Introducción al Control de Versiones con Git/GitHub usando SmartGit

Luis Nieto

Departamento de Electricidad, Electrónica y Computación
Facultad de Ciencias Exactas y Tecnología
Universidad Nacional de Tucumán

Introducción [1 | 4]

- Introducción
  - Presentación
  - Borrado de archivos
  - Movimiento/renombrado de archivos
  - Ignorar archivos

Introducción [2 | 4]

- Vuelta atrás
  - Sobre la carpeta de trabajo
  - Del último *commit*
  - A un estado específico

Introducción [3 | 4]

- Ramas
  - Trabajo con ramas
  - Combinación de ramas
  - Resolución de conflictos
  - Recomendaciones

Introducción [4 | 4]

- Repositorios remotos
  - Repositorios locales y remotos
  - Trabajo individual en GitHub
  - Trabajo colaborativo en GitHub

Presentación [1 | 21]

- **Git**
  - Sistema distribuido de control de versiones, el cual permite llevar el control de los cambios que se hacen a archivos y carpetas, permitiendo navegar por todo el historial de cambios
- **SmartGit**
  - Cliente Git gráfico

Presentación [2 | 21]

- **Definiciones**
  - **Repositorio**: se utiliza para organizar un único proyecto, pudiendo contener carpetas y archivos
  - ***Check in / Check out***: cuando alguien quiere hacer una modificación, toma una copia del trabajo del repositorio (*check out*), hace las modificaciones y reingresa la copia al repositorio (*check in*)

Presentación [3 | 21]

- **Definiciones**
  - ***Commit***: en Git los cambios guardados se llaman *commits*. Cada *commit* tiene un mensaje asociado que explica por qué se hizo determinado cambio. Los distintos *commits* forman el historial de cambios, y por lo tanto, otras personas pueden comprender lo que se viene haciendo y porqué

Presentación [4 | 21]

- Algunos videos:
  - https://www.youtube.com/watch?v=9GKpbI1siow
  - https://www.youtube.com/watch?v=H7hNgU39mlA
  - https://www.youtube.com/watch?v=SMWItqi3Y-0

Presentación [5 | 21]

- **Instalación**
  - Git: http://git-scm.com
  - SmartGit: https://www.syntevo.com/smartgit/
  - Hay enlaces para distintas plataformas

Presentación [6 | 21]

- **Inicialización de un repositorio**
  - Para llevarle el control de versiones a los archivos de una carpeta, lo primero que se debe hacer es convertirla en un repositorio Git
  - Comando Git: `git init`
  - Desde SmartGit, seleccionar **Repository > Add or Create...** y elegir la carpeta

- Si se trabajara desde la línea de comandos, dentro de la carpeta del proyecto habría que ejecutar el comando `git init`.
- Al inicializar un repositorio Git hace un primer *commit* (“*commit* inicial”).

Presentación [7 | 21]

- **Inicialización de un repositorio (continuación)**
  - Una vez que una carpeta se convierte en un repositorio Git, todos los archivos que se agreguen, modifiquen, saquen, etc. serán controlados por Git, sin importar cuán adentro de la carpeta se encuentren
  - Para saber qué archivos están en qué estado se ejecuta el comando Git `git status`

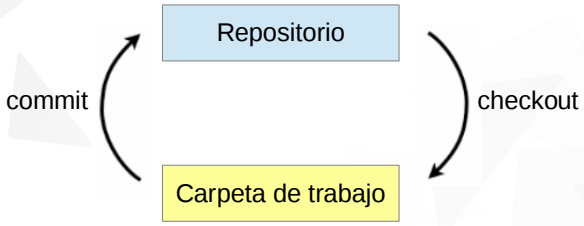
Presentación [8 | 21]

- **Inicialización de un repositorio (continuación)**
  - Al inicializar un repositorio, Git crea dentro de la carpeta otra carpeta, oculta, llamada `.git`, que es donde guarda todo lo necesario para llevar el control
  - La carpeta `.git` es el repositorio Git. Si se quisiera dejar de controlar el proyecto, se la borra

- Como el procedimiento de inicialización de un repositorio se ejecuta en cada carpera a la que se quiera llevar el control, en cada carpeta habrá una llamada `.git`.

Presentación [9 | 21]

- **Arquitectura de Git**
  - Algunos sistemas de control de versiones (no es el caso de Git) usan una arquitectura de 2 árboles



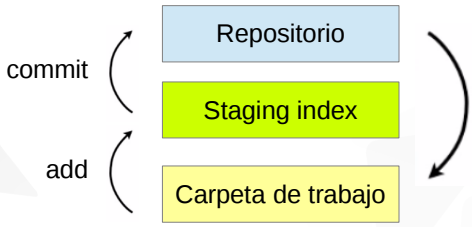
- Se los llama árboles porque representan una estructura de archivos: la carpeta de trabajo comienza en la raíz de la carpeta del proyecto, y por abajo pueden haber unas cuantas carpetas con archivos y otras carpetas adentro, las cuales a su vez pueden tener archivos y más carpetas, y así sucesivamente.

- El repositorio también contiene un conjunto de archivos.

- Cuando se quieren mover archivos desde el repositorio, se hace un *checkout*, y cuando se terminan de hacer las modificaciones se las registran (*commit*) en el repositorio (el repositorio y la carpeta de trabajo no necesariamente tienen la misma información).

Presentación [10 | 21]

- **Arquitectura de Git (continuación)**
  - Git usa una arquitectura de 3 árboles



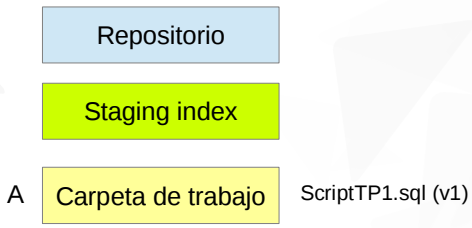
- Esta arquitectura permite que cuando se modifiquen, por ejemplo, 10 archivos en la carpeta de trabajo, no se esté obligado a pasar los 10 al repositorio, sino 5 por ejemplo: sólo a estos 5 se los pone en el *staging index*, y luego se los pasa al repositorio como un único conjunto.

- De la misma forma, se pueden sacar cosas del repositorio: se las pasa al *staging index*, y desde ahí a la carpeta de trabajo (en realidad se las pasa directamente a la carpeta de trabajo, actualizando también el *staging index)*.

- Al *staging index* también se lo conoce como *index* directamente.

Presentación [11 | 21]

- **Arquitectura de Git (continuación)**
  - Ejemplo: se agrega el archivo `ScriptTP1.sql` a la carpeta de trabajo. `ScriptTP1.sql` estará inicialmente en su primera versión (`v1`). Por simplicidad, a estos cambios se los llamará `A`:



- Observar que cuando se agrega un archivo a una carpeta que se le está llevando el control, SmartGit lo muestra con el estado "*Untracked*". Para ver la misma información desde la línea de comandos: `git status` (`ScriptTP1.sql` aparece como archivo "sin seguimiento").

- Al seleccionar el archivo, en la parte media de la pantalla se muestran las diferencias entre los 3 árboles (hay diferencias entre la carpeta de trabajo y el *index*, pero que no entre el *index* y el repositorio).

- Los cambios en la carpeta de trabajo están a la derecha, los del *index* en el centro y los del repositorio a la izquierda.

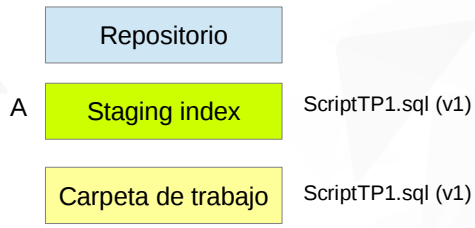
Presentación [12 | 21]

- **Arquitectura de Git (continuación)**
  - Comando Git: `git add <archivo>`
  - Desde SmartGit, para agregar archivos al *index*, seleccionar **Local > Stage** y luego se eligen los archivos (o clic derecho en el archivo y elegir **Stage**)
  - Si en lugar de seleccionar un archivo en particular se selecciona una carpeta, se agregan todos sus archivos al *index*

- Si se trabajara desde la línea de comandos, dentro de la carpeta habría que ejecutar el comando `git add ScriptTP1.sql`. Para agregar todos los archivos se especifica un punto (`.`).

Presentación [13 | 21]

- **Arquitectura de Git (continuación)**
  - Al agregar `ScriptTP1.sql` al *index*, en el mismo también existirá `ScriptTP1.sql`:

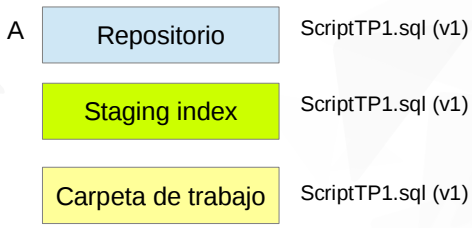


- Ahora no hay diferencias entre la carpeta de trabajo y el *index*, pero sí entre el *index* y el repositorio.

- Para ver el estado de los archivos: `git status`.

Presentación [14 | 21]

- **Arquitectura de Git (continuación)**
  - Luego se envía `ScriptTP1.sql` al repositorio
  - Comando Git: `git commit -m "<mensaje>"`
  - En SmartGit se lo selecciona y se elige **Local > Commit...**:



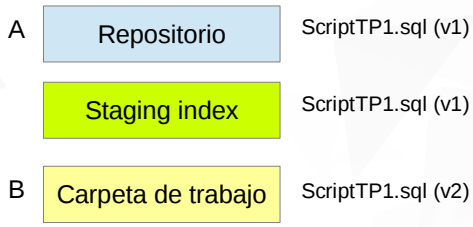
- Si se trabajara desde la línea de comandos, dentro de la carpeta del proyecto habría que ejecutar el comando: `git commit -m "<mensaje descriptivo>"`. Ejemplo: `git commit -m "ScriptTP1.sql agregado al proyecto"`.

