Flujos de trabajo en Git (1ª Parte… y puede que última)

Flujo de trabajo básico (lineal)

El flujo de trabajo más elemental consiste en usar `Git` para guardar la historia de cambios de un proyecto de manera que sea posible recuperar su estado en un punto arbitrario del desarrollo. Un caso típico es el siguiente: durante el desarrollo del proyecto se va modificando el código del mismo; en un momento determinado, se descubre que una cierta modificación ha introducido errores; si se ha tenido el cuidado de ir almacenando las distintas fases del proyecto usando `Git`, podemos revertir la situación y eliminar los errores introducidos. Evidentemente, sólo podremos regresar a un punto que hayamos tenido el cuidado de almacenar…

Supongamos un proyecto consistente en un programa, `p1`, que analiza una señal de audio para extraer ciertas estadísticas de la misma. En el proyecto tenemos los ficheros siguientes:

- `p1.c`: fichero con el código del programa principal.
- `fic_wave.c`: fichero con las funciones de acceso y lectura de señales de audio desde ficheros con formato Microsoft Wave. Este fichero está acompañado de la cabecera `fic_wave.h` con los prototipos de las funciones incluidas en él.
- `pav_analysis.c`: fichero con las funciones de análisis de la señal de audio. Acompañado de la cabecera `pav_analyisis.h`.
- `Makefile`: fichero usado por el programa `make` para mantener el programa y compilar y/o enlazar las partes necesarias para asegurar su puesta al día.
- `README.md`: fichero de texto *mágico* con la información del proyecto. De hecho, la extensión `.md` indica que el fichero responde al formato *markdown*, un formato que permite tratar el fichero como uno de texto normal y corriente, pero con indicaciones de cómo debe ser visualizado en caso de ser abierto por un programa adecuado.

Todos estos ficheros están ubicados en el directorio `P1`. Podemos visualizar el contenido de este directorio con la orden `tree` de Linux:

```
albino:~/PAV$ tree P1
P1
├── Makefile
├── README.md
├── fic_wave.c
├── fic_wave.h
├── p1.c
├── pav_analysis.c
└── pav_analysis.h

0 directories, 7 files
```

Inicialización del usuario para trabajar con Git

En la gestión de un repositorio, Git utiliza distintos datos del usuario. Entre ellos, dos son imprescindibles y Git no permitirá el trabajo con el repositorio si no están definidos previamente: el nombre del usuario y su dirección de correo electrónico. Estos datos pueden proporcionarse de manera independiente para cada proyecto o de manera global para todos los proyectos gestionados por un usuario Linux.

Por ejemplo, para definir el nombre y correo electrónico por defecto de un usuario Linux, ejecutaríamos las órdenes siguientes:

```
albino:~/PAV/P1$ git config --global user.email "albino.nogueiras@upc.edu"
albino:~/PAV/P1$ git config --global user.name "Albino Nogueiras Rodríguez"
```

Esta información por defecto se almacena en el fichero oculto del directorio raíz del usuario `~/.gitconfig`.

Alternativamente, en lugar de ejecutar los comandos anteriores, se puede editar directamente este fichero. En mi caso, el contenido del fichero es el siguiente:

```
albino:~/PAV/P1$ cat ~/.gitconfig
[user]
        email = albino.nogueiras@upc.edu
        name = Albino Nogueiras Rodríguez
[core]
        editor = vim
```

También se puede tener una identificación individualizada por proyecto omitiendo la opción `--global` al invocar el comando `git config`, o escribiendo los datos deseados en el fichero `.git/gitconfig` del repositorio.

Inicialización del directorio para trabajar con Git

Lo primero que hemos de realizar para mantener un proyecto usando Git es inicializar su repositorio a partir del directorio en el que se ubica el proyecto. Esto lo realizamos con la orden siguiente:

```
albino:~/PAV/P1$ git init
Initialized empty Git repository in /mnt/d/albino/UbuntuOnWindows/PAV/P1/.git/
```

Como el mensaje indica, esta orden crea un repositorio vacío en el directorio `.git`. Recuérdese que los nombres empezados por punto son tratados de manera especial por el Shell: en principio no se visualizan con los comandos habituales, como `ls`, `tree`, etc., pero sí pueden visualizarse con las opciones adecuadas, como `ls -a`, `tree -a`, etc. Aunque el repositorio está, a todos los efectos, vacío, el directorio `.git` contiene un montón de ficheros y subdirectorios que utiliza Git para mantener el proyecto y que, en general, no deben ser accedidos por el usuario salvo para fisgar un poco (o aprender los intríngulis de Git… pero sin tocar nada, porque eso puede ser realmente peligroso)

Para ver el estado (vacío) del repositorio, ejecutamos la orden `git status`:

```
albino:~/PAV/P1$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        Makefile
        README.md
        fic_wave.c
        fic_wave.h
        p1.c
        pav_analysis.c
        pav_analysis.h

nothing added to commit but untracked files present (use "git add" to track)
```

La salida nos indica diversas cosas:

- Estamos en la rama (*branch*) principal o maestra (*master*). Hablaremos más delante de las bifurcaciones y cómo trabajar con más de una rama de nuestro proyecto (por ejemplo, si distintos desarrolladores introducen cambios simultáneamente). Por ahora, ésta va ser nuestra única rama de trabajo.
- Todavía no hay estados almacenados (*commits*). Un commit es una copia de todos los ficheros mantenidos por Git en un momento dado. En todo momento, estaremos trabajando (modificando el contenido de sus ficheros, por ejemplo) en un commit determinado, aunque, por ahora, aún no tenemos ninguno.
- Git también nos informa de que hay una serie de ficheros que están en el directorio del proyecto, pero que no se han incorporado al repositorio (no se están siguiendo por Git). Estos ficheros aparecen en rojo para remarcar el hecho de que no están incorporados.
- Finalmente, nos indica que podemos añadir estos ficheros al repositorio, y cómo hacerlo.
    - Ya de paso, también podría decirnos cómo evitar que Git los siga, pero eso lo veremos más adelante.

Añadir ficheros (y/o directorios) al proyecto

Como indicaba el mensaje anterior de la orden `git status`, la orden para añadir ficheros y/o directorios al proyecto para hacer que Git realice su seguimiento es `git add`. Esta orden realiza una función básica: indicar a Git que actualice el índice de ficheros seguidos con el contenido actual del directorio del proyecto (y, en caso de que los hubiera, sus subdirectorios). Como casi todos los comandos de Git, el comando puede hacer muchas más cosas. Puede consultarse usando la orden `git help add`.

Añadimos el directorio actual al repositorio de Git:

```
albino:~/PAV/P1$ git add .
albino:~/PAV/P1$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   Makefile
        new file:   README.md
        new file:   fic_wave.c
        new file:   fic_wave.h
        new file:   p1.c
        new file:   pav_analysis.c
        new file:   pav_analysis.h
```

Vemos que los ficheros que antes aparecían en rojo (indicando que no se estaba realizando su seguimiento), aparecen en verde. No obstante, seguimos sin tener commits. Esto es así porque `git add` no confirma los cambios del repositorio, sino que sólo actualiza el índice de los ficheros seguidos. Estos ficheros quedan almacenados en una etapa intermedia denominada *área de preparación*.

Es importante remarcar el hecho de que, en general y en todo momento, los ficheros del proyecto estarán en tres tipos de área:

- El área de trabajo, que es, simplemente, el directorio en el que tenemos los ficheros sobre los que trabajamos.
- El área de preparación, que es el área donde vamos almacenando los cambios realizados en los ficheros.
- Las áreas de confirmación o commits, donde se almacenan los distintos estados por los que pasa el proyecto.

