**Universidad Nacional de Ingeniería**

**FACULTAD DE INGENIERÍA DE SISTEMAS**



**TRABAJO INVESTIGATIVO**

**PRUEBAS UNITARIAS**

Trabajo presentado en cumplimiento parcial del curso de Java-Programación

**Autores:**

Cristobal Ar­tica, Luis Enrique

Pilco Quevedo, Jose

Mesares Huaman, Angelica

**Docente:**

Gustavo Coronel Castillo

**LIMA – PERÚ**

**2020**

# 1. Dedicatoria

A nuestros padres, que nos guían, nos apoyan en este camino, y sobre todo son aquellas personas que siempre están a nuestro lado dándonos aliento en este nuevo camino largo que hemos emprendido; a nuestro docente que nos imparte sus conocimientos permitiéndonos adquirir bases sólidas relacionadas en el mundo de la tecnología, y a nuestros compañeros con los cuales compartimos conocimientos para aprender y alcanzar nuestras metas trazadas.

# Índice

[1. Dedicatoria 2](#_Toc32995262)

[Índice 3](#_Toc32995263)

[3. Resumen 4](#_Toc32995264)

[4. Desarrollo del tema 5](#_Toc32995265)

[4.1. Contexto 5](#_Toc32995266)

[4.2. ¿A que llamamos pruebas unitarias? 5](#_Toc32995267)

[4.3. Importancia de las pruebas unitarias 6](#_Toc32995268)

[4.3.1. Beneficios 6](#_Toc32995269)

[4.4. Buenas prácticas para su realización 7](#_Toc32995270)

[4.4.1. Estructura de las pruebas unitarias 8](#_Toc32995271)

[4.5. Características que deben tener las pruebas unitarias 9](#_Toc32995272)

[5. Casos desarrollados 10](#_Toc32995273)

[5.1. CASO 01 10](#_Toc32995274)

[5.2. CASO 02 14](#_Toc32995275)

[5.3. CASO 03 16](#_Toc32995276)

[6. Conclusiones 18](#_Toc32995277)

[7. Recomendaciones 19](#_Toc32995278)

[8. Referencias 20](#_Toc32995279)

# 3. Resumen

En el presente estudio expondrá sobre las pruebas unitarias, en cómo estas son importantes para disminuir las pruebas en las fases posteriores; no obstante, son poco practicadas ya que demanda tiempo y costos, porque los casos de pruebas se generan habitualmente de forma manual.

Hemos realizado una introducción al tema, que realizamos una descripción de las ventajas que supone su adopción, incorporar información de utilidad verificada durante la implementación de la disciplina dentro de java

La introducción nos sirvió como base para describir de manera teórica la utilización de objetos u en los casos que es necesario simular los objetos utilizados por las clases cuyo funcionamiento necesitamos verificar.

En el caso de sistemas construidos bajo paradigma Orientado a Objetos, las pruebas unitarias se centran en la clase y una de las técnicas utilizada es la técnica de McGregor y Sykes. Basándose en esta técnica y en Diseño por Contrato con JML (Lenguaje de Modelamiento de Java), se diseña y construye un prototipo funcional de una herramienta que genera asistidamente casos de prueba unitarios Concepto de prueba de Software

# 4. Desarrollo del tema

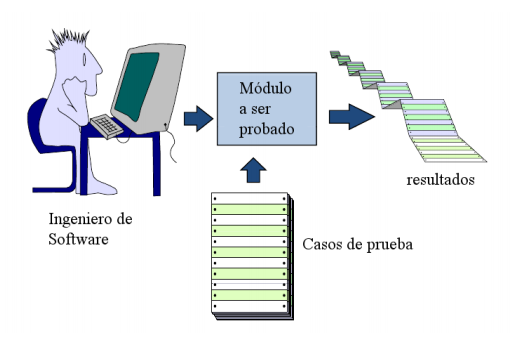
## 4.1. Contexto

El objetivo de las pruebas es presentar información sobre la calidad del producto a las personas responsables de este. Las pruebas de calidad presentarían los siguientes objetivos: encontrar defectos o bugs, aumentar la confianza en el nivel de calidad, facilitar información para la toma de decisiones, evitar la aparición de defectos.

Teniendo la afirmación anterior en mente, la información que puede ser requerida es de lo más variada. Esto hace que el proceso de *testing* sea completamente dependiente del contexto en el que se desarrolla.

El ambiente ideal de las pruebas es aquel que es independiente del desarrollo del software, de esta manera se logra objetividad en las pruebas.

A pesar de lo que muchos promueven, no existen las "mejores prácticas" como tales. Toda práctica puede ser ideal para una situación, pero completamente inútil o incluso perjudicial en otra.

Por esto, las actividades técnicas, documentación, enfoques y demás elementos que condicionarán las pruebas a realizar deben ser seleccionados y utilizados de la manera más eficiente según contexto del proyecto.

## 4.2. ¿A que llamamos pruebas unitarias?

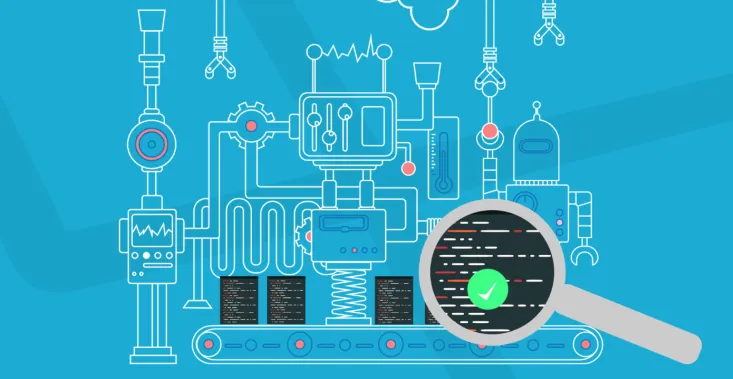
Según Yeepli (2018), una plataforma premium especializada en el desarrollo de apps móviles y sitios webs a medida, confirmo: “Las pruebas unitarias consisten en aislar una parte del código y comprobar que funciona a la perfección. Son pequeños *tests* que validan el comportamiento de un objeto y la lógica”. (párr. 3)

De acuerdo con Software Testing: Una prueba unitaria es un nivel de prueba de software en el que se prueban unidades o componentes individuales de un software para validar que cada unidad del software funcione según lo diseñado. Cabe indicar que una unidad es la parte comprobable más pequeña de cualquier software que en su mayoría tiene una o algunas entradas y una sola salida. En la programación de procedimientos, una unidad puede ser un programa individual, función, procedimiento, entre otros. En la programación orientada a objetos (POO), la unidad más pequeña es un método, que puede pertenecer a una clase base / superclase, clase abstracta o clase derivada / secundaria

De lo expuesto, podemos mencionar que las pruebas unitarias son un tipo de pruebas en donde se evalúan componentes individuales de un programa y tienen como propósito, garantizar que cada componente del programa funciona aisladamente, es decir, funciona como debe funcionar, responde como deber responder, falla como y cuando debe fallar, y acepta lo que tiene que aceptar.

