
# Práctica 1:

# Gestión del proyecto software

**Ingeniería del software**

**2º grado en ingeniería informática**

Curso 2020/2021

Aurora Ramírez Quesada

Departamento de informática y análisis numérico

# Índice

**1. Organización de la práctica** **2**

**2. Control de versiones con *Git* y *GitHub*** **4**

2.1. Introducción a los sistemas de control de versiones . . . . . 4

2.2. Conceptos básicos . . . . . 5

2.3. Gestión de ramas . . . . . 7

2.4. Instalación y configuración . . . . . 9

2.5. Comandos más útiles . . . . . 10

2.6. Repositorios colaborativos en GitHub . . . . . 11

2.7. Lenguaje *Markdown* . . . . . 11

**3. Gestión del proyecto con *Scrum* y *YouTrack*** **13**

3.1. Introducción a la metodología *Scrum* . . . . . 13

3.2. Roles, artefactos y eventos . . . . . 13

3.3. Plataforma *YouTrack* . . . . . 16

# 1. Organización de la práctica

Este documento contiene la información necesaria para realizar la primera práctica de la asignatura. Esta práctica tiene una duración de dos semanas, en las que se explicarán los conceptos básicos relacionados con la gestión del proyecto software que se va a desarrollar durante el cuatrimestre. A continuación se detallan los objetivos y la evaluación de la práctica, dividida en dos sesiones:

**Semana 1: control de versiones con *Git* y *GitHub***

- Fecha: 21/22/24 (según grupo) de septiembre de 2020.
- Objetivos:
  1. Conocer los fundamentos de los sistemas de control de versiones.
  2. Crear y manejar repositorios colaborativos de código fuente.
  3. Aprender los elementos básicos del lenguaje *Markdown*.
- Preparación: Los estudiantes deben haber instalado *Git* previamente[1] (ver apartado 2.4). También se recomienda que tengan creada una cuenta de usuario en *GitHub* (`https://github.com/`) antes del comienzo de la clase.
- Seguimiento y evaluación: esta práctica se explica como un taller de iniciación a *Git*, de forma que los estudiantes deben ir realizando los ejemplos propuestos. La práctica no tiene evaluación, y la entrega en Moodle consiste únicamente en indicar la dirección del repositorio *GitHub* creado durante el taller. Dicha entrega servirá para comprobar que el estudiante ha completado la actividad de forma síncrona durante el horario de clase.

**Semana 2: gestión del proyecto con *Scrum* y *YouTrack***

- Fecha: 28/29/01 de septiembre/octubre de 2020.
- Objetivos:
  1. Conocer los fundamentos básicos de la metodología ágil *Scrum*.
  2. Comprender la importancia de la gestión de tareas en un proyecto software.
  3. Aprender a manejar la aplicación *YouTrack*.

[1] `https://git-scm.com/downloads`

- Preparación: los estudiantes deben haber formado los grupos de trabajo (tres o cuatro personas), tras lo cual recibirán acceso a la aplicación *YouTrack* por parte de la profesora.
- Seguimiento y evaluación: tras explicar los conceptos teóricos básicos, se habilitará un proyecto en *YouTrack* para cada uno de los grupos de trabajo. Durante el resto de la sesión, se explicará el funcionamiento de la aplicación y los estudiantes irán probando sus funciones para familiarizarse con el entorno. La práctica no tiene evaluación, pero se comprobará que han realizado ejemplos sobre la plataforma. Una guía rápida del funcionamiento de *YouTrack* se encuentra disponible en Moodle.

# 2. Control de versiones con *Git* y *GitHub*

## 2.1. Introducción a los sistemas de control de versiones

Los sistemas de control de versiones (VCS, *Version Control System*) permiten el control y gestión de los cambios que experimenta un producto durante su desarrollo. Los VCS se han convertido en una herramienta esencial para los programadores, facilitando el manejo de versiones del software al registrar toda la actividad (creación, modificación, eliminación) de los artefactos que lo componen. Además, los VCS cuentan con mecanismos para recuperar copias y trabajar de forma colaborativa, siendo por tanto una pieza fundamental en el desarrollo profesional de software.

Por lo general, un VCS consta de un ***repositorio*** donde se almacenan los archivos que se encuentran bajo el control de versiones. Uno o más ***clientes*** se conectan al repositorio para leer o escribir datos en él. Se distinguen dos tipos de VCS en función de su arquitectura [1]:

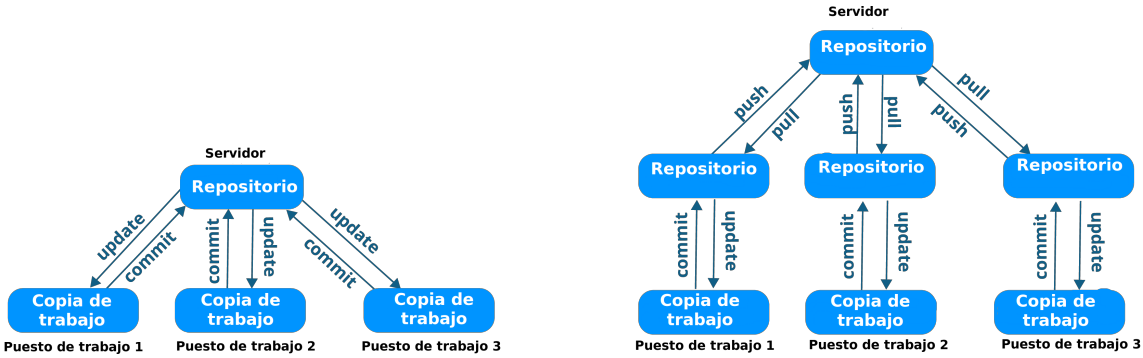
1. VCS centralizado. Existe un único repositorio central que almacena los archivos y registra los cambios realizados por los clientes. Es un modelo en el que todos los clientes tienen la visión completa del sistema. Su principal inconveniente es que el repositorio es un punto crítico ante posibles fallos. Uno de los VCS más conocidos dentro de este grupo es *Subversion* [2].
2. VCS distribuido. Existe un repositorio remoto del que cada cliente tiene una copia completa, lo que se conoce como *mirror repository*. Ante un fallo, el repositorio puede restaurarse a partir del de un cliente, pero mantener la integridad de los datos es más complejo. *Git* es uno de los VCS distribuidos más popular.

La figura 1 muestra un esquema de ambos modelos. A continuación se definen otros términos generales [2]:

**Consignar** (*commit*) Es la acción de publicar uno o más cambios en el repositorio mediante una transacción atómica. Es decir, toda la operación es una unidad que se completa de una vez o, si se produce un fallo, se revoca en su totalidad.

**Revisión** (*revision*) Estado que se crea cada vez que se realiza un *commit*. A cada revisión se le asigna un identificador único.

**Conflicto** (*conflict*) Situación que ocurre cuando varios clientes realizan cambios en sus copias locales que se solapan al intentar publicarlos en el repositorio.




(a) VCS central (b) VCS distribuido

Figura 1: Dos arquitecturas para un sistema de control de versiones. Fuente: `https://www.opentix.es/sistema-de-control-de-versiones/`

**Rama** (*branch*) Una línea de desarrollo que se crea a partir de una revisión concreta y que evoluciona de forma independiente.

En el resto de secciones se explica en detalle las particularidades de *Git*, el VCS distribuido que se utilizará en las prácticas de la asignatura.

## 2.2. Conceptos básicos

*Git* es un VCS distribuido creado en 2005 que se caracteriza por su rapidez y eficiencia para gestionar proyectos, así como por su potente sistema de ramificación [1]. Su funcionamiento se basa en el concepto de ***instantánea*** (*snapshot*). Cada vez que se realice un *commit*, el sistema evalúa el estado de cada archivo y guarda esa información, de forma que los archivos no modificados no se guardan de nuevo. La figura 2 ilustra esta idea:

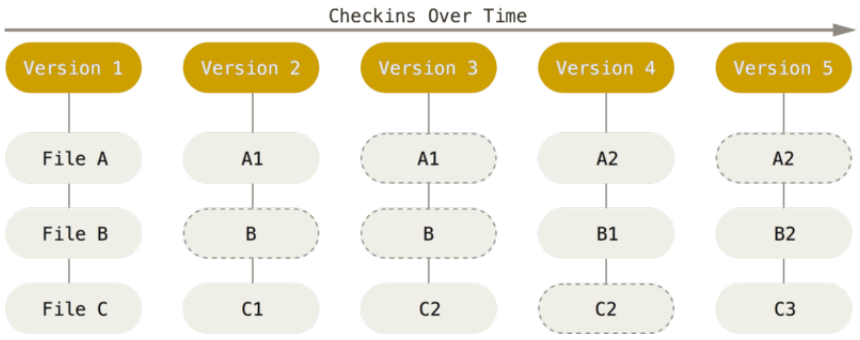



Figura 2: Funcionamiento de las *snapshots* en *Git*. Fuente: [1]

En *Git*, un archivo se encuentra en uno de los tres estados siguientes:

1. Confirmado (*committed*). Los datos están actualizados en la copia central del repositorio.
2. Modificado (*modified*). El archivo ha sufrido cambios que aún no han sido confirmados.
3. Preparado (*staged*). El archivo modificado ha sido "marcado" para su próxima confirmación.

Todos los archivos confirmados están alojados en el repositorio central, una base de datos que puede ser local o remota, cuya metainformación se almacena en un archivo con extensión `.git`. El directorio de trabajo (*working directory*) es la copia local del repositorio central, a la que se accede por el sistema de archivos del ordenador. Finalmente, el área de preparación (a veces llamado *index*) es donde se registran los cambios que van a confirmarse. La relación entre las tres áreas se muestra en la figura 3.

