El testing como parte del proceso de calidad del software

Criterios de cobertura

Contenido

- ?

Criterios de selección

- Es una **mecanismo** para decidir si una prueba es adecuada o no. Se espera que un criterio sea:
  - **Regular**: si todos las pruebas que satisfacen el criterio detectan en general los mismos errores.
  - **Válido**: si para cualquier error en el programa hay un conjunto de pruebas que satisfacen el criterio y detectan el error.
- En general, es muy difícil conseguir criterios con buenas características de regularidad y validez.

Criterios de selección

- La forma más efectiva de hacer testing es de manera exhaustiva, pero es en general impracticable.
  - Por lo tanto se necesita algún criterio para seleccionar pruebas.
- El proceso guiado por un criterio identifica un conjunto de pruebas (clase) que es representativo de todas las pruebas posibles.
  - Si dos pruebas descubren el mismo defecto, deberían pertenecer a la misma clase.
  - El producto bajo prueba se comportará de la misma manera para todos las pruebas de una misma clase.

Principales enfoques

- Existen principalmente dos ramas para definir criterios de testing:
  - Caja negra (o funcional): abarca a aquellos criterios que deciden si una suite es adecuada analizando la especificación del software a testear (pero no su código)
  - Caja blanca (o estructural): abarca a aquellos criterios que deciden si una suite es adecuada analizando la estructura del código a testear.

Testing de caja negra

En el testing funcional, el SUT se trata como si fuera una caja negra:

Como no se observa el comportamiento interno del software, es necesario contar con una descripción de qué se espera del SUT (especificación).

Para diseñar los casos de test que conforman la suite, se usa el comportamiento esperado del sistema.

Relevancia de las especificaciones

Para hacer testing de caja negra, se debe contar con una especificación del SUT.

Es más efectivo si se cuenta con especificaciones ricas:

- A nivel de rutinas: pre- y post-condiciones.
- A nivel de módulos: especificación de diseño, contratos de clases, etc.
- A nivel de sistema: una buena especificación de requisitos.

Particionado en clases de equivalencia

Consiste en dividir el espacio de entrada en clases de equivalencias:

Las clases de equivalencia deberían corresponder a casos similares (para los cuales el SUT se comportaría de la misma manera).

Motivación: si el SUT funciona correctamente para un caso de test en una clase, se supone que lo hará para el resto de los casos de la misma clase.

Problema: definir una política de particionado adecuada.

Se debe analizar la especificación y definir clases de equivalencia de casos, identificando los inputs para los cuales se especifican distintos comportamientos.

Particionado en clases de equivalencia

- Una forma básica de hacer particionado por clases de equivalencia consiste en:

  - considerar cada condición especificada sobre las entradas como una clase de equivalencia,

  - incluir clases correspondiente a entradas inválidas,

  - si las entradas correspondientes a una clase no se tratan uniformemente (e.g., salidas diferentes), particionar aún más las clases teniendo en cuenta los diferentes tratamientos.

Particionado en clases de equivalencia: ejemplo

- Supongamos que tenemos una rutina que, dada una lista y una posición en la misma, retorna el elemento en esa posición:

```
{ lista != null && (pos>=0 && pos<lista.length) }
 public Object get(List lista, int pos)
{            ...  }
{ Resultado es el elemento de la lista en la posición pos }
```

- Mirando las entradas y salidas posibles, tenemos como clases de equivalencia la combinación de las siguientes condiciones:

lista	pos	resultado
"==null"	<0	"==null"
"!=null"	>=0 && <lista.length	"!=null"
	>= lista.length	

Análisis de valores de borde

En muchos casos, el software tiene casos especiales, o extremos, propensos a errores:

- estos valores suelen encontrarse en los "bordes" de las clases de equivalencia.
- Una suite de valores de borde para un particionado es un conjunto de tests que se encuentran en los bordes de las clases de equivalencia.
- En general, usamos valores de borde para complementar una suite basada en particionado por clases de equivalencia.