## 4.3. Importancia de las pruebas unitarias

**Las pruebas unitarias** se utilizan para confirmar que una parte específica de código funciona de manera individual. Que hace “lo que se espera que haga”. Para ello, en un proyecto aislado del código general del programa, se establecen unos valores de entrada para la prueba. En la fase de ejecución, se calcula el valor que ese código devuelve con los parámetros establecidos. Por último, mediante los llamados asserts del entorno de trabajo N-Unit, se comprueba si se cumple con la previsión realizada y el código se comporta de la forma esperada.



### 4.3.1. Beneficios

A partir de su realización se puede obtener las ventajas que producen por si estas pruebas. Tenemos las siguientes:

* El estilo de vida de desarrollador será mucho más fácil, ya que la calidad de su código mejorará, se reducirán los tiempos de depuración y la corrección de incidencias y por tanto el cliente estará mucho más contento porque la aplicación hace lo que él quiere que haga, por lo que ha pagado.
* Las pruebas fomentan el cambio y la refactorización. Si consideremos que nuestro código es mejorable podemos cambiarlo sin ningún problema. Si el cambio no estuviera realizado correctamente las pruebas nos avisarán de ello. Seguramente la frase “si funciona no lo toques” a más de uno les resultará familiar. Si hubiera pruebas unitarias, no sería necesario pronunciarla.
* Se reducen drásticamente los problemas y tiempos dedicados a la integración. En las pruebas se simulan las dependencias lo que nos permite que podemos probar nuestro código sin disponer del resto de módulos. Por experiencia puede decir que los procesos de integración son más de una vez traumáticos, dejándolos habitualmente para el final del proyecto. La frase “sólo queda integrar” haciendo referencia a que el proyecto está cerca de terminar suele ser engañosa, ya que el periodo de integración suele estar lleno de curvas.
* Las pruebas nos ayudan a entender mejor el código, ya que sirven de documentación. A través de las pruebas podemos comprender mejor qué hace un módulo y que se espera de él. Nos permite poder probar o depurar un módulo sin necesidad de disponer del sistema completo. Aunque seamos los propietarios de toda la aplicación, en algunas situaciones montar un entorno para poder probar una incidencia es más costoso que corregir la incidencia propiamente dicha. Si partimos de la prueba unitaria podemos centrarnos en corregir el error de una forma más rápida y lógicamente, asegurándonos posteriormente que todo funciona según lo esperado.

## 4.4. Buenas prácticas para su realización

Cuando tiendes a realizar constantemente este tipo de pruebas irás descubriendo todas sus ventajas. Algunos ejemplos de buenas prácticas en las pruebas unitarias de software:

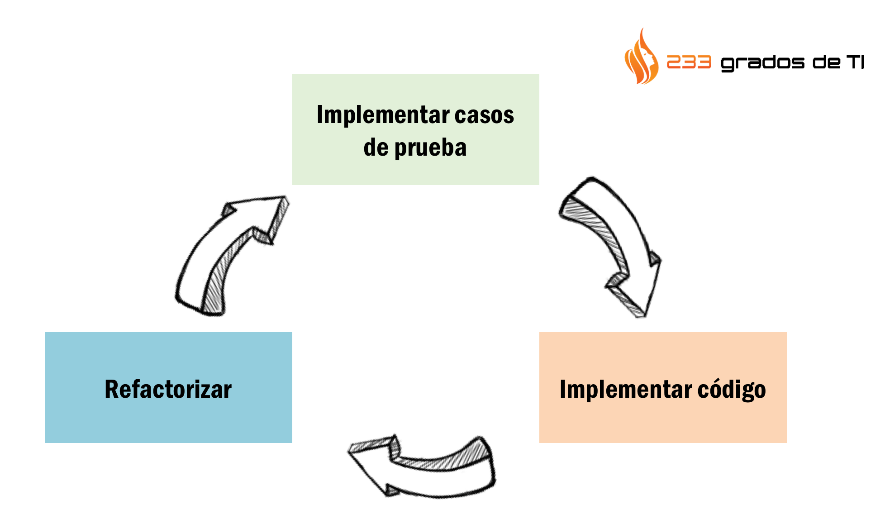
* Las pruebas unitarias deberían ser independientes. Si se produce cualquier tipo de mejora o cambio en los requerimientos, las pruebas unitarias no deberían verse afectados.
* Prueba sólo un código a la vez.
* Sigue un esquema claro. Puede parecer algo secundario, pero no lo es. Sé también consistente a la hora de nombrar tus *unit tests*.
* Cualquier cambio necesita pasar el test. En el caso de producirse un cambio en el código de cualquier módulo, asegúrate de que hay una prueba unitaria que se corresponda con ese módulo y que esta pasa las pruebas antes de cambiar la implementación.
* Corrige los bugs identificados durante las pruebas antes de continuar. Asegúrate de realizar esta corrección antes de proseguir con la siguiente fase del ciclo de vida del desarrollo de software.
* Acostúmbrate a realizar pruebas regularmente mientras programas. Cuanto más código escribas sin testar, más caminos tendrás que revisar para encontrar errores.



### 4.4.1. Estructura de las pruebas unitarias

Para llevar a cabo buenas pruebas unitarias, deben estar estructuradas siguiendo las tres A’s del Unit Testing (las A de la prueba unitarias). Se trata de un concepto fundamental respecto a este tipo de prueba**s**, que describe un proceso compuesto de tres pasos.

* Arrange (organizar). Es el primer paso de las pruebas unitarias. En esta parte se definen los requisitos que debe cumplir el código. En otras palabras, es la etapa donde se establecen los valores de entrada, asi como el valor esperado y cualquier otra preparación que se necesite para ejecutar la prueba unitaria.
* Act (actuar). Es el paso intermedio de las pruebas, el momento de ejecutar el *test* que dará lugar a los resultados que deberás analizar. En otras palabras, es la etapa donde la prueba se ejecuta o como se llama en una etapa de acción o ejecución.
* Assert (afirmar). En el último paso, es el momento de comprobar si los resultados obtenidos son los que se esperaban. Si es así, se valida y se sigue adelante. Si no, se corrige el error hasta que desaparezca. En otras palabras, es la etapa donde ser revisa que las condiciones esperadas se hayan cumplido, así como, que el método haya producido los cambios esperados o haya devuelto el valor esperado.



