

IWVG. Ecosistema Software.

<https://www.youtube.com/watch?v=B5WLg97hqng&list=PLj2IVmcP-QN1sxo5Yw4nFJy3zgm3LeJS>

- Video 1: Introducción. IDE

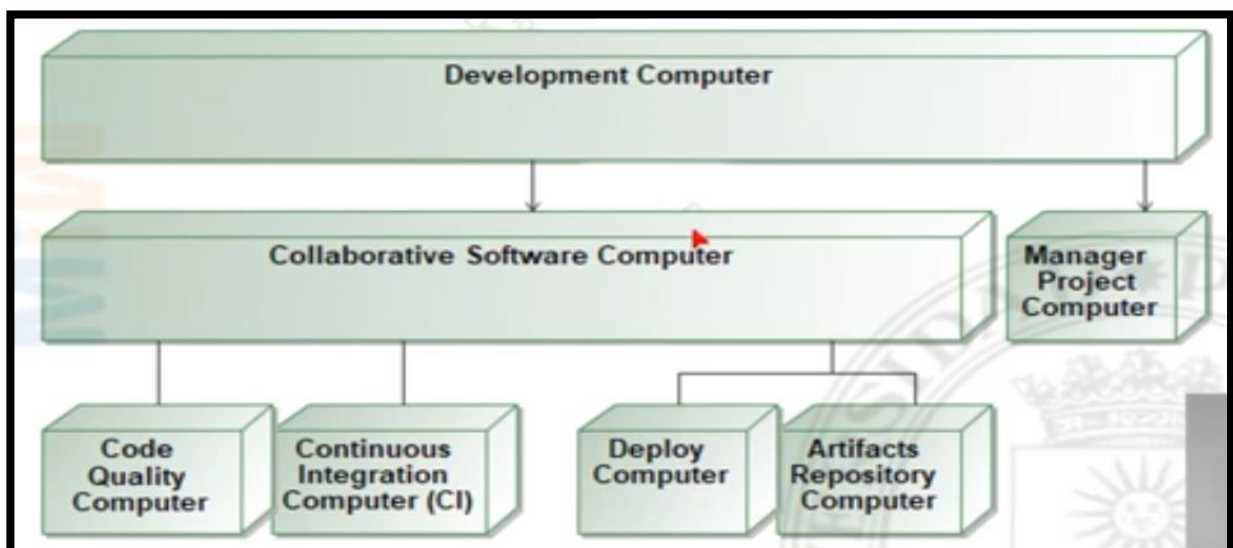
Definición de ecosistema de software:

IWVG > Ecosistema > Introducción miw.etsisi.upm.es • 2

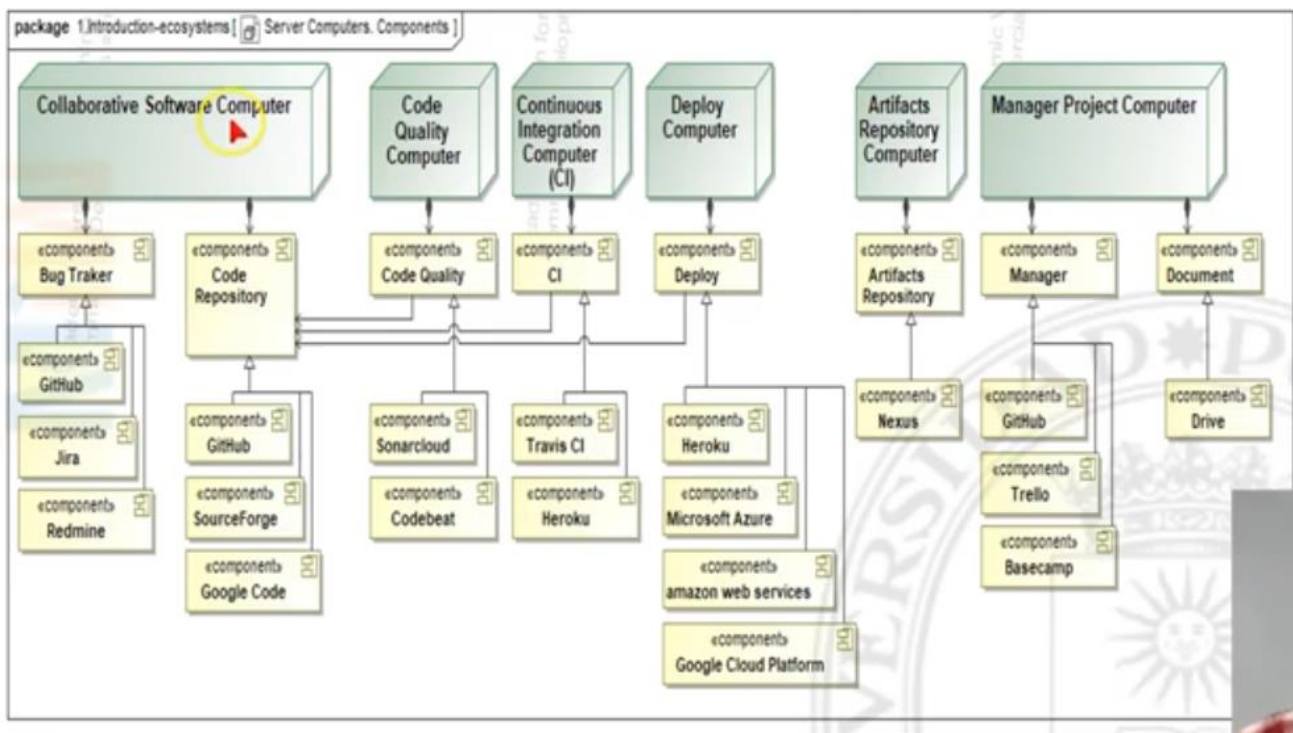
Introducción

- **Ecosistema. RAE**
 - Comunidad de los seres vivos cuyos procesos vitales se relacionan entre sí y se desarrollan en función de los factores físicos de un mismo ambiente
- **Ecosistema software**
 - Se define como un espacio de trabajo donde un conjunto de herramientas interactúan y funcionan como una unidad para el *desarrollo de software colaborativo* en todas sus fases (Gestión, Requisitos, Análisis, Diseño, Programación, Pruebas y Despliegue)
- **Forja**
 - Una forja es una plataforma para desarrollo colaborativo de software
- **Entorno de Desarrollo Integrado (IDE)**
 - Se define como una aplicación informática que proporciona servicios integrales para facilitarle al desarrollador de software

Software como un Servicio, abreviado ScuS1 (del inglés: Software as a Service, SaaS), es un modelo de distribución de software donde el soporte lógico y los datos que maneja se alojan en servidores de una compañía de tecnologías de información y comunicación (TIC), a los que se accede vía Internet desde un cliente. La empresa proveedora TIC se ocupa del servicio de mantenimiento, de la operación diaria y del soporte del software usado por el cliente. Regularmente el software puede ser consultado en cualquier computador, se encuentre presente en la empresa o no. Se deduce que la información, el procesamiento, los insumos, y los resultados de la lógica de negocio del software, están hospedados en la compañía de TIC.



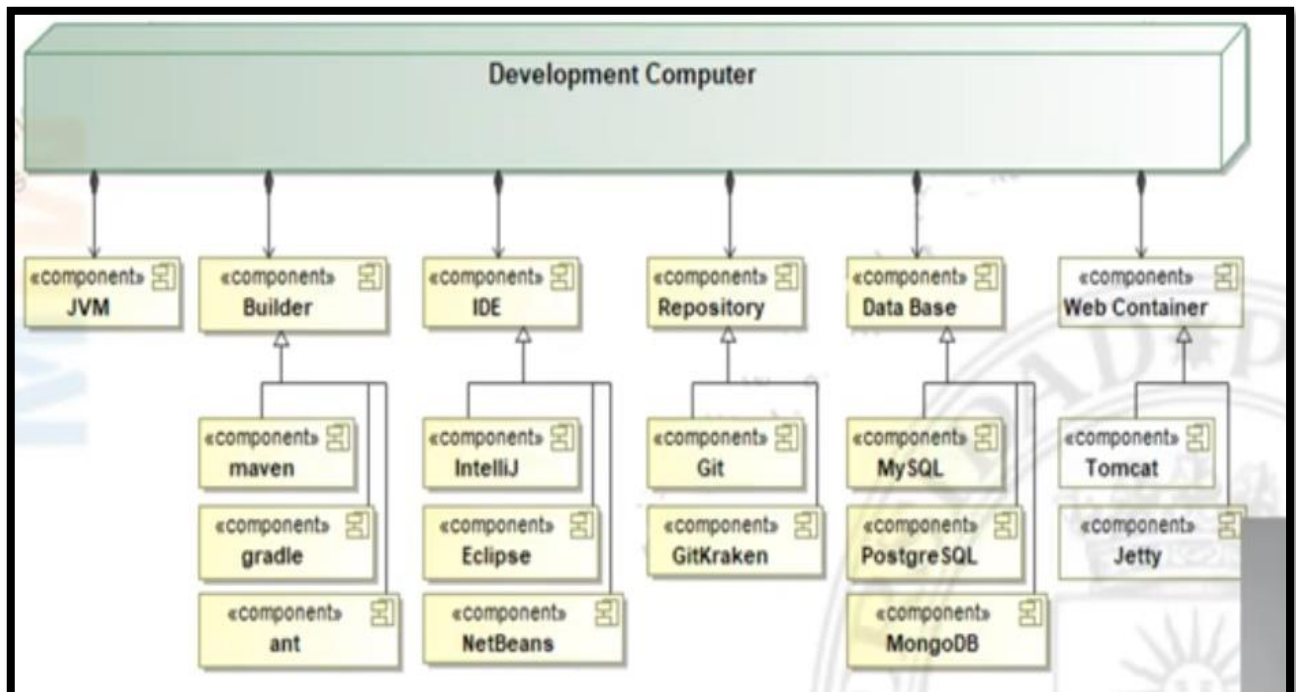
Software as a Service (SaaS)



Equipo de desarrollo:

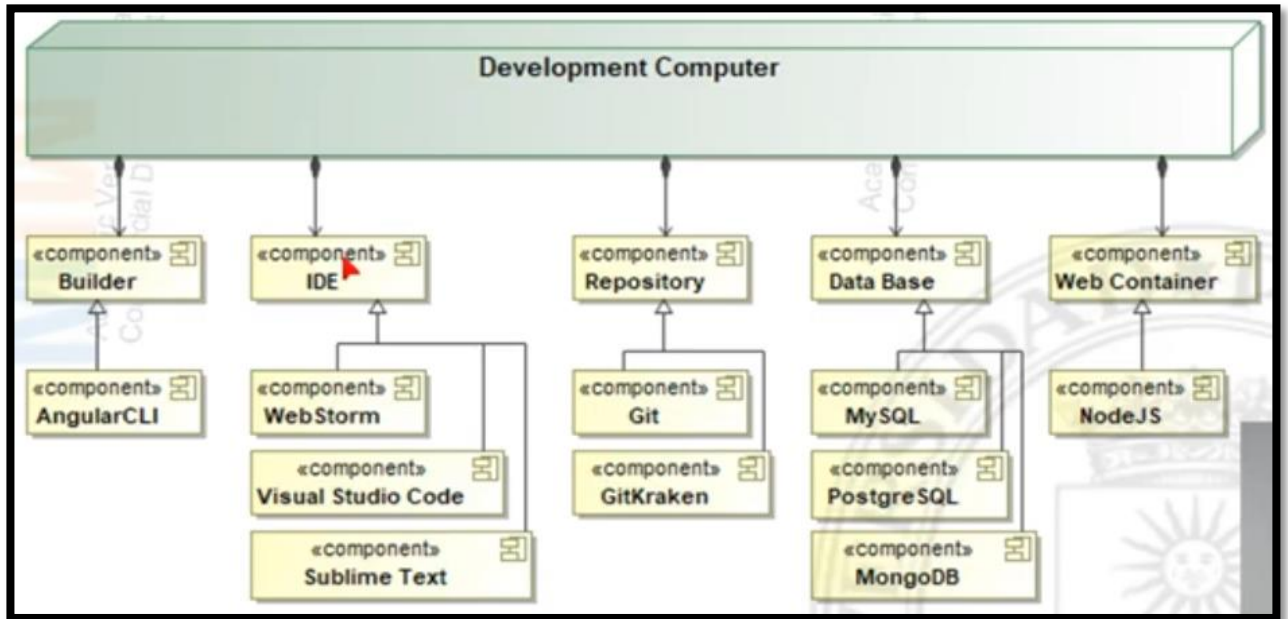
cada ecosistema que armemos depende de la tecnología a trabajar. Veamos que necesita un equipo de desarrollo para cada tecnología:

- Java.



Concepto: Una **forja** es una plataforma de desarrollo colaborativo de software. Se enfoca hacia la cooperación entre desarrolladores para la difusión de software y el soporte al usuario. En este tipo de plataformas se albergan múltiples proyectos de software, en los que los desarrolladores han de registrarse para poder contribuir. Consta de numerosas aplicaciones normalmente con interfaz web para la administración y desarrollo de estos proyectos en común.

-Angular



Para la creación del proyecto necesitamos:

- Instalar JRE y JDK
- Instalar apache-maven

Conceptos de Maven:

- **Artefacto:** Es la unidad mínima con la que trabaja Maven para gestionar sus dependencias, son los componentes software
- **Coordenadas:** Sistema con el Maven determina de forma única a cada uno de sus artefactos: *groupId:artifactId:version*
 - *Group Id:* identificación del grupo. Normalmente se utiliza el nombre del dominio, al revés: *es.upm.miw.tuvvg*
 - *Artifact Id:* identificación del artefacto: *forge*
 - *Version:* versión del artefacto: *1.0.0-SNAPSHOT, RC-1.3.4 (Release Candidate), R-1.4.5 (Release)*
- **Empaquetado:** tipo de fichero generado, normalmente *JAR* o *WAR*
- **Repositorio:** Estructura de directorios y archivos que usa Maven para almacenar, organizar y recuperar artefactos. Existen repositorios locales, privados y remotos
 - El repositorio por defecto es el repositorio central de Maven
 - Existe un repositorio privado en la empresa para los artefactos de desarrollo
 - Existe un repositorio local, donde se copian las dependencias:
%User%/.m2/repository
- **Arquetipos:** Son plantillas para definir proyectos tipo con el fin de ser reutilizados

Comandos de consola:

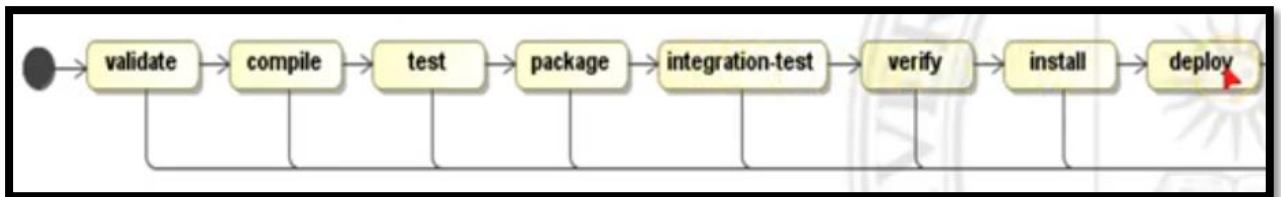
- > `mvn clean`: elimina todo lo generado en construcciones anteriores.
- > `mvn -version`
- > `mvn compile`
- > `mvn -Dmaven.test.skip=true package`
- > `mvn clean verify`

Ciclos de vida

▪ Default Lifecycle

- *validate* : Valida el proyecto si es correcto
- *compile* : Genera los ficheros .class compilando los fuentes .java
- *test* : Ejecuta los test automáticos de JUnit existentes, abortando el proceso si alguno de ellos falla.
- *package* : Genera el empaquetado final (jar, war...)
- *integration-test*: Procesar y desplegar el paquete si fuera necesario en un entorno donde se puedan ejecutar pruebas de integración
- *verify*: Ejecutar todas las comprobaciones para verificar la validez del paquete y que cumpla con los criterios de calidad
- *install* : Copia el paquete en un directorio de nuestro ordenador, de esta manera pueden utilizarse en otros proyectos maven del mismo ordenador
- *deploy* : Copia el paquete *jar* a un servidor remoto, poniéndolo disponible para cualquier proyecto maven con acceso a ese servidor remoto

Los comandos se ejecutan en orden, y cada uno realiza el anterior a menos que se le diga lo contrario (esto se hace con skip).



Plugin de objetivos

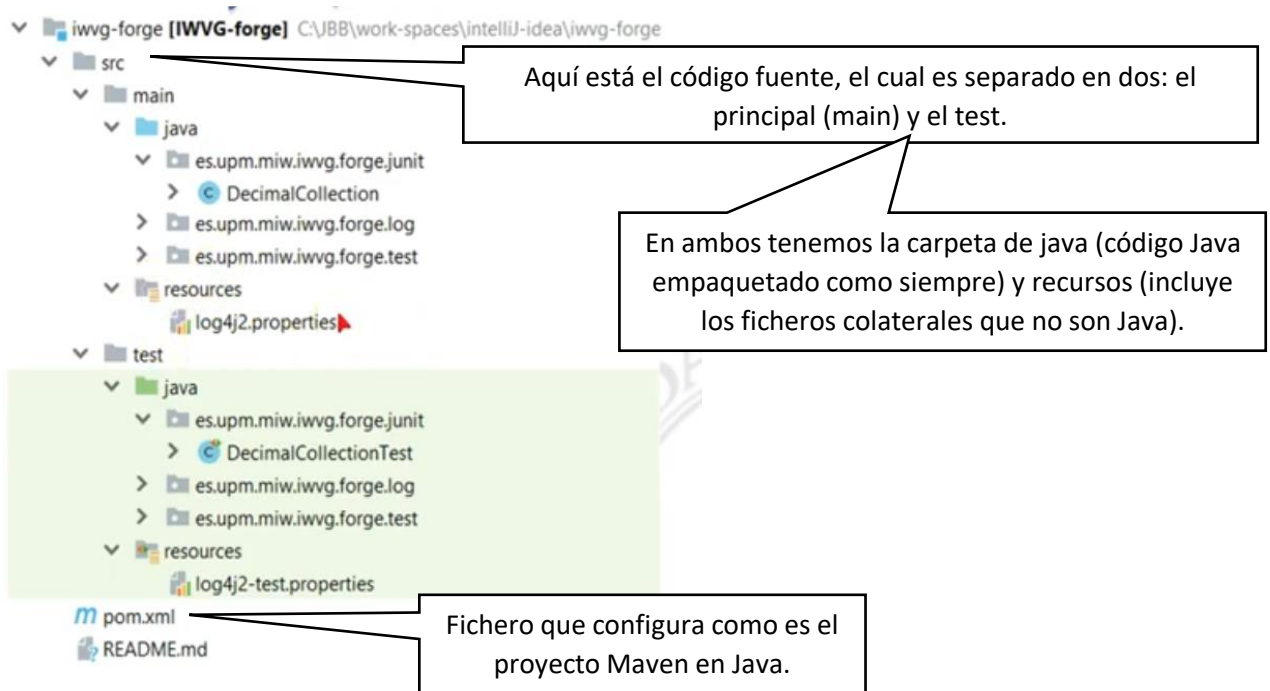
Es una tarea específica, más pequeña que una fase de construcción, que contribuye a la construcción y gestión del proyecto.

En el ciclo de vida, va después de *package*.

Comando: *[nombre del plugin]:[función del plugin a ejecutar]*

- `mvn sonar:sonar`
- `mvn spring-boot:run`

Organización de un proyecto en Java con Maven:



IDE a utilizar: IntelliJ IDEA

- IntelliJ IDEA es un entorno de desarrollo integrado (IDE), tiene una versión gratuita: *Community Edition*.
- Es una buena plataforma para el desarrollo de aplicaciones Java
 - <https://www.jetbrains.com/idea/download>
 - Versión: *Community*

Cliente GIT

- Git es un software de control de versiones, distribuido, gratuito y de código abierto
- Sitio Web: <https://git-scm.com>
- Documentación:
 - <https://git-scm.com/book/en/v2>
 - <https://git-scm.com/book/es/v2>
- **Cliente:**
 - <https://git-scm.com/download/win>
- **GUI:**
 - <https://git-scm.com/downloads/guis/>
 - <https://www.gitkraken.com/>

Primeros pasos:

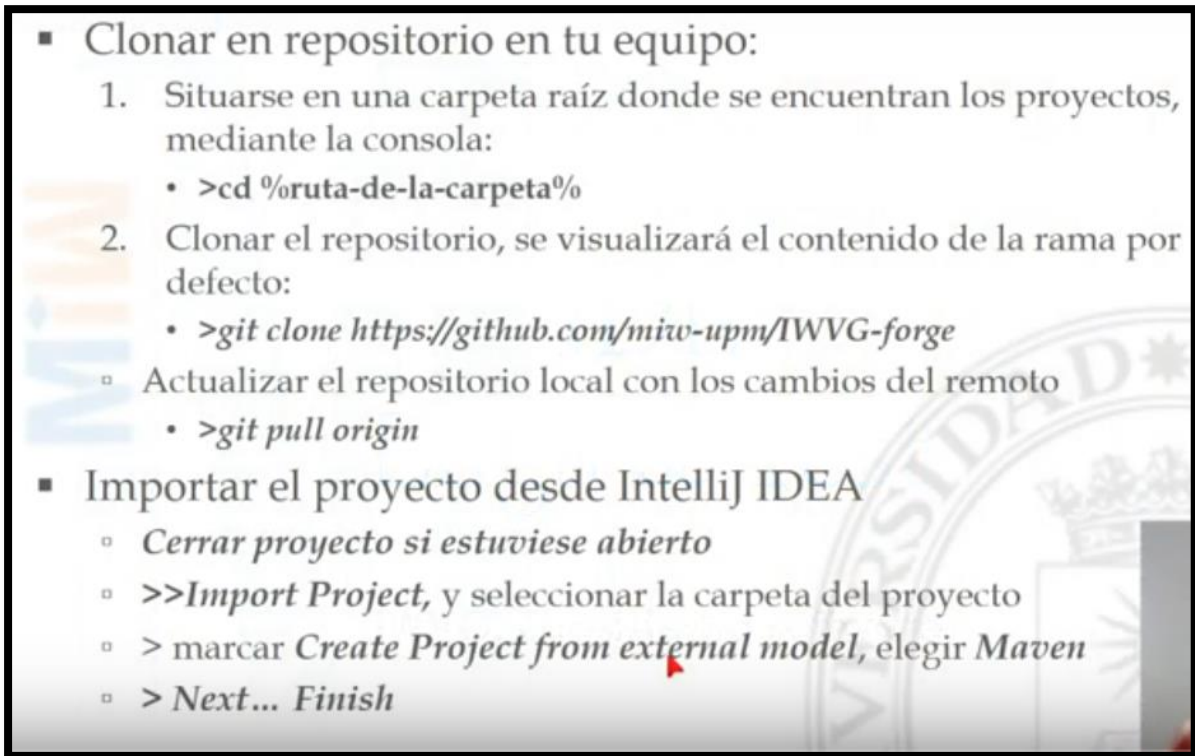
1. Instalar SDK de Java. Definir variables de entorno
2. Instalar *Maven*. Definir variables de entorno
 - Se manejará maven desde la consola, nos facilitará la creación de scripts
3. Instalar cliente Git
4. Instalar IntelliJ IDEA
5. Crear la carpeta de los workspace
6. Crear un proyecto nuevo tipo maven en el workspace. Se ofrece una plantilla a modo de ejemplo:
 - <https://github.com/miw-upm/IWVG-forge/blob/develop/docs/maven.zip>
 - Recordar cambiar el nombre de la carpeta y del proyecto en el fichero *pom.xml*
7. Importar el proyecto desde IntelliJ IDEA
 - *Cerrar proyecto si estuviese abierto*
 - **>>Import Project**, y seleccionar la carpeta del proyecto
 - **> marcar Create Project from external model**, elegir *Maven*
 - **> Next... Finish**

Comandos para corroborar por consola que Maven y Git Cliente están instalados:

- `mvn -version`
- `git --help`

- Video 2.

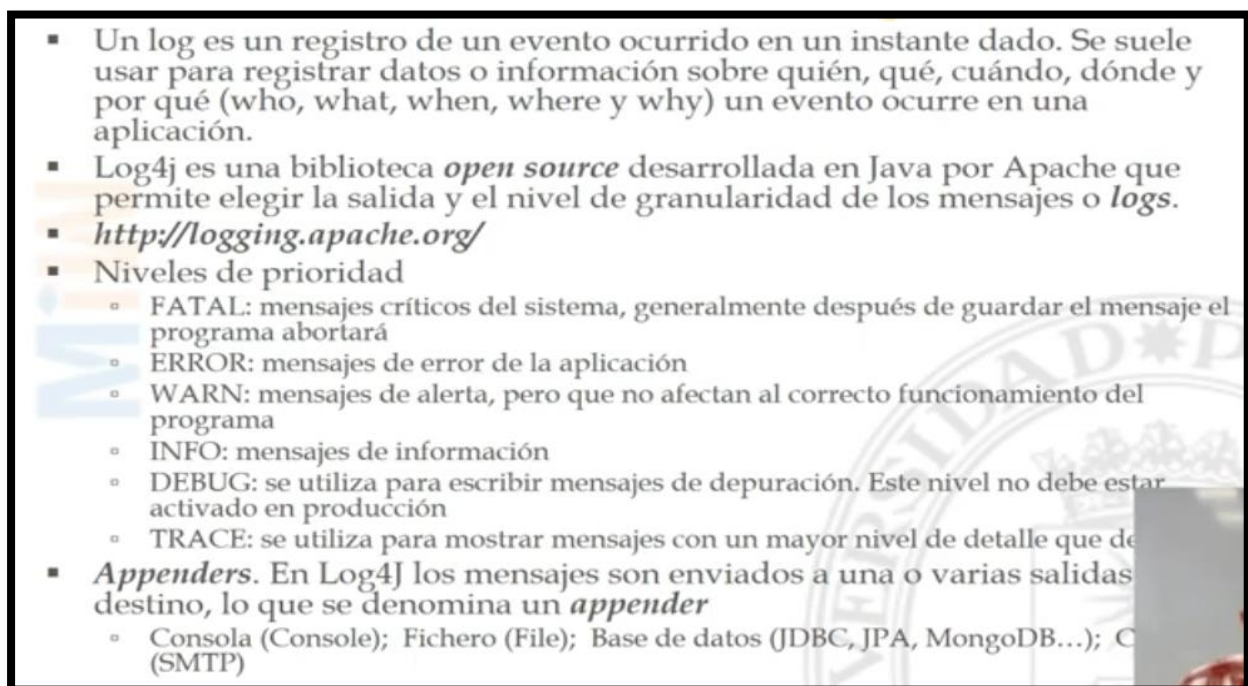
Clonar un proyecto Maven desde Github



- **Clonar en repositorio en tu equipo:**
 1. Situar en una carpeta raíz donde se encuentran los proyectos, mediante la consola:
 - `>cd %ruta-de-la-carpeta%`
 2. Clonar el repositorio, se visualizará el contenido de la rama por defecto:
 - `>git clone https://github.com/miw-upm/IWVG-forge`
 - Actualizar el repositorio local con los cambios del remoto
 - `>git pull origin`
- **Importar el proyecto desde IntelliJ IDEA**
 - *Cerrar proyecto si estuviese abierto*
 - `>>Import Project`, y seleccionar la carpeta del proyecto
 - `>` marcar *Create Project from external model*, elegir *Maven*
 - `> Next... Finish`

<https://github.com/miw-upm/iwvg-ecosystem>

Gestión de registro: Log4j



- Un log es un registro de un evento ocurrido en un instante dado. Se suele usar para registrar datos o información sobre quién, qué, cuándo, dónde y por qué (who, what, when, where y why) un evento ocurre en una aplicación.
- Log4j es una biblioteca *open source* desarrollada en Java por Apache que permite elegir la salida y el nivel de granularidad de los mensajes o *logs*.
- <http://logging.apache.org/>
- **Niveles de prioridad**
 - FATAL: mensajes críticos del sistema, generalmente después de guardar el mensaje el programa abortará
 - ERROR: mensajes de error de la aplicación
 - WARN: mensajes de alerta, pero que no afectan al correcto funcionamiento del programa
 - INFO: mensajes de información
 - DEBUG: se utiliza para escribir mensajes de depuración. Este nivel no debe estar activado en producción
 - TRACE: se utiliza para mostrar mensajes con un mayor nivel de detalle que de
- **Appenders.** En Log4J los mensajes son enviados a una o varias salidas destino, lo que se denomina un *appender*
 - Consola (Console); Fichero (File); Base de datos (JDBC, JPA, MongoDB...); C (SMTP)

- **Layouts.** Permite presentar el mensaje con el formato necesario para almacenarlo
 - Simple (*SimpleLayout*)
 - Avanzado con *patrón* (*PatternLayout*)
 - %-5level: nivel en 5 caracteres
 - %d{dd MMM-HH:mm:ss}: formato de fecha
 - %c{n}: nombre de clase con dominio de (n-1) paquetes
 - %t: método
 - %m%n: mensaje , con salto de carro (%n)
- **Configuración.** Permite varios sistemas de configuración (*properties*, *xml*, *json*, *yaml*, ...)
 - Por defecto: nivel de prioridad: *error* y appender: *Console*

En la sección main -> resources, encontramos la configuración de los log

Formato del log

Configuración de los ficheros incrementados: cada log de un día, te lo saca en un fichero

Nivel de log: Error

Consola: permite la visualización durante el desarrollo

Prueba de Log:

```
package es.upm.miw.iwvg.forge.log;

import ...

public class LoggerDemo {

    public void log() {
        Logger logger = LogManager.getLogger(this.getClass());
        logger.fatal("Log de fatal");
        logger.error("Log de error");
        logger.warn("Log de warning");
        logger.info("Log de info");
        logger.debug("Log de debug");
        logger.trace("Log de trace");
    }

    public static void main(String[] args) { new LoggerDemo().log(); }
```

Test

Las pruebas son importantísimos y ya no basta el `system.out.println`

■ Tipos

- **Pruebas Unitarias.** Los desarrolladores prueban correcto funcionamiento de un módulo de código o clase independientemente del resto. Si existen dependencias se rompen con los mocks, son clases que simulan a otras y aportan una funcionalidad limitada
- **Pruebas de Integración.** Los desarrolladores prueban los diferentes componentes que dependen de otros componentes
- **Pruebas Funcionales.** Los desarrolladores prueban al sistema como un todo
- **Pruebas de Aceptación.** Los clientes prueban la versión entregada

■ Característica de calidad:

- **Automática.** Clases que prueban clases
- **Cobertura:** % de líneas de código ejecutadas (>70%)
- **Repetibles.** Cuando parte del código ha sido modificado, se vuelven a lanzar las pruebas para comprobar que no se ha alterado su funcionalidad
- **Independiente.** Se prueban los módulos por separado

Hacer esto desde cero, es inviable. Por lo tanto, utilizamos Junit para realizar las pruebas.

- JUnit es un framework que nos ayuda a la realización de pruebas unitarias. Fue creado por Erich Gamma y Kent Beck.