Análisis de valores de borde: ejemplo

- Volvamos a mirar la rutina del ejemplo anterior

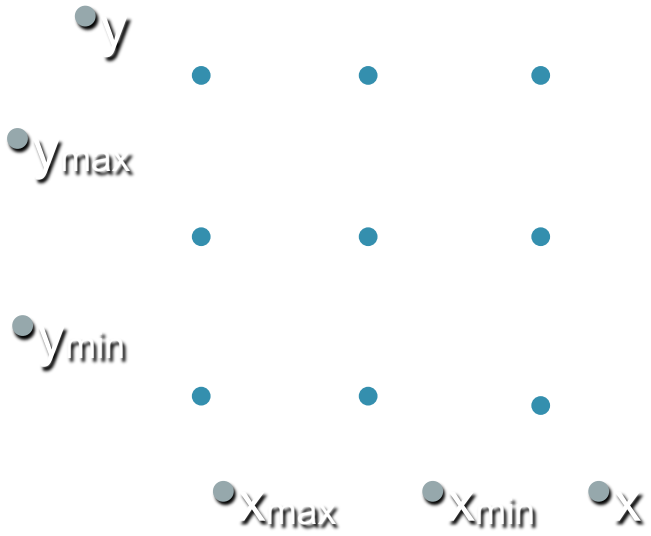
```
{ lista != null && (pos>=0 && pos<lista.length) }
 public Object get(List lista, int pos)
{            ...  }
{ Resultado es el elemento de la lista en la posición pos }
```

- Mirando las condiciones en las entradas, para cubrir los valores de borde deberíamos proveer tests que cubran las siguientes condiciones:

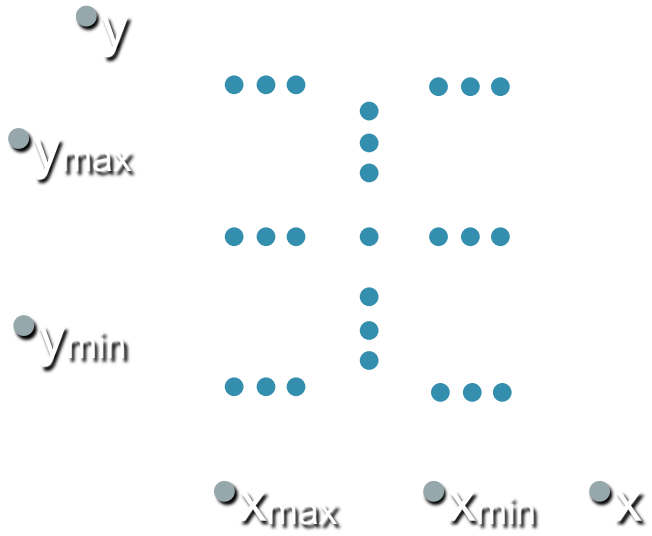
pos
"==-1"
"==0"
"==lista.length-1"
"==lista.length"

Clases de equivalencia con y sin

Valores bordes



clases de equivalencia



clases de equivalencia

valores de + borde

- Clases de equivalencia +
- Valores bordes
- Ejercicio

Técnicas de caja blanca

- A diferencia del testing de caja negra, en los **criterios/técnicas de caja blanca** se analiza el código fuente del software.
  - Es decir, los criterios de caja blanca se enfocan en la implementación.
- Muchos de los criterios exploran la estructura del código, intentando dar lugar a suites que ejerciten el código de maneras diferentes.

Cobertura de sentencias

- Se satisface **cobertura de sentencias** si todas las sentencias del programa son ejecutadas al menos una vez por algún test de la suite.
  - Una sentencia ejecutable es considerada aquella que esté asociada con código de máquina (asignaciones, evaluaciones de guardas, llamadas a funciones, inicializacion de variables, etc).
  - Es uno de los criterios de caja blanca *más débiles*.
  - Errores en condiciones compuestas y ramificaciones de programas pueden ser pasados por alto.
  - En muchos casos, con suites pequeñas se puede satisfacer este criterio.

Cobertura de sentencias: ejemplo

```
static void f(boolean b) {
  int a := 1;
  int c := 0;
  int d;
  if (b) {
     c := 2;
  };
  d := a + c;
}
```

TC #	b	Cobertura
1	TRUE	100%
2	FALSE	75%

Cobertura de sentencias: ejercicio

El siguiente programa determina si un año dado, es o no bisiesto:

```
public static boolean bisiesto(int a) {
    boolean b = false;
    if ((a%4==0) && (a%100!= 0))
            b = true;
    return b;
}
```

¿Con qué y cuántos test se logra cobertura de sentencias?

