Resumen por: Samuel Oliva
Resumen PPSS Control 1
Curso 2021/2022

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

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>
```

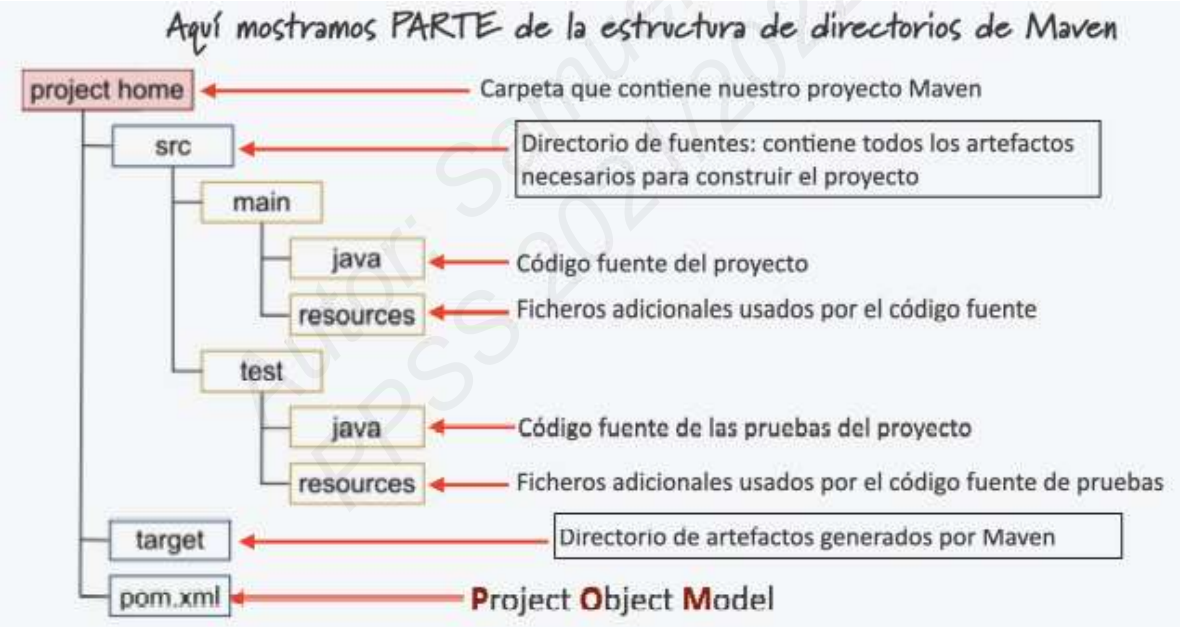
Un proyecto Maven debe contener en su raíz el pom.xml, que nos permite configurar la secuencia de acciones a realizar (build script) mediante la etiqueta <build>. También podemos indicar qué librerías (ficheros .jar) son necesarias con la etiqueta <depencencies>. De este fichero destacan 4 "secciones": **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.

