



Contenido

Tema 8: Diseño y Planificación de Pruebas Software.....	3
1. Introducción	3
2. Planificación de Pruebas Software.....	6
3. Pruebas Unitarias	6
4. Pruebas de Aceptación.....	8
5. Pruebas de Integración	9
6. JavaDoc.....	10
7. Pruebas Unitarias con <i>JUnit en Netbeans</i>	15

Tema 8: Diseño y Planificación de Pruebas Software**1. Introducción***¿Qué tipos de pruebas software existen?*

Un **conjunto de pruebas** está orientado a **comprobar** determinados aspectos de un sistema software (o de una parte de éste). Vamos a conocer de una forma resumida los **Tipos de Pruebas Software en función del objetivo** en los que se centran.

Pruebas Software Funcionales

Habitualmente podremos encontrar el comportamiento del sistema, subsistema o componente software descrito en especificaciones de requisitos o casos de uso, aunque también puede no estar documentado.

Estas pruebas **se definen a partir de funciones o características** (descritas dentro de la documentación o interpretadas por los probadores o testers) y su interoperabilidad con sistemas específicos, **pudiendo ejecutarse en todos los niveles de pruebas** (componentes, integración, sistema, etc). Es decir, se evalúa "lo que el sistema hace".

Son **Pruebas de Caja Negra** ("black-box testing") puesto que valoramos el **comportamiento externo** del sistema. Las **Pruebas de Seguridad** o las **Pruebas de Interoperabilidad** entre sistemas o componentes son casos especializados de las pruebas funcionales.



Pruebas Software no Funcionales

Incluyen las pruebas de: **Rendimiento, Carga, Estrés, Usabilidad, Mantenibilidad, Fiabilidad o Portabilidad**, entre otras. Por tanto se centran en características del software que establecen "*cómo trabaja el sistema*".

Estas pruebas también **pueden ejecutarse en todos los niveles de pruebas**. Las características no funcionales del software se pueden medir de diversas maneras, por ejemplo, por medio de tiempos de respuesta en el caso de pruebas de **rendimiento** o por número máximo de sesiones en pruebas de **estrés**.

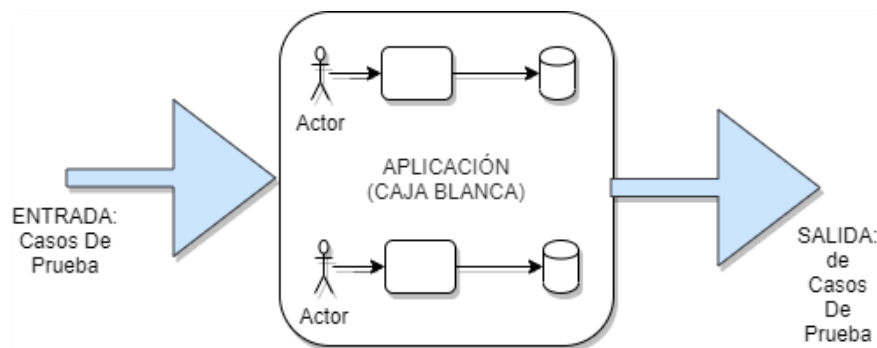
Puesto que las Pruebas software no Funcionales normalmente consideran el comportamiento externo del sistema, en la mayoría de los casos se utilizan **técnicas de Pruebas de Caja Negra**.

Pruebas Software Estructurales.

Nuevamente pueden ejecutarse en todos los niveles de pruebas (componentes, integración, sistema, etc.) y **encajan muy bien si hemos utilizado técnicas de especificación de la estructura o arquitectura del Software**. Es posible aplicar técnicas estáticas de análisis de código.

Para expresar el alcance con un conjunto de pruebas ("test suite") que ha cubierto el diseño de las pruebas, se utiliza el concepto de **Cobertura** ("Coverage"), normalmente en forma de porcentaje.

Es una técnica habitual utilizar **herramientas para calcular la cobertura del código** en el caso de Pruebas de Componentes o en Pruebas de Integración de **Componentes** (por ejemplo, trazando la jerarquía de llamadas entre elementos). Puesto que indagamos en el comportamiento interno, estas pruebas se denominan también **Pruebas de Caja Blanca** ("*white-box testing*").