- <http://www.junit.org/>

■ Test Case: Clases de prueba

- Una clase es un test, si tiene algún método de test
- Un test es un método con la anotación `@Test`
- Los test deben ser independientes entre sí, el orden no debe afectar al resultado
- El nombre del método debe describir el tipo de prueba. Empiezan con la palabra "test"

■ Test Suites: Contenedor de Test Case o Test Suites. Se crean estructura en árbol

■ Ciclo de vida

- `@BeforeAll`: Se ejecuta una sola vez antes de la batería de pruebas definida en la clase y el método debe ser *static*
- `@BeforeEach`: Se ejecuta antes de cada uno de los marcados con `@Test` (es decir, si existen varios test, se ejecuta varias veces). Suele ser una inicialización por todas las pruebas de la clase
- `@Test`: Marca un método como prueba
- `@AfterEach`: Se ejecuta después de cada uno de los `@Test`. Suele ser una liberación de recursos
- `@AfterAll`: Se ejecuta al final del proceso completo y el método debe ser *static*
- `@Disabled`: Marca un método o una clase completa para que no se ejecute
- `@Tag`: Para el filtrado de test
- `@DisplayName`: Define un nombre del test propio
- `@RepeatedTest`: Repetición de test
- `@ParameterizedTest`: Define test parametrizados con diferentes valores

Comprobaciones de JUnit:

- `assertEquals (valor_esperado, valor_real);`
 - Los valores pueden ser de cualquier tipo
 - Si son arrays, no se comprueban elemento a elemento, sólo la referencia
- `assertEquals (double_esperado, double_real, double_error);`
- `assertNotEquals`
- `assertArrayEquals (array[] esperado, array[] real)`
- `assertTrue (condición_booleana)`
- `assertFalse (condición_booleana)`
- `assertSame (Objeto esperado, Objeto real)`
 - Comprueba que son la misma referencia
- `assertNotSame`
- `assertNull (Objeto)`
 - Comprueba que el objeto es Null
- `assertNotNull (Objeto objeto)`
 - Comprueba que el objeto no es Null
- `fail ()`
 - Falla siempre
- `assertThrows`
 - Se debe lanzar una excepción
- `assertDoesNotThrow`
 - No se debe lanzar una excepción
- `assertTimeout`

Organización de los test:

- Paquetes
 - Fuentes: `src/main/java/**`
 - Pruebas: `src/test/java/**`
 - Se puede lanzar un solo test, los test de una clase o los test de un paquete y subpaquetes
- Léxico
 - Pruebas Unitarias
 - Si la clase se llama `**Test`, se ejecutarán en la fase Maven de `test`
 - Pruebas de Integración
 - Si la clase se llaman `**IT`, se ejecutan en la fase Maven de `integration-test`
- Metodología de Trabajo para mantenimiento del código
 1. Modificar el código de la clase y los test afectados
 2. Ejecutar los test de la clase para comprobar que todo sigue igual
 3. Ejecutar el test del paquete, para comprobar que todo sigue igual
 4. Ejecutar todos los test

Veamos un ejemplo de clases:

Java Class: POINT	Java Class POINT TEST
<pre>package es.upm.miw.iwvg.ecosystem.junit; public class Point { private int x; private int y; public Point(int x, int y) { this.x = x; this.y = y; } public Point(int xy) { this(xy, xy); } public Point() { this(0, 0); } public double module() { return Math.sqrt((double) this.x * this.x + this.y * this.y); } public double phase() { return Math.atan((double) this.y / this.x); } public void translateOrigin(Point origin) { this.x -= origin.getX(); this.y -= origin.getY(); } public int getX() { return this.x; } public int getY() { return this.y; } @Override public String toString() { return "Point{" + "x=" + x + ", y=" + y + '}'; } }</pre>	<pre>package es.upm.miw.iwvg.ecosystem.junit; import org.junit.jupiter.api.BeforeEach; import org.junit.jupiter.api.Test; import static org.junit.jupiter.api.Assertions.assertEquals; class PointTest { private Point point; @BeforeEach void before() { point = new Point(2, 3); } @Test void testPointIntInt() { assertEquals(2, point.getX()); assertEquals(3, point.getY()); } @Test void testPointInt() { point = new Point(2); assertEquals(2, point.getX()); assertEquals(2, point.getY()); } @Test void testPoint() { point = new Point(); assertEquals(0, point.getX()); assertEquals(0, point.getY()); } @Test void testModule() { assertEquals(3.6055, point.module(), 10e-5); } @Test void testPhase() { assertEquals(0.9828, point.phase(), 10e-5); } @Test void testTranslateOrigin() { this.point.translateOrigin(new Point(1, 1)); assertEquals(1, point.getX()); assertEquals(2, point.getY()); } }</pre>

- Video 3: Desarrollo de software colaborativo. Git.

Desarrollo de software colaborativo (Forja). Introducción.

- Consiste en un conjunto de aplicaciones para el desarrollo y mantenimiento de software de forma colaborativa
 - Sistemas de control de versiones / repositorio de código
 - Es un sistema para el almacenamiento y gestión de los diversos cambios que se realizan sobre un conjunto de ficheros, pudiendo recuperar cualquier versión
 - Listas de correo, foros y wiki
 - Sistemas de seguimiento de errores / tickets (*issues*)
- Tipos
 - Distribuidos. Aumenta la flexibilidad pero complica la sincronización y gestión. Ejemplos: Git, Mercurial... En la actualidad se están imponiendo estos sistemas
 - Centralizados. Dependiente de un responsable. Facilita la gestión pero reduce la potencia y flexibilidad. Ejemplos: CVS, Subversion...
- Forjas Web
 - GitHub. <https://github.com/>. Sitio web que lleva integrada la forja. Para código abierto es gratuito, existe una versión de pago para proyectos privados
 - Google Code. <https://code.google.com/intl/es/>
 - SourceForge. <http://sourceforge.net/>
 - Otras: BerliOS, Gforce, Savannah...

Sistema de control de version: Git

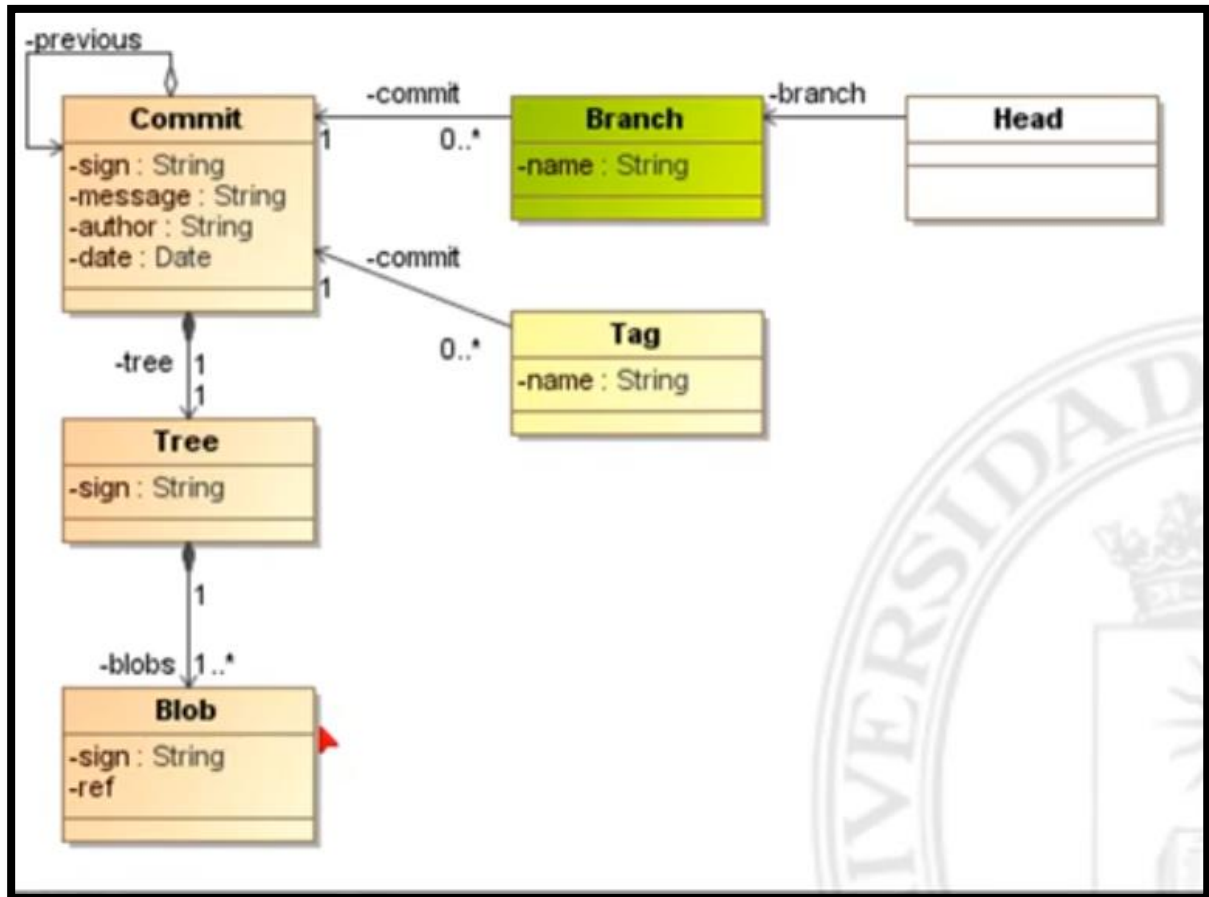
- Nace en 2005, tomando como experiencia el proyecto Bitkeeper (propietario). En 2008 nace GitHub, Forja en Web con repositorio basado en Git
- Documentación: <https://git-scm.com/book/es/v2>
- Características:
 - Control de versiones distribuido
 - Muy fiable, casi imposible perder el proyecto
 - Trabaja sin necesidad de conexión al remoto, muy rápido. Se podrá sincronizar con el remoto, **pero con asistencia...**
- Directorios
 - Locales
 - **Working directory:** área de trabajo. Es una copia descomprimida para puedan modificar (zona de trabajo del IDE)
 - **Git directory:** repositorio local, en formato comprimido (*.git)
 - Remoto
 - **Git directory:** repositorio remoto

¿en qué se basa nuestro repositorio de código?

En una instantánea o snapshot.

¿y esto qué es?

Es tomar el código completo tal y como está y tomarle una “foto” con un “commit”.



El repositorio de código almacena las instantáneas. Como han cambiado entre una y otra, no lo puedo saber.

