Tema 5. Sistemas de Control de Versiones

1. ¿Qué son los SVC?

1.1 Definición, clasificación y funcionamiento

Se llama control de versiones a la gestión de los diversos cambios que se realizan sobre los elementos de algún producto o una configuración del mismo. Una versión, revisión o edición de un producto, es el estado en el que se encuentra dicho producto en un momento dado de su desarrollo o modificación. Aunque un sistema de control de versiones puede realizarse de forma manual, es muy aconsejable disponer de herramientas que faciliten esta gestión dando lugar a los llamados sistemas de control de versiones o SVC (del inglés *System Version Control*).

Estos sistemas facilitan la administración de las distintas versiones de cada producto desarrollado, así como las posibles especializaciones realizadas (por ejemplo, para algún cliente específico). Ejemplos de este tipo de herramientas son entre otros: CVS, Subversion, SourceSafe, ClearCase, Darcs, Bazaar, Plastic SCM, Git, Mercurial, Perforce.

1.2 Terminología

Repositorio ("repository"): El repositorio es el lugar en el que se almacenan los datos actualizados e históricos de cambios.

Revisión ("revision"): Una revisión es una versión determinada de la información que se gestiona. Hay sistemas que identifican las revisiones con un contador (Ej. *subversion*). Hay otros sistemas que identifican las revisiones mediante un código de detección de modificaciones (Ej. git usa SHA1).

Etiqueta ("tag"): Los *tags* permiten identificar de forma fácil revisiones importantes en el proyecto. Por ejemplo, se suelen usar tags para identificar el contenido de las versiones publicadas del proyecto.

Rama ("branch"): Un conjunto de archivos puede ser ramificado o bifurcado en un punto en el tiempo de manera que, a partir de ese momento, dos copias de esos archivos se pueden desarrollar a velocidades diferentes o en formas diferentes de forma independiente el uno del otro.

Cambio ("change"): Un cambio (o ***diff***, o *delta*) representa una modificación específica de un documento bajo el control de versiones. La granularidad de la modificación que es considerada como un cambio varía entre los sistemas de control de versiones.

Desplegar ("checkout"): Es crear una copia de trabajo local desde el repositorio. Un usuario puede especificar una revisión en concreto u obtener la última. El término *'checkout'* también se puede utilizar como un sustantivo para describir la copia de trabajo.

Confirmar ("commit"): Confirmar es escribir o mezclar los cambios realizados en la copia de trabajo del repositorio. Los términos *'commit'* y *'checkin'* también se pueden utilizar como sustantivos para describir la nueva revisión que se crea como resultado de confirmar.

Conflicto ("conflict"): Un conflicto se produce cuando diferentes partes realizan cambios en el mismo documento, y el sistema es incapaz de conciliar los cambios. Un usuario debe resolver el conflicto mediante la integración de los cambios, o mediante la selección de un cambio en favor del otro.

Cabeza ("head"): También a veces se llama *tip* (punta) y se refiere a la última confirmación, ya sea en el tronco (*'trunk'*) o en una rama (*'branch'*). El tronco y cada rama tienen su propia cabeza, aunque HEAD se utiliza a veces libremente para referirse al tronco.

Tronco ("trunk"): La única línea de desarrollo que no es una rama (a veces también llamada línea base, línea principal o máster).

Fusionar, integrar, mezclar ("merge"): Una fusión o integración es una operación en la que se aplican dos tipos de cambios en un archivo o conjunto de archivos. Algunos escenarios de ejemplo son los siguientes:

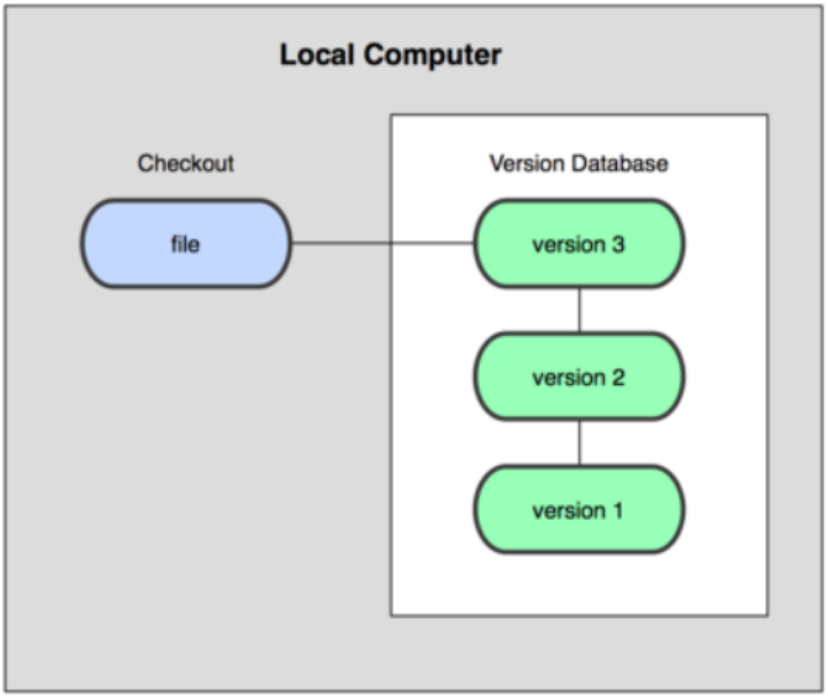
- Un usuario, trabajando en un conjunto de archivos, actualiza o sincroniza su copia de trabajo con los cambios realizados y confirmados, por otros usuarios, en el repositorio.
- Un usuario intenta confirmar archivos que han sido actualizado por otros usuarios desde el último despliegue ('checkout'), y el software de control de versiones integra automáticamente los archivos (por lo general, después de preguntarle al usuario si se debe proceder con la integración automática, y en algunos casos sólo se hace si la fusión puede ser clara y razonablemente resuelta).
- Un conjunto de archivos se bifurca, un problema que existía antes de la ramificación se trabaja en una nueva rama, y la solución se combina luego en la otra rama.
- Se crea una rama, el código de los archivos es independiente editado, y la rama actualizada se incorpora más tarde en un único tronco unificado.

1.3. Clasificación

Podemos clasificar los sistemas de control de versiones atendiendo a la arquitectura utilizada para el almacenamiento del código: locales, centralizados y distribuidos.

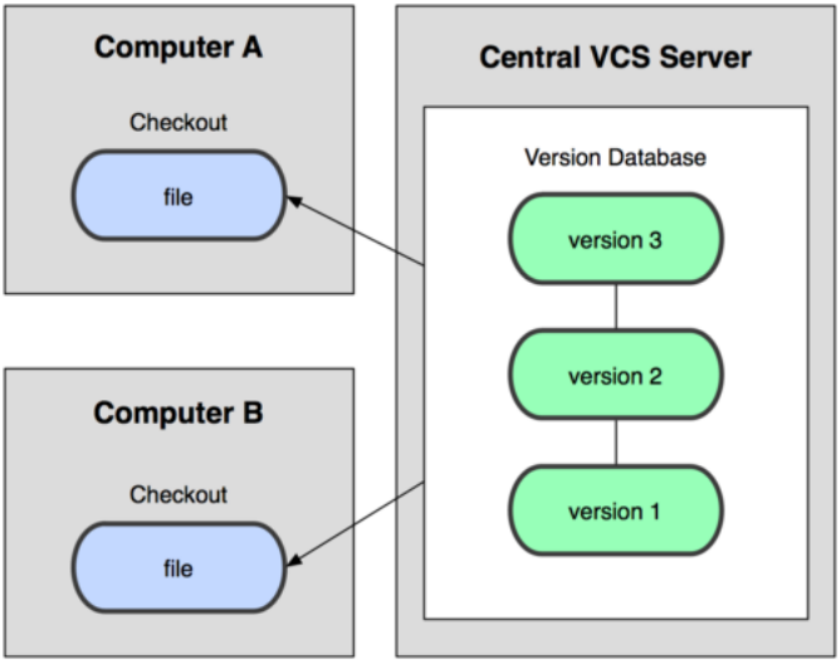
1.3.1 Locales

Los cambios son guardados localmente y no se comparten con nadie. Esta arquitectura es la antecesora de las dos siguientes.