Para confirmar los cambios realizados en el repositorio, y tener así nuestro primer commit, invocamos la orden git commit. Esta orden exige introducir un mensaje explicativo del commit. Si el mensaje no se introduce en la línea de comandos, se abrirá el editor preferido por el usuario (o el preferido por Git, si el usuario no ha indicado sus preferencias) para que éste introduzca el mensaje explicativo, que no puede estar vacío. Si el mensaje se deja vacío, el commit se aborta.

```
albino:~/PAV/P1$ git commit -m "Ficheros originales de P1"
[master (root-commit) a29a316] Ficheros originales de P1
 7 files changed, 104 insertions(+)
 create mode 100644 Makefile
 create mode 100644 README.md
 create mode 100644 fic_wave.c
 create mode 100644 fic_wave.h
 create mode 100644 p1.c
 create mode 100644 pav_analysis.c
 create mode 100644 pav_analysis.h
```

Cada vez que hacemos un commit, Git nos informa de qué ficheros del índice se han modificado y de qué modo. En este caso, nos indica que se han creado entradas para cada uno de los siete ficheros del proyecto. El modo `100644` indica que el fichero es un fichero normal no ejecutable (`100`), y que el usuario tiene permiso de lectura y escritura (`6`), mientras que el resto de usuarios del mismo grupo que el usuario y el resto de usuarios de otros grupos sólo tienen permiso de lectura (`4`).

Fijémonos en que, en la primera línea, Git nos indica que estamos en el primer commit, o commit raíz (*root*), de la rama master. También nos indica un valor hexadecimal (`a29a316`). Este valor se corresponde con los siete primeros dígitos hexadecimales de la clave SHA1 del commit. Cada commit se identifica con una clave hash SHA1 de veinte bytes (cuarenta dígitos hexadecimales), calculada a partir de datos, pretendidamente únicos, del mismo: el árbol del proyecto; la clave del commit padre del actual; el autor, el ejecutor y la fecha del commit; y el mensaje explicativo del commit. Esta clave hash es fundamental, porque sirve para identificar unívocamente cada commit (aunque podemos humanizarlos usando etiquetas…).

Es muy raro que, en un mismo proyecto, haya ambigüedad si sólo nos quedamos con unos pocos de los dígitos de la clave hash. Por este motivo, Git suele mostrar sólo los siete primeros. También es posible, para el usuario, referirse a un commit concreto usando sólo unos pocos dígitos. En este caso, Git suele conformarse con los cuatro primeros, aunque eso dependerá siempre de que no haya ambigüedad.

Podemos ver la historia del proyecto con la orden `git log`. Esta orden, que también permite muchas opciones avanzadas que podemos ver con `git help log`, nos da la sucesión de commits necesarios para llegar al actual:

```
albino:~/PAV/P1$ git log
commit a29a3164872ca19ce9a90254d5f7f362fddee49e (HEAD, master)
Author: Albino Nogueiras Rodriguez <albino.nogueiras@upc.edu>
Date:   Thu Aug 22 21:21:57 2019 +0200

    Ficheros originales de P1
```

Como apenas hemos realizado un commit, la historia mostrada por `git log` es muy escueta. En la primera línea nos informa de cuál es el commit actual, con su clave hash entera, la rama del proyecto (`master`) y la posición en la rama (`HEAD`). La posición más actual en la rama se indica siempre con la etiqueta `HEAD`. Este es un valor definido por Git. Más adelante veremos otros casos en los que Git asigna un valor por defecto a una posición concreta en el proyecto, pero `HEAD` es el más importante; al menos, por ahora.

Git log también nos informa de quién y cuándo realizó el commit, y el mensaje explicativo del mismo.

Desarrollo normal del proyecto

A partir de este momento, nuestro proyecto estará gestionado por Git. Si no queremos o necesitamos complicarnos más la vida, actuaremos como lo haríamos con cualquier otro proyecto, modificando y depurando los ficheros hasta conseguir la funcionalidad deseada.

El modo de interacción más sencillo con Git consiste en, cada vez que se introduce una modificación de calado en el proyecto, realizar un commit del mismo. Esta dinámica nos permitirá trabajar con bastante libertad, ya que sabremos que, aunque suceda algún desastre en el desarrollo, siempre tendremos la posibilidad de recuperar el trabajo perdido.

Los commits deberían estar espaciados de una manera razonable: sólo podremos recuperar los estados almacenados con ellos, por lo que será importante realizar uno cada vez que hagamos una modificación importante; pero tampoco se trata de realizar un commit cada vez que se cambia una línea de código, porque eso dificultaría innecesariamente la localización de los estados realmente importantes. El hecho de que el comando `git commit` exija un mensaje explicativo para cada commit puede servir para calibrar su frecuencia: si no sabemos qué poner en uno, o lo que ponemos es muy trivial, seguramente el commit es superfluo; si tenemos que escribir un mensaje muy complejo, explicando muchas cosas diferentes en él, seguramente hubiera sido mejor haber hecho algún commit intermedio.

Desarrollo incremental de P1

Empezamos el trabajo con el proyecto P1. Lo primero que hacemos es escribir las funciones de `pav_analysis.c` para que calculen las características de la señal de audio. Lo típico será que escribamos una de ellas, por ejemplo la función `compute_power()`, que calcula la potencia media de la trama de señal; corrijamos el código hasta que éste se compile correctamente; y comprobemos su funcionamiento con una señal de prueba. Este proceso lo repetiremos hasta que el funcionamiento de la función sea el correcto. Y el proceso completo lo realizaremos también para las funciones `compute_am()` y `compute_zcr()`.

En un caso como éste, tenemos dos opciones: realizar un commit para cada una de las tres funciones, o realizar uno único para las tres. Ambas opciones son perfectamente válidas: es fácil pensar en los mensajes explicativos si realizamos tres commits individuales; aunque lo cierto es que son cambios menores que no interaccionan entre sí y

que afectan a un único fichero, así que, si el proyecto fuera más grande, tendría más sentido agrupar los cambios de las tres funciones en un único commit.

Para conseguir un flujo de trabajo algo más complejo, optamos por realizar un commit distinto por cada función. Empezamos por `compute_power()`:

- Escribimos el código de `compute_power()` en el fichero `pav_analysis.c`.
  - Como la función llama a funciones de la librería aritmética `math.h`, también hemos tenido que modificar `Makefile` para enlazar (*link*) el programa con ella.
- Depuramos el código hasta conseguir que compile y enlace sin errores, y realice el cálculo de la potencia media correctamente.
- Una vez conseguido que `compute_power()` funcione correctamente, confirmamos los cambios con `git commit`.

```
albino:~/PAV/P1$ make
cc    -c -o p1.o p1.c
cc    -c -o pav_analysis.o pav_analysis.c
cc    -c -o fic_wave.o fic_wave.c
cc   p1.o pav_analysis.o fic_wave.o  -lm -o p1
albino:~/PAV/P1$ git commit -a -m "Escrita compute_power()"
[master b766b73] Escrito compute_power()
 2 files changed, 4 insertions(+)
```

Vemos que Git nos informa de que ha habido cambios en dos ficheros (`pav_analysis.c` y `Makefile`) que, en total, representan la inserción de cuatro líneas (hay trampa… en verdad tendría que haber más modificaciones, pero no he hecho realmente lo que digo… sólo he añadido una línea de comentario).

Podemos ver el estado del repositorio después de estos cambios:

```
albino:~/PAV/P1$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        fic_wave.o
        p1
        p1.o
        pav_analysis.o

nothing added to commit but untracked files present (use "git add" to track)
```

Vemos que Git nos informa de que hay ficheros nuevos que no se están siguiendo. En verdad, esto ya está bien: son los ficheros generados al compilar el programa y, como se generan automáticamente, no es necesario realizar su seguimiento. Para informar a Git de que no estamos interesados en seguir estos ficheros, los podemos incluir en el fichero (oculto) `.gitignore`. Este fichero admite, entre otras cosas, *wildcards*, así que un ejemplo posible es:

```
albino:~/PAV/P1$ cat .gitignore
*.o
p1
```

Si ahora ejecutamos `git status`, continuamos teniendo cosas no gestionadas por Git… el propio fichero `.gitignore`, que hemos de añadir al repositorio.

```
albino:~/PAV/P1$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .gitignore

nothing added to commit but untracked files present (use "git add" to track)
albino:~/PAV/P1$ git add .gitignore
albino:~/PAV/P1$ git status
On branch master
```

```
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   .gitignore
```

Ahora, ya tenemos el repositorio completo, siguiendo los ficheros realmente importantes e ignorando los que podemos generar de manera automática. `.gitignore` está pendiente de confirmar, pero eso ya lo haremos junto con la siguiente modificación del código.

