Práctica 2
Pruebas Unitarias – JUnit 4

Objetivos:

- Definir pruebas unitarias con **JUnit 4**.
- Entender y definir código *stub* de apoyo a las pruebas unitarias.
- Analizar la cobertura de las pruebas con el *plugin* **`EclEmma`** en el entorno Eclipse.

Previo: JUnit y eCobertura

JUnit 4 es un *framework* de pruebas que utiliza anotaciones para identificar los métodos que especifican una prueba. Estos métodos se recogen en una *clase de prueba*. Todas las clases de prueba las vamos a almacenar en una carpeta fuente que vamos a denominar `test` (ver Fig.1). El objetivo de crear esta carpeta es duplicar la estructura del proyecto en la carpeta de las pruebas. De esta forma las clases de prueba podrán acceder a las clases que se están probando sin necesidad de importarlas, se comporta como si estuvieran en el mismo paquete. Lo normal es que definamos una clase de pruebas por cada clase implementada en el proyecto. El nombre de la clase de prueba suele ser *`NombreClase`***`Test.java`**.

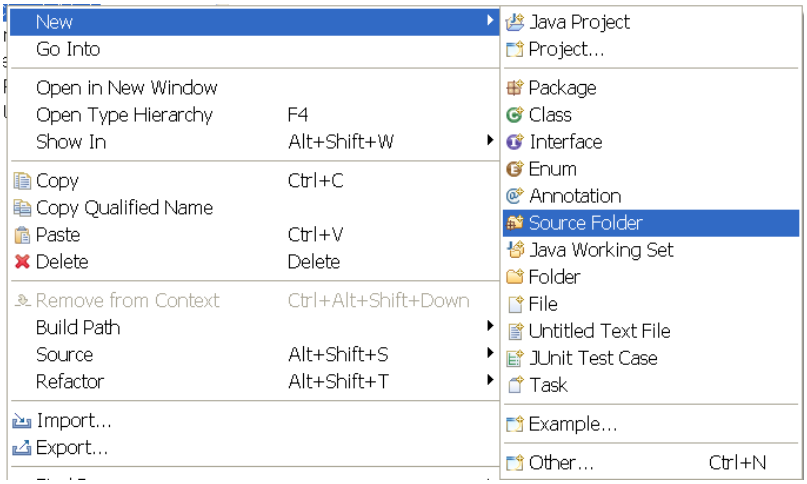

Fig. 1 Creación de carpeta fuente. Menú contextual del botón derecho del ratón pulsado sobre la carpeta del proyecto.

Para crear una clase de pruebas nos situaremos sobre la carpeta de pruebas y en el menú contextual del botón derecho del ratón seleccionaremos `New >> JUnit Test Case` (Fig. 2). La primera vez que se ejecute esta opción de menú se solicitará la instalación de la librería del *framework* de pruebas (Fig. 3).

La etiqueta que identifica un método de la clase de prueba como método de prueba es `@Test`. JUnit asume que todos los métodos de prueba se pueden ejecutar en un orden arbitrario y, por tanto**, los métodos de prueba no pueden depender unos de otros, deben ser independientes**.