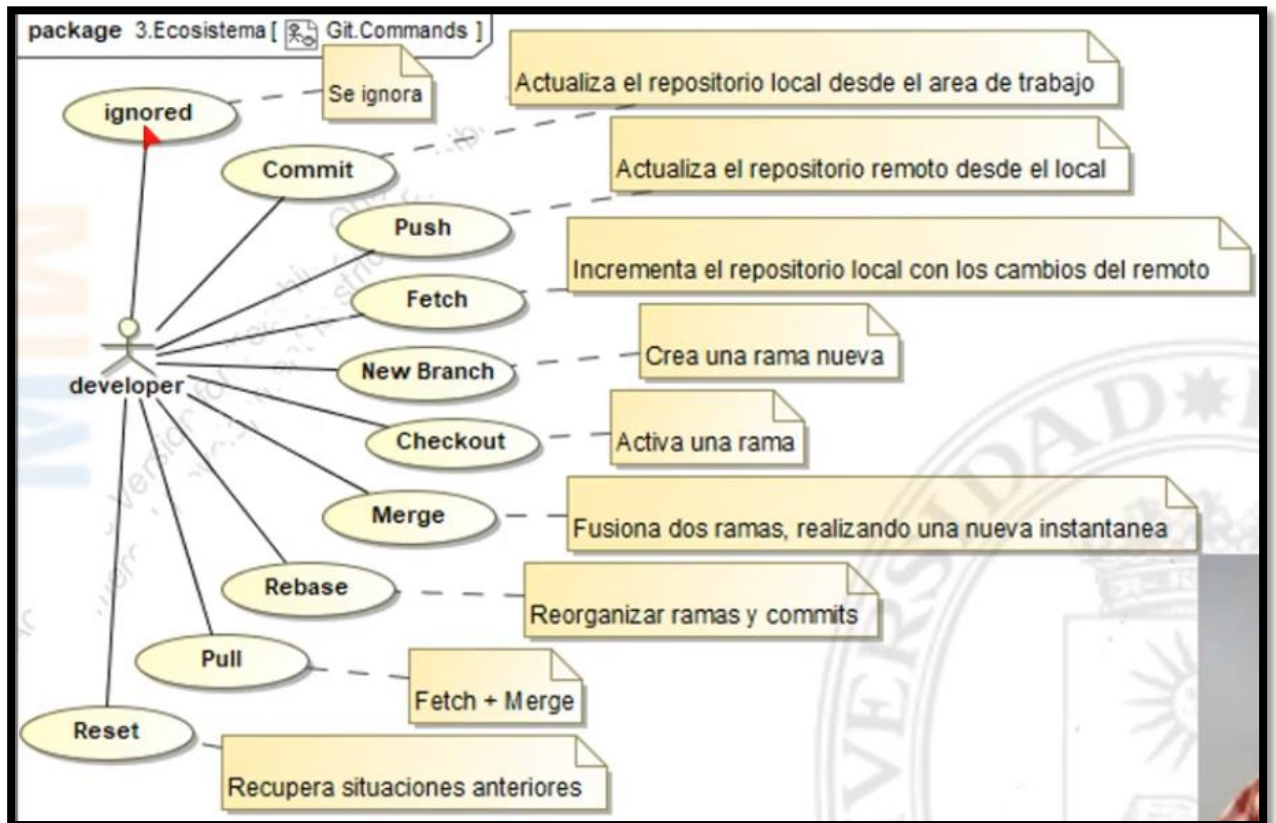
¿Qué cosas tiene una instantánea?

- Firma digital de su contenido para que nadie pueda cambiarlo
- Un mensaje
- El nombre del autor
- Fecha / hora

Al commit le apuntan las ramas. Una rama es una referencia de un commit. Estas ayudan a manejar el commit, pero el commit es lo que realmente me interesa.

La etiqueta también apunta al commit. La diferencia con la rama, es que las etiquetas no pueden cambiar de commit, pero las ramas evolucionan de commit.

Fundamentos y funciones de Git:



Comandos de Git:

- Sitio web: <https://git-scm.com/>
- Ayuda
 - `>git help --all`
 - `>git <command> -h`
- Consultas
 - `>git config --list`
 - `>git status`
 - `>git remote -v //--verbose`
 - `>git log // q para salir`
 - `>git reflog //q para salir. Referencia de los commits`
- Configurar Git
 - `>git config --global user.name " ... "`
 - `>git config --global user.email " ... "`
- Crear repositorio con todos los ficheros
 - `>git init`
 - `>git add --all`
 - `>git commit -m "mi mensaje de commit inicial"`

En el minuto 24 (aprox), muestra por terminal de IntelliJ como iniciar el proyecto en Git.

NOTA:

Con checkout cambiamos de rama. (`git checkout -b develop`), pero al poner -b, la estamos creando. Si no ponemos -b, y la rama no existe, protesta.

Error típico: al dar commit, evoluciona la rama activa. Si hemos cambiado de rama, evolucionamos una rama que no corresponde.

Recomendación: NUNCA hacer commit & push, si hubo un error, el error subió al remoto y es muy complejo arreglarlo.

¿Qué no hacer? NUNCA se cambia de rama sin cerrar un commit porque se va a perder el cambio.



Los puntos marcan el orden de los commits, pero las líneas marcan la relación entre ellos.

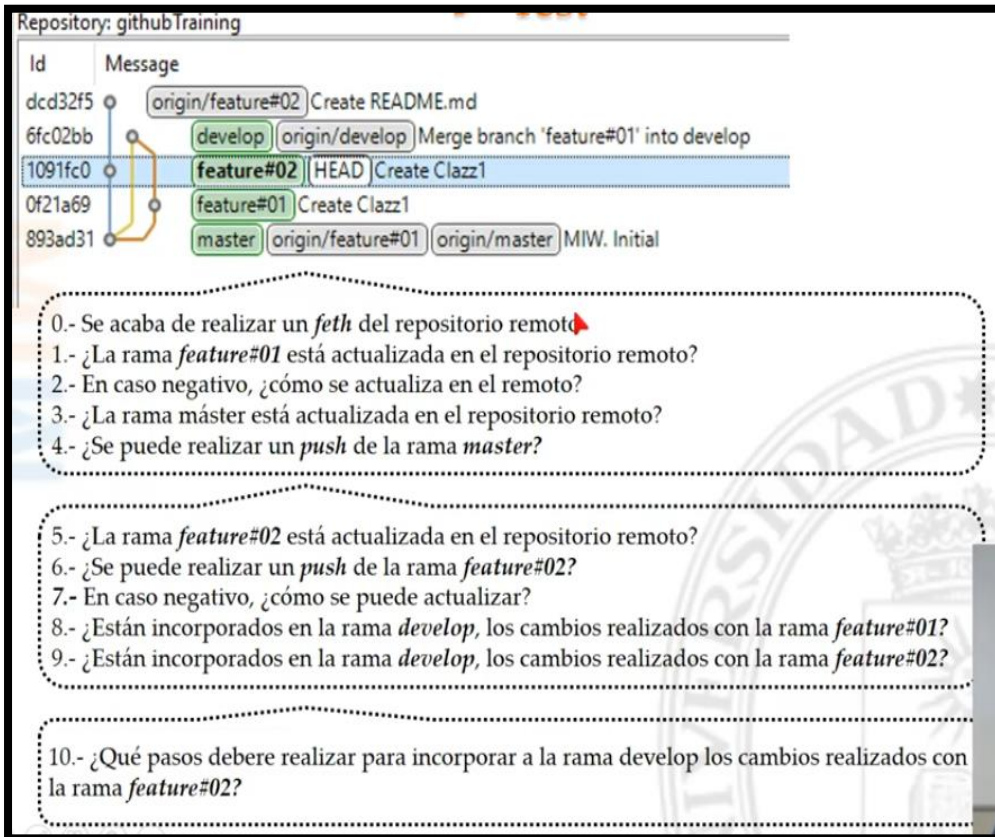
Siempre hay que estar actualizados para poder ver los cambios.

El tag (release) es muy útil para tener un .zip del código que no cambiará nunca. Si algo saliera mal más adelante, podríamos volver a un lugar seguro.

Si, ya que ambos apuntan al mismo commit.

No, ya que feature apunta a un commit y develop a otro.

No, ya que no apuntan al mismo commit. La remota es más actual. Aca necesitamos un merge local.



- 1- No, ya que no apuntan al mismo commit que *feature#02*.
- 2- Se realiza un *push*.
- 3- Si, porque apunta al mismo commit
- 4- Si
- 5- No, porque no apunta al mismo commit que *origin/feature#02*
- 6- No, porque la *feature#02* es más antigua que la remota de mejora dos y no se puede hacer un FF de arriba a abajo.
- 7- Con un merge de los cambios en *feature#02* remota con *feature#02* local.
- 8- Si
- 9- No
- 10- Para eso necesitamos merge.

- Video 4: Desarrollo de software colaborativo. Flujo de Trabajo.

Flujo de Trabajo (Workflow)

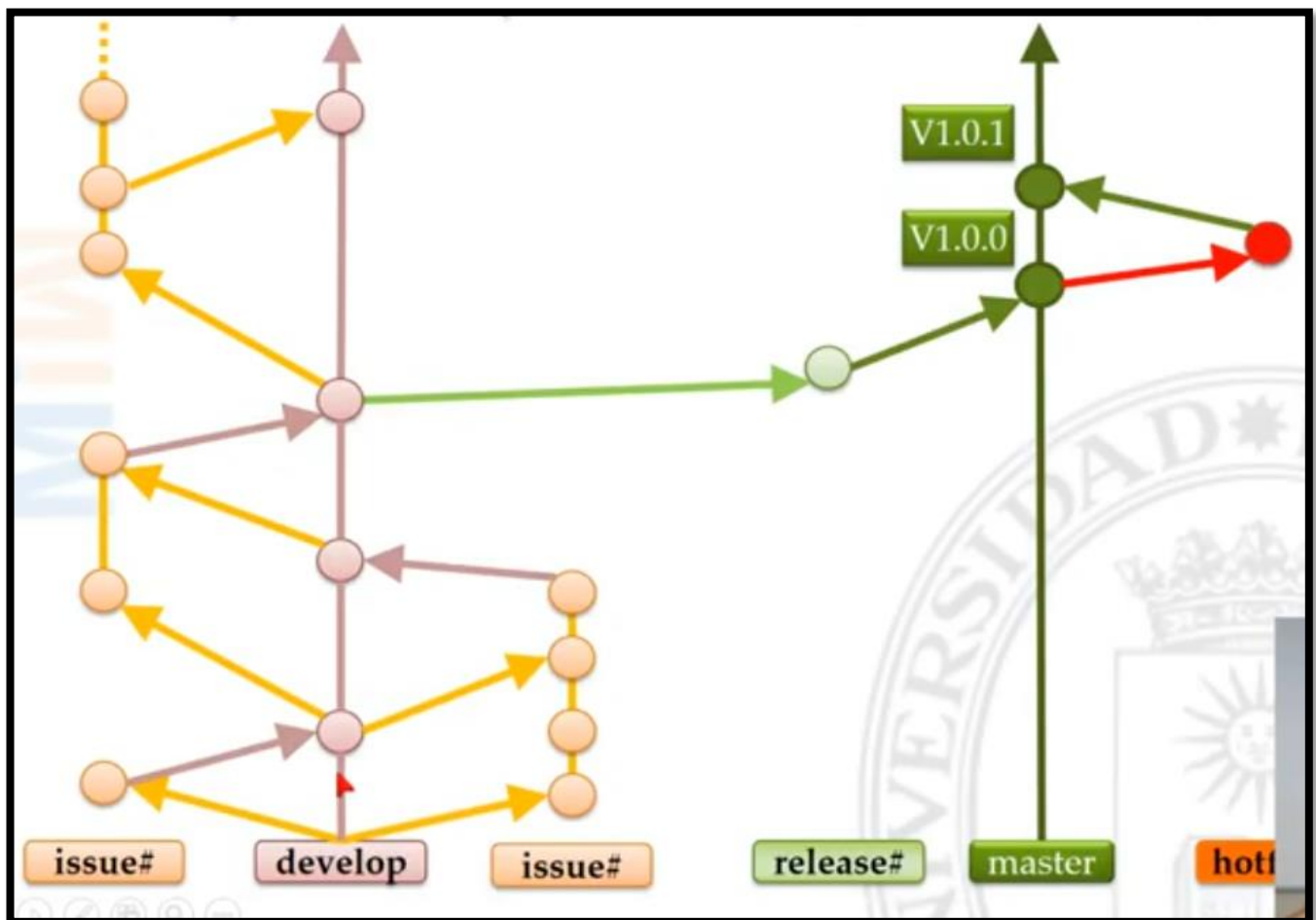
- El flujo de trabajo de un sistema de control de versiones indica cómo se relacionan los miembros del equipo para colaborar entre sí en el desarrollo del software colaborativo
 - **Flujo de trabajo centralizado (Centralized Workflow).** Todos comparten el código de un repositorio central, o en Git, todos comparten la única rama... Normalmente, se realiza una fusión al final del día, el último se come la fusión!!!
 - **Flujo de trabajo con bifurcación (Forking Workflow).** El usuario bifurca el proyecto realizando una copia completa del repositorio. Realiza las ampliaciones y realiza una petición de agregación (*pull request*). Se abre una discusión, y el dueño del repositorio decide la fusión. Normalmente, se utiliza en proyectos abiertos
 - **Flujo de trabajo ramificado (Git Workflow).** Gracias a los repositorios distribuidos, ha tomado auge por la gestión efectiva las ramas