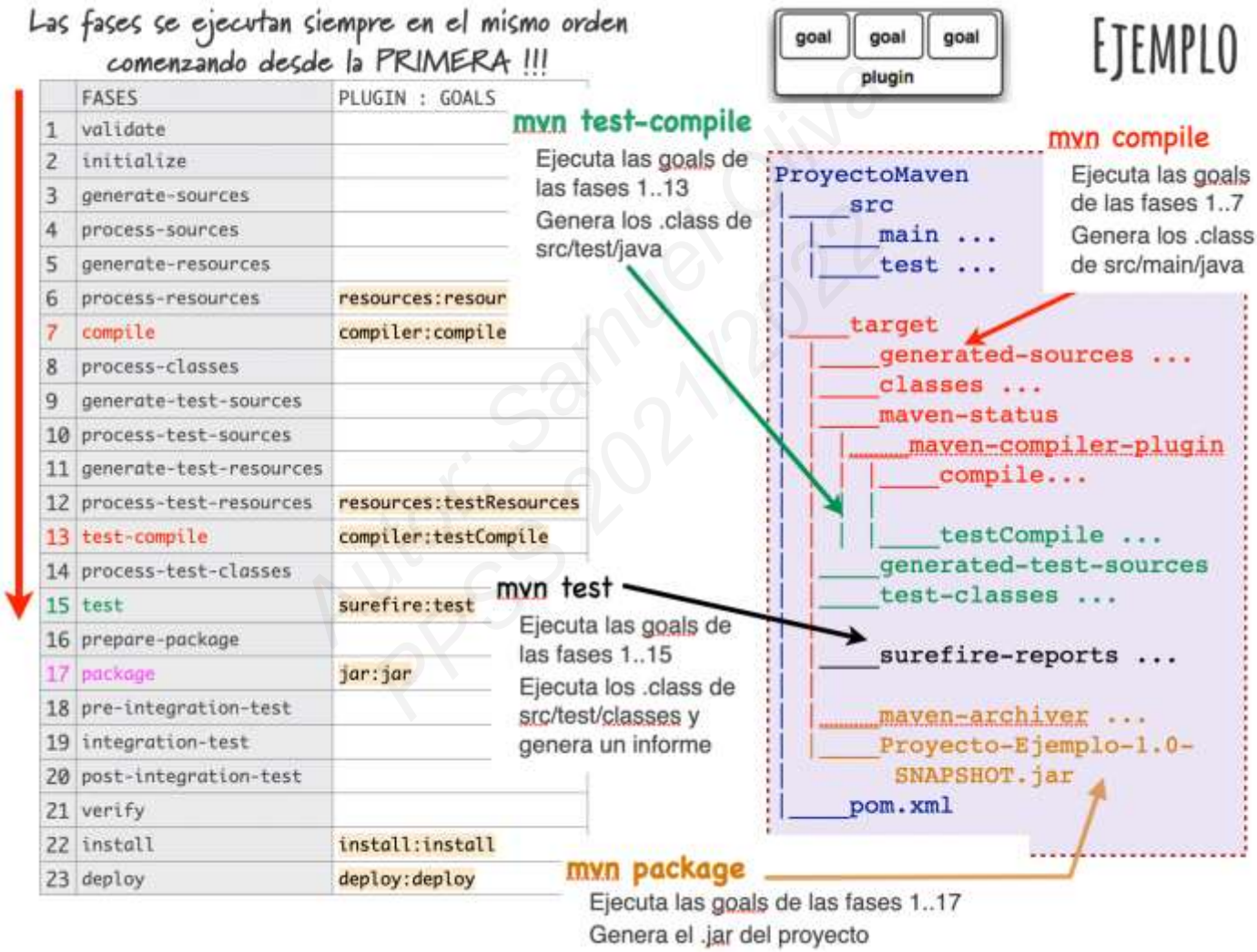
```
<dependencies>
	<!-- jUnit5 (Test, BeforeAll, BeforeEach, Assertions, Tag) -->
	<dependency>
		<!-- Directorio en .m2/repository -->
		<groupId>	org.junit.jupiter	</groupId>
		<!-- Carpeta del artefacto -->
		<artifactId>	junit-jupiter-engine	</artifactId>
		<!-- Versión del artefacto (cada version es un directorio con el .jar de la version) -->
		<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.