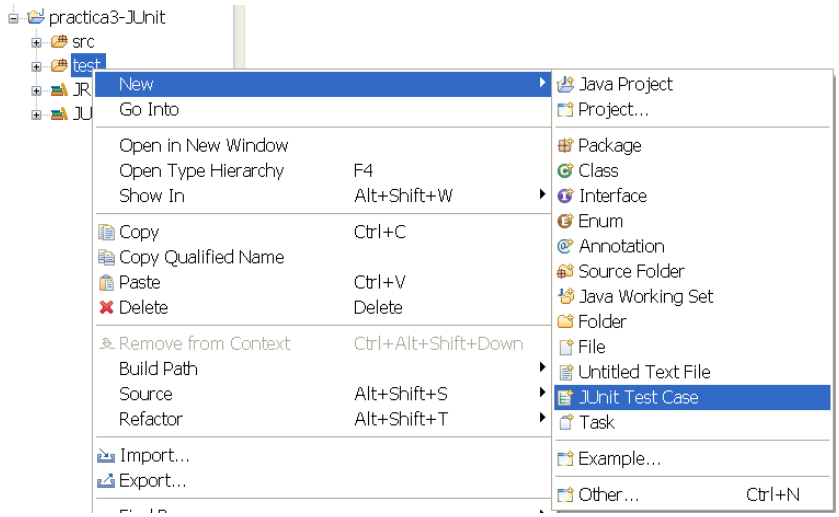

Fig. 2 Creación de una clase de prueba.

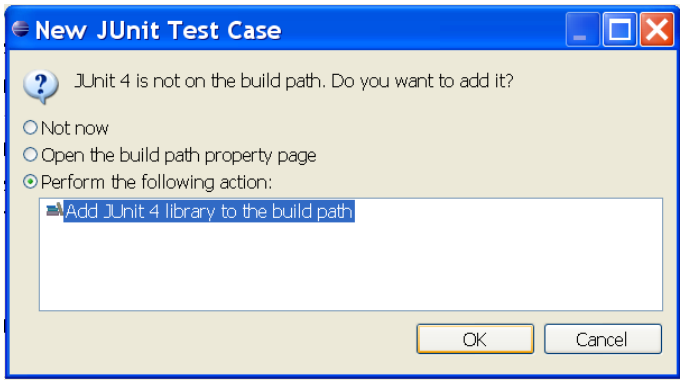

Fig. 3 Añadir la librería `JUnit 4` a la ruta de construcción.

Pongamos como ejemplo la clase `Dividir` que se encarga de realizar la división entre dos números reales. Para que esta operación se pueda realizar, es un requisito (precondición) que el divisor no sea cero (Fig. 4). Vamos a definir una clase de prueba (`DividirTest.java`) que nos ayude a localizar posibles errores en el código de la clase `Dividir` (Fig. 5).

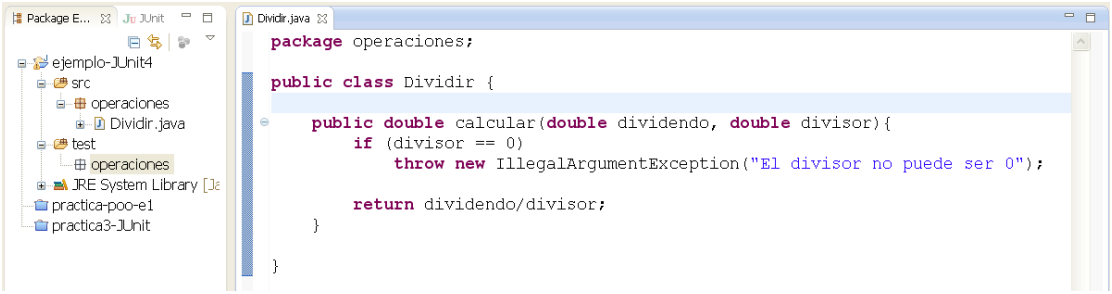

Fig. 4 Clase `Dividir`.