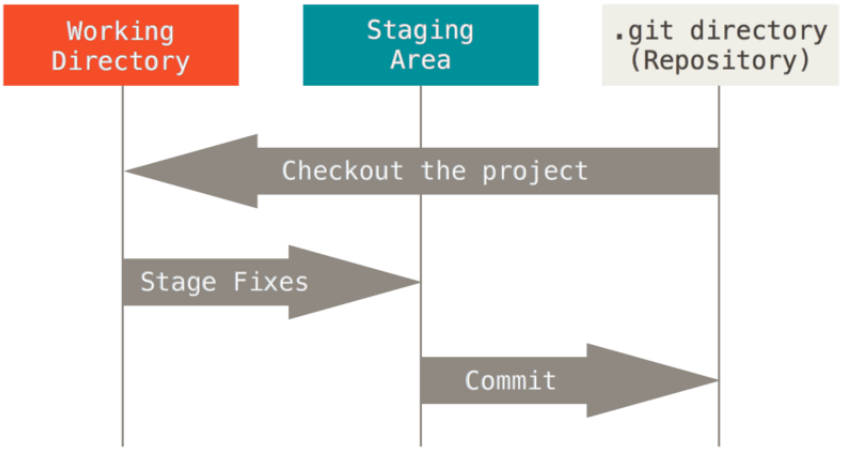



Figura 3: Flujo de trabajo en *Git*. Fuente: [1]

Por tanto, el flujo básico en *Git* consta de los siguientes pasos:

1. Actualizar la copia local del repositorio.
2. Realizar los cambios en los archivos que corresponda.
3. Registrar los cambios en el repositorio local (hacer `commit`).

La unidad de información fundamental es el ***commit***, que se compone de los siguientes elementos (ver figura 4):

- Un identificador único, creado automáticamente por una función *hash* (SHA-1) en base a su contenido.

- Referencia al "árbol" de archivos en su estado actual (visión del *snapshot*).
- Referencia al commit anterior (padre), si lo hay.
- Mensaje descriptivo, fecha y autor.

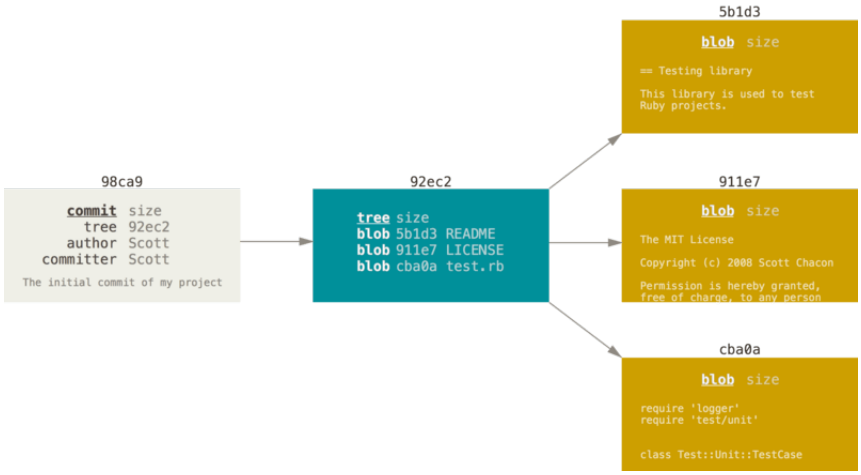



Figura 4: Estructura de un *commit* y su "árbol". Fuente: [1]

## 2.3. Gestión de ramas

El sistema de ramificación es una funcionalidad habitual de los VCS modernos, y especialmente relevante en *Git*. La idea es mantener líneas independientes de desarrollo donde se alojen cambios que no queremos que afecten a la rama principal (conocida como *master*). Por ejemplo, el código en producción (entregado al cliente) reside en la rama *master*, mientras que el desarrollo de nuevas funcionalidades se realiza en una rama aparte. Una vez la nueva funcionalidad haya sido probada, puede integrarse en la rama *master*, tal y como se representa en la figura 5.

