# Introducción

El siguiente documento es un tutorial o manual hecho por miembros del grupo para contar con un manual de referencia para usar el repositorio elegido (Git).

# Índice

[1. ¿Qué es Git?](#h.m9nxas11iv43)

[1.1. Sistema de control de versiones (SCV)](#h.ff5ew09yzgh9)

[1.2. Sistema centralizado vs. sistema distribuido](#h.62lqme7sv2bv)

[1.3. Los tres estados](#h.w7y84omxmsv2)

[2. Guía de instalación de Git](#h.5exle7oxbd8o)

[2.1. Windows](#h.e9zy6z805u2s)

[2.2. Linux](#h.z6slpfw31j85)

[3. Configuración básica inicial](#h.n2dvtiwmc8sn)

[3.1. Identificación](#h.y4pjdix1rpjh)

[3.2. Clave RSA](#h.ahpl5li5vkzh)

[3.2.1. Windows](#h.t9i02f517xsa)

[3.3.2. Linux](#h.euf1zjkd141n)

[4. Acciones básicas en Git](#h.waqv2x3xzok3)

[4.1. Añadir ficheros al índice (add)](#h.yi8jjxvvb85k)

[4.2. Borrar ficheros (rm)](#h.7nqsyzkn0r4t)

[4.3. Comprobar el estado de los ficheros (status)](#h.c62bizryc275)

[4.4. Guardar los cambios en el repositorio local (commit)](#h.x9p6ld2g2e05)

[4.5. Deshacer cambios (reset)](#h.g1k2jn7rhsrg)

[4.6. Subir cambios al repositorio remoto (push)](#h.imnp9d106ner)

[4.7. Bajar cambios del repositorio remoto (pull)](#h.8p9iu5o0gy3c)

[4.8. Trabajo con ramas (checkout)](#h.oblzcpgct320)

[4.9. Juntar la rama actual con otra (merge)](#h.dqgrzo1f21sn)

[4.10. Modificar el punto de partida de una rama (rebase)](#h.rk1tm4cw69rg)

[4.11. Más información](#h.vu0thlpvgwb2)

[5. Formas de trabajo](#h.fuc14mwlo1nn)

[5.1. Ramas remotas](#h.6g2iiujeer8w)

[5.2. Ramas locales](#h.w3un2qg08l64)

[5.3. Commits organizados](#h.aecrfpxzd9xi)

[5.4. Ramas y niveles de estabilidad](#h.galpcq8szsi)

# 1. ¿Qué es Git?

## 1.1. Sistema de control de versiones (SCV)

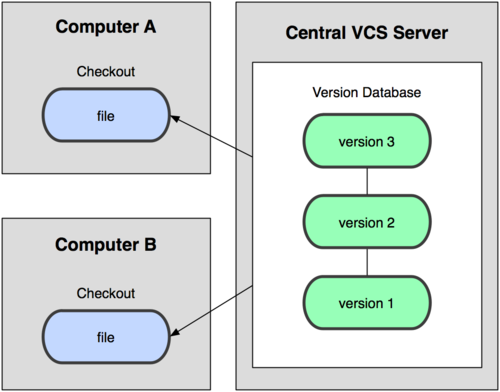
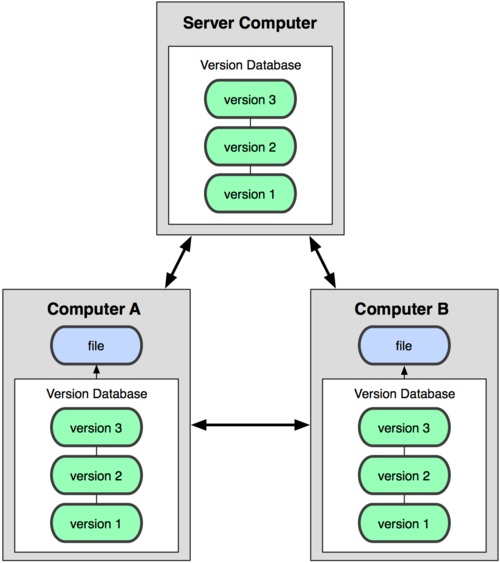
El control de versiones es un sistema que registra los cambios en un conjunto de archivos a través del tiempo para que pueda recuperar versiones específicas más tarde. Esto es Git exactamente. Gracias a esto podremos trabajar de forma concurrente sin preocuparnos de estropear el proyecto o el trabajo de los demás.

