**UNIVERSIDAD NACIONAL DE INGENIERÍA**

**FACULTAD DE INGENIERA INDUSTRIAL Y DE SISTEMAS**



**INFORME DE INVESTIGACION:**

**PRUEBAS UNITARIAS**

**NOMBRE:**

**CARO RAMIREZ, KEVIN STEVEN**

**DOCENTE:**

**CORONEL CASTILLO, ERIC GUSTAVO**

**CURSO DE PROGRAMACION EN JAVA**

**LIMA– PERÚ**

**2020**

# 

# DEDICATORIA

A mis padres que me fortalecen con su apoyo económico, moral y valores para ser competitivos y buenos profesionales.

.

# INDICE

[DEDICATORIA](#_Toc11079536) 3

[INDICE](#_Toc11079538) 4

[RESUMEN 5](#_Toc11079539)

DESARROLLO DEL TEMA………………………………………6

CASO DESARROLLO12

CONCLUSIONES [15](#_Toc11079543)

[RECOMENDACIONES 16](#_Toc11079544)

**RESUMEN:**

Una prueba unitaria es una forma de comprobar el correcto funcionamiento de una unidad de código. Una prueba unitaria será tan buena como su cobertura de código. Si nuestra cobertura es baja significa que gran parte de nuestro código esta sin probar.

**DESARROLLO DEL TEMA:**

**Pruebas Unitarias**

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. 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 también.

**Características de una buena prueba unitaria:**

Las pruebas unitarias se tienen que poder ejecutar sin necesidad de intervención manual. Esta característica posibilita que podamos automatizar su ejecución. Las pruebas unitarias tienen que poder repetirse tantas veces como uno quiera. Por este motivo, la rapidez de las pruebas tiene un factor clave. Si pasar las pruebas es un proceso lento no se pasarán de forma habitual, por lo que se perderán los beneficios que éstas nos ofrecen. Las pruebas unitarias deben poder cubrir casi la totalidad del código de nuestra aplicación. Una prueba unitaria será tan buena como su cobertura de código. La cobertura de código marca la cantidad de código de la aplicación que está sometido a una prueba. Por tanto, si la cobertura es baja, significará que gran parte de nuestro código está sin probar.

Las pruebas unitarias tienen que poder ejecutarse independientemente del estado del entorno. Las pruebas tienen que pasar en cualquier ordenador del equipo de desarrollo. La ejecución de una prueba no puede afectar la ejecución de otra. Después de la ejecución de una prueba el entorno debería quedar igual que estaba antes de realizar la prueba. Las diferentes relaciones que puedan existir entre módulos deben ser simulada para evitar dependencias entre módulos. Esto sólo se consigue si se trata el código de pruebas como el código de la aplicación. Es importante tener en cuenta que aunque estas son las características de una buena prueba, no siempre será posible ni necesario cumplir con todas estas reglas y será la experiencia la que nos guiará en la realización de las mismas.

**Mitos de las pruebas unitarias**

Aunque los beneficios de las pruebas unitarias puedan parecer claros, no es más cierto que a día de hoy se usan en muy pocos proyectos. Pero sin son tan buenas, ¿por qué no se usan? Pues la razón principal es que existen bastante desconocimiento en esta materia, poca tradición y algunos falsos mitos. Uno de los mitos es creer que escribir pruebas unitarias es escribir el doble de código; escribir el código de la aplicación y escribir el código de pruebas. Escribir una prueba nunca es escribir el doble de código, aunque lógicamente sí es escribir más código. El mito es totalmente falso. Las pruebas siempre se deben ir escribiendo a medida que se desarrolla el software. A medida de desarrollamos vamos probando nuestro código lo que nos permite asegurar que el módulo queda terminado correctamente, libre de incidencias. La realización de pruebas unitarias debe ser un proceso obligatorio en nuestros desarrollos y que no queden a la voluntad del desarrollador. Si el desarrollador no está habituado a su uso diario es muy fácil que tienda a evitar realizar este tipo de pruebas. Si no estás familiarizado con el uso de pruebas unitarias la perseverancia debe ser tu gran aliado. Debes estar convencido de que el uso de pruebas unitarias es el enfoque correcto y no ser flexible; las pruebas unitarias no son opcionales y deben realizarse a medida que se desarrolla. Dejar la implementación de pruebas para el final no es realista, ya que las pruebas nunca se llegarán a implementar y si lo hacen, serán de una baja calidad, ya que la persona que las realiza considerará que es una pérdida de tiempo, considerando que su tarea ya estaba terminada sin realizar estas pruebas.

