

P02- Automatización de pruebas: Drivers

Os recordamos que el plazo para entregar los ejercicios de esta práctica (P02) termina justo ANTES de comenzar la siguiente sesión de prácticas (P03) en el laboratorio (los grupos de los lunes, por lo tanto, el próximo lunes, los de los martes, el próximo martes, ...). Pasado este plazo los profesores no revisarán vuestro trabajo sobre P02 (de forma individual), con independencia de que se indiquen en clase las soluciones que os permitirán saber si las vuestras son correctas.

Implementación de drivers con JUnit 5

En esta práctica implementaremos los drivers y automatizaremos las pruebas unitarias teniendo en cuenta los casos de prueba que hemos diseñado en la sesión anterior. Recuerda que a partir de ahora, nuestro elemento a probar se denominará SUT (con independencia de que sea una unidad o no), y por el momento, va a representar a una UNIDAD (que hemos definido cómo un método java).

Usaremos JUnit 5 para implementar nuestros tests, pero no se trata solamente de aprender el API de JUnit, sino de usarlo siguiendo las normas que hemos indicado en la clase de teoría: por ejemplo: los tests tienen que estar físicamente separados de las unidades a las que prueban, pero tienen que pertenecer al mismo paquete, los tests tienen que implementarse sin tener en cuenta el orden en el que se van a ejecutar, cada método anotado con `@Test` debe contener un único caso de prueba...

Ejecución de drivers con JUnit 5

La ejecución de los drivers se realizará integrada en el proceso de construcción, usando Maven. Es decir, de forma automática (pulsando un botón) se ejecutarán todas las actividades conducentes a obtener los informes de prueba de la ejecución de nuestros tests (casos de prueba). Es importante que tengas clara la secuencia de acciones de dicho proceso de construcción .

Recuerda que queremos independizar nuestro proceso de construcción de cualquier IDE, por lo que usaremos la instalación de Maven de `/usr/local`. Por lo tanto, la *goal* **surefire:test** asociada por defecto a la fase test, será la encargada de ejecutar nuestros tests JUnit.


Bitbucket

El trabajo de esta sesión también debes subirlo a *Bitbucket*. Todo el trabajo de esta práctica deberá estar en el directorio **P02-Drivers**, dentro de tu espacio de trabajo.

IntelliJ (plugin Maven + *Configurations*)

Vamos a instalar un plugin para poder visualizar de forma gráfica los resultados de los tests ejecutados a través de Maven.

Para ello ve a las Preferencias de IntelliJ (**File→Settings→Plugins**), y desde la pestaña **Marketplace** busca el texto "Maven test", en el primer lugar de la lista debes ver el plugin **Maven test Support plugin**. Pulsa sobre el botón "Install", y posteriormente, desde la pestaña **Installed** tendrás que marcar la casilla que verás al lado del plugin para activarlo.

Nos aparecerá un nuevo icono en la ventana Maven: . Al pulsarlo, DESPUÉS de lanzar la ejecución de los test, nos aparecerán en diferente color dependiendo del resultado del informe JUnit.

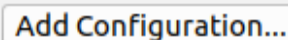
Recuerda que este plugin NO ejecuta los tests, solo lee el informe y lo visualiza.

En esta práctica necesitaremos modificar el valor de alguna variable al invocar al comando maven. En lugar "teclear" el comando cada vez, podemos crearnos desde IntelliJ un "botón" diferente para cada

construcción diferente que deseemos ejecutar. De esta forma aprovechamos las ventajas del IDE pero sin depender de él para construir el proyecto.

Vamos a usar las **Configurations** de IntelliJ.

Para crearnos una nueva *Configuration*, podemos hacerlo desde "**Run→Edit Configurations...**", o accediendo directamente a ellas desde la barra de herramientas:



En la siguiente ventana seleccionamos **Maven**, y a la derecha **Add new run configuration...**

Para cada **nueva Configuration** que crees:

- Asigne un **nombre** (puede ser cualquiera, con o sin espacios)
- **Guarda** la *configuration* en una subcarpeta de *src/resources* de nuestro proyecto Maven (si no lo hacemos así se guardará en el directorio *.idea*, y perderemos todas las configuraciones cuando subamos nuestro trabajo a Bitbucket). Simplemente selecciona "**Store as project file**" e indica la ruta de directorios de tu carpeta *src/resources/runConfigurationsIntelliJ*.
- Indica los argumentos del comando maven que queremos ejecutar (**0J0**: NO hay que incluir el comando *mvn*)
- Pulsamos sobre OK, y en la ventana Maven podremos acceder a cualquier *Configuration* que hayamos creado dentro de "**Run Configurations**" y lanzar la construcción desde aquí. De forma alternativa, también podremos hacerlo desde el desplegable que veremos en la parte superior de la ventana principal de IntelliJ

Ejercicios

Vamos a crear un proyecto Maven, que contendrá todos los ejercicios de esta sesión. Para ello elegiremos la opción "**File→New Project**". Seleccionaremos "**Maven**" en la parte izquierda de la siguiente pantalla, nos aseguraremos de que tiene asignado el JDK_11 y seleccionamos "**Next**".

