

Tema 4

Control de versiones.

Git.

Desarrollo de aplicaciones multiplataforma

Carlos Alberto Cortijo Bon



Esta obra está bajo una [Licencia Creative Commons Atribución-NoComercial-SinDerivadas 4.0 Internacional](https://creativecommons.org/licenses/by-nc-nd/4.0/).

Índice

1. Sistemas de control de versiones	1
2. Tipos de sistemas de control de versiones	1
2.1. Control de versiones local	2
2.2. Control de versiones centralizado	2
2.3. Control de versiones distribuido	2
3. Git	3
3.1. Almacenamiento de versiones de ficheros en Git	4
3.2. Estado de ficheros y flujo de trabajo en repositorio local	5
4. Instalación de Git	7
5. Comandos de Git	7
5.1. Crear repositorios (init, clone)	8
5.2. Configurar de Git (config)	9
5.3. Estado de ficheros en el área de trabajo (status)	10
5.4. Añadir y eliminar ficheros al control de versiones (add, rm)	10
5.5. Añadir ficheros cambiados al área de staging (add)	12
5.6. Ver cambios realizados (diff, difftool)	13
5.7. Confirmar cambios (commit)	14
5.8. Historial de versiones (log)	14
5.9. Etiquetar versiones (tag)	15
5.10. Obtener versiones del control de revisiones (checkout)	15
5.11. Deshacer cambios	17
5.11.1. Deshacer cambios aún no confirmados (antes de git commit) con git restore	17
5.11.2. Añadir nuevos cambios en una versión ya existente	18
5.11.3. El comando git reset	18
5.12. Crear, gestionar y fusionar ramas (branch, switch, merge)	19
5.12.1. Cambios en distintas ramas y comparación a tres bandas	20
5.12.2. Fusión de cambios (merge)	22
5.12.3. Conflictos de fusión (merge)	23
5.13. Fusión de cambios realizados en ramas (git merge)	24

1. Sistemas de control de versiones

Un **sistema de control de versiones** permite almacenar distintas versiones, a lo largo del tiempo, de todos los ficheros de un proyecto. Normalmente es un proyecto de desarrollo de *software*, y los ficheros que lo constituyen son los que permiten construir (*build*) el *software*. Este se construye a partir de los ficheros presentes en una jerarquía de directorios en un sistema de ficheros. Estos constituyen el *source tree* (árbol de código fuente), **directorio de trabajo** o *sandbox*.

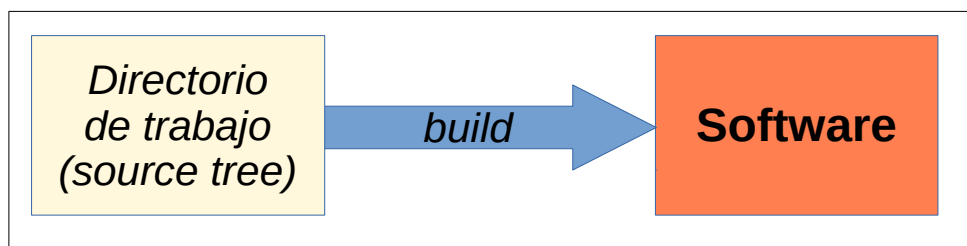


Ilustración 4.1: Proceso de construcción del software

Un sistema de control de versiones permite guardar **versiones de los ficheros individuales** que constituyen el *source tree*. Cada versión estará etiquetada (automáticamente) con una etiqueta de versión.

Un sistema de control de versiones permite también guardar **versiones del proyecto completo**. Una versión del proyecto consiste en un conjunto de ficheros, y la versión para cada uno. Normalmente estas versiones están etiquetadas o marcadas con una etiqueta que identifica la versión del proyecto. Cuando se decide que ha llegado el momento de marcar una nueva versión del proyecto, se indica una etiqueta, y se asigna esta etiqueta a la versión actual de cada uno de los ficheros incluidos actualmente en el proyecto. Más adelante será posible extraer la versión del proyecto indicando la etiqueta asignada a dicha versión del proyecto.

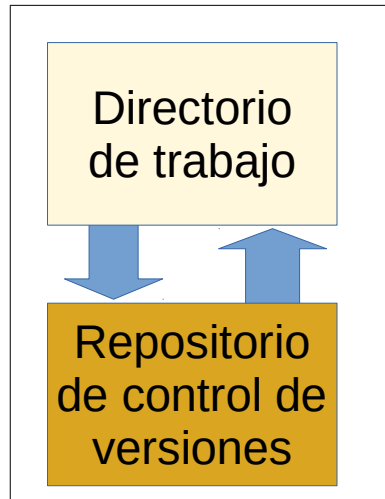


Ilustración 4.2: Repositorio

El control de revisiones trabaja con un **repositorio**. Este es una almacén que contiene todas las versiones de todos los ficheros e información adicional (*meta-datos*) acerca de sus propios contenidos. El directorio de trabajo es un directorio disponible de manera local para el propio programador, en el que tiene libertad para hacer todos los cambios que quiera creando, borrando o modificando ficheros o directorios. Normalmente, los cambios que se van haciendo en el repositorio, o parte de ellos, una vez probados, se reflejan en el repositorio, utilizando determinadas operaciones que proporciona el sistema de control de versiones. Se puede trabajar con dos tipos de repositorios:

- Repositorios locales. En la misma máquina en la que está el directorio de trabajo. Normalmente los utiliza un usuario determinado.
- Repositorios remotos. En otra máquina. Suelen ser compartidos con otros usuarios.

El conjunto de operaciones disponibles para ambos tipos de repositorio es diferente.

En un repositorio, normalmente, se añade información, pero nunca se borra. Por tanto, contiene toda la historia del proyecto, lo que permite, en cualquier momento, reproducir cualquier estado pasado en el área de trabajo. Por ejemplo, en cualquier momento se podría eliminar un fichero del proyecto en el área de trabajo, porque ya no se necesita en el *source tree* para construir el proyecto, y se puede confirmar este borrado en el repositorio. Pero este fichero no se borra del repositorio. Esto permite reproducir un estado anterior, en que este fichero formaba parte del proyecto, y construir la una versión anterior del proyecto en que este fichero formaba parte de él.

2. Tipos de sistemas de control de versiones

Existen diferentes tipos de sistemas de control de versiones, cada uno con sus características, ventajas e inconvenientes.

2.1. Control de versiones local

En un control de versiones local, el repositorio está en la máquina local, al igual que el *source tree*. Y no se comparte con otros miembros del equipo de desarrollo. En la ilustración se muestra un fichero particular del *source tree*, del que se almacenan en el repositorio varias versiones.

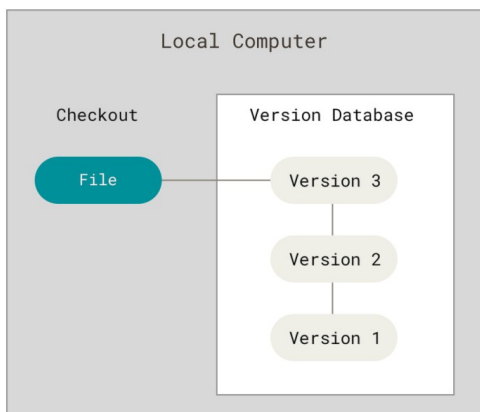


Ilustración 4.3: Control de versiones local

Las operaciones que se pueden realizar son, en general:

add: Añadir al repositorio un fichero presente en el *source tree*. Con ello, el fichero pasa a formar parte del proyecto y a estar controlado por el control de versiones.

remove: Eliminar del repositorio un fichero. Con ello no se borra su historial, de manera que las versiones antiguas siguen almacenadas en él.

checkin o commit: Guardar en el repositorio una nueva versión de un fichero situado en el *source tree*.

checkout: Obtener en el *source tree* la última versión de un fichero disponible en el repositorio. Pero también podría obtenerse una versión antigua.

El inconveniente que tiene este tipo de sistema es que solo vale para un programador, y normalmente en cualquier proyecto de desarrollo de software intervienen varios, incluso muchos. Como ventaja principal se tiene su rapidez, dado que todas las operaciones sobre el repositorio se realizan sobre ficheros locales.

2.2. Control de versiones centralizado

En un control de versiones centralizado, el repositorio está en un servidor, y es compartido por todos los miembros del equipo de desarrollo.

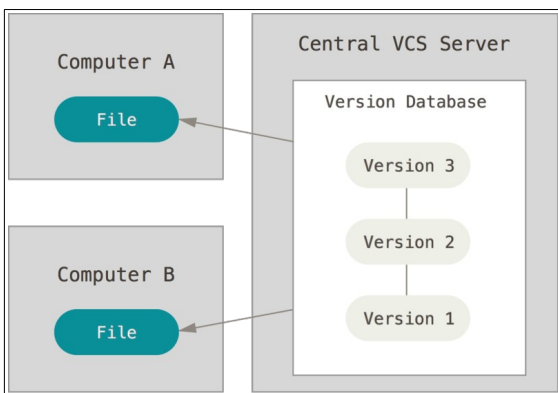


Ilustración 4.4: Control de versiones centralizado

Este modelo permite el trabajo colaborativo de todos los componentes de un equipo de desarrollo. Pero tiene el inconveniente de que el directorio compartido es un punto crítico del sistema. Si falla, falla todo el sistema y, en el mejor de los casos, los programadores pueden seguir trabajando en su directorio de trabajo, pero sin control de versiones. Es además un cuello de botella. En proyectos grandes (con muchos ficheros), en el que participan muchos programadores que realizan muchas operaciones sobre el repositorio, el rendimiento se puede degradar rápidamente.

Por lo demás, las operaciones que se pueden realizar con el sistema de control de versiones son las mismas que con un control de versiones centralizado. Pero plantean algunos problemas particulares por el hecho de que distintos programadores pueden realizar cambios simultáneamente sobre las copias locales del mismo fichero que tienen en su directorio de trabajo.

2.3. Control de versiones distribuido

En un **control de versiones distribuido**, existe un repositorio maestro compartido, pero cada programador tiene en local un control de versiones completamente funcional, y puede hacer una copia local completa del repositorio maestro compartido. Este modelo combina las ventajas del control de versiones local y compartido. Los programadores trabajan con su control de versiones local, sobre el repositorio compartido, pero pueden sincronizar los contenidos del repositorio local con los del repositorio maestro compartido.

La principal ventaja de este modelo es que permite realizar la mayor parte del trabajo sobre un repositorio local, lo que es mucho más rápido. También la seguridad, dado que si por cualquier motivo se perdiera la información en el repositorio maestro, se pueden restaurar sus contenidos a partir de los contenidos de los repositorios locales. El principal inconveniente puede ser la complejidad. Pueden ser frecuentes los conflictos entre cambios realizados localmente y cambios realizados por otros programadores. Puede ser, por tanto, difícil actualizar los contenidos de un repositorio lo-

cal para incorporar los cambios hechos por otros programadores y grabados en el repositorio maestro. Y viceversa, puede ser difícil incorporar los cambios hechos en el repositorio local al repositorio maestro.

