**PRUEBAS UNITARIAS**

En [programación](https://es.wikipedia.org/wiki/Programaci%C3%B3n), una **prueba unitaria** es una forma de comprobar el correcto funcionamiento de una unidad de código. Por ejemplo en [diseño estructurado](https://es.wikipedia.org/wiki/Dise%C3%B1o_estructurado) o en [diseño funcional](https://es.wikipedia.org/w/index.php?title=Dise%C3%B1o_funcional&action=edit&redlink=1) una función o un procedimiento, en [diseño orientado a objetos](https://es.wikipedia.org/wiki/Dise%C3%B1o_orientado_a_objetos) una clase. Esto sirve para asegurar que cada unidad funcione correctamente y eficientemente por separado. Además de verificar que el código hace lo que tiene que hacer, verificamos que sea correcto el nombre, los nombres y tipos de los parámetros, el tipo de lo que se devuelve, que si el estado inicial es válido entonces el estado final es válido

La idea es escribir casos de prueba para cada función no trivial o [método](https://es.wikipedia.org/wiki/M%C3%A9todo_(inform%C3%A1tica)) en el módulo, de forma que cada caso sea independiente del resto. Luego, con las [Pruebas de Integración](https://es.wikipedia.org/wiki/Pruebas_de_Integraci%C3%B3n), se podrá asegurar el correcto funcionamiento del sistema o subsistema en cuestión.

**CARACTERISTICAS:**

Para que una prueba unitaria tenga la *calidad suficiente* se deben cumplir los siguientes requisitos:

**Automatizable**

No debería requerirse una intervención manual. Esto es especialmente útil para [integración continúa](https://es.wikipedia.org/wiki/Integraci%C3%B3n_continua).

**Completas**

Deben cubrir la mayor cantidad de código.

**Repetibles o Reutilizables**

No se deben crear pruebas que sólo puedan ser ejecutadas una sola vez. También es útil para [integración continua](https://es.wikipedia.org/wiki/Integraci%C3%B3n_continua).

**Independientes**

La ejecución de una prueba no debe afectar a la ejecución de otra.

**Profesionales**

Las pruebas deben ser consideradas igual que el código, con la misma profesionalidad, documentación, etc.

Aunque estos requisitos no tienen que ser cumplidos al pie de la letra, se recomienda seguirlos o de lo contrario las pruebas pierden parte de su función.

**VENTAJAS**

El objetivo de las pruebas unitarias es aislar cada parte del programa y mostrar que las partes individuales son correctas. Proporcionan un contrato escrito que el trozo de código debe satisfacer. Estas pruebas aisladas proporcionan cinco ventajas básicas:

**Fomentan el cambio**

Las pruebas unitarias facilitan que el programador cambie el código para mejorar su estructura (lo que se ha dado en llamar [refactorización](https://es.wikipedia.org/wiki/Refactorizaci%C3%B3n)), puesto que permiten hacer pruebas sobre los cambios y así asegurarse de que los nuevos cambios no han introducido errores.

**Simplifica la integración**

Puesto que permiten llegar a la fase de integración con un grado alto de seguridad de que el código está funcionando correctamente. De esta manera se facilitan las [pruebas de integración](https://es.wikipedia.org/wiki/Pruebas_de_Integraci%C3%B3n).

**Documenta el código**

Las propias pruebas son documentación del código puesto que ahí se puede ver cómo utilizarlo.

**Separación de la interfaz y la implementación**

Dado que la única interacción entre los casos de prueba y las unidades bajo prueba son las interfaces de estas últimas, se puede cambiar cualquiera de los dos sin afectar al otro, a veces usando [objetos mock](https://es.wikipedia.org/wiki/Objetos_mock) (mock object) para simular el comportamiento de objetos complejos.

**Los errores están más acotados y son más fáciles de localizar**

Dado que tenemos pruebas unitarias que pueden desenmascararlos.

