P03- Automatización de pruebas: Drivers

Implementación de drivers con JUnit 5

El objetivo de esta práctica es implementar los drivers para automatizar los casos de prueba unitarios que hemos diseñado en la sesión anterior. Recuerda que a partir de ahora, nuestro elemento a probar se denominará SUT, y por el momento, va a representar a una UNIDAD (que hemos definido como un método java). La idea es que practiquéis la implementación de drivers, cuyo algoritmo es el que ya hemos explicado en clase.

En este caso hemos elegido 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...

Finalmente, la ejecución de los drivers se realizará integrada en el proceso de construcción del sistema, usando Maven. Es decir, de forma automática (pulsando un botón), se ejecutarán todas las actividades conducentes finalmente a obtener los informes de la ejecución de nuestros tests (casos de prueba).

Ejecución de drivers con JUnit 5

Es muy importante que tengas en cuenta que vamos a ejecutar nuestros tests integrados en el proceso de construcción de Maven. IntelliJ tiene incluido su propio Maven, y también su propio programa para ejecutar los tests, accesible desde el menú contextual de cada fuente de test. Pero nosotros queremos independizar nuestro proceso de construcción de cualquier IDE, de hecho hemos configurado IntelliJ para que no ejecute su propio Maven, sino el que hemos instalado en /usr/local de nuestra máquina virtual, de forma que si prescindimos del IDE y usamos Maven desde el terminal vamos a obtener siempre el mismo resultado. Esto no significa que no podáis ejecutar los tests directamente desde IntelliJ, pero recordad que tenéis que saber hacerlo a través de Maven. La *goal* ***surefire:test*** asociada por defecto a la fase test, será la encargada de ejecutar nuestros tests JUnit con Maven.

Bitbucket

El trabajo de esta sesión también debes subirlo a *Bitbucket*. Todo el trabajo de esta práctica, que será fundamentalmente código, deberá estar en el directorio **P03-Drivers**, dentro de tu espacio de trabajo, es decir, dentro de tu carpeta: ppss-2020-Gx-apellido1-apellido2

Recuerda que si trabajas desde los ordenadores del laboratorio primero deberás configurar git y clonar tu repositorio de Bitbucket.

Ejercicios

Vamos a crearnos un proyecto Maven, que contendrá todos los ejercicios de esta sesión. Para ello elegiremos la opción “Create New Project”. Seleccionaremos “Maven” en la parte izquierda de la siguiente pantalla, y nos aseguraremos de que tiene asignado el JDK_1.8.0 y seleccionamos “Next”.

Los valores para las coordenadas de nuestro proyecto serán “ppss” (para el GroupId), y “drivers” (para el ArtifactId). Usaremos la versión que aparece por defecto.

El campo de texto “Project 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 P03-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 en el subdirectorio P03-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).

➪ ➪ Ejercicio 1: *drivers* para `calculaTasaMatricula()`

Una vez "abierto" el proyecto Maven que acabas de crear, asegúrate de que tiene seleccionado el JDK 1.8. A continuación, dada la tabla de casos de prueba que has diseñado en la sesión anterior para el método ***ppss.Matricula.calculaTasaMatricula()***, se pide lo siguiente:

Nota: En caso de que no dispongas de la tabla de casos de prueba, puedes utilizar ésta:

	Datos de entrada			Resultado esperado
	edad	familia numerosa	repetidor	tasa
C1	19	falso	cierto	2000
C2	68	falso	cierto	250
C3	19	cierto	cierto	250
C4	19	falso	falso	500
C5	61	falso	falso	400

SUT

A) Necesitas el **código fuente de la unidad a probar**. Puedes copiarlo de las plantillas de la primera práctica. No olvides crear el **paquete ppss**. Asegúrate de que el código de tu SUT esté en tu disco duro en el directorio requerido por Maven. Añade en el pom las **propiedades** <project.build.sourceEncoding>, <maven.compiler.source> y <maven.compiler.target> (cópialas del pom de las plantillas de la primera práctica)

drivers

B) A continuación vamos a **implementar los drivers** asociados a tu tabla de diseño de casos de prueba (o usa la tabla que os hemos proporcionado). Primero incluye en el pom las **dependencias** necesarias. Y a continuación, muestra en el editor el código de la clase Matricula, selecciona el nombre de la clase, y desde el menú contextual pulsa sobre "Generate...→Test...". Asegúrate de que el test que se va a generar es JUnit5. Selecciona finalmente la casilla con la unidad a probar.

