

# P01A: Entorno de pruebas. Preparación.

Para poder realizar las prácticas, tendremos que:

- Descargar y poner en marcha la máquina virtual de la asignatura
- Crear un repositorio Git (remoto, en Bitbucket, y local, en nuestra máquina virtual)
- Crear una cuenta educacional en JetBrains (si no la tenéis creada ya de otras asignaturas), para poder usar IntelliJ

## Máquina virtual de la asignatura

Para las prácticas en el laboratorio utilizaremos una Máquina Virtual en la que tendrás instalado el software que vamos a necesitar. En el laboratorio estará accesible desde el escritorio cuyo icono lleva el nombre "PPSS-2020-21".

El login y password de la máquina virtual es **ppss** en ambos casos.

Para trabajar con la máquina virtual desde vuestro ordenador necesitaréis tener instalado previamente **VirtualBox** (incluyendo VirtualBox **Extension Pack**) y crear una nueva máquina virtual a partir del fichero "ppss-2020-21.vdi".

Podéis descargar el fichero comprimido ("ppss-2020-21.vdi.zip") desde el siguiente enlace: <https://drive.google.com/file/d/1C3L7NiunVFRceARUDN5KsZ60WKjvzrC> (para descomprimir el zip puedes usar *stuffit expander*, o *7zip* (el vdi ha sido coprimido con *stuffit expander*).

Para CREAR la nueva máquina virtual desde VirtualBox, con la opción **Nueva**. Usaremos los siguientes valores:

- ❖ Nombre: Podéis poner cualquier nombre arbitrario
- ❖ Tipo: Linux
- ❖ Version: Ubuntu (64 bits)
- ❖ Usar un archivo de disco duro virtual existente. A continuación seleccionamos el fichero **ppss-2020-21.vdi** de nuestro disco duro.

Una vez creada la máquina virtual, y antes de iniciarla, es posible que necesites cambiar la configuración de la pantalla para que no se muestre en un tamaño demasiado pequeño (dependerá de tu ordenador). Desde **Configuración→Pantalla** puedes probar a ponerla al 200% si lo necesitas.

Recuerda que debes instalar **GuestAdditions** para no tener problemas al redimensionar la ventana de la máquina virtual.

Finalmente seleccionamos la máquina que hemos creado y a continuación elegimos la opción **Iniciar**.

Os recomendamos que activéis la compartición del portapapeles entre la máquina anfitrión y vuestra máquina virtual desde: **Devices→Shared Clipboard→Bidirectional**. De esta forma, podremos copiar/pegar el contenido del portapapeles entre ambas máquinas.

## Git: repositorio (remoto, en *Bitbucket*)

Es una muy buena práctica (indispensable para futuros trabajos profesionales en grupo) utilizar una **herramienta de gestión de versiones**, para así tener todo nuestro trabajo accesible desde cualquier lugar, y con el historial de versiones, por si nos interesa recuperar alguna versión anterior, crear nuevas versiones a partir de ellas, etc.

Aunque nosotros no vamos a trabajar en grupo, es interesante habituarnos a utilizar un repositorio remoto para, por ejemplo, tener siempre una copia de seguridad del trabajo realizado, continuar el trabajo iniciado desde cualquier otro ordenador en el mismo punto que lo dejamos, ..., entre otras cosas.

Bitbucket es un servicio que nos permite trabajar con repositorios Git (Git es una herramienta de gestión de versiones). Lo primero que haremos será **crearnos una cuenta** (gratuita) en Bitbucket,

proporcionando vuestro nombre y apellidos, un nombre de usuario y password de vuestra elección, y una dirección de correo electrónico (usad preferiblemente vuestro correo institucional).

Una vez creada la cuenta, tenéis que crear UN ÚNICO repositorio (usando el signo "+" del panel de la izquierda) que alojará TODO el trabajo de prácticas que realicéis durante el curso. Vamos a indicar qué valores debemos usar para crear el repositorio:

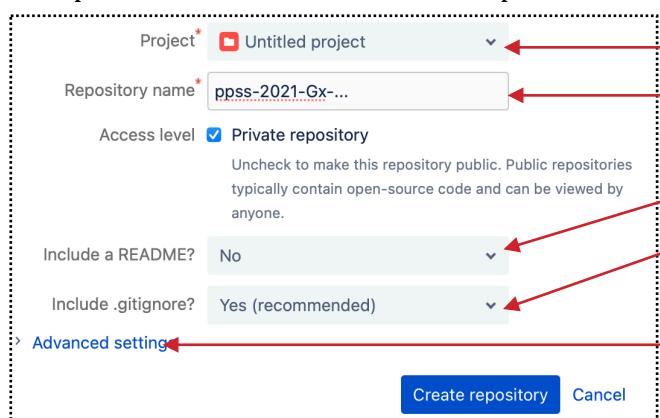
El repositorio tendrá como nombre: **ppss-2021-Gx-apellido1-apellido2**, en donde:

- ❖ **Gx** es el identificador del grupo de prácticas que tenéis asignado. Los valores posibles son: G1..G9, G40, según la siguiente tabla:

Grupo de prácticas	Identificador
Martes de 11 a 13h	G1
Lunes de 13 a 15h	G2
Martes de 9 a 11h	G3
Lunes de 09 a 11h	G4
Martes de 19 a 21h	G5

Grupo de prácticas	Identificador
Miércoles de 19 a 21h	G6
Miércoles de 9 a 11h	G7
Lunes de 17,30 a 19,30h	G8
Miércoles de 17 a 19h	G9
Martes de 11 a 13h	G40

- ❖ **apellido1** es el primer apellido (todo en minúsculas y SIN acentos)
- ❖ **apellido2** es el segundo apellido (todo en minúsculas y SIN acentos). Si algún alumno no tiene segundo apellido, entonces omitiremos esta parte



- Selecciona "Untitled Project"
- Indica el nombre del repositorio
- El repositorio será **PRIVADO**.
- No** incluyas un README.
- Deja Yes en Include .gitignore
- Debes establecer el Lenguaje usado a **"Java"** (desplegando las opciones avanzadas)

Después de crear el repositorio, deberéis dar **permiso de lectura** al usuario **eli@gcloud.ua.es**, a través de las opciones "*Repository Settings→GENERAL→Users and group access*" de vuestro proyecto (desde el panel izquierdo).

### Git: repositorio (local, en nuestra máquina virtual)

**comando git clone**  
**(para descargar una copia del repositorio)**

Una vez creado el repositorio, vamos a "descargar y sincronizar" dicho repositorio remoto en un directorio de nuestra máquina virtual. A este proceso lo llamaremos CLONAR el repositorio remoto. Si trabajas en el laboratorio, este paso lo tendrás que hacer SIEMPRE.

Supongamos que nuestro directorio de trabajo va a ser "**\$HOME/practicas**". Para CLONAR el repositorio en dicho directorio, necesitamos la URL del repositorio que acabamos de crear. Esta información podemos consultarla desde la vista "**<> Source**", seleccionado el botón Clone (arriba a la derecha de esa pantalla), y copiando el comando que nos mostrará en el portapapeles..

A continuación pegamos el contenido copiado en un terminal (desde el directorio \$HOME/practicas), y ejecutamos el comando copiado. Vemos que se crea el directorio:

### \$HOME/practicas/ppss-2021-Gx-apellido1-apellido2

Después de clonar el repositorio Git, podemos comprobar que se ha creado el directorio oculto:

```
$HOME/practicas/ppss-2021-Gx-apellido1-apellido2/.git
```

El **directorio .git** contiene nuestro repositorio local Git (que está conectado con el repositorio remoto en Bitbucket). A partir de ahora, cualquier fichero que añadamos en el directorio \$HOME/practicas/ppss-2021-Gx-apellido1-apellido2/ podrá estar sujeto al control de versiones de Git.

También veremos el fichero oculto **.gitignore** (que se ha creado junto con el repositorio). Es un fichero de texto en el que se indican todos aquellos ficheros y/o directorios que serán "ignorados" por git. Es decir, que no serán "guardados" en el repositorio para gestionar los cambios realizados sobre ellos. Puedes ver y editar el contenido del fichero .gitignore.

Cuando construyamos nuestros proyectos java con Maven, se creará sistemáticamente el directorio **target**, que contendrá, entre otras cosas, los ficheros **.class** de nuestra aplicación. No tiene sentido guardar en Bitbucket información que podemos obtener en cualquier momento (volviendo a construir el proyecto), esto va a ocupar espacio en disco y consumir un tiempo innecesario (de subidas y descargas). Puedes comprobar que una de las líneas del fichero .gitignore es "target/", que significa que se ignorará el contenido de cualquier carpeta (y todo su contenido) con ese nombre.

A continuación vamos a **configurar Git**. Recuerda que este paso deberás hacerlo SIEMPRE cuando trabajes en los ordenadores del laboratorio

**comando git config  
(ejecutar siempre en el laboratorio)**

El comando **git config** nos permitirá guardar en el fichero .git/config algunos parámetros de configuración para trabajar con git. Concretamente nuestro nombre de usuario y e-mail. Si usamos nuestro ordenador sólo hay que hacerlo la primera vez.

Configuraremos nuestra **identidad** a través de nuestro nombre y dirección de correo electrónico con los siguientes comandos:

```
> git config user.name <nombreUsuario>
```

Siendo <nombreUsuario> el nombre que mostrará Git cuando hagamos un *commit*. Poned vuestro nombre y apellidos. P.ej. "Luis Lopez Perez"

```
> git config user.email <emailUsuario>
```

Siendo <emailUsuario> el email del usuario en Bitbucket (no es necesario poner comillas)

**comando git status  
(para ver el estado de los ficheros)**

El comando **git status** nos muestra el estado de los ficheros de nuestro directorio de trabajo (en **rojo** significa que el fichero todavía no está bajo el control de Git, o sea, que si lo borramos, lo perdemos).

Para que cualquier cambio realizado sobre un fichero de nuestro directorio de trabajo sea controlado por git, primero tenemos que "marcar" dicho fichero usando el comando **git add**. Y posteriormente guardaremos el fichero marcado en nuestro repositorio local con el comando **git commit**. Para subir (copiar) nuestro repositorio local a Bitbucket usaremos el comando **git push**.

**Comandos para "guardar"  
todo nuestro trabajo en local  
en Bitbucket  
(ejecutar SIEMPRE)**

```
> git add .
> git commit -m "Mensaje obligatorio"
> git push
```

**IMPORTANTE:** Siempre debes ejecutar los comandos git desde tu directorio de trabajo, es decir, desde el directorio que contiene el subdirectorio oculto .git

> `git add .`

Con este comando marcamos TODOS los ficheros del directorio desde el que ejecutamos el comando (incluimos los ficheros del directorio actual y los de cualquier subdirectorio)

> `git commit -m "mensaje"`

"Copia" todos los ficheros "marcados" en nuestro repositorio local (en el directorio `.git`). Es OBLIGATORIO incluir un mensaje en cada commit que hagamos usando la opción "`-m`". El mensaje tiene que ir entre dobles comillas..

> `git push`

"Copia" el contenido de todo nuestro repositorio local, en el repositorio remoto en Bitbucket.

Si has hecho todo esto con **tu propio ordenador**, a partir de ahora, lo único que tendrás que hacer es utilizar estos tres comandos para sincronizar (en tu repositorio remoto) los cambios realizados en local. Recuerda que SIEMPRE deberás hacerlo desde `$HOME/practicas/ppss-2021-Gx-apellido1-apellido2` (aunque en cada práctica trabajemos en algún subdirectorio del mismo)

Si trabajas siempre utilizando los ordenadores del laboratorio, cuando llegues a casa y trabajes en tu ordenador, tendrás que (la primera vez):

**Secuencia de trabajo en el ordenador de casa  
(sólo la primera vez)**

1. clonar el repositorio de Bitbucket (`git clone`)
2. configurar nuestra identidad (`git config`)
3. trabajar en el directorio de trabajo (`ppss-2021-Gx-apellido...`)
4. subir los cambios a Bitbucket (`git add + git commit + git push`)

Para las siguientes veces: supongamos que has trabajado desde los ordenadores del laboratorio, y luego quieres seguir trabajando en casa (y en casa ya habías clonado previamente el repositorio). En ese caso tendrás que hacer primero un `git pull` desde el ordenador de tu casa para sincronizar (y descargarte) los cambios que hiciste en el laboratorio antes de continuar trabajando:

**Secuencia de trabajo en el ordenador de casa  
(el resto de veces)**

- > `git pull`
2. trabajar en el directorio de trabajo (`ppss-Gx-2021-apellido...`)
  3. subir los cambios a Bitbucket (`git add + git commit + git push`)

## LICENCIA educacional IntelliJ

Para poder usar la versión Ultimate de IntelliJ necesitáis solicitar, cada uno de vosotros, una licencia educacional desde [www.jetbrains.com](http://www.jetbrains.com) (válida durante un año). Para ello tendréis que:

1. Crearos una cuenta. Para solicitar la licencia educacional necesariamente tendréis que proporcionar vuestro e-mail institucional. Recibiréis un correo con un enlace al que tendremos que acceder para confirmar la petición.
2. Una vez que entréis en vuestra cuenta, veréis una pantalla con el mensaje "No available Licences", y varios enlaces. Seleccionamos el enlace **Apply for a free student or teacher license for educational**

purposes. Rellenamos la petición indicando que sois estudiantes (lógicamente). Y a continuación recibiréis de nuevo un correo para activar la licencia educacional.

Una vez que obtengáis la licencia, ésta será necesaria para poder ejecutar IntelliJ. Si usáis los ordenadores de los laboratorios tendréis que activarla SIEMPRE, puesto que cuando apagáis la máquina, NO se guarda ninguna información ni ningún cambio que hayáis podido hacer en la máquina virtual.

Al ejecutar IntelliJ desde los laboratorios os aparecerá una ventana desde donde seleccionaremos "Enter Key", desde donde introduciremos nuestro e-mail y password de nuestra cuenta de JetBrains.

# P01A: Entorno de pruebas

## IMPORTANTE. A TENER EN CUENTA DURANTE TODO EL CURSO

- Debes subir a Bitbucket las prácticas que realices. **SÓLO SE REVISARÁN** aquellas prácticas subidas **antes** de iniciar la sesión de la siguiente práctica.
- **DURANTE** las clases de **prácticas**:
  - con **carácter general** se darán explicaciones sobre las soluciones de la práctica anterior,
  - con **carácter individual** se realizará el seguimiento del trabajo subido por el alumno (siempre y cuando se haya hecho dentro del plazo establecido) y
  - se resolverán las dudas que surjan.
- Cada alumno tiene asignado un profesor de prácticas, que realizará el seguimiento de vuestro trabajo. No se admitirán cambios de turnos de prácticas que no sean realizados a través de la secretaría del centro.
- Recuerda que tu trabajo de prácticas te permitirá comprender y asimilar los conceptos vistos en las clases de teoría. Y que vamos a evaluar no sólo tus conocimientos teóricos sino que sepas aplicarlos correctamente. Por lo tanto, el **resultado** de tu trabajo **PERSONAL** sobre las clases en aula y en laboratorio determinará si alcanzas o no las competencias teórico-prácticas planteadas a lo largo del cuatrimestre.

Una vez que tengamos la máquina virtual preparada, nuestro repositorio Git creado y clonado en la máquina virtual, y nuestra licencia Jetbrains activa, ya estamos en disposición de comenzar con nuestro trabajo práctico.

## Maven

Maven es una **herramienta de construcción** de proyectos Java. Por definición, la construcción (*build*) de un proyecto es la secuencia de tareas que debemos realizar para, a partir del código fuente, poder usar (ejecutar) nuestra aplicación. Ejemplos de tareas que forman parte del proceso de construcción pueden ser compilación, *linkado*, pruebas, empaquetado, despliegue.... Otros ejemplos de herramientas de construcción de proyectos son *Make* (para lenguaje C), *Ant* y *Graddle* (también para lenguaje Java).

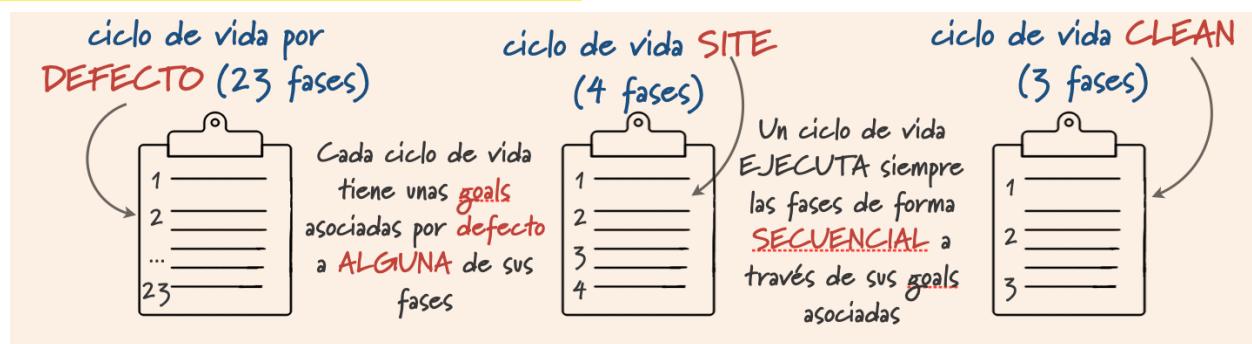
Maven puede utilizarse tanto desde línea de comandos (comando mvn) como desde un IDE.

