¿Que es prueba unitaria en java y como funciona?

Una prueba unitaria según Wikipedia es “una forma de probar el correcto funcionamiento de un módulo de código. Esto sirve para asegurar que cada uno de los módulos funcione correctamente por separado.”. Para algunos, este tipo de pruebas es una pérdida de tiempo o una inutilidad. Hay que decir que crear las pruebas supone un tiempo adicional al que dedicamos a programar la aplicación en sí. Debemos valorar si merece la pena o no, según las dimensiones y la complejidad de nuestro proyecto. Lo cierto es que si las hacemos, dependiendo del caso, podemos ahorrarnos mucho tiempo, ya que habremos automatizado las pruebas e instantáneamente sabremos si nuestra aplicación cumple con los resultados correctos. Mejoraremos la calidad del software, asegurándonos de que aunque hagamos cambios en nuestro código, cumpliremos con los requisitos esperados. En un futuro podremos modificar, ampliar o eliminar código y las pruebas unitarias seguirán sirviéndonos para realizar las comprobaciones pertinentes.?

Ejemplo de pruebas unitarias en java:

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 tipoTipoEmpleado.vendedor, y de 1500 euros si es de tipoTipoVendedor.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 tipoes 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 de particiones, también conocido como 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 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 |