Departamento de Ingeniería Informática y Matemáticas
Universitat Rovira i Virgili

Manual Práctico de Git

Sistema de Control de Versiones

Revisión 130 (07/10/15)

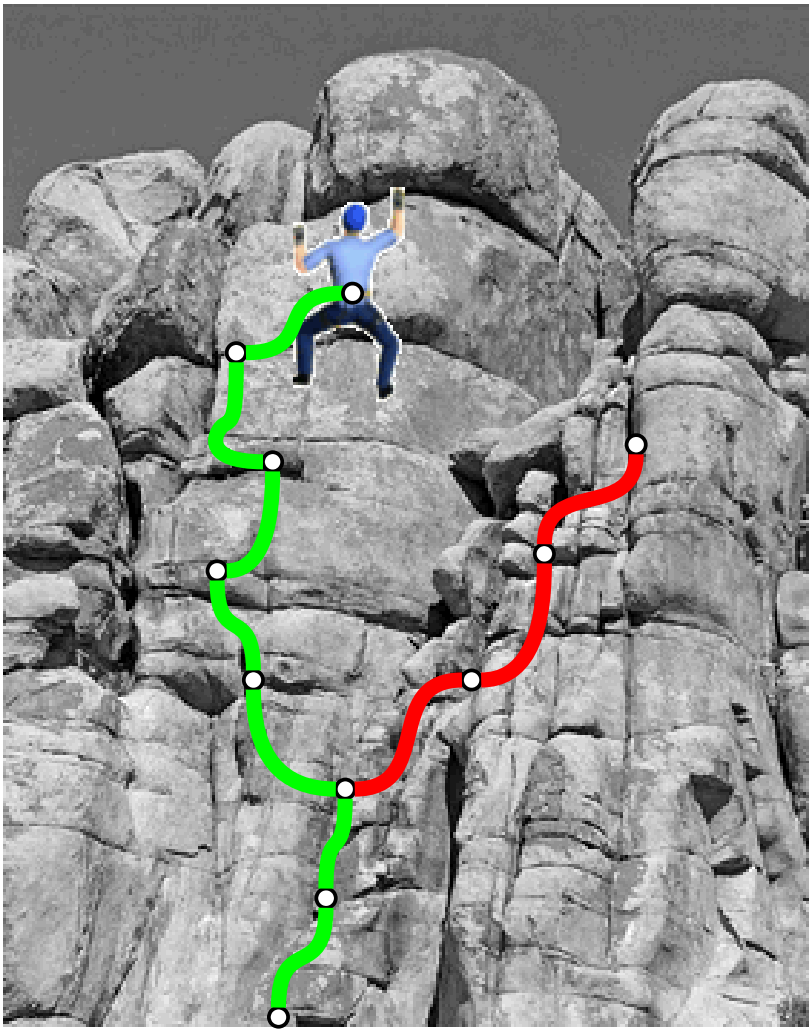
Autor: Santiago Romaní (santiago.romani@urv.cat)
Revisión: Pere Millán (pere.millan@urv.cat)

Índice de contenido

1 Idea básica

Cualquier proyecto informático se debe realizar **paso a paso**, probando las funcionalidades a medida que se van añadiendo a la aplicación. Además, en el desarrollo del proyecto puede que intervenga más de una persona, incluso con diferentes funciones: programadores, analistas, verificadores, diseñadores, traductores, etc.

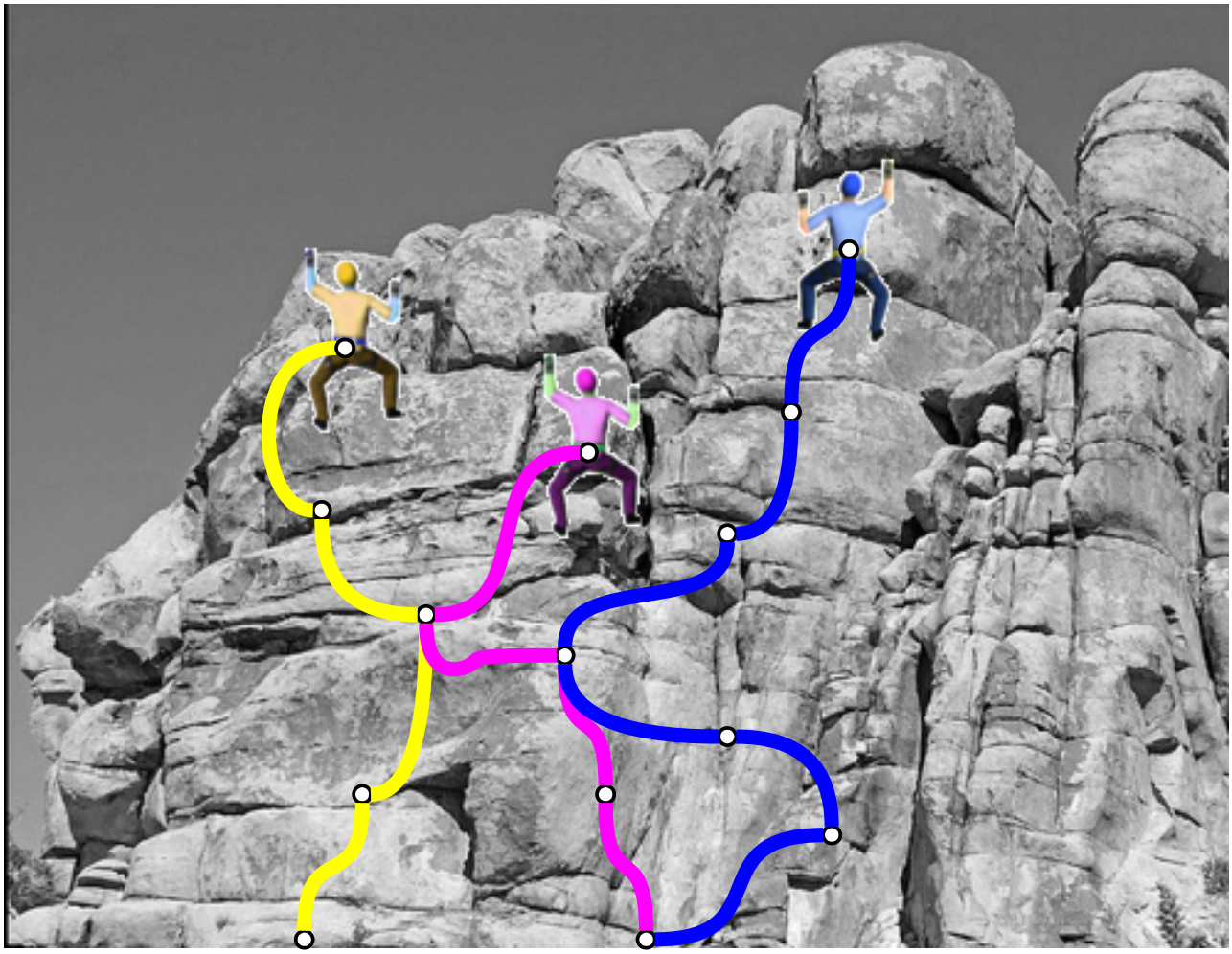
Un *Sistema de Control de Versiones* permite gestionar la evolución del contenido de **ficheros de texto** (por ejemplo, ficheros de código fuente) y la intervención de las diversas personas que trabajan en un mismo proyecto.



De manera análoga al proceso de escalar una montaña, las pequeñas modificaciones del proyecto que se van registrando en un sistema de control de versiones son como los puntos de anclaje que va colocando el escalador en la pared para enganchar su cuerda de seguridad: además de evitar una caída, los puntos de anclaje permiten volver hacia atrás en el caso de que nos hayamos equivocado de camino. Del mismo modo, un sistema de control de versiones permite **regresar a versiones anteriores del proyecto**.

Cuando se trabaja en equipo, cada componente del equipo registra los pequeños cambios que va introduciendo en el proyecto y los va **publicando en un ordenador servidor** común; de este modo, el resto de colaboradores podrán descargar esos cambios en sus propios ordenadores y fusionarlos con sus propias versiones del proyecto.

Para que las versiones de los distintos programadores no se interfieran unas con otras continuamente, los sistemas de control de versiones permiten trabajar con bifurcaciones o **ramas**, que son variantes paralelas del proyecto.



Siguiendo con la analogía anterior, las distintas ramas serían los caminos que sigue cada uno de los escaladores, aunque algunos puntos de anclaje pueden ser comunes. En un proyecto informático, dichos puntos se llaman versiones de **fusión**, las cuales **integran el código de distintas versiones en una única versión conjunta**. Al final del proyecto, todas las ramas se tendrán que fusionar en la versión definitiva del proyecto.

2 Introducción al Git

Hoy en día, existen muchos sistemas de control de versiones: ***CVS***, ***Subversion***, ***Mercurial***, ***Darcs***, ***Fossil***, etc. (http://en.wikipedia.org/wiki/Comparison_of_revision_control_software).

Git es uno de los más completos y también de los más complejos. Se desarrolló para el control de versiones de *Linux*, pero se ha adoptado en muchos desarrollos importantes (*Eclipse, Android, GNU Core Utilities, Wine, GNOME, Perl 5, KDE, Debian, Samba, DokuWiki,* etc.). Es un **sistema distribuido** y de **uso libre**.

Existen también varios servicios web gratuitos para alojar proyectos de software gestionados con Git, que ofrecen el servidor para publicar los cambios a los grupos de trabajo que no dispongan de la infraestructura necesaria: *GitHub, BitBucket, SourceForge, Google Code,* etc. (http://en.wikipedia.org/wiki/Comparison_of_open_source_software_hosting_facilities).

Sin embargo, para realizar la práctica de la asignatura dispondremos de un servidor específico dentro de la *intranet* de la Universidad, que se identifica con el nombre **git.lab.deim**. Para acceder a él, ver el documento disponible en *Moodle*: **Guía Rápida de Git**.

3 Instalación y configuración de Git

Para instalar el sistema Git en nuestro ordenador hay que descargarse el programa de la dirección de Internet <http://git-scm.com/>, seleccionando el sistema operativo apropiado.

Existen interfaces gráficas para Git, como ***Git GUI***, pero es muy recomendable empezar con la versión de **línea de comandos** por dos razones:

- porque las versiones gráficas NO incorporan todos los comandos disponibles,
- y porque no se pueden entender las opciones gráficas si primero no se conoce la funcionalidad básica de cada comando.

Dicho esto, hay que aclarar que **sí se recomienda el uso del programa *gitk***, que ofrece una representación gráfica de la evolución de todas las versiones del proyecto, lo cual resulta muy eficaz para inspeccionar el resultado de la ejecución de los diversos comandos introducidos por la línea de comandos.

El primer paso para trabajar con comandos Git es, evidentemente, iniciar una sesión de terminal o *shell*. En entornos Linux o MacOSX es relativamente fácil: hay que buscar la aplicación ***Terminal*.** En entornos Windows se puede buscar la aplicación ***Símbolo de Sistema*** en el apartado de **Accesorios** del menú de **Inicio**, o bien utilizar el terminal ***msys***, pero nosotros recomendamos el terminal específico del entorno Git para Windows, de nombre ***Git Bash***.

Antes de empezar a trabajar, hay que configurar nuestro usuario global de Git:

```
$ git config --global user.name "Santiago Romani"
$ git config --global user.email santiago.romani@urv.cat
```

Evidentemente, cada persona debe escribir su propio nombre y dirección de correo electrónico.

4 Contenido de un repositorio

Inicialmente necesitamos acceso a un repositorio Git remoto alojado en un ordenador servidor. Las personas responsables del servidor y del repositorio nos tiene que conceder los privilegios necesarios para dicho acceso.

4.1 ***git clone***

Una vez dispongamos de los privilegios necesarios, lo primero que hay que hacer es un clon del repositorio remoto (también denominado '***origin***') sobre un repositorio local.

Sin embargo, antes de realizar el clon nos tenemos que situar (con el comando '*cd* <dir>') sobre el directorio que contendrá el repositorio local. Se recomienda que sea el directorio raíz del disco "C:\", o en un subdirectorio con un nombre simple (por ejemplo "C:\Compus"), con el fin de evitar conflictos debido a una rutas del directorio que contenga espacios en blanco, como "C:\Documents and Settings". Esta recomendación también es útil para permitir una escritura rápida del directorio de trabajo:

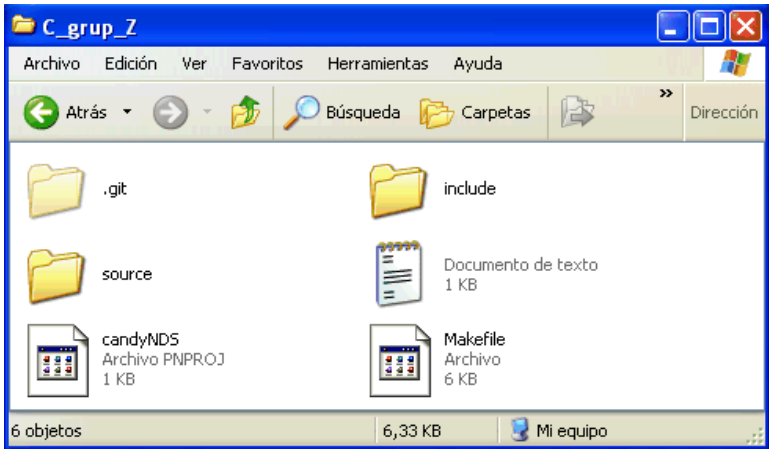
```
$ cd /c
```

A continuación ya podemos clonar el repositorio con el comando '*git clone*':

```
$ git clone <ID usuario>@git.lab.deim:<ID repo>
```

donde <ID usuario> se tiene que sustituir por nuestro identificador (el NIF), y <ID repo> se tiene que sustituir por nuestro identificador del repositorio, por ejemplo, "C_grup_Z".