Cobertura de decisiones

- Una decisión es un punto en el código en el que se produce una ramificación o bifurcación.
  - Ej.: condiciones de ciclos, condiciones de if-then-else
- Se satisface **cobertura de decisión** si todas los resultados de las decisiones del programa son ejecutadas por al menos un test de la suite.
- **Propiedad**: cobertura de decisión es más fuerte que cobertura de sentencias.
  - Si una suite satisface cobertura de decisión, también satisface cobertura de sentencias.

Cobertura de decisiones: ejemplo

```
static void f(boolean b) {
  int a := 1;
  int c := 0;
  int d;
  if (b) {
     c := 2;
  };
  d := a + c;
}
```

TC #	b	Cobertura
1	TRUE	100%
2	FALSE	

Cobertura de decisiones: ejercicio

El siguiente programa determina si una cadena es capicúa:

```
public static boolean capicua(char[] list) {
    int index = 0;
    int l = list.length;
    while (index<(l-1)) {
        if (list[index] != list[(l-index)-1])
            return false;
        index++;
    }
    return true;
}
```

¿Con qué y cuántos test se logra cobertura de decisiones?

Cobertura de condiciones

- Una decisión puede estar compuesta por una o más condiciones.
  - Una *condición* es una expresión booleana que es evaluada para determinar el resultado de una decisión.
  - Ej: `if (index < list.length) && !found …`
- Se satisface **cobertura de condición** si cada condición (de cada decisión) es ejecutada por verdadero y por falso por al menos un test de la suite.
- **No es lo mismo que todas las combinaciones!**
  - Cobertura de condición **NO** es más fuerte que cobertura de decisión (son incomparables).

Cobertura de condiciones: ejemplo

```
static void f(boolean x, y, z) {
  int a := 1;
  int c := 0;
  int d;
  if (x || (y && z) {
     c := 2;
  };
  d := a+c;
}
```

TC	x	y	z	Res.	Cob. de decisiones	Cob. de condiciones
1	TRUE	FALSE	FALSE	TRUE	50%	100%
2	FALSE	TRUE	TRUE	TRUE		
3	TRUE	TRUE	TRUE	TRUE	100%	100%
4	FALSE	FALSE	FALSE	FALSE		

Cobertura de condiciones: ejercicio

El siguiente programa determina si un valor ocurre en un arreglo:

```
public static boolean find(int[] list, int val) {
    int index = 0;
    boolean found = false;

    while (index<(list.length-1) && !found) {
        if (list[index] == val)
            found = true;
        index++;
    }
    return found;
}
```

¿Con qué y cuántos test se logra cobertura de condiciones?