- Al especificar un mensaje se debe describir lo que se está haciendo. Con los mensajes se etiqueta lo que se va haciendo, para que luego se puedan revisar los cambios y mediante los mismos saber qué contienen.

- Observar que al agregar el archivo al repositorio, no hay diferencias entre los 3 árboles.

Presentación [15 | 21]

- **Arquitectura de Git (continuación)**
  - Ahora se hace un cambio a `ScriptTP1.sql` en la carpeta de trabajo. Al hacer esto, se tendrá la versión 2 de `ScriptTP1.sql` (por simplicidad, a la diferencia entre estas 2 versiones se la llamará `B`):

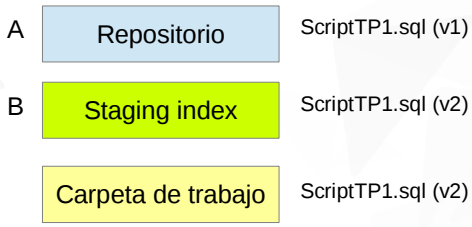


- Los nombres `A`, `B`, etc no son la forma en la que Git referencia los cambios. Primero, `A` y `B` no hacen referencia a un único archivo, sino a un conjunto de cambios. Por esta razón, en un flujo de trabajo típico, `A` representaría los cambios a 5 archivos, `B` los cambios a 3 archivos, etc. Por esta razón, `A` y `B` son instantáneas de los cambios que se hicieron, y no son archivos o versiones de archivos.

- Desde la línea de comandos, git status muestra el estado del archivo como modificado y SmartGit como "*Modified*", y muestra diferencias entre la carpeta de trabajo y el *index*, pero no entre el *index* y el repositorio.

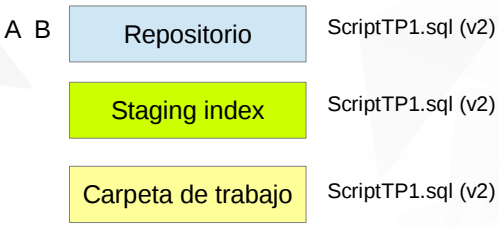
Presentación [16 | 21]

- **Arquitectura de Git (continuación)**
  - Luego se agrega `ScriptTP1.sql` al *index*:



Presentación [17 | 21]

- **Arquitectura de Git (continuación)**
  - Luego se envían los cambios al repositorio:

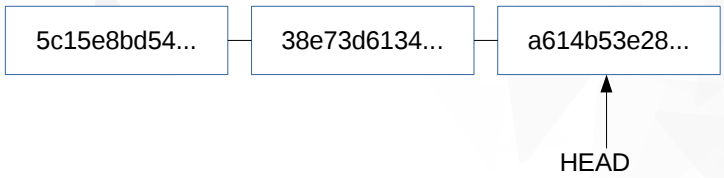


Presentación [18 | 21]

- **Arquitectura de Git (continuación)**
  - Si se ejecuta esta secuencia de pasos una vez más, se tendrá una versión 3 de `ScriptTP1.sql`, representado por `C`
  - Estos cambios se introducirán en el *index* y luego en el repositorio
  - Este es el flujo de trabajo que sigue Git

Presentación [19 | 21]

- **Arquitectura de Git (continuación)**
  - Git mantiene un puntero (sólo para el repositorio) llamado HEAD, el cual siempre apunta al último *commit* de la rama actual. A medida que se hacen nuevos *commits*, cambia el valor del puntero.



- Cada vez que Git hace un *commit* genera un identificador único para el mismo. Este identificador es un *checksum* generado mediante un algoritmo de *hash* llamado SHA-1. El número que genera este algoritmo siempre tiene 40 caracteres hexadecimales.

Presentación [20 | 21]

- **Arquitectura de Git (continuación)**
  - Comando Git para ver el historial de *commits*: `git log`
  - En SmartGit, las ventanas:
    - **Journal** muestra todo el historial de cambios (resumido)
    - **Changes** muestra las diferencias entre los 3 árboles
    - **Log** muestra todo el historial de cambios (detallado)

- Si se trabajara desde la línea de comandos, dentro de la carpeta del proyecto habría que ejecutar el comando: `git log`.
- Debido al funcionamiento del puntero HEAD, ejecutar `git log` es lo mismo que `git log HEAD` (comenzar desde el HEAD e ir hacia atrás).
- El historial muestra los *commits* ordenados descendentemente, desde el más reciente hasta el primero.
- Para ver las diferencias entre los 3 árboles se ejecuta el comando `git status`.

Presentación [21 | 21]

- **Arquitectura de Git (continuación)**
  - Trabajar con Git implica seguir esta secuencia:
    - Realizar los cambios
    - Agregar los cambios al *index*
    - Registrar los cambios en el repositorio

- En SmartGit, cuando se modifica un archivo (al cual se le viene haciendo el seguimiento) en la carpeta de trabajo, al mismo se lo puede agregar al repositorio directamente haciéndole un *commit* (SmartGit primero lo agrega al *index* y luego al repositorio).

- Para hacer lo mismo desde la línea de comandos se ejecuta: `git commit -am "<mensaje>"` (esto funciona cuando al archivo se le viene haciendo el seguimiento, no cuando el archivo recién se agrega a la carpeta de trabajo).

Borrado de archivos [1 | 2]

- Si los archivos a borrar sólo están en la carpeta de trabajo:
  - Se borran simplemente
- Si los archivos a borrar están en el repositorio (desde la línea de comandos, primera opción):
  - Se borran
  - Se ejecuta el comando `git rm <archivo>`
  - Se hace un *commit* informando del borrado

- Al borrar un archivo que estaba en el repositorio, si se ejecuta el comando `git status`, el mismo informa que el repositorio tiene un archivo `<archivo>` y que el mismo ya no está en la carpeta de trabajo, que fue borrado. También informa que cuando se le haga un cambio a un archivo al cual ya se le estaba llevando el control, se debe ejecutar `git add <archivo>`, y cuando se quiera borrar un archivo se debe ejecutar `git rm <archivo>`. En conclusión, ahora se deberá ejecutar `git rm <archivo>` y luego hacer un *commit* para registrar los cambios.

- Con esta opción se borra el archivo con algún comando/herramienta del sistema operativo, luego se pone el cambio en el *index* y finalmente se hace el *commit*.

Borrado de archivos [2 | 2]

- Si los archivos a borrar están en el repositorio (desde la línea de comandos, segunda opción):
  - Se ejecuta el comando `git rm <archivo>`
  - Se hace un *commit* informando del borrado
- Si los archivos a borrar están en el repositorio (desde SmartGit):
  - Se borran
  - Seleccionar **Local > Commit...**

- Con esta opción en un único paso se borra el archivo y se pone el cambio en el *index*, y luego se hace el *commit*.

- Al borrar un archivo que estaba en el repositorio, en SmartGit el mismo aparece con el estado "*Missing*". Se puede seleccionar **Local > Commit...** o bien hacer clic derecho en el archivo y luego **Commit..**.

Movimiento/renombrado de archivos [1 | 2]

- Si los archivos a mover/renombrar sólo están en la carpeta de trabajo:
  - Se mueven/renombran simplemente
- Si los archivos a mover/renombrar están en el repositorio (desde la línea de comandos, primera opción):
  - Se mueven/renombran
  - Se ejecutan `git add <archivo_nuevo>` y `git rm <archivo_viejo>`
  - Se hace un *commit* informando del movimiento/renombrado

- Al renombrar un archivo que estaba en el repositorio, si se ejecuta el comando `git status`, el mismo informa que se borró el archivo `<archivo_viejo>`, y que se agregó un nuevo archivo llamado `<archivo_nuevo>`, al cual todavía no le está llevando el control. Si se agrega `<archivo_nuevo>` al *index* y se borra <archivo_viejo>, Git ahora se da cuenta que se renombraron los archivos. Luego, estos cambios se pueden registrar definitavamente haciendo un *commit*.

- Con esta opción se mueve/renombra el archivo con algún comando/herramienta del sistema operativo, luego se pone el cambio en el *index* y finalmente se hace el *commit*.

Movimiento/renombrado de archivos [2 | 2]

- Si los archivos a mover/renombrar están en el repositorio (desde la línea de comandos, segunda opción):
  - Se ejecuta `git mv <archivo_viejo> <archivo_nuevo>`
  - Se hace un *commit* informando del movimiento/renombrado
- Si los archivos a mover/renombrar están en el repositorio (desde SmartGit):
  - Se mueven/renombran
  - Seleccionar **Local > Commit...**

- Con esta opción en un único paso se mueve/renombra el archivo y se pone el cambio en el *index*, y luego se hace el *commit*.
- Al renombrar un archivo que estaba en el repositorio, en SmartGit el mismo aparece con el estado "*Rename (untracked)*". Se puede seleccionar **Local > Commit...** o bien hacer clic derecho en el archivo y luego **Commit...**.

