Git

Introducción

Gorka Guardiola Múzquiz

GSYC

6 de mayo de 2021

Parte I

Sesiones de trabajo

Contenido

Problema básico de desarrollo

- ¿Alguna vez, has?...

Práctica imaginaria: enunciado

- Desarrolla un programa hexd similar a `od -tx1c`
- Imprime la salida en hexa y caracteres ascii imprimibles en otra línea

```
 $ cat hola
Hola soy un fichero.
#Para imprimir y probar

 $ hexd hola
0000000  48  6f  6c  61  20  73  6f  79  20  75  6e  20  66  69  63  68
          H   o   l   a       s   o   y       u   n       f   i   c   h
0000020  65  72  6f  2e  0a  23  50  61  72  61  20  69  6d  70  72  69
          e   r   o   .  \n   #   P   a   r   a       i   m   p   r   i
0000040  6d  69  72  20  79  20  70  72  6f  62  61  72  0a  0a  0a  0a
          m   i   r       y       p   r   o   b   a   r  \n  \n  \n  \n
0000060  0a
         \n
0000061
```

Práctica imaginaria, parte opcionales

- Hacer sólo esta parte si la otra ya funciona (hacer por separado, luego veremos)
- Imprimir los caracteres no imprimibles en notación estándar
- Otras bases (hexadecimal, por defecto $-tx$, octal $-to$ o decimal $-td$)

```
\a      audible BEL
\b      backspace
\f      form feed
\n      new line
\r      return
\t      horizontal tab
\v      vertical tab
```

Contenido

Problema de los ficheros

- Podría ir...
- ...haciendo ficheros con la fecha

```
 $ mkdir trabajo
 $ cd trabajo
 $ vi main.go        #edito los ficheros
#cada cierto tiempo
 $ cp -a ../trabajo ../version_‘date +’%d_%m_%Y’‘
 $ ls ..
trabajo
version_23_12_2018
version_24_12_2018
version_25_12_2018
version_26_12_2018
```

Problema de los ficheros

- Cada cierto tiempo tengo versiones especiales
- Ya me funciona una parte

```
#cada cierto tiempo
 $ cp -a ../trabajo ../version_‘date +’%d_%m_%Y_%s’‘_programa_inicial
 $ ls ..
trabajo
version_3_1_2018_1551442061
version_3_1_2018_1551442061_programa_inicial
version_3_2_2018_1552561741
version_3_2_2018_1552584416_imprime_chars
version_3_2_2018_15526144616_imprime_chars_y_hexa
```

Problema de los ficheros

- Es todo manual, es fácil perder el control.
- Si la lío, el software no me está ayudando.
- En cuanto tengo varias versiones se vuelve muy muy fácil liarla.

Inicializo el repo

```
 $ git config --global user.name "Gorka Guardiola"
 $ git config --global user.email "paurea@gmail.com"
#inicializo el repositorio
 $ git init pract
```

▸ Documentación ▸ Configuración ▸ Vista del usuario

Creo el programa inicial

- Los comandos go son para compilar (el programa están en golang)
- Voy de prototipo a prototipo
- Cada vez que tenga algo a lo que pueda darle nombre, hago commit
- Compruebo que compila/pasa los tests (todavía no hay) antes de nada

```
 $ git init hexd
 $ cd hexd
 $ vi main.go
 $ cat main.go
// hexd dumps binary files in various formats
package main

import "fmt"

func main() {
        fmt.Println("initial")
}
 $ go build
```

Creo el programa inicial

- Creo el commit asociado al prototipo
- Primero añado los fichero al commit (los paso a stage)
- Luego hago el commit

```
 $ git add main.go
 $ git status
 $ git commit -m 'initial program'
#Esto es parecido a cp -a ../trabajo ../version_`date +'%d_%m_%Y'`_programa_inicial
[master (root-commit) a46ec1f] initial program
 1 file changed, 8 insertions(+)
 create mode 100644 main.go
 $ git log
commit a46ec1f8c128329fa3770ecc8eef8746ee8256a0
Author: paurea <paurea@gmail.com>
Date:   Sat Dec 29 21:58:18 2018 +0100

    initial program
```

▸ Partes de un repo y estados

Ignorar ficheros

- Creo un fichero para ignorar lo que no quiero que acabe en los commit
- En este caso, el binario en la raíz y un fichero guide que uso para cortar y pegar código temporalmente
- Hago commit del fichero `.gitignore` para que forme parte del repo

```
$ vi .gitignore
$ cat .gitignore
#fichero con comandos y cortar y pegar
/guide
#el binario
/hexd
$ git add .gitignore
$ git commit -m 'ignore binaries and guide'
```

▸ .gitignore

Hacer más commits y verlos

- Sigo haciendo commits
- Los pruebo siempre antes

```
 $ vi main.go
 $ go build && go test
 $ git add main.go
 $ git commit -m 'divide in pieces a reader'
[master 50cf897] divide in pieces a reader
 1 file changed, 32 insertions(+), 3 deletions(-)
#puedo usar una hash (o sólo el comienzo si es único) para inspeccionar un commit
```

Ver la lista de commits

- Llamo a log para inspeccionar los commits
- Log tiene muchos parámetros, yo suelo definirme adog para que lo muestre bonito

```
#añado un comando para que log imprima como yo quiero
 $ git config --global alias.adog "log --all --decorate --oneline --graph"
 $ git adog
* 50cf897 (HEAD -> master) divide in pieces a reader
* 53427c2 ignore binaries and guide
* a46ec1f initial program
```

▸ Alias

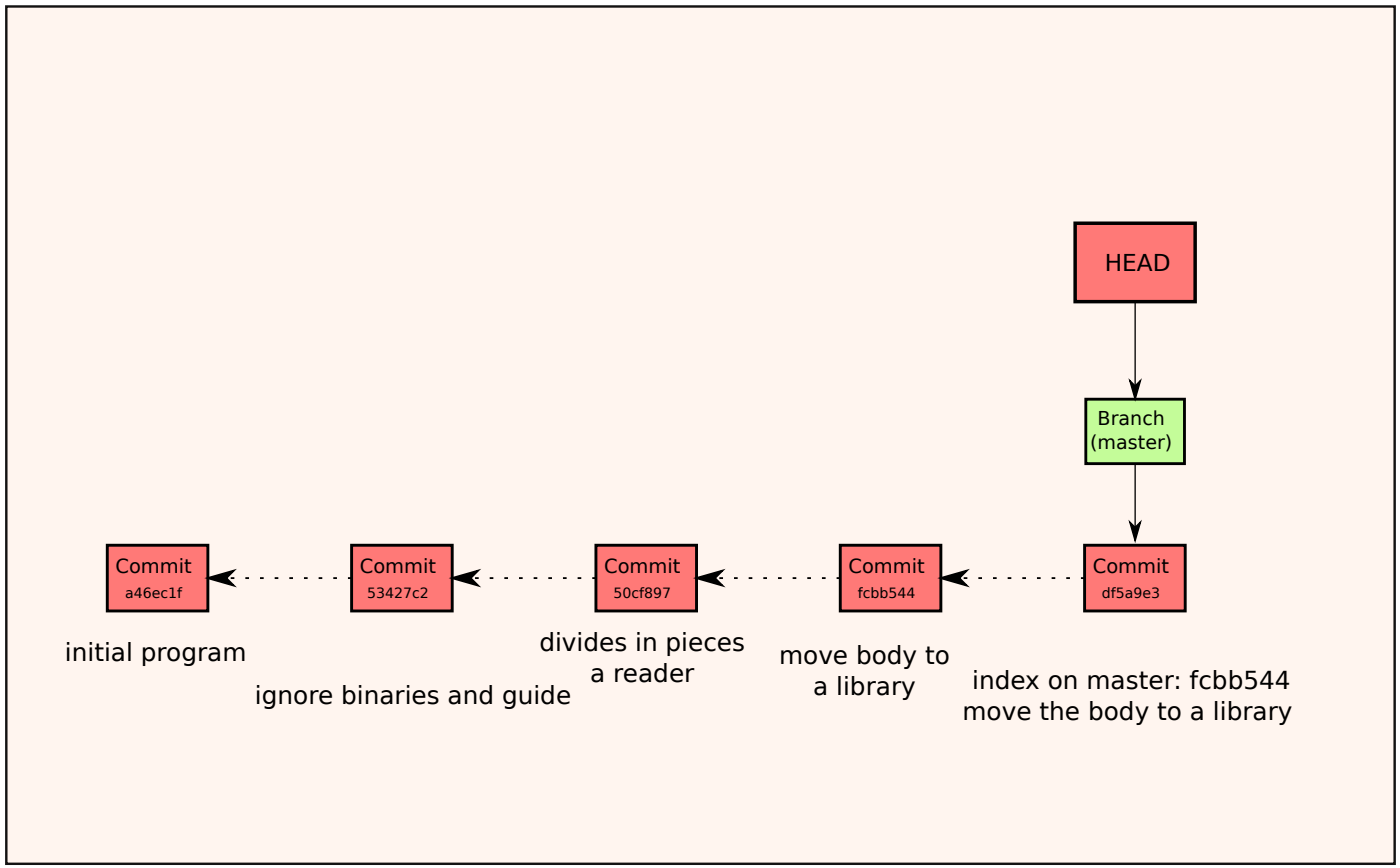
Elijo bien los nombres

- Voy haciendo una foto cada vez que tengo algo para ponerle nombre (commit)
- Ojo, los mensajes de commit y comentarios, mejor en inglés (lenguaje internacional...)
- Los nombres de los commits, como si delante pusiese, 'with this commit the program/repo will...'
- Si me interrumpen, si tengo un bug gordo... puedo ver los commits anteriores, probarlos...

Ahora tengo

- Lista de commits, cada uno con su antecesor, con nombre
- Navegables, puedo hace diffs, tengo fechas...
- Sólo con esto ya estoy mejor que con los ficheros (es automático)

Repositorio hexd



Hacer más commits y verlos

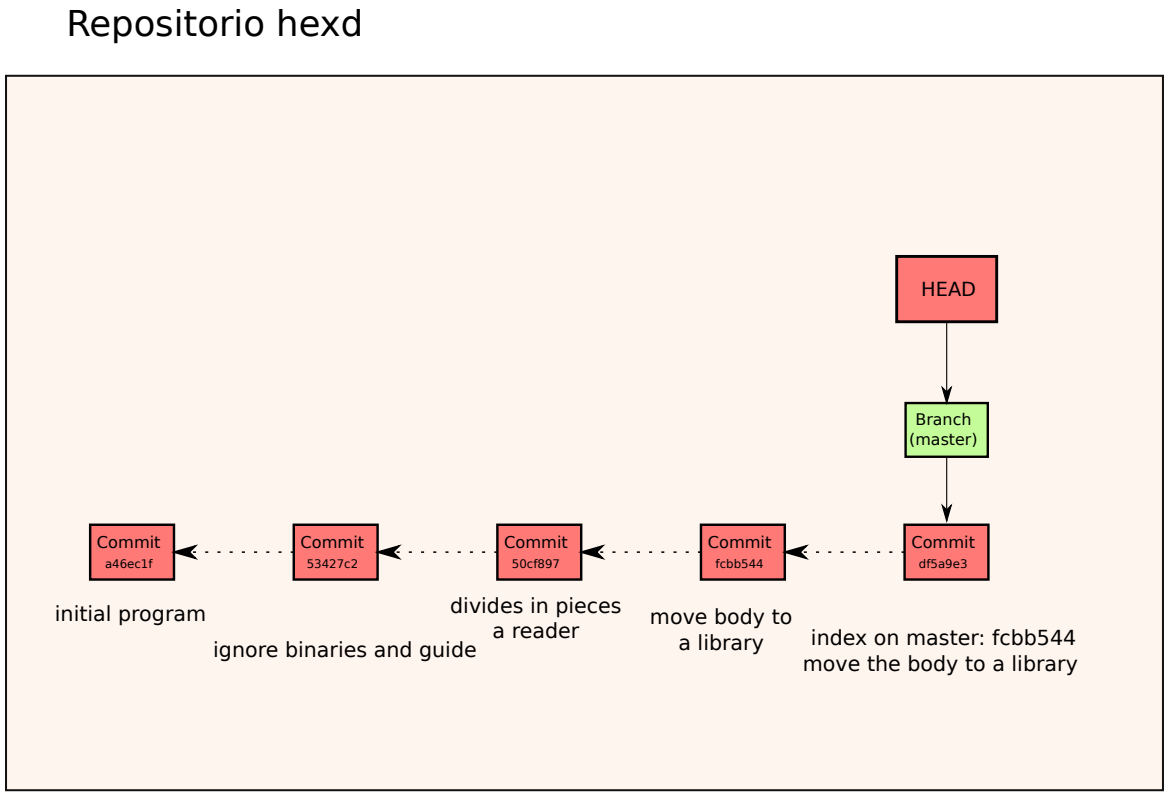
- Puedo inspeccionar cualquier commit con la hash

```
 $ git ls-tree 50cf897
100644 blob dbd62d94a38b78b37e85c01af443f40383ead313        .gitignore
100644 blob 53400f2f20189ea408c1acc9c22b1399ec4612d6        main.go
 $ git cat-file -p 53400f2f20189ea408c1acc9c22b1399ec4612d6
// hexd dumps binary files in various formats
package main

import (
        "fmt"
        "strings"
        "io"
        "log"
)
const NBytesLine=3        //fixed for now

func main() {
        //fake the file for now
        r := strings.NewReader("my file for now")
        buf := make([]byte, NBytesLine)
...
 $ git show 50cf897:main.go        #lo mismo en un paso
```

▸ Mirar cambios con diff

HEAD

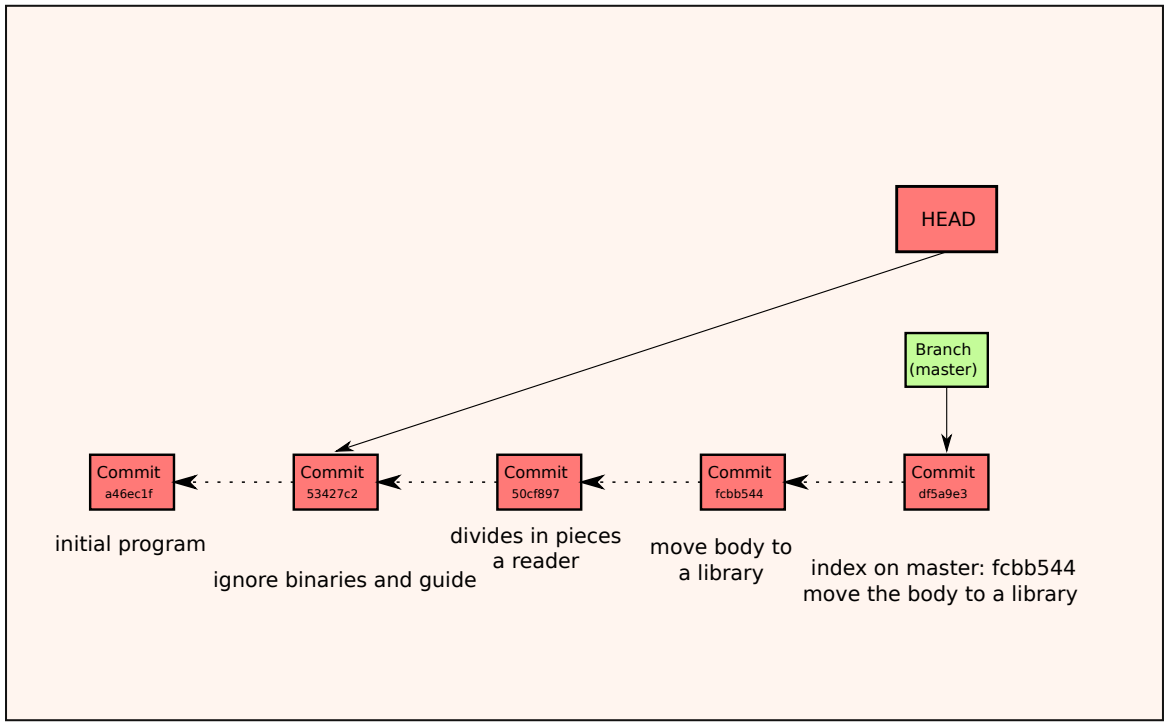
- `HEAD`: con cual estoy trabajando ahora
- Para navegar y traer ficheros al espacio de trabajo
- Commits antiguos, por ejemplo
- Hago checkout (ojo, ver `DETACHED HEAD` más adelante)
- Donde apunte `HEAD` es con lo que estoy trabajando



HEAD

- `HEAD`: con cual estoy trabajando ahora
- Para navegar y traer ficheros al espacio de trabajo
- Commits antiguos, por ejemplo
- Hago checkout (ojo, ver `DETACHED HEAD` más adelante)
- Donde apunte `HEAD` es con lo que estoy trabajando

Repositorio hexd



HEAD

- Lista de commits, cada uno con su antecesor, con nombre.
- Navegables, puedo hace diffs, tengo fechas...
- Sólo con esto ya estoy mejor que con los ficheros (es automático).

```
$ git add main.go dump/dump.go
$ git commit -m 'format a string per block'
$ git adog
* 5344987 (HEAD->master) format a string per block
* df5a9e3 index on master: fcbb544 move the body to a library
* fcbb544 move the body to a library
* 50cf897 divide in pieces a reader
* 53427c2 ignore binaries and guide
* a46ec1f initial program
```

Ahora tengo

- Quiero volver al estado del pasado, muevo el `HEAD` con checkout (normalmente, mejor crear una rama, pero todavía no sabemos)
- Teniendo todo en commit, nada sin salvar (si no `git stash`)
- Paso a estar `DETACHED HEAD`, ojo **no cambiar** sin crear una branch...

```
 $ git adog
* 5344987 (HEAD->master) format a string per block
* df5a9e3 index on master: fcbb544 move the body to a library
* fcbb544 move the body to a library
* 50cf897 divide in pieces a reader
* 53427c2 ignore binaries and guide
* a46ec1f initial program
 $ git checkout 53427c2
 $ git adog
* 5344987 (HEAD) format a string per block
* df5a9e3 index on master: fcbb544 move the body to a library
* fcbb544 move the body to a library
* 50cf897 (master) divide in pieces a reader
* 53427c2 ignore binaries and guide
* a46ec1f initial program
#aquí tengo el working como cuando hice el commit 'ignore binaries and guide'
 $ git checkout master
#vuelvo al estado inicial
```

▸ Detached head

Versiones para entregar

- De vez en cuando, tengo versiones (congelar, para entregar, por ejemplo).
- Que ya podría entregar (tienen algún fallo, pero...).
- Se pueden etiquetar.
- Para eso valen las tags.

▸ Tag

Versiones para entregar

- En cualquier momento puedo ejecutar `git tag nombre` o `git tag nombre SHA`.
- Y etiqueto el commit actual con ese nombre.

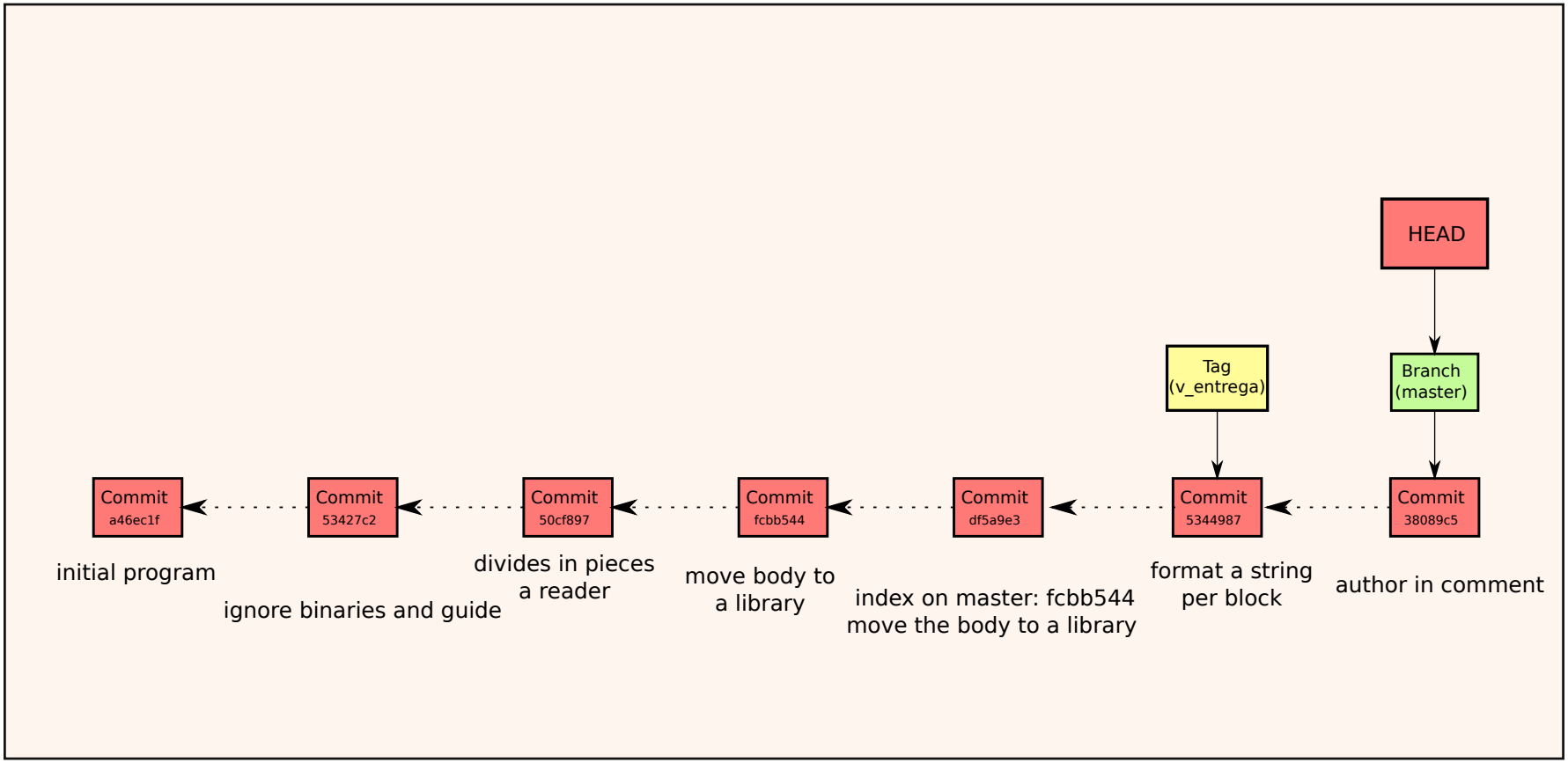
```
$ git tag v_entrega
$ vi main.go
$ git add main.go
$ git commit -m 'author in comment'
$ git adog
* 38089c5 (HEAD -> master) author in comment
* 5344987 (tag: v_entrega, optional) format a string per block
* df5a9e3 index on master: fcbb544 move the body to a library
* fcbb544 move the body to a library
* 50cf897 divides in pieces a reader
* 53427c2 ignore binaries and guide
* a46ec1f initial program
```

▸ Tag

Versiones para entregar

- En cualquier momento puedo ejecutar `git tag nombre`.
- Y etiqueto el commit actual con ese nombre.

Repositorio hexd



Versiones para entregar

- Si necesito sacar esa versión.
- Sólo tengo que hacer checkout del nombre.

```
$ git checkout v_entrega
```

▸ Tag

Contenido

Problema de los ficheros

- Ahora quiero hacer las partes opcionales.
- Pero mientras no quiero dejar de lado la obligatoria.
- Quiero seguir probándola...
- Quiero tener dos *ramas* de código (*branches*), experimental y estable.

▸ Branch

Hasta ahora

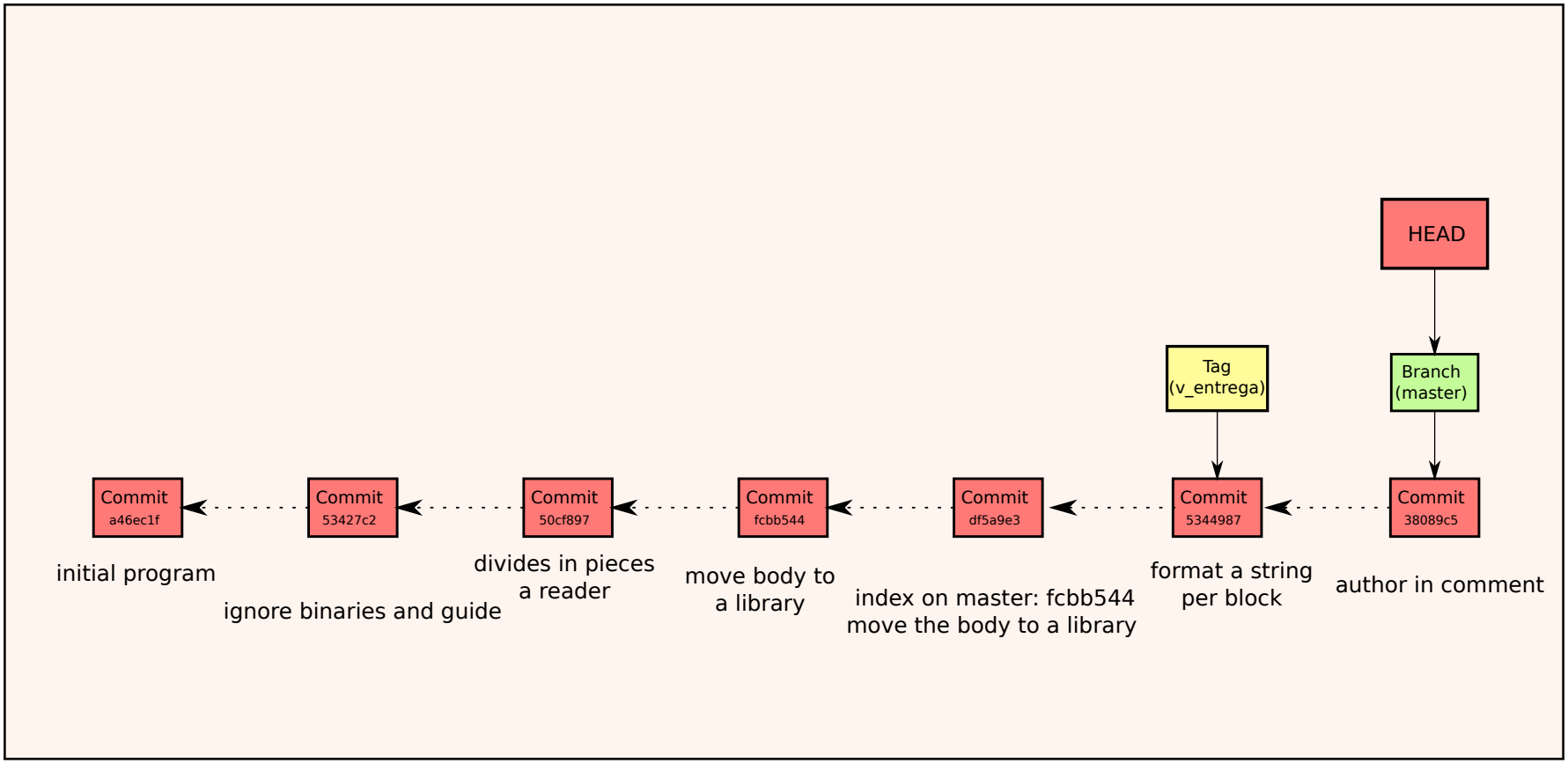
- Historia lineal de cambios

```
 $ git adog
* 38089c5 (HEAD -> master) author in comment
* 5344987 (tag: v_entrega) format a string per block
* df5a9e3 index on master: fcbb544 move the body to a library
* fcbb544 move the body to a library
* 50cf897 divides in pieces a reader
* 53427c2 ignore binaries and guide
* a46ec1f initial program
```

Hasta ahora

- Se puede visualizar con gitk

Repositorio hexd



gitk

Nueva rama

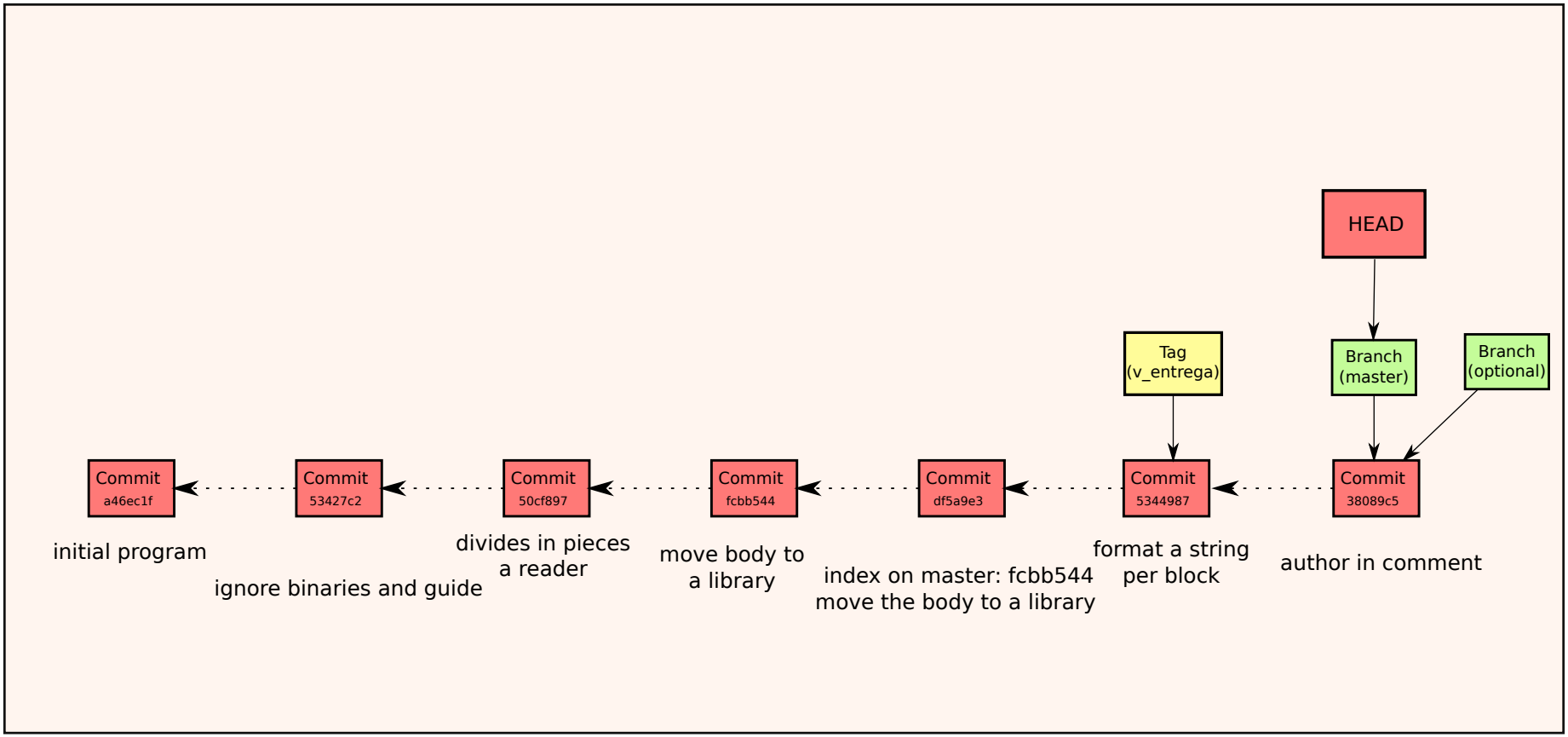
- Creo ahora la rama `optional`

```
$ git branch optional
$ git branch
* master
  optional
```

Nueva rama

- Creo ahora la rama `optional`

Repositorio hexd



Nueva rama

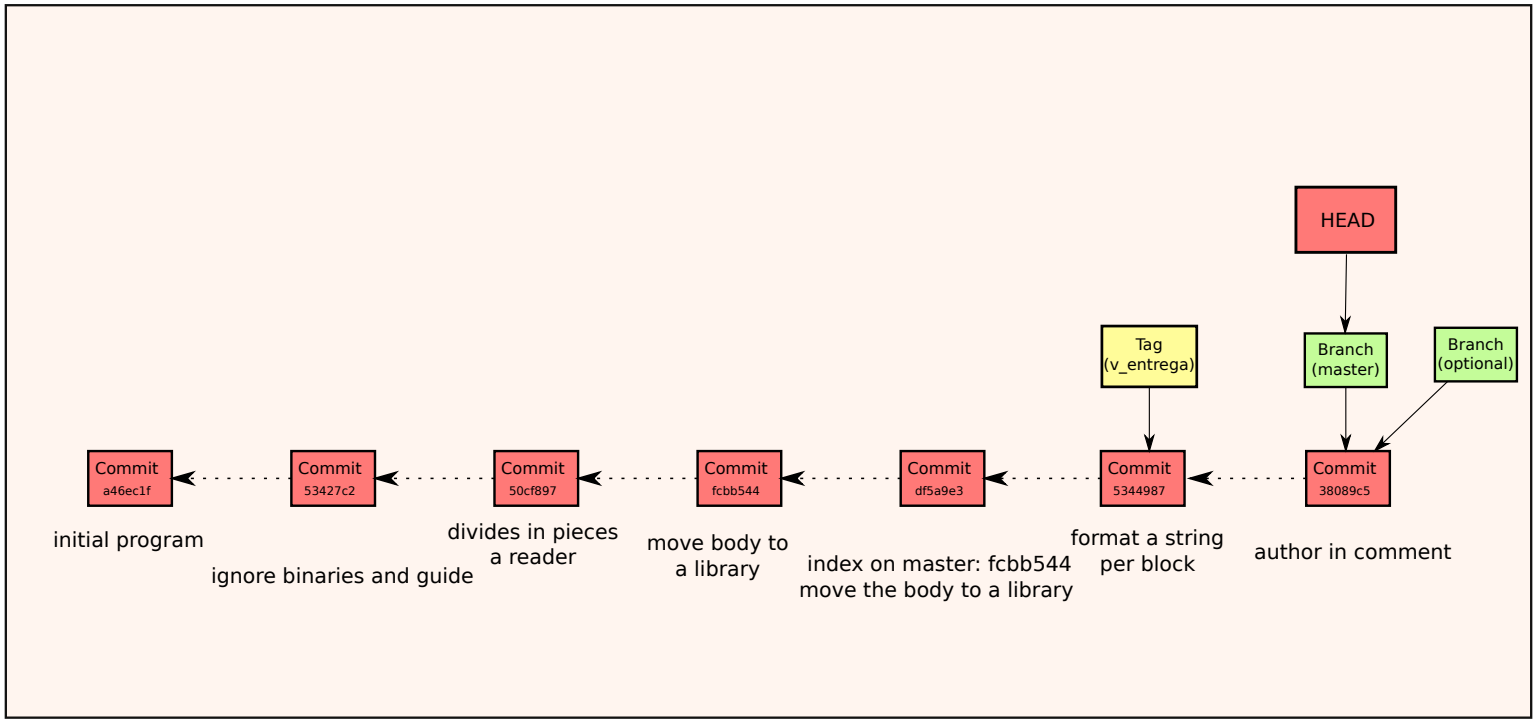
- Estoy en una rama (HEAD): `optional` o `master`.
- Los commits los hago en una rama y cada una crece por separado.
- Me paso a `optional` y hago un commit.

```
 $ git adog|sed 3q
* 38089c5 (HEAD -> master, optional) author in comment
* 5344987 (tag: v_entrega) format a string per block
* df5a9e3 index on master: fcbb544 move the body to a library
 $ git checkout optional
Switched to branch 'optional'
 $ git adog||sed 3q
* 38089c5 (HEAD -> optional, master) author in comment
* 5344987 (tag: v_entrega) format a string per block
* df5a9e3 index on master: fcbb544 move the body to a library
 $ vi main_test.go && go test
 $ git add main_test.go
 $ git commit -m 'initial tests for options'
[optional 66fb6db] initial tests for options
 1 file changed, 1 insertion(+)
 $ git adog|sed 4q
* 66fb6db (HEAD -> optional) initial tests for options
* 38089c5 (master) author in comment
* 5344987 (tag: v_entrega) format a string per block
* df5a9e3 index on master: fcbb544 move the body to a library
```