Las operaciones que los programadores pueden realizar sobre sus repositorios locales son las mismas que con un control de versiones local o centralizado. Pero hay nuevas operaciones para gestionar la interacción entre el repositorio local y el maestro. A saber:

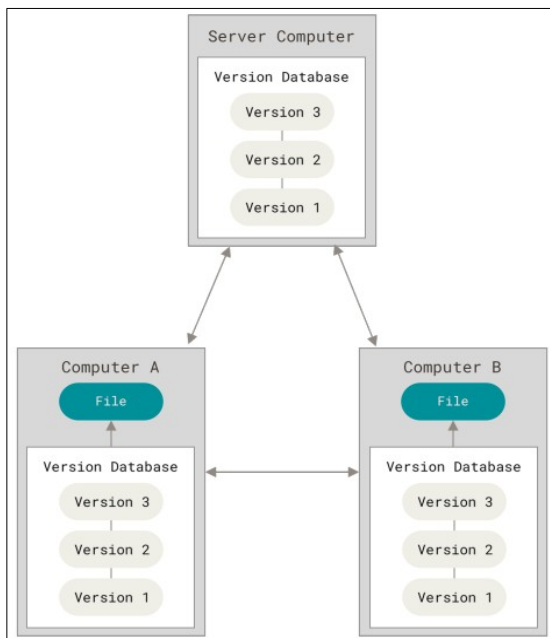


Ilustración 4.5: Control de versiones distribuido

clone: Obtener una copia local del repositorio maestro.

pull: Obtener las últimas versiones del repositorio maestro. No necesariamente se obtiene para cada fichero el contenido de la última versión, lo que podría provocar que se pierdan los cambios realizados en el repositorio local. Se pueden fusionar (*merge*) los cambios realizados en versiones posteriores del repositorio, de manera análoga a como se vio en un apartado anterior.

push: Guardar los cambios realizados sobre el repositorio local en el repositorio maestro. De nuevo, no necesariamente se guarda para cada fichero el contenido del repositorio local. Se pueden fusionar (*merge*) los cambios realizados en el repositorio local.

3. Git

Git es un sistema de control de versiones distribuido de código libre (licencia GNU). Se desarrolló inicialmente, en 2005, para gestionar el desarrollo del núcleo de Linux, y su creador fue el mismo Linus Torvalds.

Existe documentación detallada de Git en la siguiente dirección: <https://git-scm.com/book/en/v2>. En esta dirección están disponibles los contenidos del libro Pro Git de Scott Chacon and Ben Straub, publicado por Apress.

Desde el principio del mantenimiento del *kernel* (núcleo) de Linux, los cambios se realizaban mediante conjuntos de ficheros y de parches (unos ficheros especiales en los que se indicaban cambios a realizar en los ficheros de código fuente). En 2002 se empezó a utilizar un sistema de control de versiones distribuido privativo, llamado BitKeeper. En 2005, la relación entre la comunidad que desarrollaba el *kernel* de Linux y la compañía que desarrollaba BitKeeper se deterioró, y la herramienta dejó de ser ofrecida de manera gratuita. Se tomó entonces la decisión de desarrollar un nuevo sistema de control de versiones de código libre. La primera versión la desarrolló Linus Torvalds en 2005, y lo llamó Git. Los principales objetivos del nuevo sistema fueron los siguientes:

- Velocidad.
- Diseño sencillo.
- Soporte para desarrollo no lineal (muchas ramas paralelas).
- Completamente distribuido.
- Escalabilidad. Para poder gestionar grandes proyectos, como el *kernel* de Linux y otros proyectos de software libre.

3.1. Almacenamiento de versiones de ficheros en Git

La principal característica de Git con respecto a sistemas previos es que no guarda las diferencias entre versiones sucesivas de un mismo fichero de texto, sino que guarda el contenido íntegro de las diferentes versiones.

Es decir, si la versión n de un fichero de texto F es $F[n]$, la versión $n+1$ es $F[n+1]=F[n]+\Delta_n$, donde Δ_n son los cambios introducidos por la versión $n+1$, $F[n+1]$, con respecto a la versión anterior, $F[n]$. En algunos sistemas anteriores de control de versiones se guardaba la versión inicial $F[1]$ y las sucesivas diferencias que introducían las versiones posteriores: Δ_1 , Δ_2 , etc. De esta manera, para obtener el contenido de una versión k de un fichero había que empezar con la primera, $F[1]$, y aplicar sucesivamente Δ_1 , $\Delta_2, \dots, \Delta_{k-1}$.

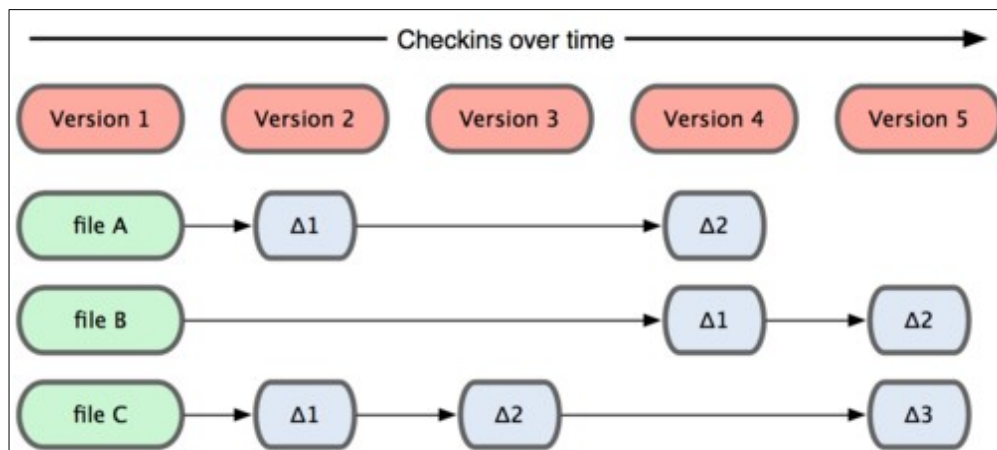


Ilustración 4.6: Almacenamiento de versiones de un proyecto en otros sistemas de control de versiones

Esto permite un almacenamiento muy compacto, pero tiene algunos inconvenientes:

- No es apropiado para ficheros binarios, que también se utilizan en muchos proyectos.
- El proceso de calcular las diferencias entre versiones y de reconstruir los ficheros aplicando sucesivamente diferencias es costoso.

El repositorio de Git, en cambio, actúa como un sistema de ficheros direccionable por contenido. Para almacenar el contenido de un fichero, se calcula el valor de la función hash SHA-1 sobre sus contenidos, y los contenidos del fichero se almacenan en el repositorio en un fichero con ese nombre, que se conoce como objeto *blob*. Si se almacenan dos ficheros con el mismo contenido, el contenido en sí solo se almacena una vez, en un único objeto *blob* cuyo nombre corresponde al contenido.

En Git, una versión determinada de un proyecto contiene, entre otras cosas, un conjunto de las referencias a las versiones de los diferentes ficheros incluidas en el proyecto.

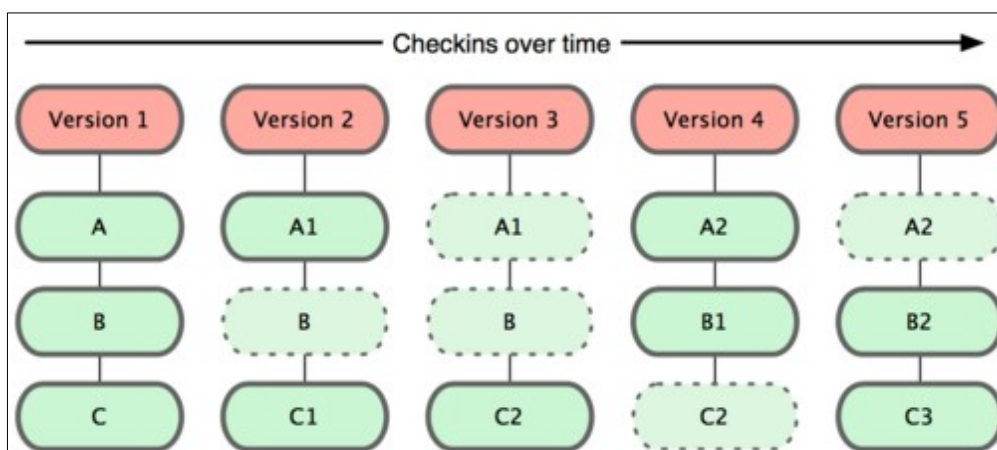


Ilustración 7: Almacenamiento de versiones de un proyecto en Git

El repositorio de Git está en un directorio con nombre `.git`. Para almacenar el contenido de un fichero, Git calcula el valor del *hash* de sus contenidos según el algoritmo SHA-1, lo que da como resultado una cadena de 40 caracteres. El contenido del fichero se almacena en el repositorio en un fichero con este nombre. Cualquier versión de cualquier fichero con los mismos contenidos tendrá, por tanto, la misma referencia, y el contenido solo se almacenará una vez en el

repositorio. Esto tiene la ventaja, además, de que permite comprobar la integridad de los ficheros. Si por algún fallo cambian los contenidos de un fichero, entonces dejará de coincidir el nombre del fichero con el *hash* de sus contenidos.

Para saber más: función de *hash* SHA1

La función de *hash* de un fichero se puede calcular, en Linux, con el comando `sha1sum`, como se muestra en el siguiente ejemplo:

```
package hola;

public class Hola {
    public static void main(String[] args) {
        System.out.println("Hola.");
    }
}
```

```
$ sha1sum Hola.java
53dafb242c141ab67ccd390b24dd669e5ed1394d  Hola.java
```

Si se hace cualquier cambio, por mínimo que sea, en el fichero, su valor de *hash* cambia completamente. Por ejemplo, si se quita el punto después de `Hola`, el resultado sería:

```
$ sha1sum Hola.java
ac514332f7934b2faebe925b323ca09e1141e075  Hola.java
```

3.2. Estado de ficheros y flujo de trabajo en repositorio local

Además del área de trabajo y el repositorio local, en Git existe un área especial llamada *staging area*. Cuando se hace una operación de *commit* en el repositorio local, los cambios se almacenan en el área de *staging*. El área de *staging* también se llama *index* o índice, porque es realmente un fichero en el que se anotan los cambios que se incluirán en la siguiente operación *commit*.

Un fichero del área de trabajo se puede encontrar en varios estados:

- **No gestionado.** O *untracked*. El fichero no está gestionado por el control de versiones.
- **Modificado.** Se han modificado sus en el área de trabajo, de manera que no se corresponden con los contenidos almacenados en el repositorio.
- **Staged.** Se han marcado los contenidos del área de trabajo para ser confirmados, pero no se han confirmado en el repositorio. Entre tanto, están en el índice o área de *staging*.
- **Confirmado.** O *committed*, o no modificado o *unmodified*. Se han confirmado los cambios en el repositorio local (directorio `.git`). Es decir, que sus contenidos coinciden con los de una versión almacenada en el repositorio local.

La siguiente ilustración muestra dónde se almacenan los contenidos más recientes de un fichero cuando se realizan las distintas operaciones con Git. Al hacer *checkout* se pasan los contenidos del repositorio local al directorio de trabajo. Los cambios realizados en este se pueden pasar al área de *staging*, y de ahí al repositorio local al hacer un *commit*. Quiere esto decir que los cambios se van guardando en el área de *staging* y solo cuando se hace *commit* se pasan todos los cambios almacenados en este área al repositorio local.

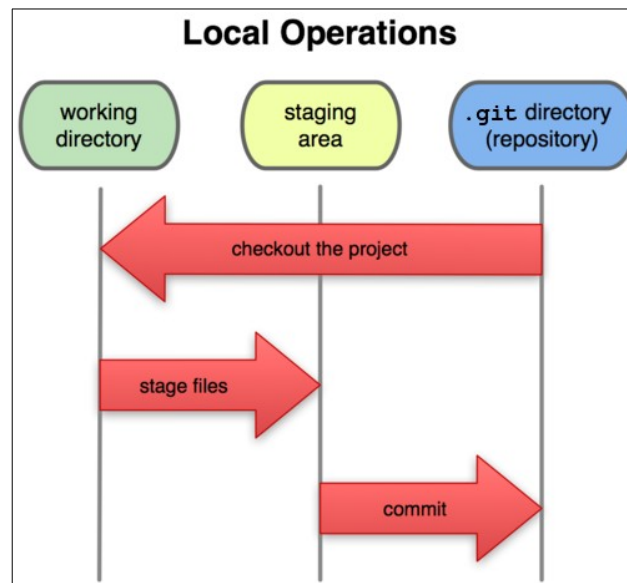


Ilustración 4.8: Almacenamiento de ficheros en Git

El flujo de trabajo básico en Git es el siguiente:

1. Modificar/crear ficheros en el directorio de trabajo.
2. Añadir los archivos modificados o creados al área de *staging* (operación `add`).
3. Confirmar los cambios añadidos al área de *staging* (operación `commit`). Esto añade una nueva versión de estos ficheros al repositorio. En el caso particular de ficheros que no existían antes en el repositorio, será la primera versión.

