc até o arco abc-e
3. Do nó abc até o arco e-f
4. Do nó abc até o arco e-g
5. Do nó abc até o arco g-h
6. Do nó abc até o arco g-i
7. Do nó abc até o arco i-j
8. Do nó abc até o arco i-k

du- todos os casos dcu e dpu combinados

all-du-paths-igual que du

Problemas:

1. Se um programa tem dois inteiros como entrada e cada um pode ser um inteiro de 32 bits, quantas entradas possíveis esse programa tem?
2. Se um programa tem 2^{64} possibilidades de entrada e um teste pode ser executado a cada milésimo de segundo, quanto tempo levaria para executar todas as entradas possíveis?
3. Um programa de folha de pagamento irá calcular o pagamento semanal tendo como entrada o número de horas trabalhadas e os valores correntes. Um trabalhador não pode exercer sua actividade mais do que 80 horas semanais e o pagamento máximo é de 50 euros/hora. Construa testes funcionais
4. Um programa calcula a área dum triângulo. As entradas são três conjuntos de coordenadas x,y. Construa testes funcionais
5. Um programa aceita duas horas (em formato 12 horas) e gera a saída do número de minutos. Construa testes funcionais
6. Defina a matriz de teste para o programa da folha de pagamentos
7. Para o problema do cálculo da área do triângulo, construa a matriz de teste
8. O seguinte pseudo-código implementa o problema do cálculo da hora, se a hora é menor do que 24 horas. Selecciona casos de teste até que a cobertura de todos os comandos seja alcançada.

```
read hr1 min1 AmorPm1  
read hr2 min2 AmorPm2  
If (hr1==12)  
    hr1=0  
If (hr2==12)  
    hr2=0  
If (AmorPm1==pm)  
    hr1=hr1+12  
If (AmorPm2==pm)  
    hr2=hr2+12  
if (min2 < min1)  
    min2=min2+1  
    hr2=hr2-1  
if(hr2<hr1)  
    hr2=hr2+24  
elapsed = min2 - min1 + 60 * (hr2 - hr1)  
print elapsed
```

9. Para o seguinte código, identifique todos os caminhos possíveis, testes de caminhos e testes de fluxos de dados

```
cin >> a >> b >> c; // node a
x = 5; y = 7;
if (a > b && b > c) {
    a = a + 1; // node b
    x = x + 6;
}
if (a = 10 || b > 20) {
    b = b + 1; // node c
    x = y + 4;
}
if (a < 10 || c = 20) { // node d
    b = b + 2; // node e
    y = 4;
}
a = a + b + 1; // node f
y = x + y;
}
if (a > 5 || c < \0) { node g
    b = c + 5; // node h
    x = x + 1;
}
cout >> x >> y; // node i
```