Esta forma se termina abandonando porque genera muchos conflictos que nadie quiere

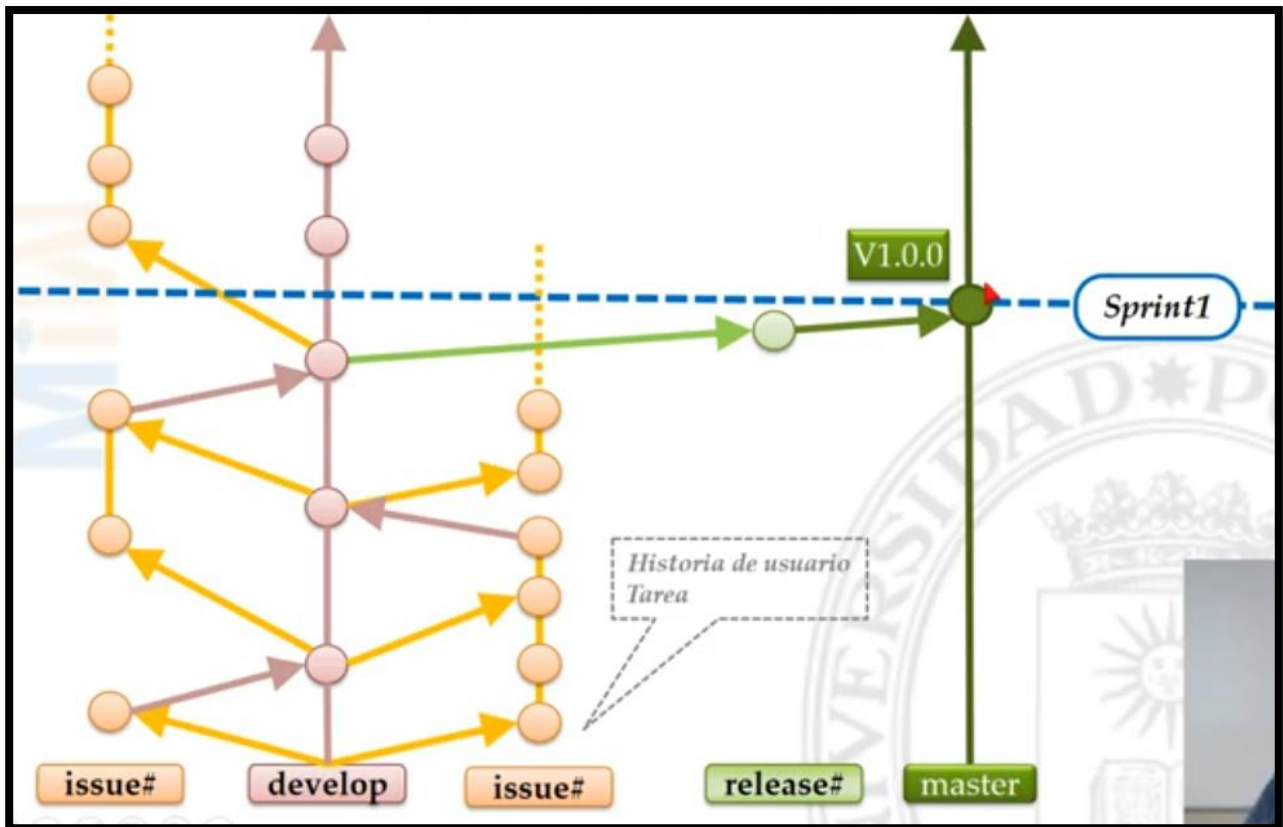
Vamos a hacer énfasis en el Flujo de trabajo ramificado (Git Workflow):

- Rama *master*: apunta al último *commit* de producción, es decir, a la última versión del código liberado
- Rama *develop*: es la rama de desarrollo, siempre estable, código que cumple los requisitos de calidad y con todos los test superados
- Ramas *issue#XX*, *feature#XX*, *topic#XX*: parte de *develop*, se añaden las nuevas características y vuelve a *develop* cuando vuelve a ser estable Rama exclusiva para hacer una mejora.
- Ramas *release#XX*: se utiliza para estabilizar un código para salir a producción. Una vez terminada, se fusiona con *master*
- Ramas *hotfix#XX*: se utilizan para corregir errores (*bugs*) el código en producción

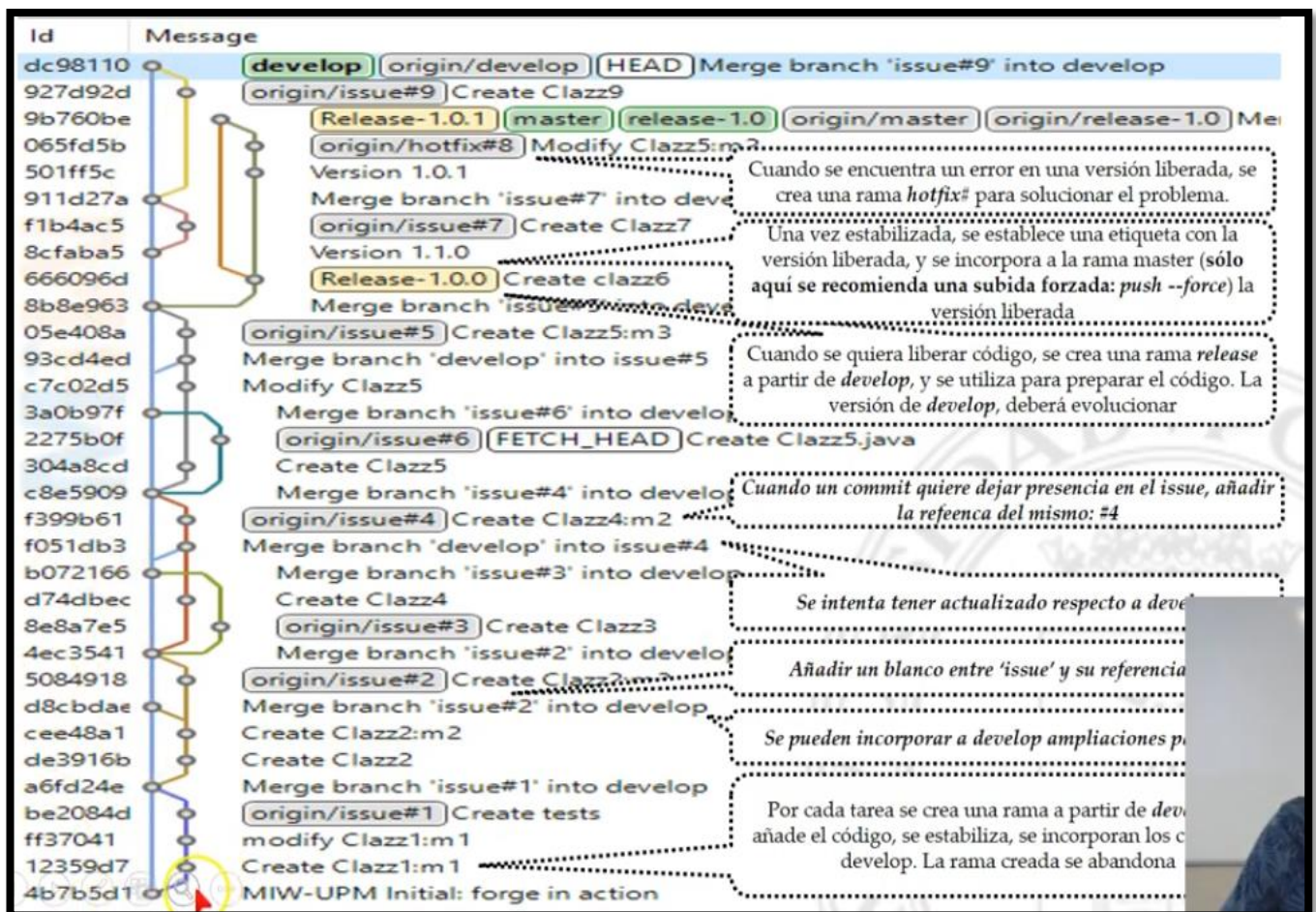
Veamos esto en forma gráfica:



Flujo de trabajo en metodologías ágiles:



Esto es un ejemplo de forja en acción:



Siempre que develop evolucione más que nuestra rama, incorporar el develop a nuestra rama, resolver los problemas, y después realizar un avance rápido y evolucionar.

- Video 5: Desarrollo de software colaborativo. GitHub. Wiki, Tickets \$ Proyectos.

Cunado necesitamos retocar un código, no vale la palabra solamente ya que esto se presta a confusiones. Para esto se utiliza una wiki. En ella va a quedar todo documentado.

