Programación para Ciencia de Datos

Francisco Javier Mercader Martínez

Índice

Tema 1: Control de Versiones & Desarrollo basado en Test

1.1) Metodologías de Desarrollo de Software

1.1.1) Introducción

A lo largo de la historia, se han descrito una serie de metodologías para mejorar y facilitar el proceso de desarrollo software. Cada metodología divide el trabajo en fases, cada una con un conjunto distinto de actividades.
El proceso de dividir el trabajo de desarrollo, normalmente en distintas fases, se conoce como **metodología de desarrollo de software.**
Estas diferentes fases de trabajo pueden incluir:

- Especificación de entregables o artefactos
- Desarrollo y verificación del código con respecto a la especificación
- Despliegue del código a sus clientes finales o entorno de producción.

1.1.2) Tipos de metodologías

- Metodología clásica - En cascada

Es un proceso de gestión de proyectos que hace hincapié en una progresión secuencial de una etapa del proceso a la siguiente.

Originario de las industrias manufacturera y de la construcción, y adoptado más tarde por la ingeniería de hardware.

Las etapas originales eran la especificación de requisitos, el diseño, la implementación, la integración, las pruebas, la instalación y el mantenimiento.

El progreso se visualizaba como un flujo de una etapa a otra (de ahí el nombre).

- Metodologías ágiles - Scrum

Es un grupo de metodologías diseñadas para ser más ligeras y flexibles que las metodologías clásicas.

Scrum utiliza ciclos de desarrollo predefinidos llamados *sprints*, normalmente de entre una semana y un mes de duración.

- Comienzan con una reunión de planificación del sprint para definir los objetivos y terminan con una revisión del sprint y una retrospectiva del sprint para discutir el progreso y cualquier problema que haya surgido durante ese sprint.

En estas reuniones, son facilitadas por el *Scrum master*.

Con dichas metodologías en mente, durante las fases de desarrollo e implantación de software surgen cuestiones que el equipo de desarrollo debe de manejar con cuidado.

- ¿Cómo se van a registrar los cambios realizados en el código fuente?
- ¿Cómo deshacer determinados cambios y volver a una versión anterior del proyecto?
- ¿Podemos llevar en paralelo diferentes versiones del proyecto?
- ¿Cómo podemos planificar las pruebas que hagamos a nuestro código?
- ¿Qué tipo de pruebas debemos de implementar? ¿A qué nivel?
- En el resto del tema veremos las herramientas que van a permitir responder y gestionar dichas herramientas en un proyecto de desarrollo del software.

1.2) Sistema de Control de Versiones

1.2.1) Introducción

Un Sistema de Control de Versiones (en adelante SCV), es un software que controla y organiza las distintas revisiones que se realicen sobre uno o varios documentos

Pero, ¿qué es una revisión?

- Una revisión es un cambio realizado sobre un documento, por ejemplo, añadir un párrafo, borrar un fragmento o algo similar.

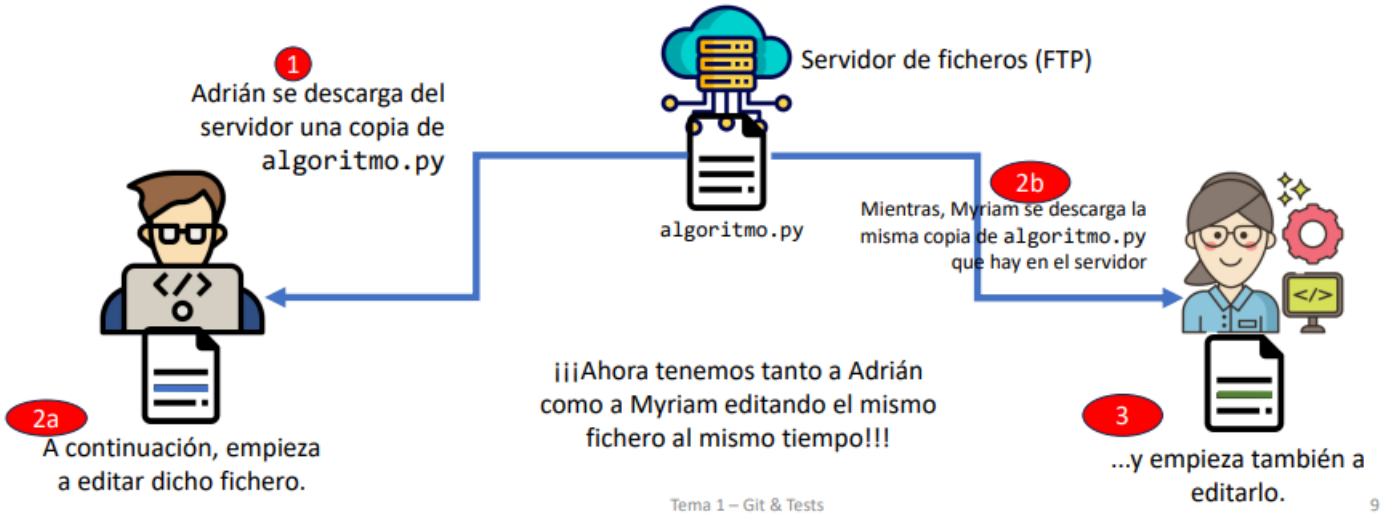
Un SVC guarda el historial de las distintas modificaciones sobre un fichero.

