P02- Automatización de pruebas: Drivers

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...

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. Por lo tanto, la *goal* ***surefire:test*** asociada por defecto a la fase test, será la encargada de ejecutar nuestros tests JUnit.

Directorio de trabajo, plantillas y Bitbucket

Todo el trabajo de esta práctica deberá estar en el directorio **P02-Drivers**, dentro de tu espacio de trabajo.

Para esta sesión proporcionamos el código fuente (ficheros java) de un proyecto maven que deberá llamarse drivers. Después de crear el proyecto tendrás que ubicar cada fichero en la carpeta correcta.

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

IntelliJ (*Maven test support plugin*)

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". Después asegúrate que, desde la pestaña ***Installed*** la casilla que parece a la derecha del plugin instalado está marcada (si no lo está, el plugin no se activará)

Nos aparecerá un nuevo icono en la ventana Maven (para verlo tienes que seleccionar el proyecto maven en dicha ventana). Al pulsarlo, DESPUÉS de lanzar la ejecución de los test, nos mostrará el informe JUnit con diferntes colores.

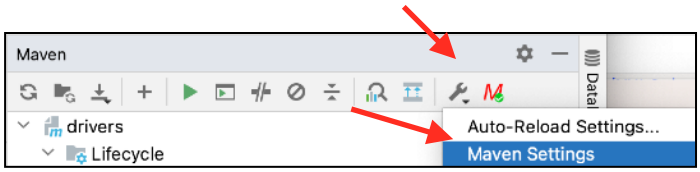
Esta nueva opción también aparece en el menú contextual del proyecto maven (desde la vista Project). Es la última opciñon del menú "***ShowTestResultsAction***".

Recuerda que este plugin **NO ejecuta los tests**, solo lee el informe generado de una construcción previa y lo visualiza.

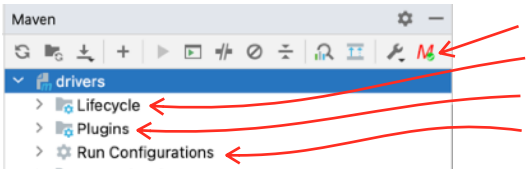
IntelliJ y construcción del proyecto (ventana Maven)

Recuerda que vamos a delegar todo el trabajo de construcción en maven, pero el externo, el que está instalado en /usr/local

Por ello, recuerda que debes macar la casilla ***Delegate IDE build/run actions to Maven*** (puedes acceder directamente con la opción Maven Setings desde la ventana de Maven, y desde ahí accedr al elemento "*Runner*"):



Para construir el proyecto usaremos la ventana Maven, desde aquí podremos:



- Ver los informes de pruebas de la última construcción
- Ejecutar cualquier fase
- Ejecutar cualquier goal
- Ejecutar cualquier Run Configuration

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 construdción a IntellJ (o lo que es lo mismo, permitimos que no todo se haga a través de Maven: 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 cuaquier 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 el proceso de implementar 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.

Ejercicios

Vamos a **crear un proyecto Maven**, que contendrá todos los ejercicios de esta sesión. Para ello elegiremos la opción "*New Project*". Y usaremos las siguientes opciones:

- ***Name***: drivers
- ***Location***: ppss-2023.../P02-Drivers/drivers (tendrás que crear la carpeta "drivers")
- ***Languaje***: Java
- ***Build system***: Maven
- ***Jdk***: 11
- Desmarcamos "Add sample code"
- Advanced settings:
  - **GroupId**: ppss
  - **ArtifactID**: 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:

- 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 dejar las dos que aparecen por defecto o sustituirlas por *maven.compiler.release*.
- Añade los ***plugins*** *maven-compiler-plugin (*v.ersión 3.10.1) y *maven-surefire-plugin*, (v. 3.0.0-M8)
- Añade la ***librería*** *junit-jupiter-engine*, v. 5.9.2, con el ***scope*** test

Puedes copiar los *plugins* y la *librería* de las plantillas de la práctica P01A o bien de las transparencias de la clase de teoría. En las siguientes prácticas iremos añadiendo más elementos a partir de esta configuración básica, por lo que deberás recordarla.

También deberías tener claro para qué sirve cada una de las secciones del pom.

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

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

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]

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. En la implementación ya hemos modificado el prototipo para que se lance la excepción. De no hacerlo así, para automatizar la tabla A tendríais que haberlo hecho vosotros.

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 teber en cuenta lo que hemos explicado en clase y ubicar correctamente cada uno de los fuentes e implementar dichos drivers con JUnit 5 correctamente. Asegúrate de que la sección de dependencias de ty pom es la correcta.
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 ventana "***Run***" de IntelliJ muestra la secuencia de *goals* ejecutadas por maven. Seleccionando la *goal* "***test***" verás el el informe JUnit en formato texto.
Crea una **nueva *Run Configuration*** con nombre "***Run CineTest***", con el comando maven "mvn **clean test -Dtest=CineTest**" (la variable *test* pertenece al plugin *surefire* e indica la clase (o clases) de los drivers que queremos ejecutar). Debes guardar todas las Run Configurations en la carpeta ***intellij-configurations***, en la raiz de tu proyecto maven.

OBSERVACIONES*:*

En el comando maven anterior ejecutamos la **fase clean,** seguida de la **fase test**. De esta forma, borramos primero el *target*, y luego realizamos la construcción. Puedes omitir la fase *clean*, pero es recomendable usarla para asegurarte de que no haya ficheros en el *target* resultantes de construcciones previas y que no necesitamos.

