

## Contenido

Maven .....	2
Pruebas de integración .....	6
Proyectos multimódulos y dependencias .....	7
DbUnit (pruebas de integración con BD) .....	8
Resumen de test de integración: .....	11
Pruebas del sistema y aceptación.....	13
Pruebas del sistema .....	13
Pruebas de aceptación (Acceptance testing).....	14
Pruebas de propiedades emergentes funcionales: Selenium.....	15
Selenium IDE .....	16
Selenium WebDriver .....	17
Pruebas de propiedades emergentes NO funcionales .....	25
JMeter .....	27
Análisis de pruebas .....	28
Herramienta automática de análisis: JaCoCo .....	29
Planificación de pruebas .....	31

## Maven

Es una herramienta de construcción de proyectos Java. La construcción (build) de un proyecto es la secuencia de tareas a partir del código fuente para poder usar (ejecutar) nuestra aplicación. La secuencia de tareas se denomina **Build Script**. La secuencia de tareas “programadas” define el proceso de construcción del proyecto.

Maven tiene predefinida 3 secuencias de tareas. Cada secuencia se denomina **ciclo de vida**. El ciclo de vida por defecto tiene 23 tareas, denominadas **fases**. Una fase es un concepto lógico que podrá tener asociado algún ejecutable que la realice. Cada fase puede tener 0 o más acciones ejecutables asociadas (**goals**).

Cada acción que se ejecuta en cada fase se denomina **GOALS**. Por ej, la fase compile tiene asociada la goal compiler:compile. Cualquier goal pertenece a un PLUGIN, lo cual no es más que un conjunto de goals. Una goal puede asociarse a una fase. Un plugin tiene 1 o varias goals.

Secuencia de tareas (Proceso de construcción) **[CICLO DE VIDA]** > (contiene varias) tareas (0 o más acciones ejecutables) **[FASES]** > (una o más) acciones ejecutables **[GOAL]** (pertenece a un PLUGIN).

Ciclo de vida por defecto de Maven:

	Fase	plugin : goal	Acciones realizadas
1	process-resources	maven-resources-plugin : resources	<u>Copia</u> /src/main/resources en <u>target</u>
2	compile	maven-compiler-plugin : compile	<u>Compila</u> *.java de /src/main/java
3	process-test-resources	maven-resources-plugin : testResources	<u>Copia</u> /src/test/resources en <u>target</u>
4	test-compile	maven-compiler-plugin : testCompile	<u>Compila</u> *.java de /src/test/java
5	test	maven-surefire-plugin : test	<u>Ejecuta</u> los <u>tests</u> unitarios
6	package	maven-jar-plugin : jar	<u>Empaque</u> *.class + recursos en un .jar
7	install	maven-install-plugin : install	<u>Copia</u> el .jar en <u>repositorio local</u> (.m2)
8	deploy	maven-deploy-plugin : deploy	<u>Copia</u> .jar en <u>repositorio remoto</u>

Una goal no es más que un código ejecutable, implementado por algún desarrollador. Las goals son **CONFIGURABLES**. Para provocar la goal se debe “importar” el plugin que la contiene en pom.xml en la sección <build>. Si una goal no tiene una fase definida por defecto ni la definimos, la goal no se ejecutará.

```
<build>
  <plugins>
    <plugin>
      <groupId> org.apache.maven.plugins </groupId>
      <artifactId> maven-compiler-plugin </artifactId>
      <version> 3.9.0 </version>
    </plugin>

    <plugin>
      <groupId> org.apache.maven.plugins </groupId>
      <artifactId> maven-surefire-plugin </artifactId>
      <version> 3.0.0-M5 </version>
    </plugin>
  </plugins>
</build>
```

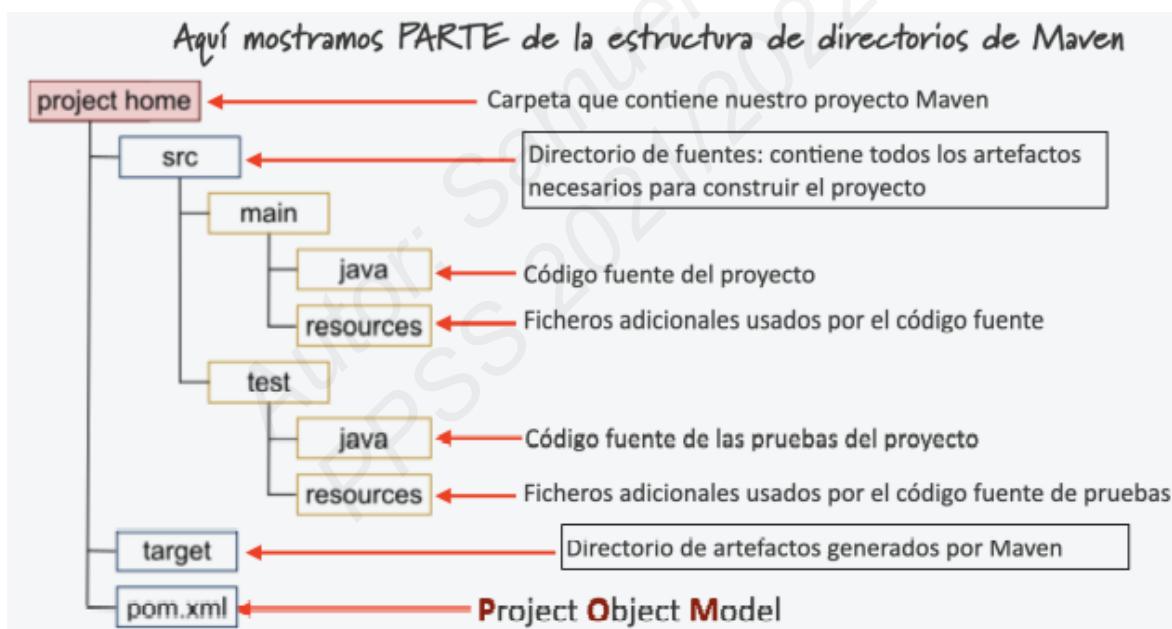
Un proyecto Maven debe contener en su raíz el pom.xml, que nos permite configurar la secuencia de acciones a realizar (build script) mediante la etiqueta <build>. También podemos indicar qué librerías (ficheros .jar) son necesarias con la etiqueta <dependencies>. De este fichero destacan 4 “secciones”: **coordenadas** (<groupId>, <artifactId>, <versión>, <package>), **propiedades** (<properties>, variables del pom), **dependencias** (<dependencies>, librerías .jar usadas), **proceso de construcción** (<build>, plugins con las goals que se ejecutarán en algún ciclo de vida).

Durante el proceso de construcción, Maven usa (y puede generar) ficheros empaquetados (**artefactos Maven**) identificados por sus coordenadas (groupId:artifactId:versión:package):

- groupId: identificador de grupo. Normalmente identifica la organización desarrolladora y puede usar puntos (org.ppss).
- artifactId: identificador del artefacto (nombre del archivo), suele ser el mismo del proyecto.
- version: versión del artefacto.
- Package: extensión del fichero, es opcional, por defecto es jar.

Los artefactos usados por Maven se almacenan en el repositorio local Maven (\$HOME/.m2/repository). Las coordenadas indican la ruta del fichero (org.ppss:practica1:1.0-SNAPSHOT -> \$HOME/.m2/repository/org/ppss/practica1/1.0-SNAPSHOT/practica1-1.0-SNAPSHOT.jar).

Estructura de directorios Maven: código fuente en **src/main/java**, código de pruebas en **src/test/java**, los ficheros y/o artefactos generados durante la construcción (.class) en directorio **target**, el cual se genera automáticamente en cada construcción del proyecto.



```

<groupId> ppss </groupId>
<artifactId> drivers </groupId>
<version> 1.0-SNAPSHOT </version>

<properties>
    <project.build.sourceEncoding>UTF-8 </project.build.sourceEncoding>
    <maven.compiler.source> 11 </maven.compiler.source>
    <maven.compiler.target> 11 </maven.compiler.target>
</properties>
  
```

Las librerías externas (**artefactos Maven**) se deben incluir en el pom.xml (sección <dependencies>) usando sus coordenadas Maven (las coordenadas que Maven usa para almacenar esas librerías en su repositorio remoto). Maven descargará la librería si no la tenemos ya. Para exportar el proyecto sólo hace falta el fichero pom.xml y el directorio src. Estas librerías de descargan en el directorio generado automáticamente .m2.

```
<dependencies>
    <!-- jUnit5 (Test, BeforeAll, BeforeEach, Assertions, Tag) -->
    <dependency>
        <!-- Directorio en .m2/repository -->
        <groupId>      org.junit.jupiter      </groupId>
        <!-- Carpeta del artefacto -->
        <artifactId>   junit-jupiter-engine   </artifactId>
        <!-- Versión del artefacto (cada versión es un directorio con el .jar de la versión) -->
        <version>     5.8.2   </version>
        <!-- A qué fase lo vamos a limitar. Indica que sólo se va a utilizar para compilar y ejecutar test -->
        <scope>       test    </scope>
    </dependency>

    <!-- @ParameterizedTest (ParametrizedTest, provider.Arguments, provider.MethodSource/ValueSource) -->
    <dependency>
        <groupId>      org.junit.jupiter      </groupId>
        <artifactId>   junit-jupiter-params   </artifactId>
        <version>     5.8.2   </version>
        <scope>       test    </scope>
    </dependency>

    <!-- EasyMock -->
    <dependency>
        <groupId>      org.easymock      </groupId>
        <artifactId>   easymock        </artifactId>
        <version>     4.3            </version>
        <scope>       test    </scope>
    </dependency>
</dependencies>
```

Para construir el proyecto, se usa el comando **mvn** seguido de la fase(s) o goal(s) que queremos realizar, indicando el plugin para las goals. Por ejemplo: mvn fase1 fase2 plugin1:goal3 plugin2:goal:4.

- Ejecutar una fase (mvn faseX) hace que se ejecuten todas las goals asociadas a las fases previas y la indicada.
- Ejecutar un goal (mvn plugin:goal) hace que se ejecute únicamente esa goal.

Direcciones que genera cada fase (partiendo del src/main, src/test y del pom.xml):

- **mvn compile:** /target/generated-sources, /target/clases, /target/maven-status/maven-compiler-plugin/compile
- **mvn test-compile:** /target/maven-status/maven-compiler-status/testCompile, /target/generated-test-sources, /target/test-classes
- **mvn test:** /target/surefire-reports
- **mvn package:** /target/maven-archiver, /target/Proyecto-Ejemplo-1.0-SNAPSHOT.jar

Las fases se ejecutan siempre en el mismo orden comenzando desde la PRIMERA !!!

FASES	PLUGIN : GOALS
1 validate	
2 initialize	
3 generate-sources	
4 process-sources	
5 generate-resources	
6 process-resources	resources:resource
7 compile	compiler:compile
8 process-classes	
9 generate-test-sources	
10 process-test-sources	
11 generate-test-resources	
12 process-test-resources	resources:testResources
13 test-compile	compiler:testCompile
14 process-test-classes	
15 test	surefire:test
16 prepare-package	
17 package	jar:jar
18 pre-integration-test	
19 integration-test	
20 post-integration-test	
21 verify	
22 install	install:install
23 deploy	deploy:deploy

### mvn test-compile

Ejecuta las goals de las fases 1..13  
Genera los .class de src/test/java

### mvn test

Ejecuta las goals de las fases 1..15  
Ejecuta los .class de src/test/classes y genera un informe

### mvn package

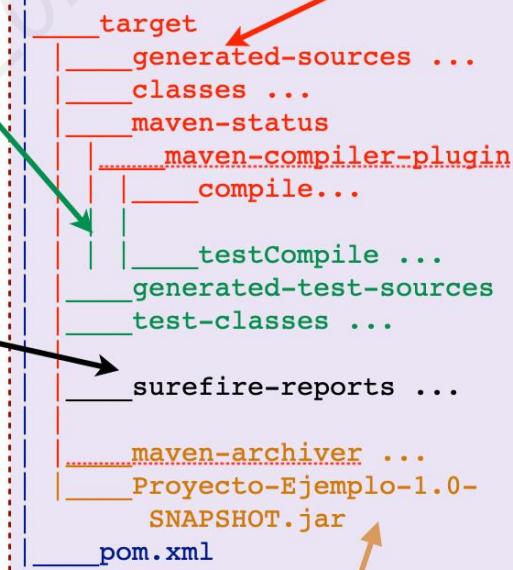
Ejecuta las goals de las fases 1..17  
Genera el .jar del proyecto



EJEMPLO

### mvn compile

Ejecuta las goals de las fases 1..7  
Genera los .class de src/main/java



Ejecuta las goals de las fases 1..17  
Genera el .jar del proyecto

## Pruebas de integración

Tras las pruebas unitarias, veremos si las piezas presentan errores (puede que tenga pero no los detectemos), y en caso de no haber encontrado, probaremos la combinación de varias piezas y probar las interacciones entre ellas. La verificación se enfoca en diferentes niveles:

- Nivel de unidades: pruebas unitarias. *Objetivo: encontrar defectos en las unidades. Cuestión fundamental: ¿Cómo aislar cada unidad del resto?*
- Nivel de integración: pruebas de integración. *Objetivo: encontrar defectos derivados de la interacción entre unidades. Cuestión fundamental: ¿Cuál es el orden a realizar para la integración?*

Se presentan problemas por malinterpretaciones, errores y descuidos (yo pensaba que tú me validabas unos datos y no es así).

