Pruebas unitarias

José A. Mañas

11.2.2016

1 Introducción

Se dice que un programa es aceptable cuando:

- hace lo que debe hacer
- no hace lo que no debe hacer

Un programador jamás debería entregar un programa sin haberlo probado. Quien recibe un programa de otro, jamás debería aceptarlo sin haberlo probado.

A partir de la especificación de lo que se espera de un programa, se prepara una batería de casos de prueba. La batería que prepara el programador le permite saber cuándo ha terminado su trabajo. La batería que prepara quien va a recibir el programa le permite saber si puede aceptarlo o no.

Los casos de prueba se pueden escribir en papel y ejecutarlos disciplinadamente de forma manual; pero esta estrategia sólo es viable para programas pequeños. En programación profesional conviene automatizar las pruebas de forma que se pueden realizar muchas veces, tanto durante el desarrollo (hasta estar satisfechos), como durante la aceptación (para dar el programa por correcto, como si en el futuro hay que modificar el programa (para cerciorarnos de que no hemos roto nada que antes funcionaba).

2 JUnit

JUnit es un paquete Java para automatizar las pruebas de clases Java.

Se puede descargar de
http://junit.org/

aunque debe decirse que viene ya instalado en los entornos de desarrollo habituales.

Usaremos la versión 4.

Los ejemplos de este documento se han realizado con junit 4.12.

Una batería de pruebas es una clase java.

```
import static org.junit.Assert.*;
import org.junit.Test;

public class Ejemplo {
	@Test
	public void test() {
		// prueba
	}
}
```

La clase puede incluir sus variables privadas y el constructor correspondiente.

Dentro de esa clase preparamos casos de prueba, que son métodos java:

los casos de prueba son métodos que • no devuelven nada: void • están etiquetados como @Test • no tienen argumentos () • contienen aserciones

@Test
public void pruebaSuma() {
assertEquals(4, 2+2);
assertEquals(2, 2+0);
assertEquals(2, 0+2);
}

3 Aserciones para probar

JUnit incluye una serie de métodos para probar que las cosas son como esperamos. Aunque se remite al lector a la documentación detallada del paquete, las siguientes funciones son básicas:

assertEquals (X esperado, X real)	compara un resultado esperado con el resultado obtenido, determinando que la prueba pasa si son iguales, y que la prueba falla si son diferentes. Usa el método equals(). Realmente el método es una colección de métodos para una amplia variedad de tipos X.
assertSame(X esperado, X real)	Ídem; pero usa == para determinar si es el objeto esperado.
assertFalse (boolean resultado)	verifica que el resultado es FALSE
assertTrue (boolean resultado)	verifica que el resultado es TRUE
assertNull (Object resultado)	verifica que el resultado es "null"
assertNotNull (Object resultado)	verifica que el resultado no es "null"
fail	sirve para detectar que estamos en un sitio del programa donde NO deberíamos estar

Consulte la documentación: http://junit.org/javadoc/latest/org/junit/Assert.html

4 Excepciones

Los métodos pueden lanzar excepciones. Lanzar una excepción no es una obligación, sino una opción del método. Si tenemos un método que en ciertas condiciones lanza una excepción, debemos probar

1. que la lanza cuando debe lanzarla
2. que no la lanza cuando no debe lanzarla

A modo de ejemplo sea esta clase con un método que lanza Exception si y sólo si el argumento es null o está vacío:

```
public class Paciente {
    public String inverso(String s)
            throws Exception {
        if (s == null || s.length() == 0)
            throw new Exception("argumento incorrecto");
        StringBuilder builder = new StringBuilder();
        for (int i = s.length() - 1; i >= 0; i--)
            builder.append(s.charAt(i));
        return builder.toString();
    }
}
```

Veamos un par de pruebas bien escritas y un par mal escritas.

