

FACULTAD DE INGENIERIA

ESCUELA ACADEMICA - INGENIERIA DE SISTEMAS

**“TRABAJO DE INVESTIGACION SOBRE PRUEBAS UNITARIAS”**

**ASIGNATURA:**

METODOLOGIAS DE PROGRAMACION

**INTEGRANTES:**

-Ascanoa Navarrete, Jhonatan

-Cisneros Haro, Anthony

-Saavedra Huaringa, Alberto

-Valencia Navarro, Jean Pierre

-Villaroel Rejano, Miguel

**DOCENTE:**

CORONEL CASTILLO, Eric Gustavo

**GRUPO:**

N° 06

**ATE- PERU**

**2017**

ÍNDICE

ÍNDICE

Pg.

DEFINICION 3

JUnit 3

¿Por qué pruebas untiarias? 4

Caracteristicas 4

¿Qué es JUnit? 5

Ventajas 6

Limitaciones 6

Ejemplos de uso 7

Conclusiones 11

Recomendaciones 12

Anexos 12

1. **DEFINICION**

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.

**JUnit:**

1. **Test Suites**

* Contenedor de **Test Cases**

1. **Test cases y asserts**

* Clases que definen un conjunto de pruebas relacionadas.

1. Implementado en el framework con los patrones de diseño:

* Comand(test case)
* Composite(test suite)

**2.¿POR QUE PRUEBAS UNITARIAS ?**

Todos los programadores saben que deben realizar pruebas a su código. Pocos lo hacen.

La respuesta generalizada al ¿Por qué no? es “No tengo tiempo”. Este apuro se convierte en un círculo vicioso. Cuanta más presión se siente, menos pruebas se realizan. Cuantas menos pruebas se realizan, menos producto se es y el código se vuelve menos estable.

Cuanto menos productivo y preciso se es, más presión se siente.

**UNA PRUEBA UNITARIA TIENE LAS SIGUIENTES CARACTERISTICAS:**

* Prueba solamente pequeñas cantidades de código: Solamente prueba el código del requerimiento especifico.
* Se aisla de otro código y de otros desarrolladores: El unit test prueba exclusivamente el código relacionado con el requerimiento y no interfiere con el trabajo hecho por otros desarrolladores.
* Solamente se aprueban los endpoints públicos: Esto principalmenteporque los disparadores de los métodos privados son métodos públicos por lo tanto se abarca el código de los métodos privados dentro de las pruebas.
* Los resultados son automatizados: Cuando ejecutamos las pruebas las podemos hacer de forma individual o de forma grupal. Estas pruebas las hace el motor de pruebas y los resultados de los mismos deben de ser precisos con respecto a cada prueba unitaria desarrollada.
* Repetible y predecible: No importa el orden ni las veces que se repita la prueba, el resultado siempre debe de ser el mismo.
* Son rapidos de desarrollar: Contrariamente a lo que peinsan los desarrolladores -> que el desarrollo de pruebas unitarias quita tiempo-los unit test por lo general deben de ser simples y rapidos de desarrollar. Dificilmente una prueba unitaria deba de tomar mas de cinco minutos en su desarrollo.

3.CARACTERISTICAS para que sea buena:

* Automatizable: No debería requerirse una intervencion manual. Esto es especialmente útil para integración continua.
* Completas: Deben cubrir la mayor cantidad de código.
* Repetibles o reutilizables: No se deben crear pruebas que solo puedan ser ejecutadas una sola vez. También es útil para integración 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.

**¿Qué es JUnit?**

es un conjunto de clases (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 fue creado por Erich Gamma y Kent Beck y es utilizado en programación para hacer pruebas unitarias de aplicaciones  Java.

JUnit es también un medio de controlar las pruebas de regresión, 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.

El propio framework incluye formas de ver los resultados (runners) que pueden ser en modo texto, gráfico (AWT o Swing) o como tarea en Ant.

En la actualidad las herramientas de desarrollo como Netbeans y Eclipse cuentan con plug-ins que permiten que la generación de las plantillas necesarias para la creación de las pruebas de una clase Java se realice de manera automática, facilitando al programador enfocarse en la prueba y el resultado esperado, y dejando a la herramienta la creación de las clases que permiten coordinar las pruebas.

Un método Assert es un método de JUnit que realiza una prueba y que retorna un “AssertionFailedError” si la prueba fallo.

