escuela técnica superior
de ingeniería informática

Gestión del código fuente
Source code management

Departamento de
Lenguajes y Sistemas Informáticos

Evolución y Gestión de la Configuración

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

Índice

Las leyes sobre la evolución

Las (8) Leyes de Lehman sobre evolución de software:

1.- **Cambio continuo** (74). Para satisfacer nuevas necesidades

2.- **Complejidad creciente** (74)

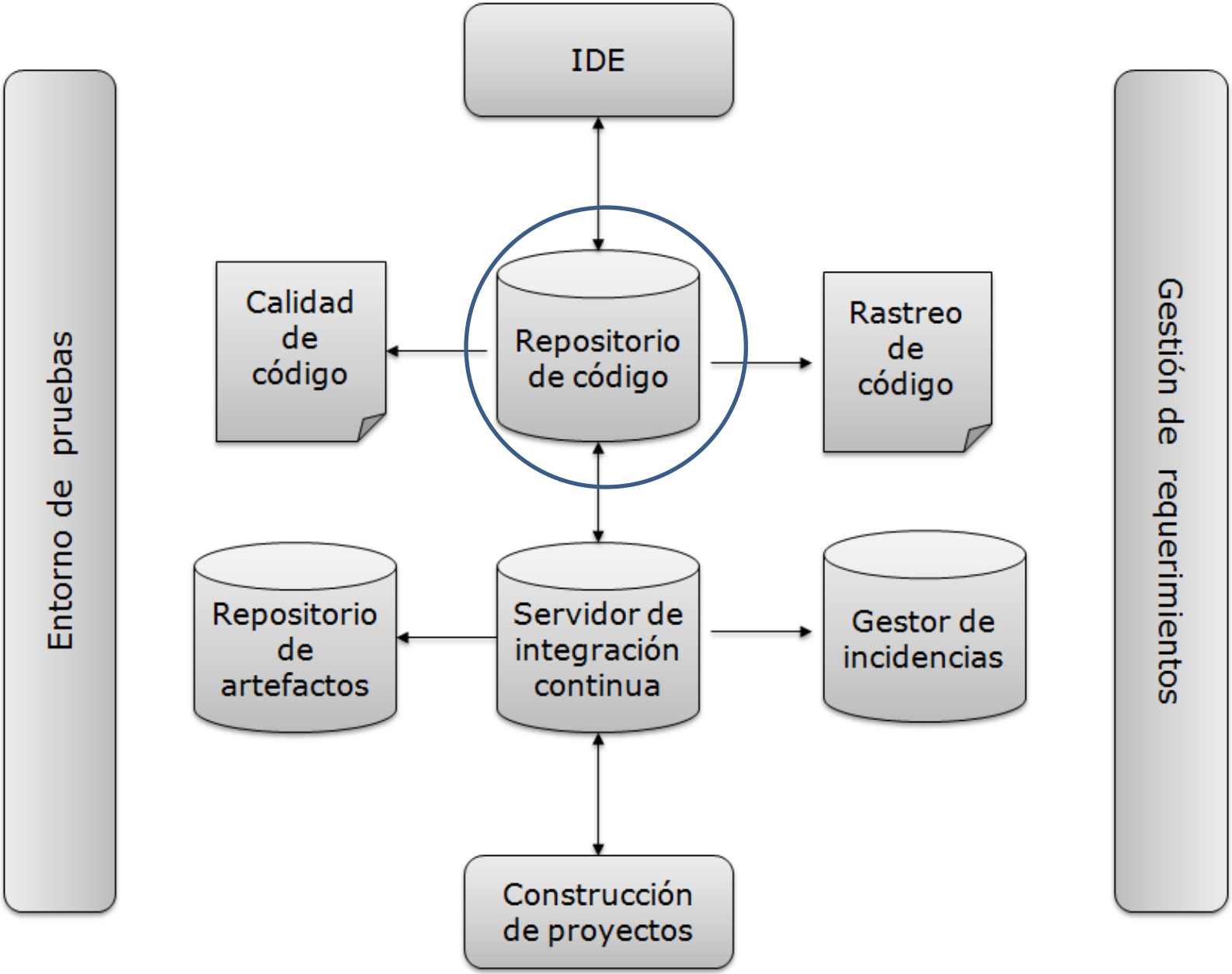
6.- **Crecimiento continuo** (91)

7.- **Calidad decreciente** (96)

"No hay nada permanente excepto el cambio"

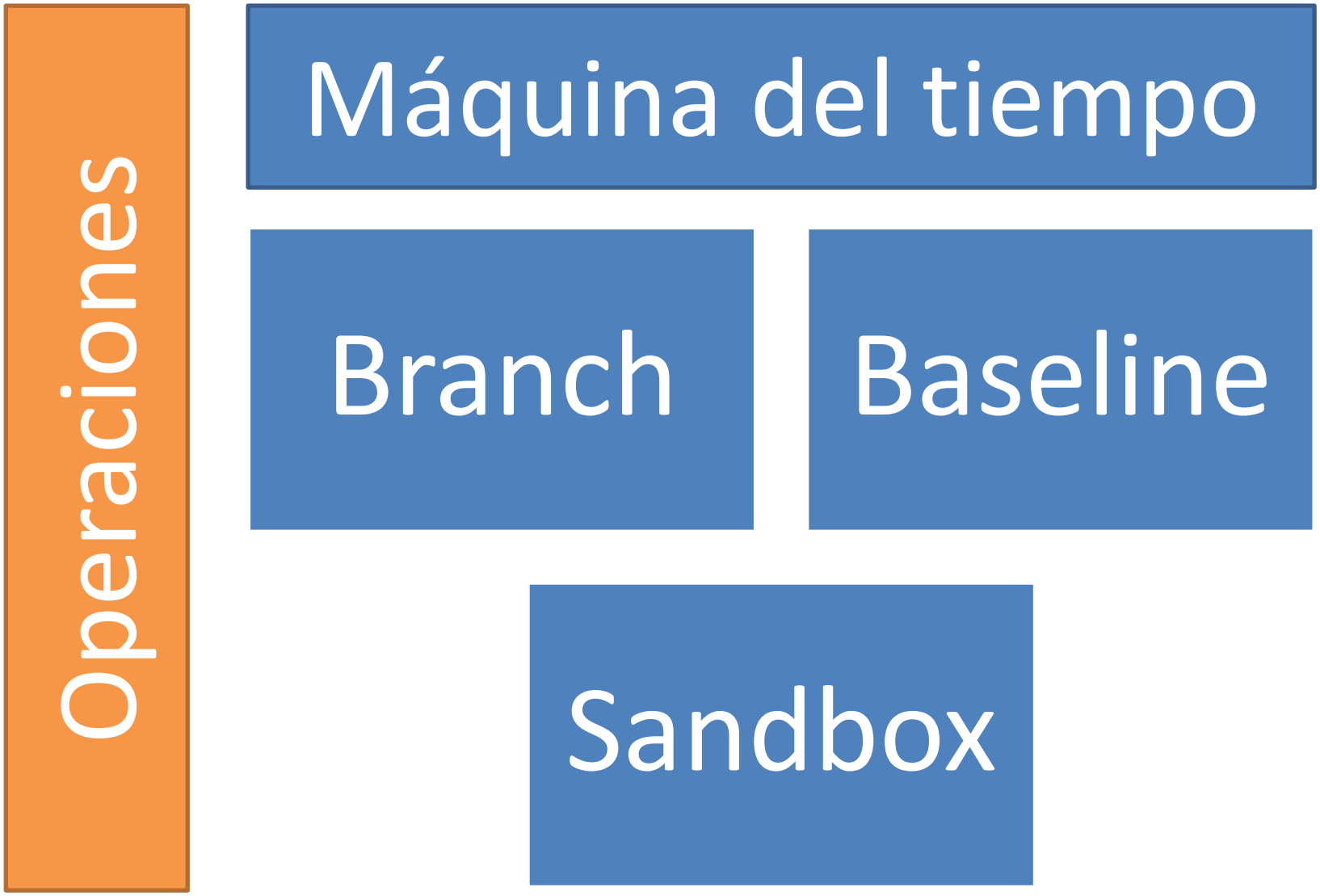
(Heráclito, 500 AdC)

Ecosistema de desarrollo



Índice

Conceptos básicos



Repositorio de código = *time machine*

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”….

