



Git. Nivel I

mayo, 2019



Objetivos del nivel

- Entender los conceptos implicados en el control de Versiones.
- Aprender a usar los comandos de Git.
- Crear repositorios locales y remotos.
- Crear y usar un repositorio en GitHub

Prerrequisitos del nivel

- Computación para Adultos Nivel I

Acerca de este manual

Este manual pertenece al Centro de Asesoramiento y Desarrollo Informático C.A. (CADI F1). Para obtener más información sobre este u otros cursos visite nuestro sitio Web www.cadif1.com, escribanos a la dirección de correo cadi@cadif1.com o visítenos en nuestra sede ubicada en la Av. Pedro León Torres con calle 59, Centro Comercial Sotavento, piso 2 oficina 27, Barquisimeto estado Lara, Venezuela. Tlf. 0251-7179247, 0251-4410268.

Las marcas mencionadas en este manual son propiedad de sus respectivos dueños. Copyright 2019. Todos los derechos reservados.



Contenido del nivel

Capítulo 1. Introducción a Git

- 1.1.- Control de Versiones.
- 1.2.- Git.
- 1.3.- Git Bash.

Capítulo 2. Repositorios Locales

- 2.1.- Creando un Repositorio.
- 2.2.- Estado Del Repositorio.
- 2.3.- Archivos Ignorados.

Capítulo 3. Agregando al Repositorio

- 3.1.- Agregar Archivos al Repositorio.
- 3.2.- Sacar Archivos Del Repositorio.

Capítulo 4. Confirmaciones. Parte 1

- 4.1.- Áreas Del Repositorio.
- 4.2.- Configuración.
- 4.3.- Confirmando Cambios.

Capítulo 5. Confirmaciones. Parte 2

- 5.1.- Confirmación Resumida.
- 5.2.- Deshacer Cambios.
- 5.3.- Modificar Confirmación.

Capítulo 6. Historial de Confirmaciones

- 6.1.- Diferencias.
- 6.2.- Historial de Confirmaciones.
- 6.3.- Historial Gráfico.

Capítulo 7. Manipulando Archivos

- 7.1.- Eliminar un Archivo Del Repositorio.
- 7.2.- Mover.
- 7.3.- Cambiar el Nombre.

Capítulo 8. Viajando en el Tiempo

- 8.2.- Comando Checkout.
- 8.3.- Comando Reset.

Capítulo 9. Ramas. Parte 1

- 9.1.- Definición.
- 9.2.- Creando Ramas.
- 9.3.- Cambiar de Rama.
- 9.4.- Eliminando Ramas.

Capítulo 10. Ramas. Parte 2

- 10.1.- Unión de Ramas.
- 10.2.- Resolviendo Conflictos.

Capítulo 11. Github. Parte 1

- 11.1.- Plataformas Para el Trabajo en Equipo.
- 11.2.- Usando Github.
- 11.3.- Creando Repositorios Remotos.