Ignorar archivos [1 | 7]

- Hay archivos a los cuales no interesa llevarles un seguimiento:
  - Código compilado
  - Archivos comprimidos
  - Logs
  - Archivos generados por el sistema operativo
  - Archivos que suelen subir los usuarios (imágenes, pdfs, videos, etc)

Ignorar archivos [2 | 7]

- La forma de decirle a Git que ignore ciertos archivos es mediante un archivo llamado `.gitignore` (dentro de la carpeta del proyecto)
- El archivo `.gitignore` tiene las reglas que sigue Git para saber a cuáles archivos les lleva el control, y a cuáles no

Ignorar archivos [3 | 7]

- Sobre las reglas del archivo `.gitignore`:
  - Pueden consistir en una lista de archivos (un archivo por línea)
  - Se pueden usar:
    - `*`: uno o más caracteres (`*.php` ignora todos los archivos terminados en `php`)
    - `?`: un único caracter
    - `[aeiou]`: conjunto de caracteres
    - `[0-9]`: rango

Ignorar archivos [4 | 7]

- Sobre las reglas del archivo `.gitignore` (continuación):
  - Se pueden negar expresiones empleando un `!` (`!index.php` no ignora el archivo `index.php`)
  - Para especificar todo un directorio, se agrega una barra `/` al final (`videos/` implica que se ignoren todos los archivos en el directorio `videos`)
  - Para agregar comentarios se agrega un `#` al comienzo de la línea, y las líneas en blanco se ignoran

Ignorar archivos [5 | 7]

- Al archivo `.gitignore` hay que llevarle el seguimiento (agregarlo al *index* y luego al respositorio)
- Links con recomendaciones sobre los archivos a ignorar:
  - https://help.github.com/articles/ignoring-files (cosas en general)
  - https://github.com/github/gitignore (cosas en particular según el lenguaje con el que se trabaje)

Ignorar archivos [6 | 7]

- Sobre las carpetas:
  - Git ignora toda carpeta vacía (Git le lleva el control a carpetas con archivos, e ignora cualquier carpeta vacía)
  - Si se quiere llevar el control de una carpeta vacía se suele crear dentro de la misma un archivo (oculto para que no moleste) vacío (llamado generalmente `.gitkeep`)

Ignorar archivos [7 | 7]

- En SmartGit, para ignorar un archivo se lo selecciona y se elige **Local > Ignore...** (o clic derecho en el archivo y seleccionar **Ignore...**)
- También se puede crear un archivo `.gitignore` dentro de la carpeta del proyecto y editarlo con la configuración deseada

- La primera vez que se edita el archivo `.gitignore`, el mismo aparece en la carpeta de trabajo con el estado "*Untracked*", por lo que se lo debe agregar al repositorio.
- Cuando se ignora un archivo, y al mismo se le hacen cambios, SmartGit no lo muestra.

Sobre la carpeta de trabajo [1 | 5]

- **Ejemplo #1:** se modifica/borra un archivo que ya está en el repositorio (sin agregar al *index*)
  - Entre la carpeta de trabajo y el *index*: **hay diferencias**
  - Entre el *index* y el repositorio: **no hay diferencias**
  - Entre la carpeta de trabajo y el repositorio: **hay diferencias**

  - Diferencias entre el *index* y la carpeta de trabajo: `git diff`
  - Diferencias entre el repositorio y el *index*: `git diff --staged`
  - Diferencias entre el repositorio y la carpeta de trabajo: `git diff HEAD`

- En todos los casos, los archivos que están en el árbol superior se marcan con un `---`, y los que están en el árbol inferior con un `+++`. Por ejemplo, en el caso de `git diff`, los archivos que están en el index se marcan con un `---`, y los que están en la carpeta de trabajo con un `+++`.

Sobre la carpeta de trabajo [2 | 5]

- **Ejemplo #1 (continuación):**
  - Desde la línea de comandos, para deshacer los cambios sobre el archivo en la carpeta de trabajo, reemplazándolos por la versión en el repositorio:

```
git checkout -- <archivo>
```

  - Desde SmartGit, elegir **Local > Discard** (sólo está disponible la opción para volver al estado del repositorio (*HEAD*))

- El comando `git checkout` reemplaza los cambios en la carpeta de trabajo con el último contenido del repositorio. Si no se especifica el doble guión asume que es una rama. No modifica el *index*. También, si en la carpeta de trabajo se hubieran agregado archivos nuevos, los mismos se mantendrían sin cambio.

- En SmartGit, también se puede hacer clic derecho en el archivo y seleccionar **Discard**.

Sobre la carpeta de trabajo [3 | 5]

- **Ejemplo #2:** se modifica/borra un archivo que ya está en el repositorio (agregando por error al *index*)
  - Entre la carpeta de trabajo y el *index*: **no hay diferencias**
  - Entre el *index* y el repositorio: **hay diferencias**
  - Entre la carpeta de trabajo y el repositorio: **hay diferencias**
  - Para deshacer los cambios en el *index*, reemplazándolos por la versión en el repositorio (volviendo al estado del ejemplo anterior):

```
git reset HEAD <archivo>
```


- El comando `git reset HEAD <archivo>` modifica el *index* para que coincida con el contenido del repositorio (también se podría haber ejecutado `git reset HEAD -- <archivo>`).
- Luego de ejecutar este comando se vuelve al estado del ejemplo anterior, por lo que para deshacer los cambios en la carpeta de trabajo, habrá que ejecutar el comando `git checkout`.
- Si en lugar de ser un único archivo el que se modificó y agregó al *index*, hubieran sido varios, para no especificar uno por uno, se ejecuta `git reset HEAD`.

Sobre la carpeta de trabajo [4 | 5]

- **Ejemplo #2 (continuación):**
  - Desde SmartGit, elegir **Local > Unstage** (se vuelve al estado del ejemplo anterior, por lo que luego habrá que hacer **Local > Discard**)

- Si se elige la opción por volver al estado del *index*, seleccionando **Discard** nuevamente se puede volver al estado del repositorio.

Sobre la carpeta de trabajo [5 | 5]

- **Ejemplo #3 (SmartGit):**
  - Desde SmartGit también se puede modificar la versión que hay en el *index* (seleccionando **Local > Index Editor**)
  - Al modificar la versión en el *index* y luego elegir **Local > Discard**, habrá 2 opciones para volver atrás: al estado del repositorio (*HEAD*) o al del *index*

Del último *commit* [1 | 3]

- **Situación:**
  - Hay un determinado archivo en el repositorio
  - Después de hacer el último *commit* uno se da cuenta que se debería haber hecho un cambio más
  - Opciones:
    - Hacer el cambio y hacer otro *commit*
    - Modificar el último *commit*

Del último *commit* [2 | 3]

- **Situación (continuación):**
  - Debido a cuestiones de integridad de Git y al algoritmo de *hash*, el único *commit* que permite una modificación es el último
  - Se hace el cambio que faltaba y se ejecuta:

```
git commit --amend -m "<mensaje>"
```

Del último *commit* [3 | 3]

- **Situación (continuación):**
  - Desde SmartGit, luego de hacer el cambio que faltaba, se hace el *commit* nuevamente y se selecciona la opción "**Amend last commit instead of creating new one**" (el último *commit* queda con este último mensaje)

- Esta opción ejecuta internamente el comando: `git commit -m <mensaje> --amend <archivo>.`

A un estado específico [1 | 6]

- El comando `git reset` permite especificar a dónde apuntará el puntero *HEAD*, y por lo tanto, permite deshacer múltiples *commits*
- Se pueden especificar 3 opciones:
  - *Soft*
  - *Hard*
  - *Mixed*

- Debido a las características de esta opción, se debe emplear con mucho cuidado.

A un estado específico [2 | 6]

- ***Soft***: mueve el puntero *HEAD* al *commit* especificado sin modificar ni el *index* ni la carpeta de trabajo (opción más segura)
- ***Hard***: mueve el puntero *HEAD* al *commit* especificado, modificando el *index* y la carpeta de trabajo para que coincidan con el repositorio
- ***Mixed*** (predeterminada): mueve el puntero *HEAD* al *commit* especificado modificando el *index* para que coincida con el repositorio

A un estado específico [3 | 6]

- **Situación**:
  - Hay un determinado archivo en el repositorio al cual se le vienen haciendo una serie de modificaciones con sus correspondientes *commits*
  - No hay diferencias entre los 3 árboles
  - Se quiere volver a un determinado estado (un determinado *commit*)

A un estado específico [4 | 6]

- **Empleando la opción *Soft***:
  - Se ejecuta: `git reset --soft <id commit>`
  - En SmartGit se elige **Local > Reset...**, se selecciona el *commit* al cual se quiere ir y se selecciona la opción *soft*
  - Como no se modifica ni el *index* ni la carpeta de trabajo, cualquier cambio realizado después del *commit* a donde se mueve el puntero *HEAD* continúa en la carpeta de trabajo (no se lo pierde)

- Observar que ahora habrá diferencias entre el repositorio y el *index*, y entre el repositorio y la carpeta de trabajo (no habrá diferencias entre el *index* y la carpeta de trabajo).

- Esta opción ejecuta internamente el comando: `git reset --soft <ID commit>`.

A un estado específico [5 | 6]

- **Empleando la opción *Mixed***:
  - Se ejecuta: `git reset --mixed <id commit>`
  - En SmartGit se elige **Local > Reset...**, se selecciona el *commit* al cual se quiere ir y se selecciona la opción *mixed*
  - Como no se modifica la carpeta de trabajo, al igual que en el caso anterior, no se pierden los cambios realizados después del *commit* a donde se mueve el puntero *HEAD*

- Observar que ahora habrá diferencias entre el repositorio y la carpeta de trabajo, y entre el *index* y la carpeta de trabajo (no habrá diferencias entre el repositorio y el *index*).

- Esta opción ejecuta internamente el comando: `git reset --mixed <ID commit>`.

A un estado específico [6 | 6]

- **Empleando la opción *Hard***:
  - Se ejecuta: `git reset --hard <id commit>`
  - En SmartGit se elige **Local > Reset...**, se selecciona el *commit* al cual se quiere ir y se selecciona la opción *hard*
  - Como se modifican el *index* y la carpeta de trabajo, cualquier cambio que se haya hecho después del *commit* a donde se mueve el puntero *HEAD* se perderá (no estará ni en el repositorio, ni en el *index* ni en la carpeta de trabajo)

- Observar que ahora no habrá diferencias entre los 3 árboles ya que esta opción al mover el puntero *HEAD* también actualiza el *index* y la carpeta de trabajo.

- Esta opción ejecuta internamente el comando: `git reset --hard <ID commit>`.