## 4.5. Características que deben tener las pruebas unitarias

* 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.
* Estas 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 simuladas para evitar dependencias entre módulos.
* Es importante conocer claramente cuál es el objetivo del test. Cualquier desarrollador debería poder conocer claramente cuál es el objetivo de la prueba y su funcionamiento. Esto sólo se consigue si se trata el código de pruebas como el código de la aplicación.
* También, 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

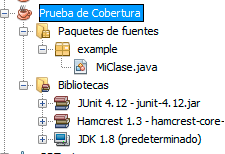
# 5. Casos desarrollados

Una Prueba Unitaria, es una forma de comprobar que nuestro código, hace lo que se supone debe hacer; es decir, se asegura que muestro código no presente fallos, errores, o cálculos inesperados, y que siempre retorne el valor correcto. Cada lenguaje de programación cuenta con sus propias herramientas para realizar estas pruebas, en el caso de java

## 5.1. CASO 01

**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«.

[](https://www.jc-mouse.net/wp-content/uploads/2017/10/cobertura_java.gif)

**Paso 2: El código**

El código que probaremos, lo estudiamos en un post anterior donde realizamos una prueba manual del algoritmo. Tenemos:

package example;

/\*\*

\* @see https://www.jc-mouse.net/

\* @author mouse

\*/

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 {

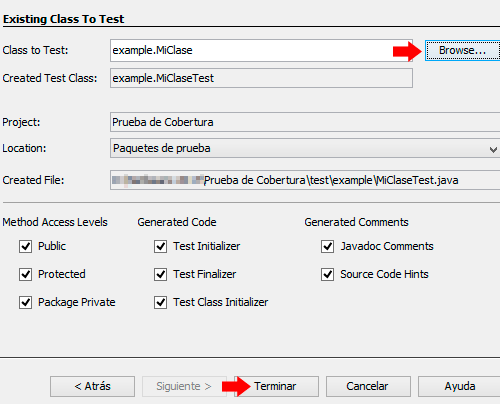
return b;

}

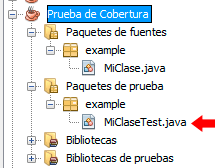
**Paso 3: Paquete de Prueba**

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».

[](https://www.jc-mouse.net/wp-content/uploads/2017/10/test_unit_class.gif)

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

[](https://www.jc-mouse.net/wp-content/uploads/2017/10/paquete-de-prueba.gif)

**Paso 4: JUnit**

Antes de realizar la prueba con JUnit, debemos estudiar un poco el código generado. 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

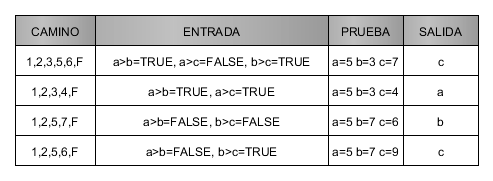
8) El resultado que esperamos obtener si la prueba tiene exito

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 detiene.

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

Como vimos en un post anterior [[Caja blanca: Prueba del camino básico](https://www.jc-mouse.net/ingenieria-de-sistemas/caja-blanca-prueba-del-camino-basico)], este algoritmo consta de 4 caminos posibles, eso quiere decir que debemos implementar 4 métodos de prueba.

[](https://www.jc-mouse.net/wp-content/uploads/2017/10/grafo-de-flujo-kawaii.gif)

Partiendo como base el método de prueba arriba mencionado, 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\_caso3() {

int a = 5;

int b = 7;

int c = 6;

MiClase instance = new MiClase();

int expResult = 7;

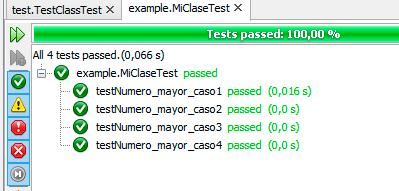
int result = instance.numero\_mayor(a, b, c);

assertEquals(expResult, result);

}

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

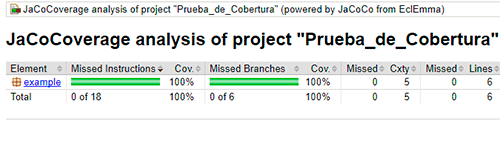
Observamos que pasamos las 4 pruebas con éxito.

[](https://www.jc-mouse.net/wp-content/uploads/2017/10/junit-test-java-example.gif)

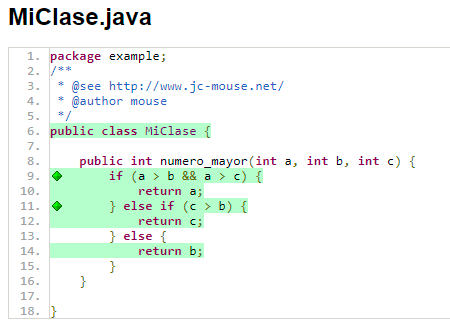
Si tuviésemos algún error, este aparecería en color rojo.

**Paso 5: JaCoCoverage**

Como mencionamos en un principio, este plugin nos permite ver más a detalle lo que pasa con el código y los caminos que este toma. Para ejecutarlo, clic derecho sobre el proyecto -> Test with JaCoCoverage. Se abrirá una página en el navegador que tengamos configurado por defecto.