## 1.2. Sistema centralizado *vs.* sistema distribuido

Existen dos tipos de SCV que se diferencian en la forma en la que almacenan los archivos: sistemas centralizados y sistemas distribuidos.

* Sistemas **centralizados**: Utilizan un único servidor en el que se almacenan todos los archivos versionados. Todos los usuarios deben acceder al servidor para trabajar con los ficheros, puesto que solamente se encuentran ahí. Si la conexión con el servidor se perdiera o el usuario no tuviera acceso a internet, ningún cambio podrá registrarse, y se deberá esperar a tener acceso de nuevo. Acceder al servidor conlleva además una cierta latencia en las operaciones.

* Sistemas **distribuidos**: Cada usuario posee una copia del repositorio en su propia máquina. El concepto de servidor central no existe desde el punto de vista del SCV, aunque comúnmente se establece un servidor remoto como *central* para poder coordinar a un cierto equipo. En caso de no disponer de una conexión con el servidor, el usuario puede realizar cambios localmente y *subirlos* más adelante, sin que esto afecte a su forma de trabajar. Además, como todos los usuarios disponen de una copia del repositorio, en caso de pérdida de datos en el servidor, el proyecto es fácil de recuperar.

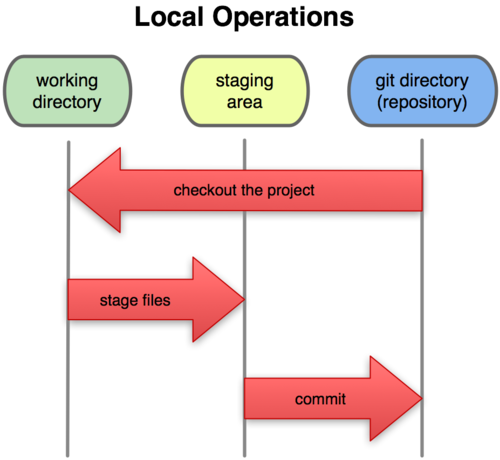
*Diagrama de un sistema centralizado (izquierda) y un sistema distribuido (derecha).*

## 1.3. Los tres estados

Cada archivo gestionado mediante Git puede estar en uno de los siguientes estados:

* **Commited**: Fichero almacenado completamente en el repositorio en su última versión
* **Modified**: Fichero modificado en el directorio de trabajo, de cuyos cambios Git aún no sabe nada.
* **Staged**: Fichero modificado en el directorio de trabajo, cuyos cambios han sido añadidos al *índice* (staging area) para formar parte del siguiente *commit*.

En ocasiones un fichero puede encontrarse a la vez en los estados *modified* y *staged* si sólo hemos añadido al índice parte del fichero o si lo hemos modificado tras añadirlo.