**Tipos de Assert:**

1. assertTrue(boolean ***test***)

assertTrue(String ***message***, boolean ***test***)

2. assertFalse(boolean ***test***)

assertFalse(String ***message***, boolean ***test***)

3. assertEquals(***expected***, ***actual***)

assertEquals(String ***message***, ***expected***, ***actual***)

4. assertSame(Object ***expected***, Object ***actual***)

assertSame(String ***message***, Object ***expected***, Object ***actual***)

5. assertNotSame(Object ***expected***, Object ***actual***)

assertNotSame(String ***message***, Object ***expected***, Object ***actual***)

6. assertNull(Object ***object***)

assertNull(String ***message***, Object ***object***)

7. assertNotNull(Object ***object***)

assertNotNull(String ***message***, Object ***object***)

8. fail()

fail(String ***message***)

**4.VENTAJAS**

Las pruebas unitarias buscan aislar cada parte del programa y mostrar que las partes individuales son correctas, proporcionando cinco ventajas básicas:

* **Fomentan el cambio**, las pruebas unitarias facilitan la reestructuración del código (refactorización), puesto que permiten hacer pruebas sobre los cambios y verificar que las modificaciones no han introducido errores (regresión).
* **Simplifican la integración**, permiten llegar a la fase de integración asegurando que las partes individuales funcionan correctamente. De esta manera se facilitan las pruebas de integración.
* **Documentan el código**, las propias pruebas pueden considerarse documentación, ya que las mismas son una implementación de referencia de como utilizar el código.
* **Separación de la interfaz y la implementación**, 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 (ver pruebas mock).
* **Menos errores y más fáciles de localizar**, las pruebas unitarias reducen la cantidad de errores y el tiempo en localizarlos.
* **Pueden mejorar el diseño**, la utilización de prácticas de diseño y desarrollo dirigida por las pruebas (Test Driven Development o TDD) permite definir el comportamiento esperado en un paso previo a la codificación.
* **Puede ser la forma más simple de verificar el funcionamiento**, en situaciones como el desarrollo de una API o un componente que brinda servicios del cual no se cuenta aún con un cliente para consumirlos.

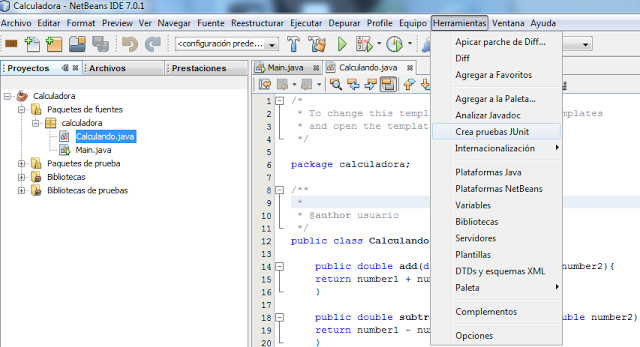
**5.LIMITACIONES**

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.[1](https://es.wikipedia.org/wiki/Prueba_unitaria#cite_note-1)​ Esta técnica se conoce como [testing aleatorio](https://es.wikipedia.org/wiki/Testing_aleatorio) (RT, por random testing). 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).

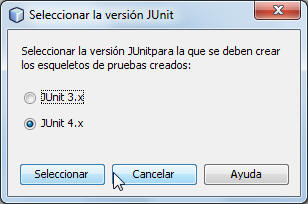
**6.EJEMPLOS DE USO**

Si desarrolláis en Java seguramente hayáis utilizado algún entorno de desarrollo (o IDE) como Eclipse o NetBeans. El siguiente tutorial está centrado en el uso de JUnit en NetBeans (usamos la versión 7.0).  
  
Las pruebas unitarias son las que realizamos sobre cada uno de los componentes que forman parte de la aplicación. Vamos a partir de una clase Java que implementa cuatro operaciones básicas de una calculadora: suma, resta, multiplicación y división. Podéis acceder al proyecto completo de este tutorial en [este enlace](https://mega.co.nz/#!hxAyyQTK!sGWJT5kHKgbFsoDAS3TaFE4lRZD7LiYQhlVKHlnW6Ro).  
  
Descomprimimos el proyecto y lo abrimos desde NetBeans. Para implementar las pruebas con JUnit abrimos la clase Calculando.java y seleccionamos el menú Herramientas > Crea pruebas

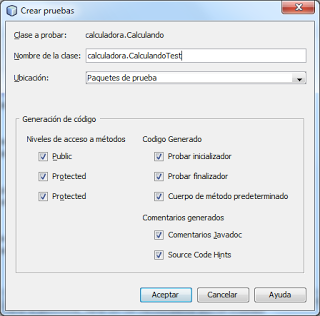
con JUnit:

[](http://2.bp.blogspot.com/-sPyjdA4tEMY/UYLHXpUiybI/AAAAAAAABJI/2MNQHb6Q4kc/s1600/junit1.png)

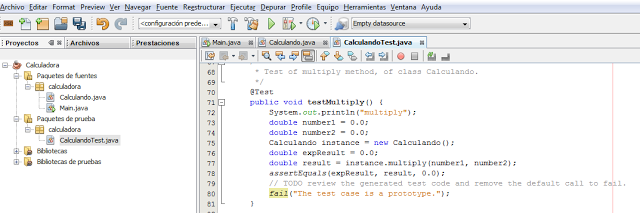
En el cuadro de diálogo siguiente elegimos JUnit 4.x