Primero indicaremos los valores para las **coordenadas** de nuestro proyecto, que serán "ppss" (para el GroupId), y "drivers" (para el ArtifactId). Usaremos la versión que aparece por defecto.

El campo de texto "**Location**" debe contener la ruta del directorio raíz de nuestro proyecto Maven. Por lo tanto asegúrate de que dicha ruta está dentro de tu directorio de trabajo, y en el subdirectorio *P02-Drivers/drivers*. Dado que la carpeta "drivers" no existe todavía, IntelliJ nos pedirá permiso para crearla (obviamente, le diremos que sí). La carpeta "drivers" creada será la carpeta raíz de tu proyecto Maven. Verás que el campo de texto "**Name**" tiene también el valor "drivers"

Pulsamos sobre "**Finish**" y puedes comprobar que en el subdirectorio *P02-Drivers/drivers* se ha creado un fichero **pom.xml**, el directorio *src*, y el directorio *.idea* (este último no tiene nada que ver con Maven, lo crea IntelliJ automáticamente en todos sus proyectos).

FICHERO POM.XML

Necesitamos modificar el fichero *pom.xml*. Para ello:

- Añade la propiedad *project.build.sourceEncoding*, con el valor *UTF-8*.
- Las propiedades *maven.compiler.source* y *maven.compiler.target*, equivalen a la propiedad *maven.compiler.release* que ya hemos usado en la práctica anterior. Puedes sustituirlas por *maven.compiler.release*, o dejar las dos que aparecen por defecto.
- Necesitamos incluir el plugin *maven-compiler-plugin* con la versión 3.9.0.
- También incluiremos el plugin *surefire* ("*maven-surefire-plugin*"), versión 3.0.0-M5.
- Puedes añadir también ahora la librería de *junit5* en la sección de dependencias, versión 5.8.2

En las siguientes prácticas iremos añadiendo más elementos a partir de esta configuración básica, por lo que deberás recordarla.

⇒ Ejercicio 1: *drivers* para *reservaButacas()*

Queremos automatizar la ejecución de los siguientes casos de prueba (Tabla A) asociados al método **ppss.P02.Cine.reservaButacas()**.

Tabla A.

	Datos de entrada		Resultado esperado	
	asientos[]	solicitados	boolean	asientos[]
C1	[]	3	false	[]
C2	[]	0	false	[]
C3	[false, false, false, true, true]	2	true	[true, true, false, true, true]
C4	[true, true, true]	1	false	[true, true, true]

Los casos de prueba de la Tabla A se han obtenido aplicando el método del camino básico, por lo que dicha tabla es efectiva y eficiente. Deberías tener claro lo que eso significa.

Nota: Recuerda que podemos obtener conjuntos de prueba alternativos usando el mismo método, y que, si bien pueden tener una cardinalidad diferente, se consideran igualmente válidos (siempre y cuando no superen el valor de CC, y se obtengan a partir de un conjunto de caminos independientes).

Por ejemplo, la Tabla B nos permite conseguir el mismo objetivo que la Tabla A

Tabla B.

	Datos de entrada		Resultado esperado	
	asientos[]	solicitados	boolean	asientos[]
C1	[false, false, false]	1	true	[true, false, false]
C2	[true]	1	false	[true]

En la carpeta **Plantillas-P02** se proporciona el código de la clase **Cine**, con la implementación del método *reservaButacas()*. Dicho método será nuestra UNIDAD a probar, y la denominaremos SUT.

A partir de aquí se pide:

A) **Implementa los drivers** de pruebas unitarias dinámicas, usando el conjunto de comportamientos a probar identificados en la **tabla A**. No uses tests parametrizados.

Debes respetar las indicaciones que hemos explicado en clase respecto a la forma correcta de implementar dichos drivers con JUnit, ubicar correctamente cada uno de los fuentes, y asegurarte de que el pom contiene la librería para poder compilar nuestros tests a través de Maven.