Pruebas Software de Regresión

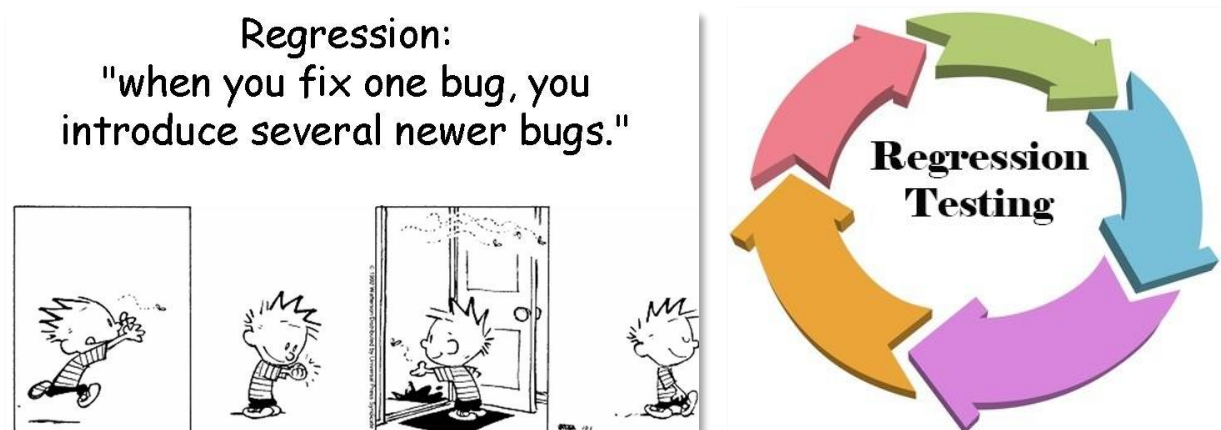
Una vez que un defecto dentro del desarrollo ha sido corregido, es necesario volver a probar el software para confirmar que el defecto ha sido eliminado y que no han surgido nuevos defectos. Son las pruebas de regresión.

Las **Pruebas de Regresión** consisten en volver a probar un componente, tras haber sido modificado, para descubrir cualquier defecto introducido, o no cubierto previamente, como consecuencia de los cambios. Los defectos pueden encontrarse tanto en el software que se ha cambiado como en algún otro componente. Se ejecutan cuando se cambia el software o su entorno. El criterio para decidir la extensión de estas Pruebas de Regresión está basado en el **riesgo de no encontrar defectos en el software que anteriormente estaba funcionando correctamente**.

Los conjuntos de pruebas de regresión ("*Regression test suites*") suelen ser bastante estables por lo que son muy buenos candidatos para **actividades de automatización de pruebas software**.

Quizá ya no os sorprenda que estas pruebas también **puedan ejecutarse en todos los niveles de pruebas** e incluyen casos de prueba de los tipos vistos anteriormente: Pruebas Funcionales, No Funcionales y Estructurales.

Ahora que ya tenemos una buena imagen de los tipos de pruebas software que podemos incorporar para **asegurar la Calidad del Software (SQA)**, es más fácil que entendamos que se trata de una actividad que requiere una elevada capacidad de adaptación de las cargas de trabajo, así como de una suficiente especialización.



2. Planificación de Pruebas Software

Dentro del desarrollo de un producto software, se utilice una metodología ágil o tradicional, es necesario que se incluya la fase de pruebas, la cual nos permitirá determinar si el producto a entregar cumple con la calidad especificada y esperada por el cliente.

Para llevar a cabo de forma correcta la planificación, se necesita un **Plan de Pruebas de Software**, el cual **tiene como propósito comunicar a todos los involucrados del proyecto las características a ser o no ser probadas, los aspectos de criterios de aprobación y fallo, criterios de suspensión y reanudación, y los riesgos.**

El **Plan de pruebas de Software** se puede aplicar a todo proyecto de software, se ajusta a las necesidades de cada proyecto de software y habrá que tener en cuenta a la hora de su diseño:

- El tamaño del proyecto.
- El tiempo disponible para ejecutarlo.
- El coste.
- El ciclo de vida del software
- ...

3. Pruebas Unitarias

Las **pruebas unitarias** o *Unit testing*, forman parte de los diferentes procedimientos que se pueden llevar a cabo dentro de la metodología ágil. Son pequeños test creados a medida para cubrir todos los requisitos funcionales y no funcionales del código y verificar sus resultados de forma individual.