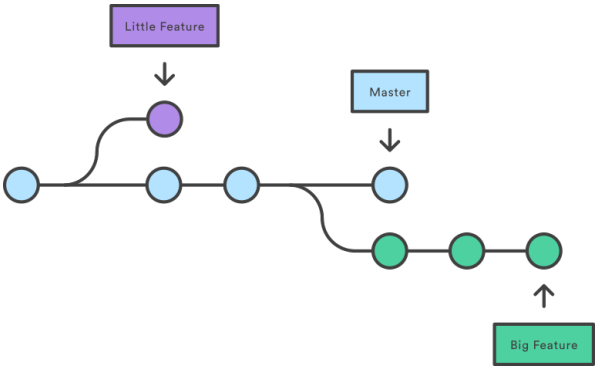
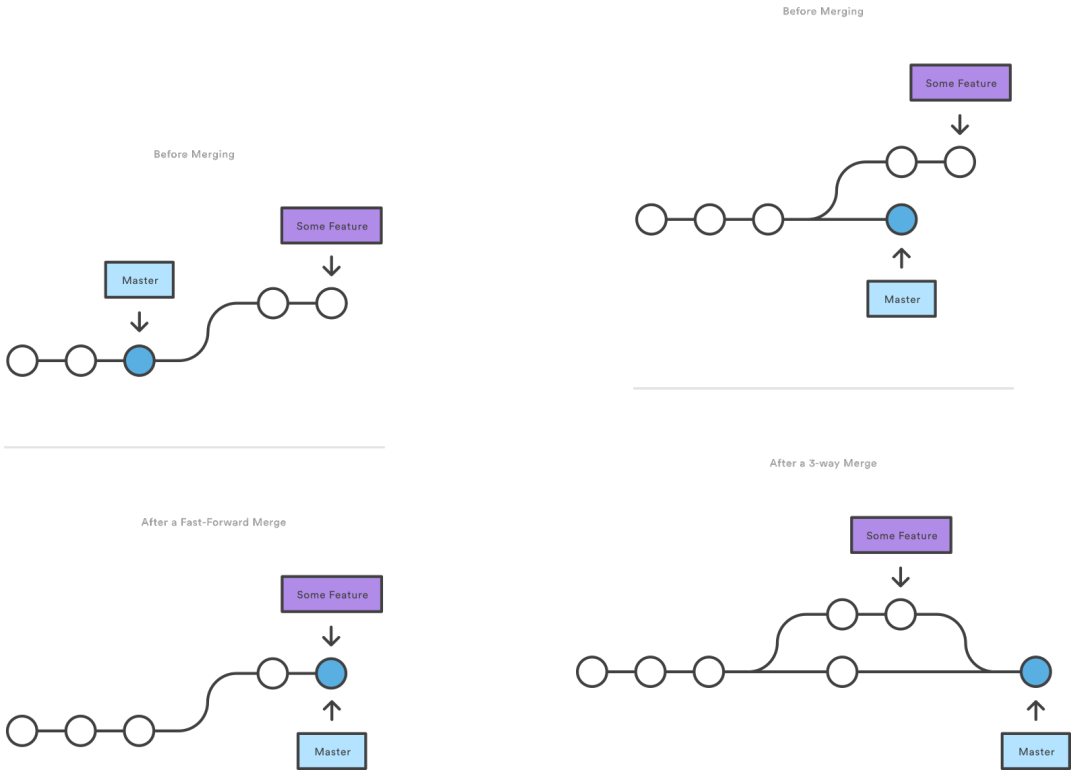



Figura 5: Ramas en un VCS. Fuente: [3]

A diferencia de otros sistemas VCS, las ramas en *Git* no implican copiar archivos entre directorios, ya que es un modelo más abstracto. Cada rama es en realidad una referencia a una secuencia de *commits*. *Git* nos permite movernos entre ramas, de forma que los *commits* que realicemos se enlazarán a la rama activa. Para saber cuál es la rama activa, *Git* mantiene un "apuntador" denominado *HEAD*.




(a) Fusión con avance rápido (b) Fusión de tres vías

Figura 6: Estrategias de fusión en *Git*. Fuente: [3]

La operación de unificar dos ramas se denomina *merge*. El proceso consiste en identificar un punto de confirmación común entre las dos ramas. En ese punto, se crea una "confirmación de fusión" (*merge commit*). Este *commit* tendrá dos "padres" y al confirmarlo, *Git* intentará mezclar las ramas automáticamente. Existen dos estrategias para realizar la fusión, tal y como se muestra en la figura 6:

- Fusión con avance rápido (*fast forward merge*). Ocurre cuando podemos recorrer el historial de revisiones de forma lineal entre ambas ramas. Por ejemplo, hemos creado una rama *some feature* a partir de la *master*, pero la rama *master* no ha sufrido cambios después. *Git* simplemente añade los *commits* de la rama *some feature* a la rama *master*. La creación del punto de fusión es opcional. Este estrategia es adecuada para cambios pequeños o corrección de errores.
- Fusión de tres vías (*3-way merge*). Si la rama *master* ha sufrido modificaciones, hacen faltan tres confirmaciones (en los dos extremos y en el predecesor común). Se

creará explícitamente un punto de fusión a continuación de los dos extremos. Esta estrategia es más habitual cuando se integran funcionalidades de mayor complejidad o que implican mayor tiempo de desarrollo.

Durante el proceso *3-way merge*, *Git* puede encontrarse con un archivo que ha sido modificado en ambas ramas, por lo que la unión no podrá realizarse automáticamente. Se ha producido un "conflicto" que requiere la intervención del usuario para resolverlo manualmente. *Git* marca la zona de conflicto en el archivo correspondiente, tal y como se muestra en la figura 7. El punto $A$ identifica el comienzo del cambio considerando el archivo que se encuentra en la rama activa donde se iba a hacer la fusión (*HEAD*). El punto $B$ representa el final del cambio en la rama actual y el comienzo del cambio en la rama bifurcada. El punto $C$ representa la línea donde acaba el cambio en la rama bifurcada. El usuario deberá modificar el archivo para mantener uno de los dos cambios y confirmarlo mediante un nuevo *commit*.

