UT4 – OPTIMIZACIÓN Y DOCUMENTACIÓN – PARTE II- CONTROL DE VERSIONES

RESULTADOS DE APRENDIZAJE ASOCIADOS
4.- Optimiza código empleando las herramientas disponibles en el entorno de desarrollo.
CRITERIOS DE EVALUACIÓN
a) Se han identificado los patrones de refactorización más usuales.
b) Se han elaborado las pruebas asociadas a la refactorización.
c) Se ha revisado el código fuente usando un analizador de código.
d) Se han identificado las posibilidades de configuración de un analizador de código.
e) Se han aplicado patrones de refactorización con las herramientas que proporciona el entorno de desarrollo.
f) Se ha realizado el control de versiones integrado en el entorno de desarrollo.
g) Se han utilizado herramientas del entorno de desarrollo para documentar las clases.

UT4 – OPTIMIZACIÓN Y DOCUMENTACIÓN – PARTE II- CONTROL DE VERSIONES

Índice de contenido

1.- Control de versiones

Un producto software se somete a innumerables modificaciones a lo largo de su ciclo de vida, lo cual hace imprescindible el uso de una herramienta que gestione los diversos cambios que experimenta durante su desarrollo o período de utilización. Git se posiciona como una de las soluciones de gestión de versiones más empleadas en el sector. Sus principales ventajas son:

a) Es un sistema de control de versiones gratuito y de código abierto.
b) Se adapta tanto a proyectos de pequeña como de gran envergadura.
c) Destaca por su facilidad de aprendizaje y su eficiente rendimiento.

En la elaboración de la mayoría de los productos software participan numerosos profesionales, lo que hace esencial que la gestión de versiones se realice de manera automatizada, evitando así la ejecución manual que podría derivar en errores significativos y en un coste elevado.

1.1.- Almacenamiento de las distintas versiones

En la práctica, los sistemas de control de versiones se pueden clasificar principalmente en dos tipos: **centralizados y distribuidos**, los cuales son los más conocidos y utilizados. Sin embargo, dentro de estas categorías, existen múltiples sistemas con diferentes características y funcionalidades. Además, es posible considerar otras formas menos comunes de gestión de versiones, aunque en esencia, la mayoría se ajusta a uno de estos dos modelos principales.

Sistemas Centralizados

- ***Subversion*** (SVN): Uno de los sistemas de control de versiones centralizados más populares.

- ***Concurrent Versions System*** (CVS): Anterior a SVN, pero con un modelo similar de centralización.

Sistemas Distribuidos

- ***Git:*** El sistema de control de versiones distribuido más utilizado, conocido por su eficiencia y flexibilidad.

- ***Mercurial:*** Otro sistema distribuido, similar a Git en muchos aspectos, pero con diferencias en la interfaz y en algunas funcionalidades.

Además de estos, existen **herramientas y plataformas** que, aunque se basen en estos sistemas, ofrecen funcionalidades adicionales para la gestión de proyectos, colaboración, y control de versiones, como:

- GitHub
- GitLab
- Bitbucket

Interesante video

¡Aprende GIT ahora! curso completo GRATIS desde cero

https://www.youtube.com/watch?v=VdGzPZ31ts8

1.2.- Aprendemos a usar Git

Git – Cheat Sheet

https://rogerdudler.github.io/git-guide/files/git_cheat_sheet.pdf

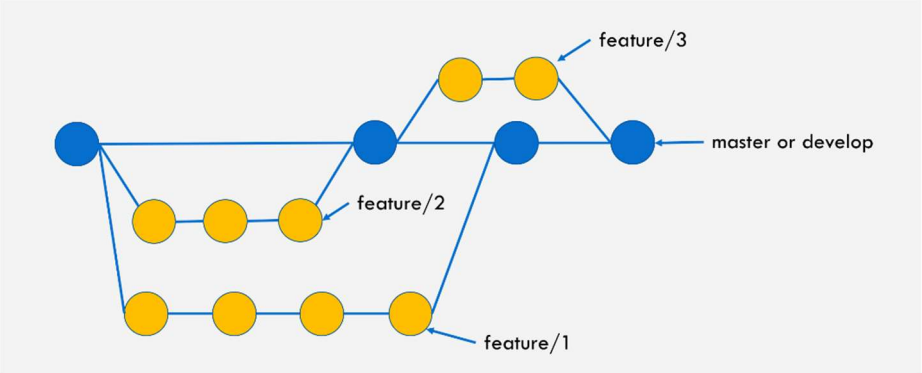
Antes de seguir, deberemos respondernos a la siguiente pregunta: ¿Qué es un repositorio?

