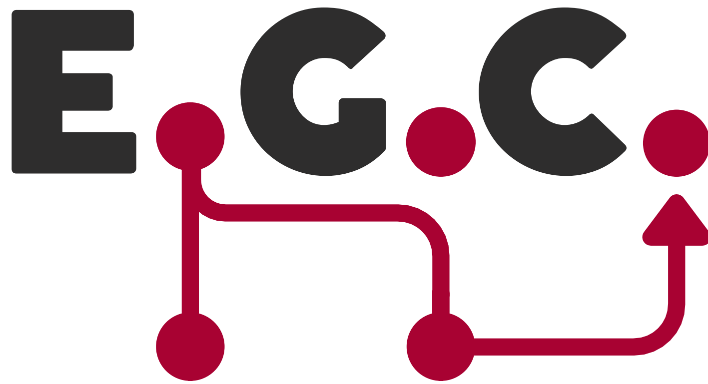


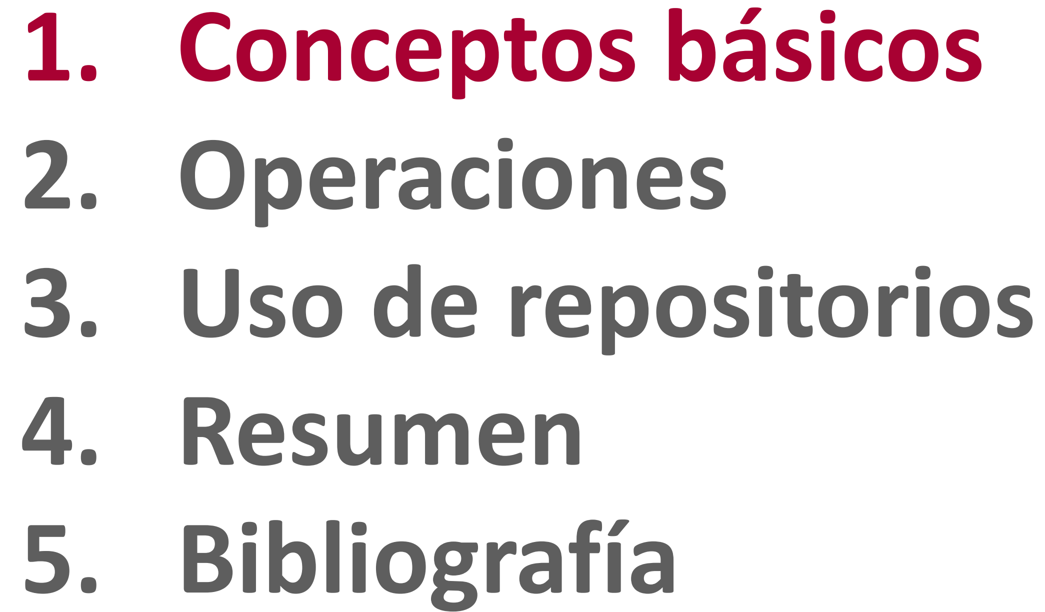
Grado en Ingeniería Informática - Ingeniería del Software

Evolución y Gestión de la Configuración

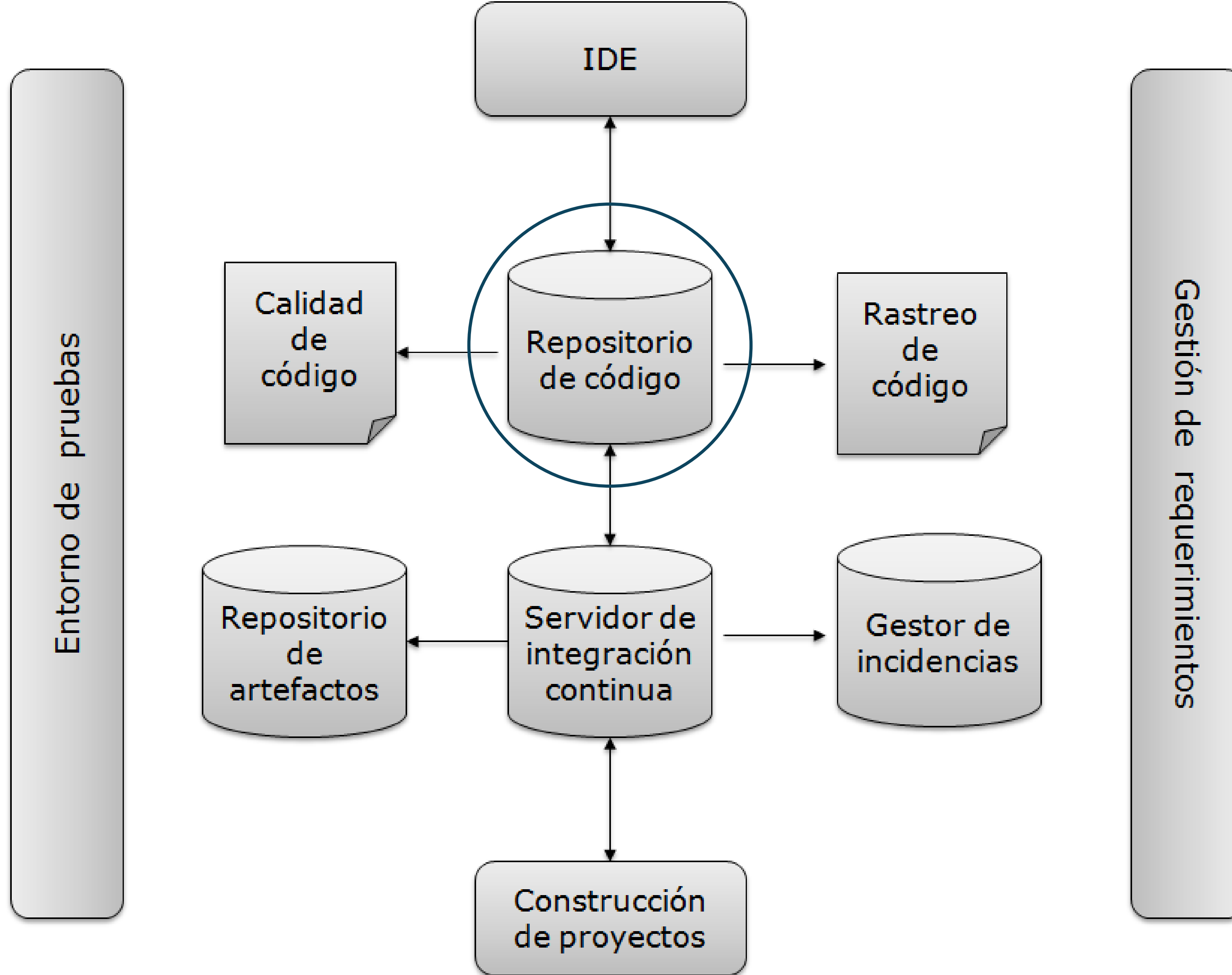
Escuela Técnica Superior de
Ingeniería Informática

Gestión del Código Fuente

Objetivo principal: saber organizar el código fuente y usar un repositorio de código de manera eficiente

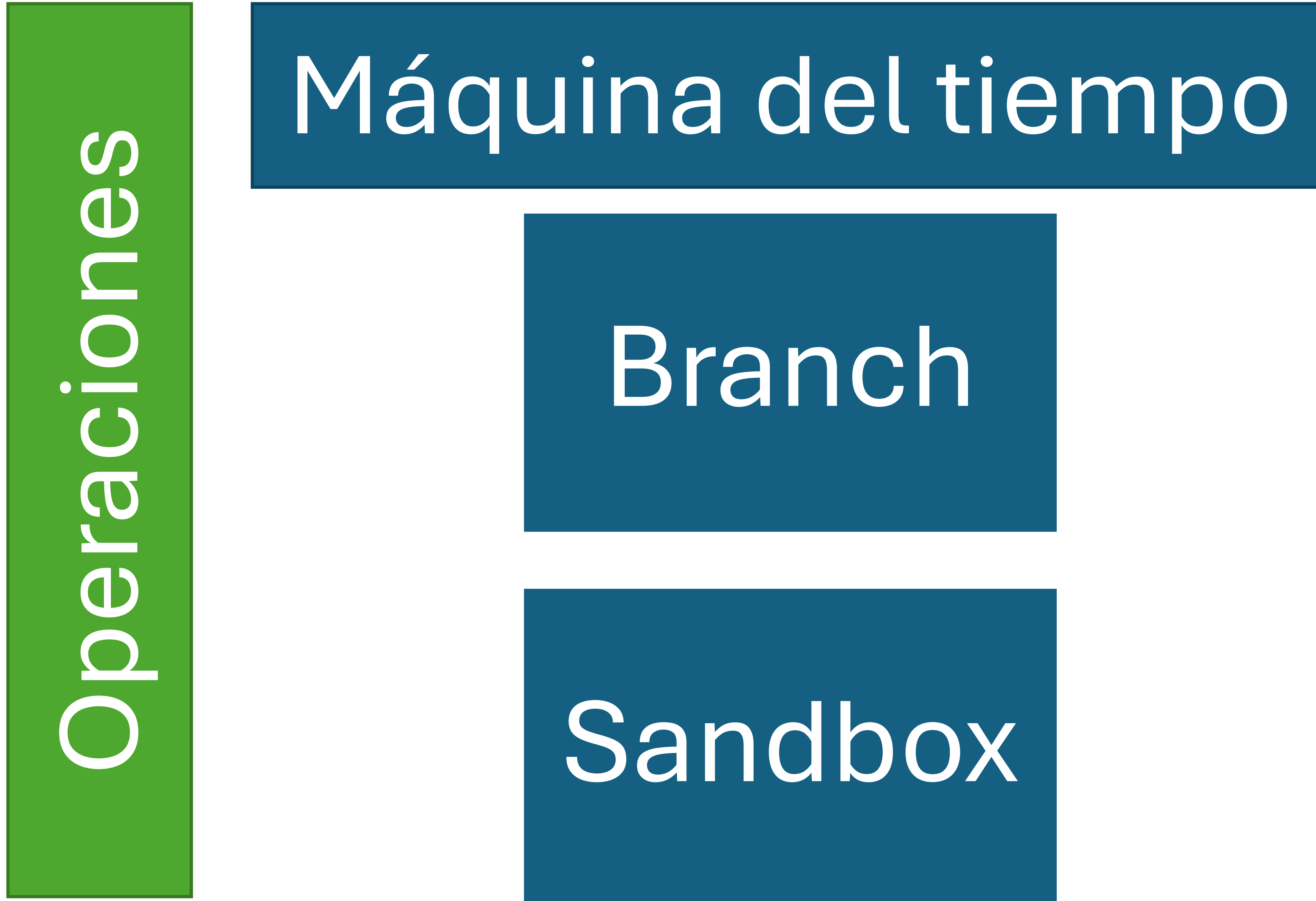

1. Conceptos básicos
2. Operaciones
3. Uso de repositorios
4. Resumen
5. Bibliografía

1. Conceptos básicos | ecosistemas de desarrollo



1. Conceptos básicos | máquina del tiempo

1. Conceptos básicos | ecosistemas de desarrollo



1. Conceptos básicos | repositorio de código

Un repositorio de código es un sistema para guardar múltiples versiones de los ficheros de un proyecto de modo que cuando se modifique un fichero se pueda acceder a las versiones anteriores.

- Repositorio de código = Sistema de control de versiones /control de código = sistema de control de revisiones = sistemas de gestión de código. = “el repositorio” = “el svn”, “el git”, “el mercurial”….

1. Conceptos básicos | git y servicios de git

¿Cuál es la diferencia entre una tecnología de repositorios de código (e.g. git) y un servicio de alojamiento de repositorios (e.g. gitlab)?

