Diseño y realización de pruebas © EDICIONES ROBLE, S.L.

Indice

Diseño y realización de pruebas

I. Introducción

Todo software debe ser probado una vez ha sido terminado. Al principio de este módulo vimos que en todos los ciclos de vida del software teníamos una fase de pruebas y verificación.

Imagen 1. Ciclo de vida en cascada

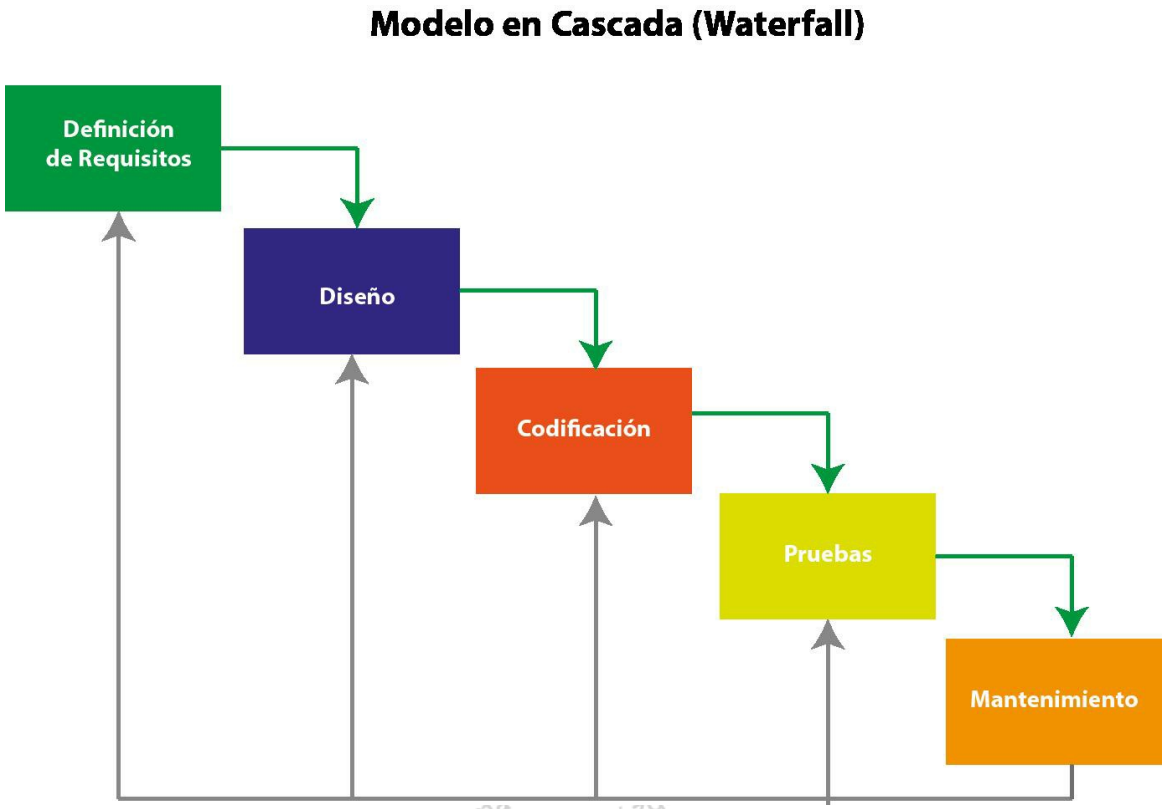


Imagen 1. Ciclo de vida en cascada.
Fuente: elaboración propia.

Esta fase es fundamental antes de entregar el producto al cliente para su explotación, aunque, dependiendo de la metodología elegida para el trabajo, las pruebas son una constante en todo proceso de desarrollo.

El software, cuando se termina, ha de someterse a una fuerte batería de pruebas para encontrar deficiencias en su funcionamiento, tanto de código como de usabilidad. Durante el proceso de desarrollo, ya sea en su totalidad o en partes, se ponen a disposición de los "testers" diferentes bloques de este para su prueba.

Para ello se dispone la aplicación en múltiples entornos de trabajo, los cuales van a ser diferentes dependiendo de la complejidad del desarrollo, pero en general se definen los entornos que mencionamos a continuación.

Imagen 2. Entornos de trabajo

Imagen 2. Entornos de trabajo.
Fuente: elaboración propia.

Entorno de desarrollo

Es el entorno inicial, donde **el equipo de desarrolladores trabajan realizando los códigos de la aplicación**. La infraestructura utilizada a nivel de software y hardware, en general, no va a coincidir con la prevista finalmente para el lanzamiento de la aplicación, de características algo más bajas con el objeto de reducir los costes de infraestructura.

Entorno de pre-producción

En esta fase se realizan las **pruebas para garantizar el funcionamiento de la aplicación** Estas pruebas serán de todo tipo, pruebas tanto para someter a la aplicación a todas las supuestas posibilidades de uso, como pruebas de carga para garantizar el rendimiento o incluso de usabilidad. Este entorno debe ser igual, en la medida de lo posible, al entorno de producción a nivel de arquitectura, de seguridad, hardware, software, etc.

El acceso a la aplicación estará permitido solo para el grupo de testeadores, y con datos que, si no son los reales, se deberían aproximar mucho.

Entorno de producción

Este es el entorno final, donde **la aplicación ya se encuentra en uso** Esta fase debería de carecer de fallos pues, de producirse, podrían causar grandes problemas a la empresa o usuarios/clientes. Por ello, antes de enviar un desarrollo a este entorno debe haber sido sometido a todas las pruebas necesarias para garantizar su correcto funcionamiento.

Al mismo tiempo, este entorno suele tener asociado una serie de planes de contingencia que deben ser implantados antes de su puesta en marcha. Pudiendo volver a versiones anteriores si fuese necesario o a la restauración de backups.

Se denomina entorno de trabajo al conjunto de estados por los que pasa el desarrollo de una aplicación.

II. Objetivos

Conocer los tipos de pruebas usados en el desarrollo del software.

Entender las características de la automatización de pruebas.

Trabajar en los ámbitos de aplicación de las pruebas y su automatización.

Comprender cómo se pueden automatizar las pruebas.

Aprender cuáles son las herramientas más usadas en la automatización de pruebas.

Conocer cuáles son los estándares sobre pruebas de software.

III. Contenidos

3.1. Pruebas en el desarrollo de software

3.1.1. Planificación

Para probar un software es necesario tener un plan de pruebas donde se realizará la programación a los que se someterá a la aplicación y la estimación de recursos que se tendrán en cuenta antes de perpetrar el proceso.