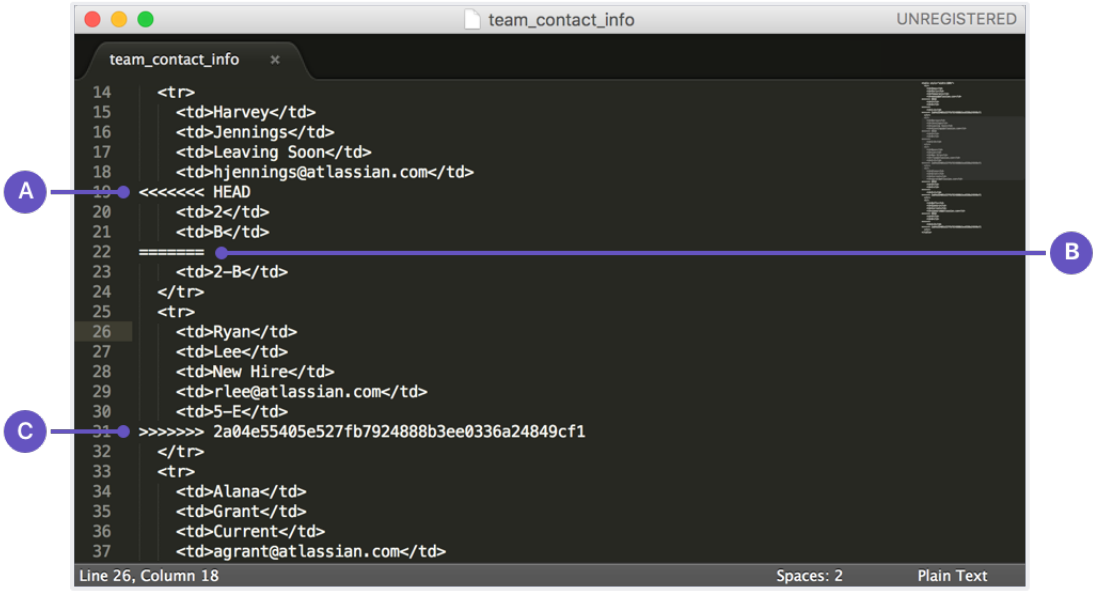



Figura 7: Visualización de un conflicto en *Git*. Fuente: `https://support.atlassian.com/bitbucket-cloud/docs/resolve-merge-conflicts/`

## 2.4. Instalación y configuración

Podemos instalar *Git* para nuestro sistema operativo accediendo a la siguiente URL: `https://git-scm.com/`. Disponemos además de un gran número de recursos para inicializarnos en su uso y aprender aspectos avanzados:

- Documentación oficial (incluye recursos externos como libros, tutoriales, etc.): `https://git-scm.com/doc`

- Tutoriales en vídeo de *GitHub* (sistema de alojamiento que utilizaremos en clase): `https://www.youtube.com/githubguides`
- Tutoriales de *Atlassian*: `https://www.atlassian.com/es/git/tutorials`

Durante el taller se explicará el uso de *Git* a través de la línea de comandos. En función del sistema operativo con el que trabaje el estudiante, existen diferentes clientes gráficos que facilitan la preparación y control de los *commits*[2]. Además, los entornos de desarrollo integrado como *Eclipse* (que se utilizará más adelante), disponen de *plugings* para utilizar *Git* sin salirnos del entorno de programación[3].

La tabla 2 detalla los comandos *Git* para su configuración básica.

| **Comando** | **Descripción** |
|---|---|
| `git config --global user.name <name>` | Establecer nuestro nombre de usuario |
| `git config --global user.email <email>` | Establecer nuestra dirección de correo |
| `git config --system core.editor <editor>` | Establecer el editor de texto |
| `git config --global color.ui true` | Habilitar el uso de color |
| `git config --list` | Ver la configuración |

Tabla 1: Comandos de configuración en Git

## 2.5. Comandos más útiles

| **Comando** | **Descripción** |
|---|---|
| `git init <nombre>` | Crear un repositorio |
| `git clone <url>` | Descargar un repositorio |
| `git status` | Lista de archivos nuevos o modificados |
| `git diff` | Muestra diferencias entre archivos o ramas |
| `git add <archivo>` | Añadir un archivo al área de preparación |
| `git commit - m <mensaje>` | Crear un *commit* |
| `git log` | Ver historial de la rama actual |
| `git branch <nombre>` | Crea una nueva rama |
| `git checkout <rama>` | Cambia a la rama indicada |
| `git merge <rama>` | Combina la rama actual y la indicada |
| `git fetch <nombre>` | Descarga cambios del repositorio remoto |
| `git pull` | Descarga e integra los cambios del repositorio remoto |
| `git push <rama>` | Sincroniza los *commits* en el repositorio remoto |

Tabla 2: Comandos más útiles en *Git*

[2] `https://git-scm.com/downloads/guis`
[3] `https://www.eclipse.org/egit/`

La tabla 1 recopila los comandos más habituales que se utilizarán en clase. El listado completo de comandos puede consultarse en: `https://git-scm.com/docs/git#_git_commands`. Para una referencia rápida, las *cheatsheet* de *GitHub*[4] (disponible en español) y *Atlassian*[5] son un buen resumen de los comandos más frecuentes.