Nota1: puedes generar la clase de los drivers de forma automática con IntelliJ seleccionando el nombre de la clase de tu SUT, y con botón derecho seleccionamos **Generate...→ Test...** Tendrás que marcar el método que vas anotar con @Test.

Nota 2: los drivers se llamarán *reservaButacasC1()*, *reservaButacasC2()*,... y así sucesivamente.

B) **Ejecuta los drivers** que has implementado usando Maven. Si detectas algún error tendrás que depurar el código. Fíjate que si tienes que modificar y/o añadir líneas cambiarás el conjunto de comportamientos implementados y probablemente los caminos independientes que has usado para crear la tabla también lo harán. Y por lo tanto la tabla de casos de prueba dejará de ser efectiva y eficiente!! No vamos a pedirte que modifiques la tabla de casos de prueba, pero debes tener en cuenta que si usamos un método de caja blanca para diseñar nuestros casos de prueba, cambios en el código implican cambios en los tests, ya que éstos dependen totalmente de dicho código!!

La ventana "Run" de IntelliJ muestra la secuencia de *goals* ejecutadas por maven. Seleccionando la *goal* "test" verás el informe JUnit en formato texto.

Crea una **nueva Configuration** con nombre **"Run CineTest"**, con el comando maven **"mvn test -Dtest=CineTest"** (la variable test pertenece al plugin *surefire* e indica la clase (o clases) de los drivers que queremos ejecutar).

🔗 Ejercicio 2: *drivers* para *contarCaracteres()*

Queremos automatizar la ejecución de los siguientes casos de prueba (Tabla C) asociados al método *ppss.P02.FicheroTexto.contarCaracteres()*.

Tabla C.

	Datos de entrada			Resultado esperado
	nombreFichero	[read(),...read()]	close()	int o FicheroException
C1	ficheroC1.txt	--	--	FicheroException con mensaje "ficheroC1.txt (No existe el fichero o directorio)"
C2	src/test/resources/ficheroCorrecto.txt	{a,b,c,-1}	No se lanza excepción	3
C3	src/test/resources/ficheroC3.txt	{a,b,IOException}	--	FicheroException con mensaje "ficheroC3.txt (Error al leer el archivo)"
C4	src/test/resources/ficheroC4.txt	{a,b,c,-1}	IOException	FicheroException con mensaje "ficheroC4.txt (Error al cerrar el archivo)"

ASUMIMOS LO SIGUIENTE:

ficheroC1.txt no existe

ficheroCorrecto.txt contiene los caracteres **abc**

ficheroC3.txt contiene los caracteres **abcd**

ficheroC4.txt contiene los caracteres **abc**

Nota: el valor "--" indica que no procede esa entrada


En la carpeta **Plantillas-P02** se proporciona el código de la clase **FicheroTexto**, con la implementación de la unidad a probar: método *contarCaracteres()*. También se proporcionan la clase **FicheroException** y el fichero de texto *ficheroCorrecto.txt* que tendrás que usar cuando ejecutes tus pruebas.

Se pide:

A) **Implementa los drivers** de pruebas unitarias dinámicas, usando el conjunto de comportamientos a probar identificados en la tabla C. No uses tests parametrizados.

Nota: sólo vas a saber implementar los tests C1 y C2. Explicaremos como implementar C3 y C4 mas adelante. Deja los tests C3 y C4 únicamente con una sentencia *Assertions.fail()* y etiquétalos como **"excluido"**

Nota: los drivers se llamarán *contarCaracteresC1()*, y *contarCaracteresC2()*

B) **Ejecuta los drivers** sólo de los tests C1 y C2 cambiando la configuración del plugin surefire desde línea de comandos. (desde IntelliJ puedes hacerlo de dos formas: desde la ventana **Terminal** (**View→Tool Windows→Terminal**), o desde la ventana de Maven, pulsando sobre el icono ). Posteriormente, crea un nuevo **Configuration** con el nombre **"FicheroTextoTest sin excluidos"**. Recuerda que el plugin *surefire* permite ejecutar sólo los drivers de una determinada clase (o conjunto de clases) usando la variable **test** (`-Dtest=<clase1>,<clase2>,...`)

⇒ Ejercicio 3: Drivers para *delete()*

En el directorio **Plantillas-P02** encontrarás el fichero **DataArray.java** con una implementación para el método **ppss.P02.DataArray.delete(int)**. También necesitarás la clase *DataException* proporcionada.

