****

**Facultad de Ingeniería de Sistemas e Informática**

**PRUEBAS UNITARIAS EN JAVA NETBEANS**

**Curso:**

Ingeniería de Software III

**Profesor:**

CORONEL CASTILLO ERIC GUSTAVO

**Integrantes:**

Prado Huerto, Jhoel

Pérez Chipana, Richard

García Montes, Julio

**Lima, Octubre 2016**

**Índice**

Resumen………………………………………………………………………………………………………………………………3

Antecedentes………………………………………………………………………………………………………………………..4

Desarrollo del Tema……………………………………………………………………………………………………………..5

MARCO TEORICO…………………………………………………………………………………….……………..…5

**CAPITULO I** DEFINICION………………………………………………………………………….……5

**CAPITULO II** Que son las pruebas unitarias……………………………………,,……………5

**CAPITULO III** Porque realizar pruebas unitarias……………………….……………………6

**CAPITULO IV** ¿Cómo Trabaja? ………………………………………………………………………7

**CAPITULO V** Otros beneficios de JUNIT…………………………………………………………7

**CAPITULO VI** Reglas de los test unitarios………………………………………………………7

**CAPITULO VII** ¿Cuándo hacerlos? …………………………………………………………………8

**CAPITULO VIII** ¿Cómo hacerlos? ………………………………………..…………………………8

**CAPITULO IX** PRUEBAS UNITARIAS CON JUNIT Y NETBEANS…….…………….……..9

Casos………………………………………………………………………………………………………………………………........13

CASO 1………………………………………………………………………………………………………………………13

CASO 2………………………………………………………………………………………………………………………19

Conclusiones…………………………………………………………………………………………………………………….……24

Recomendaciones………………………………………………………………………………………………………………….25

Bibliografía……………………………………………………………………………………………………………………….……26

Anexos……………………………………………………………………………………………………………………………….….27

**RESUMEN**

Las Pruebas Unitarias son métodos para realizar pruebas a fragmentos de código de un programa, estos fragmentos deben ser unidades estructurales del software que se está desarrollando.

En el caso de un lenguaje orientado a objetos estos fragmentos se les denominan métodos de las clases de un sistema.

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.

**ANTECEDENTES**

Decidimos investigar este tema Pruebas Unitarias por los siguientes motivos:

El objetivo de las pruebas unitarias es el aislamiento de las partes del código y la demostración de que estas partes no contienen errores.

Garantiza su correcto funcionamiento y además indica el tiempo que se demoró en ejecutar la rutina probada.

Permite un mejor trabajo en equipo por que se pueden compartir o intercambiar componentes ya probados, garantizados y listos para probarlos nuevamente ante cualquier cambio.

**PRUEBAS UNITARIAS**

**MARCO TEORICO**

**CAPITULO I - DEFINICION**

Método para realizar pruebas a fragmentos de código de un programa, estos fragmentos deben ser unidades estructurales del software que se está desarrollando.

En el caso de un lenguaje orientado a objetos estos fragmentos se les denominan métodos de las clases de un sistema.

Probar código nunca tuvo tanta importancia en el ciclo de desarrollo de una aplicación hasta hace algunos años, donde se ha desatado una revolución en los procesos de desarrollo, apareciendo nuevas y ágiles forma de construcción, donde ejecutar pruebas (o al menos pruebas unitarias) pasó de ser un proceso tedioso (como las antiguas que se ejecutaban para cumplir estándares como el ISO 9001) a ser un forma de trabajo integrada y productiva en los nuevos procesos de desarrollo.

A pesar de existir diferentes tipos y tácnicas de pruebas (unitarias, de integración, aceptación, carga y estress), en este articulo hablaremos del uso de Pruebas Unitarias (Unit Test), pues de todo el conjunto de pruebas, es considerada la más importante para garantizar un proyecto exitoso, por tanto debe ser introducida como una actividad más en el desarrollo

**CAPITULO II - Que son las pruebas unitarias**

Son pruebas dirigidas a probar clases java aisladamente y están relacionadas con el código y la

responsabilidad de cada clase y sus fragmentos de código más criticos.

Por ejemplo una clase que envie e­mails, debe ser capaz de probarse aislada chequeando que

sea capaz de enviar e­mails, y asegurandonos de probar todas las posibilidades de fallo y

éxito. En un modelo basado en pruebas, se deben definir casos de prueba para cada clase de

forma aislada, una prueba no debe depender de otras clases.

**CAPITULO III- Porque realizar pruebas unitarias**

• Asegura calidad del código entregado. Es la mejor forma de detectar errores

tempranamente en el desarrollo. No obstante, esto no asegura detectar todos los errores, por

tanto prueba de integración y aceptación siguen siendo necesarias.

• Ayuda a definir los requerimientos y responsabilidades de cada método en cada clase

probada.

• Constituye una buena forma de ejecutar pruebas de concepto. Cuando es necesario

hacer pruebas de conceptos sin integrar usar pruebas unitarias se convierte en un método

efectivo.

• Permite hacer refactoring tempranamente en el código. No es necesario todo un ciclo de

integración para hacer refactoring en la aplicación, basta con ver como se comporta un caso

de prueba para hacer refactoring unitario sobre la clase que estamos probando en cuestión.

• Permite incluso hacer pruebas de estress tempranamente en el código. Por ejemplo un

método que realize una consulta SQL que exceda los tiempos de aceptación es posible

optimizarla antes de integrar con la aplicacíón.

• Permite encontrar errores o bugs tempranamente en el desarrollo. Y está demostrado

que mientras más temprano se corrigan los errores, menos costará corregirlos.

**CAPITULO IV - ¿Cómo Trabaja?**

Esta basado en aserciones, que son condiciones que debe cumplir el resultado de la ejecución del método para demostrar que su funcionamiento es correcto.

Existen herramientas, frameworks, que facilitan este trabajo, una de ellas es el framework JUNIT.

**CAPITULO V - Reglas de los test unitarios**

Deben ser:

•Rápidos, deben ejecutarse en pocos segundos, a menudo todos a la vez.

•Independientes, una prueba no debe depender de otra.

•Repetible, el resultado del test debe dar el mismo resultado, para valores aleatorios inclusive.

•Pequeños, por que son más sencillas de modificar y comprender.

•Transparentes, el propósito de cada test debe ser claro.