Trabajo con ramas [1 | 11]

- **Situación #1**:
  - Se está trabajando en un proyecto y surge una idea sobre la cual no se está seguro que vaya a funcionar
  - En lugar de hacer varios *commits* y luego descubrir que la idea no funciona, y en consecuencia se deban deshacer los mismos, se crea una nueva rama y se trabaja en la misma
  - Si la idea no funciona, se descarta la rama, y si la idea funciona, los cambios en la rama se integran con el proyecto mediante un proceso llamado combinación (*merge*)

Trabajo con ramas [2 | 11]

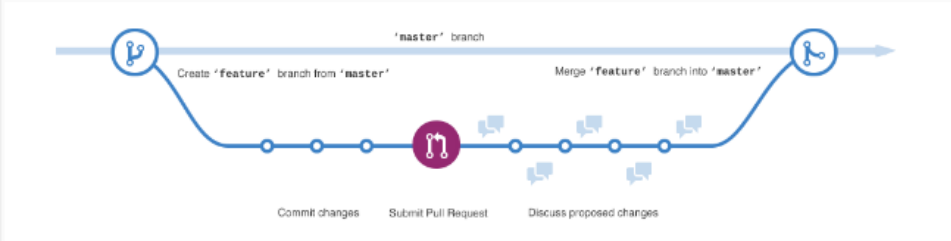
- **Situación #2**:
  - Se está trabajando en un proyecto y se quiere modificar una parte del mismo
  - Se puede crear una rama, trabajar sobre la parte que se quiere modificar, colaborar con otros, etc
  - Mientras tanto, otra gente puede seguir trabajando en el proyecto
  - Cuando se acuerden los cambios realizados en la rama, se la integra con el proyecto

Trabajo con ramas [3 | 11]

- De forma predeterminada, todo proyecto en Git tiene una rama principal llamada `master`, la cual no se puede borrar ni renombrar
- Para ver las ramas de un proyecto se ejecuta el comando `git branch`
- En SmartGit, para ver las ramas de un proyecto se elige **Window > Branches**

Trabajo con ramas [4 | 11]

- En la figura se puede ver:
  - La rama `master`
  - Una nueva rama llamada `feature`
  - La actividad en la rama `feature` antes de volver a combinarse con la rama `master`



Trabajo con ramas [5 | 11]

- **Creación de ramas:**
  - Cuando se crea una rama, se lo hace a partir de otra (rama origen)
  - Comando: `git branch <rama_nueva> <rama_origen>`
  - Para cambiar de rama, se ejecuta `git checkout <rama>`
  - Creación y cambio de rama con un único comando:
    `git checkout -b <rama_nueva> <rama_origen>`

- Cuando se crea una rama, de forma predeterminada se lo hace a partir del último *commit* de la rama origen. Si se quiere crear una rama a partir de un *commit* en particular, se debe especificar el mismo:

```
git checkout -b <rama> <id commit>
```

- Si no se especifica `rama_origen`, `rama_nueva` se crea a partir de la cual se esté parado actualmente.

Trabajo con ramas [6 | 11]

- **Creación de ramas:**
  - En SmartGit, para crear una rama:
    - Se elige **Branch > Add Branch...**
    - Se le da un nombre a la rama
    - Si se selecciona "**Add Branch & Checkout**" se cambia a la misma luego de crearse la rama (si no luego habrá que hacer el cambio manualmente)

- Si se quiere crear una rama a partir de un *commit* en particular, se hace clic derecho sobre el *commit* y se selecciona *Add Branch*.

- En caso de seleccionarse la opción "**Add Branch & Checkout**", también se ejecuta el comando: `git checkout <rama_nueva>`.

- Si se quiere crear una rama a partir de un determinado *commit*, en la ventana **Journal** se puede seleccionar el *commit*, luego hacer clic derecho y elegir "*Add Branch...*".

Trabajo con ramas [7 | 11]

- **Cambio de rama:**
  - Se elige **Branch > Check Out...**
  - Se selecciona la rama a la cual se quiere cambiar
  - Si se selecciona “**Create local branch**”, al cambiar a la rama destino se crea una copia de la misma y se cambia a la copia

Trabajo con ramas [8 | 11]

- **Sobre la creación y cambio de rama:**
  - Al crearse una rama, se sigue teniendo una única carpeta de trabajo (todos los archivos del proyecto siguen estando en la carpeta del mismo)
  - Al cambiar de rama Git hace, rápidamente, un cambio de contexto: toma todos los archivos de la carpeta de trabajo y hace que se correspondan con lo que hay en la rama

- Ejemplo: se está trabajando en la rama `master` y se cambia a la rama `prueba`: la carpeta de trabajo tendrá todos los cambios relacionados con la rama `prueba`.

- Luego se vuelve a la rama `master`: todos estos cambios desaparecen y se tienen los correspondientes a los de esta rama.

Trabajo con ramas [9 | 11]

- **Sobre la creación y cambio de rama (continuación):**
  - Si se hacen cambios en una rama, y sin hacer el *commit* se cambia a otra rama, en la carpeta de trabajo aparecen estos cambios
  - Como no se hizo el *commit*, desde cualquier rama se pueden deshacer estos cambios

Trabajo con ramas [10 | 11]

- **Renombrado de ramas:**
  - Comando: `git branch -m <rama_vieja> <rama_nueva>`
  - En SmartGit se selecciona la rama y se elige **Branch > Rename...**

- Para el comando Git también se puede especificar `--move` en lugar de `-m`.
- Al renombrar una rama se puede estar parado en la misma.

Trabajo con ramas [11 | 11]

- **Borrado de ramas:**
  - Comando: `git branch -d <rama>`
  - En SmartGit se selecciona la rama y se elige **Branch > Delete...**
  - Para borrar una rama no se debe estar parado en la misma

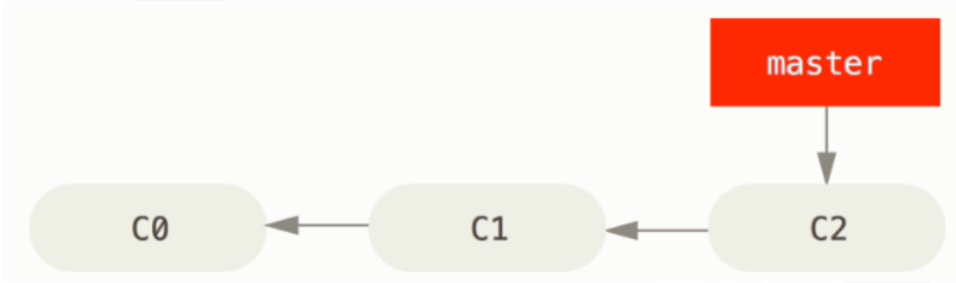
- Para el comando Git también se puede especificar `--delete` en lugar de `-d`.

Combinación de ramas [1 | 18]

- Una vez listos los cambios sobre los que se viene trabajando en una rama, se la debe integrar, por ejemplo, con la rama desde la cual se la creó
- Esto se hace mediante un procedimiento llamado **Combinación** (*merge*)

Combinación de ramas [2 | 18]

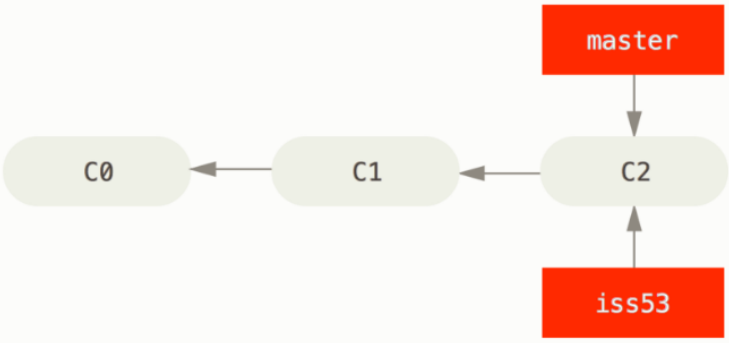
- **Situación:**
  - Se está trabajando en un proyecto sobre el cual se hicieron una serie de modificaciones con sus correspondientes *commits*



- En la figura, se hicieron 3 *commits*: primero `C0`, luego `C1` y el último es `C2`. Actualmente se está parado en la rama `master`.

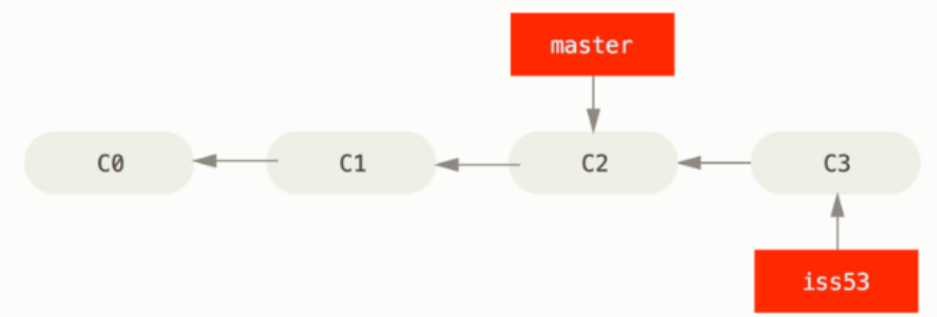
Combinación de ramas [3 | 18]

- **Situación (continuación):**
  - Se debe solucionar un determinado problema, para lo cual se crea (y se cambia) una rama llamada `iss53`



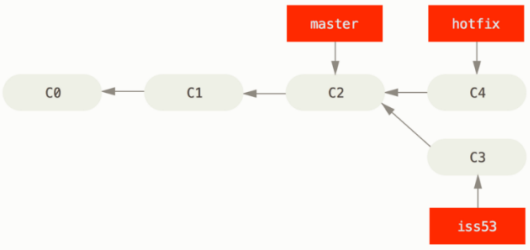
Combinación de ramas [4 | 18]

- **Situación (continuación):**
  - En la rama `iss53` se realiza, y confirma, un cambio



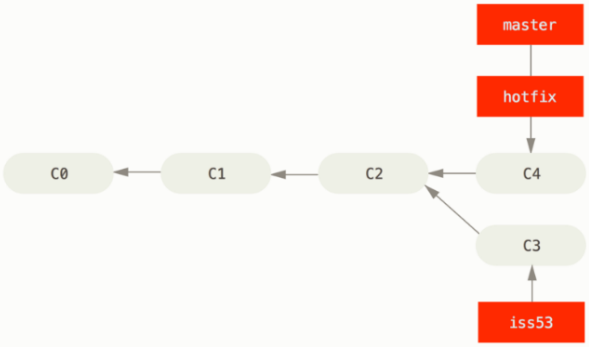
Combinación de ramas [5 | 18]

- **Situación (continuación):**
  - Se debe solucionar otro problema: se crea (a partir de la rama `master`) otra rama llamada `hotfix`
  - Se trabaja sobre `hotfix` haciendo las modificaciones (y *commits*) correspondientes



Combinación de ramas [6 | 18]

- **Situación (continuación):**
  - Una vez probadas las modificaciones en `hotfix` se las integra con la rama `master` nuevamente



Combinación de ramas [7 | 18]

- **Situación (continuación):**
  - Para combinar la rama `hotfix` con la rama `master`:
    - Pararse en la rama destino (`master`)

      ```
      git checkout master
      ```

    - Hacer la combinación

      ```
      git merge hotfix
      ```


- Una vez integrada la rama `hotfix` con la rama `master`, se puede borrar la primera.