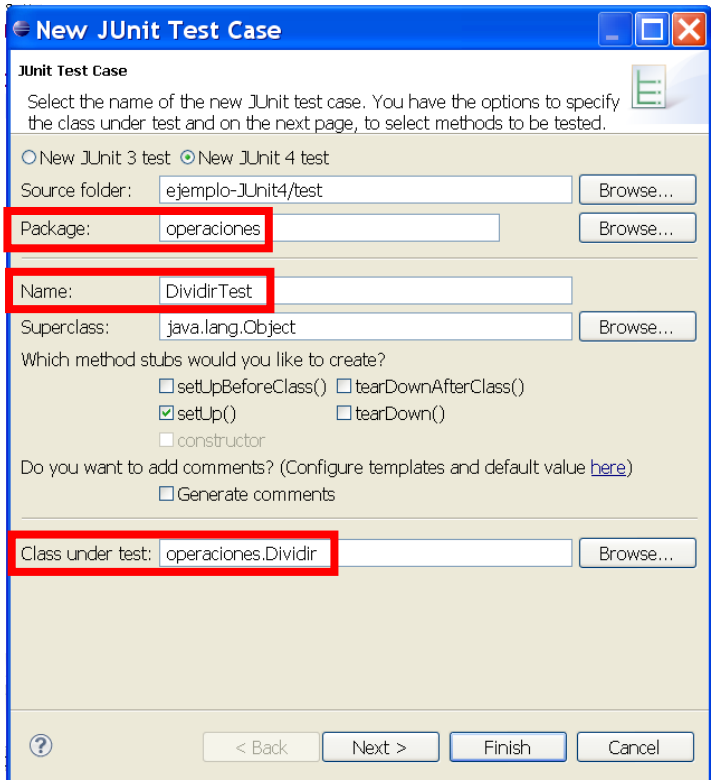

Fig. 5 Definición de la clase de prueba de la clase Dividir.

Si se establece la clase que se va a probar (*Class under test*) se puede seleccionar también, mediante un asistente, los métodos de esa clase para los que se debe crear un método de prueba (Fig. 6). La clase de prueba que se genera es la que aparece en la Fig. 7. En esta figura podemos ver que se ha generado un método de prueba `testCalcular`, con la etiqueta `@Test`, y un método denominado `setUp`, con la etiqueta `@Before`. Nótese que sólo se genera un método de prueba por cada método definido en la clase, pero **para un método, lo habitual, es que sea necesario definir más de un caso de prueba.** Cada caso de prueba se debe implementar en un método independiente. **Los nombres de los métodos de prueba deben ser representativos** para que cuando falle una prueba sepamos cuál ha sido el caso que ha fallado.

La etiqueta `@Before` establece que el método `setUp` se va a ejecutar siempre antes de la ejecución de cada método de prueba. Este método va a ser útil para inicializar objetos de prueba, esto es, objetos que tendrán el mismo valor al comienzo de la ejecución de cada prueba. En la Tabla 1 se puede ver un resumen de las etiquetas que se pueden utilizar.

Etiqueta	Descripción
`@Test`	Identifica un método como método de prueba
`@Before`	Ejecuta el método antes de cada prueba
`@After`	Ejecuta el método después de cada prueba
`@Ignore`	Ignora el método de prueba.
`@Test(expected = Exception.class)`	El método de prueba fallará si no lanza la excepción que se ha especificado.

Tabla 1 Etiquetas en JUnit 4.