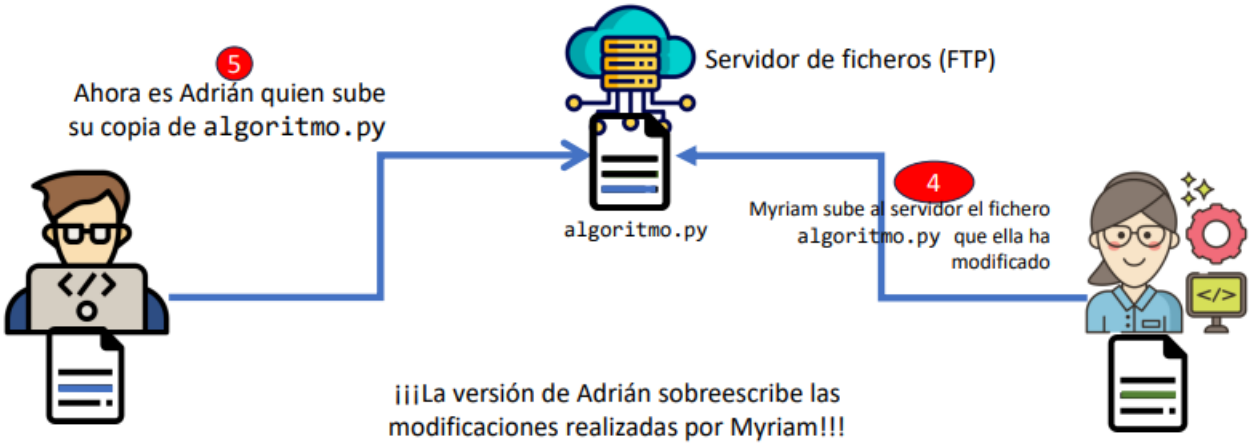
En cualquier momento podemos restaurar la revisión que queramos de un fichero.

Permite mantener una copia de seguridad de todas las modificaciones realizadas sobre un fichero, lo cual nos facilita la tarea de deshacer algo que esté mal.

Su mayor potencial surge cuando el desarrollo se hace en grupo.

- Riesgos del desarrollo en grupo sin SCV





1.2.2) Tipos

- Sistema Centralizado
  - Funcionan como un entorno clásico Cliente- Servidor
    - → Servidor: alojara el repositorio del proyecto, con toda la información de los cambios, ficheros binarios añadidos, ...
    - → El cliente trabaja con una "copia de trabajo" del servidor. Es una copia de cómo estaba el servidor en una revisión determinada.
  - El desarrollador hace cambios sobre esa copia de trabajo, y cuando considera que ha terminado con esa modificación la sube (`commit`) al servidor, el cual se encargará de fundir esos cambios en el repositorio.
  - Según la forma de controlar conflictos sobre un fichero podemos distinguir:
    - → Bloqueo de archivo: cuando alguien está trabajando con un archivo, se bloquea su acceso para el resto de usuarios.
    - → Fusión de versiones: controla qué líneas se han modificado por cada usuario.
- Sistema Distribuido
  - Similar a un sistema *Peer-to-Peer* (P2P).
  - La copia de trabajo de cada cliente es un repositorio en sí mismo, una rama nueva del proyecto central.
  - La sincronización de las distintas ramas se realiza intercambiando "parches" con otros clientes del proyecto.
  - No hay una copia original del código del proyecto, solo existen las distintas copias de trabajo.
  - Operaciones como los `commits`, no necesitan de una conexión con un servidor central.
    - → Dicha conexión solo es necesaria al "compartir" tu rama con otro cliente del sistema.
  - Cada copia de trabajo es una copia remota del código fuente y de la historia de cambios, dando una seguridad muy natural contra la pérdida de los datos.

1.2.3) Git

1.2.3.1) Introducción

Escrito en C y en gran parte construido para trabajar en el kernel de Linux.

Su principal potencial es su modelo de ramas (lo veremos más adelante).

Diseñado para operar en local de forma rápida y sin conexión.

- Pocos comandos que accedan al servidor

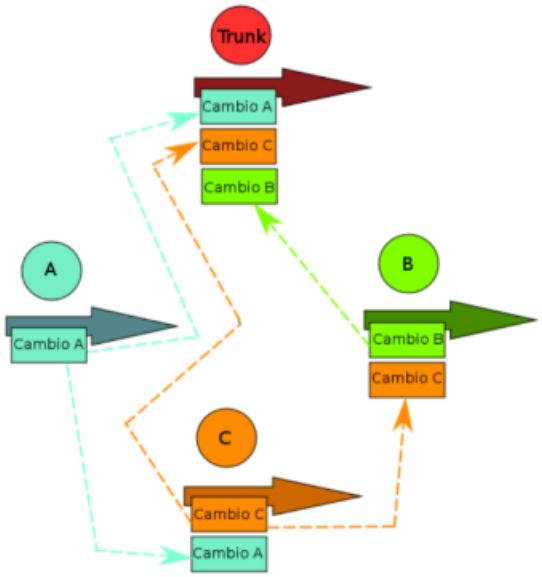
Los repositorios ocupan muy poco espacio.