Combinación de ramas [8 | 18]

- **Situación (continuación):**
  - Debido a que el *commit* `C4` está directamente a continuación del *commit* `C2`, Git simplemente mueve el puntero hacia adelante (*fast-forward*), es decir, no crea un *commit* de combinación (sólo mueve el puntero HEAD)
  - Este comportamiento se corresponde con la opción `--ff` (*fast-forward if possible*) del comando `git merge` (también se puede especificar otro modo, el cual se muestra más adelante)

- En este caso, como Git puede hacer una combinación del tipo *fast-forward*, ejecutar el comando `git merge hotfix` es lo mismo que ejecutar `git merge --ff hotfix`.
- La opción `--ff` es la predeterminada del comando `git merge`.

Combinación de ramas [9 | 18]

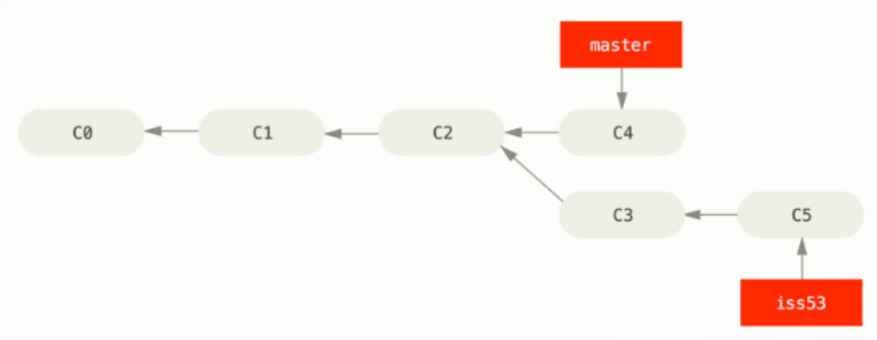
- **Situación (continuación):**
  - En SmartGit para combinar la rama `hotfix` con la rama `master`:
    - Pararse en la rama destino (`master`)
    - Elegir **Branch > Merge...**
    - Seleccionar el *commit* de la rama origen (generalmente el último)
    - Seleccionar **Create Merge-Commit**
    - Especificar el modo (si se hace un *fast-forward* o si se crea un *commit* de combinación)

- Una vez integrada la rama `hotfix` con la rama `master`, se puede borrar la primera.

- En el caso de SmartGit, a pesar que se pueda hacer una combinación del tipo *fast-forward*, lo mismo pregunta si se la quiere hacer así o si se creará un *commit* de combinación (ver más adelante).

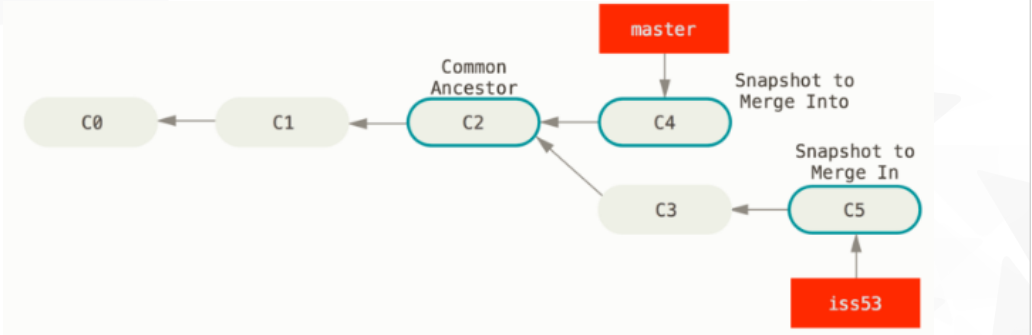
Combinación de ramas [10 | 18]

- **Situación (continuación):**
  - Se vuelve a la rama `iss53` y se continúa trabajando en la misma, produciendo el *commit* `C5`



Combinación de ramas [11 | 18]

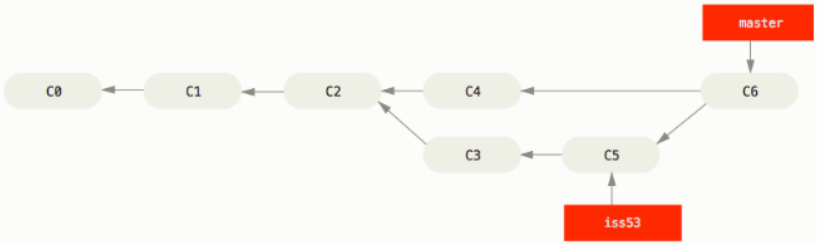
- **Situación (continuación):**
  - Una vez probadas las modificaciones en `iss53` se las integra con la rama `master` nuevamente



- Observar que ahora, el *commit* `C5` no está directamente a continuación del commit `C4`.

Combinación de ramas [12 | 18]

- **Situación (continuación):**
  - Ahora, como `C5` no viene directamente a continuación de `C4`, Git realiza la combinación en 3 pasos, usando los 2 *snapshots* apuntados por las puntas de las 2 ramas y el predecesor común a las 2



- Ahora Git, en lugar de mover el puntero de la rama hacia adelante, crea un nuevo *snapshot* como resultado de la combinación de 3 pasos y un *commit* que apunta al mismo. A este *commit* se lo llama "*commit* de combinación", y tiene la particularidad de tener más de un padre.

Combinación de ramas [13 | 18]

- Para combinar la rama `iss53` con la rama `master`:
  - Pararse en la rama destino (`master`)

```
git checkout master
```

  - Hacer la combinación

```
git merge iss53
```

  - Como ahora Git no puede hacer un *fast-forward*, crea un *commit* de combinación, por lo cual pide que se especifique un mensaje para el mismo

Combinación de ramas [14 | 18]

- Para combinar la rama `iss53` con la rama `master`:
  - Pararse en la rama destino (`master`)

```
git checkout master
```

  - Hacer la combinación

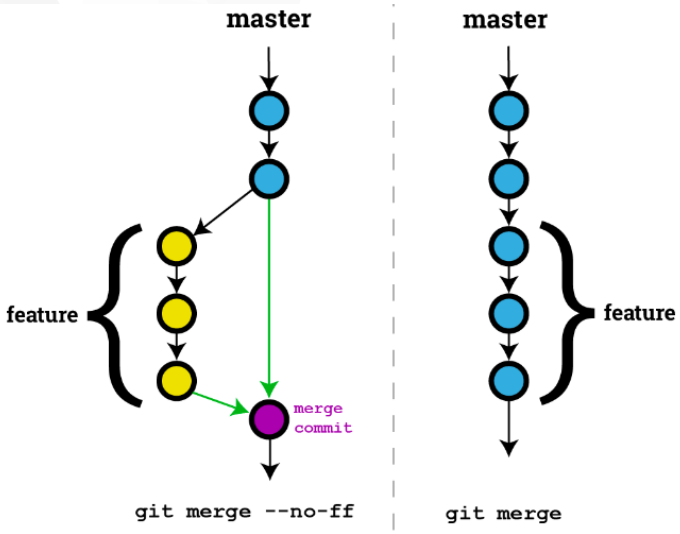
```
git merge hotfix
```

  - Como ahora Git no puede hacer un *fast-forward*, crea un *commit* de combinación, por lo cual pide que se especifique un mensaje para el mismo

Combinación de ramas [15 | 18]

- Cuando Git no puede simplemente mover el puntero *HEAD* realiza la combinación creando un *commit* de combinación
- Este comportamiento se corresponde con la opción `--no-ff` (*always create commit*) del comando `git merge`

Combinación de ramas [16 | 18]



- La opción `--no-ff` resulta útil cuando se quiere que Git mantenga un historial de los cambios.

Combinación de ramas [17 | 18]

- Git también permite especificar la opción `--ff-only`, la cual pide realizar la combinación sólo si puede hacer un *fast-forward*. Si no puede, que cancele la operación

- El hecho que se elija hacer un *fast-forward* no implica que la combinación se haga en esta modalidad. Si no se puede se creará un *commit* de combinación.

- De la misma forma, si la combinación se puede hacer haciendo un *fast-forward*, pero se elige la opción `--no-ff`, se creará el *commit* de combinación.