¿Y si no escribiéramos este código cómo probamos? Hay diferentes situaciones. La primera situación es que no se prueba. Se implementa el módulo pero no se prueba, dando por hecho que nuestra experiencia como desarrolladores hará que las incidencias sean mínimas. Cuando tenemos todos los módulos lo probamos de forma integrada a través de pruebas funcionales. Esta situación es la situación peor con la que podemos encontrarnos, pero desgraciadamente es la más habitual. Los tiempos de integración, depuración y corrección de incidencias se multiplican. El tiempo que va desde que consideramos que el producto está terminado hasta que realmente está terminado puede llegar a ser superior al tiempo invertido en el desarrollo, y lo digo por experiencia, por mala experiencia. Otra situación algo mejor, es el uso de aplicaciones de pruebas de usar y tirar. Todos hemos implementado aplicaciones tontas, generalmente de consola, para probar el módulo que estamos desarrollando y que posteriormente integraremos en un sistema mayor. La situación no es la ideal pero realmente el desarrollo de pruebas unitarias puede tener una semejanza con este tipo de aplicaciones, siendo éste último concepto más avanzado y mejorado para que pueda ser reutilizable, independientes y completas.

Estas aplicaciones de usar y tirar nos valen para la primera vez y si somos exhaustivos nos permitirá probar bien nuestro módulo. La realidad nos dice que esta aplicación se tira y que ante cualquier modificación en el módulo nos comportamos como en la situación primera, en las pruebas con sistema completa y pruebas funcionales. Otro mito, unido al anterior, es que desarrollar pruebas unitarias hace que los tiempos de desarrollo se incrementen, incrementando los costes del proyecto. Entre desarrollar y no probar y desarrollar haciendo pruebas unitarias, reconoceré que es más rápido desarrollar sin probar…..pero creo que esto no nos vale….así que entre desarrollar con pruebas unitarias y desarrollar probando con métodos más rústicos, ésta segunda es más lenta y por tanto, más cara. La manera más rápida de desarrollar software es desarrollarlo bien. Como hemos comentado en el punto anterior, si no desarrollamos código de calidad y no realizamos pruebas, los tiempos de desarrollo se podrán disparar enormemente; tiempo de integración, tiempo de depuración, tiempo de incidencias etc… Pero hay un problema mayor; la pérdida de confianza del cliente. Un producto de baja calidad, lleno de incidencias es la peor tarjeta de presentación. Por otro lado, hay que tener en cuenta que cuando más larga sea la vida de la aplicación más beneficios se obtendrán del sistema de pruebas unitarias. Muchos productos tienen un mantenimiento evolutivo que implica el desarrollo de nuevas funcionalidad y la corrección de incidencias. Sin un proceso adecuado de pruebas unitarias, los tiempos de pruebas se dispararán en fases posteriores, ya que será necesario probar y corregir tanto las nuevas funcionalidades como las funcionalidades ya existentes.

**Herramientas:**

* [Typemock](https://es.wikipedia.org/w/index.php?title=Typemock&action=edit&redlink=1): Entorno de pruebas para C++, C# y .NET - incluso puedes probar el código legado con él.
* [JUnit](https://es.wikipedia.org/wiki/JUnit): Entorno de pruebas para Java creado por [Erich Gamma](https://es.wikipedia.org/wiki/Erich_Gamma) y [Kent Beck](https://es.wikipedia.org/wiki/Kent_Beck). Se encuentra basado en SUnit creado originalmente para realizar pruebas unitarias para el lenguaje Smalltalk.
* [TestNG](https://es.wikipedia.org/wiki/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): Biblioteca para pruebas unitarias en Javascript. Creada por la fundación jQuery, ha sido reescrita para ser independiente de la biblioteca jQuery.
* [libunittest](http://libunittest.sourceforge.net/): Biblioteca portable para pruebas unitarias en C++ que usa el nuevo estándar C++11.
* [CUnit](http://cunit.sourceforge.net/): Entorno para escribir, administrar y correr test unitarios en lenguaje C.
* [PyUnit](https://wiki.python.org/moin/PyUnit): Framework para la elaboración de pruebas unitarias en python.
* [QtTest](http://doc.qt.io/qt-5/qttest-index.html): Clases para pruebas unitarias en la biblioteca Qt (C++).
* [utPLSQL](http://utplsql.org/): Framework para la elaboración de pruebas unitarias en PLSQL.

