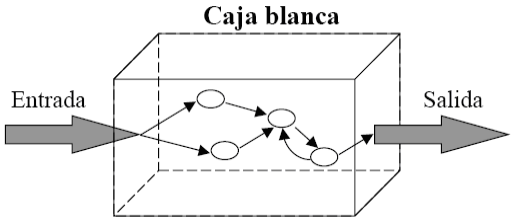
Probas unitarias

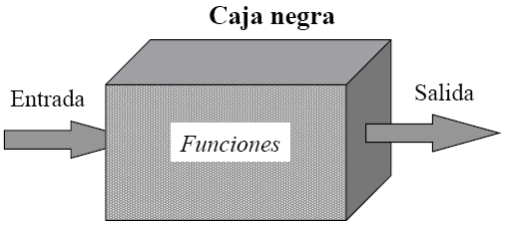
Técnicas de deseño de casos de proba

O deseño de casos de proba está limitado pola imposibilidade de probar exhaustivamente o software. Por exemplo, de querer probar tódolos valores que se poden sumar nun programa que suma dous números enteiros de dúas cifras (do 0 ao 99), deberiamos probar 10000 combinacións distintas (variacións con repetición de 100 elementos tomados de 2 en 2 = 100 elevado a 2) e aínda teriamos que probar todas as posibilidades de erro ao introducir datos (como teclear unha letra no canto dun número).
As técnicas de deseño de casos de proba teñen como obxectivo conseguir unha confianza aceptable en que se detectarán os defectos existentes, xa que a seguridade total só pode obterse da proba exhaustiva, que non é practicable sen consumir unha cantidade excesiva de recursos. Toda a disciplina de probas debe moverse nun equilibrio entre a dispoñibilidade de recursos e a confianza que achegan os casos para descubrir os defectos existentes.
Xa que non se poden facer probas exhaustivas, a idea fundamental para o deseño de casos de proba consiste en elixir algúns deles que, polas súas características, considéranse representativos do resto. A dificultade desta idea é saber elixir os casos que se deben executar xa que unha elección puramente aleatoria non proporciona demasiada confianza en detectar os erros presentes. Existen tres enfoques principais para o deseño de casos non excluíntes entre si e que se poden combinar para conseguir unha detección de defectos máis eficaz:

- Enfoque estrutural ou de caixa branca tamén chamado enfoque de caixa de cristal: Fíxase na implementación do programa para elixir os casos de proba[1].



- Enfoque funcional ou de caixa negra: Consiste en estudar a especificación das funcións, as súas entradas e saídas[2].

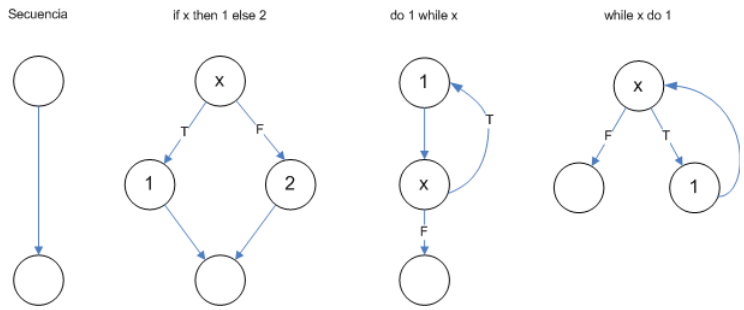


- Enfoque aleatorio: Utiliza modelos, moitas veces estatísticos, que representen as posibles entradas ao programa para crear a partir delas os casos de proba.

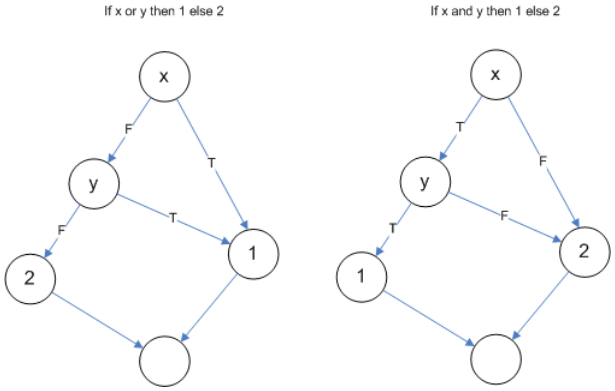
[1] Imaxe extraída de http://osl2.uca.es/wikihaskell/index.php/Pruebas_Unitarias_para_Haskell
[2] Imaxe extraída de http://osl2.uca.es/wikihaskell/index.php/Pruebas_Unitarias_para_Haskell