Directorios 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



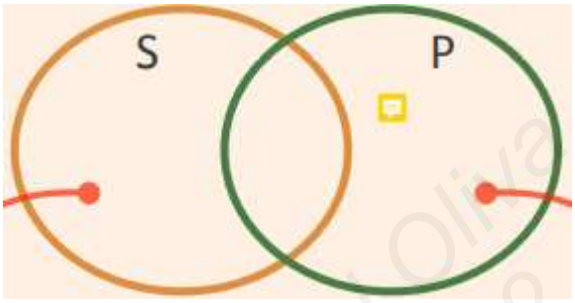
	FASES	PLUGIN : GOALS
1	validate	
2	initialize	
3	generate-sources	
4	process-sources	
5	generate-resources	
6	process-resources	resources:resour
7	compile	compiler:compile
8	process-classes	
9	generate-test-sources	
10	process-test-sources	
11	generate-test-resources	
12	process-test-resources	resources:testResources
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

Pruebas: proceso de diseño

Nuestro objetivo es encontrar defectos, no demostrar la ausencia de ellos. Si no se encuentran, no quiere decir que no haya. Las actividades del proceso de pruebas es (según ISQTB Foundation Level Syllabus):

1. Planificación y control de las pruebas: definimos los objetivos de las pruebas y nos aseguramos de cumplirlas
2. Diseño de las pruebas: decidimos con qué datos de entrada CONCRETOS probamos el código (las óptimas).
3. Implementación y ejecución de las pruebas: implementamos el driver para probar nuestro SUT fácilmente.
4. Evaluación del proceso de pruebas: comprobamos si hemos alcanzado los objetivos de pruebas planificados.

Objetivo: encontrar discrepancia entre el comportamiento del programa especificado y el implementado. Tenemos dos conjuntos: S (Specified) y P (Programmed). Queremos que ambos conjuntos sean idénticos (todo lo especificado esté implementado, y que sólo se implemente lo especificado), pero no siempre ocurre. Nuestras pruebas dinámicos (requieren ejecutar el código) deberán ser sobre casos especificados e implementados.



Los casos de pruebas se obtienen de algún método de diseño de pruebas. Usaremos una Tabla de casos de prueba para indicar las pruebas a realizar, la cual contiene los datos de entrada y el resultado esperado. Los casos de prueba consisten en **datos de entrada concretos + resultado esperado**. Puede ser un diseño de casos de pruebas basadas en la especificación o un diseño de casos de pruebas basadas en la implementación.

Cualquier método basado en la implementación sigue estos principios:

1. Analizar el código y obtener una representación en forma de grafo.
2. Seleccionar un conjunto de caminos en el grafo según algún criterio.
3. Obtener un conjunto de casos de prueba que ejercitan dichos caminos.

Dependiendo del método utilizado, podemos obtener diferentes casos de pruebas, pero siempre serán efectivos y eficientes. Sólo se aplican a nivel de unidades del programa. No puede detectar TODOS los defectos del programa, aunque seleccionemos todas las posibles entradas, no podemos detectar "caminos que faltan", es decir, comportamientos que falten al programa; de la misma manera, si no existe un código que maneja una determinada entrada diremos que falta un camino en el programa.

Cualquier método basado en la especificación sigue estos principios:

1. Analizar la especificación y PARTICIONAR el conjunto S
2. Seleccionar un conjunto de comportamientos según algún criterio
3. Obtener un conjunto de casos de prueba que ejercitan dichos comportamientos

Los métodos pueden estar centrados en pruebas unitarias, pruebas de sistema o pruebas de aceptación. En todos ellos, la identificación de DOMINIOS de entradas y salidas contribuye a PARTICIONAR los comportamientos.

Diseño de pruebas de Caja Blanca

Se basa en conceptos de teoría de grafos. El resultado esperado se obtiene SIEMPRE de la especificación. Los valores de entrada los determinamos de la IMPLEMENTACIÓN. Los comportamientos probados pueden no estar especificados. Se puede detectar comportamientos no especificados, pero no comportamientos no implementados.

Emplea grafos de flujo de control (**CFG**), el cual es un grafo dirigido donde cada nodo es 0 o más sentencias secuenciales y una única condición. Las aristas representan el flujo de ejecución entre dos conjuntos de secuencias (si tiene una condición tendrá etiqueta T o F). TODAS las sentencias de la unidad a probar deben estar representadas en el grafo.

La ejecución de una secuencia de instrucciones desde el punto de entrada a la salida se llama camino. Cada camino se corresponde con un comportamiento. Un conjunto de valores específicos de entrada provoca que se ejecute un camino específico en el programa. Sólo habrá un punto de entrada y un punto de salida.