**CAPITULO VI - Otros beneficios de JUNIT**

•Las pruebas unitarias sirven también como punto de partida para realizar desarrollo de aplicaciones orientada a pruebas “TDD”, disciplina muy usada en metodologías “ágiles”.

•Por medio de otras herramientas las pruebas unitarias también sirven para inspeccionar el código automáticamente y detectar malas prácticas de programación y de control de excepciones.

**CAPITULO VII - ¿Cuándo hacerlos?**

Dos estrategias:

Primero la funcionalidad y luego sus tests

Se implementa cada funcionalidad después de implementar el test que la probará (Test Driven Development)

Cada uno tiene que decidir de qué clases hacerlo y de cuales no (del interfaz de usuario no se suele hacer).

**CAPITULO VIII - ¿Cómo hacerlos?**

Habitualmente se utilizan las librerías XUnit (JUnit, PHPUnit, NUnit…), o el entorno de Microsoft

Habitualmente se crea un proyecto de pruebas por cada proyecto

Se suele crear una clase de pruebas por cada clase de funcionalidad

//Prueba para Sumas

public class SumarTest extends TestCase {

@org.junit.Test public void sumaTest {

int result = Sumar.suma(2,2);

assertEquals(4, result);

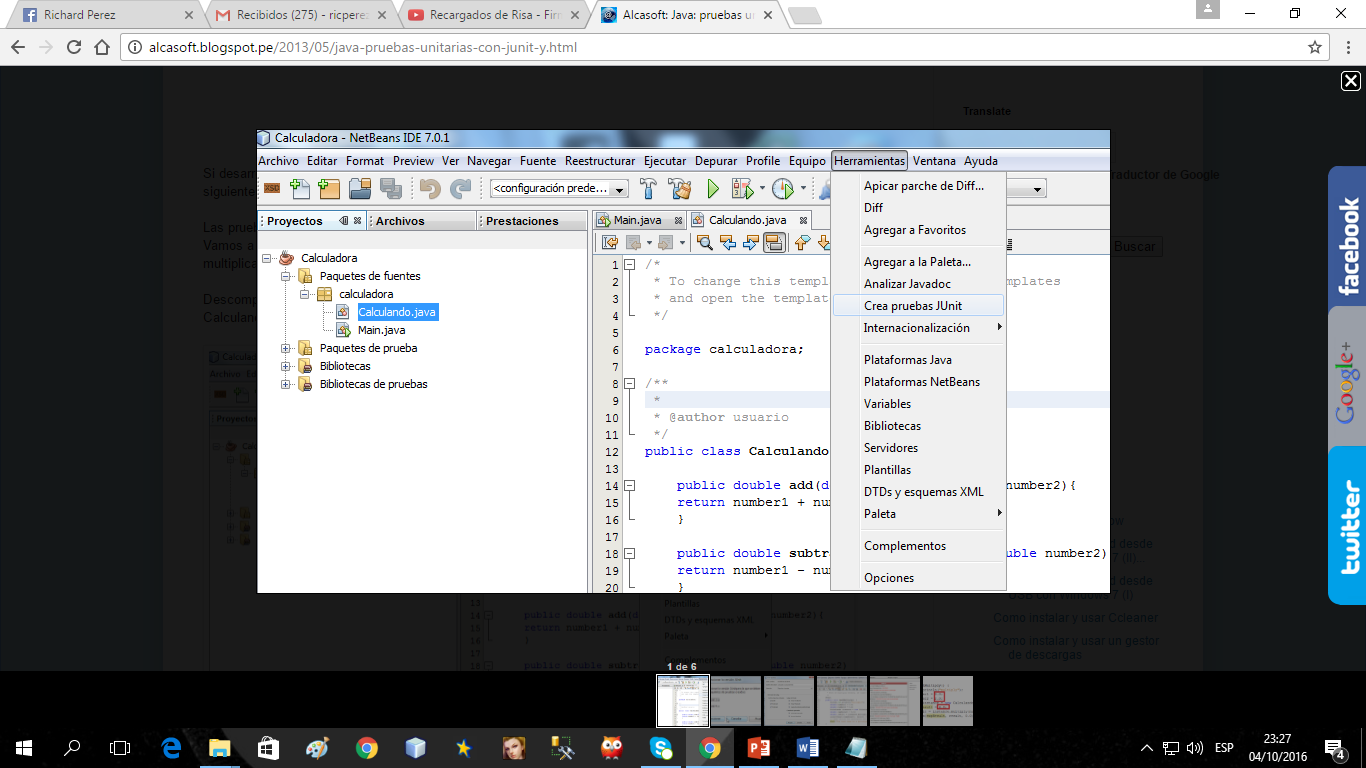
}

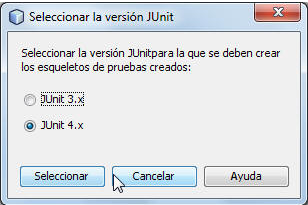
}

**CAPITULO IX - PRUEBAS UNITARIAS CON JUNIT Y NETBEANS**

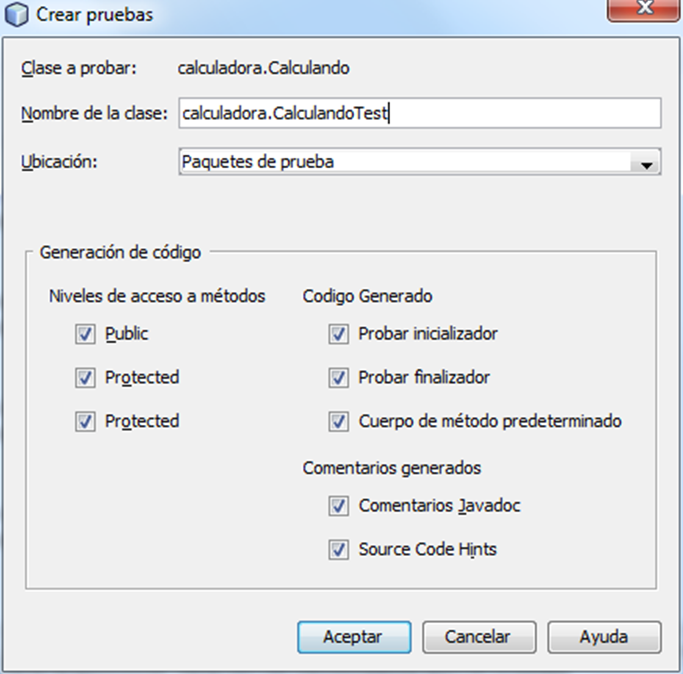
Las pruebas unitarias son las que realizamos sobre cada uno de los componentes que forman parte de la aplicación. Vamos a partir de una clase Java que implementa cuatro operaciones básicas de una calculadora: suma, resta, multiplicación y división.