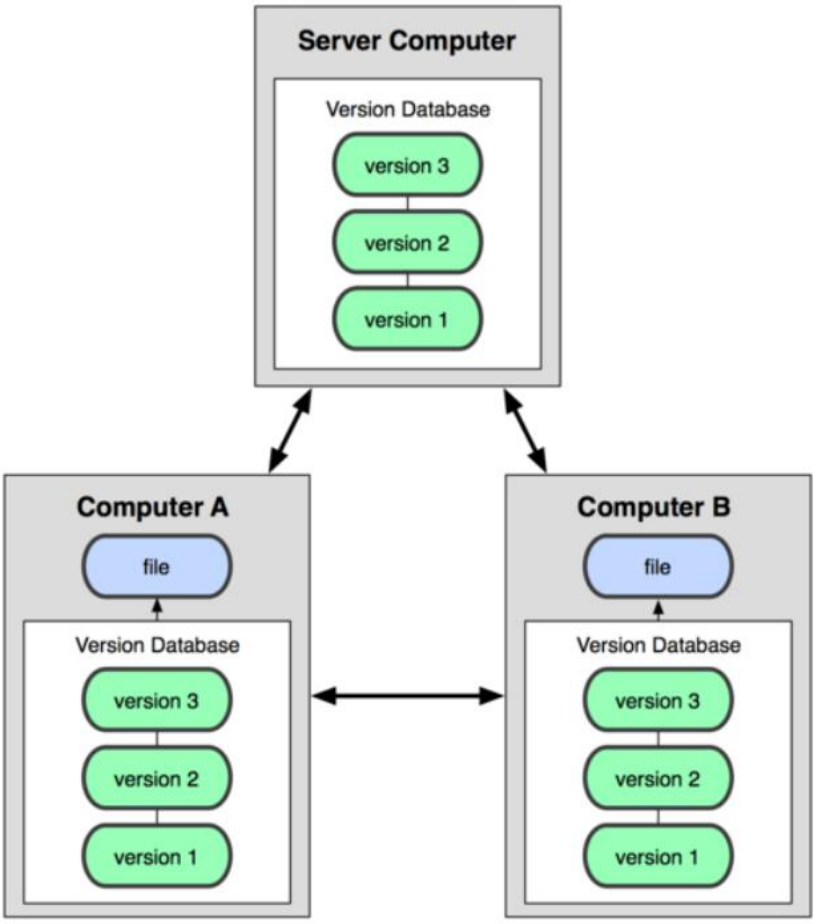
1.3.2 Centralizados

Existe un repositorio centralizado de todo el código, del cual es responsable un único usuario (o conjunto de ellos). Se facilitan las tareas administrativas a cambio de reducir flexibilidad, pues todas las decisiones fuertes (como crear una nueva rama) necesitan la aprobación del responsable. Algunos ejemplos son CVS y Subversion.



1.3.3 Distribuidos

Cada usuario tiene su propio repositorio. Los distintos repositorios pueden intercambiar y mezclar revisiones entre ellos. Es frecuente el uso de un repositorio, que está normalmente disponible, que sirve de punto de sincronización de los distintos repositorios locales. Ejemplos: Git y Mercurial.

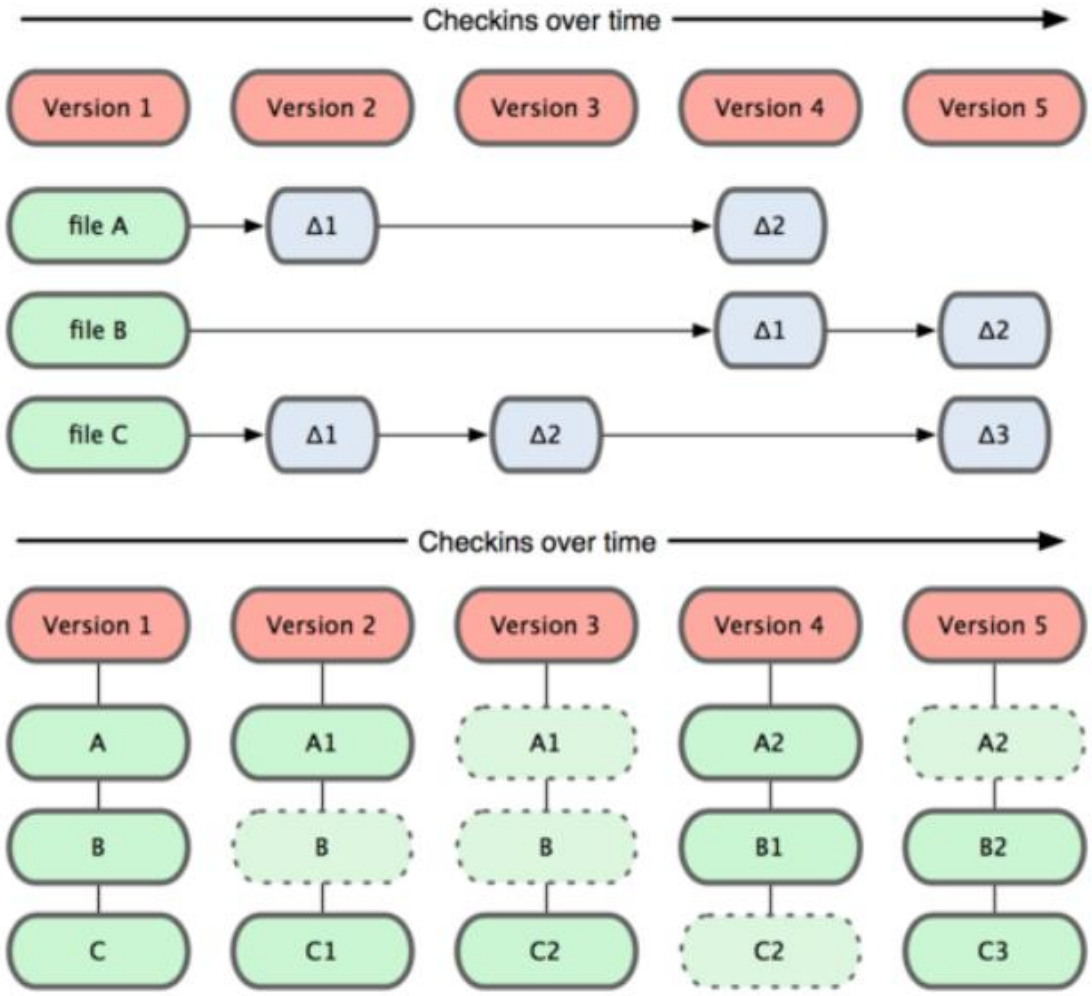


Ventajas de sistemas distribuidos

- No es necesario estar conectado para guardar cambios.
- Posibilidad de continuar trabajando si el repositorio remoto no está accesible.
- El repositorio central está más libre de ramas de pruebas.
- Se necesitan menos recursos para el repositorio remoto.
- Más flexibles al permitir gestionar cada repositorio personal como se quiera.

2. Introducción a git

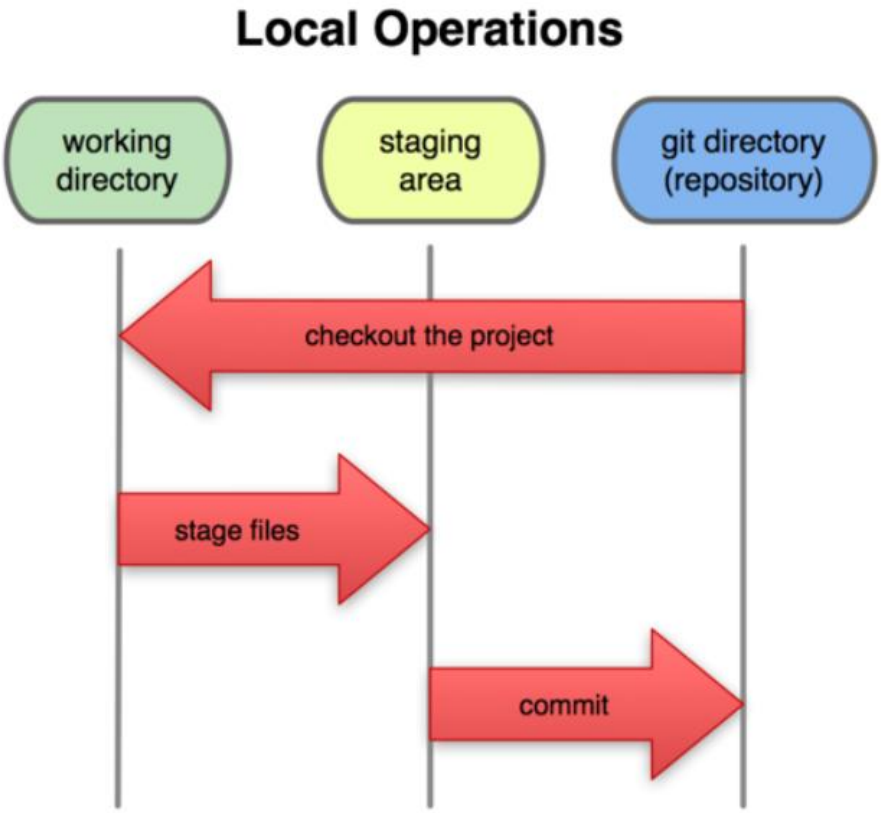
Git es un sistema de control de versiones distribuido que se diferencia del resto en el modo en que modela sus datos. La mayoría de los demás sistemas almacenan la información como una lista de cambios en los archivos, mientras que Git modela sus datos más como un conjunto de instantáneas de un mini sistema de archivos.