Realizamos las mismas operaciones al modificar `pav_analysis.c` para añadir el código de `compute_am()` y `compute_zcr()`. Para añadir un poco de variedad, en este caso vamos a pasar el fichero `pav_analysis.c` a la zona de preparación después de cada cambio, y realizamos el commit al final.

```
albino:~/PAV/P1$ # Escribimos la función compute_am()
albino:~/PAV/P1$ git add pav_analysis.c
albino:~/PAV/P1$ # Escribimos la función compute_zcr()
albino:~/PAV/P1$ git add pav_analysis.c
albino:~/PAV/P1$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   .gitignore
        modified:   pav_analysis.c

albino:~/PAV/P1$ git commit -m "Escritas compute_am() compute_zcr()"
[master 96e6f6a] Escritas compute_am() compute_zcr()
 2 files changed, 6 insertions(+)
 create mode 100644 .gitignore
albino:~/PAV/P1$ git status
On branch maste6
nothing to commit, working tree clean
```

Podemos ver el historial de cambios del proyecto con la orden `commit log`:

```
albino:~/PAV/P1$ git log
commit 96e6f6ab3d5ee4cd9a476e7e128a2cccaa602890 (HEAD -> master)
Author: Albino Nogueiras Rodríguez <albino.nogueiras@upc.edu>
Date:   Thu Aug 22 21:25:52 2019 +0200

    Escritas compute_am() compute_zcr()

commit 2efee58d945d47d48a2e3e3b01beb932c31aa0b8
Author: Albino Nogueiras Rodríguez <albino.nogueiras@upc.edu>
Date:   Thu Aug 22 21:24:08 2019 +0200

    Escrita compute_power()

commit a29a3164872ca19ce9a90254d5f7f362fddee49e
Author: Albino Nogueiras Rodríguez <albino.nogueiras@upc.edu>
Date:   Thu Aug 22 21:21:57 2019 +0200

    Ficheros originales de P1
```

Una alternativa más sucinta, y a menudo más clara, es usando la opción `–pretty=oneline`:

```
albino:~/PAV/P1$ git log --pretty=oneline
96e6f6ab3d5ee4cd9a476e7e128a2cccaa602890 (HEAD -> master) Escritas
compute_am() compute_zcr()
b766b736ed3102a9f2a3b058f1f4bb1bb9082c15 Escrita compute_power()
a29a3164872ca19ce9a90254d5f7f362fddee49e Ficheros originales de P1
```

Fijémonos en que cada commit está identificado por su clave hash SHA1 completa de 20 bytes (40 dígitos hexadecimales). Fijémonos, también, en que el orden de presentación es de más nuevo a más antiguo. Esto es así porque Git guarda la información de dónde está la versión más actual del proyecto (**HEAD**) y cuál es su predecesor. A

partir de ahí reconstruye el resto de la historia a partir del predecesor de cada uno de los estados, hasta llegar al inicio del proyecto. En verdad, esto se puede complicar mucho más cuando el proyecto tiene bifurcaciones. En ese caso, aparecerán ramas independientes (*branches*) que pueden mezclarse en una (*merge*), o ser descartadas, o rebasadas, o… A la estructura formada por todas las ramas de un proyecto se le denomina árbol (*tree*). Sea cual sea su estructura, `git log` siempre nos mostrará la historia que lleva desde el inicio del árbol hasta nuestra posición actual en él en sentido cronológico inverso.

Finalmente, modificamos `fic_wave.c`, para interpretar correctamente la cabecera de los ficheros Microsoft WAVE; `p1.c`, `pav_analysis.c` y `pav_analysis.h`, para permitir el enventanado de la señal con la ventana de Hamming; y, por último, `p1.c`, para gestionar la escritura de la salida en un fichero:

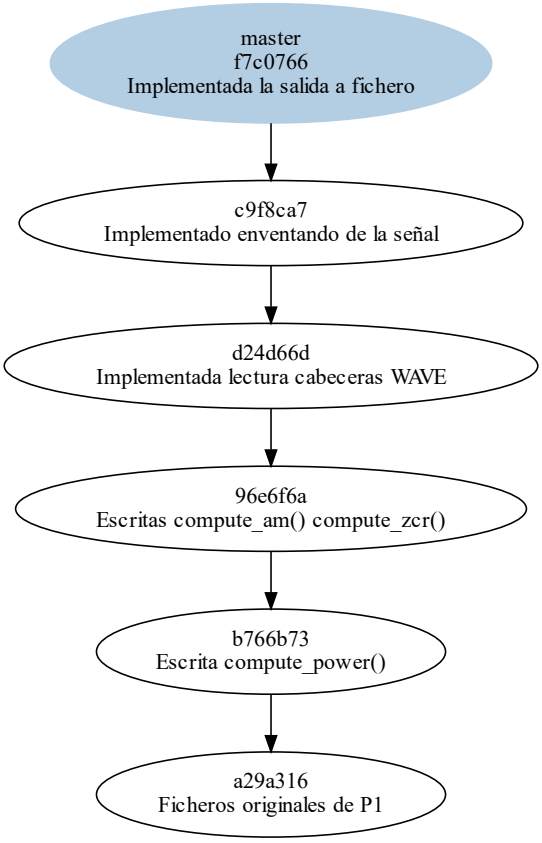
```
albino:~/PAV/P1$ git log --pretty=oneline
3b6d8800fe5edb79acc5b13cf4d57b322f19b6d7 (HEAD -> master) Escritas
compute_am() compute_zcr()
2efee58d945d47d48a2e3e3b01beb932c31aa0b8 Escrito compute_power()
9f0b9eb7b60279cd78ad943d314a85ff5a67ee54 Ficheros originales de P1
albino:~/PAV/P1$ vi fic_wave.c
albino:~/PAV/P1$ git commit -a -m "Implementada lectura cabeceras WAVE"
[master d24d66d] Implementada lectura cabeceras WAVE
 1 file changed, 2 insertions(+)
albino:~/PAV/P1$ vi p1.c pav_analysis.c pav_analysis.h
3 files to edit
albino:~/PAV/P1$ git add .
albino:~/PAV/P1$ git commit -m "Implementado enventando de la señal"
[master c9f8ca7] Implementado enventando de la señal
 3 files changed, 6 insertions(+)
albino:~/PAV/P1$ vi p1.c
albino:~/PAV/P1$ git add p1.c
albino:~/PAV/P1$ git commit -m "Implementada la salida a fichero"
[master f7c0766] Implementada la salida a fichero
 1 file changed, 2 insertions(+)
albino:~/PAV/P1$ git log --pretty=oneline
f7c076603bef32b3757d984af5e95fa6c72605fc (HEAD -> master) Implementada la
salida a fichero
c9f8ca7fe314e838d369454619b90ac458ce85ef Implementado enventando de la señal
d24d66d7a5bd57d73999b9fff76429f20b9cd037 Implementada lectura cabeceras WAVE
96e6f6ab3d5ee4cd9a476e7e128a2cccaa602890 Escritas compute_am() compute_zcr()
b766b736ed3102a9f2a3b058f1f4bb1bb9082c15 Escrito compute_power()
a29a3164872ca19ce9a90254d5f7f362fddee49e Ficheros originales de P1
```

Fijémonos en que se han introducido tres commit, cada uno con una sintaxis ligeramente diferente. En el primero, se ha realizado directamente el commit de todo lo que estuviera modificado respecto al último commit realizado. En el segundo, primero añadimos a la zona de preparación todo lo modificado y, a continuación, hacemos el commit de lo que hay en esta zona. En el tercer caso, primero añadimos un único fichero a la zona de preparación, y a continuación confirmamos lo que hay en esta zona.