1.2.3.2) Estructura general

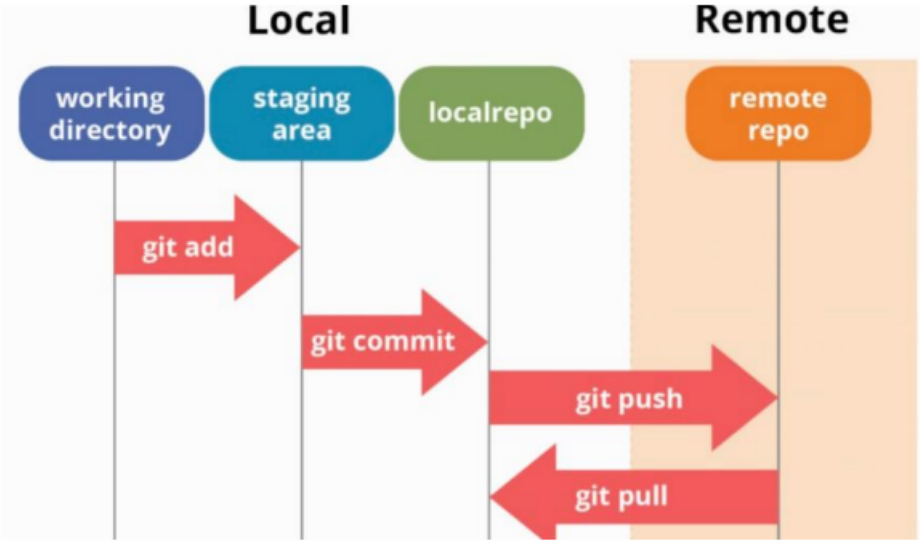
Cada desarrollador tiene su repositorio local.

Si un usuario realiza un cambio, lo hace localmente y cuando crea oportuno puede compartirlo con el resto de usuarios.

Existe una rama principal (Trunk) en la cual todos los usuarios registran los cambios realizados obteniendo de esta forma la versión final.



1.2.3.3) Git Workflow

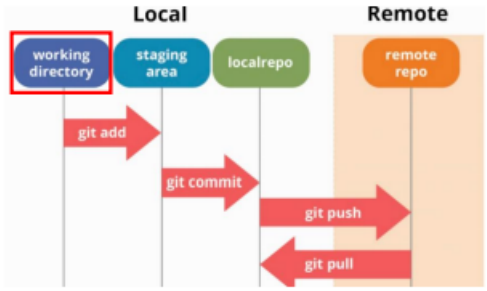


El primer paso es generar un *repositorio Git* asociado a nuestro directorio de trabajo (`working directory`) mediante el comando con `git init`.

- Esto creará dos directorios virtuales, un almacén de objetos (`.git`) y un área de montaje (o Index) que permitirán registrar los cambios realizados en el `working directory`.

En este punto hemos creado el repositorio Git, pero este se encuentra vacío.

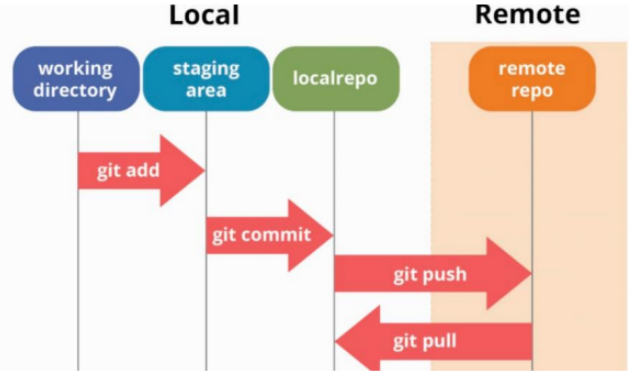
- En la figura de la derecha, el fichero `index.html` está en el directorio de trabajo pero **no** en el repositorio...