El nombre de cada driver será "C1_calculaTasaMatricula",... y así sucesivamente. Puedes usar las anotaciones que hemos visto en clase exceptuando @Tag y @ParameterizedTest..

mvn test compile

Una vez implementados los tests, compílalos usando la fase **test-compile** de Maven. Para que IntelliJ nos muestre esa fase debes deseleccionar "Show Basic Phases Only" desde la rueda dentada de la ventana Maven ¿Tienes claro qué diferencia hay entre ejecutar *mvn compiler:testCompile* y *mvn test-compile*? Puedes comprobarlo si ejecutas previamente la fase clean antes de cada uno de dichos comandos.

mvn test

C) Recuerda que en el pom todavía no hemos añadido la sección <build>. **Ejecuta los tests** desde la fase Maven correspondiente. Comprueba que NO se ejecuta ningún test, a pesar de que, por defecto, el plugin surefire está incluido en el pom. Como ya vimos en la primera práctica, JUnit5 necesita otra versión del plugin. Modifica el pom para incluir el plugin surefire con la versión correcta. Si ahora ejecutas de nuevo la fase test, deberías ver en el informe Maven que se han ejecutado todos los tests. Si quieres ver el informe de forma gráfica selecciona el icono con la "M" de color rojo a la derecha de la rueda dentada de la ventana Maven.

test parametri-zado

D) **Implementa** en una nueva clase con nombre "**MatriculaParamTest**" un test parametrizado con los datos de la tabla de casos de prueba proporcionada. Deberás modificar el pom para añadir la dependencia correspondiente. **Nota**: todas las anotaciones que hemos visto en clase, que pueden usarse conjuntamente con la anotación @Test, se pueden usar igualmente con la anotación

@ParameterizedTest. Es decir, que puedes usar @BeforeEach, @BeforeAll, ... o cualquier otra independientemente de que el driver esté anotado con @Test o con @ParemeterizedTest.

Ejecuta de nuevo la fase test de Maven, deberías obtener informes idénticos tanto para los drivers de MatriculaTest, como para los de MatriculaParamTest. Fíjate en que la goal surefire:test ejecuta TODOS los métodos anotados con @Test o @ParameterizedTest

E) **Sustituye** el caso de prueba C5 por (60, true, true, 400). Verás que detectamos un error. Depúralo.

➪ ➪ Ejercicio 2: drivers para buscarTramoLlanoMasLargo()

A partir de la tabla de casos de prueba que hayas diseñado en la sesión anterior para el método ***ppss.Llanos.buscarTramoLlanoMasLargo()***, se pide lo siguiente:

Nota: En caso de que no dispongas de la tabla de casos de prueba, puedes utilizar ésta:

TABLA A

	Datos de entrada	Resultado esperado
	lista de lecturas	Tramo
C1A	{3}	Tramo.origen =0 ; Tramo.longitud = 0
C2A	{100,100, 100, 100}	Tramo.origen = 0; Tramo.longitud = 3
C3A	{120, 140, 180, 180, 180}	Tramo.origen=2; Tramo.longitud=2

SUT

A) Añade el código fuente de la unidad en el **paquete ppss,** y la **clase Llanos** El código es el proporcionado en el enunciado de la práctica anterior. Añade también una nueva **clase Tramo**, con **dos atributos privados**: origen y longitud (de tipo entero), así como sus correspondientes **getters y setters**, más **dos constructores** para la clase: uno sin parámetros (en el que se inicializan a cero los dos atributos), y otro en el que los valores de los atributos se pasan como parámetros en el constructor. **Nota:** En el código de la práctica anterior se usa el método *setDuracion()*, debería ser *setLongitud()*.

