5. Sistemas de Control de Versiones

5.3 Git avanzado

Elena García-Morato, Felipe Ortega, Enrique Soriano, Gorka Guardiola

GSyC, ETSIT. URJC.

Laboratorio de Sistemas (LSIS)

13 abril, 2023

Universidad
Rey Juan Carlos

Tema 5 - Sistemas de Control de Versiones

5.3 Git avanzado

Contenidos

- 5.3.1 Ramas
- 5.3.2 Deshacer cambios
- 5.3.3 Pull requests
- 5.3.4 Etiquetado
- 5.3.5 Diff y patch

5.3.1 Ramas

Ciclo de cambios en archivos del proyecto

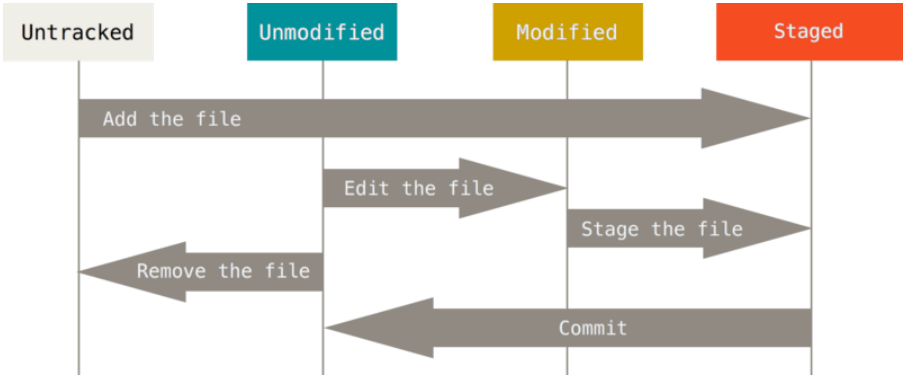


Figura 1: Resumen del ciclo de registro de cambios en los archivos de un proyecto Git. Fuente: git-scm (Sec. 2.2).

Propósito de las ramas

- Si todo el mundo trabaja simultáneamente en la misma copia del proyecto será muy habitual que surjan problemas: conflictos por cambios incompatibles al hacer un *merge*, etc.
- Casi todos los SCV incluyen soporte para crear ramas (*branches*).
- Una rama es, a todos los efectos, una versión alternativa del proyecto. En concreto, en Git se crean a partir de una instantánea del estado actual del proyecto.
- Hábito de trabajo: cuando creamos una nueva *feature* o arreglamos un *bug* primero creamos una nueva rama.
  - Los cambios en esa rama no son visibles en el resto hasta que la volvamos a integrar en otra rama o en la rama principal.
  - Así se evita filtrar código inestable en la rama principal del proyecto. También permite organizar y limpiar todos los cambios antes de integrarlos de vuelta.

Ramas en un proyecto

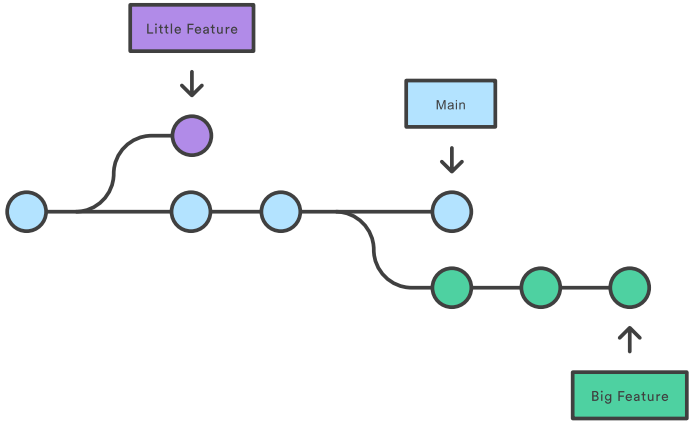


Figura 2: Esquema que ilustra la utilidad de la creación de ramas en un repo Git. Fuente: Tutorial de Atlassian Bitbucket.

Creación de ramas

- Crear nuevas ramas en Git es un proceso rápido (casi instantáneo) y no consume grandes recursos.
- Cambiar de una rama a otra es igual de fácil y rápido. Por eso, se incentiva la creación de ramas (una para cada bug, para cada nueva feature, etc.).
- Recordemos como se construye el grafo de cambios.
  - Cada *commit* contiene un puntero a una instantánea del los cambios añadidos al *stage*, nombre y e-mail del autor, mensaje descriptivo, y un puntero o varios a los ancestros.
  - El *commit* inicial no tiene ancestros, un *commit* normal tiene un ancestro y un *merge* tiene dos o más.

Creación de ramas

- Supongamos un nuevo repo en el que añadimos tres ficheros.

```
$ git init
$ git add README test.rb LICENSE
$ git commit -m "Primer commit"
```

Creación de ramas

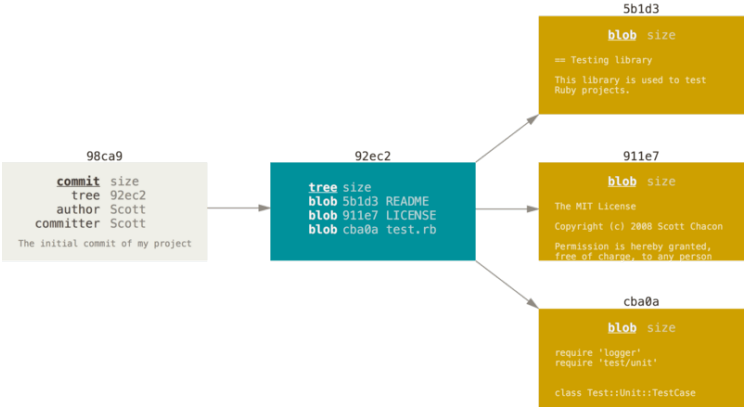


Figura 3: Commit inicial y su árbol. Fuente: Pro Git.

Creación de ramas

- Cuando hacemos dos cambios más, cada cambio guarda una instantánea del estado del proyecto.

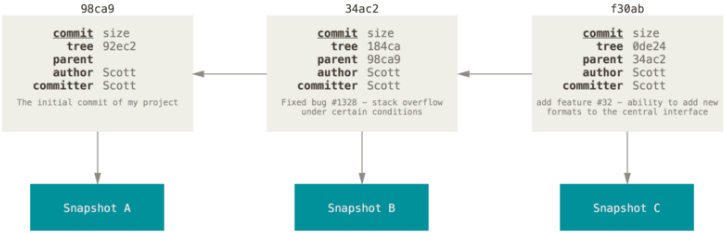


Figura 4: Grafo de cambios con tres *commits*. Fuente: Pro Git.

Creación de ramas

- Una rama es un puntero a uno de estos *commits* y su instantánea del proyecto. Por defecto, al crear un repo se crea una rama `main` o `master`.
- Al hacer *commits*, el puntero se mueve adelante automáticamente.

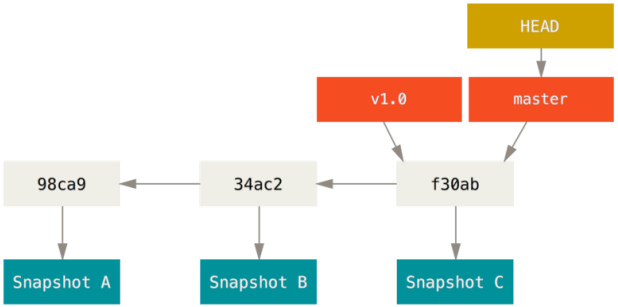


Figura 5: La rama `master` apunta al último *commit*. Aquí, también se ha etiquetado como `v1.0`. Fuente: Pro Git.

Creación de ramas

- Al crear una nueva rama surge otro puntero que apunta al mismo *commit* que la rama actual.

```
$ git branch testing
```

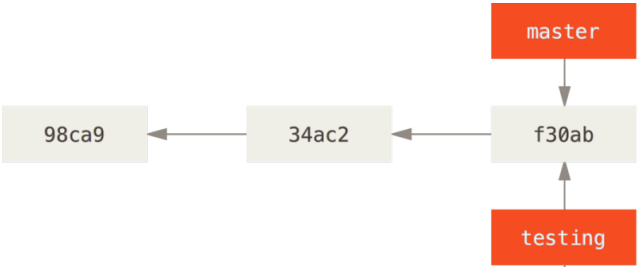


Figura 6: Se crea la rama `testing`, que también apunta al último *commit*. Fuente: Pro Git.

Creación de ramas

- El puntero HEAD es el que indica mi rama actual de trabajo.

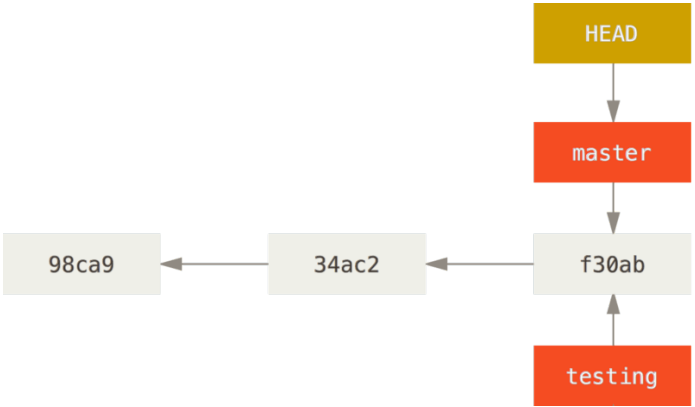


Figura 7: El puntero HEAD apunta a la rama `master`. Fuente: Pro Git.

Creación de ramas

- Vemos que las dos ramas están apuntando al mismo *commit*.

```
$ git log --oneline --decorate
f30ab (HEAD -> master, testing) Add feature #32 - ability to add new formats to central intf
34ac2 Fix bug #1328 - stack overflow under certain conditions
98ca9 Initial commit
```

Cambio de rama

- Para cambiar de rama uso el comando `git checkout branch-name`.

```
$ git checkout testing
```

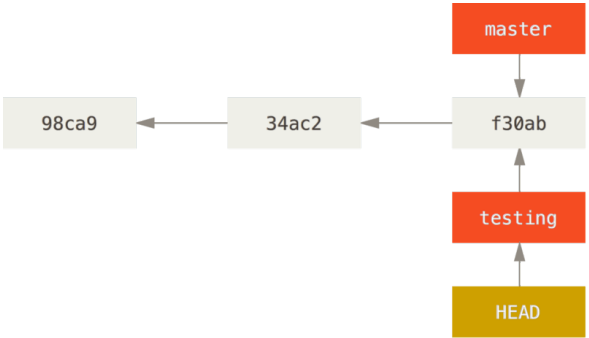


Figura 8: El puntero `HEAD` ahora apunta a la rama `testing`. Fuente: Pro Git.

Commits en la nueva rama

```
$ vim test.rb
$ git commit -a -m 'made a change'
```

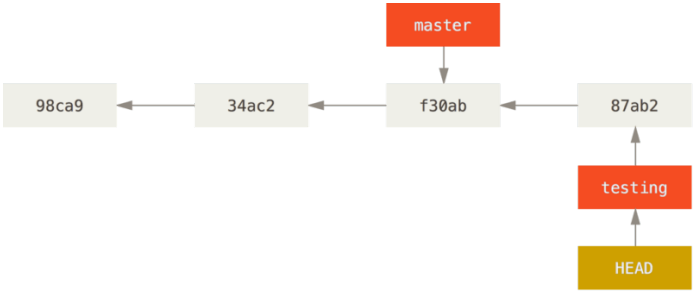


Figura 9: El puntero `HEAD` y la rama `testing` reflejan el último cambio. La rama `main` no avanza. Fuente: Pro Git.

Commits en la nueva rama

- Ahora, volvemos a la rama `master` y hacemos un *commit*.

```
$ git checkout master
$ vim test.rb
$ git commit -a -m 'made other changes'
```

- Ahora tenemos una divergencia en el historial de cambios del proyecto.

Commits en la nueva rama

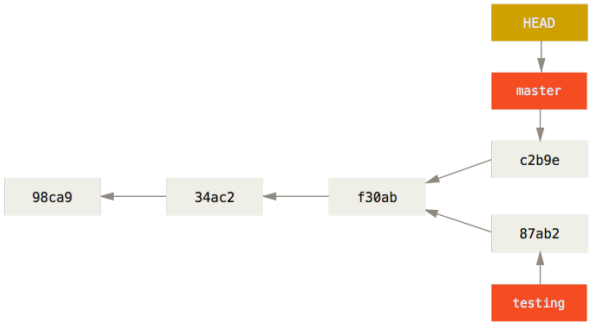


Figura 10: Nuevo historial divergente de cambios del proyecto. Fuente: Pro Git.

Mostrar el historial de cambios

- Cuidado con el comando `git log`. Por defecto, solo nos muestra el historial de cambios de **la rama actualmente en uso**.
- Para mostrar los cambios de una rama en concreto: `git log nombre-rama`.
- Para mostrar los cambios de todas las ramas: `git log -all`.

```
$ git log --oneline --decorate --graph --all
* c2b9e (HEAD, master) Made other changes
| * 87ab2 (testing) Made a change
|/
* f30ab Add feature #32 - ability to add new formats to the central interface
* 34ac2 Fix bug #1328 - stack overflow under certain conditions
* 98ca9 initial commit of my project
```

Comando `git switch`

- Desde Git v2.23 se puede usar el comando `git switch`.

```
$ git switch testing  # Cambiar a la rama testing
$ git switch -c new-branch  # Crea la nueva rama new-branch
$ git switch - # Cambia a la rama en la que estabas previamente
```

Tutoriales sobre ramas

- Ejemplo paso a paso de creación e integración de ramas (*merging*).
- Gestión de ramas.
  - Mucho cuidado con renombrar la rama principal (`main` o `master`). Podemos romper muchas utilidades y servicios integrados con nuestro repositorio.
- *Stashing* y limpieza de cambios.
  - Si tenemos cambios pendientes de *commit* en una rama Git no nos deja cambiar a otra. Primero hay que "salvar" los cambios (stash).
- Estrategias de trabajo con ramas.

5.3.2 Deshacer cambios

Modificación de *commits*

- Un caso habitual es que olvidamos añadir uno o varios archivos a un cambio y hacemos *commit*
- El fallo se puede arreglar fácilmente con el comando `git commit --amend`.
- Como resultado, el *commit* inicial se descarta y se reemplaza por completo con la nueva versión de cambios, como si el anterior nunca hubiese ocurrido.

```
$ git commit -m 'Initial commit'
$ git add forgotten_file
$ git commit --amend
```

- Precaución: Solo se debe hacer un `git commit --ammend` **sobre cambios que no se han enviado con `git push`** a otro repo remoto. Si lo hacemos y forzamos los cambios en esa rama podemos crear problemas a otros colaboradores.

Gestión del área *staging*

- Otro caso habitual es descartar los últimos cambios que hemos añadido al área de *stating*.

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

renamed:    README.md -> README
modified:   CONTRIBUTING.md
```

- Se puede observar que la propia salida del comando nos sugiere la fórmula para deshacer la operación: `(use "git reset HEAD <file>..." to unstage)`.

Gestión del área *staging*

```
$ git reset HEAD CONTRIBUTING.md
Unstaged changes after reset:
M^^ICONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)

renamed:    README.md -> README

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:   CONTRIBUTING.md
```

Gestión del área de trabajo

- Si queremos descartar por completo los últimos cambios en un archivo antes de añadirlo al área de *staging* usamos:

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

renamed:    README.md -> README
```

Gestión del área de trabajo

- Precaución: Mucho cuidado con el comando `git checkout -- <file>`. **La operación no se puede revertir**.
- Git no tiene "papelera de reciclaje" ni ninguna salvaguarda similar.
- La versión modificada del archivo se sobrescribe con la última versión almacenada (en el último *commit* realizado). Debemos estar absolutamente seguros de que es eso lo que queremos.

Comando `git restore`

- A partir de Git v2.23.0 se puede usar el comando git restore para realizar cambios equivalentes en el área de *staging* o el área de trabajo.

```
git restore --staged <file>  # Para deshacer cambios en staging area
git restore <file> # Para descartar cambios en un archivo del dir de trabajo
```

- En ese caso, los mensajes de sugerencia de `git status` ya aparecen actualizados.

5.3.3 Pull requests

Cómo contribuir a un proyecto

- Existen diversas maneras de contribuir a un proyecto software que está bajo control con Git.
- Formas de contribuir a un proyecto con Git.
- Aquí nos centramos en un caso común: la creación e incorporación de un `pull request`.
- Existe soporte específico en servicios como GitHub o GitLab para este caso.
  - GitHub: Crear pull request a partir de un fork e integrar un pull request.
  - GitLab: Crear *merge requests*.

Propósito de un *pull request*

- Queremos enviar cambios a un repositorio en el que no tenemos privilegios de edición.
- El primer paso es clonar el repo principal y crear una nueva rama para implementar nuestros cambios en ella.

```
$ git clone <url>
$ cd project
$ git checkout -b featureA
... work ...
$ git commit
... work ...
$ git commit
```

Realizar *pull request*

- Ahora, en el servicio de gestión de proyectos (GitHub, GitLab, etc.) tenemos que hacer un *fork* del proyecto principal para crear una copia que cuelgue de nuestra cuenta de usuario/a.
- Anotamos la URL del *fork* que hemos creado y lo usamos:

```
$ git remote add myfork <url>
$ git push -u myfork featureA
```

- Por último seguimos los pasos en la plataforma de gestión para crear el `pull request` de forma interactiva (instrucciones para GitHub).
- El comando `git request-pull` genera información sobre la petición que se puede enviar a los gestores del proyecto mediante un correo-e de contacto.

5.3.4 Etiquetado

Propósito de las etiquetas

- Como en otros SCV, Git permite poner etiquetas a puntos específicos en el historial del repositorio para marcar hitos relevantes.
- Normalmente se suele usar para versionado (v1.0, v1.1, etc.).
- El comando que muestra todas las etiquetas de un repo es `git tag`.
- El uso de la opción `-l` o `--list` es obligatorio si usamos comodines para patrones de búsqueda.

```
$ git tag
v1.0
v2.0
$ git tag -l "v1.8.5*" # Busca etiquetas que coinciden con un patrón
v1.8.5
v1.8.5-rc0
v1.8.5-rc1
v1.8.5.1
```

Creación de etiquetas

- Se pueden crear etiquetas de dos tipos: `lightweight` o `annotated`.
- Una etiqueta `lightweight` es como una rama sin recorrido (un puntero a una instantánea del proyecto). Sirven para marcar hitos internos o de poca importancia.
- Una etiqueta `annotated` se usa para hitos relevantes, como versiones publicadas. Son objetos completos, guardados en la base de datos del repo Git, que incluyen:
  - *Checksum*.
  - Nombre y correo-e del etiquetador.
  - Fecha.
  - Mensaje de descripción.
  - Se pueden firmar de forma segura (por ejemplo con GPG).

Creación de etiquetas

- Para crear etiquetas annotated:

```
$ git tag -a v1.4 -m "my version 1.4"
$ git tag
v0.1
v1.3
v1.4
```

Creación de etiquetas

```
$ git show v1.4
tag v1.4
Tagger: Supercoco Coder <super@coco.cc>
Date:   Fri May 6 21:19:12 2022 +0100

my version 1.4

commit ca82a6dff817ec66f44342007202690a93763949
Author: Count Counting <count@sesame-street.com>
Date:   Mon Mar 21 21:52:11 2022 -0700

Change version number
```

Creación de etiquetas

- Para crear etiquetas `lightweight`:

```
$ git tag v1.4-lw
$ git tag
...
$ git show v1.4-lw
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

Change version number
```

Creación de etiquetas

- Para anotar commits anteriores al HEAD:

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 Create write support
0d52aaab4479697da7686c15f77a3d64d9165190 One more thing

$ git tag -a v1.2 0d52aaa
```

Compartir etiquetas

- Por defecto, las etiquetas no se envían a los repos remotos en una operación `git push`.
- Se deben enviar explícitamente usando ese mismo comando:

```
$ git push origin v1.5
Counting objects: 14, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (12/12), done.
Writing objects: 100% (14/14), 2.05 KiB | 0 bytes/s, done.
Total 14 (delta 3), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
* [new tag]         v1.5 -> v1.5
```

Compartir etiquetas

- Si hay muchas etiquetas pendientes de enviar al repo remoto, se puede usar el comando `git push origin –tags`.
- Transfiere al repo remoto todas las etiquetas locales, tanto `lightweight` como `annotated` que todavía no estén allí.

```
$ git push origin --tags
Counting objects: 1, done.
Writing objects: 100% (1/1), 160 bytes | 0 bytes/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
* [new tag]         v1.4 -> v1.4
* [new tag]         v1.4-lw -> v1.4-lw
```

- Para enviar solo las etiquetas `annotated` se usa `git push <remote> –follow-tags`.

Borrar etiquetas

- Para borrar etiquetas del repo local, se usa `git tag -d <tagname>`.
- Para borrar etiquetas del repo remoto hay dos alternativas:

```
$ git push <remote> :refs/tags/<tagname>
$ git push origin --delete <tagname>
```

- Para enviar solo las etiquetas annotated se usa `git push <remote> -follow-tags`.

5.3.5 Diff y patch

Merge de branches

- Para entender como funciona debemos comprender `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

- Supongamos que tenemos un repositorio con dos ficheros `otro.txt` y `texto.txt`.
- Creamos un repo git y hacemos *commit* de ambos.

```
$ 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

- Editamos ambos ficheros: las líneas en en verde son nuevas; las líneas en en rojo se borran.

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

- Estos son los nuevos ficheros en el área de trabajo:

```
$ 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

- Ejecutamos `git diff`. Compara directorio de trabajo y *staging area*.
- El fichero resultante es un *patch*, un fichero con las diferencias

```
$ git diff
diff --git a/otro.txt b/otro.txt
index 866b380..bc52249 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 8093e19..2f4d802 100644
--- a/texto.txt
+++ b/texto.txt
@@ -1,5 +1,5 @@
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

```
$ git diff
diff --git a/otro.txt b/otro.txt
index 866b380..bc52249 100644
```

- El flag del comando `diff –git` *no existe*. Git indica de esa forma que se ejecuta `git diff`.
- 866*b*380 y *bc*52249 son hashes SHA de las versiones (*a* y *b*) del fichero a comparar (en este ejemplo, `otro.txt`).
- 100644 Son los permisos de los ficheros (tipo de fichero, rwx).

diff y patch

- Cabecera del formato unificado de `diff`, seguido de un fragmento de texto (*hunk*) que indica los cambios. En el *hunk*:
  - Cada línea que empieza por $-$ indica contenido en *a* no presente en *b*.
  - Cada línea que empieza por $+$ indica contenido que falta en *a* presente en *b*.
- `@@ -1,4 +1,6 @@` es el tamaño y longitud del texto en *a* (singo $-$) y *b* (signo $+$).
- Si sólo hay un número es la primera línea, si no, la primera línea y el número total de líneas abarcan los cambios de ese fragmento.

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

- En el ejemplo, los cambios en *a* empiezan en línea 1 y abarcan 4 líneas, los de *b* empiezan en línea 1 y abarcan 6 líneas.

diff y patch

- El fichero entero que genera `git diff` es un patch.
- Se podría aplicar sobre la versión antigua (en el ejemplo la de *staging area*) 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).

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` o `main`.

```
 $ 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
```

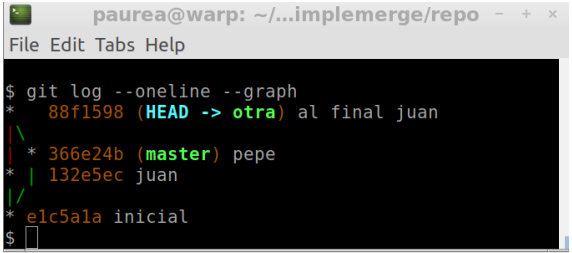
Conflictos con dos ramas

- Si no quiero hacerlo finalmente, en lugar del *commit*, puedo abortar el *merge*.

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

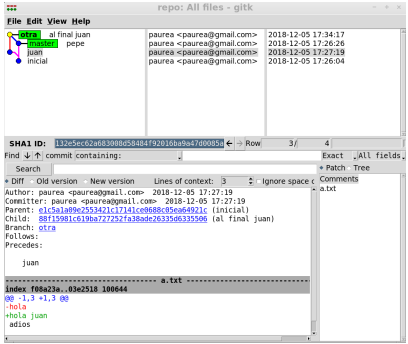
Conflictos con dos ramas

- 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, pero mejor acostumbrarse a las herramientas de línea de comando.



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
```

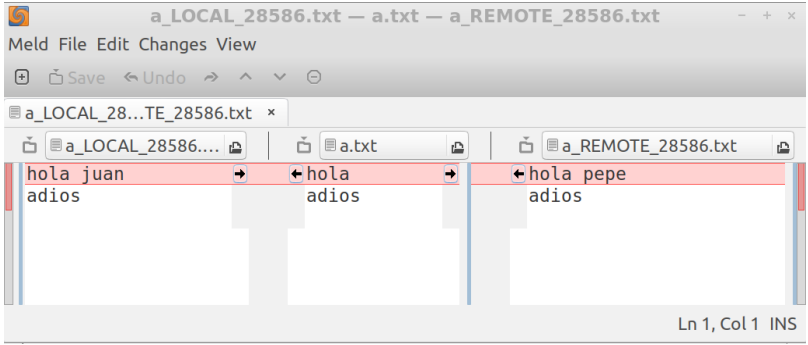
mergetool

- En el ejemplo anterior:

```
$ 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 directamente.



Resolución de conflictos

- Cuando estoy contento, cierro y salvo los cambios.

```
 $ 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'
```

Para saber más

- La referencia fundamental sobre Git accesible de forma pública es el libro Pro Git, disponible en inglés y castellano [1].
- Los libros de la serie *Head First* de O'Reilly son muy conocidos por su nivel muy accesible y su enfoque didáctico. Para este tema, se ha publicado en enero de 2022 *Head First Git* [2].
- Otra referencia muy conocida de la misma editorial es [3].

Referencias I

[1] S. Chacon y B. Straub. *Pro Git*. The Expert's Voice. Apress, 2014.

[2] Raju Gandhi. *Head First Git. A Learner's Guide to Understand Git from the Inside Out*. Head First. O'Reilly Media, 2022.

[3] P.K. Ponuthorai y J. Loeliger. *Version Control with Git*. 3ª ed. O'Reilly Media, 2022.