Git vs. GitHub

Git

First developed in 2005

Git is installed and maintained on your local system (rather than in the cloud)

One thing that really sets Git apart is its branching model

Git is a high quality version control system

GitHub

GitHub is designed as a Git repository hosting service

GitHub is exclusively cloud-based

You can share your code with others, giving them the power to make revisions or edits

GitHub is a cloud-based hosting service

Git	GitHub
Software	Service
Version control	Git repository hosting
Maintained by Linux	Maintained by Microsoft
Open-Source	Free or paid membership
No user management	Built-in user management
Locally installed	Hosted on the web
Minimal external tool configuration	Active marketplace for tool integration
Little to no competition	High competition

Sistemas Distribuidos

- ¿Qué es un sistema de control de versiones distribuido?
  - Es aquel en el que los usuarios mantienen un repositorio en local de modo que no existe un repositorio centralizado "*per se*"

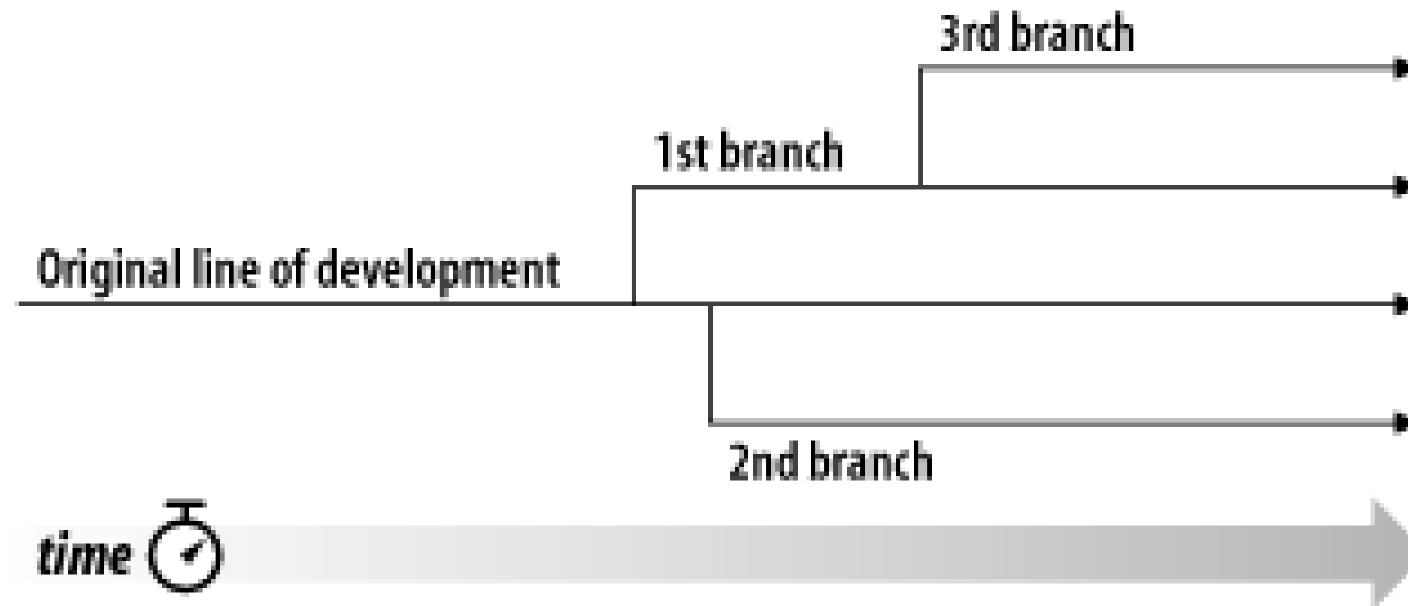
¿Por qué un sistema distribuido?

1. Conceptos básicos | repositorios distribuidos

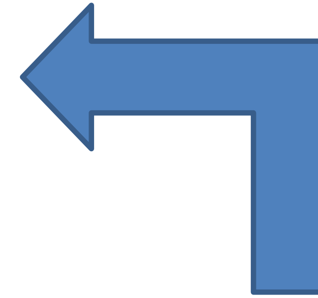
- ¿Cuáles son las principales diferencias?
  - No hay un repositorio central (aunque por convención se suele usar uno)
  - Cada usuario tiene su repositorio en local
  - Los *commits* son locales
  - Aparecen dos nuevas operaciones: ***pushing*** (especie de *commit* pero en remoto), ***pulling*** (especie de *update* del repo remoto).
  - Concepto de ***rebasing***: permite cambiar "la máquina del tiempo" del repositorio local para empaquetar los cambios en un sólo *commit* con objeto de enviarlo al repositorio remoto. Mucho cuidado con esta operación a menos que tengáis mucha destreza. Puede causar muchos problemas.

1. Conceptos básicos | branch

- **Branch**: una línea de desarrollo independiente que puede compartir parte de historia común con otras. Concepto de *branch* hija, *branch* madre



1. Conceptos básicos | sandbox



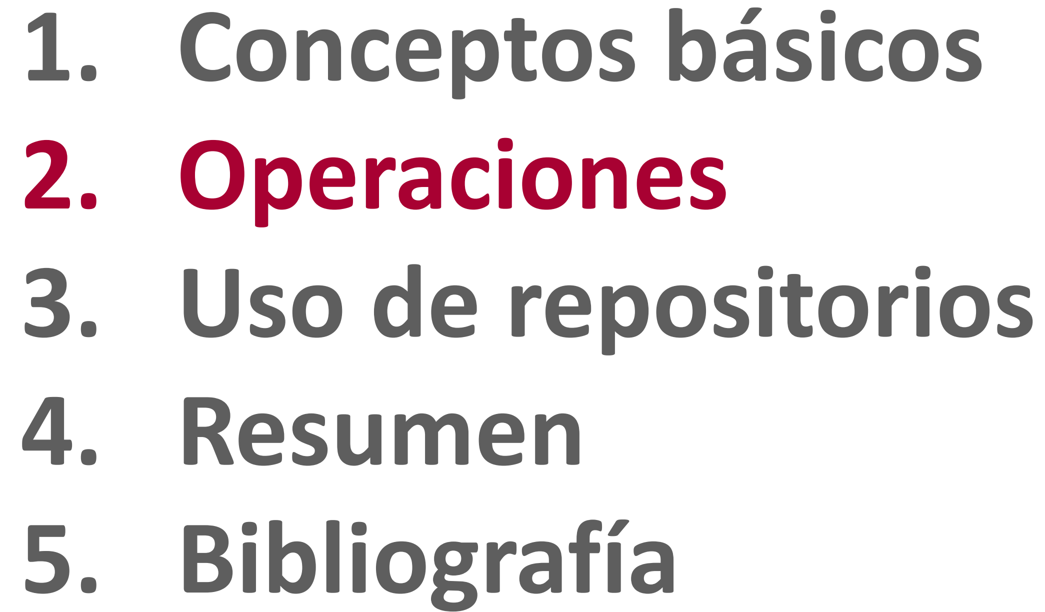
checkout

Pero…¿Sólo guardamos el código fuente? ¿qué más?

1. Conceptos básicos | contenido del repositorio

- Además del código fuente puedo guardar:
  - Pruebas, documentos, ficheros de configuración. También los binarios como imágenes, etc (discutir)
- Excepciones:
  - Los binarios
  - Los ficheros de configuración de entornos locales
  - Los archivos de log

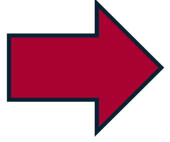
Hay que hacer uso de .gitignore


1. Conceptos básicos
2. Operaciones
3. Uso de repositorios
4. Resumen
5. Bibliografía

2. Operaciones

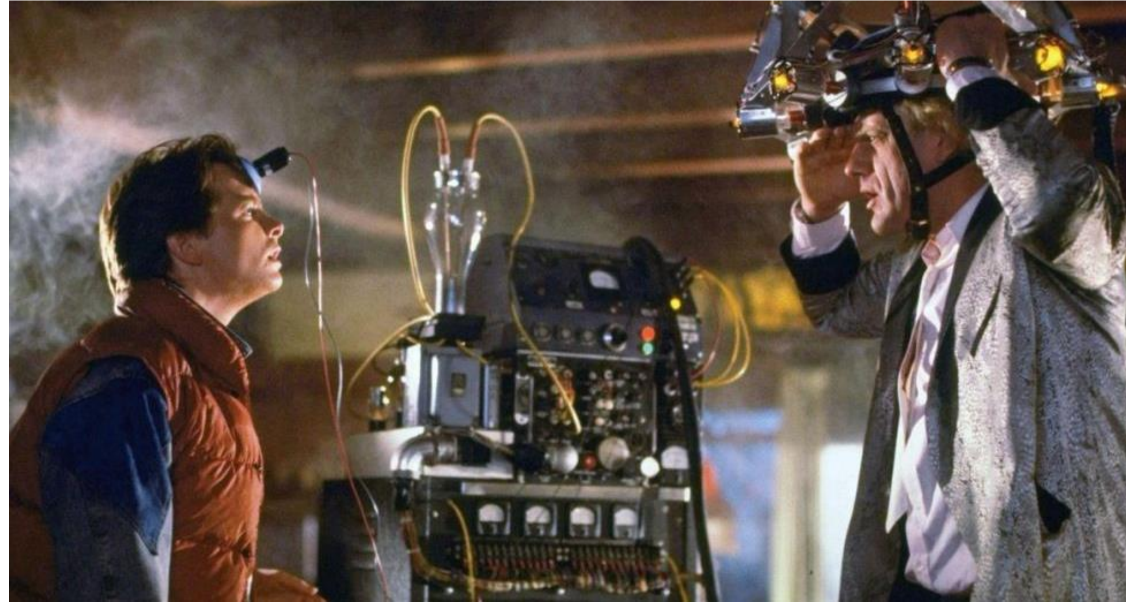
¿Cómo hacer las operaciones básicas en una "máquina del tiempo"?

2. Operaciones | tipos



- De creación
- De escritura
- De lectura
- De ramas

2. Operaciones | operaciones de creación



Clonando

“Forkeando”

Creando nuevo

En local

En remoto

2. Operaciones | operaciones de creación

Clonando

```
1 $> git clone https://github.com/EGCETSII/decide.git
```

"Forkeando"

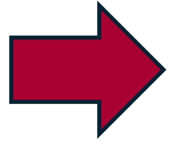
Nuevo

Cada miembro del equipo de hacer commit con un solo nombre de usuario

```
$> git config --global user.name "Your Name"
$> git config --global user.email you@example.com
$> git init
```