Nueva rama

- Trabajo en `optional`, hago el checkout

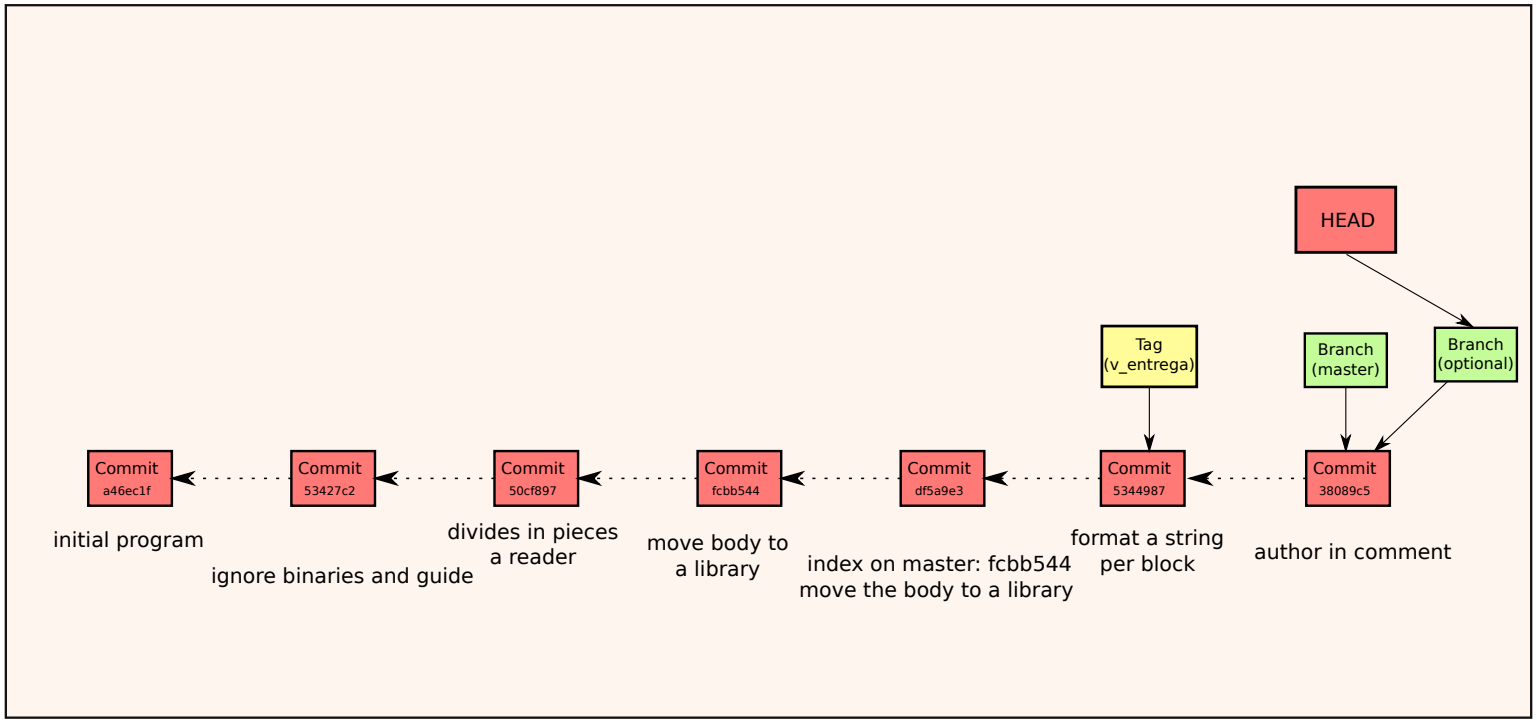
Repositorio hexd



Nueva rama

- Trabajo en `optional`, hago el checkout

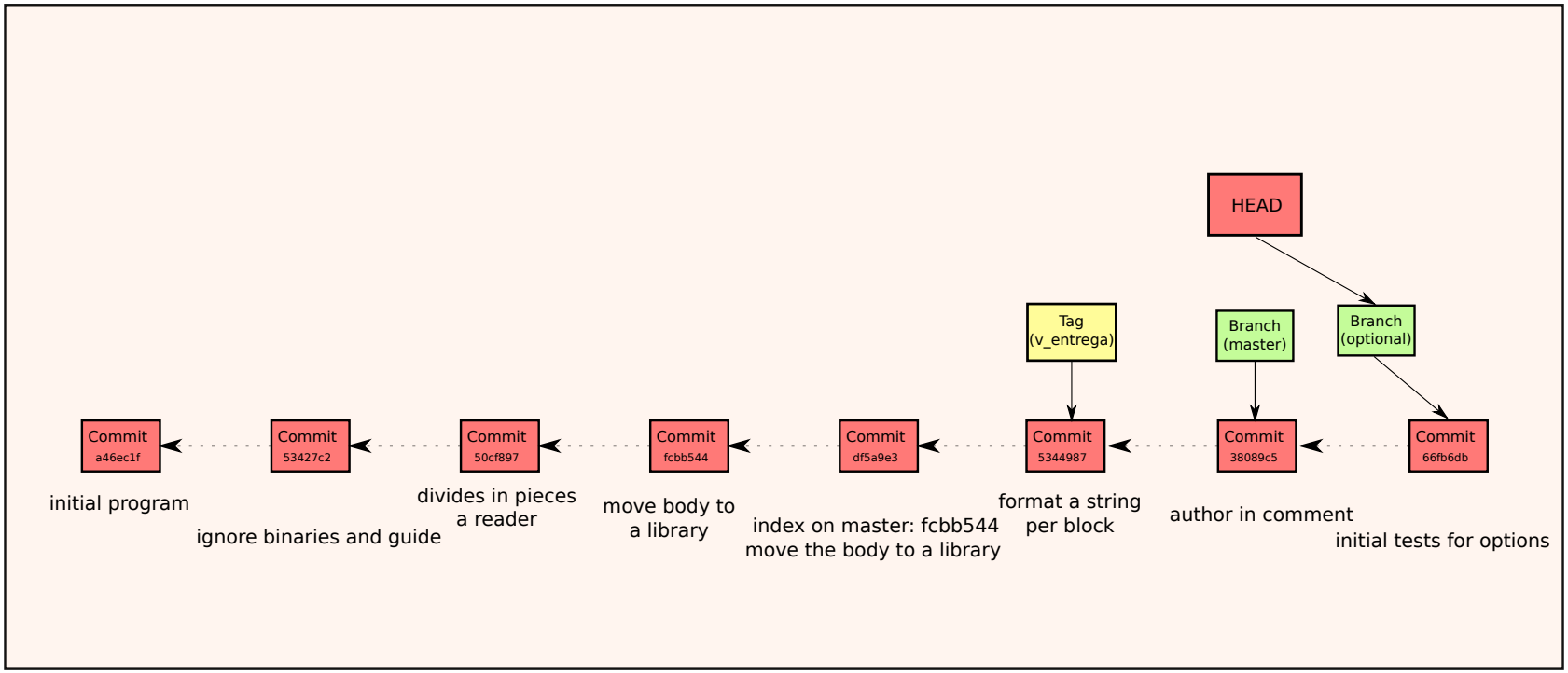
Repositorio hexd



Nueva rama

- Hago un commit a `optional`

Repositorio hexd



Nueva rama

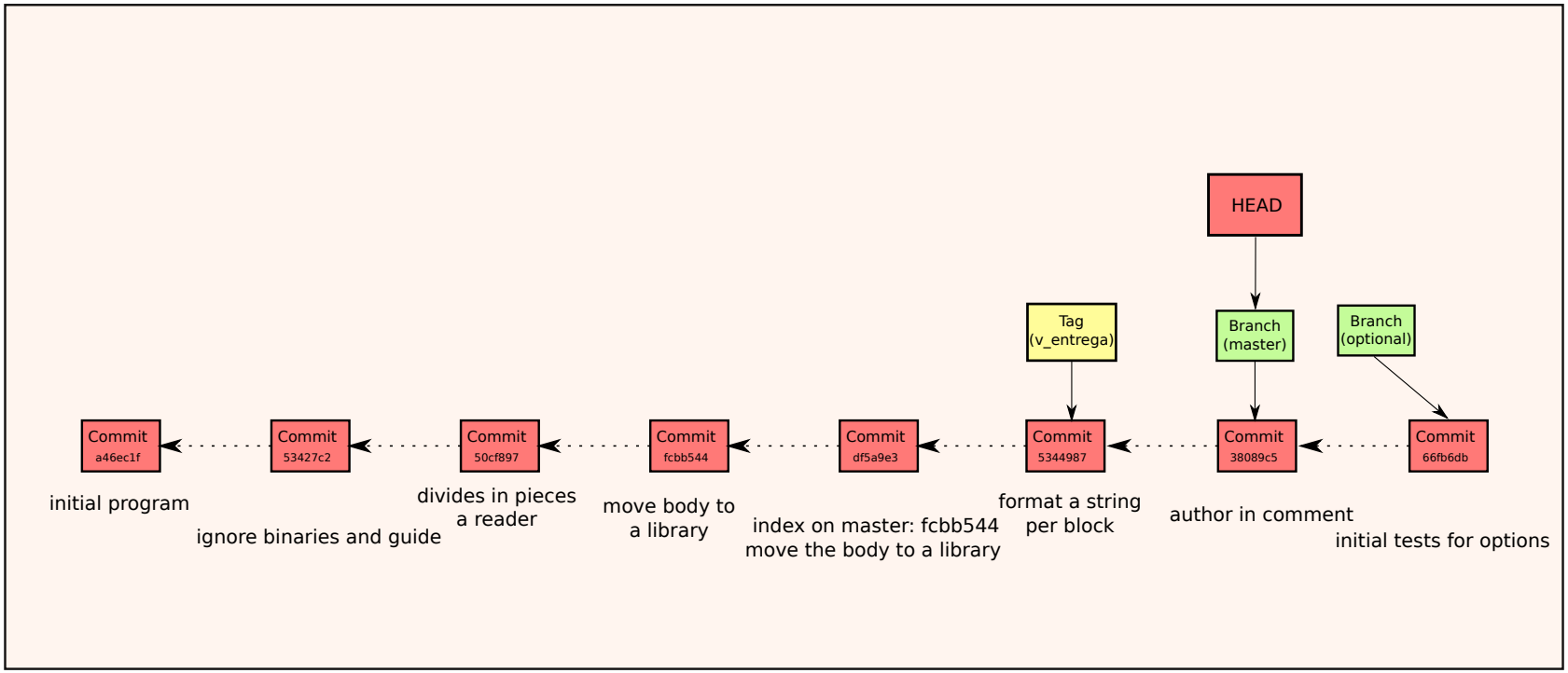
- Vuelvo a la rama principal
- arreglo un bug

```
 $ git adog|sed 3q
* 66fb6db (HEAD -> optional) initial tests for options
* 38089c5 (master) author in comment
* 5344987 (tag: v_entrega) format a string per block
 $ git checkout master
Switched to branch 'master'
 $ git adog|sed 3q
* 66fb6db (optional) initial tests for options
* 38089c5 (HEAD -> master) author in comment
* 5344987 (tag: v_entrega) format a string per block
 $ vi main.go && go test
 $ git add main.go
 $ git commit -m 'bug fix'
[master 2f02b6d] bug fix
 1 file changed, 1 insertion(+)
 $ git adog|sed 6q
* 2f02b6d (HEAD -> master) bug fix
| * 66fb6db (optional) initial tests for options
|/
* 38089c5 author in comment
* 5344987 (tag: v_entrega) format a string per block
* df5a9e3 index on master: fcbb544 move the body to a library
```

Vuelvo a la rama principal

- Hago checkout de `master`

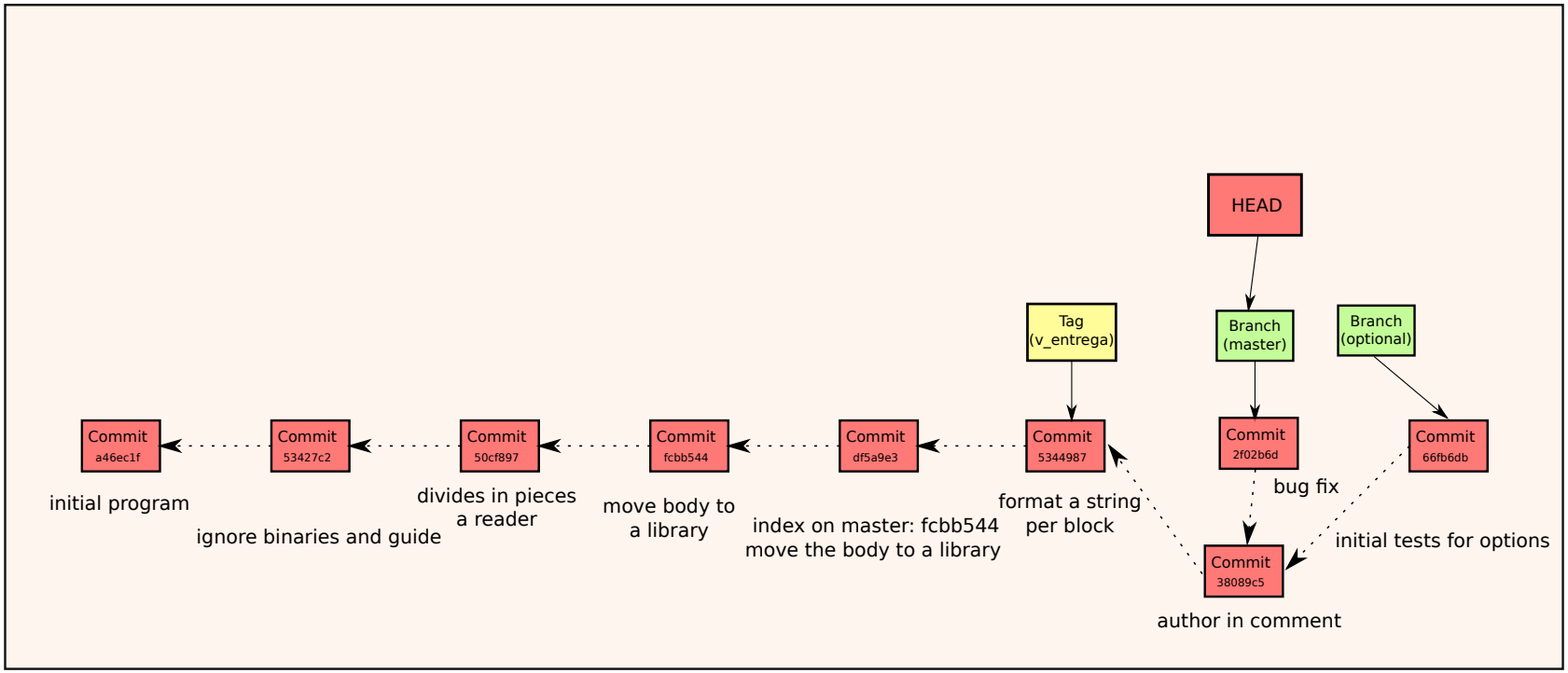
Repositorio hexd



Hago commit en master

- Hago commit en `master`

Repositorio hexd



Junto las dos líneas de desarrollo

Tres formas

- Merge ▸ Merge
- Rebase, (a 3, interactivo) ▸ Rebase
- Cherry pick ▸ Cherry pick

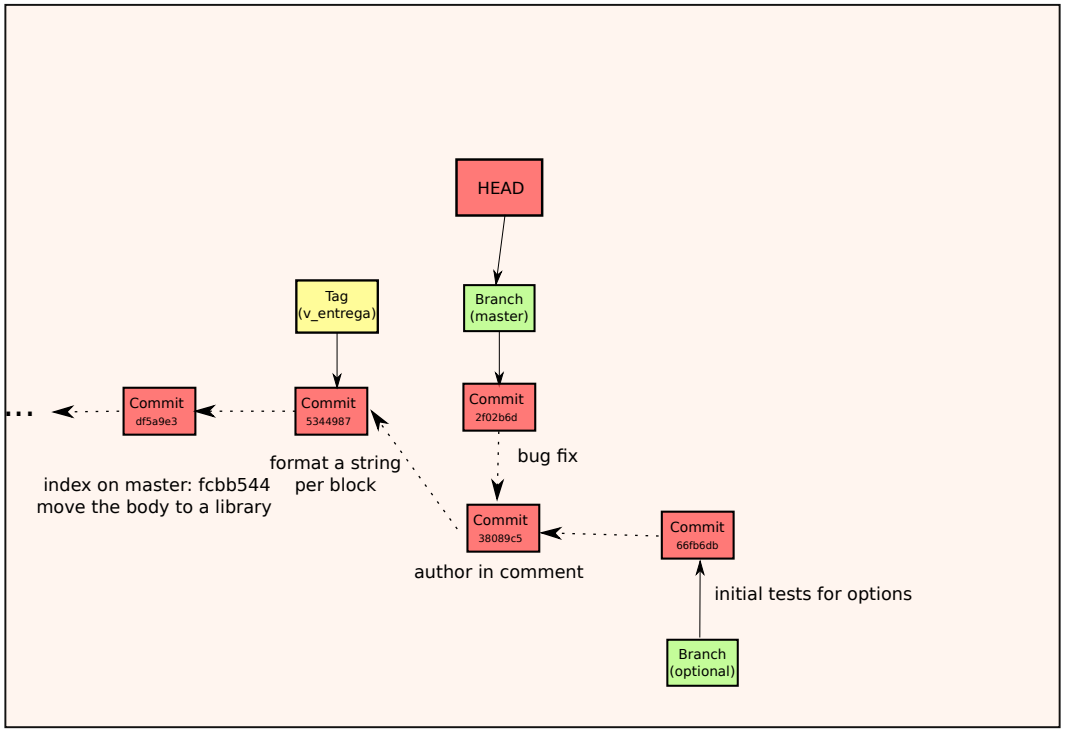
Merge

- Aplicar cambios de una rama en la otra.
- Conservando las dos.
- Queda la rama en el pasado, que se une a la que he mezclado.
- Puede haber conflictos

▸ Merge

Merge

Repositorio hexd

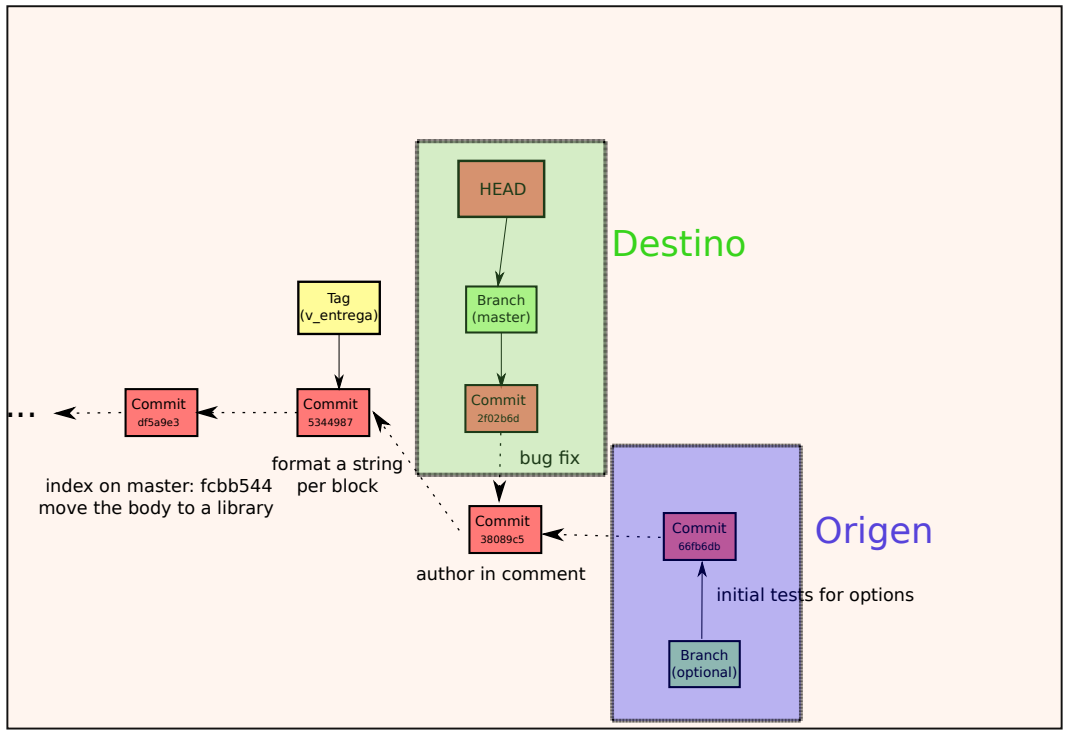


▸ Merge

Merge

```
$ git merge optional master
```

Repositorio hexd

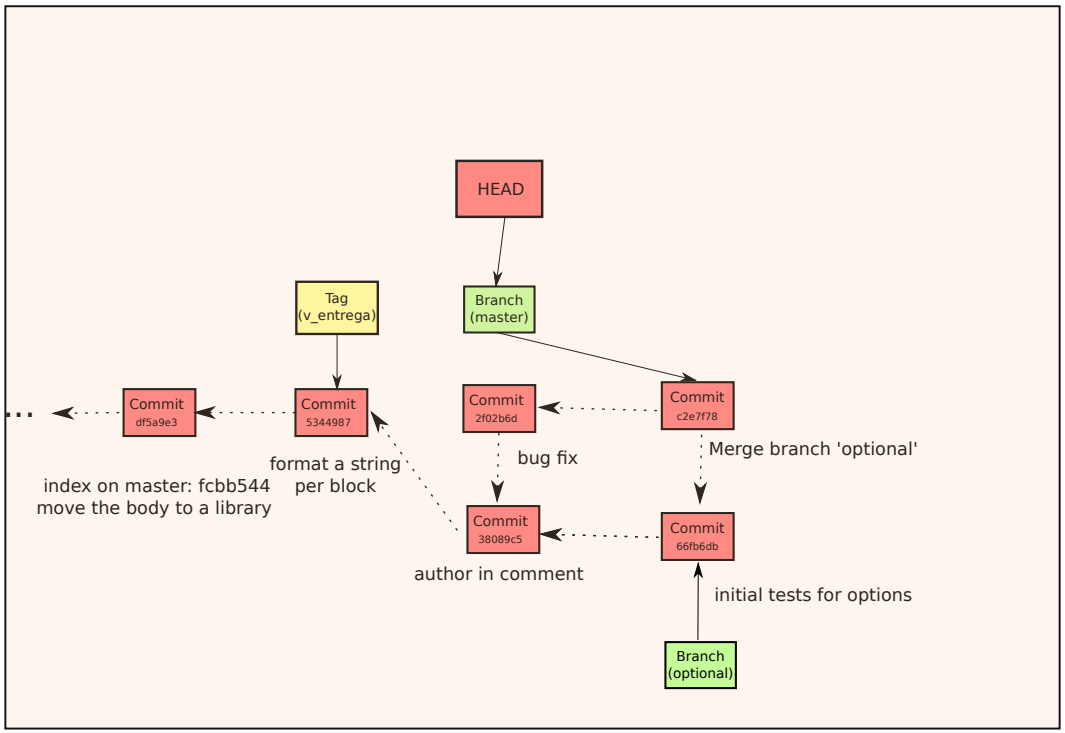


▸ Merge

Merge

```
$ git merge optional master
```

Repositorio hexd



▸ Merge

Merge

```
 $ git adog|sed 7q
*   c2e7f78 (HEAD -> master) Merge branch 'optional'
| | * 66fb6db (optional) initial tests for options
* | 2f02b6d bug fix
|/
* 38089c5 author in comment
* 5344987 (tag: v_entrega) format a string per block
```

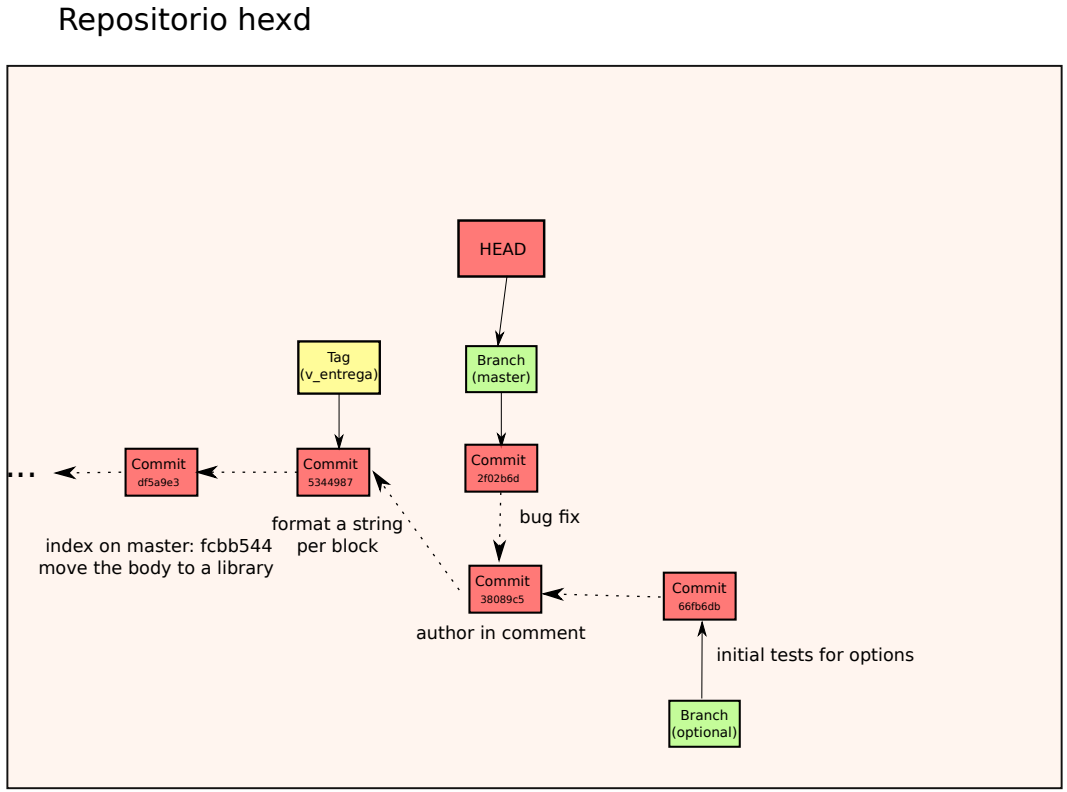
▸ Merge

Rebase

- Normal e interactivo
- Normal: corta una rama y aplica los cambios al final de la otra, al final, una sola línea en todo el tiempo.
- Interactivo: editar los commits, fusionarlos, borrarlos...

▸ Rebase

Rebase Normal

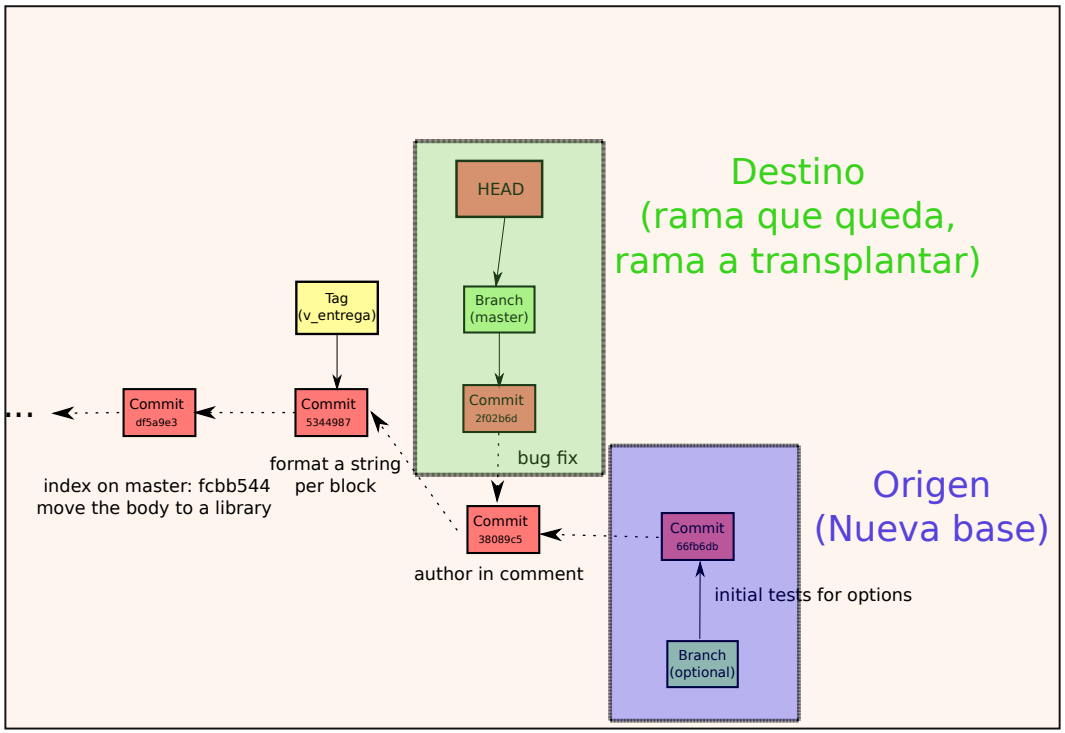


▸ Rebase

Rebase Normal

```
$ git rebase optional master
```

Repositorio hexd

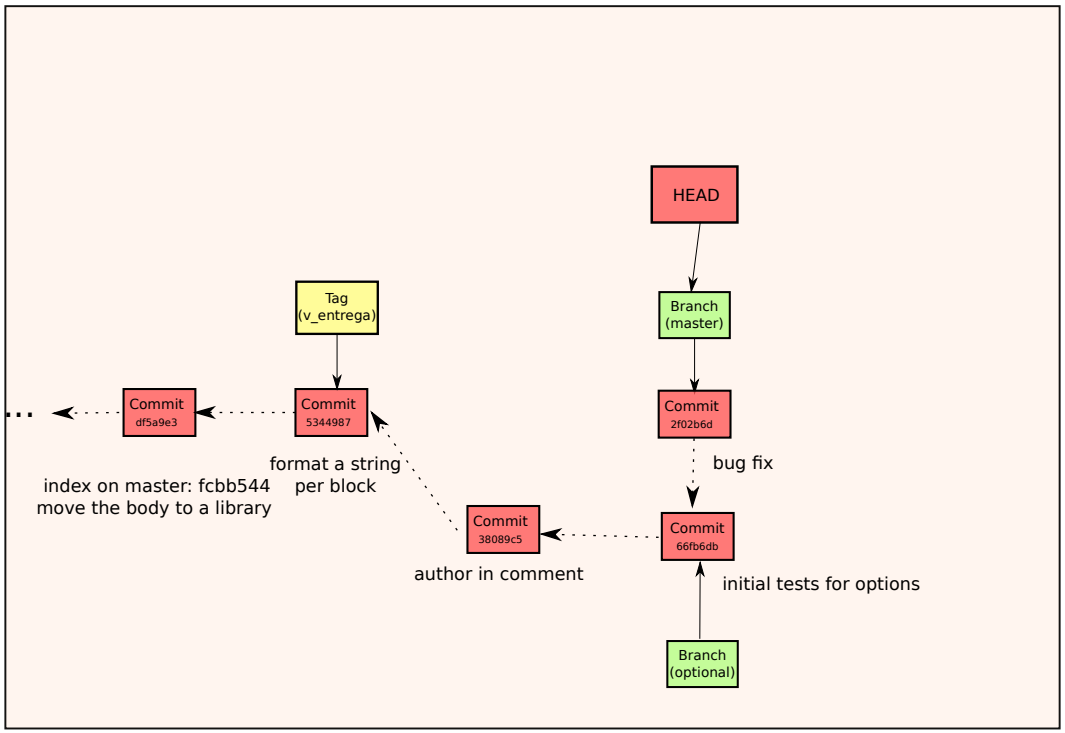


▸ Rebase

Rebase Normal

```
 $ git rebase optional master
* a71b10d (HEAD -> master) bug fix
* 66fb6db (optional) initial tests for options
* 38089c5 author in comment
* 5344987 (tag: v_entrega) format a string per block
* df5a9e3 index on master: fcbb544 move the body to a library
* fcbb544 move the body to a library
* 50cf897 divides in pieces a reader
```

Repositorio hexd



Cherry pick

- Coger commits de otro sitio y aplicarlos.
- Ojo: aparecen repetidos en el repo.

▸ Cherry pick

Conflictos

- Lo que hemos visto antes...
- ...si todo va bien, si no, hay conflictos

▸ Conflictos

Contenido

Tener un par de repos por ssh

- Tengo varias copias por si pasa algo (no exactamente backup, pero ayuda).
- Hace fácil moverse de casa al trabajo/universidad y de vuelta.
- Clono el repo con `git clone` (tengo ramas locales y remotas)

  ▸ Git clone
- Funciona igual que local, pero hay ramas remotas, concepto de fetch (o pull, que es fetch+merge) ▸ Ramas remotas

▸ Repo por ssh

Contenido

Repo central

- Hay un repo de desarrollo (personal, central).
- Concepto de repo bare ▸ Repo Bare
- Por ejemplo en GitHub/GitLab.
- Típicamente se clona, se añaden commits, se hace una Pull Request.

▸ Pull Request

Parte II

Mecanismos

Contenido

¿Qué es Git?

- ¿Alguna vez, has?...
  - Hecho revisiones de un programa (ej. una práctica)
  - Mezclado partes de un programa escrito por varios (ej. una práctica en grupo)

¿Qué es Git?

