

**TEMA MONOGRAFICO:**

PRUEBAS UNITARIAS

**INTEGRANTES:**

MALDONADO BARRANTES, BRUNO

RIVERA SANCHEZ, MARJORIT

**CURSO**

APLICACIONES MÓVILES

**PROFESOR:**

CORONEL CASTILLO, ERIC GUSTAVO

**2017**

# 

# **Dedicatoria**

Este trabajo es dedicado a las personas que desarrollan un proyecto con las ganas de ser un programador profesional. A mi familia que sin ayuda desinteresada me brindan los medios por el cual puedo seguir mi pasión. Finalmente a mis compañeros y a mi centro de estudios, que a pesar de las diferencias y tropiezos en el transcurso del ciclo, nos mantuvimos firmes y logramos terminar lo que habíamos empezado.

# **Resumen**

Esta monografía trata sobre un medio para lograr facilitar la ayuda en programas para el diseño y creación de aplicaciones móviles. Mediante búsquedas para saber dónde está el problema, a su vez poder arreglarlo, con la implementación de comandos y códigos simples. El código del que se va a hablar en esta monografía se llama “Pruebas Unitarias”, un código que se tiene que agregar al cuerpo de la aplicación, para poder verificarla, probarla y encontrar rápidamente donde está el problema. Todo esto debido a que cuando el programa encuentra un error, simplemente te muestra en qué lugar se encuentra el error de sintaxis, no te dice cuál es el problema y como arreglarlo. Es por ello que se plantea la implementación de este medio para una rápida solución.

**INDICE**

[**Dedicatoria** 2](#_Toc486076786)

[**Resumen** 3](#_Toc486076787)

[**1.** **Introducción** 5](#_Toc486076788)

[**2.** **Desarrollo** 6](#_Toc486076789)

[**3.** **Características** 7](#_Toc486076790)

[*3.1.* *Requisitos* 7](#_Toc486076791)

[*3.2.* *Ventajas* 8](#_Toc486076792)

[*3.3.* *Limitaciones* 8](#_Toc486076793)

[**4.** **Tipos de Prueba** 9](#_Toc486076794)

[*4.1.* *Caja blanca* 9](#_Toc486076795)

[4.2. *Caja negra:* 10](#_Toc486076796)

[*4.3.* *Pruebas de Integración* 10](#_Toc486076797)

[*4.4.* *Pruebas de aceptación* 11](#_Toc486076798)

[4.5. *Pruebas de regresión* 11](#_Toc486076799)

[**5.** **JUnit** 12](#_Toc486076800)

*5.1 Ejemplo1..............................................................................................................13*

*5.2 Ejemplo2..............................................................................................................16*

[**6.** **Conclusiones**](#_Toc486076801) 22

[**7.** **Referencias Bibliográficas**](#_Toc486076802) 23

# **Introducción**

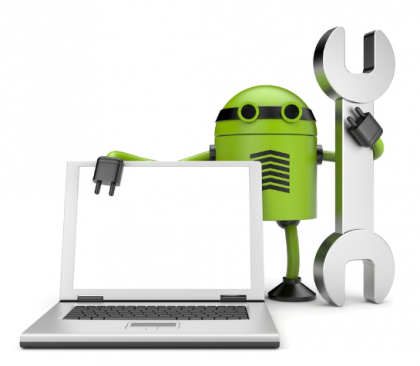
Este código te ayuda diciendo cual es el problema que puede haber en tu aplicación, donde se encuentra y a la vez, dar una solución. Este código es usado en muchas aplicaciones “Pesadas” (aplicaciones grandes, con muchas páginas, códigos y comandos). Este código no interrumpe, modifica o cambia el uso de los demás códigos o comandos que se le agreguen a la aplicación. Solo lo revisa y mira si todos los comandos funcionan perfectamente uno con otro. Sirve para todos los programa de diseño de aplicaciones como Android Studio, otras plataformas como Netbeans, Eclipse y más.

Al implementar este código les mostrara que su aspecto, el cual es fácil de manejare intuitivo, como usarlo y ejemplos en los que se puede sacar mayor provecho su uso. El uso es a gusto del usuario, puede aplicarlo para las aplicaciones que quieran crear en el futuro o para aplicaciones en las que ya estén trabajando, porque este código no necesita ser puesto al principio de la aplicación, se puede poner en cualquier momento y dará igual sus resultados. Ahora se mostrara lo principal sobre este código y cómo usarlo a la perfección.

# **Desarrollo**

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) se puede usar una función o un procedimiento, por otro lado también en [diseño orientado a objetos](https://es.wikipedia.org/wiki/Dise%C3%B1o_orientado_a_objetos). 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 tipos de parámetros y el tipo de respuesta que se devolverá. Finalmente verificar si el estado inicial es válido, por lo tanto si todo se cumple normal, el estado final arrojara el resultado valido.

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.



# **Características**

## *Requisitos*

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 continua](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.

## *Limitaciones*

Es importante darse cuenta de que las pruebas unitarias no descubrirán todos los errores del código. Algunos enfoques se basan en la generación aleatoria de objetos para amplificar el alcance de las pruebas de unidad. Esta técnica se conoce como [testing aleatorio](https://es.wikipedia.org/wiki/Testing_aleatorio). Por definición, sólo prueban las unidades por sí solas. Por lo tanto, no descubrirán errores de integración, problemas de rendimiento y otros problemas que afectan a todo el sistema en su conjunto. Además, puede no ser trivial anticipar todos los casos especiales de entradas, que puede recibir en realidad la unidad de programa bajo estudio. Las pruebas unitarias sólo son efectivas si se usan en conjunto con otras [pruebas de software](https://es.wikipedia.org/wiki/Prueba_de_software).

## 

# **Tipos de Prueba**

* Pruebas de caja blanca
* Pruebas de caja negra
* Pruebas de integración
* Pruebas de aceptación
* Pruebas de regresión

## *Caja blanca*

(Analiza sus propios códigos):

Cajas blancas es un tipo de [pruebas de software](https://es.wikipedia.org/wiki/Pruebas_de_caja_blanca) que se realiza sobre las funciones internas de un [módulo](https://es.wikipedia.org/wiki/M%C3%B3dulo_(inform%C3%A1tica)). Así como las pruebas de [caja negra](https://es.wikipedia.org/wiki/Caja_negra_(sistemas)) ejercitan los [requisitos funcionales](https://es.wikipedia.org/wiki/Requerimiento_funcional) desde el exterior del módulo, las de caja blanca están dirigidas a las funciones internas.