Al ejecutar el comando, el sistema Git intenta realizar una conexión remota con el servidor, para lo cual hay que validarse correctamente. Si todo funciona bien, se creará un directorio con el nombre del repositorio y con los ficheros de la rama actual 'master' El contenido de este directorio se puede observar desde un visor de ficheros gráfico:



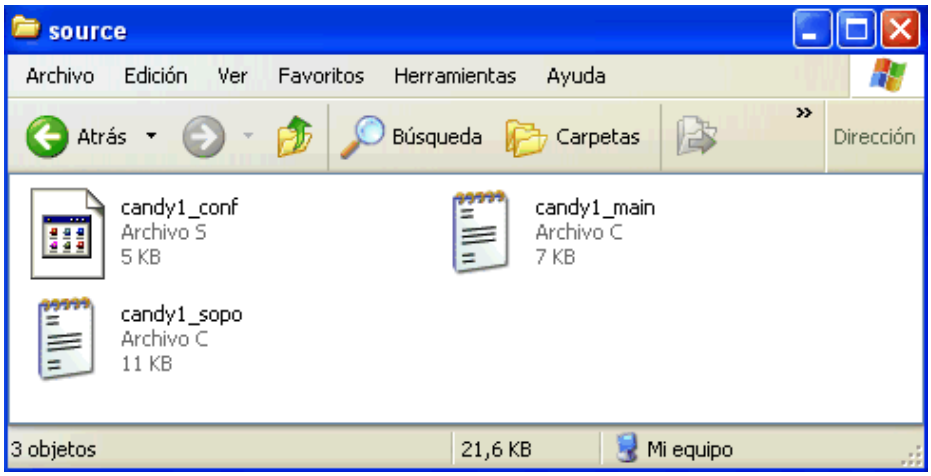
Para efectuar comandos Git sobre este repositorio, es necesario cambiar el directorio de trabajo actual:

```
$ cd <ID repo>
(master)$
```

Cuando nos posicionamos dentro del directorio que contiene el repositorio, el ***Git Bash*** nos informa de la rama seleccionada actualmente (master), dentro del *prompt*.

4.2 ***gitk***

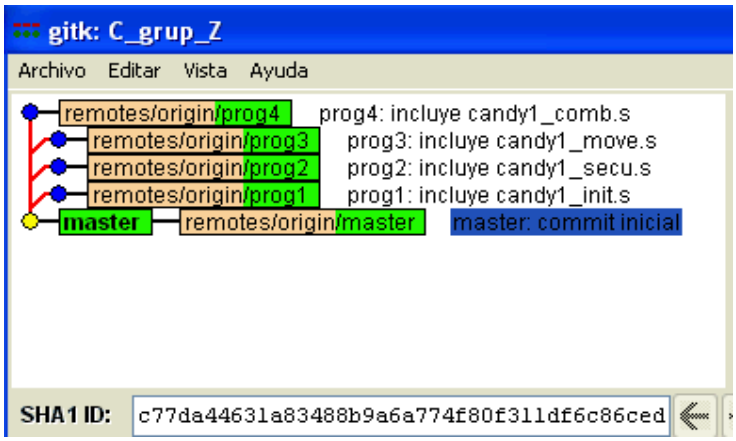
Es importante entender que un repositorio Git **no** es simplemente una carpeta alojada en un servidor remoto, lo cual se podría realizar fácilmente con un servicio como *Dropbox*, sino que contiene una estructura de versiones relacionadas entre sí que permiten cambiar o restaurar el contenido de los ficheros y directorios del proyecto con solo cambiar de versión. Para entender esto, se sugiere abrir la carpeta "source" que, inicialmente, contiene algunos ficheros fuente:



Lo que se está mostrando son los ficheros para la versión o *commit* seleccionado actualmente. Para ver todos los *commits* que proporciona el *proyecto*, podemos invocar al programa ***gitk*** desde el terminal:

```
(master)$ gitk --all --date-order &
```

La opciones indicadas sirven para mostrar todas las ramas (*--all*) y para mostrar los *commits* en el orden de su fecha de creación (*--date-order*). El símbolo *ampersand* '&' del final sirve para arrancar el programa ***gitk*** en modo *background*, lo cual le permite ejecutarse concurrentemente con la propia terminal. La representación de los *commits* del repositorio se ofrece en la parte superior izquierda de la ventana del ***gitk***:



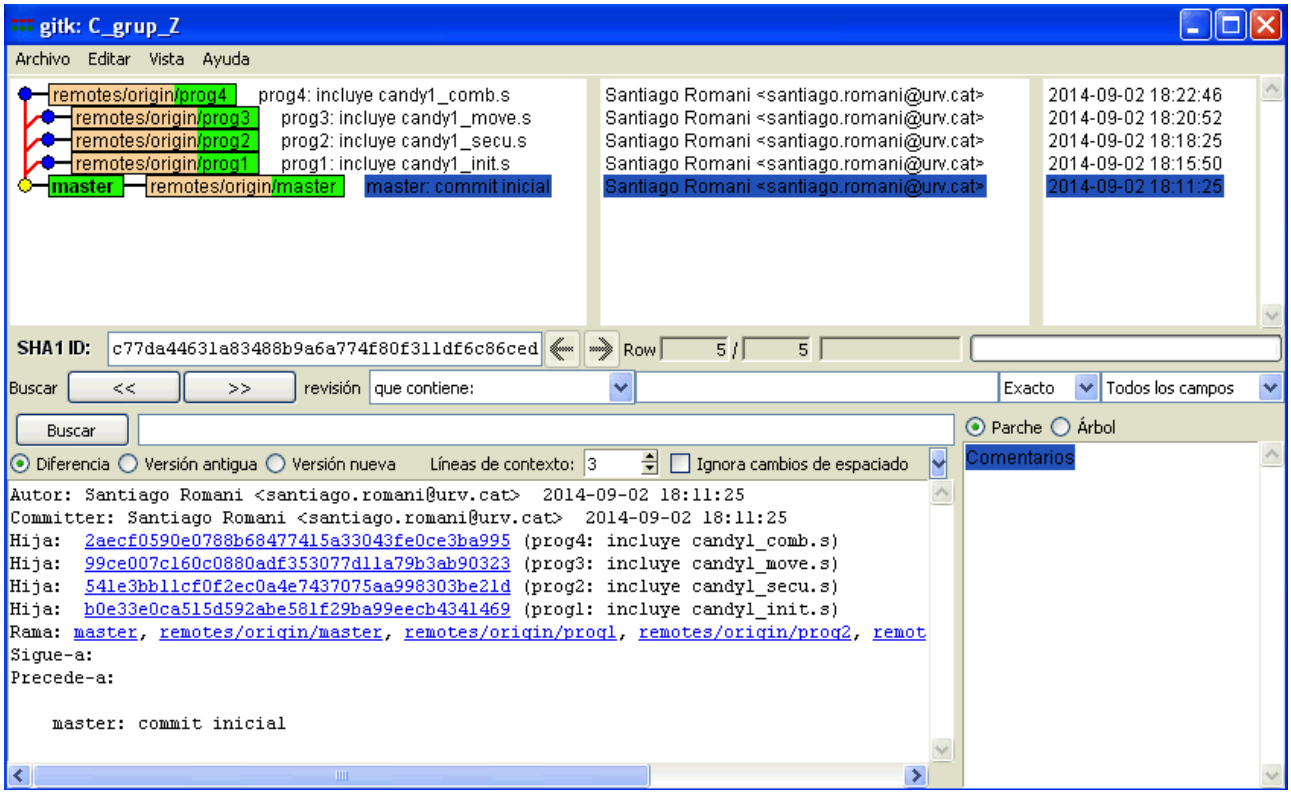
Los *commits* son los puntos azules más el punto amarillo, que representa al *commit* desplegado actualmente sobre el sistema de ficheros. Los *commits* están unidos por líneas, que representan las dependencias históricas entre las versiones.

A la derecha de los puntos se observan unas etiquetas que indican el *commit* que está referenciado por cada rama. Las etiquetas verdes son de ramas del repositorio local (de momento solo hay una, 'master'), mientras que las etiquetas que empiezan por '*remote/origin*' son de las ramas del repositorio remoto.

Más a la derecha se encuentra el mensaje identificativo de cada *commit*. Con el puntero de pantalla se puede seleccionar un *commit* concreto, cuyo comentario se resaltará con fondo azul. Este *commit* seleccionado puede ser diferente del *commit* actualmente desplegado en el árbol de directorios y ficheros (punto amarillo).

En la parte inferior de la figura anterior también se muestra el identificador SHA1 del *commit* seleccionado (fondo azul); se trata de un código de 40 dígitos hexadecimales generado automáticamente por Git a partir del contenido del *commit*, aunque para referencias posteriores podemos utilizar solo los primeros 7 dígitos (en el ejemplo, 'c77da44').

A continuación se muestra toda la ventana completa del ***gitk***, para comentar otras partes a tener en cuenta:

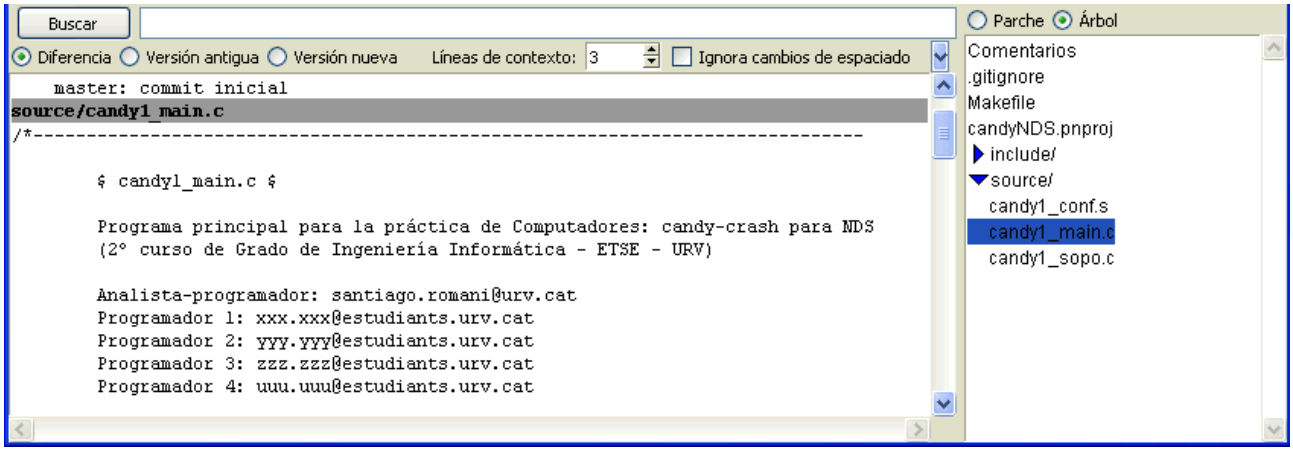


A la derecha del panel de *commits* se muestran otros dos paneles que indican el autor y la fecha de cada *commit*.

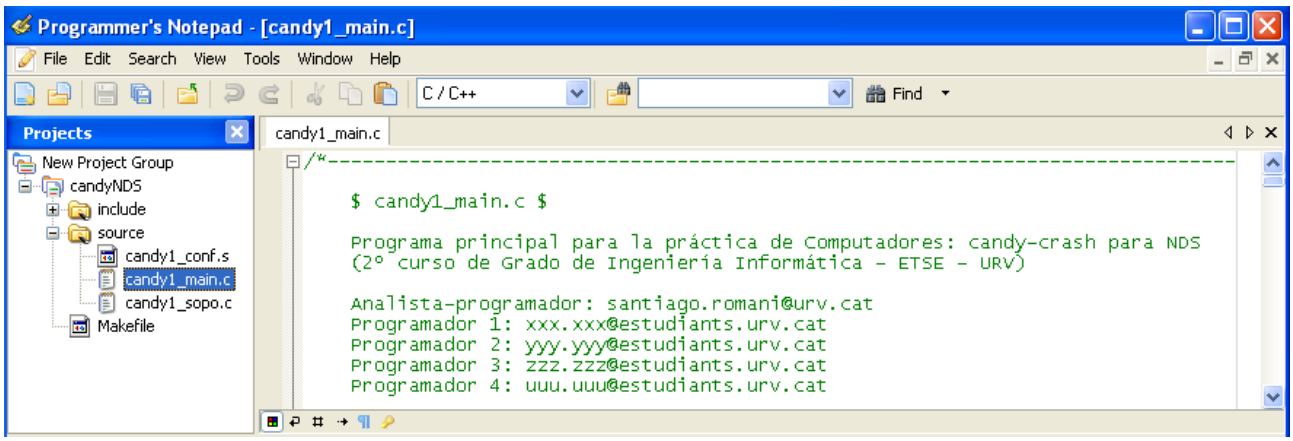
En la parte inferior derecha hay dos radio-botones para seleccionar el tipo de visualización del *commit* seleccionado, que puede ser en modo "**Parche**" o en modo "**Árbol**".

El modo "**Parche**" sirve para ver las diferencias del *commit* seleccionado con el anterior (más adelante veremos ejemplos).