2.1 Los tres estados

Git tiene tres estados principales en los que se pueden encontrar tus archivos: confirmado (***committed***), modificado (***modified***), y preparado (***staged***). **Committed** significa que los datos están almacenados de manera segura en tu repositorio **local**. ***Modified*** significa que has modificado algún archivo, pero todavía no lo has confirmado a tu repositorio local. ***Staged*** significa que has marcado un archivo modificado en su versión actual para que vaya en tu próxima confirmación.

Esto nos lleva a las tres secciones principales de un proyecto de Git: el directorio de Git (**Git directory**), el directorio de trabajo (***working directory***), y el área de preparación (***staging area***).

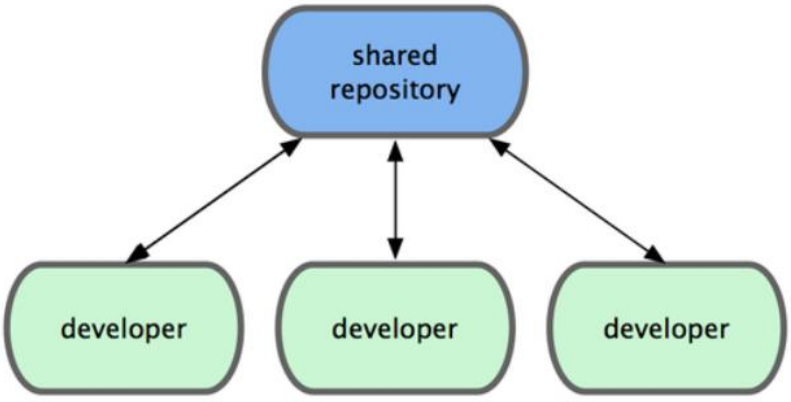


2.2 Flujos de trabajo distribuidos con git

Hemos visto en qué consiste un entorno de control de versiones distribuido, pero más allá de la simple definición, existe más de una manera de gestionar los repositorios. Estos son los flujos de trabajo más comunes en Git.

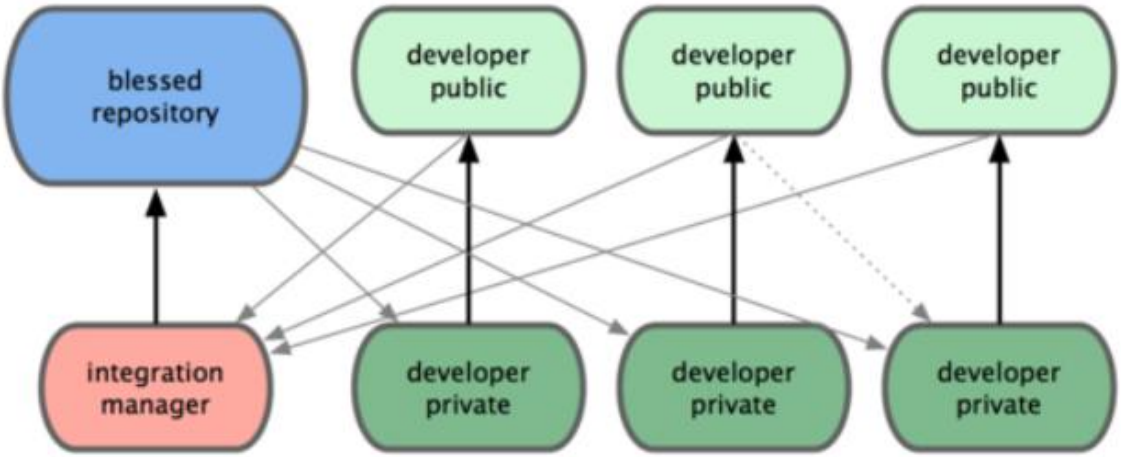
2.2.1 Flujo de trabajo centralizado

Existe un único repositorio o punto central que guarda el código y todo el mundo sincroniza su trabajo con él. Si dos desarrolladores clonan desde el punto central, y ambos hacen cambios; tan solo el primero de ellos en enviar sus cambios de vuelta lo podrá hacer limpiamente. El segundo desarrollador deberá fusionar previamente su trabajo con el del primero, antes de enviarlo, para evitar el sobreescribir los cambios del primero.



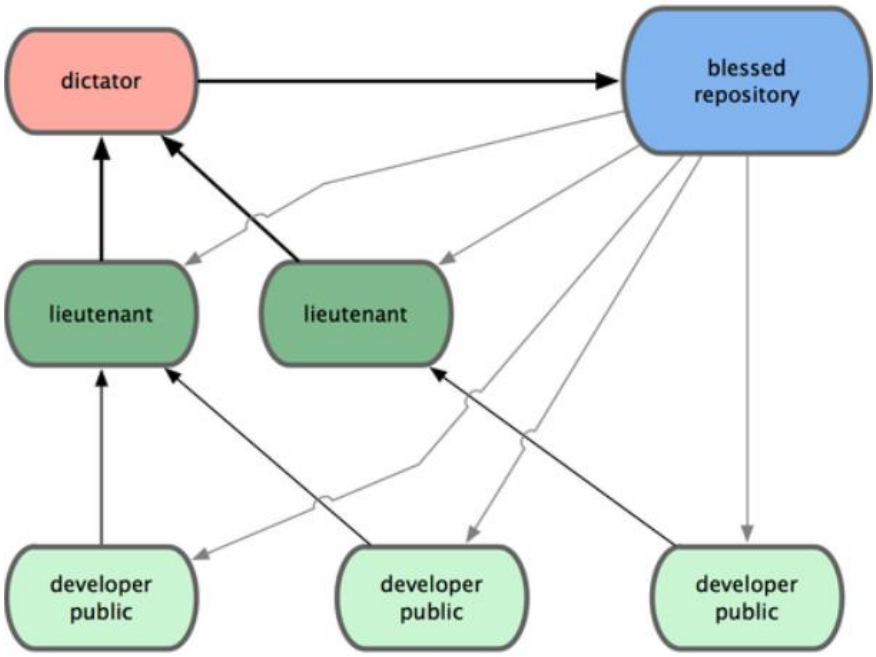
2.2.2 Flujo de trabajo del Gestor de Integraciones

Al permitir múltiples repositorios remotos, en Git es posible tener un flujo de trabajo donde cada desarrollador tenga acceso de escritura a su propio repositorio público y acceso de lectura a los repositorios de todos los demás. Habitualmente, este escenario suele incluir un repositorio canónico, representante "oficial" del proyecto.



2.2.3 Flujo de trabajo con Dictador y Tenientes

Es una variante del flujo de trabajo con múltiples repositorios. Se utiliza generalmente en proyectos muy grandes, con cientos de colaboradores. Un ejemplo muy conocido es el del *kernel* de Linux. Unos gestores de integración se encargan de partes concretas del repositorio; y se denominan tenientes. Todos los tenientes rinden cuentas a un gestor de integración; conocido como el dictador benevolente. El repositorio del dictador benevolente es el repositorio de referencia, del que recuperan (*pull*) todos los colaboradores.



3. Aspectos básicos de Git

3.1 Instalación

Instalando en Linux

Si quieres instalar Git en Linux a través de un instalador binario, en general puedes hacerlo a través de la herramienta básica de gestión de paquetes que trae tu distribución. Si estás en Fedora, puedes usar yum:

```
$ yum install git-core
```

O si estás en una distribución basada en **Debian como Ubuntu**, prueba con apt-get:

```
$ apt-get install git
```

Instalando en Windows

Instalar Git en Windows es muy fácil. El proyecto msysGit tiene uno de los procesos de instalación más sencillos. Simplemente descarga el archivo exe del instalador desde la página de GitHub, y ejecútalo:

http://msysgit.github.com/

Una vez instalado, tendrás tanto la versión de línea de comandos (incluido un cliente SSH que nos será útil más adelante) como la interfaz gráfica de usuario estándar. Se recomienda no modificar las opciones que trae por defecto el instalador.