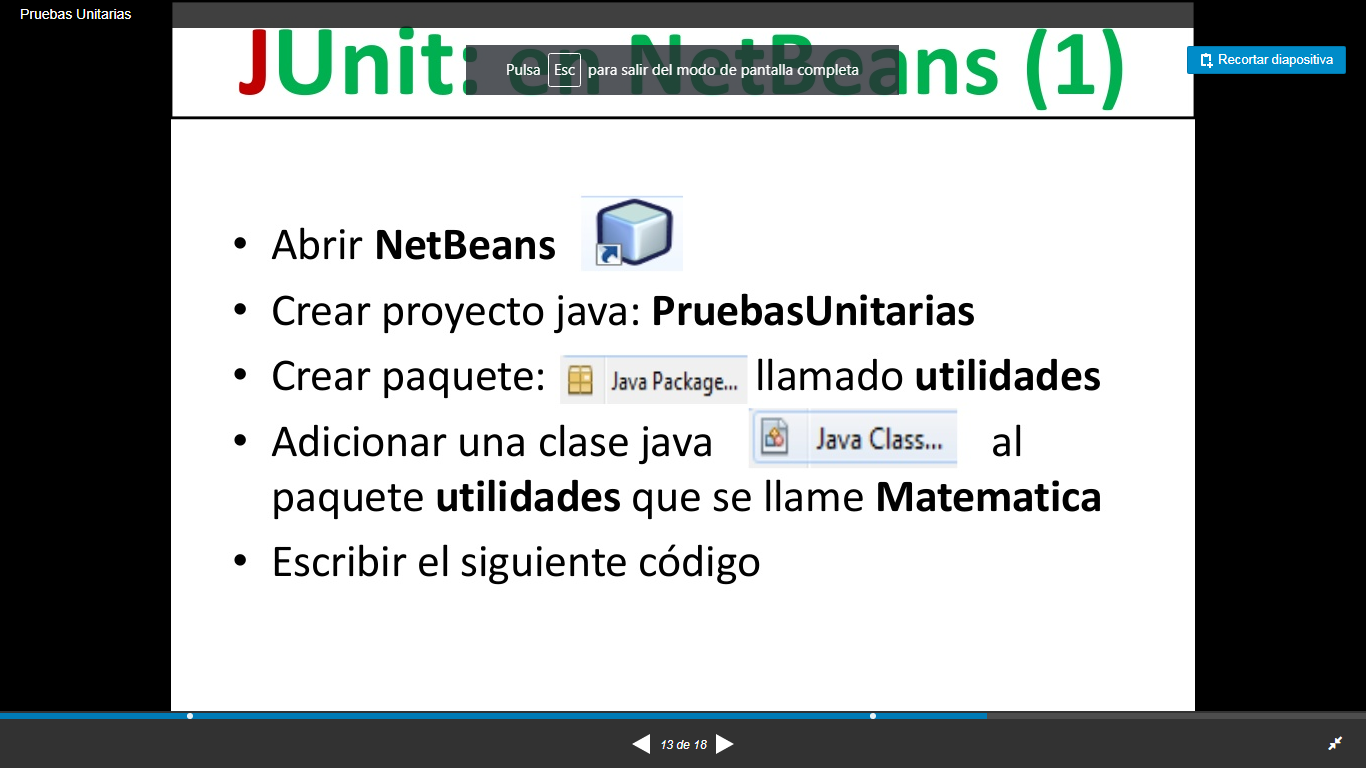
[](https://www.jc-mouse.net/wp-content/uploads/2017/10/java_analisis_unitario.gif)

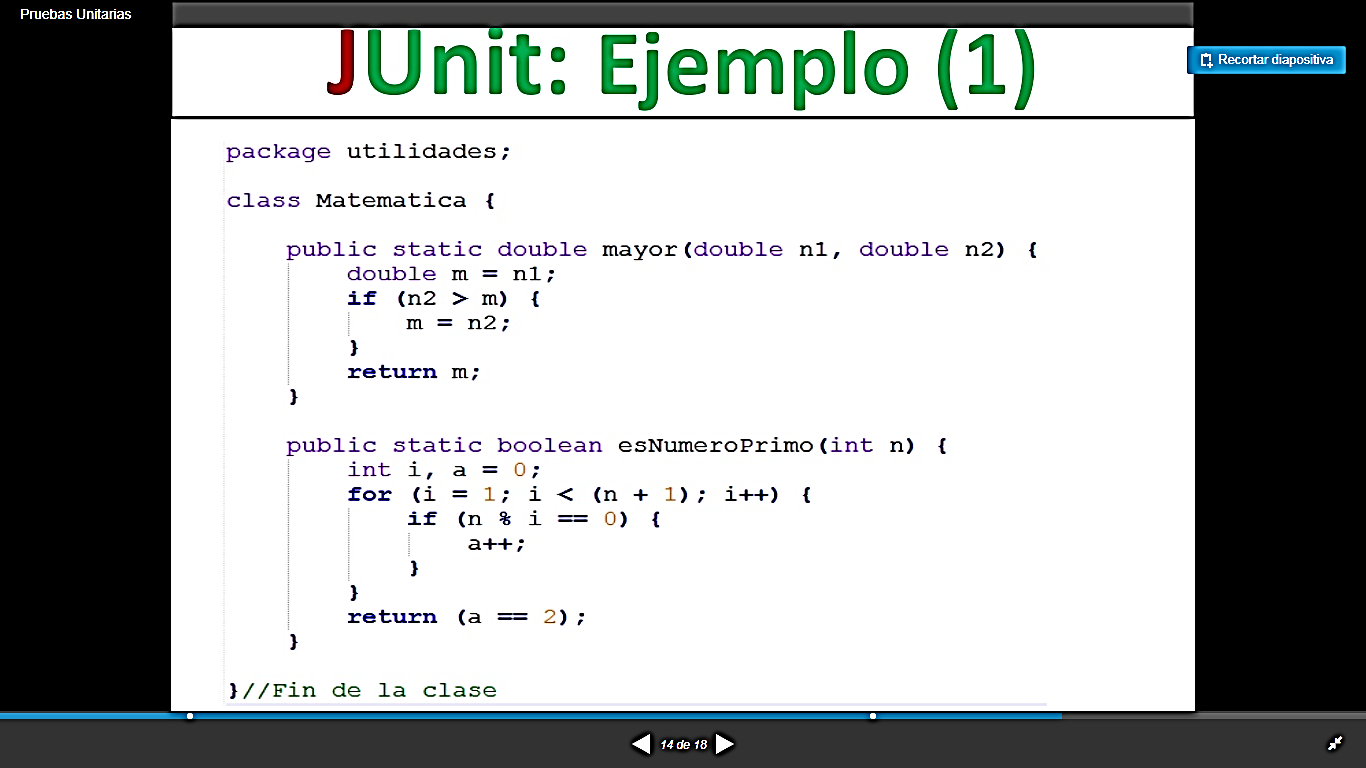
En esta página, el plugin nos brinda información importante sobre el test realizado, ademas podemos navegar por las diferentes clases que cuenta nuestro proyecto. En nuestro ejemplo pasamos todas las pruebas por lo que el código se colorea de verde.