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:

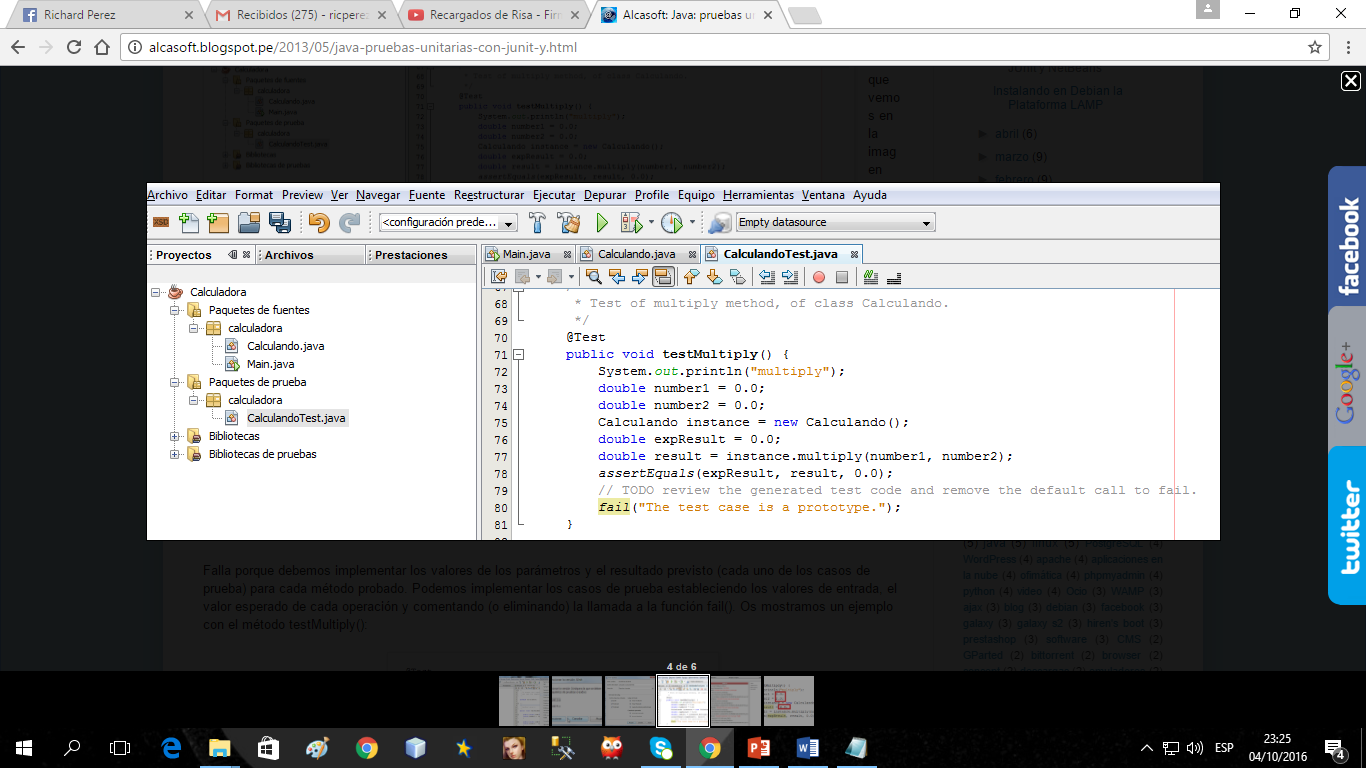
****

****

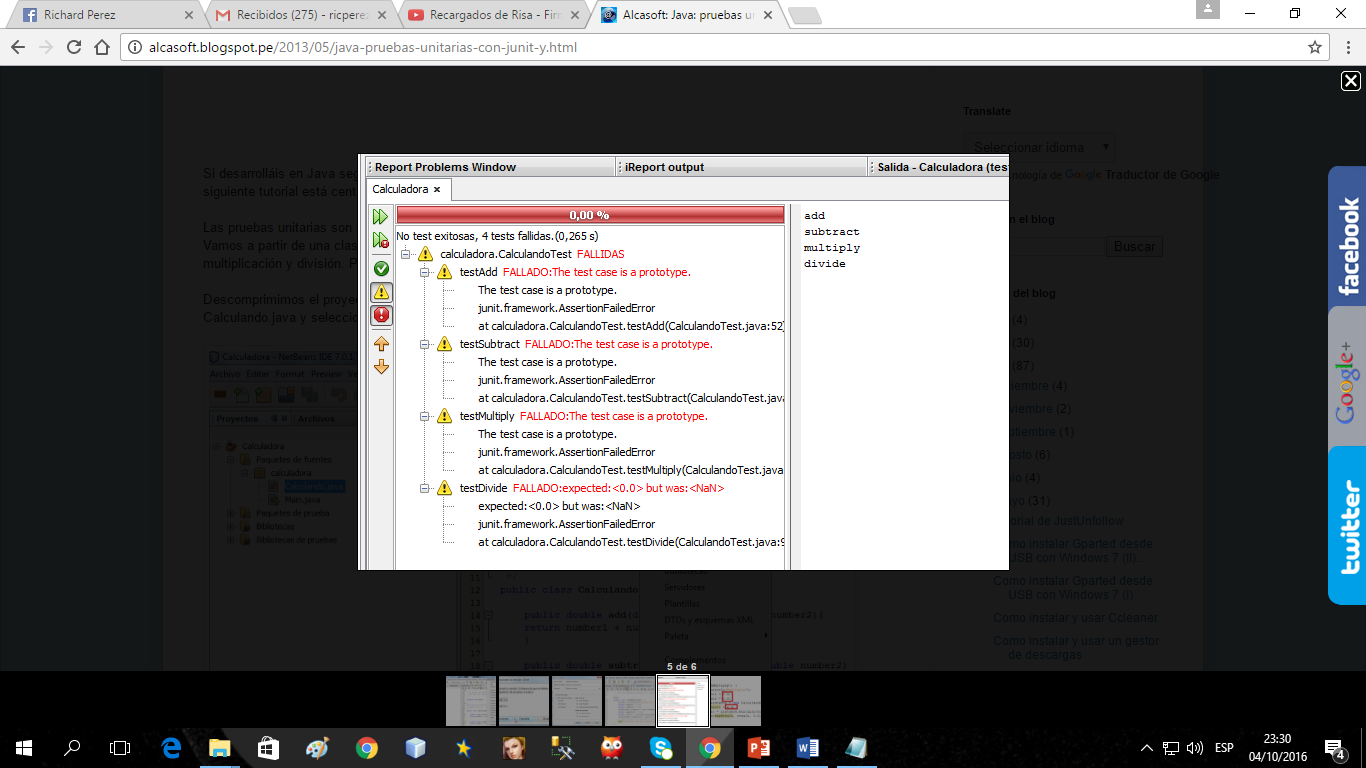
En el diálogo siguiente mantenemos las opciones por defecto y aceptamos:

****

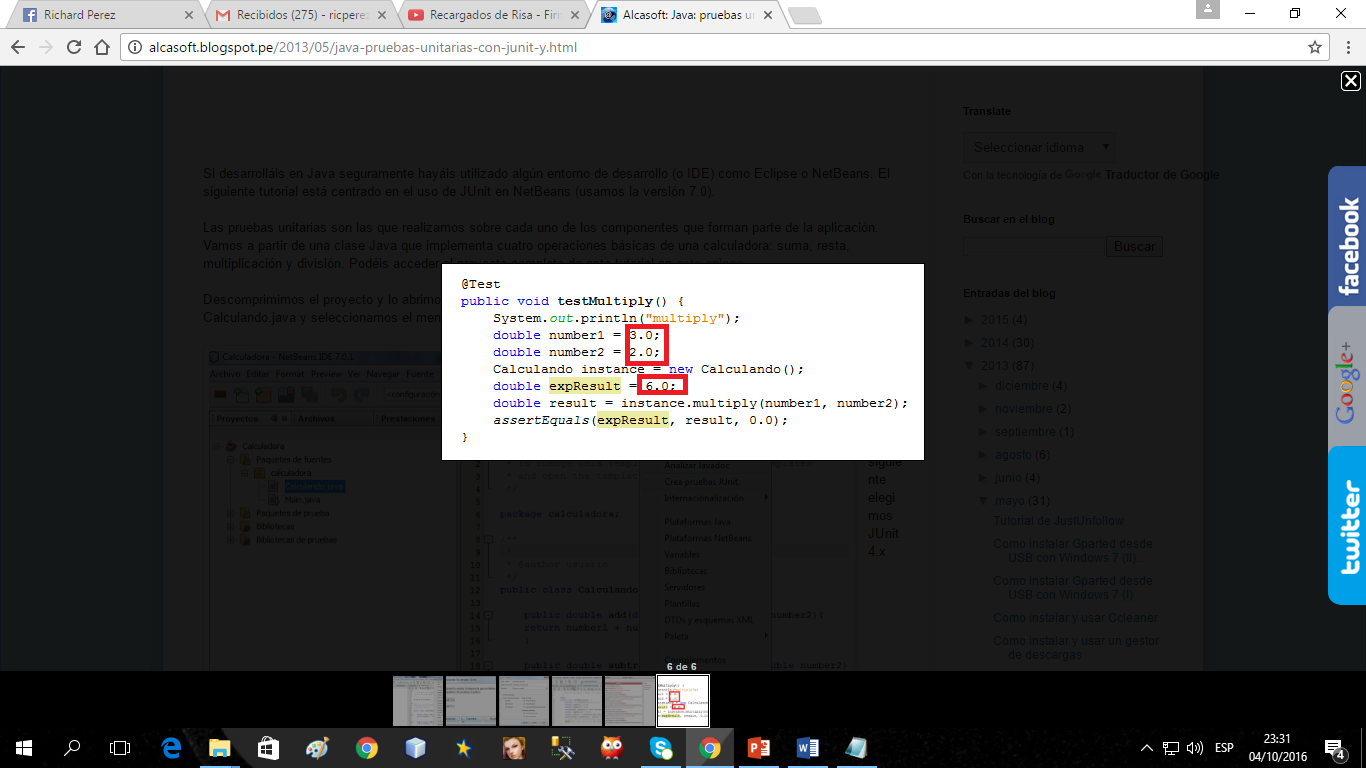
Ahora podemos acceder a la clase "CalculandoTest" creada en el paquete "Paquetes de prueba":