Tras integrar cada pieza nueva al sistema, se realizan pruebas de regresión, en los cuales volvemos a probar el sistema completo para asegurar que el nuevo módulo no perturba el comportamiento del anterior sistema ya integrado. Estas pruebas son indispensables para cuando modificamos algún componente del sistema o modificamos una funcionalidad y cuando depuramos algún defecto. Para construir el sistema debemos seguir un orden o una estrategia que nos permita construirlo con éxito. Esta construcción ordena nos permite detectar errores en las interfaces (la manera en la que se comunican con el módulo, por ejemplo los parámetros) de los componentes del sistema.

Los tipos de interfaces son:

- A través de parámetros: los métodos.
- Memoria compartida: se escriben los datos en una memoria compartida con otro, el cual los lee.
- Interfaz procedural: encapsular un conjunto de sentencias bajo un nombre, que puede recibir o no parámetros.
- Paso de mensajes: los servicios web.

Los errores más comunes derivados de la interacción son:

- Mal uso de la interfaz: pasar por parámetro un tipo de dato incorrecto o un mensaje mal codificado.
- Malentendido sobre la interfaz: asumir que recibirás unos tipos de datos (ya validados, ya formateados, etc) y que no ocurra.

Guía para diseñar las pruebas:

- Examinar el código y listar cada llamada a componentes externos. Diseñar pruebas con los valores en los extremos de los rangos, los cuales pueden revelar inconsistencias con mayor probabilidad.
- Si se pasan punteros, probar punteros nulos.
- Cuando se invoca a un componente con interfaz procedural (instrucciones bajo un nombre), intentar que este falle.
- Pruebas de estrés a sistemas con paso de mensajes.
- Si se usa memoria compartida, probar a no dejar nada o alterar el orden de llamadas.

Para una correcta construcción del sistema, se debe seguir una estrategia de integración de los componentes, los cuales son:

- Big Bang: se integran todos a la vez. Para aplicaciones pequeñas o con pocas unidades.
- Top-down: primero los componentes con mayor abstracción (dep. externas) y luego las menores. Requiere de muchos dobles, adecuado para sistemas con interfaz de usuario compleja.
- Bottom-up: primero los componentes con menor abstracción. Requiere de menos dobles, adecuado para sistemas con infraestructura o lógica de negocio compleja.
- Sandwich: mezcla de Top-down y Bottom-up.
- Dirigida por los riesgos: primero las que tengan más probabilidad de fallar o que no debe fallar (por ej, un módulo de seguridad).
- Dirigida por funcionalidades: integras por funcionalidades. Primero los módulos de una funcionalidad, luego las de otro, hasta completarlas todas.

## Proyectos multimódulos y dependencias

Es un proyecto maven compuesto por diferentes sub-proyectos relacionados/agrupados. El proyecto “padre”, el cual agrupa a los demás proyectos “hijos” (módulos) está formado por un pom (*pom aggregator*) y los sub-proyectos. En el pom el empaquetado debe ser pom (<package> pom </package>) y debe indicar los módulos que lo componen con una etiqueta <modules> y una etiqueta <module> por cada módulo. Los módulos pueden estar en cualquier ruta, pero lo más práctico es que estén en el directorio del *pom aggregator*.



El mecanismo usado por maven para gestionar los proyectos multimódulo se denomina reactor, y se encarga de recopilar todos los módulos, ordenarlos y construirlos correctamente. El orden se define mediante las dependencias entre ellos.

Beneficios de usar multimódulos:

- Reducción de duplicación: Por ejemplo, podemos **construir todos los módulos** mediante **un único comando** (aplicando el *aggregator pom*).
- Compartir propiedades: estos elementos como propiedades, dependencias y plugins se comparten mediante **herencia**. Para indicar la herencia, en el pom del hijo debemos añadir la etiqueta <parent> con los datos del padre:

```
<parent>
  <artifactId>matriculacion</artifactId>
  <groupId>ppss</groupId>
  <version>1.0-SNAPSHOT</version>
</parent>
```

En el pom del padre (*pom aggregator*):

```
<groupId>ppss</groupId>
<artifactId>matriculacion</artifactId>
<packaging>pom</packaging>
<version>1.0-SNAPSHOT</version>
```

- Dependencias entre los módulos: podemos establecer una dependencia entre los módulos para que la construcción del proyecto global sea ordenada. Tomemos como ejemplo la siguiente imagen:



La imagen indica que BO tiene una dependencia sobre PROXY y DAO y DAO una dependencia sobre COMUN. Para indicar la dependencia de DAO, debemos agregar en su POM, entre las dependencias:

```
<dependency>
  <groupId>${project.groupId}</groupId>
  <artifactId>matriculacion-comun</artifactId>
  <version>${project.version}</version>
</dependency>
```

## DbUnit (pruebas de integración con BD)

La mayor parte de las aplicaciones de empresas utilizan Base de Datos (BD). Típicamente se utiliza una arquitectura de software por capas, donde la BD se sitúa entre las capas inferiores. Las pruebas de integración requieren que existan datos en la BD, sin embargo, ¿Cómo ejecutamos pruebas de integración sobre clases que dependen de una BD?, ¿Cómo nos aseguramos que realmente se lee y escribe correctamente sobre la BD?

La respuesta es DbUnit, un *framework* que permite controlar la dependencia con la BD. No se trata de una BD doble, si no que se utiliza la BD real, ya que ese es nuestro objetivo, comprobar el escenario real. El escenario típico de ejecución de pruebas es:

1. Eliminar cualquier estado previo de la BD (antes de ejecutar cada test, no se restaura la BD después de cada test).
2. Cargar los datos necesarios para las pruebas (sólo los datos **necesarios** para el test).
3. Ejecutar las pruebas utilizando la librería DbUnit para las aserciones.

La ejecución de cada test debe ser independiente del resto.

Interfaces de DbUnit:

- **ITable**: representa una tabla. Se suele utilizar para comprobar si la operación del SUT fue correcta o no con `Assertion.assertEquals(tablaEsperada, tablaObtenida);`.
- **IDataSet**: representa una colección de tablas. Se suele utilizar para:
  - Leer las tablas de un fichero XML con `IDataSet dataSet = new FlatXmlDataSetLoader().load("/nombreTabla");`
  - Recuperar las tablas de la BD (resultado tras invocar el SUT) con `IDataSet dataBD = (IDatabaseConnection) connection.createDataSet();`
  - Obtener una tabla específica de un dataSet con `ITable tabla = dataSet.getTable("nombreTabla");`
- **FlatXmlDataSet/FlatXmlDataSetLoader**: crea datasets a partir de documentos XML con el siguiente formato (cliente es una tabla con datos id, nombre, apellido, dirección y ciudad):

```
<?xml version="1.0" encoding="UTF-8"?>
<dataset>
    <cliente id="1">
        nombre="John"
        apellido="Smith"
        direccion="1 Main Street"
        ciudad="Anycity" />
</dataset>
```

- **IDatabaseTester**: permite el acceso a la BD, creándose con los parámetros de conexión a la BD, definiendo la inicialización de esta y devolviendo la conexión a la BD con el tipo **IDatabaseConnection**.

- La construcción es:

```
String cadenaConexionDB =
"jdbc:mysql://localhost:3306/matriculacion?useSSL=false";
String driverBBDD = "com.mysql.cj.jdbc.Driver";
/*
    - driverBBDD: clase que implementa el driver (mysql-connector-java)
    - cadenaConexionDB: cadena de conexión
(JavaDataBaseConection:mysql://ruta:puerta/nombreBBDD?PARAMS)
    - username y password del usuario para acceder a la BBDD
*/
IDatabaseTester databaseTester = new MiJdbcDatabaseTester(driverBBDD,
cadenaConexionDB, "usuarioBD", "contraseñaBD");
```

- La obtención de la conexión con la BD (**IDatabaseConnection**) se obtiene con `IDatabaseConnection connection = databaseTester.getConnection();`
- La inyección del dataset inicial para la BD se realiza con `databaseTester.setDataSet(dataSet);`
- Para operar con el dataset injectado se realiza `databaseTester.onSetup(DatabaseOperation);`
  - Por defecto, `onSetup()` realiza la operación `DatabaseOperation.CLEAN_INSERT`, lo cual ejecuta una operación `DELETE_ALL` (a continuación se explica) y una operación `INSERT` del dataset.

- La operación *DatabaseOperation.DELETE\_ALL* elimina todas las filas de las tablas especificadas en el dataset. Si las tablas del dataset no se encuentran en la BD o se encuentra vacía, la BD no se ve afectada.
- La operación *DatabaseOperation.REFRESH* actualiza los datos del dataset ya existentes e inserta los nuevos. Los datos ya existentes no se ven afectados.
- La operación *DatabaseOperation.NONE* no hace nada.
- **IDatabaseConnection:** representa una conexión con una BD (un wrapper de una conexión JDBC). Las operaciones disponibles son:
  - Obtener los datos de la BD en un dataset con *IDataSet databaseDataSet = connection.createDataSet();*
  - Obtener una tabla de la BD con *ITable databaseTable = connection.createTable("nombreTabla");*
  - Contar las filas de una tabla con *int count = connection.getRowCount("nombreTabla");*
- **Assertion:** (*org.dbunit.Assertion.assertEquals*) compara objetos *IDataSet* o *ITable* con *Assertion.assertEquals(ITable, ITable);*

Para poder usar DbUnits, requeriremos de algunas **dependencias:**

- **jUnit para los tests:**

```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-api</artifactId>
  <version>5.8.2</version>
  <scope>test</scope>
</dependency>
```

- Para evitar los **warnings de DbUnit:**

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-simple</artifactId>
  <version>1.7.36</version>
</dependency>
```

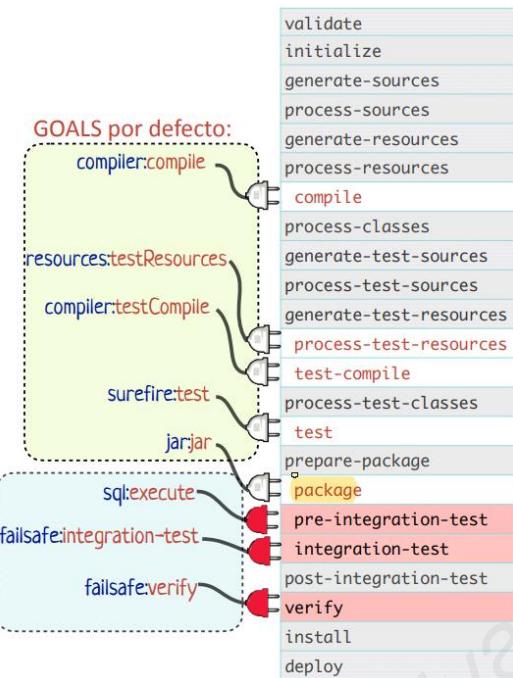
- La librería **DbUnit:**

```
<dependency>
  <groupId>org.dbunit</groupId>
  <artifactId>dbunit</artifactId>
  <version>2.7.2</version>
  <scope>test</scope>
</dependency>
```

- La librería para poder **acceder a la BD MySql:**

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.27</version>
  <scope>test</scope>
</dependency>
```