Cualquier herramienta de construcción de proyectos necesita conocer la secuencia de tareas que debe ejecutar para construir un proyecto. Dicha secuencia de tareas se denomina ***Build Script***.

Maven, a diferencia de Make o Ant (Graddle utiliza elementos de Ant y de Maven), permite definir el *build script* para un proyecto de forma declarativa, es decir, que no tenemos que indicar de forma explícita la "lista de tareas" a realizar, ni "invocar" de forma explícita la ejecución de dichas tareas.



Maven tiene predefinidas TRES secuencias de tareas (*build scripts*) para construir proyectos. Cada una de estas secuencias se denomina **ciclo de vida**.



El ciclo de vida más utilizado es el ciclo de vida por defecto (**default lifecycle**), y está formado por una lista de **23 tareas**, denominadas **fases**. Ejemplos de fases son: *compile, test, package, deploy,...* Es importante que tengas claro que **una fase es un concepto LÓGICO**, no es un ejecutable, sino que podrá tener **ASOCIADO** algún ejecutable que realice una determinada acción.

**FASES maven**  
*(cada fase puede tener asociadas cero o más acciones ejecutables)*

Una fase Maven identifica cuál debe ser la naturaleza de la acción o acciones que se ejecuten DURANTE la misma. Por ejemplo, el ciclo de vida por defecto contiene la fase "compile" y la fase "test": la primera permite asignar acciones ejecutables que lleven a cabo el proceso de compilación del proyecto, mientras que la segunda está pensada para que se ejecuten las pruebas unitarias (lógicamente, la fase de compilación será anterior a la fase de pruebas).

**GOALS y plugins**  
*(una goal puede asociarse a una fase.  
 Un plugin tiene 1 o varias goals)*

Las acciones que se ejecutan en cada una de las fases se denominan **GOALS**. Por ejemplo la fase compile tiene asociada por defecto la **goal (o acción)** denominada compiler:compile, que lleva a cabo la compilación de los fuentes del proyecto. Cualquier goal pertenece a un **PLUGIN**. Un plugin no es más que un conjunto de goals. Por ejemplo, el plugin *compiler* contiene las goals *compiler:compile* y *compiler:testCompile* (el nombre de la *goal* SIEMPRE va precedida del nombre del *plugin* separado por ":" )

El proceso de construcción de maven genera un fichero empaquetado (jar, war, ear, ...), en el directorio *target*. Cada tipo de empaquetado, tiene asociadas POR DEFECTO ciertas GOALS.

Por ejemplo, cuando nuestro proyecto se empaqueta como un .jar, las GOALS asociadas al ciclo de vida por defecto son las siguientes:

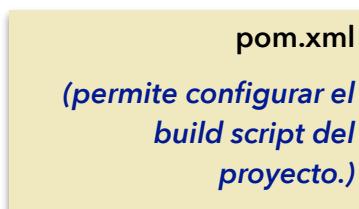
Fase	plugin : goal	acciones
process-resources	maven-resources-plugin: resources	Copia *.* de /src/main/resources en target
compile	maven-compiler-plugin: compile	Compila *.java de /src/main/java
process-test-resources	maven-resources-plugin: testResources	Copia *.* de /src/test/resources en target
test-compile	maven-compiler-plugin: testCompile	Compila *.java de /src/test/java
test	maven-surefire-plugin: test	Ejecuta los tests unitarios
package	maven-jar-plugin:jar	Empaque *.class + recursos en un jar
install	maven-install-plugin: install	Copia el fichero jar en repositorio local
deploy	maven-deploy-plugin: deploy	Copia el fichero jar en repositorio remoto

Una **goal**, por tanto, no es más que un código ejecutable, implementado por algún desarrollador. Algunos desarrolladores "deciden" que una determinada goal estará asociada POR DEFECTO a una determinada fase de algún ciclo de vida Maven. **Todas las goals son CONFIGURABLES** (disponen de un conjunto de variables propias que tienen valores por defecto y que podemos cambiar). Por ejemplo, podemos cambiar la fase a la que se asociará dicha goal. Una goal SIEMPRE está contenida en un PLUGIN.

La forma de provocar la ejecución de una goal durante una fase consiste simplemente en añadir el plugin que la contiene en el fichero pom.xml (y configurar sus valores por defecto, si es necesario). Si una goal no tiene asociada una fase por defecto, y no asociamos de forma explícita dicha goal a alguna fase, la goal **NO SE EJECUTARÁ** (aunque incluyamos su plugin en el fichero pom.xml).

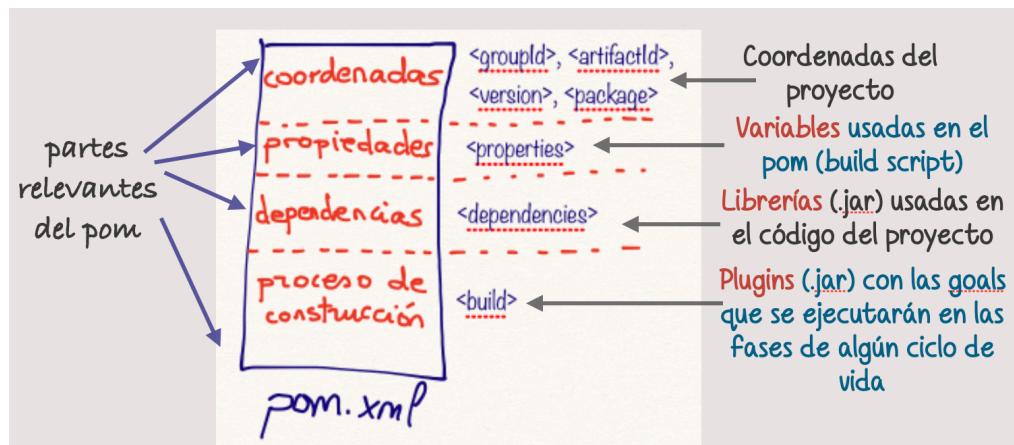
Por ejemplo, cuando el empaquetado es **jar**:

- ❖ La GOAL **compiler:testCompile** se ejecutará automáticamente durante la fase test-compile.
- ❖ La GOAL **compiler:compile** se ejecutará automáticamente durante la fase compile
- ❖ NO es necesario incluir el plugin **compiler** en el pom, a menos que queramos cambiar su configuración por defecto.



Cualquier proyecto Maven debe contener en su directorio raíz el fichero **pom.xml**. Dicho fichero nos permitirá configurar la secuencia de acciones a realizar (build script) para construir el proyecto, mediante la etiqueta **<build>**. También podremos indicar qué librerías (ficheros .jar) son necesarias para compilar/ejecutar/probar... nuestro proyecto (etiqueta **<dependencies>**).

Es importante que sepamos identificar al menos 4 "secciones" en el fichero pom.xml. Cada una de ellas se caracteriza por usar determinadas etiquetas:



Como resultado del proceso de construcción de un proyecto Maven se obtiene un **artefacto** (fichero empaquetado) que se identifica mediante sus **coordenadas**, separadas por ":". Para identificar un artefacto Maven se utilizan, como mínimo, tres elementos, o coordenadas, de forma que cualquier artefacto Maven se especifica de forma única (no hay dos artefactos con las mismas coordenadas).

Las coordenadas obligatorias que identifican de forma única a un artefacto Maven son: **groupId:artifactId:version**, en donde:

- ❖ **groupId** es el identificador de grupo. Se utiliza normalmente para identificar la organización o empresa desarrolladora y puede utilizar notación de puntos. Por ejemplo: *org.ppss*
- ❖ **ArtifactId** es el identificador del artefacto (nombre del archivo), normalmente es el mismo que el nombre del proyecto. Por ejemplo: *practica1*

- ❖ **Version** es la versión del artefacto. Indica la versión actual del fichero correspondiente. Por ejemplo: *1.0-SNAPSHOT*.

Los artefactos se almacenan en un repositorio local Maven, situado en \$HOME/.m2/repository. Las coordenadas se usan para identificar exactamente la ruta de cada fichero en el repositorio maven. Por ejemplo:

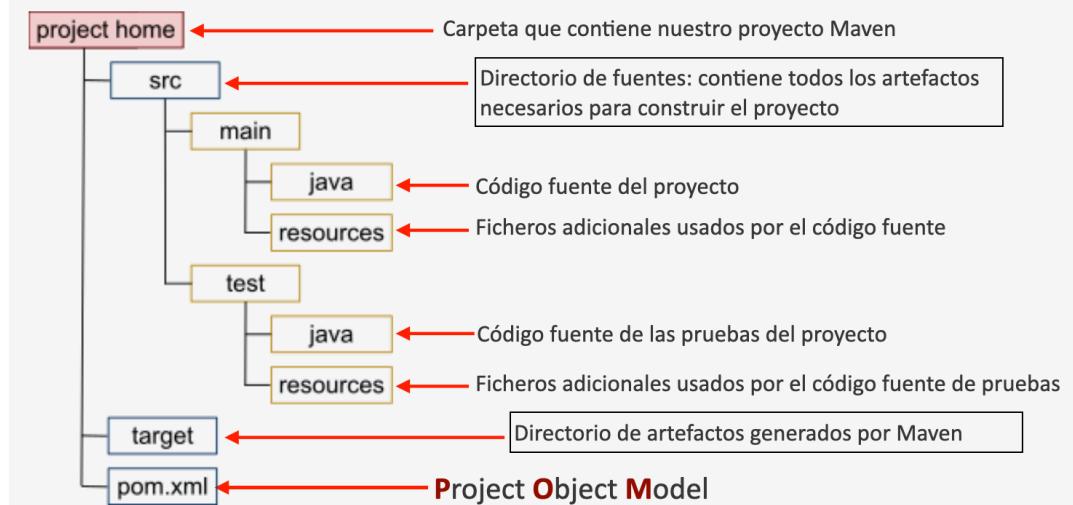
- ❖ **org.ppss:practica1:1.0-SNAPSHOT** representa al fichero \$HOME/.m2/repository/org/ppss/practica1/1.0-SNAPSHOT/practica1-1.0-SNAPSHOT.jar (por defecto, los artefactos tienen la extensión .jar)
- ❖ **org.ppss:practica1:2.0-SNAPSHOT** representa al fichero \$HOME/.m2/repository/org/ppss/practica1/2.0-SNAPSHOT/practica1-2.0-SNAPSHOT.jar
- ❖ **org.ppss:projeto3:war:1.0-SNAPSHOT** representa al fichero \$HOME/.m2/repository/org/ppss/projeto3/1.0-SNAPSHOT/projeto3-1.0-SNAPSHOT.war

### estructura de directorios Maven

*(la misma en TODOS los proyectos Maven)*

Todos los proyectos Maven usan la MISMA estructura de directorios. Así, por ejemplo, el código fuente del proyecto estará en el directorio **src/main/java**, y el código que implementa las pruebas del proyecto siempre lo encontraremos en el directorio **src/test/java**. Los artefactos generados durante la construcción, por ejemplo los ficheros .class, siempre estarán en el directorio **target** (o alguno de sus subdirectorios). El directorio target se genera automáticamente en cada construcción del proyecto, por eso no necesitamos "guardarlo" en Bitbucket.

Aquí mostramos PARTE de la estructura de directorios de Maven



Por otro lado, cualquier LIBRERÍA EXTERNA (ficheros .jar) que utilicemos en nuestro proyecto Maven, debe incluirse en el fichero pom.xml (en la sección <**dependencies**>), de forma que, si no se encuentra físicamente dicho fichero en nuestro disco duro, Maven lo descarga automáticamente de sus repositorios en la nube. Es más, si utilizamos una librería, que a su vez depende de otra, Maven automáticamente se encarga de descargarla también esta última, y así sucesivamente. Esto hace que nuestros proyectos "pesan" poco, ya que tanto el directorio target, como cualquier librería y/o plugin utilizados por el proyecto se descargarán y/o generarán automáticamente si es necesario, cada vez que construyamos el proyecto. Por tanto, si queremos "llevarnos" nuestro proyecto a otra máquina, únicamente necesitamos el fichero pom.xml, y el directorio src del proyecto (el directorio src contiene todos los fuentes del proyecto).

**repositorios locales y remotos Maven**  
*(almacenan artefactos Maven)*

Todos los artefactos generados y/o utilizados por Maven se almacenan en repositorios. Maven mantiene una serie de repositorios remotos, que alojan todos los plugins y librerías que podemos utilizar. Cuando ejecutamos Maven por primera vez en nuestra máquina, se crea el directorio **.m2** (en nuestro \$HOME), que será nuestro repositorio local. Cuando iniciamos un proceso de construcción Maven, primero se consulta nuestro repositorio local, para ver si contiene todos los artefactos necesarios para realizar la construcción. Si falta algún artefacto en nuestro repositorio local, Maven automáticamente lo descargará de algún repositorio remoto.

**Ejecución de Maven**  
*(mvn fase/goal)*

Para iniciar el proceso de construcción de Maven, usamos el comando mvn seguido de la fase (o fases) que queremos realizar, o bien indicando la goal, o goals que queremos ejecutar de forma explícita (separadas por espacios). Si incluimos una o más fases, o goals, se ejecutarán una por una en el mismo orden que hemos indicado. Por ejemplo, si tecleamos: mvn fase1 fase2 plugin1:goal3 plugin2:goal4, será equivalente a ejecutar: mvn fase1, mvn fase2, mvn plugin1:goal3, y mvn plugin2:goal4, en este orden.

El comando **mvn <faseX>** ejecuta todas las goals asociadas a todas y cada una de las fases, siguiendo exactamente el orden de las mismas en el ciclo de vida correspondiente, desde la primera, hasta la fase que hemos indicado (<faseX>).

El comando **mvn plugin:goal** ejecuta únicamente la goal que hemos especificado

Las fases se ejecutan siempre en el mismo orden  
 comenzando desde la PRIMERA !!!

FASES	PLUGIN : GOALS
1 validate	
2 initialize	
3 generate-sources	
4 process-sources	
5 generate-resources	
6 process-resources	resources:resource
7 compile	compiler:compile
8 process-classes	
9 generate-test-sources	
10 process-test-sources	
11 generate-test-resources	
12 process-test-resources	resources:testResources
13 test-compile	compiler:testCompile
14 process-test-classes	
15 test	surefire:test
16 prepare-package	
17 package	jar:jar
18 pre-integration-test	
19 integration-test	
20 post-integration-test	
21 verify	
22 install	install:install
23 deploy	deploy:deploy

**mvn test-compile**

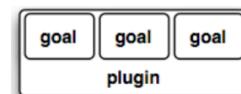
Ejecuta las goals de las fases 1..13  
 Genera los .class de src/test/java

**mvn test**

Ejecuta las goals de las fases 1..15  
 Ejecuta los .class de src/test/classes y genera un informe

**mvn package**

Ejecuta las goals de las fases 1..17  
 Genera el .jar del proyecto



**EJEMPLO**

**mvn compile**

Ejecuta las goals de las fases 1..7  
 Genera los .class de src/main/java

ProyectoMaven

src  
 main ...  
 test ...

target

generated-sources ...  
 classes ...  
 maven-status  
 maven-compiler-plugin  
 compile...

testCompile ...  
 generated-test-sources  
 test-classes ...  
 surefire-reports ...

maven-archiver ...  
 Proyecto-Ejemplo-1.0-SNAPSHOT.jar  
 pom.xml

**mvn test**

target

generated-sources ...  
 classes ...  
 maven-status  
 maven-compiler-plugin  
 compile...

testCompile ...  
 generated-test-sources  
 test-classes ...  
 surefire-reports ...

maven-archiver ...  
 Proyecto-Ejemplo-1.0-SNAPSHOT.jar  
 pom.xml

## IntelliJ IDEA Ultimate

IntelliJ es un IDE muy utilizado para trabajar con diferentes tipos de aplicaciones, entre ellas aplicaciones java y Maven. Trabajaremos SIEMPRE con proyectos Maven.