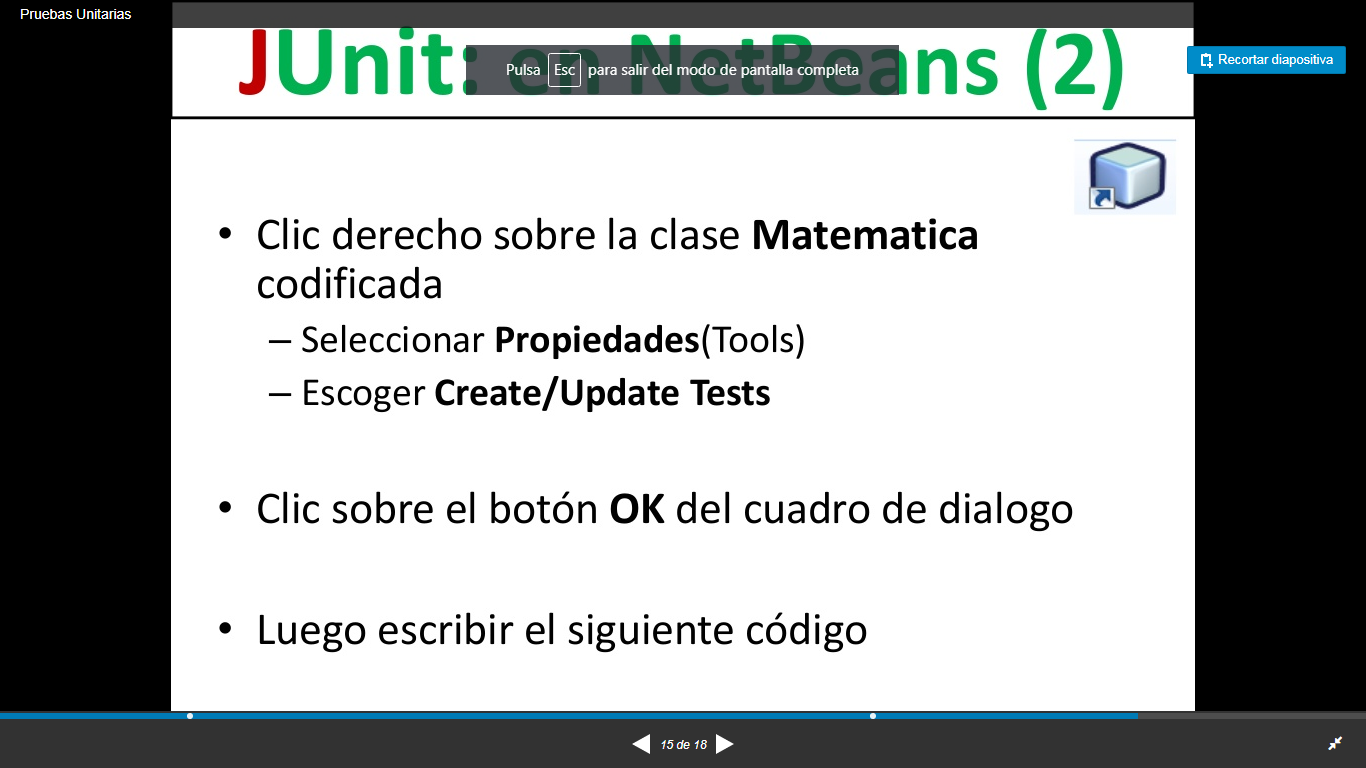
[](https://www.jc-mouse.net/wp-content/uploads/2017/10/unitary_test_java.gif)

