P01: Pruebas del software

IMPORTANTE. A TENER EN CUENTA DURANTE TODO EL CURSO

- Debes subir a GitHub las prácticas que realices. **SÓLO SE REVISARÁN** los trabajos si se han subido **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. Las dudas sobre las prácticas debes trasladarlas a tu profesor de prácticas.
- Tu trabajo de PRÁCTICAS te permitirá comprender y asimilar los conceptos vistos en las clases de TEORÍA. Es fundamental que trabajes bien las prácticas ya 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 el aula y en laboratorio determinará si alcanzas o no las competencias teórico-prácticas planteadas a lo largo del cuatrimestre.

En esta sesión vamos a familiarizarnos en el uso de las herramientas de trabajo. También te proporcionamos una documentación sobre Maven, para complementar lo que hemos trabajado en clase de teoría:

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: 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.

Ya hemos usado Maven desde línea de comandos a través del terminal. En el resto de sesiones, seguiremos usando Maven pero lanzaremos los comandos maven desde IntelliJ.

Cualquier herramienta de construcción de proyectos necesita conocer la secuencia de tareas que debe ejecutar para construir un proyecto. Dicha secuencia de tareas (o acciones) 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.

La secuencia de tareas "programadas" en un *build script* en un momento determinado, define el **proceso de construcción** de nuestro proyecto (es posible que diferentes proyectos necesiten diferentes secuencias de tareas). Por lo tanto, ejecutar el *build script* es lo mismo que ejecutar el proceso de construcción de un proyecto.

Maven proporciona 3 *build scripts* a los que llama ***ciclos de vida***, concretamente:

- ***default lifecycle*** (con 23 fases)
- ***clean lifecycle*** (formado por 3 fases)
- ***site lifecycle*** (formado por 4 fases)

Las fases de los tres *build scripts* son DIFERENTES, es decir, que dada una fase, sabemos a qué *build script* pertenece.

El ciclo de vida más utilizado es el ciclo de vida por defecto (***default lifecycle***), con 23 **fases**. Cada fase puede contener cero o más tareas (acciones), que se ejecutarán cuando se alcance dicha fase. Una fase, por lo tanto es algo lógico, es una declaración de intenciones. Cada fase está "pensada" para que se ejecuten ciertas acciones (o tareas) , a las que maven denomina ***goals.***

Cuando lanzamos el proceso de construcción, ejecutaremos las *goals* asociadas a cada una de las fases, comenzando siempre por la primera fase. Ejemplos de fases son: *compile, test, package, deploy,...*

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 la usaremos para asignar acciones ejecutables que lleven a cabo el proceso de compilación del código fuente 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*** (acción, tarea) 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 ":")

Una ***goal***, por tanto, no es más que un código ejecutable, implementado por algún desarrollador del *plugin* al que pertenece dicha goal. De hecho es un proyecto maven con empaquetado jar. 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 (propiedades) propias que tienen valores por defecto y que podemos cambiar). Por ejemplo, podemos cambiar la fase a la que se asociará dicha *goal*.

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.mxl, en la sección <build> (y configurar sus propiedades, 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
- Por lo tanto NO es necesario incluir el plugin ***compiler*** en el pom, porque ya está incluido por defecto en cualquier proyecto maven, si su empaquetado es jar. A menos, claro está, que queramos cambiar su configuración, o necesitemos una versión diferente del plugin incluido por defecto.

La única forma de conocer qué goals tiene un plugin, si éstas están asociadas por defecto o no a alguna fase, qué hace cada goal, y cómo se puede configurar cada una de ellas usando propiedades maven, es consultando la documentación de cada plugin. Por ejemplo, https://maven.apache.org/plugins/maven-compiler-plugin/

pom.xml
(configura el build script del proyecto.)

Cualquier proyecto Maven debe contener en su directorio raíz el fichero **pom.xml**. (**POM** = ***P**roject **O**bject **M**odel*) 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, tal y como hemos indicado en clase.

- **coordenadas**: usadas para identificar el artefacto Maven asociado a nuestro proyecto
- **propiedades**: pares nombre-valor usados para configurar diferentes elementos de nuestro pom
- **dependencias**: librerías externas necesarias para construir nuestro proyecto maven
- ***build***: sección desde la que podemos alterar la secuencia de goals ejecutadas por maven durante la construcción del proyecto.

artefactos Maven

(son ficheros que se identifican por sus coordenadas))

Durante el proceso de construcción, Maven usa, y también puede generar, ficheros empaquetados que se identifican de forma única mediante sus coordenadas, separadas por ":". Dichos ficheros se almacenen en REPOSITORIOS (locales y/o remotos) y se denominan **artefactos** Maven. Para identificar un artefacto Maven se requieren tres coordenadas como mínimo. Las coordenadas nos permiten LOCALIZAR en los repositorios locales y/o remotos cualquier artefacto Maven.

