

# P03- Automatización de pruebas: Drivers

En esta sesión vamos a implementar y ejecutar drivers con JUnit5:

- [Implementación de drivers](#) con JUnit5
- [Ejecución de drivers](#) con Maven
- Directorio de trabajo, [plantillas](#) y GitHub
- [Configuración del pom.xml](#) para automatizar las pruebas
- IntelliJ y construcción del proyecto ([ventana Maven](#))
- [Ejercicios](#)
  - [Ejercicio 1](#): drivers para `reservaButacas()`
  - [Ejercicio 2](#): drivers para `contarCaracteres()`
  - [Ejercicio 3](#): Drivers para `delete()`
  - [Ejercicio 4](#): drivers parametrizados
- [ANEXO 1](#): Lista de elementos *Run Configuration* creados
- [ANEXO 2](#): Tabla de casos de prueba ejercicio "realizaReserva()"
- [ANEXO 3](#): Observaciones a tener en cuenta sobre la práctica P02
- [Resumen](#)

## Implementación de drivers con JUnit 5



En esta práctica automatizaremos las pruebas unitarias teniendo en cuenta los casos de prueba que hemos diseñado en la sesión anterior. **Implementaremos** los drivers y los **ejecutaremos**. 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 como 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 estarán físicamente separados de las unidades a las que prueban, pero pertenecerán 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...

También usaremos JUnit 5 para **ejecutar** los tests, pero esto lo haremos a través del plugin surefire de maven.

## Ejecución de drivers con Maven



La ejecución de los drivers (tests Junit) 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`. La *goal* **surefire:test** asociada por defecto a la fase test, será la encargada de ejecutar nuestros tests JUnit.

Podemos ejecutar directamente el/los comandos maven que correspondan desde un **terminal**, o bien podemos usar las **Run Configurations** desde IntelliJ (recuerda que hemos explicado cómo crear una *Run Configuration* en el documento *P01-Pruebas\_del\_Software*).

## Directorio de trabajo, plantillas y GitHub

Todo el trabajo de esta práctica deberá estar en el directorio **P03-Drivers**, dentro de tu espacio de trabajo.

El directorio de plantillas: **Plantillas-P03** **NO** debes copiarlo en tu directorio de trabajo. Contiene ficheros java y xml que necesitas usar en el proyecto maven que debes crear para esta práctica. Dicho proyecto maven se llamará **drivers**

El trabajo de esta sesión también debes subirlo a *GitHub*.

## Configuración del pom.xml para automatizar las pruebas

El fichero pom.xml nos permite configurar el proceso de construcción (secuencia de acciones a ejecutar).

Dado que vamos a importar clases JUnit5 en nuestro código fuente para **implementar** los tests, necesitamos incluir el/los ficheros.jar (librerías) que contienen las clases que usemos en nuestro código. En clase de teoría hemos visto qué LIBRERÍAS necesitamos, las cuales estarán en la sección de DEPENDENCIAS del pom. En concreto, se trata del artefacto:

```
org.junit.jupiter:junit-jupiter:5.11.4
```

Para **ejecutar** los tests unitarios, usaremos el PLUGIN [surefire](#). La goal surefire:test se encargará de ejecutarlos usando JUnit5, y generará los informes de pruebas, tal y como hemos explicado en clase:

```
org.apache.maven.plugins:maven-surefire-plugin:3.5.2
```

El plugin surefire, como ya hemos indicado en clase, por defecto ignora ciertas anotaciones JUnit5, como por ejemplo @DisplayName. Además, el informe generado por el plugin solo muestra el número de tests *success*, *failure* o *error*.

Vamos a configurar el plugin para mostrar los resultados de los tests en forma de árbol (con un formato muy similar al mostrado por IntelliJ, y la herramienta [ConsoleLauncher](#) de Junit5). Para ello tenemos que configurar el plugin surefire incluyendo la dependencia:

```
me.fabriziorby:maven-surefire-junit5-tree-reporter:1.4.0
```

Con esta configuración sí se tienen en cuenta anotaciones como @DisplayName. Te recomendamos que no hagas copy/paste de las transparencias. Es mucho mejor acceder al sitio web de la librería ([maven-surefire-junit5-tree-reporter](#)) Cuidado!! la versión de surefire que tienes que usar es la 3.5.2.