La clase **DataArray** representa la tupla formada por una colección de datos enteros (hasta un máximo de 10) con valores mayores que cero, y el número de elementos almacenados actualmente en la colección. Los elementos ocupan siempre posiciones contiguas, y la primera posición será la 0. Las posiciones no ocupadas siempre tendrán el valor cero. La colección puede contener valores repetidos.

La especificación del método es la siguiente:

El método **delete(int)** borra el primer elemento de la colección cuyo valor coincida con el entero especificado como parámetro. El método lanzará una excepción de tipo *DataException* con un determinado mensaje, en los siguientes casos: cuando el elemento a borrar sea ≤ 0 (mensaje: "El valor a borrar debe ser $>$ cero"), cuando la colección esté vacía (mensaje: "No hay elementos en la colección"), cuando el elemento a borrar sea ≤ 0 y además la colección esté vacía (mensaje: "Colección vacía. Y el valor a borrar debe ser $>$ cero"), y cuando el elemento a borrar no se encuentre en la colección (mensaje: "Elemento no encontrado").

Se proporciona la siguiente tabla de casos de prueba:

Tabla D

	Datos de entrada		Resultado esperado
ID	DataArray (colección, numElem)	Elemento a borrar	(colección + numElem) o excepción de tipo DataException
C1	([1,3,5,7], 4)	5	[1,3,7], 3
C2	([1,3,3,5,7], 5)	3	[1,3,5,7], 4
C3	([1,2,3,4,5,6,7,8,9,10], 10)	4	[1,2,3,5,6,7,8,9,10], 9
C4	([], 0)	8	DataException(m1)
C5	([1,3,5,7], 4)	-5	DataException(m2)
C6	([], 0)	0	DataException(m3)
C7	([1,3,5,7], 4)	8	DataException(m4)

m1: "No hay elementos en la colección"

m2: "El valor a borrar debe ser $>$ 0"

m3: "Colección vacía. Y el valor a borrar debe ser $>$ 0"

m4: "Elemento no encontrado"

Dada la implementación anterior del método *delete()*, se pide:

A) **Implementa los drivers** de pruebas unitarias dinámicas, usando el conjunto de comportamientos a probar identificados en la tabla C. No uses tests parametrizados.

B) **Ejecuta los drivers** (de esta tabla). Deberás crear un nuevo elemento **Configuration** con el nombre **"Run DataArrayTest"**

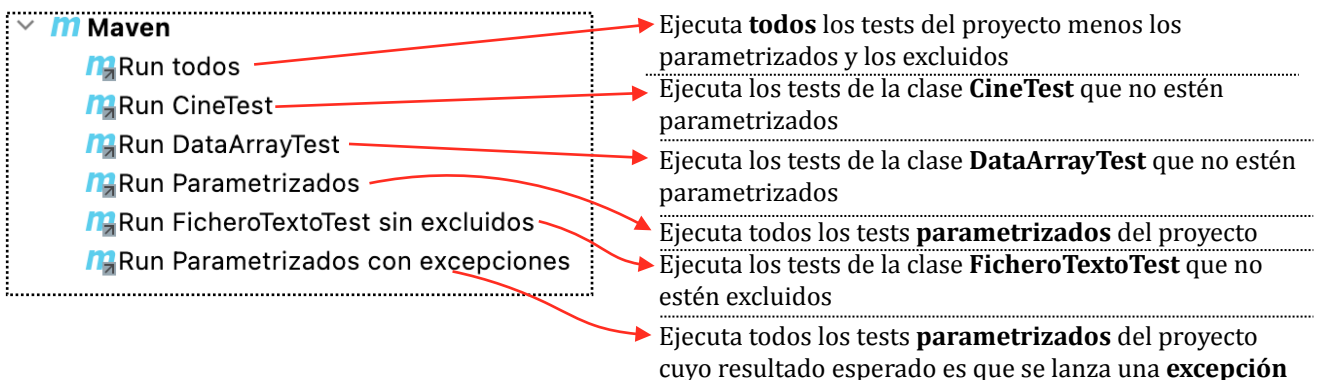
Recuerda que el plugin *surefire* permite ejecutar sólo los drivers de una determinada clase (o conjunto de clases) usando la variable **test** (`-Dtest=<clase1>,<clase2>,...`)