- Una es un sitio web cuyas páginas pueden ser editadas directamente desde el navegador, donde los usuarios crean, modifican o eliminan contenidos compartidos
- GitHub dispone de Wiki para la generación de documentación rápida y compartida a partir de lenguajes de marcado ligeros
- Dispone de varios lenguajes de marcado... [Markdown](#)

Referencia de la página y capítulo

Todos los encabezados son referenciables

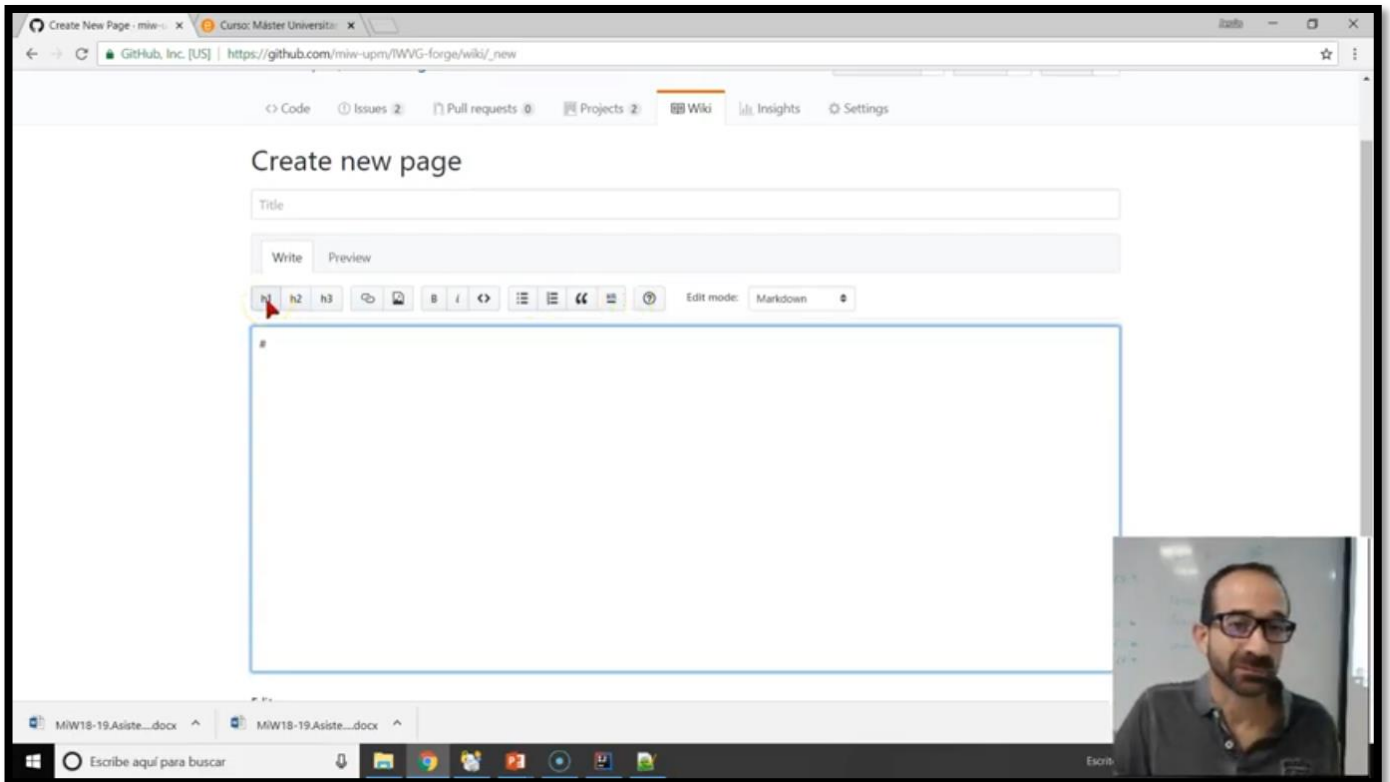
*NOTA: Lenguaje de marcado son caracteres especiales para hacer una página web con mucha rapidez.

Esto es un ejemplo de wiki en el GitHub

The screenshot shows the GitHub interface for the 'miw-upm / IWVG-forge' repository. The 'Wiki' tab is selected, displaying a page titled 'Markdown' with the subtitle 'Máster en Ingeniería Web edited this page on 13 Sep 2017 - 7 revisions'. The main content of the page is 'IWVG. Forja. wiki!' followed by a section 'Lenguaje Markdown' which is circled in yellow. Below this, there is a 'Varios' section with several lines of text and code examples. On the right side, there is a 'Pages' sidebar with a search bar and a list of links: Home, Login, Markdown, Maven template, Mi título, and Sonarcloud. At the bottom right, there is a small video feed of a man with glasses.

****NOTA:** El siguiente link no figura en el video. El siguiente link nos lleva a un documento con la sintaxis del lenguaje de marcado.
<https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/tutorial-de-markdown/>

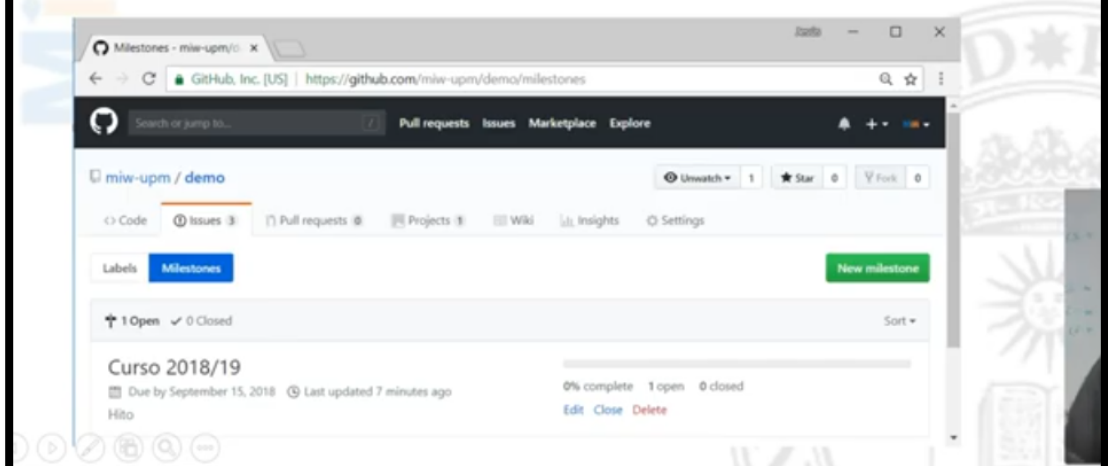
Ejemplo de creación de una nueva página con lenguaje de marcado en GitHub:



Para manejar las actividades en un flujo de trabajo, necesitamos dos cosas:

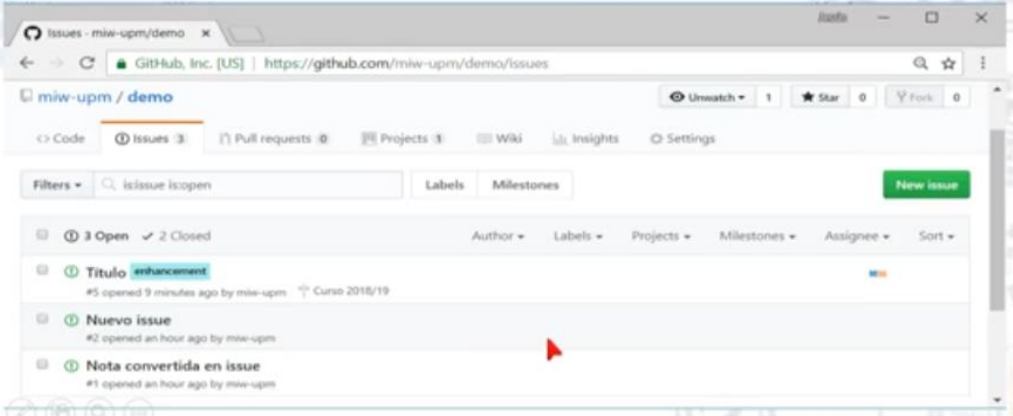
- **Hitos**

- **Milestone:** Hito (título, descripción y fecha). Representa una de fecha de referencia, normalmente asociada a un lanzamiento o finalización de un módulo. Puede estar abierto-cerrado
- Se les puede asociar tickets
- Tiene marcado un nivel de finalización, obtenido por los tickets asociados que se encuentran cerrados



- Tickets

- Ticket: tarea, error, issues... (título, asignado a..., asociado a un hito..., comentario, etiquetas) (abierto-cerrado)
 - Se abre un foro de comentarios
- Etiquetas: facilitan su identificación
 - Etiquetas de prioridad: High, Normal, Low
 - Etiquetas de tipo: Bugs, Enhancements...
- Ticket. Organización
 - Título: representa una referencia a una página de la wiki
 - Descripción muy básica, en la wiki, se detalla en profundidad

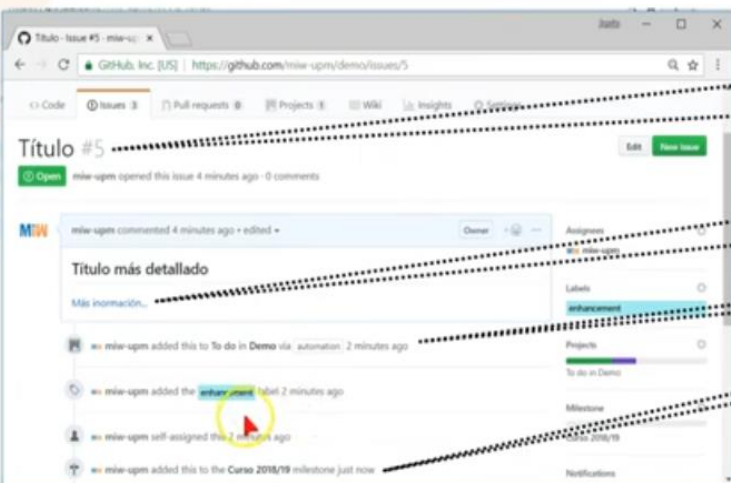


Resumen hasta el momento:

- Wiki: para documentar
- Hitos: para poner fecha límite
- Ticket: para gestionar las tareas de cada miembro del team

Organización de un ticket

- El ticket es una referencia a una página de la wiki
- Sólo tiene una breve descripción. En la wiki se establece toda la documentación



Título General, el identificador lo crea Github. La rama se llamará: 'issue#5'

Se referencia en la wiki una descripción detallada

Pertenece a un proyecto

Tiene un hito asociado

En el ticket, no se explica lo que se va a hacer. Se referencia a la wiki y es ahí donde se explica. No se hace en el ticket, porque cuando lo cerramos, este se elimina. En cambio, la wiki, no se elimina.

Historial del ticket:

Las acciones sobre el proyecto, quedan en el historial

Si en el mensaje del commit, aparece '#4', queda reflejado en el historial

En la fusión con 'develop' siempre es recomendado poner "merge 'issue #4' into develop", para que quede reflejado en el historial

Con todo lo anterior, vamos a gestionar el proyecto:

- El ticket es una referencia a una página de la wiki
- Sólo tiene una breve descripción. En la wiki se establece toda la documentación

Template: Automated kanban

Una nota se puede convertir en issue

Cuando se mueve de columna, aparece en el historial del issue

Cuando se cierra un issue se mueve automáticamente a la columna "Done"

Resumen de la gestión con GitHub:

1. Crear las notas con referencias a la wiki
2. Cuando una nota esta terminada y preparada para su realización, convertirla en ticket
3. Configuración del ticket: asignación, etiquetas, hito...
4. Cuando se inicia un ticket, se cambia de columna y se crea una rama
5. Cuando un commit debe ser reflejado en el historial del ticket, se añade en el mensaje '#xx'
6. Se desarrolla el ticket y se fusiona con *develop*:
 - "merge branch 'issue #xx' into develop"
7. Se cierra el ticket

****NOTA:** ¡Los tickets no se pueden borrar!

- Video 6: Desarrollo de software colaborativo. Integración Continua y Calidad del Código

Integración Continua

- La integración continua es una práctica de desarrollo de software, propuesto inicialmente por Martin Fowler, la cuál los miembros de un equipo integran su trabajo frecuentemente (diariamente) y se revisa automáticamente
- Filosofía muy relacionada con las metodologías ágiles y la programación extrema (XP)
- Cada integración se verifica mediante una herramienta de construcción para detectar los errores de integración tan pronto como sea posible
- Para ello se pueden utilizar un software especializado en automatizar tareas y que estas se ejecuten de forma automática. Las tareas a realizar pueden ser:
 - Compilación de los componentes
 - Obtener métricas de calidad de código
 - Ejecución de pruebas unitarias
 - Ejecución de pruebas de integración, de aceptación
- Buenos principios
 - Contribuir a menudo
 - No contribuir con código roto
 - Soluciona los builds rotos inmediatamente
 - Escribe tests automáticos
 - Todos los tests deben pasar

Es importantísimo contribuir al develop cada 4 horas aproximadamente. 2 commits al día: el último puede ser en la rama local y al día siguiente lo fusionamos con el develop.