Para configurar el plugin debes tener en cuenta que:

- queremos mostrar el informe usando <theme> con el valor ASCII (que es el valor por defecto, por lo que no será necesario indicarlo. En clase hemos puesto el ejemplo con el valor UNICODE)
- queremos configurar los detalles mostrados en el informe sobre los *failures* detectados (etiquetas <print...>, de forma que
  - ❖ se muestre la traza de la **pila de excepciones** (tanto si se trata de un error, o un failure)
  - ❖ se muestren los mensajes por la **salida estándar** (solo si resultado es error o failure)
  - ❖ se muestren los mensajes por la **salida de error** (solo si el resultado es error)

Adicionalmente necesitamos usar el **plugin compiler**:

```
org.apache.maven.plugins:maven-compiler-plugin:3.13
```

para compilar los fuentes del proyecto y para compilar los fuentes de los tests. Ya deberías saber qué goals vamos a usar de este plugin, por qué no es necesario configurar en el pom las <execution> de las goals correspondientes, y qué pasaría si no incluyésemos este plugin en el pom.

Recuerda que también debes incluir los valores de las **propiedades** para indicar que queremos usar la versión 21 de java para codificar y para generar los ejecutables, y que la codificación del texto será UTF-8

## IntelliJ y construcción del proyecto (ventana Maven)



Vamos a delegar todo el trabajo de construcción en maven, concretamente el que tenemos instalado en /usr/local

Para ello hemos marcado la casilla **Delegate IDE build/run actions to Maven** (puedes acceder directamente a dicha casilla desde la opción *Maven Settings* (icono con forma de rueda dentada) desde la ventana de Maven, y desde ahí seleccionamos el elemento **Build,Execution,Deployment→Build Tools→Maven→Runner**

### IMPORTANTE:

De forma alternativa, podemos dejar desmarcada la casilla **Delegate IDE build/run actions to Maven**.

En este caso estamos cediendo el control de la construcción a IntelliJ (o lo que es lo mismo, permitimos que no todo se haga a través de Maven). Por lo tanto, si cambiamos de IDE, las cosas puede que no se hagan de la misma manera.

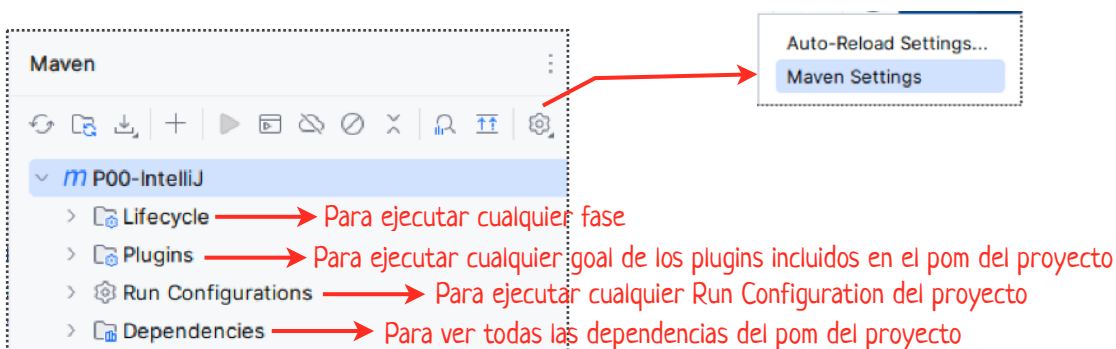
La idea es no depender totalmente de IntelliJ para construir el proyecto, de forma que cualquier construcción que hagamos del proyecto, SIEMPRE podremos hacerla exactamente igual desde un terminal, y todo funcionará de la misma forma independientemente del IDE que estemos usando.