- ¿Alguna vez, has?...

```
version_final
version_final_1.0
version_final_1.0_buena_final
version_entregar_no_funciona
version_buena_final_entregar_final
```

¿Qué es Git?

- ¿Alguna vez, has?...

```
version_con_cambios_juan_pedro
version_con_cambios_juan_func_strings
version_con_cambios_juan_pedro_func_strings_falla
aaarrrhhhhghhhhhh
```

¿Qué es Git?

- Un VCS (sistema de control de versiones)
- Distribuido
- Creado por Linus Torvalds (el de Linux) en 2005
- Para desarrollar el kernel de Linux, sustituir Bitkeeper

VCS

- Sirve para sincronizar trabajo el fuente
- Hacer, deshacer, mezclar cambios, agruparlos
- Tener variantes
- Preservar la historia del desarrollo
- Evitar: version_definitiva.1, version_redefinitiva_1.2...
- Colaborar entre varios (proyectos de software libre...)

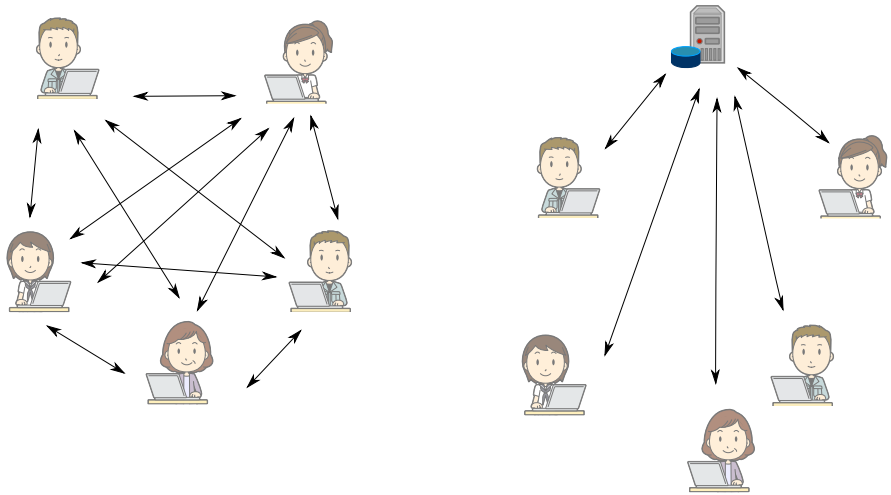
VCS

- Importante, agrupar un conjunto de cambios (commit)
- Manejarlos como una transacción, de forma atómica
- Puede haber conflictos, forma de resolverlos (cambios incompatibles)

Distribuido

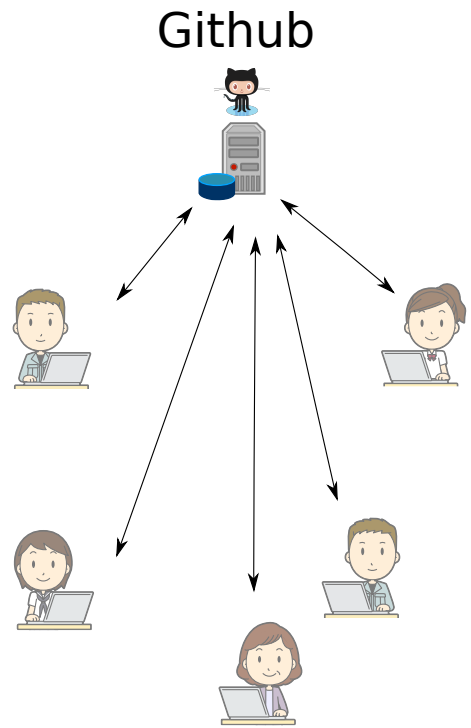
- Puedo tener varios árboles, de ficheros hay un servidor central (peers)
- Cada uno trabaja en local y luego sincronizan
- Es un conjunto de bosques de árboles de ficheros en diferentes máquinas

Distribuido v.s. Centralizado



Distribuido

- Muchas veces cómodo tener centralizado
- Por organización, simplicidad, disponibilidad, backups...
- Hay servicios de hosting de Git (Github, Gitlab)



¿Qué no es Git?

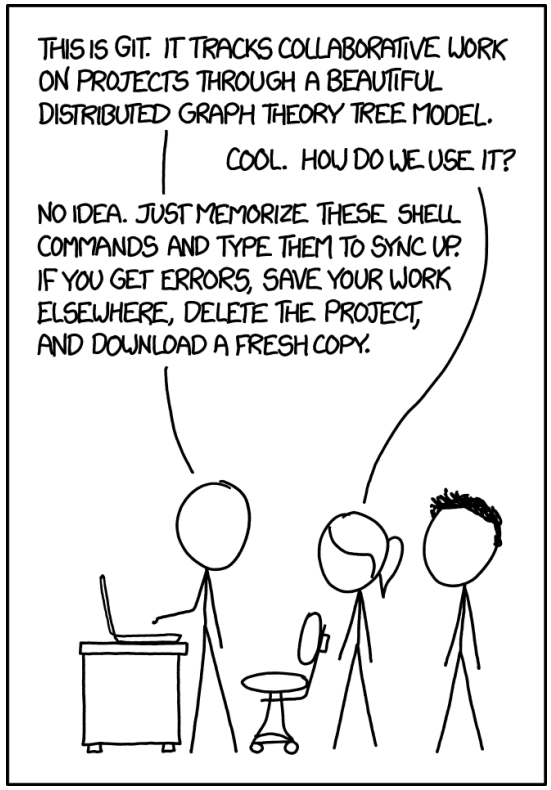
- Git no es un sistema de backup general
- Puede servir de backup para el fuente (aunque requiere intervención del usuario)
- Git no es bueno con binarios

¿Qué cosas puedo hacer?

- Añadir y modificar (editar) un fichero o varios
- Crear y mezclar un branch (rama o variante del código fuente) con y sin conflictos
- Ver la historia/historial de cambios
- Deshacer un conjunto de cambios (commit)
- Compartir, sincronizar el código con un repositorio remoto o central

Uso de Git

Para evitar



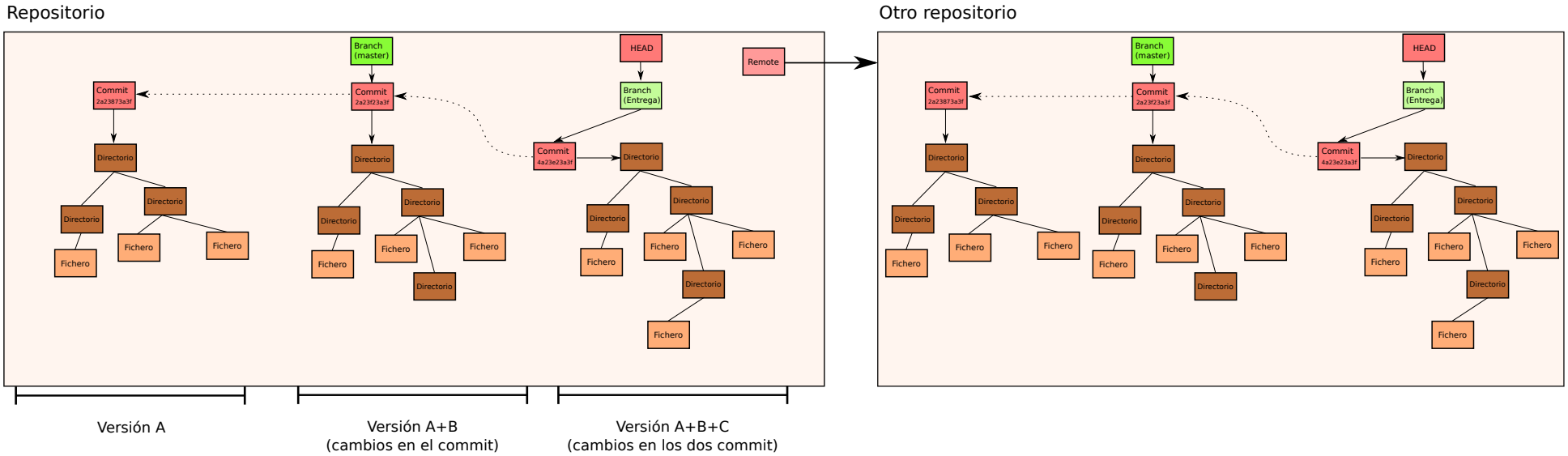
Documentación

- https://juristr.com/blog/2013/04/git-explained/
- https://git-scm.com/book/en/v2
- https://codewords.recurse.com/issues/two/git-from-the-inside-out
- Los subcomandos de git son `git XXX` entonces: `man git-XXX`
- Por ejemplo para mirar como hacer `git clone` mirar: `man git-clone`
- Los XXX (clone en el ejemplo de arriba) se suelen llamar verbos (*verbs* en la terminología de Git)

Terminología

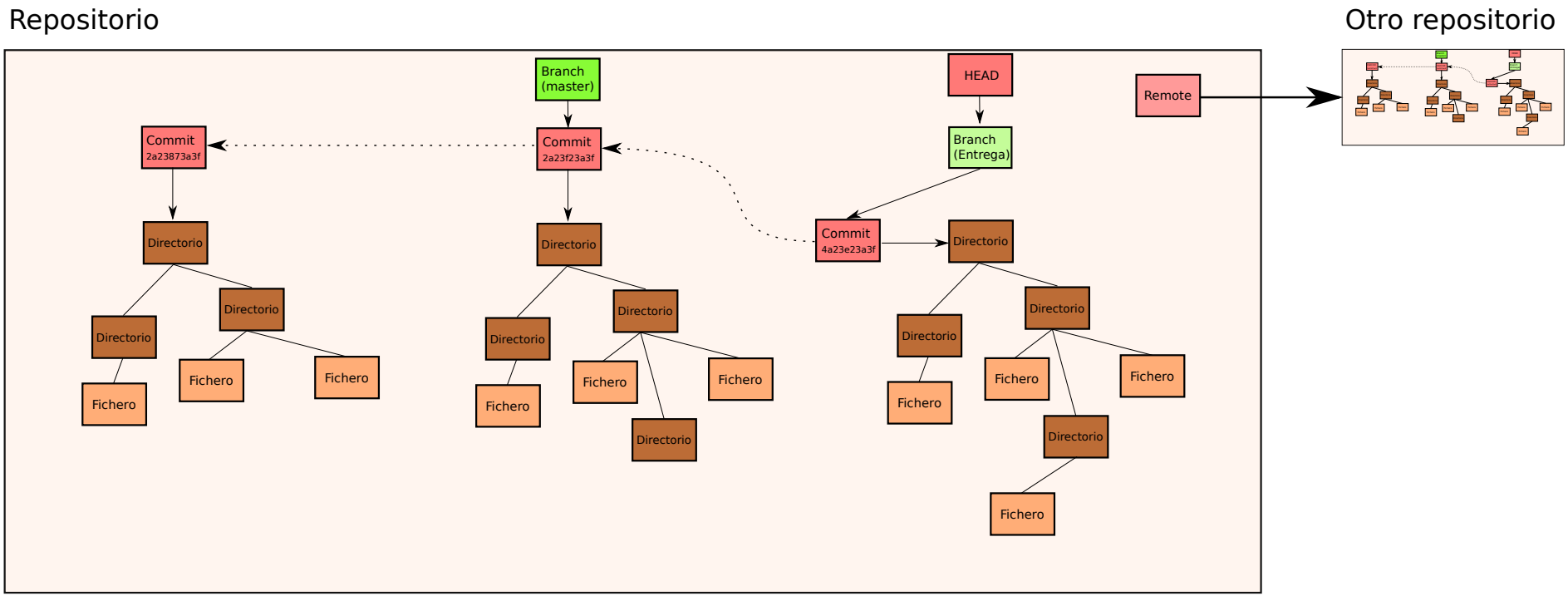
- **master**: branch principal del repositorio (de trabajo o integración)
- **clone**: copia un repositorio diferente (puede ser remoto) a un sitio local
- **commit**: manda cambios al repositorio local (en otros VCS, checkin). Un **commit** es un conjunto de cambios que se aplican juntos.
- **fetch** o **pull**: trae cambios de un repositorio diferente (o remoto), fetch trae, pull hace merge además (otros VCS, update o get latest)
- **push**: manda cambios al repositorio diferente, (puede ser remoto)
- **head**: referencia al nodo con el que estamos trabajando
- **branch** : etiqueta de un nodo (rama de código)

Vista del usuario



No están dibujadas las ramas remotas (veremos más adelante)

Vista del usuario

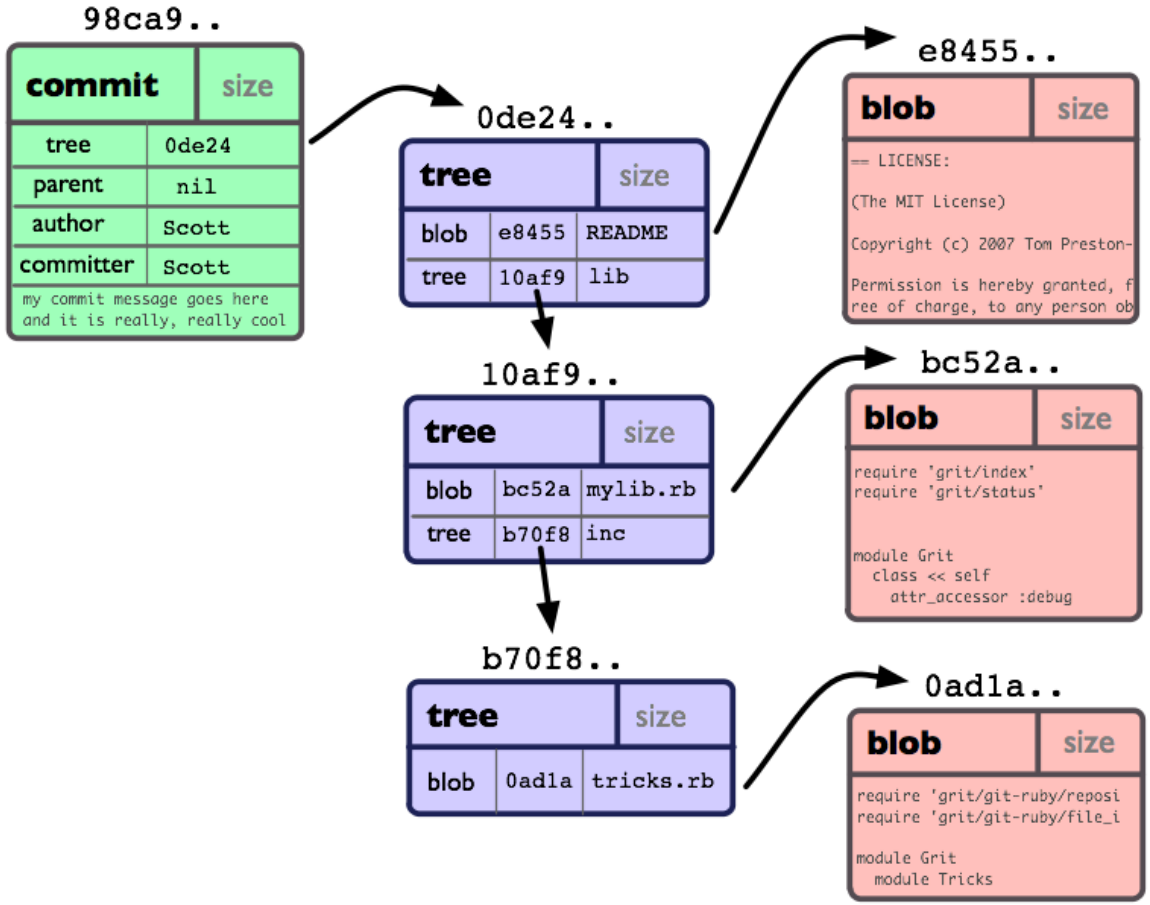


Por dentro

- Ficheros y directorios representados como blobs de bytes (y trees)
- Direccionados por hash del contenido (SHA)
- Un directorio: un **tree**, una lista de SHAs con nombres de ficheros y directorios contenidos
- Un ficheros es un **blob** de bytes: contenido
- Unas cuantas cosas más: head, etiquetas de branch...
- Metidos en directorios, hay varias formas, dentro del directorio `.git`
- Como un sistema de ficheros overlay con metadatos extra
- Se maneja con comandos `git clone`, `git pull`...

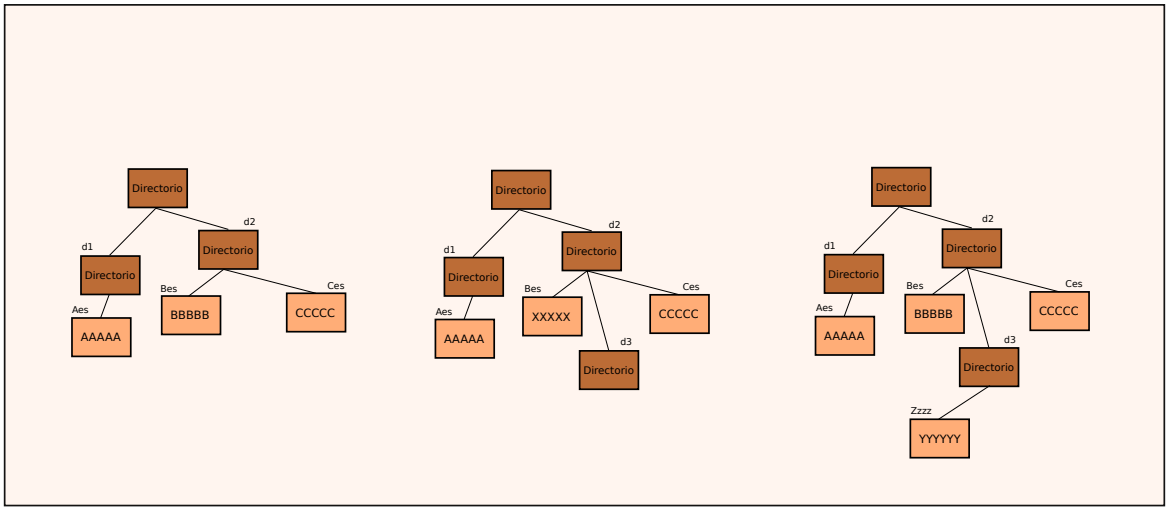
Por dentro

http://shafiulazam.com/gitbook/1_the_git_object_model.html

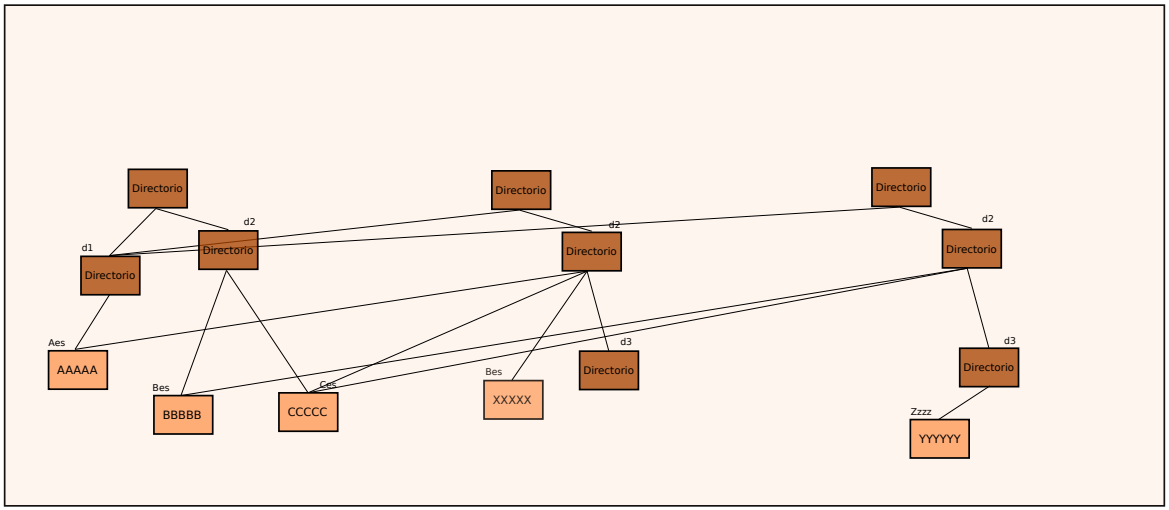


Por dentro

Repositorio



Por dentro



Por dentro

- Direccionado por contenido (hash SHA del contenido)
- El nombre verdadero (directorio o fichero) es una SHA (Un chorro hexa: $10de72403bd035e6dfe5a8bc07f4e57aef0e4e93$)
- Número obtenido de los nombres y contenidos de los subdirectorios y ficheros
- Estos a su vez de sus subdirectorios...
- Merkle tree (un cambio en un fichero, cambia la sha en el directorio padre, esta en la de su abuelo...)
- Normalmente, se añade cosas, incrementos (no se puede cambiar, al cambiar, se crea una copia nueva del subárbol hasta el raíz
- Ficheros con conjuntos de incrementos (formato packfile, además comprimido)

Por dentro

- Eficiente, sólo se guardan los cambios
- Se pueden ver desde el raíz

Contenido

Configuración

- `man git-config`
- Está en varios sitios, se miran en orden:

General:

```
$(prefix)/etc/gitconfig
$XDG_CONFIG_HOME/git/config
$HOME/.config/git/config
$HOME/.gitconfig
```

Local al repositorio:

```
.git/config
```

Configuración

- Aquí es la que se suele llamar **global**: `git config --global`
- Ahí es donde solemos configurar como usuario
- Usamos el comando

```
$HOME/.gitconfig
```

Configuración

```
$ git config --global user.name "Gorka Guardiola"
$ git config --global user.email "paurea@gmail.com"
```

Alias

- Se pueden definir comandos para que resulte mas cómodo

```
$ git config --global alias.adog "log --all --decorate --oneline --graph"
```

Configuración editor

- El que yo quiera: vim, atom...

```
$ git config --global core.editor emacs
```

Configuración mezclar cambios

- El que yo quiera, emerge, gvimdiff, p4merge...
- Para editar los ficheros viendo las diferencias
- Me puedo hacer un script que abra mi editor como yo quiera
- Por ejemplo el script extMerge, pongo un script que reciba los argumentos y a correr customMergeTool
- Git le pasa al comando: path old-file old-hex old-mode new-file new-hex new-mode
- `man git-config`

```
#caso normal, herramienta ya presente
$ git config --global merge.tool emerge
#para un programa propio o script:
$ git config --global merge.tool customMergeTool
$ git config --global mergetool.customMergeTool.cmd 'mergeacme \"$BASE\" \"$LOCAL\" \"$REMOTE\" \"$MERGED\"'
```

Contenido

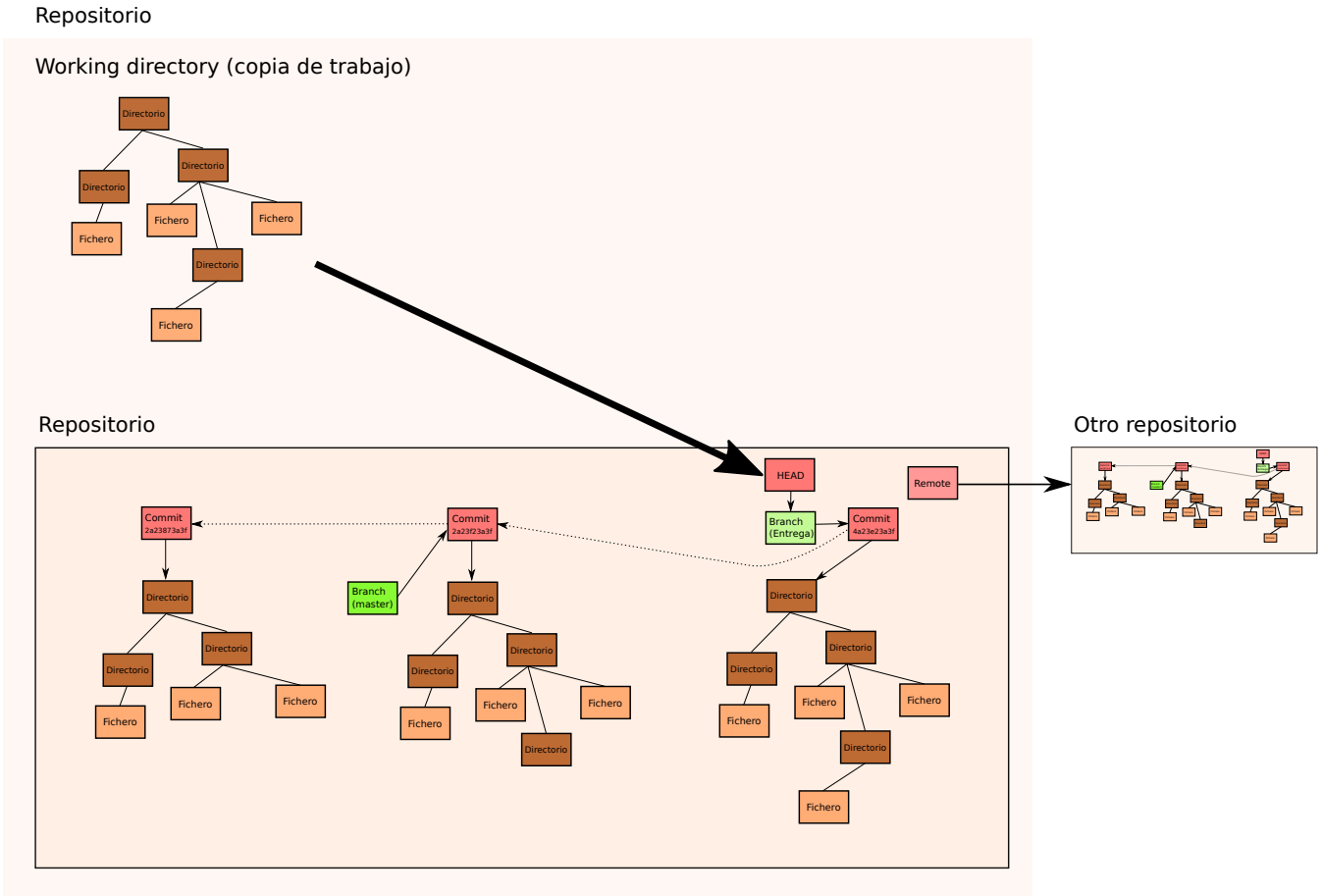
Inicialización

- Git guarda su estado en un subdirectorio `.git`
- Ahí van todos los árboles, el head...
- En el directorio fuera del `.git` tendremos nuestro espacio de trabajo

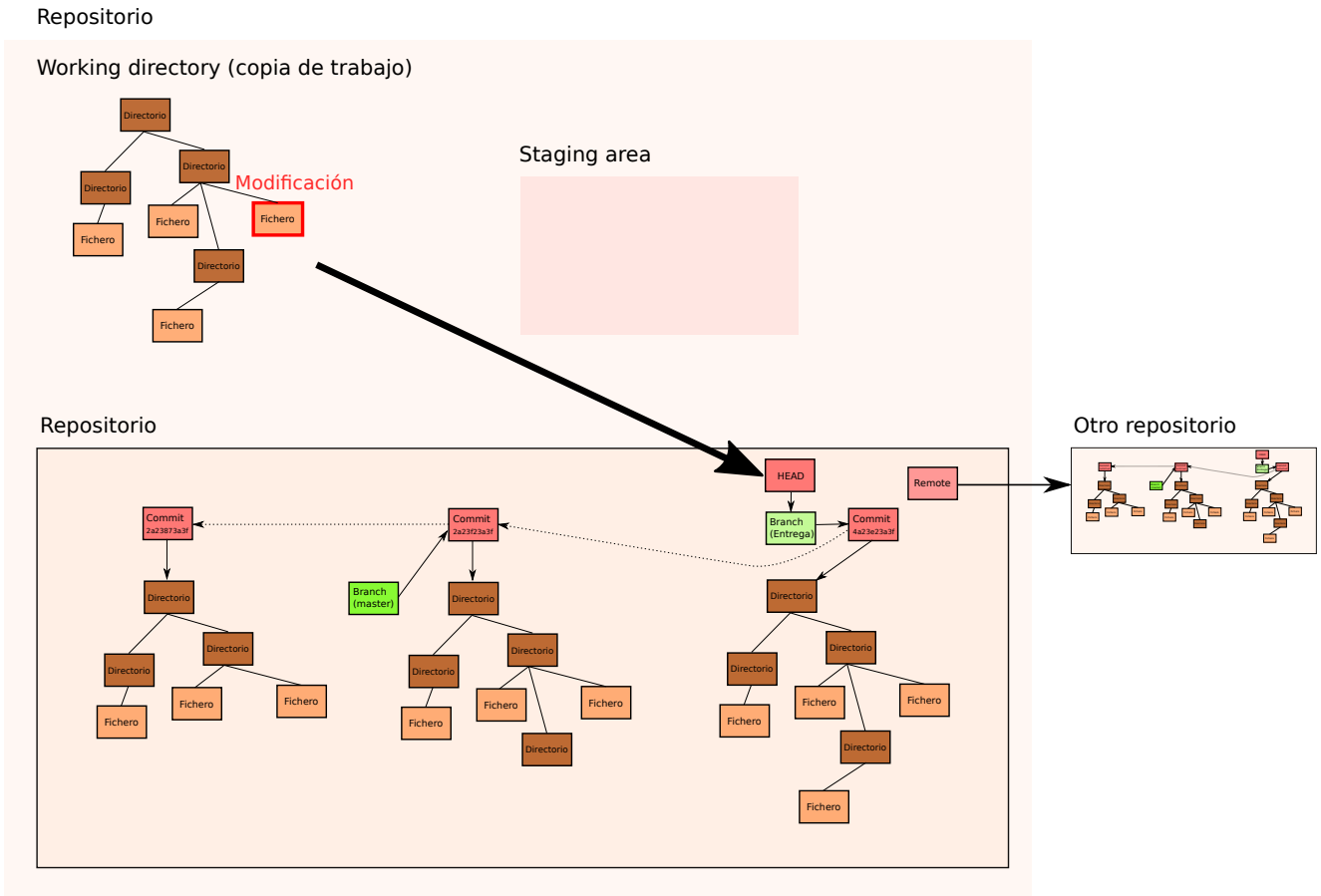
```
$ git init gitrepo
Initialized empty Git repository in /home/paurea/gitrepo/.git
```

Contenido

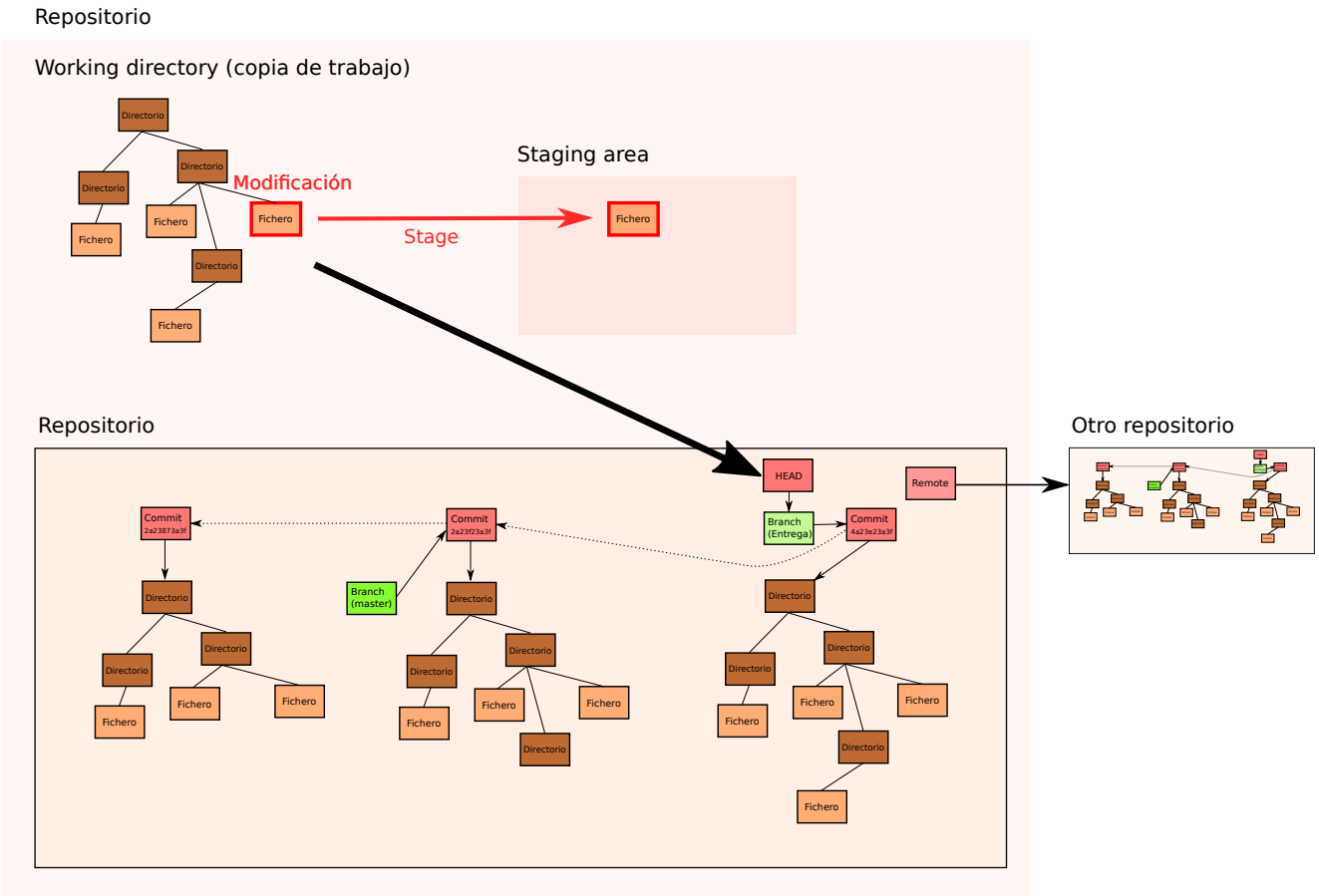
Cómo se usa


Repositorio
Working directory (copia de trabajo)
Directorio
Directorio
Directorio
Fichero
Fichero
Fichero
Directorio
Fichero
Repositorio
Otro repositorio
HEAD
Remote
Commit
2a23873a3f
Commit
2a23f23a3f
Branch
(Entrega)
Commit
4a23e23a3f
Branch
(master)
Directorio
Fichero

