P01a+P01b: Entorno de pruebas

Máquina virtual de la asignatura

Para las prácticas en el laboratorio utilizaremos una Máquina Virtual en la que tendréis instalado el software que vamos a necesitar. En el laboratorio estará accesible desde el escritorio cuyo icono lleva el nombre “PPSS-2019-20".

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, así como GuestAdditions) y crear una nueva máquina virtual a partir del fichero *"ppss-2019-20.vdi"*.

Podéis descargar el fichero comprimido ("*ppss-2019-20.vdi.zip*") desde el siguiente enlace: https://drive.google.com/open?id=1FBD03tcDw7sz0qVETm9j2V-WdJmWhr89 (el descompresor 7zip suele funcionar bien)

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-2019-20.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.

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.

Herramientas adicionales que vamos a usar

➪ Git (Gestión de versiones) ➪ Maven (Construcción o build) ➪ IntelliJ (IDE)

➪ ➪ Git: repositorio (almacenaremos el repositorio Git 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).

Repositorio GIT en Bitbucket

Una vez que tengáis creada la cuenta, deberéis 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:

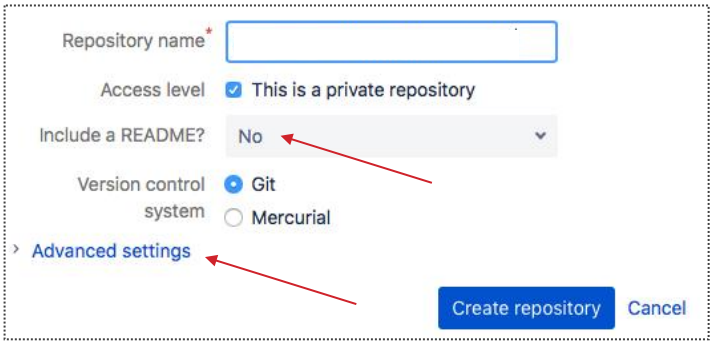
El repositorio tendrá como nombre: **ppss-2020-Gx-apellido1-apellido2,** en donde:

- ***Gx*** es el identificador del grupo de prácticas al que asistís. Los valores posibles son: G1..G9 según la siguientes tablas:

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
Martes de 17 a 19h	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



EL repositorio será **PRIVADO**.

OJO: **No** incluyáis un README.

Deberéis 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 "*Settings→Users and group access*" de vuestro proyecto (desde el panel izquierdo).

comando git clone

(para usar el repositorio en local)

Una vez creado el repositorio (vacío), vamos a ver cómo "descargarlo" en nuestra máquina. Una posibilidad es **CLONAR** el repositorio en algún directorio de nuestra máquina virtual. 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 seleccionado desde Bitbucket, a través de las opciones *"+ → Clone this repository"*, y copiamos dicho valor.

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-2020-Gx-apellido1-apellido2
```

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

```
$HOME/practicas/ppss-2020-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 cosa que añadamos en el directorio *$HOME/practicas/ppss-2020-Gx-apellido1-apellido2/* podrá estar sujeto al control de versiones de Git.

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 $HOME/.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)

A continuación creamos el fichero de texto oculto .gitignore (fíjate bien que tiene que comenzar por "punto" para que sea un fichero oculto). Éste indicará qué contenidos de nuestro directorio de trabajo queremos que sean ignorados por Git (no estarán sujetos a la gestión de versiones, en otras palabras, no nos importará "perderlos" si los borramos o modificamos su contenido).

contenido del fichero oculto .gitignore

(crear sólo la 1ª vez)

```
# ignore Maven generated target folders
target
# ignore Mac generated folders
/__MACOSX
.DS_STORE
```

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), por lo tanto "le diremos a Git" que ignore ese directorio y todo su contenido.

De la misma manera, ignoraremos las carpetas ***__MACOSX*** y ***.DS_STORE*** si la hubiera, que contiene datos irrelevantes para nuestro trabajo de prácticas, y que os aparecerán normalmente cuando descomprimáis ficheros que han sido comprimidos desde un mac.