Ahora bien, ya que estamos usando uno de los mejores IDE para trabajar con java y maven, puede ser bastante útil dejar desmarcada la casilla mencionada (temporalmente) DURANTE la implementación de los tests, de forma que podamos, en un momento dado ejecutar un único test o sólo los tests de una clase particular simplemente pulsando sobre los iconos con un triángulo verde que verás en el editor justo a la izquierda de cada uno de los tests y del nombre de la clase.

Esto también lo podemos hacer sólo con maven, pero tendremos que crearnos una run configuration para hacerlo. Si queremos ejecutar individualmente cada uno de los tests a la vez que los estamos implementando, hacer una run configuration para ello cada vez no es práctico.

Recuerda que, si bien puedes dejar desmarcada esta casilla. DEBES aprender y acostumbrarte a hacer todo a través de maven (marcando la casilla), ya que así te lo preguntaremos en el examen.

Desde la ventana **Maven** de IntelliJ (ver **Figura 1**) podemos construir el proyecto así como consultar los detalles de los elementos contenidos en nuestro pom (librerías, plugins). También podemos ejecutar cualquier Run configuration y ver qué comando maven tiene asociado.



**Figura 1.** Ventana Maven

Aunque no lo mostramos en la Figura 1, desde la ventana Maven también podremos ver cual es el "effective-pom" usado por nuestro proyecto Maven, y ejecutar cualquier goal de nuestro pom.

## Ejercicios



Vamos a **crear un proyecto Maven**, que contendrá todos los ejercicios de esta sesión, usando los siguientes datos:

- **Name:** `drivers`
- **Location:** carpeta P03-IntelliJ
- **JDK:** nos aseguraremos de que esté seleccionado JDK 21
- **groupId:** `ppss.practica3`

Una vez creado el proyecto, recuerda que debes BORRAR **SIEMPRE** el fichero `.gitignore` que ha generado IntelliJ.

Todos los ejercicios de esta sesión requieren la misma [configuración del pom](#),

**Recuerda** que es importante que tengas claro para qué sirve cada una de las secciones del pom y por qué usamos cada librería y plugin en el pom. Para que podamos controlar el proceso de construcción del proyecto es imprescindible que tengamos claro qué goals necesitamos ejecutar para poder automatizar la ejecución de nuestras pruebas.

### ➡ ➡ Ejercicio 1: *drivers* para *reservaButacas()*



Queremos automatizar la ejecución de los siguientes casos de prueba (Tabla A) asociados al método `ppss.Cine.reservaButacas()`, cuyo código está en el fichero `Cine.java` del directorio de plantillas proporcionado (**Plantillas-P03**)

**Tabla A.**

	Datos de entrada		Resultado esperado	
	asientos[]	solicitados	(boolean + asientos[]) o ReservaException	
			boolean	asientos[]
C1	[ ]	3	ButacasException con mensaje "No se puede procesar la solicitud"	
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[]) o ReservaException	
			boolean	asientos[]
C1	[false, false, false]	1	true	[true, false, false]
C2	[true]	1	false	[true]

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 tener en cuenta las normas que hemos explicado en clase y ubicar correctamente cada uno de los fuentes e implementar dichos drivers con JUnit 5 correctamente. Recuerda también que debes evitar **duplicar código** siempre que sea posible.

**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 deben tener nombres que indiquen qué comportamiento estamos ejercitando. Usa los nombres:

- C1\_reservaButacas\_should\_return\_Exception\_when\_fila\_empty\_and\_want\_3
- C2\_reservaButacas\_should\_return\_false\_when\_fila\_empty\_and\_want\_zero
- C3\_reservaButacas\_should\_return\_true\_when\_fila\_has\_3\_seats\_free\_and\_want\_2
- C4\_reservaButacas\_should\_return\_false\_when\_no\_free\_seats\_and\_want\_1

**Nota 3:** Asegúrate de que IntelliJ no te incluye un import incorrecto, y sobre todo, que no te añade nada en tu pom.xml.