drivers

TABLA A

B) A continuación implementa los **drivers** asociados a tu tabla de diseño de casos de prueba (o usa la tabla que os hemos proporcionado). La clase que contiene los drivers debe llamarse LlanosTest

El nombre de cada driver será "**C1A_buscarTramoLlano**",... y así sucesivamente. No uses tests parametrizados

Nota: Para implementar los drivers debes tener en cuenta que el resultado obtenido es un OBJETO java. Por lo tanto, para comparar el resultado real con el esperado puedes: comprobar que cada uno de los valores de los campos son iguales, o bien redefinir el método equals() de la clase Tramo.. Puedes probar a usar las dos opciones. El resultado debe ser idéntico. Para redefinir el método equals(), la forma más sencilla es hacerlo a partir del menú contextual del editor de IntelliJ seleccionando: ***"Generate...→equals() and hashCode()"***. Si haces las comprobaciones campo a campo recuerda que deberás agrupar las aserciones. Fíjate que es mucho mejor usar varias comprobaciones porque nos van a proporcionar más información en caso de fallo.

drivers

TABLA B

C) Añade a la clase LlanosTest, tres nuevos drivers asociados a la siguiente tabla adicional, obtenida también aplicando el método del camino básico. Los nombres de los drivers deben ser **C1B_buscarTramoLlano**,... y así sucesivamente.

TABLA B

	Datos de entrada	Resultado esperado
	lista de lecturas	Tramo
C1B	{-1}	Tramo.origen =0 ; Tramo.longitud = 0
C2B	{-1, -1, -1, -1}	Tramo.origen = 0; Tramo.longitud = 3
C3B	{120, 140, -10, -10, -10}	Tramo.origen=2; Tramo.longitud=2

mvn test

D) Ejecuta todos los tests y depura el código si detectas algún error. Recuerda que, aunque realices modificaciones en el código para depurarlo, TODOS los tests deben seguir en verde. También debes intentar depurar el código SIN cambiar la estructura del CFG del método que queremos probar, si no lo haces así, tendrás que revisar la tabla, puesto que es probable que deje de ser efectiva y eficiente..

test parametri-zado

E) Implementa en una nueva clase con nombre "LlanosParamTest" un test parametrizado con los datos de las tablas de casos de prueba A y B.

Ejecuta de nuevo la fase test de Maven, deberías obtener informes idénticos tanto para los drivers de LlanosTest, como para los de LlanosParamTest

➪ ➪ Ejercicio 3: selección y filtrado de las ejecuciones de los tests

-Dgroups
-DexcludedGroups
-Dtest

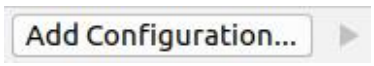
Hemos visto en clase que la goal ***surefire:test*** contiene parámetros configurables, como por ejemplo las variables "**groups**" y "**excludedGroups**". Otro parámetro interesante (cuando se están desarrollando los drivers) es "**test**", que permite ejecutar una única clase y/o método. Podéis ver la lista completa de parámetros(variables) configurables en: https://maven.apache.org/surefire/maven-surefire-plugin/test-mojo.html).

La variable test puede tener asignada una expresión regular, de forma que podamos seleccionar la ejecución de clases y/o métodos que sigan el patrón especificado, o puede usarse, que es como vamos a hacerlo nosotros, indicando clases y/o métodos concretos. Como ya hemos visto en clase, podemos cambiar la **configuración** del plugin desde el **pom**, o desde **línea de comandos**. La sintaxis desde línea de comandos es la siguiente:

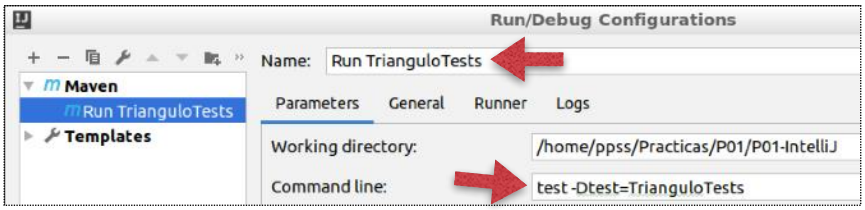
```
mvn test –Dtest=ClaseAEjecutar1, ClaseAEjecutar2#metodoX, ...
```