Criterios de selección de caminos: estructuralmente un camino es una secuencia de instrucciones en una unidad, pero semánticamente un camino es una instancia de una ejecución de una unidad. Es necesario seleccionar un conjunto de caminos bajo un criterio de selección, por ejemplo: elegir todos los caminos, ejecutar TODAS las sentencias al menos una vez, ejecutar TODAS las condiciones al menos una vez… No generaremos entradas para los tests en los que se ejecute el mismo camino varias veces, **a no ser** que este cambie el estado del sistema, haciendo que no sean idénticos.

Método del camino básico (McCabe's Basis Parth Method; método para obtener los caminos independientes a partir del grafo). El objetivo es ejecutar TODAS las sentencias del programa al menos una vez y que cada condición se haya ejecutado tanto en T como F. Un camino independiente es un camino que ejecuta al menos una nueva sentencia/condición (arista) que lo difiera del resto. El número de caminos independientes determina el número de filas de la tabla.

Complejidad ciclomática (CC) determina del número de filas de la tabla. Indica el máximo número de caminos independientes en el grafo. El valor máximo tolerable es 10. Un alto CC implica mayor complejidad y mayor esfuerzo de mantenimiento. Para reducir la CC se puede refactorizar el código incrementando la abstracción (modularizar). Métodos de calcular el CC (los dos últimos sólo son válidos si no hay saltos incondicionales):

- CC = número de arcos – número de nodos + 2
- CC = número de regiones
- CC = número de condiciones + 1

Descripción del método:

1. Construir el grafo de flujo del programa (CFG) a partir del código
2. Calcular la complejidad ciclomática (CC)
3. Obtener los caminos independientes del grafo (McCabe's / Método del camino básico)
4. Determinar los datos CONCRETOS de entrada (y salida esperada de la especificación) de la unida a probar (SUT), de forma que se ejerciten todos los caminos independientes. Estos se representarán en una tabla de casos de prueba (Camino, Entradas, Resultado esperado).

Pruebas de Caja Negra (Particiones equivalentes)

Obtendremos los casos de pruebas a partir de la ESPECIFICACIÓN. Siempre obtendremos un conjunto EFICIENTE y EFECTIVO. Podemos detectar comportamientos no implementados, pero nunca implementados pero no especificados. Con caja negra podemos preparar las pruebas antes de que se haya implementado.

Es un proceso que identifica un conjunto de CLASES DE EQUIVALENCIA para cada entrada y salida del SUT. Cada clase de entrada representa un subconjunto del total de datos posibles que tienen un mismo comportamiento (**los elementos de una partición de entrada tiene su "imagen" en la misma partición de salida**).

Una partición se identifica como a una condición de entrada/salida. Puede aplicarse a una única variable o a un subconjunto de ellas. Las variables de entrada no necesariamente son los "parámetros" del SUT. Deben ser DISJUNTAS. Los elementos de una misma partición deben tener su "imagen" en la misma partición de salida.

Las clases de equivalencia pueden clasificarse como válidas o inválidas. Las entradas inválidas normalmente tienen asociadas salidas inválidas. SOLO puede haber UNA entrada inválida por caso de prueba.

Identificación de las clases de equivalencia:

1. Identificar las particiones para CADA entrada/salida siguiendo las heurísticas (si varios valores dependen entre ellos, se unen como a una partición [por ej, que el valor de uno sea correcto o no dependiendo del otro]) (si agrupas entradas, NO debes considerar más de una partición inválida en la agrupación [una inválida debe estar compuesta por la inválida de una entrada y las demás entradas válidas]):
    a. Si la E/S especifica un rango de valores válidos (x valores entre 0 y 12), especificamos 1 válida (el rango) y 2 inválidas (por encima y por debajo)
    b. Si la E/S especifica un cantidad de valores válidos (x puede tomar entre 1 y 3 valores), especificamos 1 válida (dentro de la cantidad) y 2 inválidas (ninguna y de más)
    c. Si la E/S especifica un conjunto de valores válidos (x puede ser {a, b, c}) definidos una válida (dentro del conjunto) y una inválida (fuera del conjunto) [en Java además podemos indicar el null]
    d. Si cada valor de entrada se trata de forma diferente, definir una clase válida para cada valor de entrada (si número 1 provoca un comportamiento distinto a 2, entonces se separan)
    e. Si la E/S especifica una situación DEBE SER (x empieza por dígito), especificamos 1 válida y 1 inválida
    f. Si los elementos de una partición se tratan de forma distinta, subdividir la partición en particiones más pequeñas. (si tengo partición de día+mes, separar particiones en conjunto de partición de día + partición de mes)
2. Identificar los casos de prueba:
    a. Primero meter todas las particiones válidas, intentando meter el máximo número de clases válidas en cada una.
    b. Después metes las inválidas, una por caso de prueba.
    c. Elegir un valor concreto para cada partición.
    d. El resultado será una TABLA con tantas FILAS como CASOS DE PRUEBA obtenido.

Consejos:

- Etiquetas las particiones de una misma E/S con la misma letra.
- Si la E/S es un objeto, se debe considerar cada atributo como un parámetro diferente.
- Los objetos en Java son referenciados, por lo tanto hay que considerar el valor NULL
- Las E/S no son únicamente los parámetros.

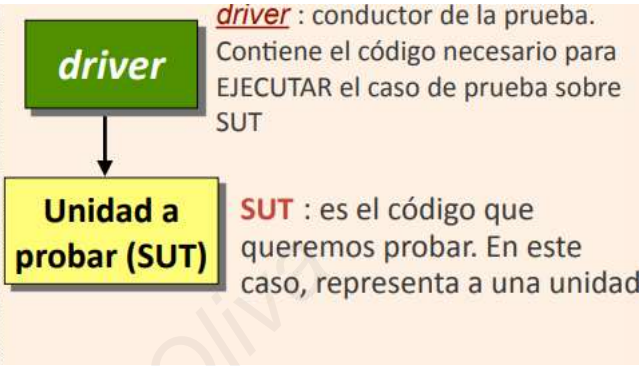
Pruebas unitarias

Implementar código (drivers) para ejecutar los tests de forma automática. El objetivo es, dado un caso de prueba, obtener le resultado real, compararlo con el esperado y emitir un informe que dirá si se ha encontrado un defecto o no con ese caso de prueba. Implementaremos tantos drivers como casos de prueba.

Queremos probar las unidades por separado. Una unidad es una "pieza" del código. Nosotros llamaremos unidad a un método Java. Se pueden realizar pruebas unitarias de forma estática o dinámica (para detectar errores necesitamos ejecutar el código). La cuestión será cómo aislar el código de cada unidad.

El algoritmo de un driver es el siguiente:

```
1. //algoritmo de un driver
2. informe driver() {
3.    d= prepara_datos_entrada();
4.    esperado= resultado_esperado;
5.    //invocamos al SUT
6.    real= SUT(d);
7.    //comparamos el resultado
8.    //real con el esperado
9.    c= (esperado == real);
10.  informe= prepara_informe(c);
11.  return informe;
12.}
```



JUnit 5 (org.junit.jupiter.api)

JUnit es un API java que permite implementar los drivers y ejecutar los casos de prueba sobre nuestra SUT (System Under Test). Se pueden implementar drivers de pruebas unitarias (SUT: una unidad) y también de integración (SUT: conjunto de unidades).

Los tests en JUnit son MÉTODOS sin parámetros, que devuelve void y está anotado con @Test (org.junit.jupiter.api.Test). Los tests deben agruparse lógicamente con el SUT (package mismo.paquete.delSUT). La clase de pruebas será la del SUT seguida de "Test" (SUTaProbarTest). No es necesario que la clase ni los tests sean "public". Las ubicaciones físicas del SUT y Driver si usamos Maven son:

- Fuentes (código) -> *Driver:* /src/test/java; *SUT:* /src/main/java
- Ejecutables (.class) -> *Driver:* /target/test-classes; *SUT:* /target/clases
- Informes -> *Driver:* /target/surefire-reports

Sentencias assert (org.junit.jupiter.api.Assertions)

Sentencias para determinar el resultado de las pruebas y emitir un informe correspondiente. Son métodos estáticos, se usan para comparar el resultado esperado con el real y el orden de los parámetros es: resultadoESPERADO, resultadoREAL [, mensajeOpcional]. Generar una excepción de tipo AssertionFailedError si no se cumple.