## 2.6. Repositorios colaborativos en GitHub

*GitHub* es una aplicación web que da soporte a la creación de repositorios utilizando *Git*, permitiendo a varios usuarios trabajar colaborativamente en proyectos. Además, proporciona funcionalidades adicionales para registrar errores (*issue tracker*), alojar documentación (web, wikis, etc.) e interaccionar con usuarios de nuestros productos. Cada estudiante deberá registrarse en la plataforma para realizar la segunda parte del taller y alojar el código y documentación que se generen durante el resto de las prácticas.

En el taller se explica cómo crear un repositorio remoto y cómo configurarlo para dar acceso a colaboradores. Una vez creado, podemos "clonarlo" en nuestro equipo para trabajar de forma local, o utilizar su interfaz para realizar algunas acciones como subir archivos y crear documentación.

## 2.7. Lenguaje *Markdown*

*Markdown* es un lenguaje de texto ligero surgido en 2004 como método de conversión entre texto plano y HTML. Tiene un sintaxis sencilla que permite dar formato a contenido web, incluyendo la creación de listas, tablas, etc. Este lenguaje está soportado por una gran cantidad de aplicaciones, entre ellas *GitHub*, convirtiéndose en una forma rápida de generar documentación. Los archivos escritos en *Markdown* tienen extensión `.md` o `.markdown`. Como ejemplo, al crear un repositorio en *GitHub* se nos permite crear un archivo `README.md` que mostrará la información básica de nuestro proyecto.

La tabla 3 detalla los comandos básicos para dar formato al texto. Para conocer más detalles del lenguaje, se recomiendan los siguientes recursos:

- Guía en español: `https://markdown.es/`
- Sintaxis y ejemplos (en inglés):
  - `https://guides.github.com/features/mastering-markdown/`
  - `https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet`

[4] `https://github.github.com/training-kit/`
[5] `https://www.atlassian.com/es/git/tutorials/atlassian-git-cheatsheet`

| **Formato** | **Sintaxis** |
|---|---|
| Encabezados | `# Primer nivel`<br>`## Segundo nivel`<br>`### Tercer nivel` |
| Negrita | `**Texto**` |
| Cursiva | `*Texto*` |
| Lista numerada | `1. Primer elemento`<br>`2. Segundo elemento` |
| Lista no numerada | `* Primer elemento`<br>`* Segundo elemento` |
| Línea de código | `‘Línea de código‘` |
| Bloque de código | `‘‘‘Bloque de código‘‘‘` |
| Enlace | `[Descripción](URL)` |
| Imagen | `![Descripción](URL o ruta)` |

Tabla 3: Sintaxis del lenguaje *Markdown*

# 3. Gestión del proyecto con *Scrum* y *YouTrack*

## 3.1. Introducción a la metodología *Scrum*

Para el desarrollo del proyecto de prácticas se seguirá una metodología ágil basada en *Scrum* [4]. *Scrum* es un marco de trabajo apropiado para el desarrollo y mantenimiento de un producto (no solo software), donde el eje principal es la entrega del producto con el máximo valor posible, siempre buscando la eficacia de las prácticas de gestión y desarrollo [4]. Para ello se emplea un modelo iterativo e incremental centrado en los tres pilares fundamentales de un método empírico:

1. Transparencia. Todos los participantes deben tener un lenguaje común que les permita entender el proceso.
2. Inspección. La metodología se fundamenta en la revisión continua de los artefactos y del progreso hacia el objetivo final. Estas revisiones deben detectar posibles desvíos sin obstaculizar el trabajo diario.
3. Adaptación. Se deben determinar las causas de los desvíos y ajustar el proceso para minimizarlos y alcanzar un producto satisfactorio para el cliente.

*Scrum* está especialmente diseñado para el trabajo colaborativo en equipos, poniendo en relieve la importancia de organizarse de forma autónoma y aprender de la propia experiencia a medida que se avanza en el problema [5]. El compromiso, coraje, foco, apertura y respeto son los valores que *Scrum* fomenta en sus equipos. Para adoptar la metodología *Scrum* es necesario conocer sus tres componentes principales (**roles**, **artefactos** y **eventos**), así como las reglas que rigen las relaciones e interacciones entre ellos. No obstante, la metodología deja abierta la elección de las estrategias concretas con las que poner en práctica estos conceptos. La figura 8 muestra una visión global de la metodología *Scrum* y sus componentes. El elemento central de *Scrum* es la iteración o *sprint*. Un *sprint* no es más que un período breve y de tiempo fijo durante el cual el equipo debe completar una parte del producto [5]. Dividiendo el trabajo en *sprints*, se busca facilitar su gestión y favorecer la adaptación a cambios. Al final de cada *sprint* se debe estar en disposición de entregar un producto que funciona, aunque no esté aún completo.

## 3.2. Roles, artefactos y eventos

La figura 9 resume los roles, artefactos y reuniones propuestos en *Scrum*. A continuación se describe cada uno de ellos [4].