El flujo de trabajo que hemos creado hasta ahora es el más sencillo posible: consiste de una sola rama (**`master`**) que hemos ido actualizando con cada commit. Es lo que se denomina un workflow lineal.

Podemos ver gráficamente este flujo de trabajo usando la utilidad `git-big-picture` (disponible en https://github.com/esc/git-big-picture/blob/master/git-big-picture):

```
albino:~/PAV/P1$ git-big-picture -f svg -act --outfile=rebase.svg
```



Deshacer commits para corregir errores

La principal ventaja de usar un sistema de control de versiones en un flujo de trabajo lineal es que podemos regresar a un estado anterior del proyecto si descubrimos que se ha cometido algún error en alguna de las versiones almacenadas. Básicamente, lo que debemos hacer es deshacer el efecto de uno o varios commits.

Dependiendo del commit que queramos deshacer, la tarea será más o menos complicada. El caso más sencillo es cuando queremos deshacer los últimos commits realizados, mientras que deshacer commits intermedios puede resultar más laborioso.

Como (casi) todo en Git, hay distintas maneras de realizar esta tarea. En las secciones siguientes se verá una de estas maneras para distintas situaciones típicas.

Deshacer el último commit para corregir errores

Imaginemos que, después de realizar un commit, descubrimos que hay un error en el mismo. Un caso especialmente doloroso, pero bastante habitual, es cuando se decide *mejorar* un código que funcionaba correctamente, y la modificación funciona peor que el original. Muchas veces, no sólo la solución mejorada no funciona, sino que en el proceso se destruye el código que sí lo hacía.

Por ejemplo: escribimos una alternativa para escribir la salida de P1 en un fichero:

```
albino:~/PAV/P1$ vi p1.c
albino:~/PAV/P1$ git -a -m "Salida a fichero mejorada"
albino:~/PAV/P1$ git commit -a -m "Salida a fichero mejorada"
[master a6ab1bd] Salida a fichero mejorada
 1 file changed, 1 insertion(+)
albino:~/PAV/P1$ git log --pretty=oneline
a6ab1bd7fd9ba491311c80f07d6678ec8c87f03b (HEAD -> master) Salida a fichero
mejorada
f7c076603bef32b3757d984af5e95fa6c72605fc Implementada la salida a fichero
c9f8ca7fe314e838d369454619b90ac458ce85ef Implementado enventando de la señal
d24d66d7a5bd57d73999b9fff76429f20b9cd037 Implementada lectura cabeceras WAVE
96e6f6ab3d5ee4cd9a476e7e128a2cccaa602890 Escritas compute_am() compute_zcr()
b766b736ed3102a9f2a3b058f1f4bb1bb9082c15 Escrita compute_power()
a29a3164872ca19ce9a90254d5f7f362fddee49e Ficheros originales de P1
```

Pero descubrimos que las modificaciones introducidas por el último commit son erróneas. Por tanto, queremos recuperar el estado que teníamos justo después de realizar el penúltimo.

Como tantas otras cosas en Git, existen varios modos de recuperar los ficheros al estado anterior a un cierto commit: `git checkout`, `git reset`, `git revert` o la recientemente añadida `git restore`… Y seguramente hay más. De todos ellos, el más interesante es, justamente, el último, `git restore`, introducido en la versión 2.23 de agosto de 2019. Desgraciadamente, este comando está aún en una fase experimental, y su comportamiento podría variar en el futuro.

Reversión de commits usando `git checkout`

Este es uno de los comandos más obtusos del ya de por si obtuso Git. Hace casi de todo; entre otras cosas, deshacer commits. En principio, su misión es la de cambiar nuestra posición actual en el árbol por otra distinta. Pero, con la opción `--`, lo que hace es mantenerse en el commit actual, recuperando los ficheros tal y como estaban justo después de realizarse el commit deseado:

```
albino:~/PAV/P1$ git checkout -- HEAD~1 .
albino:~/PAV/P1$ git log --pretty=oneline
f7c076603bef32b3757d984af5e95fa6c72605fc (HEAD -> master) Implementada la
salida a fichero
c9f8ca7fe314e838d369454619b90ac458ce85ef Implementado enventando de la señal
d24d66d7a5bd57d73999b9fff76429f20b9cd037 Implementada lectura cabeceras WAVE
96e6f6ab3d5ee4cd9a476e7e128a2cccaa602890 Escritas compute_am() compute_zcr()
b766b736ed3102a9f2a3b058f1f4bb1bb9082c15 Escrita compute_power()
a29a3164872ca19ce9a90254d5f7f362fddee49e Ficheros originales de P1
```

Aunque no vemos ningún cambio, los ficheros del directorio de trabajo están en el mismo estado que justo después de realizarse el commit `c9f8ca7`.

Reversión de commits usando `git restore`

Permite recuperar ficheros de un cierto commit al directorio de trabajo, al área de preparación o a ambos. Viene a ser un alias de `git checkout --`, porque no cambia la estructura del árbol, sólo trae los ficheros que nos interesan de un cierto commit. El commit deseado se indica con la opción `--source` y, para indicarle que deseamos recuperar los ficheros en el directorio de trabajo, se utiliza la opción `--worktree`.

```
albino:~/PAV/P1$ git restore --worktree --source=HEAD~1 .
albino:~/PAV/P1$ git log --pretty=oneline
f7c076603bef32b3757d984af5e95fa6c72605fc (HEAD -> master) Implementada la
salida a fichero
c9f8ca7fe314e838d369454619b90ac458ce85ef Implementado enventando de la señal
d24d66d7a5bd57d73999b9fff76429f20b9cd037 Implementada lectura cabeceras WAVE
96e6f6ab3d5ee4cd9a476e7e128a2cccaa602890 Escritas compute_am() compute_zcr()
b766b736ed3102a9f2a3b058f1f4bb1bb9082c15 Escrita compute_power()
a29a3164872ca19ce9a90254d5f7f362fddee49e Ficheros originales de P1
```

Reversión de commits usando `git revert`

Crea un nuevo commit eliminando los cambios introducidos por una serie de commits. Para eliminar los cambios introducidos por el último commit, haríamos:

```
albino:~/PAV/P1$ git revert HEAD -no-edit
[master 20e24dc] Revert "Revert "Implementada la salida a fichero""
 Date: Sun Oct 6 18:38:05 2019 +0200
 1 file changed, 1 insertion(+), 1 deletion(-)
albino:~/PAV/P1$ git log --pretty=oneline
20e24dc76c9fe81727cdb8fcc8433fb5d2873b63 (HEAD -> master) Revert
"Implementada la salida a fichero"
f7c076603bef32b3757d984af5e95fa6c72605fc Implementada la salida a fichero
c9f8ca7fe314e838d369454619b90ac458ce85ef Implementado enventando de la señal
d24d66d7a5bd57d73999b9fff76429f20b9cd037 Implementada lectura cabeceras WAVE
96e6f6ab3d5ee4cd9a476e7e128a2cccaa602890 Escritas compute_am() compute_zcr()
b766b736ed3102a9f2a3b058f1f4bb1bb9082c15 Escrita compute_power()
a29a3164872ca19ce9a90254d5f7f362fddee49e Ficheros originales de P1
```

Como puede verse, `git revert` ha creado un nuevo commit que tendrá el mismo contenido que el commit `c9f8ca`. La opción `--no-edit` sirve para evitar que se ponga en marcha el editor para pedirnos el mensaje del nuevo commit; en luger de eso, el mensaje será `Revert "texto del commit revertido"`.