**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 defectos.
* 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 maquetados](https://es.wikipedia.org/w/index.php?title=Objetos_maquetados&action=edit&redlink=1) (mock object - maqueta) que habilitan de forma aislada (unitaria) el comportamiento de objetos complejos.
* Los errores están más acotados y son más fáciles de localizar: Dado que tenemos pruebas unitarias que pueden desenmascararlos.

**Introducción a JUnit**

Cuando probamos un programa, lo ejecutamos con unos datos de entrada (casos de prueba) para verificar que el funcionamiento cumple los requisitos esperados. Definimos prueba unitaria como la prueba de uno de los módulos que componen un programa.

En los últimos años se han desarrollado un conjunto de herramientas que facilitan la elaboración de pruebas unitarias en diferentes lenguajes. Dicho conjunto se denomina XUnit. De entre dicho conjunto, JUnit es la herramienta utilizada para realizar pruebas unitarias en Java.

El concepto fundamental en estas herramientas es el caso de prueba (test case), y la suite de prueba (test suite). Los casos de prueba son clases o módulos que disponen de métodos para probar los métodos de una clase o módulo concreta/o. Así, para cada clase que quisiéramos probar definiríamos su correspondiente clase de caso de prueba. Mediante las suites podemos organizar los casos de prueba, de forma que cada suite agrupa los casos de prueba de módulos que están funcionalmente relacionados.

Las pruebas que se van construyendo se estructuran así en forma de árbol, de modo que las hojas son los casos de prueba, y podemos ejecutar cualquier subárbol (suite).

De esta forma, construimos programas que sirven para probar nuestros módulos, y que podremos ejecutar de forma automática. A medida que la aplicación vaya avanzando, se dispondrá de un conjunto importante de casos de prueba, que servirá para hacer pruebas de regresión. Eso es importante, puesto que cuando cambiamos un módulo que ya ha sido probado, el cambio puede haber afectado a otros módulos, y sería necesario volver a ejecutar las pruebas para verificar que todo sigue funcionando.

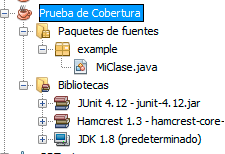
Aplicando lo anterior a Java, JUnit es un conjunto de clases opensource que nos permiten probar nuestras aplicaciones Java.

Encontraremos una distribución de JUnit, en la que habrá un fichero JAR, junit.jar, que contendrá las clases que deberemos tener en el CLASSPATH a la hora de implementar y ejecutar los casos de prueba.

**CASO DESARROLLADO:**

**Paso 1: El proyecto**

Creamos un proyecto que llamaremos «Prueba de Cobertura», agregamos un paquete «example» y las librerías arriba mencionadas. Para finalizar, creamos una clase «Miclase.java«.

****

**Paso 2. El código**

package example;

public class MiClase {

public int numero\_mayor(int a, int b, int c) {

if (a > b && a > c) {

return a;

} else if (c > b) {

return c;

} else {

return b;

}

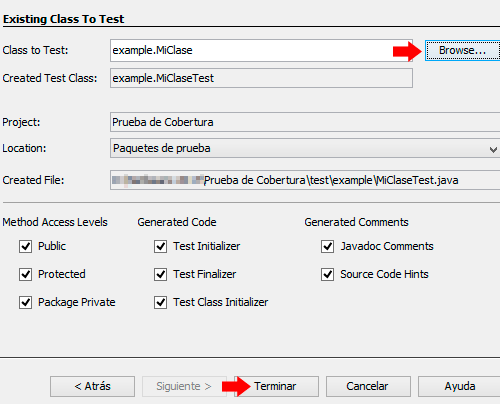
}

}

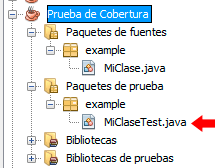
**Paso 3: Paquete de Prueba**

Ya que tenemos nuestra clase de prueba lista, procedemos a crear las clases de prueba. Clic derecho sobre nuestro proyecto -> New -> Other… En Categoría seleccionamos «**Unit Tests**» y en File Types, seleccionamos «**Test for Existing Class**» y presionamos el botón siguiente.