Procedimiento de Integración Continua

- El equipo modifica el código, y una vez testeado, se realiza un *commit + push* al repositorio
- La herramienta de CI monitoriza el repositorio y se dispara cada vez que detecta un cambio
- CI ejecuta la construcción del todo el código fuente, aplicando los tests de control de calidad, pruebas unitarias, pruebas de integración...
- CI envía correos de los resultados de la integración del nuevo código

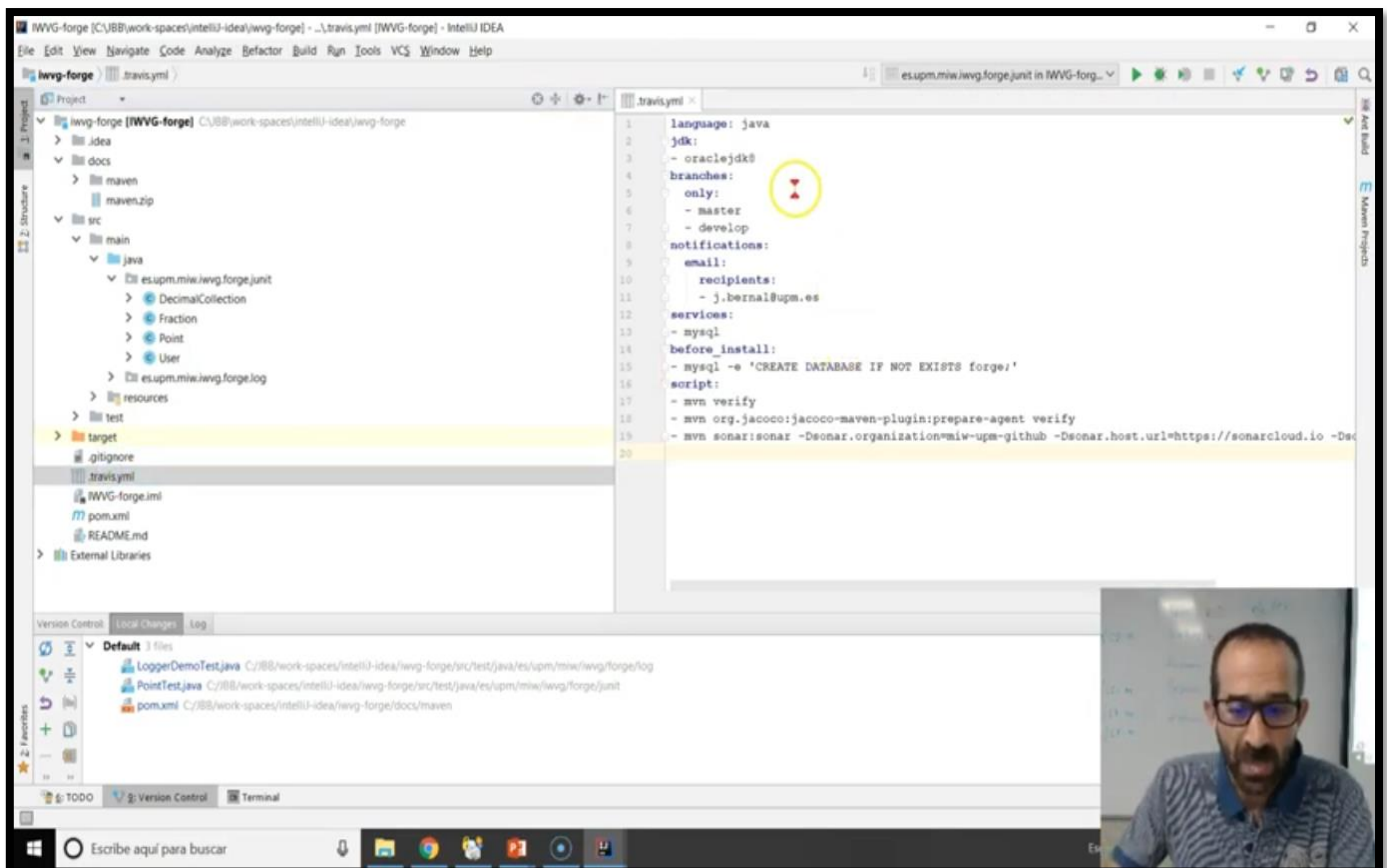
Herramientas de Integración Continua

- Travis CI
 - <https://travis-ci.org/>
 - Documentación: <http://about.travis-ci.org/docs/>
 - Travis CI es un sistema de integración continua gratuito en la nube para la comunidad OpenSource. Está integrado con GitHub y ofrece soporte para: Java, Groovy, Haskell, Node.js, PHP, Python, Ruby...
- Jenkins
 - <http://jenkins-ci.org/>
 - Jenkins es un software de integración continua de código abierto escrito en Java, evolución de Hudson
 - Jenkins tiene soporte para sistemas de control de versiones, alguno como SVN, CVS, Git y corre en un servidor de aplicaciones como ejemplo Tomcat o Jboss permitiendo la ejecución de proyectos A Maven...

IntelliJ, GitHub y Travis

- Crear un proyecto Maven en IntelliJ y conectarlo a un repositorio de GitHub
- Añadir una Clase y el correspondiente test. Probarlo
- Añadir un archivo [.travis.yml](#) al raíz del proyecto
- Para empezar a Travis CI, acceder a través de GitHub OAuth. Ir a Travis CI y seguir el vínculo de iniciar sesión, situado en la parte superior: *Sign in GitHub*
- Activar en GitHub el Servicio Hook#, se realiza desde la Web de Travis-CI. En la parte superior derecha, sobre nuestra cuenta, elegir la opción *Accounts*, activar el interruptor para cada repositorio que desea conectar en Travis CI
- A partir de ahora, el repositorio estará monitorizado por Travis CI. Cada vez que se realice un commit, ejecuta uno de los siguientes constructores en sus servidores, esta operación puede ir despacio (varios minutos)

Configuración de Travis:



Conceptos extras:

Máquina virtual

- es un software que simula un sistema de computación y puede ejecutar programas como si fuese una computadora real.

Docker:

- es un proyecto de código abierto que automatiza el despliegue de aplicaciones dentro de contenedores de software, proporcionando una capa adicional de abstracción y automatización de virtualización de aplicaciones en múltiples sistemas operativos.

Herramientas de Calidad de Código

- Sonarqube es una plataforma abierta para gestionar la calidad del código
 - <http://www.sonarqube.org/>
- Se ofrece servicio en la nube
 - <https://sonarcloud.io/>
- Abarca las siguientes características:
 - Estándares de Codificación
 - Bugs y errores potenciales
 - Duplicaciones de código
 - Pruebas Unitarias
 - Cobertura de pruebas
 - Complejidad ciclomática
 - Control de comentarios
- Ofrece soporte para más de 20 lenguajes: Java...

Sonard Cloud

- Servicio on-line de sonarqube
- Para crear un nuevo proyecto
 - >>>Help > Tutorials > Analyze a new Project
- Para generar una clave de acceso:
 - <https://sonarcloud.io/account/security/>
- Seguridad: la clave de acceso es mejor definirla como variable en Travis-ci
 - En Travis-ci, en el Proyecto, ir a *Settings (More options)*, y en el apartado *Environment Variables*, definir una variable con la clave, por ejemplo, **SONAR**
 - Para referenciarla: **\$SONAR**
- Ejecutar los comandos en el proyecto
 - `call mvn clean org.jacoco:jacoco-maven-plugin:prepare-agent install`
 - `call mvn sonar:sonar`
 - Dsonar.organization=miw-upm-github
 - Dsonar.host.url=https://sonarcloud.io
 - Dsonar.login=\$SONAR

La forma de esconder la contraseña en sonar es mediante un token en Travis

The screenshot displays the Travis CI settings interface for a repository. On the left, a sidebar lists recent builds with their status (success or failure), duration, and completion time. The main content area is divided into sections: 'Auto Cancellation', 'Environment Variables', and 'Cron Jobs'. The 'Environment Variables' section is highlighted, showing two variables: 'SONAR' and 'varEntorno', both masked with asterisks. A note below the variables states: 'Please make sure your secret key is never related to the repository, branch name, or any other guessable string. For more tips on generating keys read our documentation.' Below this, there is a table with columns for 'Name' and 'Value', and a toggle for 'Display value in build log'. The 'Cron Jobs' section at the bottom shows a job configured for the 'develop' branch, running monthly, with the option 'Always run' selected. A small video inset in the bottom right corner shows a man speaking.

Issues · miw-upm/team-8 · Travis CI GmbH [DE] | <https://travis-ci.org/miw-upm/IWVG-forge/settings>

Duration: 24 sec
Finished: 3 months ago

✓ miw-upm/APAW-themes-lay # 34
Duration: 35 sec
Finished: 3 months ago

✓ miw-upm/betca-tpv-spring # 292
Duration: 3 min 26 sec
Finished: 4 months ago

✓ miw-upm/BETCA-spring4 # 29
Duration: 2 min 39 sec
Finished: 4 months ago

✗ miw-upm/SPRING.tpv # 274
Duration: 1 min 16 sec
Finished: 11 months ago

✓ miw-upm/APAW-api-theme # 48
Duration: 1 min 32 sec
Finished: 12 months ago

✓ miw-upm/APAW-pd # 26
Duration: 1 min 4 sec
Finished: 12 months ago

Auto Cancellation

Auto Cancellation allows you to only run builds for the latest commits in the queue. This setting can be applied to builds for Branch builds and Pull Request builds separately. Builds will only be canceled if they are waiting to run, allowing for any running jobs to finish.

☐ Auto cancel branch builds ☐ Auto cancel pull request builds

Environment Variables

Notice that the values are not escaped when your builds are executed. Special characters (for bash) should be escaped accordingly.

SONAR *****

varEntorno *****

Please make sure your secret key is never related to the repository, branch name, or any other guessable string. For more tips on generating keys read our documentation.

Name	Value	Display value in build log
		<input type="checkbox"/>

Add

Cron Jobs

Branch: develop Interval: monthly Options: Always run