JUnit – Ejercicio Sencillo 1

Supongamos que tenemos una clase EmpleadoBR con las reglas de negocio aplicables a los empleados de una tienda. En esta clase encontramos los siguientes métodos con sus respectivas especificaciones:

Método	**Especificación**
float calculaSalarioBruto(TipoEmpleado tipo, float ventasMes, float horasExtra)	El salario base será 1000 euros si el empleado es de tipo TipoEmpleado.vendedor, y de 1500 euros si es de tipo TipoVendedor.encargado. A esta cantidad se le sumará una prima de 100 euros si ventasMes es mayor o igual que 1000 euros, y de 200 euros si fuese al menos de 1500 euros. Por último, cada hora extra se pagará a 20 euros. Si tipo es null, o ventasMes o horasExtra toman valores negativos el método lanzará una excepción de tipo BRException.
float calculaSalarioNeto(float salarioBruto)	Si el salario bruto es menor de 1000 euros, no se aplicará ninguna retención. Para salarios a partir de 1000 euros, y menores de 1500 euros se les aplicará un 16%, y a los salarios a partir de 1500 euros se les aplicará un 18%. El método nos devolverá salarioBruto * (1-retencion), o BRExcepcion si el salario es menor que cero.

- A partir de dichas especificaciones podemos diseñar un conjunto de casos de prueba siguiendo métodos como el método de pruebas caja negra.
- Si en lugar de contar con la especificación, contásemos con el código del método a probar también podríamos diseñar a partir de él un conjunto de casos de prueba utilizando otro tipo de métodos (en este caso se podría utilizar el método de caja blanca).

No vamos a entrar ahora en el estudio de estos métodos de prueba, sino que nos centraremos en el estudio de la herramienta JUnit. Por lo tanto, supondremos que después de aplicar un método de pruebas hemos obtenido los siguientes casos de prueba:

Método a probar	**Entrada**	**Salida esperada**
calculaSalarioNeto	2000	1640
calculaSalarioNeto	1500	1230
calculaSalarioNeto	1499.99	1259.9916
calculaSalarioNeto	1250	1050
calculaSalarioNeto	1000	840
calculaSalarioNeto	999.99	999.99
calculaSalarioNeto	500	500
calculaSalarioNeto	0	0
calculaSalarioNeto	-1	BRException
calculaSalarioBruto	vendedor, 2000 euros, 8h	1360
calculaSalarioBruto	vendedor, 1500 euros, 3h	1260
calculaSalarioBruto	vendedor, 1499.99 euros, 0h	1100
calculaSalarioBruto	encargado, 1250 euros, 8h	1760
calculaSalarioBruto	encargado, 1000 euros, 0h	1600
calculaSalarioBruto	encargado, 999.99 euros, 3h	1560
calculaSalarioBruto	encargado, 500 euros, 0h	1500
calculaSalarioBruto	encargado, 0 euros, 8h	1660
calculaSalarioBruto	vendedor, -1 euros, 8h	BRException
calculaSalarioBruto	vendedor, 1500 euros, -1h	BRException
calculaSalarioBruto	null, 1500 euros, 8h	BRException

Nota:
Los casos de prueba se pueden diseñar e implementar antes de haber implementado el método a probar. De hecho, es recomendable hacerlo así, ya que de esta forma las pruebas comprobarán si el método implementado se ajusta a las especificaciones que se dieron en un principio. Evidentemente, esto no se podrá hacer en las pruebas que diseñemos siguiendo el método de caja blanca. También resulta conveniente que sean personas distintas las que se encargan de las pruebas y de la implementación del método, por el mismo motivo comentado anteriormente.

Implementación de los casos de prueba utilizando JUnit 4.

Estrategia y Método :

Vamos a utilizar JUnit para probar los métodos anteriores. Para ello deberemos crear una serie de clases en las que implementaremos las pruebas diseñadas. Esta implementación consistirá básicamente en invocar el método que está siendo probado pasándole los parámetros de entrada establecidos para cada caso de prueba, y comprobar si la salida real coincide con la salida esperada. Esto en principio lo podríamos hacer sin necesidad de utilizar JUnit, pero el utilizar esta herramienta nos va a ser de gran utilidad ya que nos proporciona un *framework* que nos obligará a implementar las pruebas en un formato estándar que podrá ser reutilizable y entendible por cualquiera que conozca la librería. El aplicar este *framework* también nos ayudará a tener una batería de pruebas ordenada, que pueda ser ejecutada fácilmente y que nos muestre los resultados de forma clara mediante una interfaz gráfica que proporciona la herramienta. Esto nos ayudará a realizar pruebas de regresión, es decir, ejecutar la misma batería de pruebas en varios momentos del desarollo, para así asegurarnos de que lo que nos había funcionado antes siga funcionando bien.