Agrupación de aserciones: si tienes varias aserciones consecutivas, cuando se lance una, las demás no se ejecutarán. Con assertAll, se compararán todas y se lanza MultipleFailuresError en caso de fallar alguno:

```
assertAll(String heading, () -> assertEquals(…), assertArrayEquals(…), …);
```

Excepciones: podemos comprobar si queremos o no que se lance una excepción:

```
ExpectedException exception = assertThrows(ExpectedException.class, ()-> sut(…));
assertEquals("msg esperado", exception.getMessage());

SUT resultadoObtenido = assertDoesNotThrow( ()-> sut(…));
```

Anotaciones @BeforeEach, @BeforeAll, @After… (org.junit.jupiter.api.BeforeAll, org…)

Si los tests requieren las mismas acciones, podemos usarlos. El orden es: BeforeAll, BeforeEach, driver, AfterEach, BeforeEach, …, AfterEach, AfterAll. Esto también permite preparar y restaurar el estado del entorno entre cada ejecución de test.

- @BeforeEach y @AfterEach usa métodos void sin parámetros (void each ()).
- @BeforeAll y @AfterAll usa métodos estáticos void sin parámetros (static void all ()).

Etiquetado de tests: @Tag("…") (org.junit.jupiter.api.Tag)

Permite filtrar los tests mediante etiquetas. Se puede etiquetar tanto la clase como el test con @Tag("etiqueta"). Para hacer uso de las etiquetas al ejecutar los test con Maven usamos –Dgroups (definimos las etiquetas que queremos ejecutar) y –DexcludedGroups (indicamos las etiquetas que no queremos ejecutar a pesar de tener una etiqueta en –Dgroups). Podemos usar los operadores "&" (and), "|" (or, equivale a la coma ",") y "!" (not) junto a los paréntesis:

- `mvn test -Dgroups=conExcepciones&parametrizado` (ejecutamos los etiquetados conExcepciones y parametrizados, no sirve que sólo tenga uno de los dos)
- `mvn test -DexcludedGroups=excluido,parametrizado` (ejecutamos todos menos los indicados)

Test parametrizados @ParametrizedTest @ValueSource (org.junit.jupiter.params.ParametrizedTest; org.junit.jupiter.params.provider.ValueSource;)

Si el código es idéntico y sólo varia los valores concretos del caso (valores de entrada y resultado esperado), podemos usar un test parametrizado. Se implementa un único test con la anotación @ParametrizedTest y se tendrá como parámetro los valores concretos. Si sólo requiere de un parámetro podemos usar @ValueSource:

```
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;

@ParameterizedTest
@ValueSource(strings = {"racecar", "radar", "able was I ere I saw elba"})
void palindromes(String candidate) {
        assertTrue(c.isPalindrome(candidate));
}
```

Si tenemos que insertar otro tipo de dato: doubles={…}, ints{…}, longs={…}. El test se ejecuta 3 veces, cada uno con un valor del value source.

Test parametrizados @ParametrizedTest @MethodSource (org.junit.jupiter.params.ParametrizedTest; org.junit.jupiter.params.provider.MethodSource; org.junit.jupiter.params.provider.Arguments;))