Las pruebas de caja blanca se llevan a cabo, sobre un módulo concreto, para luego realizar las de caja negra sobre varios subsistemas (integración).

En los sistemas orientados a objetos, las pruebas de caja blanca pueden aplicarse a los métodos de la clase, pero según varias opiniones, ese esfuerzo debería dedicarse a otro tipo de pruebas más especializadas (un argumento podría ser que los métodos de una clase suelen ser menos complejos que los de una función de programación estructurada). Dentro de las Pruebas de Caja Blanca encontramos las llamadas coberturas (sentencia, decisión, condición y múltiple además de los mencionados caminos ciclomáticos propuestos por McCabe).

Ejemplos típicos de ello son las pruebas unitarias. Se centran en lo que hay codificado o diseñado a bajo nivel por lo que no es necesario conocer la especificación de requisitos, que por otra parte será difícil de relacionar con partes diseñadas a muy bajo nivel.

## *Caja negra:*

(Sin ver el código, prueba la funcionalidad según sus especificaciones):

Se denomina Caja Negra a aquel elemento que es estudiado desde el punto de vista de las entradas que recibe y las salidas o respuestas que produce, sin tener en cuenta su funcionamiento interno. En otras palabras, de una caja negra nos interesará su forma de interactuar con el medio que le rodea (en ocasiones, otros elementos que también podrían ser cajas negras) entendiendo qué es lo que hace, pero sin dar importancia a cómo lo hace. Por tanto, de una caja negra deben estar muy bien definidas sus entradas y salidas, es decir, su [interfaz](https://es.wikipedia.org/wiki/Interfaz); en cambio, no se precisa definir ni conocer los detalles internos de su funcionamiento.

Un [sistema](https://es.wikipedia.org/wiki/Sistema) formado por módulos que cumplan las características de caja negra será más fácil de entender ya que permitirá dar una visión más clara del conjunto. El sistema también será más robusto y fácil de mantener, en caso de ocurrir un fallo, éste podrá ser aislado y abordado más ágilmente.

Ejemplos de pruebas de caja negra son la comprobación de pruebas de integridad de la base de datos, pruebas de situaciones de excepción, o pruebas de rendimiento del sistema. Presentan una limitación en cuanto a que es prácticamente imposible reproducir todo el espectro por la innumerable cantidad de combinaciones de entrada posibles, agravada por el desconocimiento de la lógica interna.

## *Pruebas de Integración*

(Prueba cómo funciona el sistema completo, compuesto por módulos distintos):

Pruebas de integración son aquellas que se realizan en el ámbito del [desarrollo de software](https://es.wikipedia.org/wiki/Desarrollo_de_software) una vez que se han aprobado las [pruebas unitarias](https://es.wikipedia.org/wiki/Prueba_unitaria) y lo que prueban es que todos los elementos unitarios que componen el software, funcionan juntos correctamente probándolos en grupo. Se centra principalmente en probar la comunicación entre los componentes y sus comunicaciones ya sea hardware o software.

La prueba de subsistemas es un tipo de prueba de integración donde se prueba el contenido de un subsistema.

## *Pruebas de aceptación*

Las pruebas de aceptación se realizan para establecer el grado de confianza en un sistema, ésta estará determinada por su grado de adherencia a las necesidades, requerimientos y procesos de negocio solicitados por el usuario o cliente. Es en función a estos que el usuario debe decidir si acepta o no el sistema que le está siendo entregado.

Por lo tanto, las pruebas de aceptación suelen ser responsabilidad de los clientes o usuarios del sistema. Otros interesados del proyecto pueden involucrarse también.

## *Pruebas de regresión*

Son el tipo de pruebas de [software](https://es.wikipedia.org/wiki/Software) que intentan descubrir errores, carencias de funcionalidad, causados por la realización de un cambio en el programa. Se evalúa el correcto funcionamiento del software desarrollado frente a evoluciones o cambios funcionales. El propósito de éstas es asegurar que los casos de prueba que ya habían sido probados y fueron exitosos permanezcan así. Se recomienda que este tipo de pruebas sean automatizadas para reducir el tiempo y esfuerzo en su ejecución.

Este tipo de cambio puede ser debido a prácticas no adecuadas de [control de versiones](https://es.wikipedia.org/wiki/Control_de_versiones). Por lo tanto, en la mayoría de las situaciones del desarrollo de software se considera una buena práctica que cuando se localiza y corrige un bug, se grabe una prueba que exponga el bug y se vuelvan a probar regularmente después de los cambios subsiguientes que experimente el programa.

Existen herramientas de software que permiten detectar este tipo de errores, la práctica habitual en [programación extrema](https://es.wikipedia.org/wiki/Programaci%C3%B3n_extrema) es que este tipo de pruebas se ejecuten en cada uno de los pasos del ciclo de vida del desarrollo del software.

# **JUnit**

JUnit es un conjunto de bibliotecas creadas por [Erich Gamma](https://es.wikipedia.org/wiki/Erich_Gamma) y [Kent Beck](https://es.wikipedia.org/wiki/Kent_Beck) que son utilizadas en [programación](https://es.wikipedia.org/wiki/Programaci%C3%B3n) para hacer [pruebas unitarias](https://es.wikipedia.org/wiki/Prueba_unitaria) de aplicaciones [Java](https://es.wikipedia.org/wiki/Lenguaje_de_programaci%C3%B3n_Java).