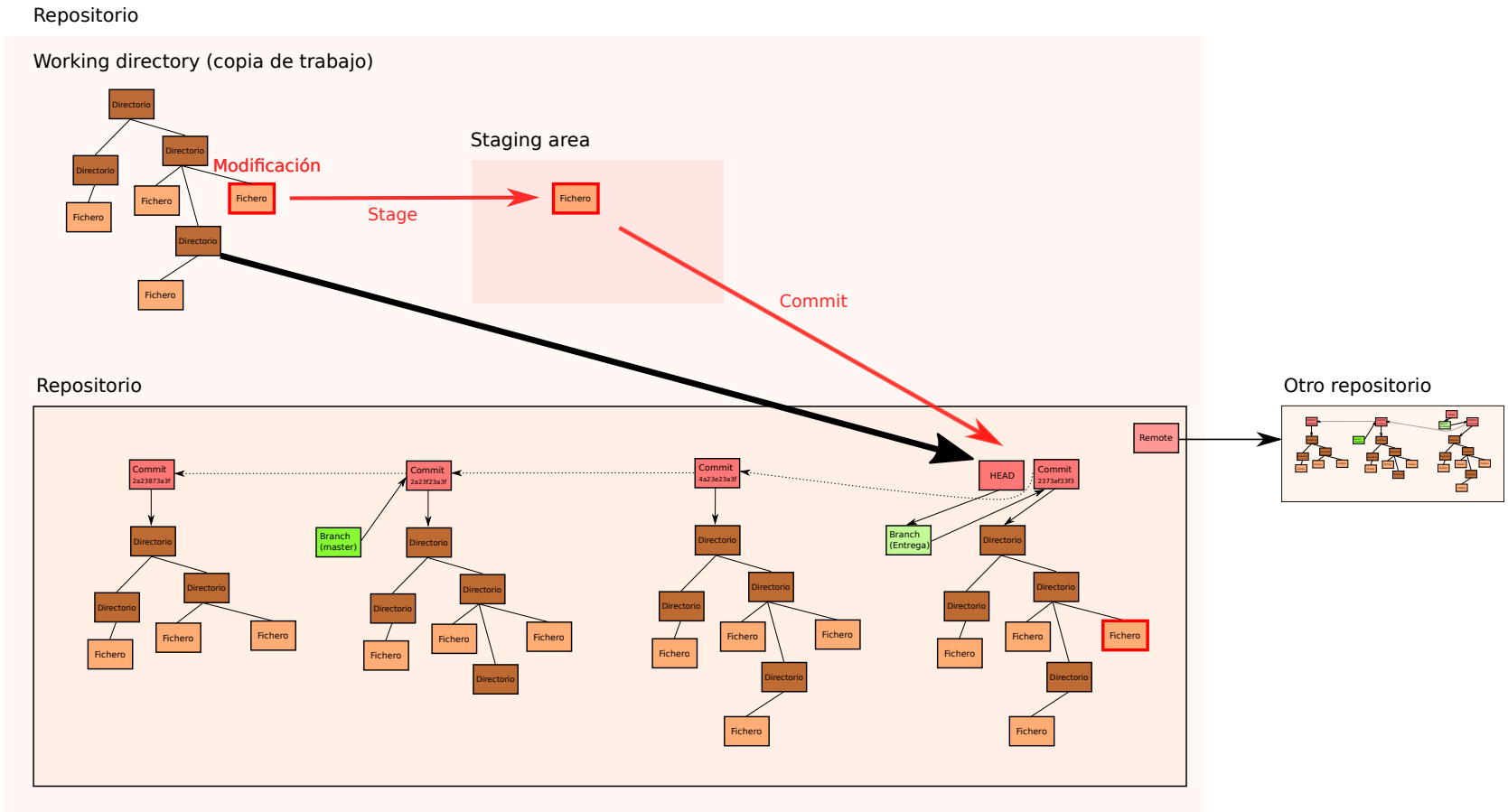
Modifico fichero(s): **git add ficheros; editor ficheros**



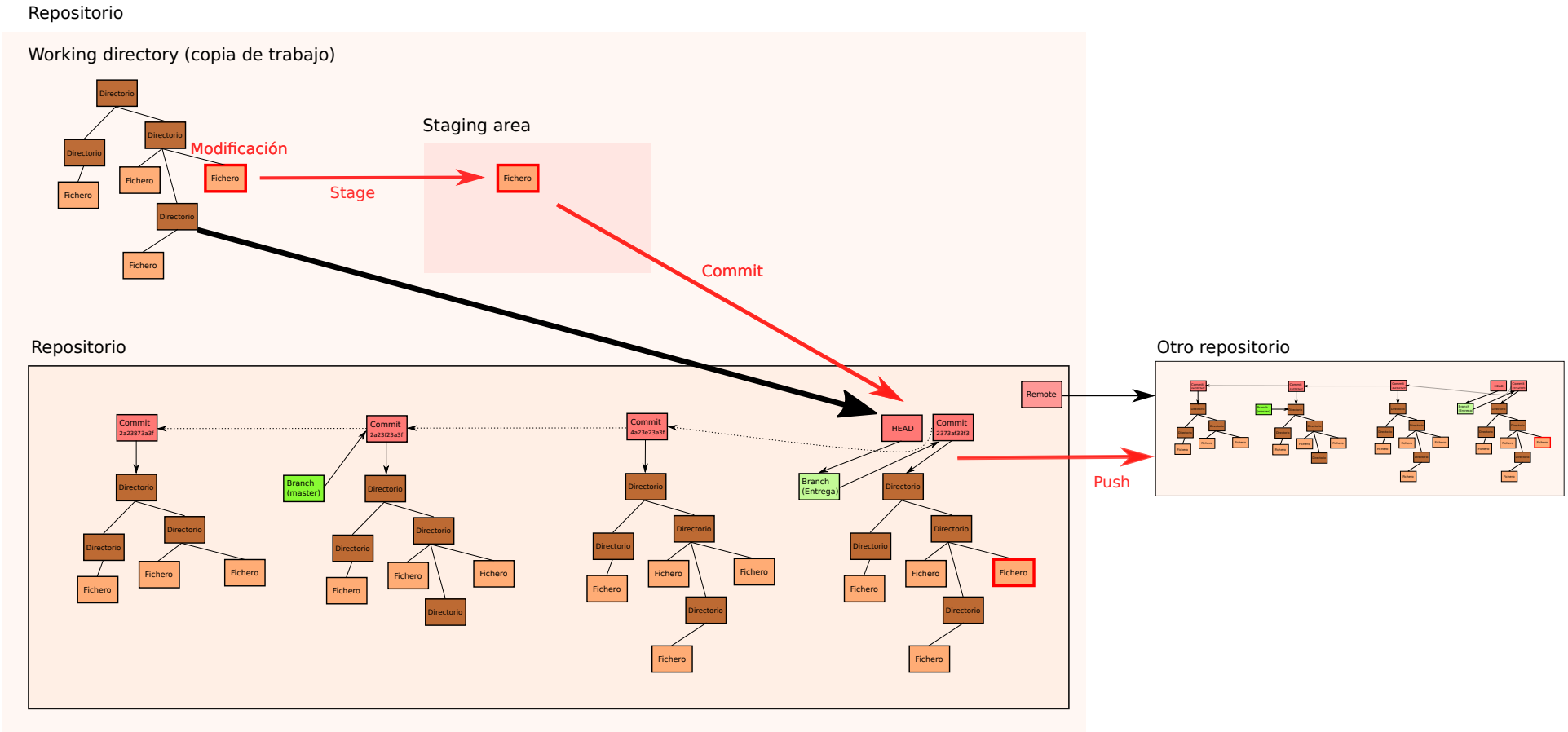
Los paso a stage: **git add ficheros**



Hago commit al repo local: **git commit**

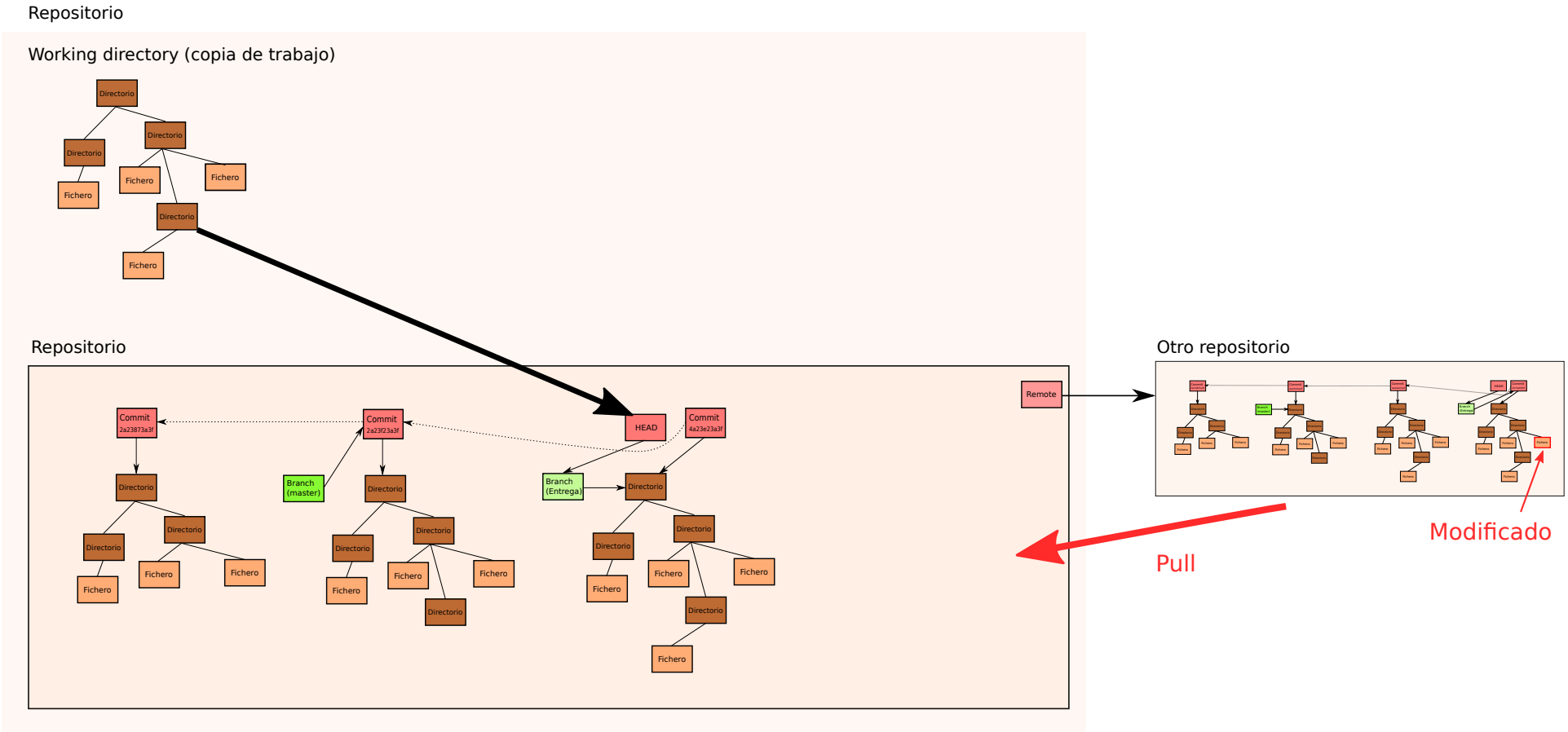


Hago push al repo remoto: **git push**



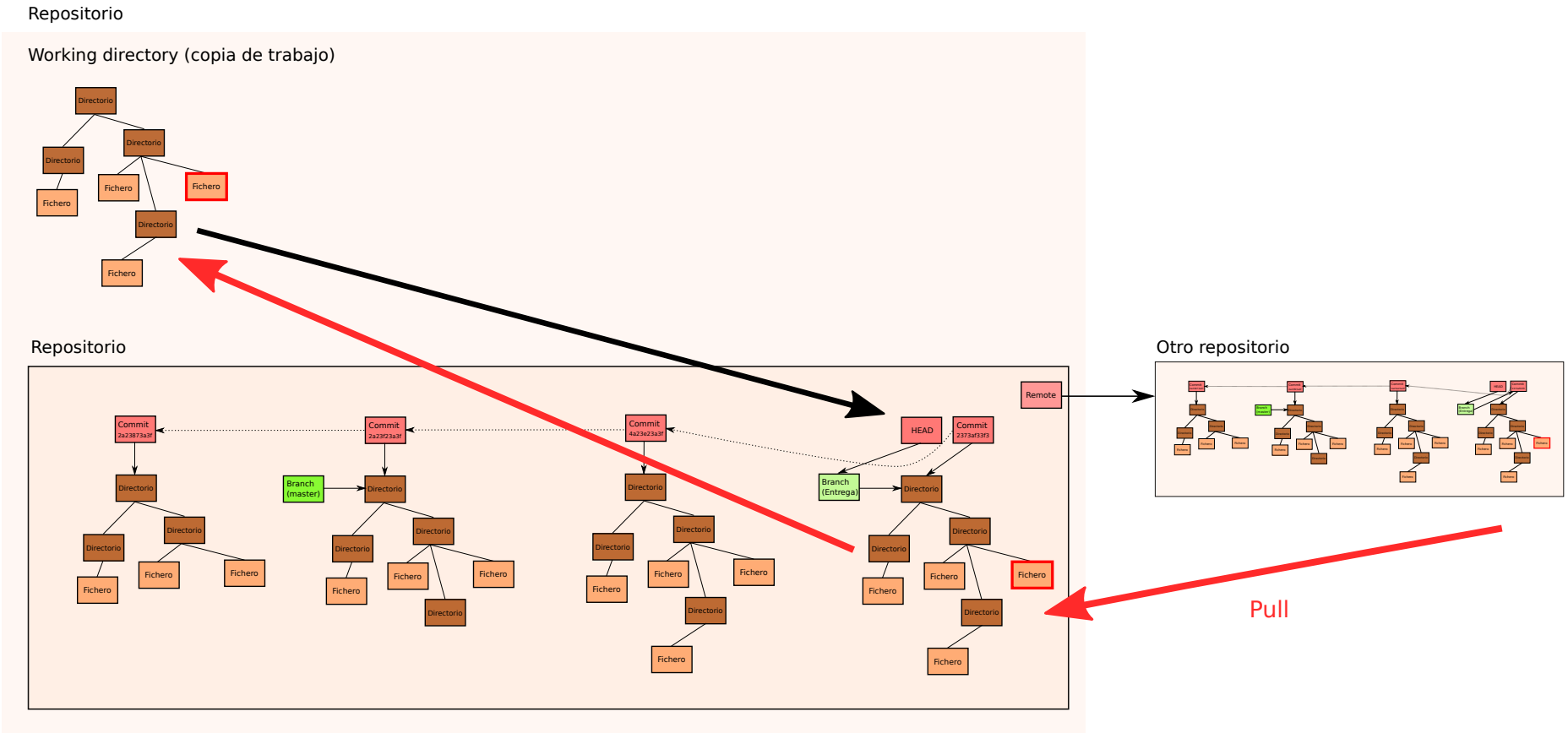
No están dibujadas las ramas remotas (veremos más adelante)

Hago pull para traer cambios del repo remoto: **git pull**



No están dibujadas las ramas remotas (veremos más adelante)

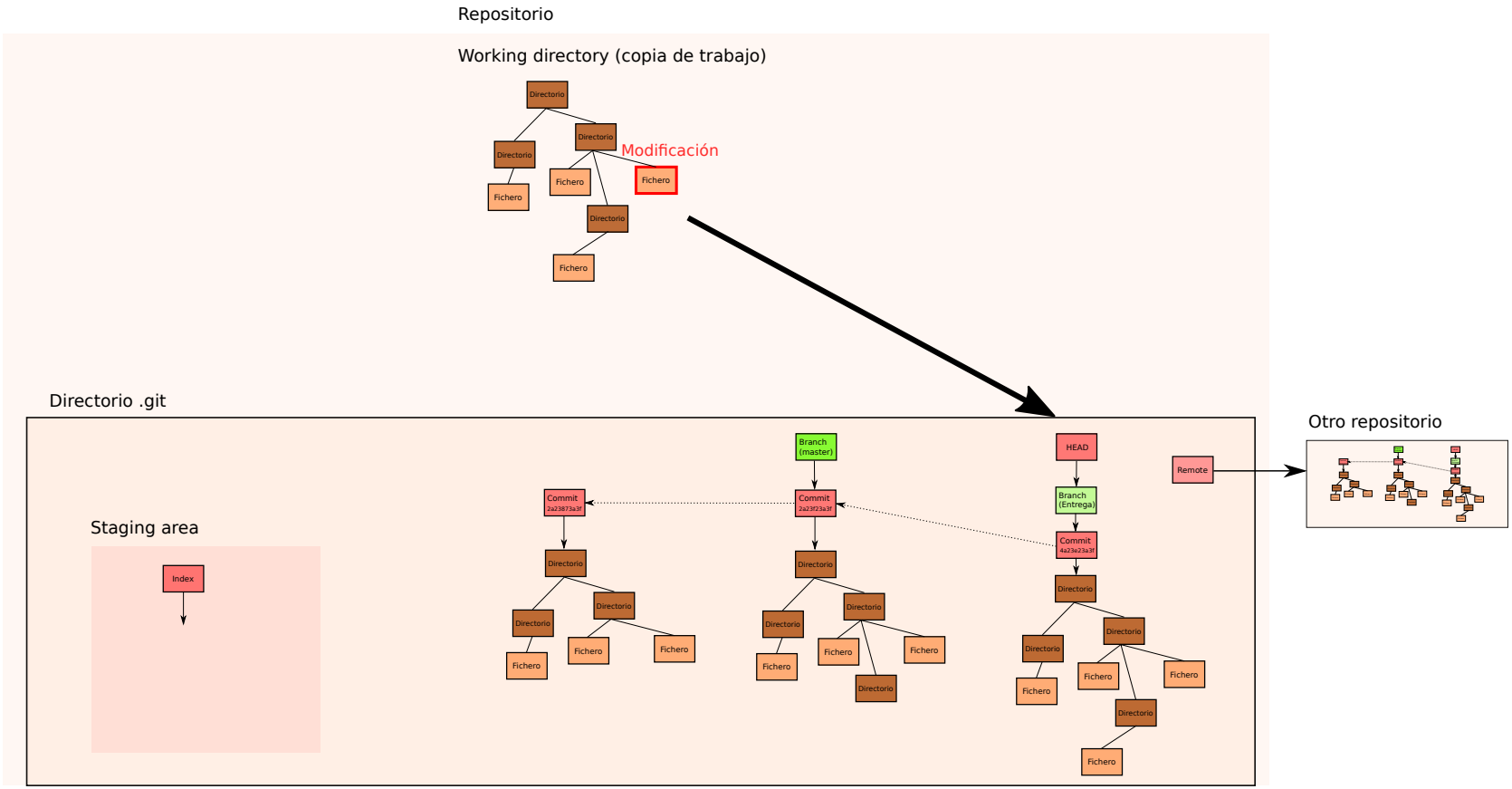
Hago pull para traer cambios del repo remoto



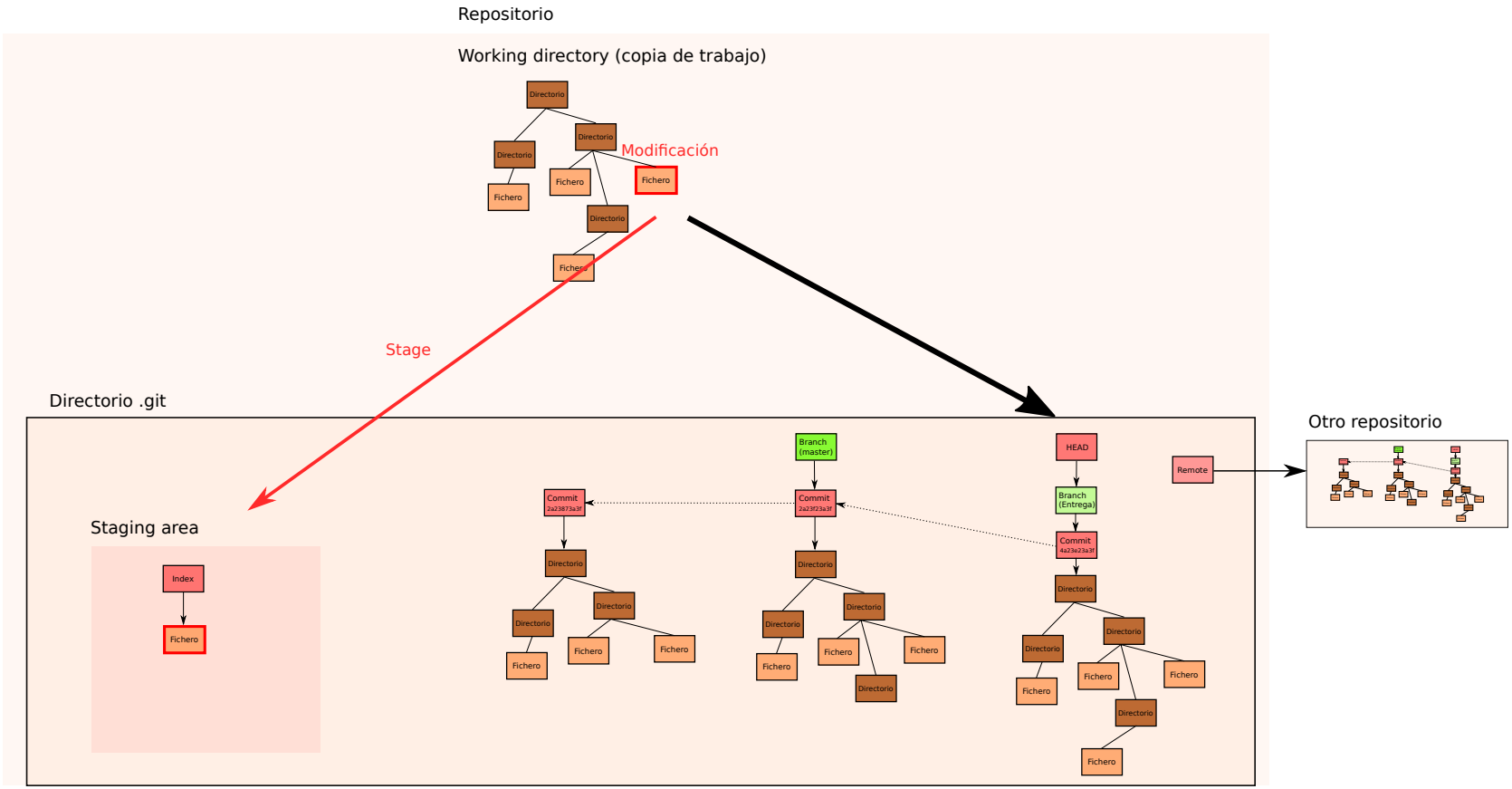
En realidad

- Staging area parte del repositorio, pero no permanente (en .git)
- También se llama índice o caché
- Hay un fichero index con los fichero a añadir
- Los blobs están en el repo, pero no forman parte de un commit (temporal)
- Commit los hace parte de un commit
- El staging lo deja todo preparado (reserva todos los recursos)

Stage area (realidad)



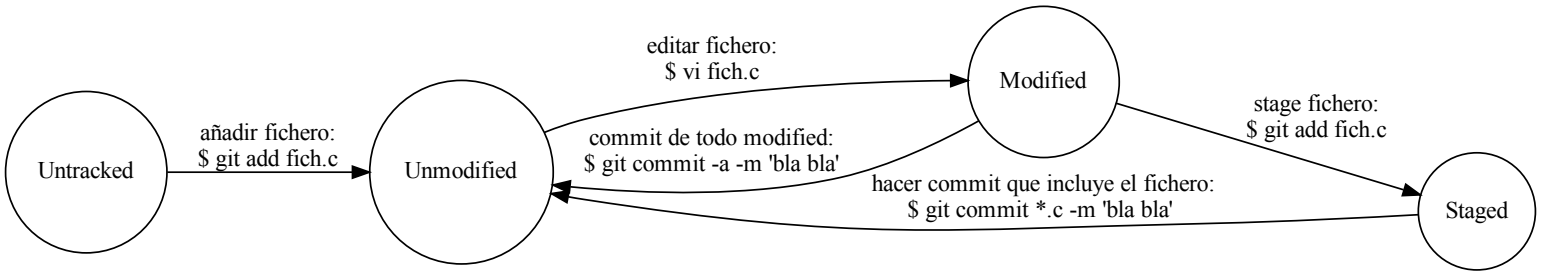
Stage area (realidad)



Estados de un fichero

- En mi directorio de trabajo
- untracked, unmodified, modified, staged
- Si a git no le importa: untracked
- Si a git le importa:
  - unmodified: está en el último commit, al día
  - modified: ha cambiado desde el último commit
  - staged: listo para hacer commit

Estados de un fichero



Contenido

Sesión

- Creo un repositorio

```
$ mkdir z
$ cd z
$ git init
Initialized empty Git repository in /home/paurea/z/.git/
```

Sesión

- Añado un fichero

```
$ touch x.c
$ git add x.c
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   x.c
```

- Añado un fichero, paso a untracked

```
$ git commit -m 'creo el primer fichero' x.c
[master (root-commit) 9657128] creo el primer fichero
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 x.c
$ git status
On branch master
nothing to commit, working tree clean
```

Sesión

- Paso a tracked
- No dice nada porque no hay cambios

```
$ git add x.c
$ git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

Sesión

- Modifico, paso a modified

```
$ vi x.c
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
        modified:   x.c
no changes added to commit (use "git add" and/or "git commit -a")
```

Sesión

- Paso a staged

```
$ git add x.c
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   x.c
$ git commit -m 'modifico el primer fichero' x.c
[master 540b592] modifico el primer fichero
 1 file changed, 2 insertions(+)
$ git status
On branch master
nothing to commit, working tree clean
```

Sesión

- Paso a staged

```
$ git log
commit 540b592ecd4229ff59c0c1f85cc0c68c3fd843e4 (HEAD -> master)
Author: paurea <paurea@gmail.com>
Date:   Thu Nov 15 13:25:20 2018 +0100

    modifico el primer fichero

commit 9657128cccb82061afa8d626af9f9eee8f69a3bd
Author: paurea <paurea@gmail.com>
Date:   Thu Nov 15 13:24:37 2018 +0100

    creo el primer fichero
```

Contenido

Fichero `.gitignore`

- Fichero `.gitignore` en el raíz del proyecto
- # **Sólo al principio**, comenta la línea
- Ficheros que git no considera (ni siquiera para untracked)
- Usa globbing, dos tipos de patrones, path, glob en general
- Si no contiene / (salvo al final) es un patrón de globbing en cualquier sitio
- El raíz es el del proyecto (/ es el directorio principal)
  - Los directorios (si queremos sólo dir) se indican con / al final
  - (! hace que el patrón se deje de ignorar i.e. lo niega)
  - Línea en blanco no hace nada
  - La barra invertida \ escapa
  - Si contiene / es un path desde el raíz (con globbing)
  - ** encaja con cualquier subpath, incluyendo las /

Fichero .gitignore

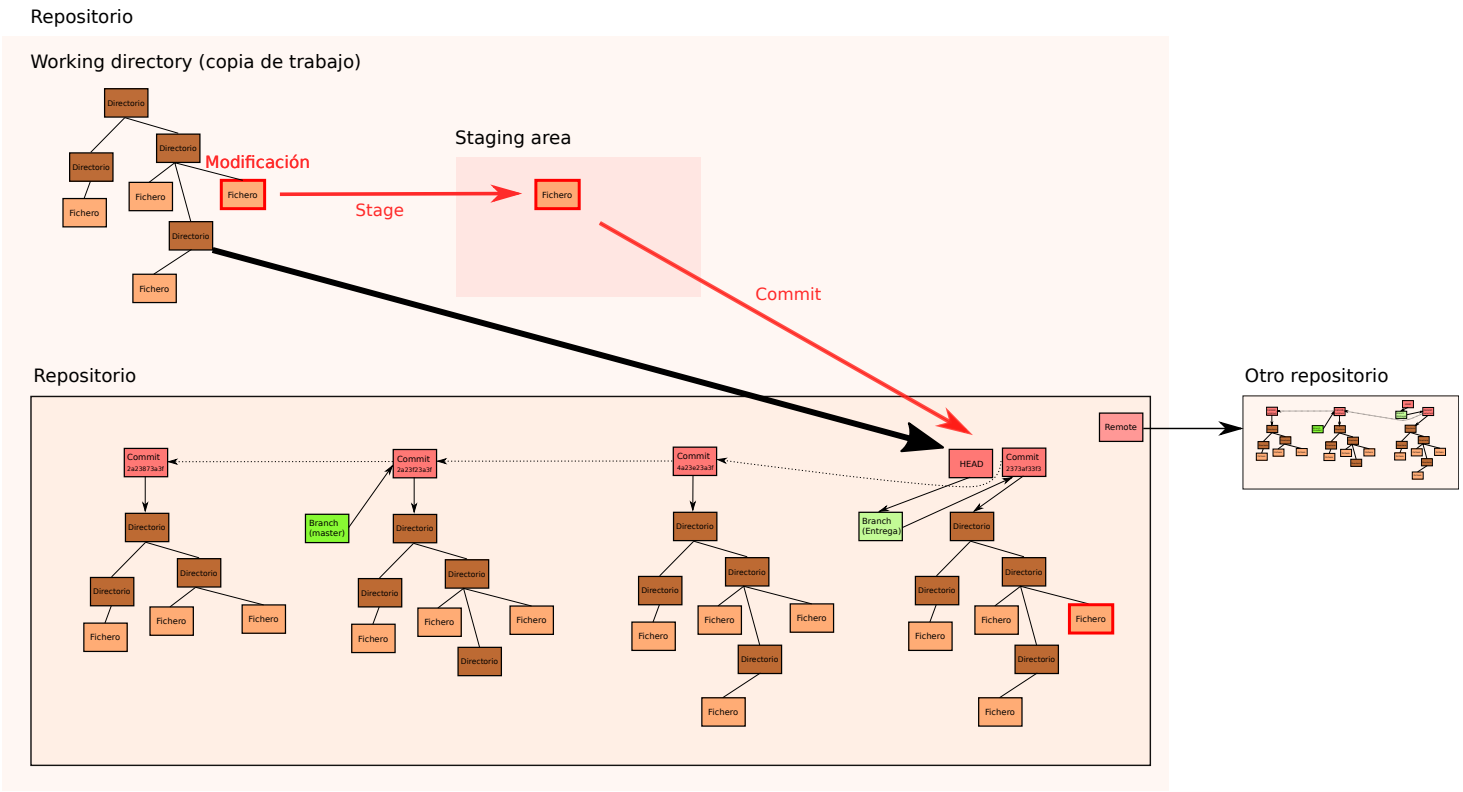
```
# una línea de comentario
# ignorar los ficheros que acaben en .a
*.a
#  no ignorar lib.a, a pesar de la regla anterior
!lib.a
#  ignorar sólo el fichero /TODO en el raíz
/TODO
#  ignorar todo en build/ en cualquier sitio
build/
#  ignorar doc/n.txt, pero no doc/server/a.txt
doc/*.txt
#  ignorar, encaja con a/x/b y con a/x/c/d/b
a/**/b
#  ignorar el contenido de bla y subdirectorios
bla/**
# ignorar, encaja con /a/d/z/b y con /e/z/b
**/z/b
```

Contenido

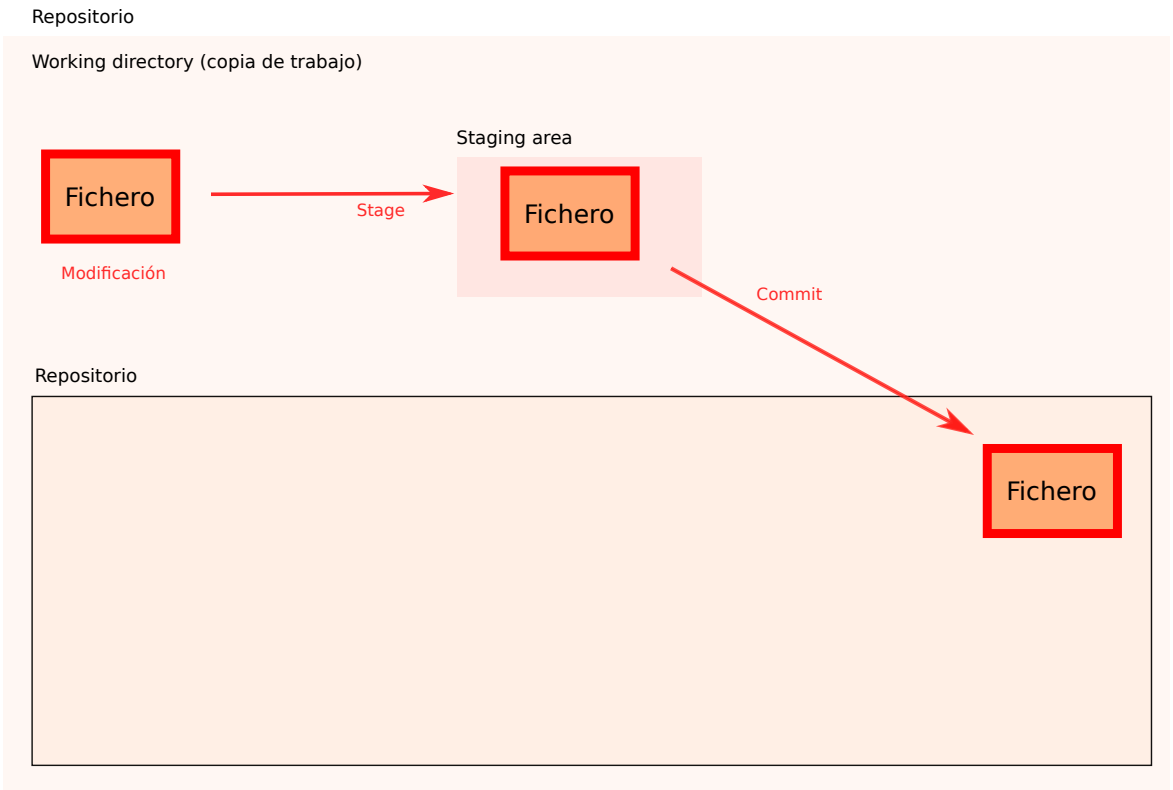
git diff

- Antes de aplicar cambios
- Tres tipos
  - Entre work y stage (`git diff`)
  - Entre stage y repo (`git diff --staged`)
  - Entre dos objetos (`git diff sha1 sha2`, ahora no lo vamos a ver)

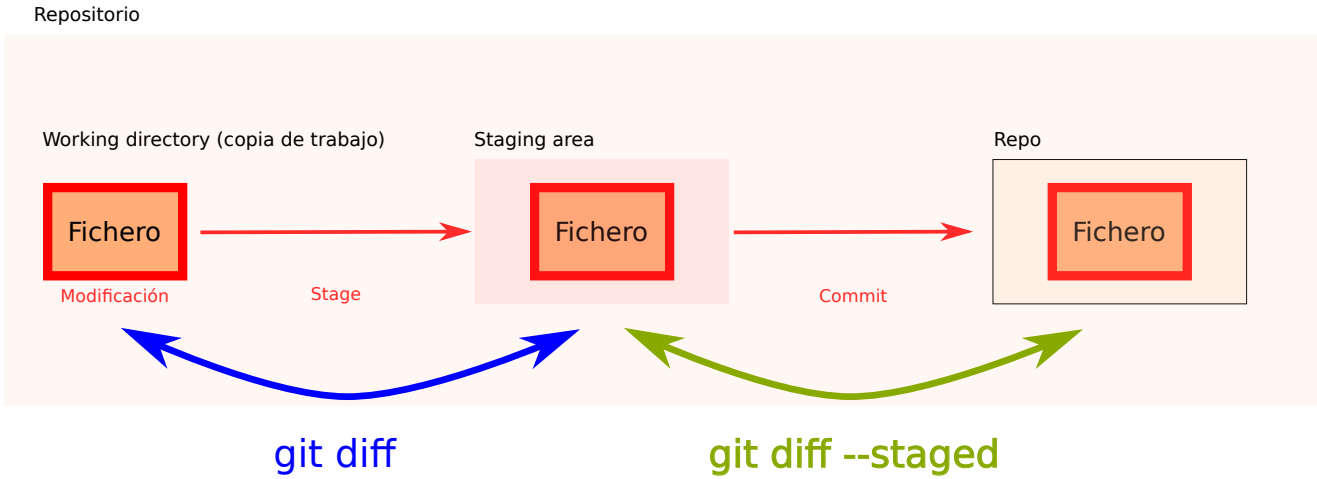
Viendo en un repositorio...



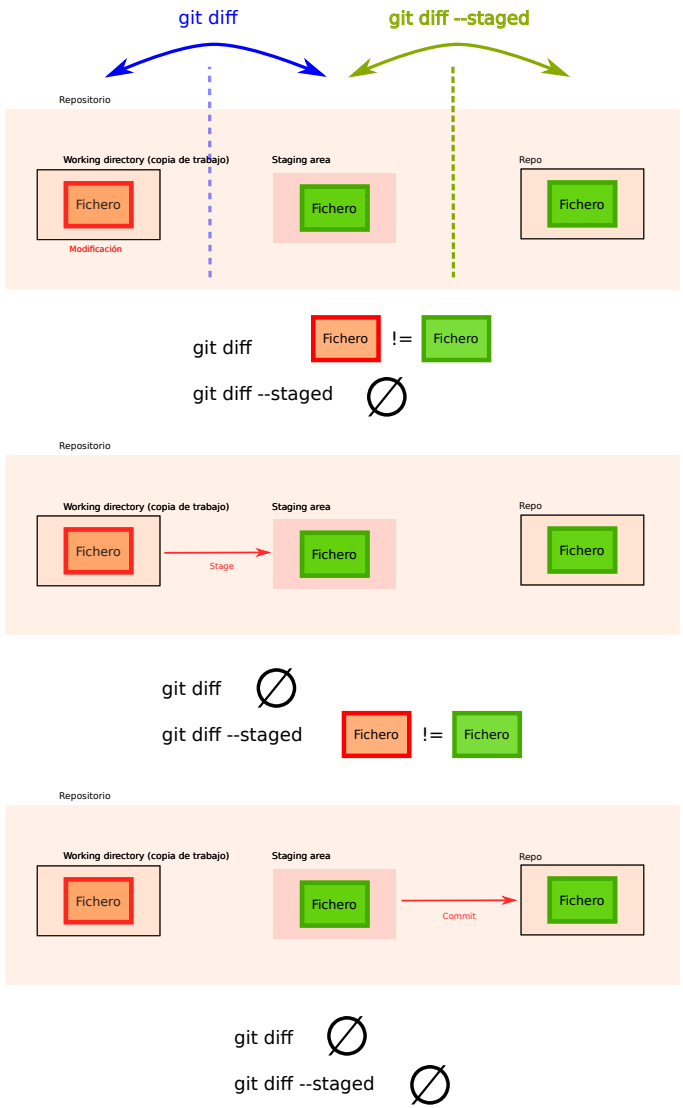
. . . lo que le pasa a un fichero



Podemos usar los dos diff



Resultado de los dos diff



Creo el fichero