El marco de pruebas es así

```
import org.junit.Before;
import org.junit.Test;

import static org.junit.Assert.assertEquals;

public class Tester {
    private Paciente paciente;

    @Before
    public void preparacion() {
        paciente = new Paciente();
    }

}
```

Y en este marco escribimos un par de pruebas para testar el lanzamiento de la interrupción.

pruebas bien escritas	

cuando
no debe saltar
	`@Test`
	`public void testBienEscrito1() throws Exception {`
	`  assertEquals("cba", paciente.inverso("abc"));`
	`}`

cuando
sí debe saltar
	`@Test(expected = Exception.class)`
	`public void testBienEscrito2() throws Exception {`
	`  assertEquals(null, paciente.inverso(null));`
	`}`

pruebas mal escritas	

cuando
no debe saltar
	`@Test(expected = Exception.class)`
	`public void testMalEscrito1() throws Exception {`
	`  assertEquals("cba", paciente.inverso("abc"));`
	`}`

cuando
sí debe saltar
	`@Test`
	`public void testMalEscrito2() throws Exception {`
	`  assertEquals(null, paciente.inverso(null));`
	`}`

4.1 Alternativa: fail()

Como alternativa a la anotación que indica que el cuerpo de la prueba debe lanzar una excepción, podemos hacer un tratamiento más fino como el siguiente

```
@Test
public void testBienEscrito3() {
   try {
      assertEquals(null, paciente.inverso(null));
      fail("deberia haber lanzado una excepcion");
   } catch (Exception e) {
      // funciona correctamente
   }
}
```

Esta forma de escribir el test permite analizar más detalle del comportamiento del código bajo pruebas; por ejemplo:

- cuando queremos un mensaje informativo particularizado
- cuando pueden lazarse varias excepciones
- cuando puede lanzarse la misma excepción en 2 o más sitios

- cuando se lanza una excepción con información adicional (campos internos) que debemos analizar para validar si están cargados como se espera; en este caso, pondríamos assert* dentro del catch

5 Métodos auxiliares

```
import static org.junit.Assert.*;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import java.util.*;

public class SetMiniTest {
   private Set<Integer> s;

   @Before
   public void setUp() {
      s = new HashSet<Integer>();
   }

   @After
   public void tearDown() {
      s = null;
   }

   @Test
   public void test0() {
      assertTrue(s.isEmpty());
   }

   @Test
   public void test1() {
      assertEquals(0, s.size());
   }

   @Test
   public void test2() {
      s.add(1);
      assertEquals(1, s.size());
   }
}
```

5.1 @Before

JUnit lo llamará antes de lanzar cada uno de los casos de prueba.

Puede ser útil cuando todos los casos de prueba requieren la misma inicialización de variables privadas y no basta con hacerlo en el constructor.

5.2 @After

JUnit lo llamará después de lanzar cada uno de los casos de prueba.

Puede ser útil para cerrar elementos abiertos durante la prueba que pudieran quedar en un estado lamentable. Por ejemplo:

- ficheros
- conexiones Internet
- conexiones a bases de datos
- ...

5.3 @Ignore

Es útil para excluir un test de la batería y que no se ejecute.

Viene a ser lo mismo que comentar el código.

5.4 @Test(timeout=100)

JUnit activa un cronómetro. Si pasados 100ms no ha terminado la prueba, suena una alarma y falla la prueba.

6 Diseño de pruebas

Diseñar pruebas es un arte que ha sido objeto de estudio durante años.

No obstante, las reglas básicas son sencillas:

- hay que prever al menos una prueba para cada función del sistema
- hay que elegir al menos un caso "normal" de datos
- cuando los datos presentan singularidades (límites o cambio de funcionamiento por rango) hay que probar los datos anexos a cada límite

Así, para probar una función que divide dos números, probaremos los casos alrededor del valor singular "0":

- numerador mayor que cero, denominador mayor que cero
- numerador mayor que cero, denominador menor que cero
- numerador mayor que cero, denominador igual a cero
- numerador menor que cero, denominador mayor que cero
- numerador menor que cero, denominador menor que cero
- numerador menor que cero, denominador igual a cero
- numerador igual a cero, denominador mayor que cero
- numerador igual a cero, denominador menor que cero
- numerador igual a cero, denominador igual a cero

6.1 Pruebas de caja negra

Se dice que las pruebas son de caja negra cuando ignoramos cómo está programada la función bajo pruebas.