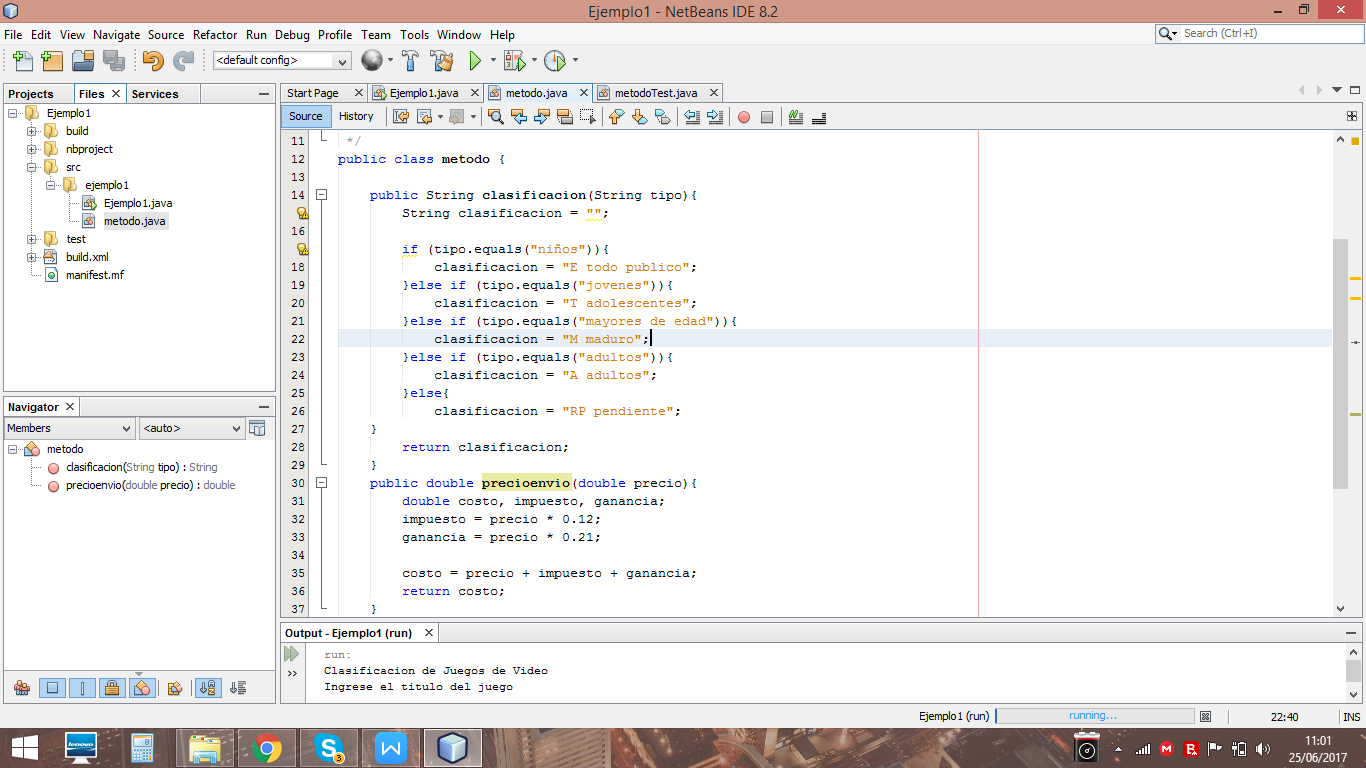
JUnit es un conjunto de clases ([framework](https://es.wikipedia.org/wiki/Framework)) que permite realizar la ejecución de clases Java de manera controlada, para poder evaluar si el funcionamiento de cada uno de los métodos de la clase se comporta como se espera. Es decir, en función de algún valor de entrada se evalúa el valor de retorno esperado; si la clase cumple con la especificación, entonces JUnit devolverá que el método de la clase pasó exitosamente la prueba; en caso de que el valor esperado sea diferente al que regresó el método durante la ejecución, JUnit devolverá un fallo en el método correspondiente.