Instalando en MacOS

En MacOS se recomienda tener instalada la herramienta homebrew. Después, es tan fácil como ejecutar:

```
$ brew install git
```

3.2 Configuración

Lo primero que deberías hacer cuando instalas Git es establecer tu nombre de usuario y dirección de correo electrónico. Esto es importante porque las confirmaciones de cambios (*commits*) en Git usan esta información, y es introducida de manera inmutable en los commits que envías:

```
$ git config --global user.name "John Doe"
```

```
$ git config --global user.email johndoe@example.com
```

También se recomienda configurar el siguiente parámetro[1]:

```
$ git config --global push.default simple
```

[1] Configura el comportamiento por defecto de ***git push***.

4. Uso básico de Git

4.1 Crear un Proyecto

Crear un programa "Hola Mundo"

Creamos un directorio donde colocar el código

```
$ mkdir curso-de-git
```

```
$ cd curso-de-git
```

Creamos un fichero hola.php que muestre Hola Mundo.

```
<?php
```

```
echo "Hola Mundo\n"; ?>
```

Crear el repositorio local

Para crear un nuevo repositorio se usa la orden `git init`

```
$ git init
```

```
Initialized empty Git repository in /home/cc0gobas/git/curso-de-git/.git/
```

Añadir la aplicación

Vamos a almacenar el archivo que hemos creado en el repositorio local para poder trabajar, después explicaremos para qué sirve cada orden.

```
$ git add hola.php
```

```
$ git commit -m "Creación del proyecto"
```

```
[master (root-commit) e19f2c1] Creación del proyecto
```

```
 1 file changed, 2 insertions(+)
```

```
 create mode 100644 hola.php
```

Comprobar el estado del repositorio

Con la orden `git status` podemos ver en qué estado se encuentran los archivos de nuestro repositorio.

```
$ git status
```

```
# On branch master
```

```
nothing to commit (working directory clean)
```

Si modificamos el archivo hola.php:

```
<?php

@print "Hola {$argv[1]}\n";
```

Y volvemos a comprobar el estado del repositorio:

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

Añadir cambios

Con la orden `git add` indicamos a git que prepare los cambios para que sean almacenados.

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

Confirmar los cambios

Con la orden `git commit` confirmamos los cambios definitivamente, lo que hace que se guarden permanentemente en nuestro repositorio.

```
$ git commit -m "Parametrización del programa"

[master efc252e] Parametrización del programa

 1 file changed, 1 insertion(+), 1 deletion(-)

$ git status

# On branch master

nothing to commit (working directory clean)
```

Diferencias entre *workdir* y *staging*.

Modificamos nuestra aplicación para que soporte un parámetro por defecto y añadimos los cambios.

```
<?php

$nombre = isset($argv[1]) ? $argv[1] : "Mundo";

@print "Hola, {$nombre}\n";
```

Esta vez añadimos los cambios a la fase de *staging* pero sin confirmarlos (*commit*).

```
git add hola.php
```

Volvemos a modificar el programa para indicar con un comentario lo que hemos hecho.

```
<?php

// El nombre por defecto es Mundo

$nombre = isset($argv[1]) ? $argv[1] : "Mundo";

@print "Hola, {$nombre}\n";
```

Y vemos el estado en el que está el repositorio

```
$ git status

# On branch master

# Changes to be committed:

#   (use "git reset HEAD <file>..." to unstage)

#

#   modified:   hola.php

#
```

```
# 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:   hola.php
#
```

Podemos ver como aparecen el archivo *hola.php* dos veces. El primero está preparado para ser confirmado y está almacenado en la zona de *staging*. El segundo indica que el directorio hola.php está modificado otra vez en la zona de trabajo (*workdir*).

Warning: Si volvieramos a hacer un git add hola.php sobreescribiríamos los cambios previos que había en la zona de *staging*.

Almacenamos los cambios por separado:

```
$ git commit -m "Se añade un parámetro por defecto"
[master 3283e0d] Se añade un parámetro por defecto
 1 file changed, 2 insertions(+), 1 deletion(-)
```

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

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

```
#
#   modified:   hola.php
#
```

```
$ git commit -m "Se añade un comentario al cambio del valor por defecto"
[master fd4da94] Se añade un comentario al cambio del valor por defecto
 1 file changed, 1 insertion(+)
```

Info El valor "." despues de git add indica que se añadan **todos** los archivos de forma recursiva.

Ignorando archivos

La orden **`git add .`** o **`git add nombre_directorio`** es muy cómoda, ya que nos permite añadir todos los archivos del proyecto o todos los contenidos en un directorio y sus subdirectorios. Es mucho más rápido que tener que ir añadiéndolos uno por uno. El problema es que, si no se tiene cuidado, se puede terminar por añadir archivos innecesarios o con información sensible.

Por lo general se debe evitar añadir archivos que se hayan generado como producto de la compilación del proyecto, los que generen los entornos de desarrollo (archivos de configuración y temporales) y aquellos que contentan información sensible, como contraseñas o tokens de autenticación. Por ejemplo, en un proyecto de C/C++, los archivos objeto no deben incluirse, solo los que contengan código fuente y los make que los generen.

Para indicarle a *git* que debe ignorar un archivo, se puede crear un fichero llamado ***.gitignore***, bien en la raíz del proyecto o en los subdirectorios que queramos. Dicho fichero puede contener patrones, uno en cada línea, que especiquen qué archivos deben ignorarse. El formato es el siguiente:

```
# .gitignore
dir1/           # ignora todo lo que contenga el directorio dir1
!dir1/info.txt  # El operador ! excluye del ignore a dir1/info.txt (sí se guardaría)
dir2/*.txt      # ignora todos los archivos txt que hay en el directorio dir2
dir3/**/*.txt   # ignora todos los archivos txt que hay en el dir3 y sus
subdirectorios
*.o             # ignora todos los archivos con extensión .o en todos los directorios
```

Cada tipo de proyecto genera sus ficheros temporales, así que para cada proyecto hay un .gitignore apropiado. Existen repositorios que ya tienen creadas plantillas. Podéis encontrar uno en https://github.com/github/gitignore

Ignorando archivos globalmente

Si bien, los archivos que hemos metido en .gitignore, deben ser aquellos ficheros temporales o de configuración que se pueden crear durante las fases de compilación o ejecución del programa, en ocasiones habrá otros ficheros que tampoco debemos introducir en el repositorio y que son recurrentes en todos los proyectos. En dicho caso, es más útil tener un *gitignore* que sea global a todos nuestros proyectos. Esta configuración sería complementaria a la que ya tenemos. Ejemplos de lo que se puede ignorar de forma global son los ficheros temporales del sistema operativo (*~, .nfs*) y los que generan los entornos de desarrollo.

Para indicar a *git* que queremos tener un fichero de *gitignore* global, tenemos que configurarlo con la siguiente orden:

```
git config --global core.excludesfile $HOME/.gitignore_global
```

Ahora podemos crear un archivo llamado .gitignore_global en la raíz de nuestra cuenta con este contenido:

```
# Compiled source #

###################

*.com

*.class

*.dll

*.exe

*.o

*.so


# Packages #

############

# it's better to unpack these files and commit the raw source

# git has its own built in compression methods

*.7z

*.dmg

*.gz

*.iso
```

```
*.jar
*.rar
*.tar
*.zip

# Logs and databases #
######################
*.log
*.sql
*.sqlite

# OS generated files #
######################
.DS_Store
.DS_Store?
._*
.Spotlight-V100
.Trashes
ehthumbs.db
Thumbs.db
*~
*.swp

# IDEs               #
######################
.idea
.settings/
.classpath
.project
```

4.2 Trabajando con el historial

Observando los cambios

Con la orden *git log* podemos ver todos los cambios que hemos hecho:

```
$ git log
commit fd4da946326fbe8b24e89282ad25a71721bf40f6
Author: Sergio Gómez <sergio@uco.es>
Date:   Sun Jun 16 12:51:01 2013 +0200

    Se añade un comentario al cambio del valor por defecto

commit 3283e0d306c8d42d55ffcb64e456f10510df8177
Author: Sergio Gómez <sergio@uco.es>
Date:   Sun Jun 16 12:50:00 2013 +0200

    Se añade un parámetro por defecto

commit efc252e11939351505a426a6e1aa5bb7dc1dd7c0
Author: Sergio Gómez <sergio@uco.es>
Date:   Sun Jun 16 12:13:26 2013 +0200

    Parametrización del programa

commit e19f2c1701069d9d1159e9ee21acaa1bbc47d264
Author: Sergio Gómez <sergio@uco.es>
Date:   Sun Jun 16 11:55:23 2013 +0200

    Creación del proyecto
```