**HERRAMIENTAS**

* JUnit: Entorno de pruebas para Java creado por Erich Gamma y Kent Beck. Se encuentra basado en SUnit creado originalmente para realizar pruebas unitarias para el lenguaje Smalltalk.
* TestNG: Creado para suplir algunas *deficiencias* en JUnit.
* [JTiger](https://es.wikipedia.org/w/index.php?title=JTiger&action=edit&redlink=1): Basado en anotaciones, como TestNG.
* [SimpleTest](https://es.wikipedia.org/w/index.php?title=SimpleTest&action=edit&redlink=1): Entorno de pruebas para aplicaciones realizadas en PHP.
* [PHPUnit](https://es.wikipedia.org/wiki/PHPUnit): Sistema para la realización pruebas unitarias en PHP.
* [CPPUnit](https://es.wikipedia.org/wiki/CPPUnit): Versión del framework para lenguajes C/C++.
* [NUnit](https://es.wikipedia.org/wiki/NUnit): Versión del framework para la plataforma.NET.
* [FoxUnit](https://es.wikipedia.org/w/index.php?title=FoxUnit&action=edit&redlink=1): Entorno OpenSource de pruebas unitarias para Microsoft Visual FoxPro
* [MOQ](https://es.wikipedia.org/w/index.php?title=MOQ&action=edit&redlink=1): Entorno para la creación dinámica de objetos simuladores (mocks). [«MOQ»](https://github.com/Moq/moq4/).
* [QUnit](https://es.wikipedia.org/w/index.php?title=QUnit&action=edit&redlink=1): Librería para pruebas unitarias en Javascript. Creada por la fundación jQuery, ha sido reescrita para ser independiente de la librería jQuery.
* [libunittest](http://libunittest.sourceforge.net/): Librería portable para pruebas unitarias en C++ que usa el nuevo estándar C++11.
* [CUnit](http://cunit.sourceforge.net/): Entorno para escribir, administar y correr test unitarios en lenguaje C.
* [PyUnit](https://wiki.python.org/moin/PyUnit): Framework para la elaboración de pruebas unitarias en python.

**EJEMPLOS**

Para empezar, vamos a ver el ejemplo más simple posible, para que podamos centrarnos en la comprensión de lo que son las pruebas unitarias y de la herramienta para llevarlas a cabo: **JUnit**. El IDE que utilizaremos será Eclipse.

**1.**Empezaremos creando la clase que queremos probar. Tendrá el siguiente código:

public class Suma {

    private int num1;

   private int num2;

   public Suma(int n1, int n2) {

       num1 = n1;

       num2 = n2;

   }

   public int sumar() {

       int resultado = num1 + num2;

       return resultado;

   }

}

Como vemos, es una clase muy sencilla que dispone de un método que permite sumar dos números.   
  
**2.** Lo siguiente es crear la clase que nos servirá para probar la clase Suma. Queremos saber si la suma se hace correctamente en tres casos: sumando dos números positivos, sumando dos números negativos y sumando un número positivo y un número negativo. El código será el siguiente: 

public class SumaTest {

   @Test

   public void sumaPositivos() {

      System.out.println("Sumando dos números positivos ...");

       Suma S = new Suma(2, 3);

       assertTrue(S.sumar() == 5);

   }

   @Test

   public void sumaNegativos() {

       System.out.println("Sumando dos números negativos ...");

       Suma S = new Suma(-2, -3);

       assertTrue(S.sumar() == -5);

   }

   @Test

   public void sumaPositivoNegativo() {

       System.out.println("Sumando un número positivo y un número negativo ...");

       Suma S = new Suma(2, -3);

       assertTrue(S.sumar() == -1);

   }

}

**VALIDACION / VERIFICACION**

\*VALIDACION

\*¿Estamos construyendo el producto correcto?

\*VERIFICACION

\*¿Estamos construyendo el producto correctamente?

\*3 MANERAS DE VERIFICAR EL SOFTWARE:

\*Pruebas: Experimentar con el programa para encontrar errores

\*Revisiones: Inspeccionar el código para detectar errores (CHECKLISTS)

\*Verificación formal: Razonar sobre el código (F(X)=Y)

**TIPOS DE PRUEBAS**

* UNITARIAS
* FUNCIONALES
* CARGAS/ESTRÉS GARANTIZAN LA CALIDAD DEL SOFTWARE
* ACEPTACION

**CLASES DE PRUEBAS**

* Herendan de la clase TESTCASE
* Define las pruebas como métodos públicos TESTXXX()
* Pueden usar los métodos ASERT() para comprobar los resultados obtenidos frente a los esperados
* Pueden utilizar los métodos setUP() y tearDown() para inicializar y liberar recursos comunes a todas las pruebas en un mismo TestCase.Cada prueba se ejecuta en el contexto de un “test fixture”

**CONCLUSIONES:**

Una prueba de la unidad gekeken el comportamiento de una unidad distinta de trabajo. Dentro de una aplicación Java, la" unidad distinta del trabajo "es a menudo (pero no siempre) un único método. Una unidad de trabajo es una tarea que no depende directamente de la terminación o cualquier otra tarea