Las coordenadas que identifican de forma única a cualquier artefacto Maven son, como mínimo: **`groupId:artifactId:version`**

- **`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*.

Opcionalmente, se puede usar una cuarta coordenada ***package.***

- **`package`** es la extensión del fichero. Indica el tipo de empaquetado. Esta coordenada es OPCIONAL. Si se omite, se asume que el empaquetado es *jar*.

Tanto las librerías externas de nuestro proyecto (dependencias), como los plugins usados por nuestro proyecto son ficheros con extensión .jar, identificados por maven por sus coordenadas y almacenados en repositorios. Son, por lo tanto, artefactos maven.

Nuestro proyecto maven también puede empaquetarse como un jar, de forma que pueda usarse como dependencia externa para otro proyecto maven.

El tipo de empaquetado asociado a un proyecto maven se indica en sus coordenadas. Dependiendo del tipo de empaquetado, un ciclo de vida maven tiene asociadas POR DEFECTO ciertas GOALS, tal y como hemos explicado en clase.

Los artefactos Maven se almacenan en un repositorio local Maven, situado en *$HOME/.m2/repository.* Las coordenadas se usan para identificar exactamente la ruta del fichero (artefacto maven) en el repositorio maven (que puede ser local o remoto). 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
- `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:proyecto3:1.0-SNAPSHOT:war` representa al fichero $HOME/.m2/repository/org/ppss/proyecto3/1.0-SNAPSHOT/proyecto3-1.0-SNAPSHOT.war

repositorios locales y remotos Maven

(almacenan artefactos Maven)

Los artefactos maven se almacenan en repositorios. Maven mantiene una serie de repositorios remotos, que alojan los plugins y librerías que podemos utilizar, por ejemplo ***Maven Central Repository*** (https://mvnrepository.com/), que actualmente cuenta con más de 50 millones de librerías!!
Cuando ejecutamos Maven por primera vez en nuestra máquina, se crea el directorio **.m2/repository** (en nuestro $HOME), que será nuestro repositorio local. Cuando iniciamos un proceso de construcción Maven, primero se consulta en 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. Si borramos el directorio .m2, éste se volverá a crear cuando realicemos una nueva construcción del proyecto.

dependencias del proyecto Maven

(librerías necesarias para construir el proyecto)

Cualquier LIBRERÍA EXTERNA (ficheros .jar) usada en nuestro proyecto, debe incluirse en la sección ***<dependencies>*** de nuestro fichero pom. Es fácil recordarlo porque representa el código adicional del que DEPENDE nuestro código. Maven se encarga de descargar dicha librería si es necesario. Es más, si utilizamos una librería, que a su vez depende de otra, Maven automáticamente se encarga de descargarse también esta última, y así sucesivamente. Esto hace que nuestros proyectos Maven "pesen" poco, ya que no será necesario incluir ni el directorio target, ni ninguna librería y/o plugin utilizados por el proyecto. Éstos se descargarán de forma automática, 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.

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 ficheros y/o 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 GitHub.

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 queramos realizar, o bien indicando la **goal**, o goals que queremos ejecutar de forma explícita (separadas por espacios). Las 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

Para crear un proyecto maven, se pueden usar lo que maven denomina "archetypes": son como "plantillas" para generar proyectos maven con diferentes configuraciones del pom.xml y con diferentes estructuras de directorios Maven (la estructura estándar de directorios maven que hemos mostrado en prácticas no está completa. Unicamente hemos indicado los directorios estándar que vamos a usar).

Hay una lista enorme de arquetipos maven que se pueden usar para crear un proyecto Maven. Un arquetipo Maven es un artefacto Maven, y por lo tanto se identifica mediante sus coordendadas.

Por ejemplo, podemos usar la goal `archetype:generate`, y usar el arquetipo `maven-archtetype-quickstart`, para crear un proyecto maven que incluye la librería de JUnit en el pom y contiene un test de ejemplo. El comando es el siguiente:

```
mvn archetype:generate \
    -DgroupId=<aqui_ponemos_el_valor_que_queramos> \
    -DartifactId=<este_será_el_nombre_del_proyecto_generado>\
    -DarchetypeArtifactId=maven-archetype-quickstart \
    -DinteractiveMode=false
```

Incluso podemos crear nuestro propio arquetipo si lo creemos conveniente.

IntelliJ IDEA Ultimate

IntelliJ es un IDE muy utilizado para trabajar con diferentes tipos de aplicaciones, entre ellas aplicaciones java y Maven. Nosotros 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 elementos 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, ficheros que contienen el 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 *proyecto* 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 

Indicamos dos formas, que usaremos en los ejercicios de esta sesión.

