Pruebas Unitarias en Java
Cubrimiento de código con Clover

Combinar distintos tipos de técnicas permite aumentar la efectividad en la detección de defectos en el software.

Un camino defectuoso en un código fuente, o incluso un defecto en una única sentencia, puede causar una tragedia irreparable al ejecutarse. Conocer qué partes del código son ejecutadas por las pruebas de software ayuda a evitar un problema cuando el sistema se encuentre en producción. Por suerte, desde hace un tiempo, existen herramientas automatizadas capaces de brindarnos información acerca del código que ha sido probado.

En la primer parte de este artículo, publicado en el último número del año pasado de esta revista, presentamos conceptos básicos de las pruebas unitarias y los drivers JUnit y TestNG. En esta segunda parte trataremos el cubrimiento de código a nivel unitario y presentaremos la herramienta Clover .
Combinar distintos tipos de técnicas permite aumentar la efectividad en la detección de defectos en el software. Revisiones personales, inspecciones, pruebas de caja negra y pruebas de caja blanca son combinadas para construir productos cercanos a cero defecto.
Las técnicas de caja blanca se basan en la estructura del código fuente para definir los casos de prueba. Distintas técnicas se enfocan en distintas estructuras del código. Por ejemplo, la técnica cubrimiento de sentencias se basa en obtener un conjunto de casos de prueba que logre ejecutar todas las sentencias del código bajo prueba. Utilizan una medida que se llama cubrimiento de código, que indica el porcentaje del código que cubrió la prueba. El cubrimiento de código es dependiente de la técnica específica que se esté utilizando. Por ejemplo, cubrimiento de sentencias indica el porcentaje de sentencias ejecutadas durante las pruebas, y trayectorias linealmente independientes indica el porcentaje de trayectorias cubiertas del total de trayectorias que son linealmente independientes.
Es un tema de investigación actual el intentar conocer la relación costo-beneficio de estas técnicas. Por lo tanto, no es trivial ni la elección de una de estas técnicas ni la selección de una estrategia completa de pruebas unitarias.

Herramienta de cubrimiento – Clover

Clover es una herramienta que mide el cubrimiento de sentencias y de decisión alcanzado por un conjunto de casos de prueba. Esta herramienta se puede utilizar de forma integrada tanto con JUnit como con TestNG.
Brinda asistencia al desarrollador de software para lograr cumplir con el cubrimiento de decisión durante las pruebas. Por ejemplo, colorea las sentencias para indicar el cubrimiento obtenido. A su vez, genera reportes que ayudan a determinar a nivel global el cubrimiento obtenido, permitiendo saber de forma rápida cuáles de las clases que no han sido probadas en profundidad tienen mayor probabilidad de contener defectos. La herramienta es sencilla de utilizar y cuenta con una interfaz gráfica amigable.

Un ejemplo sencillo

El siguiente ejemplo servirá tanto para presentar el cubrimiento de decisión usando la herramienta Clover, como para introducir el cubrimiento de trayectorias linealmente independientes y la herramienta Coview en la parte 3 de este artículo.
En la Figura 1 se muestra el método merge. 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.

```
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]=a[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]=[iter];
        k++;
    }
    return c;
}
```

La Figura 2-a presenta un caso de prueba en el cual los elementos del primer arreglo son todos menores que los del segundo arreglo. La Figura 2-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.

```
@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 2-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[]={1,2,3,4,5,8,9,10,11};
    assertArrayEquals(esperado,c);
}
```

Figura 2-b – Caso de prueba testmerge2

Ejecución usando Clover

Ejecutando únicamente el caso de prueba testmerge1 se obtiene un cubrimiento de decisión del 75%. En el método merge se tienen 4 decisiones: un while, un if y dos for. Cada una de estas decisiones puede tomar tanto el valor false como el valor true. El caso testmerge1 en su ejecución logra que el while así como el segundo for tomen tanto el valor true como el false. Sin embargo, el if y el primer for toman solamente true y false respectivamente. Es decir, 6 de las 8 posibilidades de las 4 decisiones son ejecutadas, y esto representa el 75%.
Clover pinta las sentencias ejecutadas para ayudar a visualizar cuál es el código que falta cubrir. El color verde indica que la sentencia fue ejecutada y su resultado coincide con el esperado. El color rojo (en realidad rosa) indica que la sentencia no fue ejecutada en las pruebas.

Las decisiones son pintadas de color verde cuando se han ejecutado con sus dos valores de verdad y no son pintadas en caso contrario. Estos colores brindan una rápida visualización sobre el código aún no cubierto por las pruebas.
La Figura 3 presenta el código pintado por Clover luego de ejecutar el caso de prueba testmerge1. Se pueden ver fácilmente las sentencias no ejecutadas y las decisiones que sólo fueron cubiertas parcialmente (if y primer for).
Continuando con el ejemplo ejecutamos el segundo caso de prueba: testmerge2. Con esto se pretende cumplir con el criterio de decisión. Al ejecutar este segundo caso de prueba utilizando Clover ciertas líneas de código quedan coloreadas con amarillo. Clover usa este color para mostrar sentencias que solamente fueron ejecutadas por casos de prueba que fallan. Es decir, casos de prueba cuyos resultados esperados son diferentes a los resultados realmente obtenidos. De todas formas, siempre es conveniente observar los resultados de JUnit o TestNG antes de corroborar el cubrimiento de código; solamente cuando los casos de prueba no fallan es interesante observar el cubrimiento. El código luego de ejecutados ambos casos de prueba se presenta en la Figura 4.

El defecto que provoca la falla del caso de prueba testmerge2 se encuentra en la línea a continuación del else. La sentencia c[k] = a[j]; debe ser cambiada por c[k] = b[j]; Luego de corregido este defecto se vuelven a correr los casos de prueba y todo el código queda pintado de verde. ¡El código ha sido cubierto al 100% del criterio de decisión!

```
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]=a[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]=[iter];
        k++;
    }
    return c;
}
```

Figura 3 – Cubrimiento luego de testmerge1

```
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]=a[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]=[iter];
        k++;
    }
    return c;
}
```

Figura 4 – Cubrimiento luego de ambos casos

El buen uso del cubrimiento de código

El cubrimiento de código nos brinda una idea clara de las porciones de código ejecutadas. Sin embargo, no es razonable basarse solamente en estas pruebas. Una estrategia razonable es la aplicación de revisiones personales antes de comenzar las pruebas; de esta manera la calidad del producto aumentará.
También es recomendable pensar primero en pruebas de caja negra que cubran las distintas funcionalidades de la componente, clase o método a probar. Luego de ejecutadas estas pruebas y corregidos los defectos, se puede proceder a examinar el cubrimiento de código.

Si el código no fue cubierto como se definió en la estrategia se generarán nuevos casos de prueba para lograr ese cubrimiento. Esta estrategia de verificación unitaria (revisiones personales, pruebas de caja negra, pruebas de caja blanca) brindará un producto de mayor calidad que una estrategia que solamente se basa en un tipo de técnica.

Adriana Ávila
Lucía Camilloni
Diego Vallespir
Cecilia Apa

Grupo de Ingeniería de Software, UdelaR
gris@fing.edu.uy