El modo "**Árbol**" permite ver el contenido de todos los ficheros contenidos en el *commit* seleccionado. Por ejemplo, la siguiente figura muestra el principio del fichero 'candy1_main.c':



El contenido del fichero 'candy1_main.c' también se puede ver con un editor de código fuente, como el ***Programmer's Notepad***. Para visualizar dicho fichero, podemos abrirlo directamente con el comando "**Open**" del menú **File**, o podemos abrir el proyecto de la práctica, con el comando "**Open Project**" también desde el menú **File**, para después seleccionar el fichero en el panel de contenido del proyecto:



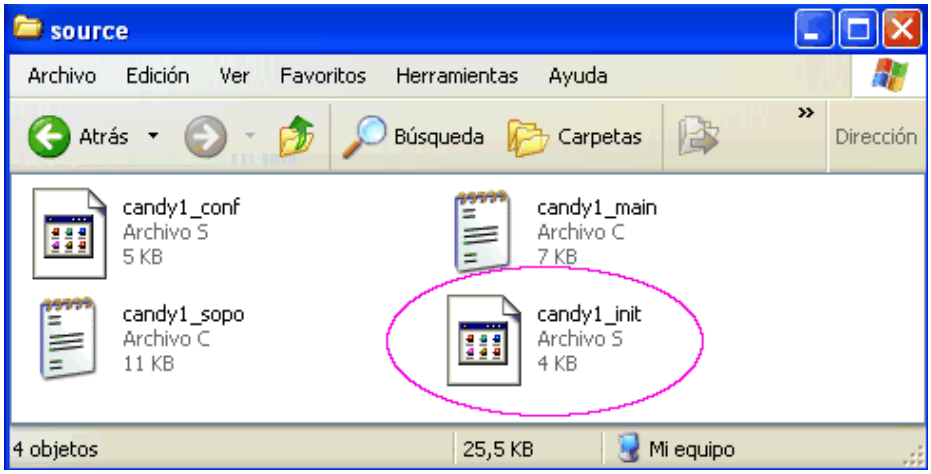
Entonces, ¿por qué es necesario mantener la misma información de dos maneras distintas, con ficheros y con la base de datos interna del Git? La respuesta es que el contenido de los ficheros se irá guardando en una serie de *commits* dentro de la base de datos del Git, de modo que **siempre podremos recuperar el estado que tenían los ficheros cuando se registró cada *commit***. Esto nos permite realizar modificaciones al código fuente con total tranquilidad, sin miedo a perder trozos de código fuente (o texto en general) que puede que tengamos que recuperar en el futuro.

4.3 *git checkout <ID rama>*

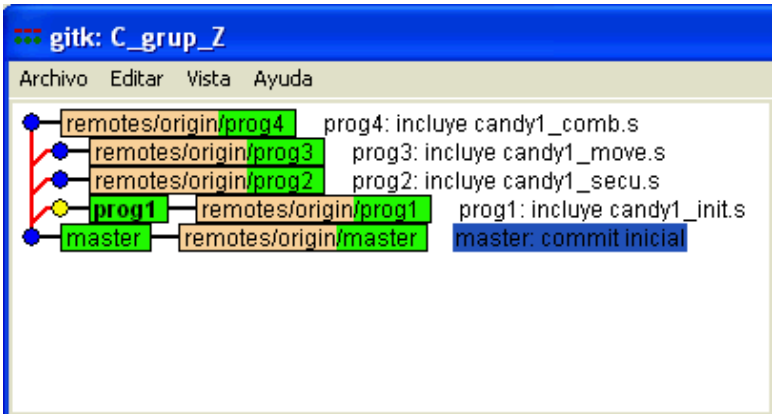
El comando '**git checkout**' permite cambiar rápidamente el *commit* desplegado sobre el sistema de ficheros. Por ejemplo, vamos a cambiar a un *commit* de una rama concreta (cada programador debe cambiar a su rama):

```
(master)$ git checkout prog1
Branch prog1 set up to track remote branch prog1 from origin.
Switched to a new branch 'prog1'
(prog1)$
```

En este momento se ha creado una nueva rama local 'prog1' (o la rama indicada), y se ha cambiado el sistema de ficheros por el contenido del *commit* apuntado por dicha rama. Esto se puede observar rápidamente en la carpeta "source" que habíamos abierto anteriormente:



Hemos resaltado con una elipse magenta el nuevo fichero que Git ha colocado en la carpeta automáticamente, puesto que dicho fichero está contenido en el nuevo *commit* desplegado sobre el sistema de ficheros; el nuevo *commit* se marcará con un punto amarillo en el panel de *commits* del ***gitk***, después de invocar el comando **Actualizar** del menú **Archivo**:



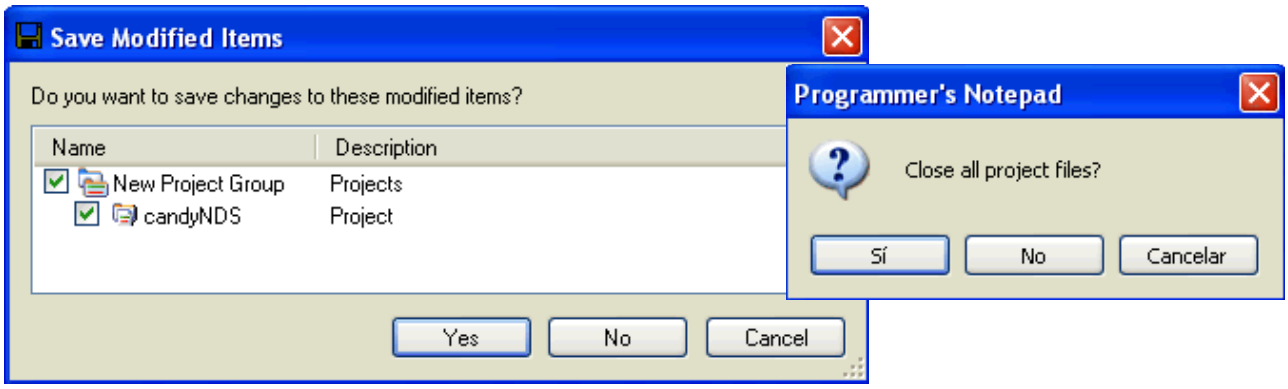
Además de la creación de la nueva etiqueta para la rama 'prog1' local, en la figura anterior se puede observar como el *commit* seleccionado para ver su contenido en ***gitk*** (fondo azul) y el *commit* desplegado en el sistema de archivos (punto amarillo) puede que no sean el mismo.

5 Modificar el repositorio local

A continuación vamos a introducir cambios en los ficheros y a crear un par de *commits* en el repositorio local, contenido en el disco duro de nuestro ordenador.

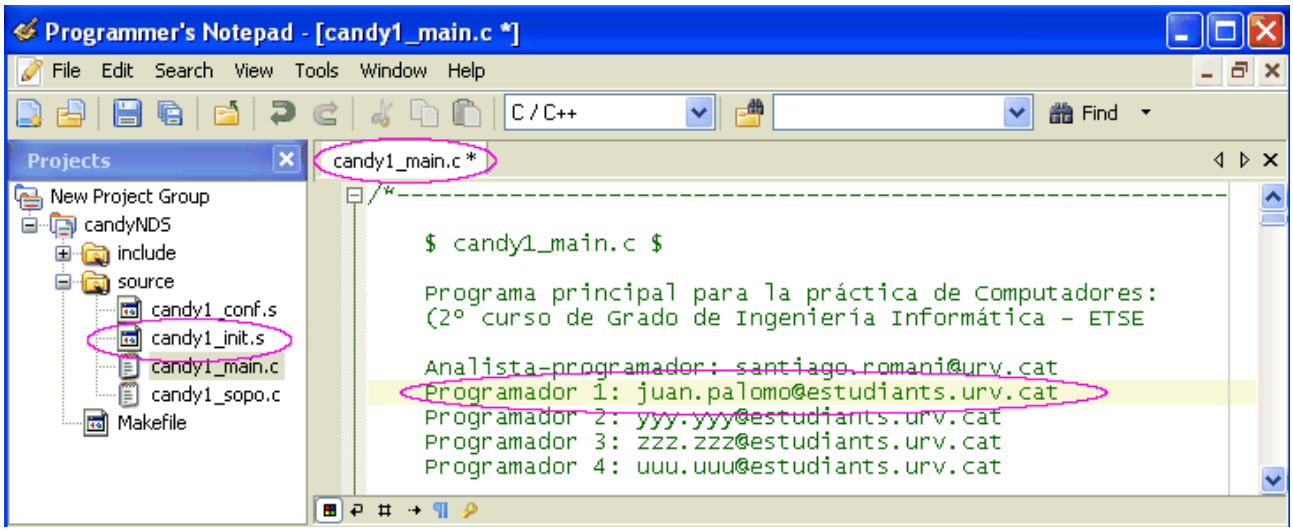
El primer cambio será muy simple: cambiaremos la línea de comentario del programador asignado con nuestro nombre.

Antes de hacer el cambio, cerraremos el proyecto de ***Programmer's Notepad*** (comando "**Close Project(s)**" del menú **File**) y lo volveremos a abrir para que se actualice al contenido actual de los ficheros:



A la pregunta del ***Programmer's Notepad*** sobre si queremos guardar los cambios en el fichero 'candyNDS.pnprj' (*Project*), habitualmente contestaremos "**No**", puesto que no se trata de ningún cambio en el código fuente, solo en la definición del proyecto que se actualiza cada vez que abrimos de nuevo el proyecto. A la pregunta de si queremos cerrar todos los ficheros del proyecto, podemos contestar "**Sí**".

Después de abrir de nuevo el proyecto, abriremos otra vez el fichero 'candy1_main.c' e introduciremos nuestro nombre en la línea de comentario correspondiente al rol que tengamos asignado:



En la figura anterior hemos resaltado, con una elipse magenta, el nuevo fichero 'candy1_init.s' que se ha desplegado con el cambio de *commit*, y que se ha incorporado al panel de proyectos del ***Programmer's Notepad*** como consecuencia de cerrar y volver a abrir dicho proyecto. Hay que tener en cuenta que cada programador dispondrá de un fichero adicional diferente. Además, se ha resaltado la línea que se ha cambiado dentro del fichero 'candy1_main.c'. Por último, se ha resaltado la marca que usa el ***Programmer's Notepad*** para indicar que el fichero 'candy1_main.c' se ha modificado.

Ahora ya podemos guardar los cambios en el fichero con el botón de salvar (icono en la barra de menús), o bien con el comando "**Save**" del menú **File**.

5.1 git status

El comando '**git status**' pide información al Git sobre el estado actual del repositorio local. Debido al cambio que hemos introducido, en este momento el comando escribirá por el terminal algo similar a lo siguiente:

```
(prog1)$ git status
# On branch prog1
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#     modified:   source/candy1_main1.c
#
no changes added to commit (use "git add" and/or "git commit -a")
```

El mensaje indica que el contenido del fichero 'candy1_main1.c' diverge del contenido que hay almacenado en la base de datos del Git para el *commit* actualmente desplegado. Además, se nos informa de las diversas opciones para añadir el cambio ('**git add**') o para deshacerlo y volver a la versión actual ('**git checkout**'), así como para crear un nuevo *commit* con dicho cambio ('**git commit -a**').

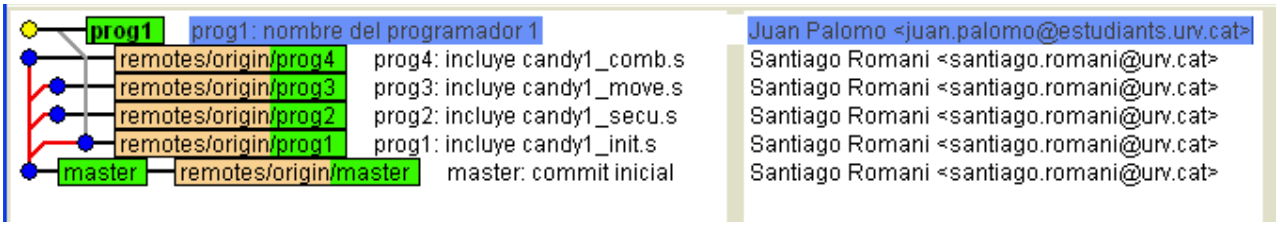
5.2 git commit

El comando '**git commit**' permite crear un nuevo *commit* con los cambios de los ficheros del proyecto sobre la rama local.

```
(prog1)$ git commit -a -m "prog1: nombre del programador 1"
[prog1 2bd527b] prog1: nombre del programador 1
 1 files changed, 1 insertions(+), 1 deletions(-)
```

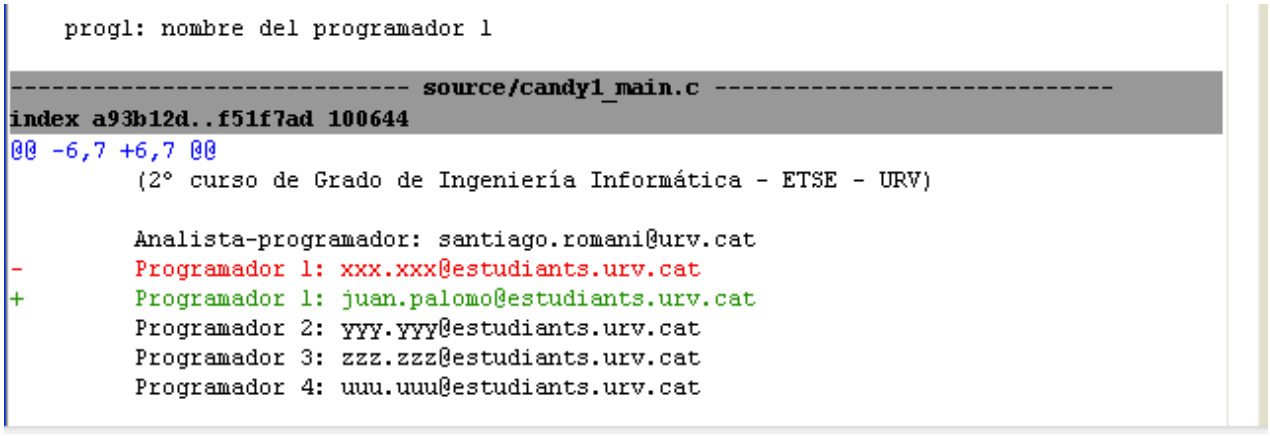
La opción '*-a*' sirve para añadir los últimos cambios en el *commit* actual (equivalente a '**git add .**') y la opción '*-m*' sirve para añadir un mensaje identificativo del contenido del *commit*. Como se puede observar, se ha empezado el mensaje con el identificador de la rama actual ("prog1:"), ya que el Git no etiqueta todos los *commits* de una rama, solo el último. Por lo tanto, este convenio nos servirá para identificar rápidamente todos los *commits* de un rol determinado. Naturalmente, cada programador deberá usar su propio identificador.