```
 $ echo aaa > file
 $ git add file
 $ git commit -m 'initial file'
[master f6647e8] initial file
 2 files changed, 2 insertions(+), 1 deletion(-)
 create mode 100644 file
 $ git status
On branch master
nothing to commit, working tree clean
```

Modifico

```
 $ echo bbb > file
 $ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   file

no changes added to commit (use "git add" and/or "git commit -a")
```

Modifico

```
 $ git diff
diff --git a/file b/file
index 72943a1..f761ec1 100644
--- a/file
+++ b/file
@@ -1 +1 @@
-aaa
+bbb
 $ git diff --staged
 $
```

Stage

```
 $ git add file
 $ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   file

 $ git diff
 $ git diff --staged
diff --git a/file b/file
index 72943a1..f761ec1 100644
--- a/file
+++ b/file
@@ -1 +1 @@
-aaa
+bbb
```

Commit

```
 $ git commit -m 'changes'
[master f227358] changes
 1 file changed, 1 insertion(+), 1 deletion(-)
 $ git status
On branch master
nothing to commit, working tree clean
 $ git diff file
 $ git diff --staged file
 $
```

Contenido

Repositorio Bare

- No tiene directorio de trabajo (ni stage)
- Vale para los servidores, para hacer **push** y **pull** de cambios
- Por convenio el directorio acaba en `.git`

```
$ git init --bare origen.git
```

Clone

- Para clonar un repositorio remoto
- Deja remote apuntando a ese repositorio (para **push** y **pull**)
- Crea el directorio sin `.git` al final y con el repositorio dentro
- El tipo de nombre (url, directorio) tiene que ver con el protocolo, puede ser ssh, git, directorio local...
- En el ejemplo, https

```
$ git clone https://github.com/paurea/dumprsync.git
```

Repositorios remotos

- Imagina que tenemos un ordenador en casa
- Y la cuenta del laboratorio
- Queremos tener el repo en ambos sitios y trabajar en ambos sitios

Repositorios remotos

- Creamos un repo bare (central) para la práctica en el home
- Tenemos un repo de trabajo en el home del labo
- Tenemos un repo de trabajo en el home de casa
- Vamos a simularlo en local (luego veremos por ssh, muy fácil)

Pruebas

- Creamos un repositorio local repocentral

```
$ cd /trabajo
$ git init --bare repocentral.git
```

Pruebas

- Hacemos un clone trabajocasa
- Veo que está apuntando a trabajocasa

```
 $ cd /trabajo
 $ git clone repocentral.git trabajocasa
Cloning into 'trabajocasa'...
warning: You appear to have cloned an empty repository.
done.
 $ cd /trabajo/trabajocasa
 $ git remote -v
origin          /trabajo/repocentral.git (fetch)
origin          /trabajo/repocentral.git (push)
```

Pruebas

- Hacemos un clone trabajolabo

```
$ git clone repocentral.git trabajolabo
Cloning into 'trabajocasa'...
warning: You appear to have cloned an empty repository.
done.
```

Pruebas

- Creo un commit de un fichero en trabajocasa

```
 $ vi tub.c
 $ git add tub.c
 $ git commit -m 'primer fichero'
[master (root-commit) 8e1cef2] fich
 1 file changed, 5 insertions(+)
 create mode 100644 tub.c
 $ git push
Counting objects: 3, done.
Writing objects: 100% (3/3), 205 bytes | 205.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To /trabajo/repocentral.git
 * [new branch]      master -> master
```

Pruebas

- Traigo el cambio a trabajolabo

```
 $  cd /trabajo/trabajolabo
 $ git pull
remote: Counting objects: 3, done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From /trabajo/repocentral
 * [new branch]      master     -> origin/master
 $ git log
commit 8e1cef28874e724e3836408226434483d203bff8e (HEAD -> master, origin/master)
Author: paurea <paurea@gmail.com>
Date:   Fri Nov 23 16:30:33 2018 +0100

    primer fichero
```

Pruebas

- Así puedo simular que tengo un servidor
- Y dos clientes
- Para probar conflictos y demás
- Usaremos **repocentral.git trabajocasa** y **trabajolabo**
- El ejemplo de antes

```
repocentral.git                          trabajocasa                                  trabajolabo
$ git init --bare repocentral.git
                                 $ git clone /trabajo/repocentral.git trabajocasa
                                 $ cd trabajocasa
                                                          $ git clone /trabajo/repocentral trabajolabo
                                                          $ cd trabajolabo
                                 $ vi tub.c
                                 $ git add tub.c
                                 $ git commit -m 'primer fichero'
                                 $ git push
                                                          $ git pull
```

Ahora de verdad en remoto

- Desde mi máquina de casa
- Creo dos repos en el labo, uno central y uno de trabajo

```
$ ssh paurea@alpha.aulas.gsyc.urjc.es
paurea@alpha.aulas.gsyc.urjc.es's password:
$ paurea@alpha$ git init --bare repocentral.git
Initialized empty Git repository in /home/gsyc/paurea/repocentral.git/
$ paurea@alpha$ git clone repocentral trabajolabo
Cloning into 'trabajolabo'...
warning: You appear to have cloned an empty repository.
done.
$ paurea@alpha$ logout
Connection to alpha.aulas.gsyc.urjc.es closed.
$ $ git clone paurea@alpha.aulas.gsyc.urjc.es:repocentral.git trabajocasa
Cloning into 'trabajocasa'...
paurea@alpha.aulas.gsyc.urjc.es's password:
warning: You appear to have cloned an empty repository.
```

Ahora de verdad en remoto

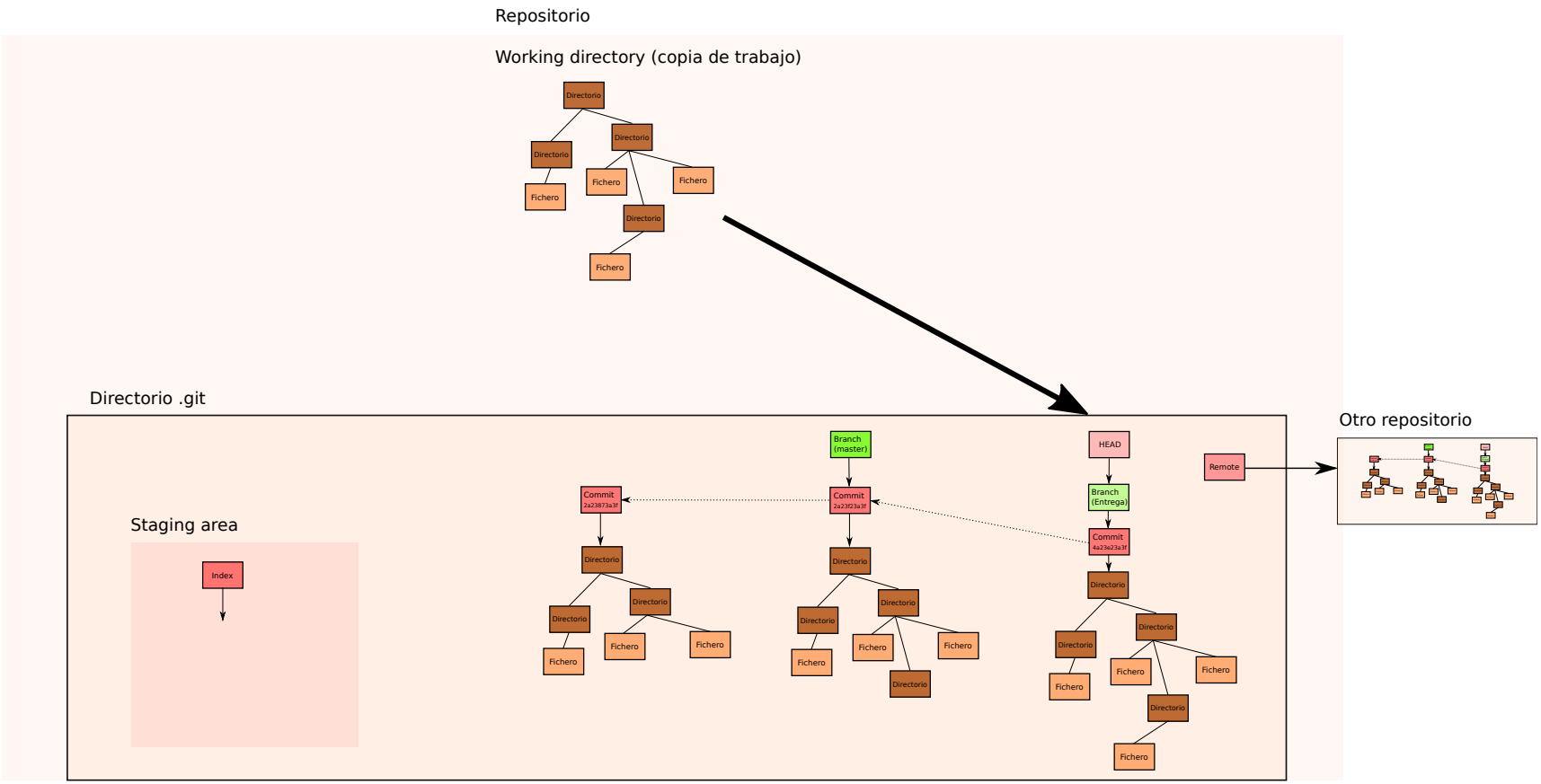
- Ya puedo trabajar en remoto

```
repocentral.git                          trabajocasa                          trabajolabo
$ ssh paurea@alpha.aulas.gsyc.urjc.es
paurea@alpha.aulas.gsyc.urjc.es's password:
paurea@alpha$ git init --bare repocentral.git
                              $ git clone paurea@alpha.aulas.gsyc.urjc.es:repocentral.git trabajocasa
                              paurea@alpha.aulas.gsyc.urjc.es's password:
                              $ cd trabajocasa
                                                        $ ssh paurea@alpha.aulas.gsyc.urjc.es
                                                        paurea@alpha.aulas.gsyc.urjc.es's password:
                                                        paurea@alpha$ git clone repocentral.git trabajolab
                                                        paurea@alpha$ cd trabajolabo
                              $ vi tub.c
                              $ git add tub.c
                              $ git commit -m 'primer fichero'
                              $ git push
                              paurea@alpha.aulas.gsyc.urjc.es's password:
                                                        paurea@alpha$ git pull
```

Contenido

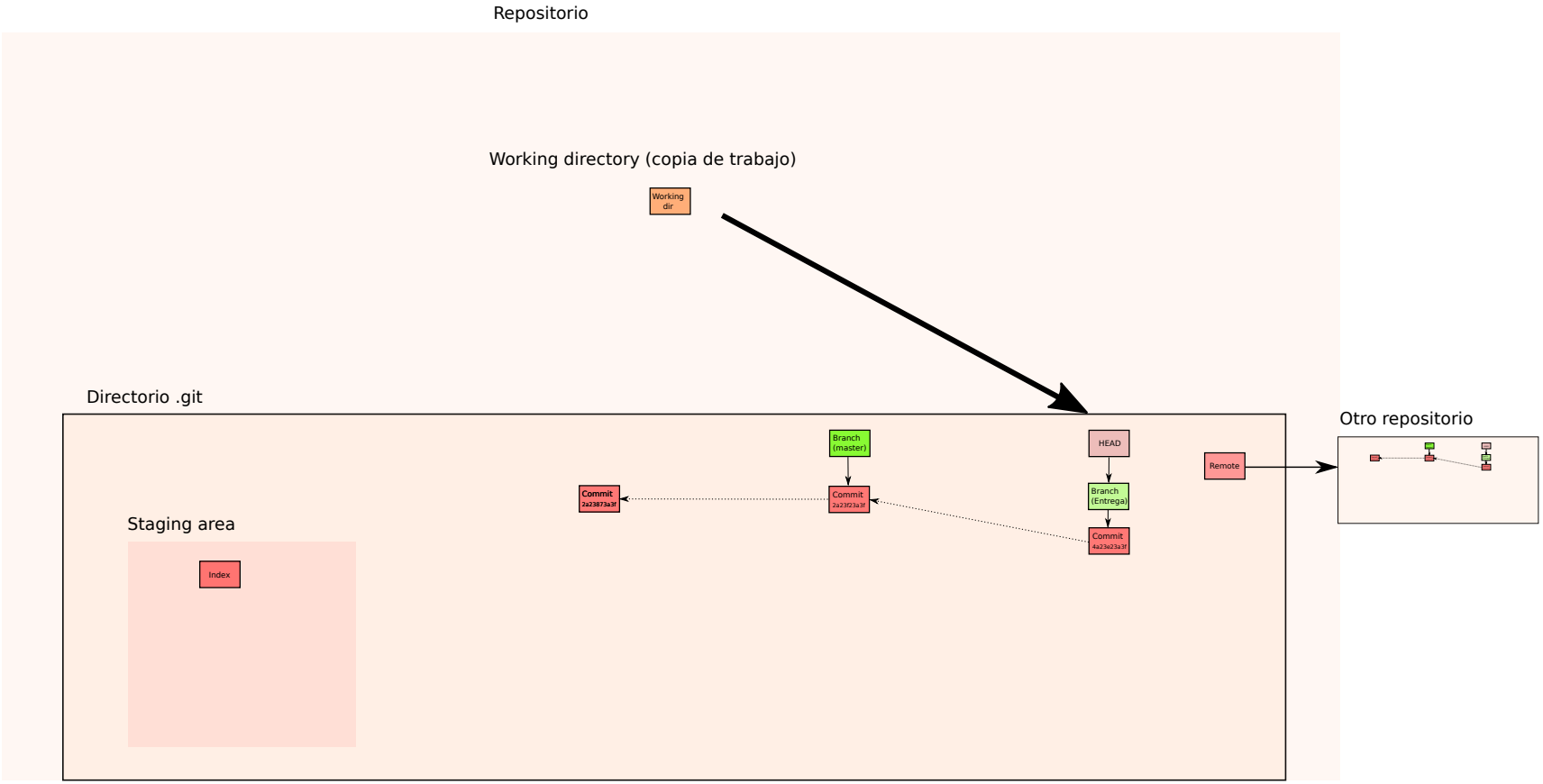
Grafo de commits

- Nos interesan los grafos de commits
- Cada commit, para el usuario, un snapshot del sistema de ficheros
- Pero hay que recordar, sólo guardamos los cambios
- Cambios a nivel de Merkle, (ficheros/líneas, dependiendo del formato)



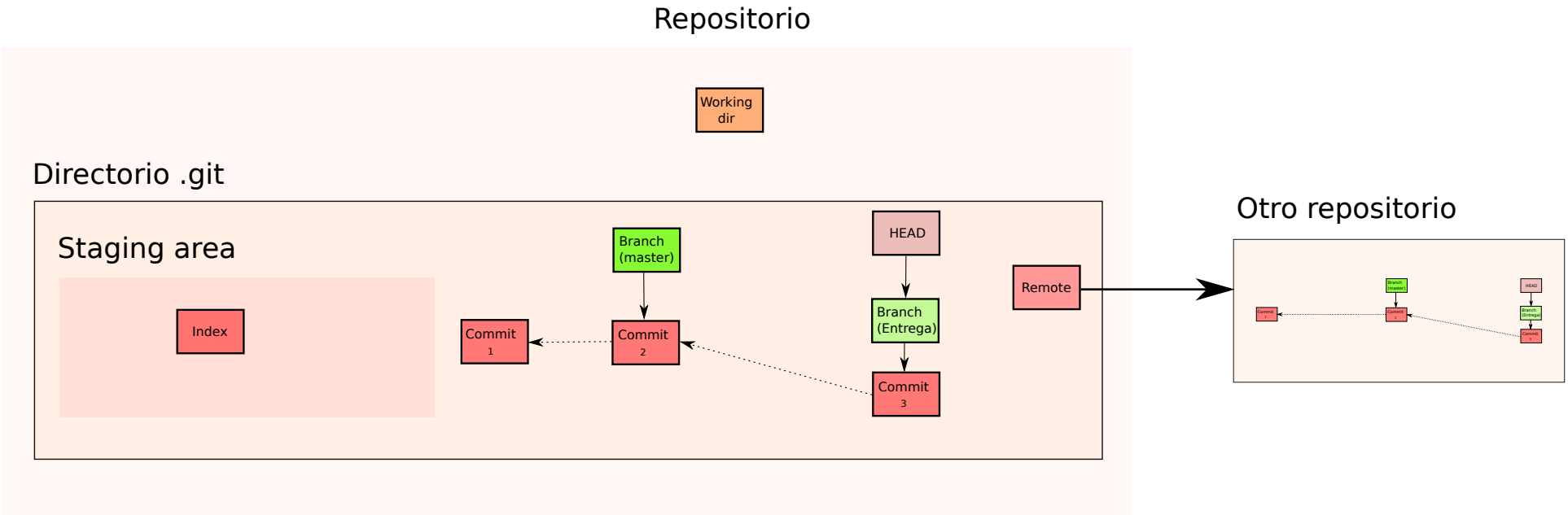
Grafo de commits

- Nos interesan los grafos de commits
- Cada commit es un conjunto de cambios
- Sobre otro commit
- Luego tenemos punteros (head, branch, tag. . .)



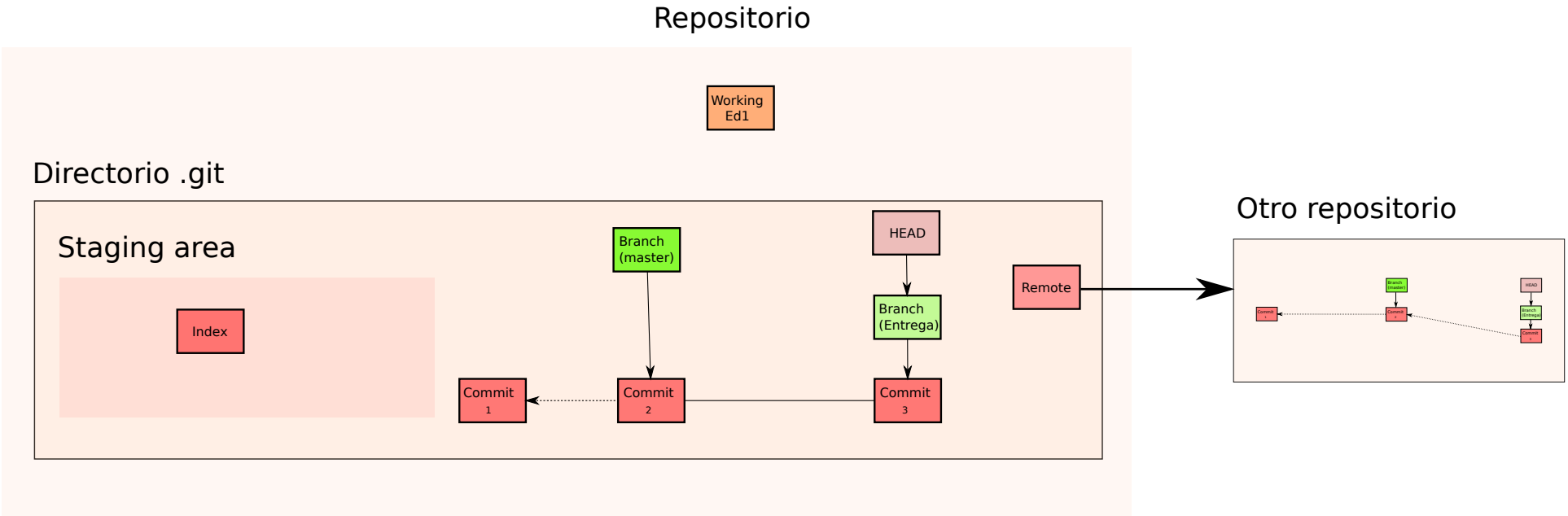
Grafo de commits

- Vamos a usar este diagrama (simplificado)
- Pero los ficheros están igual colgando debajo de los commits
- Vamos a inventarnos números y letras para identificar los commits
- En realidad, SHA (chorro hexa)



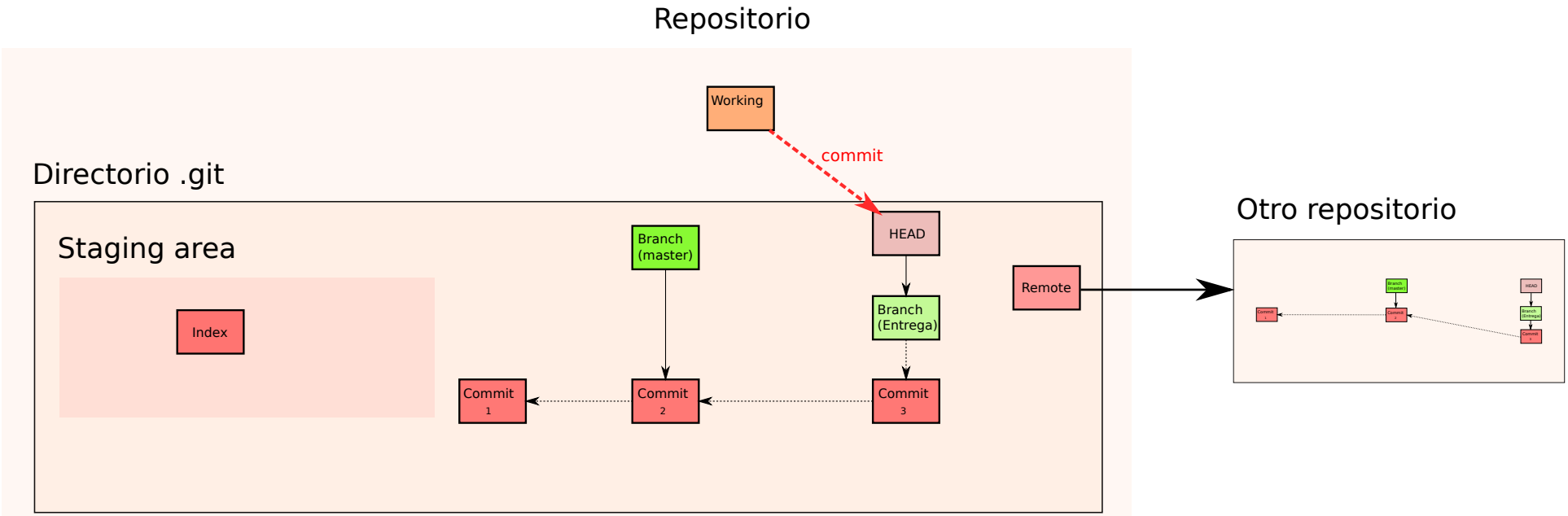
Grafo de commits

- La flecha negra (dónde trabajamos)
- En realidad es HEAD



Grafo de commits

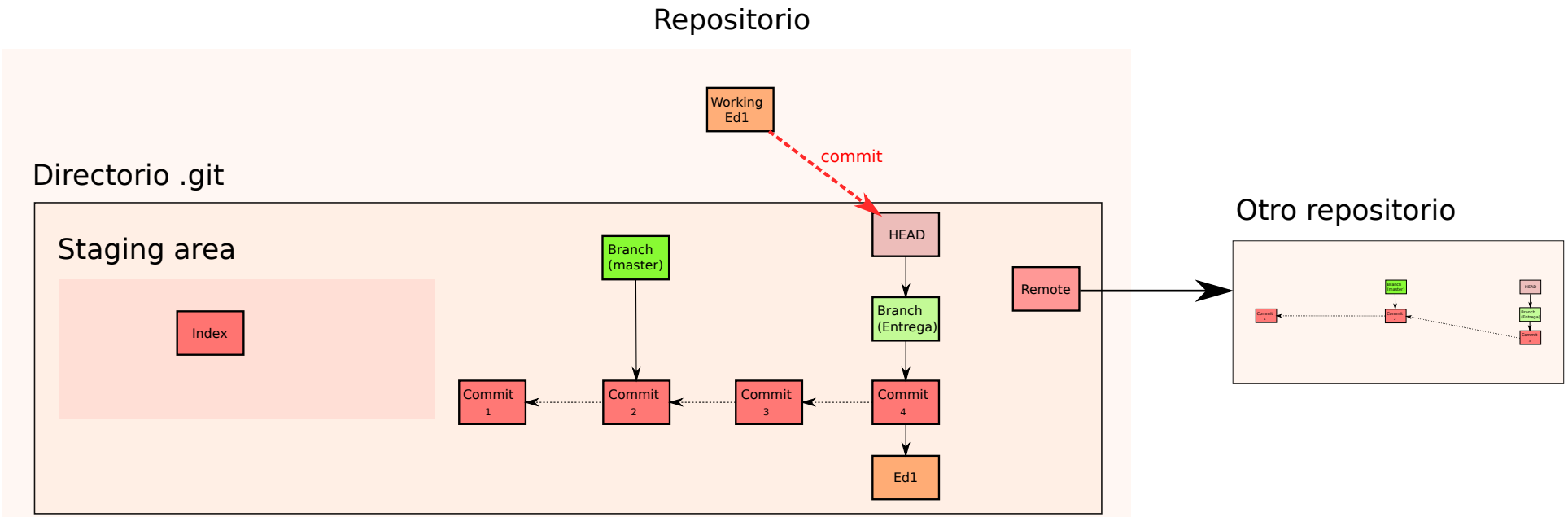
- Estando donde estamos ahora
- Hacemos un commit
- Va a HEAD, que avanza la branch actual



Grafo de commits

- Estando donde estamos ahora
- Hacemos un commit
- Va a HEAD, que avanza la branch actual

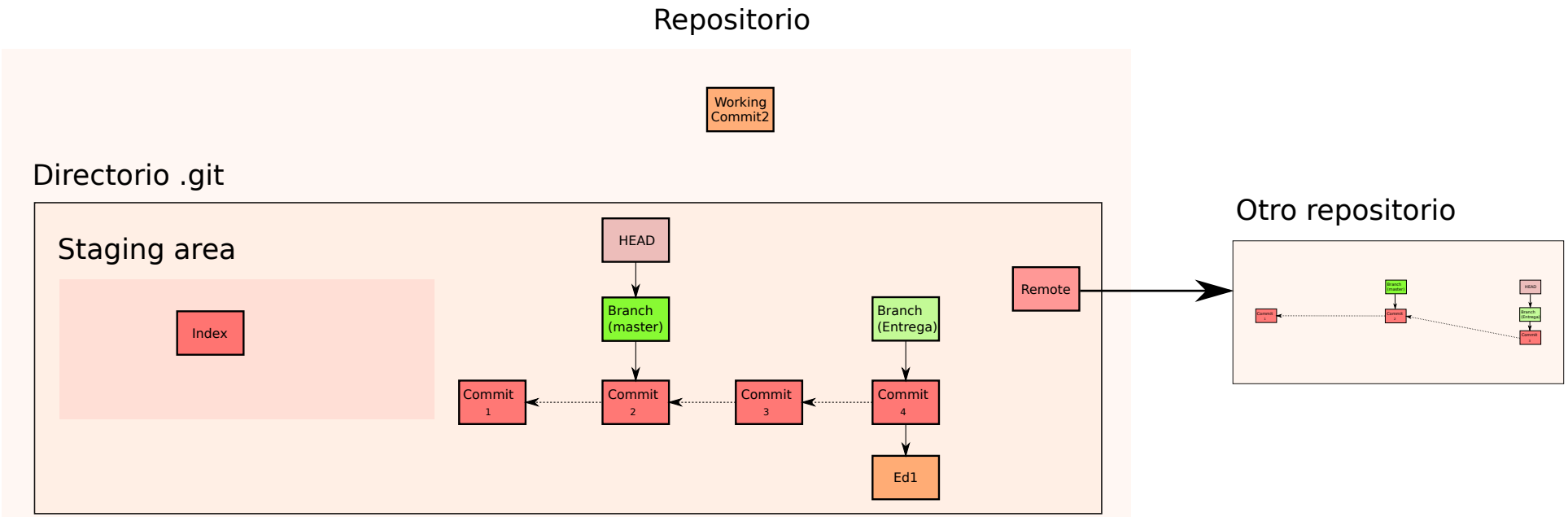
```
$  git commit -a -m 'hago ed1'
```



Grafo de commits

- Tengo ahora otra edición Ed2
- Primero tengo que cambiar el head
- Esto me sobreescribe el directorio de trabajo

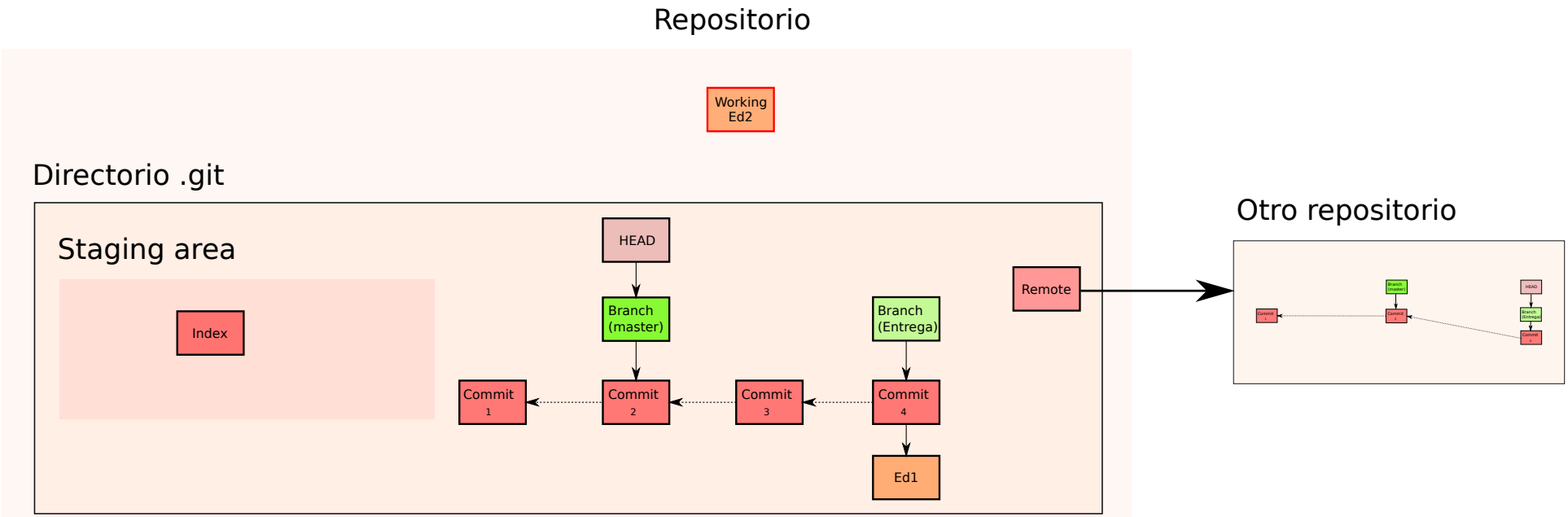
```
$  git checkout master
```



Grafo de commits

- Tengo ahora otra edición
- Que quiero que vaya a la nueva
- Ed2

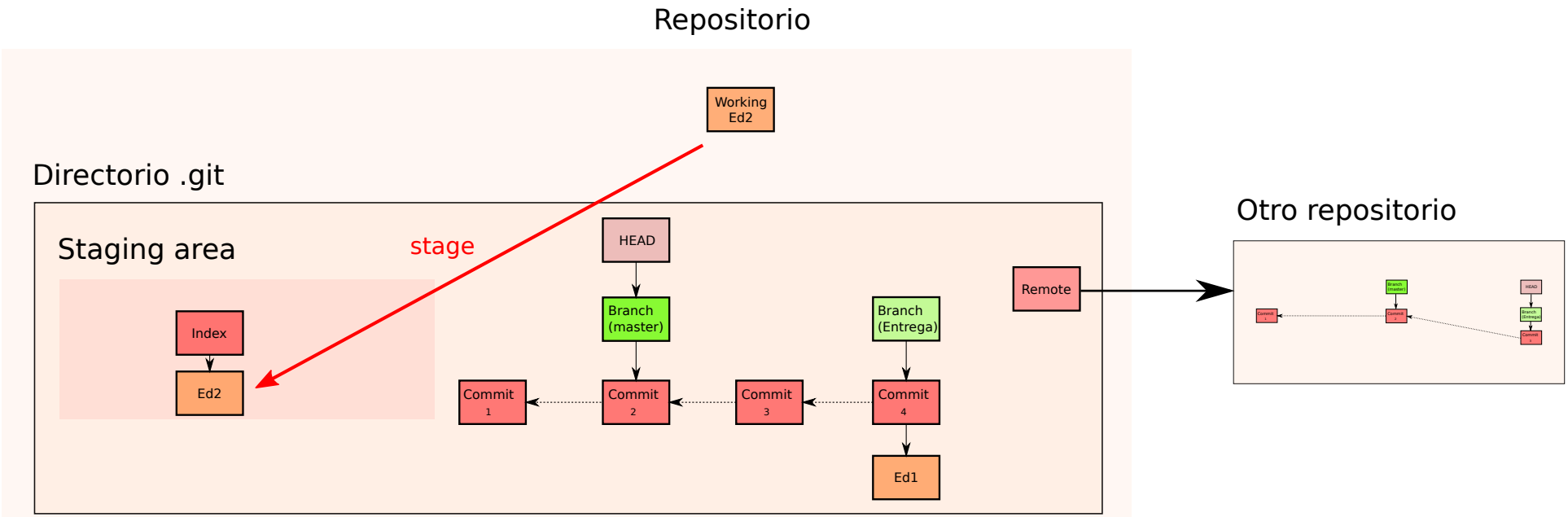
```
$  vi z.c
```



Grafo de commits

- Hago stage de los cambios
- Mete el fichero modificado en index

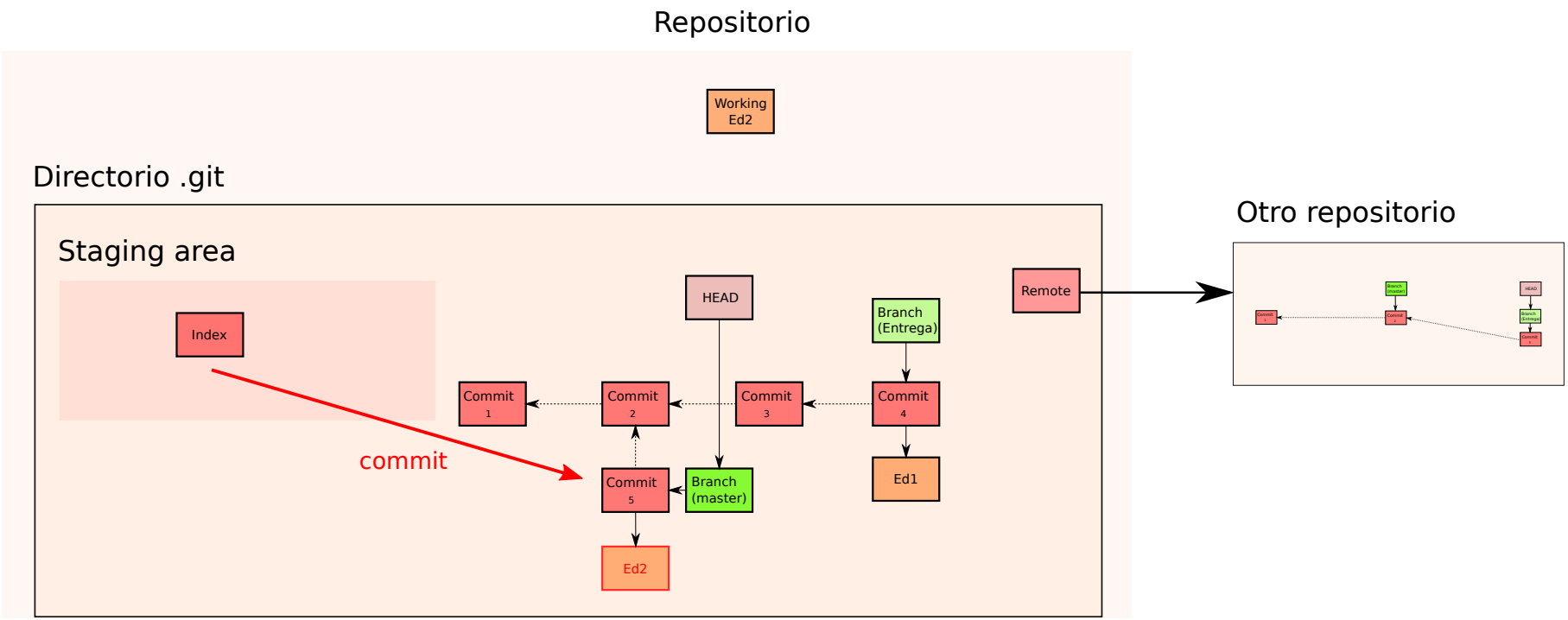
```
$  git add *.c
```



Grafo de commits

- Hago commit de los cambios
- Crea un nuevo commit apuntado al que era el HEAD
- Mueve Branch y HEAD