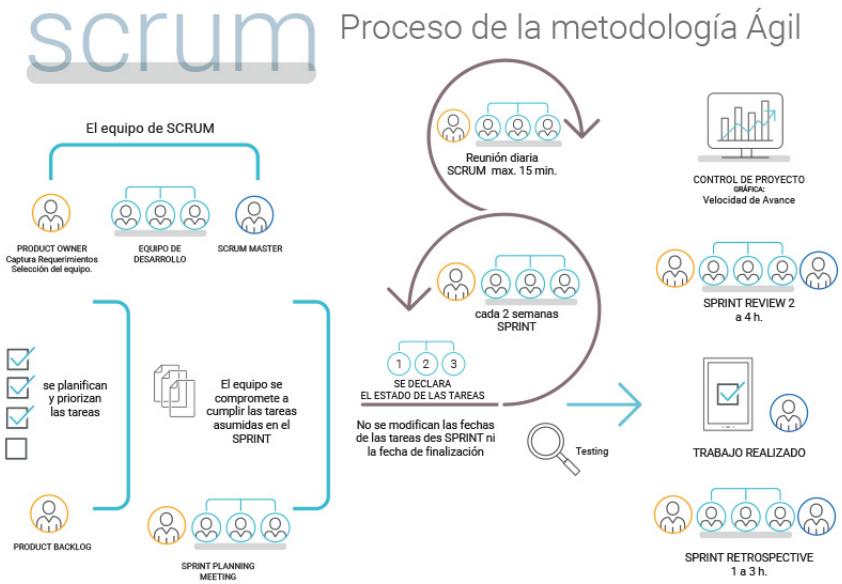



Figura 8: Metodología *Scrum*. Fuente: `https://development.grupogaratu.com/metodologia-scrum-desarrollo-software/`

**Roles**. Un equipo *Scrum* está formado por tres roles: el dueño del producto (*product owner*), el equipo de desarrollo (*development team*) y un gestor experto en la metodología (*Scrum master*):

- *Product owner*. Es el máximo responsable del trabajo que va a desarrollar el equipo de trabajo. Debe establecer los objetivos a alcanzar y asegurar que todo el mundo los entiende. Se encarga de definir la lista de requisitos a desarrollar en el *sprint* (la llamada "lista de producto") y establecer prioridades.
- *Development team*. Se compone de un grupo de profesionales que se autoorganizan para realizar el trabajo asignado al *sprint*. Todos los miembros tienen el mismo grado de responsabilidad, aunque pueden tener habilidades especializadas que dotan al grupo de un carácter multidisciplinar. Su tamaño debe ser el adecuado para que el desarrollo sea ágil y se garantice la entrega. La guía de *Scrum* recomienda equipos de entre tres y nueve miembros [4], que pueden incluir personas con otros roles.
- *Scrum master*. Es el líder encargado de garantizar que el resto de personal entiende y adopta la metodología *Scrum*. Entre sus tareas destacan ayudar al dueño de producto a crear la lista de producto para maximizar el valor del mismo, guiar al equipo de desarrollo, facilitar la organización de las reuniones que se requieran y, en general, apoyar a la organización a optimizar la puesta en práctica de la metodología.

**Artefactos**. Son documentos de trabajo en formatos diversos que dan soporte al proceso *Scrum*, permitiendo la trazabilidad y transparencia de la información clave.

- Lista de producto (*Product blacklog*). Es una lista ordenada de los requisitos del producto. Su responsable es el dueño del producto, que debe garantizar su corrección y disponibilidad. Esta lista evoluciona a medida que se va desarrollando el producto, detallando todas las características, funcionalidades, mejoras y correcciones que se van realizando. Cada elemento de la lista lleva asociado una descripción, un orden, una estimación de tiempo y un valor.
- Lista de "pendientes" (*Sprint blacklog*). Es el subconjunto de elementos de la lista de producto que se consideran en el *sprint* actual, junto a su plan de desarrollo. Debe tener el suficiente nivel de detalle para que sirva de guía a los miembros del equipo de desarrollo, quienes pueden añadir o eliminar elementos a medida que se completa el *sprint*. Permite conocer en todo momento las tareas pendientes, en desarrollo o terminadas.
- Gráficas de seguimiento (*Burn-down charts*). Aunque no se describen oficialmente en la guía de *Scrum*, es habitual generar gráficas de seguimiento para evaluar el cumplimiento de objetivos y la productividad del equipo.