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). Podemos comprobar que el fichero *.gitignore* aparece en rojo.

Para subir a Bitbucket el fichero creado (y en general, para subir cualquier cambio que hayamos hecho en el directorio de trabajo), primero tendremos que "marcar" los ficheros que queremos añadir a nuestro repositorio local (para ello usamos el comando ***git add***). A continuación ejectuar el comando ***git commit*** (para "copiar" todos los ficheros marcados en nuestro repositorio *git* local, y finalmente ejecutar ***git push*** (para "copiar" nuestro repositorio git local en Bitbucket).

Comandos para "guardar" todo nuestro trabajo en local en Bitbucket

(ejecutar SIEMPRE)

```
> git add .
> git commit –m "Añadido .gitignore"
> git push
```

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 los cambios en local. Recuerda que SIEMPRE deberás hacerlo desde $HOME/practicas/ppss-2020-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-2020-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-2020-apellido...***)
3. subir los cambios a Bitbucket (***git add + git commit + git push***)

➪ ➪ Maven

Maven es una **herramienta de construcción** de proyectos Java. Por definición, la construcción de un proyecto (*build*) es la secuencia de tareas que debemos realizar para, a partir del código fuente, poder usar 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 a realizar para construir proyectos. Cada una de estas secuencias se denomina **ciclo de vida**. El ciclo de vida más utilizado 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 1 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 tipo de empaquetado, tiene asociadas POR DEFECTO ciertas GOALS.

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.mxl (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).

Algunos ejemplos de goals incluidas por defecto cuando el empaquetado es ***jar*** son:

- La GOAL *compiler:testCompile* está asociada por defecto a la fase del ciclo de vida ***test-compile*** y realiza la compilación del código de pruebas del proyecto (cuyo código deberá estar en necesariamente el directorio /src/test/java). Esta goal se ejecutará automáticamente durante la fase test-compile.
- La GOAL *compiler:compile* está asociada por defecto a la fase del ciclo de vida ***compile*** y realiza la compilación del código fuente del proyecto (que deberá estar necesariamente en el directorio /src/main/java). Esta goal 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.

pom.xml

(permite configurar el build script del proyecto)

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>), ... entre otras cosas.

artefactos Maven

(son ficheros que se identifican por sus coordenadas))

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:proyecto3:war:1.0-SNAPSHOT*** representa al fichero $HOME/.m2/repository/org/ppss/proyecto3/1.0-SNAPSHOT/proyecto3-1.0-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.

Por otro lado, cualquier LIBRERÍA EXTERNA (ficheros .jar) que utilicemos en nuestro proyecto Maven, puede 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 descargarse también esta última, y así sucesivamente. Esto hace que nuestros proyectos "pesen" 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 respositorio 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.

En la máquina virtual proporcionada ya hemos ejecutado Maven y por lo tanto, ya existe el directorio $HOME/.m2

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). 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

➪ ➪ ➪ LICENCIA educacional IntelliJ

Para poder usar la versión Ultimate de IntelliJ necesitáis solicitar, cada uno de vosotros, una licencia educacional desde 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 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.

➪ ➪ 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 *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