En consonancia con todo lo anterior, la siguiente ilustración muestra los posibles estados de los ficheros presentes en el área de trabajo, y sus cambios de estado con cada una de las operaciones anteriores. Los ficheros presentes en el área de trabajo, pero no gestionados por Git, están en estado *untracked* o no gestionado. Con la operación `add` se pueden añadir ficheros no gestionados al área de *staging* (*Add the file*), y también añadir a dicha área ficheros sobre los que se han realizado cambios (*Stage the file*). En cualquier caso, con la siguiente operación `commit` se añadirá una nueva versión en el repositorio local de todos los ficheros presentes en el área de *staging*.

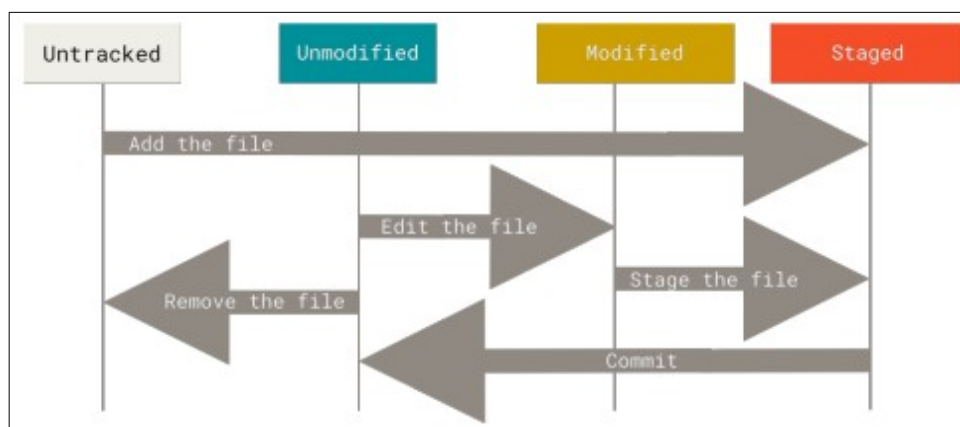


Ilustración 4.9: Estados de un fichero con Git

4. Instalación de Git

Git está disponible, al menos, para Linux, Windows y MacOS. El procedimiento de instalación es distinto según el sistema operativo. Git incluye un programa que se utiliza desde línea de comandos. Además de eso, existen diversos *front-end* gráficos, según el sistema operativo. Aquí se aprenderá a instalar y utilizar el programa de línea de comandos en Linux.

Se instala `git` en Linux con el siguiente comando:

```
$ sudo apt install git
```

Una vez hecho esto, se puede ejecutar Git con el comando `git`.

5. Comandos de Git

Si se ejecuta `git` sin ningún parámetro, se muestra ayuda acerca de su uso. Se puede ver que la forma general de uso es `git <comando> [<argumentos>]`. Los comandos principales se resaltan en gris.

```
$ git
uso: git [--version] [--help] [-C <path>] [-c <name>=<value>]
      [--exec-path=<path>] [--html-path] [--man-path] [--info-path]
      [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--bare]
      [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
      <command> [<args>]
```

Estos son comandos comunes de Git usados en varias situaciones:

comienza un área de trabajo (ver también: `git help tutorial`)

<code>clone</code>	Clona un repositorio dentro de un nuevo directorio
<code>init</code>	Crea un repositorio de Git vacío o reinicia el que ya existe

trabaja en los cambios actuales (ver también: `git help everyday`)

<code>add</code>	Agrega contenido de carpetas al índice
<code>mv</code>	Mueve o cambia el nombre a archivos, directorios o enlaces simbólicos
<code>restore</code>	Restaurar archivos de árboles de trabajo
<code>rm</code>	Borra archivos del árbol de trabajo y del índice
<code>sparse-checkout</code>	Inicializa y modifica el <code>sparse-checkout</code>

examina el historial y el estado (ver también: `git help revisions`)

<code>bisect</code>	Use la búsqueda binaria para encontrar el commit que introdujo el bug
<code>diff</code>	Muestra los cambios entre commits, commit y árbol de trabajo, etc
<code>grep</code>	Imprime las líneas que concuerdan con el patron
<code>log</code>	Muestra los logs de los commits
<code>show</code>	Muestra varios tipos de objetos
<code>status</code>	Muestra el estado del árbol de trabajo

crece, marca y ajusta tu historial común

<code>branch</code>	Lista, crea, o borra ramas
<code>commit</code>	Graba los cambios en tu repositorio

```

merge          Junta dos o más historiales de desarrollo juntos
rebase         Vuelve a aplicar commits en la punta de otra rama
reset          Reinicia el HEAD actual a un estado específico
switch         Cambiar branches
tag            Crea, lista, borra o verifica un tag de objeto firmado con GPG

colabora (mira también: git help workflows)

fetch          Descarga objetos y referencias de otro repositorio
pull           Realiza un fetch e integra con otro repositorio o rama local
push           Actualiza referencias remotas junto con sus objetos asociados

'git help -a' y 'git help -g' listan los subcomandos disponibles y algunas
guías de concepto. Consulte 'git help <command>' o 'git help <concepto>'
para leer sobre un subcomando o concepto específico.
Mira 'git help git' para una vista general del sistema.

```

Se puede obtener ayuda acerca de un comando específico por ejemplo, el comando `add`, de las siguientes maneras:

- `git add -h`. Para ayuda concisa, indicando las principales opciones.

```

$ git add -h
uso: git add [<opción>] [--] <especificación-de-ruta>...

-n, --dry-run      dry run ( ejecución en seco)
-v, --verbose      ser verboso

-i, --interactive  selección interactiva
-p, --patch        elegir hunks de forma interactiva
-e, --edit         editar diff actual y aplicar
-f, --force        permitir agregar caso contrario ignorar archivos

```

- `git help add`. Para ayuda exhaustiva, explicando las opciones en más detalle.

```

$ git --help add
GIT-ADD(1)                                Git Manual                                GIT-ADD(1)

NAME
    git-add - Add file contents to the index

SYNOPSIS
    git add [--verbose | -v] [--dry-run | -n] [--force | -f] [--interactive | -i] [--patch | -p]
            [--edit | -e] [--no-all | --[no-]ignore-removal | [--update | -u]]
            [--intent-to-add | -N] [--refresh] [--ignore-errors] [--ignore-missing] [--renormalize]
            [--chmod=(+|-)x] [--] [<paths>...]

DESCRIPTION
    This command updates the index using the current content found in the
    working tree, to prepare the content staged for the next commit. It
    typically adds the current content of existing paths as a whole, but
    with some options it can also be used to add content with only part of
    the changes made to the working tree files applied, or remove paths
    that do not exist in the working tree anymore.

(...)

```

En los siguientes apartados se explican los distintos comandos de Git y sus opciones, estando cada apartado dedicado a un tipo de operaciones. Si se necesita más información:

- Hay más información acerca de todos los comandos en <https://git-scm.com/docs>.
- En la siguiente dirección, muy interesante, se pueden ver los comandos según su relación con área de trabajo, área de staging (índice), repositorio local y repositorio remoto: <https://ndpsoftware.com/git-cheatsheet.html>.

5.1. Crear repositorios (`init`, `clone`)

Se puede crear un repositorio de git de dos formas:

- a) Para los contenidos de un directorio determinado. Para ello, hay que situarse en el directorio y ejecutar el comando:

```
$ git init
```

Esto crea un repositorio, ubicado en un nuevo directorio con nombre `.git`, para el directorio.

Por defecto, ninguno de los ficheros existentes en el directorio se incluye en el control de versiones. Hay que añadir los que interese gestionar con el control de versiones, y más adelante se explicarán los comandos que permiten hacer esto.

- b) Clonando un repositorio ya existente.

```
$ git clone <url_del_proyecto>
```

Este comando crea un nuevo directorio, que es una réplica o clon local del repositorio disponible en la ubicación especificada por `<url_del_proyecto>`. Se crea un directorio, cuyo nombre depende de `url_del_proyecto`, y dentro de él se copia toda la jerarquía de ficheros del proyecto, y un repositorio local, dentro de un directorio `.git`.

5.2. Configurar de Git (config)

Una vez instalado Git, se tiene una configuración inicial por defecto, en un fichero `~/.gitconfig`. La configuración de Git se puede hacer a tres niveles, de más general a más particular:

- Configuración general en el ámbito del sistema, en el fichero `/etc/gitconfig`. Este fichero no se crea por defecto pero, si existe, las opciones de configuración que contiene se aplican a nivel de sistema, para cualquier repositorio de cualquier usuario. Como es lógico, Se necesitan permisos de administrador para realizar cambios en este fichero.
- Configuración para un usuario en `~/.gitconfig`, donde `~` es el directorio personal del usuario. Las opciones de configuración que contiene este fichero se aplican para todos los repositorios del usuario en cuestión, y tienen preeminencia sobre las anteriores.
- Configuración para un repositorio en un fichero de nombre `config` dentro del propio repositorio, es decir, en `.git/config`. Las opciones contenidas en este fichero tienen preeminencia sobre todas las anteriores.

Se puede editar estos ficheros manualmente. Pero también se pueden consultar y modificar sus contenidos con el comando `git config`.

Para empezar, se puede consultar la configuración de Git con `git config`. La opción `--show-origin` hace que se muestre, para cada opción de configuración, el fichero desde el que se ha obtenido.

```
$ git config --list --show-origin
```

Se puede asignar un valor a una opción, de manera general para el propio usuario (opción `--global`), con el siguiente comando:

```
$ git config --global <opción> <valor>
```

Por ejemplo, se puede configurar `gedit` como editor con el siguiente comando.

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

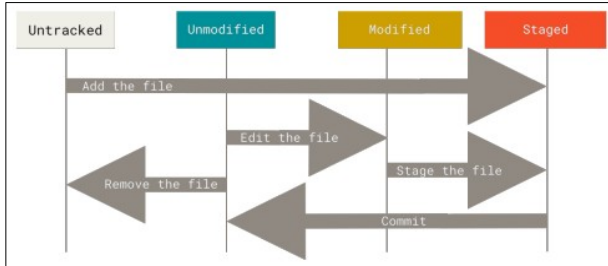
Con la opción `--global`, los cambios se hacen para el propio usuario, es decir, en el fichero `~/.gitconfig`. Si no se especifica la opción `--global`, la configuración se hace para el repositorio en el que se está situado, en el fichero `config` dentro del directorio `.git`.

Las siguientes opciones son importantes, y Es conveniente asignarles valores globalmente para el propio usuario con `git config -- global`, porque identifican al usuario en las operaciones de confirmación (`commit`) que realiza.

<code>user.name</code>	Nombre del usuario
<code>user.email</code>	Dirección de correo electrónico

5.3. Estado de ficheros en el área de trabajo (status)

El comando `git status` muestra el estado de los ficheros existentes en el área de trabajo, con respecto a los contenidos del repositorio.



Los ficheros dentro del directorio de trabajo pueden estar gestionados por el control de versiones (*tracked*) o no gestionados (*untracked*). Este último es el primer estado mostrado en el esquema anterior.

El comando `git status` muestra el estado de todos los ficheros no gestionados (*untracked*), o gestionados pero que no están en estado *unmodified*. El contenido de los ficheros en este último estado es idéntico al que hay en la versión actual del control de versiones.

