Pruebas Unitarias en Java JUnit y TestNG

Tienen una gran importancia ya que son usados como marco por muchas otras herramientas de soporte a las pruebas unitarias.

En los últimos años las pruebas unitarias han tomado mucha importancia en el desarrollo de software. Estas pruebas buscan conocer el funcionamiento de las unidades de software de forma temprana. Uno de los motivos por el cual estas pruebas son ahora más conocidas y más usadas es la incorporación de metodologías ágiles como Scrum y XP.

Al momento de incorporar pruebas unitarias al desarrollo de software varios aspectos deben ser considerados. Algunos son los siguientes:

- Qué es una unidad de software
- Quién diseña y ejecuta las pruebas, cómo lo hace y cuándo lo hace
- Cuándo se puede liberar la unidad (criterio de liberación)
- Qué medidas (métricas) se deben usar
- Qué herramientas usar

Sin considerar estos y otros aspectos la introducción de las pruebas unitarias en una organización puede ser un fracaso. Probablemente, el mayor riesgo que se corre con las pruebas unitarias es "caer" en un ciclo code-fix. Este tipo de ciclo ocurre normalmente cuando se depende únicamente de las pruebas unitarias para construir unidades de calidad. Lo único que se logra en esos casos son unidades remendadas de baja calidad. Esta discusión se presentó en el artículo "TSP/PSP en Uruguay" publicado en esta revista en su edición Isaac Newton.

En este artículo mencionaremos algunos de los factores a considerar al momento de introducir las pruebas unitarias en una organización y también presentaremos brevemente dos drivers (herramientas para ejecutar pruebas) para el lenguaje Java: JUnit y TestNG.

Quién, cómo y cuándo

Si bien existen diversos enfoques y algunos procesos de desarrollo proponen variantes, normalmente las pruebas unitarias son realizadas por la misma persona que construyó la unidad. El conocimiento de la unidad, por haberla construido, permite encontrar los defectos de forma rápida luego que alguna prueba falla. Sin embargo, existen diversas variantes. Por ejemplo, el diseño de las pruebas lo realiza una persona distinta a quien desarrolló la unidad; pero, la ejecución y corrección de defectos las realiza el desarrollador.

También al introducir las pruebas unitarias en una organización se debe considerar cómo se van a realizar las mismas. Esto implica definir la estrategia y las técnicas de pruebas unitarias a ser utilizadas. Un enfoque interesante es combinar técnicas de caja negra y técnicas de caja blanca.

Las técnicas de caja negra son pruebas que se desarrollan a partir de la especificación de la unidad a probar. En estas lo importante es probar la funcionalidad que la unidad debe cumplir.

Las técnicas de caja blanca son pruebas que se basan en el código de la unidad. Lo fundamental de estas pruebas es lograr cubrir (ejecutar) ciertas sentencias y condiciones del código.

Existen diferentes técnicas de caja negra y caja blanca. La introducción de las pruebas unitarias en una organización debe hacerse con una clara definición de qué técnicas van a ser utilizadas. Sin esta definición distintas unidades construidas por distintos desarrolladores tendrán diferencias notorias en la calidad de la unidad liberada.

Una de las definiciones más importantes a realizar es cuándo se van a diseñar y ejecutar estas pruebas. Aquí es donde se define si la organización va a realizar code-fix o no. Una mala idea es depender de las pruebas unitarias; diversos estudios empíricos indican que la efectividad de la misma es de aproximadamente 50%. Esto quiere decir que si detectamos 4 defectos durante estas pruebas (y los corregimos), la unidad tendrá aproximadamente otros 4 defectos más al ser liberada.

Por suerte la forma de no depender de estas pruebas, y sacar el mayor provecho de las mismas, es muy sencilla: revisar antes de probar. El momento de ejecutar las pruebas unitarias es siempre luego de estar seguros que lo que se ha construido es una unidad de calidad. Para esto es necesario diseñar la unidad, revisar el diseño, codificar la unidad y revisar el código de forma estática. Luego, las pruebas unitarias serán una validación de que lo que se ha construido es una unidad de calidad, detectando y corrigiendo muy pocos defectos durante las mismas. Lamentablemente, la dependencia en las pruebas unitarias es un error común en la industria de desarrollo de software.

Driver

Un driver es una pieza de software creada para simular la invocación a la unidad que se desea probar. Es quién suministra los datos de los casos de prueba a la unidad de software que está siendo probada y quién realiza el chequeo de los resultados obtenidos contra los resultados esperados.
JUnit y TestNG son dos de los drivers más utilizados en el lenguaje Java.

JUnit y TestNG

JUnit es un framework, creado por Erich Gamma y Kent Beck, utilizado para automatizar las pruebas unitarias en el lenguaje Java. Las pruebas en JUnit se realizan por medio de casos de prueba escritos en clases Java.
Una de las grandes utilidades de JUnit es que los casos de prueba quedan definidos para ser ejecutados en cualquier momento, por lo cual es sencillo, luego de realizar modificaciones al programa, comprobar que los cambios no introdujeron nuevos errores. Esto último es conocido como pruebas de regresión.
TestNG es un framework creado por el Cédric Beust también para probar software escrito en el lenguaje Java. El objetivo del mismo es cubrir con una amplia gama de necesidades existentes en los diferentes tipos de pruebas, desde las unitarias hasta las de integración.

Esta herramienta fue creada basándose en JUnit y NUnit (para .NET), pero introduciendo nuevas funcionalidades que la hacen más poderosa y fácil de usar.
Entre las funcionalidades de la herramienta se encuentran: anotaciones JDK 5, configuración flexible de pruebas, soporte para pruebas utilizando el enfoque data-driven, soporte de pasaje de parámetros y permite definir paralelismo a nivel de pruebas.
La Figura 1 y la Figura 2 presentan cómo se generan casos de prueba en JUnit y en TestNG respectivamente; notarán la inmensa similitud en la forma de codificar la prueba. El método bajo prueba es el método merge. Este método recibe dos array de enteros (cada uno ordenado de menor a mayor) y devuelve un array que es la unión ordenada de los elementos de ambos arrays. La firma del método es la siguiente:

```
public int[] merge( int[ ] a, int [ ] b)
```


```
@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);
}
```

Le indica al test runner que debe ejecutar el test

Datos de la prueba

Ejecución

Resultado esperado

Comprobación del resultado esperado con el obtenido

Figura 1 – Ejemplo en 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);
}
```

Indica que el método es parte de una prueba

Datos de la prueba

Ejecución

Resultado esperado

Comprobación del resultado esperado con el obtenido

Figura 2 – Ejemplo en TestNG

Tanto JUnit como TestNG tienen una gran importancia ya que son usados como marco por muchas otras herramientas de soporte a las pruebas unitarias; por ejemplo, herramientas de cubrimiento, de mutantes y de data-driven.
Si bien ambas cumplen con la funcionalidad de un driver, TestNG agrega al menos tres funcionalidades que no están presentes en JUnit.
Ellas son:
- Permitir pasar Objetos de Java cómo parámetros de los casos de prueba a través del soporte a las pruebas que utilizan el enfoque data-driven.
- Permite definir grupos de prueba, lo cual brinda una gran flexibilidad a la hora de ejecutar las pruebas.
- Permite definir dependencia entre pruebas. Es posible indicar qué pruebas no tienen sentido que ejecuten cuando otras fallan.

En sucesivos artículos presentaremos herramientas de cubrimientos de código y soporte a data-driven.

Adriana Ávila
Lucía Camilloni
Fernando Marotta
Diego Vallespir
Cecilia Apa

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