Desde IntelliJ podemos ejecutar cualquier comando Maven, pulsando sobre el icono correspondiente de la ventana Maven. No es necesario poner explícitamente el comando "mvn", sino la fase/s goal/s que queremos ejecutar, separadas por espacios.

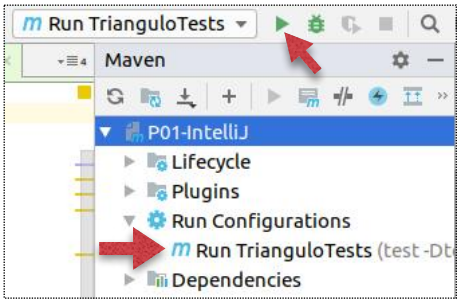
Otra opción que nos resultará más útil, si por ejemplo queremos usar varios comandos mvn de forma frecuente es crearnos elementos "Configuration", de forma que podemos tener "guardados" diferentes comandos maven para poder usarlos cuando queramos, simplemente pulsando un botón.



Para crearnos una nueva "configuración", podemos hacerlo desde "Run→Edit Configurations...", o acceder directamente desde la barra de herramientas.



IntelliJ nos proporciona "plantillas" para poder usarlas con diferentes tipos de proyectos. Nosotros crearemos "Configurations" de tipo **Maven** (pulsando sobre el **icono +**). Simplemente tendremos que elegir un nombre e indicar el comando maven asociado.



Una vez creado, nos aparecerá, como un nuevo "botón", anidado en el elemento "Run Configurations" de la ventana Maven. Podemos crear todas las "Configurations" que queramos, y también editarlas en cualquier momento. T

También podremos ejecutar cualquier "Configuration" desde la barra de herramientas, eligiendo la que nos interese, y pulsando sobre el icono con forma de triángulo verde.

Nota: Cada vez que, desde el menú contextual del fichero java de un driver, decidimos ejecutar los tests a través de IntelliJ, puedes comprobar que IntelliJ automáticamente crea una nueva "Configuration" para ejecutar los tests de esa clase, usando la plantilla correspondiente.

Teniendo esto en cuenta, así como lo que hemos visto en clase, se pide:

A) Crea "Configurations" para ejecutar las clases con los drivers del Ejercicio1, y las del Ejercicio2, con los nombres "Run ejercicio1" y "Run ejercicio2", respectivamente (usando el parámetro "test" del plugin surefire). **Nota**: Si el valor que asignamos al parámetro "test" contiene espacios en blanco, deberás usar dobles comillas para que IntelliJ no lo interprete como el nombre de una fase y/o goal, es decir usaremos: -Dtest="clase1, clase2", en vez de. -Dtest=clase1, clase2

B) Etiqueta (anotación @Tag) el código de pruebas, de forma que podamos ejecutar todos los tests parametrizados, y crea una "configuration" con nombre "Run Parametrizados", o todos los tests sin parametrizar (y añade la "configuration" con nombre "Run NO Parametrizados"

C) Etiqueta (anotación @Tag) el código de pruebas, de forma que podamos ejecutar únicamente los tests de la Tabla A del ejercicio anterior, y crea una "configuration" con nombre "Run TestsLlanosTablaA", o sólo los tests de la Tabla B (y añade la "configuration" con nombre "Run TestsLlanosTablaB".

➪ ➪ Ejercicio 4: pruebas de excepciones y agrupaciones de aserciones

Proporcionamos una implementación para el método `ppss.DataArray.delete()` (ver Figura 1)

La especificación del método es la que se indicó en la práctica P01, aunque contemplando el hecho de que se debe lanzar una excepción de tipo *DataException*, con un mensaje asociado, 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 <= y además la colección esté vacía (mensaje: "Colección vacía. Y el valor a borrar debe ser > cero"), cuando el elemento a borrar no pertenezca a la colección (mensaje: ·"Elemento no encontrado").