Si el método anotado con @ParametrizedTest requiere de varios argumentos se usará un método para obtenerlos con el cual se indica con @MethodSource:

```
@ParameterizedTest(name = "User {1}, when Alert level is {2} should have access to transporters of {0}")
@MethodSource("casosDePrueba")
void testParametrizado(boolean expected, Person user, Alert alertStatus) {
        transp.setAlertStatus(alertStatus);
        assertEquals(expected, transp.canAccessTransporter(user),
                () -> generateFailureMessage("transporter", expected, user, alertStatus));
}

private static Stream<Arguments> casosDePrueba() {
        return Stream.of(
                Arguments.of(true,  picard,  Alert.NONE   ),
                Arguments.of(true,  barclay, Alert.NONE   ),
                Arguments.of(false, lwaxana, Alert.NONE   ),
                Arguments.of(false, lwaxana, Alert.YELLOW ),
                Arguments.of(false, q,       Alert.YELLOW ),
                Arguments.of(true,  picard,  Alert.RED    ),
                Arguments.of(false, q,       Alert.RED    )
        );
}
```

El método que devuelve los argumentos devuelve un Stream con los argumentos de tipo Arguments. Por defecto el *surefire* no tendrá en cuenta el atributo name del @ParametrizedTest. Para user el atributo hay que cambiar la configuración del plugin *surefire*. Esta configuración también es necesaria para usar la anotación @DisplayName("name").

También podemos configurar las propiedades group y excludedGroups del plugin de *surefire*, sin embargo si lo configuramos directamente no podrá modificarse. También se puede modificar desde la línea de comandos, como antes se explicó (mvn test –Dgroups=etiquertas –DexcludedGroups=excluidos [-Dtest=clasesTest]).
Si queremos configurar desde el pom la configuración "por defecto" y poder modificarlo también por comando, habrá que usar una variable del pom (properties):

```
<properties>
        <filtrar.por>importantes,fase1</filtrar.por>
</properties>
...
<plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>3.0.0-M5</version>
        <configuration>
                <groups>${filtrar.por}</groups>
        </configuration>
</plugin>
```

Para poder modificar la variable (y con ello el "filtro"):

`mvn test –Dfiltrar.por=rapidos`	(Ejecuta los etiquetados con rapidos)
`mvn test`	(Ejecuta los etiquetados con importantes y fase1)
`mvn test –Dfiltrar.por=""`	(Ejecuta todos los tests, se borra el filtro)

Dependencias externas

Las pruebas unitarias dinámicas requieren ejecutar cada unidad (SUT) de forma AISLADA. Si nuestra SUT invoca a otras unidades, hay que CONTROLAR la ejecución de LAS DEPENDENCIAS EXTERNAS (colaboradores, DOCs [dependend-On Component]) para aislar el nuestro. Para poder realizar el reemplazo de los DOC necesitamos que el código tenga uno o varios SEAMS. Para conseguir un sean podemos refactorizar el SUT si es necesario.
Pasos para automaziar las pruebas:

1. Identificar las dependencias externas: una dependencia externa es otra UNIDAD que interactúa con el código a probar (SUT) y el cual no controlamos. Utilizaremos dobles para controlar el resultado del DOC y así aislar el SUT.
2. El SUT debe ser testable: necesitamos tener un SEAM para poder inyectar el doble. El doble debe implementar la misma interfaz que el colaborador o extender de la misma clase. Las formas de inyectar el doble son: como parámetro a nuestra SUT, a través del constructor, a través de un método setter o a través de una factoría local en la clase del SUT o una clase factoría. Una clase factoría implica añadir código en el src/main/java que podría ser innecesario en producción.
3. Implementamos el doble: debe ser lo más genérica posible. Este stub reemplazará al DOC en la ejecución del test.
4. Implementación del Driver: el driver se encarga de preparar los datos, de crear el doble y de inyectar el doble en la SUT antes de ejecutar las pruebas. Usaremos tantos STUBS como dependencias externas a controlar. Usar un STUB permite la verificación del resultado de las pruebas, con lo cual realizamos una verificación basada en el estado. Los drivers de pruebas UNITARIAS son DIFERENTES a los drives de pruebas de INTEGRACIÓN. NO se deben implementar dobles para los setters ni getters.