⇒ Ejercicio 4: *drivers* parametrizados

- A) Implementa un **test parametrizado** *reservaButacasC5* para la tabla A del ejercicio1.
Debes implementar el nuevo driver en la clase **CineTest** y etiquetarlo como "**parametrizado**"
- Nota:** Ejecutamos los tests a través de la goal *surefire:test*, y ésta, por defecto, no tiene en cuenta el atributo "name" opcional de la anotación *@ParameterizedTest*. Tendríamos que alterar la configuración del plugin (ver Anexo 2, si queréis probarlo).
- B) **Modifica** la *Configuration* "**Run CineTest**" para que no ejecute el test parametrizado.
- C) Implementa un **test parametrizado** con el nombre *testParametrizadoC8* para los tests C4..C7 de la Tabla C.
Debes implementar el nuevo driver en la clase **DataArrayTest** y etiquetarlo con dos etiquetas: "**parametrizado**" y "**conExcepciones**"
- D) Implementa un **test parametrizado** con el nombre *testParametrizadoC9* para los tests C1..C3 de la Tabla D.
Debes implementar el nuevo driver en la clase **DataArrayTest** y etiquetarlo con la etiqueta "**parametrizado**"
- E) **Crea** una nueva *Configuration* con nombre "**Run parametrizados**" para ejecutar todos los tests parametrizados de nuestro proyecto.
- F) **Crea** una nueva *Configuration* con nombre "**Run parametrizados con excepciones**" para ejecutar todos los tests parametrizados de nuestro proyecto etiquetados con "**conExcepciones**".
Hemos visto que `-Dgroups=etiqueta1,etiqueta2` significa que filtramos la ejecución de los tests ejecutando aquellos etiquetados con etiqueta1 o etiqueta 2.
De forma alternativa, JUnit permite usar etiquetas con expresiones booleanas (operadores "&" (and) "|" (or), "!" (not)). Incluso podemos usar paréntesis para ajustar la precedencia del operador. Por ejemplo, podríamos indicar como valor de la variable `groups`: "(firefox | chrome) & (testRapidos | testsMuyRapidos)" (<https://junit.org/junit5/docs/current/user-guide/#running-tests-tag-expressions>)
- G) **Crea** una nueva *Configuration* con nombre "**Run todos**" para ejecutar todos los tests del proyecto excepto los parametrizados y excluidos.
- H) Adicionalmente crea un fichero de texto **configurations-intellij.txt**, indicando, para cada *configuration*, el comando maven asociado. Este fichero deberá estar en *src/main/resources*

⇒ ANEXO 1: Lista de elementos *Configuration* creados

Mostramos todos los elementos **Configuration** que tenéis que crear en esta práctica.



Recuerda que todos ellos debes guardarlos en *src/resources/runConfigurationsIntellij* (si no lo haces así, los perderás al subir tu trabajo a Bitbucket).

➡ ➡ ANEXO 2: Configuración del plugin *surefire* (opcional)

Por defecto, el plugin *surefire*, no tendrá en cuenta el atributo *name* de la anotación *@ParameterizedTest*. Por lo tanto, cuando ejecute cada test, mostrará siempre el mismo nombre (ya que estamos ejecutando el mismo método varias veces)

Este atributo es interesante porque nos permite visualizar un nombre diferente para cada test ejecutado.