****

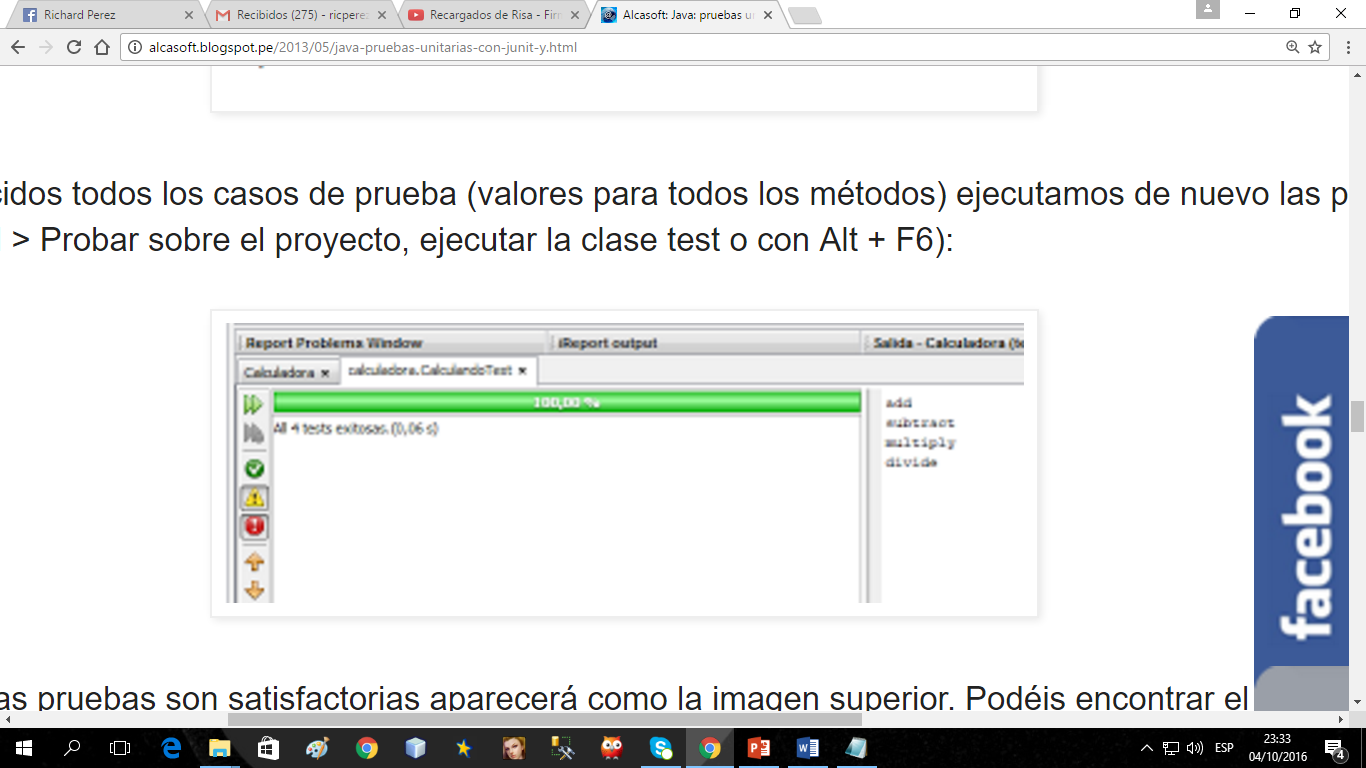
Éste método que vemos en la imagen superior es un prototipo (tiene los valores establecidos a cero y una función "fail()" que lanza un fallo en el método. Si ejecutáramos la prueba (ejecutando la clase, con Alt + F6 o con el menú contextual sobre el menú del proyecto > Probar) se produciría el siguiente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



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(). Aquí se muestra un ejemplo con el método testMultiply():

****

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.

**CASOS**

**CASO 1**

Desarrollar un método estático que tome un array de enteros como

argumento y devuelva el mayor valor encontrado en el array.

public class MayorNumero {

/\*\*

\* Devuelve el elemento de mayor valor de una lista

\* @param list Un array de enteros

\* @return El entero de mayor valor de la lista

\*/

public static int obt\_mayorNumero(int lista[]) {

return 0; // para que compile hasta que desarrollemos el método

}

}

¿Qué pruebas ocurren para el método obt\_mayorNumero?

– ¿Qué ocurre para un array con valores cualesquiera (el caso

normal)?

[3, 7, 9, 8] → 9

– ¿Qué ocurre si el mayor elemento se encuentra al principio, en

medio o al final del array?

[9, 7, 8] → 9; [7, 9, 8] → 9; [7, 8, 9] → 9

– ¿Y si el mayor elemento se encuentra duplicado en el array?

[9, 7, 9, 8] → 9

– ¿Y si sólo hay un elemento en el array?

[7] → 7

– ¿Y si el array está compuesto por elementos negativos?

[-4, -6, -7, -22] → -4

Escribimos el código del método obt\_mayorNumero:

/\*\*

\* Devuelve el elemento de mayor valor de una lista

\*

\* @param list Un array de enteros

\* @return El entero de mayor valor de la lista

\*/

public static int obt\_mayorNumero(int lista[]) {

int indice, max = Integer.MAX\_VALUE;

for (indice = 0; indice < lista.length-1; indice++) {

if (lista[indice] > max) {

max = lista[indice];

}

}

return max;

}

Escribimos el código de las pruebas (1):

import junit.framework.\*;

public class TestMayorNumero extends TestCase {

public TestMayorNumero() {

}

public void testSimple() {

assertEquals(9, MayorNumero.obt\_mayorNumero(new int[] {3, 7, 9, 8}));

}

public void testOrden() {

assertEquals(9, MayorNumero.obt\_mayorNumero(new int[] {9, 7, 8}));

assertEquals(9, MayorNumero.obt\_mayorNumero(new int[] {7, 9, 8}));

assertEquals(9, MayorNumero.obt\_mayorNumero(new int[] {7, 8, 9}));

}

public void testDuplicados() {

assertEquals(9, MayorNumero.obt\_mayorNumero(new int[] {9, 7, 9, 8}));

}

public void testSoloUno() {

assertEquals(7, MayorNumero.obt\_mayorNumero(new int[] {7}));