Clase factoría:

```
public class FactoriaIOperacionBO {
	private static IOperacionBO operacion = null;
	//Factoria con el cual recogeremos la clase
	public static IOperacionBO create(){
		if(operacion != null){
			return operacion;
		} else {
			return new Operacion();
		}
	}
	//Setter de la clase factoria
	static void setOperacion(IOperacionBO op){
		operacion=op;
	}
}
```

EasyMock (org.easymock.EasyMock)

Creación de Stubs: EasyMock.niceMock(Clase.class).

No tiene en cuenta el orden de las invocaciones, se permite las invocaciones a todos los métodos, si no están programadas se devolverá null/0/false; las llamadas a métodos se realizan con argumentos especificados (para evitarlo, se usa los EasyMock.any-(), como son anyObject(), anyInt()…).
Proceso: (import static org.easymock.EasyMock; para los any-())

1. Creación del stub:

```
Dependencia1 stub0 = EasyMock.niceMock(Dependencia1.class);
Dependencia1 partialStub = EasyMock.partialMockBuilder(Dependencia1.class)
                                   .withConstructor(2.5, "EntradaConstructor")
                                   .addMockedMethod("Methodo1", int.class, String.class)
                                   .addMockedMethod("metodoVoid", double.class).niceMock();
Dependencia1 stub1 = EasyMock.createMockBuilder(Dependencia1.class)
                                   .withConstructor(double.class, String.class)
                                   .withArgs(2.3, "ArgConstruct")
                                   .niceMock();
```

2. Programar las expectativas del stub (hay que tener en cuenta si puede lanzar excepción la dependencia):

```
EasyMock.expect(stub0.metodo1(anyInt(), anyString())).andStubReturn(9);

EasyMock.expect(stub0.metodo2(20)).andStubReturn(false);
EasyMock.expect(stub0.metodo2(EasyMock.not(EasyMock.eq(20)))).andStubReturn(true);

Assertions.assertDoesNotThrow( ()->stub1.metodoVoid() );
EasyMock.expectLastCall().asStub();
```

3. Indicamos que el stub está listo:

```
EasyMock.replay(dep0, dep1);
```

Creación de Mocks: EasyMock.strickMock(Clase.class).

Tiene en cuenta el orden de las invocaciones a sus métodos. Si queremos que también se tenga en cuenta el orden de la invocaciones entre los distintos mocks, usaremos un IMocksControl strictControl.

Proceso: (import org.easymock.IMocksControl;)

1. Creación del mock:

```
Dependencia1 mock0 = EasyMock.strickMock(Dependencia1.class);
Dependencia1 partialMock0 = EasyMock.partialMockBuilder(Dependencia1.class)
                            .withConstructor(2.5, “EntradaConstructor”)
                            .addMockedMethod(“Methodo1”, int.class, String.class)
                            .addMockedMethod(“metodoVoid”, double.class).strickMock();
//Creamos una instancia de ctrl y los mocks van a estar relacionados por ella
IMocksControl ctrl  = EasyMock.createStrickControl();
Dependencia1 dep0 = ctrl.createMock(Dependencia1.class);
Dependencia1 dep1 = EasyMock.partialMockBuilder(Dependencia1.class)
                            .withConstructor(2.5, “EntradaConstructor”)
                            .addMockedMethod(“Methodo1”, int.class, String.class)
                            .addMockedMethod(“metodoVoid”, double.class).mock(ctrl);
```

2. Programamos las expectativas del mock:

```
EasyMock.expect(dep0.metodo1(“Parametros”, “concretos”, 1))
                .andReturn(false).times(3)
                .andReturn(true).times(2)
                .andThrow(new ExpecionLanzada());

Assertions.assertDoesNotThrow( ()-> dep1.metodoVoid(3.43));
EasyMock.expectLastCall().andVoid().times(3)
                .andThrow(new ExpecionLanzada());
```

3. Indicamos que está listo:

```
EasyMock.replay(mock0, partialMock0);
ctrl.replay()
```

4. Verificamos las expectativas del mock:

```
EasyMock.verify(mock0, partialMock0);
ctrl.verify();
```