*Los tres estados.*

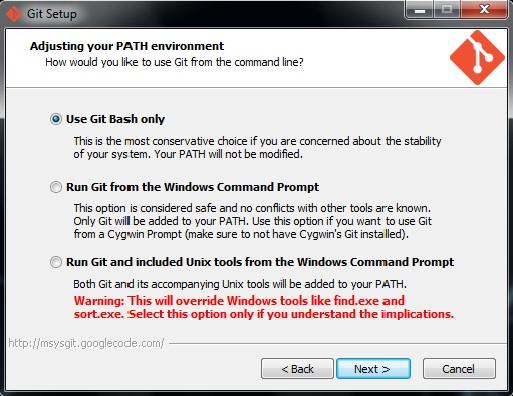
# 2. Guía de instalación de Git

## 2.1. Windows

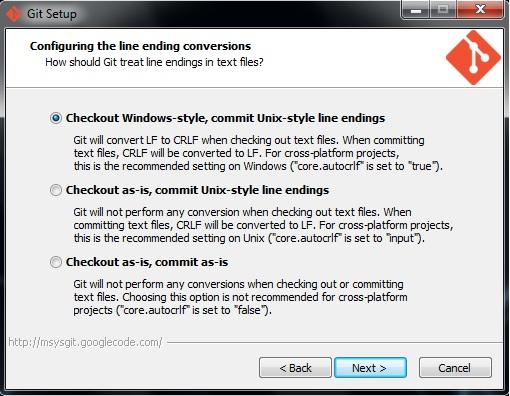
En primer lugar, descargamos Git del siguiente enlace: <http://git-scm.com/download/win>

Una vez descargado, ejecutamos el instalador y personalizamos la instalación. Los puntos donde tenemos elección son los siguientes:

* + Cómo usar Git desde la línea de comandos: Podemos elegir si queremos usar Git únicamente desde Bash (primera opción), si queremos poder utilizarlo también desde la consola de Windows (segunda opción), y si además queremos instalar utilidades Unix extra (tercera opción). Para no complicar las cosas, seleccionamos la primera opción.
* Cómo se tratan los finales de línea en los ficheros de texto: Debido a que Windows utiliza una combinación de caracteres para representar el final de línea diferente a la de los sistemas POSIX como Linux o Mac, Git debe saber cómo tratarlos. Aquí podemos elegir entre tres opciones, de las cuales nos interesa la primera. Git convertirá todos los saltos al formato de Windows en el directorio de trabajo, y al formato de POSIX en el repositorio. De esta forma el repositorio trabajará de forma uniforme con el formato POSIX, pero cada uno usará al editar los ficheros el formato que mejor le convenga.



*Ajuste de la terminal en la que se ejecuta Git.*



*Configuración del salto de línea.*

## 2.2. Linux

La instalación en Linux es muy simple, tan solo necesitamos un comando. Dependiendo de la distribución usada, el comando es diferente. En el siguiente enlace está la lista con todos los comandos: <http://git-scm.com/download/linux>

# 3. Configuración básica inicial

## 3.1. Identificación

Para que nuestros *commits* aparezcan con nuestra información, debemos configurar en Git nuestro nombre de usuario y correo electrónico:

|  |
| --- |
| **$ git config --global user.name "Your Name"**  **$ git config --global user.email** [**email@gmail.com**](mailto:email@gmail.com) |

## 3.2. Clave RSA

Para poder trabajar con GitHub sin tener que introducir nuestro usuario y contraseña cada vez que realicemos un *push* o *pull*, debemos configurar una clave RSA que nos identificará.

### 3.2.1. Windows

Generamos una nueva clave RSA:

|  |
| --- |
| **$ ssh-keygen -t rsa -C "your\_email@youremail.com" (/c/Users/you/.ssh/id\_rsa): #Generating public/private rsa key pair. Enter file in which to save the key [Press enter]** |

Tras esto nos pedirá una contraseña. Podemos elegir dejarla en blanco, aunque no es recomendable.

|  |
| --- |
| **Enter passphrase (empty for no passphrase): [Type a passphrase] Enter same passphrase again: [Type passphrase again]** |

Si todo va bien, debería salir algo parecido a esto

|  |
| --- |
| **Your identification has been saved in /c/Users/you/.ssh/id\_rsa. Your public key has been saved in /c/Users/you/.ssh/id\_rsa.pub. The key fingerprint is: #01:0f:f4:3b:ca:85:d6:17:a1:7d:f0:68:9d:f0:a2:db your\_email@youremail.com** |

A continuación introducimos el siguiente comando, que copiará al portapapeles la clave pública:

|  |
| --- |
| **$ clip < ~/.ssh/id\_rsa.pub** |

Ahora hay que añadir la clave en nuestra cuenta de GitHub. Para ello nos dirigimos a la configuración de nuestra cuenta, y en la sección *SSH Keys* damos al botón de añadir nueva clave y pegamos la clave. El nombre de la clave es irrelevante.