En esta primera práctica empezaremos a familiarizarnos con el uso de esta herramienta. Veamos primero algunos conceptos importantes:

- **Project.** Todo lo que hacemos con IntelliJ IDEA se realiza en el contexto de un **Proyecto**. Los proyectos no contienen en sí mismos artefactos tales como código fuente, *scripts* de compilación o documentación. Son el nivel más alto de organización en el IDE, y contienen la definición de determinadas propiedades. Para los que estéis familiarizados con Eclipse, un proyecto sería similar a un *workspace* de Eclipse. La configuración de los datos contenidos en un proyecto se puede almacenar en un directorio denominado **.idea**, y es creado y mantenido automáticamente por IntelliJ.
- **Module.** Un Módulo es una unidad funcional que podemos compilar, probar y depurar de forma independiente. Los módulos contienen, por lo tanto, artefactos tales como código fuente, scripts de compilación, tests, descriptores de despliegue, y documentación. Sin embargo un módulo no puede existir fuera del contexto de un proyecto. La información de configuración de un módulo se almacena en un fichero denominado **.iml**. Por defecto, este fichero se crea automáticamente en la raíz del directorio que contiene dicho módulo. Un proyecto IntelliJ puede contener uno o varios módulos. Para los que estéis familiarizados con Eclipse, un módulo sería similar a un *projecto* de Eclipse.
- **Facet.** Las **Facetas** representan varios *frameworks*, tecnologías y lenguajes utilizados en un módulo. El uso de facetas permite descargar y configurar los componentes necesarios de los diferentes frameworks. Un módulo puede tener asociadas varias facetas. Algunos ejemplos de facetas son: Android, AspectJ, EJB, JPA, Hibernate, Spring, Struts, Web, Web Services,...
- **Run/Debug Configuration.** Podemos configurar la ejecución de determinadas acciones (como por ejemplo arrancar/parar un servidor de aplicaciones, lanzar un *script* de compilación, ...), de forma que quede guardada con un determinado nombre y la podamos lanzar a voluntad, simplemente con un *click* de ratón. IntelliJ tiene varias configuraciones predefinidas, y podemos crear nuevas configuraciones a partir de éstas. Para los que estéis familiarizados con Eclipse, una configuración de ejecución en IntelliJ sería similar al mismo concepto en Eclipse.

### ⇒ Creación de un proyecto IntelliJ a partir de un proyecto Maven existente

Una posible forma de hacerlo es a partir del **fichero pom.xml** presente en cualquier proyecto Maven. Para ello simplemente:

- Desde el menú principal elegimos File→Open (u opción Open cuando abrimos IntelliJ)
- En el cuadro de diálogo seleccionamos el fichero pom.xml, y pulsamos OK. Nos preguntará si queremos abrir como fichero o como proyecto. Elegiremos como proyecto.

De forma alternativa, también podremos usar la opción File→Open y seleccionar la **carpeta** que contiene el fichero pom.xml

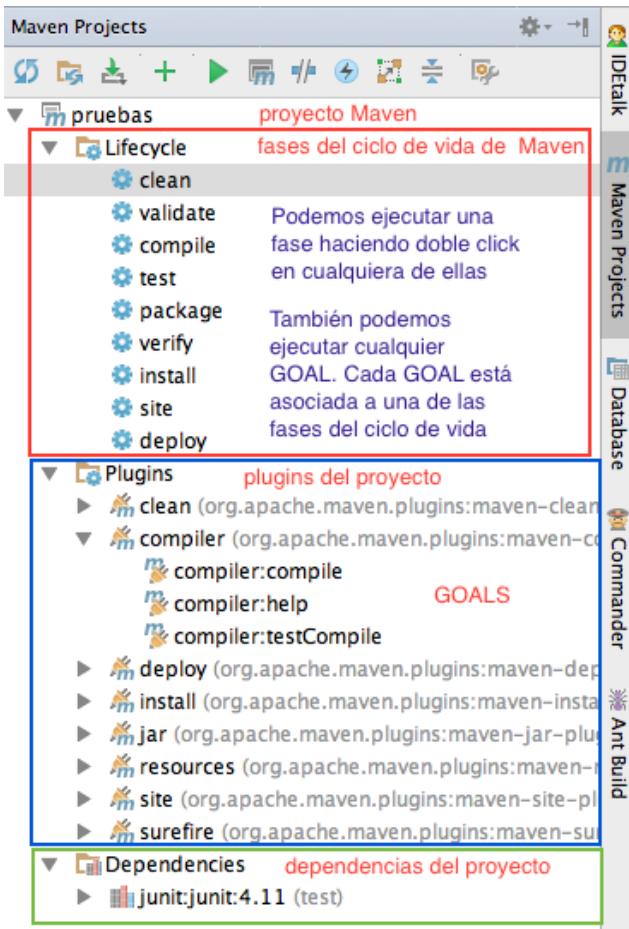
Importante: siempre que editemos la configuración que afecte de alguna manera a la construcción de nuestro proyecto (por ejemplo editando directamente el pom.xml, cambiando la ubicación y/o versión de maven del sistema...) nos aparecerá el siguiente cuadro de diálogo:



Si marcamos “enable Auto-import”, los cambios se importarán siempre de forma automática.

## ⇒ IntelliJ IDEA Maven Tool Window

IntelliJ permite mostrar diferentes “**Tool Windows**” que permiten mostrar diferentes perspectivas del proyecto. Una de estas “vistas” es la ventana de Maven (“**Maven tool window**”), que podremos mostrar si estamos trabajando con un proyecto Maven. Para mostrar la ventana tenemos que hacerlo desde **View→Tools Windows→Maven**. A continuación mostramos el aspecto de dicha ventana.



Dentro del elemento **Lifecycle**, vemos un subconjunto de fases (de los tres ciclos de vida que proporciona Maven).

Hacer doble click sobre cualquiera fase equivale a ejecutar el comando **mvn <faseX>**. Como resultado se ejecutarán todas las *goals* desde la primera fase hasta **<faseX>** que estén asociadas a cada una de ellas.

El elemento **Plugins** nos muestra los plugins asociados a las fases mostradas en el elemento **Lifecycle** por nuestro proyecto. Si añadimos algún plugin en nuestro pom también se mostrará, así como las goals que contiene. Podemos ejecutar el comando **mvn <goal>** haciendo doble click sobre cualquiera de ellas.

Podemos observar también (en gris) las coordenadas de cada plugin, y por lo tanto, sabremos la versión del mismo que estamos usando en nuestro proyecto.

El elemento **Dependencies** nos muestra todas las librerías que utiliza nuestro proyecto (bajo la etiqueta **<dependencies>** de nuestro pom.xml).

## Ejercicios

Para realizar los ejercicios de esta práctica proporcionamos, en el directorio **Plantillas-P01**, un proyecto maven (directorio **P01-IntelliJ**).

Para poder hacer los ejercicios necesitarás:

- **CLONAR** tu repositorio de Bitbucket en algún directorio, por ejemplo, supongamos que lo haces en el directorio \$HOME/practicas. En ese caso, tu directorio de trabajo, en el que debes hacer los ejercicios, será: **\$HOME/practicas/ppss-2021-Gx-apellido1-apellido2**.
- **IMPORTANTE!!!** RECUERDA que a partir de ahora, TODO lo que hagas en prácticas estará en algún subdirectorio de tu directorio de trabajo.
- **COPIA** el directorio **P01-IntelliJ** en tu directorio de trabajo y sitúate en él. A partir de aquí, se pide:

## ⇒ Ejercicio 1

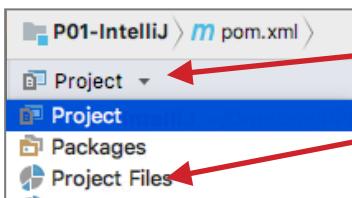
Activa tu licencia en IntelliJ. Abre el proyecto Maven P01-IntelliJ a partir del directorio que contiene el pom.xml (opción Open). Asegúrate de que la ventana **Maven** está visible. Comprueba que la plataforma SDK, versión 11 está seleccionada, desde **File→Project Structure→Platform Settings→SDKs**, y que el proyecto tiene asignada dicha versión (desde **File→Project Settings→Project→Project SDK**).

A continuación realiza lo siguiente:

Git/  
Bitbucket

Jetbrains

- A) Observa la **estructura de directorios del proyecto**. La información del proyecto puede mostrarse desde diferentes "perspectivas". Si quieres ver exactamente las carpetas físicas del disco duro debes mostrar la vista "Project Files", en lugar de "Project", que es la que tendrás por defecto (ver **Figura 1**). Puedes anotar dicha estructura en un fichero .txt para facilitar tu proceso de estudio.



La vista **Project** NO muestra la estructura física de directorios.

Para ello deberemos cambiar a la vista **Project Files**

**Figura 1.** Un mismo proyecto se puede "visualizar" de formas diferentes.

La estructura de directorios es la misma en CUALQUIER proyecto Maven. Es importante conocerla, ya que el proceso de construcción que realiza Maven asume que determinados artefactos están situados en determinados directorios. Por ejemplo, si el código fuente de las pruebas lo implementásemos en el directorio /src/main/java, no se ejecutarían dichos tests aunque lanzásemos la fase "test" de Maven

- B) Muestra en el editor la configuración de nuestro proceso de construcción (**fichero pom.xml**). Verás que el fichero xml contiene información sobre: las coordenadas, propiedades, dependencias y sobre la construcción del proyecto (etiqueta <build>)

Nuestro pom define las coordenadas de nuestro proyecto, y además hace referencia a dos artefactos: uno en la sección de dependencias y dos en la sección <build>, cada uno de los cuales tiene sus propias coordenadas. Deberías saber identificarlas. Recuerda que cualquier artefacto (fichero) generado y/o usado por maven se identifica por sus coordenadas. Nuestro proyecto maven se empaquetará de alguna forma y se generará el correspondiente artefacto, por eso tenemos que proporcionar las coordenadas de nuestro proyecto maven.

Fíjate también que los artefactos usados en nuestro pom, se usan en secciones (etiquetas) diferentes, y que cada sección tiene un propósito diferente, y por lo tanto los tipos de ficheros (artefactos) pueden ser también diferentes.

La etiqueta **<properties>** se utiliza para definir y/o asignar/modificar valores a determinadas "variables" usadas en nuestro pom.xml. Podemos usar propiedades ya predefinidas (por ejemplo la propiedad "project.build.sourceEncoding"), o podemos definir cualquier propiedad que nos interese. A partir de Maven 3 es OBLIGATORIO especificar en el pom.xml un valor para la propiedad "project.build.sourceEncoding", por lo que esta línea aparecerá en todos los ficheros pom.xml de nuestros proyectos.

Observa que hemos especificado el valor "test" para la etiqueta **<scope>** en uno de los artefactos. Dicho valor indica que el artefacto en cuestión sólo se necesita durante la compilación de los tests. Si esta etiqueta se omite, su valor por defecto es "compile" y significa que el artefacto es necesario para compilar los fuentes del proyecto.

El pom.xml de nuestra construcción incluye el plugin **maven-surefire-plugin**. Este plugin ya está incluido por defecto cuando usamos el empaquetado jar. Lo que ocurre es que la versión que se incluye por defecto es anterior a la 2.22.0 (que es la versión mínima requerida para poder compilar nuestros tests con junit5). Para ver qué versión viene incluida por defecto debes comentar el plugin en el pom, pulsar sobre el primer ícono a la izquierda de la ventana Maven Projects("Reload All Maven Projects"), y consultar la versión del plugin desde dicha ventana. Recuerda que un comentario xml empieza por "<!--" y termina por "-->".

También hemos incluido el plugin **maven-compiler-plugin**. Éste ya está incluido por defecto, pero necesitamos una versión posterior, en concreto usaremos la versión 3.8.0 para poder compilar usando jdk 11, que hemos instalado en la máquina virtual. Averigua qué versión se incluye por defecto de este plugin.

- C) Con respecto al **código fuente**, la **clase Triángulo** contiene la implementación de un método cuya especificación asociada es la siguiente: Dados tres enteros como entrada, que representan las longitudes de los tres lados de un triángulo, y cuyos valores deben estar comprendidos entre 1 y 200, el método tipo\_triangulo devuelve como resultado una cadena de caracteres indicando el tipo de triángulo, en el caso de que los tres lados formen un triángulo válido. El tipo puede ser: "Equilátero", "Isósceles", o "Escaleno". Para que los tres lados proporcionados como entrada puedan formar un triángulo tiene que cumplirse la condición de que la suma de dos de sus lados tiene que

ser siempre mayor que la del tercero. Si esto no se cumple, el método devolverá el mensaje "No es un triangulo". Si alguno de los tres lados: a, b, ó c, es mayor que 200 o inferior a 1, el método devolverá el mensaje "Valor x fuera del rango permitido", siendo x el carácter a, b, ó c, en función de que sea el primer, segundo, o tercer valor de entrada el que incumpla la condición, y con independencia de que los tres lados formen o no un triángulo.

Este es uno de los ejemplos más utilizados en la literatura sobre pruebas, quizás porque contiene una lógica clara, pero a la vez compleja. Fue utilizado por primera vez por Gruenberger en 1973, aunque en una versión algo más simple.

Fíjate que esta especificación nos proporciona el conjunto S que hemos visto en la sesión de teoría.

- D) La clase **TrianguloTest** contiene la implementación de cuatro casos de prueba (los cuatro métodos anotados con `@Test`) asociados a la especificación del apartado anterior. Después de estudiar la teoría deberías ser capaz de identificar dichos casos de prueba, y crear la correspondientes tabla. Puedes identificar cada caso de prueba como C1, C2, C3, y C4, y deberías tener claro cuántas columnas necesitas en la tabla y lo que significa cada una de ellas.

Identificador del Caso de prueba	Dato de entrada 1	...	Dato de entrada n	Resultado esperado
----------------------------------	-------------------	-----	-------------------	--------------------

Observa la implementación de cada test y verás que todos ellos siguen la misma lógica de programa. Puedes **anotar el algoritmo** que refleja dicha lógica. Recuérdalo porque lo utilizaremos también en sesiones posteriores.

## ⇒ Ejercicio 2

Vamos a "construir" el programa. En este caso sólo vamos a compilarlo. Para ello haremos doble click sobre la **fase "compile"** desde la ventana "Maven Projects". Esta acción en el IDE es equivalente a ejecutar desde línea de comando la orden: `mvn compile`. Puedes comprobar que IntelliJ ejecuta la versión de maven que hemos instalado en /usr/local de nuestra máquina virtual. Esto lo puedes hacer desde la ventana "Maven", pulsando sobre el icono que tiene un dibujo de una llave inglesa. O desde las preferencias del IDE (*Build, Execution, Deployment→Build Tools→Maven*). **Importante:** si cuando consultes la ruta de Maven ("Maven home path") aparece "Bundled Maven", cámbialo por la ruta /usr/local/apache-maven-3.6.3. Cuando ejecutes "`mvn compile`", el resultado se muestra automáticamente en una ventana en la parte inferior del IDE en donde verás los mensajes que genera Maven durante el proceso de construcción.

- A) Observa que ha aparecido un directorio nuevo en nuestro proyecto maven (en el panel de la izquierda): el directorio target. Fíjate en la nueva estructura de directorios creada, y qué artefactos contienen. Esta nueva estructura, así como la ubicación de los artefactos también es común para CUALQUIER proyecto maven, por lo que deberás conocerla. Ahora vamos a ejecutar la **fase "clean"**. Observa lo que ocurre y fíjate en las goals que se ejecutan. Ahora vuelve a compilar el proyecto. Fíjate en la secuencia de acciones que se muestran en la ventana inferior y en que NO se han ejecutado los tests

- B) Para **ejecutar los tests** vamos a hacer doble click sobre la **fase "test"**. Fíjate en la salida por pantalla, concretamente (y de momento) nos interesa la siguiente información (pulsa primero sobre el elemento P01-IntelliJ de la ventana Run para ver la pantalla de logs de Maven):

"**Tests run**" indica el número total de tests ejecutados. "**Failures**" indica el número de tests cuyo resultado esperado NO coincide con el real. Observa que junit proporciona un tercer tipo de resultado: "**Error**", del que hablaremos en sesiones posteriores. Verás que uno de los tests falla, es decir representa un fallo de ejecución (*failure*). Esto significa, como ya hemos indicado, que el valor del resultado esperado y el real NO coinciden. De hecho vemos también que por pantalla se muestra la razón del fallo de ejecución del test. Observa también algo muy importante, el resultado de la construcción es: **BUILD FAILURE**, es decir, el proceso de construcción (`mvn test`) no se ha completado con éxito puesto que se han detectado problemas en la ejecución de alguna de las goals del proceso, concretamente en la fase de pruebas.

Alternativamente, podemos ver los resultados de ejecución de los casos de prueba de forma gráfica seleccionando, desde la **ventana Maven**, el icono con una **M de color rojo** (a la derecha de la llave inglesa). Esto mostrará la ventana "Maven Test Results", en donde vemos más claramente qué test es el que ha fallado, y cuál es el resultado esperado y el real obtenido. Nota: inicialmente sólo se

muestran los tests con failures. Podemos mostrar también los tests que han pasado con éxito seleccionando el ícono “√” de la ventana Maven Test Results. Observa que cuando ejecutamos la fase “test” de maven se ejecutan TODOS los tests (es decir, también el de MatriculaTest).