Combinación de ramas [18 | 18]

- Para combinar una rama:
  - Cambiar a la rama contra la cual se va a combinar (rama destino)
  - Elegir **Branch > Merge...**
  - Seleccionar el commit de la rama origen (generalmente el último)
  - Seleccionar **Create Merge-Commit**
  - Especificar el modo (si se hace un *fast-forward* o si se crea un *commit* de combinación)

- La opción "*Merge to Working Tree*" realiza una combinación del tipo "*squash*". Para mayor información sobre este tipo de combinación se puede consultar:

https://www.syntevo.com/doc/display/SG/How+to+perform+normal+merges+and+squash+merges

Resolución de conflictos [1 | 11]

- **Situación #1:**
  - A un archivo se le hacen modificaciones en 2 ramas pero en distintas partes
  - Luego se combinan las ramas
  - Git detecta que las modificaciones están en partes distintas y combina las 2 ramas

Resolución de conflictos [2 | 11]

- **Situación #2:**
  - A un archivo se le hacen modificaciones en 2 ramas pero en las mismas partes
  - Luego se combinan las ramas
  - Git detecta que las modificaciones están en las mismas partes produciéndose un conflicto, el cual se deberá solucionar para poder realizar la combinación

- Cuando se produce un conflicto, Git queda a mitad de camino en el proceso de combinación.

Resolución de conflictos [3 | 11]

- **Situación #2 (continuación):**
  - Si se abre un archivo en el que se produjeron conflictos, se pueden ver los cambios que hace Git para indicar donde está el problema:
    - Los cambios correspondientes a la rama destino están en el bloque comprendido entre `<<<<<<< HEAD` y `=======`
    - Los cambios correspondientes a la rama origen están en el bloque comprendido entre `=======` y `>>>>>>> <rama origen>`

Resolución de conflictos [4 | 11]

- Opciones para resolver los conflictos:
  - Abortar el proceso de combinación
  - Resolver los conflictos manualmente
  - Usar una herramienta para combinación (para el caso de la línea de comandos)

Resolución de conflictos [5 | 11]

- Para abortar el proceso de combinación se usa el comando `git merge --abort`
- En SmartGit, para abortar el proceso de combinación se elige **Branch > Abort**
- Al abortar la combinación, Git saca las marcas que indicaban los lugares de conflicto

Resolución de conflictos [6 | 11]

- Resolución manual de conflictos:
  - Se debe ir archivo por archivo (en conflicto) modificándolos según las necesidades
  - Por ejemplo, suponer que deban quedar las modificaciones correspondientes a la rama destino:
    - Se borra todo lo comprendido entre `=======` y `>>>>>>> <rama origen>` (incluyendo estos 2 marcadores)
    - También se borra `<<<<<<< HEAD`

Resolución de conflictos [7 | 11]

- Resolución manual de conflictos (continuación):
  - Una vez guardados los archivos con conflicto se los agrega al *index* y se hace el *commit*
- Para resolver los conflictos manualmente, SmartGit brinda 2 opciones:
  - Editor **Conflict Resolver**
  - Ventana **Resolve**

- Si al hacer el *commit* no se especifica un mensaje, Git abre un editor con un mensaje predeterminado, el cual se puede aceptar o modificar.

Resolución de conflictos [8 | 11]

- Editor **Conflict Resolver**
  - Se abre para el archivo especificado seleccionando **Query > Conflict Resolver** (o mediante el menú contextual)
  - Editor de 3 paneles:
    - Rama destino (izquierda)
    - Resultado de la combinación (medio)
    - Rama origen (derecha)

Resolución de conflictos [9 | 11]

- Editor **Conflict Resolver (continuación)**
  - Luego de resolver el conflicto, SmartGit pregunta si se quieren llevar los archivos modificados al *index*:
    - Si se responde que sí (opción predeterminada): se completa el proceso de combinación
    - Si se responde que no: habrá que hacer un *commit* para completar el proceso de combinación

Resolución de conflictos [10 | 11]

- Ventana **Resolve**
  - Se abre para el archivo especificado seleccionando **Local > Resolve** (o mediante el menú contextual)
  - Permite especificar la acción para resolver el conflicto:
    - Dejar en el estado actual (difícil)
    - Dejar en el estado de la rama destino
    - Dejar en el estado de la rama origen
    - Abrir la ventana **Conflict Resolver**

- Si se elije la opción para dejar en el estado actual, luego se deberá editar a mano el archivo con los los marcadores empleados por Git para indicar el lugar de conflicto.

Resolución de conflictos [11 | 11]

- Ventana **Resolve (continuación)**
  - Al resolver el conflicto tomando la versión de la rama origen, o de la rama destino, se puede especificar si se quieren llevar los archivos modificados al *index*:
    - Si se los lleva al *index*: se completa el proceso de combinación
    - Si no se los lleva al *index*: habrá que hacer un *commit* para completar el proceso de combinación

Recomendaciones [1 | 2]

- Mantener las líneas lo más cortas posibles: a mayor longitud de las líneas, mayores las probablidades que se produzcan conflictos
- Mantener los *commits* chicos y relacionados con algo
- Tener cuidado con las ediciones del tipo espacios en blanco, tabulaciones, retornos de línea

Recomendaciones [2 | 2]

- Tratar de combinar con frecuencia
- Integrar los cambios de la rama `master` a medida que se progresa: si se hacen muchos *commits* a esta rama, la otra queda desincronizada, aumentando las posibilidades de conflictos

Repositorios locales y remotos [1 | 21]

- Repositorio remoto: permite que varias personas colaboren entre sí, o que una misma persona pueda clonarlo en distintas máquinas
- En el caso de Git esto se puede hacer mediante un servidor central (otro repositorio Git) en el cual se guardan los cambios para que otros los puedan ver, descargar, hacer sus propios cambios y subirlos nuevamente

- Git es un sistema de control de versiones distribuido, por lo cual no hay ninguna diferencia entre un repositorio local y uno remoto (hay *commits*, ramas, un puntero *HEAD*, etc).
- Lo único es que en el servidor hay software que le permite a Git comunicarse con otros clientes Git.
- Además, el hecho que el servidor sea el repositorio central es una convención (todos se pusieron de acuerdo en usar ese repositorio como el lugar donde sincronizar los cambios).

Repositorios locales y remotos [2 | 21]

- **Situación:**
  - Un usuario tiene en su máquina un repositorio local
  - El usuario quiere tener una copia remota de su repositorio local para que pueda trabajar desde distintas máquinas
- **Tareas:**
  - 1. Creación del repositorio remoto en GitHub
  - 2. Agregado del repositorio remoto en su máquina

Repositorios locales y remotos [3 | 21]

- **1. Creación del repositorio remoto en GitHub:**
  - Registrarse con un usuario y clave (*sign up*)
  - Iniciar la sesión en GitHub (*sign in*)
  - Crear un repositorio:
    - Decidir si será público o privado
    - Decidir si se inicializará con un archivo `README`
    - Decidir si se incluirá un archivo `.gitignore`
  - Luego de creado, tomar nota de su URL

- La URL del repositorio también se puede obtener ingresando al mismo y luego seleccionando el botón "**Clone or download**".

Repositorios locales y remotos [4 | 21]

- **2. Agregado del repositorio remoto en su máquina:**
  - Para agregar un repositorio remoto se necesita:
    - Un nombre
    - La URL del repositorio
  - Para agregar un repositorio remoto, dentro de la carpeta donde se encuentra el repositorio local se ejecuta `git remote add <nombre> <URL>`

- Para borrar un repositorio remoto se ejecuta el comando `git remote rm <nombre>`.

Repositorios locales y remotos [5 | 21]

- **2. Agregado del repositorio remoto en su máquina (continuación):**
  - Para ver los repositorios remotos que se tienen configurados, se ejecuta el comando `git remote`
  - El comando `git remote` sólo muestra una lista con los nombres de los mismos. Si se ejecuta `git remote -v` también se muestran las URLs de los mismos

- Al ejecutar el comando `git remote -v` aparecen 2 entradas: una para hacer un *fetch* y otra para hacer un *push* (ver más adelante). Por lo general las URLs que usa para hacer el *fecth* y el *push* son iguales, pero podrían ser distintas (se podría tener un repositorio del tipo sólo lectura del cual se hace el *fetch*, y otro donde se hace el *push*). En un mismo proyecto pueden haber varios repositorios remotos.

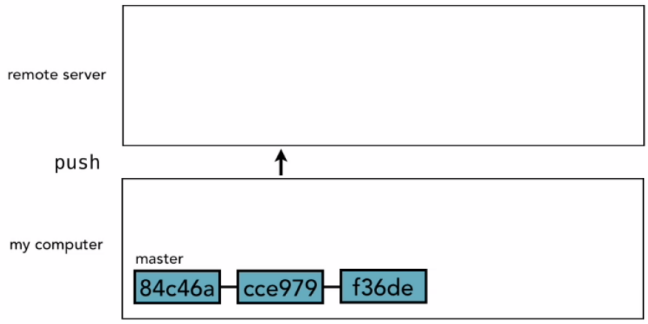
Repositorios locales y remotos [6 | 21]

- **2. Agregado del repositorio remoto en su máquina (continuación):**
  - En SmartGit, para agregar un repositorio remoto se elige **Remote > Add** y se especifica la URL y el nombre
  - En SmartGit, los repositorios remotos se muestran en la ventana **Branches** en la entrada correspondiente al nombre del repositorio

- El nombre predeterminado que usa Git para el nombre de un repositorio es `origin`.

Repositorios locales y remotos [7 | 21]

- ***Push***:
  - Consiste en llevar los *commits* de una rama local al repositorio remoto



- En la figura, se llevan los *commits* de la rama `master` al servidor remoto.

Repositorios locales y remotos [8 | 21]

- ***Push* (continuación):**
  - Para hacer el *push* de la rama `master` anterior, se ejecuta el comando:

```
git push -u <repositorio remoto> master
```

  - Opción `-u` (`--set-upstream`): permite que cuando se vuelva a hacer el *push* de la rama no se tenga que especificar el repositorio remoto (al estar parado en la rama se puede ejecutar `git push` simplemente)

- Al hacer el *push*, Git solicita un usuario y clave que tenga acceso al repositorio remoto.