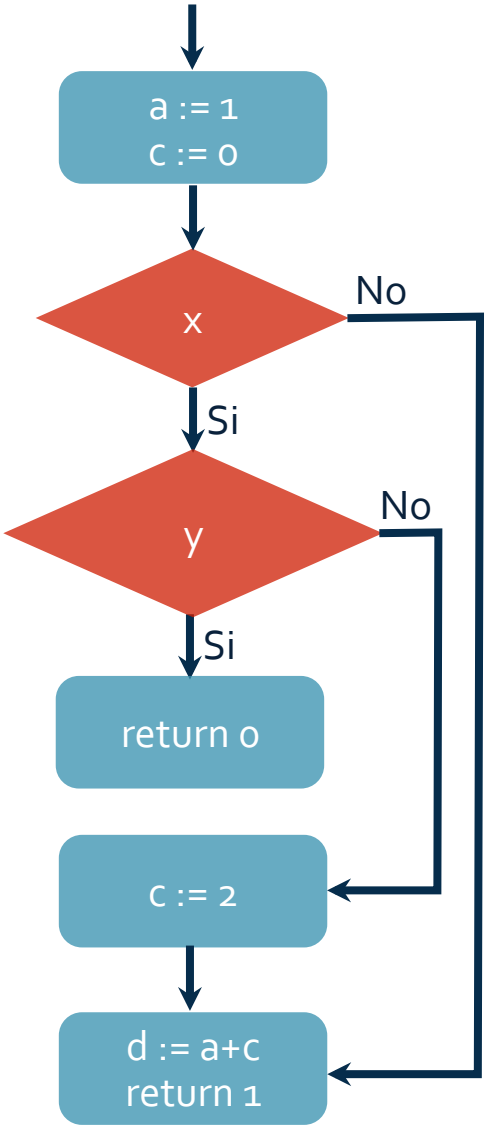
Cobertura de caminos

- El **grafo de flujo de control** de un programa es una representación, mediante grafos dirigidos, del flujo de control del programa:
  - Los nodos del grafo representan segmentos de sentencias que se ejecutan secuencialmente.
  - Los arcos del grafo representan transferencias de control entre nodos.

```
static void f(boolean x,y) {
  int a := 1;
  int c := 0;
  int d;
  if (x) {
     if (y)
        return 0;
     c := 2;
  };
  d := a+c;
}
```

Cobertura de caminos

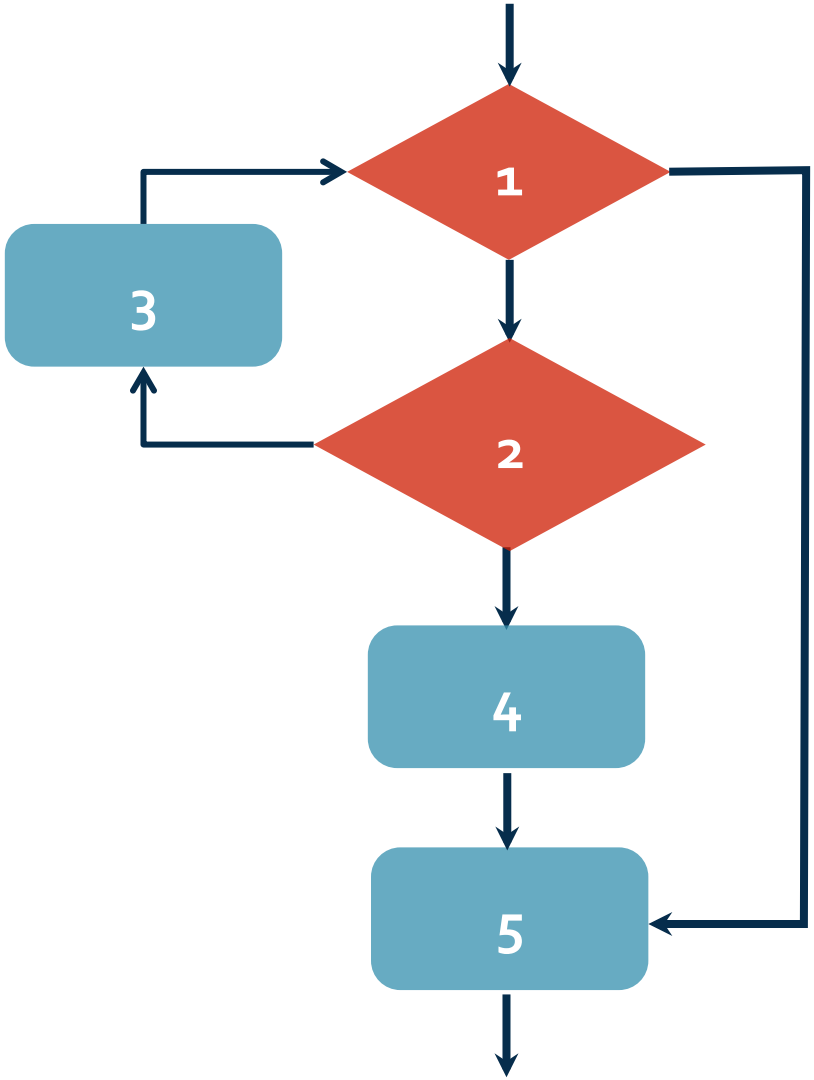
- El **grafo de flujo de control** de un programa es una representación, mediante grafos dirigidos, del flujo de control del programa:
  - Los nodos del grafo representan segmentos de sentencias que se ejecutan secuencialmente.
  - Los arcos del grafo representan transferencias de control entre nodos.