- debugging**
- C) Para poder concluir nuestro proceso de construcción con éxito: BUILD SUCCESS, necesitamos eliminar el problema/s que provoca el fallo de ejecución. En este caso se trata de averiguar por qué el informe del resultado de la ejecución del caso de prueba correspondiente es un fallo. Hemos cometido un error en la implementación del método que estamos probando, que hace que el resultado esperado (resultado que debería dar si estuviese bien implementado) no es el real. Identifica la causa y modifica el código para que el resultado real sea el correcto (recuerda que este proceso se llama DEPURACIÓN, o *debugging*). A continuación vuelve a ejecutar la fase test. Tendrás que volver a pulsar sobre el ícono con una M para ver actualizados los resultados de forma gráfica. Repite el proceso hasta que los cuatro tests estén en “verde”, y el proceso de construcción termine con: BUILD SUCCESS.
- D) Observa qué tienen en común el test C1 y un posible test adicional C5 con datos de entrada: a=7,b=7,c=7, y piensa en la conveniencia o no de incluir C5 al conjunto de tests. De la misma forma razona si son necesarios los tests C2 y C3. Añade dos posibles casos de prueba adicionales que “aporten valor” al conjunto de casos de prueba (no sean innecesarios).

**Test Case Design**

### ⇒ Ejercicio 3

La clase **Matricula** contiene el método **calculaTasaMatricula()** que devuelve el valor de las tasas de matriculación de un alumno en función de la edad, de si es familia numerosa, y si es o no repetidor, de acuerdo con la siguiente tabla (asumiendo que se aplican sobre un valor inicial de tasa=500 euros):

	Edad < 25	Edad < 25	Edad 25..50	Edad 51..64	Edad ≥ 65
Edad	SI	SI	SI	SI	SI
Familia Numerosa	NO	SI	SI		
Repetidor	SI				
Valor tasa-total	tasa + 1500	tasa/2	tasa/2	tasa -100	tasa/2

**Test Case Design**

- A) En este caso, hemos proporcionado la implementación de un único test, en la clase MatriculaTest. Rellena una tabla de casos de prueba con el test que hay implementado y piensa 5 nuevos tests que no sean “redundantes” y añádelos a la tabla. En tu opinión, ¿6 tests son suficientes o deberíamos añadir alguno más?

- B) Implementa los casos de prueba que has añadido a la tabla y ejecuta todos los tests. Recuerda que si encuentras algún error debes depurarlo.

**Test Case Code/Run**

Observa que primero (apartado A), tenemos que obtener (decidir) los casos de prueba que usaremos para comprobar que el programa funciona correctamente. Para ello tenemos que “elegir” datos de entrada concretos, y asociar un resultado esperado, de acuerdo con el comportamiento correspondiente de la especificación.

A continuación (apartado B) tenemos que implementar los tests, y finalmente ejecutarlos. Cuando obtengamos el informe de JUnit podremos ver si hemos detectado defectos en nuestro programa o no. Es muy importante que te quede claro que un informe JUnit sin errores no significa que el programa esté libre de ellos. Es decir, nuestras pruebas sólo pueden demostrar la presencia de errores (no la ausencia de los mismos).

## ⇒ Ejercicio 4

Hasta ahora hemos lanzado la ejecución de las fases del ciclo de vida de maven “clean”, “compile” y “test”. Vamos a ejercitarnos en otras dos fases importantes: la fase “package”, y la fase “install”.

**Maven  
Build  
(package)**

- Ejecuta la **fase “package”** y observa los cambios en la ventana del proyecto. Fíjate en qué acciones se han llevado a cabo para construir el proyecto y qué artefactos nuevos se han generado. Es importante que tengas claro qué artefactos se generan en cada fase, para poder utilizar de forma adecuada maven. Modifica uno de los tests, de forma que dé un resultado fallido, ejecuta la fase “clean”, y a continuación la fase “package” de nuevo, y observa lo que ocurre. De igual forma, ahora, en lugar de introducir un error en los tests, edita el fichero Matricula.java, quita un punto y coma para provocar un error de compilación y vuelve a ejecutar las fases “clean” y “package”. Tienes que tener claro el resultado obtenido y por qué se obtiene dicho resultado.
- Vuelve a reparar todos los errores introducidos y ejecuta la **fase “install”**. En este caso el resultado es menos “obvio” ya que en esta fase se “instala” (copia) el artefacto generado en la fase anterior en el repositorio local. El repositorio local de maven se encuentra en `$HOME/.m2/repository`. En el repositorio local se almacenan todos los artefactos que maven ha utilizado para construir el proyecto, más todos aquellos artefactos que hayamos “instalado”, por ejemplo, utilizando la fase install. Deberías saber la ruta exacta de cada artefacto a partir de sus coordenadas. Si borras el directorio `.m2` no importa, maven lo volverá a crear automáticamente durante la próxima ejecución del comando mvn. La primera vez que ejecutemos maven, si el repositorio local está vacío, maven se descarga todos aquellos ficheros que necesita de sus repositorios remotos. Esto significa que, a medida que vayas ejecutando maven y necesitando los artefactos (ficheros jar, war, ear, pom,...) para construir el proyecto, tu repositorio local irá creciendo. Si maven encuentra en el repositorio local el artefacto correspondiente, no será necesario proceder a su descarga, por lo que se reducirá el tiempo de construcción del proyecto.

**Maven  
Build  
(install)**

## Guardamos nuestro trabajo en Bitbucket

Recuerda que debes guardar TODO tu trabajo de prácticas en Bitbucket. Deberías “subir” a Bitbucket los ejercicios según los vas realizando (no esperes a tenerlos todos, podrías perder tu trabajo si tienes algún problema con la máquina virtual).

Para ello simplemente debes usar los comandos git que ya hemos visto, desde un terminal y **desde tu directorio de trabajo** (ppss-2021-Gx-apellido1-apellido2):

**Git/  
Bitbucket**

- `git add .`
- `git commit -m"Ejercicio P01-X terminado"`
- `git push`

## Resumen



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

GIT

- Herramienta de gestión de versiones. Nos permite trabajar con un repositorio local que podemos sincronizar con un repositorio remoto.

MAVEN

- Herramienta automática de construcción de proyectos java.. El “build script” se especifica de forma declarativa en el fichero pom.xml, en el que encontramos varias partes bien diferenciadas. Los artefactos maven generados se identifican mediante sus coordenadas.
- Los proyectos maven requieren una estructura de directorios fija. El código de los tests está físicamente separado del código fuente de la aplicación.
- Maven usa diferentes ciclos de vida. Cada ciclo de vida está formado por una secuencia ordenada de fases, cada fase puede tener asociadas unas goals. El resultado del proceso de construcción maven puede ser “Build failure” o “Build success”.

TESTS

- Un test se basa en un caso de prueba, el cual tiene que ver con el comportamiento especificado del elemento a probar.
- Una vez definido el caso de prueba para un test, éste puede implementarse como un método java anotado con @Test (anotación JUnit). Los tests se compilan y se ejecutan en diferentes momentos durante el proceso de construcción de un proyecto.
- El algoritmo de un test es siempre el mismo. Un test comprueba, dadas unas determinadas entradas sobre el elemento a probar, si su ejecución genera un resultado idéntico al esperado (es decir, se trata de comprobar si el comportamiento especificado en S coincide con el comportamiento implementado P)
- Dependiendo de cómo hayamos diseñado los casos de prueba, detectaremos más o menos errores (discrepancias entre el resultado esperado y el resultado real).

TEORIA ————— PRACTICA

# P01B- Diseño de pruebas de caja blanca

Os recordamos que el plazo para entregar los ejercicios de esta práctica (P01B) termina justo ANTES de comenzar la siguiente sesión de prácticas (P02) en el laboratorio (los grupos de los lunes, por lo tanto, el siguiente lunes, los de los martes, el siguiente martes, ...). Pasado este plazo los profesores no revisarán vuestro trabajo sobre P01B (de forma individual), con independencia de que, con carácter general, se indiquen en clase las soluciones que os permitirán saber en cualquier otro momento si las vuestras son correctas.

## Diseño de pruebas de caja blanca (*structural testing*)

En esta sesión aplicaremos el método de **diseño** de casos de prueba visto en clase (método del camino básico) para obtener un conjunto de casos de prueba de unidad (recuerda que hemos definido una unidad como un método java). Usaremos la implementación (conjunto P) para determinar el conjunto de datos de entrada. Ojo! Nunca uses la implementación para indicar el resultado esperado, por razones obvias.

Cuando apliques el método, anota SIEMPRE los PASOS que vas siguiendo e indica, de forma explícita qué es lo que se pretende con dicho paso. La idea es que las prácticas te ayuden a entender bien lo que hemos explicado en clase de teoría.

El método del camino básico tiene un objetivo que es común a todos los métodos de diseño de casos de prueba, pero también tiene un objetivo particular, que lo diferencia del resto de métodos. Cuando acabes la práctica te deben quedar muy claros ambos objetivos.

Tal y como hemos indicado en clase, cada sesión de prácticas está pensada para ayudarte a entender bien los conceptos explicados en la clase de teoría correspondiente. Por lo tanto, una vez acabada la práctica, tendrás que ser capaz de contestar y justificar razonadamente preguntas como: ¿por qué puedo obtener tablas diferentes aplicando el mismo método? ¿ambas tablas son igual de válidas? ¿por qué tienen que ser caminos independientes los caminos obtenidos a partir del grafo?, ¿qué implicaciones tiene el proporcionar menos caminos que los indicados por CC?....

En esta sesión no utilizaremos ningún software específico, pero en las siguientes sesiones automatizaremos la ejecución de los casos de prueba que hemos diseñado en esta práctica.

## Bitbucket

El trabajo de esta sesión debes subirlo a *Bitbucket*. Todo el trabajo de esta práctica deberá estar en el subdirectorio **P01B-CaminoBasico**, dentro de tu espacio de trabajo, (ppss-2021-Gx-apellido1-apellido2).

El trabajo de esta práctica debéis guardarla en documentos de **imagen (jpg, png)**, o en formato de **texto (.txt, .md)**.

## Ejercicios

Puedes hacer los ejercicios en papel y subir luego una foto a Bitbucket (en formato png o jpg). Si prefieres usar alguna herramienta, puede ser útil ésta: <https://www.draw.io>. Desde aquí podrás hacer todo el ejercicio: tanto los grafos, como la tabla, texto, etc. Cuando guardes tu trabajo, hazlo tanto en el formato por defecto (xml), por si quieras modificarlo en cualquier momento, como en formato png o jpg (para poder visualizarlo directamente desde Bitbucket).

A continuación proporcionamos la especificación y el código de las unidades a probar.

## ⇒ ⇒ Ejercicio 1

Crea la subcarpeta "**Ejercicio1**", en donde guardarás tu solución de este ejercicio. Queremos diseñar los casos de prueba para el método **calculaTasaMatricula()** (la **especificación** es la misma que la del ejercicio 3 de la práctica anterior). Dicha unidad calcula y devuelve el valor de las tasas de matriculación en función de la edad, de si es familia numerosa, y si es o no repetidor, de acuerdo con la siguiente tabla (asumiendo que se aplican siempre sobre un valor inicial de tasa=500 euros) :

	Edad < 25	Edad < 25	Edad 25..50	Edad 51..64	Edad ≥ 65
Edad	SI	SI	SI	SI	SI
Familia Numerosa	NO	SI	SI		
Repetidor	SI				
Valor tasa-total	tasa + 1500	tasa/2	tasa/2	tasa -100	tasa/2

La tabla anterior se "lee" por columnas. Por ejemplo, para la primera columna: "Si edad<25, y No familia numerosa y es repetidor" entonces el resultado es 500+1500. Las casillas en blanco indican que dicha condición es indiferente, por ejemplo "Si edad <25 y familia numerosa" independientemente de si es repetidor o no el resultado será 500/2.

Siempre se parte de un valor mínimo de tasa de 500, por lo que si ninguna de las entradas está especificada en la tabla, el resultado esperado será de 500-

Diseña los casos de prueba utilizando el método del camino básico que hemos visto en clase teniendo en cuenta que la **implementación** es la siguiente:

```

1. public float calculaTasaMatricula(int edad, boolean familiaNumerosa,
2.                                     boolean repetidor) {
3.     float tasa = 500.00f;
4.
5.     if ((edad < 25) && (!familiaNumerosa) && (repetidor)) {
6.         tasa = tasa + 1500.00f;
7.     } else {
8.         if ((familiaNumerosa) || (edad >= 65)) {
9.             tasa = tasa / 2;
10.        }
11.        if ((edad > 50) && (edad < 65)) {
12.            tasa = tasa - 100.00f;
13.        }
14.    }
15.    return tasa;
16.}
```

Debes subir tu solución a Bitbucket.

**NOTA:** Recuerda que los datos de entrada y salida esperada deben ser siempre valores concretos. Fíjate que en la tabla hay casillas en blanco, eso significa que dichos valores de entrada no afectan al resultado esperado. Aún así, tendremos que decidir un valor concreto cuando obtengamos la tabla de casos de prueba.

**NOTA:** Independientemente del método de DISEÑO de casos de prueba que usemos, todos ellos proporcionan un conjunto de casos de prueba efectivo y eficiente (evidenciar el máximo número posible de errores, con el mínimo número de pruebas), teniendo en cuenta un objetivo concreto.

En el caso del método del camino básico, el valor de complejidad ciclomática (CC) determina el número MÁXIMO de filas de la tabla. Para que el conjunto de casos de prueba sea eficiente, deberíamos generar el mínimo número de caminos con los que conseguimos recorrer todos los nodos y todas las aristas del grafo (ese número puede ser inferior al de CC).

En cualquier caso, en clase hemos dicho que vamos a considerar válida cualquier solución que contenga un número de caminos independientes menor o igual que el valor de CC. Un camino es independiente si añade al conjunto de caminos al menos un nodo o una arista que no se había recorrido ANTES.

## ▷▷ Ejercicio 2

Crea la subcarpeta "**Ejercicio2**", en donde guardarás tu solución de este ejercicio.

Se proporciona la siguiente **especificación** para el método *buscarTramoLlanoMasLargo()*:

Dada una colección de enteros, que representan lecturas de la altura de un terreno, tomadas a intervalos equidistantes de 1 kilómetro (la medida se toma siempre al inicio del intervalo), se trata de detectar cuál es el tramo llano más largo de esas lecturas, y devolver un objeto de tipo Tramo con el origen (número de kilómetro donde comienza el llano) y la longitud (número de kilómetros) del llano encontrado. Consideraremos que hemos detectado un llano cuando haya dos o más kilómetros consecutivos con la misma altura. Además, tendremos en cuenta las siguientes consideraciones:

- La lista de lecturas nunca va a tener el valor null
- El primer kilómetro se considera como kilómetro cero
- Si hay varios llanos con la misma longitud, devolveremos el primero de ellos
- Si no hay ningún llano, se devolverá un Tramo con origen: 0, y con longitud: 0
- Los llanos pueden estar por debajo, por encima, o a nivel del mar (altura 0).

Proporcionamos la siguiente **implementación** asociada a la especificación anterior:

```

1. public Tramo buscarTramoLlanoMasLargo(ArrayList<Integer> lecturas) {
2.     int lectura_anterior =-1;
3.     int longitud_tramo =0, longitudMax_tramo=0;
4.     int origen_tramo=-1, origen_tramoMax=-1;
5.     Tramo resultado = new Tramo(); //el origen y la longitud es CERO
6.
7.     for(Integer dato:lecturas) {
8.         if (lectura_anterior== dato) {//detectamos un llano
9.             longitud_tramo++;
10.            if (origen_tramo == -1 ) {//marcamos su origen
11.                origen_tramo = lecturas.indexOf(dato);
12.            }
13.        } else { //no es un llano o se termina el tramo llano
14.            longitud_tramo=0;
15.            origen_tramo=-1;
16.        }
17.        //actualizamos la longitud máxima del llano detectado
18.        if (longitud_tramo > longitudMax_tramo) {
19.            longitudMax_tramo = longitud_tramo;
20.            origen_tramoMax = origen_tramo;
21.        }
22.        lectura_anterior=dato;
23.    }
24.    switch (longitudMax_tramo) {
25.        case -1:
26.        case 0: break;
27.        default: resultado.setOrigen(origen_tramoMax);
28.                   resultado.setLongitud(longitudMax_tramo);
29.    }
30.
31.    return resultado;
32.}

```

Se pide:

- Representa el CFG asociado al código anterior, calcula su CC, y obtén el conjunto de caminos independientes.
- Selecciona datos de entrada para recorrer todos y cada uno de los caminos obtenidos, verás que aparece un camino que es **IMPOSIBLE** de recorrer con ningún dato de entrada (**pista**: el problema está en las líneas 24..29). ¿Debemos simplemente ignorar dicho camino?
- Modifica el código para eliminar las sentencias que nunca se van a ejecutar. Representa el nuevo grafo y obtén el conjunto de casos de prueba (aplicando el método del camino básico). Piensa en si podemos asegurar que el código no tiene defectos). Tiene que quedarte claro por qué la respuesta es no.

Sube tu solución a Bitbucket

### ⇒ ⇒ Ejercicio 3: método *realizaReserva()*

Crea la subcarpeta "**Ejercicio3**", en donde guardarás tu solución de este ejercicio.

Se proporciona la siguiente **especificación** para el método ***realizaReserva()***:

Se quiere llevar a cabo la reserva de una serie de libros de un socio de una biblioteca, el método recibe por parámetro el login y password del empleado de la biblioteca (que será el que realice la reserva), un identificador de un socio de la misma, y una colección de isbn's de los libros que quiere reservar. Solamente un empleado de la biblioteca con rol de bibliotecario puede realizar la reserva.

