TEMA 3: PROGRAMACIÓN MODULAR

(4) PRUEBAS:

La finalidad de las pruebas es detectar posibles defectos o errores. Por ello, se definen pensando en provocar fallos (errores al ejecutar el programa). Si la prueba falla, entonces hay defectos (lo contrario NO es cierto, es decir, no porque no detectemos fallos en las pruebas significa que el código no los contenga).

Para realizar pruebas se suele comenzar por los módulos que no utilizan otros módulos (bottom-up).

Las pruebas deben comprobar que el resultado obtenido es el esperado, ser lo más exhaustivas posibles (comprobando los posibles casos que puedan provocar más fallos), ser unitarias (probando en cada prueba una única funcionalidad) y ser lo más sencillas posibles (legibilidad).

Si una prueba falla, se debe encontrar y arreglar el fallo antes de continuar el proceso y repetir de nuevo todas las pruebas.

Hasta ahora hemos usado el método ***main*** para comprobar que un programa funciona creando objetos, llamando a métodos sobre esos objetos, comprobando condiciones e imprimiendo por consola los resultados; pero existen otras formas de hacerlo. Entre ellas se encuentra la **librería JUnit** (*http://www.junit.org*) que realiza pruebas unitarias mejor estructuradas con aserciones que comprueban condiciones sobre datos de salida y anotaciones que comprueban el rendimiento de un método o el lanzamiento de excepciones.

JUnitTest es una clase que contiene una serie de métodos test que sirven para comprobar si un método de una clase A es correcto o no. Cada método test debe realizar (como mínimo) N veces lo siguiente: (1) llamar al método con unos ciertos argumentos; y (2) comprobar que produce el resultado correcto.

Para poder utilizarlo debemos importar la librería junit-4.12.jar y ejecutar la clase como *JUnit Test*. Eclipse genera un informe en el que se muestra el resultado de la ejecución de cada test.

JUnit trabaja con anotaciones `@` → etiquetas que proporcionan información adicional acerca del elemento (método) que viene a continuación. Algunas de ellas son:
`@Test`: indica que el método que viene a continuación es un método test (prueba):
`@BeforeClass`: sólo se realiza una vez antes de que se ejecute la primera prueba
`@Before`: se ejecuta justo antes de cada prueba
`@Test (timeout = milisegundos)`: detecta si tarda mucho
`@Test (expected = exception.class)`: comprueba que un método genera una excepción cuando debe
`@AfterClass`: se ejecuta al terminar todas las pruebas
`@After`: se ejecuta después de cada prueba

Ejemplo:

```
/**
* Test method for {@link
* Almacen#test0Constructor()}.
*/
@Test
public void test0Constructor() {
      …
}
```

Para comprobar si los resultados producidos por un método que se está probando son correctos, se utilizan aserciones. Para implementar una aserción se pueden utilizar los siguientes métodos del *framework* ***JUnit***:

- `static void assertTrue(boolean condition)`
- `static void assertTrue(String msg, boolean condition)`
- `static void assertEquals(String msg, Object expected, Object actual)`
- `static void assertEquals(Object expected, Object actual)`
- `static void assertEquals(Object expected, Object actual, double range)`
- `static void assertArrayEquals(Object[] expected, Object[] actuals)`

Después de ejecutar un test, éste se marca en el informe de resultados como:

- **Error**: Se ha producido una excepción no esperada o ha vencido un timeout
- **Failure**: No se ha cumplido un aserto o no se ha generado una excepción esperada

Ejemplo Test JUnit:

```
public class TestAlmacen {
      private Almacen almacen;
      @Before
      public void setUp() {
            almacen = new Almacen(3);
      }
      /**
      * Test method for {@link
      * Almacen#test0Constructor()}.
      */
      @Test
      public void test0Constructor() {
            int[] salida = {0,0,0};
            assertArrayEquals(salida, almacen.getProductos());
      }
}
```

Antes de ejecutar cada test (prueba con @Test), inicializamos el almacen.

Comprueba si el almacen contiene 3 posiciones inicializadas con los elementos nulos (en este caso 0)

Ejercicio1: Crear una clase Subscripcion. Esta clase representa una subscripción a una publicación. Cada subscripción contiene el precio total de la subscripción en la variable precio. El precio recoge el total en euros y céntimos (2 decimales). **Por ejemplo**: 1245 representa 12 euros y 45 céntimos; mientras que 1300 representa 13 euros. El periodo de subscripción se representa en meses y se recoge en la variable periodo.

- precio : int
- periodo : int
+ Subscripcion (p : int, n : int)
+ precioPorMes () : int
    POST: calcula el precio de la subscripción mensual en euros, redondeándolo por arriba al céntimo más cercano
+ cancel ()
    POST: cancela la subscripción (periodo = 0)

Hacer una clase de tipo **Tester JUnit** (*Pulsando sobre la clase Subscripcion → New → JUnit Test Case*) que nos permita probar el método precioPorMes() en los siguientes casos:

a. Con una subscripción de 2 euros en 2 meses
b. Con una subscripción de 2 euros en 3 meses

Ejercicio2: Crear una clase OperadorAritmetico (sin variables ni constructores) que nos permita sumar, restar, multiplicar y dividir dos números. Crear una clase OperadorAritmeticoTest que nos permita probar todos y cada uno de estos métodos.

Ejercicio3: Queremos ser capaces de generar informes como:

Empresa	**Acciones**	**Precio**	**Total**
Microsoft	200	10 EUR	2000 EUR
Indra	100	50 EUR	5000 EUR
			7000 EUR

Pero que también nos permita encontrarnos con situaciones como:

Empresa	**Acciones**	**Precio**	**Total**
Microsoft	200	13 USD	2000 EUR
Indra	100	50 EUR	5000 EUR
			7000 EUR

en la que hemos tenido que utilizar el tipo de cambio **1€ = $1.30**

Para ello comenzaremos creando una clase Money con los atributos cantidad (int) y moneda (String) con su constructor (con parámetros), sus getters correspondientes, su método equals y su método toString. Además, vamos a querer sumar cantidades (implemente el método add (m : Money) : Money)***** y multiplicar la cantidad por un valor numérico entero (implemente el método times (n : int))

* **NOTA**: tenga en cuenta que las monedas deben pertenecer al mismo tipo (no intente sumar *euros* y *dólares*). Para ello sería útil que implementara un método exchange (dinero : Money, moneda : String) : Money que nos permita cambiar (en caso de ser necesario) un tipo de moneda a otro (tenga en cuenta la relación anterior **1€ = $1.30**)

Cree una clase MoneyTest que nos permita probar todos y cada uno de estos casos (compruebe los ejemplos de las tablas si no sabe por dónde empezar).