Características de las pruebas unitarias:

- **Automatizables:** Para cada uno de los test unitarios desarrollados, los resultados se pueden automatizar, de tal forma que será posible realizar las pruebas de forma individual o agrupar las pruebas dentro de grupos.
- **Completas:** El proceso de Testing consta de pequeños test que serán realizados sobre parte del código, pero finalmente, se debe poder comprobar la totalidad del código involucrado.
- **Repetibles:** Habitualmente será necesario repetir las pruebas a lo largo del tiempo, de forma individual o grupal, por lo que el resultado debe ser siempre el mismo dando igual el orden en que se realicen los test.

- **Independientes:** Las pruebas unitarias son un código aislado que se ha creado con el objetivo de comprobar otro código muy concreto, no interfiere en el trabajo de otros desarrolladores.
- **Rápidas de crear:** Como norma general, el código de los tests unitarios no debe llevar más de algunos minutos en ser creado, puesto que están diseñados para hacer que el trabajo sea más rápido, y no demorar su desarrollo.

Ventajas de realizar pruebas unitarias:

1. **Proporciona un trabajo ágil:** Permite poder detectar los errores a tiempo, de forma que sea posible corregir errores sin necesidad de tener que volver al principio y rehacer el trabajo. Gracias a que las pequeñas se van haciendo periódicamente y en pequeños packs. Disminuyendo el tiempo y el coste final de desarrollo.
2. **Calidad del código:** Derivado de la realización de pruebas continuamente y de la pronta detección de los errores, se llega un código limpio y de calidad.
3. **Detección Temprana de errores:** Los tests unitarios nos permiten detectar los errores rápidamente, analizando el código por partes, ejecutando pequeñas pruebas y de forma periódica.
4. **Facilita los cambios y favorece la integración:** Los tests unitarios nos permiten modificar partes del código sin afectar al conjunto, simplemente para poder solucionar bugs que nos encontramos por el camino.
5. **Proceso de Debugging:** Los tests unitarios ayudan en el proceso de debugging. Cuando se encuentra un error o bug en el código, solo es necesario desglosar el trozo de código testado.
6. **El diseño:** En el caso de que primero se creen los tests (*TDD*), resultará más sencillo conocer con anterioridad cómo debemos enfocar el diseño y ver qué necesidades debemos cumplir. De este modo, será más fácil llegar a una cohesión entre el código y el diseño.
7. **Reducción de coste:** Partiendo de la base que los errores se detectan a tiempo, puede llegar a optimizar los tiempos de entrega.

Es importante aclarar que las pruebas unitarias por sí solas, comprueban el código en pequeños grupos, pero no la integración total del mismo. Para ver si hay errores de integración es necesario realizar otro tipo de pruebas de software conjuntas y de esta manera comprobar la efectividad total del código.

4. Pruebas de Aceptación

Las **pruebas de aceptación** son las últimas pruebas realizadas donde el **cliente prueba el software y verifica que cumpla con sus expectativas**. Estas **pruebas** generalmente son **funcionales** y se **basan** en los **requisitos** definidos por el cliente y deben hacerse antes de la salida a producción.

Las pruebas de aceptación son fundamentales por lo cual deben incluirse obligatoriamente en el plan de pruebas de software.

Estas pruebas se realizan una vez que ya se ha probado que cada módulo funciona bien por separado, que el software realice las funciones esperadas y que todos los módulos se integran correctamente.

El **test de aceptación** termina de definir el nivel de calidad de la aplicación y le permite conocer al equipo qué tan bien supo interpretar correctamente los requerimientos del usuario o **Product Owner**.

Los usuarios y clientes suelen involucrarse en la ejecución de las pruebas de aceptación de software, siendo esto un mecanismo muy útil para ganar su confianza en el nuevo sistema o funcionalidad.

Habitualmente, el **Product Owner** le pide al equipo que realice **una demo** y valide todos los requerimientos definidos.

Para que el cliente o Product Owner apruebe las pruebas de aceptación no deben haber errores, como máximo algún error con criticidad muy baja.



5. Pruebas de Integración

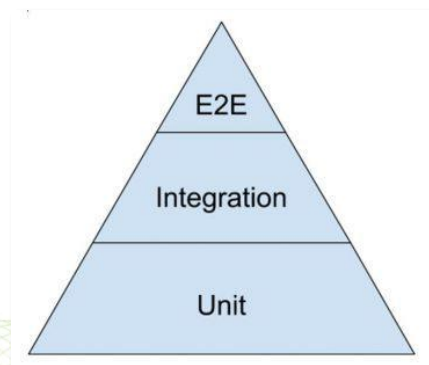
El objetivo de **las pruebas de integración** es **verificar** el correcto **ensamblaje** entre los **distintos componentes** una vez que han sido **probados unitariamente** con el fin de **comprobar** que **interactúan correctamente** a través de sus interfaces, tanto internas como externas, cubren la funcionalidad establecida y se **ajustan** a los **requisitos no funcionales** especificados en las verificaciones correspondientes.

En las pruebas de integración se examinan las interfaces entre grupos de componentes o subsistemas para asegurar que son llamados cuando es necesario y que los datos o mensajes que se transmiten son los requeridos.

Debido a que en las pruebas unitarias es necesario crear módulos auxiliares que simulen las acciones de los componentes invocados por el que se está probando y a que se han de crear componentes «conductores» para establecer las precondiciones necesarias, llamar al componente objeto de la prueba y examinar los resultados de la prueba, a menudo se combinan los tipos de prueba unitarias y de integración.



Las pruebas **E2E (End to End)**, simulan el comportamiento de un usuario real. Prueban toda la aplicación de principio a fin, cubriendo así secciones que las pruebas unitarias y las pruebas de integración no cubren.



6. Javadoc

Javadoc es una utilidad incluida en el JDK, que genera documentación automática en formato HTML, a partir de código fuente Java, mediante la inserción de comentarios con una sintaxis bien definida

Los comentarios deben seguir una serie de reglas, donde podrán convivir etiquetas *Javadoc*, junto con etiquetas *HTML*. Un ejemplo de documentación generada utilizando *Javadoc*, es el **API de Java**.

Mediante *Javadoc*, se documentan las clases, los métodos, constructores, paquetes y atributos. Esta documentación, servirá para identificar la funcionalidad de cada uno de los métodos o atributos, y facilitará la comprensión del código.

Sintaxis de Javadoc

```
/**
 * Descripción de la Clase/Método/atributo
 *
 * Etiquetas javadoc (todas ellas comienzan por @)
 *
 */
```

Dentro de los métodos y constructores, la primera frase describe lo que hace el método de forma breve, en las siguientes líneas se definirán las etiquetas Javadoc, que veremos a continuación.

En el caso de los atributos, se escribe una breve definición del mismo.

Etiquetas de Javadoc

ETIQUETA	DESCRIPCIÓN
@author	Autor de la clase.
@version	Versión de la clase.
@see	Referencia a otro componente, del mismo proyecto o de otro.
@param	Descripción parámetro.
@return	Descripción del valor de retorno. Solo si no es void.
@throws	Descripción de la excepción que puede propagar.

@deprecated	Indica que el método está obsoleto. Se mantiene por compatibilidad.
@since	Indica el nº de versión desde la que está implementado el método.

Ejemplo de Uso Javadoc

```
package com.gregoriofer.nominasycontabilidad;

import java.util.Date;

/**
 * Clase Nomina
 *
 * Contiene información sobre la nómina de cada programador
 *
 * @author EntornosDesarrollo
 * @version 1.0
 */
public class Nomina{

    //Constantes

    /**
     * Constante SALARIO_BASE
     */
    public final static double SALARIO_BASE=1100;

    /**
     * Indica el valor numérico del mes con paga extra 1
     */
    public final static int MES_EXTRA_1=6;

    /**
     * Indica el valor numérico del mes con paga extra 2
     */
    public final static int MES_EXTRA_2=12;

    //Atributos

    /**
     * Identificador de la nómina
     */
    private int idNomina;

    /**
     * Importe del Salario
     */
    private double salario;

    /**
     * Fecha de Pago de la nómina
     */
    private Date fechaPago;
```

```
//Metodos públicos

/**
 * Devuelve el id de la Nomina
 * @return idNomina
 */
public int getIdNomina() {
    return idNomina;
}

/**
 * Establece el valor del id de la Nomina
 * @param idNomina Identificador de la nómina
 */
public void setIdNomina(int idNomina) {
    this.idNomina = idNomina;
}

/**
 * Devuelve la fecha de pago de la nómina
 * @return fechaPago
 */
public Date getFechaPago() {
    return this.fechaPago;
}

/**
 * Establece de la fecha de pago de la nómina
 * @param fechaPago Fecha de pago de la nómina
 */
public void setFechaPago(Date fechaPago) {
    this.fechaPago = fechaPago;
}

/**
 * Devuelve el salarioBase
 * @return salarioBse
 */
public double getSalario() {
    return this.salario;
}

/**
 * Comprueba que la nómina ha sido abonada
 * @return <ul>
 * <li>true: la nómina ha sido abonada</li>
 * <li>false: la nómina aún no ha sido abonada</li>
 * </ul>
 */
private boolean nominaPagada() {
    if(this.fechaPago.equals("")){
        return false;
    }
    return true;
}

/**
```


CENTRO DE ENSEÑANZA CONCERTADA
"Gregorio Fernández"

```
* Método para sumar la paga extra si corresponde
* @param mes Mes actual, para confirmar si corresponde el pago
* @return <ul>
* <li>true: si se debe realizar paga extra</li>
* <li>false: si no se debe realizar</li>
* </ul>
*/
public boolean pagaExtra (int mes){

    boolean flagPagaExtra=false;
    if (mes==MES_EXTRA_1 || mes==MES_EXTRA_2){
        flagPagaExtra=true;
    }
    return flagPagaExtra;
}

//Constructor

/**
 * Constructor con 3 parametros
 * @param idNomina identificadorNomina
 * @param salario Salario a pagar
 * @param fechaPago fecha de pago de la nómina
 */
public Nomina (int idNomina, float salario, Date fechaPago){

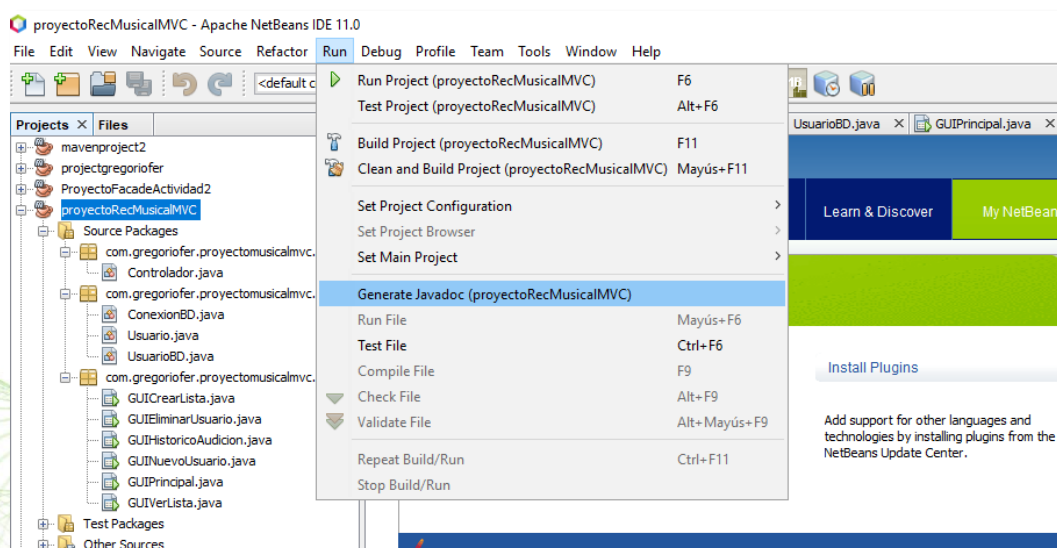
    this.idNomina=idNomina;
    this.salario=salario;
    this.fechaPago=fechaPago;

}

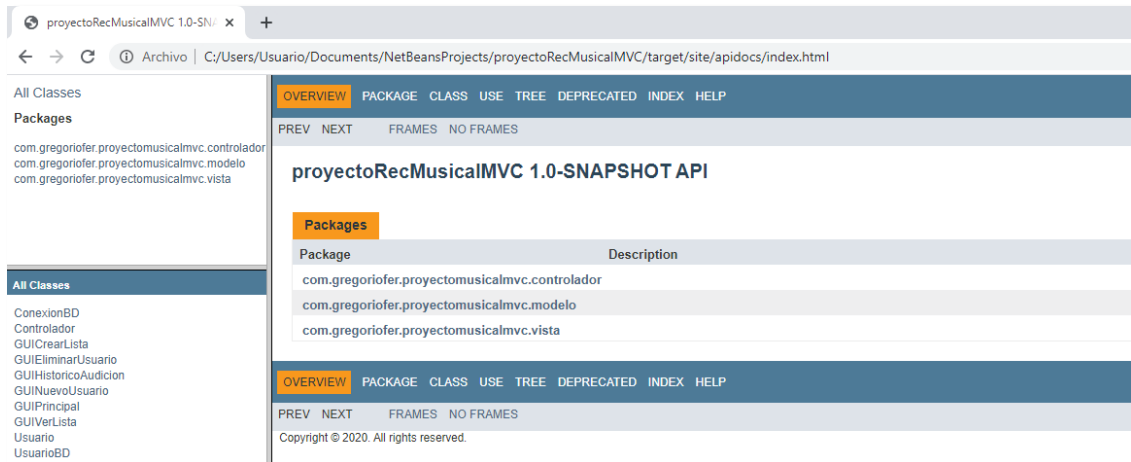
}
```