La reserva propiamente dicha (para cada uno de los libros) se hace efectiva en **otro método** (invocado desde *realizaReserva*), el cual puede lanzar varias excepciones, de forma que devolverá:

- la excepción **IsbnInvalidoException**, con el mensaje "ISBN invalido:<isbn>", si el isbn del libro que se quiere reservar no existe en la base de datos de la biblioteca (siendo <isbn> el isbn de dicho libro).
- la excepción **SocioInvalidoException**, con el mensaje "SOCIO invalido", si el identificador del socio no existe en la base de datos. En ese caso, no se podrá hacer efectiva la reserva para ninguno de los libros de la lista.
- la excepción **JDBCException**, con el mensaje "CONEXION invalida", si no se puede acceder a la BD.
- si todo va bien y se puede hacer la reserva, el método invocado termina normalmente (no devuelve nada)

El método ***realizaReserva*** termina normalmente (sin devolver nada) si todo va bien y se realiza la reserva de todos los libros de la lista pasada como parámetro. En el caso de que no se pueda hacer efectiva la reserva de algún libro, el método *realizaReserva()* devolverá una excepción de tipo **ReservaException**, con un mensaje formado por todos los mensajes de las excepciones generadas durante el proceso de reserva de cada libro, separados por ";".

Por ejemplo: suponiendo que el login y password del bibliotecario son "biblio", "1234", que el identificador de socio proporcionado existe en la base de datos, que la lista de isbn's a reservar es (12345, 23456, 34567), y que el segundo y tercer isbn no están en la base de datos, el método *realizaReserva* devolverá como resultado una excepción de tipo **ReservaException** con el mensaje: "ISBN invalido: 23456; ISBN invalido: 34567;"

En la Figura 1 se muestra una **implementación** del método *realizaReserva()*.

A partir del código y la especificación proporcionadas, diseña una tabla de casos de prueba para el método *realizaReserva()* usando el método del camino básico.

**NOTA:** En este ejercicio hay comportamientos programados que NO hemos especificado. Cuando esto ocurre, se usa un interrogante (?) como valor del comportamiento esperado. Con ello estamos indicando que no es responsabilidad del tester el completar la especificación o modificarla en modo alguno. Ante esta situación, el diseño de ese caso de prueba quedará pendiente hasta que complete la especificación por quien corresponda,

Recuerda subir tu solución a Bitbucket.

```

1. public void realizaReserva(String login, String password,
2.                             String socio, String [] isbn) throws Exception {
3.     ArrayList<String> errores = new ArrayList<String>();
4.     //El método compruebaPermisos() devuelve cierto si la persona que hace
5.     //la reserva es el bibliotecario y falso en caso contrario
6.     if(!compruebaPermisos(login, password, Usuario.BIBLIOTECARIO)) {
7.         errores.add("ERROR de permisos");
8.     } else {
9.         FactoriaB0s fd = FactoriaB0s.getInstance();
10.        //El método getOperacionB0() devuelve un objeto de tipo IOperacionB0
11.        //a partir del cual podemos hacer efectiva la reserva
12.        IOperacionB0 io = fd.getOperacionB0();
13.        try {
14.            for(String isbn: isbn) {
15.                try {
16.                    //El método reserva() registra la reserva de un libro (para un socio)
17.                    //dados el identificador del socio e isbn del libro a reservar
18.                    io.reserva(socio, isbn);
19.                } catch (IsbnInvalidoException iie) {
20.                    errores.add("ISBN invalido" + ":" + isbn);
21.                }
22.            }
23.        } catch (SocioInvalidoException sie) {
24.            errores.add("SOCIO invalido");
25.        } catch (SQLException je) {
26.            errores.add("CONEXION invalida");
27.        }
28.    }
29.    if (errores.size() > 0) {
30.        String mensajeError = "";
31.        for(String error: errores) {
32.            mensajeError += error + "; ";
33.        }
34.        throw new ReservaException(mensajeError);
35.    }
36. }

```

Figura 1. Implementación del método *realizaReserva()*

## Resumen



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



### MÉTODOS DE DISEÑO ESTRUCTURALES

- Permiten seleccionar un subconjunto de comportamientos a probar a partir del código implementado. Sólo se aplican a nivel de unidad, y son técnicas dinámicas.
- No pueden detectar comportamientos especificados que no han sido implementados.

### MÉTODO DE DISEÑO DEL CAMINO BÁSICO

- Usa una representación de grafo de flujo de control (CFG).
- El objetivo es proporcionar el número mínimo de casos de prueba que garanticen que estamos probando todas las líneas del programa, y todas las condiciones en sus vertientes verdadera y falsa, al menos una vez.
- A partir del grafo, el valor de CC indica una cota superior del número de casos de prueba a obtener.
- Una vez que obtenemos el conjunto de caminos independientes, hay que proporcionar los casos de prueba que ejerzan dichos caminos (admitiremos que el número de caminos independientes sea  $\leq$  que CC en todos los casos).
- Es posible que haya caminos imposibles o que detectemos comportamientos implementados pero no especificados.
- Los datos de entrada y los datos de salida esperados siempre deben ser concretos.
- Las entradas de una unidad no tienen por qué ser los parámetros de dicha unidad.
- El resultado esperado siempre debemos obtenerlo de la especificación de la unidad a probar.

# 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 para automatizar la ejecución de 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 (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 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 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). Es importante que tengas clara la secuencia de acciones de dicho proceso de construcción .

IntelliJ puede usar Maven, y también puede ejecutar los tests junit (sin usar maven) desde el menú contextual de cada fuente de test. IntelliJ controla los comandos maven, la versión usada, ... 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/share 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 deberá estar en el directorio **P02-Drivers**, dentro de tu espacio de trabajo, es decir, dentro de tu carpeta: ppss-2021-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 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 proporcionaremos 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` (recuerda que es obligatorio especificar esta propiedad).
- ⌚ 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` para modificar la versión a la 3.8.0. Recuerda que por defecto se incluye una versión anterior con la que no podrás compilar usando jdk 11.
- ⌚ Igualmente incluiremos el plugin `surefire` ("`maven-surefire-plugin`") para usar la versión 2.22.2.
- ⌚ Añade la dependencia para usar junit (la tienes en las transparencias de teoría, recuerda que sólo necesitas usar la que hemos indicado en las transparencias de teoría).

**Importante:** Siempre debes modificar el pom.xml sin ayuda del IDE. Y debes tener claro para qué sirve cada elemento incluido en el mismo. Este fichero nos permite configurar la construcción del proyecto e incluir la automatización de nuestras pruebas en dicho proceso. La idea es que con sólo “pulsar un botón” (es decir, ejecutar UN comando maven) podamos compilar nuestros fuentes, compilar los tests, ejecutar los tests,...

La configuración que acabamos de hacer será la configuración básica que vamos a usar, a partir de ahora en todos los proyectos.

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

### ⇒ ⇒ Ejercicio 1: drivers para `calculaTasaMatricula()`

Debes implementar y automatizar la ejecución de los casos de prueba que has diseñado en la sesión anterior para el método `ppss.Matricula.calculaTasaMatricula()`.

**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

Tareas a realizar:

**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.

**drivers** B) **Implementa los drivers** asociados a tu tabla de diseño de casos de prueba (o usa la tabla que os hemos proporcionado). IntelliJ nos permite generar la clase de pruebas de la siguiente forma: 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, compíalos 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 que tener 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) **Ejecuta los tests** desde la fase Maven correspondiente. Deberías ver en el informe Maven que se han ejecutado todos los tests. Para 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. En clase hemos explicado dónde se encuentran físicamente los fuentes, ejecutables e informes maven. Comprueba que verdaderamente es así.

D) **Implementa** en una nueva clase de pruebas con nombre "**MatriculaParamTest**". Se trata de implementar 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.

**Ejecuta** de nuevo la fase test de Maven. Si has usado la misma tabla que para el apartado B) 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:

**TABLA A**

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

	Datos de entrada	Resultado esperado
	lista de lecturas	Tramo
C1A	{3}	Tramo.origen =0 ; Tramo.longitud = 0
C2A	{100,100}	Tramo.origen = 0; Tramo.longitud = 1
C3A	{180, 180, 180}	Tramo.origen=0; Tramo.longitud=2

**SUT** A) Añade el código fuente de la unidad en el **paquete ppss**, en la clase **Llanos**. El código es el proporcionado en el enunciado de la práctica anterior (puedes copiarlo de la carpeta proporcionada **Plantillas-P02**). 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. Para generar los constructores y los métodos puedes hacerlo desde el editor, y seleccionar desde el menú contextual "Generate...", seleccionar "Conctructor", y "Getter and Setter" respectivamente.

**drivers**  
**TABLA A**

- B) Implementa los **drivers** asociados a tu tabla de diseño de casos de prueba (o usa la tabla que os hemos proporcionado, TABLA A). 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.

- 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.

**drivers**  
**TABLA B**

**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**

**test**  
**parametrizado**

- 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..
- 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 (si has usado las mismas tablas en ambos casos)

### ⇒ ⇒ 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, ...
```

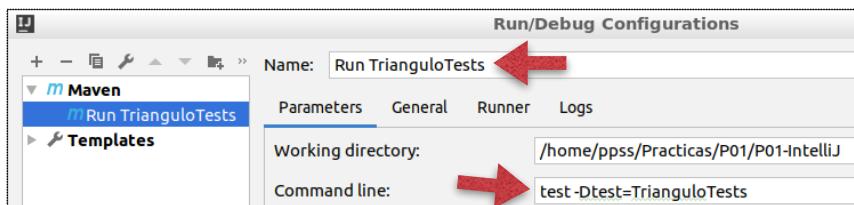


Desde IntelliJ podemos ejecutar cualquier comando Maven, pulsando sobre el icono correspondiente de la ventana Maven. No es necesario poner explícitamente el comando "mvn", sino la fase/s goal/s que queremos ejecutar, separadas por espacios.

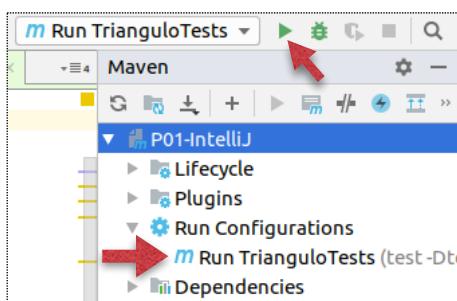
Otra opción que nos resultará más útil, si por ejemplo queremos usar varios comandos mvn de forma frecuente es crearnos elementos "Configuration", de forma que podemos tener "guardados" diferentes comandos maven para poder usarlos cuando queramos, simplemente pulsando un botón.

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

**Add Configuration...**



IntelliJ nos proporciona "plantillas" para poder usarlas con diferentes tipos de proyectos. Nosotros crearemos "Configurations" de tipo **Maven** (pulsando sobre el ícono +). 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.

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

**Nota:** Cada vez que, desde el menú contextual del fichero java de un *driver*, ejecutamos 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.

### INSTRUCCIONES para GUARDAR LAS CONFIGURACIONES DE INTELLIJ (Run Configurations)

IntelliJ guarda todas las configuraciones que hemos creado en el fichero .idea/workspace.xml.

Por otro lado si muestras el contenido del fichero .gitignore (que se generó automáticamente al crear el repositorio), verás que aparece lo siguiente:

```
# JetBrains IDE
.idea/
```

es decir, que el directorio **.idea** y todo su contenido será ignorado por git, eso significa que no vamos a guardar en nuestro repositorio local ni en el remoto una copia del fichero workspace.xml.

Si desde otro ordenador desde el que habitualmente usas para hacer las prácticas, clonas tu repositorio remoto para seguir trabajando, no tendrás las *Run Configurations* que has creado en el otro ordenador, y por lo tanto tendrás que crearlas de nuevo siquieras usarlas.

Queremos ignorar los ficheros de ese directorio, pero estamos interesados en "conservar" las configuraciones, para poder seguir usándolas si nos vamos a otro ordenador y abrimos el proyecto con IntelliJ. Por lo tanto, lo que haremos será provocar que IntelliJ guarde dichas configuraciones en otro fichero "fuera" del directorio .idea.

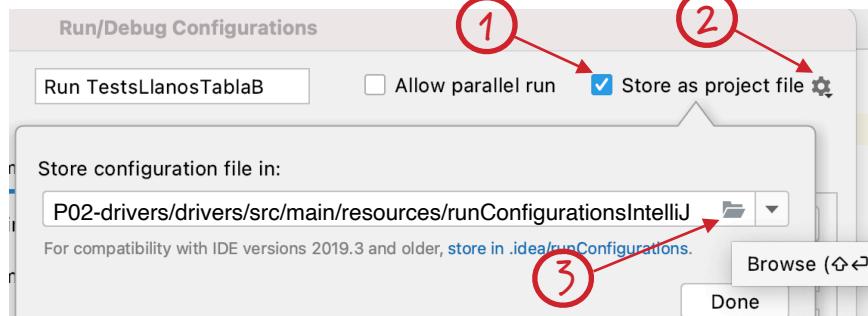
Concretamente las guardaremos en el directorio: **src/main/resources/runConfigurationsIntelliJ**.

El directorio **src/main/resources** es un directorio Maven, pensado para contener cualquier fichero adicional de nuestro proyecto que no sea código java (y que puede ser usado por nuestro código, o no, como en nuestro caso). Dentro de la carpeta src/main/resources podemos crear tantos subdirectorios como consideremos oportuno. Nosotros hemos decidido guardar las configuraciones en el subdirectorio **runConfigurationsIntelliJ**

Para ello:

- En el directorio `src/main/resources`, crea la carpeta ***runConfigurationsIntelliJ***. Puedes hacerlo dese la vista Projects, desde el menú contextual del elemento "Resources", accediendo a *New→Directory*

- Cuando añadas una nueva configuración debes marcar la casilla "***Store as project file***" e indicar la ruta donde guardaremos los ficheros asociados.



De esta forma, podremos guardar en Bitbucket dichos elementos, y seguir ignorando el directorio `.idea`.

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

- 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`
- 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***"
- 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

En el directorio **Plantillas-P02** encontrarás el fichero ***DataArray.java*** con una implementación para el método `ppss.DataArray.delete(int)`.

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 coincide 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:

	Entradas		Resultado esperado
ID	colección, numElem	Elemento a borrar	(colección + numElem) o excepción de tipo DataException
F1	[1,3,5,7], 4	5	[1,3,7], 3
F2	[1,3,3,5,7], 5	3	[1,3,5,7], 4
F3	[1,2,3,4,5,6,7,8,9,10], 10	4	[1,2,3,5,6,7,8,9,10], 9
F4	[ ], 0	8	DataException(m1)
F5	[1,3,5,7], 4	-5	DataException(m2)
F6	[ ], 0	0	DataException(m3)
F7	[1,3,5,7], 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"

Dada la especificación anterior del método `delete()`, 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 "Run Configuration" para poder ejecutar únicamente la clase **DataArrayTest**, con el nombre "**Run dataArrayTest**" (no es necesario que uses la anotación @Tag). Recuerda que tienes que guardar la nueva configuración en el directorio **runConfigurationsIntelliJ**, en la subcarpeta *resources* (del directorio *main*).

#### ⇒ ⇒ ANEXO 1: 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	reserva()	isbns	Resultado esperado
C1	"xxxx"	"xxxx"	"Luis"	(1)	{"111111"}	??
C2	"ppss"	"ppss"	"Luis"	(2)	{"111111", "222222"}	No se lanza excep.
C3	"ppss"	"ppss"	"Luis"	(3)	{"333333"}	ReservaException2
C4	"ppss"	"ppss"	"Pepe"	(4)	["111111"]	ReservaException3
C5	"ppss"	"ppss"	"Luis"	(5)	{"111111"}	ReservaException4

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 "111111", "222222".

ReservaException2: Excepción de tipo ReservaException con el mensaje: "ISBN invalido:33333;"

ReservaException3: Excepción de tipo ReservaException con el mensaje: "SOCIO invalido;"

ReservaException4: Excepción de tipo ReservaException con el mensaje: "CONEXION invalida;"

(1): No se invoca al método reserva()

(2): El método reserva() NO lanza ninguna excepción

(3): El método reserva() lanza la excepción ISBNInvalidoException

(4): El método reserva() lanza la excepción SocioInvalidoException

(5): El método reserva() lanza la excepción ConexionInvalidaException

## ⇒⇒ ANEXO 2: 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
- 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 mostrada en el anexo 1).

### 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 surefiretest 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 y qué ficheros genera nuestro proceso de construcción en cada caso.

# P03- Diseño de pruebas de caja negra

Os recordamos que el plazo para entregar los ejercicios de esta práctica (P03) termina justo ANTES de comenzar la siguiente sesión de prácticas (P04) 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 P03 (de forma individual), con independencia de que se indiquen en clase las soluciones que os permitirán saber si las vuestras son correctas.

## Diseño de pruebas de caja negra (*functional testing*)

En esta sesión aplicaremos el método de **diseño** de casos de prueba de caja negra visto en clase para obtener conjuntos de casos de prueba de unidad (método java), partiendo de la especificación de dicha unidad (**conjunto S**). Recuerda que no sólo se trata reproducir los pasos de los métodos de forma mecánica, sino que, además, debes saber qué es lo que estás haciendo en cada momento, para así asimilar los conceptos explicados.

Es importante tener presente el objetivo particular del método de diseño usado (**particiones equivalentes**), y que cualquier método de diseño nos proporciona una forma sistemática de obtener un conjunto de casos de prueba eficiente y efectivo. Obviamente, esa "sistematicidad" tiene que "verse" claramente en la resolución del ejercicio, por lo que tendrás que dejar MUY CLAROS todos y cada uno de los pasos que vas siguiendo y seguir las normas explicadas en clase sobre cómo indicar las entradas, salidas, particiones, ....

Insistimos de nuevo en que el trabajo de prácticas tiene que servir para entender y asimilar los conceptos de la clase de teoría, y no al revés.

En esta sesión no utilizaremos ningún software específico, pero en la siguiente automatizaremos la ejecución de los casos de prueba (unitarias) que hemos diseñado aquí, por lo que necesitarás tus soluciones de esta práctica para poder trabajar en la próxima clase

## Bitbucket

El trabajo de esta sesión también debes subirlo a *Bitbucket*. Todo el trabajo de esta práctica deberá estar en el directorio **P03-CajaNegra**, dentro de tu espacio de trabajo, es decir, dentro de tu carpeta: ppss-2021-Gx-apellido1-apellido2. Puedes subir los ficheros en formato png, jpg, pdf, texto, o con extensión .md.

Cada uno de los ejercicios debería estar contenido en un único fichero. Si hay más de un fichero por ejercicio, debería estar claro en el nombre del fichero lo que contiene cada uno.

## Ejercicios

A continuación proporcionamos la especificación de las unidades a probar (hemos definido una unidad como un método java). Se trata de **diseñar los casos de prueba** para cada una de las especificaciones utilizando el método de diseño de **particiones equivalentes**. Recuerda indicar claramente las entradas, salidas, las agrupaciones o no de las entradas, las particiones válidas y no válidas de cada una de las entradas/salidas y/o agrupaciones, combinaciones de particiones, y los valores concretos para las entradas y salidas en la tabla de casos de prueba, así como identificar las combinaciones de particiones usadas en cada caso de prueba. Recuerda que es imprescindible indicar las asunciones sobre los datos de entrada de la tabla en el caso de que sea necesario, y que la tabla siempre tiene que tener valores concretos.

▷▷ Ejercicio 1: especificación *importe\_alquiler\_coche()*

Crea la subcarpeta "**Ejercicio1**", en donde guardarás tu solución de este ejercicio.

En una aplicación de un negocio de alquiler de coches necesitamos una unidad denominada **importe\_alquiler\_coche()**. Dicha unidad calcula el importe del alquiler de un determinado tipo de coche durante un cierto número de días, a partir de una fecha concreta, y devuelve el importe del alquiler. Si no es posible realizar los cálculos devuelve una excepción de tipo ReservaException. El prototipo del método es el siguiente:

TipoCoche es un tipo enumerado cuyos posibles valores son: (TURISMO, DEPORTIVO). Nos indican que si la fecha de inicio proporcionada no es posterior a la actual, entonces se lanzará la excepción ReservaException con el mensaje "Fecha no correcta". Si el tipo de coche no está disponible durante los días requeridos, o se intenta hacer una reserva de más de 30 días, entonces se lanzará la excepción ReservaException con el mensaje "Reserva no posible". El precio de la reserva por día depende del número de días reservados, según la siguiente tabla:

1 día	100 euros
2 días o más	50 euros/día

Diseña los casos de prueba para la especificación anterior utilizando el método de particiones equivalentes.

▷ Ejercicio 2: especificación *generaTicket()*

En una aplicación de un comercio, tenemos un método que genera tickets de venta en función de los artículos comprados por un determinado cliente. Concretamente, el prototipo del método será el siguiente:

Para dicho método, dados un cliente y la lista de artículos que desea comprar (identificados por su código), el método *generaTicketc()* genera un ticket de compra que incluye, para **cada artículo**, las **unidades** solicitadas, y el **precio total** para dicho **artículo**. También incluye el precio **total** de la **compra** (resultante de sumar todos los totales de todos los artículos comprados). La lista de artículos puede contener **códigos repetidos**. Si por ejemplo queremos comprar dos unidades de un artículo, el código de ese artículo puede aparecer dos veces en la lista. Cada **cliente** se caracteriza por su **nif**, y su **estado**. El nif habrá sido validado previamente en otra unidad. En el caso de que no tengamos registrado el nif del cliente, o se trate de un cliente con valor null, se lanzará la excepción **BOException** con el mensaje "**El cliente no puede realizar la compra**". El **estado** del cliente puede ser "**normal**", si no tiene cuentas pendientes de abonar, o "**moroso**" (si tiene pagos pendientes). En el caso de que el cliente sea "**moroso**", se comprobará si su deuda es superior a 1000 euros, en cuyo caso se generará la excepción BOException con el mensaje "**El cliente no puede realizar la compra**" (en otro caso se continuará con el proceso de compra). Si alguno de los artículos no existe en la base de datos se generará la excepción BOException con el mensaje "**El artículo no está en la BD**". Si en algún momento se produce un error de acceso a la base de datos se generará la excepción BOException con el mensaje "**Error al recuperar datos del articulo**".

Diseña los casos de prueba para la especificación anterior utilizando el método de particiones equivalentes.

A continuación se proporcionan las estructuras de datos utilizadas en el método a probar:

```
public class Cliente {
    String nif;
    EstadoCliente estado;
    float deuda; ...
}
enum EstadoCliente {normal,moroso};
```

```
public class Articulo {
    String cod;
    float precioUnitario;
    ...
}
```

```
public class Ticket {
    Cliente cliente;
    List<LineaVenta> lineas;
    float precioTotal; ...
}
```

```
public class LineaVenta {
    Articulo articulo;
    int unidades;
    float precioLinea;
}
```

### ▷▷ Ejercicio 3: especificación *matriculaAlumno()*

Supongamos que queremos probar un método que realiza el proceso de matriculación de un alumno. Se trata del método MatriculaBO.matriculaAlumno():

**unidad**

```
public MatriculaTO matriculaAlumno(AlumnoTO alumno,
                                    List<AsignaturaTO> asignaturas) throws BOException
```

Dicho método tiene como entradas los datos de un alumno, contenidos en un objeto AlumnoTO más la lista de asignaturas de las que se quiere matricular. El método devuelve un objeto MatriculaTO que contiene: la información sobre el alumno, la lista de asignaturas de las que se ha matriculado con éxito, y una lista con el informe de error para cada una de las asignaturas de las que no se haya podido matricular. Los tipos de datos que vamos a utilizar son los siguientes:

```
public class AlumnoTO implements Comparable<AlumnoTO> {
    String nif; // NIF del alumno
    String nombre; // Nombre del alumno
    String direccion; // Direccion postal del alumno
    String email; // Direccion de correo electronico del alumno
    List<String> telefonos; // Lista de telefonos del alumno
    Date fechaNacimiento; // Fecha de nacimiento del alumno
    ...
}
```

```
public class AsignaturaTO {
    int codigo;
    String nombre;
    float creditos;
    ...
}
```

```
public class MatriculaTO {
    AlumnoTO alumno;
    List<AsignaturaTO> asignaturas;
    List<String> errores;
    ...
}
```

El método *matriculaAlumno()*, dada la información sobre los datos del **alumno** y las **asignaturas** de las que se quiere matricular, hace efectiva la matriculación, actualizando la base de datos utilizando la información que recibe como entrada. Asume que el método recibirá un objeto de tipo AlumnoTO no nulo. Los datos del alumno (excepto el nif) han sido validados previamente. En el caso particular del nif, éste podrá ser nulo, un nif válido, o un nif no válido.

Si el nif del alumno es nulo, se devuelve un error (de tipo **BOException**) con el mensaje "**El nif no puede ser nulo**". **Nota:** a menos que se diga lo contrario cuando se devuelva un error, éste será de tipo BOException.

Si el nif no es válido, devolverá el mensaje de error: "**Nif no válido**". Si el alumno no está dado de alta en la base de datos, se procederá a dar de alta a dicho alumno (con independencia de que luego se produzca un error o no en las asignaturas a matricular). Al comprobar si el alumno está o no dado de alta, puede ser que se produzca un error de acceso en la base de datos, generándose el mensaje de error "**Error al obtener los datos del alumno**", o bien que se produzca algún error durante el proceso

de dar de alta, generándose un error con el mensaje: "**Error en el alta del alumno**". Si la lista de asignaturas de matriculación es vacía o nula, se genera el mensaje de error: "**Faltan las asignaturas de matriculación**". De la misma forma, si para alguna de las asignaturas de la lista ya se ha realizado la matrícula, se devolverá el mensaje de error: "**El alumno con nif nif\_alumno ya está matriculado en la asignatura con código código\_asignatura**".

**Nota1:** asumimos que todas las asignaturas que pasamos como entrada ya existen en la BD.

**Nota2:** todos los accesos a la BD se realizan a través de una unidad externa.

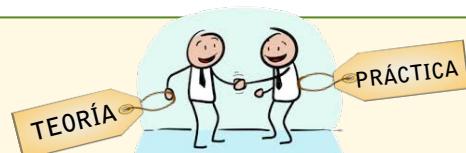
El número máximo de asignaturas a matricular será de 5. Si se rebasa este número se devolverá el error "**El número máximo de asignaturas es cinco**". El proceso de matriculación sólo se llevará a cabo si no ha habido ningún error en todas las comprobaciones anteriores. Durante el proceso de matrícula, puede ocurrir que se produzca algún error de acceso a la base de datos al proceder al dar de alta la matrícula para alguna de las asignaturas. En este caso, para cada una de las asignaturas que no haya podido hacerse efectiva la matrícula se añadirá un mensaje de error en la lista de errores del objeto MatriculaTO. Cada uno de los errores consiste en el mensaje de texto: "**Error al matricular la asignatura cod\_asignatura**" (siendo cod\_asignatura el código de la asignatura correspondiente). El campo asignaturas del objeto MatriculaTO contendrá una lista con las asignaturas de las que se ha matriculado finalmente el alumno.

Diseña los casos de prueba para la especificación anterior utilizando el método de particiones equivalentes. En el caso del objeto de tipo **AlumnoTO**, puedes considerar como dato de entrada únicamente el atributo **nif**. En el caso de los objetos de tipo **AsignaturaTO** puedes considerar únicamente el dato de entrada el atributo **codigo**, que representa el código de la asignatura.

## Resumen



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



### MÉTODOS DE DISEÑO DE CAJA NEGRA

- Permiten seleccionar de forma SISTEMÁTICA un subconjunto de comportamientos a probar a partir de la especificación. Se aplican en cualquier nivel de pruebas (unitarias, integración, sistema, aceptación), y son técnicas dinámicas.
- El conjunto de casos de prueba obtenidos será eficiente y efectivo (exactamente igual que si aplicamos métodos de diseño de caja blanca, de hecho, la estructura de la tabla obtenida será idéntica).
- Pueden aplicarse sin necesidad de implementar el código (podemos obtener la tabla de casos de prueba mucho antes de implementar, aunque no podremos detectar defectos sin ejecutar el código de la unidad a probar).
- No pueden detectar comportamientos implementados pero no especificados.
- A diferencia de los métodos de caja blanca, pueden aplicarse no solamente a unidades sino a "elementos" "más grandes" (con más líneas de código): pruebas de integración, pruebas del sistema y pruebas de aceptación

### MÉTODO DE PARTICIONES EQUIVALENTES

- Es imprescindible obtener las entradas y salidas de la unidad a probar para poder aplicar correctamente el método. Este paso es igual de imprescindible si aplicamos un método de diseño de pruebas de caja blanca, ya que de ello dependerá la estructura de la tabla de casos de prueba obtenida.
- Cada entrada y salida de la unidad a probar se partitiona en clases de equivalencia (todos los valores de una partición de entrada tendrán su imagen en la misma partición de salida). Las particiones pueden realizarse sobre cada entrada por separado, o sobre agrupaciones de las entradas, dependiendo de si el carácter valido/inválido de una partición de una entrada, depende de otra de las entradas. Cada partición (tanto de entrada como de salida, agrupadas o no) se etiqueta como válida o inválida. Todas las particiones deben ser disjuntas.
- El objetivo es proporcionar el número mínimo de casos de prueba que garanticen que estamos probando todas y cada una de las particiones, y que todas las particiones inválidas se prueban de una en una (sólo puede haber una partición inválida de entrada en cada caso de prueba). Para ello es importante seguir un orden a la hora de combinar las particiones: primero las válidas, y después las inválidas, de una en una).
- Cada caso de prueba será una selección de un comportamiento especificado. Puede ocurrir que la especificación sea incompleta, de forma que al hacer las particiones, no podamos determinar el resultado esperado. En ese caso debemos poner un "interrogante" como resultado esperado. El tester NO debe completar/cambiar la especificación.

# P04- Dependencias externas 1: stubs

Os recordamos que el plazo para entregar los ejercicios de esta práctica (P04) termina justo ANTES de comenzar la siguiente sesión de prácticas (P05) 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 P04 (de forma individual), con independencia de que se indiquen en clase las soluciones que os permitirán saber si las vuestras son correctas.

## Dependencias externas

En esta sesión implementaremos **drivers** para automatizar pruebas unitarias, teniendo en cuenta que las unidades a probar pueden tener **dependencias externas**, que necesitaremos controlar a través de sus dobles. El objetivo es realizar las pruebas aislando la ejecución de nuestra unidad. De esta forma nos aseguramos de que cualquier defecto que detectemos estará exclusivamente en el código de nuestro SUT, excluyendo así cualquier código del que dependa.

Las dependencias externas (métodos) tendremos que sustituirlas por sus dobles, concretamente por STUBS, por lo que la idea es que el doble controle las **entradas indirectas** de nuestro SUT. El doble reemplazará, durante las pruebas a la dependencia externa real que se ejecutará en producción.

Recuerda que NO se puede alterar "temporalmente" el código a probar (SUT), pero sí se puede REFACTORIZAR, para que el código contenga un **SEAM** (uno por dependencia externa), de forma que sea posible injectar el doble durante las pruebas, y que reemplazará al colaborador correspondiente. Es importante que tengas claro que hay diferentes refactorizaciones posibles y que cada una de ellas tiene diferentes "repercusiones" en el código en producción.

Los drivers que vamos a implementar realizan una **verificación basada en el estado**, es decir, el resultado del test depende únicamente del resultado de la ejecución de nuestro SUT. Observa que el algoritmo del driver es el mismo que el de sesiones anteriores, pero añadiendo más acciones en la fase de preparación de los datos, ya que tendremos que crear el doble, programar su resultado e injectarlo en nuestra SUT, antes de ejecutarla. Para implementar los drivers usaremos JUnit5.. Para ejecutarlos usaremos Maven y Junit5.

## Bitbucket

El trabajo de esta sesión también debes subirlo a *Bitbucket*. Todo el trabajo de esta práctica deberá estar en el directorio **P04-Dependencias1**, dentro de tu espacio de trabajo, es decir, dentro de tu carpeta: ppss-2021-Gx-apellido1-apellido2..

## Ejercicios

En las sesiones anteriores, hemos trabajado con un proyecto IntelliJ con un único módulo (nuestro proyecto Maven). En esta sesión vamos a crear también un proyecto IntelliJ, pero inicialmente estará **vacío**, e iremos añadiendo los módulos (proyectos Maven), en cada uno de los ejercicios.

Para crear el proyecto IntelliJ, simplemente tendremos que realizar lo siguiente:

- **File→New Project.** A continuación elegimos "Empty Project" y pulsamos sobre Next,
- **Project name : "P04-stubs". Project Location: "\$HOME/ppss-2021-Gx.../P04-Dependencias1/P04-stubs".** Es decir, que tenemos que crear el proyecto dentro de nuestro directorio de trabajo, y del directorio P04-Dependencias1.

De forma automática, IntelliJ nos da la posibilidad de añadir un nuevo módulo a nuestro proyecto (desde la ventana **Project Structure**), antes de crearlo. Cada ejercicio lo haremos en un módulo diferente.

Vamos a añadir un primer módulo que usaremos en el Ejercicio1. En la ventana que nos muestra IntelliJ, desde **Project Settings → module**, pulsamos sobre "**+→New Module**":

- Seleccionamos **Maven**, y nos aseguramos de elegir el **JDK 11**
- **No** edites los campos **Name** y **Location**
- **GroupId: "ppss"; ArtifactId: "gestorLlamadas".**
- Asegúrate de que los campos **Name** y **Location** tienen los valores: **Name: "gestorLlamadas".**  
**Location: "\$HOME/ppss-2021-Gx.../P04-Dependencias1/P04-stubs/gestorLlamadas".**

Finalmente pulsamos sobre OK (automáticamente IntelliJ marcará los directorios de nuestro proyecto como directorios estándar de Maven, de forma que "sabrá" cuáles son los directorios de fuentes, de recursos, de pruebas,...).

**NOTA:** Recuerda que debes **modificar el pom** convenientemente para poder ejecutar tus tests JUnit a través del plugin surefire. Esto lo tendrás que hacer **para cada módulo nuevo** que añadimos al proyecto. Cuando modifiques el pom, para asegurarte de que IntelliJ "se ha dado cuenta" de dicho cambio, puedes usar la opción "**Maven→Reload Project**" desde el menú contextual del **módulo** que contiene el fichero pom.xml

### ⇒ ⇒ Ejercicio 1: drivers para `calculaConsumo()`

Una vez que hemos creado el **módulo gestorLlamadas** en nuestro proyecto IntelliJ, vamos a usarlo para hacer este ejercicio.

Se trata de automatizar las pruebas unitarias sobre el método **GestorLlamadas.calculaConsumo()** (que pertenece al paquete **ppss.ejercicio1**) utilizando verificación basada en el estado.

A continuación indicamos el código de nuestro SUT, y los casos de prueba que queremos automatizar.