Lo único que podemos hacer es tirar de especificación para identificar qué son casos singulares y casos límite. Y podemos añadir sospechas que tengamos.

Lo bueno de las pruebas de caja negra es que prueban lo que necesitamos, sin preocuparse de cómo lo hacemos. Son ideales para desarrollo incremental en donde reemplazamos código validando que se siguen pasando los tests.

Lo malo de las pruebas de caja negra es que no miran el código y pueden haber singularidades que no probamos de antemano y pueden ocurrir en ejecución real.

6.2 Pruebas de caja blanca

Se dice que las pruebas son de caja blanca cuando tenemos acceso al código de la función bajo pruebas.

En este caso, además de las pruebas oportunas de caja negra, podemos ver la cobertura sobre el código:

- si hemos ejecutado todas las sentencias al menos una vez
- si hemos probado todas las condiciones al menos 2 veces (cierto y falso)
- si hemos probado todos los bucles:
    - while: 0, 1 y más de 1 vez
    - until: 1, 2 y más de 2 veces
    - for: 0, 1 y más de 1 vez

Lo bueno de las pruebas de caja blanca es que probamos lo que hemos codificado y no pueden quedar singularidades que pudieran ocurrir en ejecución real sin haberse probado antes.

Lo malo de las pruebas de caja blanca es que no probamos lo que necesitamos y podemos acabar con un código que ejecuta perfectamente algo que no es lo que necesitamos.

Las pruebas de caja blanca son poco útiles para un desarrollo incremental usando pruebas como validación continua.

6.3 Pruebas aleatorias

A veces puede que no estemos muy seguros de haber probado suficientes casos. Esto ocurre cuando lo que probamos es una caja negra y la funcionalidad no es simple ni el algoritmo utilizado internamente es evidente.

En estos casos las pruebas no son definitivas pues podemos no haber probado cierta circunstancia que en la realidad provoca un error.

Supongamos que la probabilidad de que una función falle es p.

La probabilidad de que una prueba pase sin detectar el fallo es (1-p).

La probabilidad de que n pruebas pasen sin detectar el fallo es $(1-p)^n$.

El número de pruebas que tenemos que pasar para que la probabilidad de que haya un fallo y no lo hayamos encontrado sea menor que x se calcula como

$(1-p)^n < x \rightarrow n > \log(x)/\log(1-p)$

Algunos números:

p = 1% = 0.1

n= 1 → $(1-p)^n = 0.9 = 90\%$

n= 2 → $(1-p)^n = 0.81 = 81\%$

n= 5 → $(1-p)^n = 0.59 = 59\%$

x = 10% = 0.1 → n > 21

x = 1% = 0.01 → n > 43

En palabras, con un número razonable de pruebas aleatorias podemos obtener bajas probabilidades de que los errores no se detecten.

En la práctica, las pruebas deberían orientar su aleatoriedad hacia las zonas más sospechosas de fallo o, simplemente, hacia escenarios conflictivos. Lo que no deja de ser un arte.

Las pruebas aleatorias tienen de bueno que prueban muchas cosas pensando poco. Todo depende de que la aleatoriedad sea coherente con lo que nos encontremos después en la realidad.

Un problema de las pruebas aleatorias es que podemos encontrarnos con un error que no seamos capaces de reproducir. Para ello, o bien convertimos la aleatoriedad en pseudo-aleatoriedad (usando series previsibles de datos que parecen aleatorios) o registramos sistemáticamente el caso de prueba y el resultado para poder reproducirlo en el futuro.

Si para generar datos aleatorios se usa la clase Random de la biblioteca de java, pueden generarse datos impredecibles haciendo

```
Random random = new Random();
```

o se puede generar siempre la misma secuencia imponiendo una semilla inicial

```
long semilla = 0; // cualquier valor, lo importante es que sea fijo
Random random = new Random(semilla);
```

7 Inglés

Se incluye un sucinto glosario de terminología inglesa relacionado con la prueba de programas.

acierto	success
batería de pruebas	test suite
caso de prueba	test case
fallo	failure
probar	to test
prueba	test
pruebas de aceptación	acceptance test suite
repetición de pruebas	regression testing

8 Ejemplo: conjuntos

Se adjunta una batería de pruebas para comprobar que funciona correctamente la clase java.util.HashSet como implementación de la clase java.util.Set.