Además, vamos a tener que configurar la construcción del proyecto para incluir unos **plugins**:



- **Maven compiler plugin** para compilar las unidades (compiler:compile en fase compile) y los tests unitarios (compiler:testCompiler en fase test-compile):

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.9.0</version>
</plugin>
```

- **Surefire plugin** para la ejecución de los tests unitarios (surefire:test en la fase test):

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>3.0.0-M5</version>
</plugin>
```

- **Failsafe plugin** para ejecutar los tests de integración failsafe:integration-test ejecuta los tests de integración en fase integration-test y failsafe:verify detiene la ejecución si falla un test de integración en fase verify:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-failsafe-plugin</artifactId>
    <version>3.0.0-M5</version>
    <executions>
        <execution>
            <goals>
                <!-- Por defecto la goal failsafe:integration-test
                está asociada a la fase integration test,
                ejecuta los @Test en clases IT*.java, *IT.java o *ITCase.java-->
                <goal>integration-test</goal>
                <!-- Por defecto la goal failsafe:verify
                está asociada a la fase verify,
                detiene la ejecución si un test de integración falla -->
                <goal>verify</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

- **La goal sql:execute** para inicializar las tablas de la BD antes de ejecutar los tests de integración (fase pre-integration-test):

```

<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>sql-maven-plugin</artifactId>
    <version>1.5</version>
    <dependencies>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>8.0.27</version>
        </dependency>
    </dependencies>
    <configuration>
        <driver>com.mysql.cj.jdbc.Driver</driver>
        <url>jdbc:mysql://localhost:3306/?useSSL=false</url>
        <username>root</username>
        <password>ppss</password>
    </configuration>
    <executions>
        <!-- Ejecuta la sentencia sql en la fase pre-integration-test -->
        <execution>
            <id>create-customer-table</id>
            <phase>pre-integration-test</phase>
            <goals>
                <goal>execute</goal>
            </goals>
            <configuration>
                <srcFiles>
                    <srcFile> src/test/resources/sql/create_table_customer.sql
                    </srcFile>
                </srcFiles>
            </configuration>
        </execution>
    </executions>
</plugin>

```

## Resumen de test de integración:

- **INICIALIZACION DEL CONTROLADOR DE LA BD:**
  - IDatabaseTester databaseTester = new JdbcDatabaseTester(driverBBDD, cadenaConexionDB, "root", "ppss");
  - IDatabaseConnection connection = databaseTester.getConnection();
- **Preparar datos entrada**
- **LECTURA DEL DATASET INICIAL PARA LA BD E INSERCIÓN DEL DATASET**
  - IDataset dataInit = new FlatXmlDataFileLoader().load("/tabla2.xml");
  - databaseTester.setDataSet(dataInit);
  - databaseTester.onSetup();
- **EJECUTAR SUT**
  - Assertions.assertDoesNotThrow(() -> SUT(**datos entrada**));
  - 
  - SUTEException exception = Assertions.assertThrows(SUTEException.class, () -> SUT(**datos entrada**));
 ) ;
- **LECTURA DEL ESTADO RESULTANTE DE LA BD TRAS EJECUTAR EL SUT**
  - IDataset actualDataset = connection.createDataSet();
  - ITable actualTable = actualDataset.getTable("alumnos");
- **LECTURA DEL ESTADO ESPERADO DE LA BD**
  - IDataset expectedDataset = new FlatXmlDataFileLoader().load("/tabla3.xml");
  - ITable expectedTable = expectedDataset.getTable("alumnos");
- **RESULTADO DEL TEST**
  - Assertion.assertEquals(expectedTable, actualTable);
 ○
  - Assertions.assertAll(
 () -> Assertions.assertEquals(msgEsperado, exception.getMessage()),
 () -> Assertion.assertEquals(expectedTable, actualTable)
 );

La implementación completa de un tests con DbUnit es:

```
private IDatabaseTester databaseTester;
private IDatabaseConnection connection;

@BeforeEach
public void setUp() throws Exception {

    //Cadena de conexión: ruta:puerta/nombreBBDD?PARAMS
    String cadenaConexionDB =
"jdbc:mysql://localhost:3306/matriculacion?useSSL=false";
    String driverBBDD = "com.mysql.cj.jdbc.Driver";
    /*
    Necesitas para poder usar la BBDD:
        - clase que implementa el driver (mysql-connector-java)
        - cadena de conexión
    (JavaDataBaseConection:mysql://ruta:puerta/nombreBBDD?PARAMS)
        - username y password del usuario para acceder a la BBDD
    */
    databaseTester = new MiJdbcDatabaseTester(driverBBDD,
        cadenaConexionDB, "root", "ppss");

    //obtenemos la conexión con la BD (para luego crear el DataSet de la BBDD)
    connection = databaseTester.getConnection();
}

@Test
public void testA1() throws Exception {
    //Primero: conectar con la BBDD (IDatabaseTester y IDatabaseConnection)

    //Preparar SUT
    //Datos de Entrada
    AlumnoTO alumno = new AlumnoTO();
    alumno.setNif("33333333C");
    alumno.setNombre("Elena Aguirre Juarez");
    //Si no debe ser null, si no vacio, indicamos string vacio
    alumno.setDireccion("");
    alumno.setEmail("");
    alumno.setFechaNacimiento(LocalDate.of(1982, 02, 22));

    //Preparar BBDD -> leer ficheros e introducirlos
    IDataset dataInit = new FlatXmlDataFileLoader().load("/tabla2.xml");
    //Inyectamos el dataset en el objeto databaseTester e inicializamos BBDD
    databaseTester.setDataSet(dataInit);
    databaseTester.onSetup(DatabaseOperation.CLEAN_INSERT);

    //SUT
    Assertions.assertDoesNotThrow(() -> alumnoDAO.addAlumno(alumno));

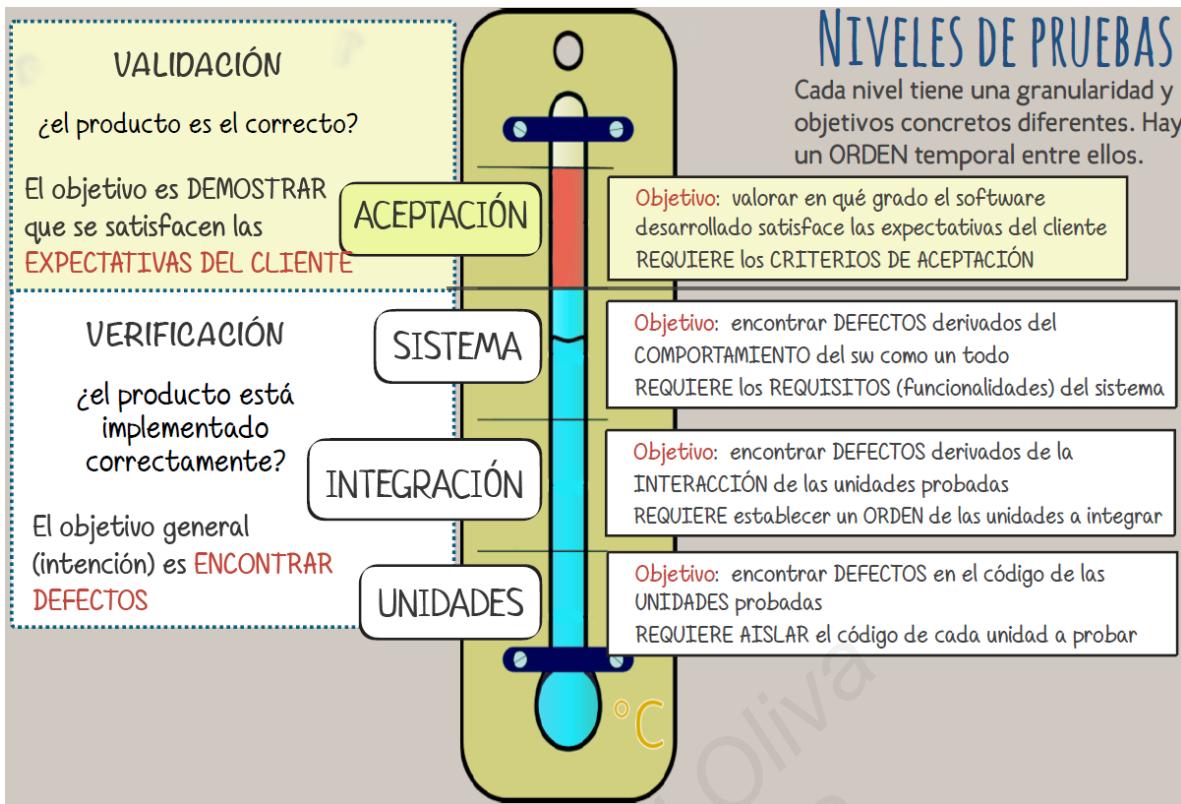
    //Recuperar cambios en BBDD
    IDataset databaseDataSet = connection.createDataSet();
    //ITable actualTable = connection.createTable("alumnos");

    //Recuperamos los datos de la tabla alumnos
    ITable actualTable = databaseDataSet.getTable("alumnos");

    //Lectura de fichero de BBDD esperado
    IDataset dataEsperado = new FlatXmlDataFileLoader().load("/tabla3.xml");
    ITable expectedTable = dataEsperado.getTable("alumnos");

    //Comparar
    Assertion.assertEquals(expectedTable, actualTable);
}
```

## Pruebas del sistema y aceptación



Componente = UNIDAD o conjunto de UNIDADES.

## Pruebas del sistema

Objetivo: encontrar defectos en las funcionalidades (desde el punto de vista del desarrollador), defectos derivados del SW como un todo. Se trata de la última fase del proceso de VERIFICACIÓN (¿el producto está implementado correctamente?).

- Son pruebas dinámicas y obtienen los casos de prueba a partir de las especificaciones del sistema (caja negra). Usa todos los componentes del sistema, se prueba si los componentes son compatibles, si interaccionan correctamente y transfieren el dato adecuado en el tiempo adecuado a través de sus interfaces.
- Se diferencian de las pruebas de integración por:
  - Se incluyen componentes reutilizables desarrollados por “terceros” (APIs externos, por ej.).
  - Los comportamientos probados son los especificados para el sistema completo (por ej., una compra completa online).

### Diseño:

- Diseño basado en casos de uso:
  - Permite probar interacciones entre componentes mostrados en los diagramas de secuencias. La prueba hace que se lleven a cabo diferentes interacciones entre los componentes implicados.
  - Los diagramas de secuencia ayudan a diseñar los casos de prueba adecuados ya que muestran qué datos de entrada y salidas deben producirse entre los componentes.
  - Puesto que no podemos realizar pruebas exhaustivas, debemos seleccionar algunos casos de uso concretos según algunas políticas de prueba:
    - Todas las funciones del sistema que sean accedidas desde menús deben ser probadas.
    - Desde un menú, probar todas las combinaciones de funciones.
    - Si se dan entradas de usuario, se deben probar correctas e incorrectas.
    - Las funciones más utilizadas en uso normal, deben ser las más probadas.

- Diseño de transición de estados:

- Se utiliza un sistema con estados (por ej., un sistema de pedidos con estados de pedido). Un estado viene dado por un conjunto de valores del sistema y puede ir cambiando.
- A partir de la especificación se obtiene un grafo que representa los estados en los nodos y la transición entre los estados con las aristas.
- A partir del grafo, se selecciona un conjunto de caminos mínimos para conseguir un objetivo (por ej., recorrer todas las transiciones del estado [McCabe's]).
- Finalmente se determina los datos de entrada y salida para recorrer esos caminos.

## Pruebas de aceptación (Acceptance testing)

Objetivo: validar las expectativas del cliente. Valorar en qué grado el SW satisface las expectativas del cliente. Requiere criterios de aceptación. Forma parte de la fase de VALIDACIÓN (¿el producto es correcto?).

- Tras las pruebas del sistema, el producto está preparado para ser entregado al cliente, lo último que falta es que cumpla con sus expectativas. Los clientes ejecutan tests de aceptación basándose en sus expectativas. Si el producto supera estas pruebas, el *sistema pasa a producción*.
- Son pruebas orientadas a determinar si el sistema satisface los criterios de aceptación, normalmente firmadas al inicio del proyecto (los criterios de aceptación se definen en etapas tempranas, pero los probamos al final y después del proceso de verificación).
- Hay dos categorías de pruebas de aceptación:
  - User acceptance testing (UAT): dirigidas por el cliente para asegurar que satisface los criterios de aceptación contractuales. Pruebas α [en el lugar de desarrollo y por usuarios conocidos] y pruebas β [puede ser abierta o reservada a usuarios anónimos seleccionados].
  - Business acceptance testing (BAT): dirigidas por la organización a modo de ensayo de las UAT para asegurar que las pasarán.