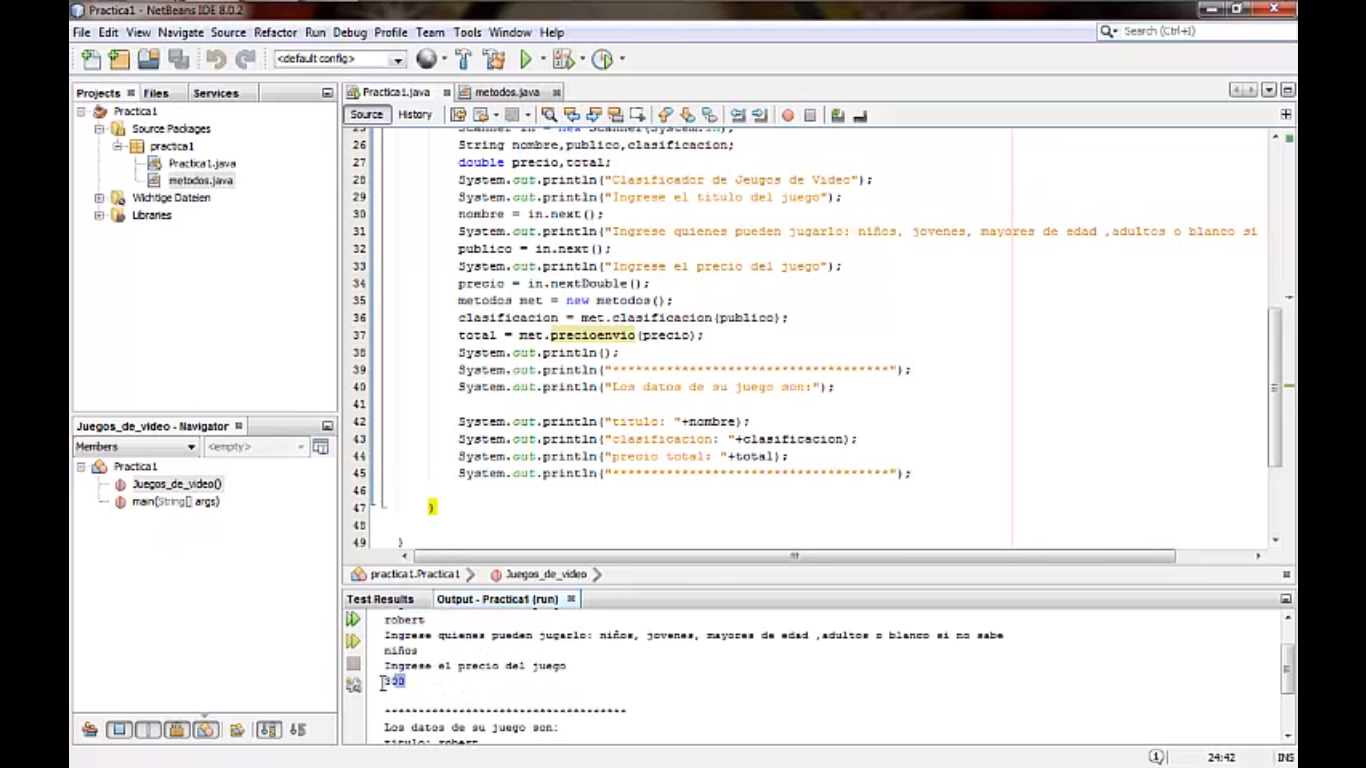
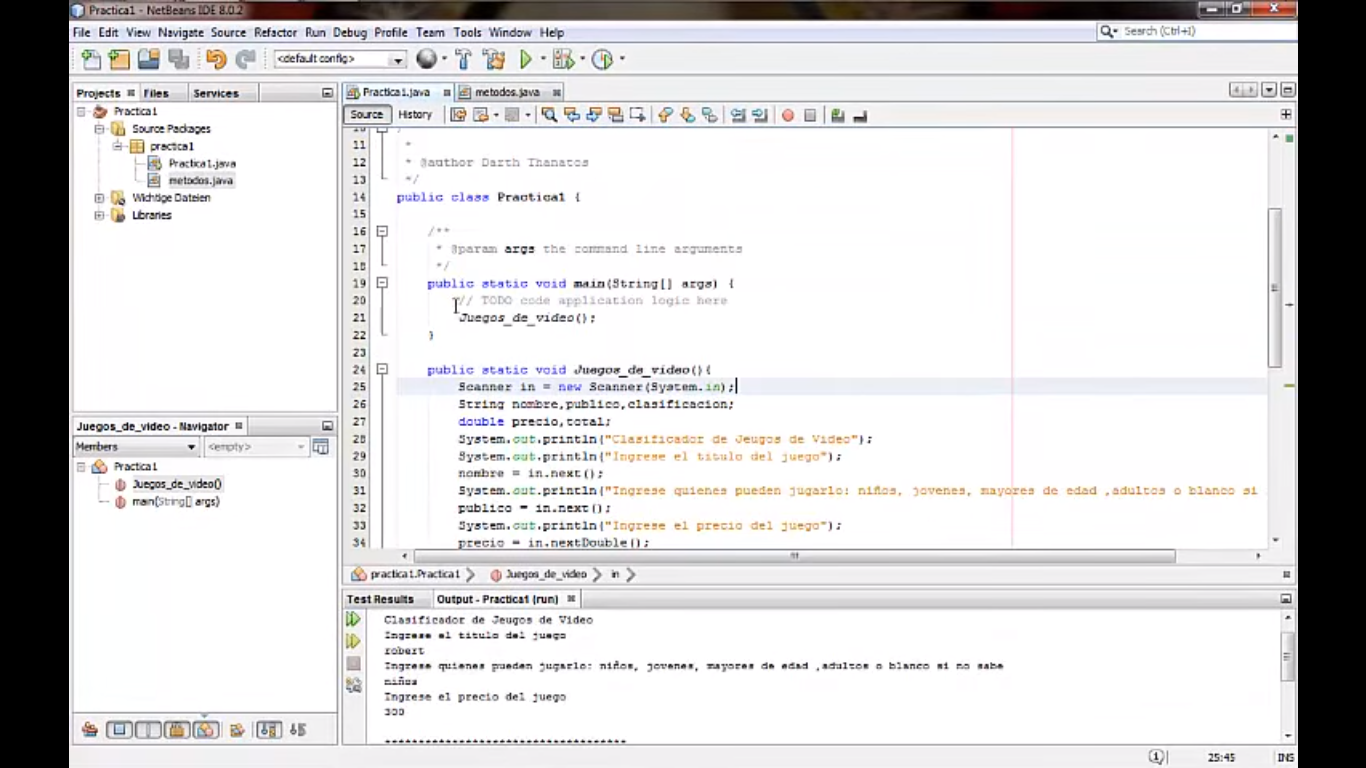
JUnit es también un medio de controlar las [pruebas de regresión](https://es.wikipedia.org/wiki/Pruebas_de_regresi%C3%B3n), necesarias cuando una parte del código ha sido modificado y se desea ver que el nuevo código cumple con los requerimientos anteriores y que no se ha alterado su funcionalidad después de la nueva modificación.