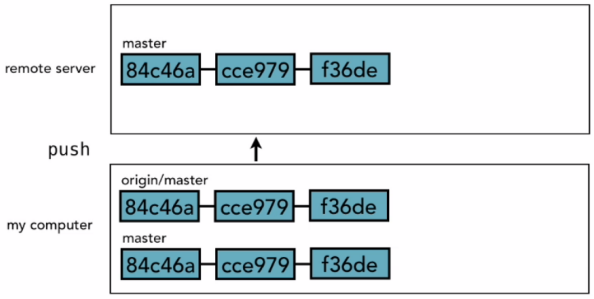
Repositorios locales y remotos [9 | 21]

- ***Push* (continuación):**
  - En SmartGit, para hacer el *push*:
    - Pararse en la rama que se desee
    - Elegir **Remote > Push...**
    - Luego de hacer el *push*, SmartGit muestra la rama local e informa cómo está con respecto a la rama remota

- Al hacer el *push*, SmartGit solicita la primera vez un usuario y clave que tenga acceso al repositorio remoto (no es necesario emplear una clave maestra). La clave maestra es una clave que protege las demás claves usadas en la autenticación de servidores. Se puede usar una o no.

Repositorios locales y remotos [10 | 21]

- ***Push* (continuación):**
  - Al hacer el *push* en el servidor remoto se crea la misma rama que la local (`master` en este caso)
  - En el repositorio local se crea otra rama: `origin/master`



- La rama que crea el servidor remoto es igual a la local, con los mismos *commits* (incluso los mismos IDs).

Repositorios locales y remotos [11 | 21]

- **Tipos de ramas:**
  - Ramas locales:
    - Ramas que sólo puede ver el usuario local (están en el repositorio local)
    - Ejemplo: rama `master` (local, antes del *push*)
    - A estas ramas también se las llama "*Non-Tracking*"

Repositorios locales y remotos [12 | 21]

- **Tipos de ramas (continuación):**
  - Ramas remotas:
    - Ramas que están en un repositorio remoto
    - Se crean haciendo un *push* de una rama local
    - Ejemplo: rama `master` (remota)

Repositorios locales y remotos [13| 21]

- **Tipos de ramas (continuación):**
  - Ramas "*Remote-Tracking*":
    - Ramas "locales" que son copias de ramas remotas (como un *cache* local de las ramas remotas)
    - En el ejemplo, cuando se hizo el *push* de `master` al repositorio remoto, en el repositorio local se creó una rama local "*Remote-Tracking*" llamada `origin/master` (`origin/master` mira a `master` en `origin`)

Repositorios locales y remotos [14 | 21]

- **Tipos de ramas (continuación):**
  - Ramas "*Remote-Tracking*":
    - Se actualizan haciendo un *fetch* (o un *pull*) por ejemplo
    - Para verlas se ejecuta el comando `git branch -r`

- Las ramas locales se pueden ver con el comando `git branch`, mientras que las ramas "*Remote-Tracking*" con el comando `git branch -r`.

- Git no mantiene 2 copias de los objetos entre las ramas "`master`" y "`origin/master`", sino punteros.

Repositorios locales y remotos [15 | 21]

- **Tipos de ramas (continuación):**
  - Ramas "*Local-Tracking*":
    - Son ramas locales que miran otras ramas (generalmente del tipo "*Remote-Tracking*", pero no necesariamente)
    - En el ejemplo, al hacer el *push* la opción `-u` configura a `master` para que mire a `origin/master`, quien a su vez mira a `master` en el repositorio remoto

- A estas ramas también se las conoce como "*Tracking*" solamente.

- Al especificar la opción `-u`:
- En el repositorio local, la rama `master` al mirar a `origin/master` es una rama "*Local-Tracking*".
- En el repositorio local, la rama `origin/master` es una rama "*Remote-Tracking*".
- En el repositorio remoto, la rama `master` es local en el mismo, y remota para el repositorio local.

Repositorios locales y remotos [16 | 21]

- **Tipos de ramas (continuación):**
  - Ramas "*Local-Tracking*":
    - Si `master` fuera "*Non-Tracking*" cada vez que se haga un *push* se deberá especificar el repositorio remoto
    - Para distinguir entre las ramas *Tracking* de las *Non-Tracking* se usa el comando `git branch -vv`

- Si no se hubiera especificado la opción `-u`:
- En el repositorio local, la rama `master` no mira a `origin/master`, y por lo tanto es una rama "*Non-Tracking*".
- En el repositorio local, la rama `origin/master` es una rama "*Remote-Tracking*".
- En el repositorio remoto, la rama `master` es local en el mismo, y remota para el repositorio local.

Repositorios locales y remotos [17 | 21]

- **Tipos de ramas (continuación):**
  - Para crear una rama "*Local-Tracking*" para que mire a otra, se puede usar:

```
git checkout -b <rama> <rama a mirar>
```

  - Una rama "*Non-Tracking"* se puede transformar en "*Tracking"* con el comando:

```
git branch -set-upstream-to=
 <repositorio remoto>/<rama> <rama non-tracking>
```


- La opción `-b` de `git checkout` hace que después de crear la rama se cambie a la misma. La opción `--track` de `git branch` crea y cambia a la rama.

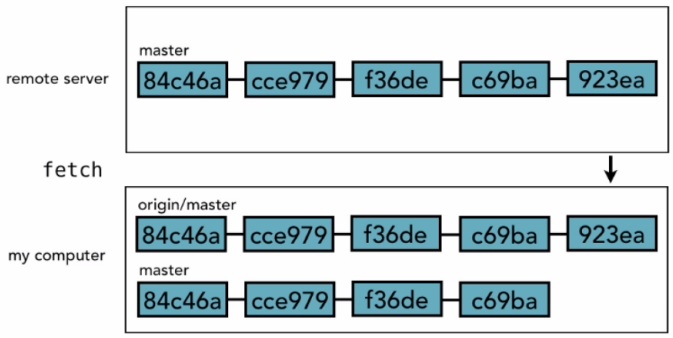
- En el archivo `.git/config` se puede cómo guarda Git las ramas para distinguir las *Tracking* de las *Non-Tracking*. Editando este archivo se puede también transformar una rama *Tracking* en *Non-Tracking*, y viceversa.

Repositorios locales y remotos [18 | 21]

- **Tipos de ramas (continuación):**
  - En SmartGit, si en algún momento se quiere que una rama deje de mirar a otra, se la selecciona y se elige **Branch > Stop Tracking...**
  - Para que vuelva a mirar a otra, se elige **Branch > Set Tracked Branch** y se elige la rama

Repositorios locales y remotos [19 | 21]

- ***Fetch*:**
  - Trae los *commits* de una rama remota a una rama "*Remote-Tracking*" (`origin/master`)



- El *fetch* mantiene la sincronización entre `origin/master` (local) y `master` en el servidor remoto.

Repositorios locales y remotos [20 | 21]

- ***Fetch* (continuación):**
  - Para hacer el *fetch* de la rama `master` anterior:
    - Ejecutar el comando:

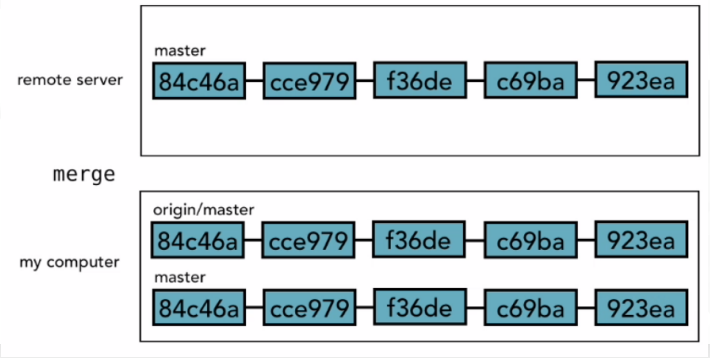
```
git fetch <repositorio remoto> master
```

  - En SmartGit, se elige **Remote > Fetch**

- Al hacer el *fetch* puede no especificarse la rama, con lo cual se hace para todas las ramas. Si hubiera un único repositorio remoto, se puede omitir el nombre del repositorio.

Repositorios locales y remotos [21 | 21]

- ***Fetch* (continuación)**
  - Para que los cambios queden reflejados en la rama `master` local se debe hacer una combinación (*merge*)



Trabajo individual en GitHub [1 | 10]

- **Situación:**
  - Máquina A: tiene un repositorio local Git
  - Máquina B: quiere tener una copia local del repositorio
- **Tareas:**
  - 1. Crear el repositorio remoto en GitHub
  - 2. Agregar el repositorio remoto en la máquina A
  - 3. Hacer un *push* desde la máquina A
  - 4. Clonar el repositorio remoto en la máquina B
  - 5. Crear el resto de ramas locales

Trabajo individual en GitHub [2 | 10]

- **1. Crear el repositorio remoto en GitHub**
  - Ya visto
- **2. Agregar el repositorio remoto en la máquina A**
  - Ya visto
- **3. Hacer un *push* desde la máquina A**
  - Hacer el *push* por cada rama que se quiera subir

Trabajo individual en GitHub [3 | 10]

- **4. Clonar el repositorio remoto desde la máquina B:**
  - Del sitio de GitHub obtener la URL del repositorio que se quiere clonar
  - Ejecutar: `git clone <URL> <carpeta>`

- La URL del repositorio se puede obtener ingresando al mismo y luego seleccionando el botón "*Clone or download*".

- Cuando no se especifica la carpeta donde estará el repositorio, Git crea una cuyo nombre es la última parte de la URL del repositorio que se está clonando sin el .git. Por ejemplo, si la URL del repositorio a clonar es:

  https://github.com/luisenieto/EjemplosGit.git

- Se crearía la carpeta "`EjemplosGit`".