Criterios de aceptación: deben ser definidos y acordados entre el proveedor y el cliente.

- Cuestión: ¿Qué criterios debe satisfacer el sistema para ser aceptados por el cliente?
  - El principio básico es asegurar que la calidad del sistema es aceptable.
  - Los criterios de aceptación deben ser medibles y cuantificables (con porcentajes, por ejemplo, no sirve decir "mucho", "muy", "poco", etc).
  - Ejemplos de atributos de calidad:
    - Corrección funcional y completitud: ¿hace lo que se quiere que haga y están todas las características presentes?
    - Exactitud, integridad de datos, rendimiento, fiabilidad y disponibilidad, robustez...

Propiedades emergentes: cualquier atributo incluido en los criterios de aceptación es una propiedad emergente.

- Son propiedades/características que sólo cobran sentido cuando encontramos el sistema completo ya que no puede calcularse a partir de las propiedades de sus componentes individuales.
- Para identificar si un atributo es una propiedad emergente debes preguntarte si tiene sentido aplicarlo a una pieza o sólo al programa completo.
- Dos tipos de propiedades emergente:
  - Funcionales: muestran el propósito del sistema después de integrar sus componentes. Requiere que esté todo el sistema funcionando para poder abordar la funcionalidad.
  - No funcionales: relacionadas con el comportamiento en un entorno habitual de producción.

Diseño: Un grupo que no esté implicado en el proceso de desarrollo debería ser quien realice las pruebas de aceptación para no tener consideraciones técnicas. Se trata de determinar si el sistema es lo suficientemente bueno: cumple los criterios de aceptación. Normalmente se trata de un proceso black-box, se centra en la “funcionalidad” y no en la implementación. Los métodos de diseño se diseñan mediante caja negra.

- Método basado en requerimientos: pruebas de validación, cada requerimiento debe ser “testable”.
  - Un requerimiento implica una condición, una restricción, un requisito o algo necesario. Deben estar especificados de manera que se pueda diseñar una prueba a partir de él (y probar que está presente, si no podemos demostrar que el requerimiento está presente en el sistema, no sirve de nada).
  - Un requerimiento es:
    - Condición o capacidad necesitado por un usuario para resolver un problema.
    - Condición o capacidad que necesita el sistema para poder satisfacer una determinada especificación.
  - Dado un requerimiento, mediante particiones equivalentes podemos definir varios tests para “cubrir” todo el requerimiento.
- Método basado en escenarios: escenarios que describen la forma en la que el sistema debería usarse.
  - Un escenario es una descripción de un ejemplo de interacción del usuario con el sistema. El escenario debe incluir *asunciones iniciales*, una *descripción del flujo normal de eventos* y de *situaciones excepcionales*, y una *descripción del estado final* del sistema.
  - A partir de un escenario, podemos extraer múltiples requerimiento, y por ende, múltiples tests. Debemos realizar un listado de los requerimientos representados en el escenario.
  - En un escenario se prueba tanto los requerimientos individuales como la combinación de varios de ellos.
  - El tester debe tomar el rol del escenario y probar tanto situaciones erróneas como correctas.
  - No podemos realizar una prueba de todos los escenarios posibles, no podemos hacer pruebas exhaustivas ya que el tiempo es un recurso limitado. Debemos seleccionar los escenarios más adecuados.

## Pruebas de propiedades emergentes funcionales: Selenium

- Las pruebas de propiedades emergentes funcionales tienen como objetivo comprobar que el sistema ofrece la funcionalidad esperada.
  - Se diseñan con técnicas de caja negra y prueban la funcionalidad del sistema a través de la interfaz de usuario.
    - Diseño de pruebas basado en requerimientos.
    - Diseño de pruebas basado en escenarios.
- Selenium es un conjunto de herramientas de pruebas open-source para automatizar pruebas de propiedades emergentes funcionales sobre aplicaciones Web (navegadores):
  - Selenium WebDriver (API): tests robustos, pueden ser escalables y distribuirse en diferentes entornos.
  - Selenium IDE (extensión de navegador): permite crear scripts de pruebas utilizando la aplicación web tal y como un usuario haría.
  - Selenium Grid: servidor proxy que permite ejecutar tests en paralelo usando múltiples máquinas y diferentes navegadores.

## Selenium IDE

Ventajas: rápida implementación de tests, no requiere experiencia con lenguajes programación, la búsqueda de elementos en la página es muy fácil y rápida.

Inconvenientes: tests inflexibles (no pueden variarse), duplicación de código, no soporta gestión de errores y se pueden integrar en el proceso de construcción.

Características:

- Botón de grabación, editor de scripts, el driver se implementa como un script de comandos selenese.
- El proyecto se puede guardar en un fichero con extensión .side en formato Json. De cada comando se almacena: id, comentario, comando, target, target optionales, value.
- Un comando Selenece se compone de: target (referencia a elemento HTML), value (texto, patrón o variable).

4 tipos de comandos:

- **Actions**: interactúan con elementos de la página:
  - open -> target: URL. Abre una pestaña con ese URL. (*open http://localhost:8080/jpetstore*)
  - click -> target: locator. Hace click. (*click linkText=Enter the Store*)
  - type -> target: locator; value: texto. Escribe el texto en el locator. (*type name=username z*)
- **Accessors**: almacenan valores en variables:
  - store -> target: texto a guardar; value: variable. (*store hola variable*)
  - store (text, xpath count) -> target: locator; value: variable. Guarda el texto /el tamaño del array(conteo) del locator. (*store text xpath=/div[2]/label[1] variable*)
  - execute script -> target: script en JavaScript con return [return "a"; / return \${variable} – 1;]; value: nombre de la variable para return [variable, conteo, resultado...]. (*execute script return"a"; variable*)
- **Assertions**: verifican si se cumple la condición:
  - verify (text, title, element present) -> target: locator; (text o title) value: texto [User\* o User Name]. Registra el fallo y continúa. (*verify title JPetStore Demo / verify text name=user Usuario*)
  - assert (text, title, element present) -> target: locator; (text o title) value: texto. Detiene la ejecución si falla. (*assert title JPetStore Demo / assert text name=user Usuario*)
  - assert -> target: variable; value: valor. (*assert variable 3*)
  - waitFor (element present/visible) -> target: locator; value: tiempo en ms a esperar antes de fallar y detener la ejecución. (*wait for element present name=applyButton 1000*)
- **Control flow**: alteran el flujo secuencial de los comandos (las variables vienen de store o execute script):
  - Condicionales:
    - if -> target: condición [\${variable} == 2]. (*if \${variable}==2 .... else .... end*)
    - else if -> target: condición.
    - else.
    - end: indica el fin de la condición o del else.
  - Bucles:
    - times -> target: número de veces a repetir; end. (*times 5 .... end*)
    - do, repeat if -> target: condición (\${valor}< 4). (*do... repeat if \${valor}< 4*)
    - while -> target: condición; end. (*while \${valor}< 4 ... end*)

**Locator** [locatorType=location] (campo target):

- linkText: texto escrito (en <a>). (*linkText=Return to FISH*)
- name: campo name. (*name = username*)
- id: campo id. (*id = 143*)
- xpath: dirección xpath (relativa o absoluta). (*xpath=[/html/body/form[1]/input[2] | css=a:nth-child(3)>img)*)

## Selenium WebDriver

Permite evitar las limitaciones de Selenium IDE pudiendo usarse en varios lenguajes de programación, como Java.

## Inicialización del WebDriver + definición del headless

```
//Definimos si debe ser headless o no
ChromeOptions chromeOptions = new ChromeOptions();
boolean headless = Boolean.parseBoolean(System.getProperty("chromeHeadless"));
chromeOptions.setHeadless(headless);

//Inicializamos driver
WebDriver driver = new ChromeDriver(chromeOptions);
driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
```

## Inicialización del WebDriver sin headless (+ obtención de title)

```
//Inicializamos driver
WebDriver driver = new ChromeDriver();
driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
String title = driver.getTitle();
```

## Localización de objetos

```
//Buscamos un searchBox con name=="q"  
WebElement searchBox = driver.findElement(By.name("q"));
```

## Formas de localizar un elemento:

- By.name(): atributo name.
  - By.id(): atributo id.
  - By.tagName(): nombre del tag/tipo de elemento (input, form, label, etc).
  - By.className(): atributo class.
  - By.linkText(): texto de una etiqueta <a>.
  - By.partialLinkText(): texto parcial de una etiqueta <a>.
  - By.xpath(): dirección xpath (absoluta o relativa).
  - By.cssSelector(): patrones para seleccionar un elemento, los cuales son:
    - css=tag#id                    (css="form#loginForm" para <form id="loginForm"> </form>)
    - css=tag.class                (css="input.passfield" para <input class = "required passfield"/>)
    - css=tag.class[attribute=value]  
                                  (css="input.required[type='text']" para <input class="required" type="text"/>)
    - css=tag[attribute=value]    (css="input[name='username']" para <input name="username"/>)
    - css=tag:contains("inner text") (css="font:contains('Password:')"c para <font>Password:</font>)

## Acciones sobre WebElements

- `click(): driver.findElement(By.xpath("//div/ul/li[6]/a")).click();` (link)  
`driver.findElement(By.cssSelector("input[value='Business']")).click();` (radioBox)  
`driver.findElement(By.linkText("Register here")).click();` (link)
  - `sendKeys(caracteres):`  
`driver.findElement(By.cssSelector("input[name=middlename]")).sendKeys(usuario);`
  - `clear():` sobre un elemento textbox, limpia el texto.
  - `submit():` sobre un elemento form, envía el form.
  - `getText(); isDisplayed(); isEnabled(); isSelected();`
  - Seleccionar en dropBox (el menú accesories, al apuntarle se despliega, queremos seleccionar shoes):

```
Actions pointAccesories = new Actions(driver);
Action pointAccesories.moveToElement(accesories).moveToElement(shoes).click();
pointAccesories.perform();
```

## Tiempos de espera

- Implícito:

```
driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
```

- Explícito:

```
WebDriverWait wait = new WebDriverWait(driver, 10);
wait.until(ExpectedConditions.titleContains("selenium"));
```

## Múltiples acciones

Usamos la clase Actions combinado con la clase Action con los WebElements:

```
// Add all the actions into the Actions builder
Actions builder = new Actions(driver);
// Generate the composite action
Action compositeAction = builder.keyDown(Keys.CONTROL)
    .click(driver.findElement(By.id("button1")))
    .click(driver.findElement(By.id("button2")))
    .click(driver.findElement(By.id("button3")))
    .keyUp(Keys.CONTROL)
    .build();
// Perform the composite action.
compositeAction.perform();
```

## Acciones con ratón (y Actions)

click(WebElement), moveToElement(WebElement), doubleClick() y contextClick() [click derecho].

```
new Actions(driver)
    .moveToElement(driver.findElement(By.id("button1"))).doubleClick()
    .moveToElement(driver.findElement(By.id("button2"))).click()
    .build().perform();
```

## Acciones con teclado (y Actions)

keyDown(Keys), keyUp(Keys) y sendKeys(String).

```
new Actions(driver)
    .keyDown(Keys.CONTROL)
    .click(driver.findElement(By.id("input1")))
    .click(driver.findElement(By.id("input2")))
    .sendKeys("HOLA TODOS")
    .keyUp(Keys.CONTROL)
    .build().perform();
```

## Operaciones de navegación

Driver.switchTo.window(windowHandle). (Obtiene handle de la primera ventana, abre una segunda, obtiene su handle [con ambos handles podremos cambiar entre ambas ventanas]. Cierra la ventana actual y cambia a la otra)

```
String window1 = driver.getWindowHandle();
driver.findElement(By.linkText("Google Search")).click();
//Obtengo los handleIds que tengo actualmente
Set<String> setIds = driver.getWindowHandles();
//Handle de la nueva pestaña
String[] handleIds = setIds.toArray(new String[setIds.size()]);
String window2 = handleIds[1];
driver.switchTo().window(window2);
driver.close();
driver.switchTo().window(window1);
```

## ¿Dónde guardar el page object?

Podemos hacerlo todo en src/test/java, con lo cual failsafe ejecutará realizará la búsqueda de los patrones del HTML. Los tests no serán flexibles (si cambia el diseño de la página, habrá que volver a cambiar todos los tests).