```
$  git commit -m 'hago ed2'
```



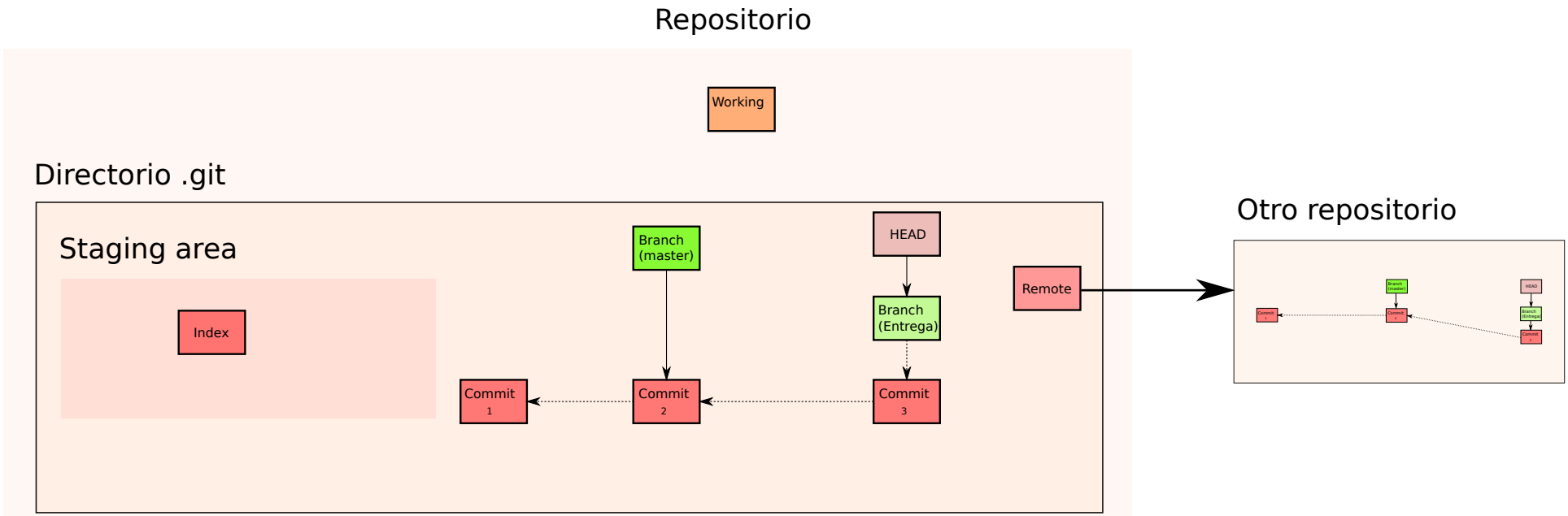
Grafo de commits

- Mensajes, describen el commit (el parámetro `git -m 'mensaje del commit'`)
- Escribir con cuidado

Creación: git branch

- Para crear una nueva branch
- La que hay por defecto es master
- No hace checkout (crear + checkout es `git checkout -b`)

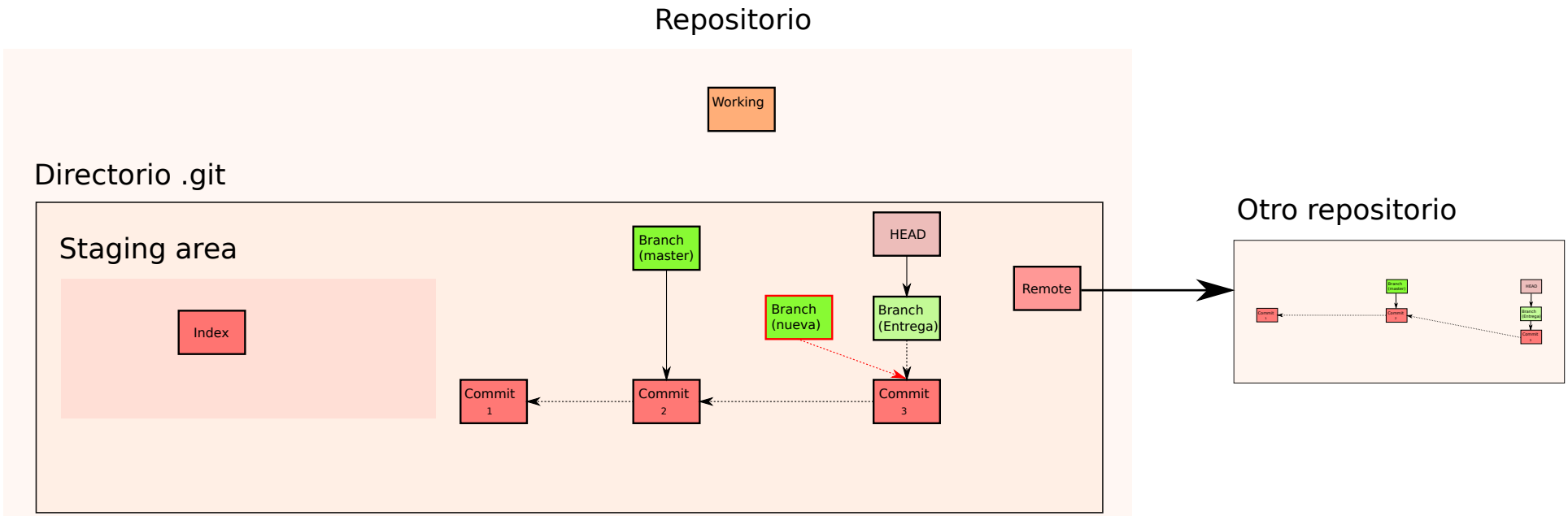
```
$  git branch nueva
```



Creación: git branch

- Para crear una nueva branch
- La que hay por defecto es master
- No hace checkout (crear + checkout es `git checkout -b`)

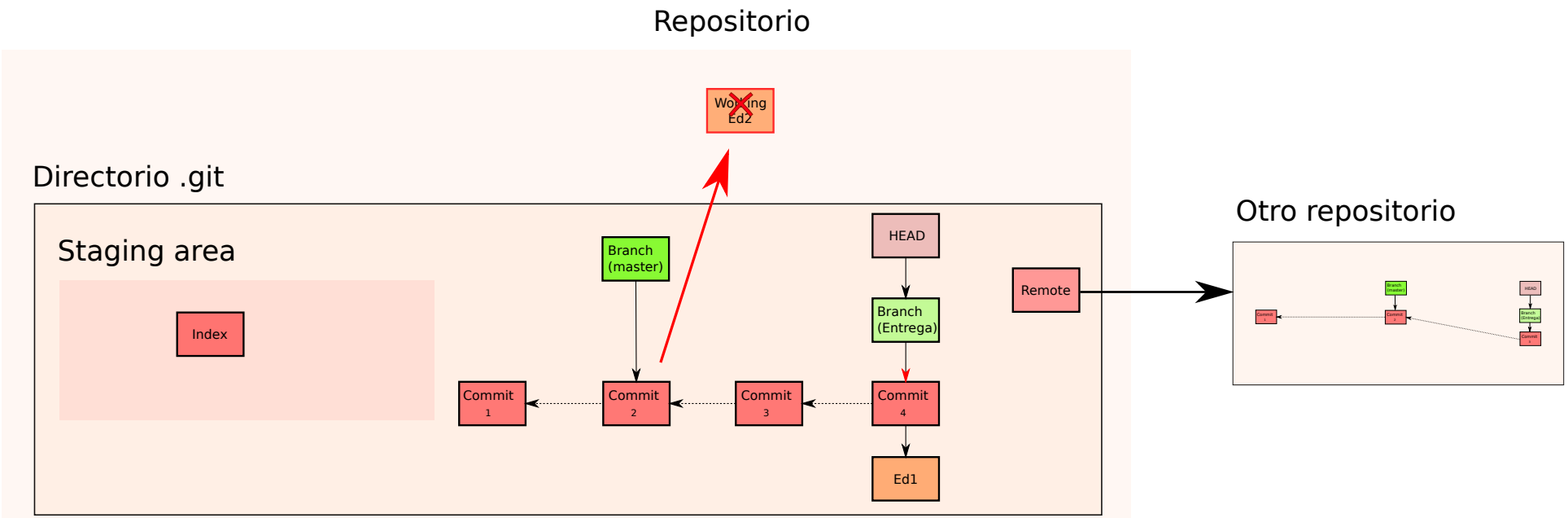
```
$  git branch nueva
```



Ojo

- Cuando hago checkout
- De no tener modificaciones sin commit
- O la lío y pierdo los cambios (git no se deja)

```
$  vi x.c
$  git checkout master  $ #acabo de perder lo que hice en x.c
```



Checkout problemático

- Ejemplo de error con dos branches
- master y nuevo

```
 $ git init repocentral
Initialized empty Git repository in /home/paurea/GITEX/badbranch/repocentral/.git/
 $ ls
guide  repocentral
 $ cd repocentral
 $ vi x.c
 $ git add x.c
 $ git commit -m 'creo x.c'
[master (root-commit) 7307527] creo x.c
 1 file changed, 3 insertions(+)
 create mode 100644 x.c
 $ git branch nuevo
 $ git checkout nuevo
Switched to branch 'nuevo'
 $ vi x.c
 $ git add x.c
 $ git commit -m 'modifico nuevo'
[nuevo 25e3fa2] modifico nuevo
 1 file changed, 2 insertions(+)
```

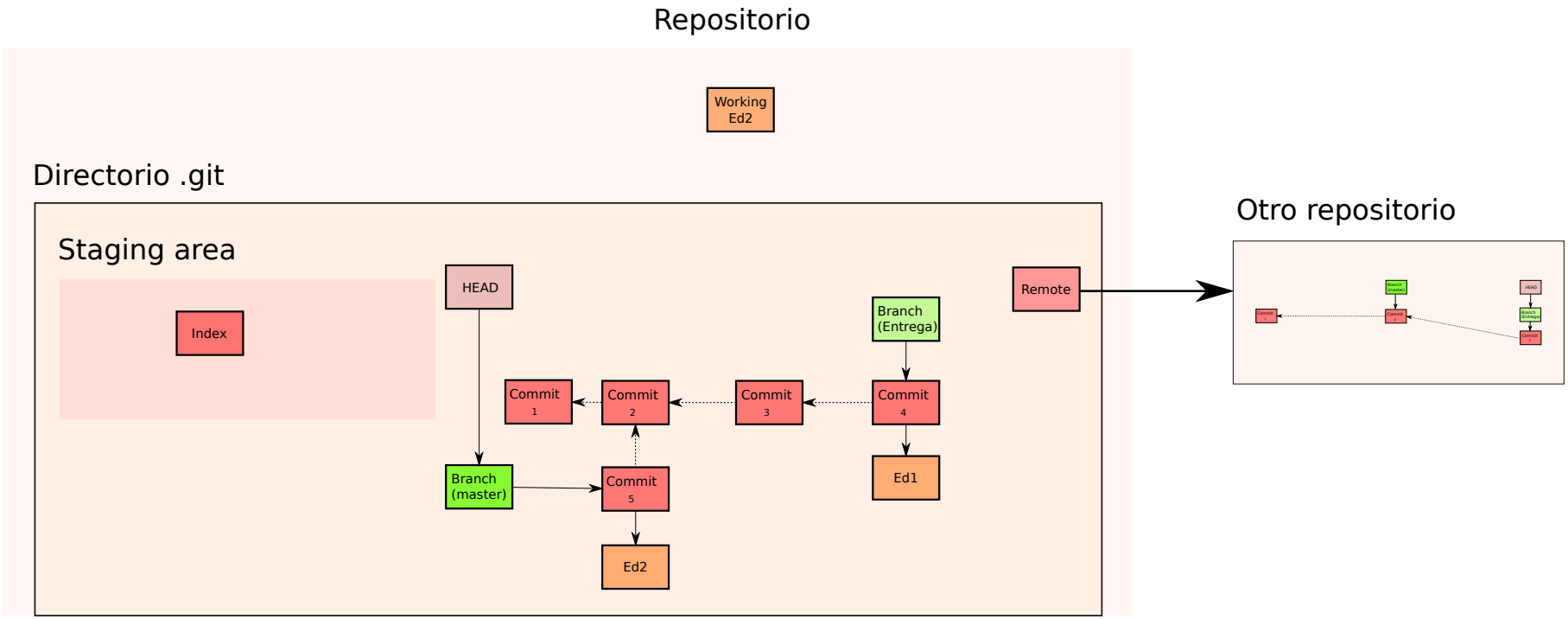
Checkout problemático

- Git no se deja

```
 $ vi x.c
 $ git checkout master
error: Your local changes to the following files would be overwritten by checkout:
        x.c
Please commit your changes or stash them before you switch branches.
Aborting
```

Merge de branches

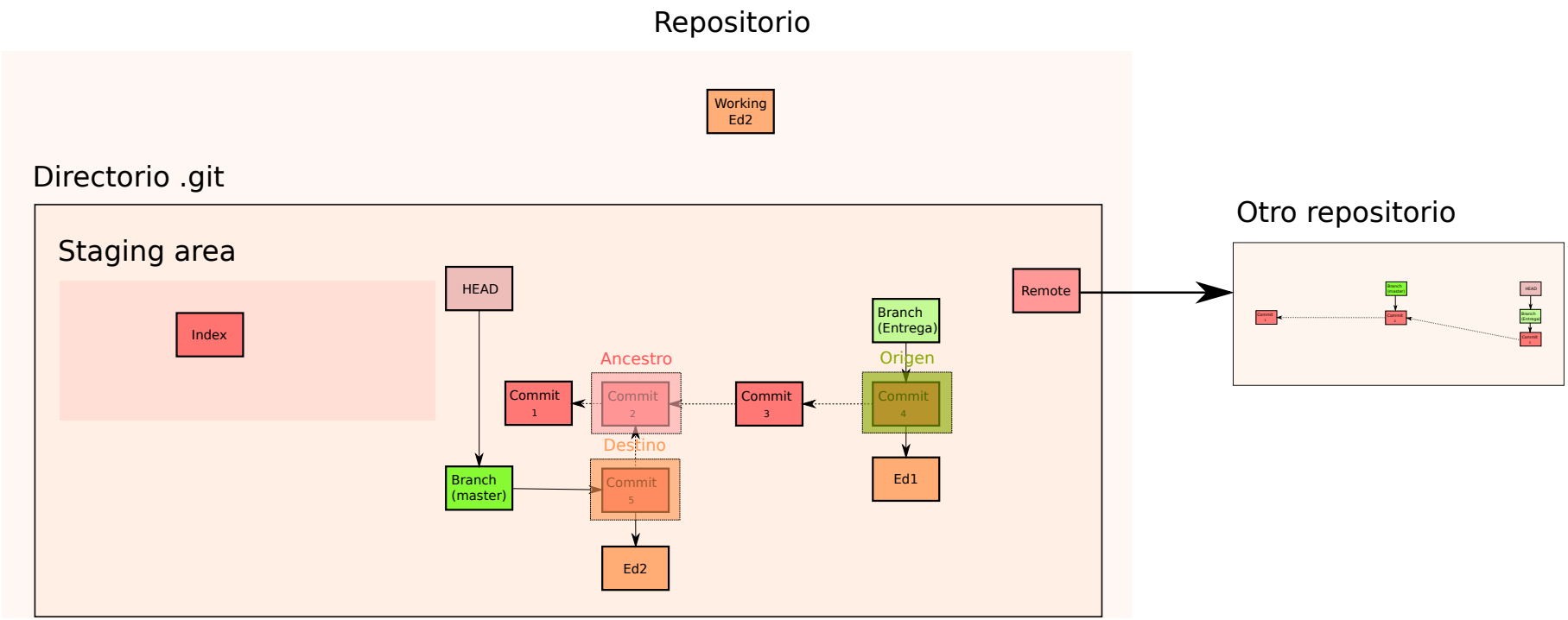
- Supongamos que estamos en master
- Ojo, es importante dónde tengo el HEAD
- Y tenemos Entrega
- Y queremos mezclar los cambios



Merge de branches

- Queremos un nuevo commit
- Que contenga los cambios de Entrega aplicados a master
- Git busca un ancestro común

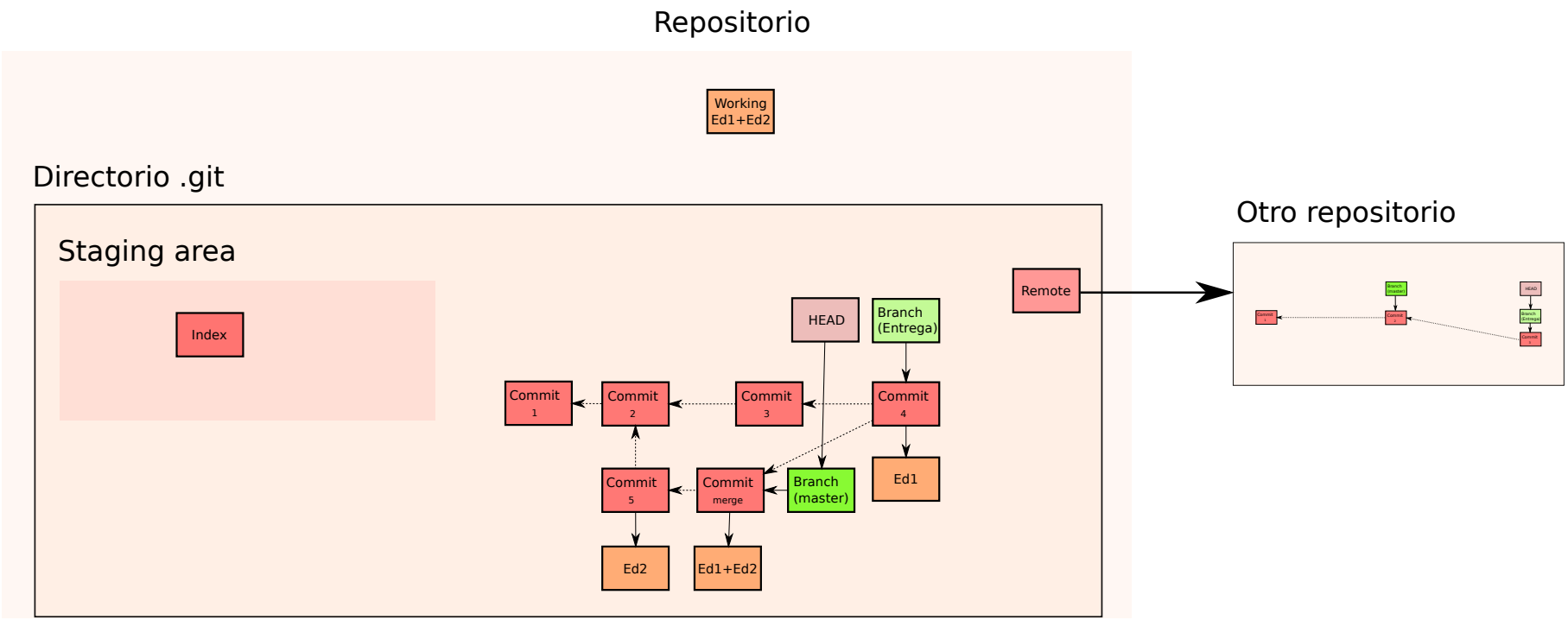
```
$ git merge Entrega
```



Merge de branches

- Si los ficheros no existen los crea
- Si las líneas no existen las añade
- Si todo va bien. . . (Si la misma línea editada: habrá conflicto)

```
$ git merge Entrega
```



Contenido

Merge de branches

- Para entender como funciona, a qué nos referimos con $Ed1 + Ed2$
- Hay que entender bien `diff` y `patch` (comandos clásicos de Unix)
- `diff` encuentra el mínimo número de diferencias entre dos ficheros/directorios
- `patch` permite aplicar esas diferencias
- Git tiene sus versiones `git diff` y `git patch`

diff y patch

- Imaginemos que tenemos un repositorio
- Con dos ficheros `texto.txt` y `texto.txt`

```
 $ cat otro.txt
De acuerdo a Greimas, es un enunciado ya sea gráfico o fónico que nos
permite visualizar las palabras que escuchamos que es utilizado para
manifestar el proceso lingüístico. Mientras Hjelmslev usa ese término
para designar el todo de una cadena lingüística ilimitada (§1).
 $  cat texto.txt
También es una composición de caracteres imprimibles (con grafema)
generados por un algoritmo de cifrado que, aunque no tienen sentido
para cualquier persona, sí puede ser descifrado por su destinatario
original. En otras palabras, un texto es un entramado de signos con una
intención comunicativa que adquiere sentido en determinado contexto.
```

diff y patch

- Los editamos
- Las líneas en en verde son nuevas
- Las líneas en en rojo desaparecen

```
----------------------otro.txt--------------------
De acuerdo a Greimas, es un enunciado ya sea gráfico o fónico que nos
permite visualizar las palabras que escuchamos que es utilizado para
 decir aquí que proceso
manifestar el proceso lingüístico. Mientras Hjelmslev usa ese término
 meter aquí detalles
para designar el todo de una cadena lingüística ilimitada (§1).


 ----------------------texto.txt--------------------
También es una composición de caracteres imprimibles (con grafema)
generados por un algoritmo de cifrado que, aunque no tienen sentido
 para cualquier persona, sí puede ser descifrado por su destinatario
original. En otras palabras, un texto es un entramado de signos con una
 definida e intensa
intención comunicativa que adquiere sentido en determinado contexto.
```

diff y patch

- Este es el nuevo fichero

```
$ cat otro.txt
De acuerdo a Greimas, es un enunciado ya sea gráfico o fónico que nos
permite visualizar las palabras que escuchamos que es utilizado para
decir aquí que proceso
manifestar el proceso lingüístico. Mientras Hjelmslev usa ese término
meter aquí detalles
para designar el todo de una cadena lingüística ilimitada (§1).

$  $ cat texto.txt
También es una composición de caracteres imprimibles (con grafema)
generados por un algoritmo de cifrado que, aunque no tienen sentido
original. En otras palabras, un texto es un entramado de signos con una
definida e intensa
intención comunicativa que adquiere sentido en determinado contexto.
```

diff y patch

- Podemos correr diff
- Recordemos que compara directorio de trabajo y stage
- El fichero resultante es un patch, un fichero con las diferencias

```
 $ git diff
diff --git a/otro.txt b/otro.txt
index dc0660f..e4e2aa5 100644
--- a/otro.txt
+++ b/otro.txt
@@ -1,4 +1,6 @@
 De acuerdo a Greimas, es un enunciado ya sea gráfico o fónico que nos
 permite visualizar las palabras que escuchamos que es utilizado para
 +decir aquí que proceso
 manifestar el proceso lingüístico. Mientras Hjelmslev usa ese término
 +meter aquí detalles
 para designar el todo de una cadena lingüística ilimitada (§1).
diff --git a/texto.txt b/texto.txt
index ebe0425..c45344e 100644
--- a/texto.txt
+++ b/texto.txt
@@ -1,7 +1,7 @@
 También es una composición de caracteres imprimibles (con grafema)
 generados por un algoritmo de cifrado que, aunque no tienen sentido
 -para cualquier persona, sí puede ser descifrado por su destinatario
 original. En otras palabras, un texto es un entramado de signos con una
 +definida e intensa
 intención comunicativa que adquiere sentido en determinado contexto.
```

diff y patch

- El flag del comando de arriba `diff --git` *no existe*
- Es como git indica que estás corriendo `git diff`
- *dc*0660*f* y *e*4*e*2*aa*5 son hashes SHA de las versiones (*a* y *b*) de ficheros a comparar
- 100644 Son los permisos de los ficheros (tipo de fichero, rwx)

```
$ git diff
diff --git a/otro.txt b/otro.txt
index dc0660f..e4e2aa5 100644
```

diff y patch

- Cabecera del formato unificado del formato diff seguido de un trozo de texto (hunk)
- El signo menos $-$ significa que son cosas de *a* no presentes en *b*
- El signo menos $+$ significa que son cosas que faltan en *a* presentes en *b*
- `@@ -14 +1,6 @@`, es el tamaño y longitud del texto en *a* y *b*

```
--- a/otro.txt
+++ b/otro.txt
@@ -1,4 +1,6 @@
```

diff y patch

- Las líneas que son diferentes van precedidas del carácter definido en la cabecera:
  - El signo menos $-$ significa que son cosas de *a* no presentes en *b*
  - El signo menos $+$ significa que son cosas que faltan en *a* presentes en *b*

```
--- a/otro.txt
+++ b/otro.txt
@@ -1,4 +1,6 @@
 También es una composición de caracteres imprimibles (con grafema)
 generados por un algoritmo de cifrado que, aunque no tienen sentido
-para cualquier persona, sí puede ser descifrado por su destinatario
 original. En otras palabras, un texto es un entramado de signos con una
+definida e intensa
 intención comunicativa que adquiere sentido en determinado contexto.
```

diff y patch

- El fichero entero es un patch
- Se podría aplicar sobre la versión antigua (en el ejemplo la de stage) para generar la nueva
- Esto se haría con el comando `git apply` que es equivalente al comando de unix `patch apply` (pero para el formato de git)

```
 $ git diff
diff --git a/otro.txt b/otro.txt
index dc0660f..e4e2aa5 100644
--- a/otro.txt
+++ b/otro.txt
@@ -1,4 +1,6 @@
 De acuerdo a Greimas, es un enunciado ya sea gráfico o fónico que nos
 permite visualizar las palabras que escuchamos que es utilizado para
 +decir aquí que proceso
 manifestar el proceso lingüístico. Mientras Hjelmslev usa ese término
 +meter aquí detalles
 para designar el todo de una cadena lingüística ilimitada (§1).
diff --git a/texto.txt b/texto.txt
index ebe0425..c45344e 100644
--- a/texto.txt
+++ b/texto.txt
@@ -1,7 +1,7 @@
 También es una composición de caracteres imprimibles (con grafema)
 generados por un algoritmo de cifrado que, aunque no tienen sentido
 -para cualquier persona, sí puede ser descifrado por su destinatario
 original. En otras palabras, un texto es un entramado de signos con una
 +definida e intensa
 intención comunicativa que adquiere sentido en determinado contexto.
```

Contenido

Merge

- Volviendo a `git merge`
- Lo que hace es sacar el parche entre el antepasado común y el origen
- Y aplicarlo en el destino

Conflictos

- Si no se pueden calcular las diferencias con `diff`
- Porque las dos versiones editan *la misma línea*
- Surge un conflicto
- Hay que resolverlo
- Vamos a ver un ejemplo práctico con dos ramas

Conflictos con dos ramas

- Tenemos dos ramas, master y otra

```
 $ git init repo
Initialized empty Git repository in /home/paurea/GITEX/simplemerge/repo/.git/
 $ cd repo/
 $ vi a.txt
 $ cat a.txt
hola
adios
 $ git add a.txt
 $ git commit -m 'inicial'
[master (root-commit) e1c5a1a] inicial
 1 file changed, 3 insertions(+)
 create mode 100644 a.txt
 $ git branch otra
```

Conflictos con dos ramas

- Editamos la misma línea del mismo fichero en ambas

```
 $ vi a.txt
 $ cat a.txt
hola pepe
adios
 $ git add a.txt
 $ git commit -m 'pepe'
[master 366e24b] pepe
 1 file changed, 1 insertion(+), 1 deletion(-)
 $ git checkout otra
Switched to branch 'otra'
 $ cat a.txt
hola
adios
 $ vi a.txt
 $ git add a.txt
 $ cat a.txt
hola juan
adios
 $ git commit -m juan
[otra 132e5ec] juan
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Conflictos con dos ramas

- Tenemos un conflicto
- Las dos versiones de las líneas en conflicto están entre <<<<<<< *HEAD* y >>>>>>> *master*
- La de arriba es la versión de la línea de la rama HEAD (donde apunta HEAD, en este caso la rama otra)
- Lo de abajo es la versión de la rama master

```
 $ git diff master otra
diff --git a/a.txt b/a.txt
index 9ea163b..03e2518 100644
--- a/a.txt
+++ b/a.txt
@@ -1,3 +1,3 @@
-hola pepe
+hola juan
 adios

 $ git merge master
Auto-merging a.txt
CONFLICT (content): Merge conflict in a.txt
Automatic merge failed; fix conflicts and then commit the result.
 $ cat a.txt
<<<<<<< HEAD
hola juan
=======
hola pepe
>>>>>>> master
adios
```

Conflictos con dos ramas

- Edito el fichero
- Puedo usar mergetool o un editor

```
 $ git mergetool #o editar con vi
 $ git add a.txt
 $ git commit -m 'al final juan'
[otra 88f1598] al final juan
 $ cat a.txt
hola juan
adios
```