Un **repositorio**, en el contexto del desarrollo de software y el control de versiones, es un espacio de almacenamiento centralizado donde se guardan, organizan, y gestionan los archivos de código fuente de un proyecto, junto con su historial de cambios. Este permite a los desarrolladores y colaboradores rastrear y controlar las modificaciones realizadas sobre el proyecto a lo largo del tiempo, facilitando la colaboración, el mantenimiento, y la distribución del código.

Los repositorios pueden ser **locales** (ubicados en la máquina de un desarrollador) o **remotos** (alojados en un servidor o plataforma en la nube, como GitHub, GitLab, o Bitbucket).

Los sistemas de control de versiones, como Git, permiten a los usuarios trabajar con estos repositorios para:

- Añadir cambios (***commits***)
- Crear ramas (***branches***) para el desarrollo de nuevas características de forma aislada
- Fusionar cambios (***merge***)



1.2.1.- Cómo crear un repositorio local

Para iniciar un repositorio local, ejecuta el siguiente comando:

```
git init
```

Tras crearlo, es necesario configurar el nombre y la dirección de correo electrónico que utilizarás para "firmar" los commits en el nuevo repositorio. Para ello, utiliza los siguientes comandos de Git:

```
git config user.name "Tu nombre"
git config user.email tu_direccion_de_email
```

Si prefieres que esta configuración se aplique a todos los repositorios en tu sistema, puedes añadir la opción **--global**.

Sin embargo, no es recomendable si participas en diversos proyectos o equipos, dado que es posible que necesites utilizar diferentes direcciones de correo electrónico.

De querer hacerlo globalmente, los comandos serían:

```
git config --global user.name "Tu nombre"
git config --global user.email tu_direccion_de_email
```

1.2.2.- Integrar un repositorio remoto

Para conectar un repositorio local a un remoto en Git, utiliza el comando **git remote add origin <URL del repositorio remoto>**. "origin" es un nombre convencional para referirse al repositorio remoto principal, pero puedes usar cualquier otro nombre y añadir múltiples repositorios remotos si lo necesitas.

Por ejemplo:

```
git remote add origin https://github.com/j2logo/tutorial-flask.git
```

Si prefieres empezar trabajando directamente con un repositorio remoto, puedes clonarlo usando **git clone <URL del repositorio remoto>**. Este comando copia el repositorio remoto a tu máquina local, creando un nuevo repositorio local ya vinculado al remoto.

Por ejemplo:

```
git clone https://github.com/j2logo/tutorial-flask.git
```

1.2.3.- Comandos Git para confirmar los cambios: *add* y *commit*

Para confirmar cambios en tu repositorio local de Git, sigue dos pasos:

1) Primero, selecciona los archivos que han sido modificados y que deseas incluir en tu próxima confirmación (**commit**). Esto se hace con el comando "**git add"**, seguido del nombre del archivo o archivos específicos. Si prefieres incluir todos los archivos modificados de una vez, utiliza "**git add ."**

Por ejemplo:

```
git add <nombre_fichero>
```

o para todos los archivos:

```
git add .
```

2) Después de seleccionar los archivos, realiza la confirmación de los cambios con "**git commit"**, acompañado de un mensaje que describa

lo que has modificado o añadido al proyecto. Este paso registra tu cambio en el historial del repositorio.

Ejecuta:

```
git commit -m "Mensaje del commit"
```

1.2.4.- Integrando los cambios en tu repositorio remoto

Para subir tus cambios del repositorio local al remoto en Git, utiliza el comando **git push** seguido del nombre del repositorio remoto (comúnmente "origin") y el nombre de la rama local que deseas actualizar o enviar.

Por ejemplo:

```
git push origin <nombre_rama_local>
```

Recuerda que "**origin**" es el nombre estándar para referirse al repositorio remoto principal, pero este puede variar si le has asignado un nombre diferente.

Pero… ¿Cómo sé cómo se llama la "rama local"? ¿Y si se me olvida el nombre?