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. Se nos preguntará si confiamos en dicho proyecto, y le diremos que sí.

De forma alternativa, también podremos usar la opción File→Open y seleccionar la **carpeta** que contiene el fichero pom.xml. Nos preguntará si confiamos en dicho proyecto, y le diremos que sí.

Al abrir el proyecto, la barra con los menús de opciones no es visible. En su lugar, veremos un icono en el extremo izquierdo la barra superior con 4 líneas horizontales. Lo seleccionaremos cada vez que necesitemos usar alguna de las opciones de la barra de menús de IntelliJ.

En la máquina virtual usamos openjdk 21. Siempre que abramos un proyecto, nos aseguraremos de que lo tiene asociado. Para ello, desde la barra de menús de IntelliJ seleccionamos ***File → Project Structure → Project Settings → Project → SDK***, y seleccionamos: "**21 java version 21.0.5**". En el cuadro de texto ***Languaje level***, seleccionaremos "*21-Record patterns, pattern matching for switch*". Para guardar los cambios pulsamos sobre el botón "***Apply***". Desde ***Platform Settings → SDKs*** nos aseguraremos de que tenemos seleccionada la librería jdk con la versión 21

Podemos FIJAR estas propiedades para NUEVOS PROYECTOS, desde ***File-→ New Projects Setup → Structure... → Project Settings → Project → SDK***, seleccionamos: "**21 java version 21.0.5**". En el cuadro de texto ***Languaje level***, seleccionaremos "*Record patterns, pattern matching for switch*". Para guardar los cambios pulsamos sobre el botón "***Apply***". Finalmente, desde ***Platform Settings → SDKs*** nos aseguraremos de que tenemos seleccionada la librería jdk con la versión 21.

➪ Creación de un proyecto IntelliJ Maven nuevo 

Podemos crear un proyecto Maven básico, que simplemente contenga la estructura estándar de directorios y un fichero pom.xml con el mínimo contenido posible.

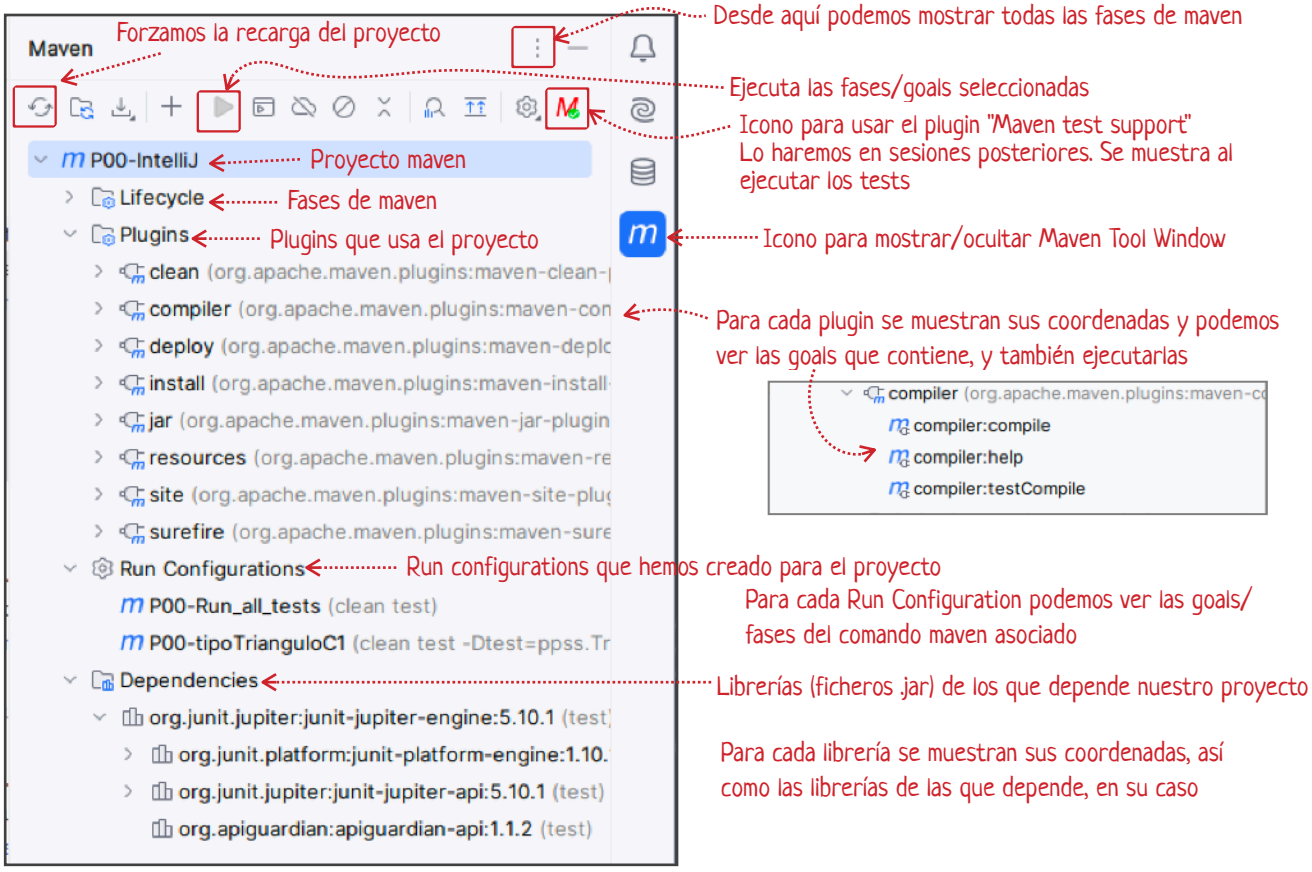
Para ello seleccionamos: ***File-→ New →Project... →Java***