Para observar el nuevo *commit*, tenemos que actualizar la ventana del ***gitk***:



Observar que hay un nuevo *commit* (punto amarillo) etiquetado como el último de la rama '**prog1**' local (etiqueta verde), pero diferente del último *commit* de la misma rama en el servidor '**remotes/origin/prog1**', aunque hay una línea (gris) que los une, lo cual indica que el último es una modificación del anterior.

Además, si en el ***gitk*** activamos la visualización del último *commit* en modo "**Parche**", podremos observar los cambios del *commit* seleccionado respecto a su *commit* anterior:



```
    prog1: nombre del programador 1

---------------------------- source/candy1_main.c ----------------------------
index a93b12d..f51f7ad 100644
@@ -6,7 +6,7 @@
          (2º curso de Grado de Ingeniería Informática - ETSE - URV)

          Analista-programador: santiago.romani@urv.cat
-         Programador 1: xxx.xxx@estudiants.urv.cat
+         Programador 1: juan.palomo@estudiants.urv.cat
          Programador 2: yyy.yyy@estudiants.urv.cat
          Programador 3: zzz.zzz@estudiants.urv.cat
          Programador 4: uuu.uuu@estudiants.urv.cat
```

En la zona de cambios (parches), las filas eliminadas se marcan con un carácter '-' y se escriben en rojo, mientras que las filas añadidas se marcan con un carácter '+' y se escriben en verde. El resto de filas en negro se añaden para mostrar el contexto de las líneas cambiadas, pero dichas filas tienen el mismo contenido en las dos versiones, así como el resto de filas que no se muestran. El modo "**Parche**" es muy útil para distinguir rápidamente las modificaciones introducidas en las versiones de todos los ficheros del proyecto.

Ahora vamos a introducir cambios más significativos en otros ficheros. Para ello, hay que descargarse, del espacio ***Moodle*** de la asignatura, un fichero llamado '**code_snippets_1.txt**'. Allí se encuentran trozos de rutinas para cada uno de los ficheros de la práctica. Hay que abrir el fichero con el ***Programmer's Notepad*** y copiar/pegar el trozo proporcionado para el fichero que tengamos asignado.

Para el caso del programador 1, hay que insertar el código de la rutina mod_random() en el fichero '**candy1_init.s**'. Para cada programador hay un trozo de rutina en un fichero diferente. Después de insertar el código y guardar el fichero, hay que hacer otro '**git commit**':

```
(prog1)$ git commit -a -m "prog1: incluye mod_random() en candy1_init.s"
[prog1 245831b] prog1: incluye mod_random() en candy1_init.s
 1 files changed, 20 insertions(+), 0 deletions(-)
```

En este caso no hay ninguna línea a eliminar, solo se han añadido nuevas líneas (20). Actualizando el ***gitk*** podremos observar un mapa de *commits* similar al siguiente:

Con este ejemplo se ve, por primera vez, un *commit* que no tiene etiqueta de rama: el *commit* anterior al actual. En este caso se muestra la utilidad de añadir la marca de la rama en la que estamos trabajando en el mensaje del *commit*, lo cual no es obligatorio para Git, pero si recomendable para nuestro trabajo.

Otra observación a tener en cuenta es que nuestra rama local ya se encuentra 2 *commits* por delante del contenido de la rama correspondiente en el servidor. El comando '**git status**' notifica esta situación por el terminal de la siguiente forma:

```
(prog1)$ git status
# On branch prog1
# Your branch is ahead of 'origin/prog1' by 2 commits.
#
nothing to commit (working directory clean)
```

Evidentemente, en algún momento habrá que subir los cambios al servidor para que sean accesibles al resto de los compañeros del equipo. Sin embargo, antes de explicar como realizar dicha actualización, en el siguiente apartado vamos a introducir algunos errores a propósito sobre la rama local para aprender cómo se puede recuperar el contenido de versiones anteriores.

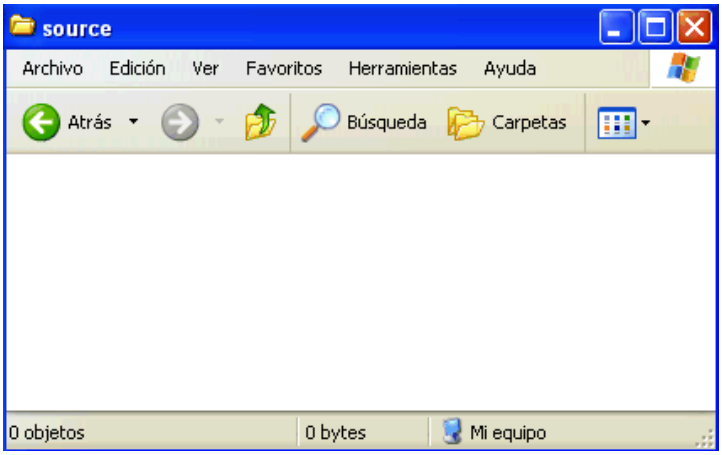
6 Recuperación de contenidos

A continuación vamos a introducir errores a propósito, para luego volver a un estado anterior del proyecto.

Primero vamos a borrar todos los ficheros fuente. Esto se puede hacer directamente desde el visor de ficheros y carpetas del Windows, o bien con el siguiente comando:

```
(prog1)$ rm source/*
```

Se puede comprobar en el visor de Windows que todos los ficheros del directorio "source" han desaparecido (!):



El comando '**git status**' nos informa de la diferencia entre el árbol de directorios y ficheros del proyecto y el contenido del *commit* actual en la base de datos del Git:

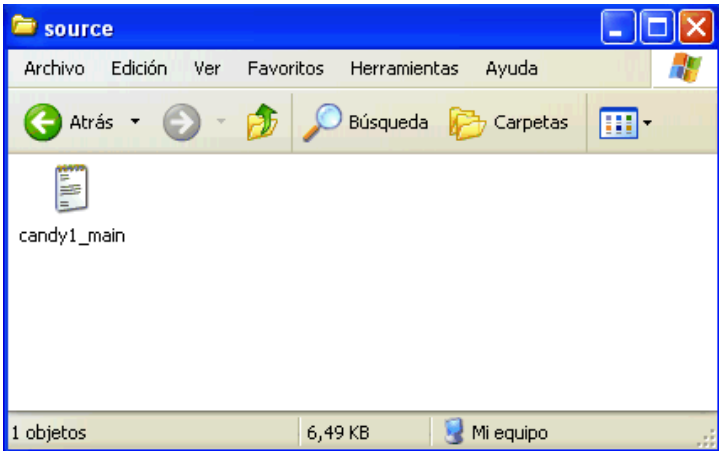
```
(prog1)$ git status
# On branch prog1
# Your branch is ahead of 'origin/prog1' by 2 commits.
#
# Changes not staged for commit:
#   (use "git add/rm <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       deleted:    source/candy1_conf.s
#       deleted:    source/candy1_init.s
#       deleted:    source/candy1_main.c
#       deleted:    source/candy1_sopo.c
#
no changes added to commit (use "git add" and/or "git commit -a")
```

6.1 *git checkout -- <file>*

Supongamos que queremos recuperar el fichero 'candy1_main.c'; esto lo podemos conseguir con el siguiente comando:

```
(prog1)$ git checkout -- source/candy1_main.c
```

Si visualizamos otra vez el contenido de la carpeta, observaremos que hemos recuperado el fichero indicado:

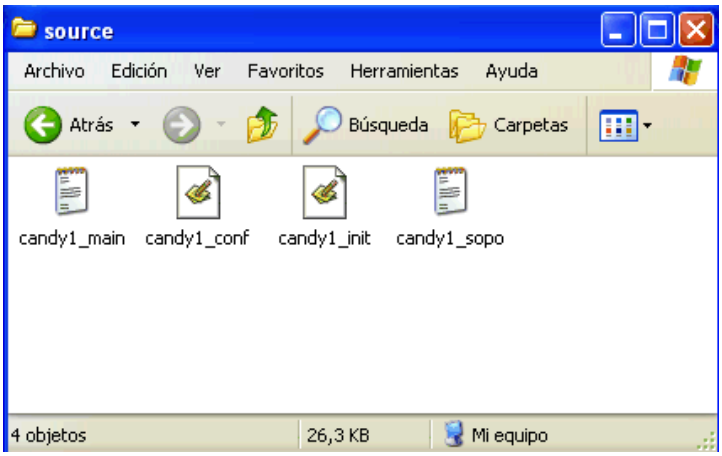


6.2 *git reset --hard*

Supongamos que queremos recuperar todos los ficheros que faltan; esto lo podemos conseguir con el siguiente comando:

```
(prog1)$ git reset --hard
HEAD is now at 245831b prog1: incluye mod_random() en candy1_init.s
```

Esta vez, el contenido de la carpeta "source" será idéntico al que teníamos antes de eliminar todos los ficheros:



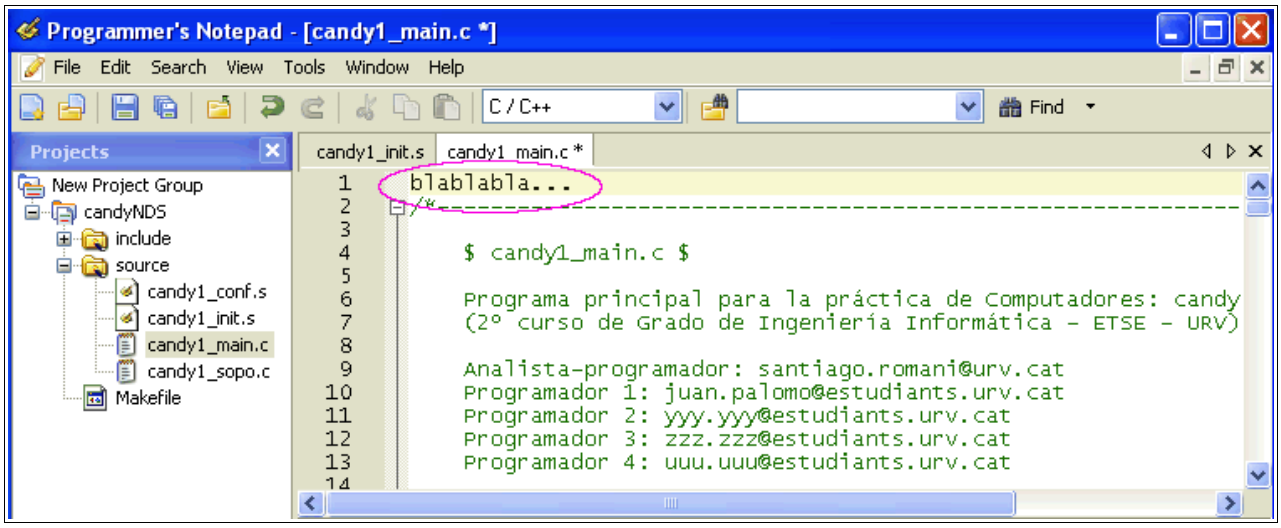
Hay que aclarar que los comandos de recuperación '**git checkout -- <file>**' y '**git reset --hard**' no solo sirven para restablecer ficheros borrados, sino que pueden recuperar el estado de ficheros modificados o incluso eliminar nuevos ficheros añadidos erróneamente.

La idea clave siempre es que el árbol de directorios y ficheros nos permite introducir cambios de la manera habitual (editor de texto, comandos de sistema, etc.), pero el contenido válido del proyecto es lo que hay registrado en la base de datos del Git.

6.3 *git checkout <ID commit>*

Ahora vamos a introducir un error en un fichero y vamos a crear el *commit* correspondiente, con el fin de observar como podemos "ignorar" un *commit* erróneo de la base de datos local del Git.

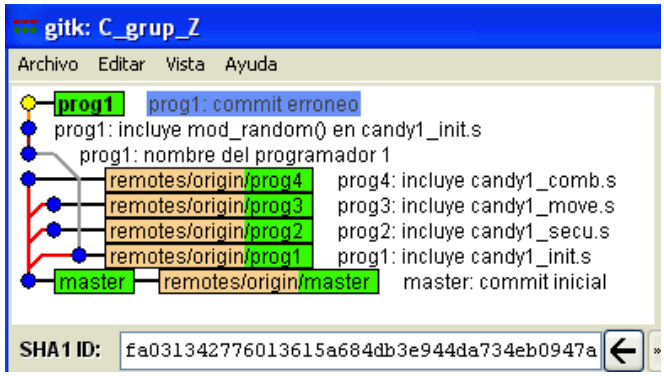
Primero hay que editar un fichero del proyecto. Por ejemplo, con el ***Programmer's Notepad*** añadiremos la siguiente línea al principio del fichero 'candy1_main.c':



Guardaremos este cambio y generaremos el *commit* erróneo:

```
(prog1)$ git commit -a -m "prog1: commit erroneo"
[prog1 fa03134] prog1: commit erroneo
 1 files changed, 1 insertions(+), 0 deletions(-)
```

La actualización del ***gitk*** nos permitirá observar el nuevo *commit*:



Ahora se supone que nos damos cuenta de que el *commit* que acabamos de insertar en la base de datos local del Git es incorrecto (!). Para volver al *commit* anterior, o a un *commit* cualquiera de la base de datos, podemos utilizar el comando '**git checkout <ID commit>**', donde <ID commit> es el identificador del *commit* donde nos queremos trasladar.