Si queréis usar el atributo "name" para ver cómo funciona, simplemente tendréis que cambiar la configuración del plugin *surefire* de la siguiente forma:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>3.0.0-M5</version>
  <configuration>
    <statelessTestsetReporter
      implementation="org.apache.maven.plugin.surefire.extensions.junit5.JUnit5Xml30StatelessReporter">
      <usePhrasedTestCaseClassName>true</usePhrasedTestCaseClassName>
      <usePhrasedTestCaseMethodName>true</usePhrasedTestCaseMethodName>
    </statelessTestsetReporter>
  </configuration>
</plugin>
```

Esta configuración también es necesaria si queremos usar la anotación *@DisplayName*, que nos permitirá mostrar el nombre especificado, en lugar del nombre real del método ejecutado <https://junit.org/junit5/docs/current/user-guide/#writing-tests-display-names>

Podréis ver el atributo "name" y el texto de la anotación *@Display* en el informe de pruebas a través del plugin que hemos instalado en IntelliJ (SIEMPRE después de ejecutar el comando maven correspondiente, ya que este plugin no ejecuta los tests, simplemente consulta el informe de tests y lo muestra por pantalla.

➡ ➡ ANEXO 3: Tabla de casos de prueba ejercicio "realizaReserva()"

En esta sesión no implementaremos (todavía) drivers para la tabla diseñada en el ejercicio 3 de la práctica anterior. Aunque implementaremos los drivers más adelante, dicha tabla es la siguiente:

	login	password	ident. socio	isbn	{reserva()}	Resultado esperado
C1	"xxx"	"xxx"	"Luis"	{"1111"}	--	??
C2	"ppss"	"ppss"	"Luis"	{"1111", "2222"}	{NoExcep, NoExcep.}	No se lanza excep.
C3	"ppss"	"ppss"	"Luis"	{"3333"}	{IsbnEx}	ReservaException1
C4	"ppss"	"ppss"	"Pepe"	{"1111"}	{SocioEx}	ReservaException2
C5	"ppss"	"ppss"	"Luis"	{"1111"}	{JDBCEx}	ReservaException3

Suponemos que el login/password del bibliotecario es "ppss"/"ppss"; que "Luis" es un socio y "Pepe" no lo es; y que los isbn registrados en la base de datos son "1111", "2222".

-- significa que no se invoca al método reserva()

NoExcep. → El método reserva no lanza ninguna excepción

IsbnEx → Excepción *IsbnInvalidoException*

SocioEx → Excepción *SocioInvalidoException*

JDBCEx → Excepción *JDBCException*

ReservaException1: Excepción de tipo *ReservaException* con el mensaje: "ISBN invalido:3333; "

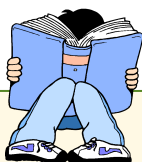
ReservaException2: Excepción de tipo *ReservaException* con el mensaje: "SOCIO invalido; "

ReservaException3: Excepción de tipo *ReservaException* con el mensaje: "CONEXION invalida; "

➡ ➡ ANEXO 4: Observaciones a tener en cuenta sobre la práctica P01B

- Nunca puede haber sentencias en el código que NO estén representadas en algún nodo
- Un nodo solo puede contener sentencias secuenciales, y NUNCA puede contener más de una condición
- El grafo tendrá un único nodo inicio y un único nodo final
- Todas las sentencias de control tienen un inicio y un final, debes representarlos en el grafo.
- Todos los caminos independientes obtenidos tienen que ser posibles de recorrer con algún dato de entrada,
- Cada caso de prueba debe recorrer "exactamente" todos los nodos y aristas de cada camino independiente.
- No podemos obtener más casos de prueba que caminos independientes hayamos obtenido
- Todos los datos de entrada deben de ser concretos
- Posiblemente necesitaremos hacer asunciones sobre los datos de entrada (como en la tabla del ejercicio 2)

Resumen



¿Qué **conceptos** y **cuestiones** me deben quedar CLAROS después de hacer la práctica?



IMPLEMENTACIÓN DE LOS TESTS

- Es necesario haber diseñado previamente los casos de prueba para poder implementar los drivers.
- El código de los drivers estará en `src/test/java`, en el mismo paquete que el código a probar. Nuestra SUT será una unidad, por lo tanto, estaremos automatizando pruebas unitarias, usando técnicas dinámicas. Tendremos UN driver para CADA caso de prueba.
- Debemos cuidar la implementación de los tests, para no introducir errores en los mismos.
- Para compilar los drivers "dependemos" de la librería JUnit 5, que habrá que incluir en el pom. Antes de compilar el código de los drivers, es imprescindible que se hayan generado con éxito los `.class` del código a probar (de `src/main/java`). Es decir, nunca haremos pruebas sobre un código que no compile.
- Debes usar correctamente tanto las anotaciones Junit (@) como las sentencias explicadas en clase (Assertions)

EJECUCIÓN DE LOS TESTS

- La ejecución de los tests se integra en el proceso de construcción del sistema, a través de MAVEN, (es muy importante tener claro que "acciones" deben llevarse a cabo y en qué orden).. La goal `surefire:test` se encargará de invocar a la librería JUnit en la fase "test" para ejecutar los drivers.
- Podemos ser "selectivos" a la hora de ejecutar los drivers, aprovechando la capacidad de Junit5 de etiquetar los tests..
- En cualquier caso, el resultado de la ejecución de los tests siempre será un informe que ponga de manifiesto las discrepancias entre el resultado esperado (especificado), y el real (implementado). Junit nos proporciona un informe con 3 valores posibles para cada test: pass, fault y error.
- Si durante la ejecución de los tests, alguno de ellos falla, el proceso de construcción se DETIENE y termina con un BUILD FAILURE.
- Debes tener claro que comandos Maven debemos usar para integrar la automatización de las pruebas en el proceso de construcción del proyecto y qué ficheros genera nuestro proceso de construcción en cada caso.