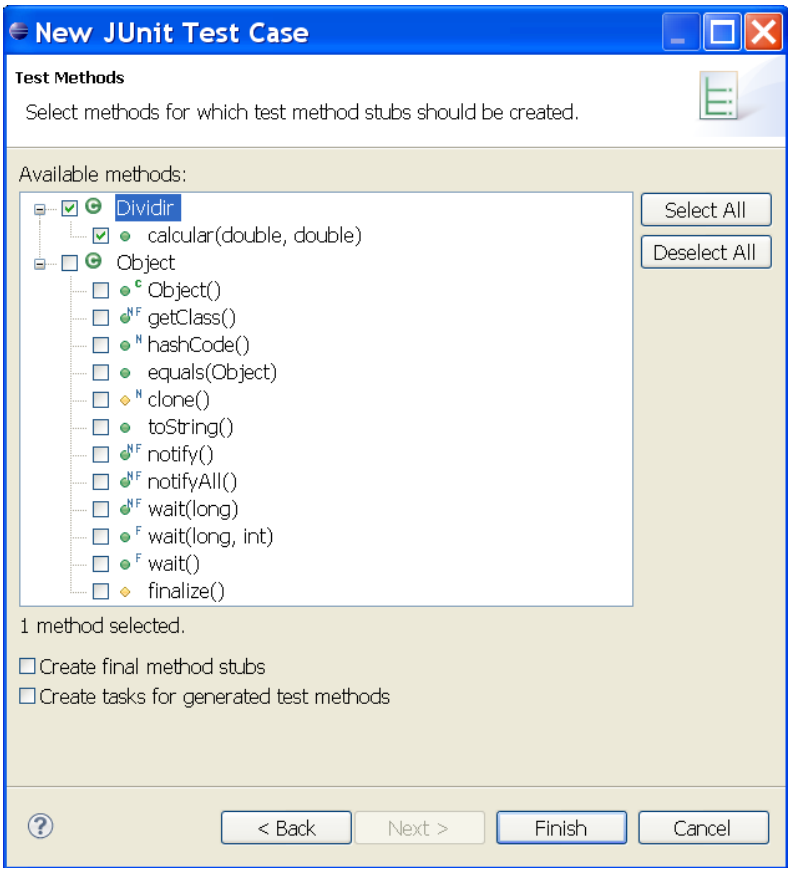

Fig. 6 Métodos de la clase Dividir que se van a probar.

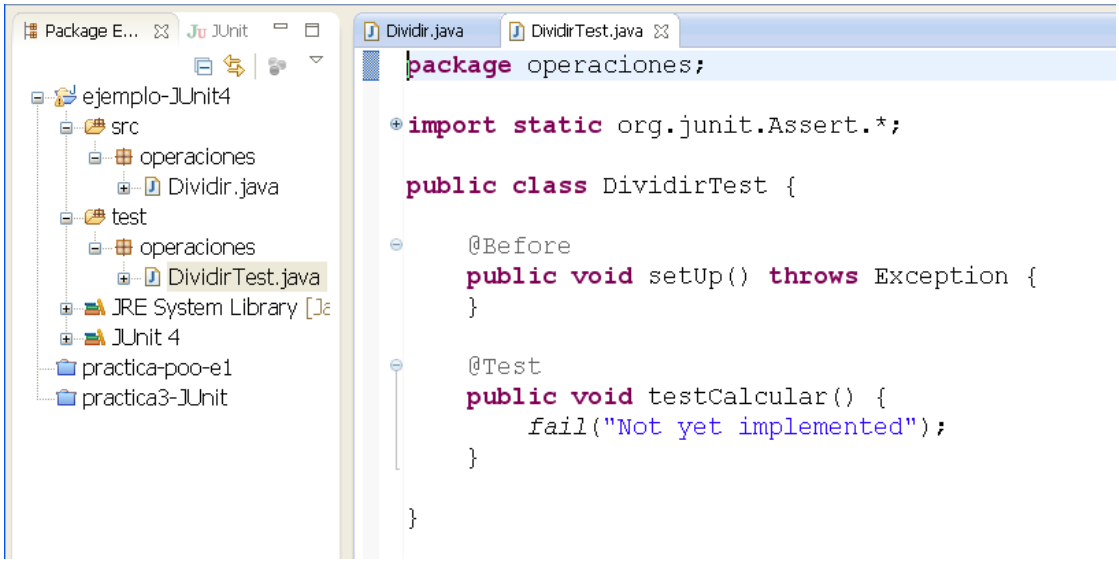

Fig. 7 Clase de prueba generada para la clase Dividir.

Para la definición de los métodos de prueba, JUnit proporciona métodos de clase (`static`) en la clase `org.junit.Assert` para probar ciertas condiciones. Estos métodos permiten especificar el mensaje de error, el resultado esperado y el resultado actual que se obtiene tras la ejecución del método. La Tabla 2 resume los métodos disponibles. Los parámetros que aparecen entre `[]` son opcionales.