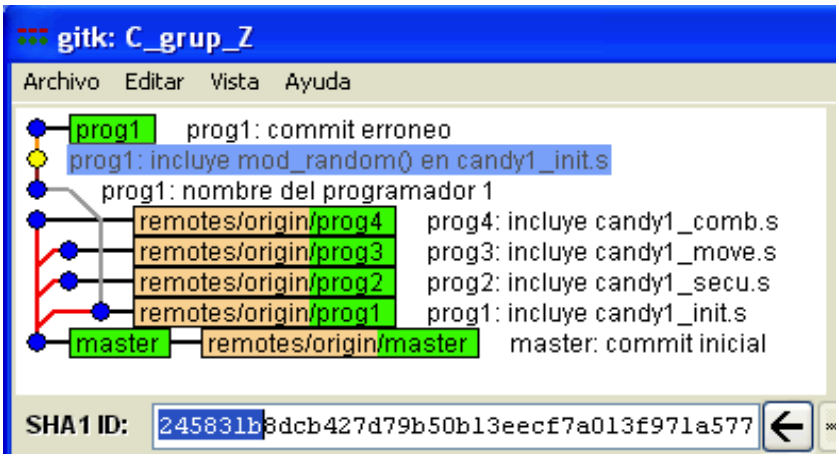
Existen diversas modos de especificar un *commit*, como por ejemplo, usando la etiqueta de una rama que esté referenciando el *commit* (ver apartado 4.3). En este caso, sin embargo, queremos posicionarnos sobre un *commit* que no está referenciado por ninguna etiqueta de rama. Por lo tanto, podemos utilizar los 7 primeros dígitos del identificador SHA1 del último *commit* que hemos creado en el apartado 5.2 (hay que consultar estos dígitos en el ***gitk***, puesto que en cada repositorio serán diferentes):

```
(prog1)$ git checkout 245831b
Note: checking out '245831b'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b new_branch_name

HEAD is now at 245831b... prog1: incluye mod_random() en candy1_init.s
```

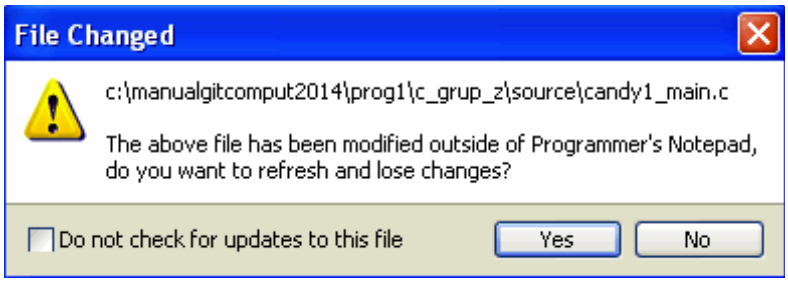
Este mensaje nos avisa que estamos en modo '**detached HEAD**', que significa que el *commit* desplegado actualmente no está siendo apuntado por ninguna etiqueta de rama. La misma información la podemos obtener con '**git status**':

```
((245831b))$ git status
# Not currently on any branch.
nothing to commit (working directory clean)
```

También podemos actualizar el ***gitk*** para observar cómo el *commit* desplegado actualmente no está referenciado por ninguna etiqueta de rama:



Sin embargo, el fichero 'candy1_main.c' ha vuelto a su estado anterior. Esto se puede observar volviendo al ***Programmer's Notepad,*** el cual detectará la actualización del fichero y nos preguntará si queremos trasladar esa actualización al visor del fichero:



Con la opción "**Yes**" restableceremos el contenido anterior del fichero y desaparecerá la línea errónea.

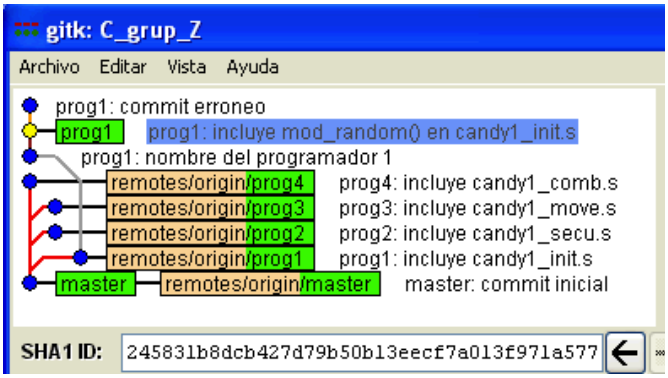
6.4 *git branch -f <ID rama> <ID commit>*

En modo '**detached HEAD**' podemos seguir introduciendo cambios en los ficheros y crear nuevos *commits* a partir del *commit* actual. Sin embargo, es muy recomendable que una etiqueta de rama apunte al *commit* desplegado actualmente, puesto que las etiquetas de rama están pensadas para referenciar los *commits* de forma simbólica, es decir, sin tener que introducir los 7 primeros dígitos del código SHA1.

En el estado actual del ejemplo, tenemos que forzar la etiqueta de nuestra rama de trabajo 'prog1' (o la del programador que tengamos asignado) a que apunte al *commit* actual. Esto se puede conseguir con '**git branch -f <ID rama> <ID commit>**':

```
((245831b))$ git branch -f prog1 245831b
```

Actualizando el ***gitk*** observaremos una evolución de *commits* similar a la siguiente:



Sin embargo, todavía tendremos que invocar un '**git checkout <ID rama>**' para que ***Git Bash*** reconozca esta actualización del puntero de rama:

```
((245831b))$ git status
# Not currently on any branch.
nothing to commit (working directory clean)

((245831b))$ git checkout prog1
Switched to branch 'prog1'
Your branch is ahead of 'origin/prog1' by 2 commits.

(prog1)$
```

Tal como está ahora el historial de *commits* del proyecto, el *commit* erróneo **no** se ha eliminado de la base de datos. Esto no es problema puesto que cada *commit* ocupa muy poco espacio de memoria (solo se guardan los cambios respecto al *commit* anterior). Cuando se añadan nuevos *commits* a partir del *commit* actual, el *commit* erróneo quedará "descolgado" de las líneas de evolución de *commits* y no interferirá de ningún modo con los *commits* correctos.

En determinadas configuraciones del historial de *commits*, es posible borrar definitivamente los *commits* colgantes (*dangling commits*) con el comando '**git prune**'. En cualquier caso, dentro del entorno ***gitk***, si en vez de invocar la opción "***Actualizar***" del menú **Archivo**, invocamos la opción "***Reload***" del mismo menú, los *commits* colgantes ya no se mostrarán.

7 Sincronización con el repositorio remoto

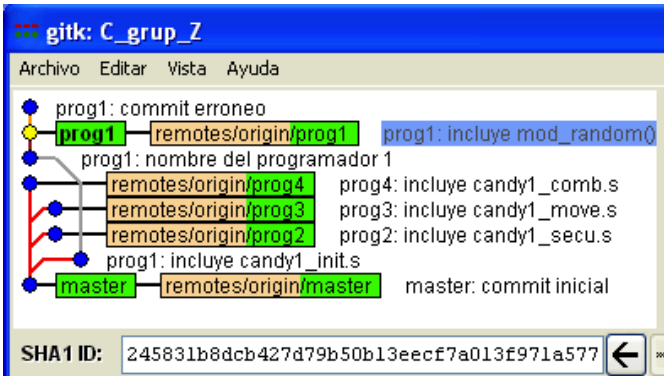
A continuación vamos a subir los *commits* del repositorio local al repositorio remoto alojado en el servidor. También tendremos que bajar (y fusionar) los *commits* que suban nuestros compañeros en el mismo repositorio remoto.

7.1 git push origin <ID rama>

El comando '**git push origin <ID rama>**' permite subir todos los *commits* de la rama <ID rama> que todavía no estén registrados en la rama correspondiente del servidor. Por ejemplo:

```
(prog1)$ git push origin prog1
Counting objects: 12, done.
Compressing objects: 100% (8/8), done.
Writing objects: 100% (8/8), 1.14 KiB, done.
Total 8 (delta 5), reused 0 (delta 0)
Unpacking objects: 100% (8/8), done.
To git.lab.deim:C_grup_Z
   b0e33e0..245831b  prog1 -> prog1
```

Con un solo comando '**git push**' se han subido 2 *commits* nuevos en el servidor (excluido el *commit* erróneo). Actualizando el ***gitk*** observaremos una visualización similar a la siguiente:



En la figura anterior hay que observar que la etiqueta 'remote/origin/prog1' está al lado de la etiqueta 'prog1'. Esto significa que la base de datos del repositorio local (en nuestro ordenador) **está sincronizada** con la base de datos del repositorio remoto (en el servidor).

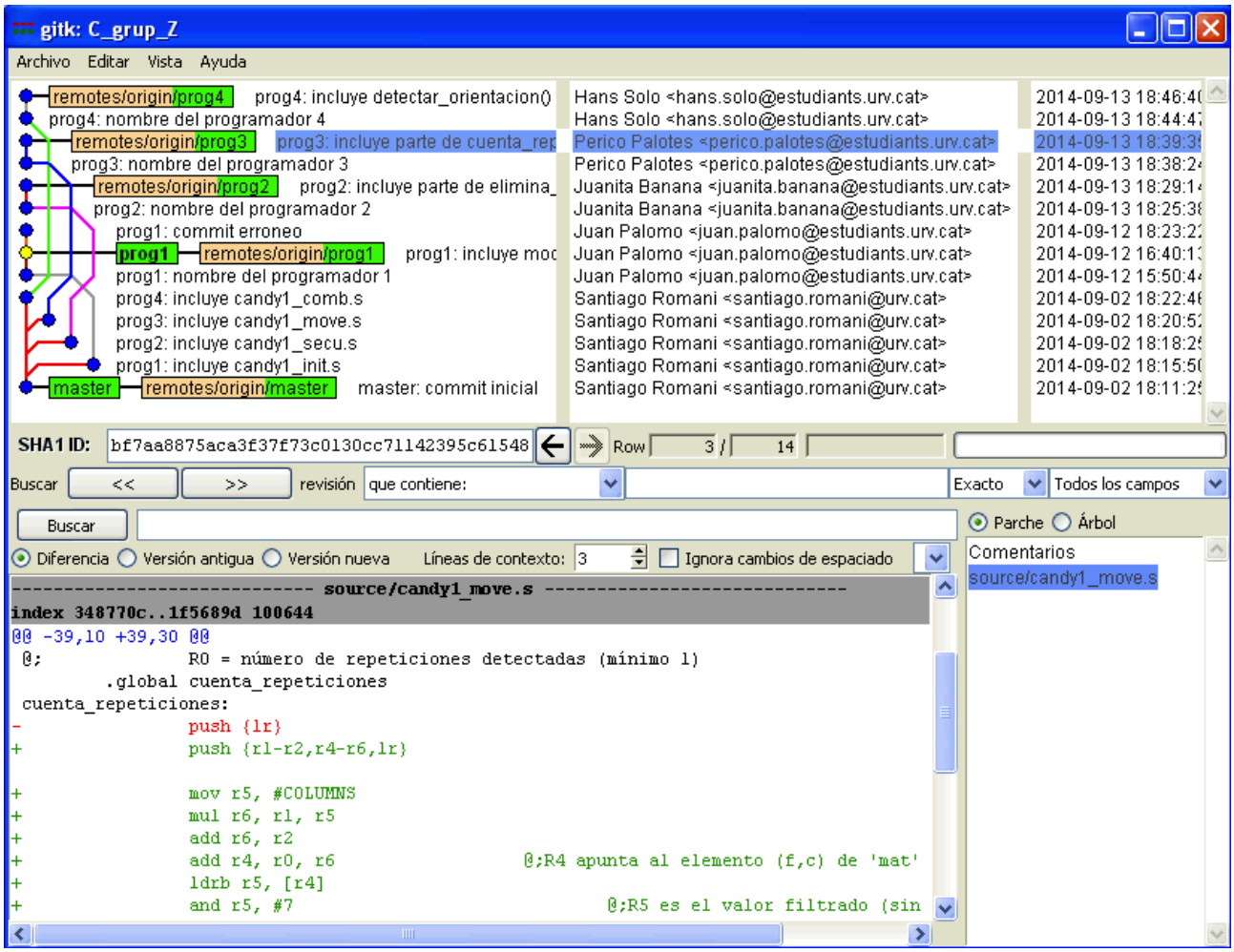
ATENCIÓN: Es muy importante entender este paso, puesto que la práctica **no** se considerará entregada si los *commits* realizados en el repositorio local no están subidos correctamente al servidor.

7.2 *git fetch*

El comando '**git fetch**' permite descargar las últimas actualizaciones del repositorio remoto sobre nuestro repositorio local. Es decir, con este comando podremos ver el contenido que ha subido al servidor cada programador hasta el momento actual:

```
(prog1)$ git fetch
remote: Counting objects: 34, done.
remote: Compressing objects: 100% (24/24), done.
remote: Total 24 (delta 15), reused 0 (delta 0)
Unpacking objects: 100% (24/24), done.
From git.lab.deim:C_grup_Z
   541e3bb..3c9e2de  prog2      -> origin/prog2
   99ce007..bf7aa88  prog3      -> origin/prog3
   2aecf05..48c2dbb  prog4      -> origin/prog4
```

Actualizando el ***gitk*** observaremos una evolución de *commits* similar a la siguiente:



En la parte superior de la ventana del ***gitk*** se pueden ver los *commits* que han subido los compañeros en sus respectivas ramas remotas, junto con el nombre del programador y la fecha de generación de cada *commit*. Hay que observar también que las ramas remotas no

contienen los *commits* erróneos introducidos en la sección 6.3, aunque esos *commits* estén registrados localmente en el ordenador de cada programador.