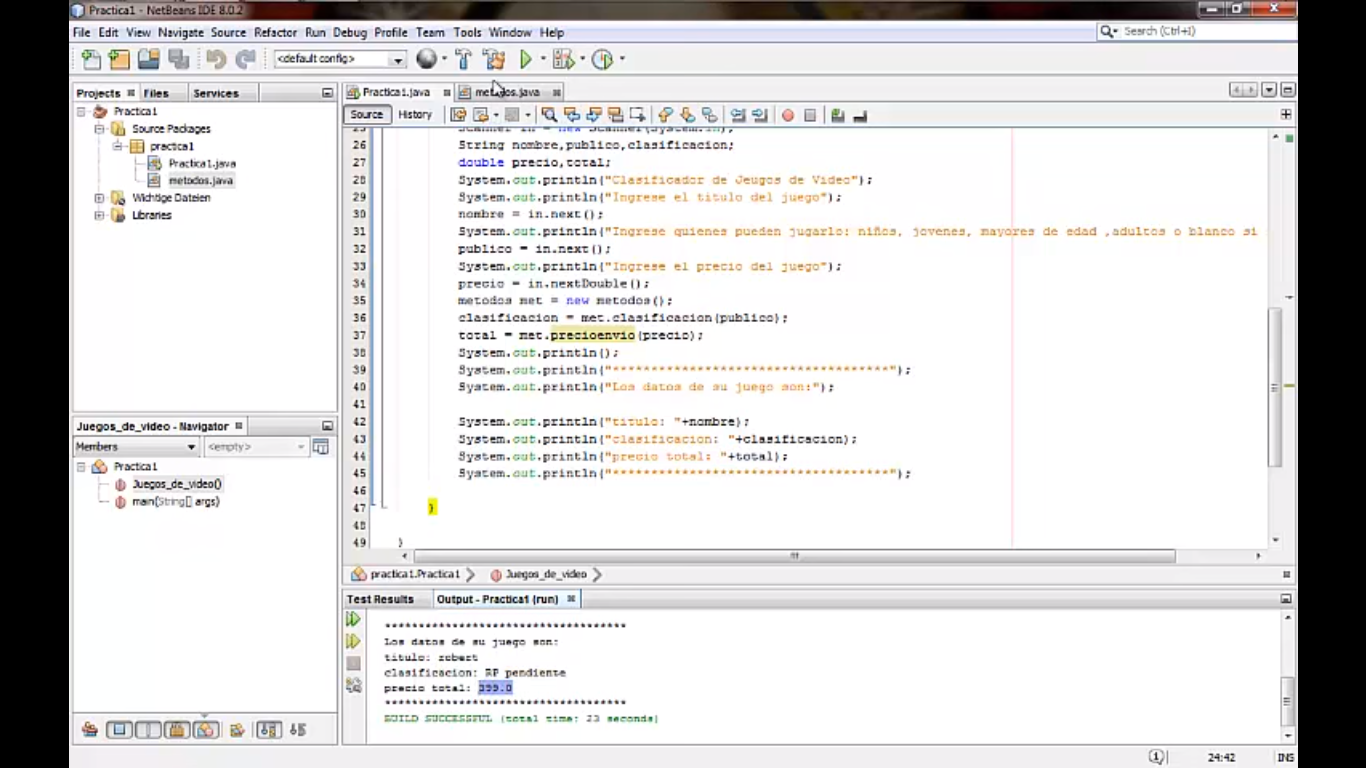
En la actualidad las herramientas de desarrollo como [NetBeans](https://es.wikipedia.org/wiki/NetBeans) y [Eclipse](https://es.wikipedia.org/wiki/Eclipse_(software)) cuentan con plug-ins que permiten la generación de las plantillas necesarias para la creación de las pruebas de una clase Java. De esta manera obtendremos su realización de manera automática, facilitando al programador enfocarse en la prueba y el resultado esperado. De esta manera, dejando a la herramienta la creación de las clases que permiten coordinar las pruebas



# *5.1 Ejemplo 1*

* Aquí veremos la creación y ejecución de pruebas unitarias usando JUnit en Netbeans en un proyecto java.
* En este proyecto tenemos 2 clases: La clase clase main que manda a llamar a Juegos\_de\_video().   
  Éste es un menú muy sencillo, donde nos dice el titulo de un juego, su precio y por último nos da un registro donde se puede ver el título, la clasificación y un precio total.



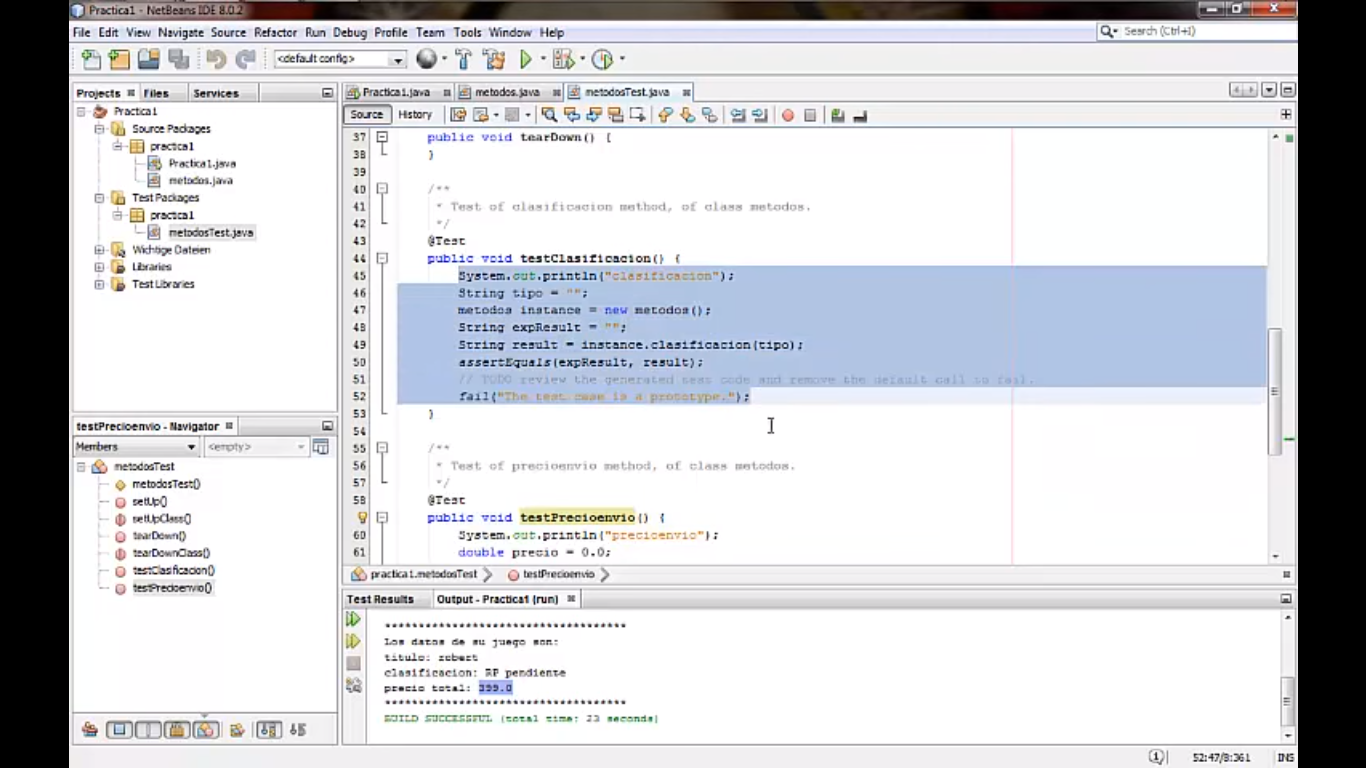


# Los métodos van a crear un método auxiliar en la cual se va a poder testear la entrada y salida de todos los datos. Netbeans en sus versiones más recientes trae configura el plugins para realizar pruebas unitarias, para ello utilizaremos el framework JUnit.