Para el establecimiento de los estándares asociados a cada caso de prueba a realizar se prestará atención a los requisitos establecidos en la primera fase del desarrollo.

La planificación es parte de la documentación de todo proyecto de software, donde **se establecerán ya desde un inicio el tipo y la forma en que se llevarán a cabo**. Asimismo, se debe contar con los recursos personales, y de hardware y con el tiempo para su realización.

plan de pruebas

Aunque un plan de pruebas variará dependiendo del tipo de proyecto, sus **partes más importantes** serían las siguientes:

El proceso de prueba

Descripción de cada una de las fases del proceso de pruebas.

Requisitos de trazabilidad

La planificación debe asegurar que se prueban todos los requisitos de forma individual.

Elementos a probar

Los elementos que van a ser probados.

Cronograma de pruebas

Un calendario de pruebas donde asignamos los recursos de personal, medios y tiempos.

estructura de un plan de pruebas

Para que un plan de pruebas resulte efectivo debe constar, como mínimo, de las siguientes partes:

Procedimientos de registro de prueba

La realización de las pruebas no solo se limita a ejecutar y verificar que todo funcione. Es imprescindible que los resultados obtenidos se documenten, por lo que deberán registrarse de forma sistemática. También tenemos que saber que prácticamente todos los desarrollos de software son sometidos a diferentes auditorías, por lo que la documentación detallada de cada proceso es fundamental.

Requisitos de hardware y software

En la elaboración del plan de pruebas es preciso definir los recursos de software y hardware que serán necesarios para poder realizar los test pertinentes.

Plan de contigencias

Se deben plantear posibles procedimientos o recursos para solucionar los imprevistos. Así como es imprescindible que un desarrollo en producción tenga medios de respaldo, en las fases de desarrollo también debemos disponer de los sistemas necesarios para solucionar posibles problemas, ya sean de personal o del sistema.

Definición de los casos de pruebas

Como su nombre indica, define y delimita los casos de prueba que deben aplicarse al sistema.

Puedes descargarte de esta página una plantilla para la realización de un plan de pruebas de software.

3.1.2. El proceso de pruebas

Durante el proceso de pruebas se valida y verifica la aplicación. Parece entenderse que “validación” es sinónimo de “verificación”, pero son términos con significados diferentes.

Validación

La validación es el proceso que comprueba que el sistema se esté construyendo de la forma adecuada.

Verificación

Mientras que la verificación se refiere a si el sistema está bien diseñado y sin errores de funcionamiento.

Es decir, la validación se refiere a la comprobación de que el sistema cumplirá con las necesidades reales del cliente, y la verificación a que carece de errores o fallos.

3.2. Tipos de pruebas

3.2.1. Pruebas estáticas

En las pruebas estáticas no es necesario ejecutar la aplicación; este tipo de pruebas**se centra en la inspección de los requisitos, prototipos o errores de en el código fuente** buscando posibles errores con el objetivo de eliminar defectos en fases tempranas del desarrollo.

Estas revisiones se pueden hacer de forma manual o con software específico dentro de las herramientas CASE.

En muchas ocasiones se trata de la revisión de la documentación, como, por ejemplo:

- Especificaciones de requisitos.
- Documentos de diseño.
- Códigos fuente.
- Planes de prueba.
- Casos de prueba.
- Scripts de prueba.
- Documentos de ayuda de usuario.
- Contenido de la página web.

3.2.2. Pruebas dinámicas

Aquí **se comprueba el** correcto o, mejor dicho, esperado **comportamiento del código generado**, de tal modo que la aplicación se someterá a estas pruebas cuando ya esté, al menos en parte, realizada. Para efectuar este tipo de pruebas es necesario ejecutar el código, de manera que se revisará su ejecución y se comprobará que cumple los requerimientos establecidos.

Además de la ejecución se comprobará que el consumo de recursos del sistema como uso de la CPU, memoria, etc. Dentro de este tipo de pruebas tenemos las **funcionales** y las **no funcionales**.

funcionales

En las pruebas funcionales se examinan los datos de entrada y se analizan los resultados obtenidos. Por tanto, su metodología es probar y validar que el software hace lo que se esperaba de él.

características

Características de una **prueba funcional**:

- Identificación de las funciones que el software debe realizar.
- Generación de los datos de entrada en base a las especificaciones de las funciones.
- Determinación de los elementos de salida en base a las especificaciones de la función.
- Ejecución de la prueba en base a los parámetros establecidos.
- Comparación de los resultados obtenidos frente a los esperados.

no funcionales

En contraposición, las denominadas pruebas no funcionales, no están dirigidas a probar las funciones, sino a otros aspectos de la aplicación, como pueden ser el rendimiento, la carga, la usabilidad, la seguridad, etc.

3.2.3. Pruebas estructurales (caja blanca)

Para testear la aplicación se elige una entrada. A partir de esta se examina la aplicación por los distintos caminos posibles, obtenidos a partir del código y por los que se puede ejecutar la aplicación, revisando las salidas que se producen.

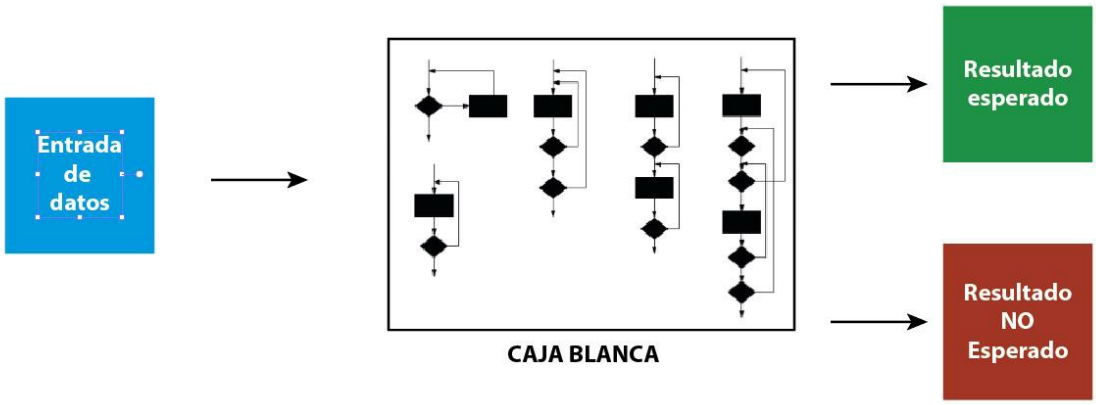


Imagen 3. En las pruebas estructurales el diseño y la implementación del sistema son conocidos por la persona que realiza la prueba.
Fuente: elaboración propia.