En lugar del último commit, se puede revertir cualquier serie de commits de la historia del proyecto. Por ejemplo: `git revert HEAD~4..HEAD~3` intentaría revertir los cambios introducidos por los commits que van desde cinco antes del actual (`HEAD~4`), hasta cuatro antes(`HEAD~3`). En estos casos, es posible que `git revert` no pueda acabar la operación por sí sólo debido a la aparición de conflictos. Si esto ocurre, `git revert` dejará los ficheros en el directorio de trabajo y marcará los segmentos conflictivos con las cadenas de texto `<<<<<<`, `======` y `>>>>>>`. Nosotros tendremos que editar los ficheros conflictivos y realizar el commit manualmente.

Reversión de commits usando `git reset`

El comando `git reset` sirve para reposicionarse en un commit distinto al actual. Después de ejecutarlo, el `HEAD` apunta al commit seleccionado. En principio, los ficheros del directorio de trabajo no se ven afectados (se quedan los que teníamos antes de la operación). Con la opción `--hard` se consigue que también los ficheros del directorio de trabajo se actualicen a los que había justo después de realizarse el commit seleccionado. Todo lo que había en la historia después de este commit desaparece de la rama principal:

```
albino:~/PAV/P1$ git reset --hard HEAD~1
HEAD is now at f7c0766 Implementada la salida a fichero
albino:~/PAV/P1$ git log --pretty=oneline
f7c076603bef32b3757d984af5e95fa6c72605fc (HEAD -> master) Implementada la
salida a fichero
c9f8ca7fe314e838d369454619b90ac458ce85ef Implementado enventando de la señal
d24d66d7a5bd57d73999b9fff76429f20b9cd037 Implementada lectura cabeceras WAVE
96e6f6ab3d5ee4cd9a476e7e128a2cccaa602890 Escritas compute_am() compute_zcr()
b766b736ed3102a9f2a3b058f1f4bb1bb9082c15 Escrita compute_power()
a29a3164872ca19ce9a90254d5f7f362fddee49e Ficheros originales de P1
```

En esta orden `HEAD~1` indica el penúltimo commit, porque éste es el commit al que queremos regresar. Análogamente, podríamos referirnos al `N`-ésimo último commit con la expresión `HEAD~N`.

Aunque se ha deshecho el último commit, éste no se borra del repositorio, sino que lo único que se hace es cambiar la etiqueta `HEAD` al commit anterior al último. De alguna manera, el `HEAD` anterior desaparece del flujo de trabajo, pero no del repositorio, con lo que, si fuera necesario, podríamos volver a él con posterioridad. Podemos ver cómo queda el repositorio, incluyendo los commits perdidos, con la orden `git reflog`:

```
albino:~/PAV/P1$ git reflog
f7c0766 (HEAD -> master) HEAD@{0}: reset: moving to HEAD~1
a6ab1bd HEAD@{1}: commit: Salida a fichero mejorada
f7c0766 (HEAD -> master) HEAD@{2}: commit: Implementada la salida a fichero
c9f8ca7 HEAD@{3}: commit: Implementado enventando de la señal
d24d66d HEAD@{4}: commit: Implementada lectura cabeceras WAVE
96e6f6a HEAD@{5}: commit: Escritas compute_am() compute_zcr()
b766b73 HEAD@{6}: commit: Escrita compute_power()
a29a316 HEAD@{7}: commit (initial): Ficheros originales de P1
```

Deshacer un commit intermedio para corregir errores (sin conflictos)

Otra situación es descubrir que una modificación intermedia es errónea. En este caso nos gustaría eliminar el commit que introduce el error, pero sin eliminar todo el trabajo realizado con posterioridad. Habitualmente, un problema de este tipo dará lugar a conflictos más o menos difíciles de resolver. Esto ocurrirá cuando los ficheros que queremos recuperar deshaciendo el commit erróneo son modificados en otros commits posteriores.

En nuestro proyecto P1, la lectura de la cabecera Microsoft WAVE sólo afecta al fichero `fic_wave.c`, que no es modificado al implementar el enventanado de la señal o la escritura de la salida en fichero. Así pues, podemos eliminar el commit conflictivo sin perder el trabajo realizado en los siguientes.

La orden necesaria para eliminar un commit intermedio, manteniendo los cambios introducidos por commits posteriores es `git rebase`. La sintaxis de este comando es bastante compleja (ejecutar `git help rebase` para ver las distintas opciones). En este caso, si queremos eliminar el commit `d24d6`, la orden es:

```
albino:~/PAV/P1$ git rebase --onto 96e6f d24d6 master
First, rewinding head to replay your work on top of it...
Applying: Implementado enventando de la señal
Applying: Implementada la salida a fichero
albino:~/PAV/P1$ git log --pretty=oneline
9a96fad9956ac4095cae440b67dd1838e9497103 (HEAD -> master) Implementada la
salida a fichero
2bd99bde6f4bfe6172a5535e70257dfa95315058 Implementado enventando de la señal
96e6f6ab3d5ee4cd9a476e7e128a2cccaa602890 Escritas compute_am() compute_zcr()
b766b736ed3102a9f2a3b058f1f4bb1bb9082c15 Escrita compute_power()
a29a3164872ca19ce9a90254d5f7f362fddee49e Ficheros originales de P1
```

Lo que hace esta orden es aplicar las modificaciones de los commits que van desde el estado dejado después del `d24d6` (`Implementada lectura cabecera WAVE`) hasta el final de la rama **`master`**, sobre el estado dejado después del commit `96e6f` (`Escritas compute_am() compute_zcr()`). De este modo se eliminan las modificaciones introducidas en el commit (o commits) que queremos eliminar.

Fijémonos en que las claves SHA1 de los commits aplicados sobre el eliminado se han modificado. Esto es así porque, efectivamente, los ficheros correspondientes son distintos. Además, aunque no los veamos en la historia del proyecto, los commits eliminados siguen estando presentes en el repositorio. Es decir, todos los commits antiguos son todavía accesibles (por ejemplo, con el comando `git checkout`).

Deshacer un commit intermedio para corregir errores (pretendidamente, con conflictos)

Desgraciadamente, cuando queremos eliminar un commit intermedio, la situación más habitual es que aparezcan conflictos. Eso ocurre, por ejemplo, cuando en el commit a eliminar se introdujeron cambios en ficheros que vuelven a ser modificados en commits posteriores. Lo que queremos es que esos ficheros regresen a su estado original y que luego se les apliquen los mismos cambios que se hicieron en los ficheros modificados. Aunque en ocasiones Git es tan listo que es capaz de hacer esto sin problemas, lo habitual es que no sea inmediato qué cambios hacer y/o dónde aplicarlos, con lo que Git no podrá realizar la tarea por sí solo. Lo bueno es que Git sí es capaz de aislar estas situaciones conflictivas y facilitarnos el proceso de resolución.

El funcionamiento de la detección y resolución de conflictos es uno de los más complicados de Git aunque, de cara al usuario, todo se reduce, en buen parte, a tener fe en Git. Como veremos más adelante, Git es capaz por sí solo de resolver muchas situaciones (es decir, de evitar los conflictos) y, cuando no es capaz de evitarlos, de proporcionar mecanismos para encontrarlos y resolverlos manualmente. En la próxima sección, dedicada al flujo de trabajo paralelo, entraremos en más detalle en qué consiste un conflicto y cómo los gestiona Git. En esta sección, sin entrar en mucho detalle, veremos cómo aparecen y de qué mecanismos disponemos para resoverlos.

