Capítulo 6

Pruebas unitarias con JUnit

Contenidos

Introducción

Llegados a este punto ya somos capaces de escribir aplicaciones Java utilizando los principios de la POO. Sabemos cómo definir clases, como utilizar la herencia o la composición para ampliar el comportamiento de nuestras clases. Incluso somos capaces de controlar las situaciones anómalas que pueden darse durante la ejecución de nuestras aplicaciones a través de la gestión de excepciones.

El siguiente paso que usualmente se suele dar es comprobar la validez del código realizando pruebas. A veces, estas pruebas son *caseras*, probamos unos cuantos valores de entrada en nuestra aplicación, valores de los que conocemos cual es la salida esperada, y confiamos que en el resto de casos nuestro código esté libre de errores. Y en este momento es cuando la confianza se convierte en engaño, nuestro código está plagado de errores que no hemos sido capaces de detectar y que tarde o temprano saldrán a la luz sumiéndonos en la oscuridad, paradojas de la vida.

La primera idea que debemos fijar es que hacer pruebas de código no debe ser una opción, es un requerimiento, por defecto, en todo desarrollo de proyectos informáticos.

La segunda idea que debemos fijar es que las pruebas de código no deben ser manuales, si no automatizadas. Si son manuales, por aburrimiento o falta de tiempo acabaremos por no hacerlas. Las pruebas automatizadas forman parte del código del proyecto, son tan importantes como el código que se está probando y por lo tanto debemos dedicarles el mismo empeño que al desarrollo del código de nuestra aplicación.

En este capítulo no se va a mostrar cómo diseñar buenas pruebas de código, y lo que es más importante, no se va a mostrar cómo escribir código que se pueda probar fácilmente, en la sección de referencias de este capítulo se dan algunos títulos que son de lectura obligada a todo desarrollador que quiera poner en práctica la prueba de código en sus proyectos.

En este capítulo se va a mostrar cómo utilizar una herramienta para comprobar código. Esta excelente herramienta es JUnit.

6.1. ¿Qué son las pruebas unitarias?

Las pruebas unitarias se realizan sobre una clase, para probar su comportamiento de modo aislado, independientemente del resto de clases de la aplicación. Este requisito a veces no se cumple, piensa en el código de una clase que accede a una base de datos, y que la prueba de la clase se base en el resultado que se recupera de la base de datos, resulta imposible comprobar esta clase de modo aislado, aunque existen técnicas (como los *Mock Objects*) que minimizan estas dependencias.

6.1.1. Principios FIRST para el diseño de pruebas unitarias

Cuando se diseñan pruebas unitarias es importante seguir los principios *FIRST*. Cada una de las letras de esta palabra inglesa está relacionada con un concepto. Veámoslos:

Fast : La ejecución del código de pruebas debe ser rápida. Si las pruebas consumen demasiado tiempo acabaremos por no hacerlas.

Independent : Una prueba no puede depender de otras. Cada prueba debe ser unitaria, debe poder realizarse de modo aislado.

Repetable : Las pruebas se deben poder repetir en cualquier momento y la cantidad de veces que sea necesario. El resultado de una prueba debe ser siempre el mismo.

Self-validating : Sólo hay dos posibles resultados de una prueba: «La prueba pasó con éxito» o «La prueba falló».

Timely : Las pruebas han de escribirse en el momento de escribir el código, y no al final de toda la fase de desarrollo [1]

[1] La metodología de desarrollo *Test Driven Development (TDD)* lleva este principio al inicio del proceso de desarrollo de tal modo que las pruebas de código se escriben antes que el propio código que se intenta probar.

6.2. Pruebas unitarias con JUnit

JUnit es una herramienta para realizar pruebas unitarias automatizadas. JUnit está integrada en *Eclipse*, no es necesario descargarse ningún paquete ni instalar un nuevo plug-in para utilizarla. *Eclipse* facilita la creación de pruebas unitarias.

Para mostrar con un ejemplo cómo se escriben pruebas unitarias de código con JUnit vamos a utilizar las clases `ConversorTemperaturas` y `TemperaturaNoValidaException` que vimos en el capítulo anterior.