**Eventos**. Una de las singularidades de *Scrum* es la definición y organización de eventos (bloques de tiempo con duración máxima) que optimicen el aprovechamiento del tiempo fomentando así la productividad. Son necesarios para garantizar la transparencia e inspección, ya que en ellos se definen las tareas a realizar y se revisa el trabajo concluido. Todos los eventos giran en torno al *sprint*:

- Reunión diaria (*daily Scrum meeting*). Es un reunión corta, de 15 minutos y que se recomienda que sea de pie para evitar su prolongación. En ella, el equipo de desarrollo distribuye las tareas a realizar en ese día y evaluar si el progreso es el adecuado.
- Planificación (*Sprint planning*). Se establece la funcionalidad que va a incorporarse al producto en el siguiente *sprint* y se decide cómo se realizará el trabajo para lograr dicho objetivo. Aunque el peso de esta actividad recae en el *product owner*, quién habrá priorizado la lista de producto, es importante que también participe el equipo de desarrollo para aportar su visión sobre el desarrollo y garantizar que todos comprenden el objetivo.
- Revisión (*Sprint review*). Al terminar el *sprint*, todo el equipo analiza el trabajo realizado junto al cliente. Se trata de una reunión informal donde se informa del estado actual, se demuestra el funcionamiento del producto y se recopila información que sea de utilidad para planificar los siguientes *sprints*.
- Retrospectiva (*Sprint retrospective*). Se realiza tras la revisión del *sprint* y su máximo responsable es el *Scrum master*. Su finalidad es analizar el *sprint* respecto a las personas, procesos, etc. con el fin de identificar problemas y plantear mejoras.

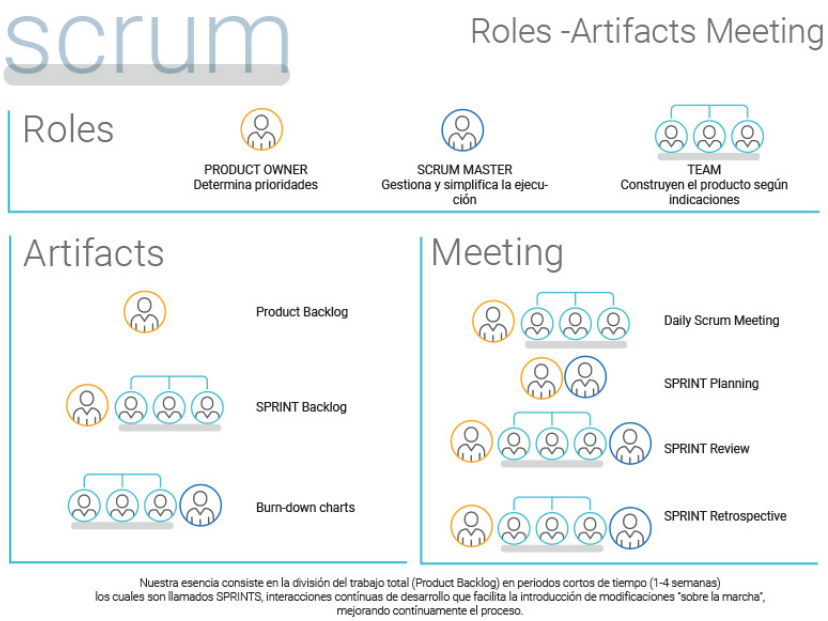



Figura 9: Roles, artefactos y reuniones en *Scrum*. Fuente: `https://development.grupogaratu.com/metodologia-scrum-desarrollo-software/`

## 3.3. Plataforma *YouTrack*

*YouTrack* es una plataforma web, también disponible como aplicación de escritorio, para la gestión de proyectos. Sus principales funcionalidades son:

1. Gestión de incidencias. Permite crear y hacer el seguimiento de las incidencias que surgen durante el desarrollo el proyecto.
2. Base de conocimiento. Es un área donde se pueden crear artículos, notas sobre reuniones, documentación, etc. Soporta el lenguaje *Markdown*.
3. Generación de informes. Dispone de una gran variedad de gráficos e informes para analizar el progreso del proyecto y cómo se distribuye el trabajo entre los miembros del equipo.
4. Gestión del tiempo. Permite incorporar estimaciones a las tareas y realizar su seguimiento, así como crear diagramas de Gantt para la planificación global.

Además, *YouTrack* permite la creación de "paneles ágiles" inspirados en el concepto de *sprint*. Aunque no implementa los elementos propios de *Scrum*, la plataforma ofrece la posibilidad de crear tareas en cada *sprint* y asignarles un responsable, una prioridad y una estimación de tiempo. Las tareas pueden estar en cuatro estados: abiertas, en progreso, por verificar y terminadas. Por defecto, cada proyecto en *YouTrack* tiene dos paneles:

uno para la gestión del proyecto (*project management*) y otro para el desarrollo (*project development*).

# Referencias


[1] S. Chacon and B. Straud, *Pro Git*. Apress, 2nd ed., 2014. Disponible en: https://git-scm.com/book/es/v2.

[2] B. Collins-Sussman, B. W. Fitzpatrick, and C. M. Pilato, *Version Control with Subversion*. O'Reilly Media, rev. 2147 ed., 2006. Disponible en: http://svnbook.red-bean.com/.

[3] Atlassian, *Tutorial Git*, 2020. Disponible en: https://www.atlassian.com/es/git/tutorials.

[4] K. Schwaber and J. Sutherland, *La guía de Scrum*. 2016. Disponible en: https://www.scrumguides.org/docs/scrumguide/v2016/2016-Scrum-Guide-Spanish.pdf.

[5] Atlassian, *Agile Coach*, 2020. Disponible en: https://www.atlassian.com/es/agile/scrum.