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

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*"

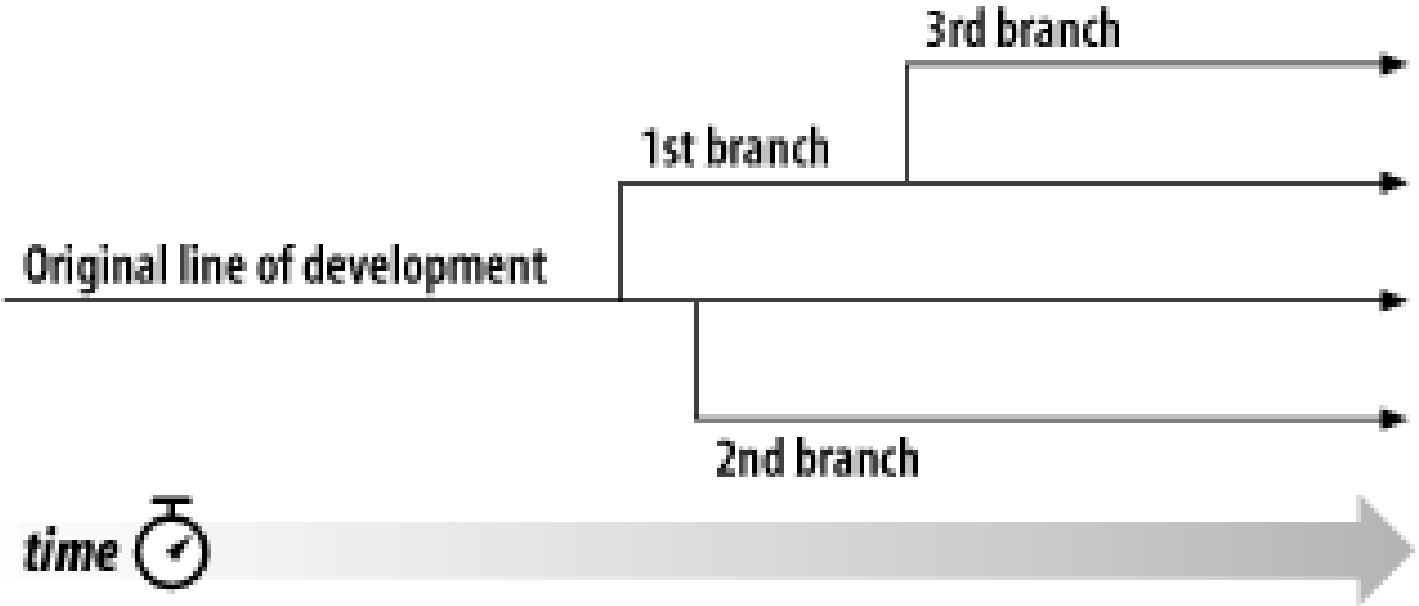
¿Por qué un sistema distribuido?

Sistemas 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.

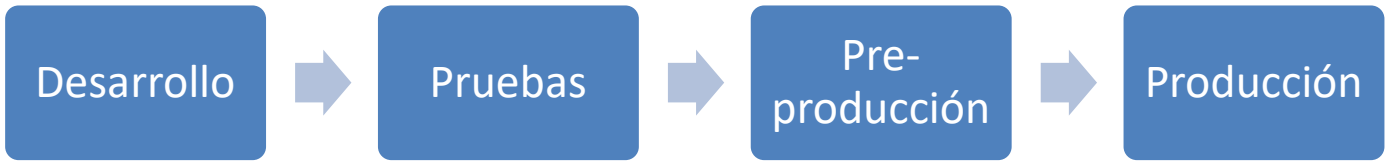
Branch

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



Baseline

- **Baseline:** Los cambios en el software requieren del paso por una serie de estados. Identificar los estados importantes de los artefactos de software es lo que se llama crear un "baseline" del producto.
- También se suele conocer como "*tagging*" "*labeling*" "*snapshoting*"
- Las "baseline" deben ser **inmutables**.
- Estados frecuentes:



Sandbox

- **Sandbox/workspace/working copy**

checkout

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

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

- También 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

Ejercicios

- Imagina que has hecho durante un un tiempo un archivo sobre el que ya no quieres seguir hacienda seguimiento ¿Qué harías?
- Imagina que tienes un patrón para ignorar un conjunto de archivos (ej. *.png), pero hay uno determinado que quieres empezar a hacer seguimiento (ej. logo.png) ¿Cómo lo harías?

Índice

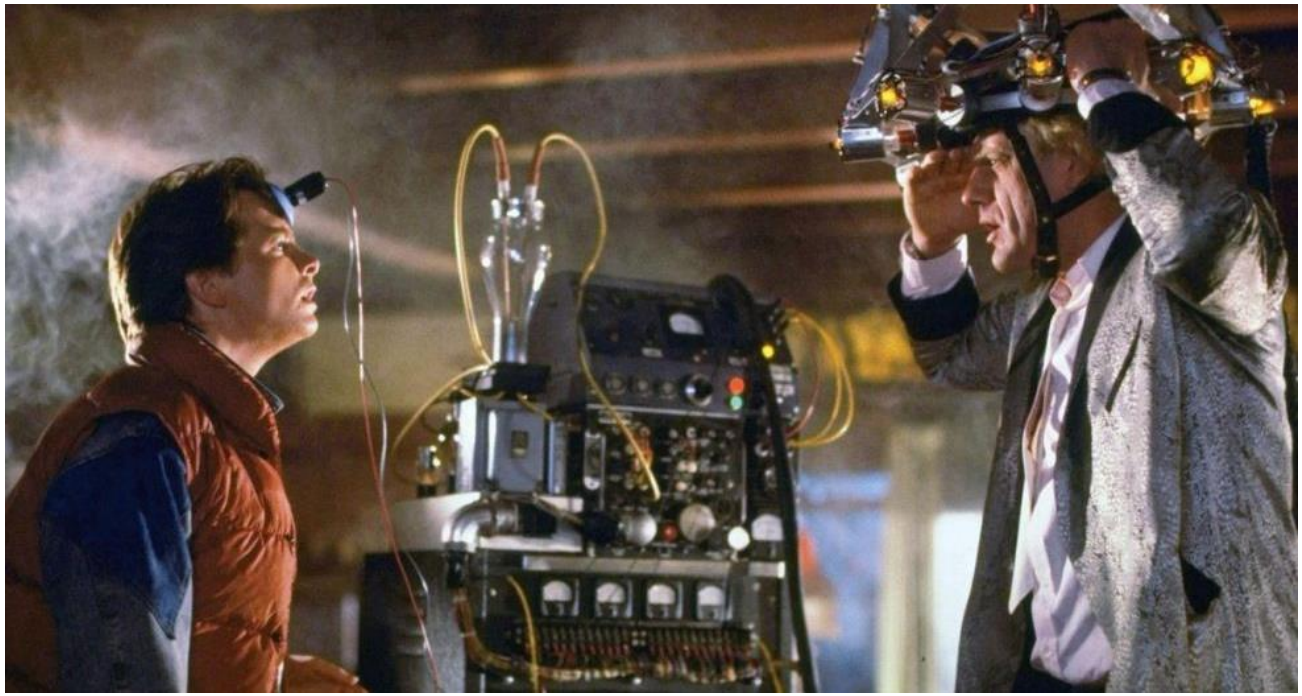
Operaciones

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

Tipos de operaciones

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

Crear el repositorio



Clonando

“Forkeando”