6.2.1. Creación de clases de prueba

Para crear una clase de prueba en *Eclipse* seleccionamos *File* → *New* → *Other...*, en la ventana de diálogo que se abrirá seleccionamos *Java* → *JUnit* → *JUnit Test Case*, se abrirá una nueva ventana de diálogo, en la parte superior seleccionamos *New JUnit 4 test*, introducimos el nombre para la clase de prueba y su paquete, y por comodidad, en la parte inferior de esta ventana pulsamos *Browse* y seleccionamos la clase que deseamos probar, que en nuestro caso es `ConversorTemperaturas`, y pulsamos *Next*, en la nueva ventana veremos que se nos ofrece la posibilidad de seleccionar los métodos que queremos probar, en nuestro caso vamos a seleccionar `celsiusAFharenheit(double)` y *celsiusAReamur(double)* y pulsamos *Finish*. La clase de prueba que se creará automáticamente será la que se muestra en el Listado 6.1.

Cabe destacar varios puntos de este Listado:

1. Fíjate en el uso del `import static` de la línea 3, es útil para no tener que incluir el nombre de la clase `Assert` cuando utilizamos alguno de sus métodos estáticos, como `fail`. Un `import static` me permite utilizar todos los métodos `static` de una clase sin necesidad de anteponer al método el nombre de la clase [2].

2. Observa que se han creado dos métodos de prueba, una para cada método que seleccionamos sobre la clase a probar, y que la signatura de ambos es **public void** nombreDelMetodo(). Los métodos de prueba deben ser públicos no retornar ningún valor y su lista de argumentos debe estar vacía.

3. Fíjate que sobre cada uno de los métodos de prueba aparece la anotación `@Test` que indica al compilador que es un método de prueba.

4. Por defecto, cada uno de los métodos de prueba tiene una llamada a `fail("Mesaje con descripción.")`.

```
1 package test;
2
3 import static org.junit.Assert.*;
4
5 import org.junit.Test;
6
7 public class TestConversorTemperaturas {
```

[2] Los `static import` se introdujeron en la versión 5 de Java, y aunque son cómodos de utilizar, el uso de JUnit es un caso, pueden provocar confusión en el programador, ya que al no aparecer el nombre de la clase tendremos la duda de si el método pertenece a la clase actual o a un clase de la que se ha hecho un `static import`. En general, el uso de los `static import` está desaconsejado.

```
8
9  @Test
10 public final void testCelsiusAFherenheit() {
11  fail("Not yet implemented");
12 }
13
14 @Test
15 public final void testCelsiusAReamur() {
16  fail("Not yet implemented");
17 }
18
19 }
```

Listado 6.1: Código generado automáticamente por *Eclipse* para una clase de prueba.

6.2.2. La anotación `@Test`

Como ya se ha dicho, la anotación `@Test` sirve para indicar que un determinado método es un método de prueba. Vamos a escribir el primer código de prueba tal y como se muestra en el Listado 6.2. Fíjate que también hemos añadido **throws** TemperaturaNoValidaException para indicar que no queremos gestionar esta posible excepción en el código del método de prueba.

```
1  @Test
2  public void testCelsiusAFharenheit() throws
       TemperaturaNoValidaException {
3   ConversorTemperaturas conversor = new ConversorTemperaturas();
4   assertEquals(32, conversor.celsiusAFharenheit(0), 0);
5  }
```

Listado 6.2: Un método de prueba.

Lo primero que hacemos en el método de prueba es crear una instancia de la clase `ConversorTemperaturas`, y después utilizar el método assertEquals(valorEsperado, valorObtenido, error). Este método comprueba que la diferencia entre el `valorEsperado` y el `valorObtenido` es menor que `error`. Si es así, se ha pasado la prueba, de lo contrario la prueba falla. En nuestro caso, se está aseverando que la diferencia entre el valor que devuelve el método `celsiusAFharenheit(0)` y el valor `32` es cero. Escribamos el código para la segunda de las pruebas tal y como se muestra en el Listado 6.3.

```
1  @Test
2  public void testCelsiusAReamur() throws TemperaturaNoValidaException {
3   ConversorTemperaturas conversor = new ConversorTemperaturas();
4   assertEquals(0, conversor.celsiusAReamur(0), 0);
5  }
```

Listado 6.3: Un segundo método de prueba.

De modo análogo al primer método de prueba, en este caso estamos aseverando que la diferencia entre el valor que devuelve la llamada al método `celsiusAReamur(0)` y el valor `0` es cero.

Para ejecutar las pruebas desde *Eclipse* pulsa el botón derecho del ratón sobre la clase de prueba y en el menú emergente selecciona la opción *Run As* → *JUnit Test*, verás que se abre una nueva vista con el resultado de la ejecución de las pruebas, que en nuestro caso es *Runs: 2/2 Errors: 0 Failures: 0* que nos

indica que se han realizado 2 pruebas, ninguna de ellas a provocado un error y ninguna de ellas a provocado un fallo.

¿Cual es la diferencia entre un fallo y un error en el contexto de las pruebas unitarias con JUnit?. Un fallo es una aseveración que no se cumple, un error es una excepción durante la ejecución del código. Generemos un fallo de modo artificial para ver el resultado, cambiemos la línea `assertEquals(32, conversor.celsiusAFharenheit(0), 0);` por esta otra `assertEquals(0, conversor.celsiusAFharenheit(0), 0);`, y ejecutemos de nuevo la prueba, en este caso obtendremos un fallo que nos informará que el valor esperado de la prueba era `0` mientras que el valor obtenido es `32.0`.

Añadamos otro método de prueba que genere un error, para ver la diferencia con un fallo, tal y como se muestra en el Listado 6.4.

```
1 public void testTemperaturaNoValidaFharenheit() throws
      TemperaturaNoValidaException {
2  ConversorTemperaturas conversor = new ConversorTemperaturas();
3  conversor.celsiusAFharenheit(-400);
4 }
```

Listado 6.4: Un método de prueba que genera un error.

Al ejecutar de nuevo las pruebas esta vez obtendremos la excepción *La temperatura no puede ser menor que -273ºC*. ¿Y si lo que queremos es precisamente comprobar que se lanza la excepción?, es decir, ¿Y si nuestra prueba pasa precisamente si se genera la excepción? Para ello basta con añadir el atributo `expected=TemperaturaNoValidaException.class` a la anotación `@Test` quedando de este modo `@Test(expected=TemperaturaNoValidaException.class)`. Si ejecutamos de nuevo las pruebas veremos que todas ellas pasan.

Otra técnica que no utiliza el atributo `expected` de la anotación `@Test` para comprobar que se produce una excepción es la mostrada en el Listado 6.5. Esta vez el método está etiquetado únicamente con `@Test`, y detrás de la línea de código que esperamos que produzca la excepción escribimos `fail("Para temperaturas por encima de -273 la prueba debe pasar.")`. Si la excepción se produce al ejecutarse la línea 5, la ejecución de la prueba continuará en el bloque `catch (TemperaturaNoValidaException e)` y la prueba pasará, que es lo que esperamos.

Si no se produce la excepción en la línea 5, se ejecutará la sentencia `fail(...)` y la prueba no pasará, cosa que será indicativa de que algo ha ido mal ya lo que intentamos probar es que la excepción sí que se produce.

```
1 @Test
2 public void testTemperaturaNoValidadFahrenheit() {
3  ConversorTemperaturas conversor = new ConversorTemperaturas();
4  try {
5   conversor.celsiusAFharenheit(-400);
6   fail("Para temperaturas por encima de -273 la prueba debe pasar.");
7  } catch (TemperaturaNoValidaException e) {
8  }
9 }
```

Listado 6.5: Un método de prueba que genera un error.

Este segundo método de prueba de excepciones es el recomendado, ya que es más fácil interpretar qué es lo que se está intentando probar.

6.2.3. Las anotaciones `@Before` y `@After`

Si revisas el código de los tres métodos de prueba anteriores verás que lo primero que hacemos es crear una instancia de la clase `ConversorTemperaturas`. JUnit nos proporciona un mecanismo para extraer el código que se repite en todos los métodos, y que debe ejecutarse antes de cualquiera de ellos, a través de las anotaciones `@Before` y `@After`. Si anotamos un método con `@Before` su código será ejecutado antes de cada uno de los métodos de prueba, si tenemos tres métodos de prueba será ejecutado antes de cada uno de los tres métodos. Por su lado, si anotamos un método con `@After` será ejecutado después de la ejecución de cada uno de los métodos de prueba. Por lo tanto, podemos usar la anotación `@Before` para iniciar todas las infraestructuras necesarias a la ejecución de las pruebas y la anotación `@After` para limpiar estas infraestructuras.

En nuestro caso, la clase de prueba quedaría tal y como se muestra en el Listado 6.6.

```
1 import static org.junit.Assert.*;
2
3 import org.junit.After;
4 import org.junit.Before;
5 import org.junit.Test;
6
7 import conversor.ConversorTemperaturas;
8 import conversor.TemperaturaNoValidaException;
9
10 public class TestConversorTemperaturas2 {
11  private ConversorTemperaturas conversor;
12
13  @Before
14  public void creaConversorTemperaturas() {
15   conversor = new ConversorTemperaturas();
16  }
17
18  @After
19  public void destruyeCnversorTemperarturas() {
20   conversor = null;
21  }
22
23  @Test
24  public void testCelsiusAFharenheit() throws
        TemperaturaNoValidaException {
25   assertEquals(32, conversor.celsiusAFharenheit(0), 0);
26  }
27
28  @Test
29  public void testCelsiusAReamur() throws TemperaturaNoValidaException {
30   assertEquals(0, conversor.celsiusAReamur(0), 0);
31  }
32
33  @Test(expected=TemperaturaNoValidaException.class)
34  public void testTemperaturaNoValidaFharenheit() throws
        TemperaturaNoValidaException {
35   conversor.celsiusAFharenheit(-400);
36  }
37 }
```

Listado 6.6: Uso de las anotaciones `@Before` y `@After`.

Las anotaciones `@Before` y `@After` las puedes utilizar tantas veces como te sea necesario, puede haber más de un método anotado con alguna de estas anotaciones. Todos los métodos que estén anotados con `@Before` se ejecutarán antes de cada uno de los métodos de prueba y todos los métodos que estén anotados con `@After` se ejecutarán después de cada uno de los métodos de

prueba.

6.2.4. Las anotaciones `@BeforeClass` y `@AfterClass`

Podemos mejorar un poco más nuestra clase de prueba con el uso de dos nuevas etiquetas `@BeforeClass` y `@AfterClass`. Fíjate que la clase que estamos probando `ConversorTemperaturas` no tiene estado, y por lo tanto el resultado de las llamadas a sus métodos es independiente del orden en el que se hagan, por lo que no es necesario crear una instancia nueva antes de cada una de las pruebas, si no que la misma instancia nos sirve para las tres pruebas.

Si anotamos un método de una clase de prueba con `@BeforeClass` ese método se ejecutará una única vez antes de la ejecución de cualquier método de prueba. Por otro lado, si anotamos un método de una clase de prueba con `@AfterClass` el método será ejecutado una única vez después de haberse ejecutado todos los métodos de prueba, tal y como se muestra en el Listado 6.7.

```
1 import static org.junit.Assert.assertEquals;
2
3 import org.junit.AfterClass;
4 import org.junit.BeforeClass;
5 import org.junit.Test;
6
7 import conversor.ConversorTemperaturas;
8 import conversor.TemperaturaNoValidaException;
9
10 public class TestConversorTemperaturas3 {
11  private static ConversorTemperaturas conversor;
12
13  @BeforeClass
14  public static void creaConversorTemperaturas() {
15   conversor = new ConversorTemperaturas();
16  }
17
18  @AfterClass
19  public static void destruyeCnversorTemperarturas() {
20   conversor = null;
21  }
22
23  @Test
24  public void testCelsiusAFharenheit() throws
        TemperaturaNoValidaException {
25   assertEquals(32, conversor.celsiusAFharenheit(0), 0);
26  }
27
28  @Test
29  public void testCelsiusAReamur() throws TemperaturaNoValidaException {
30   assertEquals(0, conversor.celsiusAReamur(0), 0);
31  }
32
33  @Test(expected=TemperaturaNoValidaException.class)
34  public void testTemperaturaNoValidaFharenheit() throws
        TemperaturaNoValidaException {
35   conversor.celsiusAFharenheit(-400);
36  }
37 }
```

Listado 6.7: Uso de las anotaciones `@BeforeClass` y `@AfterClass`.

Fíjate en el importante detalle que aparece en el Listado 6.7, los métodos anotados con `@BeforeClass` y `@AfterClass` deben ser ambos `static` y por lo tanto, los atributos a los que acceden también deben ser `static`, tal y como vimos en 2.6.

6.2.5. Pruebas con batería de datos de entrada

Cada uno de los métodos de prueba de los ejemplos anteriores utiliza un trío de datos, valor esperado, valor real y error para comprobar cada uno de los casos de prueba. Si queremos escribir una nueva prueba para otro trío de valores es tedioso crear un método sólo para él. JUnit proporciona un mecanismo para probar baterías de valores en vez de únicamente tríos aislados.

Lo primero que debemos hacer es anotar la clase de prueba con `@RunWith(Parameterized.class)` indicando que va a ser utilizada para realizar baterías de pruebas. La clase de prueba ha de declarar un atributo por cada uno de los parámetros de la prueba, y un constructor con tantos argumentos como parámetros en cada prueba. Finalmente necesitamos definir un método que devuelva la colección de datos a probar anotado con `@Parameters` . De este modo, cada uno de los métodos de prueba será llamado para cada una de las tuplas de valores de prueba.

En el Listado 6.8 se muestra un ejemplo de clase de prueba para una batería de pruebas sobre la clase `ConversorTemperaturas`.

```
1 import static org.junit.Assert.*;
2
3 import java.util.Arrays;
4 import java.util.Collection;
5
6 import org.junit.AfterClass;
7 import org.junit.BeforeClass;
8 import org.junit.Test;
9 import org.junit.runner.RunWith;
10 import org.junit.runners.Parameterized;
11 import org.junit.runners.Parameterized.Parameters;
12
13 import conversor.ConversorTemperaturas;
14 import conversor.TemperaturaNoValidaException;
15
16 @RunWith(Parameterized.class)
17 public class TestConversorTemperaturas4 {
18  private double celsius;
19  private double fharenheit;
20  private double reamur;
21  private double error;
22  private static ConversorTemperaturas conversor;
23
24  public TestConversorTemperaturas4(double celsius, double fharenheit,
        double reamur, double error) {
25   this.celsius = celsius;
26   this.fharenheit = fharenheit;
27   this.reamur = reamur;
28   this.error = error;
29  }
30
31  @Parameters
32  public static Collection<Object[]> datos() {
33   return Arrays.asList(new Object[][]{
34     {0.0, 32.0, 0.0, 0.0}, // {celsius, fharenheit, reamur, error}
35     {15, 59.0, 12.0, 0.0},
36     {30, 86.0, 24.0, 0.0},
37     {50, 122.0, 40.0, 0.0},
38     {90, 194.0, 72.0, 0.0}
39   });
40  }
41
42  @BeforeClass
43  public static void iniciaConversor() {
44   conversor = new ConversorTemperaturas();
45  }
46
47  @AfterClass
```

```
48  public static void eliminaConversor() {
49   conversor = null;
50  }
51
52  @Test
53  public void testCelsiusAFharenheit() throws
        TemperaturaNoValidaException {
54   assertEquals(fharenheit, conversor.celsiusAFharenheit(celsius), error)
         ;
55  }
56
57  @Test
58  public void testCelsiusAReamur() throws TemperaturaNoValidaException {
59   assertEquals(reamur, conversor.celsiusAReamur(celsius), error);
60  }
61 }
```

Listado 6.8: Ejemplo de definición de una clase que realiza una batería de pruebas.

De modo resumido, estos son los pasos para definir una clase que ejecuta baterías de pruebas:

1. Anotar la clase de prueba con `@RunWith(Parameterized.class)`.
2. Declarar un atributo en la clase por cada parámetro de prueba.
3. Definir un constructor con tantos argumentos como parámetros de prueba.
4. Definir un método que devuelva una colección con todas las tuplas de prueba, y anotarlo con `Parameters`.

Internamente y de modo esquemático, lo que JUnit hace en el caso de las baterías de pruebas es crear una instancia de la clase de prueba a partir del constructor con tantos argumentos como parámetros en cada una de las tuplas de prueba, y seguidamente llama a cada uno de los métodos de prueba.

6.2.6. Ejecutar varias clases de prueba. Test Suites

Lo común, como hemos visto, es tener varias clases de prueba ya que a veces no tiene sentido una única clase donde se realicen todas las pruebas. Piensa por ejemplo en las pruebas parametrizadas, quizás tengas algún caso de prueba sobre el que no tenga sentido realizar pruebas parametrizadas, como por ejemplo comprobar que se produce una excepción.

Por lo tanto, si tenemos varias clases de prueba, ¿Cómo podemos ejecutar todas las pruebas, o al menos algunas de ellas sin tener que ejecutarla cada clase de modo independiente? Para ello existe un mecanismo en JUnit llamado *Test Suites*, que no son más que agrupaciones de clases de prueba que se ejecutan una tras otra.

Básicamente lo que hacemos es anotar una clase para que JUnit la reconozca como una suite de pruebas y con otra anotación añadimos a esta clase todas las clases de prueba que nos interese ejecutar, tal y como se muestra en el Listado 6.9.

```
1 import org.junit.runner.RunWith;
2 import org.junit.runners.Suite;
3 import org.junit.runners.Suite.SuiteClasses;
4
```

```
5 @RunWith(Suite.class)
6 @SuiteClasses({
7  TestConversorTemperaturas.class,
8  TestConversorTemperaturas2.class,
9  TestConversorTemperaturas3.class,
10  TestConversorTemperaturas4.class
11 })
12 public class AllTests {
13 }
```

Listado 6.9: Ejemplo de una suite de pruebas *Test Suite*.

De este modo bien podemos ejecutar una única clase de prueba para probar el funcionamiento correcto de una clase en particular de todas las que forman nuestro proyecto, o bien podemos ejecutar toda la *suite* de clases de prueba para probar todo nuestro proyecto.

6.3. Cobertura de las pruebas

Una duda que nos surge al realizar pruebas unitarias es la cantidad de líneas de código que han cubierto las pruebas, ¿Ha quedado algún fragmento de código que no se ha ejecutado ni una sola vez para todas las pruebas? ¿Cómo podemos saberlo?.

Lo ideal es que nuestras pruebas cubran el 100 % del código de la clase que estamos probando. Pero no caigas en el engaño de pensar que por cubrir con pruebas el 100 % del código estás cubriendo todos los casos posibles de la ejecución de ese código, ni mucho menos. Puedes cubrir el 100 % de la ejecución del código con casos triviales que nunca fallarán y no sacarán a la luz los posibles errores de tu código.

Para que veas con claridad la diferencia entre cobertura de código y pruebas exhaustivas el Listado 6.10 te muestra un método a probar y un par de métodos de prueba. Los métodos de prueba cubren el 100 % del código del método que se está probando pero, ¿Qué pasa si alguna de las referencias que se pasan al método de prueba es `null`? Evidentemente el método que se está probando contiene un error que no será descubierto por las pruebas aunque estas estén cubriendo el 100 % del código.

```
1 // Método que se va a probar
2 public int quienEsMayor(Persona primera, Persona segunda) {
3  if(primera.edad > segunda.edad) return -1;
4  if(primera.edad < segunda.edad) return 1;
5  else return 0;
6 }
7
8 // Tres métodos de prueba
9 @Test
10 public void masViejoElPrimero() {
11  Persona primera = new Persona();
12  primera.setEdad(100);
13  Persona segunda = new Persona();
14  segunda.setEdad(50);
15  assertEquals(-1, quienEsMayor(primera, segunda), 0);
16 }
17
18 @Test
19 public void masViejoElSegundo() {
20  Persona primera = new Persona();
21  primera.setEdad(50);
22  Persona segunda = new Persona();
23  segunda.setEdad(100);
```

```
24  assertEquals(1, quienEsMayor(primera, segunda), 0);
25 }
26
27 @Test
28 public void mismaEdad() {
29  Persona primera = new Persona();
30  primera.setEdad(50);
31  Persona segunda = new Persona();
32  segunda.setEdad(50);
33  assertEquals(0, quienEsMayor(primera, segunda), 0);
34 }
```

Listado 6.10: Los tres métodos cubren el 100 % del código del método que se está probando, pero este método contiene un error ya que no se comprueba que las referencias sean distintas de `null`

Existe modelos teóricos que dan las pautas a seguir para garantizar que las pruebas son exhaustivas, de modo que se contemplan todos los posibles casos de fallo de nuestro código. La referencia [5] presenta de modo exhaustivo las pruebas de código que se han de realizar para garantizar que se cubre el 100 % de los posibles caminos de ejecución.

6.3.1. EclEmma y su plug-in para Eclipse

Afortunadamente existen excelentes herramientas que miden, de modo automático, la cobertura de nuestras pruebas unitarias. Una de esas herramientas, aunque no la única es *Ecl-Emma* de la que existe un *plug-in* para *Eclipse* y cuya página web es `http://www.eclemma.org/`. Para instalar este *plug-in* basta seguir los mismo pasos que se mostraron en la Sección 4.5, pero siendo la dirección del *plug-in* la que se indica en la página web de la herramienta.

Una vez instalado este *plugin* te aparecerá un nuevo botón en *Eclipse* a la izquierda del botón ejecutar. Si pulsas este botón cuando está seleccionada una clase de prueba, se abrirá una nueva vista de nombre *Coverage* en la que se te mostrará todos los resultados de la cobertura de la prueba, y lo que es de gran ayuda, de cada una de las líneas de código de la clase que se está probando se coloreará su fondo en *verde*, si la línea ha sido cubierta completamente por la prueba; *amarilla*, si ha sido cubierta sólo parcialmente; o *roja*, si no ha sido cubierta.

En la vista *Coverage* se muestra para cada clase probada, una tabla con el porcentaje de líneas de código cubiertas por la prueba.

Sin duda, la herramienta *Ecl-Emma* y su *plugin* para *Eclipse* son excelentes herramientas que contribuyen a aumentar la calidad de nuestro código.

Lecturas recomendadas.

- Un excelente libro, de autores españoles, donde se trata de un modo completo las pruebas de software es la referencia [5]. De lectura obligada si se quiere poner en práctica las pruebas de software.

- Otra excelente referencia de autores españoles es [6]. En la primera parte de este libro se expone con claridad los principios del Diseño de Software Dirigido por Pruebas. En la segunda parte del libro se aplica lo expuesto en la primera parte al desarrollo de una aplicación de ejemplo. Aunque los

lenguajes de programación que muestran los ejemplos con .Net y Python, la aplicación a Java con JUnit es directa.

- El capítulo 10 de [13] presenta las cómo realizar pruebas unitarias con JUnit, y en el capítulo 12 muestra como trabajar con *Cobertura* para analizar el grado de cobertura de nuestras pruebas. *Cobertura* es otra excelente herramienta para el análisis de cobertura de las pruebas unitarias.

- Otro excelente título que debería figurar en todos las estanterías de un buen desarrollador es *Clean code* de Robert C. Martin, también conocido como *uncle Bob*. En el capítulo 9 de esta referencia [10] está dedicado a las buenas prácticas en el diseño de test unitarios de prueba.