Durante estas pruebas, el testeador se centra en determinar cómo trabaja la ejecución del programa, por lo que la someterá a diferentes partes del código, como bucles, y a **casos de pruebas con distintos valores**.

Pruebas de caja blanca

Las pruebas de caja blanca se engloban dentro de los siguientes tipos de pruebas:

Pruebas unitarias

Se testean los posibles resultados con parámetros.

Pruebas de integración

Se testean los posibles caminos entre las unidades

Pruebas de sistema

Se testean diferentes caminos entre los subsistemas.

ventajas de las pruebas de caja blanca

- Se pueden realizar en fases tempranas del desarrollo, pues podemos ir testeando las partes del código que estén realizadas.
- Se pueden llevar a cabo pruebas complejas, puesto que es posible verificar todos y cada uno de los posibles caminos.

inconvenientes

- El testeador debe tener profundos conocimientos de programación y ejecución, puesto que las pruebas están basadas en la interpretación del código fuente.
- Todo proceso de software tiene continuas modificaciones de código, por lo que debemos someter el programa a la batería de pruebas con cada cambio.

3.2.4. Pruebas de integración

Con las pruebas de integración comprobamos que las interacciones entre los distintos módulos de la aplicación funcionan correctamente.

Una aplicación está formada por multitud de elementos, que interactúan entre sí, enviando y/o procesando datos. Con estas pruebas no solo comprobamos que cada módulo hace lo que se le pide, sino que interactúa con el resto de los módulos de forma correcta. Un ejemplo sería la interfaz de una aplicación, que se debe comunicar con el núcleo controlador del programa.

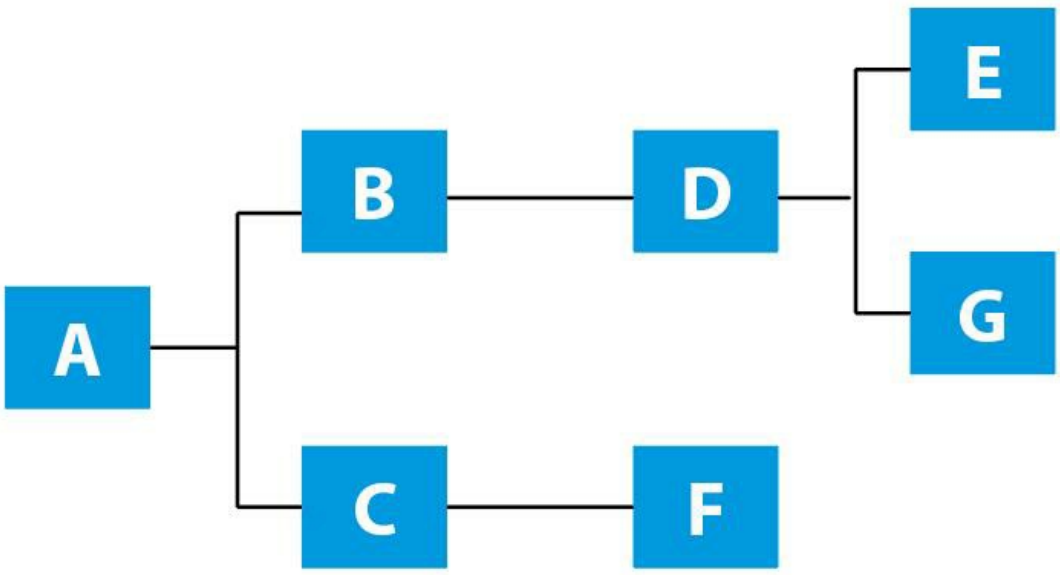


Imagen 4. Las pruebas de integración detectan los errores derivados de las interacciones entre las unidades que se integran.
Fuente: elaboración propia.

aspectos a atender

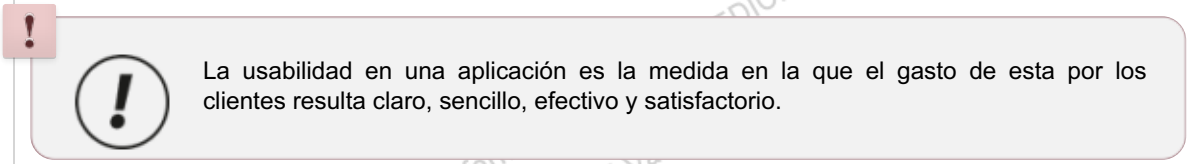
Para realizar las pruebas de integración es importante atender a los siguientes aspectos:

- Definir todas las interacciones entre cada unidad.
- Probar por separado cada módulo para asegurarse de que funciona antes de ser integrado.
- Realizar pruebas repetitivas de regresión según se van incorporando componentes.

3.2.5. Pruebas de usabilidad y accesibilidad

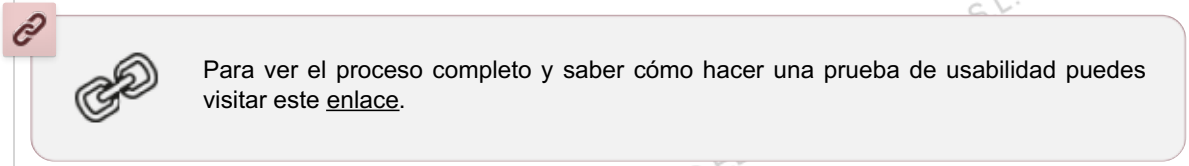
Usabilidad

En una aplicación es importante que ejecute los métodos que le codificamos correctamente. Pero también debemos tener en cuenta que el usuario es el que la utilizará. Por lo tanto, **el modo en que se presente es fundamental para el éxito**.



La usabilidad en una aplicación es la medida en la que el gasto de esta por los clientes resulta claro, sencillo, efectivo y satisfactorio.

Para perpetrar estas pruebas se escoge un grupo de personas (normalmente cinco) y se les propone una serie de supuestos a realizar. Estas personas no deben tener un perfil técnico, aunque sí ser expertos en temas de usabilidad, como usuarios finales de distintos perfiles.



Para ver el proceso completo y saber cómo hacer una prueba de usabilidad puedes visitar este enlace.

Accesibilidad

> Según la W3C, se habla de que un sitio web es accesible cuando “el acceso se realiza de forma universal, independientemente del tipo de hardware, software, infraestructura de red, idioma, cultura, localización geográfica y capacidades de los usuarios”.

Cada una de las categorías principales de discapacidad requiere un determinado tipo de adaptación. Se trata de realizar aplicaciones adaptadas para que accedan usuarios con discapacidad visual, de audición, de motricidad, cognitiva, etc.

Para más información sobre las pruebas de accesibilidad puedes visitar esteenlace.