```
//paquete ppss.ejercicio1
public class GestorLlamadas {
    static double TARIFA_NOCTURNA=10.5;
    static double TARIFA_DIURNA=20.8;
    public int getHoraActual() {
        Calendar c = Calendar.getInstance();
        int hora = c.get(Calendar.HOUR);
        return hora;
    }

    public double calculaConsumo(int minutos) {
        int hora = getHoraActual();
        if(hora < 8 || hora > 20) {
            return minutos * TARIFA_NOCTURNA;
        } else {
            return minutos * TARIFA_DIURNA;
        }
    }
}
```

	minutos	hora	Resultado esperado
C1	10	15	208
C2	10	22	105

Debes tener claro en qué DIRECTORIOS debes situar cada uno de los fuentes.

Recuerda que el código en producción estará en src/main/java, y el código de pruebas estará en src/test/java

**IMPORTANTE:** Para implementar el driver tenemos que: detectar las dependencias externas, comprobar si nuestro SUT es testable, implementar los dobles, y finalmente implementar el driver. Tienes que tener claro cada uno de los pasos para saber lo que estás haciendo en cada momento. Esto debes hacerlo para todos los ejercicios.

## ⇒ ⇒ Ejercicio 2: drivers para *calculaConsumo()* Versión 2

Seguiremos trabajando en el módulo **gestorLlamadas** del ejercicio anterior. A partir de la tabla de casos de prueba del ejercicio 1, automatiza las pruebas unitarias sobre la siguiente implementación alternativa de **GestorLlamadas.calculaConsumo()** utilizando verificación basada en el estado. En este caso, la unidad a probar pertenece al paquete **ppss.ejercicio2** (que deberás crear), del **módulo gestorLlamadas**.

Para este ejercicio necesitamos también la clase Calendario

```
//paquete ppss.ejercicio2
public class Calendario {
    public int getHoraActual() {
        throw new UnsupportedOperationException ("Not yet implemented");
    }
}

//paquete ppss.ejercicio2
public class GestorLlamadas {
    static double TARIFA_NOCTURNA=10.5;
    static double TARIFA_DIURNA=20.8;

    public Calendario getCalendario() {
        Calendario c = new Calendario();
        return c;
    }

    public double calculaConsumo(int minutos) {
        Calendario c = getCalendario();
        int hora = c.getHoraActual();
        if(hora < 8 || hora > 20) {
            return minutos * TARIFA_NOCTURNA;
        } else {
            return minutos * TARIFA_DIURNA;
        }
    }
}
```

## ⇒ ⇒ Ejercicio 3: drivers para *calculaPrecio()*

Para este ejercicio añadiremos un nuevo módulo (*New→Module...*). El valor de "Add as a module to" y "Parent", debe ser *<none>*. No edites los campos **Name** y **Location**. El valor de **groupId** será "**ppss**" y el valor de **artifactId** será "**alquiler**"). Asegúrate de que el campo **Name** sea "**alquiler**" y que el valor de **Location** sea: "**\$HOME/ppss-2021-Gx-.../P04-Dependencias1/P04-stubs/alquiler**".

La unidad a probar en este ejercicio es el método **calculaPrecio**, el cual calcula el precio de alquiler de un determinado tipo de coche durante un determinado número de días, a partir de una fecha que se pasa también como parámetro. El precio para cada día depende de si es festivo o no, en cuyo caso se suma o se resta un 25% sobre un precio base obtenido mediante una consulta a un servicio externo. La comprobación de si es festivo o no puede lanzar una excepción, en cuyo caso, ese día no se contabilizará en el precio. Si no se ha producido ningún error en las comprobaciones, se devolverá un ticket con el precio total, en caso contrario se lanzará una excepción con un mensaje indicando los días en los que se han producido errores

Proporcionamos el siguiente código del método *AlquilaCoches.calculaPrecio()*..

```
public class AlquilaCoches {
    protected Calendario calendario = new Calendario();

    public Ticket calculaPrecio(TipoCoche tipo, LocalDate inicio, int ndias)
                                throws MensajeException {
        Ticket ticket = new Ticket();
        float precioDia, precioTotal = 0.0f;
        float porcentaje = 0.25f;

        String observaciones = "";
        IService servicio = new Servicio();
        precioDia = servicio.consultaPrecio(tipo);
        for (int i=0; i<ndias; i++) {
            LocalDate otroDia = inicio.plusDays((long)i);
            try {
                if (calendario.es_festivo(otroDia)) {
                    precioTotal += (1+ porcentaje)*precioDia;
                } else {
                    precioTotal += (1- porcentaje)*precioDia;
                }
            } catch (CalendarioException ex) {
                observaciones += "Error en dia: "+otroDia+"; ";
            }
        }

        if (observaciones.length()>0) {
            throw new MensajeException(observaciones);
        }

        ticket.setPrecio_final(precioTotal);
        return ticket;
    }
}
```

```
public class Ticket {
    private float precio_final;
    //getters y setters
}
```

Este método calcula el precio de alquiler de un determinado tipo de coche durante un determinado número de días, a partir de una fecha que se pasa también como parámetro. El precio para cada día depende de si es festivo o no, en cuyo caso se suma o se resta un 25% sobre un precio base obtenido mediante una consulta a un servicio externo. La comprobación de si es festivo o no puede lanzar una excepción, en cuyo caso, ese día no se contabilizará en el precio. Si no se ha producido ningún error en las comprobaciones, se devolverá un ticket con el precio total, en caso contrario se lanzará una excepción con un mensaje indicando los días en los que se han producido errores

Debes tener en cuenta que el tipo `LocalDate` representa una fecha y pertenece a la librería estándar de Java. La sentencia `inicio.plusDays(i)` devuelve la fecha resultante de añadir “*i*” días a la fecha “*inicio*”.

Podemos obtener una representación de tipo `String` a partir de un `LocalDate`, de la siguiente forma:

```
String fechaS = fecha.toString(); //el valor de fechaS es : "aaaa-mm-dd"
```

Podemos crear un objeto de tipo `LocalDate` a partir de un `String` mediante:

```
LocalDate fecha = LocalDate.of(2021, Month.MARCH, 2);
```

Las clases `Calendario` y `Servicio` están siendo implementadas por otros miembros del equipo.

Tendréis que crear las clases `Calendario`, `Servicio`, así como la interfaz `IService` y las excepciones `CalendarioException` y `MensajeException`. Son clases que se usarán en producción (por lo tanto deben estar en `src/main/java`), pero que no es necesario implementar. Si no las definimos, lógicamente el código no compilará.

Tipo coche es un tipo enumerado:

```
public enum TipoCoche {TURISMO, DEPORTIVO, CARAVANA}; //ej. de uso: TipoCoche.TURISMO
```

Queremos automatizar las pruebas unitarias sobre `calculaPrecio()` usando verificación basada en el estado, a partir de los siguientes casos de prueba:

**IMPORTANTE:** si necesitas refactorizar no puedes añadir ningún atributo en la clase que contiene nuestro SUT.

Asumimos que el precio base para cualquier día es de 10 euros, tanto para caravanas como para turismos.

Id	Datos Entrada				Resultado Esperado
	Tipo	fechafinicio	dias	festivo	Ticket (importe) o MensajeException
C1	TURISMO	2021-05-18	10	false para todos los días	Ticket (75)
C2	CARAVANA	2021-06-19	7	true solo para los días 20 y 24	Ticket (62,5)
C3	TURISMO	2021-04-17	8	false para todos los días, y lanza excepción en 18, 21, y 22	("Error en dia: 2021-04-18; Error en dia: 2021-04-21; Error en dia: 2021-04-22;")

**Nota:** el formato de la fecha es "aaaa-mm-dd" (año-mes-dia)

#### ⇒ ⇒ Ejercicio 4: drivers para *reserva()*

Para este ejercicio añadiremos un nuevo módulo (*New→Module...*). El valor de "*Add as a module to*" y "*Parent*", debe ser *<none>*. No edites los campos *Name* y *Location*. El valor de *groupId* será "*ppss*" y el valor de *artifactId* será "*reserva*". Asegúrate de que el campo *Name* sea "*reserva*" y que el valor de *Location* sea: "*\$HOME/ppss-2021-Gx-.../P04-Dependencias1/P04-stubs/reserva*".

Dado el código de la unidad a probar, que proporcionamos más adelante, se trata de implementar y ejecutar los drivers (usando verificación basada en el estado) automatizando así las pruebas unitarias de la siguiente tabla de casos de prueba.

	login	password	ident. socio	Acceso BD	isbns	Resultado esperado
C1	"xxxx"	"xxxx"	"Luis"	(1)	{"11111"}	ReservaException1
C2	"ppss"	"ppss"	"Luis"	(2)	{"11111", "22222"}	No se lanza excep.
C3	"ppss"	"ppss"	"Luis"	(3)	{"33333"}	ReservaException2
C4	"ppss"	"ppss"	"Pepe"	(4)	{"11111"}	ReservaException3
C5	"ppss"	"ppss"	"Luis"	(5)	{"11111"}	ReservaException4

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 "11111", "22222".

ReservaException1: Excepción de tipo ReservaException con el mensaje: "ERROR de permisos;"

ReservaException2: Excepción de tipo ReservaException con el mensaje: "ISBN invalido:33333;"

ReservaException3: Excepción de tipo ReservaException con el mensaje: "SOCIO invalido;"

ReservaException4: Excepción de tipo ReservaException con el mensaje: "CONEXION invalida;"

(1): No se invoca al método reserva()

(2): El método reserva() NO lanza ninguna excepción

(3): El método reserva() lanza la excepción ISBNInvalidoException

(4): El método reserva() lanza la excepción SocioInvalidoException

(5): El método reserva() lanza la excepción ConexionInvalidaException

Para este ejercicio, si tienes que refactorizar, usa una clase factoría.

El código de la unidad a probar es el siguiente:

```
//paquete ppss
public class Reserva {

    public boolean compruebaPermisos(String login, String password, Usuario tipoUsu) {
        throw new UnsupportedOperationException("Not yet implemented");
    }

    public void realizaReserva(String login, String password,
                               String socio, String [] isbn) throws Exception {

        ArrayList<String> errores = new ArrayList<>();
        if(!compruebaPermisos(login, password, Usuario.BIBLIOTECARIO)) {
            errores.add("ERROR de permisos");
        } else {
            IOperacionB0 io = new Operacion();
            try {
                for(String isbn: isbn) {
                    try {
                        io.operacionReserva(socio, isbn);
                    } catch (IsbnInvalidoException iie) {
                        errores.add("ISBN invalido" + ":" + isbn);
                    }
                }
            } catch (SocioInvalidoException sie) {
                errores.add("SOCIO invalido");
            } catch (JDBCException je) {
                errores.add("CONEXION invalida");
            }
        }
        if (errores.size() > 0) {
            String mensajeError = "";
            for(String error: errores) {
                mensajeError += error + "; ";
            }
            throw new ReservaException(mensajeError);
        }
    }
}
```

```
//paquete ppss
public enum Usuario {
    BIBLIOTECARIO, ALUMNO, PROFESOR
}
```

Las excepciones debes implementarlas en el paquete "ppss.excepciones" (por ejemplo):

```
//paquete ppss.excepciones
public class JDBCException extends Exception { }

//paquete ppss.excepciones
public class ReservaException extends Exception {
    public ReservaException(String message) { super(message); }
}
```

Definición de la interfaz (paquete: ppss):

```
//paquete ppss
public interface IOperacionB0 {
    public void operacionReserva(String socio, String isbn)
        throws IsbnInvalidoException, JDBCException, SocioInvalidoException;
}
```

## Resumen



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



### DEPENDENCIAS EXTERNAS

- Cuando realizamos pruebas unitarias la cuestión fundamental es AISLAR la unidad a probar para garantizar que si encontramos evidencias de DEFECTOS, éstos se van a encontrar "dentro" de dicha unidad.
- Para aislar la unidad a probar, tenemos que controlar sus entradas indirectas, proporcionadas por sus colaboradores o dependencias externas. Dicho control se realiza a través de los dobles de dichos colaboradores
- Durante la pruebas realizaremos un reemplazo controlado de las dependencias externas por sus dobles de forma que el código a probar (SUT) será IDÉNTICO al código de producción.
- Un DOBLE siempre heredará de la clase que contiene nuestro SUT, o implementará su misma interfaz, y será inyectado en la unidad a probar durante las pruebas. Hay varios tipos de dobles, concretamente usaremos STUBS
- Para poder inyectar el doble durante las pruebas, es posible que tengamos que REFACTORIZAR nuestro SUT, para proporcionar un "seam enabling point"

### IMPLEMENTACIÓN DE LOS TESTS

- El driver debe, durante la preparación de los datos, crear los dobles (uno por cada dependencia externa), y debe inyectar dichos dobles en la unidad a probar a través de uno de los "enabling seam points" de nuestro SUT.
- A continuación el driver ejecutará nuestro SUT (pasándole las entradas DIRECTAS del SUT, mientras que las entradas INDIRECTAS las obtendrá de los STUBS). El driver comparará el resultado real obtenido y finalmente generará un informe.
- Dado que el informe de pruebas dependerá exclusivamente del ESTADO resultante de la ejecución de nuestro SUT, nuestro driver estará realizando una VERIFICACIÓN BASADA EN EL ESTADO.

# P05- Dependencias externas 2: mocks

Os recordamos que el plazo para entregar los ejercicios de esta práctica (P05) termina justo ANTES de comenzar la siguiente sesión de prácticas (P06) 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 P05 (de forma individual), con independencia de que se indiquen en clase las soluciones que os permitirán saber si las vuestras son correctas.

Para realizar esta práctica disponéis de dos semanas (en el calendario aparece como P05A y P05B).

## Dependencias externas

En esta sesión implementaremos de nuevo drivers para realizar pruebas unitarias, pero en esta ocasión realizaremos una **verificación basada en comportamiento**. En este tipo de verificación el doble puede provocar que el test falle, ya que éste realiza comprobaciones para ver si la interacción de nuestra unidad con el doble es la esperada.

Ya hemos trabajado con un tipo de doble, al que hemos denominado **STUB**. Un stub controla las entradas indirectas de nuestro SUT, y se usa para realizar una **verificación basada en el estado**. En este caso, el informe de pruebas depende única y exclusivamente de si el resultado real obtenido al ejecutar nuestro SUT de forma aislada, coincide con el resultado esperado.

Ahora practicaremos con otro tipo de doble: un **MOCK**. Cuyo propósito es diferente de un stub, y por lo tanto, su implementación también lo es. Un mock es un punto de observación de las salidas indirectas de nuestro SUT, y además, y esto es importante, registra TODAS las interacciones de nuestro SUT con el doble, de forma que si el doble no es usado por nuestro SUT de la forma esperada (se llama al doble un cierto número de veces, en un determinado orden, y con unos parámetros concretos, que hemos "programado" previamente), entonces el test fallará, con independencia de que el resultado real de la ejecución de nuestro SUT coincida o no con el esperado. Por lo tanto, un mock puede hacer que nuestro test falle, mientras que un STUB nunca va a ser la causa de un informe fallido de pruebas (ya que no importa cómo interacciona nuestro SUT con el stub, ni siquiera si es invocado o no desde el SUT). Por otro lado, la implementación de un mock requerirá más líneas de código, puesto que no solamente tiene que devolver un cierto valor, sino que tiene que registrar todas las interacciones con nuestro SUT y hacer las comprobaciones oportunas que podrán provocar un fallo en nuestro informe de pruebas. Los mocks permiten realizar una **verificación basada en el comportamiento**.

Dado que la implementación de un mock no es tan simple como la de un stub, vamos a usar una **librería**, que nos permitirá crear cada doble de forma dinámica, cuando estemos ejecutando nuestro test, usando un API java. La librería que vamos a usar es **EasyMock**. Y nos va a permitir implementar tanto mocks como stubs. Por lo tanto también vamos a poder usar la librería EasyMock para implementar drivers con una verificación basada en el estado.

El proceso para implementar los drivers es el MISMO, con independencia del tipo de verificación que se realice. Así, en cualquier caso tendremos que:

1. Identificar las **dependencias externas**, que serán reemplazadas por sus dobles durante las pruebas.
2. Conseguir que nuestro SUT sea testable
3. Implementar los dobles
4. Implementar el driver (realizando un tipo de verificación u otro)

Recuerda que el objetivo no es solamente practicar con el framework, sino entender bien las diferencias entre los dos tipos de verificaciones de nuestros drivers, así como el papel de los dobles usados en ambos. La realización de los ejercicios también contribuirá a reforzar vuestros conocimientos sobre maven, y cómo se organiza nuestro código cuando usamos este tipo de proyectos .

Para implementar los drivers usaremos JUnit5 y EasyMock.. Para ejecutarlos usaremos Maven, y Junit5.

## Bitbucket

El trabajo de esta sesión también debes subirlo a *Bitbucket*. Todo el trabajo de esta práctica deberá estar en el directorio **P05-Dependencias2**, dentro de tu espacio de trabajo, es decir, dentro de tu carpeta: ppss-2021-Gx-apellido1-apellido2..

## Ejercicios

En esta sesión también vamos a crear un proyecto IntelliJ **vacío**, e iremos añadiendo los módulos (proyectos Maven), en cada uno de los ejercicios.

Para **crear el proyecto IntelliJ**, simplemente tendremos que realizar lo siguiente:

- **File→New Project.** A continuación elegimos “Empty Project” y pulsamos sobre Next,
- **Project name : "P05-mocks". Project Location: "\$HOME/ppss-2021-Gx.../P05-Dependencias2/P05-mocks".** Es decir, que tenemos que crear el proyecto dentro de nuestro directorio de trabajo, y del directorio P05-Dependencias2.

De forma automática, IntelliJ nos da la posibilidad de añadir un nuevo módulo a nuestro proyecto (desde la ventana **Project Structure**), antes de crear el proyecto. Cada ejercicio lo haremos en un módulo diferente. Recuerda que CADA módulo es un PROYECTO MAVEN.

Vamos a añadir un primer módulo que usaremos en el Ejercicio1. En la ventana que nos muestra IntelliJ, desde **Project Settings →module**, pulsamos sobre "**+→New Module**":

- Seleccionamos **Maven**, y nos aseguramos de elegir el **JDK 11**
- **No** edites los campos **Name** y **Location**
- **GroupId: "ppss"; ArtifactId: "gestorLlamadasMocks".**
- Asegúrate de que los campos **Name** y **Location** tienen los valores: **Name: "gestorLlamadasMocks". Location: "\$HOME/ppss-2021-Gx.../P05-Dependencias2/P05-mocks/gestorLlamadasMocks".**

Finalmente pulsamos sobre OK.

**NOTA:** Recuerda que debes modificar el pom convenientemente para cada módulo

### ⇒⇒ Ejercicio 1: drivers para *calculaConsumo()*

Una vez que hemos creado el **módulo gestorLlamadasMocks** en nuestro proyecto IntelliJ, vamos a usarlo para hacer este ejercicio.

Se trata de automatizar las pruebas unitarias sobre el método **GestorLlamadas.calculaConsumo()** (que pertenece al paquete **ppss**) utilizando verificación basada en el comportamiento.

**Recuerda** que la implementación de los dobles siempre la haremos con EasyMock si queremos hacer una verificación basada en el **comportamiento**.

A continuación indicamos el código de nuestro SUT, y los casos de prueba que queremos automatizar.