# 

# En esta opción nos va salir varios tipos de generación de métodos y nos da oportunidad incluso de crear pruebras de integración con el “Integration Test” pero no es el caso, también nos muestra los framework disponibles y en este caso utilizaremos el JUnit.

# Nos generó un nuevo test packages. Aquí a creado métodos para cada una de las funciones que teníamos establecidas en nuestro anterior clase, ahora a la primera no podremos generar nuestras pruebas unitarias la razón es que la clase que nos crea nos pide un tipo de entrada que es la respuesta que esperamos, instancia a la clase métodos y el result va a llamar el método de la clase y vamos a tener que compararlo, en este momento no tenemos ningún modo de compararlo entonces tendremos que realizar un if para ver si la respuesta esperada es correcta y meter el fail dentro de ese if

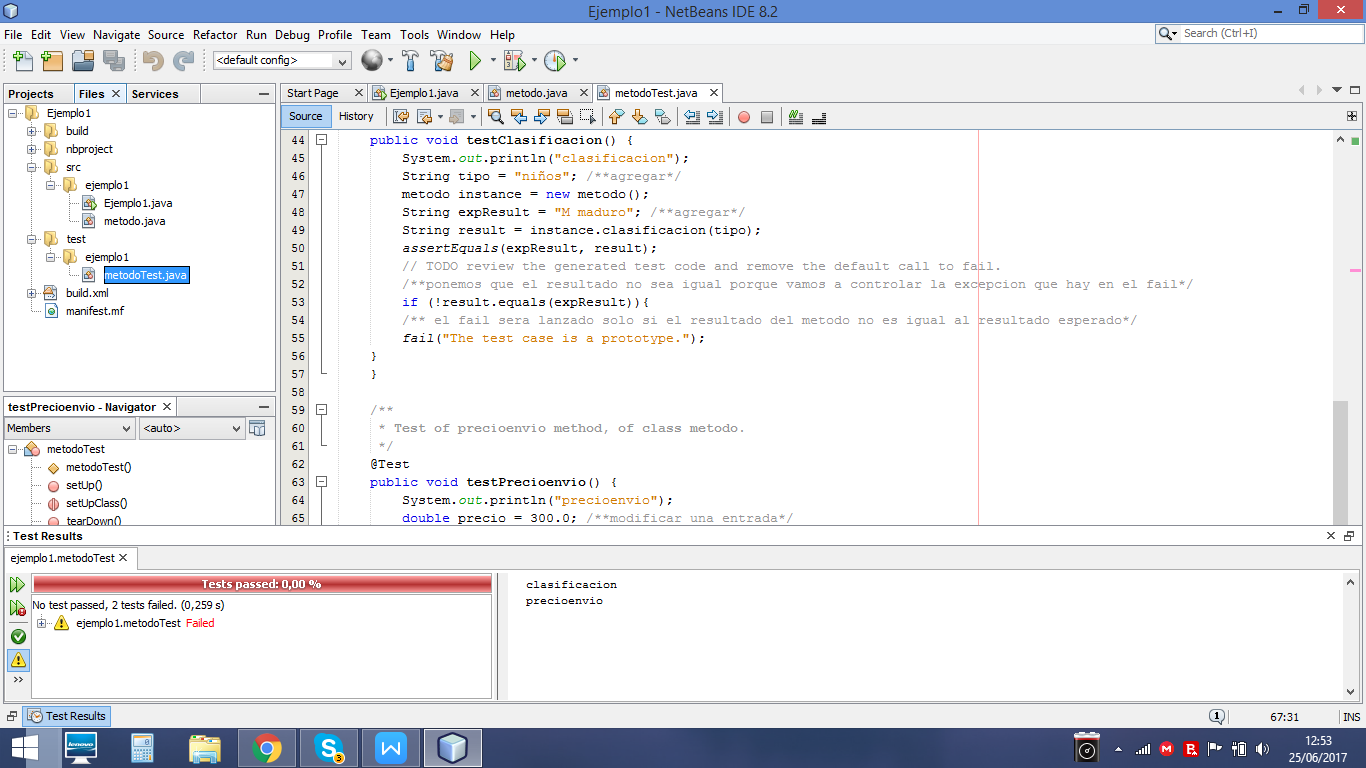


# 

# Y ahora como podemos ver las 2 pruebas fueron satisfactorias porque ambos resultados fueron los esperados

# 

# Si ahora modificamos niños y le decimos por algún motivo sale “M maduros”, la prueba unitaria tiene que fallar, del mismo modo si le damos un dato que no tiene que ser igual, la prueba unitaria tiene que fallar y nos tiene que indicar que la prueba falló porque el resultado esperado no es mismo al resultado que está arrojando la función en este caso.



# *5.2 Ejemplo 2*

# Aquí vemos una simple operación de suma de “a+b” para mostrar un resultado que puede variar según lo que se ponga, hecho para netbeans y con java.