Creando nuevo

En local

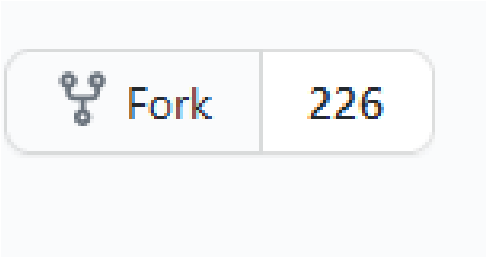
En remoto

Crear el repositorio

Clonando

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

"Forkeando"



Nuevo

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

Tipos de operaciones

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

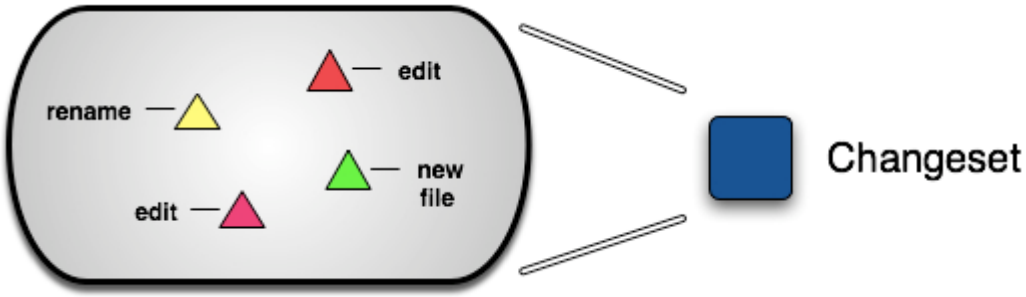
Estados de un *configuration item*

Sandbox
(sin seguimiento)

Parte de la historia
(con seguimiento)

Ignorado

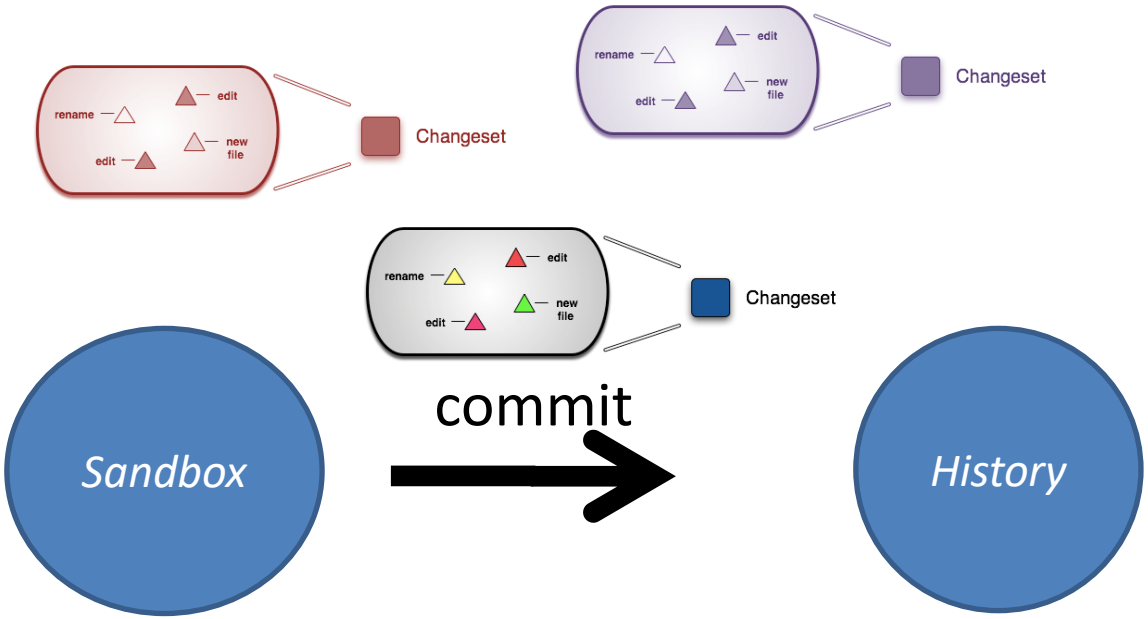
Changeset: conjunto de cambios que deben ser tratados como un conjunto indivisible (en svn se conoce como “revision”, en git como “commit”).