Comprobamos que todo es correcto:

|  |
| --- |
| **$ ssh -T git@github.com** |

Aparecerá el siguiente mensaje:

|  |
| --- |
| **The authenticity of host 'github.com (207.97.227.239)' can't be established.RSA key fingerprint is 16:27:ac:a5:76:28:2d:36:63:1b:56:4d:eb:df:a6:48. Are you sure you want to continue connecting (yes/no)? [Type yes]** |

Finalmente aparecerá este mensaje, que nos indica la cuenta con la que acabamos de identificarnos.

### 3.3.2. Linux

Generamos una nueva clave RSA:

|  |
| --- |
| **$ ssh-keygen -t rsa -C "your\_email@youremail.com" (/c/Users/you/.ssh/id\_rsa): #Generating public/private rsa key pair. Enter file in which to save the key [Press enter]** |

Tras esto nos pedirá una contraseña. Podemos elegir dejarla en blanco, aunque no es recomendable.

|  |
| --- |
| **Enter passphrase (empty for no passphrase): [Type a passphrase] Enter same passphrase again: [Type passphrase again]** |

Si todo va bien, debería salir algo parecido a esto.

|  |
| --- |
| **Your identification has been saved in ~/.ssh/id\_rsa. Your public key has been saved in ~/.ssh/id\_rsa.pub. The key fingerprint is: #01:0f:f4:3b:ca:85:d6:17:a1:7d:f0:68:9d:f0:a2:db your\_email@youremail.com** |

A continuación introducimos los siguientes, que copiará al portapapeles la clave pública:

|  |
| --- |
| **$ sudo apt-get install xclip #Downloads and installs xclip $ xclip -sel clip < ~/.ssh/id\_rsa.pub #Copies the contents of the id\_rsa.pub file to your clipboard** |

Ahora hay que añadir la clave en nuestra cuenta de GitHub. Para ello nos dirigimos a la configuración de nuestra cuenta, y en la sección *SSH Keys* damos al botón de añadir nueva clave y pegamos la clave. El nombre de la clave es irrelevante.

Comprobamos que todo es correcto:

|  |
| --- |
| **$ ssh -T git@github.com** |

Aparecerá el siguiente mensaje:

The authenticity of host 'github.com (207.97.227.239)' can't be established.RSA key fingerprint is 16:27:ac:a5:76:28:2d:36:63:1b:56:4d:eb:df:a6:48.  
Are you sure you want to continue connecting (yes/no)? [Type yes]

Finalmente aparecerá este mensaje, que nos indica la cuenta con la que acabamos de identificarnos.

# 4. Acciones básicas en Git

A continuación se explican los comandos básicos de Git y su uso. Para cualquier duda podemos consultar la ayuda de git, mediante git help <comando> o git <comando> --help.

## 4.1. Añadir ficheros al índice (add)

Para poder realizar *commits*, antes debemos añadir las modificaciones al índice (*staging area*). Para ello utilizaremos el comando git add <ficheros>. Como parámetros pasamos los nombres de los ficheros o directorios que queremos añadir.

Una versión muy útil de este comando es git add -u <ficheros>, que añadirá todos los cambios realizados en los ficheros de los que Git ya tenga algún registro.

Git incluye una herramienta para añadir ficheros al índice de forma interactiva, usable desde git add -i <ficheros>. Su uso puede consultarse aquí: <http://git-scm.com/book/en/Git-Tools-Interactive-Staging>

## 4.2. Borrar ficheros (rm)

Tanto si deseamos borrar un fichero de un repositorio como si queremos que un fichero que ha desaparecido de nuestro directorio de trabajo figure como eliminado, necesitamos usar el comando git rm <ficheros>.