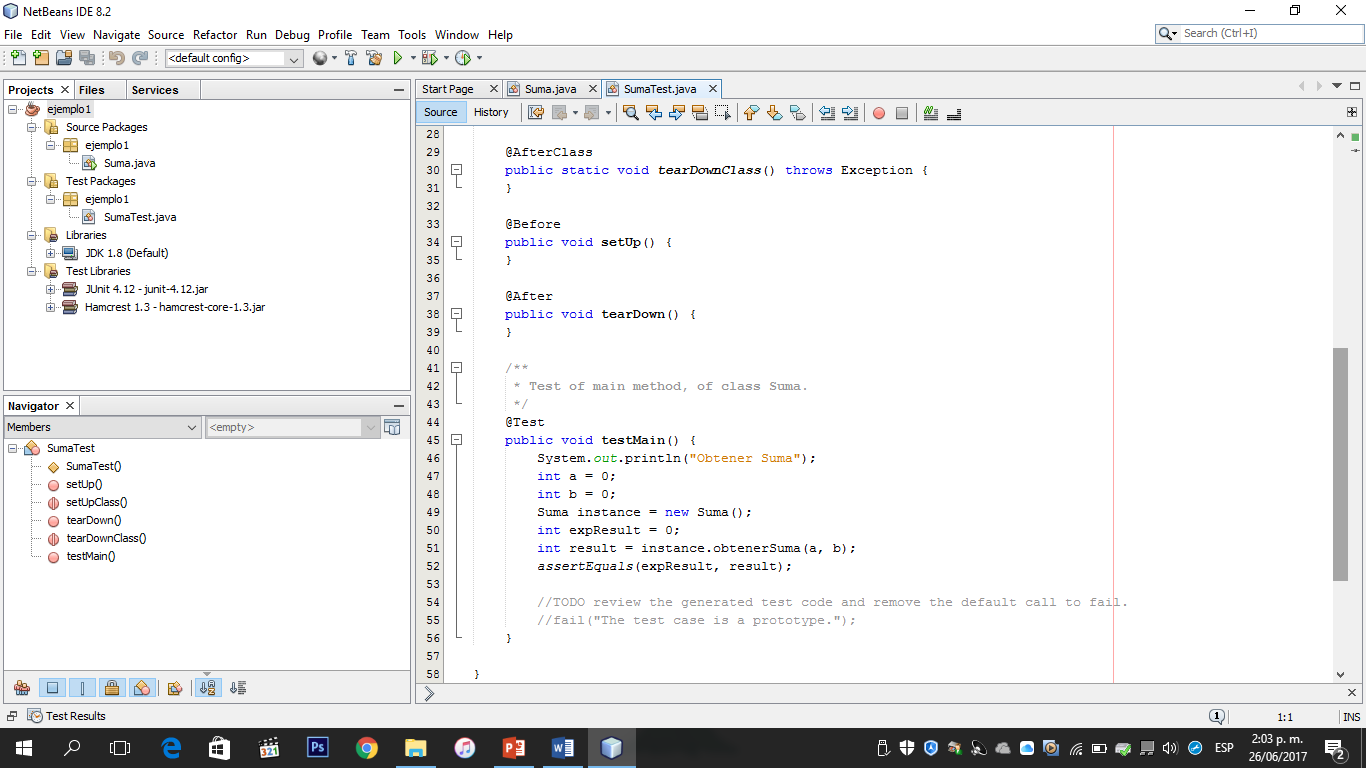
# Ahora crearemos un JUnit o prueba unitaria para poder confirmar si la operación resultara con excito. Se le hace click derecho a la opcion de arriba en projectos y se elije “New” y despues “Others”.