Capítulo 12. Repositorios Remotos

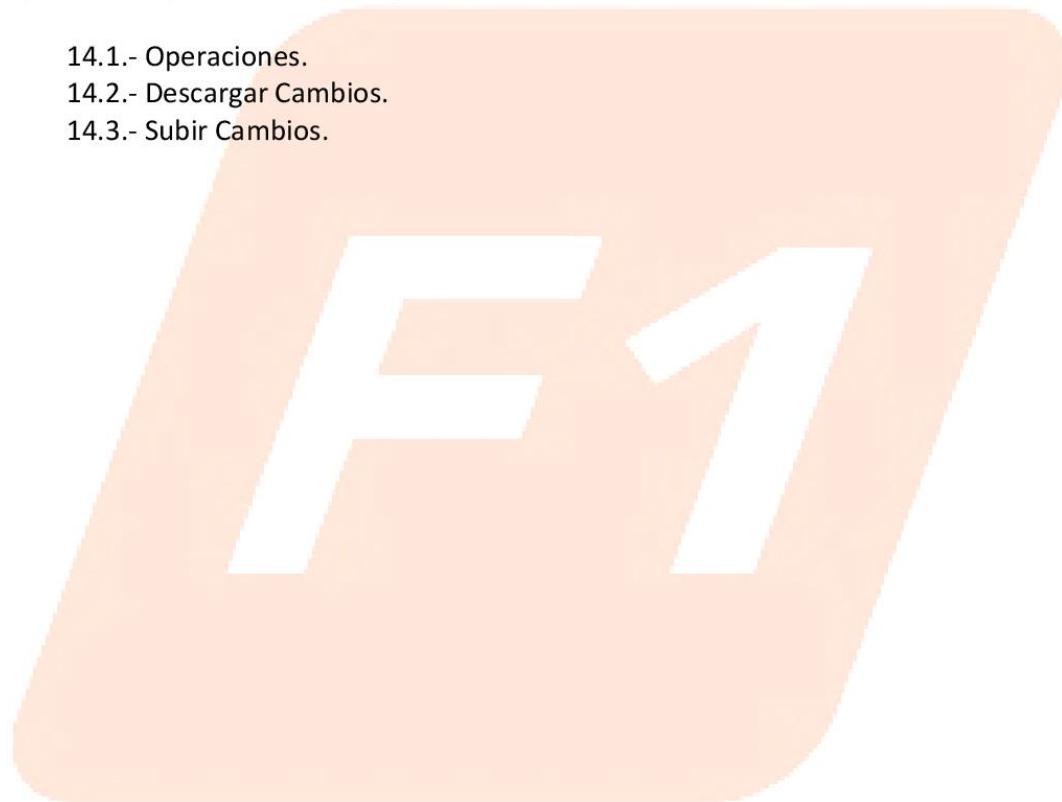
- 12.1.- Repositorio Remoto.
- 12.2.- Configurar Repositorio Remoto.
- 12.3.- Clonando Repositorios.

Capítulo 13. Github. Parte 2

- 13.1.- Contribuyendo en Proyectos de Terceros.
- 13.2.- Pull Request.
- 13.3.- Organizaciones.
- 13.4.- Equipos.

Capítulo 14. Operaciones Con Repositorios Remotos

- 14.1.- Operaciones.
- 14.2.- Descargar Cambios.
- 14.3.- Subir Cambios.

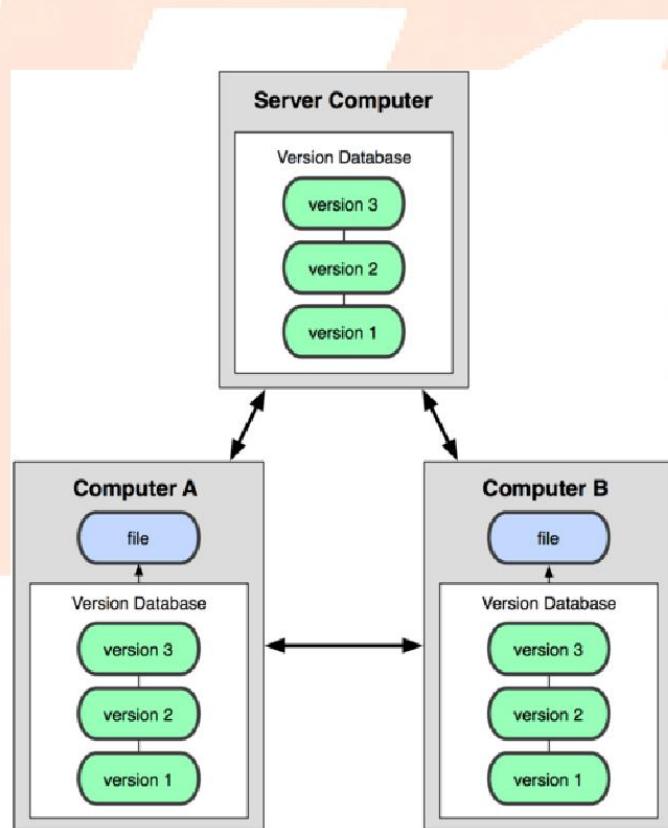


Capítulo 1. INTRODUCCIÓN A GIT

1.1.- Control de Versiones

Se llama control de versiones a la gestión de los diversos cambios que se realizan sobre los elementos de algún producto o una configuración del mismo. Una versión, revisión o edición de un producto, es el estado en el que se encuentra el mismo en un momento dado de su desarrollo o modificación.

En el desarrollo de software, un Sistema de Control de Versiones (SCV o CVS por las siglas Control Versión System) brindará apoyo para gestionar de manera eficiente, cronológica y detallada las diversas versiones que se van generando en el proyecto, con el objetivo de agilizar los tiempos de desarrollo y facilitando el trabajo en equipo.



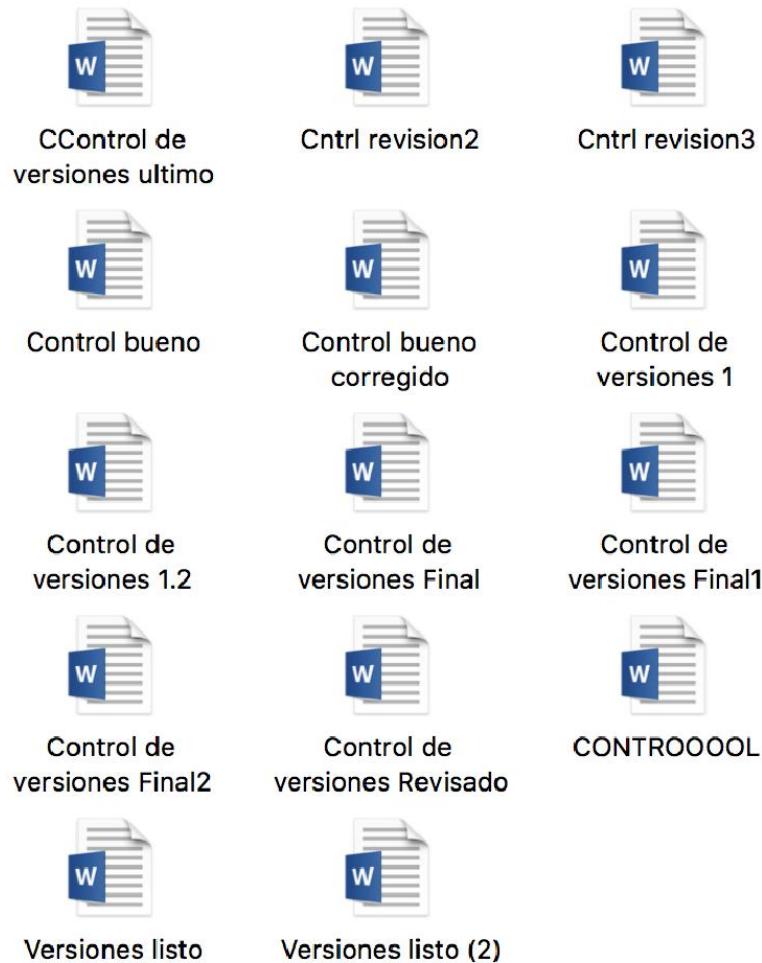
Durante los 70's hasta inicios del 2000, muchos profesionales (principalmente de software) se encontraban con 3 problemas:

- Proyectos difíciles de gestionar y liderar
- Riesgos a sobreseguir con mi código el avance formal del equipo
- La centralización y poca probabilidad de trabajar remotamente

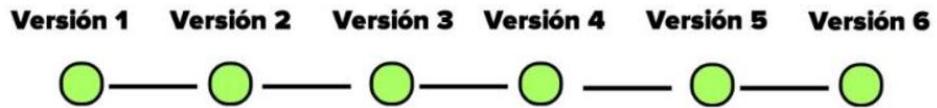
Para resolver estos problemas, se crearon diferentes sistemas de control de versiones:

- SCCS (1972)
- RCS (1982)
- CVS (1986-1990)
- SVN (2000)
- BitKeeper SCM (2000)
- Mercurial (2005)

Es muy frecuente encontrar desarrollos de proyectos en los cuales existen gran cantidad de archivos, donde cada uno es una versión mejorada del anterior lo cual hace prácticamente incierto cuáles son los archivos más actualizados representando un problema muy poco práctico. Los sistemas de control de versiones buscan eliminar el problema reflejado en la siguiente imagen:



Un sistema de control de versiones busca gestionar ágilmente proyectos. Parte de su principal propósito es que se pueda regresar a un estado anterior del proyecto o conocer, incluso, toda su evolución en el tiempo. Desde sus inicios hasta donde se encuentra actualizado. Se pueden ver a los SCV como máquinas del tiempo, que permiten regresar a cualquier momento que se necesite en el proyecto.



1.2.- Git

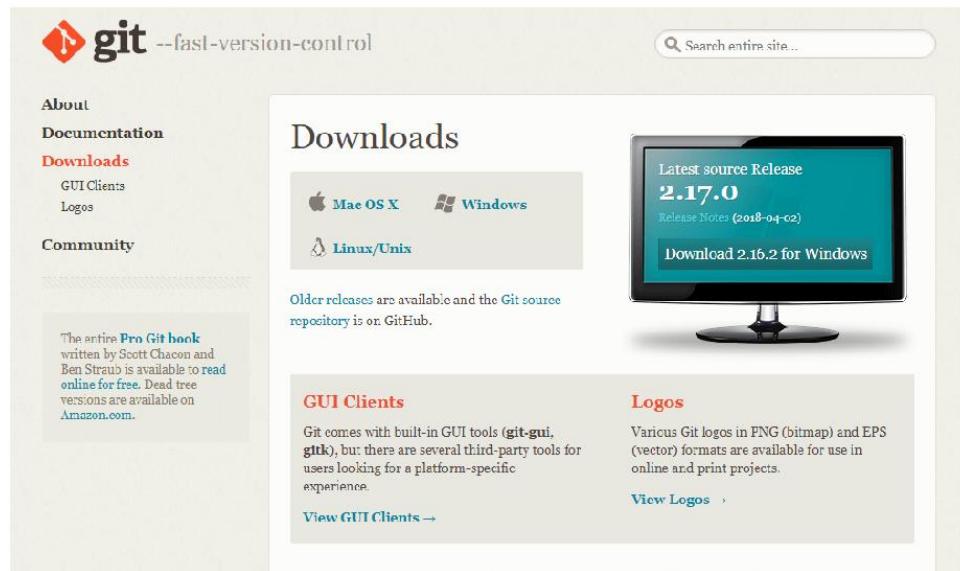
Git es un sistema de control de versiones elaborado por Linus Torvalds de libre distribución y de código abierto diseñado para manejar todo, desde proyectos pequeños a muy grandes con rapidez y eficiencia.

El núcleo de Linux es un proyecto de software de código abierto con un alcance bastante grande. Durante la mayor parte del mantenimiento del núcleo de Linux (1991-2002), los cambios en el software se pasaron en forma de parches y archivos. En 2002, el proyecto del núcleo de Linux empezó a usar un SCV propietario llamado BitKeeper. Luego el equipo de desarrollo de Linux encabezado por Linus Torvalds implementó Git como su propio SCV.



El sitio oficial del proyecto Git es <https://git-scm.com>. Desde la sección de descarga se puede obtener los instaladores para diversos sistemas operativos. El sitio oficial provee

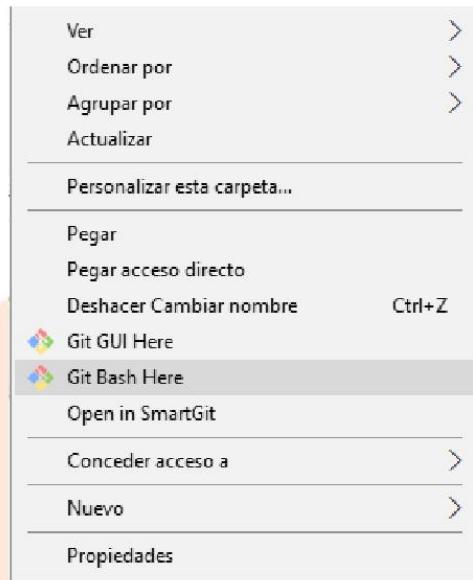
un apartado de documentación para aprender los aspectos principales de la aplicación. En la imagen se muestra la página de descarga.



The screenshot shows the 'Downloads' section of the official Git website. It features links for Mac OS X, Windows, and Linux/Unix. A prominent feature is a monitor displaying the latest source release, version 2.17.0, with a 'Download' button. Below the monitor, there's a note about older releases and a link to the Pro Git book. On the left, there's a sidebar with links for About, Documentation, Downloads (selected), GUI Clients, Logos, and Community. A sidebar also mentions the availability of the 'Pro Git book' on Amazon.com.

En el caso de Windows, Git ofrece un archivo ejecutable que permite realizar la instalación del software. Como típico instalador de Windows, la instalación es muy sencilla, sólo se debe seguir el proceso del asistente de instalación haciendo clic en “siguiente” en todos los pasos del mismo. Para la instalación de Git no se requiere ningún software especializado previamente instalado en el computador.





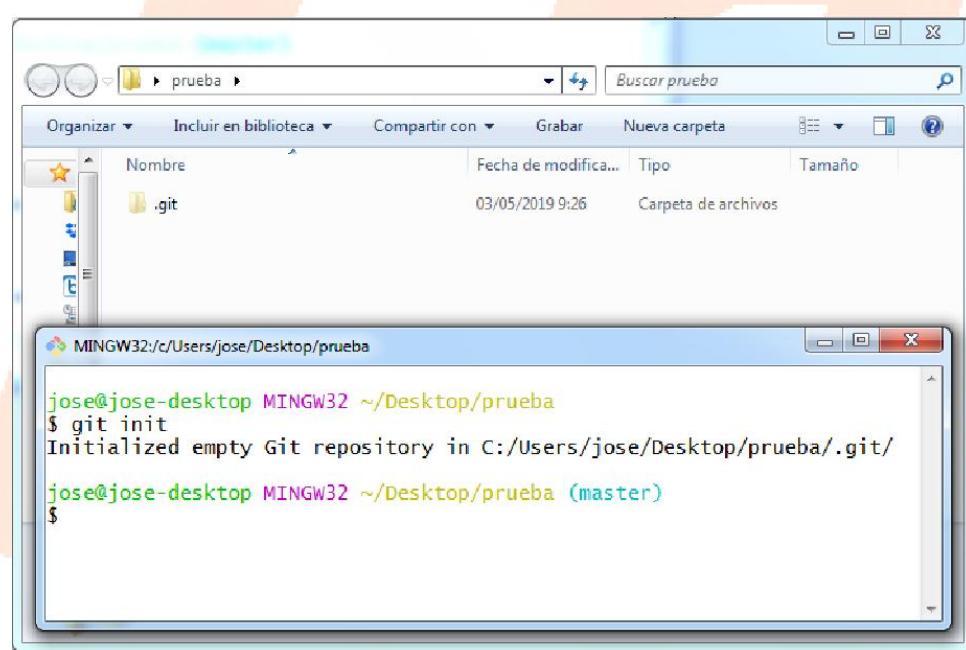
Capítulo 2. REPOSITORIOS LOCALES

2.1.- Creando un Repositorio

Para comenzar un proyecto en Git se debe hacer uso del comando “git init”. Basta con posicionarse en el directorio en el cual se quiere iniciar un repositorio de Git, para luego usar el comando:

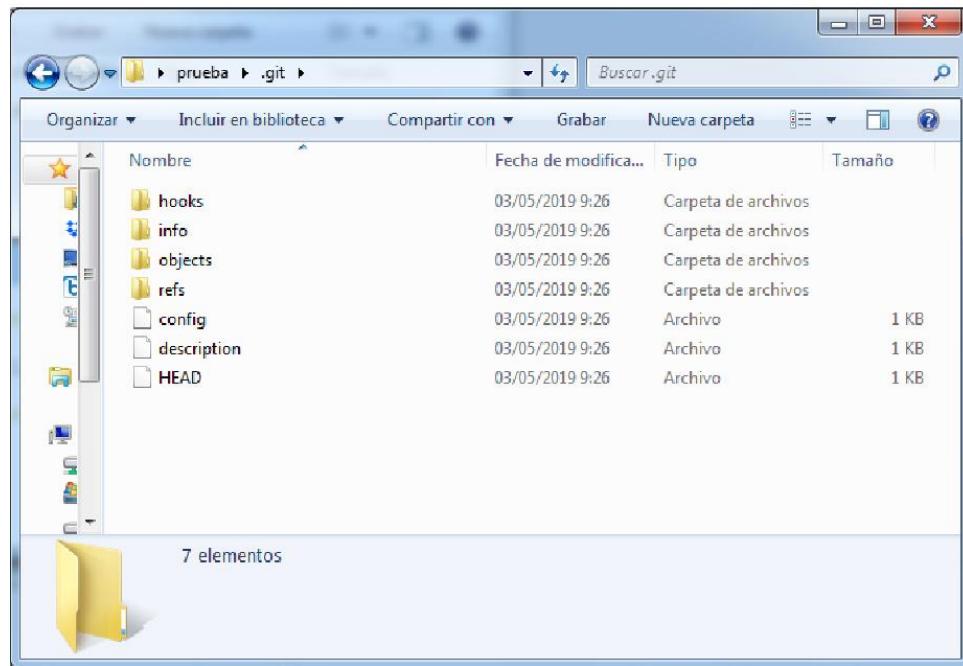
```
git init
```

En la imagen se muestra un ejemplo de la creación de un nuevo repositorio en el directorio "prueba".



En el directorio se crea una carpeta oculta con el nombre .git, en la cual se almacenan archivos y subdirectorios necesarios para llevar el control de versiones del repositorio.

El usuario no debería interactuar directamente con esta carpeta, ya que es de uso específico del programa.



Cualquier carpeta del sistema puede ser un contenedor o repositorio para Git. Un repositorio puede crearse en un directorio vacío o en un directorio que contenga archivos, por lo tanto, no es necesario iniciar el proyecto a la par que el repositorio. De hecho, estos se pueden crear aún cuando ya se ha avanzado con el código.

2.2.- Estado Del Repositorio

Cada archivo del directorio de trabajo puede estar en uno de estos dos estados:

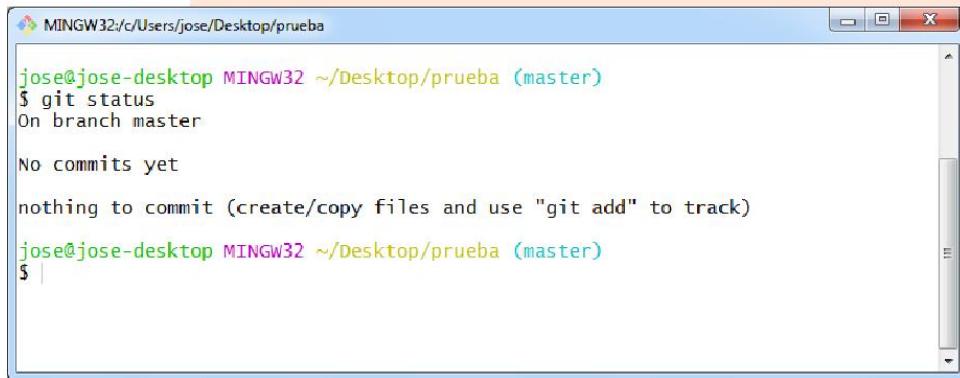
- untracked: sin seguimiento.
- tracked: bajo seguimiento.

Untracked es el estado por defecto que tiene un archivo al ser agregado a la carpeta del repositorio. Si al crear el repositorio ya existen archivos en la carpeta, todos los archivos

estarán untracked, es decir, aún no están siendo controlados por el sistema de control de versiones.

Para conocer el estado de los archivos de un repositorio se ejecuta el comando “git status”. Este comando mostrará los archivos (si existen) que han tenido alguna modificación o que se hayan agregado recientemente.

En el caso de un repositorio vacío (sin archivos) al hacer un git status debe mostrar un mensaje similar al de la siguiente imagen:



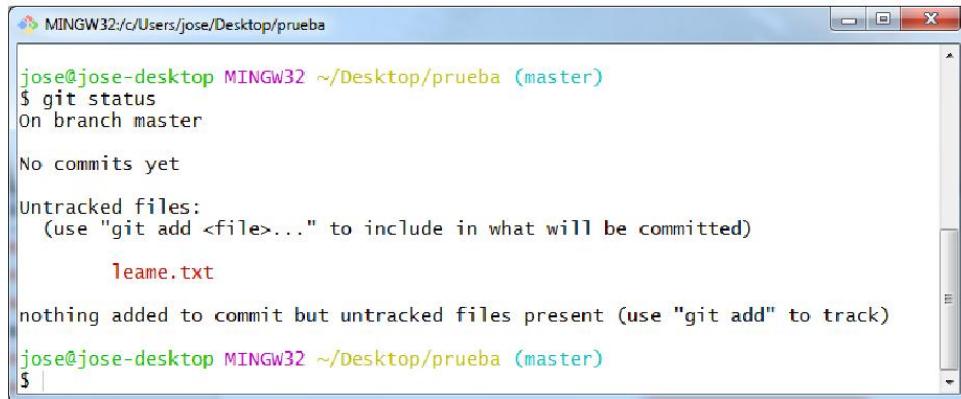
```
MINGW32:/c/Users/jose/Desktop/prueba
jose@jose-desktop MINGW32 ~/Desktop/prueba (master)
$ git status
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)

jose@jose-desktop MINGW32 ~/Desktop/prueba (master)
$
```

En la siguiente imagen se muestra el resultado del comando "git status" luego de haber agregado a la carpeta un archivo de texto con el nombre “leame.txt”. Como se puede ver, el archivo "leame.txt" tiene el estado "untracked":



```
jose@jose-desktop MINGW32 ~/Desktop/prueba (master)
$ git status
On branch master

No commits yet

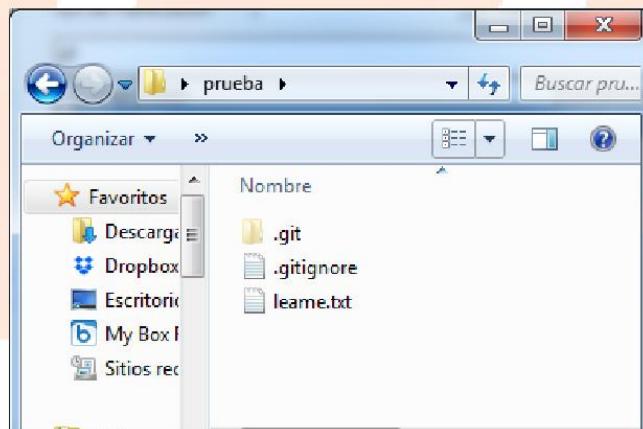
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    leame.txt

nothing added to commit but untracked files present (use "git add" to track)
jose@jose-desktop MINGW32 ~/Desktop/prueba (master)
$
```