A continuación, presionando el botón «browse…» buscamos y seleccionamos la clase que testearemos, «Miclase«, dejamos todas las opciones tal cual están y finalizamos presionando el botón «Terminar».



Se creara otra clase en la sección de «**Paquete de prueba**» como se ve a continuación



**Paso 4: JUnit**

En la parte inferior de la clase *MiClaseTest* encontramos el siguiente método:

1 @Test

2 public void testNumero\_mayor() {

3 System.out.println("numero\_mayor");

4 int a = 0;

5 int b = 0;

6 int c = 0;

7 MiClase instance = new MiClase();

8 int expResult = 0;

9 int result = instance.numero\_mayor(a, b, c);

10 assertEquals(expResult, result);

11 // TODO review the generated test code and remove the default call to fail.

12 fail("The test case is a prototype.");

13 }

donde:

1) @Test: Los métodos marcados con esta anotación, le indican a JUnit que es código que queremos que se ejecute. Son estos métodos donde se implementan el código de pruebas.

4,5,6) Las variables del método que probaremos

7) Instancia a nuestra de prueba

8) El resultado que esperamos obtener si la prueba tiene exito

9) la llamada al método de prueba, el resultado se almacena en otra variable

10) assertEquals: Es uno de varios métodos con los que cuenta JUnit, este método en particular compara si dos objetos son iguales, de no ser así, lanzará una excepción y la prueba se detie ne.

12) fail: Este método hace que la prueba termine con fallo

Creamos 4 métodos con sus respectivos valores de entrada y valores esperados, es decir:

@Test

public void testNumero\_mayor\_caso1() {

int a = 5;

int b = 3;

int c = 7;

MiClase instance = new MiClase();

int expResult = 7;

int result = instance.numero\_mayor(a, b, c);

assertEquals(expResult, result);

}

@Test

public void testNumero\_mayor\_caso2() {

int a = 5;

int b = 3;

int c = 4;

MiClase instance = new MiClase();

int expResult = 5;

int result = instance.numero\_mayor(a, b, c);

assertEquals(expResult, result);

}

@Test

public void testNumero\_mayor\_caso3() {

int a = 5;

int b = 7;

int c = 6;

MiClase instance = new MiClase();

int expResult = 7;

int result = instance.numero\_mayor(a, b, c);

assertEquals(expResult, result);

}

@Test

public void testNumero\_mayor\_caso4() {

int a = 5;

int b = 7;

int c = 9;

MiClase instance = new MiClase();

int expResult = 9;

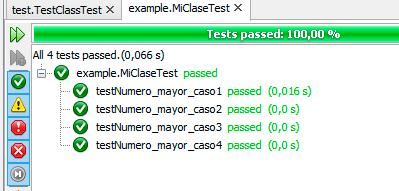
int result = instance.numero\_mayor(a, b, c);

assertEquals(expResult, result);

}

Para ejecutar el test con JUnit, clic derecho sobre la clase MiClaseTest.java -> run file

Observamos que pasamos las 4 pruebas con éxito.



Si tuviésemos algún error, este aparecería en color rojo.

**CONCLUSIONES:**

* Las pruebas unitarias deben ejecutarse sin necesidad de intervención manual y debe repetirse cuantas veces quieras.
* También tienen que poder ejecutarse independientemente del estado del entorno.
* Mejora la calidad del código, se reducirán los tiempos de depuración y la corrección de incidencias.
* Las pruebas nos ayudan a entender mejor el código, ya que sirven de documentación.
* En las pruebas se simulan las dependencias lo que nos permite que podamos probar nuestro código sin disponer del resto de los módulos.
* Una de sus ventajas es que los errores están más acotados y son más fáciles de encontrar. Dado que tienen pruebas unitarias que pueden desenmascáralos.
* Las pruebas nos permiten poder probar o depurar un módulo sin la necesidad de disponer del sistema completo.
* La primera vez que utilicemos las pruebas unitarias nos llevará algo de tiempo ya que tenemos que familiarizarnos con las herramientas y el Framework.

**Recomendación:**

Hagan Pruebas Unitarias, para tener una buena calidad de código y tener mayor seguridad de su trabajo