Supongamos que hemos diseñado la siguiente tabla de casos de prueba:

Entradas		**Resultado esperado**
colección, numElem	**Elemento a borrar**	**(colección + numElem + array que se devuelve) o excepción de tipo DataException**
[1,3,5,7,0,0,0,0,0,0], 4	5	[1,3,7,0,0,0,0,0,0,0], 3, [1,3,7,0,0,0,0,0,0,0]
[1,3,3,5,7,0,0,0,0,0], 5	3	[1,3,5,7,0,0,0,0,0,0], 4, [1,3,5,7,0,0,0,0,0,0]
[1,2,3,4,5,6,7,8,9,10], 10	4	[1,2,3,5,6,7,8,9,10,0], 9, [1,2,3,5,6,7,8,9,10,0]
[0,0,0,0,0,0,0,0,0,0], 0	8	DataException(m1)
[1,3,5,7,0,0,0,0,0,0],4	-5	DataException(m2)
[0,0,0,0,0,0,0,0,0,0], 0	0	DataException(m3)
[1,3,5,7,0,0,0,0,0,0], 4	8	DataException(m4)

m1: "No hay elementos en la colección"
m2: "El valor a borrar debe ser > cero"
m3:"Colección vacía. Y el valor a borrar debe ser > cero"
m4: "Elemento no encontrado"

Implementa en una clase **DataArrayTest**, los *drivers* correspondientes a la tabla de casos de prueba proporcionada. Deberás agrupar las aserciones y comprobar que el mensaje de las excepciones generadas es el correcto.

Opcionalmente, puedes parametrizar los tests.

Añade una nueva "configuration" para poder ejecutar únicamente la clase DataArrayTest, con el nombre "Run DataArrayTest" (no es necesario que uses la anotación @Tag)

Figura 1. Código del método delete() y de la clase DataException:

```
public class DataArray {
    ...//el resto de código es el mismo que el de las plantillas de P01
public int[] delete(int elem) throws DataException {
        int i=0;
        boolean encontrado = false;

        if ((numElem==0)&&(elem<=0)) {
            throw new DataException("No hay elementos en la colección. "+
                    "Y el valor a borrar debe ser > 0");
        } else if (numElem==0) {
            throw new DataException("No hay elementos en la colección");
        } else if (elem<=0) {
            throw new DataException("El valor a borrar debe ser > cero");
        } else {
            //buscamos el elemento en la colección
            while (i < numElem && !encontrado) {
                if (coleccion[i] == elem) {
                    encontrado = true;
                } else {
                    i++;
                }
            }
            if (encontrado) {
                while (i < (numElem - 1) && coleccion[i] != 0) {
                    coleccion[i] = coleccion[i + 1];
                    i++;
                }
                if (i == (numElem - 1)) {
                    coleccion[numElem - 1] = 0;
                }
                numElem--;
            } else throw new DataException("Elemento no encontrado");
        }
        return coleccion;
    }
}
```

```
public class DataException extends Exception {
    public DataException() { }

    public DataException(String message) {
        super(message);
    }
}
```

Resumen

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

IMPLEMEMENTACIÓN DE LOS TESTS

- El resultado del diseño de casos de prueba es necesario para poder implementar los drivers.
- Implementaremos el código de pruebas en src/test/java, en el mismo paquete que el código a probar. Usaremos las anotaciones Junit5 para implementar los drivers para cada caso de prueba. Nuestra SUT será una unidad, por lo tanto, estaremos automatizando pruebas unitarias, usando técnicas dinámicas.
- 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 (en src/main/java)

EJECUCIÓN DE LOS TESTS

- La ejecución de los tests se integra en el proceso de construcción del sistema, para lo que usaremos 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 para ejecutar los drivers.
- Podemos ser "selectivos" a la hora de ejecutar los tests, 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.