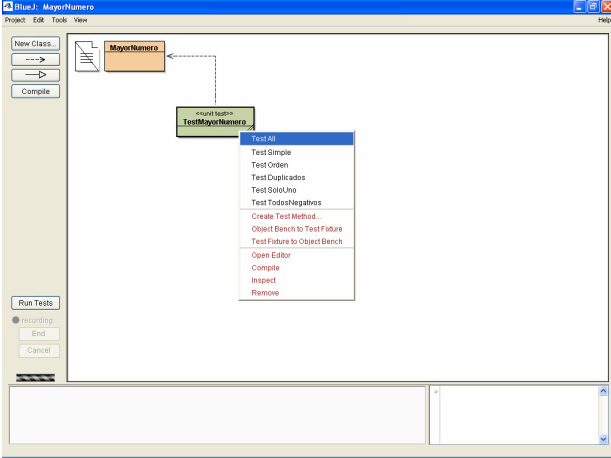
}

public void testTodosNegativos() {

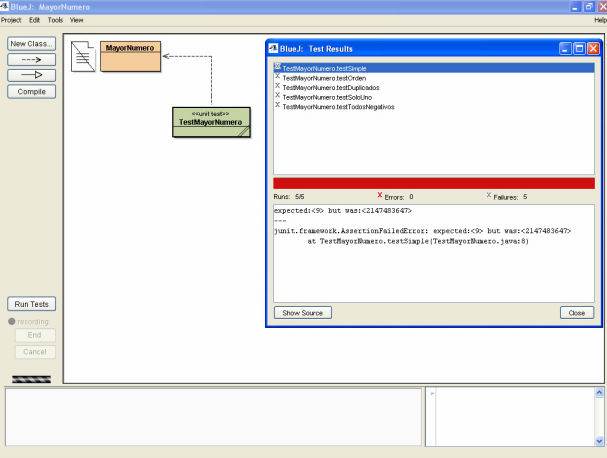
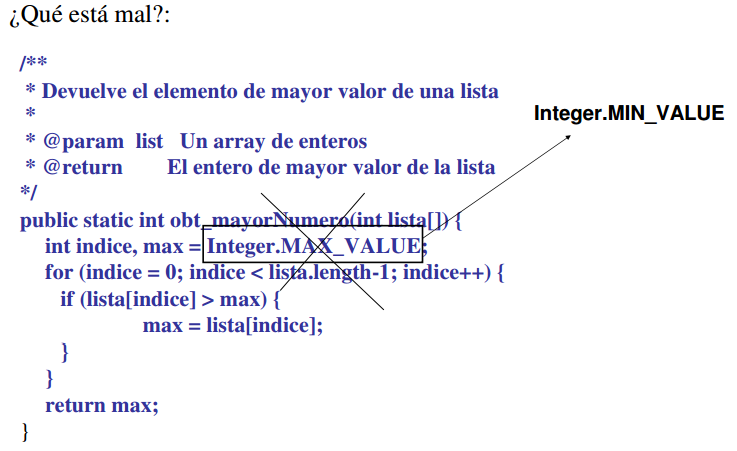
assertEquals(-4, MayorNumero.obt\_mayorNumero(new int[] {-4, -6, -7, 22}));

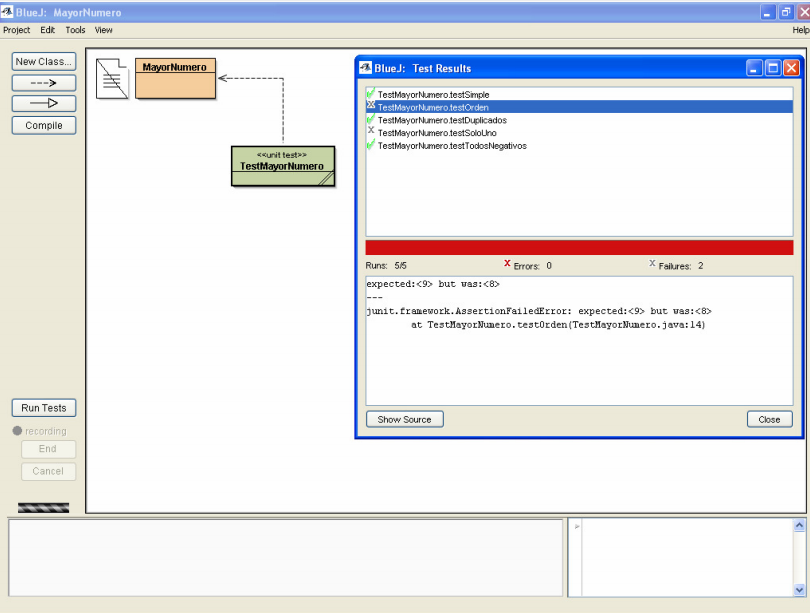
}

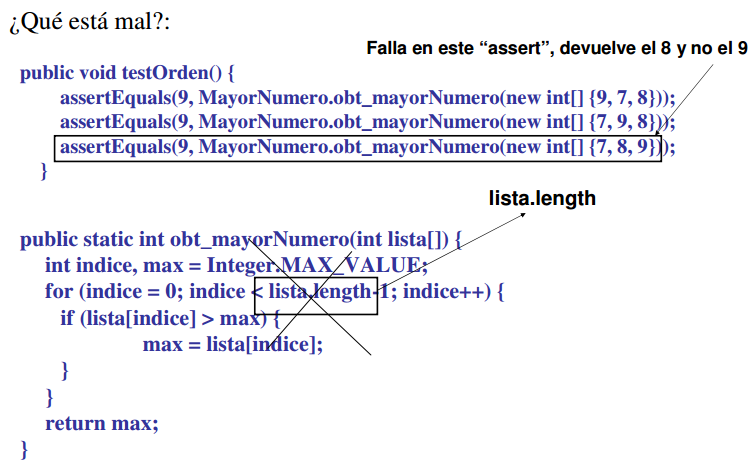
} // Fin de la clase

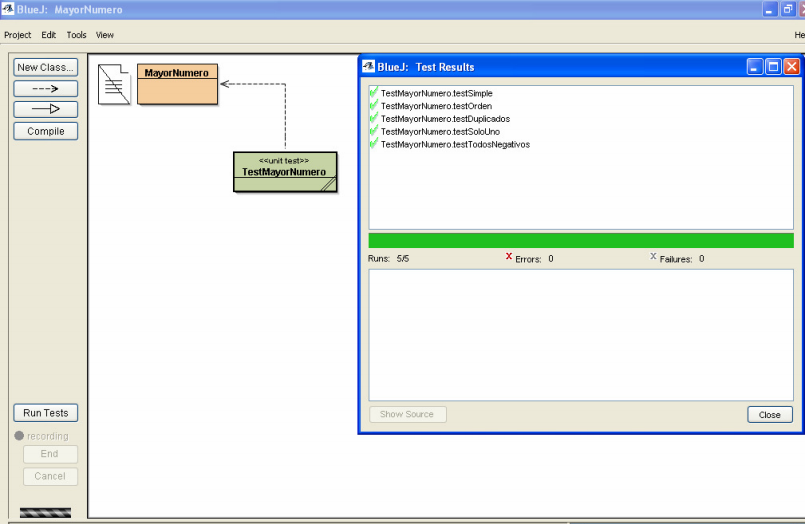
Ejecutamos las pruebas:

Ejecutamos las pruebas:



Volvemos a ejecutar las pruebas (hasta que pasen todas):



Otra vez a ejecutar las pruebas,ahora ya pasan todas.

**CASO 2**

public class PDAdminTemplateTests extends TestCase {

/\*\* servicio de operaciones sobre TAM \*/

private TAMOperations tam;

/\* (non­Javadoc)

 \* @see junit.framework.TestCase#setUp()

 \*/

protected void setUp() throws Exception {

super.setUp();

ApplicationContext app =

new ClassPathXmlApplicationContext("applicationContext.xml");

this.tam = (TAMOperations) app.getBean("tamOperations");

}

/\* (non­Javadoc)

 \* @see junit.framework.TestCase#tearDown()

 \*/

protected void tearDown() throws Exception {

super.tearDown();

}

/\*

 \* Test method for 'cl.interplanet.dao.tam.support.PDAdminTemplate.createUser(...)'