- B) **Ejecuta los drivers** que has implementado usando Maven. Para ello debes crear una **nueva Run Configuration** con nombre **Run\_CineTest**, y asociarle el comando maven:

► mvn clean test -Dtest=CineTest

La propiedad **test** pertenece al plugin **surefire** e indica la clase (o clases) de los drivers que queremos ejecutar).

TODAS las *Run Congurations* maven siempre las guardaremos en la carpeta **intellij-configurations**, en la raíz de tu proyecto maven. Tendrás que crear dicha carpeta.

Si detectas algún *failure* 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"** en dicha ventana verás el informe JUnit en forma de árbol.

Incluye la anotación **@DisplayName("Tests asociados a la clase Cine")** a la clase que contiene los drivers y observa la diferencia entre usarla y no usarla.

- C) **Informe de pruebas.** Durante la construcción del proyecto, Maven nos muestra el informe de pruebas. En clase hemos explicado que diferentes herramientas visualizan los informes de pruebas de forma diferente (en forma de texto, con colores, con diferentes iconos, ...), pero la información de todos ellos es la MISMA, y se obtiene a partir del informe que proporciona JUnit. Ya debes saber dónde encontrar dicho informe. Abre el fichero **txt** correspondiente para ver el contenido de dicho informe.

Adicionalmente, Maven puede generar informes de pruebas en formato **html**. Cambia el nombre de la *Run Configuration* **Run\_CineTest**, a **Run\_CineTest\_with\_html\_report**, y modifica el comando maven para que quede así:

► mvn clean jxr:test-jxr surefire-report:report -Dtest=CineTest

La goal **jxr:test-jxr** genera un html de los fuentes de los tests en el directorio **target/reports/xref-test** con líneas numeradas para poder referenciarlas con hiperenlaces desde el html generado por el plugin **surefire-report**.

La goal **surefire-report:report** ejecuta la fase **test** y a continuación genera un informe en formato **html** en el directorio **target/reports**

Después de ejecutar esta *Run\_Configuration*, abre en un navegador el fichero **target/reports/surefire.html** para ver qué contiene. Puedes hacerlo desde IntelliJ seleccionando el fichero y la opción **Open In → Browser → Firefox** (o el navegador que tengas instalado).

Verás que puedes navegar por los tests a nivel de paquete, y clase.

Se usan iconos diferentes con temática meteorológica para cada resultado de ejecución de los tests (por ejemplo un sol, una nube gris con un rayo, ...)

## ⇒ Ejercicio 2: *drivers* para *contarCaracteres()*



Queremos automatizar la ejecución de los siguientes casos de prueba (Tabla C) asociados al método *ppss.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 archivo o el 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-P03** 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 1:** 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 2:** los drivers deben tener nombres que indiquen qué comportamiento estamos ejercitando.

Usa los nombres:


- ▶ C1\_contarCaracteres\_should\_return\_Exception\_when\_file\_does\_not\_exist
- ▶ C2\_contarCaracteres\_should\_return\_3\_when\_file\_has\_3\_chars
- ▶ C3\_contarCaracteres\_should\_return\_Exception\_when\_file\_cannot\_be\_read
- ▶ C4\_contarCaracteres\_should\_return\_Exception\_when\_file\_cannot\_be\_closed

B) **Ejecuta los drivers** sólo de los tests C1 y C2 cambiando la configuración del plugin surefire desde línea de comandos.

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>,...` )

Para usar Maven mediante línea de comandos desde IntelliJ puedes hacerlo de dos formas:

– desde la ventana **Terminal** (**View**→**Tool Windows**→**Terminal** ), o

– desde la ventana **Maven**, pulsando sobre el icono 

Después de probar la construcción desde línea de comandos, crea una nueva **Run Configuration** con el nombre **Run\_FicheroTextoTest\_sin\_excluidos**



### 🔗 Ejercicio 3: Drivers para *delete()*



En el directorio **Plantillas-P03** encontrarás el fichero ***DataArray.java*** con una implementación para el método ***ppss.DataArray.delete(int)***. También necesitarás la clase *DataException* proporcionada.