- ***Name***: aquí tecleamos el nombre del proyecto (nombre de la carpeta que contiene el fichero pom.xml)
- ***Location***: ruta en nuestro disco duro donde se creará nuestro proyecto maven
- ***JDK***: nos aseguraremos de que esté seleccionado JDK 21
- Desde ***Advanced Settings*** podemos indicar el valor de las coordenadas ***groupId*** y ***artifactId*** (por defecto el valor de artifactId será el mismo que el nombre del proyecto. Te recomendamos que no lo cambies)

Después de crear el proyecto, verás que ha creado un fichero `.gitignore`. BÓRRALO siempre. NO es necesario ya que tenemos nuestro propio fichero *.gitignore* en nuestro directorio de trabajo. Para borrar el ficheros, simplemente situamos el ratón sobre él, y desde el menú contextual (es decir, con botón derecho del ratón), seleccionamos ***Delete...***

IntelliJ IDEA Maven Tool Window

IntelliJ permite mostrar diferentes ***"Tool Windows"*** en las que se visualizan diferentes perspectivas (vistas) del proyecto. Una de estas "vistas" es la ventana de Maven (***"Maven tool window"***), que usaremos cuando trabajemos con un proyecto Maven. A continuación mostramos el aspecto de dicha ventana, usando un proyecto maven cualquiera de ejemplo.



Usaremos la ventana *Maven Tool Window* para ejecutar fases/goals del proyecto y también para consultar las versiones de las librerías y plugins usados

IntelliJ IDEA Run Configurations

Usaremos las ***Run Configurations*** para realizar diferentes construcciones del proyecto, utilizando comandos maven.

IntelliJ proporciona diferentes plantillas. Nosotros trabajaremos con las plantillas Maven.

A continuación mostramos una imagen con un proyecto maven de ejemplo

Cuando creemos una *Run Configuration* siempre marcaremos la opción ***Store as project file***, y la guardaremos en la carpeta con el nombre ***intellij-configurations***, al mismo nivel que el pom. No se trata de un directorio maven, sino de una carpeta que hemos añadido para no "perder" las *run configurations* al subir el trabajo de prácticas a GitHub. Por defecto, las *run configurations* se guardan en la carpeta ***.idea***, la cual es ignorada por git.

Para **crear** una *run configuration* usaremos el icono "**+**" de la ventana emergente que aparece cuando seleccionamos "***Edit Configurations"***

Ejercicios

El directorio **Plantillas-P01** contiene un varios ficheros que necesitaremos para realizar los ejercicios.

Para poder hacer los ejercicios debes:

- **Desde tu directorio de trabajo** (directorio que contiene la carpeta oculta .git) **crea la carpeta P01-IntelliJ (este será el directorio que vamos a usar durante esta sesión. y todo tu trabajo deberá estar en esta carpeta)**

  IMPORTANTE!!! RECUERDA que a partir de ahora, TODO lo que hagas en prácticas estará en algún subdirectorio de tu directorio de trabajo.

➪ Ejercicio 1: proyecto maven (nuevo) simpleMavenProject 

Crea un primer proyecto maven (en la carpeta P01-IntelliJ) usando los siguientes datos:

- ***Name***: `simpleMavenProject`
- ***Location***: carpeta P01-Intellij
- ***JDK***: nos aseguraremos de que esté seleccionado JDK 21
- ***groupId:*** `ppss.P01`

Observa qué información tiene el fichero pom.xml del proyecto que acabas de crear. Fíjate en la sección de **PROPIEDADES**. Estas propiedades las vamos a usar en TODOS los proyectos maven del curso.

Verás que se ha creado la estructura de directorios Maven pero no hay ningún fichero de código java.