Por ejemplo, supongamos que el commit que queremos eliminar es el que implementa el enventanado de la señal. Éste afecta a los ficheros `pav_analysis.c`, `pav_analysis.h` y `p1.c`. Los dos primeros no vuelven a ser modificados, pero `p1.c` sí es modificado cuando introducimos la escritura de la salida en fichero. Intentamos ejecutar `git rebase` igual que hicimos en el apartado anterior:

```
albino:~/PAV/P1$ git rebase --onto HEAD~2 HEAD~1 master
First, rewinding head to replay your work on top of it...
Applying: Implementada la salida a fichero
albino:~/PAV/P1$ git log --pretty=oneline
8f4a7b402118ca43bda2bb0c2cbcde8fc86a1502 (HEAD -> master) Implementada la
salida a fichero
d24d66d7a5bd57d73999b9fff76429f20b9cd037 Implementada lectura cabeceras WAVE
96e6f6ab3d5ee4cd9a476e7e128a2cccaa602890 Escritas compute_am() compute_zcr()
b766b736ed3102a9f2a3b058f1f4bb1bb9082c15 Escrita compute_power()
a29a3164872ca19ce9a90254d5f7f362fddee49e Ficheros originales de P1
```

A destacar dos cosas: la primera, que ahora hacemos referencia a los commits que participan en el comando usando la nomenclatura `HEAD~N`, en lugar de los primeros dígitos de su clave hash SHA1; la segunda, que a pesar de lo dicho en el párrafo anterior y de que el autor intentó provocar una situación conflictiva modificando dos veces el mismo fichero, Git es más listo que el autor y ha sido quien de realizar el rebase sin conflictos. Sirva esta triste experiencia de prueba de que Git es capaz de hacer cosas increíbles… e increíblemente útiles.

Deshacer un commit intermedio para corregir errores (efectivamente, con conflictos)

Segundo intento: volvemos al flujo de trabajo original, y añadimos un commit con una modificación de `p1.c` que sí provoque conflictos con el enventanado de la señal.

```
albino:~/PAV/P1$ vi p1.c
albino:~/PAV/P1$ git commit -a -m "Modificado el enventanado de la señal"
[master 8b7ab2d] Modificado el enventanado de la señal
 1 file changed, 1 insertion(+), 1 deletion(-)
albino:~/PAV/P1$ git log --pretty=oneline
e454c3561b8f7220e1a6c651f231ffc28e91e913 (HEAD -> master) Modificado el
enventanado de la señal
f7c076603bef32b3757d984af5e95fa6c72605fc Implementada la salida a fichero
c9f8ca7fe314e838d369454619b90ac458ce85ef Implementado enventando de la señal
d24d66d7a5bd57d73999b9fff76429f20b9cd037 Implementada lectura cabeceras WAVE
96e6f6ab3d5ee4cd9a476e7e128a2cccaa602890 Escritas compute_am() compute_zcr()
b766b736ed3102a9f2a3b058f1f4bb1bb9082c15 Escrita compute_power()
a29a3164872ca19ce9a90254d5f7f362fddee49e Ficheros originales de P1
```

Intentamos el rebase para eliminar el commit que implementó la lectura de la cabecera de ficheros WAVE:

```
albino:~/PAV/P1$ git rebase --onto d24d6 c9f8c master
First, rewinding head to replay your work on top of it...
Applying: Implementada la salida a fichero
Applying: Modificado el enventanado de la señal
Using index info to reconstruct a base tree...
M       p1.c
Falling back to patching base and 3-way merge...
Auto-merging p1.c
CONFLICT (content): Merge conflict in p1.c
error: Failed to merge in the changes.
Patch failed at 0002 Modificado el enventanado de la señal
Use 'git am --show-current-patch' to see the failed patch

Resolve all conflicts manually, mark them as resolved with
"git add/rm <conflicted_files>", then run "git rebase --continue".
You can instead skip this commit: run "git rebase --skip".
To abort and get back to the state before "git rebase", run "git rebase --
abort".
```

Ahora sí tenemos un conflicto, y Git avisa de ello de una manera un tanto escandalosa. Aunque el mensaje pueda abrumar un poco al principio, toda la información que proporciona es sumamente interesante. Por ejemplo, Git nos informa que únicamente hay problemas de conflictos en el fchero `p1.c`. También es interesante la recomendación de ejecutar `git am --show-current-patch`. Al hacerlo, obtenemos información precisa de cuáles son las líneas del fichero que provocan el conflicto:

```
albino:~/PAV/P1$ git am --show-current-patch
commit e454c3561b8f7220e1a6c651f231ffc28e91e913 (master)
Author: Albino Nogueiras Rodríguez <albino.nogueiras@upc.edu>
Date:   Fri Aug 23 12:23:08 2019 +0200

    Modificado el enventanado de la señal

diff --git a/p1.c b/p1.c
index 042e398..39214b6 100644
--- a/p1.c
+++ b/p1.c
@@ -14,7 +14,7 @@ int main(int argc, char *argv[]) {
     short *buffer;
     FILE  *fpWave;

-       // TODO: Implementar enventanado
+       // TODO: Implementar enventanado otra vez
```

```
    if (argc != 2 && argc != 3) {
```

Por otro lado, si miramos el fichero conflictivo, veremos que Git ha añadido marcadores para indicar las líneas que provocan el conflicto: una cadena de símbolos menor que (<) seguidos del commit actual; el texto actual de las líneas conflictivas después de eliminar los commits rebasados; una cadena de símbolos igual (=); y una cadena de símbolos mayor (>) seguidos del mensaje del commit a aplicar.

```
albino:~/PAV/P1$ vi p1.c
albino:~/PAV/P1$ head -21 p1.c  | tail -5
<<<<<<< HEAD
=======
        // TODO: Implementar enventanado otra vez

>>>>>>> Modificado el enventanado de la señal
```

Estos bloques son fáciles de localizar al editar el fichero y proporcionan bastante información de cómo resolver el conflicto. Ciertamente, no es sencillo de hacer, pero el hecho de que Git nos ayude a localizar el origen de los conflictos resulta de gran ayuda.

El proceso de resolución implica editar el fichero en cuestión, eliminar las marcas de bloque, dejar la versión definitiva del archivo (esto puede llevar trabajo) e indicar a Git de que ya hemos resuelto el conflicto con los comandos `git add` y `git rebase --continue`:

```
albino:~/PAV/P1$ vi p1.c
albino:~/PAV/P1$ git add p1.c
albino:~/PAV/P1$ git rebase --continue
Applying: Modificado el enventanado de la señal
albino:~/PAV/P1$ git log --pretty=oneline
f15f9760e714a42deb916b3ab8c9dabc5de5c1cc (HEAD -> master) Modificado el
enventanado de la señal
9b5865e25da732e5df166e8492037ee9f44db0ea Implementada la salida a fichero
d24d66d7a5bd57d73999b9fff76429f20b9cd037 Implementada lectura cabeceras WAVE
96e6f6ab3d5ee4cd9a476e7e128a2cccaa602890 Escritas compute_am() compute_zcr()
b766b736ed3102a9f2a3b058f1f4bb1bb9082c15 Escrita compute_power()
a29a3164872ca19ce9a90254d5f7f362fddee49e Ficheros originales de P1
```

Flujo de trabajo paralelo

En la sección anterior hemos visto los fundamentos de un flujo de trabajo lineal en Git. Para ello, hemos supuesto un usuario llamado **albino** que realizaba todo el trabajo de un proyecto denominado P1. Uno de los puntos fuertes de Git es su capacidad para permitir el trabajo colaborativo de más de un usuario. En este caso vamos a suponer que dos usuarios, **blanca** y **carles**, se reparten el trabajo de este mismo proyecto.

Evidentemente, Blanca y Carles pueden organizar su trabajo de manera lineal, usando un flujo de trabajo como el de la sección anterior, pero eso implica que deben sincronizarse para que los cambios de uno no afecten a los del otro y asegurarse de que la versión final incorpora el código válido de ambos, y sólo éste. Una buena manera de conseguirlo es que siempre trabajen juntos, pero eso no es siempre lo más conveniente (sobre todo, si además de Blanca y Carles, también participan David, Ernest, Feli, etc.).

Git proporciona un mecanismo que permite que cada usuario trabaje en su propia copia del repositorio, sólo interactuando con el resto cuando ello es necesario. Por ejemplo, cuando se tiene que construir la versión definitiva del proyecto a partir de las contribuciones de los distintos desarrolladores. Este mecanismo es el uso de ramas (*branch*) independientes para cada uno, que pueden mezclarse (*merge*) para unificar las contribuciones.

El proceso de mezcla puede dar lugar a conflictos que habrá que resolver manualmente. Es importante, por ello, que cada usuario procure trabajar en partes del proyecto que minimicen la probabilidad de que ello ocurra.

