# UD4 - Diseño y realización de pruebas



## 1. Introducción a las pruebas de software

Las pruebas de software forman parte de una de las fases del ciclo de vida del software y tratan de detectar defectos cometidos en fases anteriores.

El objetivo de las pruebas de software es la validación y verificación del mismo.

- **Validación**: se realiza al finalizar por completo el desarrollo para determinar si satisfacen los requisitos especificados.
- **Verificación**: se realiza al final de cada fase para comprobar el cumplimiento de los requisitos de esa fase.

> **Ejemplo de la importancia de realizar pruebas de código**
>
> ARIANE 5 es un cohete de un solo uso diseñado para colocar satélites en órbita geoestacionaria y para enviar cargas a órbitas bajas. El vuelo 501 (04/06/1996) fue la primera prueba de vuelo del sistema de lanzamiento del Ariane 5. Fracasó porque 37 segundos después del lanzamiento, la lanzadera explotó debido al mal funcionamiento del software de control. El motivo de la explosión fue un fallo de software, el módulo de control no se había probado lo suficiente.
>
> Vídeo: TBT Launch: Ariane 5 Flight 501 (6-4-1996).

Esisten dos aspectos a los que hay que prestar atención para realizar pruebas en el software, en ellos hay que considerar:

1. La **estrategia de aplicación de las pruebas**, donde se fijan los elementos que van testear. (Punto 2 de esta unidad)
2. Las **técnicas de diseño de casos de prueba** que se van a utilizar para cada uno de los elementos seleccionados. (Punto 3 de esta unidad)

Otro concepto importante relacionado con las pruebas de código es la **depuración**, que es el proceso de identificar y corregir errores de programación. Es conocido también por el término inglés *debugging*, cuyo significado es "eliminación de bugs", manera en que se conoce informalmente a los errores de programación.

# 2. Estrategia de las pruebas de software

Las pruebas siempre se empiezan porlas partes pequeñas de la aplicación y se va incrementando poco a poco el alcance. Las pruebas que se suelen realizar son unitarias, de integración, de sistema y de validación y suelen realizarse secuencialmente.

## 2.1. Pruebas unitarias

Las pruebas unitarias son las primeras a las que se somete nuestro software y prueban las clases u objetos de nuestro código. En programación orientada a objetos, además de las clases tendremos que probar los métodos individualmente.

## 2.2. Pruebas de integración

En las pruebas de integración se comprueba si las clases que forman nuestro programa funcionan correctamente cuando interactuan. Se pueden realizar siguiendo dos estrategias diferentes:

- Prueba basada en hebra: integra el conjunto de clases requeridas para responder a una entrada concreta.
- Prueba basada en uso: primero prueba el funcionamiento de las clases independientes y después el de las dependientes de las anteriores.

## 2.3. Pruebas de sistema

Las pruebas de sistema comprueban el funcionamiento de un sistema integrado de hardware y software para comprobar si cumplpe los requisitos especificados. Se comprueban los requisitos funcionales y no funcionales, la documentación de usuario y el rendimiento del programa.

Las pruebas de sistema se pueden agrupar en:

- **Pruebas de recuperación**: se fuerza un fallo en el sistema y se comprueba que se puede recuperar correctamente. Se realiza en sistemas que deben tolerar fallos y que requieran recuperaciones rápidas.
- **Pruebas de seguridad**: tratan de encontrar lagunas de seguridad y proteger al sistema ante ataques imprevistos que podrían resultar en la pérdida de datos o información confidencial.
- **Pruebas de esfuerzo**: ponen al sistema ante situaciones extremas para comprobar su funcionamiento. Se prueban, por ejemplo, situaciones que requieran la memoria máxima de la que dispone el sistema.
- **Pruebas de rendimiento**: se realizan en sistemas en los que, además de requisitos funcionales, se deben cumplir ciertos requisitos de rendimiento como por ejemplo un tiempo de respuesta máximo.

- **Pruebas de despliegue**: comprueban el funcionamiento del sistema en diferentes plataformas o equipos en los que se va a poder utilizar. Por ejemplo, una web debemos probarla en diferentes navegadores.

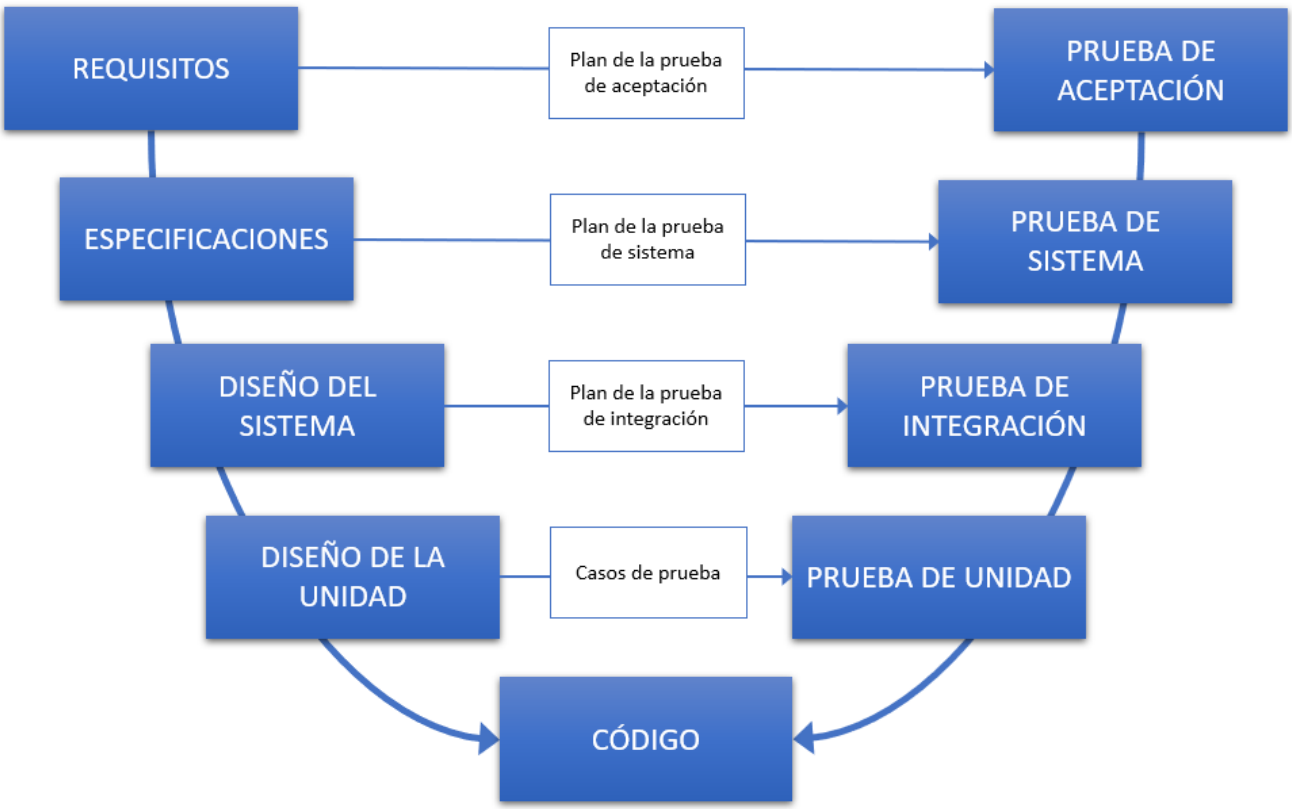
### 2.4. Pruebas de validación

El objetivo de las pruebas de validación es comprobar si el software es válido para el usuario y si está preparado para ser implementado en el entorno donde se va a usar.

Para considerar un programa como válido es importante tener claros los criterios de aceptación. Para ello es necesario fijar estos criterios en la ERS ( especificación de requisitos del sistema).

En las pruebas de validación participa el usuario final del producto y decide, junto al equipo de pruebas, si se ha alcanzado la versión final del software y si está listo para su explotación.

Para trabajar con la integración de las pruebas en el ciclo de vida se hace una modificación del modelo en cascada que conocemos como modelo en V. En este modelo se representan las fases de desarrollo de software junto a las de prueba.

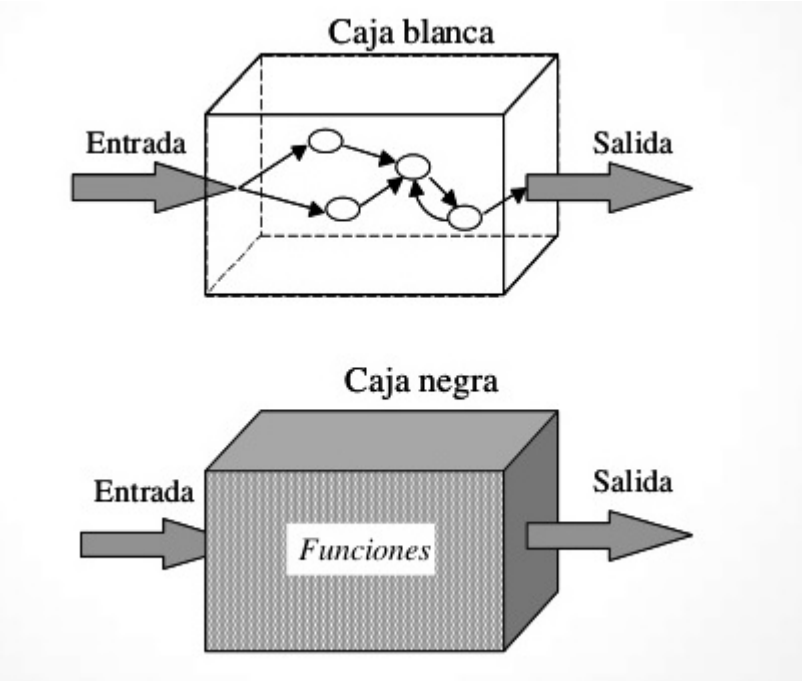



## 3. Diseño de casos de prueba

Las técnicas de diseño de casos de prueba son las diferente metodologías que tenemos para generar pruebas a nuestro software en función del objetivo. Tenemos dos tipos principales de pruebas:

- **Pruebas de caja blanca** o pruebas estructurales: examinan el código fuente y su ejecución. Los casos de prueba se seleccionan en función del conocimiento que se tiene de la estructura del método y es necesario disponer del código fuente.

- **Pruebas de caja negra** o pruebas funcionales: estudian el sistema desde fuera. Los casos de prueba se basan en las especificaciones del módulo, por lo que no se requiere conocimiento de su estructura interna, no es necesario disponer del código fuente.




> Práctica 1: Pruebas estructurares o de caja blanca

> Práctica 2: caja negra

> Practica 3: combinar caja blanca y negra

## 4. Documentación de pruebas

Se deben documentar el diseño y el resultado de las pruebas, para ello se crean dos documentos:

- **Plan general de pruebas**: documento con la planificación de las pruebas que se van a realizar. Incluye el enfoque, los recursos necesarios, las actividades a realizar en las prueba, el personal responsable y los riesos asociados.
- **Especificación del diseño de las pruebas**: detalla el plan general de pruebas. Este documento incluye otros dos:
    - Especificaciones de casos de prueba: recoge los datos de entrada que se van a utilizar en cada prueba junto a los resultados esperados y las dependencias entre casos de prueba.
    - Especificaciones de procedimientos de prueba: fija los pasos a seguir para la ejecución de las pruebas.

## 5. Depuración

Dentro del ciclo de vida del software estamos en la fase de pruebas, justo después de codificación o implementación


En esta fase, una ejecución del software o de parte de él nos permite identificar y corregir errores de programación.

Así pues, **la depuración** consiste en buscar y corregir errores que puede tener el código.

Tenemos herramientas software que nos asisten en esta tarea. Se les suele llamar **depuradores** y están, habitualmente, integrados en el IDE.

El depurador simula una ejecución real del software. En realidad se está ejecutando sobre una especie de máquina virtual que controla el estado de la memoria y del procesador virtual.

Estas herramientas nos van a permitir, entre otras cosas, parar la ejecución del software, ejecutarlo paso a paso, ver los valores de variables, objetos, etc.

Los pasos para realizar una depuración de código son:

- Ejecutaremos el código con todas las opciones posibles
- Anotaremos los problemas que surgen que pueden ser de dos tipos:
    - Errores funcionales (el programa no funciona o funciona mal).
    - Errores de coherencia con el diseño (el programa funciona pero no hace lo que debe hacer).
- Solucionamos los problemas

Para solucionar los problemas se recomienda seguir también una serie de pautas:

- Ejecutar el código en **modo debugging** para poder utilizar las herramientas del IDE para depuración
- **Localizar** el problema ¿En qué línea o zona del código se produce el error?

- Encontrar la **causa** del problema ¿Porqué se produce el error?
- Encontrar la **solución** del problema. Normalmente requerirá varias pruebas

Las utilidades que tienen las herramientas del IDE para depuración son:

- **Puntos de ruptura o breakpoints**: Se trata de puntos de interrupción de la ejecución del código que podemos colocar donde queramos.
  Cuando el depurador llega a ese punto se detiene. Podemos colocar tantos punto de estos como queramos. Incluso,una vez parada la ejecución podemos reanudarla para que se haga solo una instrucción. De esta forma podemos ver que efecto tiene sobre el programa cada una de las líneas de código.
  La ejecución se para en el primer breakpoint que hayamos colocado.
  Tenemos varias utilidades para trabajar a partir de un breakpoint.
- **Watchpoints**: Permiten añadir una variable que no aparece en la ventana de variables para consultar su valor. Permite combinar variables con expresiones algebraicas.
- **Cambiar el valor de una variable**: Se puede modificar el valor de una variable de forma manual.

> Práctica 4: debugger

# 6. Pruebas automáticas

JUnit es un conjunto de bibliotecas que se utilizan para hacer pruebas unitarias de aplicaciones Java.

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.

Se puede encontrar más información sobre JUnit en la web del proyecto.

## Framework

JUnit 5 se compone de 3 módulos principales:

- Plataforma: sirve como base para lanzar un marco de prueba en la JVM y define la API TestEngine para desarrollar un marco de prueba que se ejecuta en la plataforma.
- Júpiter: incluye el nuevo modelo de programación para escribir pruebas y la extensión modelo para escribir extensiones en JUnit 5.
- Vintage: proporciona un TestEngine que permite la compatibilidad con versiones anteriores de JUnit 4 o incluso JUnit 3.

## Configurar Junit5 con Maven

En el fichero pom.xml

```
<dependencies>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-api</artifactId>
            <version>5.6.0</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-params</artifactId>
            <version>5.6.0</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-engine</artifactId>
            <version>5.6.0</version>
            <scope>test</scope>
        </dependency>
 </dependencies>
 <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>2.22.2</version>
            </plugin>
        </plugins>
   </build>
```

Las tres dependencias son generadas automáticamente por NetBeans con Maven pero el plugin hay que incorporarlo cuando queremos utilizar ciertas anotaciones como por ejemplo @Disabled.

**Imports**

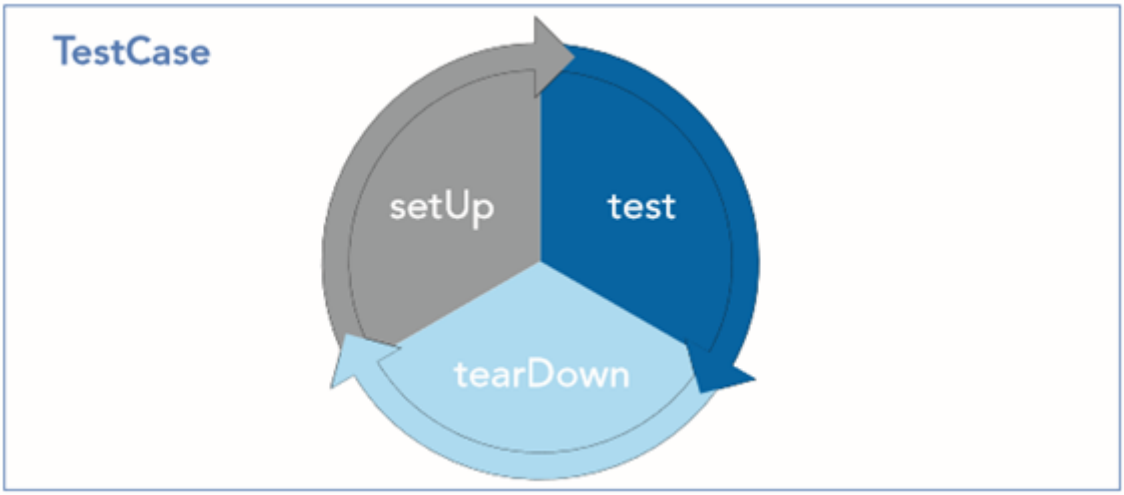
La importación de las versiones de Junit5:

```
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*; // paquete que contiene las
afirmaciones
```

## Ciclo de vida de Junit

Un test Junit puede contener varios test de métodos. Cada método identificado como test será ejecutado siguiendo el ciclo de vida de Junit.
Este consiste en tres partes: setUp, test y tearDown, y todo método se ejecuta siguiendo esta secuencia.
Los métodos de prueba deben indicarse con la anotación **@Test**.

Para crear los métodos @Test seguimos el patrón AAA (Arrange, Act, Assert).




Ejemplo: Tenemos una clase calculando con el método subtract()

```
@Test
public void testSubtract() {

        //Arrange
        double number1 = 0.0;
        double number2 = 0.0;
        Calculando instance = new Calculando();
        double expResult = 0.0;

        //Act
        double result = instance.subtract(number1, number2);

        //Assert
          assertEquals(expResult, result, 0.0);
```

## Anotaciones de Junit 5

| ANOTACIÓN | USO |
|---|---|
| @Test | Indica que un método es un método de prueba. A diferencia de la anotación de JUnit 4 @Test, esta anotación no declara ningún atributo, ya que las extensiones de prueba en JUnit Jupiter funcionan en función de sus propias anotaciones dedicadas. Dichos métodos se heredan a menos que se anulen. |

| ANOTACIÓN | USO |
|---|---|
| @TestClassOrder | Se utiliza para configurar el orden de ejecución de la clase de prueba para @Nested clases de prueba en la clase de prueba anotada. Estas anotaciones se heredan. |
| @TestMethodOrder | Se utiliza para configurar el orden de ejecución del método de prueba para la clase de prueba anotada. |
| @TestInstance | Se utiliza para configurar el ciclo de vida de la instancia de prueba para la clase de prueba anotada. Estas anotaciones se heredan. |
| @DisplayName | Declara un nombre para mostrar personalizado para la clase de prueba o el método de prueba. Estas anotaciones no se heredan. |
| @BeforeEach | Indica que el método anotado debe ejecutarse antes de cada @Test método de la clase actual. Dichos métodos se heredan a menos que se anulen. |
| @AfterEach | Indica que el método anotado debe ejecutarse después de cada @Test método de la clase actual. Dichos métodos se heredan a menos que se anulen. |
| @BeforeAll | Indica que el método anotado debe ejecutarse antes que todos los @Test métodos de la clase actual. Dichos métodos se heredan (a menos que estén ocultos o anulados) y deben serlo (a menos que se use el ciclo de vida de la instancia de prueba static "por clase" ). |
| @AfterAll | Indica que el método anotado debe ejecutarse después de todos los @Test métodos de la clase actua. Dichos métodos se heredan (a menos que estén ocultos o anulados) y deben serlo (a menos que se use el ciclo de vida de la instancia de prueba static "por clase" ). |
| @Nested | Indica que la clase anotada es una clase de prueba anidada no estática. |
| @Tag | Se usa para declarar etiquetas para filtrar pruebas, ya sea a nivel de clase o de método. |
| @Disabled | Se utiliza para deshabilitar una clase de prueba o un método de prueba |

## Asserts (Afirmaciones)

JUnit proporciona métodos estáticos en la clase Assertion para probar ciertas condiciones. Estos métodos de afirmación típicamente comienzan con assert y permiten especificar el mensaje de error, el esperado y el resultado real. Un método assert compara el valor real devuelto por una prueba para el valor esperado, y se produce una **AssertionException** si la prueba de comparación falla.

### Método assertArrayEquals()

El método **assertArrayEquals()** probará si dos matrices son iguales entre sí. En otras palabras, si las dos matrices contienen el mismo número de elementos, y si todos los elementos de la matriz son iguales entre sí. Para comprobar si hay elemento de la igualdad, los elementos de la matriz se comparan utilizando sus métodos equals(). Más específicamente, los elementos de cada matriz se comparan uno a uno usando su

método equals(). Eso quiere decir, que no es suficiente que las dos matrices contienen los mismos elementos. También deben estar presentes en el mismo orden. Ejemplo:

```
public class PersonaTest {
    @Test
    public void testCompararStringArray() {
        String[] arrayEsperado = {"uno", "dos", "tres"};
        String[] arrayPrueba   = {"uno", "dos", "tres"};
        assertArrayEquals(arrayEsperado, arrayPrueba);
    }
}
```

Si las matrices son iguales, los assertArrayEquals() continuará sin errores. Si las matrices no son iguales, se produce una excepción, y la prueba es abortada. Cualquier código de prueba después de los assertArrayEquals() no se ejecutará.

**Método assertEquals() y assertNotEquals()**

El método **assertEquals()** compara si dos objetos son iguales, utiliza el método equals(). Ejemplo:

```
public class PersonaTest {
    @Test
    public void testComprarIgual() {
      assertEquals("uno", "dos");
    }
}
```

Si los dos objetos son iguales de acuerdo con la aplicación de sus equal(), el método de los assertEquals() devolverá normalmente. De lo contrario, el método assertEquals() lanzará una excepción, y la prueba se detendrá ahí.
El método **assetNotEquals()** compara si dos objetyos son distintos, utiliza el método equals().

**Método assertTrue() y assertFalse()**

Los métodos **assertTrue()** y **assertFalse()** simplemente validan un resultado si es verdadero o falso.Ejemplo:

```
public class PersonaTest {
    @Test
    public void testComprarTrue() {

      boolean comprobar= calculo>10;
      assertTrue(comprobar);
    }
}
```

### Método assertNull() y assertNotNull()

Los métodos **assertNull()** y **assertNotNull()** simplemente validan un resultado si es nulo o no.Ejemplo:

```
public class PersonaTest {
    @Test
    public void testCompararNull() {

        assertNull(persona);
    }
}
```

### Método assertSame() y assertNotSame()

Los métodos **assertSame()** y **assertNotSame()** prueban si dos referencias a objetos apuntan al mismo objeto o no. No es suficiente que los dos objetos son iguales. Debe ser exactamente el mismo objeto al que apunta. Ejemplo:

```
public class PersonaTest {
    @Test
    public void testCompararIgualPorReferencia() {
        assertSame("String", "String");
    }
}
```

### Test de Timeout

Podemos realizar sencillos test de rendimiento verificando que un test no exceda un tiempo límite de ejecución.
Ejemplo: La prueba dada PASARÁ porque la ejecución del método se completa en 2 segundos, mientras que assertTimeout() espera 3 segundos.

```
@org.junit.jupiter.api.Test
public void testGetSumaTiempo() throws InterruptedException {
    Assertions.assertTimeout(Duration.ofSeconds(3), () -> {
        getValue();
    });
}

public String getValue() throws InterruptedException {
    TimeUnit.SECONDS.sleep(2);
    return "";
}
```

> Práctica 5: JUnit 5