En la parte inferior de la ventana se ha visualizado el parche del último *commit* subido por el programador 3, simplemente seleccionado dicho *commit* en el panel superior (fondo azul). Es un ejemplo de cómo, con el ***gitk***, podemos inspeccionar los cambios ("**Parche**") o el código fuente ("**Árbol**") de los *commits* que nuestros compañeros han subido al servidor, sin necesidad de crear las ramas locales de los otros programadores.

8 Fusión de *commits*

A continuación vamos a fusionar *commits*, es decir, a mezclar el código fuente de distintas versiones del proyecto, típicamente para incluir el trabajo de los programadores en la rama 'master', aunque se pueden realizar otros tipos de fusiones.

Este proceso puede resultar complicado porque las diferencias en el contenido de las diversas versiones de cada fichero provocan **conflictos que el Git no puede resolver**. En este caso, Git simplemente marcará las diferencias dentro de cada fichero, y somos nosotros, los programadores, los que tenemos que editar el fichero y crear la versión definitiva.

En realidad, la dificultad de la fusión no reside en editar los ficheros, sino en **integrar el código**, es decir, conseguir que dentro de una misma versión **funcionen** los códigos que diversos programadores han creado por separado.

ATENCIÓN: para intentar minimizar el número de fusiones, vamos a realizarlas todas desde un único ordenador local, es decir, uno de los programadores tomará el rol de 'master' y realizará dichas fusiones con la ayuda de los otros compañeros de proyecto.

8.1 *git merge --no-ff <ID rama>*

El comando '**git merge**' tratará de fusionar el *commit* desplegado actualmente con otro *commit* cualquiera, habitualmente referenciado mediante un identificador de rama <ID rama>, lo cual se puede interpretar como una fusión entre ramas.

Lo primero que haremos será colocarnos en la rama 'master', puesto que es la rama sobre la que se tiene que ir generando la versión definitiva del proyecto. En este manual supondremos que será el programador de la rama 'prog1' el que tomará el rol de 'master', pero podría ser cualquiera de los otros programadores:

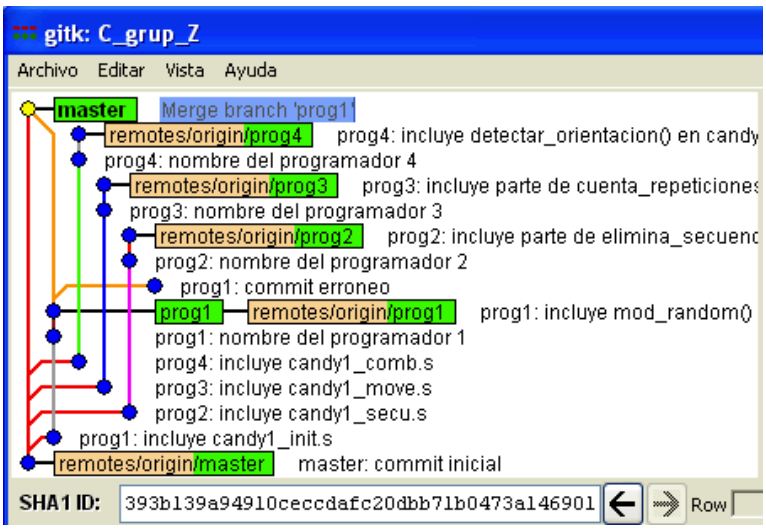
```
(prog1)$ git checkout master
Switched to branch 'master'
```

A continuación fusionaremos el último *commit* de la rama 'master' con el último *commit* de nuestra rama local de trabajo (en este caso 'prog1'), utilizando el comando '**git merge**':

```
(master)$ git merge --no-ff prog1
Merge made by recursive.
 candyNDS.pnproj       |    2 +-
 source/candy1_init.s |  146 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 source/candy1_main.c |    2 +-
 3 files changed, 148 insertions(+), 2 deletions(-)
 create mode 100644 source/candy1_init.s
```

Este primer *merge* no produce ningún conflicto, puesto que el último *commit* de 'prog1' (o de la rama que corresponda) era una derivación del *commit* anterior de 'master', de modo que Git asume que simplemente hay que trasladar a la rama 'master' los últimos cambios introducidos en la rama 'prog1'.

Con la actualización del contenido del ***gitk*** podremos observar el nuevo *commit* de fusión:



El *commit* de fusión lo ha generado Git automáticamente (no hace falta hacer '**git commit**'), y se puede distinguir porque confluyen dos líneas de evolución de *commits*. El mensaje del *commit* también se ha fijado automáticamente.

La opción '***--no-ff***' impide que el Git aplique una fusión ***fast-fordward***, lo que provocaría que la rama 'master' se posicionara sobre el *commit* apuntado por la rama 'prog1'. En general, queremos mantener las ramas separadas, por lo tanto, **siempre adjuntaremos la opción *--no-ff*** (*no-fastforward*) al comando '**git merge**'.

Ahora vamos a fusionar la rama 'master' con la rama 'prog2'. Si repetimos el mismo patrón que el 'git merge' anterior, seguramente se producirá un error:

```
(master)$ git merge --no-ff prog2
fatal: 'prog2' does not point to a commit
```

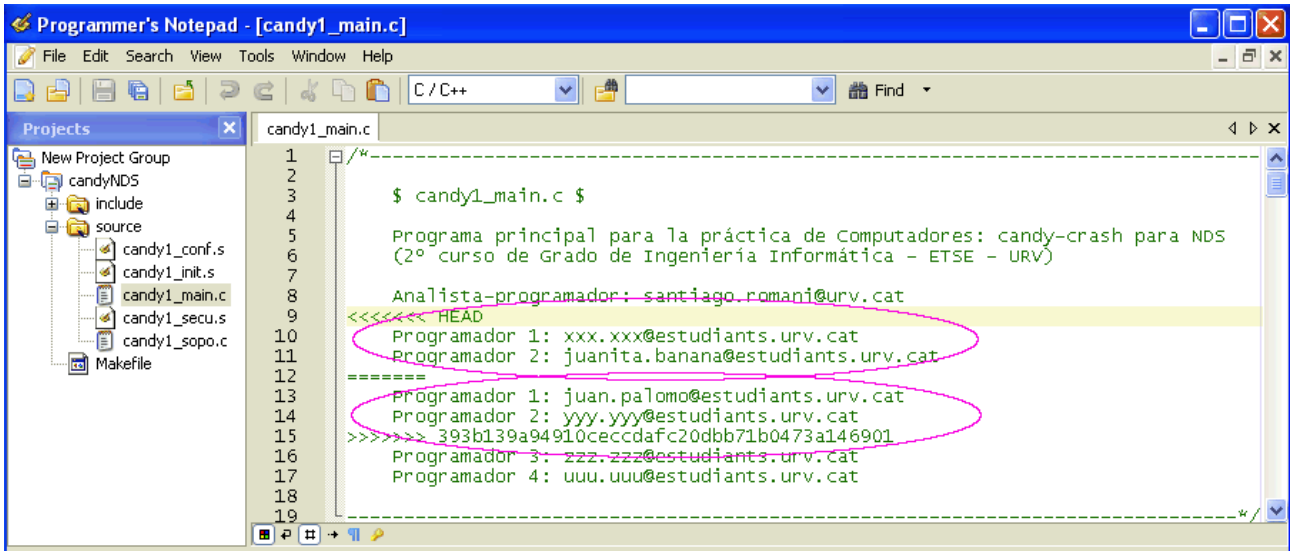
Esto es debido a que estamos trabajando en el ordenador del 'prog1', que no necesita tener una rama local de nombre 'prog2'. Para solucionar este pequeño inconveniente, podemos utilizar la referencia remota al último *commit* subido por 'prog2' al servidor:

> **ATENCIÓN**: se supone que, antes de hacer las fusiones, todos los programadores habrán subido su última versión al servidor y el programador que toma el rol de 'master' habrá realizado un '**git fetch**' para obtener dichas versiones.

```
(master)$ git merge --no-ff remotes/origin/prog2
Auto-merging source/candy1_main.c
CONFLICT (content): Merge conflict in source/candy1_main.c
Automatic merge failed; fix conflicts and then commit the result.

(master|MERGING)$
```

En este caso, Git solo ha encontrado un fichero conflictivo, el 'source/candy1_main.c'. El resto de los ficheros han sido incorporados sin ningún problema a la rama 'master', bien porqué tenían el mismo contenido en los dos *commits*, como por ejemplo, el 'source/candy1_conf.s', bien porqué eran propios de cada rama: 'source/candy1_init.s', proveniente de la rama 'prog1', y 'source/candy1_secu.s', de la rama 'prog2'.

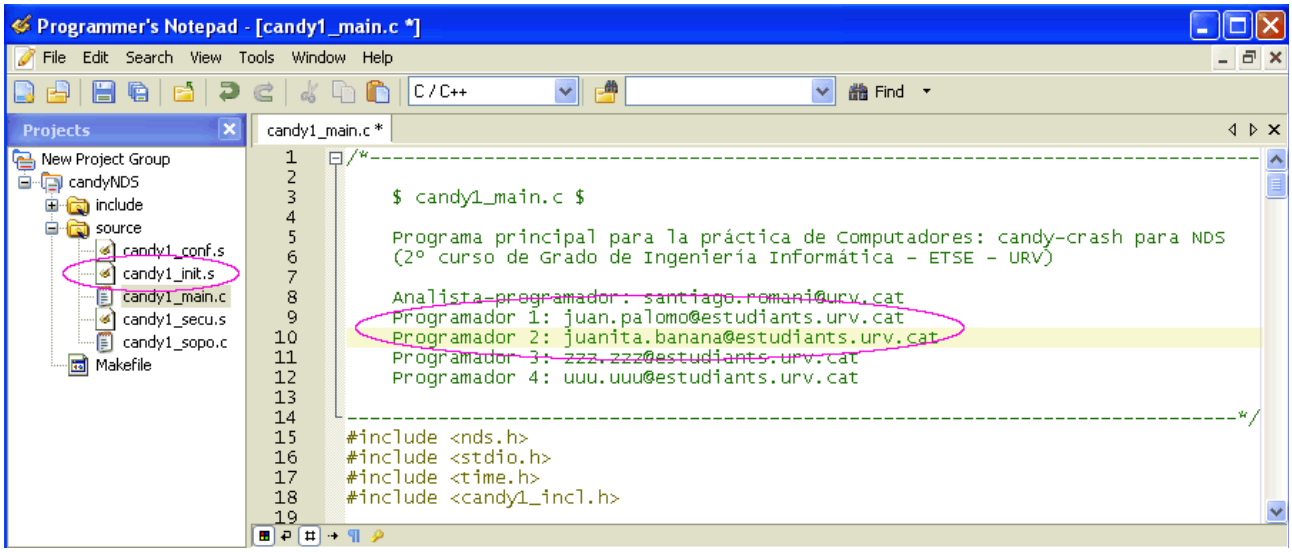
Para solucionar los conflictos de 'candy1_main.c', tendremos que editar el fichero con el ***Programmer's Notepad***:



En el gráfico anterior, sobre la ventana de edición del fichero 'candy1_main.c' se ha resaltado (con elipses magenta) cómo el Git ha indicado las líneas donde ha encontrado conflictos y no los ha podido resolver. Concretamente, se han generado las siguientes marcas:

```
<<<<<<< HEAD
texto contenido en el commit desplegado actualmente (HEAD)
...
=======
texto contenido en el commit con el que hay que fusionar
...
>>>>>>> (ID commit a fusionar)
```

En este ejemplo, es fácil darse cuenta de cuál tiene que ser el contenido definitivo:



Manualmente, hay que quitar todas las marcas que ha insertado Git y las líneas de nuestro código fuente a descartar, dejando solo las líneas definitivas, que en el ejemplo corresponden a los identificadores (correo electrónico) de los programadores 1 y 2.

A parte del contenido final, en el gráfico anterior también se ha resaltado el fichero 'candy1_init.s', que se ha introducido debido al *commit* de fusión subido por el programador 1, ya que la rama del programador 2 no contenía este fichero, sino el 'candy1_secu.s'.

Una vez modificado y guardado el contenido definitivo de 'candy1_main.c', hay que crear **manualmente** un nuevo *commit* de fusión con los nuevos cambios:

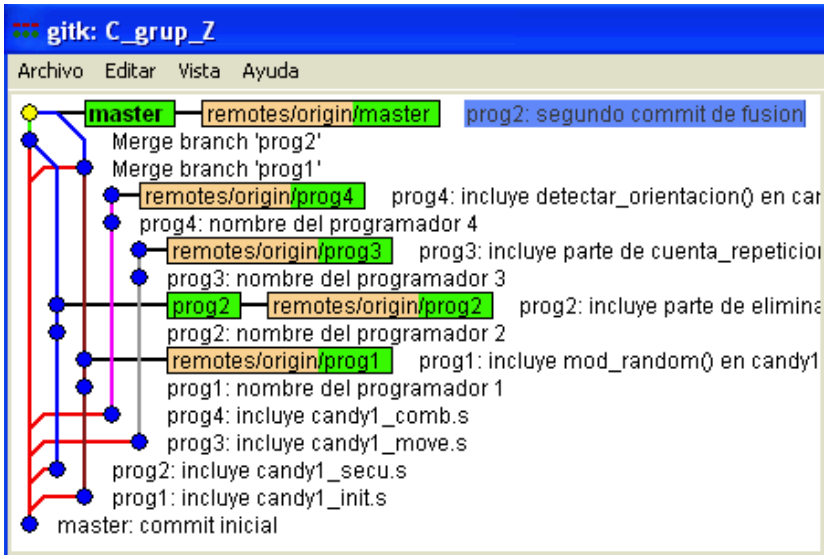
```
(master|MERGING)$ git commit -a -m "master: segundo commit de fusion"
[master eaa3b1c] master: segundo commit de fusion

(master)$
```

Hay que observar que, en el *prompt* del ***Git Bash***, el identificador de la rama después de realizar un 'git merge' conflictivo se marca con el mensaje "**|MERGING**", para indicar que el proceso de fusión todavía no ha terminado.