Este comando eliminará el fichero si existe y lo marcará como borrado en el repositorio. Si lo que queremos es quitar un fichero del repositorio conservando la versión existente en el directorio de trabajo, debemos usar la versión git rm --cached <ficheros>.

## 4.3. Comprobar el estado de los ficheros (status)

Antes de realizar cualquier acción sobre el repositorio, es conveniente saber previamente en qué estado se encuentra. Para ello utilizaremos el comando git status o su versión de listado reducido git status -s.

Este comando nos informará de todos aquellos ficheros que no estén presentes o hayan sufrido cambios desde el último *commit*. Los ficheros actualizados no aparecerán en el listado de este comando.

## 4.4. Guardar los cambios en el repositorio local (commit)

El centro de todo el sistema de versiones es el *commit*, el cual podemos realizar mediante el comando git commit. Un *commit* almacenará en el repositorio todos los cambios del índice introducidos con otros comandos (particularmente con git add), y los almacenará en forma de *delta*, es decir, modificaciones desde el último *commit*.

Existen algunas opciones dignas de mención para este comando:

* + -m "mensaje": Se usará el mensaje indicado como mensaje de *commit*. Si no incluimos esta opción, Git lanzará el editor por defecto (vim en Windows).
  + -a: Añade automáticamente todos los ficheros de los que Git tiene registro al índice antes de realizar el *commit*. Esta opción equivale a ejecutar el comando git add -u justo antes de realizar el *commit*.

## 4.5. Deshacer cambios (reset)

Este comando es demasiado grande y complejo como para que os lo explique porque, sinceramente, no termino de saber qué narices hace. Consultad la página del manual (<http://www.kernel.org/pub/software/scm/git/docs/git-reset.html>) y la sección de Pro Git (<http://git-scm.com/book/en/Git-Basics-Undoing-Things>).

## 4.6. Subir cambios al repositorio remoto (push)

Para compartir todos los cambios que realicemos con el resto del mundo, debemos realizar un *push*. la sintaxis de este comando es git push <remote> <branch>. En nuestro caso, la forma concreta de este comando será git push origin develop.

Mediante este comando, enviamos todos los commits de una cierta rama a un repositorio remoto, a su rama correspondiente. Esto sólo podremos hacerlo si la rama remota y la local comparten una base, es decir, *no se han realizado cambios en la rama remota desde que la obtuvimos*. Si esto no ocurre, Git nos informará y deberemos realizar un *pull* para incluir los cambios antes de nada.

## 4.7. Bajar cambios del repositorio remoto (pull)

Para poder ver los cambios realizados en el proyectos por otros usuarios, debemos descargar esos cambios mediante el comando git pull. Al descargar los cambios en una rama ya existente, Git automáticamente realizará un *merge*, lo cual puede llevar a conflictos si se ha modificado el mismo fichero varias veces.

## 4.8. Trabajo con ramas (checkout)

Para organizar distintas versiones paralelas de trabajo, Git funciona en lo que se denominan *ramas*. Una rama no es más que un conjunto de cambios que llevan a un estado del proyecto. Las ramas tienen generalmente una base común, a partir de la cual divergen.

Mediante el comando git checkout <rama>, podemos cambiar de rama en cualquier momento. Al cambiar de rama, todos los ficheros del directorio de trabajo se actualizarán a los de la nueva rama, permitiendo modificarla.

La variante git checkout -b <rama> nos permite crear una nueva rama. La nueva rama será idéntica a la rama desde la cual ejecutamos el comando, y Git automáticamente cambiará a esta nueva rama.

d0Hay que tener en cuenta que Git no nos dejará cambiar de rama inmediatamente si hay cambios sin guardar tanto en el directorio de trabajo como en el índice. Antes de cambiar de rama, debemos realizar un *commit* o descartar los cambios.

## 4.9. Juntar la rama actual con otra (merge)

Probablemente el elemento más importante de Git, por su potencia y su uso, es el *merge*. Este comando nos permite mezclar dos ramas diferentes en una, juntando todos los cambios.