Cobertura de caminos

- Se satisface cobertura de caminos si todos los caminos del grafo de flujo de control son recorridos por al menos una test de la suite.
  - **Es un criterio muy fuerte**: conseguirlo puede requerir suites muy grandes.
  - Si una suite satisface cobertura de caminos, satisface cobertura de decisiones y por lo tanto, de sentencias.
- Suelen imponerse restricciones al criterio para hacerlo practicable:
  - **cobertura de caminos simples**: requiere cubrir caminos sin repetición de arcos.
  - **cobertura de caminos elementales**: requiere cubrir caminos sin repetición de nodos.

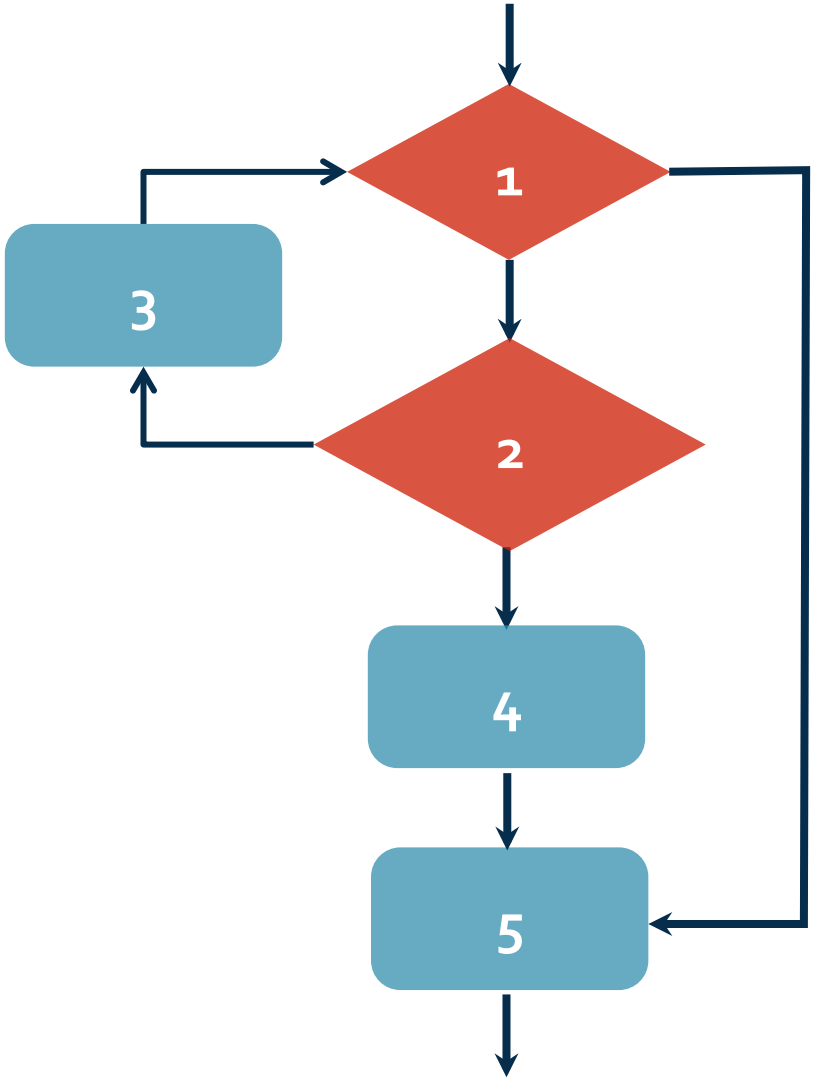
Cobertura de caminos



Caminos posibles:

- 1, 5
- 1,2,4,5
- 1,2,3,1,2,4,5
- 1,2,3,1,2,3,1,2,4,5
- 1,2,3,1,2,3,1,2,3,1,2,4,5
- …
- 1,2,3,1,5
- 1,2,3,1,2,3,1,5
- 1,2,3,1,2,3,1,2,3,1,5
- …

Cobertura de caminos



Caminos simples:

- 1, 5
- 1,2,4,5
- 1,2,3,1,5

Caminos elementales:

- 1,5
- 1,2,4,5
- 1,2,3,1,5
- 1,2,3,1,2,4,5

EclEmma Jacoco

Es una herramienta para medir cobertura de una test suite, de acuerdo a algunos criterios de caja blanca:

- Diseñada para Java+JUnit.
- Se puede instalar como un plugin de Eclipse.
- Muestra visualmente el código cubierto + estadísticas de cobertura.