Podemos guardar en src/main/java las clases del patrón de diseño pago object (POM, Page Object Model), dejando en los tests sólo el código de test:

- Esto facilita la mantenibilidad y reduce la duplicación de código en los tests.
- La idea es independizar los tests de las páginas HTML.
- Básicamente, para cada página tendremos una clase en el cual los atributos serán los elementos (WebElements) y los métodos serán los SERVICIOS de la página (registrarnos, cambiar a otra página, insertar un elemento a la cesta, etc).
- El API de WebDriver proporciona elementos para implementar este patrón (POM):
  - Anotación `@FindBy` para injectar los WebElements de una página web. Para que funcione, debemos usar el PageFactory.
  - Clase `PageFactory` para obtener las clases que representan las páginas HTML.

### Ejemplo de POM sin PageFactory

**Página inicial.** Se encarga de abrir el URL y de hacer el enlace a la página de login.

```
public class HomePage {  
    WebDriver driver;  
    WebElement account;  
    WebElement logIn;  
    String pTitle;  
  
    public HomePage(WebDriver driver) {  
        this.driver = driver;  
        this.driver.get("http://demo-store.seleniumacademy.com/");  
        account = driver.findElement(By.xpath("//*[@id='header']//div/div[2]/div/a"));  
        pTitle = driver.getTitle();  
    }  
  
    public CustomerLoginPage accessLoginPage() {  
        account.click();  
  
        logIn = driver.findElement(By.xpath("//*[@id='header-account']/div/ul/li[6]/a"));  
        logIn.click();  
  
        return new CustomerLoginPage(driver);  
    }  
  
    public String getPageTitle() {  
        return pTitle;  
    }  
}
```

**Página de login.** Tiene las direcciones de los elementos preparados, en el constructor inicializa los WebElements posibles. Tras hacer login, se nos redirige a la página de MyAccount.

```
public class CustomerLoginPage {
    WebDriver driver;
    WebElement email;
    WebElement password;
    WebElement logIn;
    WebElement errorMsg;
    String pTitle;

    String cssEmail = "input[name=\"login[username]\"]";
    String cssPassword = "input[name=\"login[password]\"]";
    String xpathLoginButton = "//*[@id=\"send2\"]";
    String xpathMsgErr =
"//*[@id=\"top\"]/body/div/div[2]/div/div/div[2]/ul/li/ul/li/span";

    public CustomerLoginPage(WebDriver driver) {
        this.driver = driver;
        email = driver.findElement(By.cssSelector(cssEmail));
        password = driver.findElement(By.cssSelector(cssPassword));
        logIn = driver.findElement(By.xpath(xpathLoginButton));
        pTitle = driver.getTitle();
    }

    public MyAccountPage loginOK(String user, String password) {
        email.sendKeys(user);
        this.password.sendKeys(password);
        logIn.click();
        return new MyAccountPage(driver);
    }

    public String loginFail(String user, String password) {
        email.sendKeys(user);
        this.password.sendKeys(password);
        logIn.click();
        errorMsg = driver.findElement(By.xpath(xpathMsgErr));
        return errorMsg.getText();
    }

    public String getPageTitle() {
        return this.pTitle;
    }
}
```

**Página MyAccount.** Una simple página que no interactúa con ningún elemento.

```
public class MyAccountPage {
    WebDriver driver;
    String pTitle;

    public MyAccountPage(WebDriver driver) {
        this.driver = driver;
        pTitle = driver.getTitle();
    }

    public String getPageTitle() {
        return this.pTitle;
    }
}
```

## Ejemplo de POM CON PageFactory

**Página principal**, abre el URL de la página. Los atributos anotados con @FindBy se crearán cuando estén presentes en la página. Puede redirigir a la página de shoes.

```
public class MyAccountPage {  
    private WebDriver driver;  
  
    private @FindBy(xpath = "//*[@id='nav']/ol/li[3]/a")  
    WebElement accesories;  
  
    private @FindBy(xpath = "//*[@id='nav']/ol/li[3]/ul/li[4]/a")  
    WebElement shoes;  
  
    private String pTitle;  
  
    public MyAccountPage(WebDriver driver){  
        this.driver = driver;  
        this.driver.get("http://demo-store.seleniumacademy.com/customer/account/");  
  
        this.pTitle = driver.getTitle();  
    }  
  
    public ShoesPage goToShoesPage(){  
        //Creo la secuencia de acciones para ir al ShoesPage (Apunto al menu  
        Accessories y click a shoes)  
  
        Actions pointAccesories = new Actions(driver);  
        pointAccesories.moveToElement(accesories).moveToElement(shoes).click();  
        pointAccesories.perform();  
  
        return PageFactory.initElements(driver, ShoesPage.class);  
    }  
  
    public String getPageTitle(){  
        return this.pTitle;  
    }  
}
```

**Página shoes**. Tiene métodos para seleccionar zapatos a comparar, y cuando hace click en comparar, se abre otra pestaña, con lo cual cambia el driver a la nueva página.

```
public class ShoesPage {  
    private WebDriver driver;  
  
    private WebElement shoe;  
  
    private @FindBy(css = "button[title='Compare']")  
    WebElement compareButton;  
    private @FindBy(partialLinkText = "Clear All")  
    WebElement clearAllButton;  
    private @FindBy(xpath = "//*[@id='top']/body/div/div[2]/li/span")  
    WebElement clearedListMsg;  
  
    private String pTitle;  
  
    public ShoesPage(WebDriver driver){  
        this.driver = driver;  
        this.pTitle = driver.getTitle();  
    }
```

```

public void selectShoeToCompare(int number) {
    //Busco el boton Add to Compare del elemento number como primer zapato
    shoe =
driver.findElement(By.xpath("//*[@id=\"top\"]/body/div/ul/li["+number+"]/a"));
    //Hago click en Add
    JavascriptExecutor jse = (JavascriptExecutor) driver;
    jse.executeScript("arguments[0].scrollIntoView()", shoe);
    shoe.click();
}

public ProductComparisonPage compareShoes() {
    //Handle de la pestaña actual (ShopPage)
    String handleIdShoes = driver.getWindowHandle();

    //Click boton compare (con scroll)
    JavascriptExecutor jse = (JavascriptExecutor) driver;
    jse.executeScript("arguments[0].scrollIntoView()", compareButton);
    compareButton.click();

    //Obtengo los handleIDs que tengo actualmente
    Set<String> setIds = driver.getWindowHandles();

    //Handle de la nueva pestaña (Comparison Page)
    String[] handleIds = setIds.toArray(new String[setIds.size()]);
    String handleIdCompare = handleIds[1];

    driver.switchTo().window(handleIdCompare);

    ProductComparisonPage comparisonPage = PageFactory.initElements(driver,
ProductComparisonPage.class);
    //El handle ID de compare será su "myHandle"
    comparisonPage.setMyHandleID(handleIdCompare);
    //El handle ID de shoesPage será su "myHandleFrom"
    comparisonPage.setMyHandleIDFrom(handleIdShoes);

    return comparisonPage;
}

public String clearShoesList(){
    //Hago scroll hasta el boton de clear
    JavascriptExecutor jse = (JavascriptExecutor) driver;
    jse.executeScript("arguments[0].scrollIntoView()", clearAllButton);
    clearAllButton.click();

    //Aceptamos la alerta guardando el mensaje
    Alert alerta = driver.switchTo().alert();
    String msg = alerta.getText();
    alerta.accept();
    return msg;
}

public String getPageTitle(){ return this.pTitle; }

public String getClearedMessage(){ return clearedListMsg.getText(); }
}

```

**Página de comparación.** Tiene la operación de cerrar la ventana, devolviendo el handle a la ventana original.

```
public class ProductComparisonPage {
    private WebDriver driver;

    private @FindBy(xpath = "//*[@id=\"top\"]/body/div/div[3]/button")
        WebElement closeButton;
    private String myHandleId;
    private String myHandleIdFROM;
    private String pTitle;

    public ProductComparisonPage(WebDriver driver) {
        this.driver = driver;
        pTitle = driver.getTitle();
    }

    public void setMyHandleID(String myHandleId) {
        this.myHandleId = myHandleId;
    }

    public void setMyHandleIDFrom(String myHandleIdFROM) {
        this.myHandleIdFROM = myHandleIdFROM;
    }

    public ShoesPage close() {
        //Cierro la pestaña dando click al boton de cerrar
        //closeButton.click();
        driver.close();
        //Vuelvo el driver al handleId de la primera pestaña, anteriormente almacenada
        driver.switchTo().window(myHandleIdFROM);

        return PageFactory.initElements(driver, ShoesPage.class);
    }

    public String getPageTitle(){
        return this.pTitle;
    }
}
```

**El test.** Se van abriendo las páginas conforme se obtienen los objetos que lo controlan. Mediante los títulos se realiza un seguimiento.

```
@Test
public void compareShoes() {
    // (1) Comprobamos My Account
    MyAccountPage accountPage = PageFactory.initElements(driver, MyAccountPage.class);
    assertEquals("My Account", accountPage.getPageTitle());

    // (2) Seleccionamos Accessories->Shoes
    ShoesPage shoesPage = accountPage.goToShoesPage();

    // (3) Comprobamos Shoes-Accesories
    assertEquals("Shoes - Accessories", shoesPage.getPageTitle());

    // (4) Seleccionamos 2 zapatos
    shoesPage.selectFirstShoeToCompare(5);
    shoesPage.selectSecondShoeToCompare(6);

    // (5) Click en compare
    ProductComparisonPage productComparisonPage = shoesPage.compareShoes();

    // (6) Comprobamos Products Comparison List
    assertTrue(productComparisonPage.getPageTitle().contains("Products Comparison List"));

    // (7) Cerramos la ventana productos
    shoesPage = productComparisonPage.close();

    // (8) Comprobamos Shoes-Accesories
    assertEquals("Shoes - Accessories", shoesPage.getPageTitle());

    // (9) Limpiamos la lista
    String msgAlerta = shoesPage.clearShoesList();
    assertTrue(msgAlerta.toLowerCase().contains("are you sure you would like to remove all products from your comparison?"));

    // (10) Verificamos el mensaje The comparison list was cleared
    assertEquals("The comparison list was cleared.", shoesPage.getClearedMessage());
}
```

Uso de Cookies. Mediante la clase Cookies, podemos guardar y recuperar las sesiones de un usuario.

- Cookies.storeCookiesToFile(usernameLogin, password); usamos este método para iniciar sesión en una cuenta y almacenar su sesión en una cookie.
- Cookies.loadCookiesFromFile(driver); recupera la sesión almacenada en una Cookie.

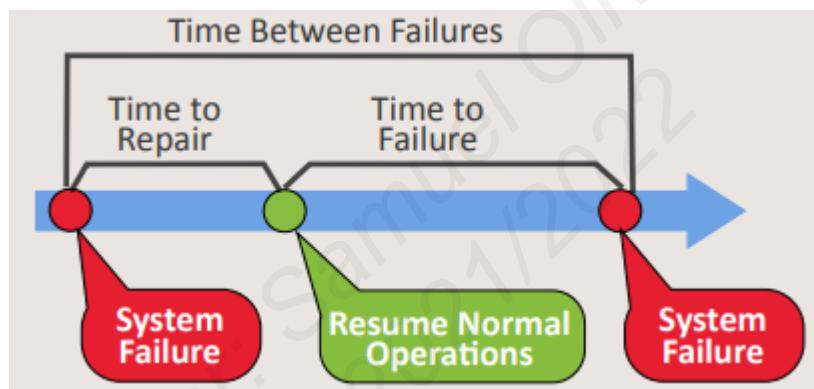
## Pruebas de propiedades emergentes NO funcionales

Las propiedades emergentes no funcionales hacen referencia al cómo debe comportarse en sistema realizando sus funciones. Muchos se categorizan como “-ilidades”:

- Fiabilidad: probabilidad de que no falle durante un tiempo determinado (fiabilidad del 90%).
- Disponibilidad: tiempo durante el cual está disponible el sistema (24/7, 24h de 7d a la semana).
- Mantenibilidad: capacidad de soportar cambios, los cuales pueden ser:
  - Correctivos: provocado por errores en la aplicación.
  - Adaptativos: provocado por cambios de hardware/software sobre el que se ejecuta la aplicación.
  - Perfectivos: añadir/modificar funcionalidades para ampliar/mejorar la aplicación.
- Escalabilidad: capacidad de mantener el tiempo de respuesta ante cambios en usuarios activos.
- Robustez: capacidad de seguir funcionando pese a fallos (que sea capaz de recuperarse).