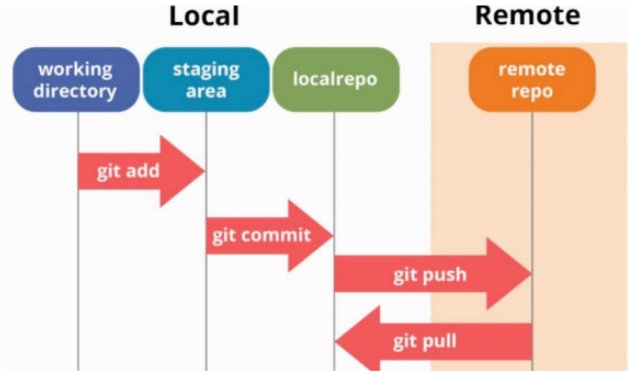
Con `git init` asociamos a nuestro directorio de trabajo donde tenemos todos los archivos de un proyecto:

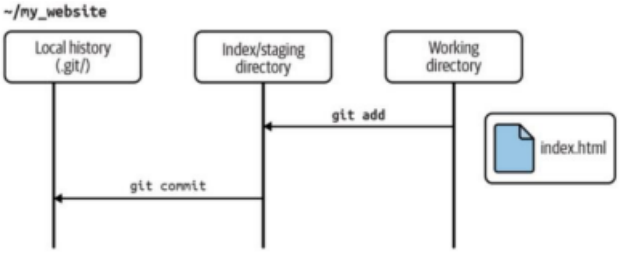
1) Nuestro repositorio local
2) Nuestro repositorio `git`

Para añadir los ficheros del `working directory` que queramos tener controlados dentro del repositorio Git, deberemos ejecutar el comando `git add` (Ej: `git add index.html`). Esto deja el archivo en una fase o área preparatoria (`staging area`).
Con esto indicamos a Git que queremos incluir la versión más reciente del fichero como parte de la revisión del repositorio.
De forma análoga, podemos indicar a git que queremos quitar un determinado fichero del repositorio mediante `git rm`.

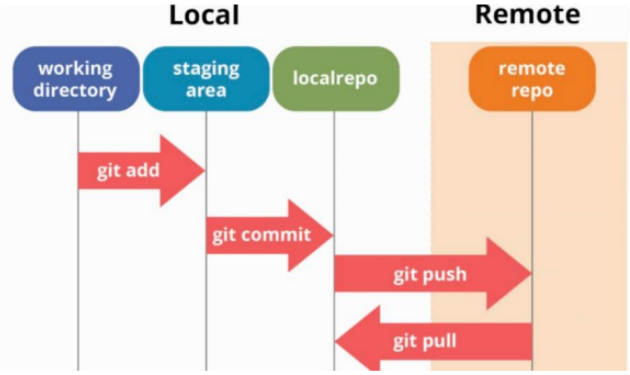


El siguiente paso lógico es consignar el nuevo fichero en el repositorio mediante el comando `git commit`.
Todos estos comandos lo que han hecho ha sido mover el fichero `index.html` del directorio de trabajo al directorio índice (`index`) y finalmente al subdirectorio con el historial local.





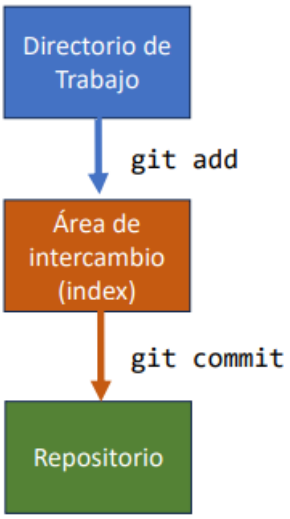
El *commit* que hemos realizado anteriormente es local, para actualizar la rama principal (Trunk) que se encuentra en un servidor deberemos usar el comando `git push`.
También, podríamos bajarnos los cambios que se realizado en dicha rama principal con `git pull`.



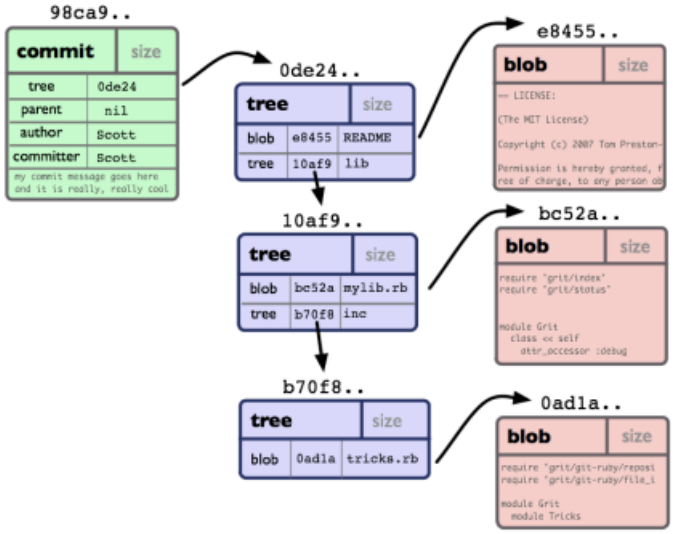
1.2.3.4) Componentes de un repositorio Git

Dentro de un repositorio Git nos encontramos con dos estructuras de datos

- El almacén de objetos: contiene los ficheros de datos originales y todos los mensajes de log, fechas y otra información necesaria para reconstruir el estado del proyecto en una determinada fecha.
  - Tiende a crecer de tamaño conforme editamos ⟶ Git comprime dicho almacén en *packfiles*.
  - Cada fichero se almacena como un *blob* y se indexa por su hash SHA-1 por lo que es muy difícil encontrar dos objetos con el mismo nombre.
  - Se copia a un nuevo repositorio cuando se ejecuta `git clone`.
- Área de intercambio o Índice: es el área intermedia donde se puede configurar el "aspecto" que tendrá la entrega antes de hacer `git commit`. Es privado del repositorio Git al cual pertenece.




```
.
|-- README
`-- lib
    |-- inc
    |   `-- tricks.rb
    `-- mylib.rb
```



1.2.3.5) Gestión de ramas

Una rama es una línea de desarrollo que existe de forma independiente a otra, pero que comparte una historia común en algún punto temporal anterior.

Se puede decir que una rama siempre nace como una copia de algo y a partir de ahí pasa a generar su propia historia.

Git permite generar muchas ramas (branches) dentro de un proyecto (y por tanto múltiples líneas de desarrollo).

Una rama se implementa en git como un puntero a un commit ya existente.

Git lleva un control de los commits realizados sobre dicha rama de forma independiente.

Existen múltiples razones que justifican el uso de ramas dentro de un proyecto:

- Para representar diferentes estados en un proyecto (estable, en desarrollo, release candidate, etc.)
- Para representar una nueva versión del proyecto (porque sabes que determinados clientes van a querer seguir usando la versión anterior).
- Permite centrarse en una determinada funcionalidad del producto que estemos desarrollando.

Cuando se crea un repositorio, se crea una rama principal con nombre `master`.

Aunque un proyecto pueda contener múltiples ramas, un desarrollador trabajará sobre una rama en concreto en un momento determinado.

En cada momento, se tendrá una única rama activa.

- Inicialmente, al crear un repositorio, la rama master constituirá la rama activa.

Cada vez que ejecutemos `git commit` se aplicará sobre la rama activa.

El nombre de la rama hace en realidad referencia al `commit` más reciente realizado (`HEAD` en terminología `git`)

- En la figura de la derecha vemos una rama `development` que se ha fusionado con la rama `main`, de ahí que solo haya un `HEAD`.

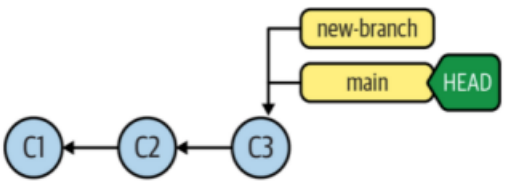
1.2.3.6) Creación de ramas

Para crear una rama derivada de la rama activa en la que nos encontremos debemos ejecutar el comando `git branch <nombre_de_la_rama> <punto_inicial>`.

- Por defecto `punto_inicial` hace referencia al HEAD de la rama activa.

Al ejecutar dicho comando simplemente hemos creado un nueva rama **pero** no nos hemos movido a la misma (el HEAD sigue apuntando a la rama origen)

En el ejemplo de la figura hemos creado una nueva rama `new-branch` pero la rama activa sigue siendo `main`.



Para poder movernos a una rama que hayamos creado previamente deberemos ejecutar `git checkout <nombre_de_la_rama>`

Podemos obtener un listado de todas las ramas que tenemos en nuestro proyecto con el comando `git branch`.

Existe la posibilidad de crear una rama y moverse automáticamente a ella mediante el comando `git checkout -b <nombre_de_la_rama>`

Para eliminar una rama podemos ejecutar `git branch -d <nombre_de_la_rama>`

- **No debe de ser la rama activa.**

1.2.3.7) Fusión de ramas

Una rama puede fusionarse con otra siempre que estén en el mismo repositorio.

Una fusión entre dos ramas es realmente un nuevo `commit` que representa dicha unificación.

Al hacer la fusión, la rama activa es la rama que recibe la fusión con otra rama.

Para ello, usamos el comando `git merge <rama_a_fusionar>`

- La siguiente secuencia permite integrar dentro de `main_branch` los cambios realizados en la rama `modified_branch`

  1) `git checkout main_branch`

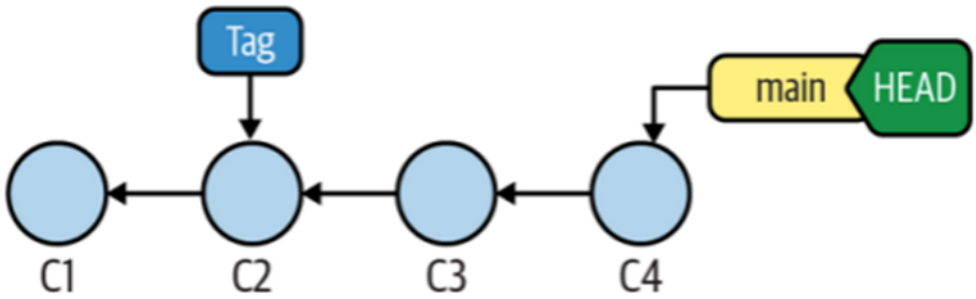
  2) `git merge modified_branch`

1.2.3.8) Gestión de tags

Una rama apunta al `commit` más reciente (`HEAD`) pero ¿qué pasa si queremos hacer un `commit` a partir de un punto anterior a `HEAD`?

Git nos permite etiquetar un `commit` en particular mediante `tags`.

- Esto nos permite definir puntos de referencia estáticos dentro del proyecto, porque representan, por ejemplo, un punto estable de desarrollo dentro del proyecto.



1.2.3.9) Creación de tags

Para crear un tag dentro de la rama activa debemos ejecutar `git tag <nombre_del_tag>`

Con `git tag` podemos ver el listado de todos los tags de la rama activa.

Una vez creado un tag podremos generar, por ejemplo, una rama a partir de él con `git branch <nombre_de_la_rama> <nombre_del_tag>`

- Ejemplo: crear una rama llamada `mi_nueva_rama` a partir del tag v1.1.

  – `git branch mi_nueva_rama v1.1`

1.2.3.10) `branches vs tags`

Un `tag` y un `branch` pueden parecer similares pero sirven para objetivos diferentes.

Un `tag` es estático pues siempre apunta a un punto específico de una rama.

- Sirven para marcar, por ejemplo, cuando una determinada versión de un producto fue lanzado al marcado.
- Pueden también usarse como *marcadores temporales*.

Las ramas son *dinámicas* pues van avanzando conforme se van realizando `commits`.

Por tanto, a la hora de decidir si crear un `tag` o un `branch` debemos de preguntarnos si el punto de referencia es estático o dinámico.

1.3) Desarrollo Impulsado por Pruebas

1.3.1) Introducción

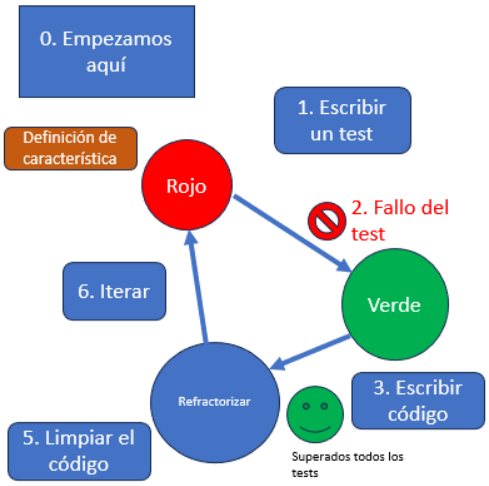
Las pruebas de software son el acto de examinar los artefactos y el comportamiento del software que se está probando mediante validación y verificación.

El desarrollo impulsado por pruebas (*Test-Driven Development*, TDD) es una metodología de desarrollo de software que enfatiza la escritura de pruebas antes de escribir el código real.

- Sigue un ciclo específico de escribir pruebas, ejecutarlas y luego escribir el código para que las pruebas pasen.
- TDD promueve un enfoque disciplinado del desarrollo de software y puede conducir a un código más confiable, mantenible y bien diseñado.

TDD implica los siguientes pasos:

1) Escribir una prueba: Comenzar escribiendo una prueba que defina el comportamiento esperado de una funcionalidad específica. A menudo se escribe como una función que afirma ciertas condiciones o resultados.
2) Ejecutar la prueba para verificar que falle. Este es un paso importante porque garantiza que la prueba sea válida y que el comportamiento esperado aún no esté implementado.
3) Escribir el código: Implementar la cantidad mínima de código necesaria para que la prueba pase. Concéntrate en resolver el problema específico que aborda la prueba no en agregar funcionalidad innecesaria.



4) Ejecutar la prueba nuevamente: Para verificar si el cambio de código ha logrado que la prueba pase. Si la prueba falla, volver al paso 3.
5) Refactorizar el código: Una vez que la prueba pasa, refactorizar el código para mejorar su diseño, estructura o rendimiento manteniendo la aprobación de las pruebas. Garantiza que el código se mantenga limpio y mantenible.
6) Repetir el ciclo: Pasar a la siguiente característica o pieza de funcionalidad y volver al paso 1.

1.3.2) Tests unitarios vs test de integración

A lo largo del grado, probablemente ya has creado una prueba sin darte cuenta.

- ¿Recuerdas cuando ejecutaste tu aplicación y la usaste por primera vez? ¿Revisaste las funciones y experimentaste usándolas?

- Eso se conoce como **prueba exploratoria** y es una forma de prueba manual.

La prueba exploratoria es una forma de prueba que se realiza sin un plan.

- ¡solo estás explorando la aplicación!

Frente a esta forma manual, las pruebas automatizadas son la ejecución de su plan de pruebas mediante un *script* en lugar de una persona.

- En este sentido, podemos distinguir entre *pruebas unitarias* y *pruebas de integración*.

Las **pruebas unitarias** prueban unidades individuales (módulos, funciones, clases) de forma aislada del resto del programa.

Sólo validan las unidades individuales, por lo que sólo se debe inicializar la unidad deseada en las clases de prueba.

Para ignorar el servicio externo o las dependencias técnicas externas, como el origen de datos, podemos usar simuladores para imitar sus comportamientos.

Las pruebas unitarias deben ejecutarse en muy poco tiempo ya que no dependen de ninguna fuente o servicio externo.

El propósito de las **pruebas de integración** es verificar si varios módulos desarrollados por separado funcionan juntos como se espera.

Se realiza activando muchos módulos y ejecutando pruebas de nivel superior contra todos ellos para garantizar que funcionen juntos.

- Estos módulos pueden ser partes de un solo ejecutable o separados.

La principal diferencia entre la prueba de integración en comparación con la prueba unitaria es que se inicializan todos los servicios internos.

- Se puede implementar una base de datos de prueba que tenga los mismos nombres de tabla y columna que la de producción para probar las capas externas.

1.3.3) Definir tests en Python

A fin de poder escribir tests en python, tenemos la primitiva `assert`.

- `assert expression[, assertion_message]`

La instrucción `assert` de Python permite escribir comprobaciones en el código fuente.

Se pueden utilizar para probar si ciertas suposiciones siguen siendo verdaderas mientras desarrollamos una aplicación.

Si algún `assert` devuelve `False`, entonces existe un error en el código fuente de la aplicación.

```
1 assert sum([1, 2, 3]) == 6, "Debe sumar 6" # (No devolverá nada por consola)
2
3 assert sum([1, 1, 1]) == 6, "Debe sumar 6"
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: Debe sumar 6
```

1.3.4) Frameworks para tests

En python exiten múltiples test runners, los más destacados son:

- `unittest` (https://docs.python.org/3/library/unittest.html): incorporado a la biblioteca estándar de Python desde la versión 2.1. Contiene tanto un marco como un ejecutor de pruebas.
- `nose` (https://nose.readthedocs.io/en/latest/): Más eficiente que `unittest` para gestionar centenares o miles de tests. Es compatible con cualquier prueba escrita con `unittest` y puede usarse como un reemplazo directo del mismo.
- `pytest` (https://docs.pytest.org/en/7.4.x/): Admite la ejecución de casos de prueba `unittest`. La verdadera ventaja de `pytest` radica en la escritura de casos de prueba como una serie de funciones en un archivo Python que comienzan con el nombre `test_`.

1.3.5) Definición de test con `pytest`

`pytest` espera que nuestras pruebas se encuentren en archivos cuyo nombre empiecen por `test_` o terminen por `_test.py`.

Por ejemplo, en un fichero `test_capitalize.py` podríamos tener:

```
def capital_case(x):
  return x.capitalize()
def test_capital_case():
  assert capital_case('semáforo') == "Semáforo"
```

A continuación, ejecutando `pytest` sobre la línea de comandos veríamos que nuestra función `capital_case` pasa el test definido.

Si vamos un paso más allá podemos testear si nuestra función `capital_case` comprueba que el parámetro de entrada es un *string* o no. Por tanto, añadimos al fichero `test_capitalize.py` las siguientes líneas:

```
import pytest
def test_capital_case():
  assert capital_case('semáforo') == 'Semáforo'
def test_raises_exception_on_non_string_arguments():
  with pytest.raises(TypeError):
    capital_case(9)
```

Mediante `pytest.raises`, afirmamos que nuestra función debe lanzar un `TypeError` en caso de que el argumento pasado no sea una cadena.

- ¿Pasará o no pasará el test nuestra función `capital_case`?

1.4) Conclusiones

Los sistemas de control de versiones es una herramienta muy útil en el desarrollo de proyectos software pues controla y organiza las distintas **revisiones** que se realicen sobre uno o varios documentos.

Podemos distinguir entre SCV centralizados o distribuidos.

Git es un SCV distribuido cuyo principal potencial es su modelo de ramas.

También podemos etiquetar determinados *commits* con tags.

El desarrollo impulsado por pruebas (*Test-Driven Development*, TDD) es una metodología de desarrollo de software que enfatiza la escritura de pruebas antes de escribir el código real.

Debemos distinguir entre test unitarios vs tests de integración.

Existen diferentes frameworks para el desarrollo de las pruebas, como `nose`, `pytest` o `unittest`.

La instrucción `assert` es la base sobre la que construir la definición de tests en python.

`pytest` permite la ejecución de pruebas de forma sencilla mediante el uso de un nomeclatura especial.

Tema 2: Programación orientada a objetos

2.1) Características de la Programación Orientada a Ojetos

La Programación Orientada a Ojetos (POO) es un paradigma de desarrollo de software.

> Enfoque centrado en los conceptos (abstracciones) del dominio de aplicación $\longrightarrow$ **Objetos**

La etapa de modelado del programa es fundamental

- Identificar los objetos que requería la aplicación.
- Definir las relaciones entre dichos objetos.
- Utilizar un lenguaje formal para especificar los modelos.

Abstracción: permite centrarnos en las propiedades de los tipos de datos y no en la implementación.

Modularidad: permite descomponer el software en componentes (clases, funciones) que se pueden combinar para resolver el problema original.

Encapsulación: permite agrupar en un mismo módilo tanto en la estructura como el comportamiento de los tipos de datos.

Ocultación de Información: permite establecer la visibilidad de las propiedades de un módulo, diferenciando la parte pública y la parte privada.

Herencia: permite definir unas clases a partir de otras.

Polimorfismo: permite que una entidad pueda hacer referencia a objetos de diferente tipo en tiempo de ejecución. Relacionado con el concepto de ligadura dinámica.

2.2) Desarrollo orientado a objetos

Identificar los objetos relevantes al problema.

Describir los tipos de objetos y su propiedades.

Encontrar las operaciones para los tipos de objetos.

Identificar relaciones entre objetos.

Utilizar los tipos de objetos y relaciones para estructurar el software.

> ¿Cómo podemos formalizar el proceso?

2.3) UML

Unified Modelling Language

Lenguaje visual para el modelado de software.
Se basa en el uso de diagramas. Tipos:

- Estructura: Diferentes entidades que componen un programa.
- Comportamiento: Respuesta que va a dar el programa y pasos para lograrlo.

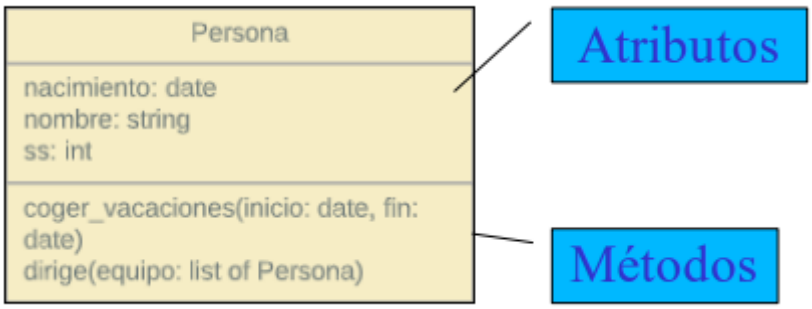
Utilización:

- Diagramas de clases
- Diagramas de casos de uso
- Diagramas de secuencia

2.4) Clases y Objetos

Una clase (concepto) es la definición de un objeto (realización).

- Nombre
- Atributos ⟶ datos que definen su estado (clase/instancia).
- Métodos ⟶ funciones que implementa (clase/instancia).



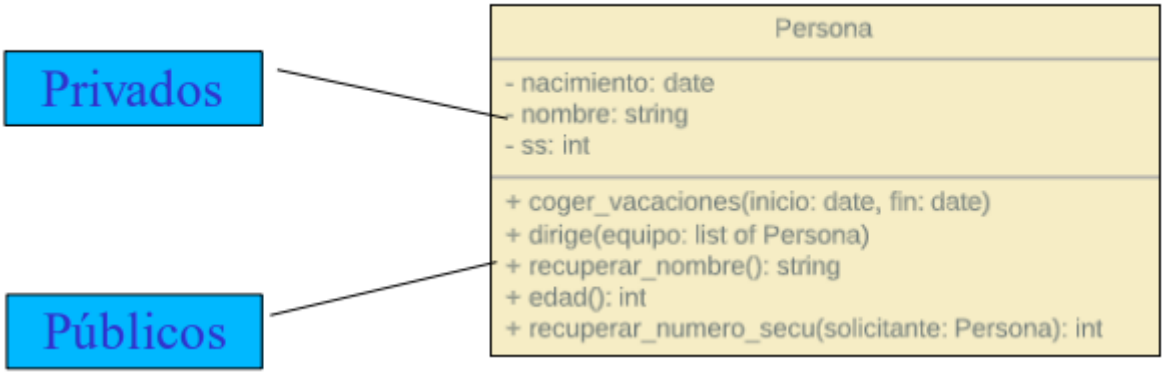
Instanciación: proceso mediante el cual se crea un objeto a partir de su clase.

2.5) Encapsulación

2.5.1) Visibilidad

La visibilidad regula el acceso a la información dentro de una clase ⟶ proceso de encapsulación.

- Pública (+) miembro accesible para las demás clases del programa.
- Protegida (#) miembro accesible sólo a las clases derivadas (herencia).
- Privada (-) miembro sólo accesible por su clase.



2.5.1.1) Caso de uso

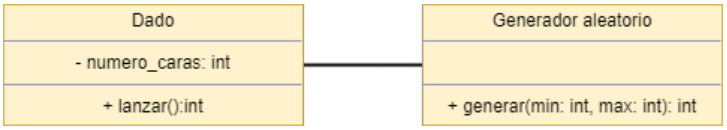
Se desea modelar mediante UML un supermercado en donde se tienen una serie de productos (frescos, refrigerados, congelados y en conserva), guardados en almacenes, y proporcionados por una serie de proveedores.

Se desea también modelar los clientes de dicho supermercado así como su cesta de la compra y la forma en la que la llenan introduciendo productos.

2.5.2) Dependencia

Una dependencia se establece cuando una clase o uno de sus miembros necesita de otra clase para funcionar.

- Permite la libertad de elegir la estructura interna.
- Preserva el uso de externo de una clase durante su ciclo de vida



Gestión de las dependencias mediante visibilidad.