Operaciones frecuentes

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

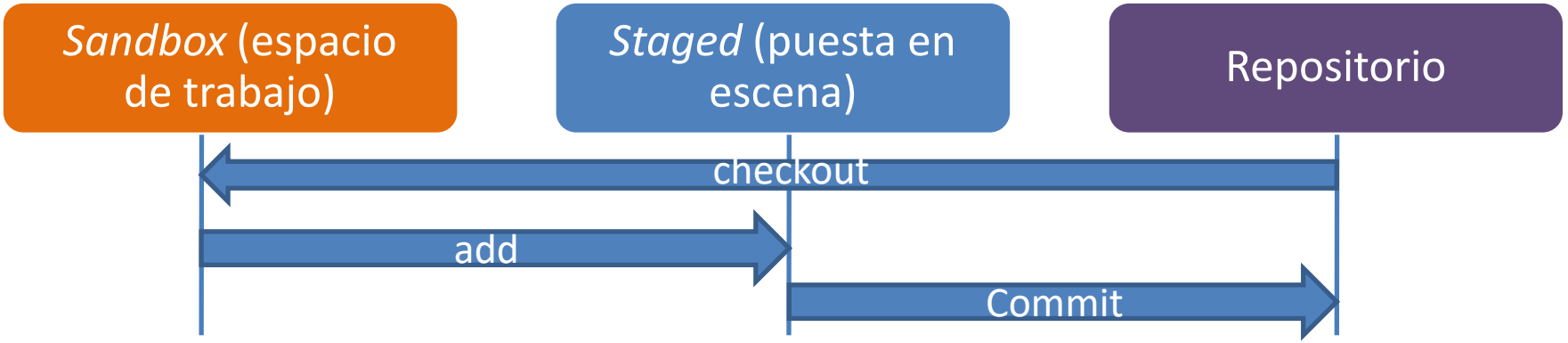
Lo ideal es tener *commits* atómicos



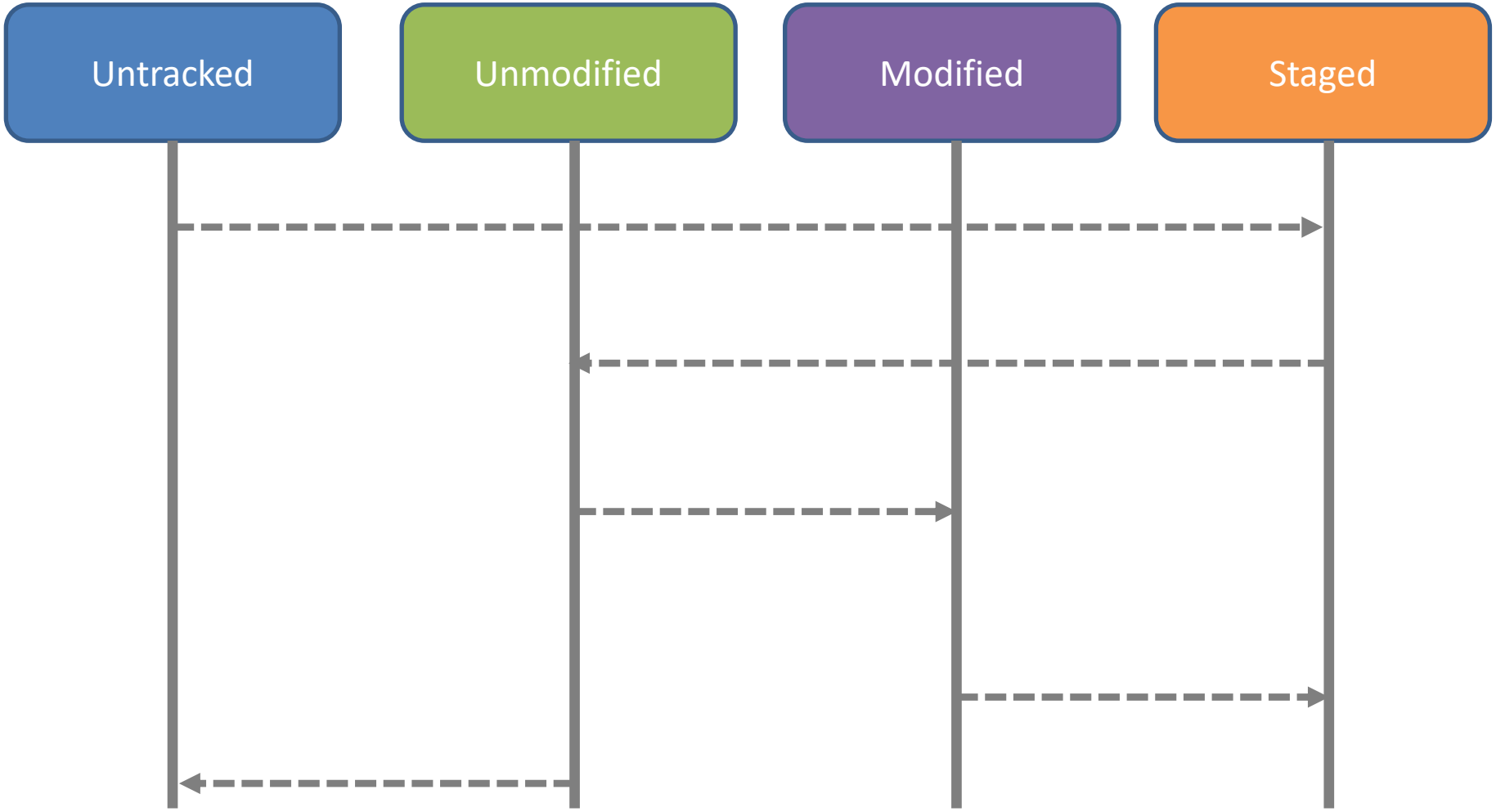
Operaciones frecuentes

¿Qué hay en un *commit*?

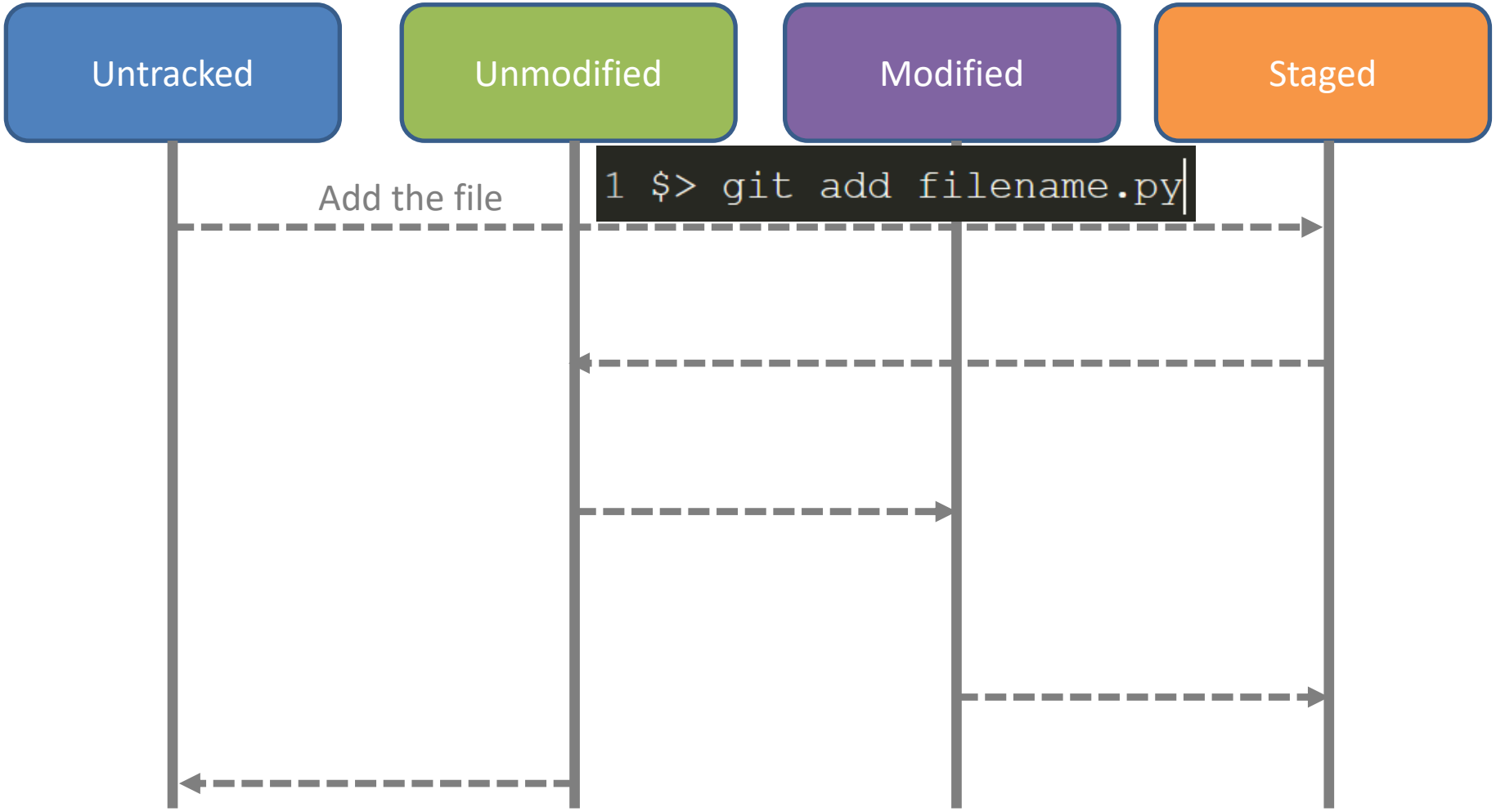
- Datos modificados + metadatos
- Un fichero *rastreado (*con seguimiento*)* puede estar en tres estados (en el caso de GIT):
  - Confirmado (*committed*)
  - Modificado (*modified*)
  - Preparado (*staged*)