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

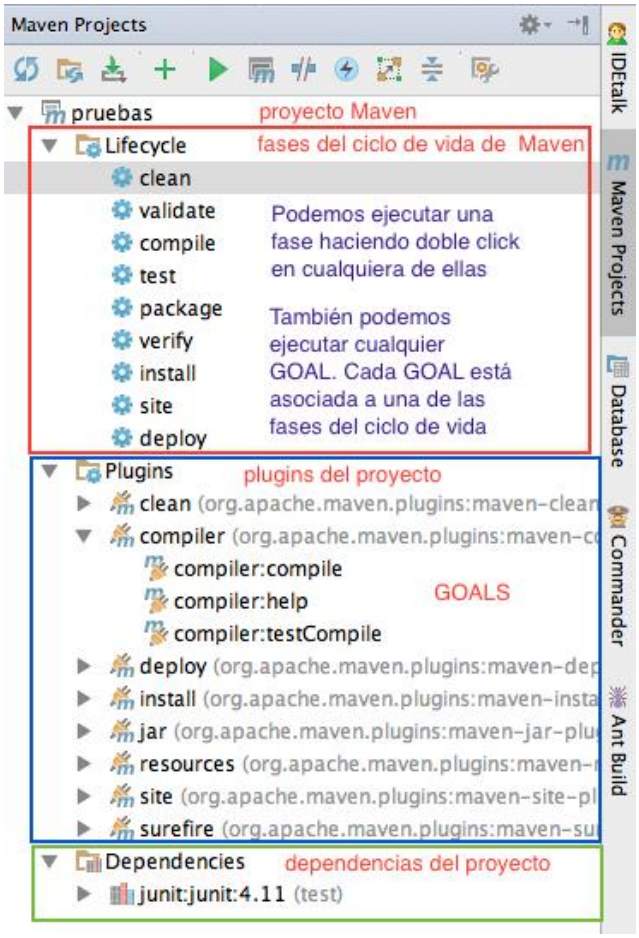
- Desde el menú principal elegimos File→Open (u opción Open cuando abrimos IntelliJ)
- En el cuadro de diálogo seleccionamos el fichero pom.xml, y pulsamos OK. Nos preguntará si queremos abrir como fichero o como proyecto. Elegiremos como proyecto.

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 cuatro 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**). El propósito de esta práctica es familiarizarnos con el entorno de trabajo: IntelliJ, Maven, Git, e introducir el proceso de automatización de pruebas.

Para poder hacer los ejercicios necesitarás:

Git/ Bitbucket

- **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-2020-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

Jetbrains

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 1.8 está seleccionada, desde *File→Project Structure→Platform Settings→SDKs*, y que el proyecto tiene asignada dicha versión (desde *Project Settings→Project→Project SDK*).

A continuación realiza lo siguiente:

A) Anota 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**).

Maven *Directory Layout*

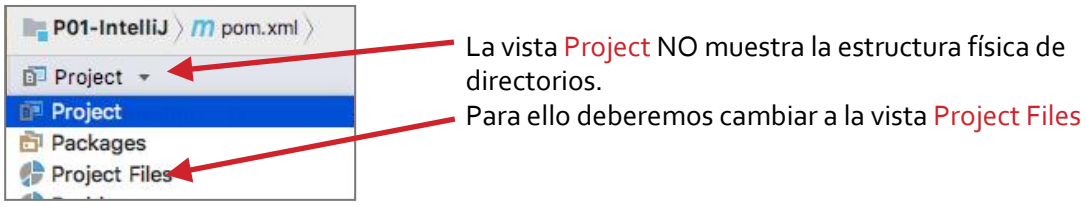


Figura 1. Un mismo proyecto se puede visualizar de formas diferentes.

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>)

Maven *pom.xml*

Nuestro pom define las coordenadas de nuestro proyecto, y además hace referencia a dos artefactos: uno en la sección de dependencias y otro en la sección <build>, indica cuáles son sus coordenadas: tanto las del proyecto, como la de los dos artefactos referenciados en él. ¿Por qué nuestro proyecto tiene definidas unas coordenadas?

¿Por qué los artefactos de nuestro pom están en secciones (etiquetas) diferentes y qué tipo de ficheros son?

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). Averigua qué versión viene incluida por defecto comentando el plugin en el pom, y consultando la información del plugin desde la ventana Maven Projects.

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: "Equilatero", "Isosceles", 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.