3.2.6. Pruebas de caja negra (detección de errores)

Las pruebas de caja negra suelen ser de tipo funcional, pero también incluyen algunas no funcionales, como, por ejemplo, una prueba de rendimiento.

Las pruebas de caja negra o pruebas de “comportamiento”**no requieren que el testeador vea el código que se está ejecutando**, de ahí su nombre de “caja negra”. Estas pruebas se realizan desde el punto de vista del usuario, por lo que no es necesario que el testeador tenga un perfil técnico.

Básicamente consisten en someter a la aplicación a diferentes pruebas de uso y comprobar que el resultado sea el buscado. Con este tipo de pruebas podemos localizar los siguientes errores:

Errores

- Funciones incorrectas o que no existan.
- Errores en la interfaz.
- Errores en el acceso a los datos.
- Errores en el comportamiento.
- Errores en el inicio o la terminación de la aplicación.

Gracias a estas pruebas podemos encontrar errores de funcionamiento, como un error en la interfaz o fallo al procesar algún dato. Además, detectaremos si la aplicación cumple con las especificaciones, puesto que se realizan con respecto al usuario, en muchos casos personal, del propio cliente.

La gran **desventaja** de las pruebas de caja negra es que **requieren que la aplicación esté prácticamente terminada**, al menos los módulos para probar. Por lo tanto, un solo error suele producir grandes cambios en el desarrollo.

3.2.7. Pruebas de seguridad

Las pruebas de seguridad son las encargadas de confirmar que el **software está libre de amenazas internas o externas** procedentes, bien de personas, bien de un software malicioso.

Como dicen los expertos en seguridad: "la seguridad total no existe", pero es bueno realizar una serie de comprobaciones, para garantizar que estamos exentos de los peligros más frecuentes.

En resumen, con estas pruebas se verifica:

- Si existe una autenticación adecuada.
- Si el control de acceso es adecuado.
- Si el software es capaz de mantener la confidencialidad de los datos.
- La disponibilidad del software en caso de ataque por parte de *hackers* y programas maliciosos.

3.2.8. Pruebas de rendimiento

No siempre disponemos de la información necesaria para saber cuáles son las necesidades reales que tendrá una aplicación. Normalmente dichas necesidades se van incrementando con el tiempo de vida del software.

Pero sí que debemos **establecer unos parámetros iniciales que garanticen su funcionamiento**, partiendo de un número de usuarios que puedan acceder concurrentemente o teniendo en cuenta la carga de datos máxima a la que se someterá la aplicación.

Los **valores** que comúnmente se suelen medir son:

Tiempo de respuesta

Es la medida más solicitada en las pruebas de rendimiento. Afecta de forma directa a la experiencia con la aplicación del usuario final.

Uso de recursos

Suelen ser de tipo narrativo, como por ejemplo, “durante las pruebas no se alcanzaron nunca valores superiores al 60 % de utilización de la CPU”.

Para probar estas variables existen **dos tipos de pruebas clásicas** a las que debemos someter nuestra aplicación.

Pruebas de carga

Aquí se somete a la aplicación a una carga de peticiones que se considera más elevada a la que soportará la aplicación cuando esté en producción.

Pruebas de estrés

Este tipo de pruebas tiene como objetivo comprobar la robustez, la disponibilidad y la fiabilidad de una aplicación cuando es sometida a condiciones extremas.

3.3. Pruebas de código

3.3.1. Herramientas de testeo

La fase de pruebas en todo proyecto de software es un proceso laborioso. Es obvio que contar con herramientas que nos ayuden a automatizar las tareas donde configurando parámetros podamos construir una serie de pasos que nuestros códigos deban superar.

Imagen 5. Proceso de automatización de pruebas

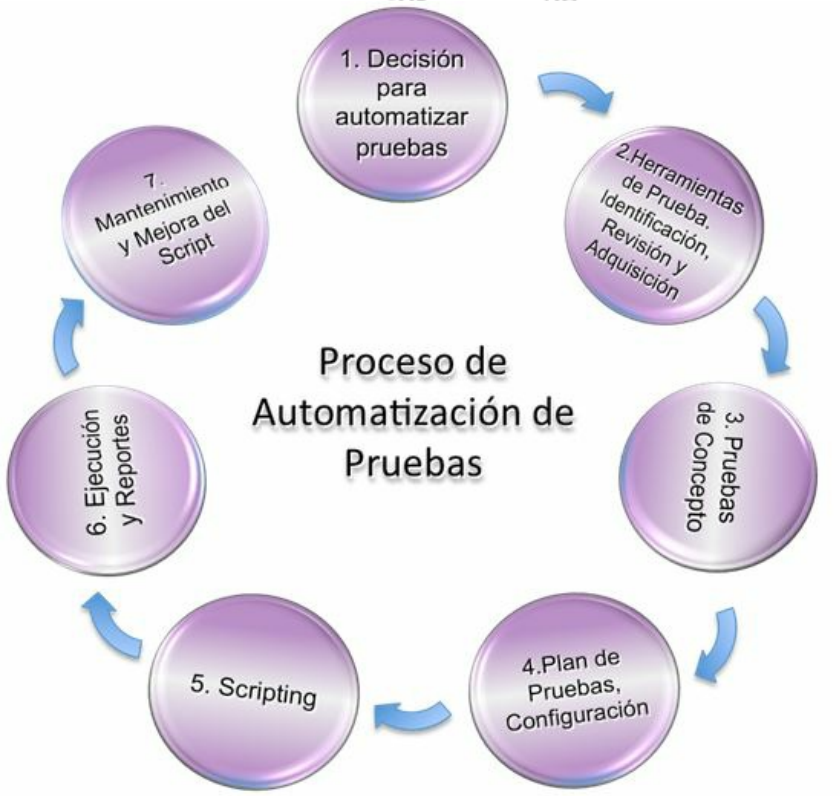


Imagen 5. Proceso de automatización de pruebas.
Fuente: http://toka-solutions.com/servicios.html

Para automatizar las pruebas tenemos muchas herramientas, cada una con funcionabilidades diferentes. Vamos a clasificarlas según su **objetivo**:

Gestor de pruebas

Este tipo de aplicaciones es una de las más usadas por los desarrolladores. Normalmente son frameworks de los propios lenguajes o herramientas integradas en los IDE, como el caso de JUnit.

Con JUnit podemos automatizar las pruebas unitarias de una aplicación Java. Mediante el conjunto de clases de Java podemos confirmar si los métodos implementados a partir de unos valores de entrada comprueban los datos de salida. Si estos datos no son los esperados, nos devolverá un error.