Generar la documentación JavaDoc en nuestro proyecto.

Run -> Generate Javadoc



Documentación generada en formato *html*.



proyectoRecMusicalMVC 1.0-SNAPSHOT API

Package	Description
com.gregoriofer.proyectomusicalmvc.controlador	
com.gregoriofer.proyectomusicalmvc.modelo	
com.gregoriofer.proyectomusicalmvc.vista	

Copyright © 2020. All rights reserved.

7. Pruebas Unitarias con *JUnit* en *Netbeans*

El framework *JUnit*

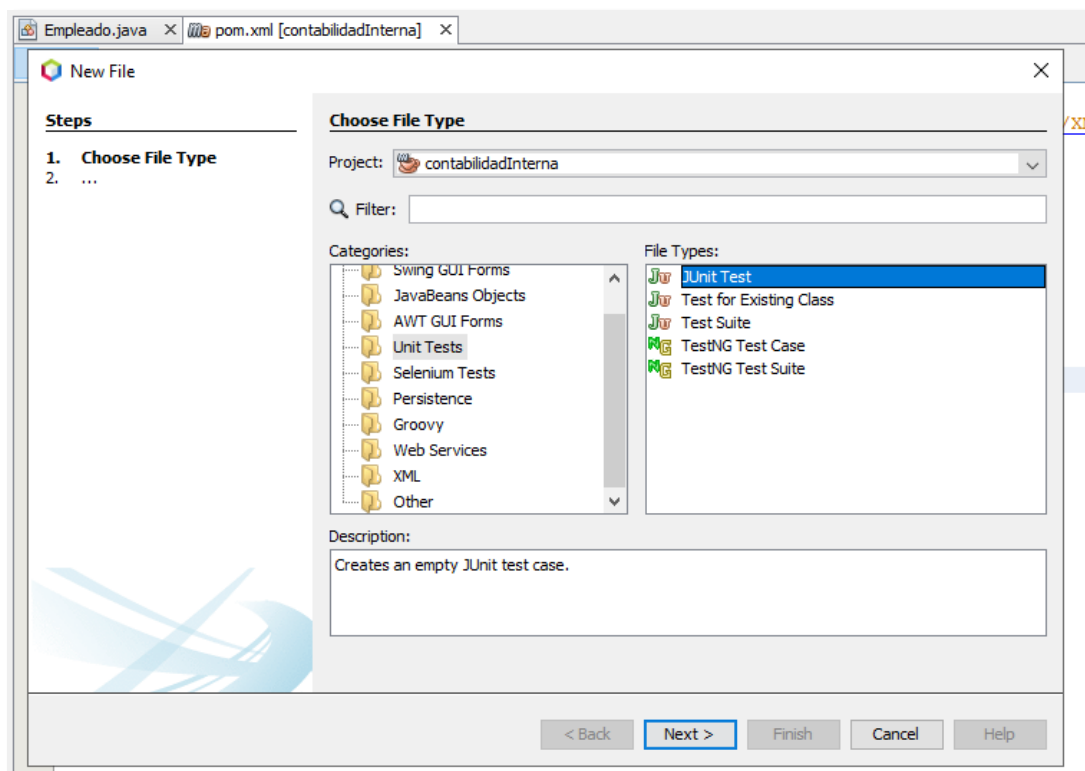
Existen diferentes frameworks para realizar pruebas unitarias para prácticamente cualquier lenguaje de programación.