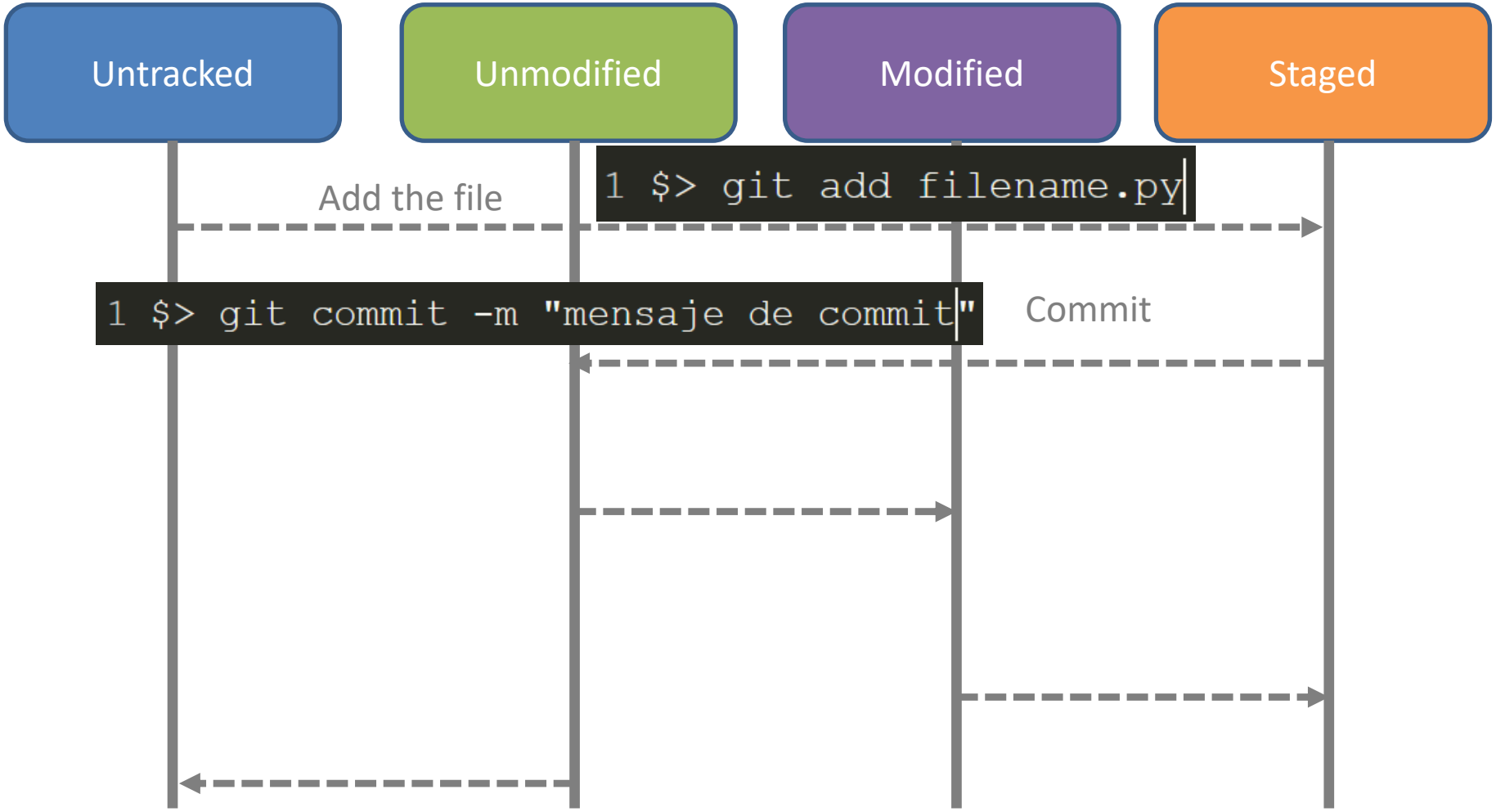
Estados de un *configuration item*



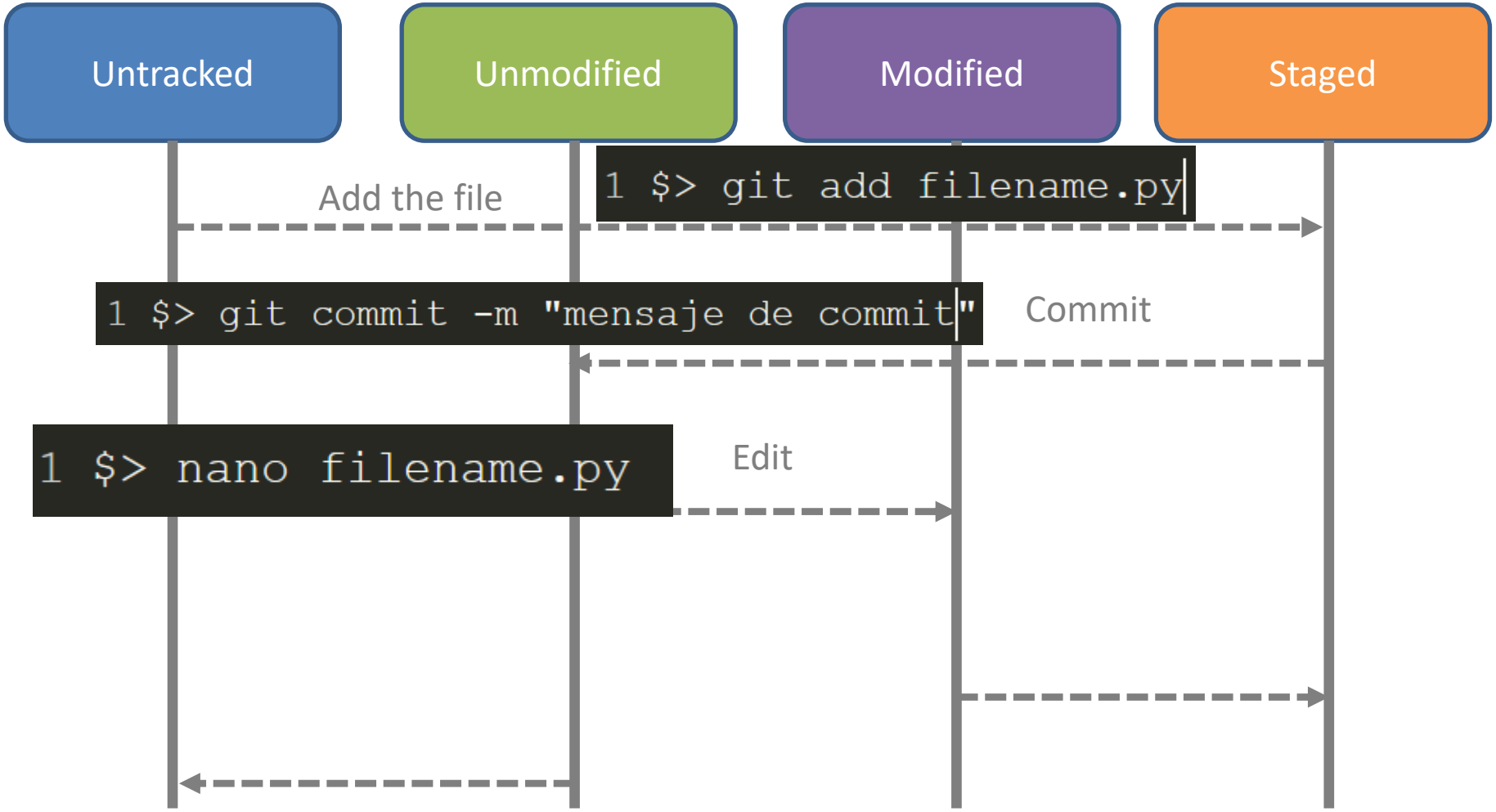
Estados de un *configuration item*



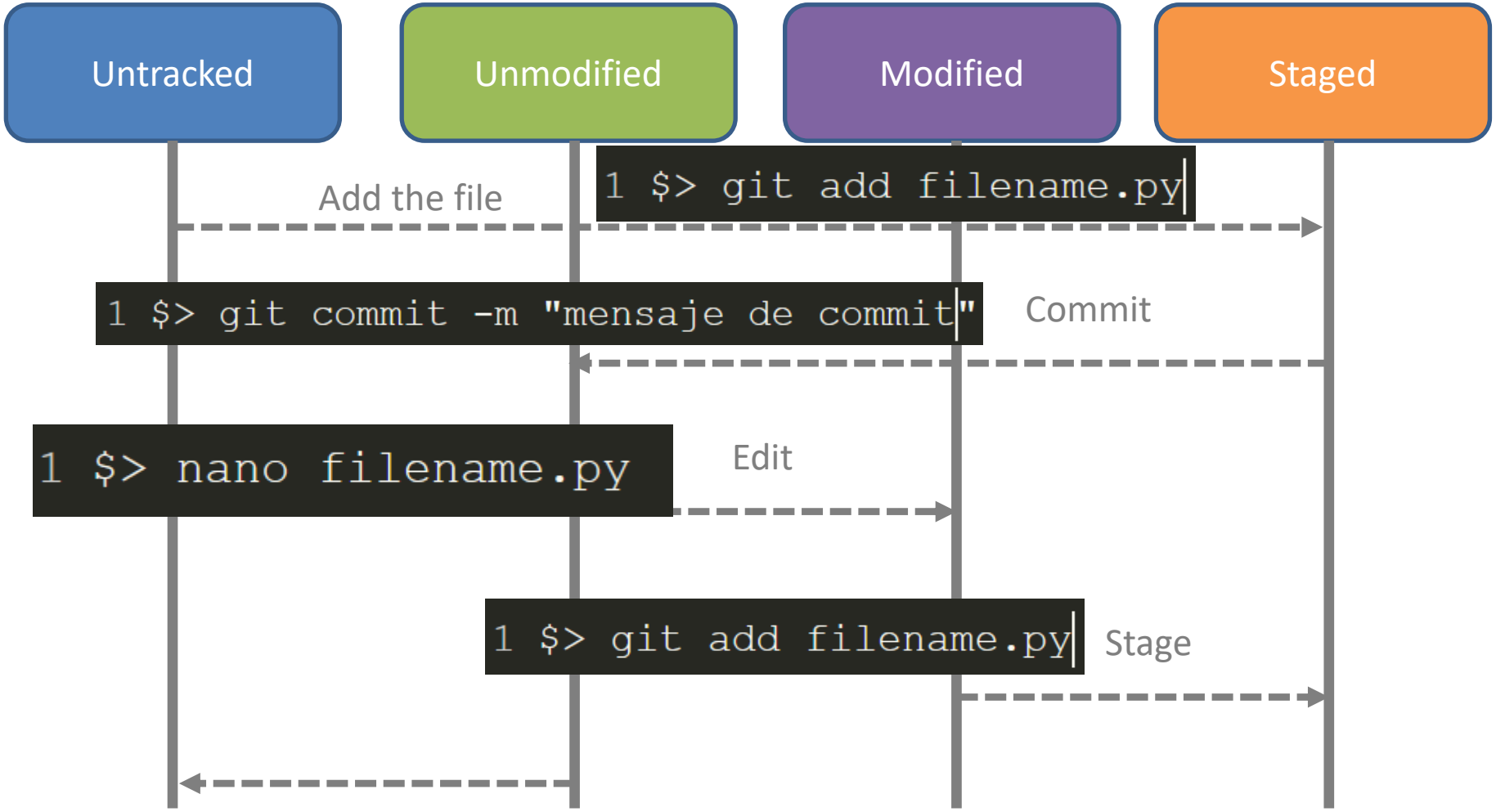
Estados de un *configuration item*



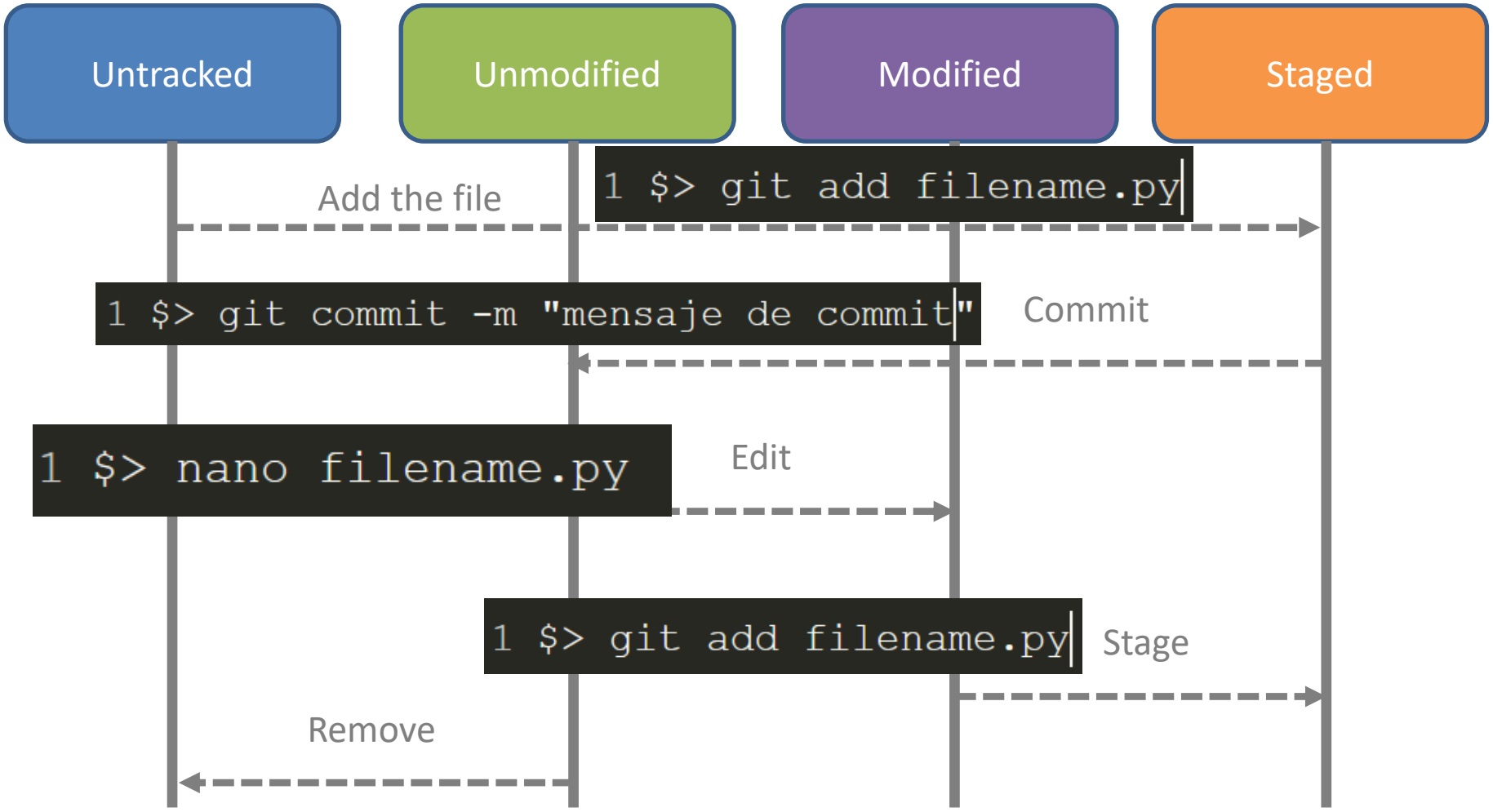
Estados de un *configuration item*



Estados de un *configuration item*



Estados de un *configuration item*




```