Analizador dinámico

Con esta herramienta podremos introducir fragmentos de código en nuestro programa para testear su comportamiento.

Por ejemplo, las veces que se ejecuta un bucle.

Oráculo

Es una herramienta que **predice los resultados** que se generan **a partir de los datos de prueba**. Esta herramienta puede ser la misma aplicación de una versión anterior o bien un prototipo creado para este fin.

Comparador de ficheros

Herramienta que **compara los resultados obtenidos con otros de pruebas previas**. Se utiliza en pruebas de regresión, donde se parangonan pruebas de distintas versiones.

Generador de informes

Proporciona la **definición de informes para los resultados de las pruebas**.

Generador de datos de pruebas

Genera los datos de pruebas para el programa en cuestión.

Estos se pueden obtener desde una BBDD o bien de forma aleatoria.

Simulador

Pueden ser, por ejemplo, pruebas de interfaces de usuario, que son **programas guiados por scripts** que simulan numerosas interacciones de los usuarios.

3.3.2. Automatización de pruebas

En el punto anterior ya hemos comentado cómo mediante el uso de herramientas podemos agilizar el trabajo de la realización de las pruebas, que pueden ser de dos tipos:

Pruebas codificadas

Se automatizan las pruebas unitarias utilizando casos de uso.

Pruebas de usuario

Se graban acciones que realizarán los usuarios sobre la interfaz de la aplicación a evaluar. Con esto se consigue que las acciones repetitivas puedan ser ejecutadas las veces que haga falta.

Existen **entornos de pruebas** dependiendo del lenguaje de programación:

JUnit

Sistema para realizar pruebas unitarias con el lenguaje JAVA.

TestNG

Creado para suplir algunas deficiencias en JUnit.

JTiger

Basado en anotaciones, como TestNG, también para Java.

SimpleTest

Entorno de pruebas para aplicaciones realizadas en PHP.

PHPUnit

Sistema para la realización de pruebas unitarias en PHP.

CPPUnit

Versión del framework para lenguajes C/C++.

FoxUnit

Entorno de pruebas para Microsoft Visual FoxPro.

MOQ

Entorno para la creación dinámica de objetos simuladores (mocks).

QUnit

Librería para pruebas unitarias en JavaScript. Creada por la fundación jQuery.

3.4. Pruebas unitarias con JUnit

3.4.1. Automatización de pruebas

Mediante **las pruebas unitarias garantizamos que una parte del código funcione como se espera** Este tipo de pruebas son codificadas por el usuario con una serie de funciones o métodos de una clase.

Este conjunto de clases (en el caso de JAVA), se denomina Frameworks y nos proporciona las herramientas a modo de métodos y anotaciones, para llevar a cabo las pruebas codificando algoritmos que comparan el resultado dado por el programa con aquel esperado.

1

Para JAVA tenemos el caso de JUnit. Este framework resulta muy útil para realizar y automatizar las pruebas. Como ya hemos comentado, una prueba unitaria se caracteriza por realizar una comparación del resultado esperado de un método con lo retornado por la codificación.

Esta **comparación es válida con cualquier tipo de dato**, ya sean numéricos, cadenas de texto, booleanos o incluso excepciones. El único requisito para que una prueba unitaria resulte efectiva es que tenga únicamente dos posibilidades de resolución: (correcta/incorrecta).

2

JUnit forma parte de las versiones más actuales de Netbeans y Eclipse, aunque si usamos alguna versión que no lo traiga incluido, se puede instalar desde el “market” de cada aplicación fácilmente.

3

ventajas

Las ventajas principales de JUnit son las que enumeramos a continuación:

- Es de código abierto.
- Podemos crear anotaciones para identificar los métodos de ensayo.
- Funciona a base de aserciones.
- Es sencillo de aplicar.
- Se pueden ejecutar de forma automática.
- Permite realizar suites de pruebas.
- Además del error nos proporciona información.

4

Las clases principales que nos encontramos en JUnit son:

Assert

Contienen un conjunto de métodos de aserción.

TestCase

Contienen un caso de prueba definido como accesorio para ejecutar varias pruebas.

TestResult

Disponen de métodos para recoger los resultados de la ejecución de un caso de prueba.

5

Además, contamos con **anotaciones** como @RunWith y @Suite, que nos permitirán ejecutar baterías de pruebas. JUnit se encarga de ejecutar todos los métodos testxxx() que contiene la clase.

6

En el siguiente ejemplo el único método con este nombre es testConcat(), por lo que JUnit lo ejecutará. Dentro de testConcat() hay una llamada al método assertTrue(). Este método comprueba si la expresión que recibe como argumento es cierta, y trasmite el resultado a JUnit.

```
import junit.framework.*;
//Un test de ejemplo sobre la clase String.
public class EjemploTest extends TestCase {
    public void testConcat()  {
            String s = "hola";
        String s2 = s.concat(" que tal");
            assertTrue(s2.equals("hola  que tal"));
    }
    public static Test suite() {
        return new TestSuite(EjemploTest.class);
    }
    public static void main (String[] args) {
        junit.textui.TestRunner.run(suite());
    }
}
```

7

Para conocer más a fondo el uso de JUnit, tanto en NetBeans como en Eclipse, y su funcionamiento básico, te recomendamos las siguientes lecturas:

- **Tutorial JUnit (Eclipse):** JUnit es una biblioteca de Java que te ayuda a realizar pruebas unitarias.
- **Tutorial JUnit (NetBeans):** este tutorial presenta los conceptos básicos de escritura y ejecución de pruebas de unidad JUnit en el IDE NetBeans.

3.4.2. La clase JUnit

Como sucede con todo lo referente a JAVA, al hablar de JUnit estamos refiriéndonos a un conjunto de clases, desde las cuales ejecutaremos los métodos para implementar las pruebas. Estas clases se heredan de junit.framework.TestCase. En el caso de los métodos de prueba veremos que comienzan por test.

```
import junit.framework.*;
public class CuentaTest extends TestCase {
...
}
```

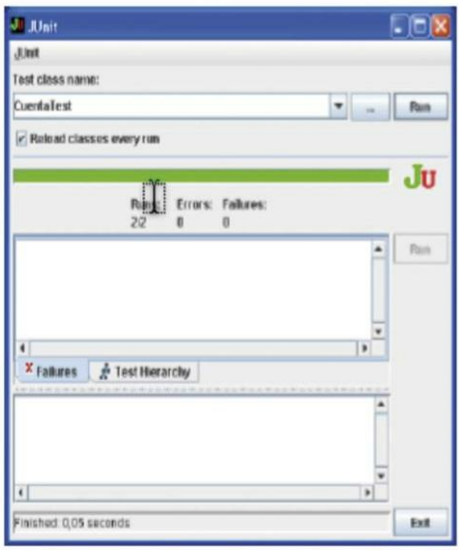
El procedimiento para realizar las pruebas es sencillo: una vez tenemos creada la clase, definimos los métodos (Test), donde ejecutaremos los métodos del programa. Después, mediante los assert, comprobaremos si el resultado es el que le indicamos.

```
public void testCuentaNueva () {
  Cuenta cuenta = new Cuenta();
   assertEquals(cuenta.getSaldo(), 0.00);
}

public void testIngreso () {
  Cuenta cuenta = new Cuenta();
     cuenta.ingresar(100.00);
  assertEquals(cuenta.getSaldo(), 100.00);
}
```

En el código anterior vemos dos métodos de prueba. En el primero usamos el constructor de nuestro programa y comprobamos que el saldo esté en 0. En el segundo utilizamos el método ingresar, donde añadimos 100 € y comprobamos que el nuevo saldo sea el esperado. Una vez que codifiquemos nuestros casos de prueba, la herramienta JUnit se encargará de validar los resultados y mostrarlos.

Si el caso de prueba falla



Funcionan correctamente

Imagen 6. Ventanas de JUnit para mostrar el resultado de la prueba.
Fuente: elaboración propia.

3.4.3. Métodos JUnit

JUnit dispone de los métodos Assert para probar que los resultados sean los esperados por la ejecución de nuestro código. Estos métodos permiten especificar un error, el resultado esperado y el resultado real.

Estos métodos son:

assertArrayEquals()
assertEquals()
assertTrue()
assertFalse()
assertNull()
assertNotNull()
assertSame()
assertNotSame()
assertThat()

Veamos algunos **ejemplos** de aplicación:

assertArrayEquals()

```
package ar.com.ladooscurojava.model.test;
import org.junit.Assert;
import org.junit.Test;
/**
 * @author ciber
 */
public class PersonaTest {
   @Test
   public void testCompararStringArray() {

      String[] arrayEsperado = {"Juan", "María", "Jose"};
      String[] arrayPrueba  = {"Juan", "María", "Jose"};
      Assert.assertArrayEquals(arrayEsperado, arrayPrueba);
   }
}
```

Con el método assertArrayEquals() comparamos el array de prueba con el array generado por nuestro código, determinando si el resultado es el esperado.

assertEquals()

```
package ar.com.ladooscurojava.model.test;
import org.junit.Assert;
import org.junit.Test;

/**
 * @author ciber
 */
public class PersonaTest {
   @Test
   public void testComprarIgual() {
      Assert.assertEquals("dato1", "dato2");
   }
}
```

Con el método assertEquals() comprobamos si el dato resultante tiene el valor deseado.

assertTrue() y assertFalse()

```
package ar.com.ladooscurojava.model.test;
import org.junit.Assert;
import org.junit.Test;
/**
 * @author ciber
 */
public class VehiculoTest {
   @Test
   public void testComprarTrue() {
      Assert.assertTrue(true);
   }
}
```

Con los métodos assertTrue() y assertFalse() validamos si un resultado es verdadero o falso.

3.4.4. Anotaciones JUnit

Para crear un método test dentro de una clase, tenemos que añadir la anotación “@test”.

A continuación, detallamos algunas **anotaciones** existentes:

@RunWith

Se le asigna una clase a la que se invocará en lugar del ejecutor por defecto.

@Before

Indicamos que el método se debe ejecutar antes de cada test (precede al método setUp).

@After

Indicamos que el siguiente método se debe ejecutar después de cada test.

@Test

Indicamos que es un método de test.

Ejemplo de anotación @Test:

```
public class Empleado{

@Test

public void testSueldo(){

float resultadoReal = Empleado.calcularSueldo(2000);

float esperado = 2200;

assertEquals(resultadoReal,esperado,0.01):

}

}
```

En algunos casos el resultado de la ejecución puede ser una excepción. Mediante la anotación @Test podemos indicar la excepción esperada.

```
@Test(expected=BRException.class)

public void testSueldo(){

Empleado.sueldoCalc(2500);

}
```

Otra forma de realizar la prueba es utilizar el método fail, que nos indica un fallo en la misma. De este modo, mediante un bloque try-catch, podemos capturar la excepción esperada si el método falla.

```
@Test

public void testSueldo(){

try{

Empleado.sueldoCalc(2500);

fail('Se esperaba un a excepcion')

}catch(BRException e)
```

3.4.5. Suites de pruebas

Cuando desarrollamos un programa, la cantidad de métodos puede crecer considerablemente. Esto provoca que debamos agrupar diferentes casos de prueba para ejecutarlos a la vez. Mediante las suites podemos agrupar métodos de prueba para ejecutarlos de forma conjunta.

```
@RunWith(Suite.class)

@SuiteClasses({Prueba1.class,Prueba2.class})

public class MiSuite{}
```

3.4.6. Ejemplo de uso

Vamos a realizar un ejemplo de uso de JUnit sobre una aplicación muy sencilla que calcule el precio con IVA de algún producto.

Imagen 7. Diseño de nuestra aplicación

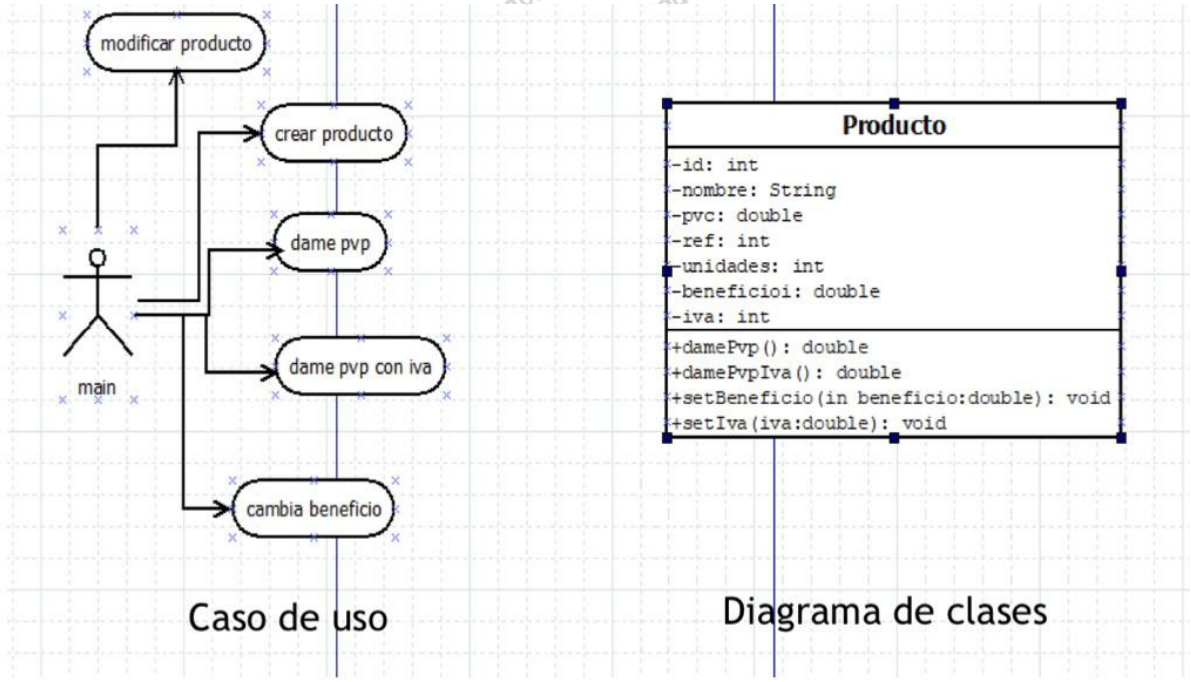


Imagen 7. Diseño de nuestra aplicación.
Fuente: elaboración propia.

Declaración de atributos y constructores.

Un par de métodos para calcular el PVP y el precio con IVA.

Una vez disponemos de nuestro pequeño programa, vamos a generar la clase de pruebas con JUnit.

Imagen 8. Crear una clase JUnit

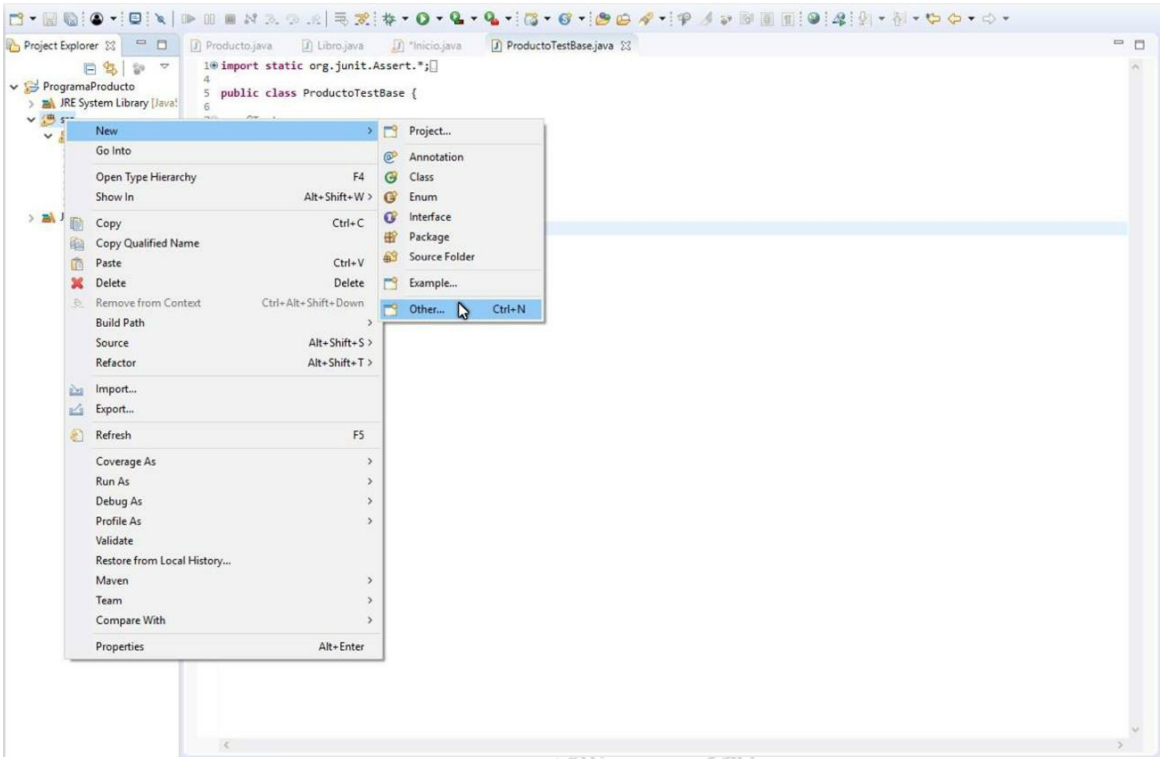


Imagen 8. Crear clase JUnit.
Fuente: elaboración propia.

Al ser la primera vez que lo utilizamos, Eclipse no nos mostrará el acceso en el menú inicial, por lo que pulsamos "Other...".

Imagen 9. Configuración de una clase

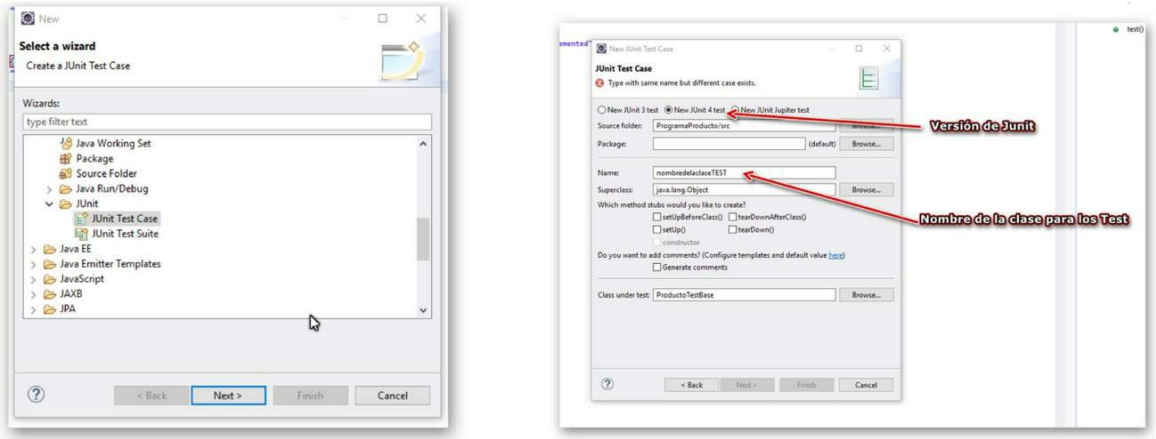


Imagen 9. Configuración de la clase.
Fuente: elaboración propia.

Seleccionamos la opción JUnit Test Case y en la ventana seleccionamos la versión de JUnit que vamos a utilizar y el nombre que le daremos a la clase de pruebas.

Programa

```
@Test
public void testDamePvp(){
Producto prod = new Producto();
//LÓGICA DE LA CLASE
prod.setPvc(100.00);
prod.setIva(1.21);
prod.setBeneficio(1.3);
double res = 130.00;
//RESULTADO ESPERADO
assertEquals(res, prod.damePvp(), 0.01);
}
@Test
public void testDamePvpIva(){
Producto prod = new Producto();
//MÉTODO DE PRUEBA
prod.setPvc(100.00);
prod.setIva(1.21);
prod.setBeneficio(1.3);
//ESPERADO
double res = 1.21*1.3*100.00;
assertEquals (res, prod.damePvpIva(),
0.01);
}
```

Métodos de prueba, el primero compara el valor del precio con PVP y el segundo con el IVA añadido.

Imagen 10. Resultado de la prueba

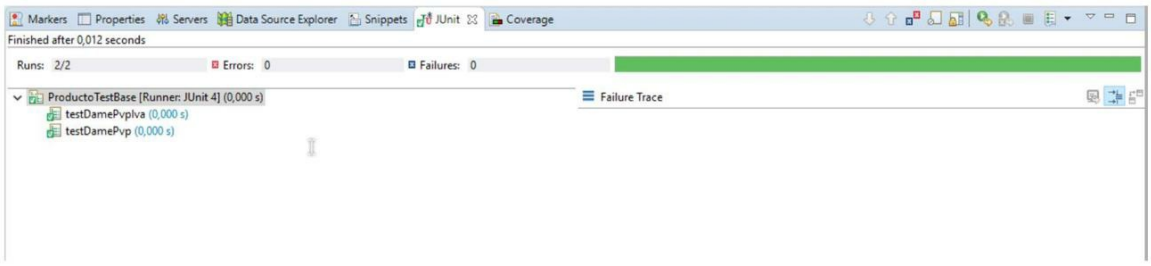


Imagen 10. Resultado de la prueba.
Fuente: elaboración propia.

3.4.7. Pruebas parametrizadas

En algunas ocasiones, no es suficiente comprobar el resultado con un solo dato, por lo que se necesita enviar una batería de datos para confirmar que el método funciona correctamente. Con JUnit podemos parametrizar pruebas, enviando mediante una lista una serie de datos para comprobar.

Ejemplo de prueba parametrizada:

```
@RunWith (Paremeterized.class)

public class ProductoParametrosTest {

private double pvc;

private double esperado;

public ProductoParametrosTest( double pvc, double esperado)
{

// TODO Auto-generated constructor stub

Producto prod = new Producto();

prod.setPvc(pvc);

this.pvc = prod.damePvp();

this.esperado = esperado;

}

@Parameters

public estatic Collection<Object[]> data (){

130.00} };

 }

Objetc[][] = data = new Object [][] { {10.00,13.00} , {100.00,

return Arrays.asList(data);

@Test

public void test() {

asserteEquals(pvc, esperado, 0)

 }

}
```


IV. Actividades interactivas

Actividad 1: Relaciona los siguientes conceptos

V. Resumen

En esta unidad, hemos aprendido:

- Los tipos de pruebas que podemos realizar para garantizar la calidad de nuestro software.
- La importancia de tener una estrategia o plan de pruebas para asegurar el buen funcionamiento de nuestro software.
- Las pruebas unitarias que se suelen realizar con componentes de software asociados a bloques de código, clases o métodos.
- Las herramientas proporcionadas por los frameworks de los lenguajes y los propios entornos de desarrollo.

VI. Lecturas obligatorias

- Guía Paso a Paso sobre Pruebas de Usabilidad para Sitios Web.
- Tutorial JUnit (Eclipse): JUnit es una biblioteca de Java que te ayuda a realizar pruebas unitarias.
- Tutorial JUnit (NetBeans): Este tutorial presenta los conceptos básicos de escritura y ejecución de pruebas de unidad JUnit en el IDE NetBeans.

Ejercicios

Ejercicio 1

Duración estimada del ejercicio

180 minutos

Este ejercicio es **opcional**, si tienes alguna duda puedes contactar con tu tutor.

Enunciado

Crea un proyecto java con la clase calculadora:

```
2  public class Calculadora {
3
4      private int num1;
5      private int num2;
6
7      public Calculadora(int a, int b) {
8          num1 = a;
9          num2 = b;
10     }
11
12     public  int suma() {
13         int resul = num1 + num2;
14         return resul;
15     }
16
17     public  int resta() {
18         int resul = num1 - num2;
19         return resul;
20     }
21
22     public  int multiplica() {
23         int resul = num1 * num2;
24         return resul;
25     }
26
27     public  int divide() {
28         int resul = num1 / num2;
29         return resul;
30     }
31 }
```

Crea con JUnit una clase de prueba para la clase Calculadora.

Modifica el método resta() de la clase Calculadora y añade los métodos resta2() y divide2() que se exponen a continuación. Crea después los métodos test para probar los tres métodos. Los métodos son:

```
17      public  int resta() {
18          int resul = num1 - num2;
19          return resul;
20      }
21
22      public boolean resta2() {
23          if(num1>=num2){
24              return true;
25          }else {
26              return false;
27          }
28      }
29
30      public Integer divide2() {
31          if(num2==0) {
32              return null;
33          }
34          int resul = num1/num2;
35          return resul;
36      }
37
```

Utiliza los métodos assertTrue(), assertFalse(), assertNull(), assertNotNull o assertEquals() según convenga.

Realiza pruebas parametrizadas para el método suma().

Metodología

- Revisa la información sobre los métodos a usar en:http://junit.sourceforge.net/javadoc/org/junit/package-summary.html
- Utilizando JUnit crea la clase de test y realizar las pruebas solicitadas.
- El alumno entregará el informe en PDF con los códigos y explicando los procesos generados subiéndolo a la plataforma.

Recursos

Enlaces de Interés

Casos de prueba: JUnit. 2018
http://www.jtech.ua.es/j2ee/publico/lja-2012-13/sesion04-apuntes.html
Dept. Ciencia de la Computación e IA (2014). Casos de prueba: JUnit. 2018, de Universidad de Alicante.

Bibliografía

- **Ramos, A., Ramos, Mª. J. *Entornos de desarrollo*. España. Ed. Garceta; 2014.** : Ramos, A., Ramos, Mª. J. *Entornos de desarrollo*. España. Ed. Garceta; 2014.

Glosario.

- **W3C**: (World Wide Web Consortium) es la comunidad internacional que desarrolla estándares que aseguran el crecimiento de la Web a largo plazo.