Recuerda que, para **ejecutar los tests a través de Maven** debes marcar la casilla "*Delegate IDE build/run actions to Maven*".

Después de ejecutar los tests, puedes usar el ***plugin*** que hemos instalado en IntelliJ para **ver el informe** con un aspecto similar al que muestra la construcción de IntelliJ.

Adiconnalmente, puedes configurar el pom para que maven muestre el informe en forma de **árbol** (ver **Anexo 2**)

➪ ➪ 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-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*(), *contarCaracteresC2*(), ...

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 el terminal, crea una nueva ***Run onfiguration*** con el nombre "***FicheroTextoTest sin excluidos***"

➪ ➪ Ejercicio 3: Drivers para *delete()*

En el directorio **Plantillas-P02** 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 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 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.

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

➪ ➪ Ejercicio 4: *drivers* parametrizados

A) Implementa un ***test parametrizado*** *reservaButacasC5* para la tabla A del ejercicio1.
Debes implementar el nuevo driver en la clase ***CineTes***t 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).

Nota: Fíjate que sólo podrás prarmetrizar 3 de los 4 drivers.

B) **Modifica** la *Run 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 D.
Debes implementar el nuevo driver en la clase ***DataArrayTes***t 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 ***DataArrayTes***t y etiquetarlo con la etiqueta "***parametrizado***"

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.

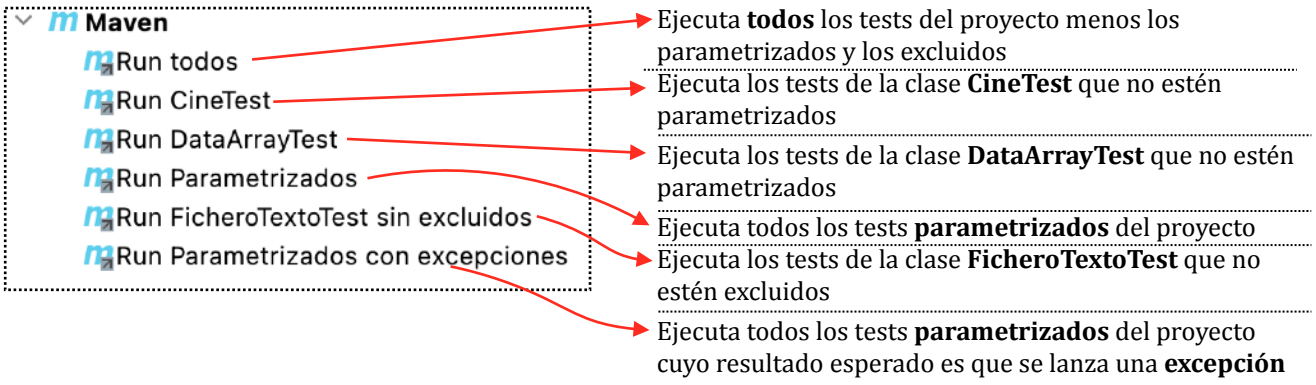
De forma alternativa, JUnit permite usar etiquetas con expresiones booleanas (operadores "&" (and) "|" (or), "!" (not)). Incluso podemos usar paréntesis para ajustar la precedencia del operador. Por ejemplo, podríamos indicar como valor de la variable groups: "(firefox | chrome) & (testRapidos | testsMuyRapidos)" (https://junit.org/junit5/docs/current/user-guide/#running-tests-tag-expressions)

G) **Crea** una nueva *Run Configuration* con nombre "***Run todos*** para ejecutar todos los tests del proyecto excepto los parametrizados y excluidos.

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 **P02-Drivers**.

➪ ➪ ANEXO 1: Lista de elementos *Run Configuration* creados

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



Recuerda que todos ellos debes guardarlos en ***src/main/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.JUnit5Xml30Statele
ssReporter">
           <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.

También podemos configurar el plugin surefire para que muestre los resultados en f**orma de árbol** (https://github.com/fabriciorby/maven-surefire-junit5-tree-reporter). Se puede configurar en formado UNICODE (mostrado en el recuadro), o en formato ASCII (en cuyo caso simplemente tienes que cambiar el valor del atributo implementation por:
`"org.apache.maven.plugin.surefire.extensions.junit5.JUnit5StatelessTestsetInfoTreeReporter"`

```
<plugin>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>3.0.0-M8</version>
    <dependencies>
        <dependency>
            <groupId>me.fabriciorby</groupId>
            <artifactId>maven-surefire-junit5-tree-reporter</artifactId>
            <version>1.1.0</version>
        </dependency>
    </dependencies>
    <configuration>
        <reportFormat>plain</reportFormat>
        <consoleOutputReporter>
            <disable>true</disable>
        </consoleOutputReporter>
        <statelessTestsetInfoReporter

implementation="org.apache.maven.plugin.surefire.extensions.junit5.JUnit5StatelessTests
etInfoTreeReporterUnicode">
        </statelessTestsetInfoReporter>
    </configuration>
</plugin>
```

➪ ➪ 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	isbns	{reserva()}	Resultado esperado
C1	"xxxx"	"xxxx"	"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 isbns registrados en la base de datos son "11111", "22222".

-- 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:33333; "
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
- 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 metodo 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 caminos independientes hayamos obtenido
- Todos los datos de entrada deben de ser concretos, y el resultado esperado también.
- 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. 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 comporamiento 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?

IMPLEMEMENTACIÓN 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 libreria 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 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 en el proceso de construcción del proyecto y qué ficheros genera nuestro proceso de construcción en cada caso.