Probas unitarias estruturais

Para traballar coas técnicas estruturais imos realizar grafos de fluxo dos programas ou funcións a probar. Isto non é estritamente necesario pero debuxalos axuda a comprender o funcionamento das técnicas de proba de caixa branca. Os grafos que se usarán serán grafos fortemente conexos, é dicir, sempre existe un camiño entre calquera par de nodos que se elixan e para subsanar que o nodo primeiro e o último estean directamente conectados, engadirase un arco ficticio que os una.
Grafos básicos de fluxo:



Grafos para condicións múltiples:



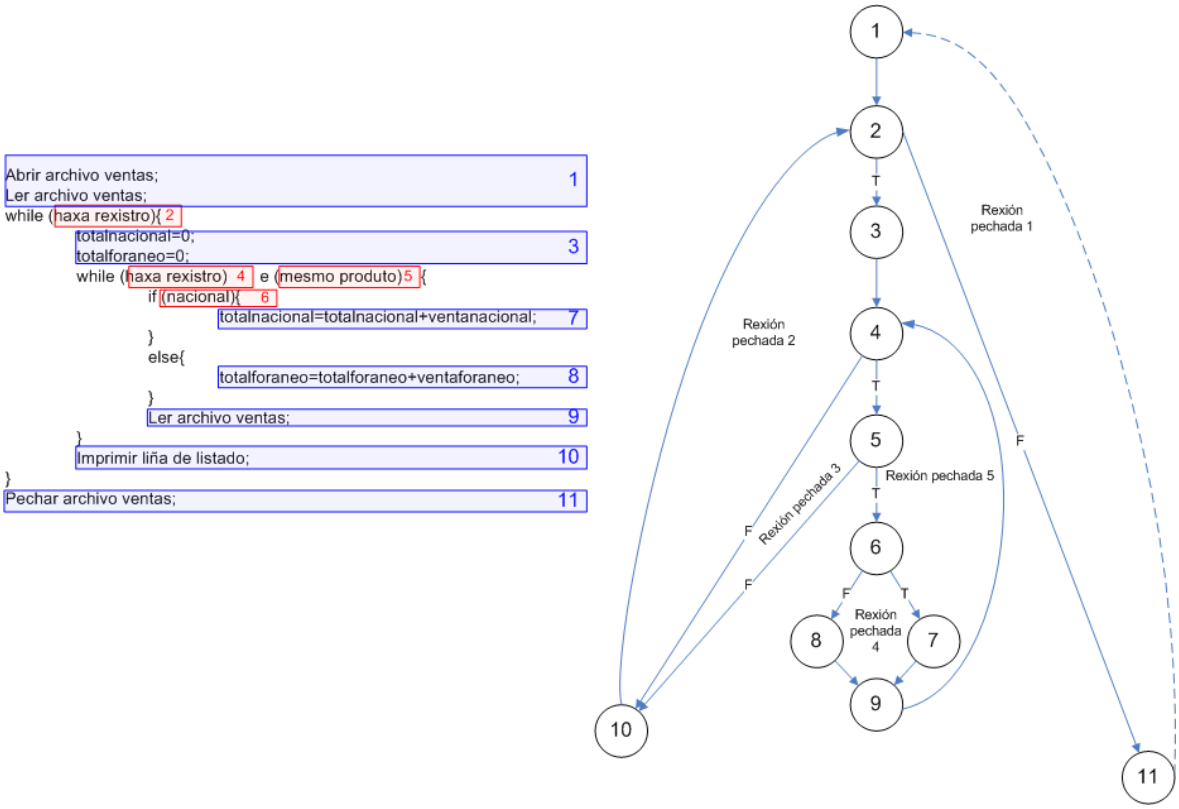
Criterios de cobertura lóxica de Myers

O deseño de casos ten que basearse na elección de camiños importantes que ofrezan unha seguridade aceptable en descubrir defectos. Utilízanse os chamados criterios de cobertura lóxica. Os criterios de cobertura lóxica de Myers móstranse ordenados de menor a maior esixencia, e por tanto, custo económico:

- Cobertura de sentenzas: xerar os casos de proba necesarios para que cada sentenza se execute, polo menos, unha vez.
- Cobertura de decisións: xerar casos para que cada decisión teña, polo menos unha vez, un resultado verdadeiro e, polo menos unha vez, un falso.
- Cobertura de condicións: xerar os casos de proba necesarios para que cada condición de cada decisión adopte o valor verdadeiro polo menos unha vez e o falso polo menos unha vez. Por exemplo, a decisión: if ((a==3) || (b==2)) ten dúas condicións: (a==3) e (b==2).
- Criterio de decisión/condición: consiste en esixir o criterio de cobertura de condicións obrigando a que se cumpra tamén o criterio de decisións.