1 $> git rm --cached filename.py
```

Tipos de operaciones

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

Operaciones frecuentes

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

checkout

OJO: puede haber “juguetes” nuevos

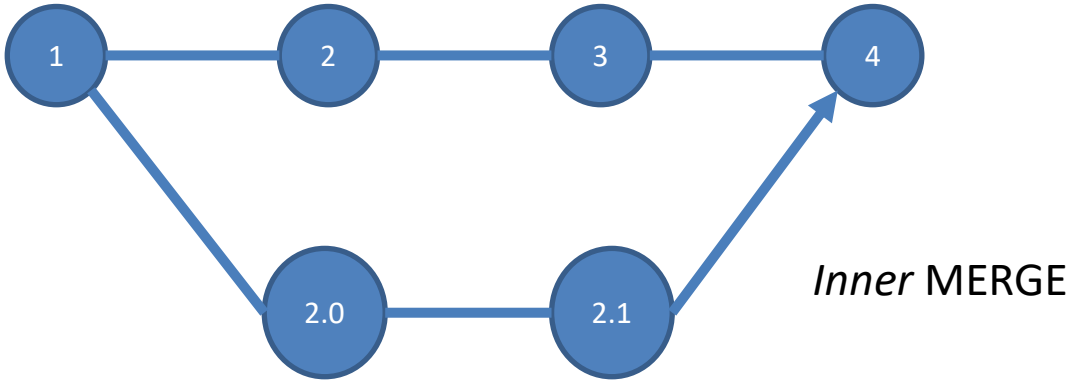
Recordad tener clara la respuesta a esta pregunta:
¿tengo que meter los ficheros de configuración de mi sandbox en el área de trabajo?