 \*/

public void testCreateUser() {

this.tam.createUser("TATA",

new PDRgyUserName("cn=TATA,cn=Users,dc=achs,dc=cl", "Bravo", "Bravisimo"),

"ESTE ES EL TATA", "000001", null, true, false, true);

}

/\*

 \* Test method for 'cl.interplanet.dao.tam.support.PDAdminTemplate.deleteUser(...)

 \*/

public void testDeleteUser() {

this.tam.deleteUser("TATA", false);

}

/\*

 \* Test method for 'cl.interplanet.dao.tam.support.PDAdminTemplate.importUser(...)

 \*/

public void testImportUser() {

this.tam.importUser("21493497",

new PDRgyUserName("CN=Cuenta CI. Informacion,CN=Users,DC=achs,DC=cl",

null, null), true);

}

/\*

 \* Test method for 'cl.interplanet.dao.tam.support.PDAdminTemplate.setAuthRuleText(...)

 \*/

public void testSetAuthRuleText() {

String txt = "<xsl:choose><xsl:when " +

"test=\"(azn\_engine\_target\_resource='" +

"/WebSEAL/marte­marte/webapp/010002')\">" +

"!FALSE!</xsl:when><xsl:otherwise>" +

"!TRUE!</xsl:otherwise>" +

"</xsl:choose>";

this.tam.setAuthzRuleText("app­mantencion­rule", txt);

}

...

}

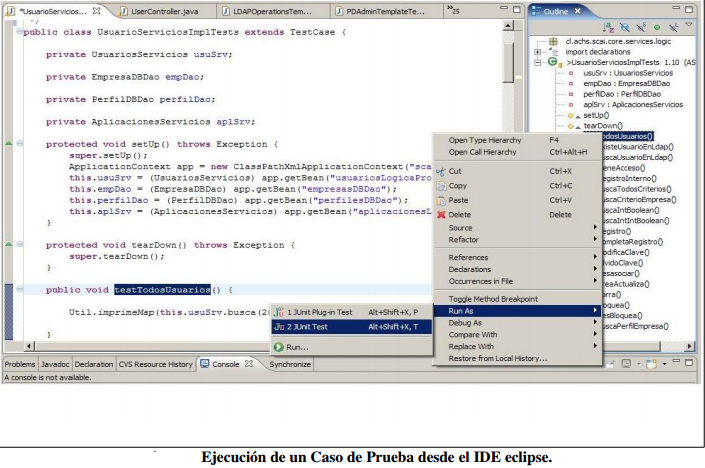
Es posible usar una serie de test runners provistos por JUnit para correr nuestros test cases que se

ejecutan y muestran los resultados de formas diferentes. Los más usados son el Text runner y el

Swing runner que lo muestra de forma gráfica. Pero en lo particular yo prefiero ejecutar JUnit

desdeel IDE3 o mejor aún, desde ant, desde donde es posible además automatizar la ejecución de

los casos de prueba generando reportes en html de cada ejecución.

A continuación algunos pantallazos del uso de JUnit en eclipse



Con el uso de JUnit + Ant es posible generar reportes sobre los resultados de la ejecución de todos

los casos de prueba4,  por ejemplo usando ant podemos tener un build.xml como el de la siguiente

figura.

<target name="run­tests" depends="compile­test" description="Ejecuta todos los casos de prueba y genera reportes">

<property name="reports.dir" value="${target.junit.reports.dir}"/>

<property name="summary.dir" value="${target.junit.summary.dir}"/>

<mkdir dir="${reports.dir}"/>

<mkdir dir="${summary.dir}"/>

<junit printsummary="yes" haltonfailure="no" haltonerror="no">

<jvmarg line="­Djava.awt.headless=true"/>

<!­­ Must go first to ensure any jndi.properties files etc take precedence  ­­>

<classpath location="${target.testclasses.dir}"/>

<classpath location="${target.classes.dir}"/>

<!­­ Need files loaded as resources ­­>

<classpath location="${test.dir}"/>

<classpath refid="all­libs"/>

<formatter type="plain" usefile="false"/>

<formatter type="xml"/>

<batchtest fork="yes" todir="${reports.dir}">

<fileset dir="${target.testclasses.dir}" includes="${test.includes}" excludes="${test.excludes}"/>

</batchtest>

</junit>

<junitreport todir="${reports.dir}">

<fileset dir="${reports.dir}">

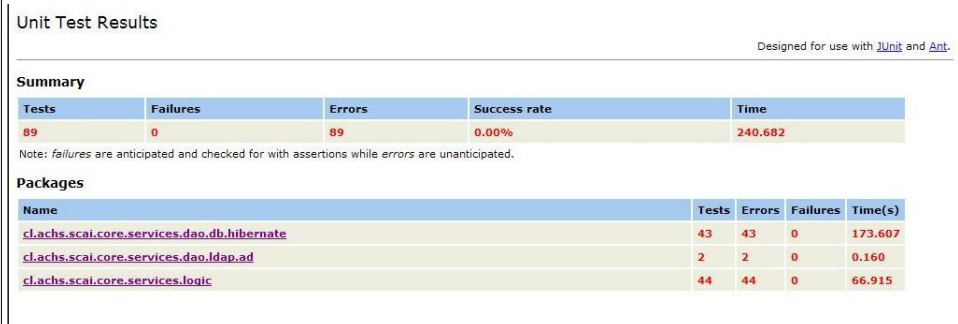
<include name="TEST­\*.xml"/>

</fileset>

<report todir="${summary.dir}"/>

</junitreport>

</target>

Y ante la ejecución del comando > ant run­tests se genera el siguiente resporte

**CONCLUSIONES**

• Las pruebas unitarias son parte de los entregables que toda aplicación debería considerar.

• No es necesario imprimir los resultados en consola basta con utilizar aserciones que se encargarán de comparar los resultados obtenidos frente a los esperados y si falla muestra automáticamente los resultados obtenidos que están fallando.

• Las pruebas unitarias permiten un mejor trabajo en equipo y es considerado un método ágil.

• Junit es un framework que facilita la ejecución de las pruebas unitarias.