Método	Descripción
fail(mensaje)	Hace que falle el método de prueba. Se puede utilizar, por ejemplo, para indicar una parte del código que no debe alcanzarse.
assertTrue([mensaje], boolean condicion)	Comprueba que la condición booleana sea `true`. La prueba falla si no lo es. También está disponible *assertFalse*.
assertEquals([mensaje], esperado, actual, tolerancia)	Comprueba que valores reales (`float` o `double`) coinciden. La tolerancia el error máximo permitido entre esperado y actual.
assertNull([mensaje], object)	Comprueba que `object == null`.
assertNotNull([mensaje], object)	Comprueba que `object != null`.
assertSame([mensaje], esperado, actual)	Comprueba que `esperado == actual`. Es decir, las dos variables referencian al mismo objeto.
assertNotSame([mensaje], esperado, actual)	Comprueba que `esperado != actual`. Es decir, las dos variables referencian a objetos distintos.

Tabla 2 Asertos en JUnit 4.

En el caso de la clase de prueba de la división podríamos definir dos métodos de prueba que comprueben una división correcta y una división por cero, en cuyo caso debe saltar la excepción `IllegalArgumentException`. El contenido de la clase de prueba se muestra en el Fig. 8.

Dividir.java DividirTest.java

```
package operaciones;

import static org.junit.Assert.*;

public class DividirTest {
    private Dividir dividir;

    @Before
    public void setUp() throws Exception {
        dividir = new Dividir();
    }

    @Test
    public void testCalcular() {
        assertEquals(3.0, dividir.calcular(6.0, 2.0), 0);
    }

    @Test (expected = IllegalArgumentException.class)
    public void testCalcularDivisionCero() {
        assertEquals(3.0, dividir.calcular(6.0, 0.0), 0);
    }

}
```

Fig. 8 Clase de prueba para la división.

Para ejecutar la clase de prueba podemos situar el ratón sobre la clase que se quiere ejecutar y seleccionar `Run As >> JUnit test`. En la vista de ejecución de las pruebas (Fig. 9) se muestra el resultado y se marcan las pruebas que han tenido éxito y cuáles no. Si un caso de prueba no ha tenido éxito se puede deber a un ***error*** o un ***fallo*** *(Failure)*. Un ***error*** indica que el test ha fracasado debido a que se ha producido una excepción durante la ejecución del caso de prueba, posiblemente la

ejecución del método que se está probando ha provocado una excepción. Un ***fallo*** indica que el test ha fracasado debido a que uno de los asertos no se cumple.

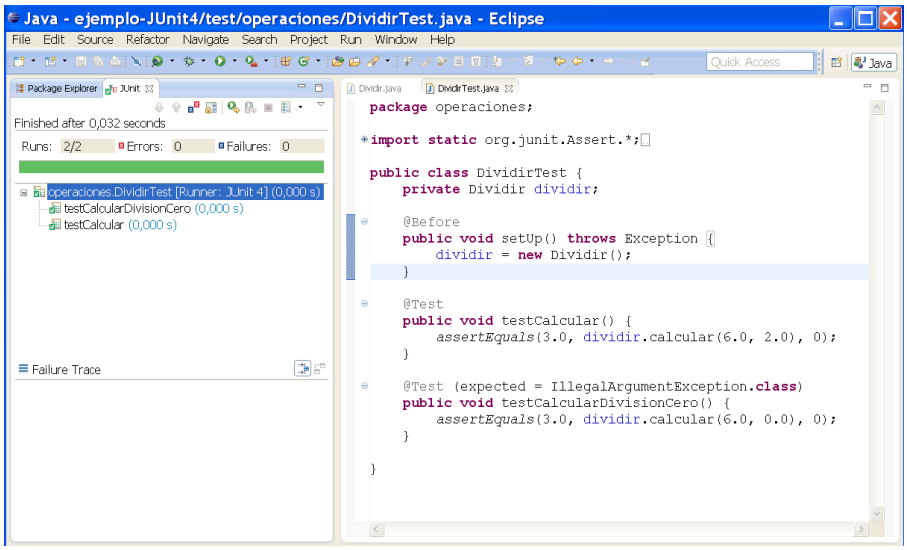

Fig. 9 Ejecución de las pruebas.