## 5.2. CASO 02

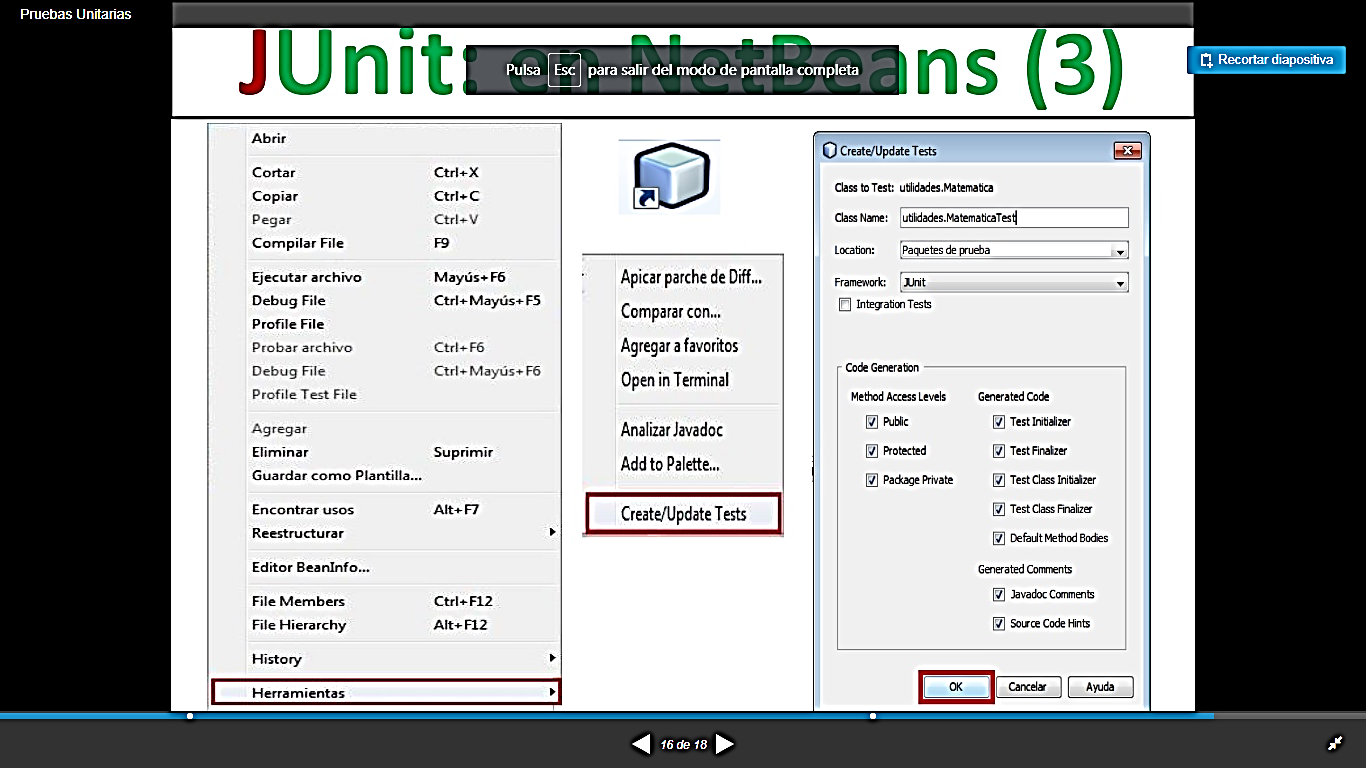
**PASO 1:**

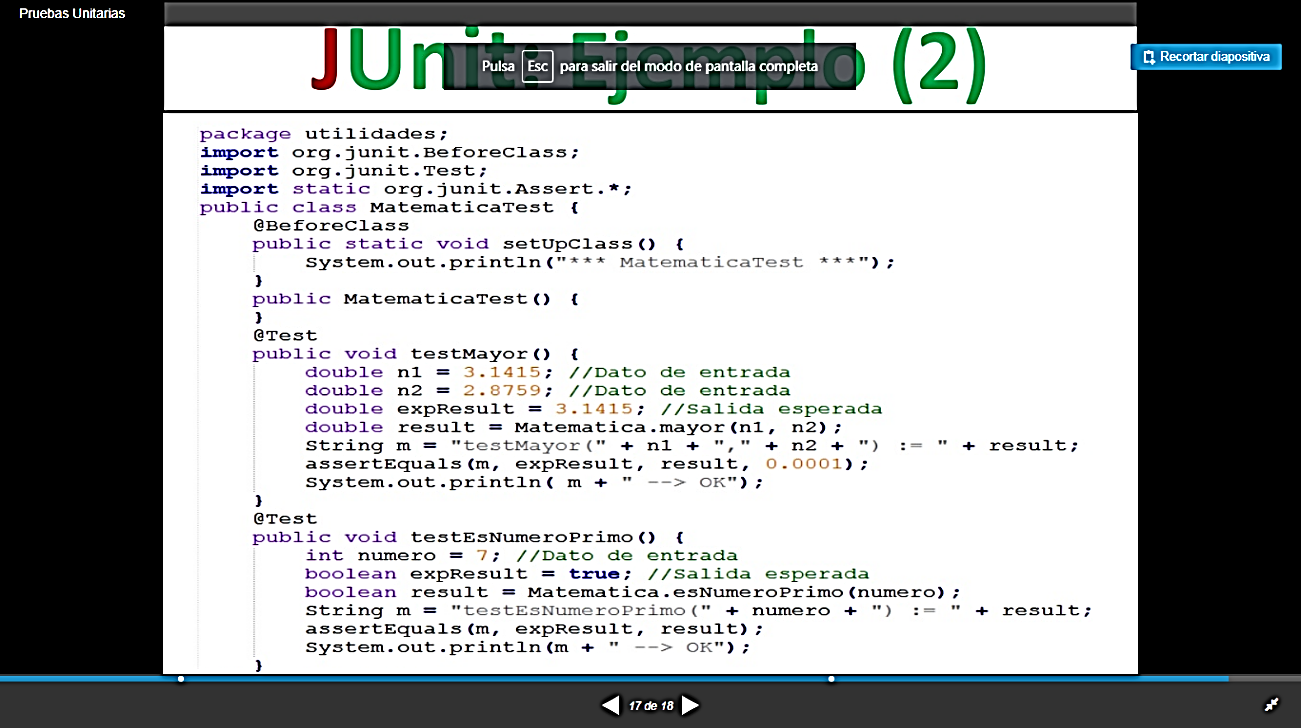


PASO 2: Escribir el siguiente código

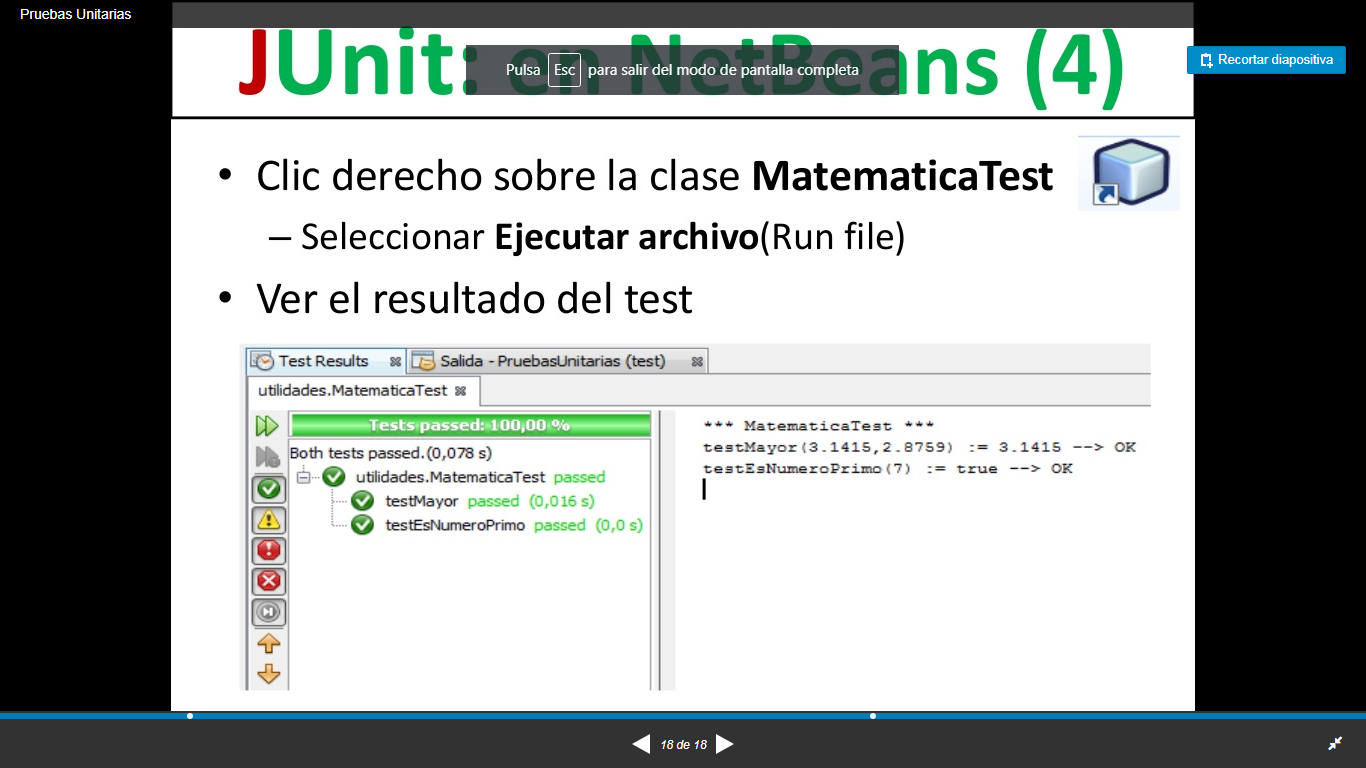
**PASO 3:**

**PASO 4:**

****

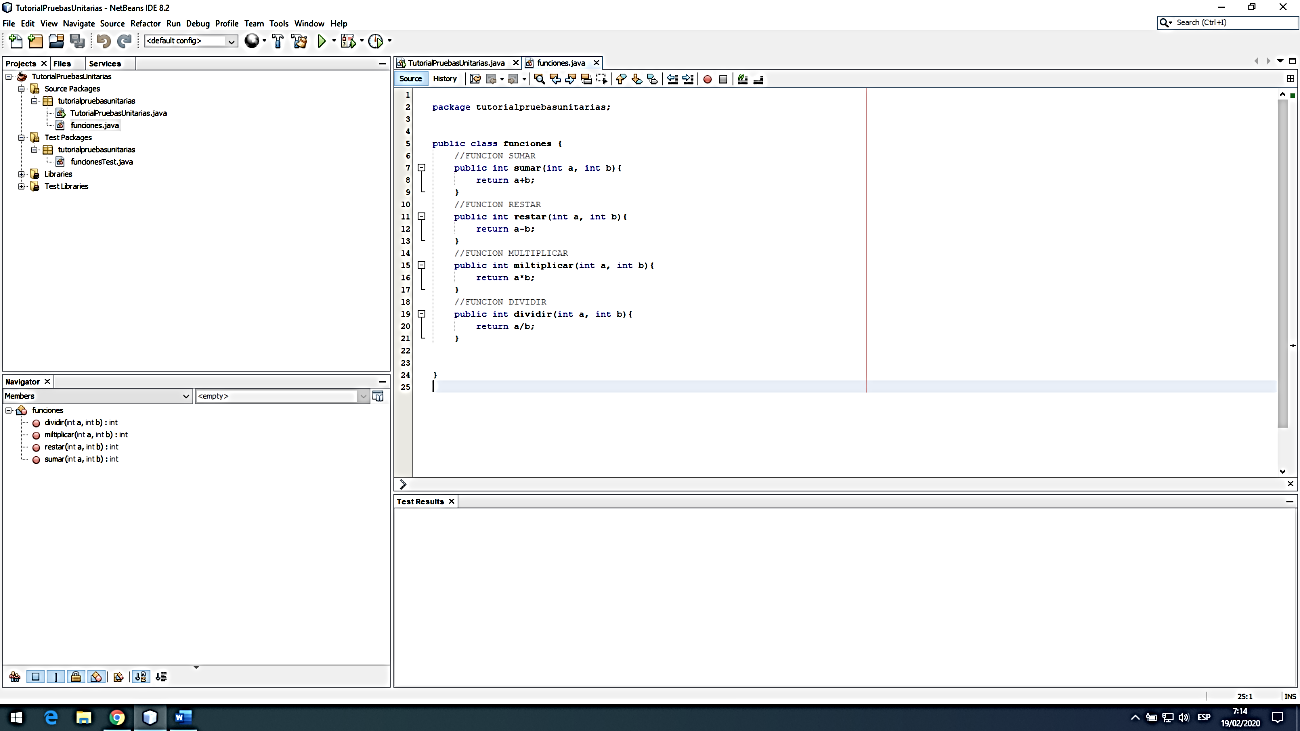
**PASO 5:**

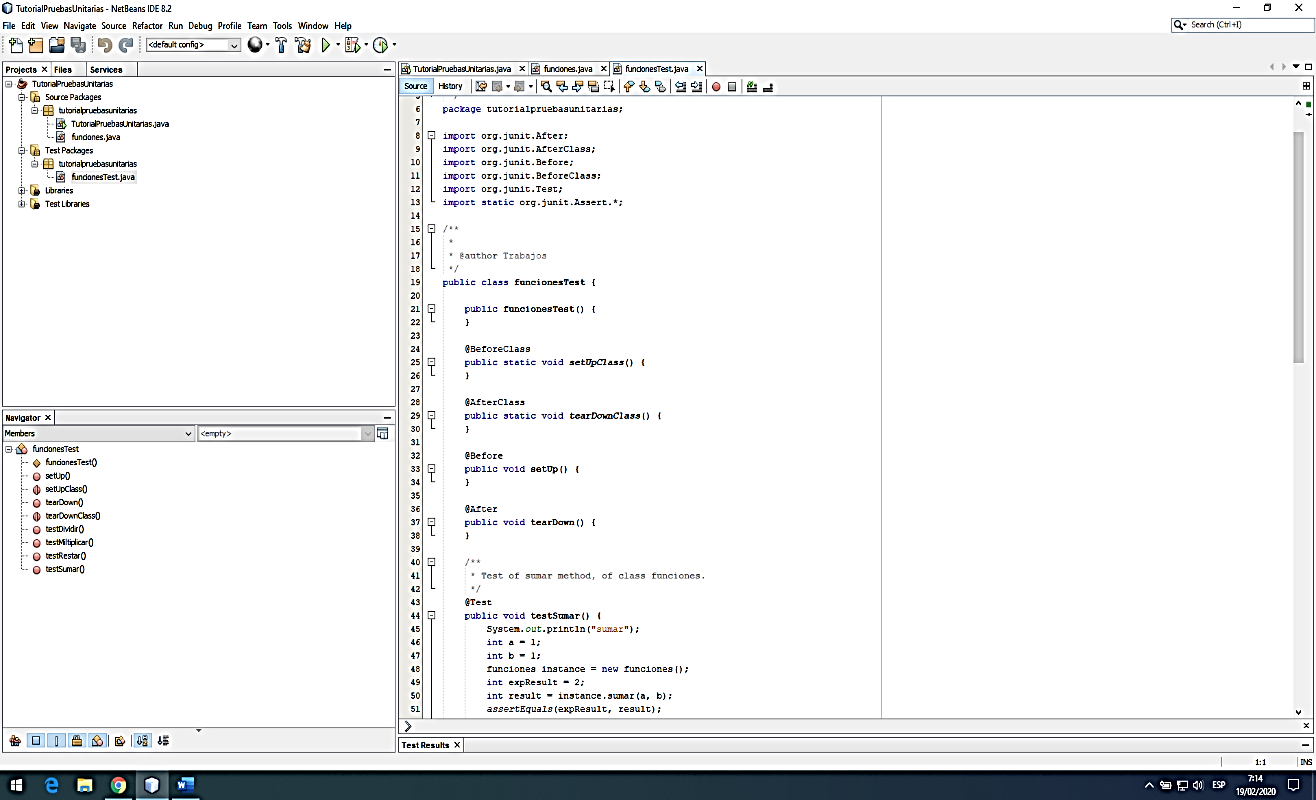
**PASO 7:**

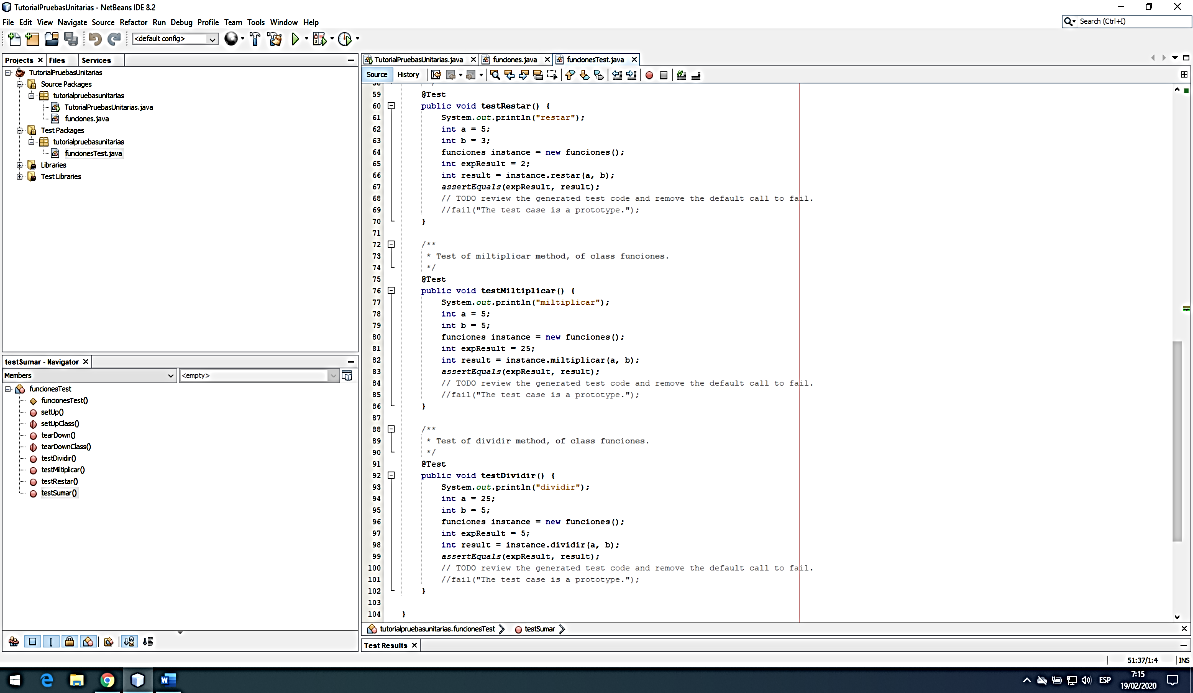


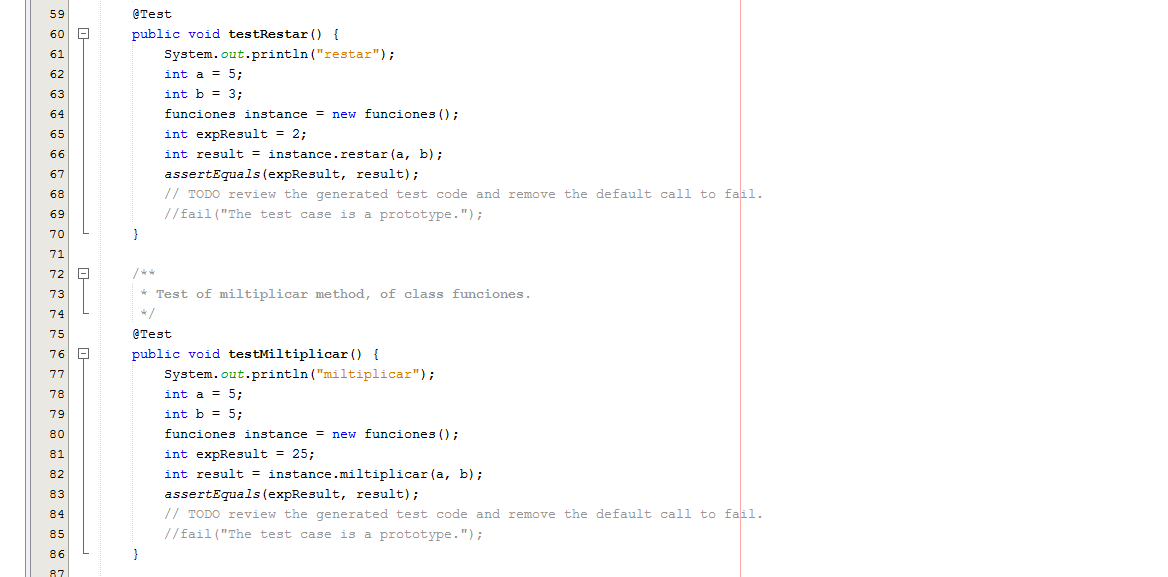
## 5.3. CASO 03

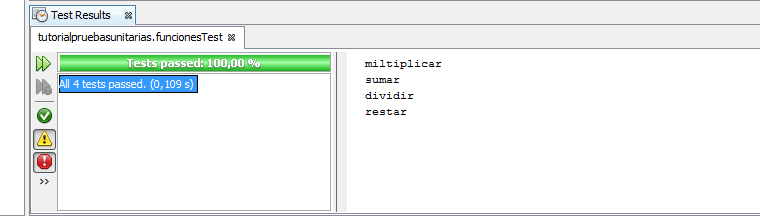
CLASE FUNCIONES: Operadores Aritméticos



**TEST:**



****

**RESULTADOS**

# 6. Conclusiones

Después de lo expuesto podemos concluir que el objetivo fundamental de las pruebas unitarias es asegurar el correcto funcionamiento de las interfaces o flujo de datos entre componentes de manera tal que a la hora de realizar una unificación de los diferentes componentes que conforman el sistema en general, que favorezca el desarrollo de la aplicación que se quiere realiza java.