También es posible ver versiones abreviadas o limitadas, dependiendo de los parámetros:

```
$ git log --oneline
```

```
fd4da94 Se añade un comentario al cambio del valor por defecto

3283e0d Se añade un parámetro por defecto

efc252e Parametrización del programa

e19f2c1 Creación del proyecto

git log --oneline --max-count=2

git log --oneline --since='5 minutes ago'

git log --oneline --until='5 minutes ago'

git log --oneline --author=sergio

git log --oneline --all
```

Una versión muy útil de git log es la siguiente, pues nos permite ver en que lugares está master y HEAD, entre otras cosas:

```
$ git log --pretty=format:'%h %ad | %s%d [%an]' --graph --date=short

* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto
(HEAD, master) [Sergio Gómez]

* 3283e0d 2013-06-16 | Se añade un parámetro por defecto [Sergio Gómez]

* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]

* e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]
```

Crear alias

Como estas órdenes son demasiado largas, Git nos permite crear alias para crear nuevas órdenes parametrizadas. Para ello, podemos configurar nuestro entorno con la orden *git config* de la siguiente manera:

```
git config --global alias.hist "log --pretty=format:'%h %ad | %s%d [%an]' --graph --
date=short"
```

Puedes configurar incluso alias para abreviar comandos. Algunos ejemplos de alias útiles:

```
git config --global alias.br branch

git config --global alias.co checkout

git config --global alias.ci commit

git config --global alias.st "status -u"

git config --global alias.cane "commit --amend --no-edit"
```

4.3 Recuperando versiones anteriores

Cada cambio es etiquetado por un hash, para poder regresar a ese momento del estado del proyecto se usa la orden *git checkout*.

```
$ git checkout e19f2c1

Note: checking out 'e19f2c1'.


You are in 'detached HEAD' state. You can look around, make experimental

changes and commit them, and you can discard any commits you make in this

state without impacting any branches by performing another checkout.


If you want to create a new branch to retain commits you create, you may

do so (now or later) by using -b with the checkout command again. Example:


  git checkout -b new_branch_name


HEAD is now at e19f2c1... Creación del proyecto

$ cat hola.php

<?php

echo "Hello, World\n";
```

El aviso que nos sale nos indica que estamos en un estado donde no trabajamos en ninguna rama concreta. Eso significa que los cambios que hagamos podrían "perderse" porque si no son guardados en una nueva rama, en principio no podríamos volver a recuperarlos. Hay que pensar que Git es como un árbol donde un nodo tiene información de su nodo padre, no de sus nodos hijos, con lo que siempre necesitaríamos información de dónde se encuentran los nodos finales o de otra manera no podríamos acceder a ellos.

Por lo tanto, si volvemos a un commit anterior de una rama y se hacen cambios y commits aquí, esos commits quedarán “huérfanos” (no en ninguna rama), a menos que creemos una nueva rama antes:

```
git checkout -b prueba-antigua
```

Volver a la última versión de la rama master.

Usamos git checkout indicando el nombre de la rama:

```
$ git checkout master

Previous HEAD position was e19f2c1... Creación del proyecto
```

Etiquetando versiones

Para poder recuperar versiones concretas en la historia del repositorio, podemos etiquetarlas, lo cual es más fácil que usar un hash. Para eso usaremos la orden git tag.

```
$ git tag v1
```

Ahora vamos a etiquetar la versión inmediatamente anterior como v1-beta. Para ello, podemos usar los modificadores ^ o ~ que nos llevarán a un ancestro determinado. Las siguientes dos órdenes son equivalentes:

```
$ git checkout v1^

$ git checkout v1~1

$ git tag v1-beta
```

Si ejecutamos la orden sin parámetros nos mostrará todas las etiquetas existentes.

```
$ git tag

v1

v1-beta
```

Y para verlas en el historial:

```
$ git hist master --all

* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto
(tag: v1, master) [Sergio Gómez]

* 3283e0d 2013-06-16 | Se añade un parámetro por defecto (HEAD, tag: v1-beta)
[Sergio Gómez]

* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]

* e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]
```

Borrar etiqueta

Para borrar etiquetas:

```
git tag -d nombre_etiqueta
```

Visualizar cambios

Para ver los cambios que se han realizado en el código usamos la orden *git diff*. La orden sin especificar nada más, mostrará los **cambios que no han sido *añadidos* aún**, es decir, todos los cambios que se han hecho antes de usar la orden git add. Después se puede indicar un parámetro y dará los cambios entre la versión indicada y el estado actual. O para comparar dos versiones entre sí, se indica la más antigua y la más nueva. Ejemplo:

```
$ git diff v1-beta v1

diff --git a/hola.php b/hola.php

index a31e01f..25a35c0 100644

--- a/hola.php

+++ b/hola.php

@@ -1,3 +1,4 @@

 <?php

+// El nombre por defecto es Mundo

 $nombre = isset($argv[1]) ? $argv[1] : "Mundo";

 @print "Hola, {$nombre}\n";
```

Existen más tipos de git diff

- *git diff* → Compara el Working Directory con el Staged Area. Es decir, cambios no agregados.
- *git diff –staged* → Compara el staging area con el último commit (HEAD). Es decir, cambios agregados.
- *git diff HEAD* → Compara el Working Directory con el último commit (HEAD). Es decir, todos los cambios sin commitear.

5. Uso Avanzado de git

5.1 Deshacer cambios

Deshaciendo cambios antes de la fase de staging.

Volvemos a la rama máster y vamos a modificar el comentario que pusimos:

```
$ git checkout master

Previous HEAD position was 3283e0d... Se añade un parámetro por defecto

Switched to branch 'master'
```

Modificamos *hola.php* de la siguiente manera:

```
<?php

// Este comentario está mal y hay que borrarlo

$nombre = isset($argv[1]) ? $argv[1] : "Mundo";

@print "Hola, {$nombre}\n";
```

Y comprobamos:

```
$ git status

# On branch master

# Changes not staged for commit:

#   (use "git add <file>..." to update what will be committed)

#   (use "git checkout -- <file>..." to discard changes in working directory)

#

#   modified:   hola.php

#

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

El mismo Git nos indica que debemos hacer para añadir los cambios o para deshacerlos. Lo que vamos a hacer es restaurar el archivo hola.php a su versión del último commit (HEAD).

```
$ git checkout hola.php

$ git status

# On branch master

nothing to commit, working directory clean

$ cat hola.php

<?php
```

```
// El nombre por defecto es Mundo

$nombre = isset($argv[1]) ? $argv[1] : "Mundo";

@print "Hola, {$nombre}\n";
```

Deshaciendo cambios antes del commit

Vamos a hacer lo mismo que la vez anterior, pero esta vez sí añadiremos el cambio al *staging* (sin hacer *commit*). Así que volvemos a modificar *hola.php* igual que la anterior ocasión:

```
<?php

// Este comentario está mal y hay que borrarlo

$nombre = isset($argv[1]) ? $argv[1] : "Mundo";

@print "Hola, {$nombre}\n";
```

Y lo añadimos al *staging:*

```
$ git add hola.php

$ git status

# On branch master

# Changes to be committed:

#   (use "git reset HEAD <file>..." to unstage)

#

#   modified:   hola.php

#
```

De nuevo, Git nos indica qué debemos hacer para deshacer el cambio:

```
$ git reset HEAD hola.php

Unstaged changes after reset:

M   hola.php

$ git status

# On branch master

# Changes not staged for commit:

#   (use "git add <file>..." to update what will be committed)

#   (use "git checkout -- <file>..." to discard changes in working directory)

#

#   modified:   hola.php
```

```
#

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

$ git checkout hola.php
```

Y ya tenemos nuestro repositorio limpio otra vez. Como vemos hay que hacerlo en dos pasos: uno para borrar los datos del *staging* y otro para restaurar la copia de trabajo.

Deshaciendo *commits* no deseados

Si a pesar de todo hemos hecho un *commit* y nos hemos equivocado, podemos deshacerlo con la orden *git revert*. Modificamos otra vez el archivo como antes:

```
<?php

// Este comentario está mal y hay que borrarlo

$nombre = isset($argv[1]) ? $argv[1] : "Mundo";

@print "Hola, {$nombre}\n";
```

Pero ahora sí hacemos *commit*:

```
$ git add hola.php

$ git commit -m "Ups... este commit está mal."

master 5a5d067] Ups... este commit está mal

 1 file changed, 1 insertion(+), 1 deletion(-)
```

Bien, una vez confirmado el cambio, vamos a deshacer el cambio con la orden git revert. Es importante comprender que git revert no borra commits anteriores, sino que crea un nuevo commit que anula el último:

```
$ git revert HEAD --no-edit

[master 817407b] Revert "Ups... este commit está mal"

1 file changed, 1 insertion(+), 1 deletion(-)

$ git hist

* 817407b 2013-06-16 | Revert "Ups... este commit está mal" (HEAD, master)
[Sergio Gómez]

* 5a5d067 2013-06-16 | Ups... este commit está mal [Sergio Gómez]

* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto
(tag: v1) [Sergio Gómez]

* 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta) [Sergio
Gómez]

* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
```

```
* e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]
```

Borrar commits de una rama

El anterior apartado revierte un commit, pero deja huella en el historial de cambios. Para hacer que no aparezca hay que usar la orden **git reset**.