La batería de pruebas está organizada en varios casos de prueba cada uno centrado en una función de las proporcionadas por la clase. Y en cada caso de prueba se buscan los casos singulares

- conjunto vacío frente a conjunto con elementos
- miembros que ya pertenecen al conjunto frente a miembros que no pertenecen al conjunto

8.1 SetTest

```
import static org.junit.Assert.*;

import java.util.ConcurrentModificationException;
import java.util.HashSet;
import java.util.Set;

import org.junit.Test;

public class SetTest {
	private String e1 = "elemento 1";
	private String e2 = "elemento 2";
	private String e3 = "elemento 3";
	private String e4 = "elemento 4";
	private String e5 = "elemento 5";

	public SetTest() {
		// constructor
		// útil si hay que inicializar variables privadas
	}

	@Test
	public void testAdd() {
		Set<String> set = new HashSet<String>();
		assertEquals(0, set.size());
		assertTrue(set.add(e1));
		assertEquals(1, set.size());
		assertTrue(set.add(e2));
		assertEquals(2, set.size());
		assertFalse(set.add(e1));
		assertEquals(2, set.size());
	}

	@Test
	public void testRemove() {
		Set<String> set = new HashSet<String>();
		set.add(e1);
		set.add(e2);
		set.add(e3);
		assertEquals(3, set.size());
		assertTrue(set.remove(e2));
		assertEquals(2, set.size());
		assertFalse(set.remove(e2));
		assertEquals(2, set.size());
		assertTrue(set.remove(e1));
```

```
        assertEquals(1, set.size());
        assertTrue(set.remove(e3));
        assertEquals(0, set.size());
    }

    @Test
    public void testClear() {
        Set<String> set = new HashSet<String>();
        set.add(e1);
        set.add(e2);
        set.add(e3);
        assertEquals(3, set.size());
        set.clear();
        assertEquals(0, set.size());
    }

    @Test
    public void testIsEmpty() {
        Set<String> set = new HashSet<String>();
        assertTrue(set.isEmpty());
        set.add(e1);
        set.add(e2);
        set.add(e3);
        assertFalse(set.isEmpty());
        set.remove(e3);
        assertFalse(set.isEmpty());
        set.clear();
        assertTrue(set.isEmpty());
    }

    @Test
    public void testContains() {
        Set<String> set = new HashSet<String>();
        set.add(e1);
        set.add(e2);
        set.add(e3);
        assertTrue(set.contains(e1));
        assertTrue(set.contains(e2));
        assertTrue(set.contains(e3));
        assertFalse(set.contains(e4));
        set.add(e2);
        assertTrue(set.contains(e1));
        assertTrue(set.contains(e2));
        assertTrue(set.contains(e3));
        assertFalse(set.contains(e4));
    }

    @Test
    public void testEquals() {
        Set<String> set1 = new HashSet<String>();
        set1.add(e1);
        set1.add(e2);
        set1.add(e3);
        Set<String> set2 = new HashSet<String>();
        assertFalse(set1.equals(set2));
        set2.add(e1);
        assertFalse(set1.equals(set2));
        set2.add(e2);
        assertFalse(set1.equals(set2));
```

```
		set2.add(e3);
		assertTrue(set1.equals(set2));
		assertTrue(set2.equals(set1));
		set2.add(e4);
		assertFalse(set1.equals(set2));
		assertFalse(set2.equals(set1));
	}

	@Test
	public void testIteration() {
		Set<String> set1 = new HashSet<String>();
		set1.add(e1);
		set1.add(e2);
		set1.add(e3);
		Set<String> set2 = new HashSet<String>();
		for (String e : set1)
			set2.add(e);
		assertEquals(set1, set2);
	}

	@Test(expected = ConcurrentModificationException.class)
	public void testConcurrentModification() {
		Set<String> set = new HashSet<String>();
		set.add(e1);
		set.add(e2);
		set.add(e3);
		for (String e : set)
			set.remove(e);
		assertTrue(set.isEmpty());
	}

}
```