Para saber el nombre de la rama local en la que te encuentras trabajando en Git, puedes utilizar el comando:

git branch

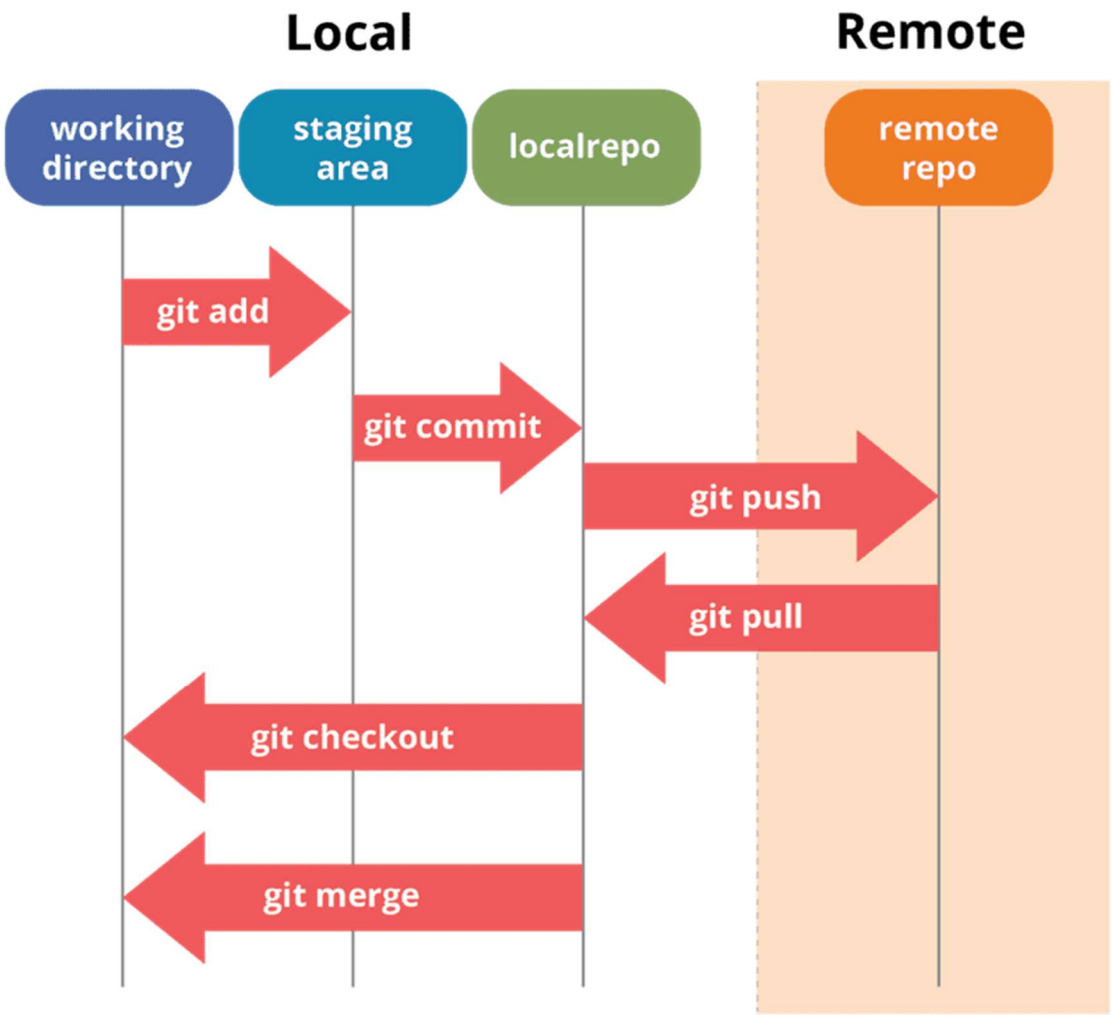
Al ejecutar este comando, Git listarán todas las ramas locales que existen en tu repositorio. La rama en la que te encuentras actualmente estará

marcada con un asterisco (*) al lado de su nombre. Esto te ayuda a identificar rápidamente en qué rama estás trabajando sin confusiones.

Si quieres ampliar…

Comando Git principales
https://j2logo.com/comandos-git-principales/

Esquema de la arquitectura de los repositorios:



2.- Referencias bibliográficas

- Moreno Pérez, J.C. *Entornos de desarrollo.* Editorial Síntesis.
- Ramos Martín, A. & Ramos Martín, M.J. *Entornos de desarrollo*. Grupo editorial Garceta.