2. Operaciones | tipos

De creación

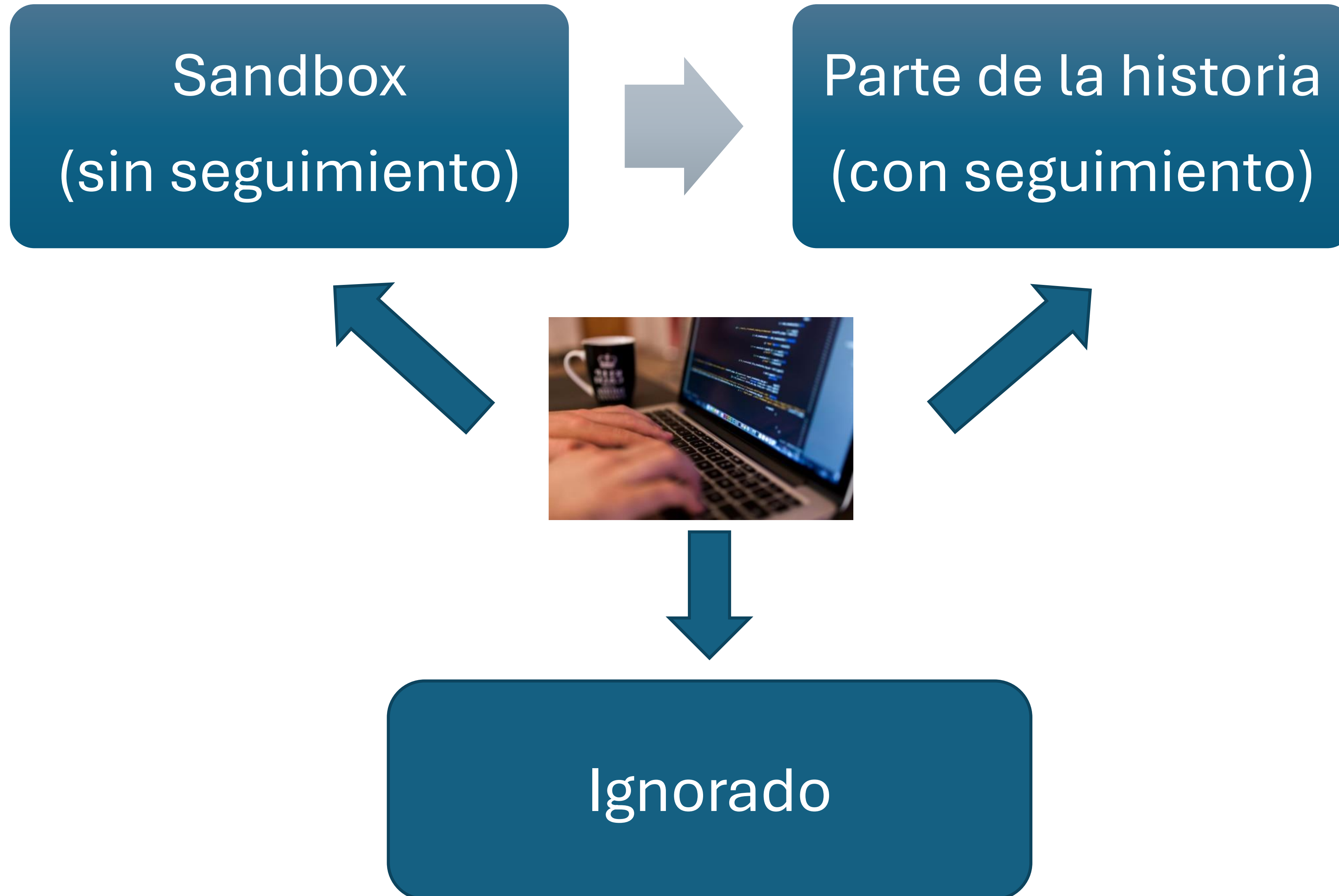


De escritura

De lectura

De ramas

2. Operaciones | operaciones de escritura



2. Operaciones | ejercicio

Imagina que has hecho un *commit* de un archivo (debug.log) sobre el que ya no quieres seguir haciendo seguimiento ¿Qué harías?

2. Operaciones | ejercicio

- Imagina que has hecho un commit de un archivo (debug.log) sobre el que ya no quieres seguir haciendo seguimiento ¿Qué harías?

```
echo debug.log >> .gitignore
git rm --cached debug.log
#rm 'debug.log'
git commit -m "Start ignoring debug.log"
```

1. Añadir debug.log al fichero .gitignore
2. Elimina debug.log del repositorio pero permanece en el directorio de trabajo como fichero ignorado (--cached)
3. Confirmar

2. Operaciones | ejercicio

Imagina que tienes un patrón para ignorar un conjunto de archivos (ej. *.log), pero hay uno determinado del que quieres empezar a hacer seguimiento (ej. debug.log) ¿Cómo lo harías?

2. Operaciones | ejercicio

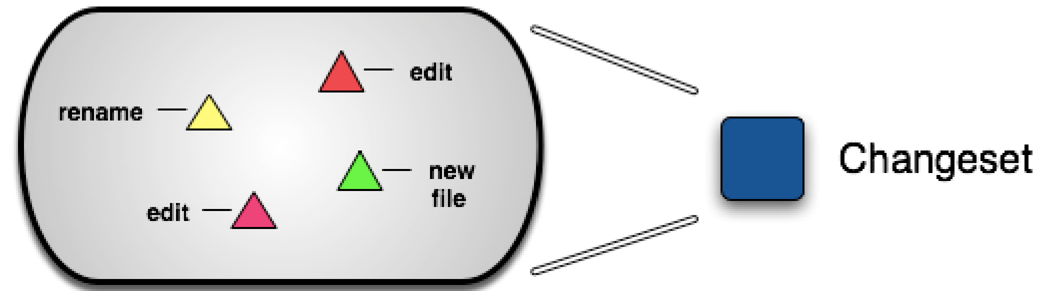
- Imagina que tienes un patrón para ignorar un conjunto de archivos (ej. *.log), pero hay uno determinado del que quieres empezar a hacer seguimiento (ej. debug.log) ¿Cómo lo harías?

```
$ echo !debug.log >> .gitignore

$ cat .gitignore
*.log
!debug.log

$ git add debug.log

$ git commit -m "Adding debug.log"
```

1. Añadir ! para que se haga la excepción (patrón de anulación)
2. Verificar contenido del .gitignore
3. Añadir debug.log al seguimiento
4. Confirmar

2. Operaciones | conjunto de cambios

Changeset: conjunto de cambios que deben ser tratados como un conjunto indivisible (en svn se conoce como "revision", en git como "commit").

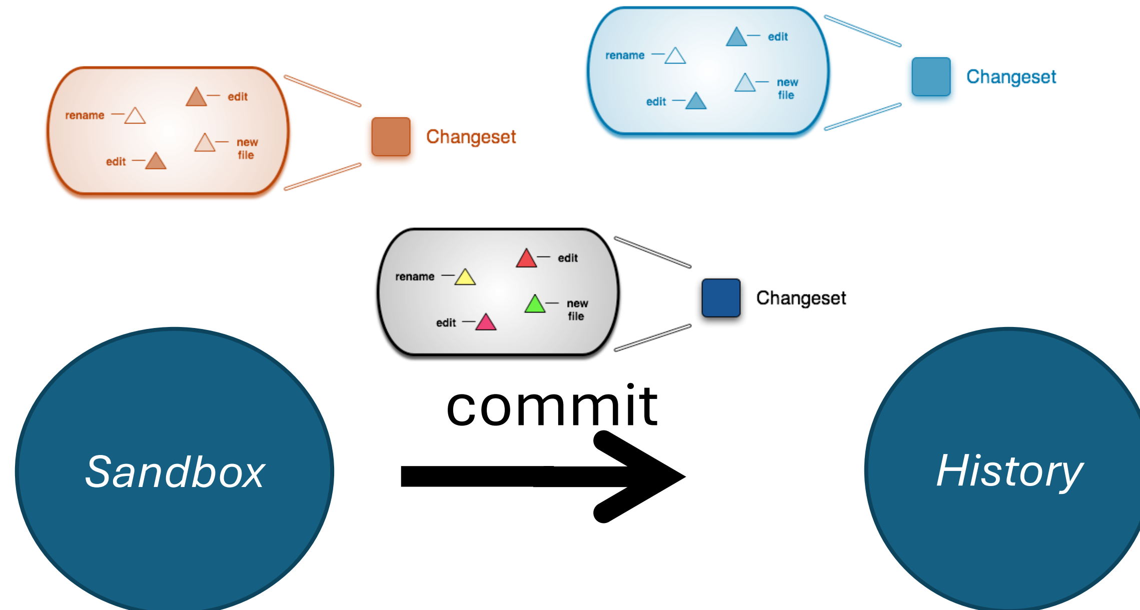


OJO: en git "commit" se usa tanto para una operación como para nombrar al conjunto de cambios

2. Operaciones | de escritura

- **Add**: Añadir un elemento a la "máquina del tiempo" de modo que su estado y versiones sean controlados por la misma (con seguimiento)
- **Commit**: Guardar en la máquina del tiempo un conjunto de "changesets". Pueden ser atómicos o no.

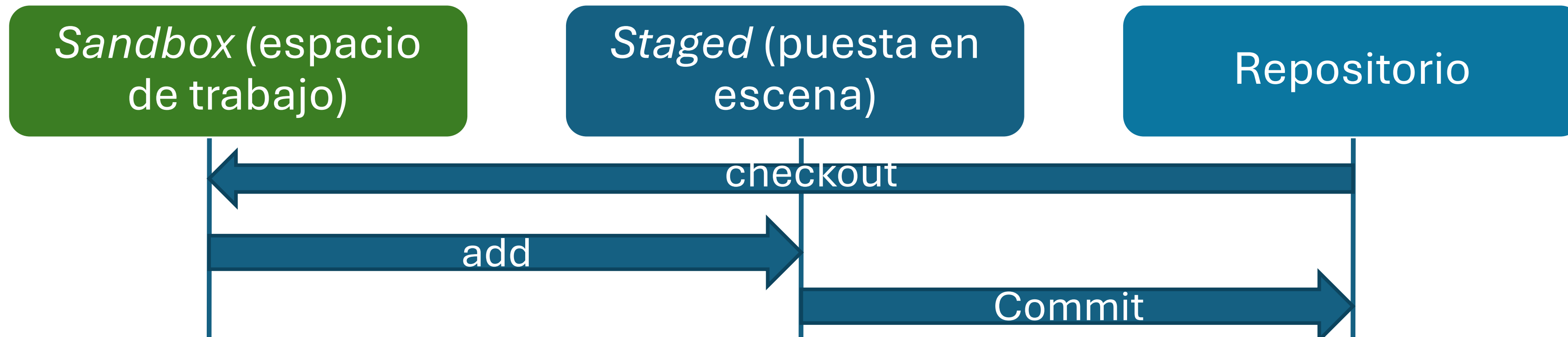
Lo ideal es tener *commits* atómicos



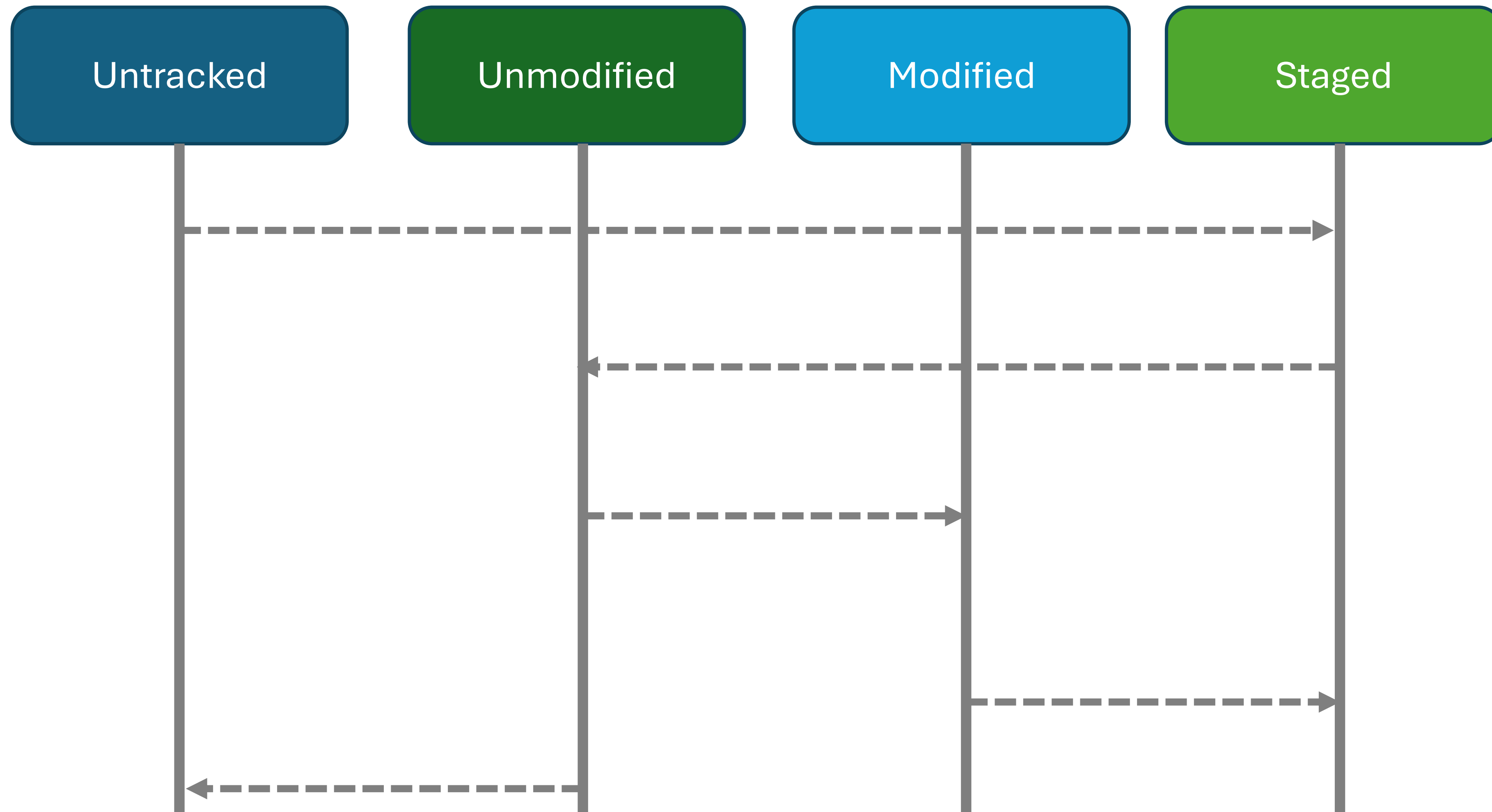
2. Operaciones | de escritura

¿Qué hay en un *commit*?