En el directorio de plantillas tenéis los ficheros `Ejemplo.java` (que forma parte del código fuente de nuestro proyecto) y `EjemploTest.java` (que contiene la implementación de un caso de prueba para el método *Ejemplo.fechaValida(int dia, int mes, int anyo)*.

COPIA los ficheros anteriores en el proyecto maven (donde corresponda). Lógicamente, deberás tener en cuenta el paquete al que pertenecen ambas clases.

Observarás que, para la clase `FechaTest`, IntelliJ detecta que ciertas clases no están disponibles (las clases que importamos). Dichas clases no pertenecen a la librería estándar de Java. Pertenecen al api de JUnit, que usaremos para poder implementar y ejecutar nuestros tests.

No te preocupes de momento por estos avisos de IntelliJ.

Desde la ventana Maven de IntelliJ vamos a **COMPILAR** los fuentes del proyecto usando la fase que ya hemos explicado en clase ¿Cuál es el resultado de la construcción del proyecto y por qué da ese resultado?

Observa las GOALS que se ejecutan. ¿Por qué no aparecen los plugins correspondientes en el fichero pom.xml de nuestro proyecto?

Vamos a la ventana Maven, ahora fíjate en el contenido del elemento `Plugins`: ahí aparecen plugins que NO tienes indicados de forma explícita en tu fichero pom.xml.

super POM
(pom "padre" cualquier proyecto maven)

effective POM
(configuración particular para un proyecto maven)

Cualquier proyecto Maven, dependiendo de su empaquetado, tiene ciertas goals asociadas por defecto a ciertas fases. De la misma forma, si las dependencias no se encuentran en el repositorio local, Maven busca en ciertos repositorios remotos. Toda esa información se encuentra indicada en el denominado "super POM", del cual "heredan" todos los proyectos Maven.

La configuración que hemos indicado de forma explícita en nuestro fichero pom.xm + la configuración que se hereda del super POM se denomina "effective POM". (puedes consultar esta referencia)

Obviamente, la configuración "explícita" de nuestro pom.xml podemos verla abriendo el fichero. Pero, ¿cómo podemos sabe cuál es su configuración "efectiva", es decir la resultante de "combinar" la configuración que heredamos más la que nosotros hemos configurado?

Para mostrar cuál es la configuración "real" (o "efectiva") de nuestro pom, podemos usar el comando `mvn help:effective-pom`

En la ventana Maven de IntelliJ, el elemento `Plugins` muestra precisamente eso. Por eso vemos, por ejemplo, que nuestro pom tiene asociado el plugin compiler, y por eso hemos podido compilar los fuentes de nuestro proyecto.

Fíjate en la goal que hemos ejecutado al compilar el proyecto (`compiler:compile`). Puedes ver que el plugin compiler tiene más de una goal (desplegando el elemento correspondiente en la ventana maven de IntelliJ).

Vamos a consultar la documentación de dicho plugin (en clase ya hemos dicho que la única forma de conocer las goals que tiene un plugin, qué hace cada una de ellas, y cómo podemos configurar dicho plugin es consultando su documentación.

Puedes buscar en el navegador las palabras "maven compiler plugin" o acceder directamente a https://maven.apache.org/plugins/maven-compiler-plugin/

Una vez en la página, si accedes al apartado `Goals` podrás ver qué hace cada goal. Y si accedes a cada una de ellas, te indica si dichas goals están asociadas o no a alguna fase por defecto. Esto es importante, porque si una goal no está asociada por defecto a ninguna fase y no la asociamos de forma explícita a ninguna de ellas, entonces dicha goal NO se ejecutará aunque incluyamos el plugin en el pom.

También puedes comprobar que el plugin tiene dos goals, que hacen cosas diferentes y están asociadas a fases diferentes!! ¿cuáles son esas goals?

Volvamos a nuestro proyecto Maven. Ahora vamos a **COMPILAR** los fuentes de nuestros **TESTS**, usando la fase que ya debes conocer. Fíjate que, a pesar de no haber indicado de forma explícita la goal para hacer esto, podemos ejecutar sin problema por lo que acabamos de explicar.

Ahora debes ver una construcción fallida (como era de esperar, y nos había avisado IntelliJ).

Nuestros tests no compilan porque las clases (de la librería JUnit) que usamos en dicho código NO están indicadas en el pom.

Por lo tanto debemos **INCLUIR LA LIBRERÍA** (fichero .jar) en el pom, que contiene las clases que necesitamos para poder compilar nuestro código de pruebas. Vamos a buscarla en uno de los repositorios remotos de maven: *Maven Central Repository* (que os hemos indicado en la página 4 de este documento).