Para implementar las pruebas en JUnit utilizaremos dos elementos básicos:

- Por un lado, marcaremos con la anotación @Test los métodos que queramos que JUnit ejecute. Estos serán los métodos en los que implementemos nuestras pruebas. En estos métodos llamaremos al método probado y comprobaremos si el resultado obtenido es igual al esperado.
- Para comprobar si el resultado obtenido coincide con el esperado utilizaremos los métodos assert de la librería JUnit. Estos son una serie de métodos estáticos de la clase Assert (para simplificar el código podríamos hacer un *import* estático de dicha clase), todos ellos con el prefijo assert-. Existen multitud de variantes de estos métodos, según el tipo de datos que estemos comprobando (assertTrue, assertFalse, assertEquals, assertNull, etc). Las llamadas a estos métodos servirán para que JUnit sepa qué pruebas han tenido éxito y cuáles no.

Cuando ejecutemos nuestras pruebas con JUnit, se nos mostrará un informe con el número de pruebas éxitosas y fallidas, y un detalle desglosado por casos de prueba. Para los casos de prueba que hayan fallado, nos indicará además el valor que se ha obtenido y el que se esperaba.

Además de estos elementos básicos anteriores, a la hora de implementar las pruebas con JUnit deberemos seguir una serie de buenas prácticas que se detallan a continuación:

- La clase de pruebas se llamará igual que la clase a probar, pero con el sufijo -Test. Por ejemplo, si queremos probar la clase MiClase, la clase de pruebas se llamará MiClaseTest.
- La clase de pruebas se ubicará en el mismo paquete en el que estaba la clase probada. Si MiClase está en el paquete es.ua.jtech.lja, MiClaseTest pertenecerá e ese mismo paquete. De esta forma nos aseguramos tener acceso a todos los miembros de tipo protegido y paquete de la clase a probar.
- Mezclar clases reales de la aplicación con clases que sólo nos servirán para realizar las pruebas durante el desarrollo no es nada recomendable, pero no

queremos renunciar a poner la clase de pruebas en el mismo paquete que la clase probada. Para solucionar este problema lo que se hará es crear las clases de prueba en un directorio de fuentes diferente. Si los fuentes de la aplicación se encuentran normalmente en un directorio llamado src, los fuentes de pruebas irían en un directorio llamado test.

- Los métodos de prueba (los que están anotados con @Test), tendrán como nombre el mismo nombre que el del método probado, pero con prefijo test-. Por ejemplo, para probar miMetodo tendríamos un método de prueba llamado testMiMetodo.
- Aunque dentro de un método de prueba podemos poner tantos assert como queramos, es recomendable crear un método de prueba diferente por cada caso de prueba que tengamos. Por ejemplo, si para miMetodo hemos diseñado tres casos de prueba, podríamos tener tres métodos de prueba distintos: testMiMetodo1, testMiMetodo2, y testMiMetodo3. De esta forma, cuando se presenten los resultados de las pruebas podremos ver exactamente qué caso de prueba es el que ha fallado.

<u>En nuestro caso:</u>

Si partimos de

```
enum TipoEmpleado {
    vendedor, encargado
};
class BRException extends Exception {

    public BRException(String msj) {
        super(msj);
    }
}
public class EmpleadoBR {
   //private String nombre;
   //otros atributos
    float calculaSalarioBruto(TipoEmpleado tipo, float ventasMes, float horasExtra)
            throws BRException {...19 lines }

    public float calculaSalarioNeto(float salarioBruto) throws BRException
    {...15 lines }
```

Los test podrían ser como:

```
@Test
public void testCalculaSalarioBruto1() throws Exception {
    System.out.println("calculaSalarioBruto");
    TipoEmpleado tipo = TipoEmpleado.vendedor;
    float ventasMes = 2000.0F;
    float horasExtra = 8.0F;
    EmpleadoBR instance = new EmpleadoBR();
    float expResult =1360.0F;
    float result = instance.calculaSalarioBruto(tipo, ventasMes, horasExtra);
    assertEquals(expResult, result, 0.01);

}
```

```java
@Test
public void testCalculaSalarioNeto1() throws Exception {
    System.out.println("calculaSalarioNeto1");
    float salarioBruto = 2000.0F;
    EmpleadoBR instance = new EmpleadoBR();
    float expResult = 1640.0F;
    float result = instance.calculaSalarioNeto(salarioBruto);
    assertEquals(expResult, result, 0.01);

}
```

Se recomienda escribir un método por cada prueba a realizar

En algunos casos de prueba, lo que se espera como salida no es que el método nos devuelva un determinado valor, sino que se produzca una excepción.

En este caso lo que hacemos es llamar al método probado dentro de un bloque try-catch que capture la excepción esperada. Si al llamar al método no saltase la excepción, llamamos a fail() para indicar que no se ha comportado como debería según su especificación

```java
@Test
public void testCalculaSalarioNeto9() {
    try {
        float salarioBruto = -1.0F;
        EmpleadoBR instance = new EmpleadoBR();
        instance.calculaSalarioNeto(salarioBruto);
        fail("Se esperaba excepcion BRException");
    } catch (BRException e) {
    }
}
```