- Criterio de condición múltiple: No caso de que se considere que a avaliación das condicións de cada decisión non se realiza de forma simultánea, pódese considerar que cada decisión multicondicional descomponse en varias decisións unicondicionales.
- Criterio de cobertura de camiños: Débese executar cada un dos posibles camiños do programa polo menos unha vez. Defínese camiño como unha secuencia de sentenzas encadeadas desde a sentenza inicial do programa até a sentenza final. O número de camiños, mesmo nun programa pequeno, pode ser impracticable para as probas. Para reducir o número de camiños a probar, pode utilizarse a complexidade ciclomática de McCabe.

Exemplo de grafo de fluxo fortemente conexo dun anaco de pseudocódigo:



Complexidade ciclomática de McCabe

A complexidade ciclomática é unha métrica que nos indica o número de camiños independentes que ten un grafo de fluxo. McCabe definiu como un bo criterio de proba o probar un número de camiños independentes igual ao da métrica. Un camiño independente é calquera camiño que introduce, polo menos, un novo conxunto de sentenzas de proceso ou unha condición, respecto dos camiños existentes. En termos do diagrama de fluxo, un camiño independente está constituído polo menos por unha aresta que non fose percorrida nos camiños xa definidos anteriormente. Na identificación dos distintos camiños débese ter en conta que cada novo camiño debe ter o mínimo número de sentenzas novas ou condicións novas respecto dos que xa existen. Desta maneira téntase que o proceso de depuración sexa máis sinxelo.
A complexidade de McCabe V(G) pódese calcular das seguintes tres maneiras, a partir dun grafo de fluxo G fortemente conexo:

- $V(G) = a - n + 2$
  - a: número de arcos ou arestas sen contar o que une o primeiro nodo có último

- n: número de nodos

- V(G) = r
  - r: número de rexións pechadas do grafo contando a que forma o arco que une o primeiro nodo có último

- V(G) = c + 1
  - c: número de nodos de condición

Por exemplo, a complexidade ciclomática de McCabe para o grafo de fluxo do anaco de pseudocódigo anterior é 5, é dicir, é o valor obtido por calquera das tres fórmulas anteriores:

- V(G) = 14 – 11 + 2 = 5
- V(G) = 5
- V(G) = 4 + 1

Unha vez calculada a complexidade ciclomática debemos elixir tantos camiños independentes como o valor da complexidade. Para elixir estes camiños, comezamos definindo un camiño inicial e a continuación imos creando novos camiños variando o mínimo posible o camiño inicial. Para executar un camiño pode ser necesaria a súa concatenación con algún outro.
Para o exemplo anterior teriamos 5 camiños que se representan mediante o número dos nodos do grafo que recorren e os bucles mediante puntos suspensivos despois do nodo de control do bucle:

- 1-2-11
- 1-2-3-4-10-2-...
- 1-2-3-4-5-10-2-...
- 1-2-3-4-5-6-7-9-4-...
- 1-2-3-4-5-6-8-9-4-...

Probas unitarias funcionais

¿Chegaría cunha proba de caixa branca para considerar probado un anaco de código?. A resposta é non, e demóstrase co seguinte código:

```
if ((x+y+z)/3==x)
             then print("X, Y, Z son iguales")
else print("X, Y, Z no son iguales")
```

Hai dous camiños posibles que se recorren cos valores x=5, y=5, z=5 e x=2, y=3, e z=7, que confirman a validez do código, pero cos valores x=4, y=5, e z=3 fallaría o código. Do que se deduce, que se necesitan outro tipo de probas como as probas funcionais para complementar as probas estruturais.
As probas funcionais ou de caixa negra céntranse no estudo da especificación do software, da análise das funcións que debe realizar, das entradas e das saídas. As probas funcionais exhaustivas tamén adoitan ser impracticables polo que existen distintas técnicas de deseño de casos de caixa negra.