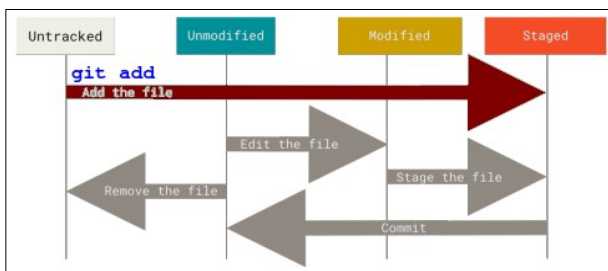
Por lo demás, los ficheros gestionados por el control de versiones (*tracked*) pero que no están en estado *unmodified*, pueden estar en los siguientes estados:

- **modified.** El fichero presente en el área de trabajo contiene modificaciones con respecto al existente en el repositorio, y estos cambios no se han pasado a estado *staged* para confirmarlos más adelante. Es decir, el fichero contiene cambios que no está previsto confirmar por el momento en la próxima operación `commit`.
- **staged.** El fichero presente en el área de trabajo contiene modificaciones con respecto al existente en el repositorio, y se ha solicitado que se incluya una nueva versión del fichero en la próxima operación `commit`.

5.4. Añadir y eliminar ficheros al control de versiones (add, rm)

Se puede hacer que un fichero que está en estado *untracked* pase a estar gestionado por el control de versiones con el comando

```
$ git add <fichero>
```



Con esto, el fichero pasa de estado *untracked* a estado *staged*. Con la próxima operación `commit`, se añadirá una nueva versión del fichero (la primera) al control de versiones.

Ignorar ficheros con .gitignore

Se puede hacer que, Git ignore determinados tipos de ficheros. Para ello se crea un fichero `.gitignore` que incluye los patrones de ficheros que hay que ignorar. Por ejemplo, si el contenido del fichero fuera este:

```
*.o
*.a
*~
```

Git ignoraría los ficheros cuyo nombre termina en `.o`, en `.a`, y en `~`.

También se pueden añadir directorios en este fichero. Por ejemplo:

```
obj/
bin/
```

Las dos líneas anteriores harían que se ignoraran todos los ficheros en los directorios `obj` y `bin`.

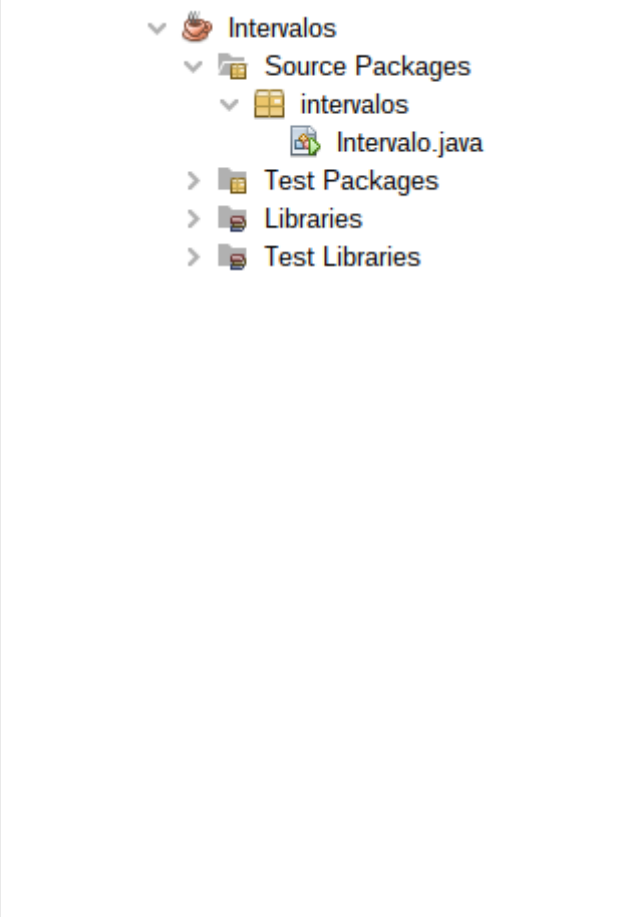
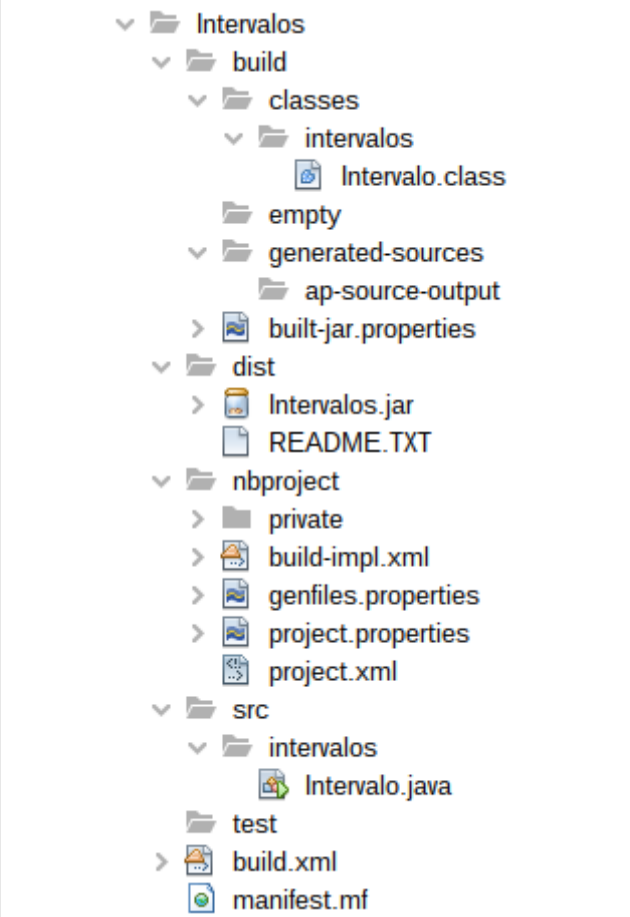
Pueden haber ficheros `.gitignore` en directorios por debajo del inicial, y entonces sus contenidos se tendrán en cuenta dentro de esos directorios.

No es mala idea incluir el propio fichero `.gitignore` en el repositorio, con `git add .gitignore`.

Actividad 1

Solo deben incluirse en el control de versiones los ficheros necesarios para generar el programa y todos los ficheros adicionales que este necesite durante su ejecución. En el caso particular de un proyecto de Java, está claro que hay que incluir los ficheros `.java`, pero no los ficheros `.class`, porque estos últimos contienen el bytecode que se generan cuando se compilan los ficheros `.java`.

Abre un proyecto de Java que hayas desarrollado (vale cualquiera, por simple que sea). Construye el proyecto (opción “build”) y mira en las vistas de proyecto y de ficheros, y en esta última, fíjate en los ficheros que existen dentro del directorio del proyecto. Deberías haber algo análogo a esto.

Vista de proyectos	Vista de ficheros
	

La operación `clean` borra todos los ficheros que no son estrictamente necesarios. Para ejecutarla sobre el proyecto, se puede pulsar con el botón derecho sobre el proyecto y seleccionar la opción “clean”.

En la vista de ficheros se puede ver que se han borrado algunos directorios y ficheros. Solo se han dejado los necesarios para generar el programa ejecutable. Entre ellos, por supuesto, los ficheros `.java` con el código fuente. Pero no solo esos.

Elige un proyecto cualquiera de Netbeans para un programa en Java. Averigua en qué directorio están sus ficheros. Copia los contenidos de este directorio en otro lugar. Sitúate en el directorio donde está la copia que acabas de crear y

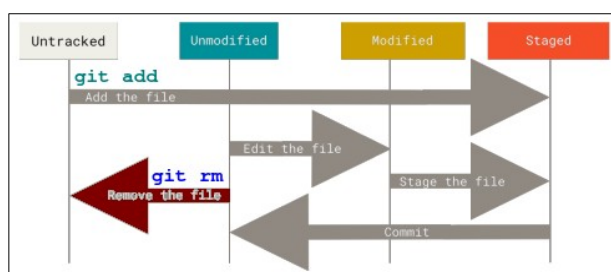
ejecuta el comando `git init`. Añade al control de versiones todos los ficheros necesarios con `git add` y `git commit`. Si ejecutas `git status`, verás en estado *untracked* (sin seguimiento) el resto de ficheros. Crea un fichero `.gitignore` con contenidos apropiados para que se ignoren estos ficheros. Cuando ejecutas el comando `git status`, no se deberían mostrar estos ficheros. Investiga acerca del contenido y la utilidad de los distintos ficheros que han quedado. Deberías entender para qué sirve cada uno de los ficheros que has incluido en el control de versiones.

Por último, incluye el propio fichero `.gitignore` en el control de versiones.

En <https://github.com/github/gitignore> hay ficheros cuyo contenido podría ser útil como punto de partida para los contenidos de `.gitignore` para proyectos de desarrollo en distintos lenguajes de programación. Uno de ellos es Java. Una vez creado tu fichero `.gitignore`, compara sus contenidos con los del fichero que existe en esta página web para Java.

Se puede eliminar un fichero del área de trabajo y del control de versiones con el comando `git rm`. El cambio se anota en el área de *staging*, donde aparece con estado *deleted*, y se confirma con la siguiente operación `commit`.

```
$ git rm <fichero>
```



Si el fichero está en estado *modified* (cambiado con respecto al repositorio) o *staged* (se han añadido cambios en el área de *staging*), no se permitirá esta operación, a no ser que se utilice la opción `-f` para forzar el borrado.

Git tiene también un comando `mv`. El comando

```
$ git mv <fichero_origen> <fichero_destino>
```

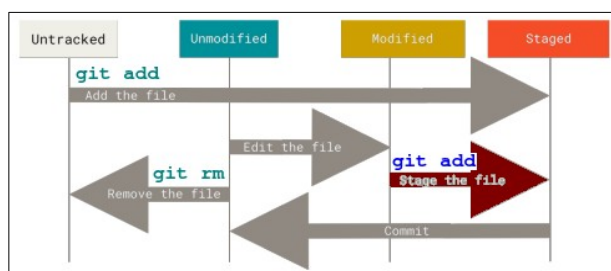
Es equivalente a

```
$ mv <fichero_origen> <fichero_destino>
$ git rm <fichero_origen>
$ git add <fichero_destino>
```

5.5. Añadir ficheros cambiados al área de *staging* (*add*)

Se puede hacer que un fichero que está en estado *modified* pase al área de *staging* y, por tanto, al estado *staged*. Con ello, se añadirá una nueva versión del fichero en el control de versiones con la siguiente operación `commit`.

```
$ git add <fichero>
```



Con esto, el fichero pasa de estado *untracked* a estado *staged*. Con la próxima operación `commit`, se añadirá una nueva versión del fichero (la primera) al control de versiones.

Atención: Si se ejecuta `git add <fichero>` con un fichero en estado *untracked*, y después se modifica, en el área de *staging* está el fichero con los contenidos que tenía cuando se añadió inicialmente, sin los últimos cambios. Si se ejecuta el comando `git status`, el fichero aparecerá dos veces, una por los cambios que se incluyeron en el área de *staging* con el comando `git add`, y otra por los últimos cambios. Para que se incluyan los últimos cambios en el área de *staging*, debe ejecutarse de nuevo `git add <fichero>`. Entonces, al ejecutar `git status`, el fichero solo aparecerá una vez, en el área de *staging*. Otra posibilidad es hacer `git commit` para confirmar los cambios en el área de *staging*, con lo que el comando `git status` solo mostrará el fichero una vez, en estado *modified*.

Se pueden pasar todos los ficheros modificados al área de *staging* con el comando:

```
$ git add .
```

Actividad 2

¿Actúa el comando anterior de forma recursiva? Es decir, ¿Añade solo los ficheros cambiados o en estado *untracked* situados en el directorio actual, o también los situados en directorios dentro del directorio actual?

5.6. Ver cambios realizados (*diff*, *difftool*)

Uno de los estados en los que puede estar un fichero presente en el área de trabajo es *modified*. Esto significa que se han realizado cambios en él con respecto a la última versión almacenada en el control de versiones.