Una vez que se han definido las pruebas, resulta de utilidad conocer si éstas cubren todo los caminos posibles del código que se está probando. En el entorno Eclipse se puede instalar el *plugin* para el análisis de la cobertura de las pruebas denominado **EclEmma**. Una vez instalado el *plugin*, seleccionando la clase de prueba nos aparecerá la opción del menú contextual del botón derecho `Cover As >> JUnit test`. La ejecución de esta opción colorea el código marcando en verde las sentencias que se han ejecutado y en rojo aquellas partes que no están cubiertas por las pruebas (Fig. 11). Además, se muestra el informe de los porcentajes de cobertura en la vista *Coverage* que se abre automáticamente en el panel inferior. Vemos en la Fig. 11 que se ha cubierto el 100% del código de la carpeta del código fuente (`src`).

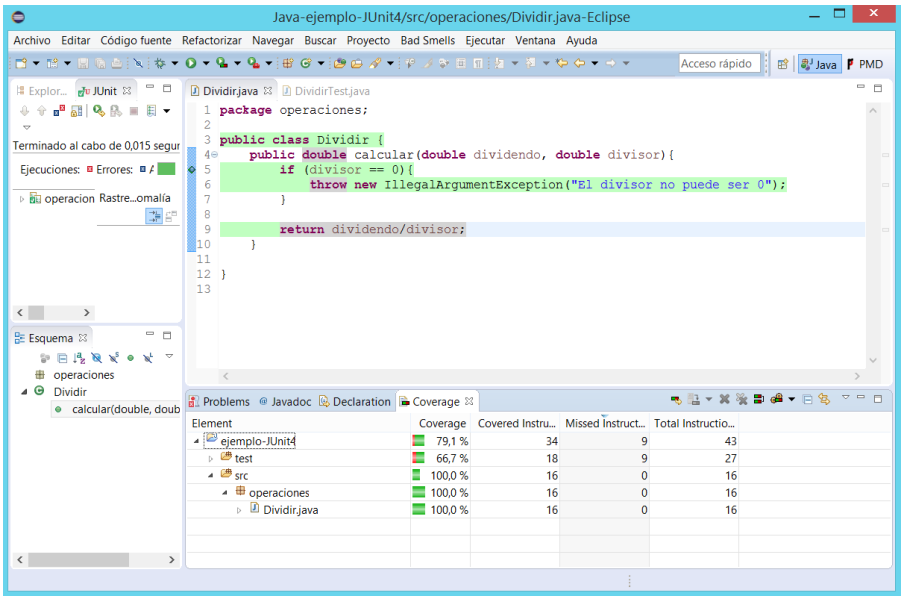

Fig. 10 Cobertura del código.

Ejercicios

1. Descarga el proyecto Eclipse que se proporciona en el Aula Virtual y renómbralo siguiendo el formato descrito en la práctica 1.
2. Crea una "carpeta fuente" denominada `test` (con la misma estructura que la carpeta `src`). En esta carpeta se incluirá los tipos y clases que hay que implementar.
3. Crear una interfaz que defina el tipo `Inventario` en la carpeta `src` y una clase que la implemente que represente un *stub* del `Inventario` real en el paquete adecuado de la carpeta `test`.
4. Define la clase `CafeteraTest` que incluya los casos de prueba para los métodos de la clase `Cafetera: addReceta, editarReceta, hacerCafe`. El objetivo es utilizando la especificación para diseñar casos de prueba que permitan encontrar los errores ocultos en el código de los métodos indicados de la clase `Cafetera`. Una vez identificado cada error se debe marcar con la etiqueta `//FIXME`, comentar el código donde se produce cada error y arreglarlo debidamente.
5. Comprobar el porcentaje de cobertura para comprobar que es el máximo posible.

Entregables

La entrega se realizará en la tarea del Aula Virtual asignada a la práctica con fecha de entrega 14/11/2021 23:55 horas y se debe subir a la tarea del Aula Virtual el espacio de trabajo de Eclipse con el que se ha trabajado. Para ello se debe comprimir la carpeta del proyecto, **no el espacio de trabajo** Eclipse.