Las propiedades emergentes deben ser cuantificables: se utilizan **métricas** para evaluar el nivel de las propiedades emergentes no funcionales:

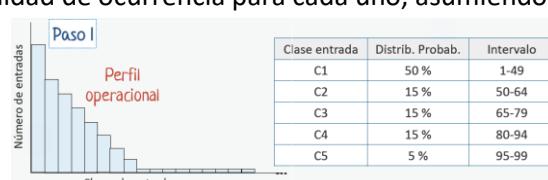
- **Fiabilidad**: pruebas aleatorias basándonos en un perfil operacional (perfil con probabilidades reales de cada operación). Se usan métricas MTTF (Mean Time To Failure), MTTR (Mean Time To Repair) y MTBF=MTTF+MTTR (Mean Time Between Failures).



- **Disponibilidad**: MTTR para medir el “downtime” del sistema. La idea es minimizar MTTR.
- **Mantenibilidad**: MTTR (tiempo consumido en analizar un defecto correctivo, diseñar la modificación, implementar el cambio, probarlo y distribuirlo).
- **Escalabilidad**: número de transacciones por unidad de tiempo (10000 transacciones/segundo). Se puede incrementar la escalabilidad siempre y que no pasemos el límite de almacenamiento de datos, ancho de banda o velocidad del procesador.

Ejemplos de pruebas:

1. **Pruebas de carga**: validan el rendimiento en términos de tratar un número específico de usuarios manteniendo un ratio de transacciones (debe tardar 2s por transacción cuando hay 10000 usuarios).
2. **Pruebas de stress**: forzar peticiones por encima del límite y hasta cuanto aguanta y la capacidad de recuperarse. Evalúa la fiabilidad (prob de no fallar) y la robustez (capacidad de recuperarse) cuando se supera la carga normal.
3. **Pruebas estadísticas**: evalúa la fiabilidad en un entorno real. Consiste en:
  1. **Construir un perfil operacional**: refleja el uso real del sistema (patrón de entradas), e identifica las clases y la probabilidad de ocurrencia para cada uno, asumiendo un uso normal.



## 2. Genera conjunto aleatorio de datos de prueba que reflejen el perfil operacional.

Paso 2

Se generan números aleatorios entre 1 y 99, por ejemplo:

13-94-22-24-45-56-81-19-31-69-45-9-38-21-52-84-86-97-...

Se derivan casos de prueba según su distribución de probabilidad:

C1-C4-C1-C1-C1-C2-C4-C1-C1-C3-C1-C1-C1-C2-C4-C4-C5-...

## 3. Ejecuta las pruebas: mide el número de fallos y tiempo entre fallos, calculando la fiabilidad.

Paso 3

... a continuación deberíamos ejecutar las pruebas midiendo el número de fallos y el tiempo entre fallos

Las propiedades emergentes no funcionales influyen en el rendimiento del sistema. Para evaluar el rendimiento necesitamos las siguientes actividades:

- **Identificar los criterios de aceptación:** identificar y cuantificar las propiedades emergentes no funcionales que determinan el rendimiento más aceptable.
- **Diseñar los tests:** estudiamos el patrón de uso de la aplicación (perfil operacional) para que los casos de prueba estén basado en escenarios reales.
- **Preparar el entorno de prueba:** el entorno debe ser lo más realista posible (ancho de banda, hardware, etc).
- **Automatizar las pruebas.**
- **Analizar los resultados:** obtendremos una gran colección de datos como resultado. Habrá que estudiarlos para obtener las características del programa.

Consideraciones del rendimiento:

- Posibles fallos relacionados con el rendimiento pueden provocar grandes retoques en el código:
  - Las propiedades no funcionales están condicionadas por la ARQUITECTURA del sistema.
  - La arquitectura se suele determinar en las primeras fases del desarrollo. Cambiar la arquitectura puede implicar cambiar “todo”.
  - El coste de reparar un error es proporcional al intervalo desde que se produjo hasta que se detecta.
- Minimizaremos los problemas de las propiedades emergentes mediante una buenas estrategias de pruebas combinado con una arquitectura software que considere el rendimiento desde el inicio del desarrollo:
  - Con una arquitectura iterativa de desarrollo las iteraciones iniciales son para construir prototipos exploratorios y comprobar los requisitos no funcionales.

## JMeter

Presentamos:

- Uno o más **grupos de hilos** (Thread Groups): hilos de ejecución, son independientes entre ellos, podemos definir cuantos hilos ejecutar. RAMP-UP: crea los hilos de forma gradual.
- **Controladores lógicos** (Logic Controllers): determinan el control del flujo (Simple controller [no tiene ningún efecto, sólo agrupa], Loop Controller [repite hijos X veces], Only once controller, Interleave controller [Alterna la ejecución de los hijos por cada iteración]).
- **Samplers, Listeners, Timers, Assertions...**: Operadores que realizan acciones o modifican los procesos:
  - **Samplers**: realizan peticiones [GET, POST, etc.] HTTP/FTP/HDBC... Request, se ejecutan en el orden del plan. Un sampler se puede anidar en un grupo de hilos o un controlador. Como hijo pueden tener un Assertions o un Timers.
  - **Listeners**: muestran y almacenan en disco los resultados de las peticiones realizadas. Proporcionan acceso a información que JMeter va acumulando:
    - Todos los listeners guardan los mismos datos, se diferencian en la forma en que la presentan
    - Ejemplos: Simple Data Writer, Aggregate Graph, Graph result...
  - **Timers**: Realizan pausas **antes de realizar** cada petición de cada hilo, afecta a todos los samplers del mismo nivel o inferiores. Hay diferentes tipos: Constant timer, Uniform random timer (aleatorias), Gaussian random timer (random bajo distribución gausiana). Si hay varios, se aplican todos. Para que sólo afecte a un sampler, debe ser su hijo. Para que se aplique después de un sampler: se añade como hijo al siguiente sampler o lo añadimos en un Flow Control Action Sampler (sampler que va después, y sirve para poner timers).
  - **Assertions**: comprueba las respuestas recibidas mediante el texto, el código, etc. Mira el resultado esperado. En la práctica, buscamos en el Text Response por un Substring introducido.
- **Elementos de configuración**: Trabajan junto a los samplers, los cuales modifican la configuración de estos. Tenemos HTTP Request Default para cuando tenemos samplers con la misma URL, HTTP Cookie Manager para almacenar y enviar cookies como si fuera un navegador, además de poder añadir nuevas.
- **Representación de los datos**:
  - **Result Tree/Assertion Result**: ve la ejecución de los samplers y sus aserciones.
  - **Graph Results**: muestra en un gráfico de puntos cada información de cada Request y su evolución.
  - **Aggregate Report**: muestra información de la ejecución:
    - **# Sampler**: cuantos samplers se han ejecutado.
    - **Average**: tiempo medio de respuesta (en ms).
    - **Median**: tiempo medio de respuestas, el de la mitad (el 50%).
    - **90% Line**: percentil 90%, el tiempo de respuesta situado en el puesto 90%.
    - **Min**: tiempo mínimo de respuesta para un sampler.
    - **Max**: tiempo máximo de respuesta para un sampler.
    - **% Error**: porcentaje de peticiones con errores.
    - **Throughput (Rendimiento)**: número de peticiones por segundo/min/hora. Se calcula como num peticiones/tiempo total de respuesta (tiempo de respuesta = tiempo desde que se envía la primera petición hasta que se recibe la última respuesta, incluido temporizadores).
    - **Kb/sec**: rendimiento expresado en Kilobytes por segundo.

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Received ...	Sent KB/sec
/jpetstore...	20	13	10	24	24	25	5	25	0.00%	148.1/sec	833.62	112.70
Home	20	12	9	20	23	26	5	26	0.00%	138.9/sec	205.62	53.03
Main catal...	20	8	7	13	19	19	3	19	0.00%	119.0/sec	561.76	25.34
Sign In	20	9	8	15	17	23	4	23	0.00%	101.5/sec	388.92	82.69
Login	20	49	35	93	94	119	13	119	0.00%	45.0/sec	231.47	93.92
Reptiles	20	11	9	21	24	24	6	24	0.00%	46.4/sec	168.21	11.42
Sign Out	20	151	99	371	385	700	5	700	0.00%	21.7/sec	112.21	34.94
<b>TOTAL</b>	<b>140</b>	<b>36</b>	<b>11</b>	<b>84</b>	<b>107</b>	<b>385</b>	<b>3</b>	<b>700</b>	<b>0.00%</b>	<b>11.4/sec</b>	<b>48.18</b>	<b>9.95</b>

Tenemos un Interleave Controller de Reptiles/Birds/Dogs. Como sólo se ha realizado 1 iteración, sólo se muestra Reptiles. Si hicieramos 3 iteraciones, se mostrarían Reptiles, Birds y Dogs, con 20 samples cada uno (el resto 60).

## Análisis de pruebas

Uso de métricas para el análisis de pruebas.

El proceso de pruebas: tiene 4 fases:

- Planificación y control de las pruebas: definir los objetivos de las pruebas y establecer el qué, cómo y cuándo vamos a realizarlas.
- Diseño de las pruebas: decidir con qué datos vamos a probar el código. En cada nivel el proceso de diseño cambia.
- Implementación y ejecución de las pruebas: la idea es ejecutarlas de forma automática. En cada nivel de pruebas usaremos diferentes herramientas.
- Evaluar metas y emitir un informe: como resultado tendremos información del proceso realizado, obteniendo unos informes diferentes en cada nivel.

Para analizar y extraer conclusiones sobre las pruebas, necesitamos poder cuantificar el proceso y resultado, es decir, usar una métrica para medir las pruebas. Una métrica se define como una medida cuantitativa del grado en el que un sistema, componente o proceso posee un determinado atributo:

- Si no podemos medir, no podemos saber si estamos alcanzando los objetivos.
- Tiene que haber una relación entre lo que medimos y lo que queremos conocer.
- Se puede medir casi cualquier cosa: líneas probadas, número de errores encontrados, número de pruebas...

Debemos ser **efectivos** con las pruebas. Las causas por las que podemos no ser efectivos son:

- Si las pruebas están mal diseñadas, puede que el código quede "pobremente" probado, a pesar de ejecutar todas las líneas. Es complicado de detectar de forma automática.
- Podemos dejar de probar partes del código, esto se trata de un problema de la COBERTURA de código. Se puede analizar la cobertura de las pruebas. Una cobertura completa no indica la calidad de las pruebas, sólo la extensión.
- Niveles de cobertura:
  - A nivel de líneas.
  - A nivel de bloque.
  - A nivel de función.
  - A nivel de decisiones: cada condición se cumple como verdadero y falso, de forma grupal [if(a || b && c) -> que el resultado del if sea true y false]. Una decisión es una expresión booleana formada por condiciones y 0 o más operadores booleanos.
  - A nivel de condiciones: cada unidad de las condiciones se cumplen como verdadero y false [if(a || b && c) -> se mira tanto por a, como por b, como por c]. Una condición es el elemento mínimo de una expresión booleana [para if((A>0)&(B>0)), (A>0) es una condición y (B>0) es otra].