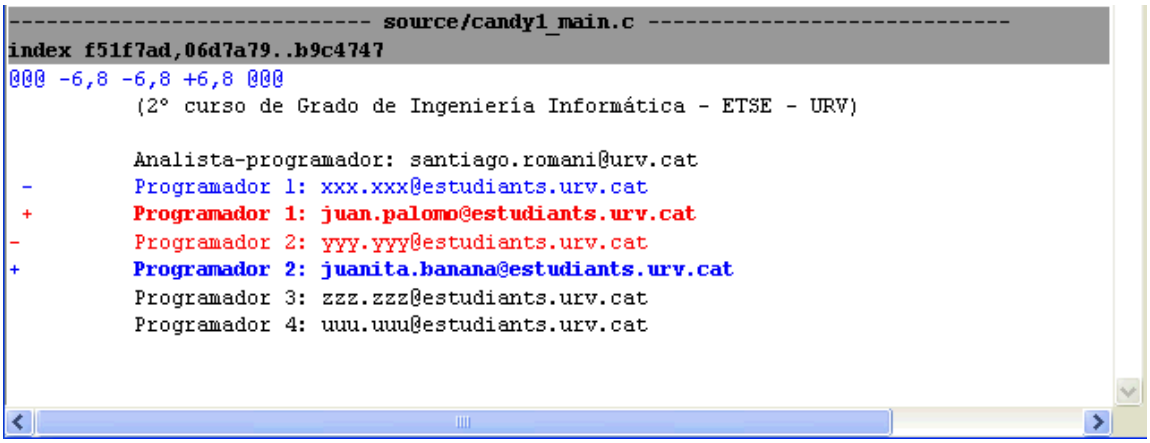
Si no queremos continuar con esa fusión, para volver al estado anterior solo es necesario introducir el comando '**git merge --abort**'.

Si realizamos un nuevo *commit*, el Git entenderá que continuamos adelante y que hemos resuelto los conflictos, de modo que desaparecerá el mensaje "|MERGING" de detrás del nombre de la rama.

Si actualizamos el ***gitk*** del repositorio local donde estamos realizando las fusiones, obtendremos una visualización similar a la siguiente:



Además, en el contenido del "**Parche**" del último *commit* de fusión, el ***gitk*** nos informa de los cambios respecto a los *commits* fusionados del siguiente modo:



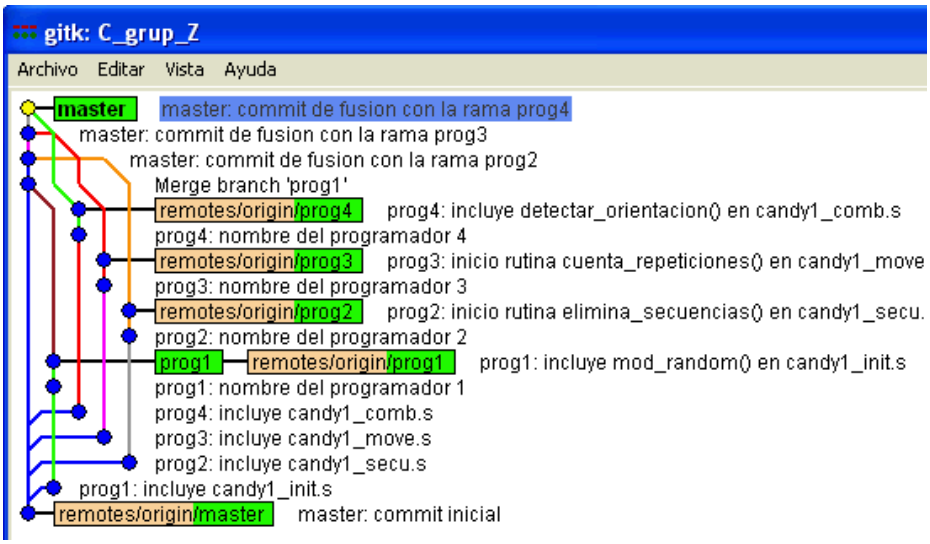
En rojo muestra las líneas del *commit* desplegado actualmente ('master') y en azul muestra las líneas del *commit* fusionado ('remotes/origin/prog2'), añadiendo un '+' para las líneas añadidas, además de usar texto en negrita, y un '-' para las líneas eliminadas.

Para completar la fusión del código fuente de todos los programadores, repetiremos el proceso anterior con el resto de las ramas, es decir, con 'remotes/origin/prog3' y con 'remotes/origin/prog4'.

```
(master)$ git merge --no-ff remotes/origin/prog3
Auto-merging source/candy1_main.c
CONFLICT (content): Merge conflict in source/candy1_main.c
Automatic merge failed; fix conflicts and then commit the result.

<<< arreglar conflictos en 'source/candy1_main.c' >>>

(master|MERGING)$ git commit -a -m "master: tercer commit de fusion"
[master ac8540f] master: tercer commit de fusion
```

```
(master)$ git merge --no-ff remotes/origin/prog4
Auto-merging source/candy1_main.c
CONFLICT (content): Merge conflict in source/candy1_main.c
Automatic merge failed; fix conflicts and then commit the result.

<<< arreglar conflictos en 'source/candy1_main.c' >>>

(master|MERGING)$ git commit -a -m "master: cuarto commit de fusion"
[master 0caa3ad] master: cuarto commit de fusion

(master)$
```

Si actualizamos el ***gitk*** de nuevo, observaremos como hemos obtenido una versión definitiva del proyecto en la rama 'master', donde confluyen las líneas de evolución de todas las demás ramas:



En este ejemplo, solo había conflictos en un mismo fichero, y solo en los comentarios con los identificadores de los programadores. Sin embargo, en un entorno real, los conflictos serán mucho más complejos de resolver porque afectarán a trozos de código que nos obligarán a redefinir las estructuras de nuestros algoritmos.

Finalmente, hay que publicar todos los *commits* de la rama 'master' en el servidor, para que esa última versión del proyecto sea visible al resto de los programadores:

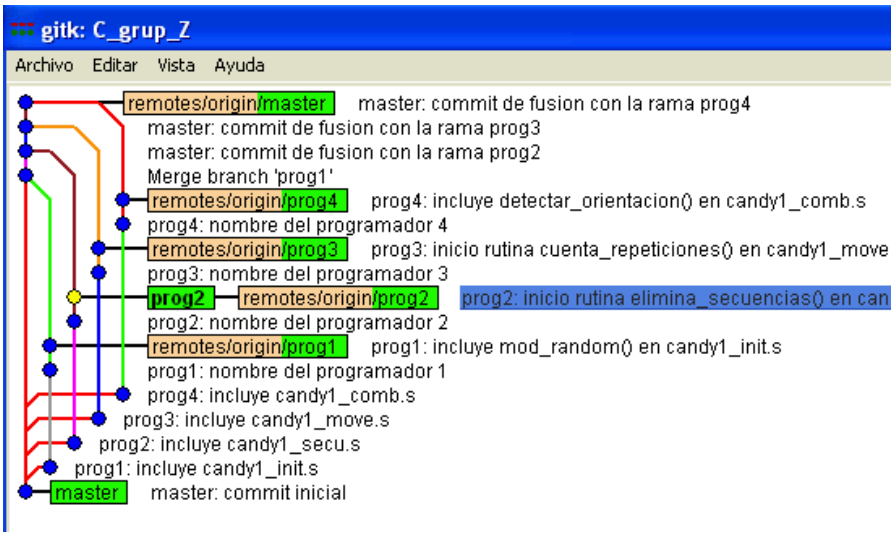
```
(master)$ git push origin master
Counting objects: 1, done.
Writing objects: 100% (1/1), 231 bytes, done.
Total 1 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (1/1), done.
To git.lab.deim:C_grup_Z c77da44..393b139 master -> master

(master)$
```

8.2 *git pull origin <ID rama>*

El comando '**git pull**' sirve para bajar y fusionar los últimos *commits* de una rama concreta desde el servidor. Esto permitirá al resto de programadores actualizar los *commits* de fusión generados sobre la rama 'master'.

Como ejemplo, vamos a mostrar los pasos que debe realizar el programador 2, pero estos mismos pasos deben ser aplicados por los programadores 3 y 4. El programador 1 no necesita sincronizar los *commits* de fusión de la rama 'master', porque ya los tiene en su repositorio local.

Primero, el programador 2 puede hacer un '**git fetch**' para obtener todos los nuevos *commits* del servidor. En el visor del historial de versiones del gitk podremos observar una evolución similar a la siguiente:



Como vemos, la rama 'master' local ha quedado muy por detrás del último *commit* de la rama 'remotes/origin/master': para sincronizarlas deberemos hacer un '**git pull**', pero antes hay que cambiar de la rama 'prog2' a la rama 'master', con el comando '**git checkout**':

```
(prog2)$ git checkout master
Switched to branch 'master'
Your branch is behind 'origin/master' by 16 commits, and can be fast-forwarded.

(master)$
```

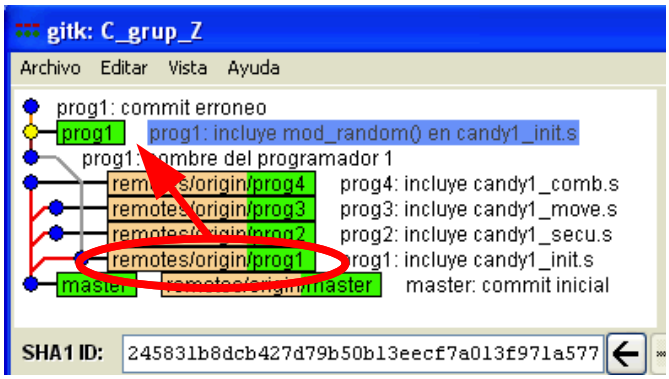
Al cambiar de rama, el Git nos informa de la diferencia de *commits* entre las ramas 'master' local y remota. Además, el mensaje del Git nos indica que se puede realizar un "avance rápido" de la rama local ("*can be fast-forwarded*"). Es el momento de hacer un '**git pull**':

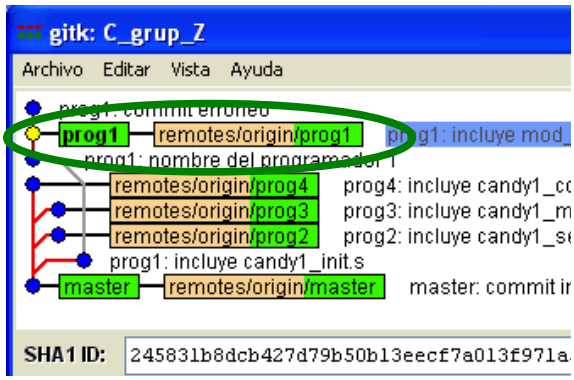
```
(master)$ git pull origin master
remote: Counting objects: 1, done.
remote: Total 1 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (1/1), done.
From e:/Docencia/Computadors.2014-15/Practica/C_grup_Z
 * branch            master     -> FETCH_HEAD
Updating abd91b6..0caa3ad
Fast-forward
 source/candy1_comb.s |  162 ++++++++++++++++++++++++++++++++++++++++++++++++++
 source/candy1_init.s |  146 ++++++++++++++++++++++++++++++++++++++++++++++
 source/candy1_main.c |    8 +-
 source/candy1_move.s |  124 ++++++++++++++++++++++++++++++++++++++
 source/candy1_secu.s |  134 ++++++++++++++++++++++++++++++++++++++++
 5 files changed, 570 insertions(+), 4 deletions(-)
 create mode 100644 source/candy1_comb.s
 create mode 100644 source/candy1_init.s
 create mode 100644 source/candy1_move.s
 create mode 100644 source/candy1_secu.s
```

Actualizando el ***gitk*** del programador 2 observaremos la actualización de la etiqueta local 'master', que quedará igualada a la etiqueta remota 'remotes/origin/master'.

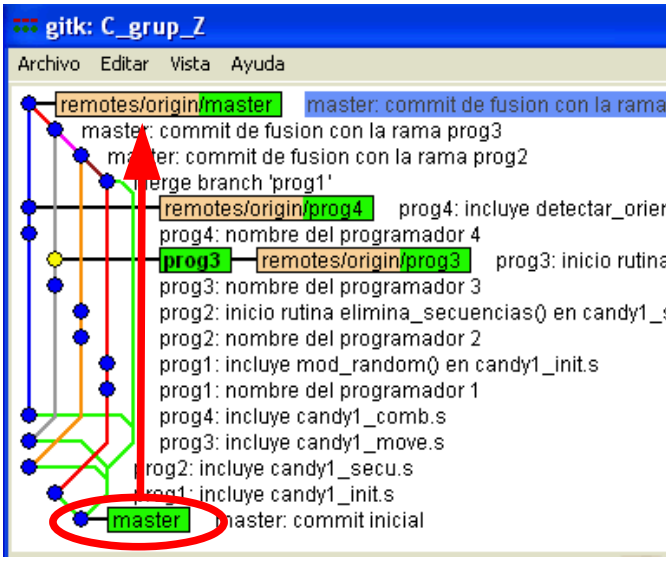
Por lo tanto, el proceso de sincronización de ramas con el servidor se puede entender en los siguientes términos:

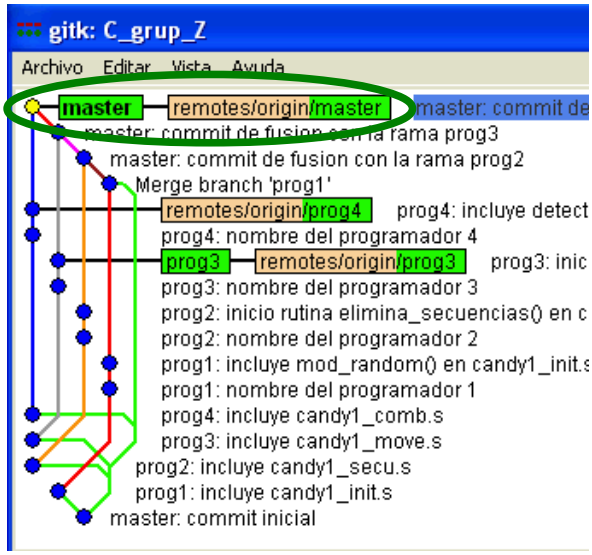
- '**git push**' sirve para avanzar la etiqueta remota de una rama hasta el *commit* apuntado por la etiqueta local,





- '**git pull**' sirve para avanzar la etiqueta local de una rama hasta el *commit* apuntado por la etiqueta remota.