En el caso de Java podemos incluso elegir entre varias opciones, pero utilizaremos **JUnit**, puede que la más consolidada.

Página principal de este framework. => <https://junit.org/junit5/>

Crear pruebas unitarias

Debemos saber que *JUnit* está integrado en *Apache Netbeans*, por lo que no necesitaremos instalar ningún software adicional.

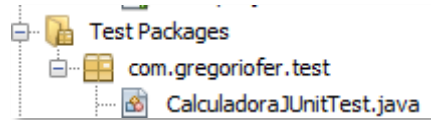


Supongamos que queremos crear unos test para la siguiente clase.

```
public class Calculadora {  
  
    private float resultado;  
  
    public float suma(float sumando1, float sumando2) {  
        return resultado = sumando1+ sumando2;  
    }  
  
    public float resta(float minuendo, float sustraendo) {  
        return resultado = minuendo - sustraendo;  
    }  
  
    public float multiplicacion(float factor1, float factor2) {  
        return resultado = factor1 * factor2;  
    }  
  
    public float division(float dividendo, float divisor) {  
        return resultado = dividendo / divisor;  
    }  
  
    public float getResultado() {  
        return resultado;  
    }  
}
```


La etiqueta @Test

En el paquete test crea la clase *CalculadoraTest*, y dentro de esa clase crea el siguiente método:



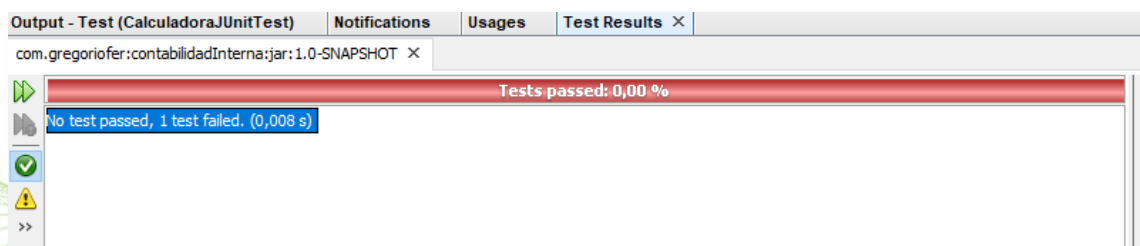
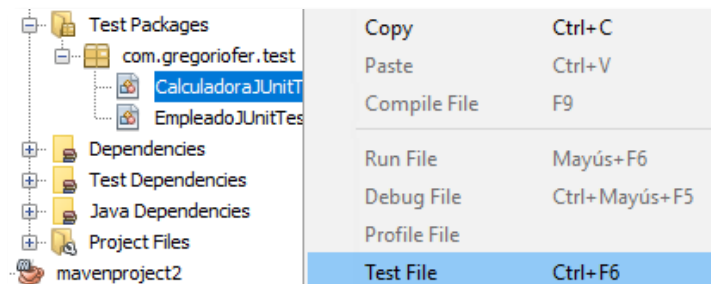
La etiqueta **@Test** marca un método como método de prueba. Debe incluirse inmediatamente antes del código test.

Los métodos de prueba siempre deben ser de tipo **public void** y comenzar por la palabra **'test'**

```
@Test
public void testSuma() {
    fail("Not yet implemented");
}
```

El método *fail(String)* hace fallar el test.

