Pruebas Unitarias en Java

CodeCover es una herramienta de cubrimiento de código que brinda información acerca de distintos tipos de cubrimiento.

En el número anterior de esta revista vimos cómo realizar pruebas unitarias utilizando Clover como herramienta de cobertura de código. Clover brinda información sobre el cubrimiento de sentencias y de decisión pero no acerca de cómo han sido cubiertos los ciclos (while, for, do-while) del método bajo prueba.
Los ciclos son una fuente importante de fallas en los programas y por este motivo hay que construirlos, revisarlos y probarlos con cuidado. En este artículo presentaremos el cubrimiento de ciclos y la herramienta CodeCover que da soporte al mismo.

Cubrimiento de ciclos

Los distintos criterios de cubrimiento de ciclos definen la forma en la cuál deben ser ejecutados los ciclos al momento de realizar las pruebas. Para cierta ejecución un ciclo puede no ser ejecutado, ser ejecutado una única vez o ejecutarse varias veces. Estos son los casos que el criterio de ciclos exige sean ejercitados durante las pruebas. Cabe aclarar que, para ciclos del tipo do-while, se excluye el caso de no ejecución ya que el mismo es imposible.
Por ejemplo, un método que realiza una búsqueda de un elemento en un arreglo podría contener un while que recorre el arreglo hasta que el mismo se termina o hasta que el elemento ha sido encontrado: while (index < arreglo.length && !encontrado) {
Los tres casos exigidos por el criterio de cubrimiento de ciclos se pueden cumplir de la siguiente manera:
No ejecutar el ciclo – El caso de prueba a utilizar es que el arreglo sea vacío
Ejecutar una única vez el ciclo - Se pueden usar dos casos distintos: un arreglo con un único elemento que no es el buscado o un arreglo cuyo primer elemento es el buscado.
Ejecutar más de una vez el ciclo – También se pueden usar dos casos distintos: un arreglo que contiene al elemento buscado pero que no está en la primera posición o un arreglo que no contiene al elemento buscado y sea de largo mayor a 1.
Entonces, para este ejemplo, son al menos necesarios tres casos de prueba para lograr cumplir con el criterio de cubrimiento de ciclos.
Existen distintos tipos de cubrimientos de ciclos y estos se diferencian en cómo cada uno exige cubrir los ciclos llamados anidados. Un ciclo es anidado cuando se encuentra dentro de otro ciclo. Los ciclos anidados y su cubrimiento quedan por fuera del alcance de este artículo.

Medición del cubrimiento de ciclos con CodeCover

CodeCover es una herramienta de cubrimiento de código que brinda información acerca de distintos tipos de cubrimiento, uno de estos es el cubrimiento de ciclos.
Para cada ciclo de un método de una clase en Java, CodeCover definirá distintos ítems candidatos. Estos se corresponden con las distintas posibilidades ya analizadas de los ciclos: no se ejecuta el ciclo ninguna vez, se ejecuta una única vez y se repite varias veces. El porcentaje total de cubrimiento luego de ejecutado un conjunto de casos de prueba se calcula dividiendo el número de ítems candidatos cubiertos entre la cantidad total de ítems candidatos.
Para visualizar de forma sencilla el cubrimiento logrado luego de ejecutadas las pruebas CodeCover resalta el código con diferentes colores. Cada ciclo del código es pintado indicando si todos, alguno o ninguno de los ítems candidatos que contiene fueron cubiertos. Esto es útil para la identificación de casos de prueba faltantes para cumplir con el cubrimiento.
Se utiliza el color verde para indicar que todos los ítems del ciclo fueron cubiertos, el amarillo para cubrimiento parcial (al menos un ítem fue cubierto pero no todos) y el rojo si ninguno de los ítems fue cubierto.
En los casos en los que el cubrimiento de cierto ciclo no es total (color amarillo), es posible consultar el detalle de los ítems candidatos que no fueron cubiertos por el conjunto de casos de prueba.

Esta información sirve para construir nuevos casos de prueba que lleven a cumplir con el 100% del cubrimiento.

Un ejemplo sencillo

En la Figura 1 se presenta el método merge; mismo método que usamos como ejemplo en el artículo anterior (partiendo de la versión corregida). Este recibe dos arreglos de números enteros ordenados de menor a mayor. El método "toma" los elementos de los dos arreglos y devuelve otro arreglo con esos elementos ordenados de menor a mayor.

Ejecución usando CodeCover

Utilizamos con CodeClover los mismos casos de prueba que se presentaron en el artículo anterior. La Figura 3.a presenta un caso de prueba en el cual los elementos del primer arreglo son todos menores que los del segundo arreglo. La Figura 3.b presenta otro caso de prueba en donde los elementos de los arreglos pasados se deben intercalar. Ambos casos están codificados en JUnit y conforman el conjunto de casos de prueba de nuestro ejemplo.

El método merge contiene 3 ciclos (1 while y 2 for) que determinan 9 ítems candidatos. Nos referiremos a los ítems candidatos como Cero, Uno y Muchos para los casos en que el ciclo no es ejecutado, es ejecutado una vez y es ejecutado varias veces, respectivamente.

Al ejecutar los dos casos de prueba se obtiene un cubrimiento del 55,6%. Este corresponde a la ejecución de 5 de los 9 ítems. En la Figura 2 se muestra el código resaltado correspondiente a la ejecución de estos casos usando CodeCover. Se observa que los ciclos (while y for) dentro del método están pintados con amarillo; esto indica que todos los ciclos tienen únicamente un cubrimiento parcial de los ítems candidatos.

```
public int[] merge (int[] a, int[] b){
    int i = 0;
    int j = 0;
    int k = 0;
    int c[] = new int[a.length + b.length];
    while(i < a.length && j < b.length){
        if( a[i] < b[j] ){
             c[k] = a[i];
             k++;
             i++;
        }else{
            c[k] = b[j];
            k++;
            j++;
        }
    }
    for(int iter = i; iter < a.length; iter++){
        c[k] = a[iter];
        k++;
    }
    for(int iter = j; iter < b.length; iter++){
        c[k] = b[iter];
        k++;
    }
    return c;
}
```

Figura 1 – Método merge

```
public int[] merge (int[] a, int[] b){
    int i = 0;
    int j = 0;
    int k = 0;
    int c[] = new int[a.length + b.length];
    while(i < a.length && j < b.length){
        if( a[i] < b[j] ){
             c[k] = a[i];
             k++;
             i++;
        }else{
            c[k] = b[j];
            k++;
            j++;
        }
    }
    for(int iter = i; iter < a.length; iter++){
        c[k] = a[iter];
        k++;
    }
    for(int iter = j; iter < b.length; iter++){
        c[k] = b[iter];
        k++;
    }
    return c;
}
```

Figura 2 – Cubrimiento CodeCover

```
@org.junit.Test
public void testmerge1(){
    int a[] = {1,2,3,4,5,6};
    int b[] = {7,8,9,10,11};
    Operaciones oper = new Operaciones();
    int c[] = oper.merge(a,b);
    int esperado[] = {1,2,3,4,5,6,7,8,9,10,11};
    assertArrayEquals(esperado,c);
}
```

Figura 3.a – Caso de prueba testmerge1

```
@org.junit.Test
public void testmerge2(){
    int a[] = {1,2,5,11};
    int b[] = {3,4,8,10};
    Operaciones oper = new Operaciones();
    int c[] = oper.merge(a,b);
    int esperado[] = {1,2,3,4,5,8,10,11};
    assertArrayEquals(esperado,c);
}
```

Figura 3.b – Caso de prueba testmerge2

El while ha sido solamente cubierto en el ítem Muchos. Para el primer for se han cubierto los ítems Cero y Uno. En cambio, para el segundo for los ítems cubiertos son Cero y Muchos. Esto indica que para cumplir con el cubrimiento de ciclos son necesarios casos de prueba en los cuales los ítems Cero y Uno del while, Muchos del primer for y el ítem Uno del segundo for sean ejecutados.
El siguiente paso para lograr el 100% del cubrimiento es analizar el tipo de casos de prueba que se necesita para cumplir con cada uno de los ítems que no han sido cubiertos.

Ítem Cero del While – Caso de prueba en el cual al menos uno de los dos arreglos sea vacío. De esta forma la decisión se hace falsa y no se ingresa nunca al while.

Ítem Uno del While – Caso de prueba en el cual a[0] es menor a b[0] y el arreglo a es de largo igual a 1; obviamente el arreglo b tiene que ser al menos de largo 1. El caso inverso también sirve. Esto provoca que se ingrese una única vez al while.
Ítem Muchos del primer for – Caso de prueba en el cual el arreglo a tiene al final al menos 2 elementos que son mayores que el último elemento del arreglo b (En caso que b sea vacío, cualquier arreglo con al menos 2 elementos sirve para el arreglo a). Esto provoca salir del while sin haber ordenado dichos elementos en el arreglo c y por ende, se recorrerá al menos 2 veces el primer for.

Ítem Uno del segundo for – Caso de prueba en el cual el arreglo b tiene en su última posición un elemento mayor que el de la última posición del arreglo a (en caso que el arreglo a sea vacío. b puede ser cualquier arreglo con un único elemento). Además, en caso de tener más de 1 elemento, el elemento en la penúltima posición de b debe ser más chico que el de la última posición de a. Un caso así provoca salir del while con un único elemento por ordenar en el arreglo c, ese elemento es el último de b y por ende se recorrerá el segundo for una única vez.
Para cumplir con el 100% del cubrimiento es necesario agregar casos de prueba que satisfagan los tipos de casos descritos. Los casos presentados en la figura 4 cumplen con este requerimiento. Entonces, estos casos, ejecutados en conjunto con los dos anteriores, logran cubrir los 9 ítems candidatos definidos por CodeCover, logrando el 100% del cubrimiento de ciclos.

¡Al ejecutar los cuatro casos de prueba CodeCover indica con el color verde sobre el while y los 2 for que el cubrimiento ha sido alcanzado de forma satisfactoria!

```
@org.junit.Test
public void testmerge3(){
    int a[] = {};
    int b[] = {8};
    Operaciones oper = new Operaciones();
    int c[] = oper.merge(a,b);
    int esperado[] = {8};
    assertArrayEquals(esperado,c);
}
```

```
@org.junit.Test
public void testmerge4(){
    int a[] = {4,5};
    int b[] = {3};
    Operaciones oper = new Operaciones();
    int c[] = oper.merge(a,b);
    int esperado[] = {3,4,5};
    assertArrayEquals(esperado,c);
}
```

Figura 4 – Casos de prueba agregados

Carmen Bogado
Diego Vallespir
Grupo de Ingeniería de Software, UdelaR
gris@fing.edu.uy