Los ficheros manejados en un proyecto de desarrollo de software son en general ficheros de texto con código fuente en un lenguaje de programación, o lenguajes de marcas como XML o HTML. Y los cambios que se realizan sobre ellos consisten en la inserción, el borrado o la modificación de líneas individuales o de bloques de líneas contiguas.

Existen programas que permiten ver gráficamente las diferencias, como por ejemplo `meld` en Linux. Se puede ver que los cambios en la nueva versión consisten en un intervalo de líneas que se han borrado, y en una nueva línea que se ha añadido. Una herramienta así es todo lo que necesitaría un programador que utiliza un control de versiones local para ver los cambios que ha realizado con respecto a la última versión.

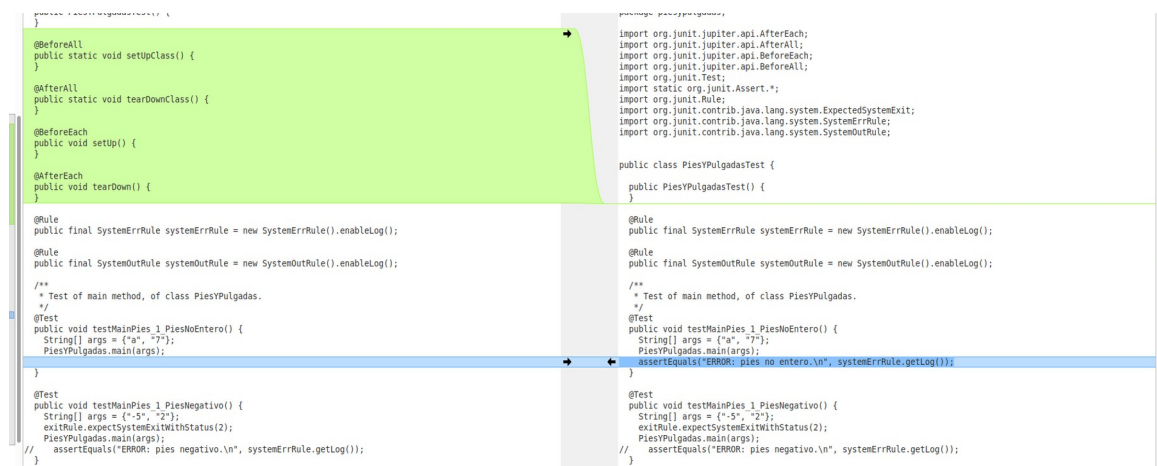


Ilustración 4.10: Cambios realizados en un fichero de texto, mostrados con `meld`

El comando `git status` muestra sobre qué ficheros se han realizado cambios que no se han pasado al área de *staging*, y sobre qué ficheros se han realizado cambios que están en el área de *staging*.

Se puede ver el detalle de los cambios realizados (Δ) con el comando `git diff`. Este comando se puede utilizar para mostrar:

- Cambios realizados sobre ficheros, pero que todavía no se han pasado al área de *staging*. Para ello, se puede ejecutar el siguiente comando.

```
$ git diff
```


- Cambios que se han pasado al área de staging, pero que todavía no están confirmados (con `commit`). Es decir, que no se han guardado en una nueva versión en el repositorio. Para ello se puede utilizar el siguiente comando.

```
$ git diff --staged
```

Este comando muestra la diferencia en el formato de `diff`, que es un tanto incómodo para leer. Puede mostrarse utilizando diversas herramientas, como por ejemplo `meld` o `kdifff3`. Para ver cuáles de ellas están instaladas, se puede utilizar el comando

```
$ git difftool --tool-help
```

Se puede asignar un valor apropiado a la variable de configuración `diff.tool` para que `git difftool` utilice una herramienta en particular. Se puede asignar un valor de los mostrados por `git difftool --tool-help`.

Para ver los cambios con una herramienta (la que se haya configurado o, si no se ha configurado ninguna, una que elija Git) en lugar de con `diff`, se puede utilizar el comando.

```
$ git difftool
```

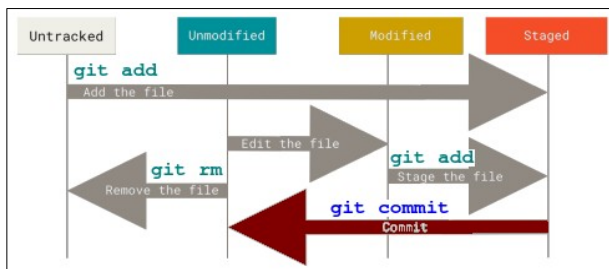
Se puede especificar un nombre de fichero para que solo muestre las diferencias para ese fichero.

Por lo demás, las opciones son las mismas para `git diff` y para `git difftool` (por ejemplo, `--staged`).

5.7. Confirmar cambios (*commit*)

Se pueden confirmar cambios realizados y que están en el área de *staging* con el comando

```
$ git commit
```



Con esto, se añade una nueva versión del fichero al control de versiones, y el fichero pasa de estado *staged* a estado *unmodified*.

Se puede especificar un texto descriptivo de los cambios realizados con la opción `-m`. Si no, Git lanzará un editor de texto para crear este texto. Ya se ha visto que se puede configurar este editor con la opción de configuración `core.editor`. Como ayuda, en el texto a editar se incluirá una lista con los ficheros que irán en el `commit`. Con la opción `-v`, se incluirá además el resultado de `git diff` para cada uno.

Esta operación no solo guarda una nueva versión de los ficheros particulares, sino que guarda una imagen (*snapshot*) del proyecto completo, que se puede recuperar más adelante.

La opción `-a` hace que se incluyan ficheros cambiados pero no añadidos al área de *staging* (es decir, en estado *modified*). Puede ser útil, pero hay que usarla con precaución.

5.8. Historial de versiones (*log*)

Se puede consultar el historial de versiones con

```
$ git log
```

```
commit faecca5626af700e44d9ed13e4e1b17563d94b9b (HEAD -> master)
Author: carlos <carloscortijo@iesportada.org>
Date: Tue Apr 13 13:00:34 2021 +0200

    Se añade un fichero y se cambia otro

commit 32e58ea98ee1604b06877d48b405dac5628e117c
Author: carlos <carloscortijo@iesportada.org>
Date: Tue Apr 13 12:58:10 2021 +0200

    Commit directo

commit 017b1f1d5ee5013778c25fe9668e4135fc81f23d
Author: carlos <carloscortijo@iesportada.org>
Date: Tue Apr 13 12:25:34 2021 +0200
```

Con `git log --oneline` se muestra un listado más escueto, con la información para cada versión en una única fila. Se puede ver que cada versión tiene un identificador único, que es el principio de un valor de *hash* para la versión.

```
faecca5 (HEAD -> master) Se añade un fichero y se cambia otro
32e58ea Commit directo
017b1f1 Primera versión
```

5.9. Etiquetar versiones (*tag*)

Se pueden etiquetar versiones con `git tag`. Resulta más cómodo referirse a una versión por una etiqueta que por el código interno que le asigna Git.

Una manera sencilla de añadir una etiqueta a una versión, por ejemplo la 32e58 anterior, es la siguiente:

```
$ git tag -a unaversión -m "Una versión de prueba" 32e5
```

-a significa *annotate*, y es para añadir la etiqueta que viene a continuación. La opción -m es para dar un texto descriptivo, y por último viene el identificador de la versión que se quiere etiquetar.

En adelante, se puede utilizar la etiqueta *unaversión* como identificador de la versión, en sustitución de su código que empieza por 32e58.

Las etiquetas aparecen al lado del identificador de la versión en el listado de versiones o *commits*.

```
faecca5 (HEAD -> master) Se añade un fichero y se cambia otro
32e58ea (tag: unaversión) Commit directo
017b1f1 Primera versión
```

5.10. Obtener versiones del control de revisiones (*checkout*)

El comando `git checkout` obtiene versiones del control de revisiones en el área de trabajo. Con este comando, se obtiene en el área de trabajo la última versión disponible en el repositorio.

Pero se puede especificar un identificador de versión o una etiqueta (*tag*), si existe para la versión. Las versiones existentes, y sus identificadores y etiquetas, se pueden consultar con `git log`.

```
$ git checkout <id_versión>
$ git checkout <etiqueta>
```

Es interesante el mensaje que se muestra al hacer esto:

```
$ git checkout 32e58ea
Nota: actualizando el árbol de trabajo '32e58ea'.
```

Te encuentras en estado 'detached HEAD'. Puedes revisar por aquí, hacer cambios experimentales y confirmarlos, y puedes descartar cualquier commit que hayas hecho en este estado sin impactar a tu rama realizando otro checkout.

Si quieres crear una nueva rama para mantener los commits que has creado, puedes hacerlo (ahora o después) usando -b con el comando checkout. Ejemplo:

```
git checkout -b <nombre-de-nueva-rama>
```

HEAD está ahora en 32e58ea Commit directo

El estado *detached HEAD* significa que la versión actual, con la que se compara el área de trabajo, no es la última de la rama actual. El concepto de rama no se ha explicado todavía. Por ahora, y en relación a la operación que se acaba de realizar, baste decir que la versión actual no es la última, y que no se pueden crear versiones intermedias entre dos ya existentes. Si se hicieran cambios que interesara guardar, habría que hacerlo en una nueva rama, creada a partir de la versión actual de la rama principal. Más adelante se explicará más acerca de las ramas y su gestión.

Si ahora se ejecuta el comando `git status`, se obtiene la siguiente información:

```
$ git status
HEAD desacoplada en 32e58ea
nada para hacer commit, el árbol de trabajo está limpio
```

HEAD es un puntero que apunta la versión con la que se está trabajando que, como se dice, es 32e58ea, porque se acaba de obtener esa versión con `git checkout 32e58ea`. En este caso no es la última, y eso es lo que significa “HEAD desacoplada”, que es una traducción del término *detached HEAD*, que se mostró al hacer `git checkout 32e58ea`.

El comando `git log` indica la posición del puntero HEAD dentro del historial de versiones. El resultado de `git log --oneline` sería ahora el siguiente.

```
32e58ea (HEAD, tag: unaversión) Commit directo
017b1f1 Primera versión
```

Como última versión del proyecto se muestra ahora aquella de la que se ha hecho *checkout* (32e58ea). Esto es lógico, porque se ha restaurado el estado del proyecto cuando esta era la última versión. Pero se pueden mostrar también las más recientes con la opción `--all`. Con `git --oneline --all`, se mostraría:

```
faecca5 (master) Se añade un fichero y se cambia otro
32e58ea (HEAD, tag: unaversión) Commit directo
017b1f1 Primera versión
```

Se puede volver a la cabeza de la rama principal con `git checkout master`. Esto obtiene la última versión de la rama master.

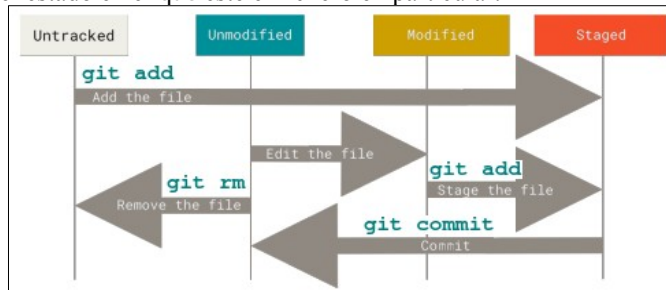
```
$ git checkout master
La posición previa de HEAD era 32e58ea Commit directo
Cambiado a rama 'master'
```

Realmente, no hay otra rama que la rama master, y nunca se ha salido de ella. Ahora, se puede verificar que el puntero HEAD apunta otra vez a la última versión de la rama master con `git --oneline --all`.

```
faecca5 (HEAD -> master) Se añade un fichero y se cambia otro
32e58ea (tag: unaversión) Commit directo
017b1f1 Primera versión
```

5.11. Deshacer cambios

Ahora que ya se sabe hacer casi todo, es el momento de aprender cómo se puede deshacer. Más en concreto: cómo se pueden deshacer los cambios realizados sobre un fichero en particular con respecto a la última versión existente en el repositorio. Eso depende del estado en el que esté el fichero en particular.