Si ejecutamos el test, tendremos el siguiente resultado:



CENTRO DE ENSEÑANZA CONCERTADA
"Gregorio Fernández"

Vamos a definir de forma completa el código de la prueba de la Suma:

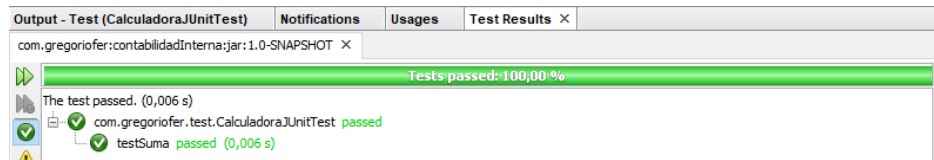
```
@Test
public void testSuma() {

    Calculadora calculadora = new Calculadora();
    assertEquals(2f, calculadora.suma(1, 1));

}
```

`assertEquals(valorEsperado, métodoProbado)`

Compara el valor esperado con el devuelto.



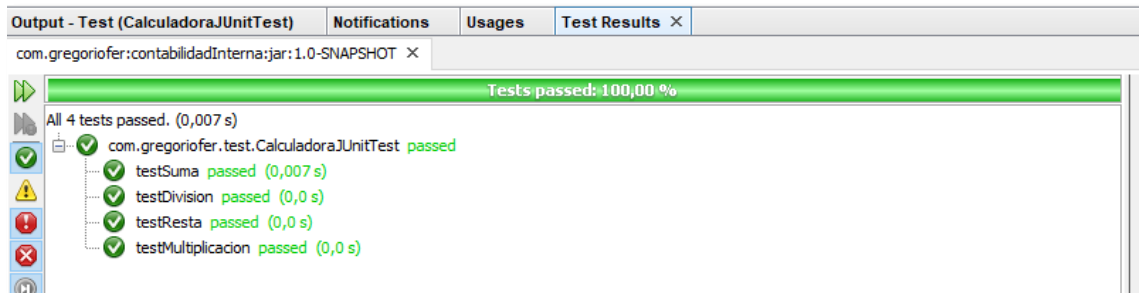
Como vemos, el test se ejecuta correctamente. Vamos a escribir los test para el resto de los métodos:

```
@Test
public void testResta() {
    Calculadora calculadora = new Calculadora();
    assertEquals(3f, calculadora.resta(4, 1));
}

@Test
public void testMultiplicacion() {
    Calculadora calculadora = new Calculadora();
    assertEquals(6f, calculadora.multiplicacion(2, 3));
}

@Test
public void testDivision() {
    Calculadora calculadora = new Calculadora();
    assertEquals(5f, calculadora.division(10, 2));
}
```

Si volvemos a ejecutar todos tests, veremos el siguiente resultado:



La etiqueta @BeforeEach

En el caso anterior, hemos tenido que crear una instancia de la clase Calculadora.

Sería muy útil poder indicar que un método se ejecuta repetidamente antes de cualquier test, y en él crear las instancias que necesitamos. Es la funcionalidad que proporciona la etiqueta @BeforeEach, marca un método (Habitualmente setUp()) que se ejecuta siempre antes que cualquier test

```
private Calculadora calculadora;

@BeforeEach
public void setUp() {
    calculadora = new Calculadora();
}

@Test
public void testSuma() {
    assertEquals(2f, calculadora.suma(1, 1));
}

@Test
public void testResta() {
    assertEquals(3f, calculadora.resta(4, 1));
}

@Test
public void testMultiplicacion() {
    assertEquals(6f, calculadora.multiplicacion(2, 3));
}

@Test
public void testDivision() {
    assertEquals(5f, calculadora.division(10, 2));
}
```

La etiqueta @AfterEach

De la misma forma, la etiqueta @AfterEach nos permite realizar tareas de limpieza después de ejecutar cada uno de los test.

```
private Calculadora calculadora;

@BeforeEach
public void setUp() {
    calculadora = new Calculadora();
}

@AfterEach
public void finish() {
    calculadora = null;
}
```

Aserciones Disponibles para las pruebas con JUnit.

- assertEquals(resultado esperado, resultado actual)
Exitoso si el valor esperado es idéntico al resultado de la invocación
- assertNull(objeto)
Exitoso si el objeto es null.
- assertNotNull(objeto)
Complementario del anterior.
- assertTrue(condición)
Exitoso si la condición pasada es verdadera.
- assertFalse(condición)
Exitoso si la condición pasada es falsa.
- assertEquals(Objecto1, objeto2)
Exitoso si las referencias de los objetos son idénticas.
- assertEquals(Objecto1, objeto2)
Complementario del anterior.