```
//paquete ppss
public class GestorLlamadas {
    static double TARIFA_NOCTURNA=10.5;
    static double TARIFA_DIURNA=20.8;

    public Calendario getCalendario() {
        Calendario c = new Calendario();
        return c;
    }

    public double calculaConsumo(int minutos) {
        Calendario c = getCalendario();
        int hora = c.getHoraActual();
        if(hora < 8 || hora > 20) {
            return minutos * TARIFA_NOCTURNA;
        } else {
            return minutos * TARIFA_DIURNA;
        }
    }
}
```

	minutos	hora	Resultado esperado
C1	22	10	457,6
C2	13	21	136,5

```
//paquete ppss
public class Calendario {
    public int getHoraActual() {
        throw new
        UnsupportedOperationException
        ("Not yet implemented");
    }
}
```

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

Para este ejercicio añadiremos un nuevo módulo (*New→Module...*) a nuestro proyecto IntelliJ. Recuerda que los valores de "*Add as a module to*" y "*Parent*", deben ser *<none>*, el *groupId* será "*ppss*" y el valor de *artifactId* será "*premio*"). Asegúrate de que el campo *Name* sea "*premio*" y que el valor de *Location* sea: "*\$HOME/ppss-2021-Gx-.../P05-Dependencias2/P05-mocks/premio*"

A continuación mostramos el código del método **ppss.Premio.compruebaPremio()**

```
public class Premio {
    private static final float PROBABILIDAD_PREMIO = 0.1f;
    public Random generador = new Random(System.currentTimeMillis());
    public ClienteWebService cliente = new ClienteWebService();

    public String compruebaPremio() {
        if(generaNumero() < PROBABILIDAD_PREMIO) {
            try {
                String premio = cliente.obtenerPremio();
                return "Premiado con " + premio;
            } catch(ClienteWebServiceException e) {
                return "No se ha podido obtener el premio";
            }
        } else {
            return "Sin premio";
        }
    }

    // Genera numero aleatorio entre 0 y 1
    public float generaNumero() {
        return generador.nextFloat();
    }
}
```

Se trata de implementar los siguientes tests unitarios sobre el método anterior, utilizando verificación basada en el **comportamiento**:

- A) el número aleatorio generado es de 0,07, el servicio de consulta del premio (método obtenerPremio) devuelve "entrada final Champions", y el resultado esperado es "Premiado con entrada final Champions"
- B) el número aleatorio generado es de 0,03, el servicio de consulta del premio (método obtenerPremio) devuelve una excepción de tipo ClientWebServiceException, y el resultado esperado es "No se ha podido obtener el premio"
- C) el número aleatorio generado es de 0,3 y el resultado esperado es: "Sin premio"

### ⇒ ⇒ Ejercicio 3: drivers para *calculaPrecio()*

Para este ejercicio añadiremos un nuevo módulo (*New→Module...*). El valor de "**Add as a module to**" y "**Parent**", debe ser *<none>*, el **groupId** será "*ppss*" y el valor de **artifactId** será "*alquiler*"). Asegúrate de que el campo **Name** sea "*alquiler*" y que el valor de **Location** sea: "*\$HOME/ppss-2021-Gx.../P05-Dependencias2/P05-mocks/alquiler*".

Proporcionamos el siguiente código del método *ppss.AlquilaCoches.calculaPrecio()*..

```
public class AlquilaCoches {
    protected Calendario calendario = new Calendario();

    public Ticket calculaPrecio(TipoCoche tipo, LocalDate inicio, int ndias)
            throws MensajeException {
        Ticket ticket = new Ticket();
        float precioDia, precioTotal = 0.0f;
        float porcentaje = 0.25f;

        String observaciones = "";
        IService servicio = new Servicio();
        precioDia = servicio.consultaPrecio(tipo);
        for (int i=0; i<ndias; i++) {
            LocalDate otroDia = inicio.plusDays((long)i);
            try {
                if (calendario.es_festivo(otroDia)) {
                    precioTotal += (1+ porcentaje)*precioDia;
                } else {
                    precioTotal += (1- porcentaje)*precioDia;
                }
            } catch (CalendarioException ex) {
                observaciones += "Error en dia: "+otroDia+" ";
            }
        }

        if (observaciones.length()>0) {
            throw new MensajeException(observaciones);
        }

        ticket.setPrecio_final(precioTotal);
        return ticket;
    }
}
```

```
public class Ticket {
    private float precio_final;
    //getters y setters
}
```

La especificación es la misma que la de la práctica anterior.

Recordemos que el tipo *LocalDate* representa una fecha y pertenece a la librería estándar de Java. La sentencia *inicio.plusDays(i)* devuelve la fecha resultante de añadir "i" días a la fecha "inicio".

Podemos obtener una representación de tipo *String* a partir de un *LocalDate*, de la siguiente forma:

```
String fechaS = fecha.toString(); //el valor de fechaS es : "aaaa-mm-dd"
```

Podemos crear un objeto de tipo *LocalDate* a partir de un *String* mediante:

```
LocalDate fecha = LocalDate.of(2021, Month.MARCH, 2);
```

Las clases Calendario y Servicio están siendo implementadas por otros miembros del equipo.

Tipo coche es un tipo enumerado:

```
public enum TipoCoche {TURISMO,DEPORTIVO,CARAVANA}; //ej. de uso: TipoCoche.TURISMO
```

Queremos automatizar las pruebas unitarias usando verificación basada en el comportamiento, a partir de los siguientes casos de prueba. Ten en cuenta que **no podemos alterar** en modo alguno la invocación a nuestra unidad desde otras unidades, **ni tampoco podemos añadir** ningún atributo **ni ningún método** en la clase de nuestro SUT.

Asumimos que el precio base para cualquier día es de 10 euros, tanto para caravanas como para turismos.

Id	Datos Entrada				Resultado Esperado
	Tipo	fechalinicio	dias	festivo	
C1	TURISMO	2021-05-18	10	false para todos los días	Ticket (75)
C2	CARAVANA	2021-06-19	7	true solo para los días 20 y 24	Ticket (62,5)
C3	TURISMO	2021-04-17	8	false para todos los días, y lanza excepción en 18, 21, y 22	("Error en dia: 2021-04-18; Error en dia: 2021-04-21; Error en dia: 2021-04-22;")

**Nota:** el formato de la fecha es "aaaa-mm-dd" (año-mes-dia)

**Nota 2:** En la implementación de los drivers puedes "relajar" la comprobación del parámetro en la invocación al método `es_festivo` y usar `anyObject()` en los tres tests.

**Nota 3:** Implementa una versión adicional del driver para el caso de prueba C3, al que puedes llamar C3A, en la que no "relajes" la comprobación del valor del parámetro (no uses `anyObject()` en la invocación del método `es_festivo()`). Obviamente, necesitarás mas líneas de código para especificar las expectativas de los dobles. Como podrás comprobar, con una verificación basada en el comportamiento en la que importan "todas" las interacciones de nuestro SUT con las dependencias externas (orden de las invocaciones, los valores de los parámetros, y número de invocaciones), provocan que nuestros drivers sean más "frágiles" y dependientes de la implementación, tal y como se ha explicado en la clase de teoría.

#### ⇒ ⇒ Ejercicio 4: drivers para `reserva()`

Para este ejercicio añadiremos un nuevo módulo (**New→Module...**) a nuestro proyecto IntelliJ. El valor de "**Add as a module to**" y "**Parent**", debe ser `<none>`, el **groupId** será "`ppss`" y el valor de **artifactId** será "`reserva`"). El nombre del módulo será `reserva`. Asegúrate de que el **Content root** y **Module file location** sean: `$/HOME/ppss-2021-Gx.../P05-Dependencias2/P05-mocks/reserva`"

Proporcionamos el código del método `ppss.Reserva.realizaReserva()`, así como la tabla de casos de prueba correspondiente.

La especificación es casi la misma que la de la práctica anterior, excepto que la completamos indicando que si alguien diferente del bibliotecario intenta hacer la reserva el método devuelve la excepción `ReservaException` con el mensaje "ERROR de permisos".

Código del método **ppss.Reserva.realizaReserva()**:

```
public class Reserva {

    public boolean compruebaPermisos(String login, String password, Usuario tipoUsu) {
        throw new UnsupportedOperationException("Not yet implemented");
    }

    public void realizaReserva(String login, String password,
                               String socio, String [] isbn) throws ReservaException {

        ArrayList<String> errores = new ArrayList<String>();
        if(!compruebaPermisos(login, password, Usuario.BIBLIOTECARIO)) {
            errores.add("ERROR de permisos");
        } else {
            FactoriaB0s fd = new FactoriaB0s();
            IOperacionB0 io = fd.getOperacionB0();
            try {
                for(String isbn: isbn) {
                    try {
                        io.operacionReserva(socio, isbn);
                    } catch (IsbnInvalidoException iie) {
                        errores.add("ISBN invalido" + ":" + isbn);
                    }
                }
                } catch (SocioInvalidoException sie) {
                    errores.add("SOCIO invalido");
                } catch (JDBCException je) {
                    errores.add("CONEXION invalida");
                }
            }
        if (errores.size() > 0) {
            String mensajeError = "";
            for(String error: errores) {
                mensajeError += error + "; ";
            }
            throw new ReservaException(mensajeError);
        }
    }
}
```

Las excepciones debes implementarlas en el paquete ppss.excepciones

La definición de la interfaz IOperacionB0 y del tipo enumerado son las mismas que en la práctica anterior.

La definición de la clase FactoriaB0s es la siguiente

```
public class FactoriaB0s {
    public IOperacionB0 getOperacionB0(){
        throw new UnsupportedOperationException("Not yet implemented");
    }
}
```

Tabla de casos de prueba:

	login	password	ident. socio	Acceso BD	isbns	Resultado esperado
C1	"xxxx"	"xxxx"	"Pepe"	No se accede a la BD	{"22222"}	ReservaException1
C2	"ppss"	"ppss"	"Pepe"	Acceso a BD sin excepciones	{"22222", "33333"},	No se lanza excep.
C3	"ppss"	"ppss"	"Pepe"	BD genera ex. ISBN... en 11111	{"11111"}	ReservaException2
C4	"ppss"	"ppss"	"Luis"	BD genera ex. SOCIO... en 22222	{"22222"}	ReservaException3
C5	"ppss"	"ppss"	"Pepe"	BD genera ex. ISBN... en 11111 BD genera ex. "CONEX.." en 33333	{"11111","22222", "33333"}	ReservaException4

ReservaException1: Excepción de tipo ReservaException con el mensaje: "ERROR de permisos;"

ReservaException2: Excepción de tipo ReservaException con el mensaje: "ISBN invalido:11111;"

ReservaException3: Excepción de tipo ReservaException con el mensaje: "SOCIO invalido;"

ReservaException4: Excepción de tipo ReservaException con el mensaje: "ISBN invalido:11111; CONEXION invalida;"

**Nota:** Suponemos que el login/password del bibliotecario es "ppss"/"ppss"; que "Pepe" es un socio y "Luis" no lo es; y que los isbn registrados en la base de datos son "22222", "33333".

Tienes que tener en cuenta que **no podemos alterar** en modo alguno la invocación a nuestra unidad desde otras unidades, **ni tampoco podemos añadir** ningún atributo en la clase de nuestro SUT, **ni añadir código** adicional en el directorio de fuentes.

A partir de la información anterior, y utilizando la librería EasyMock, se pide lo siguiente:

- A) Implementa los drivers utilizando verificación basada en el **comportamiento**. La clase que contiene los tests se llamará, por ejemplo, ReservaMockTest.java
- B) Implementa los drivers utilizando verificación basada en el **estado**. La clase que contiene los tests se llamará, por ejemplo, ReservaStubTest.java

## Resumen



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



### VERIFICACIÓN BASADA EN EL ESTADO

- El driver de una prueba unitaria puede realizar una verificación basada en el estado resultante de la ejecución de la unidad a probar.
- Si la unidad a probar tiene dependencias externas, éstas serán sustituidas por stubs durante las pruebas. Los dobles controlarán las entradas indirectas de nuestro SUT. Un stub no puede hacer que nuestro test falle (el resultado del test no depende de la interacción de nuestro SUT con sus dependencias externas)
- Para poder usar los dobles (stubs), éstos tienen que poder injectarse en nuestro SUT a través de los "enabling seam points".

### VERIFICACIÓN BASADA EN EL COMPORTAMIENTO

- El driver de una prueba unitaria puede realizar una verificación basada en el comportamiento, de forma que no sólo se tenga en cuenta el resultado real, sino también la interacción de nuestro SUT con sus dependencias externas (cuántas veces se invocan, con qué parámetros, y en un orden determinado).
- Los dobles usados si realizamos una verificación basada en el comportamiento se denominan mocks. Un mock constituye un punto de observación de las salidas indirectas de nuestro SUT, y además registra la interacción del doble con el SUT. Un mock sí puede provocar que el test falle.
- Para poder usar los dobles (mocks), éstos tienen que poder injectarse en nuestro SUT a través de los "enabling seam points".
- Para implementar los dobles usaremos la librería EasyMock. Para ello tendremos que crear el doble (de tipo Mock, o StrictMock), programar sus expectativas, indicar que el doble ya está listo para ser usado y finalmente verificar la interacción con el SUT
- EasyMock también nos permite implementar drivers usando verificación basada en el estado (usando stubs). Para ello tendremos que crear el doble (de tipo NiceMock), programar sus expectativas (relajando los valores de los parámetros) e indicar que el doble ya está listo para ser usado por nuestro SUT".