Demo y ejercicio

- EclEmma Jacoco
- http://update.eclemma.org/
- Completar la cobertura de StaticRoutines

Testing basado en mutación

- Según este criterio, una suite es adecuada si es efectiva para descubrir bugs inyectados:
  - los bugs se inyectan en el código, mutando operaciones del mismo
  - cada variante del código original es un mutante
- El objetivo: matar todos los mutantes:
  - un mutante está muerto si existe un test de la suite que no falla en el original, pero si en el mutante.

Mutantes: ejemplo

```
public static boolean bisiesto(int a) {
      boolean b = false;
      if ((a%4==0) && (a%100!= 0))
         b = true;
      return b;
}
```

- El siguiente es un mutante del anterior:

```
public static boolean bisiesto(int a) {
      boolean b = false;
      if ((a%4==0) || (a%100!= 0))
         b = true;
      return b;
}
```

µJava

- Es una herramienta para evaluar suites de acuerdo a mutación:
  - Diseñada para Java+JUnit.
  - Crea mutantes a partir del código original:
    - Mutantes de clase: modificador de acceso, variables ocultas, sobrecarga de métodos, asignaciones compatibles, casting, etc.
    - Mutantes de métodos: operadores aritméticos, relacionales, condicionales, de shift, lógicos, etc.
  - Corre suites JUnit y mide % de mutantes muertos
  - Reporta los mutantes que quedaron vivos.

Demo

- μJava

Testing aleatorio

- Según esta perspectiva, en lugar de diseñar o elegir test basado en algún criterio, se crean test aleatoriamente.
  - Principalmente para testing de regresión.
- Pero generar test aleatoriamente es complejo:
  - no toda secuencia de métodos es válida;
  - los test redundantes no agregan confianza.

Testing aleatorio

- El siguiente test es ilegal:

```
@Test public void test() {

    Date d1 = new Date(2011,7,22);
    d1.setMonth(-1);
    Date d2 = new Date(2012,7,4);
    boolean v = d1.before(d2);

    assertTrue(v);
}
```

- Modificar el mes con un valor negativo lanza una excepción.
- Extender esa porción del test case de cualquier manera seguirá siendo ilegal.

Testing aleatorio

- Los siguientes test son redundantes:

```
@Test public void test() {

    Node n = new Node();
    Tree t = new Tree(n);
    Node m = new Node();
    boolean v = t.isRoot(m);
    assertTrue(v == false);
}
```

```
@Test public void test() {

    Node n = new Node();
    Tree t = new Tree(n);
    Node m = new Node();
    boolean v = t.isRoot(m);
    boolean w = t.isLeaf(m);
    assertTrue(v == false);
}
```

Randoop

- Randoop es una herramienta para la generación automática de tests de unidad.
- Basada en generación aleatoria.
- Incorpora técnicas eficientes para:
  - intentar eliminar casos de test inválidos,
  - intentar no producir casos de test redundantes.

Randoop

- La generación se realiza en cuatro pasos:
  1. Se crea una secuencia de métodos, eligiendo métodos y sus argumentos aleatoriamente, extendiendo sencuencias anteriores de forma incremental.
  2. Se descarta la secuencia si ya fue creada anteriormente (equivalencia módulo nombre de variables).
  3. Se ejecuta la secuencia para comprobar que ningún "contrato" sea violado.
  4. Se determina qué valores de la secuencia pueden ser utilizados como entradas para nuevas llamadas a métodos cuando se cree una nueva secuencia. Se filtran redundantes/ilegales.

Demo

- Randoop

Introducción al testing unitario