Además, hay que saber que el comando '**git pull**' es equivalente a un '**git fetch**' seguido de un '**git merge**'. Por lo tanto, el último comando 'git pull origin master' se hubiese podido escribir como 'git merge remotes/origin/master', sabiendo que previamente habíamos realizado un 'git fetch'.

8.3 Actualización de las ramas de los programadores

Al finalizar la fase 1 de la práctica, será necesario realizar el proceso de fusión de todos los códigos fuente de los distintos programadores sobre la rama 'master', como se ha visto en los apartados anteriores.

Sin embargo, para continuar con la fase 2, todos los programadores deberán continuar modificando el proyecto en su rama de trabajo. Esto significa que tendremos que trasladar los cambios del último *commit* de la rama 'master' sobre la rama específica de cada programador. Vamos a realizar los pasos necesarios sobre el programador 1, pero el resto de programadores también deberán realizar dichos pasos en sus respectivos repositorios locales.

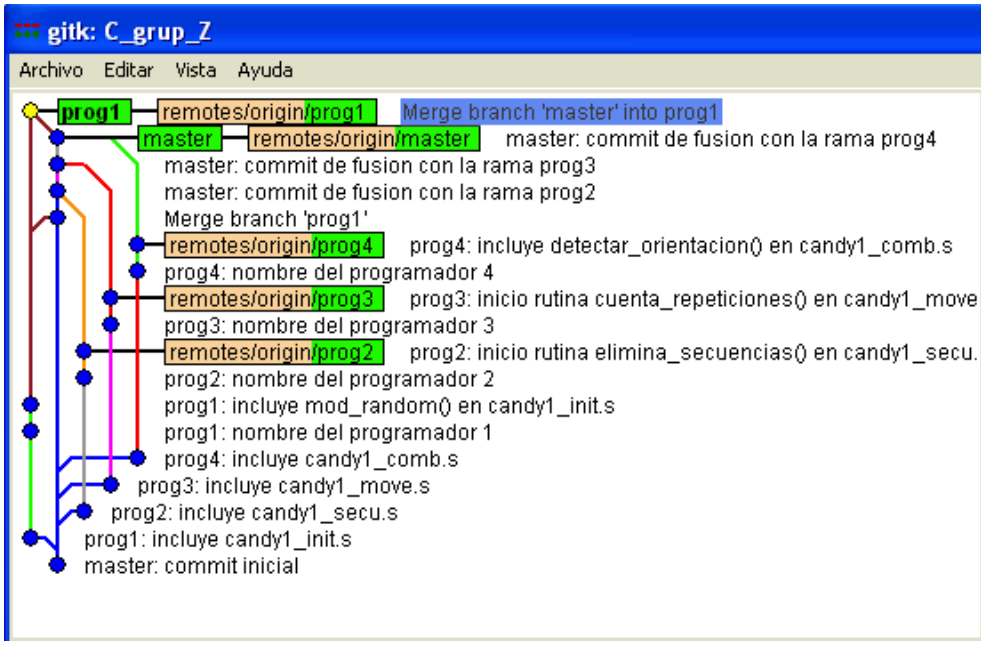
Primero, hay que cambiar otra vez a la rama del programador, y después realizar otro '**git merge**' con la rama 'master' actualizada:

```
(master)$ git checkout prog1
Switched to branch 'prog1'

(prog1)$ git merge --no-ff master
Merge made by recursive.
 source/candy1_comb.s |  162 ++++++++++++++++++++++++++++++++++++++++++++++++++
 source/candy1_main.c |    6 +-
 source/candy1_move.s |  126 ++++++++++++++++++++++++++++++++++++
 source/candy1_secu.s |  134 ++++++++++++++++++++++++++++++++++++++
 5 files changed, 426 insertions(+), 4 deletions(-)
 create mode 100644 source/candy1_comb.s
 create mode 100644 source/candy1_move.s
 create mode 100644 source/candy1_secu.s
```

Solo queda realizar un '**git push**' para actualizar el nuevo *commit* de fusión de la rama 'prog1' sobre la rama remota 'remotes/origin/prog1':

```
(prog1)$ git push origin prog1
Counting objects: 1, done.
Writing objects: 100% (1/1), 237 bytes, done.
Total 1 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (1/1), done.
To git.lab.deim:C_grup_Z
   245831b..ad6371c  prog1 -> prog1
```

Al actualizar la visualización del ***gitk*** podremos observar que volvemos a estar en nuestra rama de trabajo, preparados para realizar nuevos cambios a partir de la última versión común (integrada):



A partir de este momento, los nuevos *commit* que realicemos sobre la rama del programador no afectarán a la rama 'master', ya que seguirán su evolución particular. Al final de la fase 2, se tendrá que repetir todo el proceso de fusión para obtener la versión definitiva del proyecto, una vez más sobre la rama 'remotes/origin/master'.

9 Otros conceptos de Git

A continuación se adjuntan otros conceptos que pueden resultar de utilidad cuando trabajamos con el Git. Además, en el documento "**Guía rápida de Git**" que se proporciona en el espacio *Moodle* de la asignatura hay unas tablas resumen para recordar la sintaxis de cada comando.

9.1 *Identificadores de versión*

Además de usar los 7 primeros dígitos SHA-1 como identificador de un *commit*, también se pueden usar las siguientes referencias:

un trozo del mensaje descriptivo	":/texto" (por ejemplo ":/incluye fichero candy1_init.s")
el *commit* actual	HEAD
un *commit* anterior al actual	HEAD^ o HEAD~1
varios *commits* anteriores al actual	HEAD~*n* (por ejemplo, HEAD~3)
el último *commit* de una rama	*<ID rama>* (por ejemplo, *master*)
varios *commits* anteriores a una rama	*<ID rama>~n (por ejemplo, master~2)*

9.2 *El fichero ".gitignore"*

Git únicamente guarda los cambios de ficheros de texto. Si los cambios son ficheros binarios (como imágenes o ejecutables), Git guardará **todo el contenido de los ficheros modificados** en su base de datos. Por esta razón se recomienda disponer de un fichero especial '.gitignore' en la base del directorio que contiene el repositorio local, para especificar los tipos de ficheros que **no** queremos que se registren en la base de datos del Git:

```
(prog1)$ cat .gitignore
*.d
*.o
*.elf
*.nds
*.pnps
*.DS_Store
```

En este caso, forzaremos al Git a que ignore los cambios en los ficheros objeto (*.o), los ficheros ejecutables (*.elf) y el fichero con el contenido de la ROM para la plataforma NDS (*.nds), entre otros.

9.3 *git init --bare <ID repo>*

El comando '**git init**' permite crear un nuevo repositorio en el directorio de trabajo en el que esté posicionado el terminal. La opción '*--bare*' sirve para que este repositorio solo contenga la base de datos de Git, sin el árbol de ficheros y directorios del proyecto. Es decir, podemos crear nuestro propio repositorio remoto, dentro del disco duro de un ordenador al cual tengamos acceso; si el ordenador está conectado a Internet, tendremos nuestro propio servidor Git (si se modifican adecuadamente los permisos de acceso al directorio ".git").

9.4 *git remote -v*

El comando '**git remote**' permite gestionar los servidores remotos contra los que queremos sincronizar nuestro repositorio local. Con la opción '*-v*' se muestra un listado de los servidores actuales:

```
(prog1)$ git remote -v
origin      tunelgit:C_grup_X (fetch)
origin      tunelgit:C_grup_X (push)
```

En este caso solo tenemos un servidor (*origin*), aunque hay dos direcciones diferenciadas, una para el '**git fetch**' y la otra para el '**git push**'.

Esta información está almacenada en la base de datos del Git (directorio ".git", dentro del repositorio local), con lo cual, si movemos el directorio que contiene el repositorio local a otra ubicación del disco, Git continuará sincronizándose correctamente.

9.5 *git add*

El comando '**git add**' se tendrá que utilizar si añadimos un fichero que no estaba registrado en la base de datos del Git. En los ejemplos de este manual, no ha sido necesario invocar este comando porque el repositorio remoto ya incluía todos los ficheros necesarios, pero habrá que tener esto en cuenta en el caso que queramos añadir nuevos ficheros (por ejemplo, al empezar la fase 2 de la práctica).

9.6 *git rm*

Al contrario del comando anterior, '**git rm**' sirve para eliminar un fichero de la base de datos del Git que ya no queramos que forme parte de nuestro proyecto. Hay que entender que invocando un comando '**rm**' (si '**git**' delante) solo estamos borrando los ficheros del árbol de ficheros y directorios, de modo que Git siempre tratará de restituir dichos ficheros desde su base de datos.

9.7 *git merge --abort*

Después de hacer un '**git merge**' o un '**git pull**' que nos informe de que hay conflictos que resolver manualmente, podemos decidir **no** fusionar y continuar nuestro trabajo actual para hacer un *merge* más adelante. Para ello, simplemente hay que ejecutar '**git merge ––abort**' inmediatamente después del *merge* conflictivo.

9.8 *git commit --amend*

Si hemos introducido un error en el último *commit*, ya sea en el contenido de los ficheros o simplemente en el mensaje descriptivo, y todavía no se ha publicado el *commit* erróneo en el servidor, es posible modificar el contenido de los ficheros y realizar un '**git commit ––amend**' para sustituir el contenido del *commit* erróneo.

9.9 *git diff*

Si queremos ver las diferencias entre dos versiones de un mismo fichero o ficheros registrados en dos *commits* distintos de nuestro repositorio local, podemos realizar el comando '**git diff <ID commit old> <ID commit new> -- <ID ficheros>**', donde <ID commit old> y <ID commit new> son los identificadores de las versiones antigua y nueva, respectivamente (ver apartado 9.1 sobre identificadores de *commits*), y <ID ficheros> es el *path* del fichero o ficheros que se quieren comparar (se pueden usar metacarácteres). Por ejemplo:

```
(master)$ git diff HEAD~3 HEAD -- include/*
```

mostraría las diferencias entre todos los ficheros del directorio "include" según estaban registrados en tres *commits* anteriores al actual y com están registrados en el *commit* actual.

10 Referencias adicionales

En este manual se ha tratado de dar una visión lo suficientemente profunda para que el programador pueda manejar Git con la seguridad de que la evolución de su trabajo siempre estará convenientemente archivada.

Sin embargo, este manual no ofrece una visión exhaustiva del entorno Git, puesto que quedan por cubrir otras funcionalidades, como el etiquetado de versiones (*tags*) o la reubicación de la base de una rama ('**git rebase**').

Quizás la funcionalidad más importante de las que no se han tratado en este manual, sea la posibilidad de que cada repositorio de cada ordenador local pueda actuar como repositorio remoto para otros ordenadores locales. Esta característica es la que define al sistema Git como un entorno de control de versiones distribuido.

De todos modos, a continuación se ofrece un listado de referencias de Internet para que el lector interesado pueda profundizar en éstos y otros aspectos de Git:

Referencia	***Descripción***
`http://en.wikipedia.org/wiki/Git_(software)`	Introducción a Git en la Wikipedia.
`http://git-scm.com` `http://git-scm.com/documentation`	Página oficial de Git. Apartado específico sobre documentación.
`http://schacon.github.com/git/user-manual.html`	Manual de usuario oficial de Git: visión general del sistema Git, con una extensión no excesivamente larga pero introduce conceptos bastante complejos.
`http://schacon.github.com/git/gittutorial.html` `http://schacon.github.com/git/gittutorial-2.html`	Tutorial oficial de Git (en dos partes): ofrece una visión más rápida que el manual, pero puede resultar un poco confusa si se toma como punto de partida.
`http://schacon.github.com/git/git.html`	Páginas del Manual de Git: contiene la descripción exhaustiva de todos los comandos de Git, la misma descripción que ofrece el comando 'git <command> --help' pero como páginas html.
`http://gitref.org/`	Otro tutorial de Git: empieza con conceptos más básicos que el tutorial oficial.
`http://byte.kde.org/~zrusin/git/git-cheat-sheet-medium.png`	Una referencia de Git en una sola página.
`http://excess.org/article/2008/07/ogre-git-tutorial/`	Una presentación gravada en vídeo de los conceptos más importantes de Git (dispone de transparencias en PDF): utiliza una notación gráfica muy clara de los *commits* que se proponen como ejemplo.
`http://pcottle.github.io/learnGitBranching/`	Un entorno web interactivo para experimentar con comandos Git sobre *commits* simbólicos, con explicaciones y ejercicios rápidos de realizar.