Accede al repositorio central de Maven y busca `junit-jupiter` (en el cuadro de búsqueda de la parte superior de la página). Te aparecerán varios resultados. Para cada uno de ellos, debajo del nombre, se muestra el valor de las coordenadas *groupId* y *artifactId* con el formato *valorGroupId >> valorArtifactId.* Elige la entrada cuyo groupId es `org.junit.jupiter`, y su artifactId tiene como valor `junit-jupiter`.

Selecciona dicha entrada. Estamos interesados en usar la versión 5.11.4. Selecciónala y copia desde la pestaña Maven, la etiqueta <dependency> de esta librería. Finalmente pégala en el fichero pom.xml de tu proyecto Maven y observarás que los avisos de IntelliJ desaparecen de tu código de pruebas.

Ahora comprueba que ya puedes compilar los fuentes de los tests.

NOTA: Si IntelliJ sigue mostrándote advertencias en rojo, recarga de nuevo el proyecto Maven, puedes hacerlo usando el primer icono de la izquierda en la parte superior de la ventana Maven. De esta forma, fuerzas a que IntelliJ vuelva al leer el fichero pom.xml que has modificado.

Desde la ventana maven (y accediendo a la documentación del plugin), averigua qué goal se ejecuta cuando ejecutamos los tests (es el plugin que verás en la última posición de la lista de plugins que te muestra IntelliJ).

EJECUTA LOS TESTS. El resultado debe ser una construcción exitosa.

Como ya hemos ejecutado varias fases en esta práctica y en la anterior, Maven se habrá descargado los artefactos necesarios para construir el proyecto. **Consulta tu repositorio local y LOCALIZA**:

- la librería de JUnit que acabas de añadir (usa sus coordenadas para localizar el fichero jar)
- el plugin compiler que estás usando para compilar (usa sus coordenadas para localizar el fichero jar)

Queremos usar una versión más reciente tanto del compilador de java como del plugin para ejecutar nuestros tests. Como ambos están incluidos ya por defecto en nuestro pom, lo que haremos para cambiar las versiones es simplemente incluir ambos plugins en el pom pero con la versión que nos interesa.