Maven Source Code

Este es uno de los ejemplos más utilizados en la literatura sobre pruebas, quizá 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 es 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. ¿Cuáles son exactamente? Identifícalos como C1, C2, C3, y C4, y muéstralos en una tabla con cuatro filas con la siguiente información:

Maven Test Code

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

Esta tabla se denomina "Tabla de casos de prueba". Recuérdala porque la utilizaremos en sesiones posteriores cuando diseñemos los casos de prueba.

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

Maven Test Code

Guarda tu trabajo en Bitbucket (contesta a las preguntas usando ficheros de texto). Ver los comandos git al final del documento.

➪ ➪ 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 (*File→Settings→Build, Execution, Deployment→Build Tools→Maven*). Cuando ejecutes "mvn compile", el resultado se muestra automáticamente una ventana en la parte inferior del IDE en donde verás los mensajes que genera Maven durante el proceso de construcción.

Maven Build (compile)

A) Observa que ha aparecido un directorio nuevo en nuestro proyecto maven (en el panel de la izquierda). Anota 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. Ahora vamos a ejecutar la **fase "clean"**. Observa lo que ocurre y anota 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.

Maven Build (clean)

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:

Maven Build (test)

"**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 icono "√" 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).

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. Obviamente 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 (repite el proceso hasta que los cuatro tests estén en "verde", y el proceso de construcción termine con: BUILD SUCCESS

debugging

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 razona la conveniencia o no de incluir C5 al conjunto de tests. De la misma forma razona si son necesarios los tests C2 y C3. Basándote en tu razonamiento anterior, piensa en dos posibles casos de prueba adicionales que "aporten valor" al conjunto de casos de prueba (no sean innecesarios) justificando tu respuesta.

Test Case Design

Guarda tu trabajo en Bitbucket. Ver los comandos git al final del documento.

➪ ➪ 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

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, ¿son suficientes o deberíamos añadir alguno más?

Test Case Design

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.

C) Pregunta a tus compañeros si han sido capaces de encontrar errores en el código ejecutando alguno de los tests implementados. Indica en qué te has basado para decidir los 5 nuevos casos de prueba que te hemos pedido, y contrasta con tus compañeros en qué se han basado ellos para tomar su decisión.

Test Case Code/Run

Guarda tu trabajo en Bitbucket. Ver los comandos git al final del documento.

➪ ➪ Ejercicio 4

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

A) Ejecuta la **fase “package”** y observa los cambios en la ventana del proyecto. Anota qué acciones se han llevado a cabo para construir el proyecto y qué artefactos nuevos se han generado. Justifica el tipo de dichos nuevos artefactos. 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, observa lo que ocurre y explica por qué ocurre esto. 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 “package”. Explica lo que ha pasado y por qué.

Maven Build (package)

B) 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. Indica la ruta exacta del artefacto generado y justifica el por qué de dicha ruta. 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. 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)

Puesto que para la construcción de nuestro proyecto hemos necesitado utilizar junit, dicha librería estará también en nuestro repositorio local. Búscala e indica exactamente la ruta en la que se encuentra.

Guarda tu trabajo en Bitbucket. Ver los comandos git al final del documento.

➪ ➪ Ejercicio 5

La clase **DataArray** representa una colección de datos enteros mayores que cero almacenados en un Array. Inicialmente cualquier objeto de tipo DataArray tendrá cero elementos (y las posiciones no ocupadas del array tendrán el valor cero).

A continuación mostramos la especificación de los métodos add(int) y delete(int), que añaden y borran un elemento en la colección, respectivemente.

El método **add(int)** sólo añade un nuevo elemento si el entero que se quiere añadir es mayor que cero y no hemos alcanzado el límite de elementos, que será 10. El elemento se añade en la primera posición vacía del array, y el número de elementos se incrementa en uno si se consigue añadir el elemento. No importa si el elemento a añadir tiene un valor repetido en la colección, en este caso, se añadirá de nuevo, en la posición que le corresponda (en la posición contigua al último hueco libre, siempre y cuando no se alcance el máximo de elementos).