- No sólo se puede clonar un repositorio remoto: también se puede clonar uno local haciendo `git clone <carpeta rep.> <carpeta copia>`

Trabajo individual en GitHub [4 | 10]

- **4. Clonar el repositorio remoto desde la máquina B (continuación):**
  - Con el comando anterior al clonar un repositorio:
    - 1. Si no se especifica la carpeta, se crea una
    - 2. La carpeta se inicializa como repositorio
    - 3. Al repositorio local se le agrega un repositorio remoto con el nombre `origin` de forma predeterminada
    - 4. Se descargan todos los archivos y *commits*

Trabajo individual en GitHub [5 | 10]

- **4. Clonar el repositorio remoto desde la máquina B (continuación):**
  - Cuando se clona un repositorio:
    - 5. Crea y hace el *checkout* de una rama inicial a partir de la rama predeterminada del repositorio (generalmente `master`)
    - 6. Por cada rama en el repositorio remoto se crea su correspondiente rama "*Remote-Tracking*" en el repositorio local

Trabajo individual en GitHub [6 | 10]

- **4. Clonar el repositorio remoto desde la máquina B (continuación):**
  - Sobre el *checkout* de la rama inicial (5):
    - Un repositorio puede tener varias ramas
    - Una de estas ramas es en la que se está parado
    - Dentro de la carpeta `.git`, el archivo `HEAD` tiene la información de la rama sobre la que se está parado. Si un repositorio tiene las ramas `master` y `rama1` y se está parado en `rama1`:

      ```
      ref: refs/heads/rama1
      ```


- Si se estuviera parado en `master` mostrarría:

```
ref: refs/heads/master.
```

Trabajo individual en GitHub [7 | 10]

- **4. Clonar el repositorio remoto desde la máquina B (continuación):**
  - Sobre el *checkout* de la rama inicial (5):
    - Siguiendo con el ejemplo, luego de clonar el repositorio anterior, al ejecutar `git branch` para ver las ramas locales, se vería:

      ```
      * rama1
      ```

    - Es decir, Git hice el *checkout* sólo de la rama en la que se estaba parado en el repositorio remoto al momento de clonarse

Trabajo individual en GitHub [8 | 10]

- **4. Clonar el repositorio remoto desde la máquina B (continuación):**
  - Sobre el *checkout* de la rama inicial (5):
    - Si se quisiera hacerle el checkout a una rama distinta a la que se estaba parado, se especifica la opción `--branch <rama>` al comando `git clone`
    - Si el respotorio que se está clonando está en GitHub, en el mismo se puede cambiar la rama a la cual se le hará el *checkout* desde el menú **Settings**, opción **Branches**

Trabajo individual en GitHub [9 | 10]

- **4. Clonar el repositorio remoto desde la máquina B (continuación):**
  - Sobre la creación de las ramas "*Remote-Tracking*" (6):
    - Siguiendo con el ejemplo, luego de clonar el repositorio anterior, al ejecutar `git branch -r` para ver las ramas del tipo "*Remote-Tracking*", se vería:

```
origin/HEAD -> origin/rama1
origin/master
origin/rama1
```

Trabajo individual en GitHub [10 | 10]

- **4. Clonar el repositorio remoto desde la máquina B (continuación):**
  - Sobre la creación de las ramas "*Remote-Tracking*" (6):
    - Como se puede ver del ejemplo anterior, además de las ramas `master` y `rama1`, se crea la rama "*Remote-Tracking*" `origin/HEAD`
    - Esta rama se puede borrar ejecutando:

```
git remote set-head origin -d
```

Trabajo individual en GitHub [11 | 10]

- **4. Clonar el repositorio remoto desde la máquina B (continuación):**
  - Para clonar un repositiro en SmartGit elegir **Repository > Clone...**
  - Especificar si se incluirán o no submódulos
  - Elegir la rama a la que se hará el *chekout* (*Check Out Branch*)

- A veces los proyectos de software comparten partes en común con otros proyectos de software. Git cuenta con una característica llamada submódulo, la cual permite embeber un repositorio Git dentro de otro.

- Un submódulo es un repositorio anidado embebido en una carpeta dedicada dentro de la carpeta de trabajo (la cual pertenece al repositorio padre).

- Generalmente se clona todo el repositorio (*Fetch all Heads and Tags*). Si se quiere clonar una rama o etiqueta en particular, no se selecciona esta opción.

Trabajo individual en GitHub [12 | 10]

- **4. Clonar el repositorio remoto desde la máquina B (continuación):**
  - Especificar si se clonará todo el repositorio o una rama en particular (*Fetch all Heads and Tags*)
  - Especificar la ubicación local del repositorio a clonar

- Generalmente se clona todo el repositorio (*Fetch all Heads and Tags*). Si se quiere clonar una rama o etiqueta en particular, no se selecciona esta opción.

- En la ventana **Branches** se puede cambiar el nombre del repositorio remoto (`origin` es el valor predeterminado).

Trabajo individual en GitHub [13 | 10]

- **5. Crear el resto de ramas locales:**
  - Siguiendo con el ejemplo anterior, se le había hecho el *checkout* a la rama `rama1`
  - No se puede trabajar sobre las ramas remotas. Si sólo se quiere ver qué hay en la rama remota `master`, se puede hacer:

```
git checkout origin/master
```

  - En SmartGit se selecciona la rama remota y al hacer el *checkout* se elige que no se cree la rama local

Trabajo individual en GitHub [14 | 10]

- **5. Crear el resto de ramas locales (continuación):**
  - Si se quiere trabajar sobre la rama, se debe crear la rama local:

```
git checkout master
```

  - En SmartGit se selecciona la rama remota y al hacer el *checkout* se elige que se cree la rama local

- Hacer el checkout de una rama local a partir de una rama del tipo "Remote-Tracking" crea automáticamente una rama "Tracking".
- El comando `git checkout master` (en este caso) hace lo siguiente: si no encuentra la rama `master`, pero existe una rama del tipo "*Remote-Tracking*" en un único repositorio remoto con el mismo nombre, hace:

```
git checkout master origin/master
```

Trabajo individual en GitHub [15 | 10]

- **Situación:**
  - En la máquina A hay un repositorio local Git, el cual está subido a GitHub y clonado en la máquina B
  - Se quiere trabajar desde cualquier máquina
- **Tareas:**
  - 1. Hacer un *push* de la rama en la máquina A
  - 2. Hacer un *fetch* en la máquina B
  - 3. Combinar ramas remotas y locales en la máquina B

Trabajo individual en GitHub [16 | 10]

- **1. Hacer un *push* de la rama en la máquina A:**
  - Si se está trabajando en `rama1`, después de hacer el último *commit*, al ejecutar `git status` se informa que la misma está delante de `origin/rama1` en una cantidad de *commits*. Así informa Git que hay cambios en `rama1` (local) que no están en `rama1` (remota)
  - En SmartGit, después de hacer el *push* aparecería como `rama1 > origin`

- Esto es así porque `rama1` está mirando a `origin/rama1`.

Trabajo individual en GitHub [17 | 10]

- **1. Hacer un *push* de la rama en la máquina A (continuación):**
  - Ahora se hace el push de `rama1`. Si ahora se ejecuta `git status` se informa que la misma está actualizada con `origin/rama1`
  - En SmartGit, después de hacer el *push* aparecería como `rama1 = origin`

Trabajo individual en GitHub [18 | 10]

- **2. Hacer un *fetch* en la máquina B:**
  - Después de hacer el *fetch*, si se ejecuta `git status` sobre `rama1` se informa que `origin/rama1` está delante de `rama1` en una cantidad de *commits*. Así informa Git que hay cambios en `rama1` (remota) que no están en `rama1` (local)
  - En SmartGit, después de hacer el *push* aparecería como `rama1 < origin`

Trabajo individual en GitHub [19 | 10]

- **3. Combinar ramas remotas y locales en la máquina B:**
  - Después de combinar las ramas remotas y locales, si se ejecuta `git status` sobre `rama1` se informa que la misma está actualizada con `origin/rama1`
  - En SmartGit, después de combinar aparecería como `rama1 = origin`

Trabajo colaborativo en GitHub [1 | 2]

- Para agregar colaboradores para un proyecto en GitHub, se elige **Settings > Collaborators** (desde GitHub)
- Cuando se agrega un colaborador, GitHub le envía un correo y cuando el colaborador acepta la invitación, tiene acceso al repositorio y lo puede clonar para empezar a trabajar

Trabajo colaborativo en GitHub [2 | 2]

- Flujo de trabajo en un repositorio remoto:
  - Al comenzar la sesión de trabajo, hacer un *fetch* de los últimos cambios en el servidor
  - Hacer un *merge* entre las ramas "*Remote-Tracking*" y locales
  - Hacer los *commits* en las ramas locales
  - Antes de hacer el *push* al repositorio remoto, volver a hacer un *fetch* y combinar

- La razón de volver a hacer un *fetch* y combinar antes de hacer el *push* se debe a que mientras se estuvo trabajando localmente se pudieron haber producido cambios que se deban tener en cuenta para el trabajo que se está haciendo.

Resumen [1 | 2]

- Introducción a Git
- Borrado de archivos
- Movimiento / renombrado de archivos
- Cómo se ignoran archivos
- Vuelta atrás sobre la carpeta de trabajo
- Vuelta atrás del último *commit*
- Vuelta atrás a un estado específico

Resumen [2 | 2]

- Trabajo con ramas
- Combinación de ramas
- Resolución de conflictos
- Recomendaciones
- Repositorios remotos
- Repositorios locales y remotos
- Trabajo individual en GitHub
- Trabajo colaborativo en GitHub

Otro material [1 | 2]

- https://git-scm.com/
- https://www.git-tower.com/help/mac/commit-history/undo-commits
- https://www.atlassian.com/git/tutorials/undoing-changes/git-revert
- https://stackoverflow.com/questions/4693588/git-what-is-a-tracking-branch
- https://stackoverflow.com/questions/16408300/what-are-the-differences-between-local-branch-local-tracking-branch-remote-bra/24785777
- http://www.syntevo.com/doc/display/SG/Branches

Otro material [2 | 2]

- https://dzone.com/articles/git-differences-between-local-branch-non-tracking-1
- https://stackoverflow.com/questions/354312/why-is-origin-head-shown-when-running-git-branch-r