Los ficheros en estado *untracked* solo están en el área de trabajo y Git no los está gestionando y, por lo tanto, no proporciona ninguna forma de deshacer cambios que se hayan realizado en ellos. Si esto fuera necesario en alguno, entonces se ha tardado demasiado en empezar a gestionarlo con Git.

5.11.1. Deshacer cambios aún no confirmados (antes de `git commit`) con `git restore`

Estos son cambios realizados en algún fichero gestionado por Git pero aún no confirmados. Los cambios realizados pueden estar en el área de trabajo (en cuyo caso el fichero está en estado *modified*) o en el área de *staging* (en cuyo caso el fichero está en estado *staged*), e incluso en ambos lugares. Como ya se ha visto, el comando `git status` muestra una lista con los ficheros que hay en cada una de estas situaciones.

```
$ git status
En la rama master
Cambios a ser confirmados:
  (usa "git restore --staged <archivo>..." para sacar del área de stage)
    modificado:      f.txt

Cambios no rastreados para el commit:
  (usa "git add <archivo>..." para actualizar lo que será confirmado)
  (usa "git restore <archivo>..." para descartar los cambios en el directorio de trabajo)
    modificado:      hola.txt
```

Piensa

Describe un escenario en el que para un mismo fichero podrían haber cambios tanto en el área de *staging* como en el área de trabajo, y cambios distintos. Es decir, un escenario en el que el comando `git status` mostraría el nombre de un mismo fichero tanto en las dos listas anteriores, a saber: cambios a ser confirmados y cambios no rastreados para el commit.

Como se puede ver, el comando `git status` recomienda el comando `git restore` para deshacer los cambios realizados en los ficheros en estado *staged* y en estado *modified*.

Estado <i>modified</i>	<code>git restore <fichero>...</code>
Estado <i>staged</i>	<code>git restore --staged <fichero>...</code>

Nota: alternativas a `git restore`

En versiones de git previas a la 2.23.0, el comando `git status` recomendaba comandos distintos a `git restore` para deshacer cambios aún no confirmados con `git commit`.

```
$ git status
En la rama master
Cambios a ser confirmados:
(usa "git reset HEAD <archivo>..." para sacar del área de stage)

    modificado:      f.txt

Cambios no rastreados para el commit:
(usa "git add <archivo>..." para actualizar lo que será confirmado)
(usa "git checkout -- <archivo>..." para descartar los cambios en el directorio de
trabajo)

    modificado:      hola.txt
```

El comando `git checkout`, como ya se ha visto, sirve para obtener los contenidos de los ficheros de la última versión. Se puede hacer para todos los ficheros con `git checkout`, o bien para ficheros particulares con `git checkout -- <archivo>...` Los dos guiones (`--`) indican que lo que viene a continuación es un nombre de fichero. Normalmente no es necesario, pero en algunos comandos podría referirse también a otra cosa (por ejemplo, una rama), y entonces los dos guiones antepuestos indican que lo que viene a continuación es el nombre de un fichero.

5.11.2. Añadir nuevos cambios en una versión ya existente

Se pueden añadir nuevos cambios a una versión ya existente. Si después de hacer `git commit` se realizan nuevos cambios y se quiere no crear una nueva versión con estos nuevos cambios, sino incluir estos nuevos cambios en la última versión, se puede utilizar la opción `--amend` de `git commit`. También se puede incluso utilizar esta opción para cambiar los comentarios del `commit`. En el siguiente ejemplo, se usa la opción `-a`, para evitar utilizar previamente un comando `git add` para añadir previamente los cambios al área de *staging*.

```
$ (cambios en el área de trabajo)
$ git commit -a -m "comentario nueva versión"
$ (más cambios en área de trabajo, que se quieren incluir en la misma versión)
$ git commit --amend -a -m "comentario para misma versión, con más cambios"
```

5.11.3. El comando `git reset`

Este es un comando que permite hacer muchas cosas distintas, y que hay que utilizar con precaución. Supongamos que no se está en estado *detached HEAD*, es decir, que el puntero `HEAD` apunta a la última versión de la rama actual, que suponemos que es `master` (en breve se explicará más acerca de las ramas, por el momento se asume que solo hay una, llamada `master`).



`git reset` permite hacer cosas que también se pueden hacer con comandos ya vistos como `git checkout` o `git restore`. Por ejemplo:

- Eliminar cambios realizados en ficheros en estado *staged*, es decir, que ya se han pasado al área de *staging*.

```
$ git reset HEAD
```

Esto también se puede hacer solo para determinados ficheros indicados mediante su nombre.

```
$ git reset HEAD <archivo>...
```

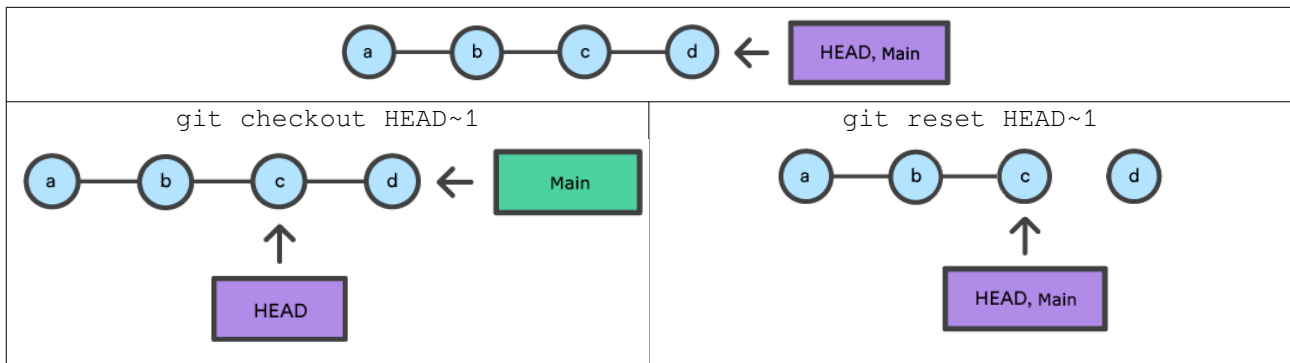
(Nota: como ya se ha visto, también se puede utilizar el comando `git restore` para hacer lo anterior).

Pero el comando `git reset` se puede utilizar para eliminar versiones, cosa que no se puede hacer con comandos vistos anteriormente como `git checkout` o `git restore`.

- Se puede eliminar el último commit realizado, lo que equivale a eliminar la última versión del repositorio, con el comando:

```
$ git reset --soft HEAD~1
```

HEAD~1 se refiere a la versión anterior a la versión a la que apunta el puntero HEAD. Supongamos que esta es la última de la rama actual, lo que será normalmente el caso, a menos que se esté en estado *detached head*. A continuación se muestra la diferencia entre `git reset` y `git checkout`.



En realidad, y como se puede ver, la última versión no se elimina inmediatamente. Pero sí pasa a estar desconectada del resto de versiones. Y se eliminará cuando se ejecute el gestor de basura interno. Esto se hace cada cierto tiempo, que podrían ser 30 días en una configuración por defecto.

En cambio, con `git checkout`, el puntero de la rama actual (en este caso, `main`), sigue apuntando a la última versión, que sigue conectada a la versión posterior. Y dado que el puntero `HEAD` no apunta a la última versión de la rama actual, se está en estado *detached HEAD*.

Como cabría esperar, se puede hacer referencia a versiones anteriores, si existen, con `HEAD~2`, `HEAD~3`, etc.

Con la opción `--soft` no se modifican los contenidos del área de trabajo para que sean conformes con esta versión anterior. Si se quisiera cambiar los contenidos del área de trabajo para hacerlos conformes con esta última versión, habría que utilizar la opción `-hard`.

```
$ git reset --hard HEAD~1
```

5.12. Crear, gestionar y fusionar ramas (*branch*, *switch*, *merge*)

Con lo que se ha visto hasta ahora, las distintas versiones de un proyecto forman una secuencia. Pero es posible crear una nueva rama para un proyecto a partir de una versión cualquiera. Esto se hace con el comando:

```
$ git branch <nombre_rama>
```

Por ejemplo, se puede crear una nueva rama con `git branch rama1`, y después consultar las ramas existentes con `git branch`, que entonces mostraría la siguiente información:

```
* master
  rama1
```

Esto significa que existen dos ramas: `master` y `rama1`, y que la rama actual es `master`.

Para cambiar a una rama, se puede ejecutar el comando:

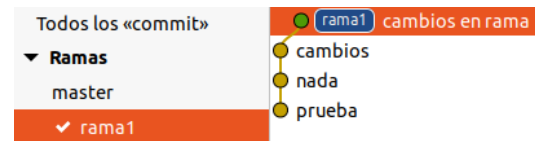
```
$ git switch <nombre_rama>
```

Si se ejecuta ahora `git branch rama1`, el comando `git branch` muestra lo siguiente:

```
master
* rama1
```


Ahora se pueden hacer cambios en esta rama y confirmarlos con `git commit -m "cambios en rama"`. Se puede ver el historial con `git log --oneline`, pero también con la herramienta `gitg`.

```
d0c66e0 (HEAD -> rama1) cambios en rama
4bf32b8 cambios
cbc7814 nada
b3a0b95 prueba
```



Se sigue viendo un historial de versiones lineal porque, aunque se ha creado una nueva versión en la nueva rama `rama1`, no se ha creado ninguna nueva en la rama `master`. Pero la herramienta `gitg` sí muestra visualmente el inicio de la nueva rama. Es de reseñar que como parte del historial de la nueva rama, aparece el historial de la rama `master`, hasta el momento de la bifurcación.

Ahora se puede volver a la rama `master` con `git switch master`, hacer algunos cambios, y crear una nueva versión con `git commit -m "cambio en la rama principal"`. Con `git log` no se mostrará el historial de la otra rama, pero con la opción `--all` se puede mostrar el historial de todas las ramas, y con `--graph` se muestra en forma de grafo.

Con el siguiente comando, se puede ver que el conjunto de versiones ya es una secuencia.

```
$ git log --all --oneline --graph
```

```
* 2742ebf (HEAD -> master) cambio en rama principal
| * d0c66e0 (rama1) cambios en rama
|/
* 4bf32b8 cambios
* cbc7814 nada
* b3a0b95 prueba
```

Y como siempre, se muestra adónde apunta el puntero `HEAD`, que ahora es al final de la rama `master`.

Ahora en cada rama se podría avanzar por separado. Posiblemente interese centrarse, en la nueva rama que se ha creado, centrarse en un aspecto específico y especialmente problemático o difícil, mientras que en la principal se siguen haciendo cambios diversos. Una vez que se ha terminado el trabajo en la nueva rama y todo está bien, puede interesar incorporar los cambios que se han hecho en ella en la rama principal. Es decir, hacer una fusión o *merge* de los cambios realizados en ambas ramas. Esto consiste en un *commit* especial, que crea una nueva versión en la que confluyen ambas ramas, y que incluye los cambios realizados en ambas.

El uso de ramas es habitual en proyectos en los que trabajan varias personas, y más cuanto menos estrecha sea la relación entre ellas. Pero la confirmación (*commit*) de los cambios plantea nuevos problemas, porque se pueden producir conflictos entre los cambios que se hacen en distintas ramas. A continuación se explica cuándo suceden estos conflictos y de qué manera se pueden solucionar.