Para ello usamos git merge <rama>. Esto introducirá todos los cambios de la rama especificada en nuestra rama actual. *Nótese que la única rama modificada es la actual, nunca la especificada en el comando*. Tras ello, Git automáticamente guarda los cambios introducidos por *merge* con un nuevo *commit*, y deja nuestro espacio de trabajo limpio y listo para seguir trabajando. Es de notar que, de la misma forma, *merge* se quejará si intentamos ejecutarlo en una rama con cambios sin almacenar en un *commit*.

En ocasiones podemos necesitar revisar y editar los cambios de un *merge* antes de realizar el *commit* final, o incluso podemos querer editar el mensaje de dicho *commit*. Para ello podemos usar la opción --no-commit. Nunca se nos debe olvidar realizar el *commit* manualmente tras esto.

Generalmente, Git es lo suficientemente *listo* como para mezclar incluso aquellos ficheros que han sido editados en ambas ramas en múltiples ocasiones. Sólo cuando la misma línea se ha editado en ramas diferentes, Git entenderá que hay un conflicto y dejará en manos del usuario su resolución.

Tanto si existe un conflicto como si elegimos no realizar el *commit* automáticamente, Git se quedará en un estado intermedio en el que todos los cambios se han introducido en el espacio de trabajo pero no se han guardado. En el caso de un conflicto, debemos resolverlo, añadir el fichero conflictivo al índice mediante git add, y posteriormente realizar un *commit*. En ambos casos, es recomendable realizar un *commit* inmediatamente después de revisar y corregir posibles errores de mezcla y conflictos, y no realizar ningún otro cambio.

## 4.10. Modificar el punto de partida de una rama (rebase)

Con rebase podemos modificar el commit original del cual partió la rama. La sintaxis del comando es git rebase <rama>, lo cual modificará la base de la rama actual por el commit de la rama indicada.

Su uso principal es el de integrar en la rama actual los cambios de la rama de partida como si se hubieran realizado *antes* de los commits de nuestra propia rama. Esto también se puede conseguir mediante git merge, aunque el resultado es diferente.

Más información y un montón de diagramas en <http://git-scm.com/book/en/Git-Branching-Rebasing>.

## 4.11. Más información

Para más información y mayores referencias y datos, es recomendable leerse el libro on-line [Pro Git](http://git-scm.com/book). Cualquier cosa que no haya quedado clara de este mini-manual, viene mejor explicada en el libro con total seguridad.

# 5. Formas de trabajo

## 5.1. Ramas remotas

El repositorio remoto contará en todo momento con dos ramas fijas, master y develop, así como de una serie de ramas que se irán creando según sean necesarias para coordinar a subgrupos del grupo que estén trabajando en una misma característica.

La rama master no debe modificarse bajo ningún concepto: de ello se encargará una única persona designada en el grupo, que periódicamente incorporará los cambios estables a tiempo para las entregas.

La rama develop será la que modifiquemos, directa o indirectamente, al realizar cambios en el proyecto. Sólo debe modificarse directamente para cambios menores como corrección de *bugs* o pequeños ajustes realizados por una sola persona que no encajen en ramas específicas creadas posteriormente.

Las *ramas específicas* se crearán cuando sea necesario, y serán los puntos de coordinación de todo el grupo. Cada vez que se decida implementar una cierta característica en el proyecto, se creará una de estas ramas, y los que estén asignados a estas tareas trabajarán en esta nueva rama. Alguien, una sola persona dentro de cada rama, se encargará de incorporar los cambios en develop, cuidando de no subir código que no compile o rompa el resto de la aplicación.

## 5.2. Ramas locales

Aunque cada uno es libre de trabajar en casa como quiera, es recomendable seguir una metodología de trabajo concreta.

Cada vez que vayamos a ponernos a trabajar, descargaremos todos los cambios de la rama específica que tengamos asignada. A partir de aquí, crearemos una rama para cada nueva tarea que realicemos. Trabajaremos en esta nueva rama durante todo o parte del día, y al finalizar, mezclaremos los cambios en nuestra rama específica y procederemos a subirla de nuevo al servidor. La rama local que creamos inicialmente ya no sirve para nada y podemos proceder a borrarla usando git branch rm <rama> sin problema.