[](http://3.bp.blogspot.com/-ll6AULWneS4/UYLHqQHnBHI/AAAAAAAABJQ/qCCkZi-L4K0/s1600/junit2.png)

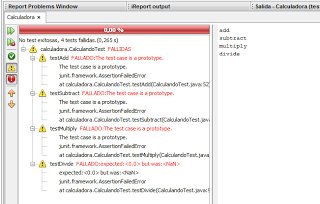
En el diálogo siguiente mantenemos las opciones por defecto y aceptamos:

[](http://4.bp.blogspot.com/-4x4np4vrK6I/UYLH0-it-ZI/AAAAAAAABJY/Pbu8tMIDR1g/s1600/junit3.png)

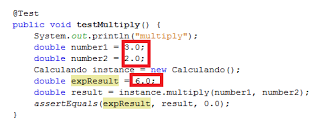
Ahora podemos acceder a la clase "CalculandoTest" creada en el paquete "Paquetes de prueba":

[](http://2.bp.blogspot.com/-kFbAK4RpL60/UYLINeD9Y_I/AAAAAAAABJg/XZcdSvh38UE/s1600/junit4.png)

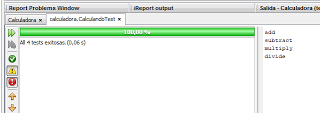
Éste método que vemos en la imagen superior es un prototipo (tiene los valores establecidos a cero y una función "fail()" que lanza un fallo en el método. Si ejecutáramos la prueba (ejecutando la clase, con Alt + F6 o con el menú contextual sobre el menú del proyecto > Probar) se produciría el siguiente resultado:

[](http://2.bp.blogspot.com/-48gveMNdlps/UYLIrW-1rfI/AAAAAAAABJo/O0FzxB2wQvk/s1600/junit5.png)

Falla porque debemos implementar los valores de los parámetros y el resultado previsto (cada uno de los casos de prueba) para cada método probado. Podemos implementar los casos de prueba estableciendo los valores de entrada, el valor esperado de cada operación y comentando (o eliminando) la llamada a la función fail(). Os mostramos un ejemplo con el método testMultiply():

[](http://4.bp.blogspot.com/-KzJt_W6gykc/UYLJxKBzh3I/AAAAAAAABJ0/xTTwREULK_s/s1600/junit6.png)

Una vez establecidos todos los casos de prueba (valores para todos los métodos) ejecutamos de nuevo las pruebas (menú contextual > Probar sobre el proyecto, ejecutar la clase test o con Alt + F6):



Como vemos si las pruebas son satisfactorias aparecerá como la imagen superior. Podéis encontrar el código de la clase de ejemplo y de test en el proyecto en el enlace del primer párrafo:

**7. CONCLUSIONES**

**-Al desarrollar un nuevo software o sistema de información, la primera etapa de pruebas a considerar es la etapa de pruebas unitarias, se encuentran presentes las pruebas de caja negra y de caja blanca**

**-se analiza el proceso interno del sistema para evaluar las inconsistencias que pueda estar presentando el sistema**

**-Este tipo de pruebas debe ser realizado por personal especializado**

**-El objetivo fundamental de las pruebas unitarias es asegurar el correcto funcionamiento de las interfaces.**

**-…**

**8.Recomendaciones**

* **Organización por clases**
* **Organización con namespaces**
* **Cómo nombrar una prueba unitaria**
* **El patrón: Esperado, Obtenido y Comparación**
* **Organización de escenarios de prueba**

**Anexos**

<https://books.google.com.pe/books?id=XIcXXFlO1FgC&pg=PA175&dq=pruebas+unitarias+java&hl=es&sa=X&ved=0ahUKEwiUz5D2nOzXAhWDDZAKHTwACuYQ6AEIWzAJ#v=onepage&q=pruebas%20unitarias%20java&f=false>

<https://www.google.com/search?tbm=bks&q=pruebas+unitarias+java>

<https://books.google.com.pe/books?id=PZQoZ9KTNaEC&pg=PA63&dq=pruebas+unitarias+java&hl=es&sa=X&ved=0ahUKEwiUz5D2nOzXAhWDDZAKHTwACuYQ6AEIJTAA#v=onepage&q=pruebas%20unitarias%20java&f=false>

<https://books.google.com.pe/books?id=ofv7CwAAQBAJ&pg=PA124&dq=pruebas+unitarias+java&hl=es&sa=X&ved=0ahUKEwiUz5D2nOzXAhWDDZAKHTwACuYQ6AEILzAC#v=onepage&q=pruebas%20unitarias%20java&f=false>

<https://books.google.com.pe/books?id=3EQdUbkOVGIC&pg=PA11&dq=pruebas+unitarias+java&hl=es&sa=X&ved=0ahUKEwiUz5D2nOzXAhWDDZAKHTwACuYQ6AEIPDAE#v=onepage&q=pruebas%20unitarias%20java&f=false>