**RECOMENDACIONES**

Las pruebas deben usarse en todo el ciclo de desarrollo de un proyecto, sobre todo debe

formar parte del dia a dia del programador, formulando filosofías como la TDD (Test Driven

Development) donde el método es probar antes de implementar. Cada uno tiene que decidir de

que clases hacerlo y de cuales no (del interfaz de usuario no se suele hacer).

Comprobar que se obtienen los resultados esperados. En caso de que el código

funcione correctamente, ¿cómo lo sabré? Prueba en los extremos pues es donde “viven” la

mayoría de los errores.

Algunos síntomas de que hay un error en los extremos pueden ser: valores a

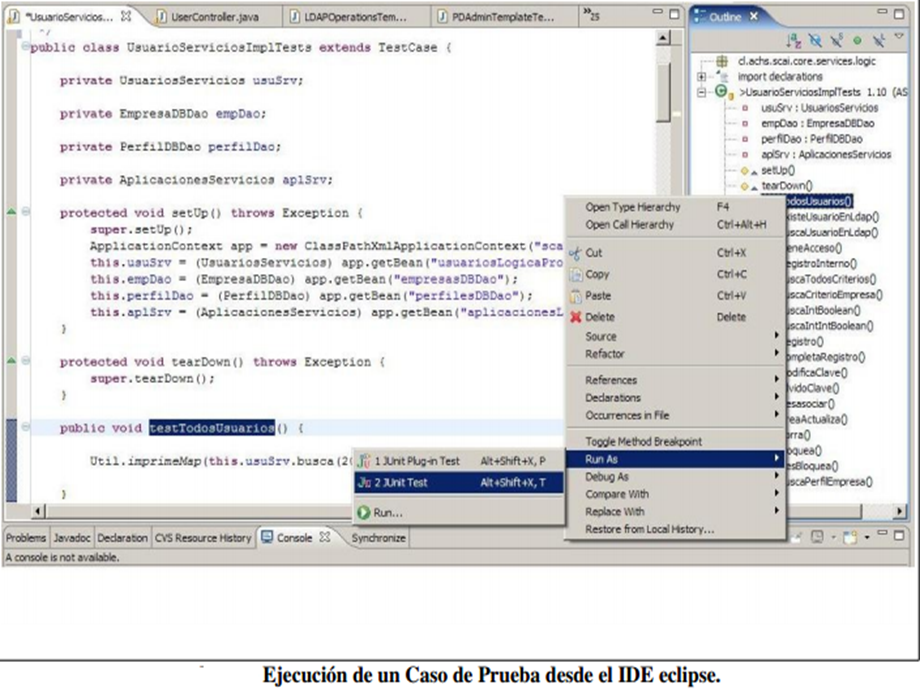
null o vacíos cuando no debería ser así, valores muy superiores a lo esperado,

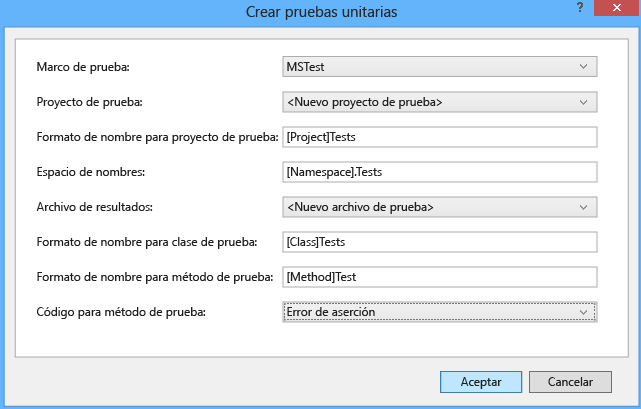
elementos duplicados en listas o que desaparecen, listas desordenadas cuando

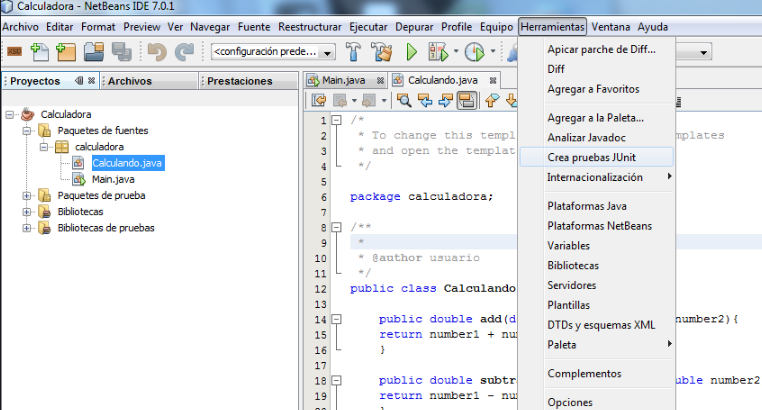
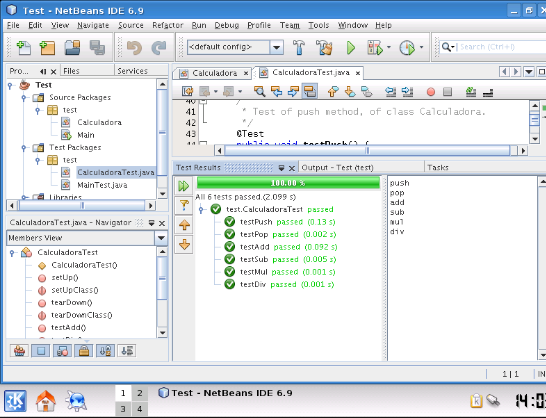
deberían estar ordenadas, acciones que ocurren fuera de orden, etc.

**BIBLIOGRAFIA**

* Desarrollo de aplicaciones web en el entorno servidor. IFCD0210- Rafael Luis Granados La Paz- Malaga. España.
* Diseño Ágil con TDD- Carlos Blé Jurado- 2007 -Malaga. España.
* PROGRAMACION EN JAVA- McGraw-Hill – 2011 - Madrid. España.
* Programación orientada a objetos con Java- 2007.
* Técnicas cuantitativas para la gestión en la ingeniería del software- 2014- Isabel Ramos Román, José Javier Dolado Cosín-2011 - España.
* Desarrolle aplicaciones Windows con pruebas unitarias – 2009, Córdova. España.

**ANEXOS**

****

****