```
$ git reset --hard v1

HEAD is now at fd4da94 Se añade un comentario al cambio del valor por defecto

$ git hist

* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto
(HEAD, tag: v1, master) [Sergio Góme

* 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta) [Sergio
Gómez]

* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]

* e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]
```

Lo que realmente hace es:

- Deshace todos los commits posteriores a v1[2]
- Borra los cambios que tengas en *staging* y en el *working directory*
- Deja tu repositorio exactamente como estaba en v1

> La orden *reset* es una operación delicada. Debe evitarse si no se sabe bien lo que se está haciendo, sobre todo cuando se trabaja en repositorios compartidos, porque podríamos alterar la historia de cambios lo cual puede provocar problemas de sincronización.

Como curiosidad, existen los siguientes tipos de git reset:

Modo	***Qué cambia***	***Qué conserva***	***Uso típico***
--soft	Solo mueve el HEAD	Mantiene staging y archivos	Deshacer un commit, pero mantener los cambios listos para volver a commitear
--mixed (defecto)	Mueve el head y limpia el staging	Mantiene los archivos (working dir)	Deshacer commit y quitar el staging, pero conservar el código modificado.
--hard	Mueve el head, limpiar staging y borra cambios del working dir	Nada se conserva	Volver todo exactamente a un commit anterior.

[2] El resto de los cambios no se han borrado (aún), simplemente no están accesibles porque git no sabe cómo referenciarlos. Si sabemos su hash podemos acceder aún a ellos. Pasado un tiempo, eventualmente Git tiene un recolector de basura que los borrará. Se puede evitar etiquetando el estado final.

Modificar un commit

Esto se usa cuando hemos olvidado añadir un cambio a un commit que acabamos de realizar. Tenemos nuestro archivo *hola.php* de la siguiente manera:

```
<?php

// Autor: Sergio Gómez

// El nombre por defecto es Mundo

$nombre = isset($argv[1]) ? $argv[1] : "Mundo";

@print "Hola, {$nombre}\n";
```

Y lo confirmamos. El parámetro -a hace un git add antes de hacer *commit* de todos los archivos modificados o borrados (de los nuevos no), con lo que nos ahorramos un paso.

```
$ git commit -a -m "Añadido el autor del programa"

[master cf405c1] Añadido el autor del programa

 1 file changed, 1 insertion(+)
```

Ahora nos percatamos que se nos ha olvidado poner el correo electrónico. Así que volvemos a modificar nuestro archivo:

```
<?php

// Autor: Sergio Gómez <sergio@uco.es>

// El nombre por defecto es Mundo

$nombre = isset($argv[1]) ? $argv[1] : "Mundo";

@print "Hola, {$nombre}\n";
```

Y en esta ocasión usamos commit --amend que nos permite modificar el último estado confirmado, sustituyéndolo por el estado actual:

```
$ git add hola.php

$ git commit --amend -m "Añadido el autor del programa y su email"

[master 96a39df] Añadido el autor del programa y su email

 1 file changed, 1 insertion(+)

$ git hist

* 96a39df 2013-06-16 | Añadido el autor del programa y su email (HEAD, master)
[Sergio Gómez]

* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto
(tag: v1) [Sergio Gómez]

* 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta) [Sergio
Gómez]
```

```
* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]

* e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]
```

Nunca modifiques un *commit* que ya hayas sincronizado con otro repositorio o que hayas recibido de él. Estarías alterando la historia de cambios y provocarías problemas de sincronización.

5.2 Moviendo y borrando archivos

Mover un archivo a otro directorio con git

```
Para mover archivos usaremos la orden git mv:

$ mkdir lib

$ git mv hola.php lib

$ git status

# On branch master

# Changes to be committed:

#   (use "git reset HEAD <file>..." to unstage)

#

#   renamed:    hola.php -> lib/hola.php

#
```

Mover y borrar archivos.

Podíamos haber hecho el paso anterior con la órden del sistema *mv* y el resultado hubiera sido el mismo. Lo siguiente es a modo de ejemplo y no es necesario que lo ejecutes:

```
$ mkdir lib

$ mv hola.php lib

$ git add lib/hola.php

$ git rm hola.php
```

Y, ahora sí, ya podemos guardar los cambios:

```
$ git commit -m "Movido hola.php a lib."

[master 8c2a509] Movido hola.php a lib.

 1 file changed, 0 insertions(+), 0 deletions(-)

 rename hola.php => lib/hola.php (100%)
```

6. Ramas

6.1 Administración de Ramas

Crear una nueva rama

Cuando vamos a trabajar en una **nueva funcionalidad**, es conveniente hacerlo en una nueva rama, para no modificar la rama principal y dejarla inestable. Aunque la orden para manejar ramas es *git branch* podemos usar también *git checkout*.

Vamos a crear una nueva rama:

```
git branch hola
```

Si usamos git branch sin ningún argumento, nos devolverá la lista de ramas disponibles.

La orden anterior no devuelve ningún resultado y tampoco nos cambia de rama, para eso debemos usar *checkout*:

```
$ git checkout hola

Switched to branch 'hola'
```

Hay una forma más rapida de hacer ambas acciones en un solo paso. Con el parámetro -b de git checkout podemos cambiarnos a una rama que, si no existe, se crea instantáneamente.

```
$ git checkout -b hola

Switched to a new branch 'hola'
```

Modificaciones en la rama secundaria

Añadimos un nuevo archivo en el directorio lib llamado HolaMundo.php:

```
<?php


class HolaMundo

{

   private $nombre;


   function __construct($nombre)

   {

      $this->nombre = $nombre;
```

```
    }


    function __toString()
    {
        return sprintf ("Hola, %s.\n", $this->nombre);
    }
}
```

Y modificamos hola.php:

```
<?php
// Autor: Sergio Gómez <sergio@uco.es>
// El nombre por defecto es Mundo
require('HolaMundo.php');


$nombre = isset($argv[1]) ? $argv[1] : "Mundo";
print new HolaMundo($nombre);
```

Podríamos confirmar los cambios todos de golpe, pero lo haremos de uno en uno, con su comentario.

```
$ git add lib/HolaMundo.php
$ git commit -m "Añadida la clase HolaMundo"
[hola 6932156] Añadida la clase HolaMundo
 1 file changed, 16 insertions(+)
 create mode 100644 lib/HolaMundo.php
$ git add lib/hola.php
$ git commit -m "hola usa la clase HolaMundo"
[hola 9862f33] hola usa la clase HolaMundo
 1 file changed, 3 insertions(+), 1 deletion(-)
```

Y ahora con la orden git checkout podemos movernos entre ramas:

```
$ git checkout master
Switched to branch 'master'
$ git checkout hola
Switched to branch 'hola'
```

Modificaciones en la rama master

Podemos volver y añadir un nuevo archivo a la rama principal:

```
$ git checkout master

Switched to branch 'master'
```

Creamos un archivo llamado README.md en la raíz de nuestro proyecto con el siguiente contenido:

```
# Curso de GIT

Este proyecto contiene el curso de introducción a GIT
```

Y lo añadimos a nuestro repositorio en la rama en la que estamos:

```
$ git add README.md

$ git commit -m "Añadido README.md"

[master c3e65d0] Añadido README.md

 1 file changed, 3 insertions(+)

 create mode 100644 README.md

$ git hist --all

* c3e65d0 2013-06-16 | Añadido README.md (HEAD, master) [Sergio Gómez]

| * 9862f33 2013-06-16 | hola usa la clase HolaMundo (hola) [Sergio Gómez]

| * 6932156 2013-06-16 | Añadida la clase HolaMundo [Sergio Gómez]

|/

* 81c6e93 2013-06-16 | Movido hola.php a lib [Sergio Gómez]

* 96a39df 2013-06-16 | Añadido el autor del programa y su email [Sergio
Gómez]

* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto
(tag: v1) [Sergio Gómez]

* 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta)
[Sergio Gómez]

* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]

* e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]
```

Y vemos como git hist muestra la bifurcación en nuestro código.

6.2 Fusión de ramas y resolución de conflictos

Mezclar ramas

Podemos incorporar los cambios de una rama a otra con la orden git merge

```
$ git checkout hola

Switched to branch 'hola'

$ git merge master

Merge made by the 'recursive' strategy.