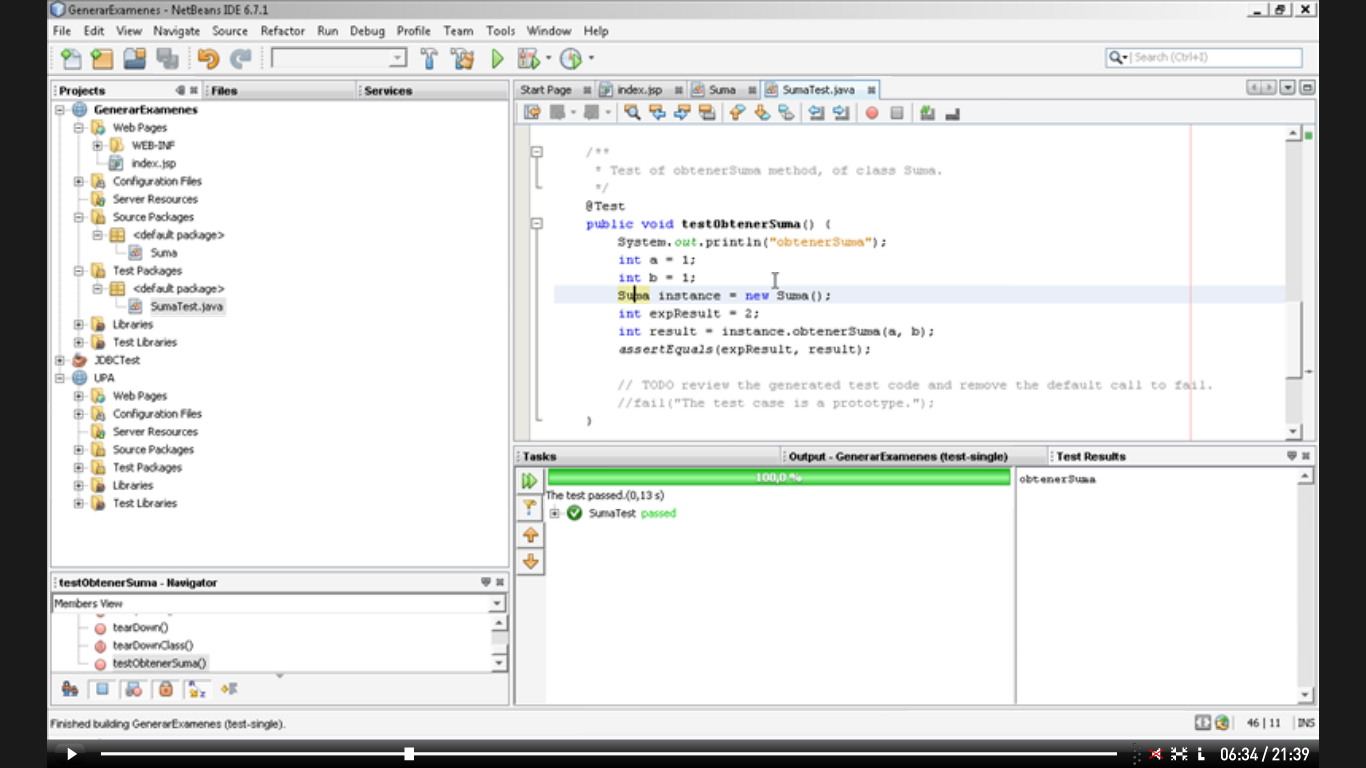
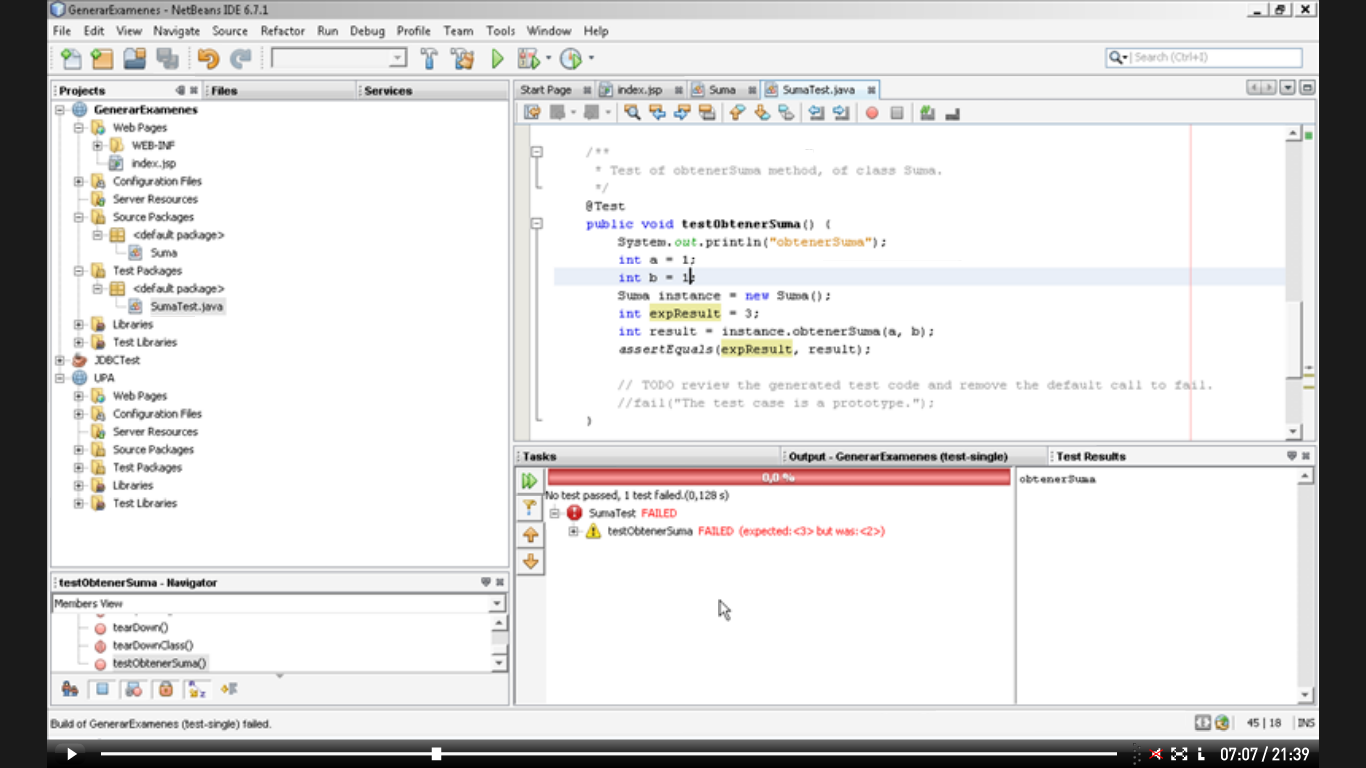
# En las categorías se selecciona la llamada “Unit Test” y en el tipo de File se elige la llamada “Test for Excisting File” o prueba para archivo existente, elegiremos el archivo java donde creamos la operación básica de matemáticas.

# Cuando se haya elegido se tienen que seleccionar todas las opciones disponibles para que el JUnit pueda hacer todas las formas de revisión posible.

# 

# Una vez creado se les abrirá una viñeta nueva, con las operaciones ya hechas para la operación escogida y el resultado para la operación que debería ser.



* Pueden ver que si se pone que a=0 y b=0 sumados te daran 0; pero si pones que a=1 y b=1 sumados te daran 3, la respuestas saldra erronea y te diran cual deberia ser la respuesta.

# **6. Conclusiones**

Al desarrollar un nuevo software o sistema de información, la primera etapa de pruebas a considerar es la etapa de pruebas unitarias. En la cual como mencionamos anteriormente, se encuentran presentes las pruebas de caja negra y de caja blanca en las cuales, en una se realiza un análisis de los datos de entrada y de salida. Mientras que en otro se analiza el proceso interno del sistema para evaluar las inconsistencias que pueda estar presentando. Todo esto para llegar a una corrección de los mismos y proseguir con la nueva fase del proceso de desarrollo del sistema. Podemos destacar que las pruebas unitarias son 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.

Este tipo de pruebas debe ser realizado por personal especializado en la aplicación de pruebas a nivel de software, el cual debe estar familiarizado en el uso de herramientas de depuración y pruebas, igualmente deben conocer el lenguaje de programación en el que se está desarrollando la aplicación.

El objetivo fundamental de las pruebas unitarias es asegurar el correcto funcionamiento de las interfaces, o flujo de datos entre componentes. De manera que a la hora de realizar una unificación de los diferentes componentes que conforman el sistema en general, exista una congruencia que favorezca el desarrollo de la aplicación que se quiere realizar.



# **7. Referencias Bibliográficas**

* JUnit. Pruebas Unitarias Introducción - dit/UPM. *JUnit Pruebas Unitarias (Dpto. de Ingeniería de Sistemas Telemáticos)*

Recuperado de: http://dit.upm.es/~legf/doc/lprg/Transparencias%20LPRG/Pruebas%20Unitarias.pdf

* Barrientos, Pablo Andrés (25 de abril de 2014). [*Enfoque para pruebas de unidad basado en la generación aleatoria de objetos*](http://sedici.unlp.edu.ar/handle/10915/34969)*.* p. 101. Consultado el 21 de junio del 2017.
* Bertolino A. (2003). *Software Testing Research and Practice*. Electronic Edition (Springer link).
* S. Sanchez, M. Sicilia, D. Rodriguez. Ingenieria del software. Un enfoque desde la guía SWEBOK. Alfaomega. 2012.