5.12.1. Cambios en distintas ramas y comparación a tres bandas

Cuando sobre un fichero F se realizan cambios en ramas independientes y finalmente se quieren fusionar, se tiene la siguiente situación:

- La última versión (n) del fichero en el control de versiones es un fichero $F[n]$. Esta es la versión del fichero presente en la versión del proyecto a partir del cual este se bifurca (*fork*), con la creación de una nueva rama (*branch*).
- El programador A ha realizado determinados cambios (Δ_a), y tiene $F[n] + \Delta_a$ en su área de trabajo.
- El programador B ha realizado determinados cambios (Δ_b), y tiene $F[n] + \Delta_b$ en su área de trabajo.

Para comparar los cambios realizados por dos programadores en un fichero con respecto a la última versión común, se puede realizar una **comparación a tres bandas** (*3-way comparison*), con programas como `kdiff3`. En el siguiente ejemplo se muestra a la izquierda la última versión común, es decir, $F[n]$. Después el fichero con los cambios realizados por A (es decir, $F[n] + \Delta_a$), y después los cambios realizados por B (es decir, $F[n] + \Delta_b$).



Los cambios realizados por ambos programadores se muestran resaltados en rojo. En este caso particular, no hay conflictos entre los cambios realizados por ambos programadores, porque han realizado cambios en una misma línea ni en bloques de líneas que se solapan.

Una herramienta clásica para mostrar diferencias entre ficheros es `diff`. Es un comando al que se le pasan dos ficheros, y muestra los cambios realizados sobre el primero para obtener el segundo. A continuación se muestra el resultado de utilizarlo para comparar la versión base con cada una de las dos nuevas versiones: la del programador A y la del programador B. Se puede verificar que la información mostrada por `diff` corresponde exactamente a los cambios realizados por ambos programadores.

diff base fichero_de_A (Δ_a)	diff base fichero_de_B (Δ_b)
<pre> 19,34d18 < < @BeforeAll < public static void setUpClass() { < } < < @AfterAll < public static void tearDownClass() { < } < < @BeforeEach < public void setUp() { < } < < @AfterEach < public void tearDown() { < } < 49c33 < < --- > assertEquals("ERROR: pies no entero.\n", systemErrRule.getLog()); </pre>	<pre> 14c14,17 < < --- > /** > * > * Documentación de SystemRules: > https://stefanbirkner.github.io/system-rules/ > */ 41a45,46 > @Rule > public final ExpectedSystemExit exitRule = ExpectedSystemExit.none(); 47a53 > exitRule.expectSystemExitWithStatus(1); </pre>

La salida de `diff` es una secuencia de diferencias. Cada diferencia se encabeza con una línea en la que se indica:

- Líneas afectados en el fichero inicial. Si es una, aparece su número. Si son varias, aparece el número de la primera y de la última, separados por comas.

- Tipo de cambio realizado:
 - d (delete): borrado.
 - c (change): cambio.
 - a (add): inserción.
- Líneas del fichero final en las que están los cambios realizados.

Después viene la descripción del cambio realizado. En el caso de un borrado (d) o inserción (a), se muestran las líneas borradas o insertadas. En el caso de un cambio (c), se muestran las líneas del fichero original y las líneas por las que se sustituyen en el fichero final.

No hay **conflictos** porque los cambios realizados sobre el fichero original por ambos programadores se han hecho en rangos de filas que no se solapan. El primer programador ha realizado cambios en los rangos de filas del fichero original (19 a 34) y (49), y el segundo en los rangos (14), (41) y (47).

5.12.2. Fusión de cambios (*merge*)

Para fusionar los cambios hay que hacer un *commit* especial, en el que se crea una nueva versión del proyecto que contiene los cambios realizados en ambas ramas. Para un fichero en el que a partir de una versión común $F[n]$ se han realizado cambios Δ_a y Δ_b en respectivas ramas a y b, hay que guardar en el repositorio una nueva versión de un fichero que contenga los cambios hechos en ambas ramas.

$$F[n+2] = F[n] + \Delta_a + \Delta_b = F[n] + \Delta_b + \Delta_a$$

Esto no plantea ningún problema si no hay conflictos entre los cambios realizados por ambos programadores, es decir, entre Δ_a y Δ_b . Como ya se ha visto, esto será así cuando no han realizado cambios sobre la misma línea ni en dos bloques de líneas que se solapan. El proceso de incorporar distintos conjuntos de cambios con respecto a un mismo fichero original se llama en inglés *merge* (o fusión). Por supuesto, nada garantiza que todos los cambios introducidos vayan a funcionar correctamente juntos. Pero esa es otra cuestión.

Existen comparadores a tres bandas que permiten realizar la operación *merge*. Se muestra a continuación cómo se haría para el ejemplo anterior con `k3diff`. Existe una opción que muestra una ventana de fusión, en la que se incorporan los cambios realizados por ambos programadores, es decir, $F[n] + \Delta_a + \Delta_b$.

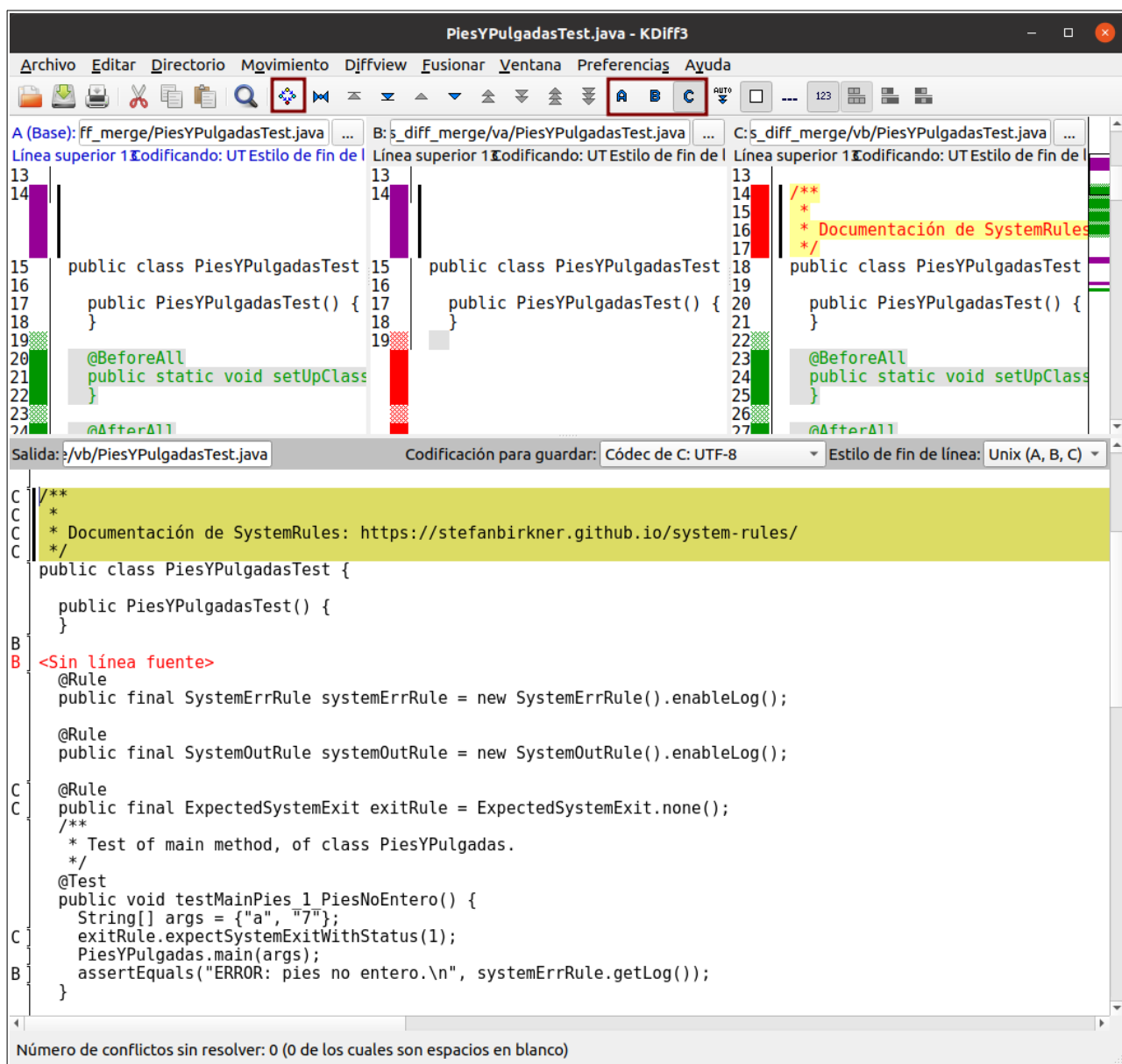


Ilustración 4.12: Merge o fusión a tres bandas (3-way merge)

Los cambios correspondientes a las distintas versiones se etiquetan con:

- A. Correspondientes a la versión base.
- B. Los realizados por el primer programador (A). Estos pertenecen, por tanto, a Δ_a .
- C. Los realizados por el primer programador (B). Estos pertenecen, por tanto, a Δ_b .

En la ventana de *merge* o fusión, como se ve, se incluyen por defecto los cambios realizados por el programador A (etiquetados como B) y por el programador B (etiquetados como C). Pero hay tres botones en la parte superior, que se muestran resaltados, y que permiten, para cada cambio, seleccionar entre el contenido de la versión inicial (A) o de las dos nuevas versiones (B y C).

5.12.3. Conflictos de fusión (*merge*)

Con frecuencia hay conflictos entre los cambios realizados por dos programadores sobre una misma versión de un fichero. Es decir, ambos realizan modificaciones sobre una misma línea o sobre bloques de líneas que se superponen, como en el ejemplo siguiente. Es una variación del ejemplo anterior en el que, en dos líneas diferentes, se han hecho cambios distintos en las versiones de los programadores A y B. En la ventana de *merge* aparecen ambos conflictos, y

en la comparación de código arriba aparece el detalle del segundo. Se resaltan los cambios con el mismo color en ambas ventanas para ayudar a identificarlos y relacionarlos. En este caso, el programador puede elegir los cambios de un programador u otro, e incluso editar manualmente la línea.