La clase ***DataArray*** representa la tupla (c,n), siendo **c** una colección de datos enteros (hasta un máximo de 10) con valores mayores que cero, y **n** 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  $\leq 0$  (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  $\leq 0$  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 <i>DataException</i>
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	<i>DataException</i> (m1)
C5	( [1,3,5,7], 4 )	-5	<i>DataException</i> (m2)
C6	( [ ], 0 )	0	<i>DataException</i> (m3)
C7	( [1,3,5,7], 4 )	8	<i>DataException</i> (m4)

Asumimos que en todos los casos el array tiene 10 elementos. En la tabla se han omitido las últimas posiciones del array con valor 0.

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 proporcionada del método *delete()*, se pide:

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

**Nota 1:** los drivers deben tener un nombre con el formato **Cx\_nombreMetodo\_should\_yyy\_when\_zzz**, tal y como hemos hecho en el resto de ejercicios.

- B) **Ejecuta los drivers** (sólo los de esta tabla, y sin usar @Tag). Deberás crear un nuevo elemento **Run Configuration** con el nombre **Run\_DataArrayTest**

## 



- A) Implementa un **test parametrizado** con nombre `C5_reservaButacas` para la tabla A del ejercicio1.  
Debes implementar el nuevo driver en la clase `CineTest` y etiquetarlo como "**parametrizado**" (sólo podrás parametrizar 3 de los 4 drivers)

Usa la anotación `@DisplayName("reservaButacas_")`.

En el método `C5_reservaButacas`, debes añadir un parámetro adicional de tipo `String`. Dicho `String` se mostrará por pantalla cuando el resultado esperado no coincida con el real. Adicionalmente, usaremos dicho parámetro de tipo `String` en el atributo **name** de la anotación `@ParameterizedTest`. Tienes que usar dicho atributo para que, al ejecutar los tests, se muestre por pantalla lo siguiente:

- ▶ `reservaButacas_[1]` should be `false` when we want `0` and `fila has no seats`
- ▶ `reservaButacas_[2]` should be `true` when we want `2` and `there are 2 free seats`
- ▶ `reservaButacas_[3]` should be `false` when we want `1` and `all seats are already reserved`

**Nota:** Los valores `1`, `2` y `3` dependerán del orden de ejecución de los tests en tu máquina, y son los valores asociados a la variable `{index}`. Las partes del mensaje en azul se corresponden con los parámetros del test que se está ejecutando.

- B) **Modifica** la *Run Configuration* `Run_CineTest_with_html_report` para que NO ejecute el test parametrizado.
- C) Implementa un **test parametrizado** con el nombre `C8_deleteWithExceptions` para los tests C4..C7 de la Tabla D.  
Debes implementar el nuevo driver en la clase `DataArrayTest` y etiquetarlo con dos etiquetas: "**parametrizado**" y "**conExcepciones**"

Usa la anotación `@DisplayName("delete_With_Exceptions_")` y el atributo **name** de la anotación `@ParameterizedTest` para que al ejecutar los tests se muestre lo siguiente:

- ▶ `delete_With_Exceptions_[1]` Message exception should be `"No hay elementos en la colección"` when we want delete `8`
- ▶ `delete_With_Exceptions_[2]` Message exception should be `"El valor a borrar debe ser > 0"` when we want delete `-5`
- ▶ `delete_With_Exceptions_[3]` Message exception should be `"Colección vacía. Y el valor a borrar debe ser > 0"` when we want delete `0`
- ▶ `delete_With_Exceptions_[4]` Message exception should be `"Elemento no encontrado"` when we want delete `8`

**Nota:** Los valores `1`,...,`4` dependerán del orden de ejecución de los tests en tu máquina, y son los valores asociados a la variable `{index}`. Las partes del mensaje en azul se corresponden con los parámetros del test que se está ejecutando

- D) Implementa un **test parametrizado** con el nombre `C9_deleteWithoutExceptions` para los tests C1..C3 de la Tabla D.  
Debes implementar el nuevo driver en la clase `DataArrayTest` y etiquetarlo con la etiqueta "**parametrizado**"