 README.md | 3 +++

 1 file changed, 3 insertions(+)

 create mode 100644 README.md

$ git hist --all

*   9c6ac06 2013-06-16 | Merge commit 'c3e65d0' into hola (HEAD, hola)
[Sergio Gómez]

|\

* | 9862f33 2013-06-16 | hola usa la clase HolaMundo [Sergio Gómez]

* | 6932156 2013-06-16 | Añadida la clase HolaMundo [Sergio Gómez]

| |

| * c3e65d0 2013-06-16 | Añadido README.md [Sergio Gómez]

|/

* 81c6e93 2013-06-16 | Movido hola.php a lib [Sergio Gómez]

* 96a39df 2013-06-16 | Añadido el autor del programa y su email [Sergio
Gómez]

* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto
(tag: v1) [Sergio Gómez]

* 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta)
[Sergio Gómez]

* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]

* e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]
```

De esa forma se puede trabajar en una rama secundaria incorporando los cambios de la rama principal o de otra rama.

Resolver conflictos

Un conflicto es cuando se produce una fusión que Git no es capaz de resolver. Vamos a modificar la rama master para crear uno con la rama hola.

```
$ git checkout master

Switched to branch 'master'
```

Modificamos nuestro archivo *hola.php* de nuevo:

```
<?php

// Autor: Sergio Gómez <sergio@uco.es>

print "Introduce tu nombre:";

$nombre = trim(fgets(STDIN));

@print "Hola, {$nombre}\n";
```

Y guardamos los cambios:

```
$ git add lib/hola.php

$ git commit -m "Programa interactivo"

[master 9c85275] Programa interactivo

 1 file changed, 2 insertions(+), 2 deletions(-)

$ git hist --all

*   9c6ac06 2013-06-16 | Merge commit 'c3e65d0' into hola (hola) [Sergio
Gómez]

|\

* | 9862f33 2013-06-16 | hola usa la clase HolaMundo [Sergio Gómez]

* | 6932156 2013-06-16 | Añadida la clase HolaMundo [Sergio Gómez]

| | * 9c85275 2013-06-16 | Programa interactivo (HEAD, master) [Sergio Gómez]

| |/

| * c3e65d0 2013-06-16 | Añadido README.md [Sergio Gómez]

|/

* 81c6e93 2013-06-16 | Movido hola.php a lib [Sergio Gómez]

* 96a39df 2013-06-16 | Añadido el autor del programa y su email [Sergio
Gómez]

* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto
(tag: v1) [Sergio Gómez]

* 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta)
[Sergio Gómez]
```

```
* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]

* e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]
```

Volvemos a la rama hola y fusionamos:

```
$ git checkout hola

Switched to branch 'hola'

$ git merge master

Auto-merging lib/hola.php

CONFLICT (content): Merge conflict in lib/hola.php

Automatic merge failed; fix conflicts and then commit the result.
```

Si editamos nuestro archivo lib/hola.php obtendremos algo similar a esto:

```
<?php

// Autor: Sergio Gómez <sergio@uco.es>

<<<<<<< HEAD

// El nombre por defecto es Mundo

require('HolaMundo.php');


$nombre = isset($argv[1]) ? $argv[1] : "Mundo";

print new HolaMundo($nombre);

=======

print "Introduce tu nombre:";

$nombre = trim(fgets(STDIN));

@print "Hola, {$nombre}\n";

>>>>>>> master
```

La primera parte marca el código que estaba en la rama donde trabajábamos (HEAD) y la parte final el código de donde fusionábamos. Resolvemos el conflicto, dejando el archivo como sigue:

```
<?php

// Autor: Sergio Gómez <sergio@uco.es>

require('HolaMundo.php');


print "Introduce tu nombre:";

$nombre = trim(fgets(STDIN));
```

```
print new HolaMundo($nombre);
```

Y resolvemos el conflicto confirmando los cambios:

```
$ git add lib/hola.php

$ git commit -m "Solucionado el conflicto al fusionar con la rama master"

[hola a36af04] Solucionado el conflicto al fusionar con la rama master
```

Rebasing vs Merging

Rebasing es otra técnica para fusionar distinta a merge y usa la orden git rebase. Vamos a dejar nuestro proyecto como estaba antes del fusionado. Para ello necesitamos anotar el hash anterior al de la acción de *merge*. El que tiene la anotación *"hola usa la clase HolaMundo"*.

Para ello, podemos usar la orden `git reset` que nos permite mover HEAD donde queramos.

```
$ git checkout hola

Switched to branch 'hola'

$ git hist

*   a36af04 2013-06-16 | Solucionado el conflicto al fusionar con la rama master (HEAD, hola) [Sergio Gómez]

|\

| * 9c85275 2013-06-16 | Programa interactivo (master) [Sergio Gómez]

* |   9c6ac06 2013-06-16 | Merge commit 'c3e65d0' into hola [Sergio Gómez]

|\ \

| |/

| * c3e65d0 2013-06-16 | Añadido README.md [Sergio Gómez]

* | 9862f33 2013-06-16 | hola usa la clase HolaMundo [Sergio Gómez]

* | 6932156 2013-06-16 | Añadida la clase HolaMundo [Sergio Gómez]

|/

* 81c6e93 2013-06-16 | Movido hola.php a lib [Sergio Gómez]

* 96a39df 2013-06-16 | Añadido el autor del programa y su email [Sergio Gómez]

* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto (tag: v1) [Sergio Gómez]

* 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta) [Sergio Gómez]

* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]

* e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]
```

```
$ git reset --hard 9862f33

HEAD is now at 9862f33 hola usa la clase HolaMundo
```

Y nuestro estado será:

```
$ git hist --all

* 9862f33 2013-06-16 | hola usa la clase HolaMundo (HEAD, hola) [Sergio Gómez]

* 6932156 2013-06-16 | Añadida la clase HolaMundo [Sergio Gómez]

| * 9c85275 2013-06-16 | Programa interactivo (master) [Sergio Gómez]

| * c3e65d0 2013-06-16 | Añadido README.md [Sergio Gómez]

|/

* 81c6e93 2013-06-16 | Movido hola.php a lib [Sergio Gómez]

* 96a39df 2013-06-16 | Añadido el autor del programa y su email [Sergio Gómez]

* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto (tag: v1)
[Sergio Gómez]

* 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta) [Sergio Gómez]

* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]

* e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]
```

Hemos desecho todos los *merge* y nuestro árbol está *"limpio"*. Vamos a probar ahora a hacer un rebase. Continuamos en la rama hola y ejecutamos lo siguiente:

```
$ git rebase master

First, rewinding head to replay your work on top of it...

Applying: Añadida la clase HolaMundo

Applying: hola usa la clase HolaMundo

Using index info to reconstruct a base tree...

M   lib/hola.php

Falling back to patching base and 3-way merge...

Auto-merging lib/hola.php

CONFLICT (content): Merge conflict in lib/hola.php

error: Failed to merge in the changes.

Patch failed at 0002 hola usa la clase HolaMundo

The copy of the patch that failed is found in: .git/rebase-apply/patch


When you have resolved this problem, run "git rebase --continue".
```

```
If you prefer to skip this patch, run "git rebase --skip" instead.

To check out the original branch and stop rebasing, run "git rebase --abort".
```

El conflicto, por supuesto, se sigue dando. Resolvemos guardando el archivo hola.php como en los casos anteriores:

```
<?php

// Autor: Sergio Gómez <sergio@uco.es>

require('HolaMundo.php');


print "Introduce tu nombre:";

$nombre = trim(fgets(STDIN));

print new HolaMundo($nombre);
```

Añadimos los cambios en *staging* y en esta ocasión, y tal como nos indicaba en el mensaje anterior, no tenemos que hacer git commit sino continuar con el *rebase*:

```
$ git add lib/hola.php

$ git status

rebase in progress; onto 269eaca

You are currently rebasing branch 'hola' on '269eaca'.