Análisis de cobertura de código: hay 7 formas de cuantificar la cobertura de código:

1. **State coverage (cobertura de líneas)**: 100% -> se han ejecutado todas las líneas.
2. **Branch (decisión) coverage (cobertura de decisiones/ramas)**: se ejecutan todas las ramas o decisiones en sus estados true y false. Un 100% de ramas quiere decir un 100% de líneas.
3. **Condition coverage**: se ejecuta cada condición en su condición verdadera y falsa. Para if((A>0) & (B>0)), tenemos que probar (A>0) verdadero y falso, y (B>0) verdadero y falso. Esto puede ser en dos casos de prueba: true-false, false-true [los operadores & y | calcula ambas condiciones, no adelanta casos]. Si el operador fuese && o || necesitaríamos de 3 casos de prueba: para ||, probamos false-true, true-\*, false-false; para && probamos true-true, true-false, false-\*.
4. **Condition decision coverage (cobertura de condiciones y decisiones)**: buscamos el 100% de condiciones sumándole el 100% de decisiones. Para ello, además de que cada condición sea probada como true y

*false, la decisión también debe ser probada como true y false. Para `if((A>0) & (B>0))`, tenemos que probar ( $A>0$ ) verdadero y falso, y ( $B>0$ ) verdadero y falso, además del `if(true)` y `if(false)`. Para ello, podemos tener dos casos de prueba: true-true, false-false.*

5. **Multicondition coverage:** debemos probar todas las combinaciones de condiciones posibles. Para `if((A>0) & (B>0))`, tenemos que probar que sea true-true, true-false, false-true y false-false. Implica cobertura de condiciones, decisiones (ramas) y condiciones/decisiones.
6. **Loop coverage (cobertura de bucles):** una cobertura de bucles implica ejecutar el bucle 0 veces, 1 vez, n veces siendo n un valor típico, y m veces siendo m el valor máximo (adicionalmente, se puede considerar  $m-1$  y  $m+1$ ).
7. **Path coverage (cobertura de caminos):** recorrer todos los caminos posibles del código. Un código con 10 if's necesitaríamos  $2^{10} = 1024$  tests! Si además tenemos un bucle que va entre 1 y 20 iteraciones, con 5 caminos distintos dentro del bucle, serían  $10^{14}$  caminos distintos. 100% cobertura de caminos implica 100% bucles, ramas y sentencias (líneas), pero no implica 100% condiciones múltiples.

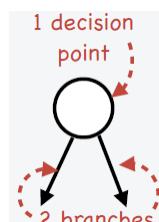
## Herramienta automática de análisis: JaCoCo

JaCoCo genera un informe con las siguientes métricas:

- Instructions: instrucciones byte code de Java.
- Branches: llama branch tanto a decisiones como condiciones: 2 decisiones con 4 condiciones = 6 branches [( $a \& b$ ) -> 2 decisiones + 4 condiciones; ( $a \mid\mid b$ ) -> 2 decisiones + 2 condiciones ya que cuando a es true, estamos en decisión true, con lo cual no contamos a (considerar grafo, el total de aristas del if es el branch)].
- CC: ¿Cuántos casos son necesarios para cubrir todos los caminos linealmente independientes?
- Líneas: se ejecuta al menos una instrucción byte code de la línea [`if(a || b)`; si se detiene en a, no se ejecutan todas las instrucciones byte code (faltó comprobar b), pero sí la línea].
- Métodos: se ejecuta al menos una línea del método. Se consideran constructores e inicializadores estáticos como métodos.
- Clases: una clase es ejecutada cuando al menos un método es ejecutado.

Hay diferentes niveles de análisis: proyecto, paquete, clase y método.

CC en JaCoCo: (branches- decision points + 1): 2 branches – 1 decision point +1 = 2 CC



- ¿Para qué sirve la CC? Mide la complejidad de un código (mayor CC, más complejo [necesita refactorizar]) y nos da una cota superior del número máximo de tests a realizar para ejecutar toda las líneas y condiciones del código.

Plugin JaCoCo para Maven:

- Usa un mecanismo llamado “Java Agent” que inserta los mecanismos de análisis de cobertura “on-the-fly”. Se pre-procesa los ficheros .class, y durante la ejecución se guarda el análisis de cobertura en un fichero .exec.
- **jacoco:prepare-agent:** prepara la propiedad argLine para pasarlal al JVM durante la ejecución de los tests. Se asocia a la fase `initialize` e indica qué clases deben ser instrumentadas y el fichero con los datos resultantes (`target/jacoco.exec`).
- **jacoco:prepare-agent-integration:** similar, pero a la fase `pre-integration-test` y el fichero es `target/jacoco-it.exec`.

- **jacoco:report**: a partir del `.exec`, genera un informe html, xml y cvs en `target/site/jacoco`. Asociada a la fase `verify`. Obtiene un informe con la cobertura de los tests unitarios.
- **jacoco:report-integration**: lo mismo pero usando el fichero `jacoco-it.exec`, almacenando en `target/site/jacoco-it`, y asociado a la fase `verify`. Obtiene un informe de la cobertura de sus clases en los tests de integración.
- **Jacoco:report-aggregate**: obtiene un informe de los tests de integración y unitarios de los módulos de los que depende.
- **jacoco:check**: comprueba si se ha alcanzado un mínimo de cobertura, y detiene la construcción del proyecto si no se alcanza. Se asocia a la fase `verify` y usa los ficheros `jacoco.exec`. Se puede configurar la cobertura:
  - **Nivel de cobertura (<element>)**: BUNDLE (aplicación), PACKAGE, CLASS, SOURCEFILE, METHOD
  - **Contador a configurar (<counter>)**: INSTRUCTION, LINE, BRANCH, COMPLEXITY, METHOD, CLASS.
  - **Valores a mirar (<value>)**: TOTALCOUNT, COVEREDCOUNT, MISSEDCOUNT, COVEREDRATIO, MISSEDRATIO.
  - **Definir valor máximo o mínimo**: <máximo>, <mínimo>.

**<counter>**

JaCoCo		instructions	branches	CC	lines	methods	classes						
Element		Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cqty	Missed	Lines	Missed	Methods	Missed	Classes
org.jacoco.examples		58%	64%	24	53	97	193	19	38	6	12		
org.jacoco.core		97%	93%	107	1,388	115	3,347	21	720	2	139		
org.jacoco.agent.rt		77%	84%	31	121	62	310	21	74	7	20		
jacoco-maven-plugin		90%	81%	35	183	44	407	8	110	0	19		
org.jacoco.cli		97%	100%	4	109	10	275	4	74	0	20		
org.jacoco.report		99%	99%	4	572	2	1,345	1	371	0	64		
org.jacoco.ant		98%	99%	4	163	8	429	3	111	0	19		
org.jacoco.agent		86%	75%	2	10	3	27	0	6	0	1		
Total		1,355 of 27,352	95%	143 of 2,125	93%	211	2,599	341	6,333	77	1,504	15	294

TOTALCOUNT COVEREDCOUNT

COVEREDRATIO

<value>

```

<rule>
  <element>BUNDLE</element>
  <limits>
    <limit>
      <counter>COMPLEXITY</counter>
      <value>COVEREDRATIO</value>
      <minimum>0.60</minimum>
    </limit>
  </limits>
</rule>
  
```

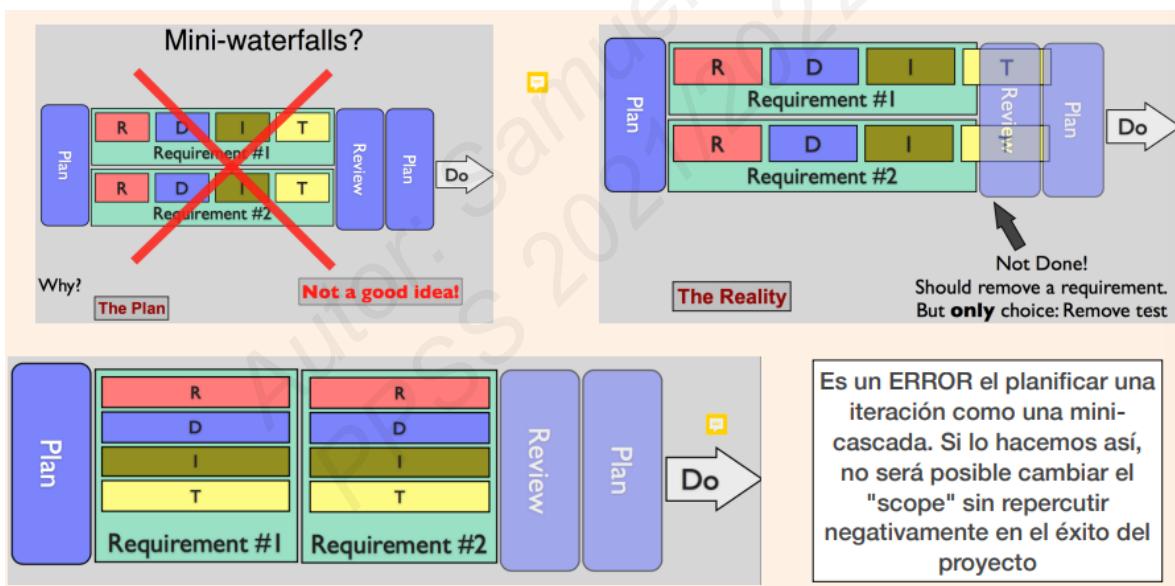
## Planificación de pruebas

El objetivo de un proyecto es materializar la idea que tiene el cliente, para ello, tenemos diferentes formas de planificar el proyecto:

- Planificación predictiva: soporta muy mal los cambios, desde el inicio se planifica todo. Un proyecto está en riesgo si su planificación se extiende más allá del horizonte del planificador y no ha considerado el tiempo necesario a realizar ajustes necesarios. Se trata del modelo secuencial (Waterfall lifecycle).
- Planificación adaptativa: tiene varios horizontes, se tratan de los modelos iterativos y ágil. Una aproximación iterativa no implica una mayor productividad, si no que la retroalimentación permite que sea menos costoso reparar los fallos debido a que se detectan antes.
  - Iterativo: tiene 2 horizontes, release e iteración. Reseñase se trata de definir al inicio todas las tareas para cada iteración, iteración se trata de en cada iteración, definir las tareas a realizar.
  - Ágil: tiene 3 horizontes: release, iteración y dia.

Modelo iterativo:

- Los modelos iterativos suelen estar conducidos por el cliente, con lo cual pueden validar sus expectativas cuanto antes.
- Las iteraciones deben ser **time-boxed**, no deben simular mini-waterfalls, si no centrarse en una funcionalidad por momento, para evitar que si nos quedamos cortos de tiempo, que se corte la fase final en todas las funcionalidades implementándose, y que lo que ocurra es que no se llegue a implementar la última funcionalidad, teniendo el resto correcto.



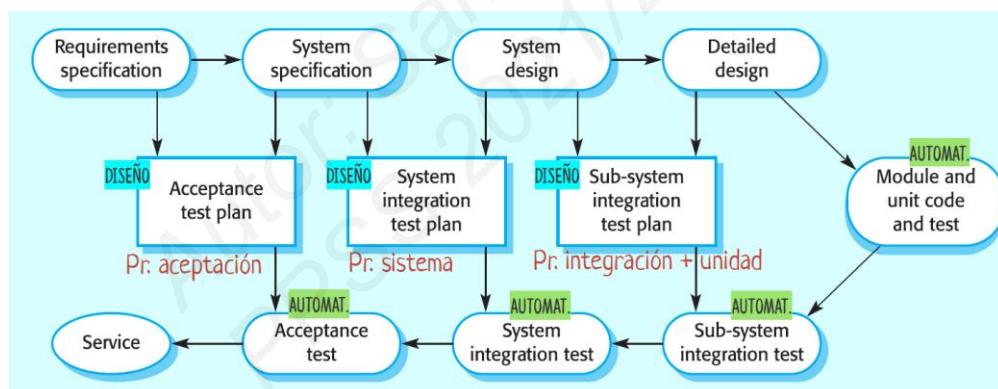
Diseño de las pruebas:

- El desarrollo del software debe equilibrar 3 factores: tiempo, dinero y funcionalidad. Para conseguir un resultado aceptable, hay que priorizar las pruebas más importantes y fijar criterios para conseguir unos objetivos y saber cuándo parar.
- El objetivo es que las pruebas sean efectivas (encuentren errores, para tener oportunidad de mejorar el código) y eficientes (mayor número de errores en menor número de pruebas).
- El **proceso de prueba** se forma por:
  - Test planning and control: planificar qué, cómo, quién y cuándo se van a hacer las pruebas, y qué se va a hacer si los planes no se ajustan a la realidad.
  - Test analysis and design: determinar casos de pruebas a utilizar y definir datos concretos + resultado esperado.