Usa la anotación `@DisplayName("delete_Without_Exceptions_")` y añade un parámetro adicional de tipo `String` para que al ejecutar los tests se muestre lo siguiente (el valor de dicho parámetro adicional también se mostrará cuando el resultado esperado no coincida con el real):

- ▶ `delete_Without_Exceptions_[1]` should be `[1, 3, 7]` when we want delete `5`
- ▶ `delete_Without_Exceptions_[2]` should be `[1, 3, 5, 7]` when we want delete `3`
- ▶ `delete_Without_Exceptions_[3]` should be `[1,2,3,5,6,7,8,9,10]` when we want delete `4`

**Nota:** Los valores `1`,...,`3` dependerán del orden de ejecución de los tests en tu máquina, y son los valores asociados a la variable `{index}`. Las partes del mensaje en azul se corresponden con los parámetros del test que se está ejecutando

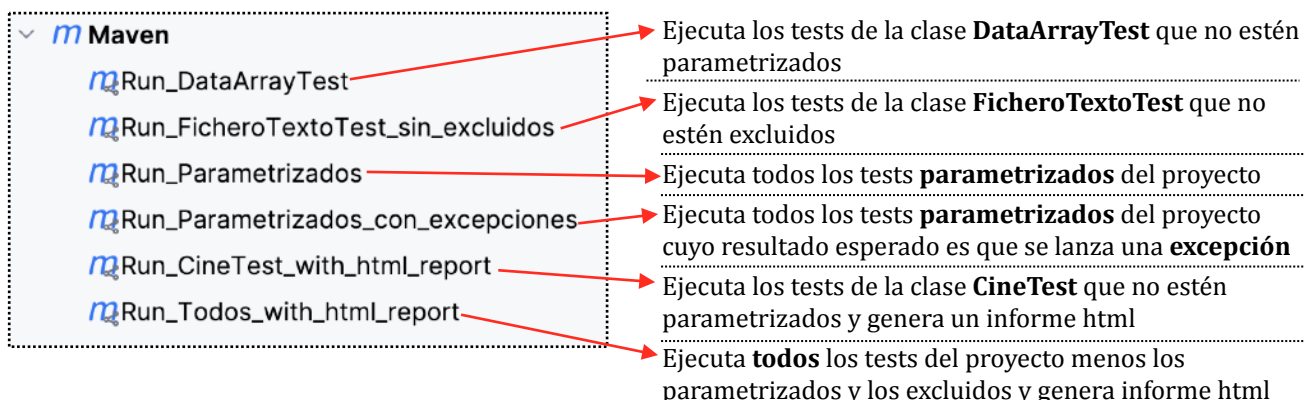


- E) Crea una nueva *Run Configuration* con nombre **Run\_parametrizados** para ejecutar todos los tests parametrizados de nuestro proyecto.
- F) Crea una nueva *Run Configuration* con nombre **Run\_parametrizados\_con\_excepciones** para ejecutar todos los tests parametrizados de nuestro proyecto etiquetados 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) & (testsRapidos | testsMuyRapidos)" (<https://junit.org/junit5/docs/current/user-guide/#running-tests-tag-expressions>)
- G) Crea una nueva *Run Configuration* con nombre **Run\_todos\_with\_html\_report** para ejecutar todos los tests del proyecto excepto los parametrizados y excluidos, y además genere un informe adicional en formato html
- H) Finalmente, copia todos los comandos maven de todas las *Run Configurations* que hemos creado en el fichero **comandosMaven.txt**, que encontrarás en el directorio de plantillas. Este fichero debe estar en la carpeta **P03-Drivers**.

## ➡ ➡ ANEXO 1: Lista de *Run Configuration* creados



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



Recuerda que todos ellos debes guardarlos en la carpeta **intellij-configurations** (al mismo nivel que el fichero pom.xml). De no hacerlo así, perderás las *Run configurations* al subir tu trabajo a GitHub.