Escribir pruebas automáticas son muy importantes sobre el cual se apoyan las otras prácticas, que juntas va a permitir mejorar mucho el proceso de desarrollo y la calidad del sistema.

Las pruebas unitarias cumplen con un estándar para que sean de mayor comprensión para el programador y sea más fácil lograr una automatización en las etapas que se requieren en el programa.

Escribir pruebas automáticas es seguramente el pilar más importante sobre el cual se apoyan las otras prácticas, que juntas van a permitir mejorar mucho el proceso de desarrollo y la calidad del sistema.

# 7. Recomendaciones

Se expondrá posteriormente algunas recomendaciones relacionadas al empleo de las pruebas unitarias, para buscar una mejor detección de analogías en el proceso de los programas

* Las pruebas unitarias demuestran que la lógica del código está en buen estado y que funcionará en todos los casos.
* Aumentan la legibilidad del código y ayudan a los desarrolladores a entender el código base, lo que facilita hacer cambios más rápidamente.
* Los test unitarios bien realizados sirven como documentación del proyecto.
* Se realizan en pocos milisegundos, por lo que podrás realizar cientos de ellas en muy poco tiempo.
* Las pruebas unitarias permiten al desarrollador refactorizar el código más adelante y tener la garantía de que el módulo sigue funcionando correctamente. Para ello se escriben casos de prueba para todas las funciones y métodos, para que cada vez que un cambio provoque un error, sea posible identificarlo y repararlo rápidamente.
* Considere mejorar la calidad final del código ya que, al estar realizando pruebas de manera continua, al finalizar el código será limpio y de calidad.
* Se debe considerar la estructura de etapas que necesiten las pruebas unitarias para la buena ejecución de los programas.

# 8. Referencias

<https://www.yeeply.com/blog/que-son-pruebas-unitarias/#practicas>

Pruebas unitarias en netbeans.PDF

Pruebas unitarias.WORD

<https://si.ua.es/es/documentacion/c-sharp/documentos/pruebas/07pruebasunitarias.pdf>

<https://www.fundanet.es/blog/pruebas-unitarias-y-de-integracion/>