Clases de equivalencia

Cada caso de equivalencia debe cubrir o máximo número de entradas. Debe tratarse o dominio de valores de entrada dividido nun número finito de clases de equivalencia que cumpran que a proba dun valor representativo dunha clase permite supor "razoablemente" que o resultado obtido (se

existen defectos ou non) será o mesmo que o obtido probando calquera outro valor da clase. O método para deseñar os casos consiste en identificar as clases de equivalencia e crear os casos de proba correspondentes.
Imos ver algunhas regras que nos axudan a identificar as clases de equivalencia tendo en conta as restricións dos datos que poden entrar ao programa:

- De especificar un rango de valores para os datos de entrada, como por exemplo, "o número estará comprendido entre 1 e 49", crearase unha clase válida: 1<= número <= 49 e dúas clases non válidas: número < 1 e número > 49.
- De especificar un número de valores para os datos de entrada, como por exemplo, "código de 2 a 4 caracteres", crearase unha clase válida: 2<=número de caracteres do código <=4, e dúas clases non válidas: menos de 2 caracteres e máis de 4 caracteres.
- Nunha situación do tipo "debe ser" ou booleana como por exemplo, "o primeiro carácter debe ser unha letra", identificarase unha clase válida: é unha letra e outra non válida: non é unha letra.
- De especificar un conxunto de valores admitidos que o programa trata de forma diferente, crearase unha clase para cada valor válido e outra non válida. Por exemplo, se temos tres tipos de inmobles: pisos, chalés e locais comerciais, faremos unha clase de equivalencia por cada valor e unha non válida que representa calquera outro caso como por exemplo praza de garaxe.
- En calquera caso, de sospeitar que certos elementos dunha clase non se tratan igual que o resto da mesma, deben dividirse en clases de equivalencia menores.

Para crear os casos de proba séguense os pasos seguintes:

- Asignar un valor único a cada clase de equivalencia.
- Escribir casos de proba que cubran todas as clases de equivalencia válidas non incorporadas nos anteriores casos de proba.
- Escribir un caso de proba para cada clase non válida ata que estean cubertas todas as clases non válidas. Isto faise así xa que se introducimos varias clases de equivalencia non válidas xuntas, poida que a detección dun dos erros, faga que xa non se comprobe o resto.

Exemplo: Unha aplicación bancaria na que o operador proporciona un código de área (número de 3 díxitos que non empeza nin por 0 nin por 1), un nome para identificar a operación (6 caracteres) e unha orde que disparará unha serie de funcións bancarias ("cheque", "depósito", "pago factura" ou "retirada de fondos"). Todas as clases numeradas son:

Entrada	Clases válidas	Clases inválidas
Código área	(1) 200 <=código <=999	(2) código < 200 (3) código > 999
Nome para identificar operación	(4) 6 caracteres	(5) menos de 6 caracteres (6) más de 6 caracteres
Orden	(7) “cheque” (8) “depósito” (9) “pago factura” (10) “retirada fondos”	(11) “divisas”

Os casos de proba, supoñendo que a orde de entrada dos datos é: código-nome-orden son os seguintes:

- Casos válidos:

Código	Nome	orde	Clases

200	Nómina	cheque	(1) (4) (7)
200	Nómina	depósito	(1) (4) (8)
200	Nómina	pago factura	(1) (4) (9)
200	Nómina	retirada fondos	(1) (4) (10)

- Casos no válidos:

Código	Nome	orde	Clases
180	Nómina	cheque	(2)
1032	Nómina	cheque	(3)
200	Nómin	cheque	(5)
200	Nóminas	cheque	(6)
200	Nómina	divisas	(11)

Análise de valores límite (AVL)

A experiencia constata que os casos de proba que exploran as condicións límite dun programa producen un mellor resultado para a detección de defectos. Podemos definir as condicións límite para as entradas, como as situacións que se encontran directamente arriba, abaixo e nas marxes das clases de equivalencia e dentro do rango de valores permitidos para o tipo desas entradas. Podemos definir as condicións límite para as saídas, como as situacións que provocan valores límite nas posibles saídas. É recomendable utilizar o enxeño para considerar todos os aspectos e matices, ás veces sutís, para a aplicación do AVL. Algunhas regras para xerar os casos de proba:

- Se para unha entrada especifícase un rango de valores, débense xerar casos válidos para os extremos do rango e casos non válidos para situacións xusto máis aló dos extremos. Por exemplo a clase de equivalencia: "-1.0 <= valor <= 1.0", casos válidos: -1.0 e 1.0, casos non válidos: -1.01 e 1.01, no caso no que se admitan 2 decimais.
- Se para unha entrada especifícase un número de valores, hai que escribir casos para os números máximo, mínimo, un máis do máximo e un menos do mínimo. Por exemplo: "o ficheiro de entrada terá de 1 a 250 rexistros", casos válidos: 1 e 250 rexistros, casos non válidos: 251 e 0 rexistros.
- De especificar rango de valores para a saída, tentarán escribirse casos para tratar os límites na saída. Por exemplo: "o programa pode mostrar de 1 a 4 listaxes", casos válidos: 1 e 4 listaxes, casos non válidos: 0 e 5 listaxes.
- De especificar número de valores para a saída, hai que tentar escribir casos para tratar os limites na saída. Por exemplo: "desconto máximo será o 50%, o mínimo será o 6%", casos válidos: descontos do 50% e o 6%, casos non válidos: descontos do 5.99%, e 50.01% se o desconto é un número real.
- Se a entrada ou saída é un conxunto ordenado (por exemplo, unha táboa, un arquivo secuencial,...), os casos deben concentrarse no primeiro e no último elemento.

Conxectura de erros:

A idea básica desta técnica consiste en enumerar unha lista de erros posibles que poden cometer os programadores ou de situacións propensas a certos erros e xerar casos de proba en base a dita lista. Esta técnica tamén se denominou xeración de casos (ou valores) especiais, xa que non se obteñen en base a outros métodos senón mediante a intuición ou a experiencia. Algúns valores a ter en conta para os casos especiais poderían ser os seguintes:

- O valor 0 é propenso a xerar erros tanto na saída como na entrada.
- En situacións nas que se introduce un número variable de valores, como por exemplo unha lista, convén centrarse no caso de non introducir ningún valor e un só valor. Tamén pode ser interesante que todos os valores sexan iguais.
- É recomendable supor que o programador puidese interpretar mal algo nas especificacións.
- Tamén interesa imaxinar as accións que o usuario realiza ao introducir unha entrada, mesmo coma se quixese sabotar o programa. Poderíase comprobar como se comporta o programa se os valores de entrada están fóra dos rangos de valores límites permitidos para ese tipo de variable. Por exemplo, se unha variable de entrada é de tipo *int*, deberíase comprobar o que ocorre se o valor de entrada está fóra do rango de valores permitido para un *int* ou mesmo se ten decimais ou é unha letra. Poderíase comprobar que na entrada non vai camuflado código perigoso. Por exemplo, comprobar a posible inxección de código nunha entrada a unha base de datos.
- Completar as probas de caixa branca e de caixa negra para o caso de bucles. Procurar que un bucle se execute 0 veces, 1 vez ou máis veces. De coñecer o número máximo de iteracións do bucle (n), habería que executar o bucle 0, 1, n-1 e n veces. Se hai bucles aniñados, os casos de proba aumentarían de forma exponencial, polo que se recomenda comezar probando o bucle máis interior mantendo os exteriores coas iteracións mínimas e ir creando casos de proba cara o exterior do aniñamento.

Probas unitarias aleatorias

Nas probas aleatorias simúlase a entrada habitual do programa creando datos para introducir nel que sigan a secuencia e frecuencia coas que poderían aparecer na práctica diaria, de maneira repetitiva (próbanse moitas combinacións). Para iso utilízanse habitualmente ferramentas denominadas xeradores automáticos de casos de proba.

Enfoque recomendado para o deseño de casos

As distintas técnicas vistas para elaborar casos de proba representan aproximacións diferentes. O enfoque recomendado consiste na utilización conxunta de ditas técnicas para lograr un nivel de probas “razoable”. Por exemplo:

- Elaborar casos de proba de caixa negra para as entradas e saídas utilizando clases de equivalencia, completar co análise do valor límite e coa conxectura de erros para engadir novos casos non contemplados nas técnicas anteriores.
- Elaborar casos de proba de caixa branca baseándose nos camiños do código para completar os casos de proba de caixa negra.