## ➡ ➡ ANEXO 2: 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	lista isbn	{reserva()}	Resultado esperado
C1	"xxx"	"xxx"	"Luis"	{"11111"}	--	??
C2	"ppss"	"ppss"	"Luis"	{"11111", "22222"}	{NoExcep, NoExcep.}	No se lanza excep.
C3	"ppss"	"ppss"	"Luis"	{"33333"}	{IsbnEx}	ReservaException1
C4	"ppss"	"ppss"	"Pepe"	["11111"]	{SocioEx}	ReservaException2
C5	"ppss"	"ppss"	"Luis"	{"11111"}	{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 isbnns registrados en la base de datos son "11111", "22222".

**{reserva()}** → Representa la lista de salidas en sucesivas ejecuciones del método reserva()

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

**NoExcep.** → El método reserva no lanza ninguna excepción

**IsbnEx** → Excepción IsbnInvalidoException

**SocioEx** → Excepción SociolInvalidoException

**JDBCEx** → Excepción JDBCException

**ReservaException1:** Excepción de tipo ReservaException con el mensaje: "ISBN invalido:33333; "

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

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

### ➡ ➡ ANEXO 3: Observaciones a tener en cuenta sobre la práctica P02



- 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
- Si un método termina "normalmente" debe llegar a la sentencia return del final del método (o a la última sentencia si se trata de un método void). Si el método termina debido a que se lanza una excepción y no se captura, o se usa algún salto incondicional (break, continue, goto, un return que no es el del final del método...), entonces el return final del método (en caso de haberlo) puede que NO se ejecute.
- Todas las sentencias de control tienen un inicio y un final, debes representarlos en el grafo.
- Todos los caminos independientes obtenidos tienen que poder recorrerse con algún dato de entrada..
- Cada caso de prueba debe recorrer EXACTAMENTE todos los nodos y aristas de cada camino independiente. Por ejemplo, si los datos de entrada provocan varias iteraciones en un bucle entonces el camino asociado a esa entrada debe indicar exactamente esas iteraciones.
- No podemos obtener más casos de prueba que número de caminos independientes seleccionados (ni menos)
- Todos los datos de entrada deben de ser concretos, y el resultado esperado también. El resultado esperado siempre lo obtendremos de la especificación.
- Posiblemente necesitaremos hacer asunciones sobre los datos de entrada (como en las tablas de los ejercicios 2 y 3)
- Para poder hacer la tabla necesitamos identificar las ENTRADAS y SALIDAS del método a probar. Puede haber entradas "directas", que son los datos que entran "directamente" (por ejemplo, a través de los parámetros). Pero el método puede tener entradas "indirectas", que serán proporcionadas por la invocación a otra unidad (método java). Por ejemplo, en el ejercicio 3, el resultado de invocar a io.reserva() es una entrada indirecta. En el ejercicio 2, el resultado de invocar a close() es otra entrada indirecta. Hemos identificado un comportamiento a probar como una correspondencia entre datos de entrada y resultado esperado. Cada comportamiento asocia un resultado esperado a unos datos de entrada. Por lo tanto, cualquier dato que necesitemos conocer (sea un parámetro o no) para poder indicar el resultado esperado será una entrada.

## 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 (usamos técnicas dinámicas). Tendremos UN driver para CADA caso de prueba.
- Debemos cuidar la implementación de los tests, para no introducir errores en los mismos. Usaremos siempre el patrón AAA
- Es muy importante que el nombre cada test refleje lo que hace el test. Para ello usaremos el formato `<id>_<nombre_método>_should_<resultado_esperado>_when_<condiciones_sobre_las_entradas>`.
- 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)
- La compilación del código de los tests se lleva a cabo durante la fase `test-compile`, y se realiza DESPUÉS de compilar con éxito el código fuente de nuestro proyecto. Es decir, no compilaremos los fuentes de los tests si el código a probar tiene errores de compilación.

## EJECUCIÓN DE LOS TESTS

- La ejecución de los tests se integra en el proceso de construcción 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 durante 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 unitarios, 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 unitarias en el proceso de construcción del proyecto y qué ficheros genera nuestro proceso de construcción en cada caso.