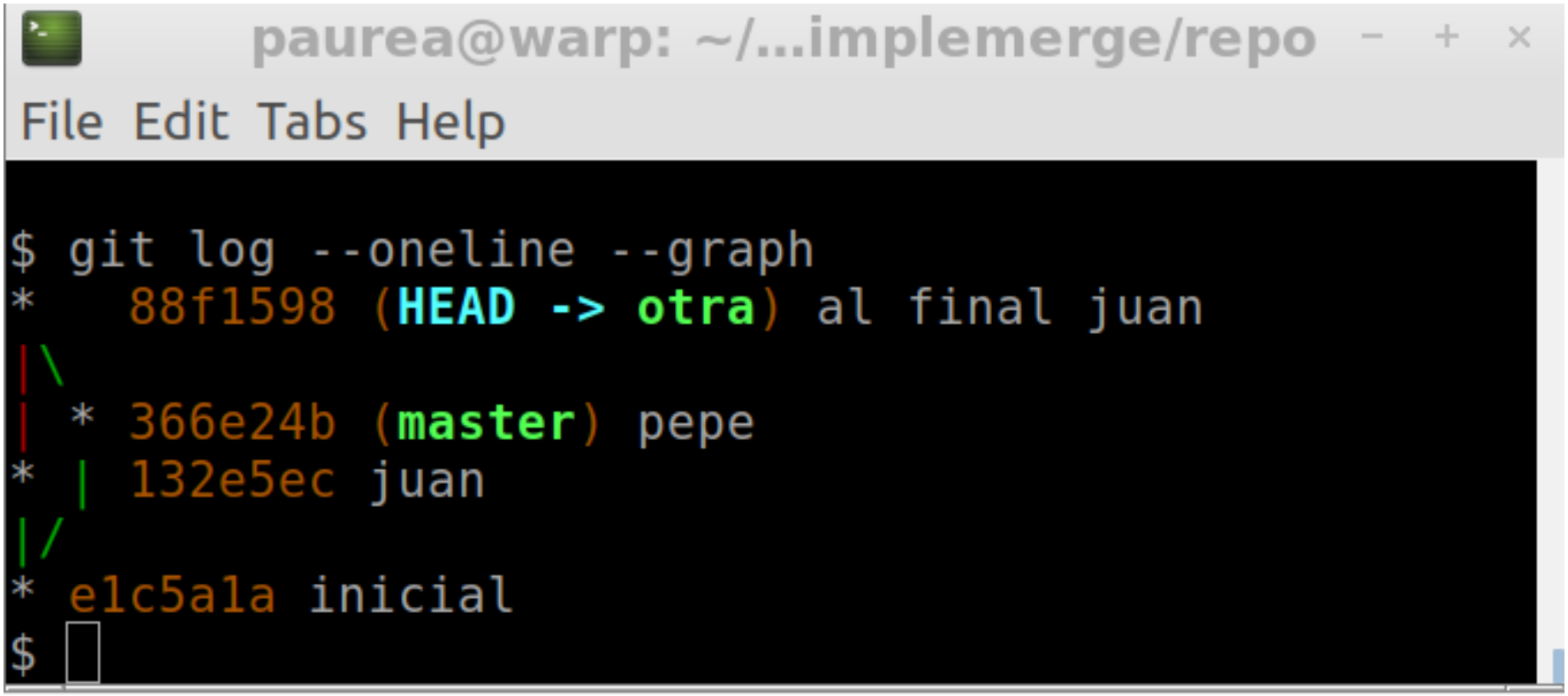
Conflictos con dos ramas

- Si no quiero hacerlo finalmente, en lugar del commit, puedo abortar

```
$ git mergetool #o editar con vi
$ git merge --abort
```

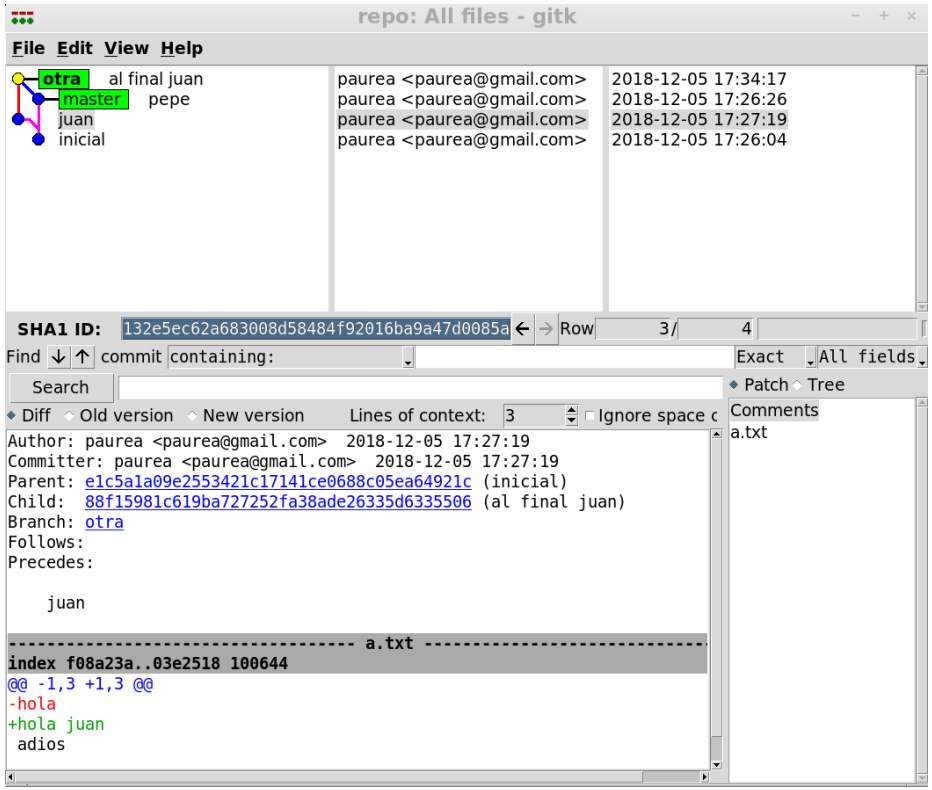
Conflictos con dos ramas

- Puedo ver el grafo de commits con `git log`
- Muchas opciones, pruébalas
- Interesantes: `--oneline --graph --all --decorate`
- Cada rama que diverge de un color, los números SHAs de commits



Conflictos con dos ramas

- Si quiero ver una versión más gráfica del grafo
- Puedo usar `gitk` (`apt install gitk`) en el directorio
- Mejor acostrumbrarse a las herramientas de línea de comando
- Usar gitk para aclararse mientras me acostumbro



mergetool

- Se da de alta en la configuración
- Permite ver lado a lado los ficheros en conflicto
- Hay varias, también puedo usar un IDE como Atom y viene todo integrado

mergetool

- Ejemplo, la herramienta `meld`

```
$ apt install meld
...
$ git config --global merge.tool meld
```

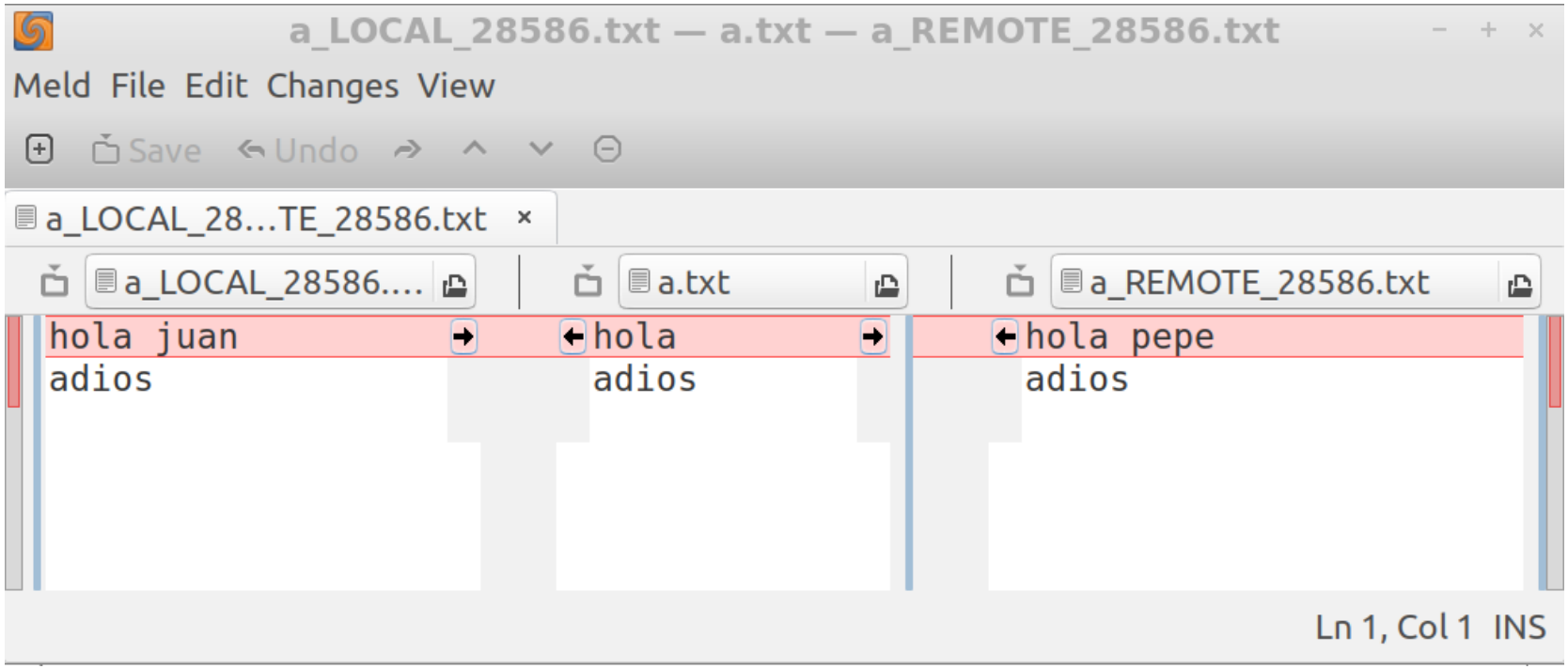
mergetool

- En el ejemplo de antes

```
 $ git merge master
Auto-merging a.txt
CONFLICT (content): Merge conflict in a.txt
Automatic merge failed; fix conflicts and then commit the result.
 $ git mergetool          #chant your mind to my mind
```

Resolución de conflictos

- A la izquierda una versión
- A la derecha otra
- En medio la que voy a dejar, muevo las líneas con las flechas o edito



Resolución de conflictos

- Cuando estoy contento, cierro y salvo

```
 $ git mergetool
Merging:
a.txt

Normal merge conflict for 'a.txt':
  local: modified file
  remote: modified file
Was the merge successful [y/n]? y
 $ rm a.txt.orig
 $ git commit -m a.txt 'mezclados'
```

Contenido

Ramas remotas

- En realidad, si tengo un repo remoto
- Son ramas especiales `repo/rama`
- No las puedo tocar, se actualizan con fetch (y pull)
- Lo último que vi de ese repo
- Volvemos al ejemplo del labo

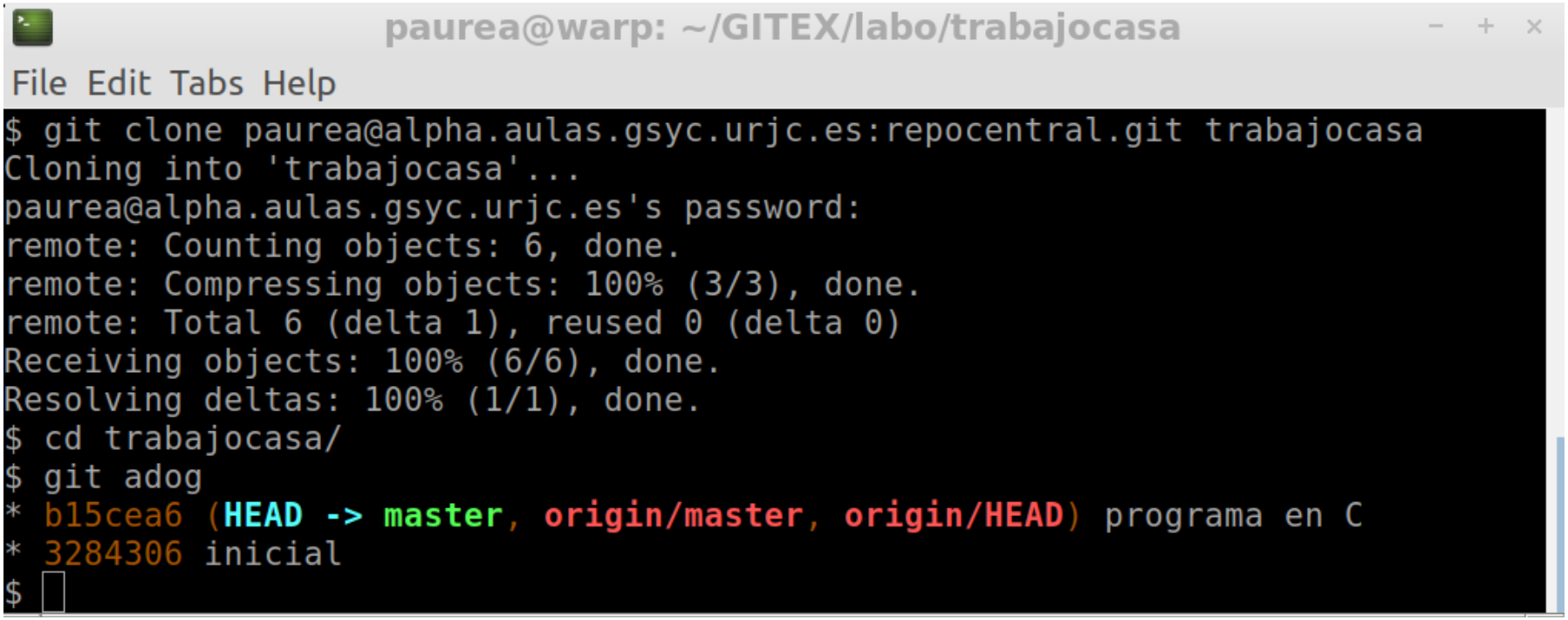
Ramas remotas

- Estoy en casa
- He clonado mi repocentral, con unos cuantos cambios
- Ejecuto mi nuevo comando

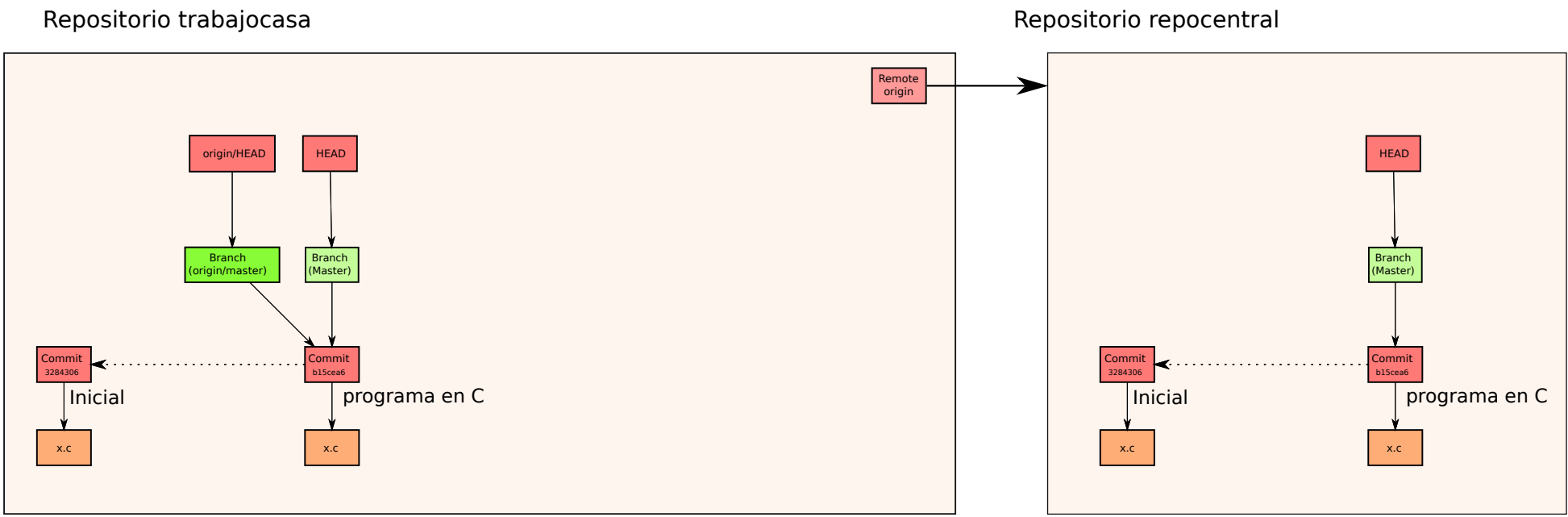
```
 $ git clone paurea@alpha.aulas.gsyc.urjc.es:repocentral.git trabajocasa
Cloning into 'trabajocasa'...
paurea@alpha.aulas.gsyc.urjc.es's password:
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 6 (delta 1), reused 0 (delta 0)
Receiving objects: 100% (6/6), done.
Resolving deltas: 100% (1/1), done.
 $ git adog
git adog
* b15cea6 (HEAD -> master, origin/master, origin/HEAD) programa en C
* 3284306 inicial
```

Ramas remotas

- En la shell te dibuja las cosas de colores (puedo verlo también con `gitk`)



Ramas remotas

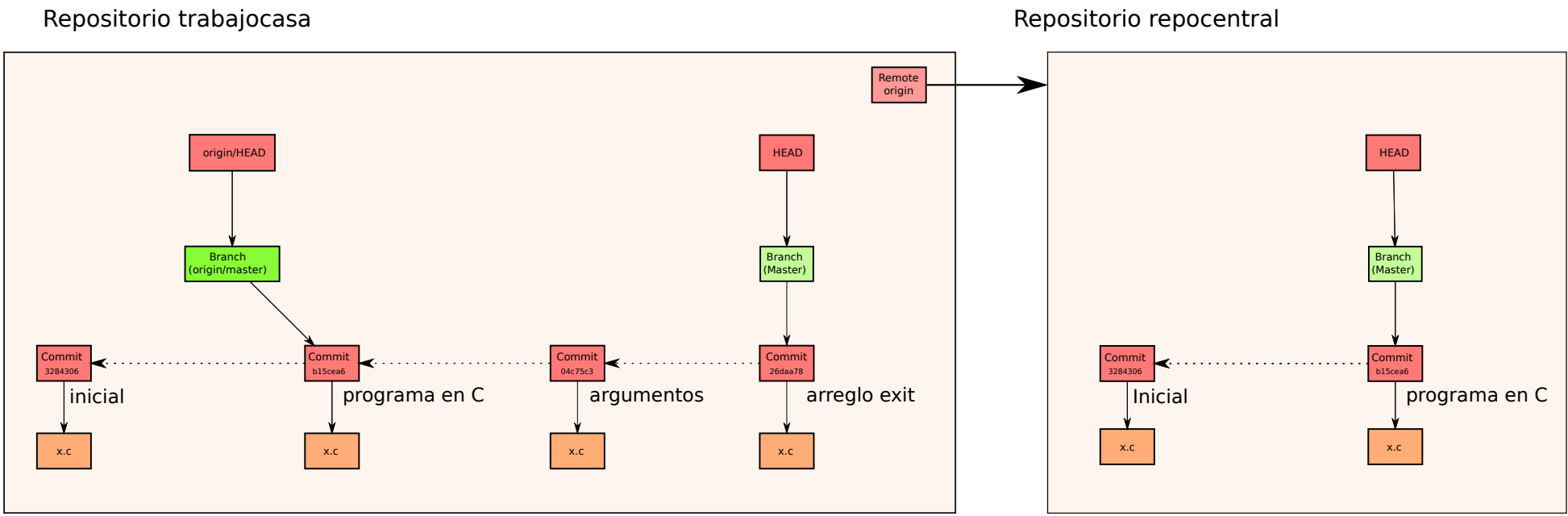


Ramas remotas

- Edito el fichero

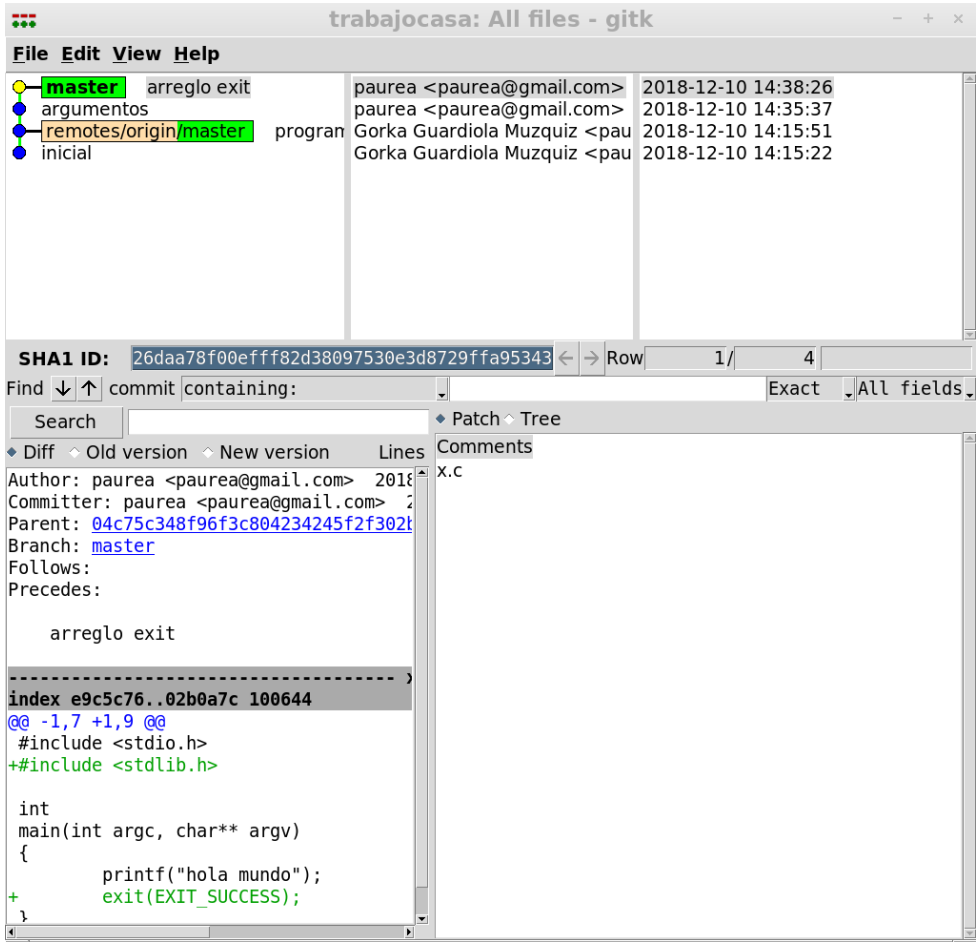
```
 $ vi x.c
 $ git commit -m 'argumentos'
$ git commit -m 'argumentos'
[master 04c75c3] argumentos
 1 file changed, 1 insertion(+), 1 deletion(-)
 $ git adog
* 04c75c3 (HEAD -> master) argumentos
* b15cea6 (origin/master, origin/HEAD) programa en C
* 3284306 inicial
 $ vi x.c
 $ git add x.c
 $ git commit -m 'arreglo exit'
[master 26daa78] arreglo exit
 1 file changed, 2 insertions(+)
 $ git adog
* 26daa78 (HEAD -> master) arreglo exit
* 04c75c3 argumentos
* b15cea6 (origin/master, origin/HEAD) programa en C
* 3284306 inicial
 $ cat x.c
#include <stdio.h>
#include <stdlib.h>

int
main(int argc, char** argv){
        printf("hola mundo");
        exit(EXIT_SUCCESS);
}
```

Ramas remotas



Ramas remotas

- Visto con gitk



Ramas remotas

- En la shell

Ramas remotas

- Ver las ramas
- Todas, incluidas las remotas se muestran con -a
- El head local se muestra con un asterisco
- Hay dos ramas remotas, HEAD remota y master remota

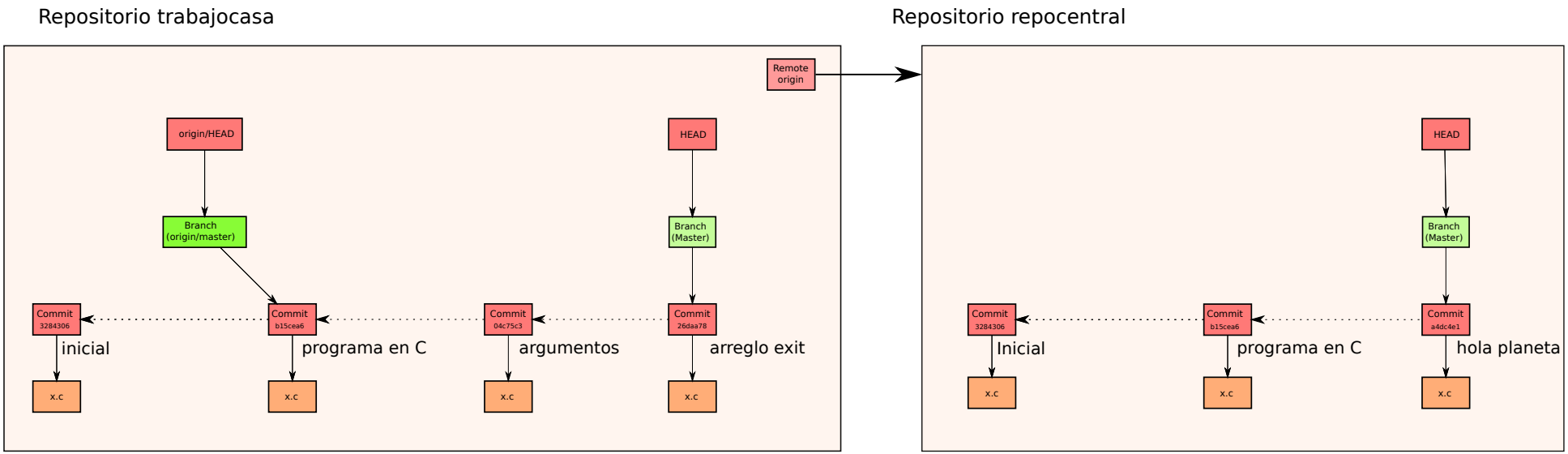
```
 $ git branch
* master
 $ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
 $ git remote
origin
```

Ramas remotas

- Si alguien en paralelo toca

```
 $ ssh paurea@alpha.aulas.gsyc.urjc.es
paurea@alpha.aulas.gsyc.urjc.es's password:
 $ cd trabajolabo
 paurea@alpha$ git adog
* b15cea6 (HEAD -> master, origin/master) programa en C
* 3284306 inicial
 paurea@alpha$ cat x.c
#include <stdio.h>

int
main()
{
        printf("hola planeta");
}
 paurea@alpha$ git add x.c
paurea@alpha$ git commit -m 'hola planeta'
[master a4dc4e1] hola planeta
...
 1 file changed, 1 insertion(+), 1 deletion(-)
 paurea@alpha$ git push
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 318 bytes | 318.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To /home/gsyc/paurea/repocentral
   b15cea6..a4dc4e1  master -> master
 paurea@alpha$ git adog
* a4dc4e1 (HEAD -> master, origin/master) hola planeta
* b15cea6 programa en C
* 3284306 inicial
```

Ramas remotas

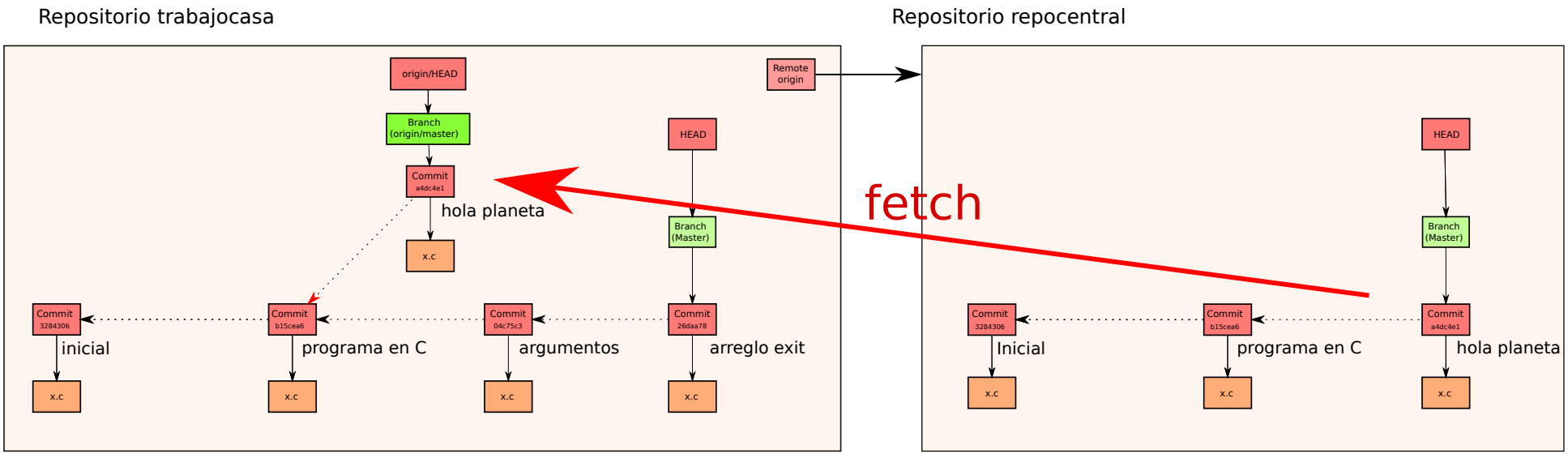


Ramas remotas

- Si alguien en paralelo toca

```
 $ git adog
* 26daa78 (HEAD -> master) arreglo exit
* 04c75c3 argumentos
* b15cea6 (origin/master, origin/HEAD) programa en C
* 3284306 inicial
$ git fetch
paurea@alpha.aulas.gsyc.urjc.es's password:
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From alpha.aulas.gsyc.urjc.es:repocentral
   b15cea6..a4dc4e1  master     -> origin/master
$ git adog
* a4dc4e1 (origin/master, origin/HEAD) hola planeta
| * 26daa78 (HEAD -> master) arreglo exit
| * 04c75c3 argumentos
|/
* b15cea6 programa en C
* 3284306 inicial
```

Ramas remotas



Ramas remotas

- Despues del fetch

Ramas remotas

- Hago el merge

```
 $ git merge
Auto-merging x.c
CONFLICT (content): Merge conflict in x.c
Automatic merge failed; fix conflicts and then commit the result.
 $ cat x.c
#include <stdio.h>
#include <stdlib.h>

int
main(int argc, char** argv)
{
<<<<<<< HEAD
        printf("hola mundo");
        exit(EXIT_SUCCESS);
=======
        printf("hola planeta");
>>>>>>> refs/remotes/origin/master
}
```

Ramas remotas

- Resuelvo el conflicto

```
 $ vi x.c
 $ cat x.c
#include <stdio.h>
#include <stdlib.h>

int
main(int argc, char** argv)
{
        printf("hola planeta");
        exit(EXIT_SUCCESS);
}
 $ git add x.c
 $ git commit -m 'mezcla planeta y mundo'
[master 5c80cd5] mezcla planeta y mundo
```

Ramas remotas

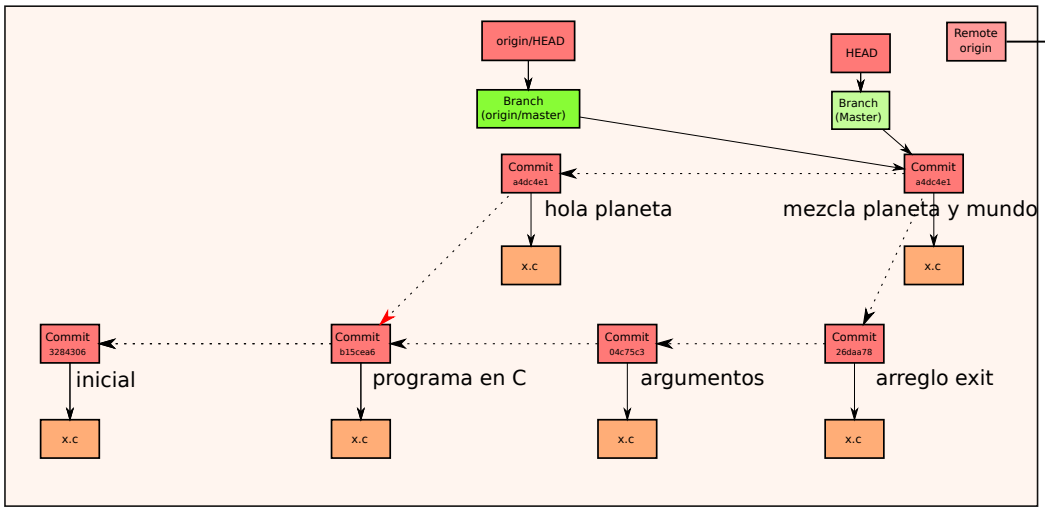
- Hago el merge

```
 $ git push
paurea@alpha.aulas.gsyc.urjc.es's password:

 $ git adog
*   5c80cd5 (HEAD -> master) mezcla planeta y mundo
| | * a4dc4e1 (origin/master, origin/HEAD) hola planeta
* | 26daa78 arreglo exit
* | 04c75c3 argumentos
|/
* b15cea6 programa en C
* 3284306 inicial
 $ git push
paurea@alpha.aulas.gsyc.urjc.es's password:
Counting objects: 9, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (9/9), 850 bytes | 850.00 KiB/s, done.
Total 9 (delta 1), reused 0 (delta 0)
To alpha.aulas.gsyc.urjc.es:repocentral.git
   a4dc4e1..5c80cd5  master -> master
 $ git adog
*   5c80cd5 (HEAD -> master, origin/master, origin/HEAD) mezcla planeta y mundo
| | * a4dc4e1 hola planeta
* | 26daa78 arreglo exit
* | 04c75c3 argumentos
|/
* b15cea6 programa en C
* 3284306 inicial
 $
```