En concreto queremos usar la versión 3.13.0 para el compilador de java, y la versión 3.5.2 para el plugin para ejecutar los tests.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.13.0</version>
</plugin>
```

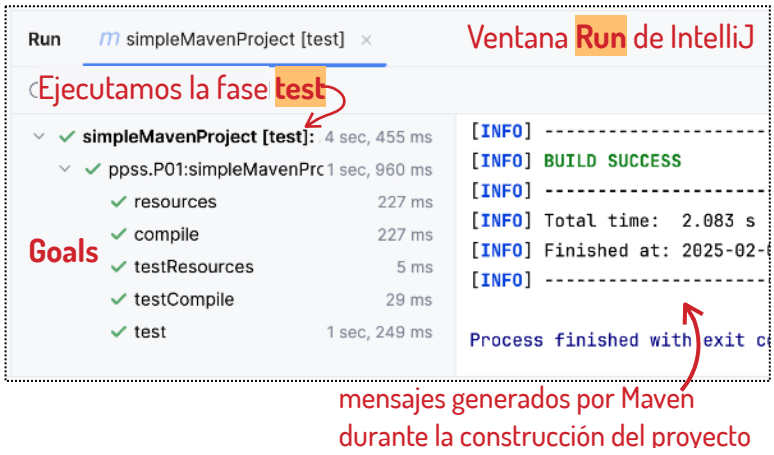
```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>3.5.2</version>
</plugin>
```

Una vez añadidos en el pom, en la sección que corresponda, comprueba que tu "pom efectivo" que muestra IntelliJ es diferente (ya que usa artefactos diferentes).

Comprueba que puedes ejecutar los tests y que obtienes una construcción exitosa. Fíjate que si ejecutas la fase para ejecutar los tests, también vas a necesitar ejercitar la nueva goal que has añadido para compilar.

Maven siempre te muestra todo lo que va ejecutando en forma de texto.

IntelliJ nos muestra, para dicha construcción, todas las golas (pero sin indicar el nombre del plugin) que ejecuta Maven.



En la ventana Maven de IntelliJ vemos algunas fases de Maven (de los tres ciclos de vida). Te habrás dado cuenta de que no se muestran TODAS las fases de los 3 ciclos de vida de maven. Las fases mostradas por defecto se basan en una selección de fases más frecuentemente usadas. Si quieres ver todas las fases usa el icono de opciones (en la parte superior derecha de la ventana maven, mira la imagen del apartado *Maven Tool Window* de este documento) y desmarca la opción ***Show Basic Phases Only***

Finalmente familiarízate con las fases **PACKAGE** e **INSTALL**. Ejecútalas y anota:

- qué goals se ejecutan en cada caso,
- qué hace, qué artefacto genera y dónde se genera durante la fase package,
- qué hace exactamente la fase install y dónde se copia exactamente en tu disco duro el artefacto correspondiente .

¿Para qué nos puede servir que el empaquetado de nuestro proyecto esté almacenado en nuestro repositorio local Maven?

Recuerda subir este ejercicio a GitHub, usando los comandos que ya debes conocer.

➪ Ejercicio 2: construcción del proyecto: run configurations

Vamos a crear ***Run Configurations*** para "programar" diferentes construcciones del proyecto. Para ello usaremos las indicaciones del apartado correspondiente de este documento.

Vamos a crear 3 *Run Configurations* (recuerda que debes pulsar el desplegable a la derecha de "Current File"):

- *Run Configuration* con el nombre `SimpleProject_CompileAll`
  Queremos eliminar cualquier resultado de construcciones previas y compilar todos los fuentes, tanto los del proyecto como los de los tests
- *Run Configuration* con el nombre `SimpleProject_Run_tests`
  Queremos eliminar cualquier resultado de construcciones previas y ejecutar los tests
- *Run Configuration* con el nombre `SimpleProject_Compile_only_tests`
  Queremos eliminar cualquier resultado de construcciones previas y únicamente compilar el código de los tests

Recuerda que TODAS las Run Configurations debes guardarlas en la carpeta ***intellij-configurations*** (en la raíz de tu proyecto Maven). Dicha carpeta NO es una carpeta Maven, y no se creará automáticamente al crear el proyecto maven.

Ejecuta las Run Configuration que has creado y comprueba que hacen lo que se indica. Debes tener claro cuál es el resultado, qué ficheros se generan y dónde en cada proceso de construcción.

En el directorio de plantillas dispones del fichero **ejercicio2.txt**. Indica los comandos maven asociados a cada Run Configuration y copiar dicho fichero en tu directorio P01-IntelliJ.

Ejercicio 3: comportamientos especificados, programados y probados

Proporcionamos la especificación del método fechaValida(). Dicho método, a partir de tres valores enteros que representan el día, mes y año, devuelve el valor cierto o falso en función de que formen una fecha válida o no.

Como información adicional recordamos qué reglas deben cumplirse para considerar que un año es bisiesto:

- Todos los años bisiestos son divisibles entre 4.
- Aquellos años que son divisibles entre 4, pero no entre 100, son bisiestos.
- Los años que son divisibles entre 100, pero no entre 400, no son bisiestos.
- Sin embargo, los años divisibles entre 100 y entre 400 sí que son bisiestos.

Tal y como hemos visto en clase, el conjunto S, o conjunto de comportamientos especificados se obtiene a partir de la especificación.

La implementación del método `fechaValida()` constituye un conjunto P de comportamientos programados.

Y fnalmente, los tests constituyen un conjunto T de comportamientos probados.

Teniendo en cuenta que en clase hemos definido un comportamiento como una tupla con datos de entrada + resultado (tanto los datos de entrada como el resultado deben ser valores **CONCRETOS**), se pide:

- Indica tres comportamientos que pertenezcan al conjunto S
- Indica dos comportamientos que pertenezcan al conjunto P
- Enumera el conjunto de comportamientos del conjunto T

En clase hemos indicado que cada comportamiento probado recibe el nombre de CASO DE PRUEBA.

El proceso de **diseño de pruebas** consiste en seleccionar un conjunto de CASOS DE PRUEBA, y que este proceso debe hacerse de forma EFICIENTE y EFECTIVA

Es muy importante, no sólo que contestes a las preguntas, sino que tengas claro por qué has contestado eso y no otra cosa.

El resultado del proceso de diseño se refleja en una TABLA de casos de prueba.

Una vez concluido con el proceso de diseño, el siguiente paso consiste en AUTOMATIZAR la ejecución de dicha tabla.

Indica qué dos subprocesos forman la automatización de las pruebas. En el proyecto Maven en el que estamos trabajando ¿hemos realizado la automatización de las pruebas? Indica exactamente cuáles son los resultados de ambos subprocesos en dicho proyecto.