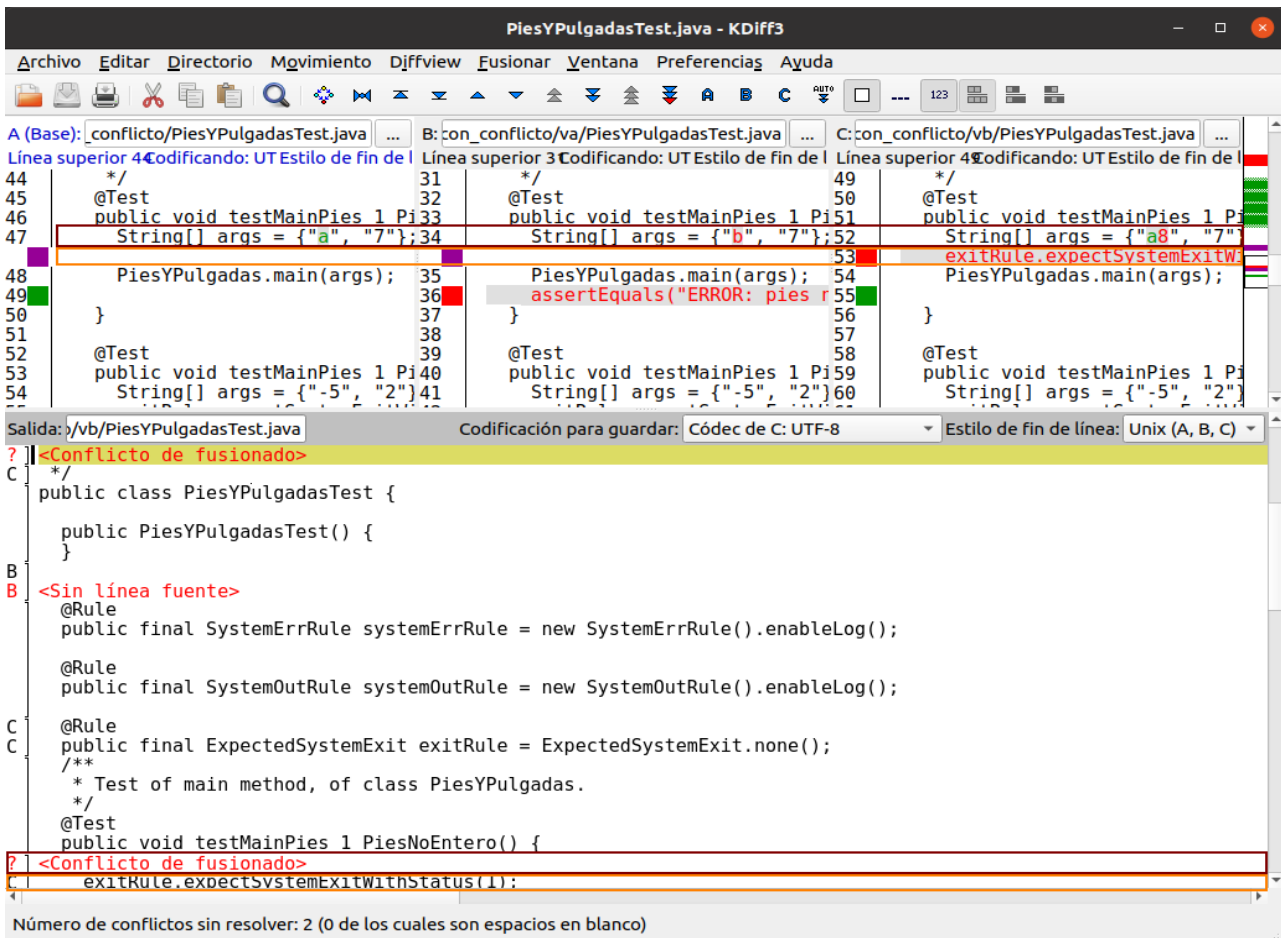


Ilustración 4.13: Comparación a tres bandas con conflictos y merge (fusión) de los cambios

Cuando hay conflictos, es necesario resolverlos manualmente. En la parte de abajo de la ilustración anterior se ve una ventana de edición. Sus contenidos se inician con la versión original, y en ella se pueden incorporar los cambios realizados por cualquiera de ambos programadores, y además se puede editar manualmente.

5.13. Fusión de cambios realizados en ramas (git merge)

Volviendo al caso que se planteaba inicialmente, se tiene un proyecto en el que se ha creado una rama `rama1` a partir de la rama inicial `master`, y después se han creado nuevas versiones en ambas.

Con el siguiente comando, se puede ver el conjunto de versiones, que ya no es una secuencia.

```
$ git log --all --oneline --graph
```

```
* 2742ebf (HEAD -> master) cambio en rama principal
| * d0c66e0 (rama1) cambios en rama
|/
* 4bf32b8 cambios
* cbc7814 nada
* b3a0b95 prueba
```

La última versión común es 4bf32b8. Si llamamos a esta versión C, entonces se tiene que:

- $2742ebf = C + \Delta_m$, donde Δ_m son los cambios que se han hecho en la rama `master`.
- $D0c66e0 = C + \Delta_b$, donde Δ_b son los cambios que se han hecho en la nueva rama `rama1`.

Cuando se ejecuta el comando `git merge otra_rama` desde una rama, se crea una nueva versión que incluye los cambios realizados en la rama actual (HEAD) y también los realizados en `otra_rama`.

```
$ git merge otra_rama
```

Esta nueva versión será igual, por tanto, a $C + \Delta_m + \Delta_b$. Esto es un *merge* a tres bandas en el que, como se vio en una sección previa, pueden o no haber conflictos entre los cambios realizados en ambas ramas, Δ_m y Δ_b .

A continuación se muestra un ejemplo de la salida de este comando cuando no hay conflicto entre los cambios en ambas ramas. Entonces se hacen automáticamente los cambios en los ficheros del área de trabajo, y se muestran detalles acerca de los ficheros que se han cambiado para hacer el *merge* o fusión, y cuántos y de qué tipo son. Por supuesto, se puede utilizar el comando `git diff` para ver cuáles son exactamente los cambios en cada fichero.

```
Merge made by the 'recursive' strategy.
a.txt | 4 +---
1 file changed, 1 insertion(+), 3 deletions(-)
```

La salida del comando `git log --all --oneline --graph` sería entonces la siguiente. En ella se puede ver la nueva versión creada que finaliza la rama `rama1`. En la nueva versión que se ha creado (`dd8e47a`) se han incorporado sus cambios.

```
* dd8e47a (HEAD -> master) Merge branch 'rama1': prueba de merge sin conflicto
S
| \
| * d0c66e0 (rama1) cambios en rama
| * | 2742ebf cambio en rama principal
|/
* 4bf32b8 cambios
* cbc7814 nada
* b3a0b95 prueba
```

Herramienta gitg

Existen diversas herramientas que permiten ver las distintas ramas de un proyecto de manera gráfica e interactiva. Entre ellas está `gitg`. También `gitk`.

En la parte izquierda muestra las distintas ramas existentes. Se pueden ver todas, o bien se puede elegir una rama en particular. Si se selecciona la rama creada anteriormente, `rama1`, se puede ver su historial, que comienza con la primera versión de la rama a partir de la cual se creó (`master`), y termina en la versión que se fusionó (con `git merge`) desde la rama `master`.



En el ejemplo anterior, la herramienta Git ha podido fusionar automáticamente los cambios en ambas ramas (Δ_m y Δ_b) porque no había ningún conflicto entre ellos. Para cada fichero para el que hay algún conflicto, se crea, como resultado de la fusión fallida, un fichero con el mismo nombre y con un bloque como el siguiente para cada conflicto.

```
(Líneas comunes antes del conflicto)
<<<<<<< (cambios en rama master, desde la que se hace merge)
La resolución de conflictos es complicada;
Dejémoslo estar.
|||||
La resolución de conflictos es complicada.
=====
```

Con Git es fácil.

>>>>>> (cambios en otra rama, cuyos cambios se quieren incorporar)

(Líneas comunes después del conflicto)

Donde:

- <<<<<< precede a los cambios en la rama master (incluidos en Δ_m).
- >>>>>> precede a los cambios en la otra rama (incluidos en Δ_b).
- | | | | | Precede al contenido de la versión común más reciente (C).

Estos ficheros se pueden editar manualmente, para sustituir cada bloque de los anteriores con el contenido apropiado, de forma que finalmente el fichero quede con los contenidos que debe haber en la nueva versión, en la que confluirán ambas ramas.

Otra opción muy interesante es ejecutar el comando

```
$ git mergetool
```

Este invocará la herramienta configurada, como se explica a continuación, para cada fichero para el que ha habido conflicto. Si es, por ejemplo, `kdiff3`, esta herramienta sabe interpretar adecuadamente los contenidos en el formato anterior, muestra el contenido de la versión base y de la de cada rama, y proporciona herramientas para resolver fácilmente los conflictos. Ya se explicó el planteamiento general de una fusión o *merge* a tres bandas, y el funcionamiento de esta herramienta, en una sección previa.

Configuración de la herramienta para *merge* a tres bandas

La siguiente opción de configuración, a la que se le puede asignar un valor con `git config`, o modificando directamente los ficheros de configuración, indica la herramienta a utilizar para realizar un *merge* a tres bandas con el comando `git mergetool`.

<code>merge.tool</code>	Herramienta a utilizar para <i>merge</i> a tres bandas con <code>git mergetool</code> .
-------------------------	---

Con Git se pueden utilizar directamente las siguientes herramientas: entre otras: `kdiff3`, `gvimdiff`, `meld`, y `tkdiff`.

@PEND: comandos que muestran detalle de lo incluido en una versión (? `git show ...`)

@PEND: comando `blame`

Actividad 3

(se recomienda realizar esta práctica sin rellenar el documento que se pide, sencillamente siguiendo los pasos. Una vez realizada, y entendido lo que se ha hecho, repetir la práctica, rellenando el documento para documentar y explicar todo lo hecho en cada paso)

Debes entregar un fichero de OpenOffice Writer explicando paso a paso cómo haces esta práctica. En él debes incluir al menos los comandos que ejecutas, y capturas de pantallas con la información que muestran. Para algunos de los pasos, puede ser necesario, o en todo caso, se puede utilizar, más de un comando.

1. Crea un directorio `prac_final_Git_apellido1_apellido2_nombre`, donde `apellido1`, `apellido2` y `nombre` son tus apellidos y nombre. Cámbiate a él. Esta es el área de trabajo. Crea un

repositorio de Git con `git init`.

2. Configura `kdiff3` como la herramienta para *merge* de Git (opción `merge.tool`), y también como herramienta para *diff* (opción `diff.tool`). Las configuraciones (esta y cualquier otra que haya que hacer) debe ser solo para el repositorio actual, no global a nivel de usuario.
3. Muestra esta configuración, y el fichero en el que están, con las opciones apropiadas de `git config`.
4. Crea un fichero `trad.txt`, con el siguiente contenido:

```
[1]
en:keyboard
[2]
en:source code
[3]
en:compiler
[4]
en:library
[5]
en:repository
[6]
en:update
```

5. Crea una primera versión del proyecto que contenga este fichero con estos contenidos, con la etiqueta `textos_orig_apellido1_apellido2_nombre`, donde *apellido1*, *apellido2* y *nombre* son tus apellidos y nombre.
6. Muestra el historial de versiones con `git log`. Debe aparecer una sola versión, en la rama `master`.
7. Crea una nueva versión que añada después de cada línea con un texto del tipo “en: término” una línea con el texto “es:”. Muestra el historial de versiones con `git log`. Deben aparecer las dos versiones anteriormente descritas en la rama principal (`master`).
8. Muestra las diferencias de los contenidos de las dos versiones anteriores con `git diff`, y con `git difftool`.
9. Crea una nueva rama `traductorexterno`.
10. Crea una nueva versión en la rama principal (`master`), en la que, en cada una de las líneas “es:”, se añade la traducción al español del término en inglés anterior, pero solo para los números pares.
11. Crea una nueva versión en la rama `traductorexterno`, en la que se haga igual que se ha hecho en el paso anterior, pero para los números impares. Y además, también para el término “library”. En ambas ramas debe haber una traducción distinta para este término. En una de ellas debe ser “biblioteca”, y en la otra podría ser “librería”.
12. Haz un *merge* de la rama `traductorexterno` con la rama `master`, con el comando `git merge`. Debe haber al menos un conflicto, para el fichero `trad.txt`, para la línea donde está la traducción al español de “library”. Revisa los contenidos de este fichero, e incluye una captura de pantalla con sus contenidos.
13. Ejecuta `git mergetool`. Debe ejecutarse `kdiff3`, de acuerdo con la configuración que se ha hecho en un paso anterior, y mostrarse el conflicto en la línea donde está la traducción para “library”. Utiliza las herramientas de `kdiff3` para resolver el conflicto, a favor de “biblioteca”, graba los cambios en la ventana de *merge* (abajo), y toma una captura de pantalla antes de salir.
14. Para verificar que todo está correcto, con el comando `git log --oneline --all --graph`, deben verse ambas ramas y la nueva versión que se ha creado al fusionarlas en la rama `master`. También puedes verificarlo con las herramientas `gitg` o `gitk` (que seguramente tendrás que instalar previamente).
15. Añade en una nueva versión del proyecto, con etiqueta `entrega`, el documento de OpenOffice con las explicaciones de cómo has hecho toda la práctica, incluyendo capturas de pantalla. Debe estar en un directorio `doc`.
16. Crea un fichero de tipo `zip` o `gzip` con los contenidos del directorio `prac_final_Git_apellido1_apellido2_nombre` que has creado al principio, y entrega este fichero.

@PEND: GitHub.

Referencias:

- <https://www.atlassian.com/es/git/tutorials/undoing-changes/git-reset>
- @PEND: añadir algunas más.