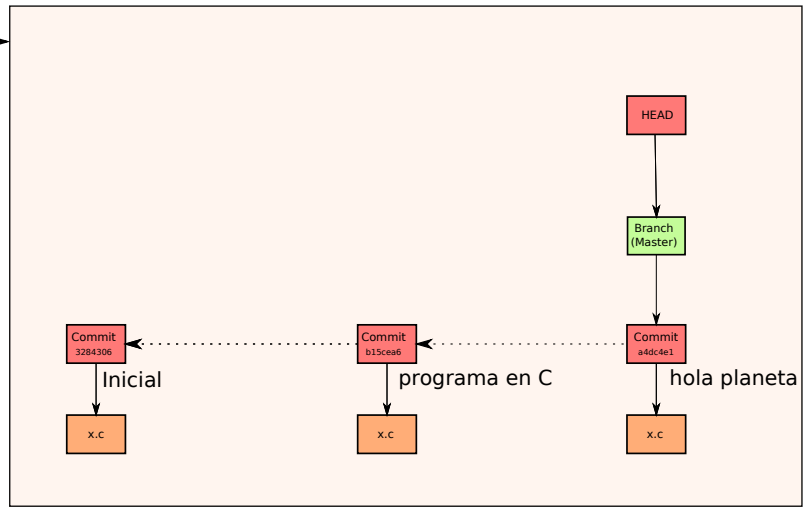
Ramas remotas

- Despues del merge

Repositorio trabajocasa

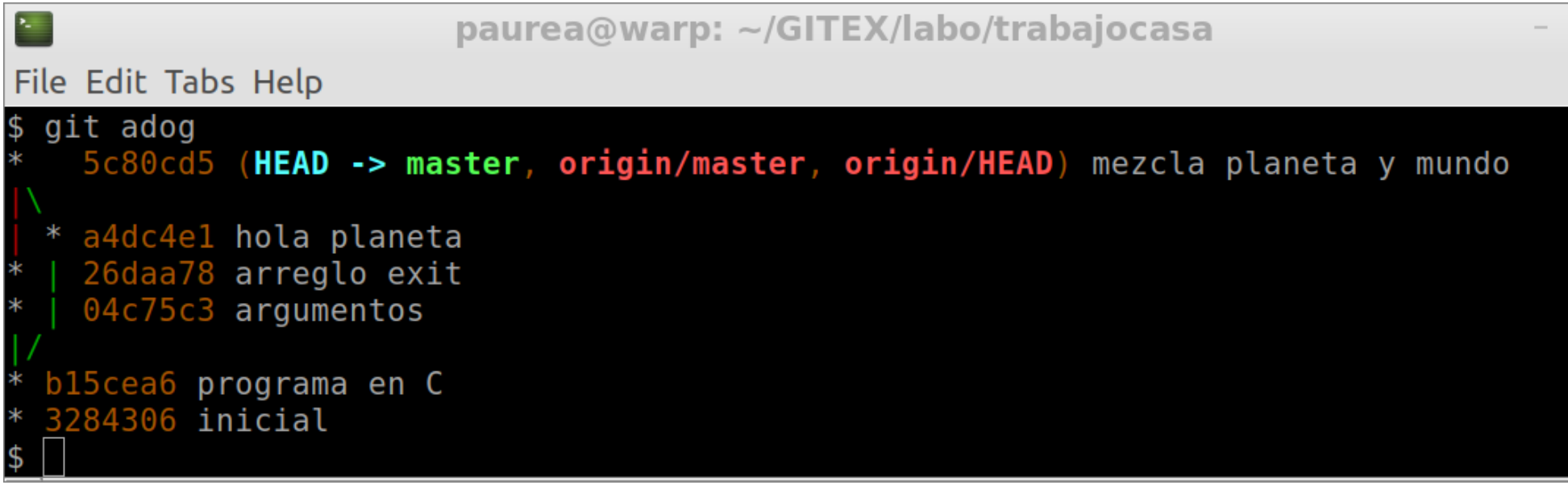


Repositorio repocentral



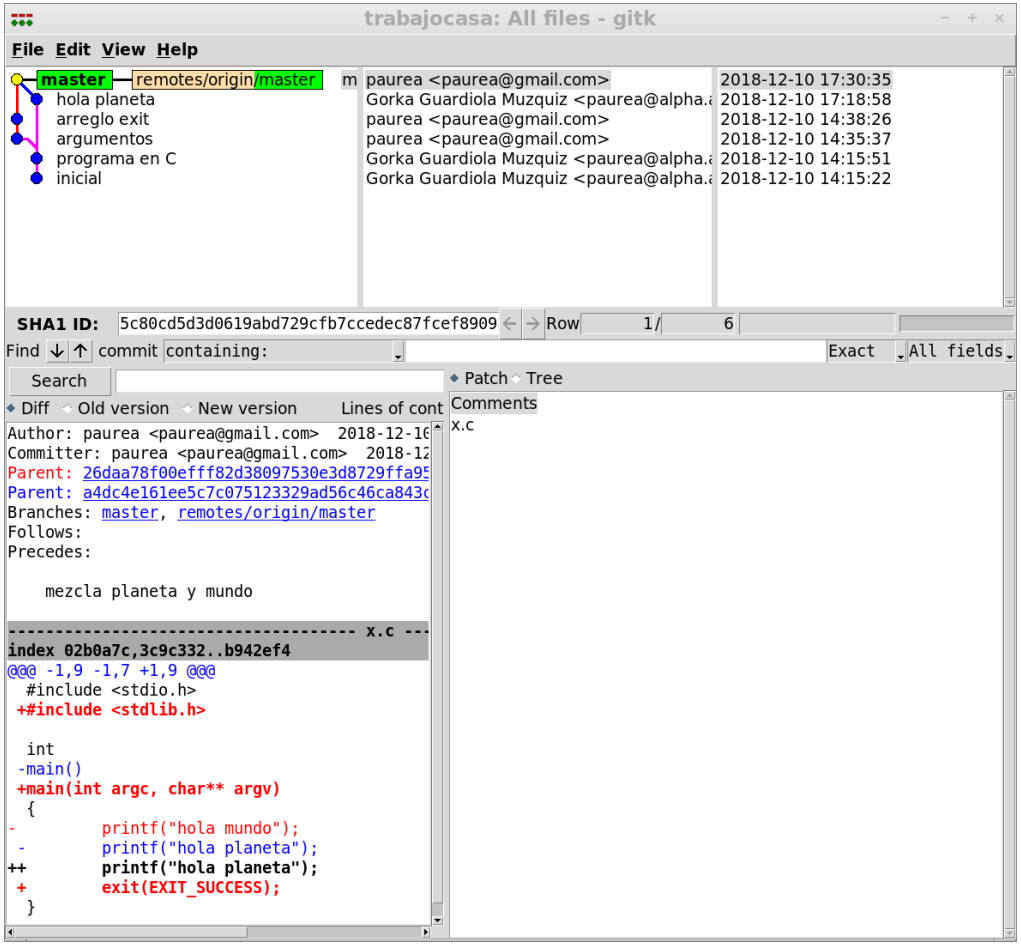
Ramas remotas

- Despues del merge, shell



Ramas remotas

- Despues del merge, gitk



Ramas

- `git fetch` seguido de `git merge FETCH_HEAD`
- Que se llama `git pull`

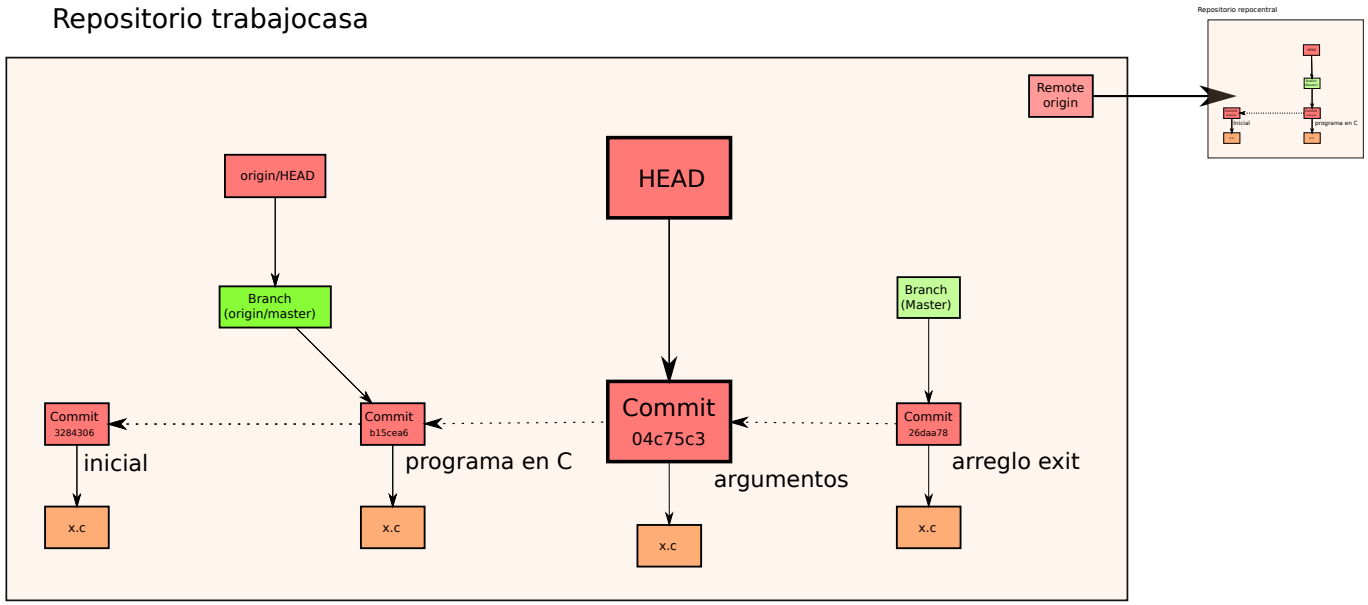
Contenido

Detached head

- Puedo apuntar HEAD a donde quiera, commit que quiera
- Haciendo `git checkout SHA`
- Ojo, si no apunto a una rama, estoy fuera creando commits en el aire
- Luego difíciles de recuperar.
- Puedo hacer el checkout y crear una rama (luego la borro con -d) `git checkout -b mirar SHA`

Detached head

- `git checkout 04c75c3`



Contenido

Detached head

- Cuando hago un checkout de un commit, mi HEAD no apunta a una rama
- Ver luego los commits, complicado
- Me puedo recuperar haciendo que se apunte de nuevo
- `git reset --hard HEAD`: OJO: pierde todo el estado: pone index y trabajo igual al HEAD
- Lo mismo pero soft, `git reset --soft HEAD`, sólo cambia head pero deja index y trabajo (peligroso)
- puedo hacer uno soft seguido de stash (ver más adelante) si hace falta
- Seguido de `git checkout branch` y vuelvo a una rama

Tags y cómo trabajo

- Haciendo `git tag vers3.4.5 ramabuena`
- O haciendo `git tag vers3.4.5 SHA`
  - Etiqueta navegable, para hacer releases, se puede firmar digitalmente
  - Ojo, push y demás no lo meten en el repo remoto, hay que hacerlo a mano
  - Una `git push origin vers3.4.5`
  - Todas `git push --tags`
  - Ver *semantic versioning*: `Major.Minor.Patch` `https://semver.org/`
    - **Major**, cambia api, ejemplo 3
    - **Minor** funcionalidad compatible, ejemplo 4
    - **Patch** arregla bugs, ejemplo 5
    - Ejemplo: 3.4.5
  - Luego puedo hacer `git checkout vers3.4.5`

Git stash

- Es una pila de estados que contienen index y trabajo, para luego
- Se salva y se recupera en un commit o en otro
- Algunos, como pop y apply pueden tener conflictos (se guarda un parche y se aplica al sacarlo)
- `git stash`: similar a push
- `git stash push`
- `git stash pop`
- `git stash list`
- `git stash apply`: aplica la cima al estado actual, no hace pop

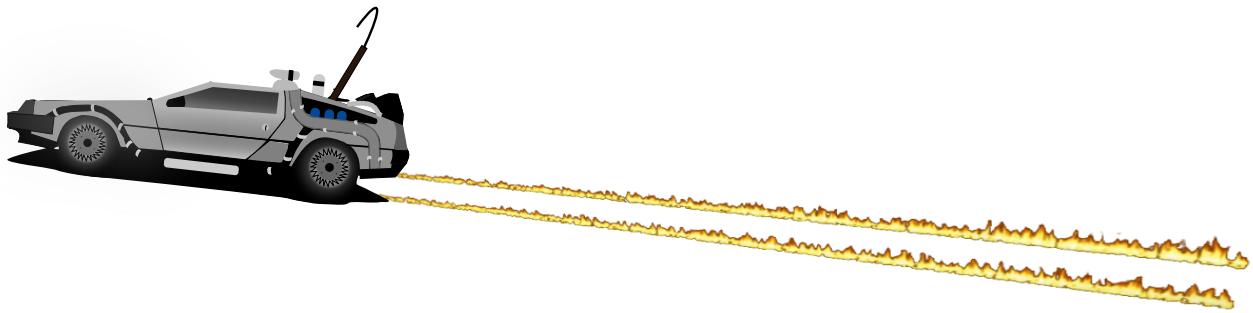
Arreglar el último commit

- `git commit --amend`
- Hace commit, pero sobreescribiendo el último

Mirar ficheros/árboles

- `git ls-tree SHA-del-commit`
- `git ls-tree SHA-del-árbol`
- `git cat-file SHA-del-fichero`
- `git show SHA-del-commit:dir/dirx/fichero`

Edición del grafo



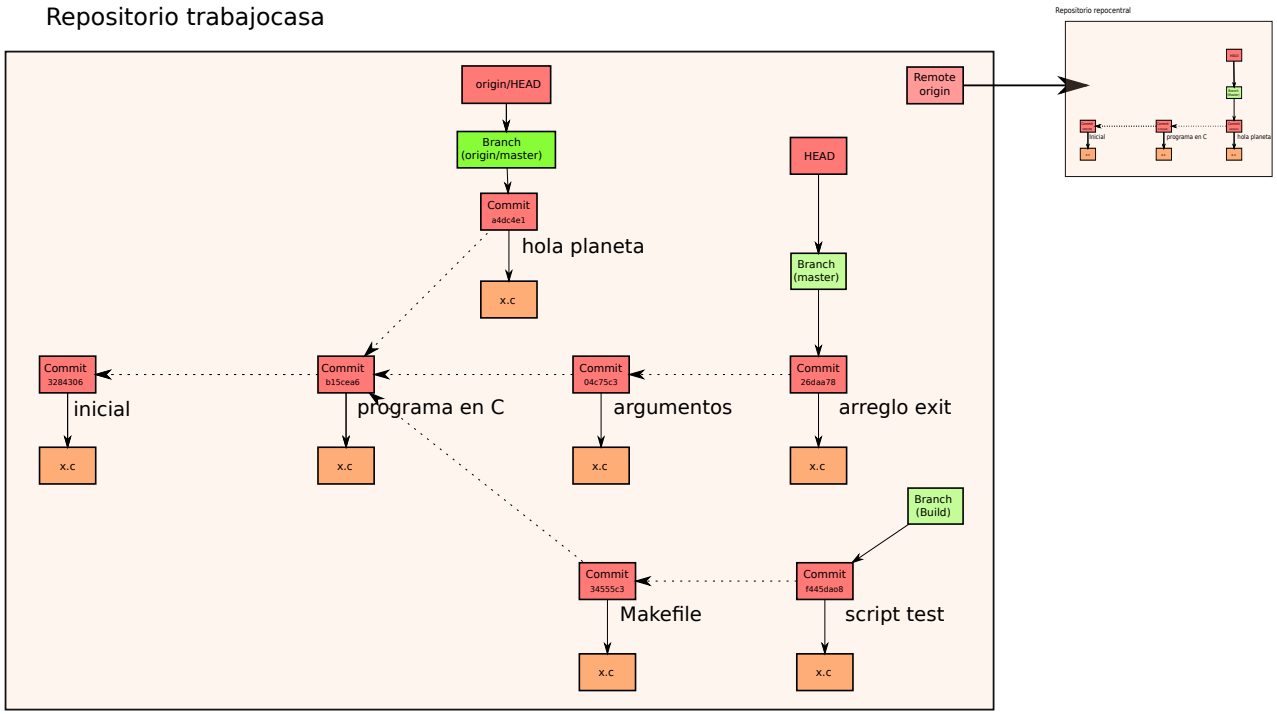
- git rebase, tres tipos
  - Mezcla ramas
    - ★ Simple
    - ★ Con tres ramas (origen, para calcular el antepasado común, destino)
  - Interactivo

Primer tipo, rebase simple

- Fusionar dos ramas
- Haciendo los commits de una en otra

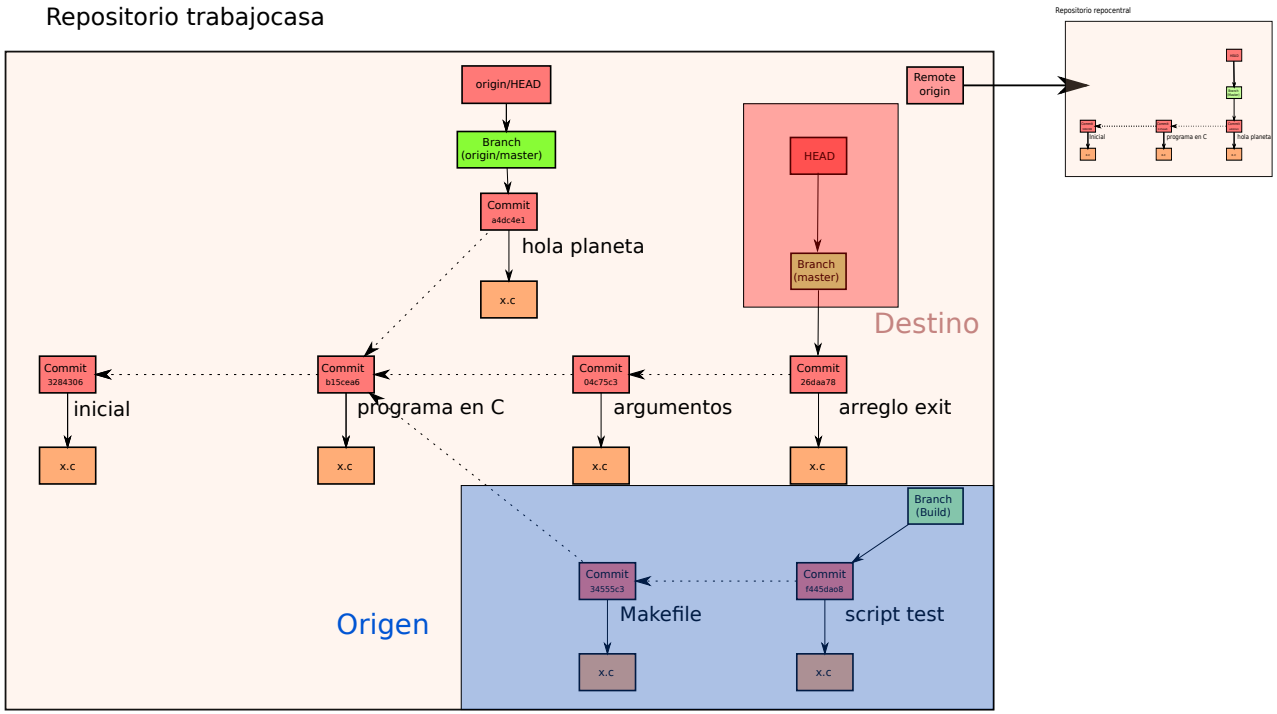
Rebase simple

```
$ git checkout master
```



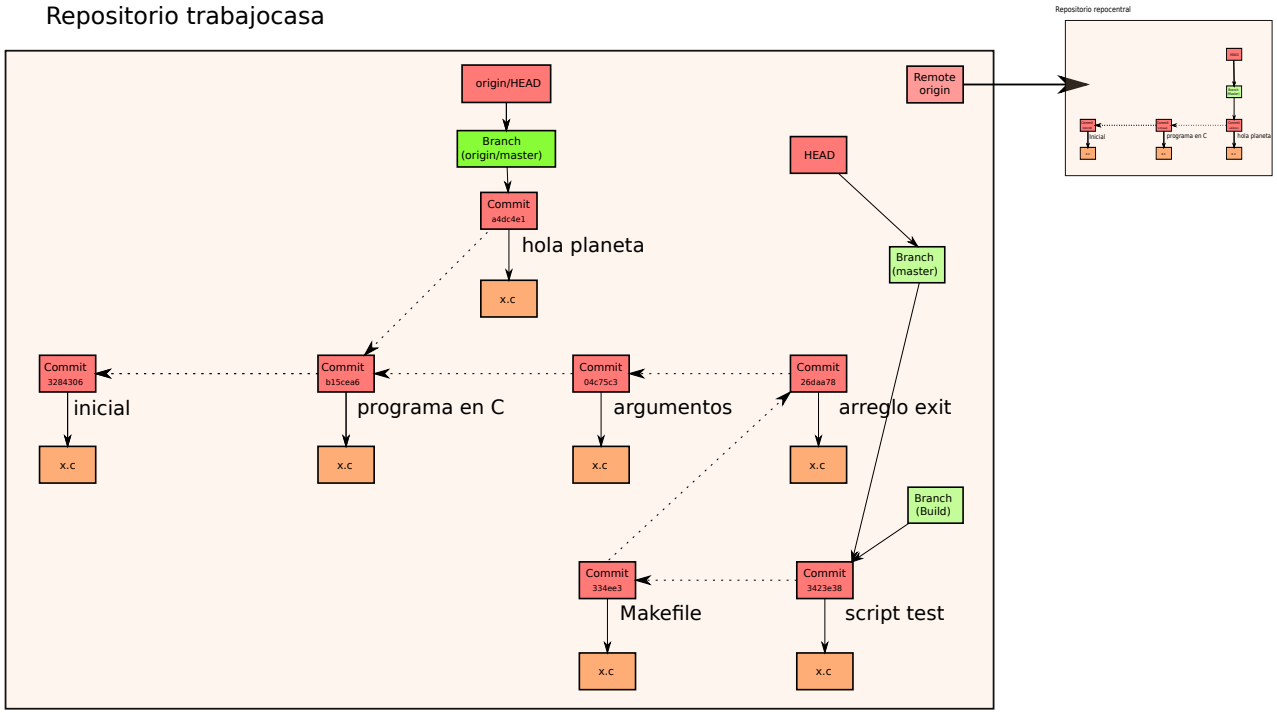
Rebase simple

```
$ git rebase Build
```



Rebase simple

- Queda una sóla línea de commits
- Puede haber conflictos

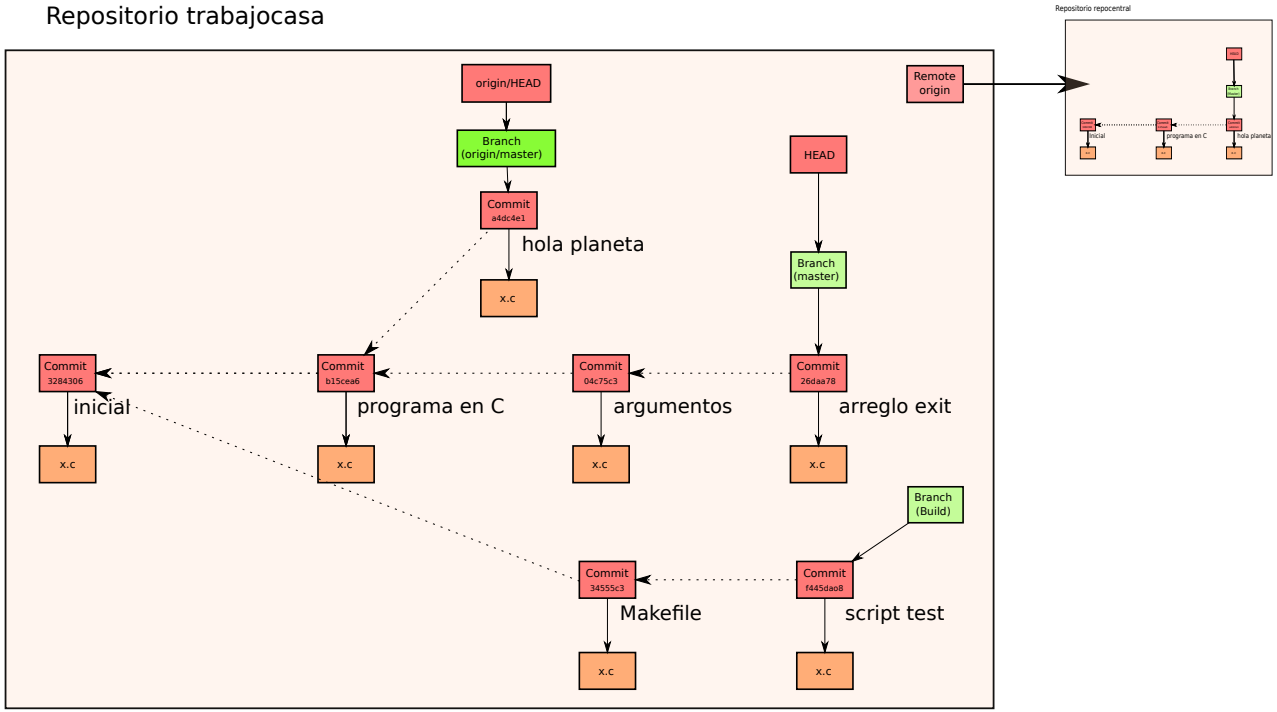


Primer tipo, rebase complicado

- Fusionar dos ramas con un tercera de referencia
- Origen (rama de la que vienen los commits)
- Destino (rama a la que se aplican los commits)

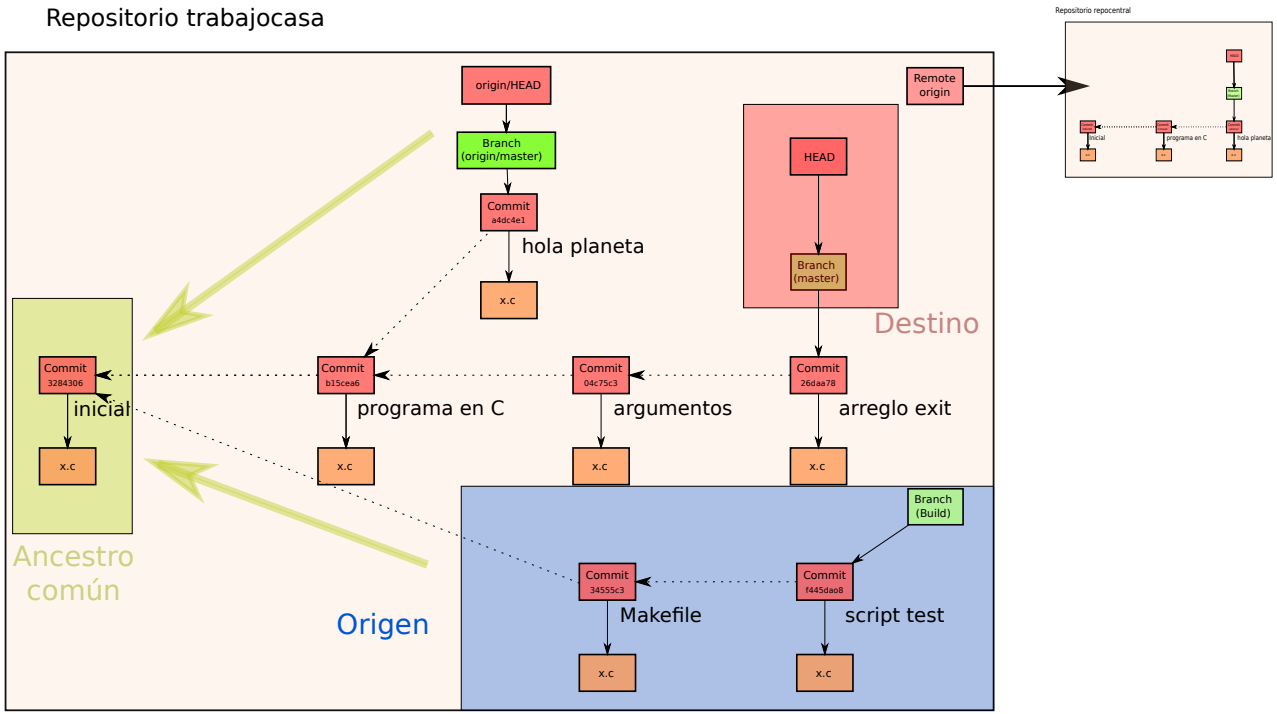
Rebase complicado

```
$ git checkout master
```



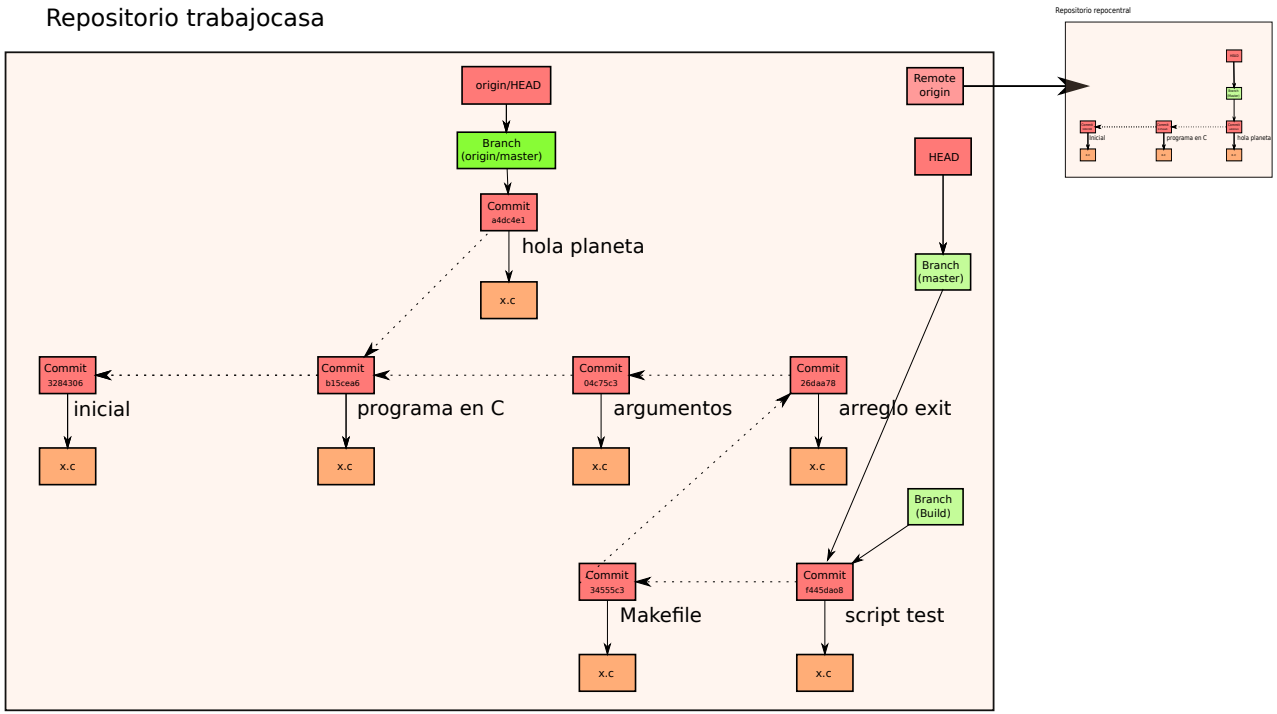
Rebase complicado

```
$ git rebase --onto master origin/master Build
```



Rebase complicado

```
$ git rebase --onto master origin/master Build
```



Rebase

- Ojo, en ambos casos, los commits cambian de SHA
- Ojo, no hacer estas cosas con commits visibles/compartidos (otra gente puede depender de ellos)

Rebase interactivo

- Se ejecuta y se abre un fichero con comandos
- Los comandos
- `--root` *root* para editar el primer commit (desde el) `git rebase -i --root $tip`
  - Pick: dejar el commit como está
  - Edit: cambiar el commit (luego, de forma interactiva)
  - Reword: cambiar el mensaje de commit
  - Drop: borrar el commit
  - Squash: generar un commit con el mensaje del anterior y ambos en mezclados
  - Exec: ejecutar un comando de shell

Rebase interactivo

```
 $ git rebase -i HEAD~4   #hace 4 commits o SHA o una referencia

#edita el fichero a continuación
pick 234234c Programa inicial
pick 56dbd1 Opcion -a
pick 2527dec Opcion -b
pick 0e5b2d1 Makefile

 # Rebase 170afb6..02e5bd1 onto 170afb6 (2 command(s))
 #
 # Commands:
 # p, pick = use commit
 # r, reword = use commit, but edit the commit message
 # e, edit = use commit, but stop for amending
 # s, squash = use commit, but meld into previous commit
 # f, fixup = like "squash", but discard this commit's log message
 # x, exec = run command (the rest of the line) using shell
 # d, drop = remove commit
```

Rebase interactivo

```
#salvo el fichero y voy editando, etc.
pick 234234c Programa inicial
pick 56dbd1 Opcion -a
squash 2527dec Opcion -b
edit 0e5b2d1 Makefile

 # Rebase 170afb6..02e5bd1 onto 170afb6 (2 command(s))
 #
 # Commands:
 # p, pick = use commit
 # r, reword = use commit, but edit the commit message
 # e, edit = use commit, but stop for amending
 # s, squash = use commit, but meld into previous commit
 # f, fixup = like "squash", but discard this commit's log message
 # x, exec = run command (the rest of the line) using shell
 # d, drop = remove commit
```

Flujo compartido

- Crear ramas locales
- Fundir en master (merge/rebase)
- Ojo, no hacer rebase de nada público
- Hacer push sólo de master

Ramas remotas

- Crear ramas en el repo compartido: **git push origin ramalocal**
- En realidad estoy creando una rama remota
- Para no tener que estar que estar haciendo: **git pull origin ramalocal**
- La marco para que la siga (con pull se trae todas): **git push –set-upstream origin ramalocal**

Contenido

Colaboración

- Pull request, patches
- Lo suyo es que haya tags (aquí la release v1.0)
- Tengo un clon y un repo mío público central
- El otro puede hacer pull (también puedo generar un parche con el mismo comando)

```
 $ git push https://git.pa/mirepopublico master
 $ git request-pull v1.0 https://git.pa/mirepopublico master
The following changes since commit b7a2ad67819d251cc36a8e6ccf70c3fc9b53b23e:

  hola (2018-12-18 12:54:59 +0100)

are available in the Git repository at:

  https://git.pa/mirepopublico

for you to fetch changes up to b7a2ad67819d251cc36a8e6ccf70c3fc9b53b23e:

  hola (2018-12-18 12:54:59 +0100)

----------------------------------------------------------------
```

Colaboración (github)

- Hago push, pongo tags/miro commits
- Uso el interfaz web:
  - `https://help.github.com/articles/creating-a-pull-request/`
  - `https://help.github.com/articles/merging-a-pull-request/`
- Puedo usar Hub para la línea de comando `https://hub.github.com/`

Contenido

git cherry-pick

- `git cherry-pick SHA`, por ejemplo `git cherry-pick d467740`
- Lo aplica en el branch actual (con otro nombre)
- Si hay conflicto, se resuelve y `git cherry-pick --continue` o SHA `git cherry-pick --abort`
- Peligro: mismo commit con SHA diferente, tener cuidado

Hooks y automatización

- Hay un directorio `.git/hooks` con scripts que se ejecutan
- El nombre dice cuando se aplican, por ejemplo `pre-push.sample`
- Quitar el `.sample` del final y dar permisos de ejecución
- Hooks para tests, formateo...