En el directorio de plantillas dispones del fichero **ejercicio3.txt**. Contesta ahí las cuestiones planteadas en este ejercicio y copia dicho fichero en el directorio P01-IntelliJ.

Para terminar, en el directorio de plantillas hay un fichero **tests.txt**, con el código de dos tests adicionales. Copia dichos métodos en el fichero que contiene los tests de tu proyecto maven, y vuelve a lanzar el proceso de construcción. Verás que la construcción no tiene éxito. Observa el informe de ejecución de los tests que maven te muestra en pantalla, depura el defecto y asegúrate de que efectivamente el defecto desaparece, y que no introduces ningún defecto adicional

Debes tener claro que cualquier cambio que hagas en el código puede provocar que aparezca un nuevo defecto que no estaba antes.

Por lo tanto vas a tener que repetir el proceso de pruebas muchísimas veces durante el desarrollo. Por ello es impensable NO automatizar el proceso de pruebas. Recuerda que las pruebas son necesarias si queremos tener éxito en el desarrollo del proyecto, que inevitablemente vamos a cometer errores (mistake), y que reparar las consecuencias de esos errores va a consumir mucho de nuestro tiempo de desarrollo.

Recuerda también que en esta sesión, hemos visto que:

Es importante tener claro que la **secuencia de pasos** para realizar las pruebas es ésta:

- primero tenemos que obtener (diseñar) los casos de prueba que usaremos para comprobar que el programa funciona correctamente (apartado A). 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 tenemos que implementar los tests, y
- finalmente ejecutarlos para obtener el informe de pruebas.

El **informe de pruebas** obtenido (proporcionado por JUnit), nos indica 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).

La viñeta mostrada como inicio del tema de teoría, es pura ficción. La "magia" no existe. Si no hacemos pruebas, y las hacemos bien (de forma efectiva y eficiente), nuestro proyecto NO tendrá éxito.

Resumen

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

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: coordenadas, propiedades, dependencias y plugins. (Hay más secciones, pero de momento sólo veremos estas cuatro)
- Un artefacto maven es un fichero usado y/o generado por el proceso de construcción de maven y que se identifica mediante sus coordenadas y se almacena en un repositorio Maven (local o remoto). Dichas coordenadas nos permiten conocer en qué ruta de nuestro repositorio local (y/o remoto) se almacena dicho fichero, así como el nombre exacto del mismo.
- Los proyectos maven requieren una estructura de directorios fija. El código de los tests está físicamente separado del código fuente de la aplicación.
- Maven tiene predefinidos tres ciclos de vida (o build scripts) para construir un proyecto. Cada ciclo de vida está formado por una secuencia ordenada de fases, cada fase puede tener asociadas unas goals. Una goal siempre pertenece a un plugin. El resultado del proceso de construcción maven puede ser Build failure o Build success.
- El comando mvn permite ejecutar tanto una fase (de alguno de los ciclos de vida) como una goal. En el primer caso se ejecutan todas las goals asociadas a todas las fases del ciclo de vida, desde la primera, hasta la fase especificada. En el segundo caso únicamente ejecutaremos la goal indicada..
- Construiremos el proyecto a través de IntelliJ. Los comandos maven a ejecutar los guardaremos en ficheros xml llamados Run Configurations. Dichos ficheros los crearemos en la carpeta intellij-configurations (NO es una carpeta de maven, y el nombre de dicha carpeta es arbitrario)
- También podemos ejecutar cualquiera de las fases de los ciclos de vida de maven directamente desde la ventana Maven Tools. Igualmente podemos ver/ejecutar cualquiera de las goals de los plugins que usa nuestro pom. La ventana Maven Tools muestra el effectiive pom de nuestro proyecto Podemos ver las coordenadas de todos los artefactos maven de nuestro pom..

COMPORTAMIENTOS ESPECIFICADOS, IMPLEMENTADOS Y PROBADOS

- Definimos un comportamiento como una tupla formada por valores CONCRETOS de entrada + resultado asociado.
- Definimos el conjunto S de comportamientos especificados (se obtiene a través de la especificación. El conjunto P es el conjunto de comportamientos implementados, y el conjunto T es el conjunto de comportamientos probados
- Cada comportamiento probado se denomina CASO DE PRUEBA, y constituye una fila de la denominada TABLA de casos de prueba.
- Para realizar pruebas dinámicas tenemos que: 1) Obtener (diseñar) los casos de prueba que usaremos para evidenciar la presencia de defectos en el código implementado. 2) Automatizar la ejecución de dicha tabla, implementando los tests (cada test implementa un comportamiento probado), y ejecutando dichos tests de forma automática (integrados en el proceso de construcción del proyecto..
- El proceso de diseño es el que condiciona la bondad de nuestras pruebas. El objetivo es realizar una selección de comportamientos a probar EFECTIVA y EFICIENTE. Es imposible realizar pruebas exhaustivas.
- En ningún caso vamos a demostrar que el software no tiene defectos. Cuando ejecutamos los tests sólo podremos demostrar la presencia de defectos. .Por lo tanto, como testers, buscaremos detectar el máximo número de defectos posibles (efectividad) con el menor número posible de casos de prueba (eficiencia)
- Para detectar la presencia de un defecto en el código, vamos a EJECUTAR dicho código. Esta forma de proceder para detectar defectos se conoce como pruebas DINÁMICAS, y es la aproximación que vamos a seguir durante todo el curso.