Tipos de operaciones

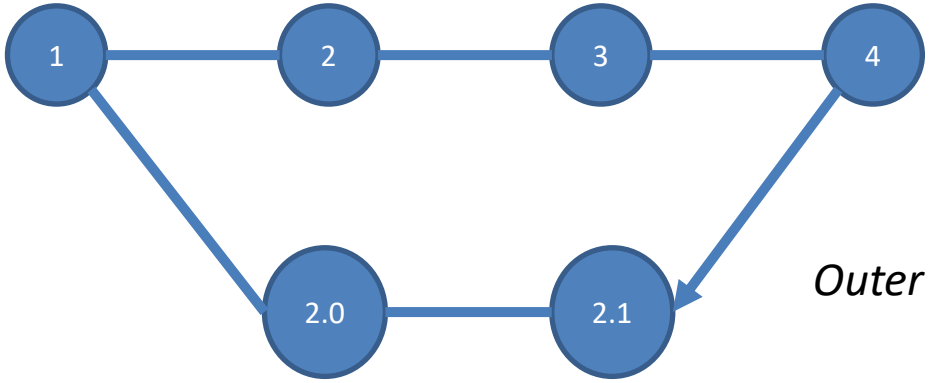
- De creación
- De escritura
- De lectura
- De ramas

Operaciones frecuentes

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



Inner MERGE



Outer MERGE

Si queries aprender “jugando” de manera visual sobre commandos de git

- https://git-school.github.io/visualizing-git/

Operaciones frecuentes

- **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*

- Introducción
- Conceptos básicos
- Operaciones
- Gestión de ramas
- Uso de repositorios
- Principios
- Resumen
- Bibliografía

Problema a resolver

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?

Motivos para la creación de ramas [Humble & Farley]

- **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.

Índice

¿Cómo y cuándo hacer commit?

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

¿Para qué?

¿Cómo y cuándo hacer commit?

- 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

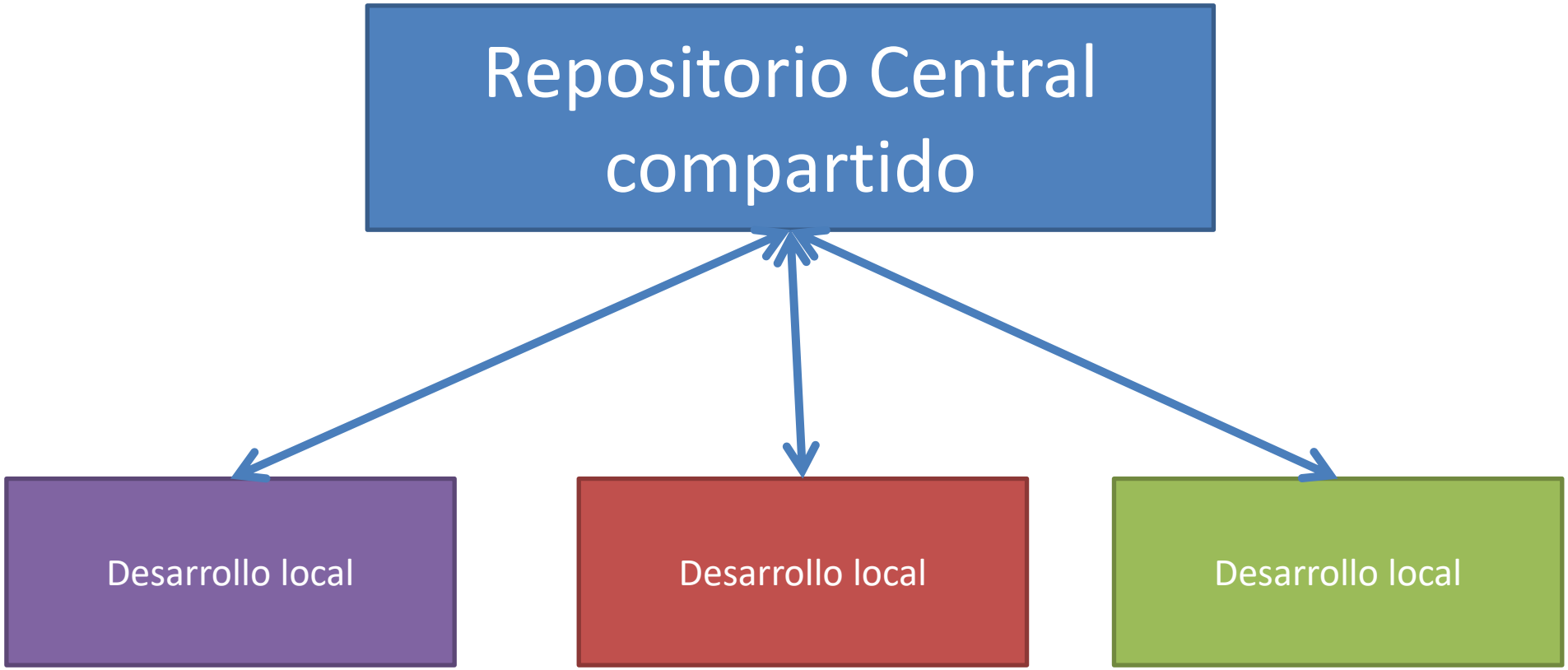
```
<type>(<scope>): <subject>
<BLANK LINE>
<body>
<BLANK LINE>
<footer>
```

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

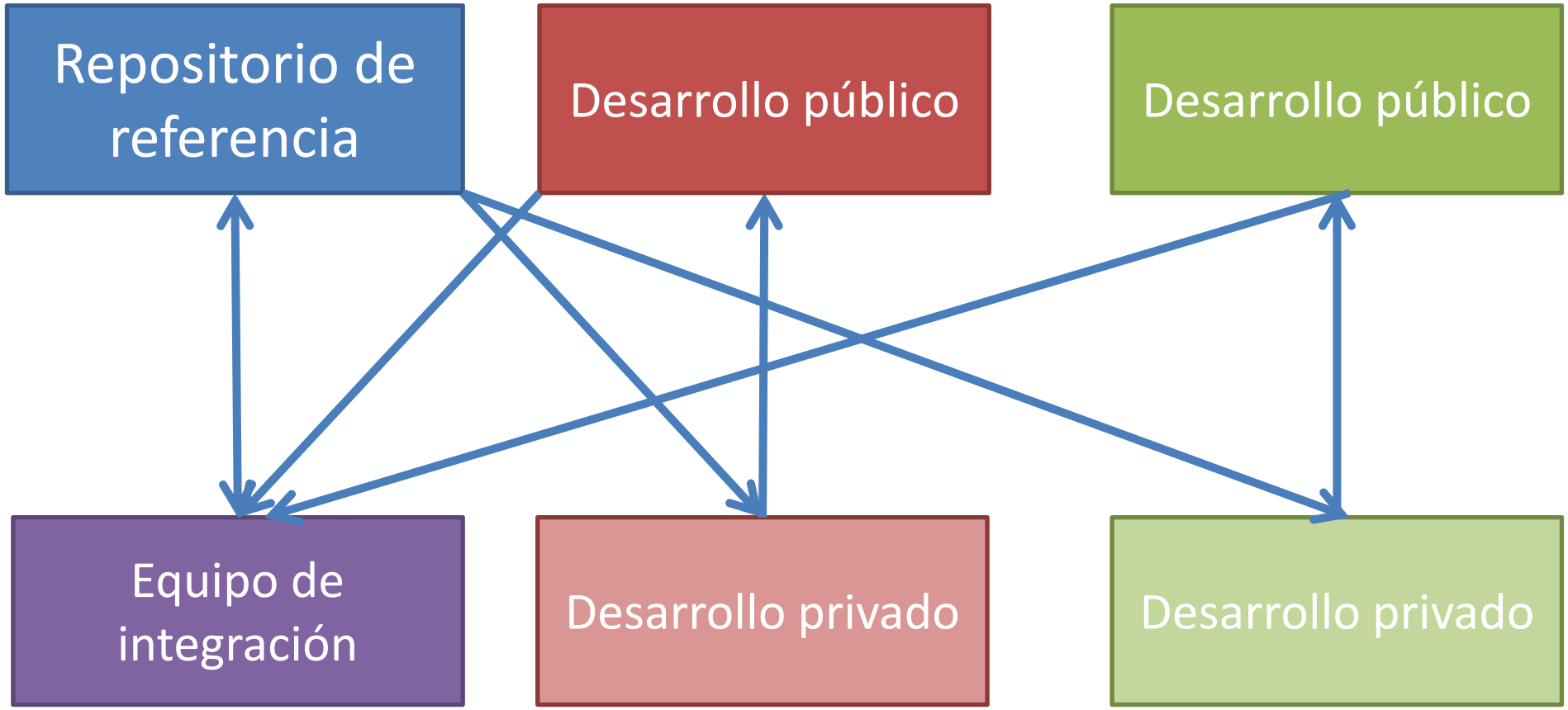
Usage models – principios básicos

- Todo lo que hay en master 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