Lo primero que tienen que hacer cada uno de los usuarios es crear una copia personal del repositorio del proyecto original, que está en el subdirectorio `PAV/P1` del directorio raíz de albino. Esta acción la hacemos clonando (*clone*) el repositorio original con el comando `git clone`:

```
blanca:~/PAV$ git clone ~albino/PAV/P1
Cloning into 'P1'...
```

```
done.
blanca:~/PAV$ cd P1
blanca:~/PAV/P1$ git log --pretty=oneline
f4c6ce96929c5a282a6b1fb4330b89a1d4c53c71 (HEAD -> master, origin/master,
origin/HEAD) Ficheros originales de P1
```

```
carles:~/PAV$ mkdir P1
carles:~/PAV$ cd P1
carles:~/PAV/P1$ P1$ git clone ~albino/PAV/P1 .
Cloning into '.'...
done.
carles:~/PAV/P1$ git log --pretty=oneline
f4c6ce96929c5a282a6b1fb4330b89a1d4c53c71 (HEAD -> master, origin/master,
origin/HEAD) Ficheros originales de P1
```

Fijémonos en que, tras la clonación del repositorio, ambos usuarios tienen una copia idéntica del original, salvo que ahora aparecen dos nuevas etiquetas: `origin/master` y `origin/HEAD`. Estas etiquetas indican la rama y commit del que procede el commit actual. `origin` es la etiqueta que apunta al repositorio original. Este es un repositorio externo o remoto (*remote*). Nuestro repositorio puede hacer referencia a tantos repositorios remotos como sea necesario. Podemos ver los repositorios remotos referenciados (o no) en nuestro proyecto con la orden `git remote`:

```
blanca:~/PAV/P1$ git remote -v
origin  /mnt/d/albino/UbuntuOnWindows/PAV/P1 (fetch)
origin  /mnt/d/albino/UbuntuOnWindows/PAV/P1 (push)
```

Esta orden nos indica que `origin` apunta a un directorio del usuario albino tanto para extraer (*fetch*) como para introducir (*push*). También podemos hacer que blanca vea el repositorio de carles como un remoto y viceversa:

```
blanca:~/PAV/P1$ git remote add carles ~carles/PAV/P1
blanca:~/PAV/P1$ git remote -v
carles  /mnt/d/albino/UbuntuOnWindows/carles/PAV/P1 (fetch)
carles  /mnt/d/albino/UbuntuOnWindows/carles/PAV/P1 (push)
origin  /mnt/d/albino/UbuntuOnWindows/PAV/P1 (fetch)
origin  /mnt/d/albino/UbuntuOnWindows/PAV/P1 (push)
```

A continuación, tanto blanca como carles desarrollan su parte del proyecto. Por un lado, blanca se encarga de implementar las funciones de análisis (`pav_analysis.c`) y las funciones de lectura de cabeceras Wave (`fic_wave.c`):

```
blanca:~/PAV/P1$ vi pav_analysis.c
blanca:~/PAV/P1$ git commit -a -m "Escritas funciones de análisis"
[master 71308ac] Escritas funciones de análisis
 1 file changed, 6 insertions(+)
blanca:~/PAV/P1$ vi fic_wave.c
blanca:~/PAV/P1$ git commit -a -m "Implementada lectura cabecera WAVE"
[master c03915f] Implementada lectura cabecera WAVE
 1 file changed, 2 insertions(+)
blanca:~/PAV/P1$ git log --pretty=oneline
c03915f244d49452a64d5d1c4bcd6dac4006b3d7 (HEAD -> master) Implementada lectura
cabecera WAVE
71308aca0f804194cedc551b61cb23ddb16d4bf0 Escritas funciones de análisis
f4c6ce96929c5a282a6b1fb4330b89a1d4c53c71 (origin/master, origin/HEAD) Ficheros
originales de P1
```

Por otro lado, carles se encarga de implementar la salida a fichero (`p1.c`) y el enventanado de la señal (`p1.c`, `pav_analysis.c` y `pav_analysis.h`):

```
carles:~/PAV/P1$ vi p1.c
carles:~/PAV/P1$ git add p1.c
carles:~/PAV/P1$ git commit -m "Implementada la salida a fichero"
[master 7a6095a] Implementada la salida a fichero
 1 file changed, 1 insertion(+)
```

```
carles:~/PAV/P1$ vi p1.c pav_analysis.c pav_analysis.h
3 files to edit
carles:~/PAV/P1$ git add p1.c pav_analysis.c pav_analysis.h
carles:~/PAV/P1$ git commit -m "Implementado el enventanado de la señal"
[master 915d851] Implementado el enventanado de la señal
 3 files changed, 6 insertions(+)
carles:~/PAV/P1$ git log --pretty=oneline
915d851de8eef50c6b2b7e9d8936b56ef3cb5f08 (HEAD -> master) Implementado el
enventanado de la señal
7a6095a4fc459e31d4c17ca05eecca2ec2d16cc3 Implementada la salida a fichero
f4c6ce96929c5a282a6b1fb4330b89a1d4c53c71 (origin/master, origin/HEAD) Ficheros
originales de P1
```

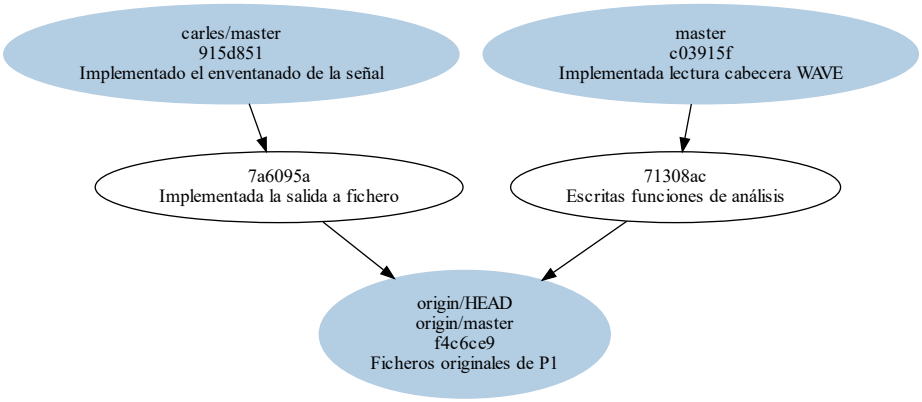
Ahora, lo que toca es unir las modificaciones introducidas por blanca y carles en un único repositorio. En un sistema con repositorio principal centralizado, lo que se haría es mezclar (*merge*) los cambios de ambos con el contenido del repositorio principal. Pero en este caso no existe tal repositorio principal, así que la mezcla se hará sobre uno de los dos usuarios… y luego el otro recuperará la versión definitiva del primero.

Integración sin conflictos

Lo primero que tiene que hacer blanca es recopilar el estado del repositorio de carles en su propio repositorio. Para ello ha de ejecutar el comando `git fetch`:

```
blanca:~/PAV/P1$ git fetch carles
remote: Counting objects: 8, done.
remote: Compressing objects: 100% (8/8), done.
remote: Total 8 (delta 6), reused 0 (delta 0)
Unpacking objects: 100% (8/8), done.
From /mnt/d/albino/UbuntuOnWindows/carles/PAV/P1
 * [new branch]      master     -> carles/master
```

Fijémonos que, como git fetch indica, el repositorio de blanca tiene ahora una nueva rama con la rama master de carles. Podemos ver el flujo de trabajo de blanca:

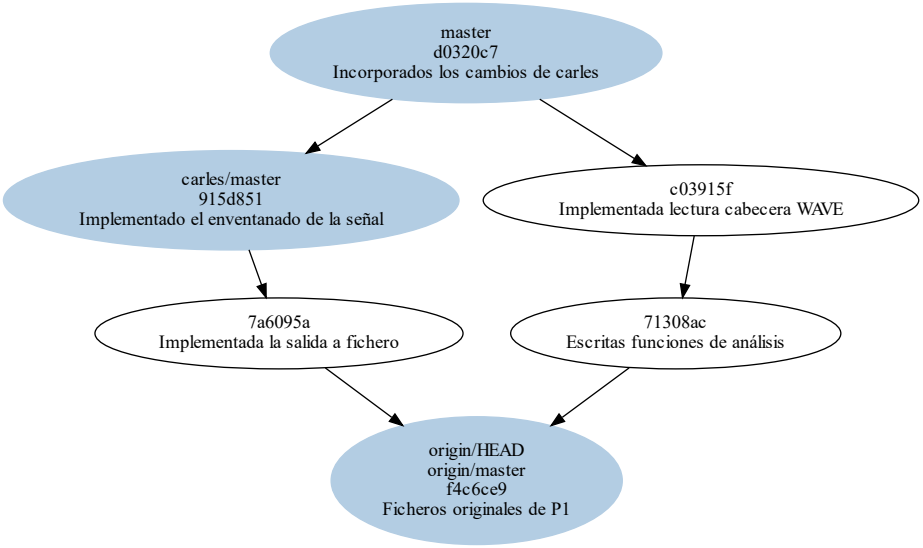


Ahora, blanca ha de mezclar (*merge*) los cambios producidos en las dos ramas:

```
blanca:~/PAV/P1$ git merge carles/master -m "Incorporados los cambios de
carles"
Auto-merging pav_analysis.c
Merge made by the 'recursive' strategy.
 p1.c           | 3 +++
 pav_analysis.c | 2 ++
 pav_analysis.h | 2 ++
 3 files changed, 7 insertions(+)
blanca:~/PAV/P1$ git log --pretty=oneline
d0320c7ee69d53db80dfa6ded784495fd456e6d2 (HEAD -> master) Incorporados los
cambios de carles
915d851de8eef50c6b2b7e9d8936b56ef3cb5f08 (carles/master) Implementado el
enventanado de la señal
7a6095a4fc459e31d4c17ca05eecca2ec2d16cc3 Implementada la salida a fichero
c03915f244d49452a64d5d1c4bcd6dac4006b3d7 Implementada lectura cabecera WAVE
71308aca0f804194cedc551b61cb23ddb16d4bf0 Escritas funciones de análisis
```

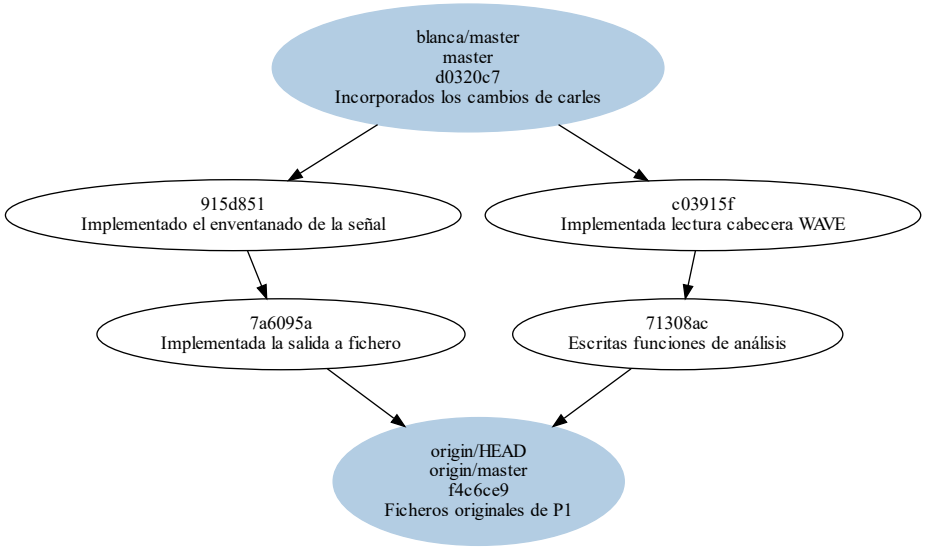
```
f4c6ce96929c5a282a6b1fb4330b89a1d4c53c71 (origin/master, origin/HEAD) Ficheros
originales de P1
```

Ahora, blanca ya dispone de la versión final del proyecto, con los cambios introducidos por carles y ella misma.



Falta que carles reciba en su repositorio estos cambios para que él también disponga de esta versión final. Lo que h de hacer es traer el repositorio de blanca con `git fetch` y mezclar los cambios con `git merge`:

```
carles:~/PAV/P1$ git remote add blanca ~blanca/PAV/P1
carles:~/PAV/P1$ git fetch blanca
remote: Counting objects: 9, done.
remote: Compressing objects: 100% (9/9), done.
remote: Total 9 (delta 6), reused 0 (delta 0)
Unpacking objects: 100% (9/9), done.
From /mnt/d/albino/UbuntuOnWindows/blanca/PAV/P1
 * [new branch]      master     -> blanca/master
carles:~/PAV/P1$ git merge blanca/master
Updating 915d851..d0320c7
Fast-forward
 fic_wave.c     | 2 ++
 pav_analysis.c | 6 ++++++
 2 files changed, 8 insertions(+)
carles:~/PAV/P1$ git log --pretty=oneline
d0320c7ee69d53db80dfa6ded784495fd456e6d2 (HEAD -> master, blanca/master)
Incorporados los cambios de carles
915d851de8eef50c6b2b7e9d8936b56ef3cb5f08 Implementado el enventanado de la
señal
7a6095a4fc459e31d4c17ca05eecca2ec2d16cc3 Implementada la salida a fichero
c03915f244d49452a64d5d1c4bcd6dac4006b3d7 Implementada lectura cabecera WAVE
71308aca0f804194cedc551b61cb23ddb16d4bf0 Escritas funciones de análisis
f4c6ce96929c5a282a6b1fb4330b89a1d4c53c71 (origin/master, origin/HEAD) Ficheros
originales de P1
```



Fijémonos en que los flujos de trabajo de blanca y carles son casi idénticos: apenas varía un par de etiquetas, pero el contenido de los commits (y sus claves hash SHA1) es el mismo. Es más, cada commit es idéntico al original en todos los aspectos, incluido su autor. Podemos verlos usando la versión completa de `git log`:

```
blanca:~/PAV/P1$ git log
commit d0320c7ee69d53db80dfa6ded784495fd456e6d2 (HEAD -> master,
carles/master)
Merge: c03915f 915d851
Author: Blanca Zengotitagoitia Balcells <blanca.zengotitagoitia@upc.edu>
Date:   Fri Aug 23 23:15:00 2019 +0200

    Incorporados los cambios de carles

commit 915d851de8eef50c6b2b7e9d8936b56ef3cb5f08
Author: Carles Castellnou Garcia <carles.castellnou@upc.edu>
Date:   Fri Aug 23 22:46:31 2019 +0200

    Implementado el enventanado de la señal

commit 7a6095a4fc459e31d4c17ca05eecca2ec2d16cc3
Author: Carles Castellnou Garcia <carles.castellnou@upc.edu>
Date:   Fri Aug 23 22:45:24 2019 +0200

    Implementada la salida a fichero

commit c03915f244d49452a64d5d1c4bcd6dac4006b3d7
Author: Blanca Zengotitagoitia Balcells <blanca.zengotitagoitia@upc.edu>
Date:   Fri Aug 23 22:33:36 2019 +0200

    Implementada lectura cabecera WAVE

commit 71308aca0f804194cedc551b61cb23ddb16d4bf0
Author: Blanca Zengotitagoitia Balcells <blanca.zengotitagoitia@upc.edu>
Date:   Fri Aug 23 22:31:47 2019 +0200

    Escritas funciones de análisis

commit f4c6ce96929c5a282a6b1fb4330b89a1d4c53c71 (origin/master, origin/HEAD)
Author: Albino Nogueiras Rodríguez <albino.nogueiras@upc.edu>
Date:   Fri Aug 23 01:58:11 2019 +0200

    Ficheros originales de P1
```