De esta manera estamos consiguiendo varias cosas:

* Si alguno de nuestros compañeros actualiza el código, lo hará en la rama específica, y no en la rama local nueva. Esto puede evitar conflictos si actualizamos frecuentemente la rama específica para ver los cambios del compañero.
* Si el código que estamos escribiendo no nos sirviera o quisiéramos descartarlo, basta con borrar la rama.
* Es posible que queramos avanzar en una cierta característica mientras arreglamos un bug que acabamos de detectar. En lugar de corregir el bug en nuestra rama local y no incorporarlo hasta que acabemos, podemos volver a la rama específica, crear otra rama extra, corregir el bug y subir la corrección. Todo ello con nuestro código a medias, en una rama aparte, de la que nadie sabe nada. Posteriormente podemos incorporar los cambios a esta rama con un *merge* o un *rebase*, para tener el error corregido.

## 5.3. Commits organizados

Para que todos podamos saber en todo momento qué está pasando en la otra punta del proyecto, conviene mantener un orden y una organización en los *commits*. Esta organización se consigue con los siguientes aspectos aspectos:

* **Agrupar cambios similares en un *commit*, separar cambios diferentes en varios *commits*.**

Aunque pueda parecer una tontería, es conveniente agrupar los cambios en los *commits* por su lógica y no por su fecha. Si el mismo día, incluso a diferentes intervalos, hemos trabajado en dos partes distintas del proyecto, es conveniente integrarlas en el repositorio con *commits* separados.

Incluso si dentro del mismo fichero tocamos partes diferenciadas de código, conviene guardarlas en varios *commits*. Recordemos que Git trae una herramienta que permite añadir *trozos* de ficheros al índice en lugar de ficheros completos.

* + - **Evitar los *supercommits*.**

Debemos evitar siempre realizar *commits* con cientos de cambios. En lugar de juntar todo nuestro trabajo de un día en un solo *commit*, conviene separarlo en pequeños grupos. Una forma de hacerlo es siguiendo el anterior punto, otra es realizar pequeños *commits* cada cierto tiempo.

Los *commits* no tienen por qué incluir cambios completos, depurados o testeados. Aunque sí conviene que el código esté mínimamente limpio y compile, dejar una característica a medio hacer en un *commit* no está prohibido.

* **Describir los cambios de cada commit correcta y completamente.**

Git obliga a incluir un mensaje con cada *commit*. Este mensaje debe incluir una descripción de los cambios realizados en dicho *commit*, de forma que leyendo este mensaje sepamos exactamente qué ha cambiado.

El formato que utilizaremos se compondrá de una primera línea, muy breve, que dará una idea general de los cambios, y un cuerpo donde explicaremos los cambios de forma detallada. El formato exacto del cuerpo depende de cada uno: Podemos listar los cambios uno a uno, de forma concisa, redactar una explicación de lo que se ha hecho, o ambas.

## 5.4. Ramas y niveles de estabilidad

Cada una de las categorías en las que hemos dividido el sistema de trabajo corresponde a un nivel de estabilidad:

* + - * Las **ramas locales** corresponden a código activo y, por tanto, inestable. Cualquier cosa tiene cabida aquí.
      * Las **ramas específicas** corresponden a código activo pero compartido. El código que se suba aquí no debería fallar a niveles básicos como la compilación, pero puede ser inestable a nivel de usabilidad.
      * La **rama *develop*** corresponde a código en desarrollo pero con coherencia interna. El código que se suba aquí debe estar funcionando, aunque sea a nivel básico, y no romper otras funcionalidades.
      * La **rama *master*** corresponde a código *estable*, susceptible de ser entregado al profesor. Todo lo que se suba a esta rama estará testeado apropiadamente y no deberá contener fallos.