Modelo centralizado

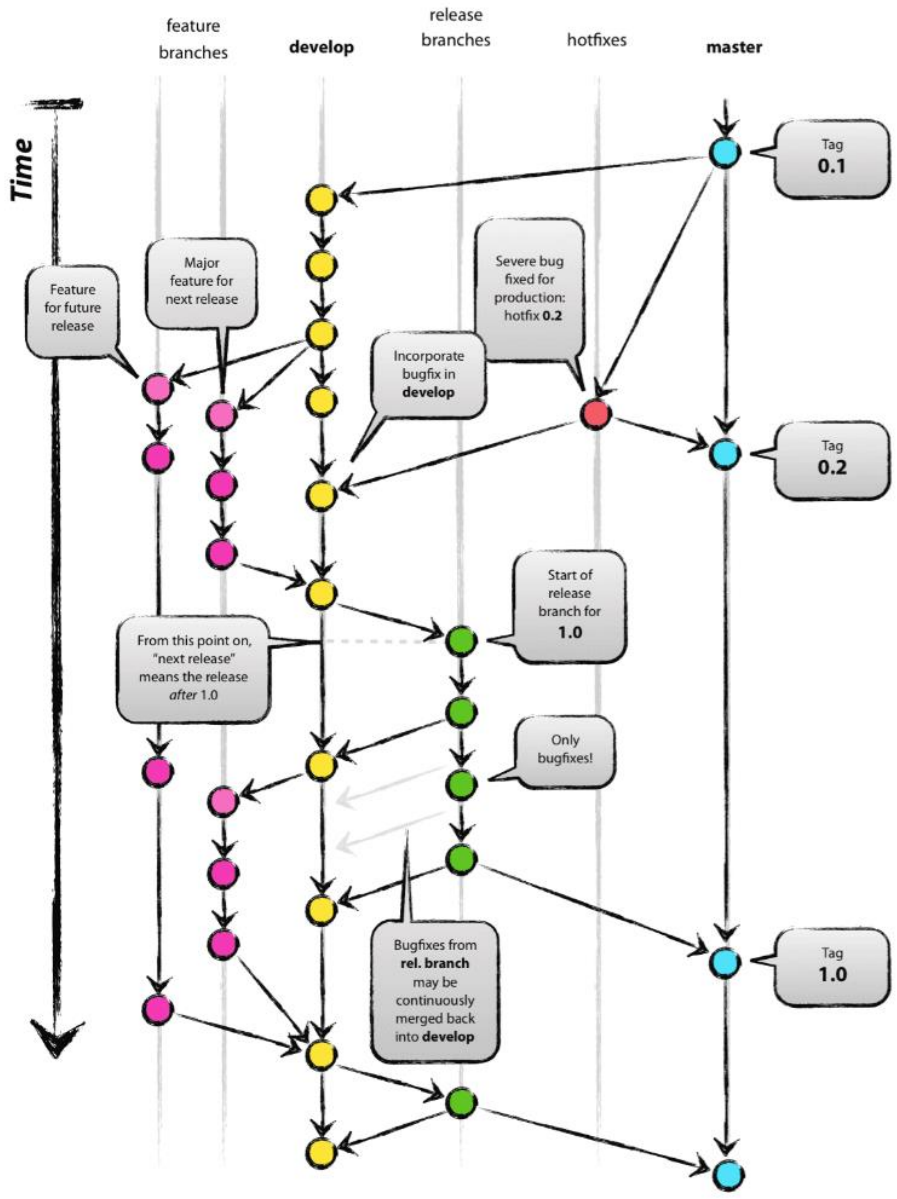


Modelo de integración



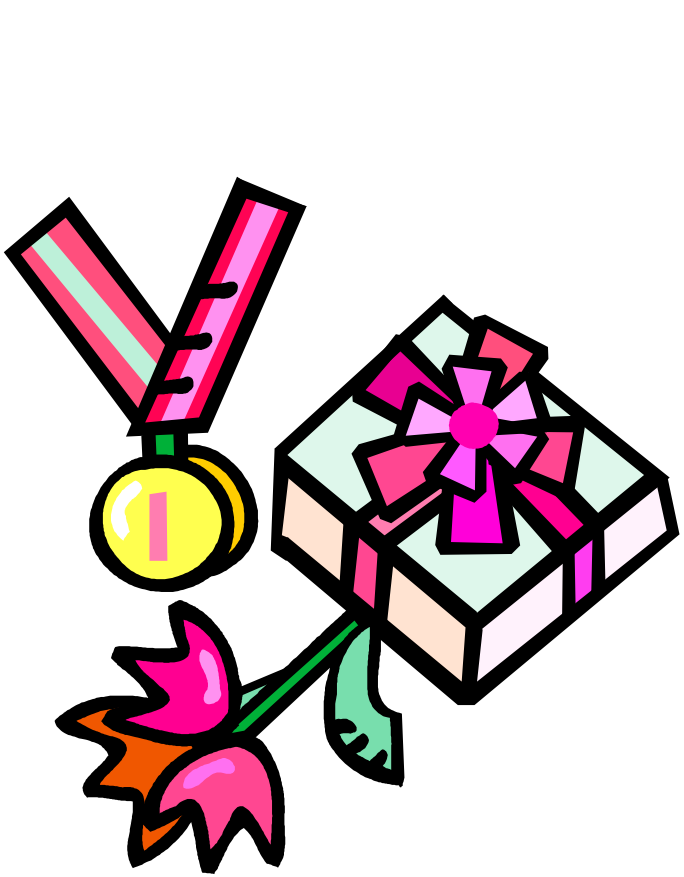
1. Existe un repositorio público de referencia
2. Los desarrolladores/as crean una copia privada para proponer cambios
3. Las contribuciones se hacen públicas en un repositorio
4. Se le envía una propuesta de integración al equipo de integración
5. El equipo de integración verifica que los cambios sean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

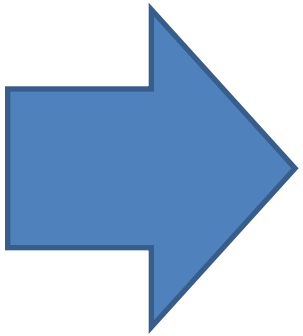
Git flow



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

Git Flow





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.

Índice

Source code management

Principios [Aiello]

- El código siempre está a salvo, *nunca* se pierde (efecto Y2K)
- El código sigue una línea temporal
- Gestionar las variantes del código debe ser fácil usando *branches*
- Si hay distintas *branches* estas deben poder fusionarse facilmente (*merge*)
- La gestión del código debe ser ágil y reproducible
- Debe permitir la trazabilidad y rastreo de los cambios
- Debe ayudar a mejorar la productividad y la calidad del código

Índice

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
- Definir una política de gestión de commits atómicos
- Consensuar y usar una política de nombres de *commit*

Resumen

- ¿Qué hemos aprendido?
  - El cambio es inevitable
  - Si no se gestiona bien puede haber muchos problemas
  - Los conceptos básicos de gestión del código: branches, repo, tag, sandbox,…
  - Distintas estrategias de gestión de las ramas
  - Escenarios típicos de uso de un SCV
- ¿Qué veremos en las siguientes lecciones?
  - En prácticas de laboratorio se trabaja con gestores de versiones para poner en práctica distintos escenarios y también para gestionar las peticiones de cambio
  - Cómo integrar el repositorio de código en el ciclo de CI

Índice

Bibliografía