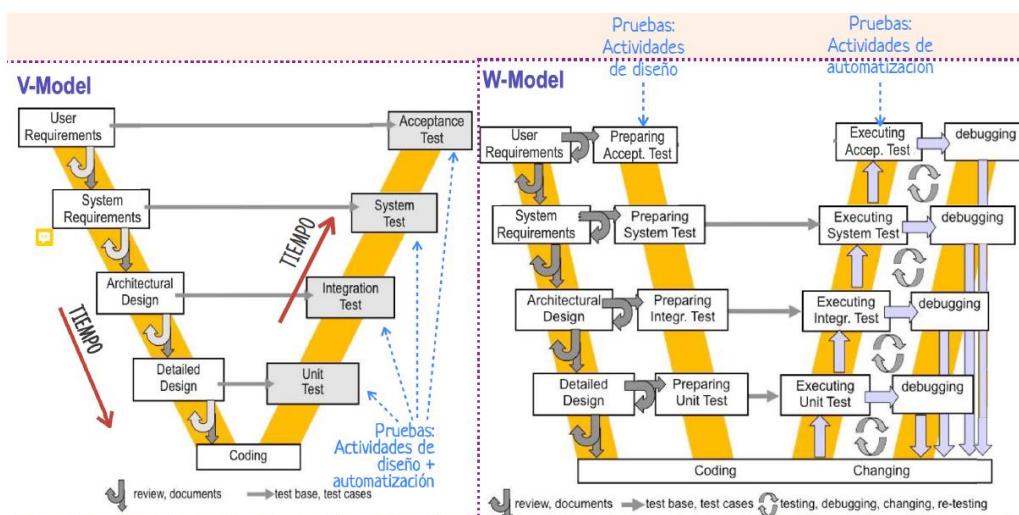
- Test implementation and execution: implementar los tests. Es la parte más visible, pero los pasos anteriores son necesarios.
  - Evaluating exit criteria and reporting: se verifica si se alcanza los completion criteria (cobertura, etc) y generar informe.
  - Test closure activities: pruebas finalizadas, asegurar que los informes están disponibles, etc.
- Tenemos que integrar las actividades de pruebas con el resto de actividades de desarrollo en el plan del proyecto. Para ello, tenemos diversos niveles de pruebas:

Secuencia temporal de niveles de pruebas (DINÁMICAS)				
OBJETIVO (cuantificable)	VERIFICACIÓN. Objetivo: encontrar defectos Realizada por los desarrolladores		VALIDACIÓN. Objetivo: ver en qué grado se satisfacen las expectativas del cliente. Requieren al usuario	
	UNIDAD	INTEGRACIÓN	SISTEMA	ACEPTACIÓN
Encontrar defectos en las unidades. Deben de probarse de forma AISLADA	Encontrar defectos en la interacción de las unidades. Debe establecerse un ORDEN de integración	Encontrar defectos derivados del comportamiento del sistema como un todo.	Valorar en qué GRADO se satisfacen las expectativas del cliente. Basadas en criterios de ACEPTACIÓN (prop. emergentes) cuantificables	
DISEÑO	Camino básico (CB) Particiones equivalentes (CN)	Guías (consejos) en función de los tipos de interfaces (CN)	Basado en casos de uso (CN) Transición de estados (CN)	Basado en requerimientos (CN) Basado en escenarios (CN) Pruebas de rendimiento (CN) Pruebas $\alpha$ y $\beta$ (CN)
AUTOMATIZACIÓN (implement. + ejecución)	Java, Maven, JUnit, EasyMock	Java, Maven, JUnit, DbUnit, EasyMock	Java, Maven, JUnit, DbUnit, Webdriver	Java, Maven, JUnit, Webdriver: JMeter Usuario ( $\alpha$ y $\beta$ )

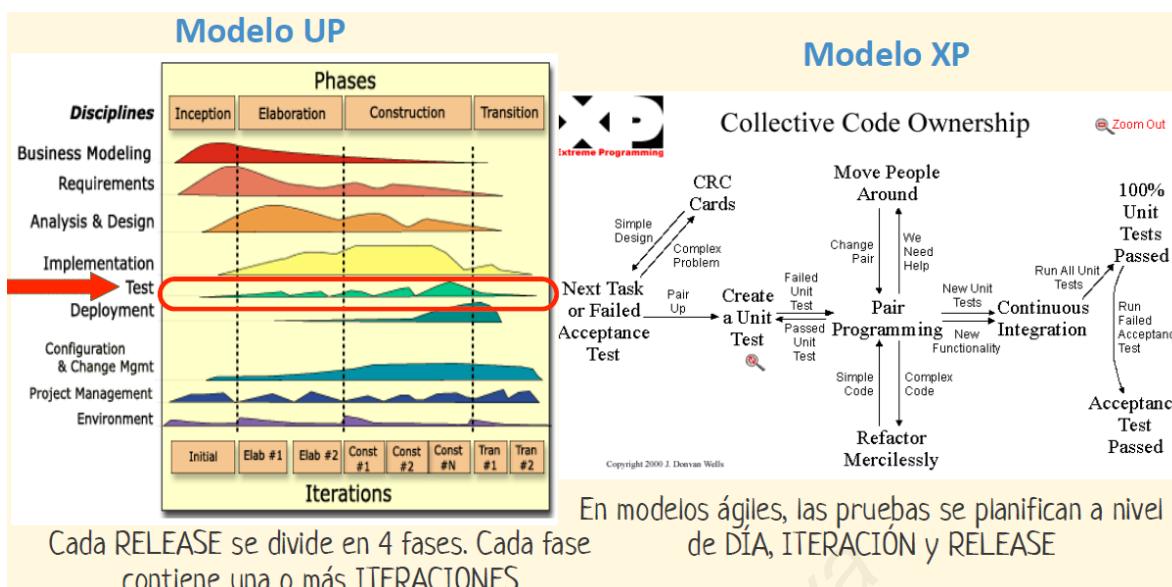
- Debemos saber también cuando diseñar cada tipo de prueba



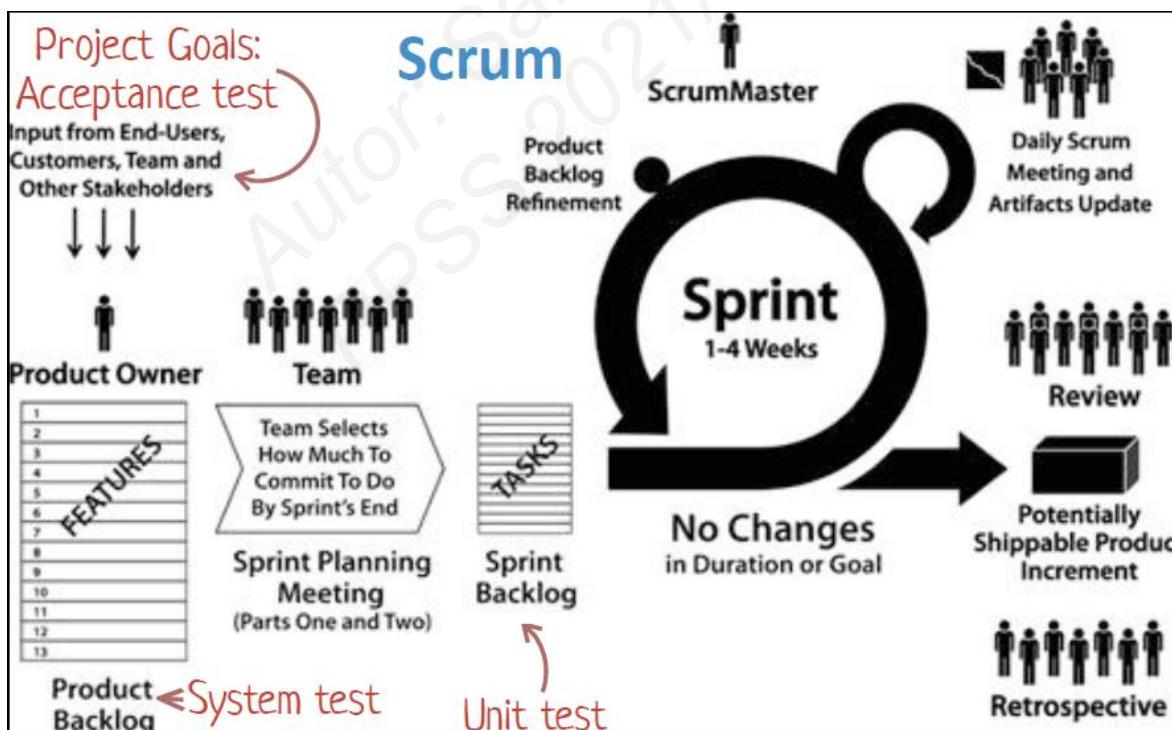
- Pruebas en **modelos secuenciales**: se contempla un equilibrio entre las pruebas estáticas y dinámicas.



- Pruebas en **modelos iterativos y ágiles**, se planifican a nivel de iteración y release (conjunto de iteraciones).
  - El modelo XP trata de diseñar primer los tests, luego implementar el código para que pase los tests, todos los días hay un plan que debemos cumplir.



- El modelo scrum es más flexible, partes de una conversación con quien tienes el negocio, defines las características a implementar, que forman parte de una serie de tareas que vas haciendo en iteraciones que se llaman sprint (formado por iteraciones, cada iteración del mismo tamaño). En el scrum no hay gestor de proyecto, tenemos el scrumMaster, que guía un poco al grupo, pero sigue siendo uno más. Primer implementa los tests y luego implementamos, y luego a saco con la integración.



Aspectos de cada plan:

- Aspecto del plan XP:
  - Story cards: historias de usuario que indica lo que quiere hacer, en el reverso se tiene los criterios de aceptación.
  - Task list: lista de tareas para las story de cada iteración.

- Release plan board: plan de entregas (release) con la historia general de lo que se hará en la iteración.
- Iteration plan board: tablero que divide en tareas lo que se va a implementar, escrito en tarjetas. Cada uno coge una tarjeta, lo trabaja, pone su nombre y va anotando lo que tarda, y en acabar lo vuelve a colocar en el tablero. De esta manera, cada uno se autoasigna la tarea e indica lo que duró y el estado del mismo.
- Aspecto del scrum:
  - Scrum task board: tablero con los tasks en tarjetas en una columna de pendientes, con en el reverso los tests a realizar. Tendrá una columna de tareas en proceso y otra de completados.

Prácticas de pruebas:

- TDD: los tests guían al resto de cosas. Aquí no habrá código sin probar, ya que si no existe el test, no se crea el código. Todo lo que creamos, es para pasar una prueba.
  - Pasos:
    - Primero escribo un test.
    - Luego escribo código para pasar el test.
    - Refactorizo el código.
  - Aproximaciones para crear el proyecto:
    - **Clásica:** empiezo por abajo, lo más pequeño, lo creo, y cuando ya lo haya probado y refactorizado múltiples veces, añado otro módulo superior, yendo hacia arriba. Implementando primero las unidades de bajo nivel, y luego las de alto nivel. Necesitaremos pocos dobles.
    - **BDD:** empezar por arriba, creas test de una unidad de alto nivel, con lo cual necesitarás muchos dobles (mocks o stubs), y vas sustituyendo los dobles con sus tests y su implementación.
- Integraciones continuas:
  - En XP es necesario. Consiste en integrar continuamente el código del proyecto. Para ello tendremos un proceso que únicamente se dedicará a eso, cogerá las unidades y las integra (servidor de integraciones continuas).
  - El desarrollador sube el código probado con pruebas unitarias, la máquina central lo recoge y realiza prueba de integración (normalmente lo hace en intervalos cortos), comprueba si hay un fallo, y en caso de haberlo, lo informa para arreglarlo lo más pronto posible.
- BDD: empieza por arriba, empieza por lo más cercano al usuario. Cuando preguntas a un cliente qué quiere, no te dice lo que quiere, si no lo que cree que debes hacer para conseguir lo que quiere.
  - Recoge las conversaciones con el cliente y transcribe los objetivos que importan para el desarrollo, obteniendo ejemplos reales de los requisitos funcionales. Entonces vienen “los 3 amigos” (tester, product owner y developer) para definir las pruebas.
  - El ciclo de desarrollo de BDD es: escribe la historia a implementar, crea un test con las especificaciones, automatiza y ejecuta el test, informa del report, y en caso de ser favorable sube el código.
  - BDD no implementa tests unitarios, si no especificaciones de bajo nivel. Convierte los tests en un “Given *initial stage* When *action under test* Then *outcome*” para definir la especificación.
  - De la conversación del cliente con “los 3 amigos”, salen los tests, los cuales también son especificaciones de bajo nivel.