El método **delete(int)** borra el entero especificado como parámetro de la colección de enteros almacenados en el array. El proceso de borrado no tiene ningún efecto en la colección de elementos si el elemento a borrar no pertenece a la colección. Además, si se hace efectivo el borrado, no deben quedar "huecos", de forma que los siguientes elementos al elemento borrado se desplazarán una posición hacia la izquierda en el array, y lógicamente el número de elementos se decrementará en uno. Las posiciones del array vacías deberán contener el valor cero. El método devuelve el array de elementos resultante después de realizar el borrado.

A partir de la especificación anterior, se pide:

A) Crea una **tabla** para el método *delete()* con los **casos de prueb**a que pienses que deberías probar para asegurarte de que una futura implementación es correcta. Recuerda: una fila por caso de prueba. El identificador del caso de prueba tendrá que aparecer en el nombre del método correspondiente que implemente dicho caso de prueba.

Test Case Design

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

B) **Implementa el método delete()**, y asegúrate de que no tiene errores de compilación

Source Code

C) Vamos a crear la clase que contendrá los tests para los métodos *add()* y *delete()*. Lo haremos a través del asistente de IntelliJ. Para ello seleccionamos en el código (DataArray.java) el nombre de la clase "DataArray" y a continuación pulsamos el botón derecho del ratón (para mostrar el menú contextual) y elegimos la opción "Generate... → Test...". La librería a utilizar debe ser "Junit5", marcaremos las casillas de los métodos add() y delete() y pulsamos sobre "OK". Esto nos creará una nueva clase java denominada **DataArrayTest** en el directorio src/test/java (dentro del paquete ppss), con dos métodos anotados con @Test. Una vez creada la clase **implementa tres tests para el método add()**, considerando estas situaciones:

Test Case Code

- Añadir un elemento a una colección vacía
- Añadir un elemento a una colección de más de un elemento
- Añadir un elemento a una colección llena

Para implementar los tests, puedes generar de forma automática el prototipo de cada método seleccionando el nombre de la clase de puebas ("DataArrayTest"), y eligiendo desde su menú contextual, la opción "Generate...→ Test Method...".

D) Implementa los tests de la tabla del apartado anterior para el método delete(). Depura los errores para que todos los tests estén en "verde".

Test Case Code/Run/Debug

Guarda tu trabajo en Bitbucket. Ver los comandos git en el siguiente apartado.

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-2020-Gx-apellido1-apellido2):

Git/ Bitbucket

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

Y ya para terminar...

MUY IMPORTANTE:

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.

> Recuerda también que la asimilación de los conceptos requiere una dedicación PERSONAL de forma **CONTINUADA** durante TODO el cuatrimestre.

La evaluación será individual, en presencia del profesor, y de forma ESCRITA. Por lo que es muy importante que seáis capaces de EXPRESAR, de forma clara y ordenada, y por ESCRITO, los conocimientos teórico-prácticos adquiridos.

> Por todo ello te aconsejamos el comenzar a trabajar desde YA en el ejercicio de ciertas actividades, como tomar apuntes en clase, analizar y anotar la "conexión" de cada sesión práctica con la de teoría, y muy especialmente, a destacar las ideas fundamentales y posteriormente trabajar el detalle (y no al revés). Todo esto os facilitará que consigáis los objetivos planteados.

Resumen

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

TEORIA — PRACTICA

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 de construcción maven puede ser "Build fauilure" 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 verifica que: dadas unas determinadas entradas sobre el elemento a probar, su ejecución genera un resultado idéntico al esperado (es decir, estamos comprobando si el conjunto S coincide con el conjunto P)
- Dependiendo de cómo hayamos diseñado los casos de prueba, detectaremos más o menos errores (discrepancias entre el resultado esperado y el resultado real).
- Hay diferentes tipos de tests, en función de su objetivo, el instante de tiempo en que se realiza y las técnicas usadas

INTELLIJ

- Es un IDE que usaremos para implementar código de pruebas en proyectos java con maven.