  (all conflicts fixed: run "git rebase --continue")


Changes to be committed:

  (use "git reset HEAD <file>..." to unstage)


    modified:   lib/hola.php

$ git rebase --continue

Applying: hola usa la clase HolaMundo
```

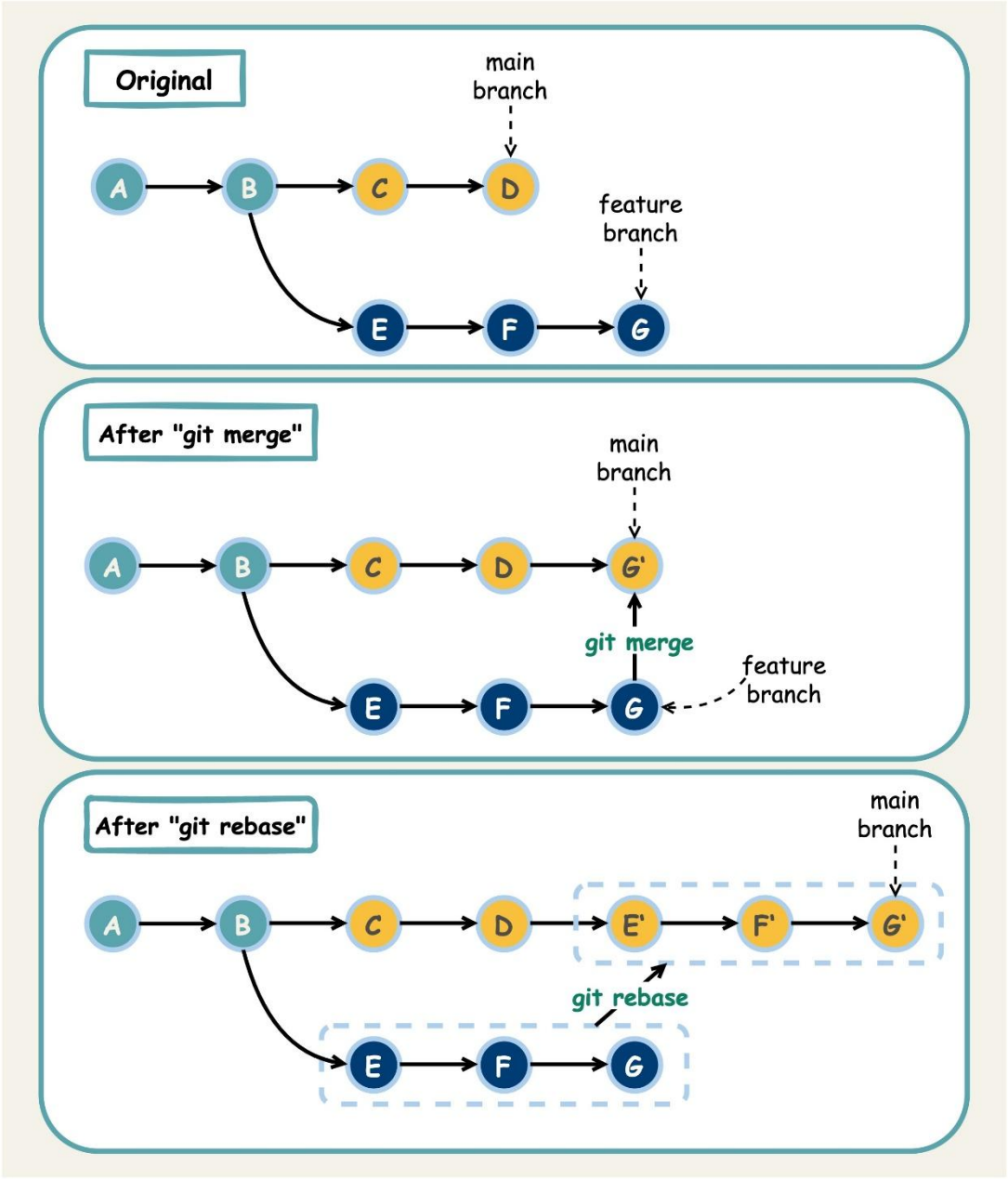
Y ahora vemos que nuestro árbol tiene un aspecto distinto, mucho más limpio:

```
$ git hist --all

* 9862f33 2013-06-16 | hola usa la clase HolaMundo (HEAD -> hola) [Sergio Gómez]

* 6932156 2013-06-16 | Añadida la clase HolaMundo [Sergio Gómez]

* 9c85275 2013-06-16 | Programa interactivo (master) [Sergio Gómez]

* c3e65d0 2013-06-16 | Añadido README.md [Sergio Gómez]

* 81c6e93 2013-06-16 | Movido hola.php a lib [Sergio Gómez]

* 96a39df 2013-06-16 | Añadido el autor del programa y su email [Sergio Gómez]

* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto (tag:
v1) [Sergio Gómez]

* 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta) [Sergio
Gómez]

* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]

* e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]
```

Lo que hace rebase es volver a aplicar todos los cambios a la rama master, desde su nodo más reciente. Eso significa que se modifica el orden o la historia de creación de los cambios. Por eso rebase no debe usarse si el orden es importante o si la rama es compartida.

Git Merge vs. Git Rebase

blog.bytebytego.com



6.3 Mezclando con la rama master

Ya hemos terminado de implementar los cambios en nuestra rama secundaria y es hora de llevar los cambios a la rama principal. Usamos git merge para hacer una fusión normal:

```
$ git checkout master

Switched to branch 'master'

$ git merge hola

Updating c3e65d0..491f1d2

Fast-forward

 lib/HolaMundo.php | 16 ++++++++++++++++

 lib/hola.php      |  4 +++-

 2 files changed, 19 insertions(+), 1 deletion(-)

 create mode 100644 lib/HolaMundo.php

 $ git hist --all

 * 9862f33 2013-06-16 | hola usa la clase HolaMundo (HEAD -> master, hola) [Sergio Gómez]

 * 6932156 2013-06-16 | Añadida la clase HolaMundo [Sergio Gómez]

 * 9c85275 2013-06-16 | Programa interactivo [Sergio Gómez]

 * c3e65d0 2013-06-16 | Añadido README.md [Sergio Gómez]

 * 81c6e93 2013-06-16 | Movido hola.php a lib [Sergio Gómez]

 * 96a39df 2013-06-16 | Añadido el autor del programa y su email [Sergio Gómez]

 * fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto (tag: v1) [Sergio Gómez]

 * 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta) [Sergio Gómez]

 * efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]

 * e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]
```

Vemos que indica que el tipo de fusión es *fast-forward*. Este tipo de fusión tiene el problema que no deja rastro de la fusión, por eso suele ser recomendable usar el parámetro --no-ff para que quede constancia siempre de que se ha fusionado una rama con otra.

Vamos a volver a probar ahora sin hacer *fast-forward*. Reseteamos *master* al estado *"Programa interactivo"*.

```
$ git reset --hard 9c85275

$ git hist --all
```

```
* 9862f33 2013-06-16 | hola usa la clase HolaMundo (HEAD -> hola) [Sergio Gómez]

* 6932156 2013-06-16 | Añadida la clase HolaMundo [Sergio Gómez]

* 9c85275 2013-06-16 | Programa interactivo (master) [Sergio Gómez]

* c3e65d0 2013-06-16 | Añadido README.md [Sergio Gómez]

* 81c6e93 2013-06-16 | Movido hola.php a lib [Sergio Gómez]

* 96a39df 2013-06-16 | Añadido el autor del programa y su email [Sergio Gómez]

* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto (tag: v1)
[Sergio Gómez]

* 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta) [Sergio Gómez]

* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]

* e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]
```

Vemos que estamos como en el final de la sección anterior, así que ahora mezclamos:

```
$ git merge -m "Aplicando los cambios de la rama hola" --no-ff hola

Merge made by the 'recursive' strategy.

 lib/HolaMundo.php | 16 ++++++++++++++++

 lib/hola.php      |  4 +++-

 2 files changed, 19 insertions(+), 1 deletion(-)

 create mode 100644 lib/HolaMundo.php

$ git hist --all

*   2eab8ca 2013-06-16 | Aplicando los cambios de la rama hola (HEAD -> master) [Sergio
Gomez]

*\

| * 9862f33 2013-06-16 | hola usa la clase HolaMundo (hola) [Sergio Gómez]

| * 6932156 2013-06-16 | Añadida la clase HolaMundo [Sergio Gómez]

|/

* 9c85275 2013-06-16 | Programa interactivo (master) [Sergio Gómez]

* c3e65d0 2013-06-16 | Añadido README.md [Sergio Gómez]

* 81c6e93 2013-06-16 | Movido hola.php a lib [Sergio Gómez]
```

```
* 96a39df 2013-06-16 | Añadido el autor del programa y su email [Sergio Gómez]

* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por defecto (tag: v1) [Sergio Gómez]

* 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta) [Sergio Gómez]

* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]

* e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]
```

En la siguiente imagen se puede ver la diferencia: