Validación y Verificación de Software

Criterios de cobertura: Caja blanca/Caja negra | Clases de Equivalencia | Valores de borde | Cobertura basada en flujo de control | JaCoCo | Mutación | Pitest

Criterios de testing

Un criterio de testing es una mecanismo para decidir si una suite es adecuada o no. En general, se espera que un criterio de test cumpla con:

Regularidad: Un criterio es regular si todos los conjuntos de tests que satisfacen el criterio detectan en general los mismos defectos.

Validez: Un criterio es válido si para cualquier defecto en el programa hay un conjunto de tests que satisfaga tal criterio y detecte el defecto.

En general, es muy difícil conseguir criterios con buenas características de regularidad y validez.

Principales enfoques de testing

La forma más efectiva de hacer testing es de manera exhaustiva, pero es en general impracticable.

Luego, se necesita algún criterio para seleccionar tests.

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 el 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 tests 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 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 tests, 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

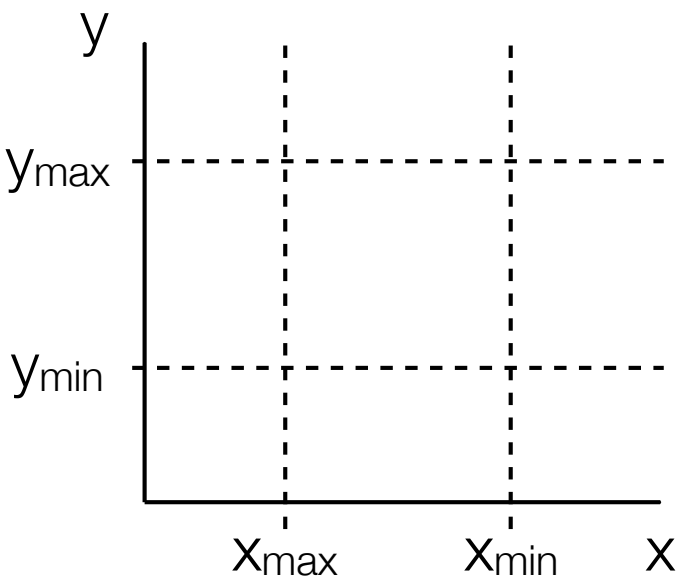
Volvamos a mirar la rutina del ejemplo anterior

```
{ 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 }
```

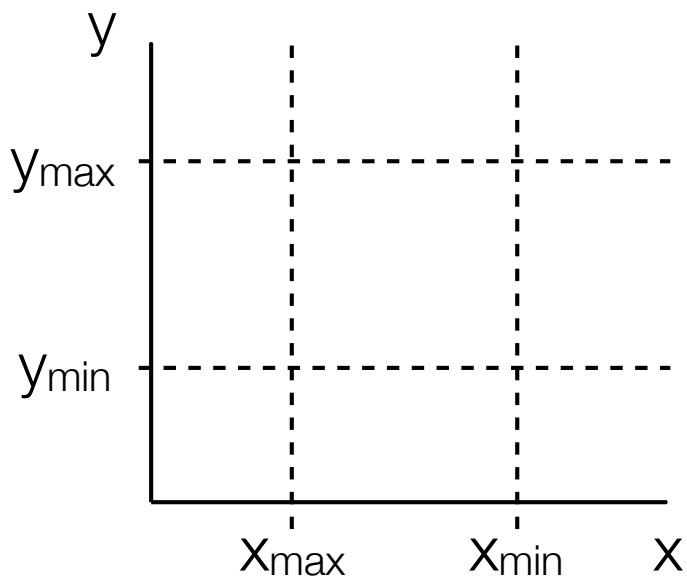
Mirando las condiciones en las entradas, para cubrir los valores de borde deberíamos proveer tests que cubran las siguientes condiciones:

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

Testing de caja blanca

A diferencia del testing de caja negra, en los criterios de test de caja blanca se analiza el código del SUT para decidir si una suite es adecuada o no.

Es decir, los criterios de caja blanca se enfocan en la implementación.

Muchos de los criterios exploran la estructura del código a testear, intentando dar casos que ejerciten el código de maneras diferentes.

Cobertura de sentencias

Una test suite satisface el criterio de cobertura de sentencias si todas las sentencias del programa son ejecutadas al menos una vez por algún test de la suite.

- 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 (Caja Blanca)

Una test suite satisface este criterio si todas las sentencias del programa son ejecutadas al menos una vez por algún test de la suite.

```
/**
 *  Return whether an array is palindromic
 * @param An array if char
 * @return return true if the given array is palindromic.
 */
public static boolean capicua(char[] list) {
    int index = 0;
    int l = list.length;
    boolean res = true;
    while(index<(l-1)){
        if(list[index] != list[(l-index)-1]){
                res= false;
        }
        index++;
    }
    return res;
}
```

```
@Test
public void testCapicua() {
    char [] a ={'h', 'o','l','a'};
    boolean b  = capicua(a);
    assertFalse(b);

}
```

Cobertura de sentencias

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.

Puntos de Decisión

```
/**
 *  Return whether an array is palindromic
 * @param An array if char
 * @return return true if the given array is palindromic.
 */
public static boolean capicua(char[] list) {
    int index = 0;
    int l = list.length;
    boolean res = true;
    while(index<(l-1)){
        if(list[index] != list[(l-index)-1]){
                res= false;
        }
        index++;
    }
    return res;
}
```

Cobertura de decisión

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

Una suite satisface el criterio de cobertura de decisión si todas las decisiones del programa son ejecutadas por true y por false al menos una vez.

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 Decisión (Caja Blanca)

Una suite satisface el criterio de cobertura de decisión si todas las decisiones del programa son ejecutadas por true y por false al menos una vez.

Cobertura de decisión vs branches- agregar condición decision etc

```
/**
 *  Return whether an array is palindromic
 * @param An array if char
 * @return return true if the given array is palindromic.
 */
public static boolean capicua(char[] list) {
    int index = 0;
    int l = list.length;
    boolean res = true;
    while(index<(l-1)){
        if(list[index] != list[(l-index)-1]){
                res= false;
        }
        index++;
    }
    return res;
}
```

```
@Test
public void testNoCapicua() {
    char [] a ={'h', 'o','l','a'};
    boolean b  = capicua(a);
    assertFalse(b);
}
```

```
@Test
public void testCapicua() {
    char [] a ={'n', 'e','u','q','u','e','n'};
    boolean b  = capicua(a);
    assertTrue(b);
}
```

Cobertura de Decisión

Podemos encontrar suites de test más chicas que satisfagan Cobertura de Decisión para capicúa.

```
 @Test
 public void testNoCapicua() {
     char [] a ={'a', 'n','t','o','r','c','h','a'};
     boolean b  = capicua(a);
     assertFalse(b);
}
```

En general se obtienen suites de test débiles para detectar, por ejemplo, errores en condiciones complejas.

Cobertura de caminos

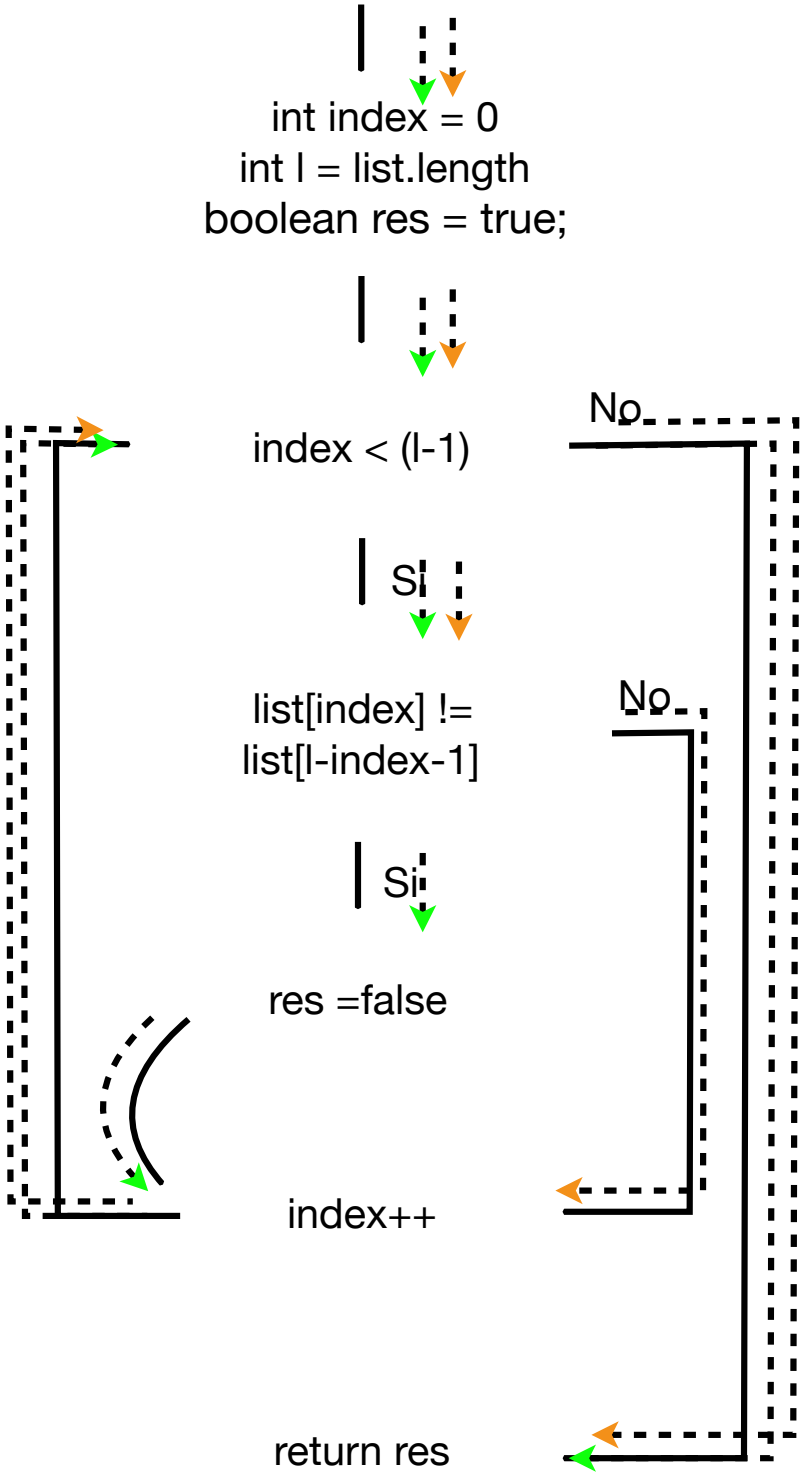
Una suite satisface el criterio de cobertura de caminos si todos los caminos de ejecución del programa bajo prueba son recorridos al menos una vez.

Es un criterio muy fuerte: conseguirlo puede requerir suites muy grandes e incluso infinitas

Grafos de flujo de control

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

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 de longitud n

Herramientas para medir cobertura

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.

Lectura recomendada

JaCoCo: https://www.eclemma.org/

Software Unit Test Coverage and Adequacy (http://www.cs.toronto.edu/~chechik/courses07/csc410/p366-zhu.pdf)