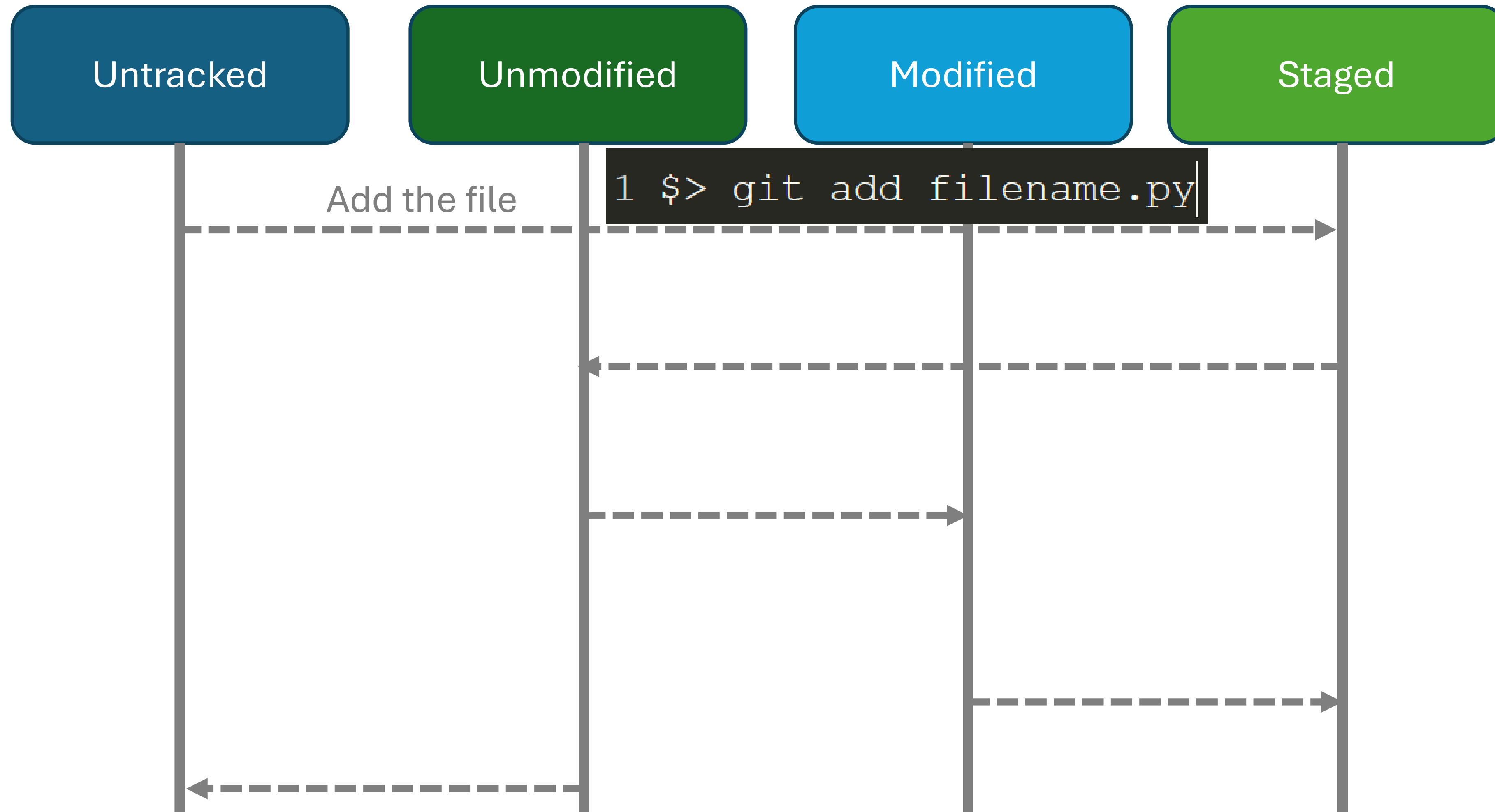
- Datos modificados + metadatos
- Un fichero *rastreado* (con seguimiento) puede estar en tres estados (en el caso de GIT):
  - Confirmado (*committed*)
  - Modificado (*modified*)
  - Preparado (*staged*)



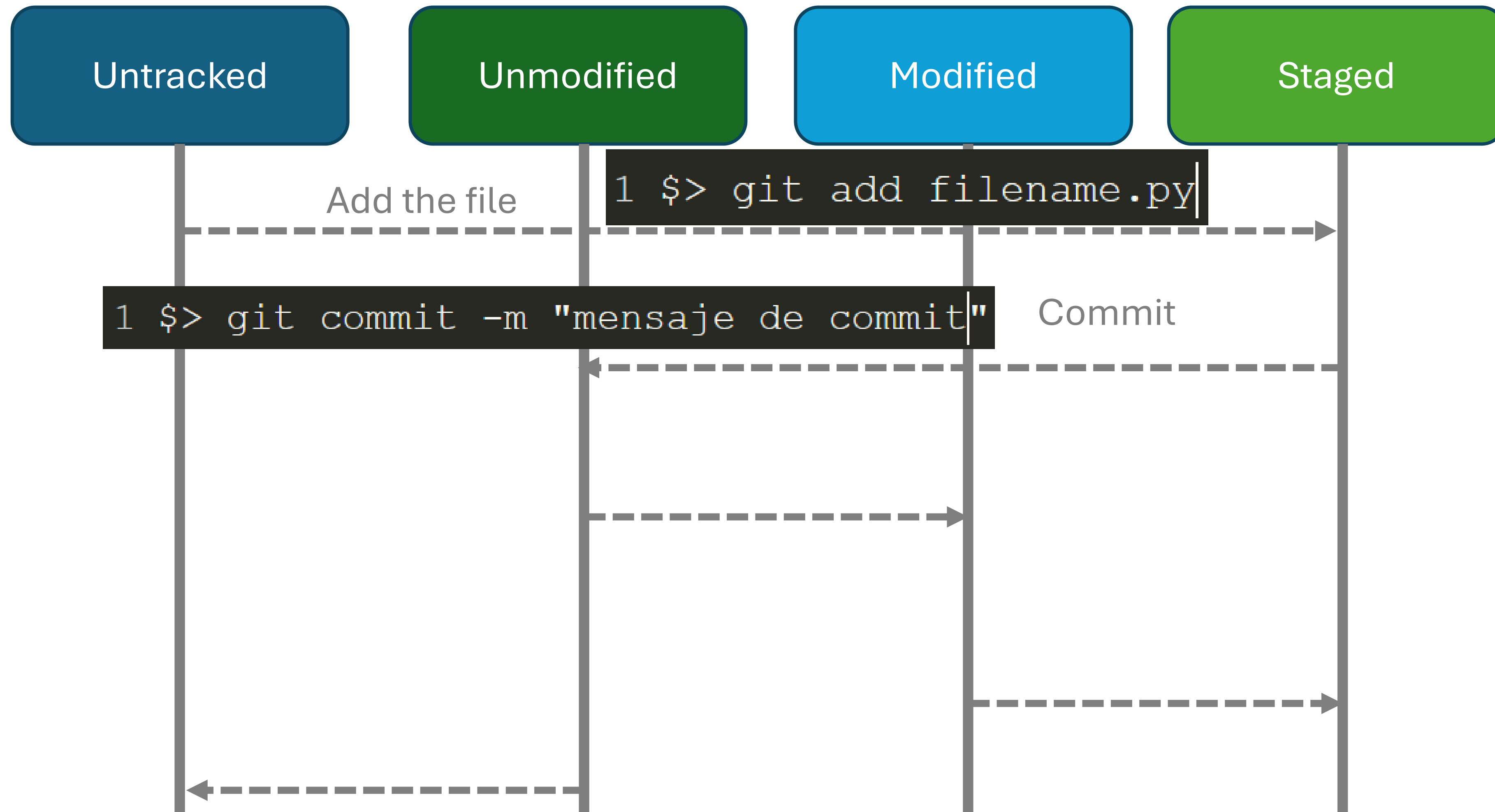
2. Operaciones | flujo frecuente



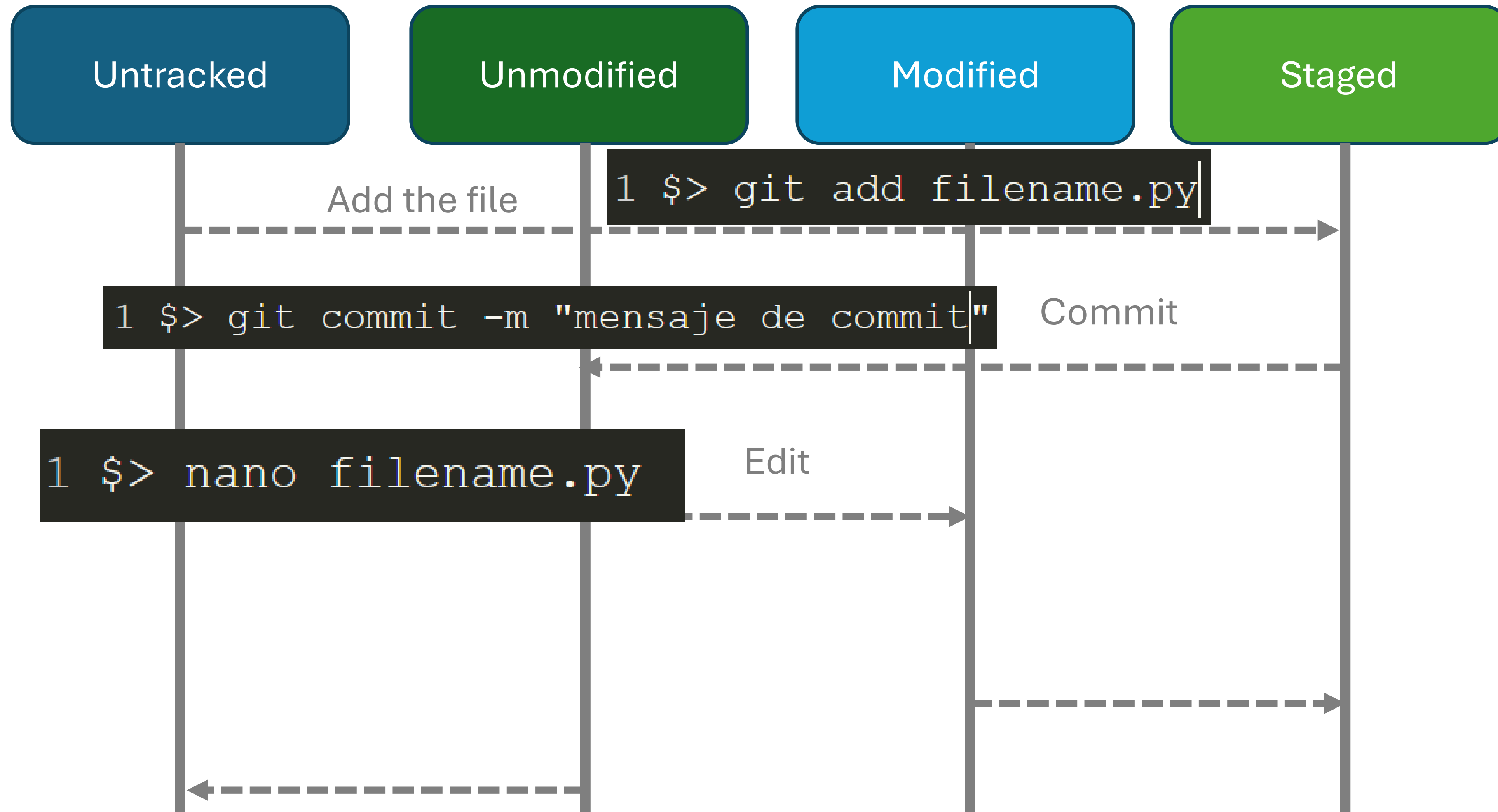
2. Operaciones | flujo frecuente



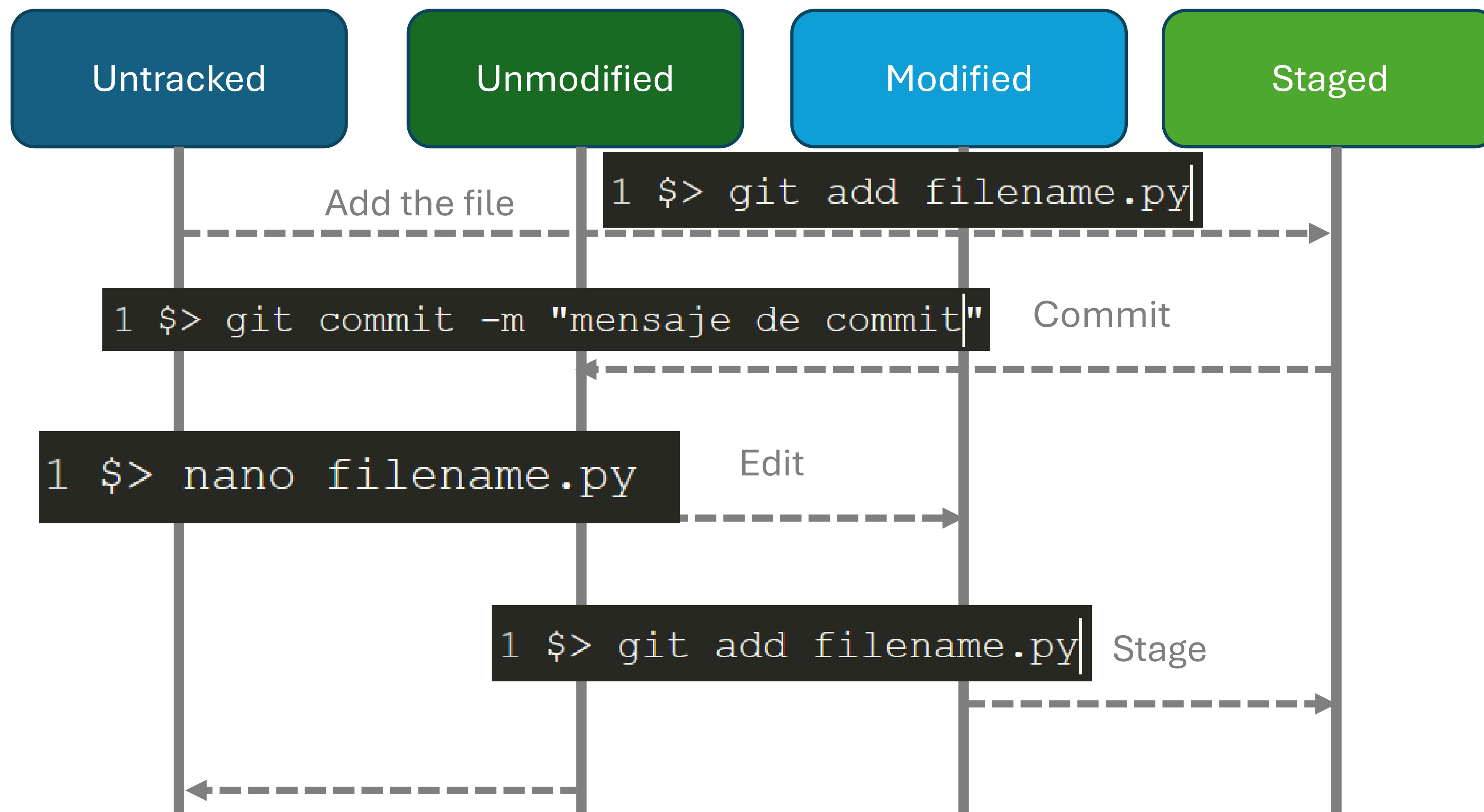
2. Operaciones | flujo frecuente



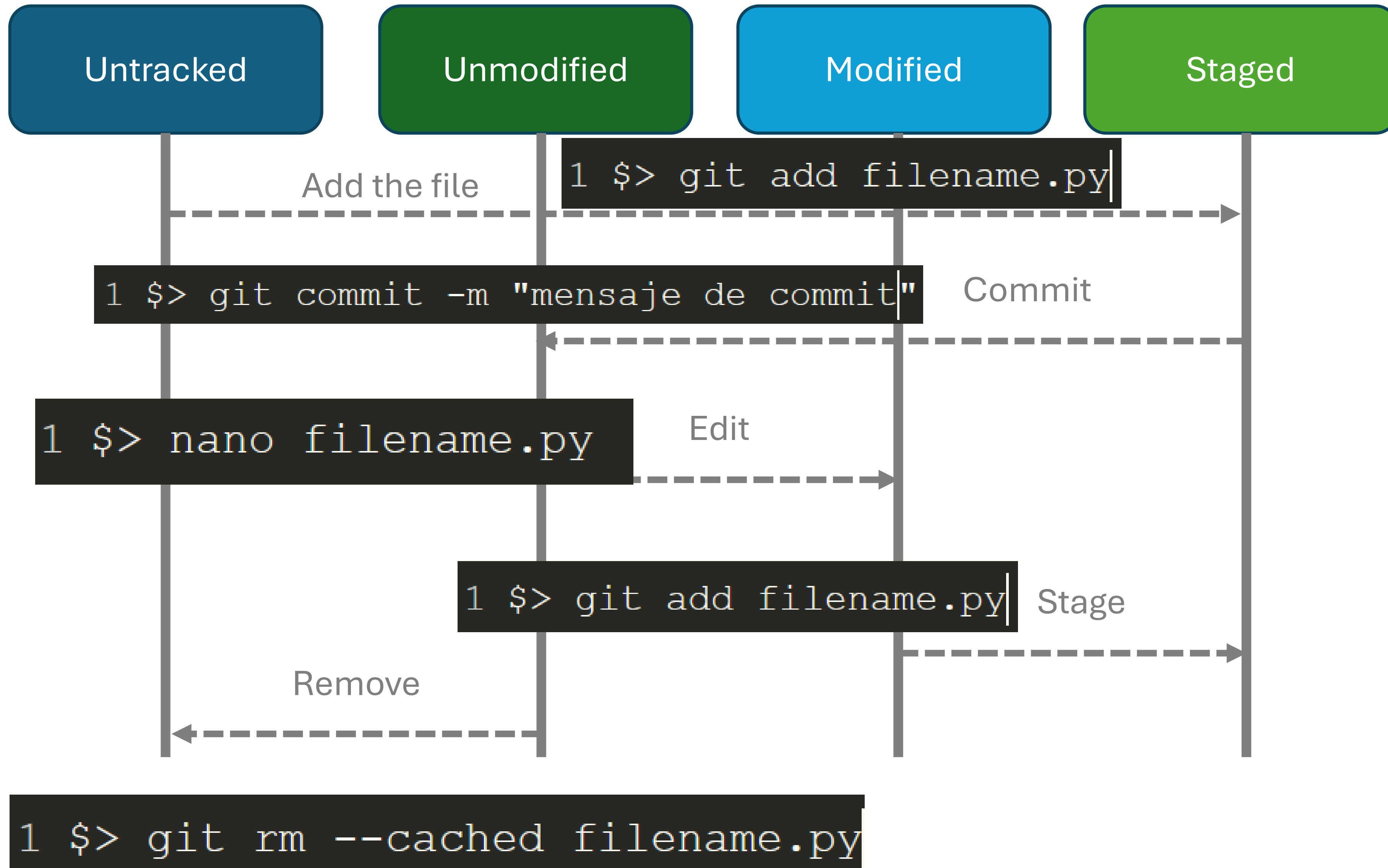
2. Operaciones | flujo frecuente



2. Operaciones | flujo frecuente



2. Operaciones | flujo frecuente



2. Operaciones | commits atómicos

¿Por qué lo ideal es tener *commits* atómicos?

2. Operaciones | commits atómicos

Tengo varios cambios hechos que afectan a varios ficheros. Si un cambio afecta a un conjunto de ficheros y otro cambio afecta a otro conjunto de ficheros distinto ¿cómo hago para hacer un commit atómico?

```
$ git status
 M templates/profile/summary.html
 M services.py
```

2. Operaciones | commits atómicos

```
$ git status
 M templates/profile/summary.html
 M services.py

$ git add templates/profile/summary.html

$ git status
 A templates/profile/summary.html
 M services.py

$ git commit -m "Add div HTML elements"
```

2. Operaciones | commits atómicos

Ahora tengo varios cambios hechos sobre el mismo fichero pero que afectan a partes distintas, ¿puedo hacer commits atómicos?

```
$ git status
M services.py
```

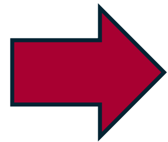
2. Operaciones | commits atómicos

```
$ git status
M services.py
$ git add -p

…

$ git status

¿qué más?
```

2. Operaciones | tipos

De creación

De escritura


De lectura

De ramas

2. Operaciones | de lectura

- **<u>Checkout</u>**: tomar los artefactos del repositorio y llevarlo al área de trabajo (*sandbox*) para realizar modificaciones.

checkout

OJO: puede haber "juguetes" nuevos

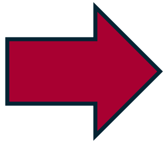
Recordad tener clara la respuesta a esta pregunta:
¿tengo que meter los ficheros de configuración de mi sandbox en el área de trabajo?

2. Operaciones | tipos

De creación

De escritura

De lectura



De ramas

2. Operaciones | de ramas

- **Merging:** Es el proceso de mezclar dos o más ramas de artefactos.

https://learngitbranching.js.org/?locale=es_ES

2. Operaciones | de ramas - merge

El ciclo típico de mezclar ramas es:

- Me pongo en la rama sobre la que quiero mezclar los cambios
- Mezclar los cambios sobre la rama actual desde la rama origen (*source*)
- Resolver conflictos
- Hacer commit del merge
- Abortar el merge

```
$ git checkout main


$ git merge feature_x


…


$ git commit -m "Resolve merge conflicts"
$ git merge --abort
```

¿Se nos olvida algo?

2. Operaciones | de ramas - merge

El ciclo típico de mezclar ramas es:

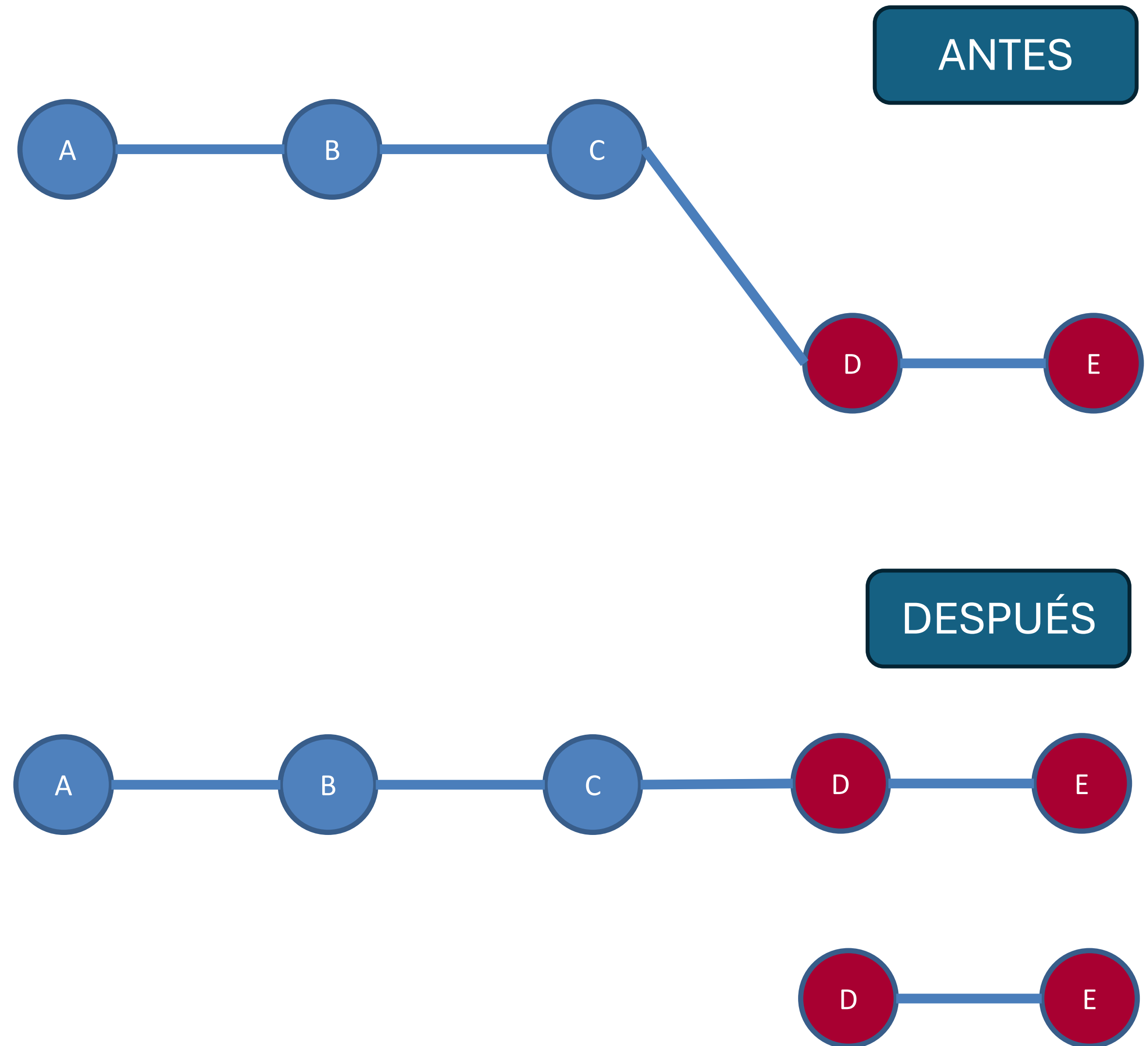
- Me pongo en la rama sobre la que quiero mezclar los cambios
- **Me aseguro que la rama no está desactualizada**
- Mezclar los cambios sobre la rama actual desde la rama origen (*source*)
- Resolver conflictos
- Hacer commit del merge
- Abortar el merge

```
$ git checkout main

$ git pull origin main

$ git merge feature_x

...

$ git commit -m "Resolve merge conflicts"
$ git merge --abort
```

2. Operaciones | de ramas – tipos de merge

Fastforward

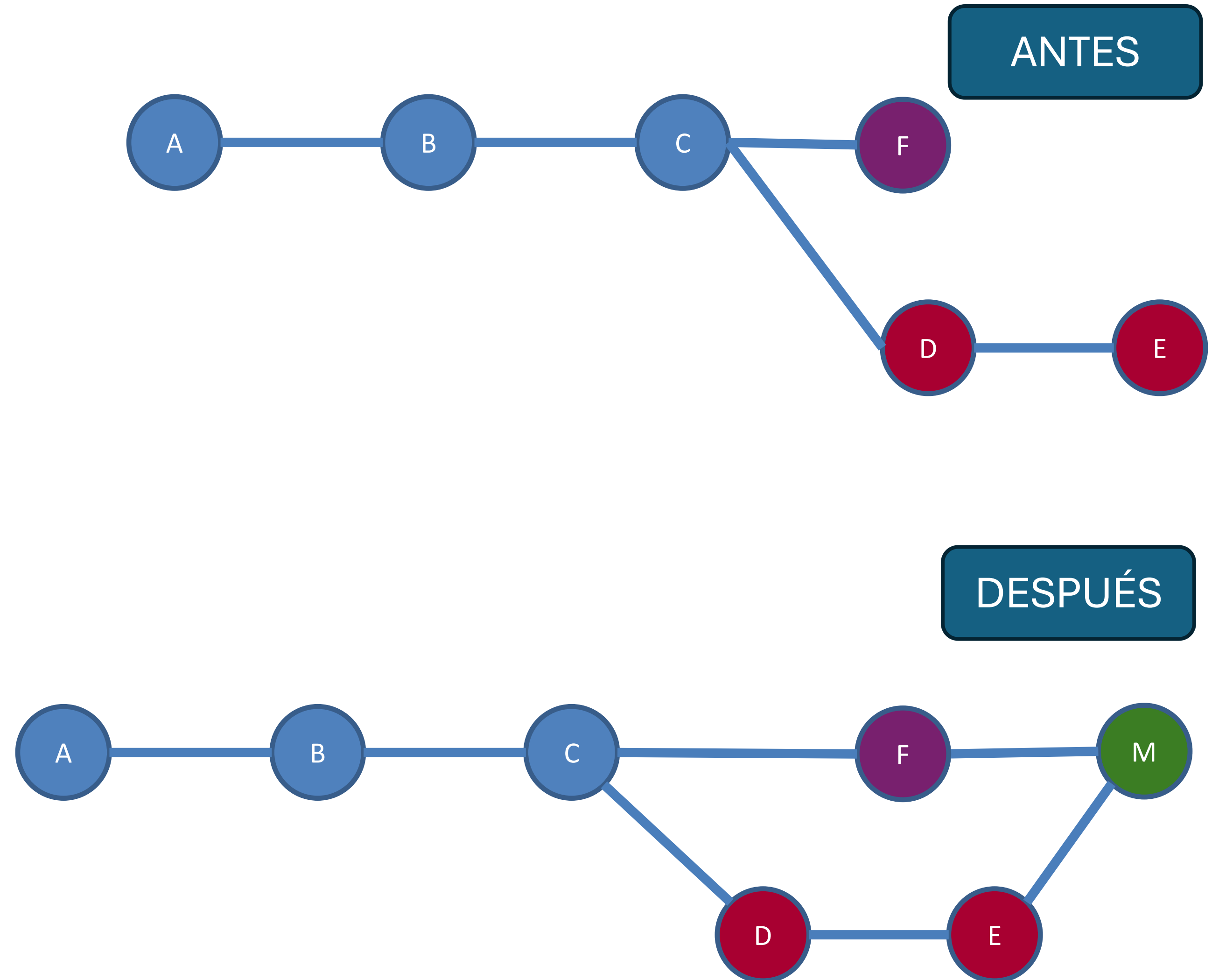
- Cuando la rama destino no ha tenido *commits*
- Git no hace un *commit* de *merge*
- La historia permanece lineal
- Es como si se hubiese hecho *commits* sobre la rama destino



2. Operaciones | de ramas – tipos de merge

Recursive (3-way)

- Cuando la rama origen y destino han tenido *commits*
- Git hace un *commit* de *merge*
- La historia no es lineal



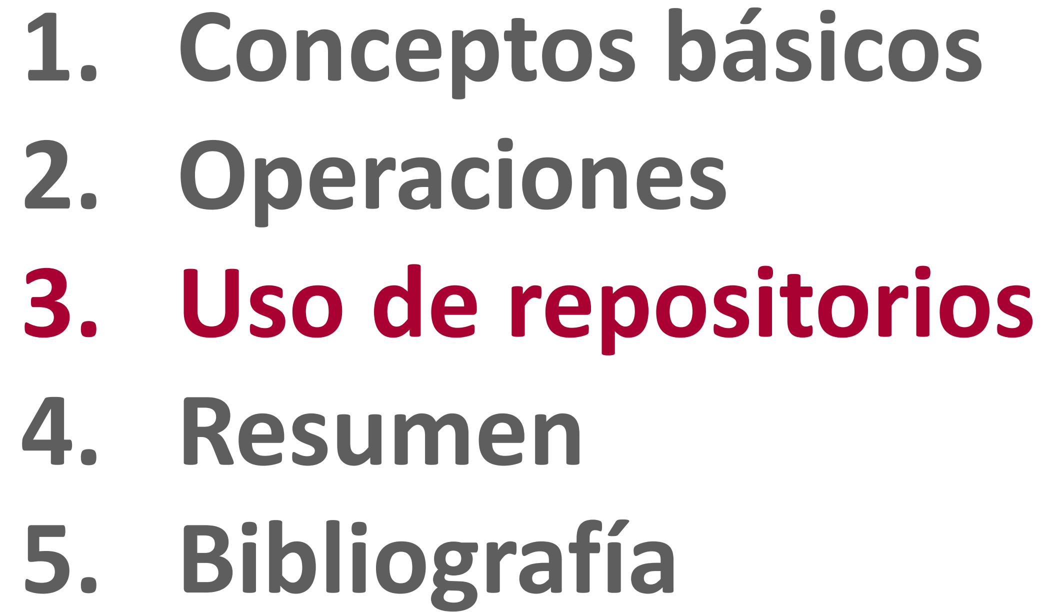
2. Operaciones | conflictos

- **Resolución de conflictos**. Cuando se hace *merge* pueden aparecer conflictos. Algunos pueden resolverse de manera automática, otros requieren intervención manual. ***Resolver conflictos es duro***.

- ***Diff***. Hacer *diff* significa mirar la diferencia que hay entre artefactos de un repositorio.

- A partir de un *diff* se puede generar un ***patch***, es decir, un conjunto de cambios a realizar a un conjunto de artefactos. Cuando un conjunto de cambios se aplica a una serie de artefactos, se dice que se ha aplicado un *patch*.
  (ver http://es.wikipedia.org/wiki/Diff probar con https://www.diffchecker.com/)

Conflictos *sintácticos* vs conflictos *semánticos*


1. Conceptos básicos
2. Operaciones
3. Uso de repositorios
4. Resumen
5. Bibliografía

3. Uso de repositorios | modelo de uso

Definir el "*usage model*" o modo de uso de la/s herramienta/s:

¿Cómo se gestionan las ramas? ¿qué permisos se dan a los usuarios? ¿cómo se hacen los *merge*? ¿con qué frecuencia? ¿por qué motivos se crean las ramas? ¿cuándo se eliminan?, etc, etc, etc...

¿Por qué crear ramas?

3. Uso de repositorios | modelo de uso

- **Físicos**: Se crean ramas según la distribución “física” del proyecto, es decir, de su arquitectura.
- **Funcionales**: Según aspectos funcionales como pueden ser características (*features*), corrección de defectos (*bugfixing*)
- **Entorno**: por ejemplo, distintas versiones del sistema operativo / plataforma.
- **Organizacionales**: Para organizar el trabajo en equipos, tareas, subproyectos, ...
- **Procedimentales**: Para pasar por distintas etapas de los proyectos.

3. Uso de repositorios | commits

Es necesario establecer una guía de cómo y cuándo hacer *commit*

¿Para qué?

3. Uso de repositorios | commits

- Al establecer una guía de cómo y cuándo hacer *commit:*
  - Ordenamos y sistematizamos el trabajo
  - Permitimos generar *changelogs* automáticamente
  - Facilitamos la visualización y navegación de la historia del repositorio.
- Hay diversas guías en las que se puede basar el equipo para desarrollar la suya propia

Format of the commit message

```
<type>(<scope>): <subject>
<BLANK LINE>
<body>
<BLANK LINE>
<footer>
```

http://karma-runner.github.io/0.10/dev/git-commit-msg.html

3. Uso de repositorios | commits

¿Se puede asistir en la gestión de plantillas de *commit*?

3. Uso de repositorios | commits

2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)

What Makes a Good Commit Message?

Yingchen Tian*
Beijing Institute of Technology
Beijing, China
tianyc10@foxmail.com

Yuxia Zhang*†
Beijing Institute of Technology
Beijing, China
yuxiazh@bit.edu.cn

Klaas-Jan Stol
University College Cork and Lero
School of Computer Science and IT
Cork, Ireland
k.stol@ucc.ie

Lin Jiang
Beijing Institute of Technology
Beijing, China
jianglin17@bit.edu.cn

Hui Liu†
Beijing Institute of Technology
Beijing, China
liuhui08@bit.edu.cn

ABSTRACT

A key issue in collaborative software development is communication among developers. One modality of communication is a commit message, in which developers describe the changes they make in a repository. As such, commit messages serve as an "audit trail" by which developers can understand how the source code of a project has changed—and why. Hence, the quality of commit messages affects the effectiveness of communication among developers. Commit messages are often of poor quality as devel-

KEYWORDS

Commit-based software development, open collaboration, commit message quality

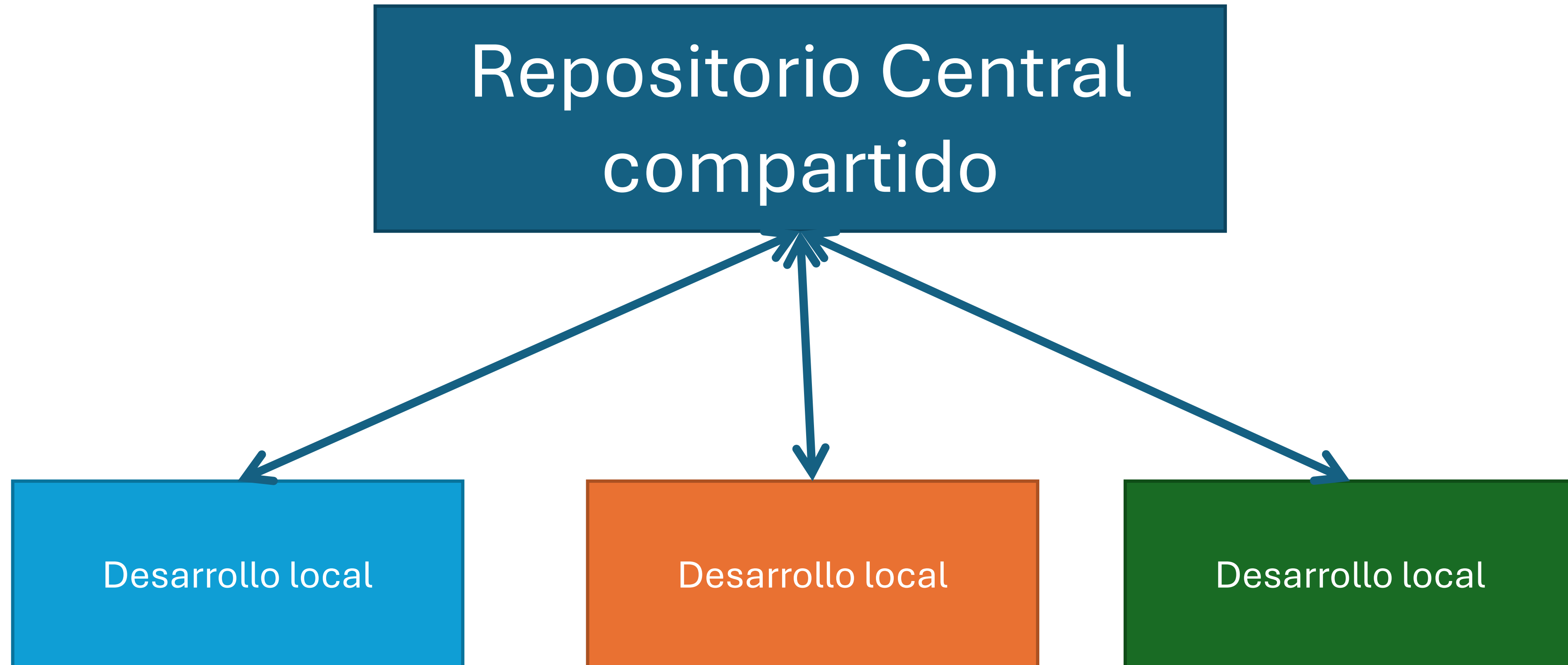
ACM Reference Format:
Yingchen Tian, Yuxia Zhang, Klaas-Jan Stol, Lin Jiang, and Hui Liu. 2022. What Makes a Good Commit Message?. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510205

https://doi.org/10.1145/3510003.3510205

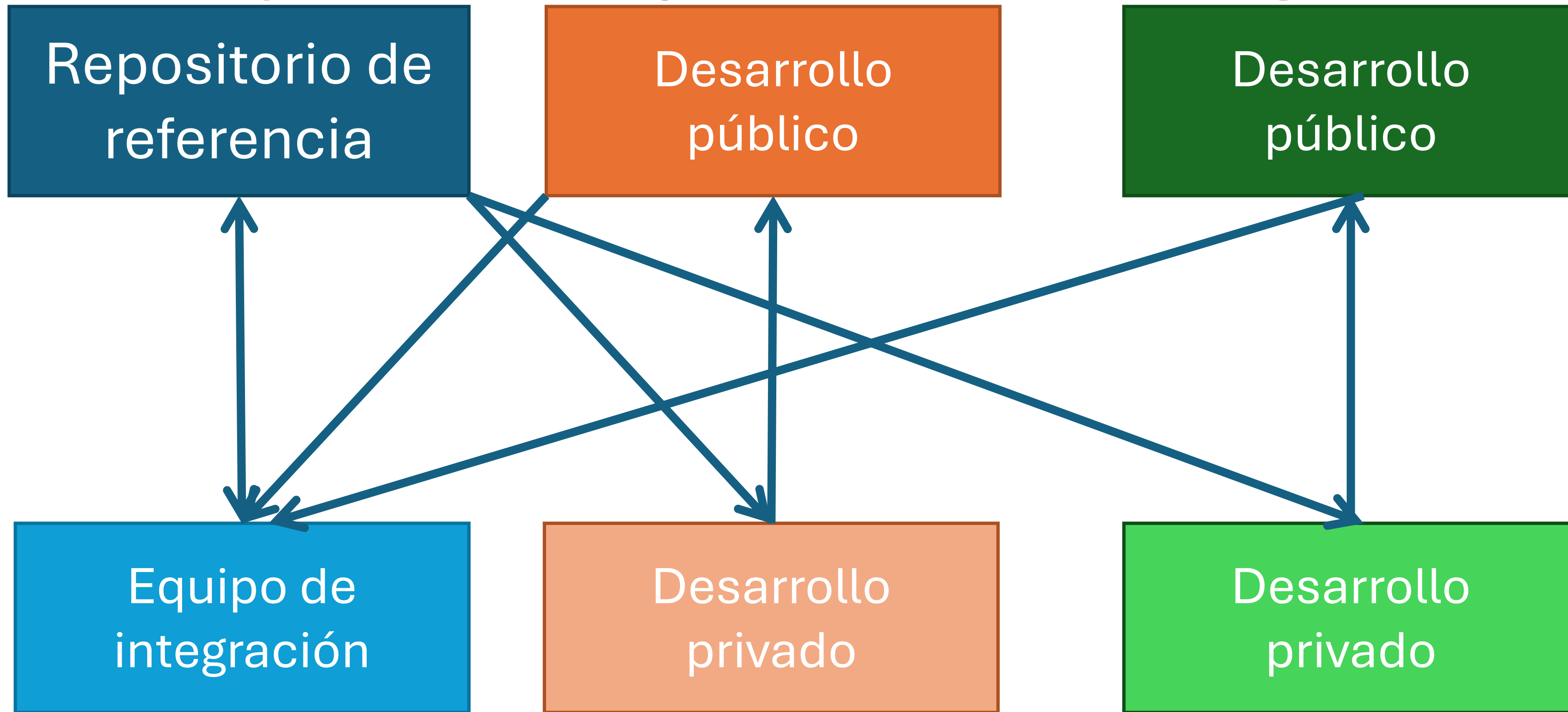
3. Uso de repositorios | principios básicos

- Todo lo que hay en *main* debe poder ser ejecutado/desplegado
- Los *commits* deben ser gestionados correctamente siguiendo unas guías
- Debe haber una relación entre *commits* y gestión de cambio/incidencias/*issues*
- Se debe tener planificado y ser consciente de cómo afecta el modelo de uso a las etapas de gestión de la construcción e integración continua

3. Uso de repositorios | modelo centralizado

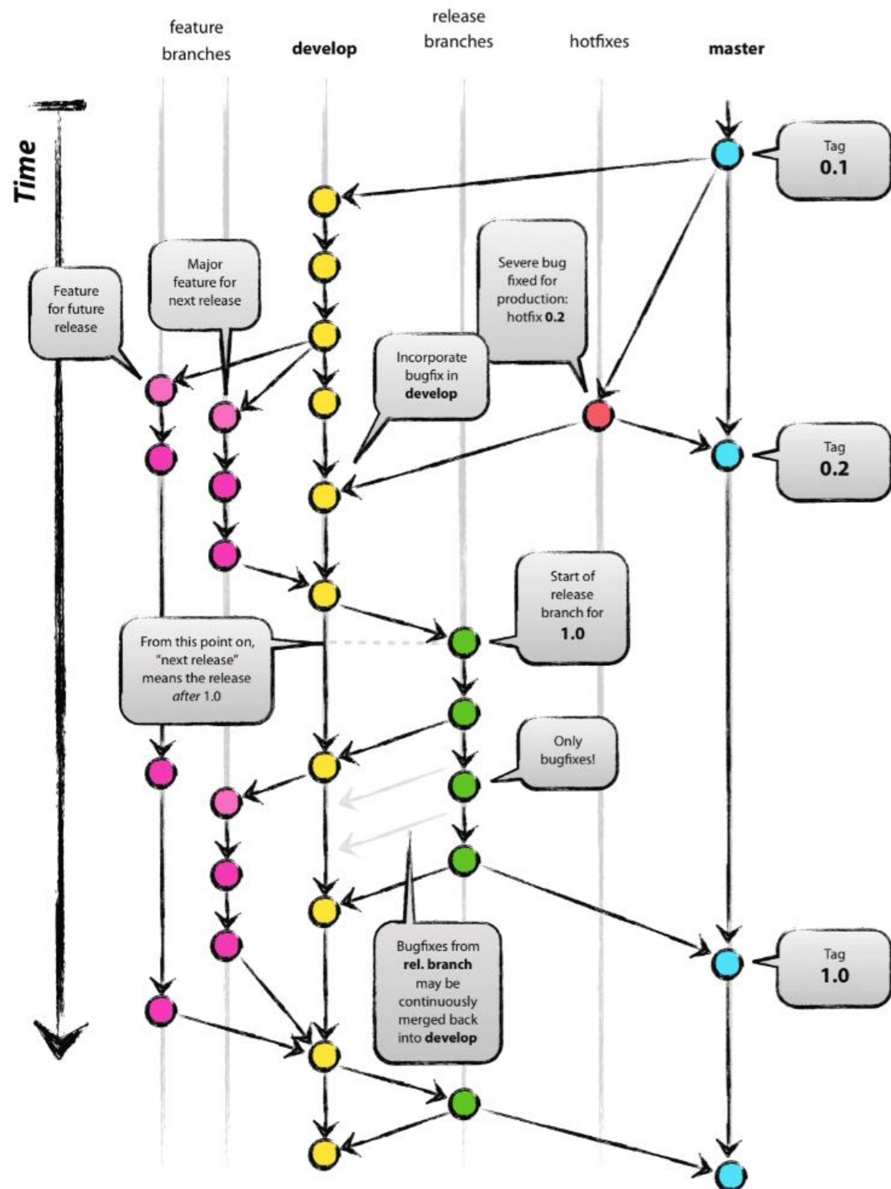


3. Uso de repositorios | modelo de integración



1. Existe un repositorio público de referencia
2. Los desarrolladores/as crean una copia privada para proponer cambios
3. Las contribuciones se hacen públicas en un repositorio
4. Se le envía una propuesta de integración al equipo de integración
5. El equipo de integración verifica que los cambios sean correctos en local
6. Acepta los cambios que sean necesarios y los hacen públicos en el repositorio de referencia

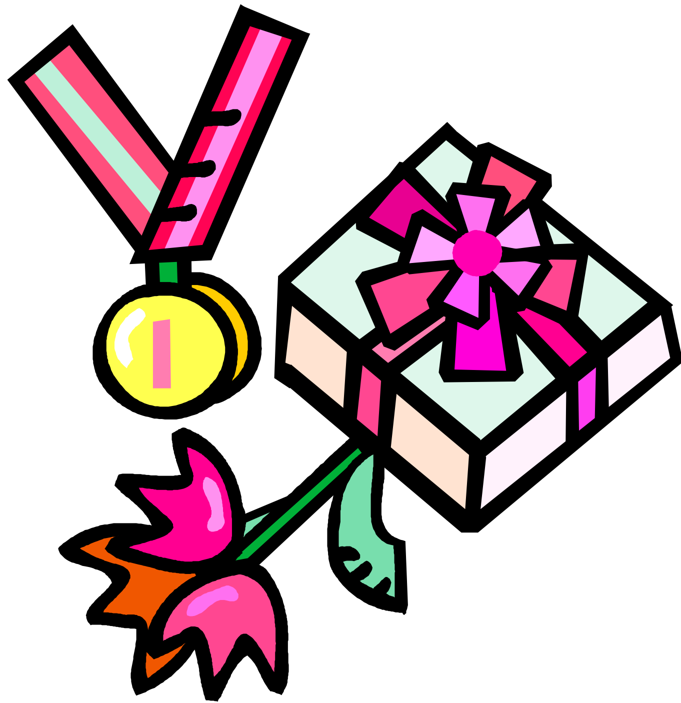
3. Uso de repositorios | gitflow

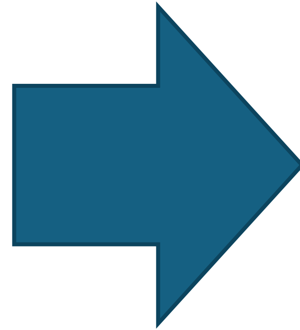


NO usar git flow

http://nvie.com/posts/a-successful-git-branching-model/

3. Uso de repositorios | gitflow





Note of reflection (March 5, 2020)

https://nvie.com/posts/a-successful-git-branching-model/

This model was conceived in 2010, now more than 10 years ago, and not very long after Git itself came into being. In those 10 years, git-flow (the branching model laid out in this article) has become hugely popular in many a software team to the point where people have started treating it like a standard of sorts — but unfortunately also as a dogma or panacea.

During those 10 years, Git itself has taken the world by a storm, and the most popular type of software that is being developed with Git is shifting more towards web apps — at least in my filter bubble. Web apps are typically continuously delivered, not rolled back, and you don't have to support multiple versions of the software running in the wild.

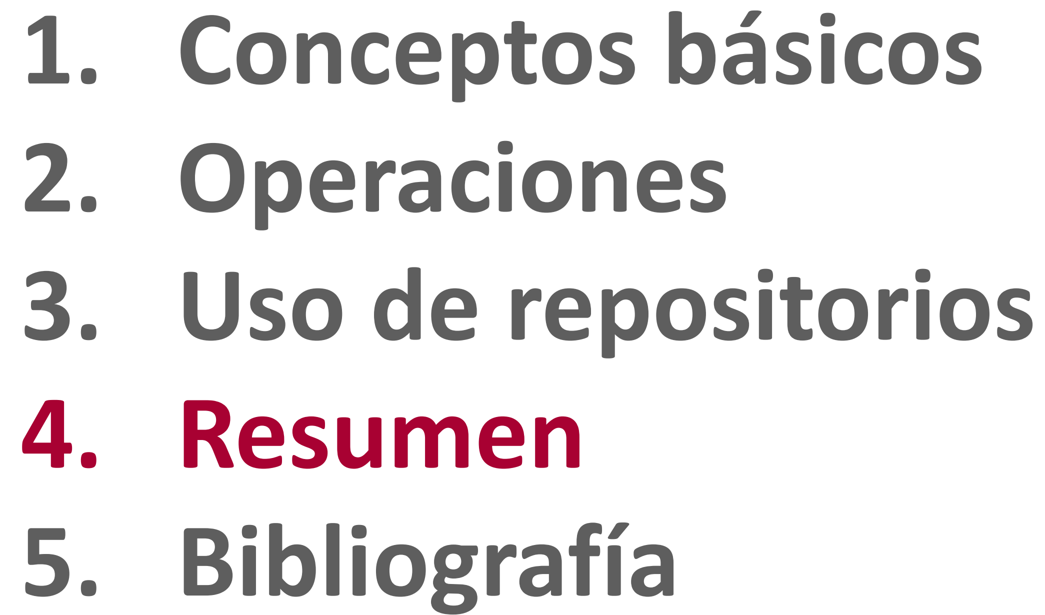
This is not the class of software that I had in mind when I wrote the blog post 10 years ago. If your team is doing continuous delivery of software, I would suggest to adopt a much simpler workflow (like GitHub flow) instead of trying to shoehorn git-flow into your team.

If, however, you are building software that is explicitly versioned, or if you need to support multiple versions of your software in the wild, then git-flow may still be as good of a fit to your team as it has been to people in the last 10 years. In that case, please read on.

To conclude, always remember that panaceas don't exist. Consider your own context. Don't be hating. Decide for yourself.

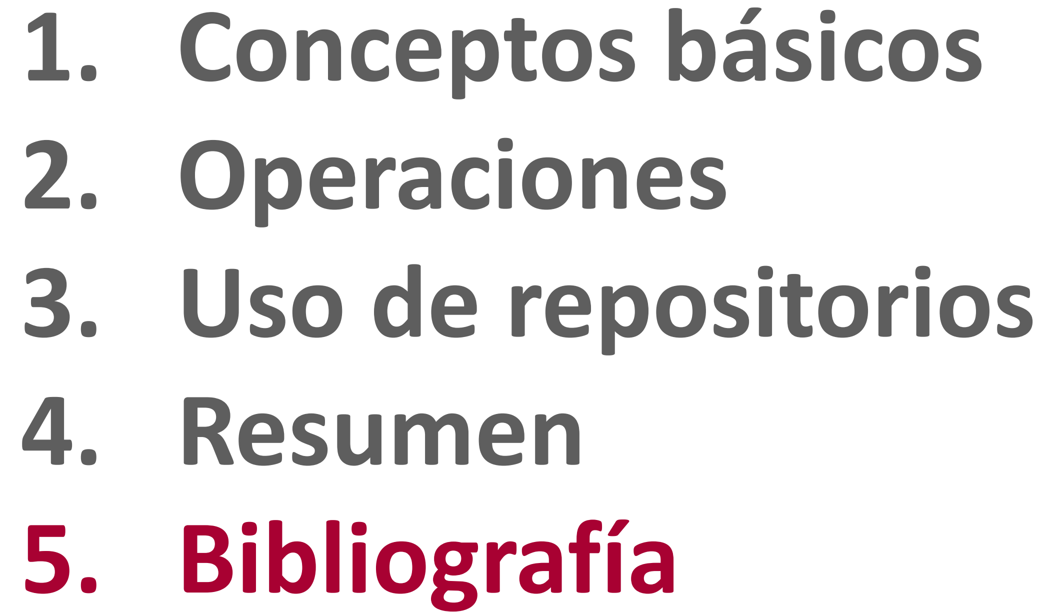
3. Uso de repositorios | ramas

- Usar un modelo de ramas por features => al menos una rama *main* y una o varias de *features*
- No usar un modelo de ramas por personas (a menos que esté realmente justificado)
- Las ramas que se dejan usar, se destruyen. Puede haber ramas que no se usen que sigan, pero debe haber una justificación para ello.
- Tener una rama por entorno de despliegue que se tenga: *production* (*main*), *preproduction*
- Hacer *merge* frecuentes entre las ramas (principio de integración continua). De nada sirve tener un servidor de CI si luego no se hacen *merge* periódicos.

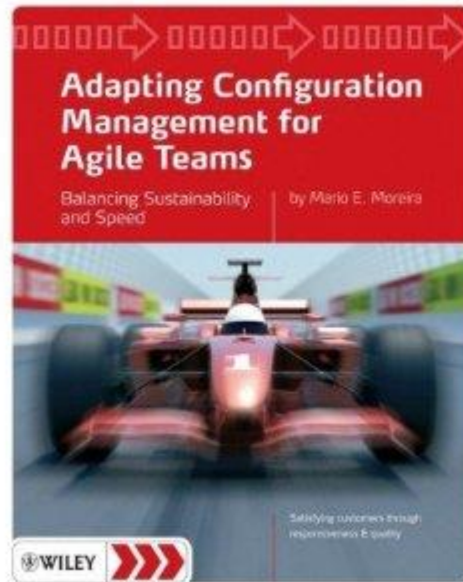

1. Conceptos básicos
2. Operaciones
3. Uso de repositorios
4. Resumen
5. Bibliografía

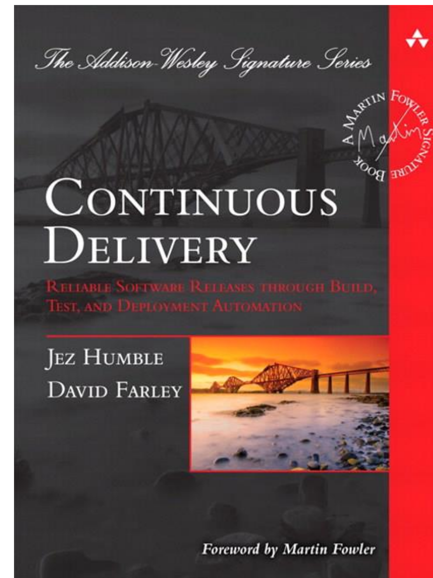
4. Resumen

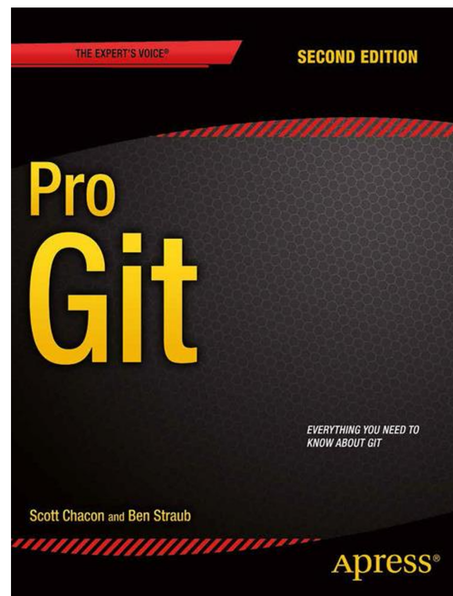
- Definimos de qué archivos vamos a hacer seguimiento y de cuáles no
- Decidimos cómo organizar el repositorio del equipo y del proyecto y su modelo de uso (*usage model*)
- Limpiar todas las ramas que no se vayan a utilizar
- Establecer un único nombre para cada persona que contribuya al proyecto
- Definir una política de gestión de *commits* atómicos
- Consensuar y usar una política de nombres de *commit*

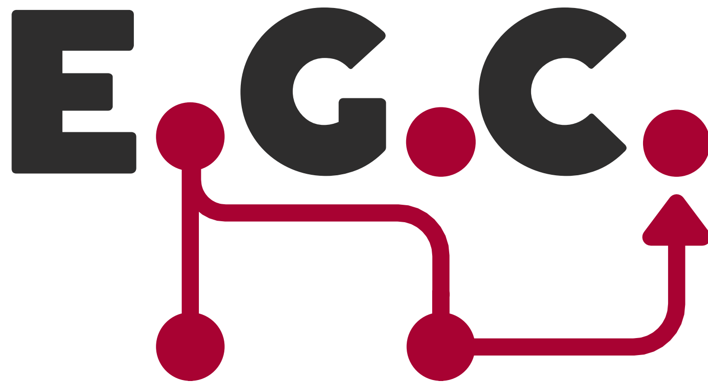

1. Conceptos básicos
2. Operaciones
3. Uso de repositorios
4. Resumen
5. Bibliografía

5. Bibliografía









Grado en Ingeniería Informática - Ingeniería del Software

Evolución y Gestión de la Configuración

Escuela Técnica Superior de
Ingeniería Informática

¡Gracias!