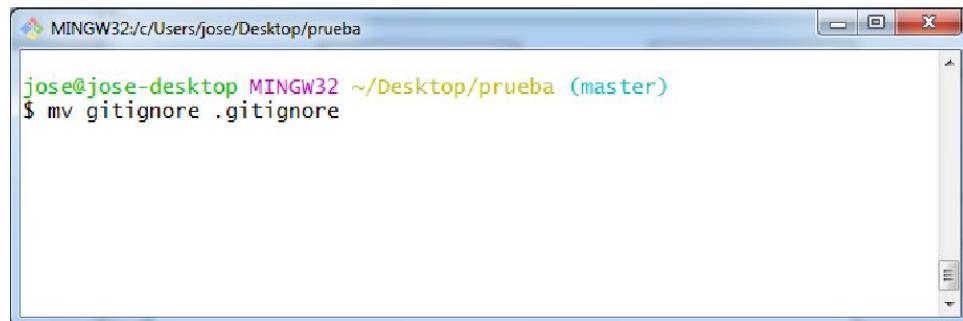
2.3.- Archivos Ignorados

Existen circunstancias donde se necesita tener archivos en el directorio, pero no es necesario que sean tomados en cuenta por Git, es decir, que sean ignorados. Para especificar cuáles archivos deben ser ignorados, Git busca y lee un archivo con el nombre .gitignore en el directorio del repositorio. La siguiente imagen muestra un repositorio con un archivo .gitignore:



El archivo .gitignore es un archivo de texto. Para crearlo en Windows existe un pequeño problema: el explorador de Windows no permite crear un archivo que tenga un nombre comience con un punto ". ". Para crear el archivo, se crea sin el punto, pero luego en la

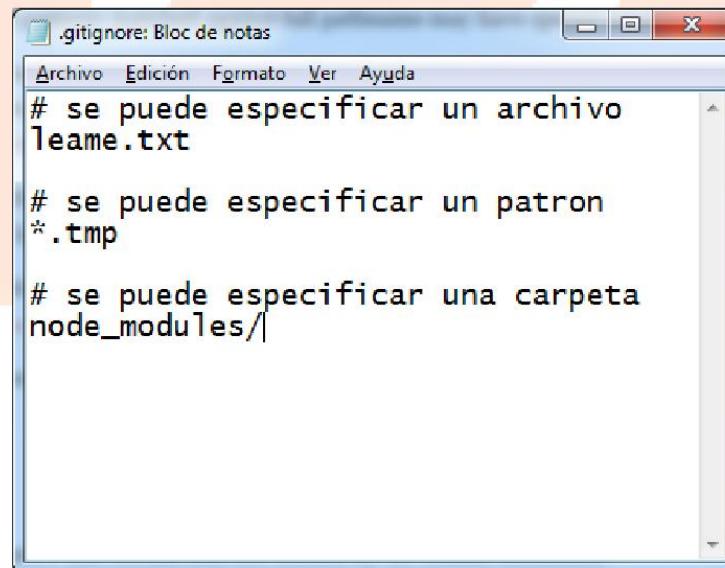
ventana de comandos se le cambia el nombre con el comando "mv", como se ve en la imagen:



```
jose@jose-desktop MINGW32 ~/Desktop/prueba (master)
$ mv gitignore .gitignore
```

El contenido del archivo .gitignore es muy variado. Puede contener:

- nombres de carpetas
- nombres de archivos
- patrones de nombres

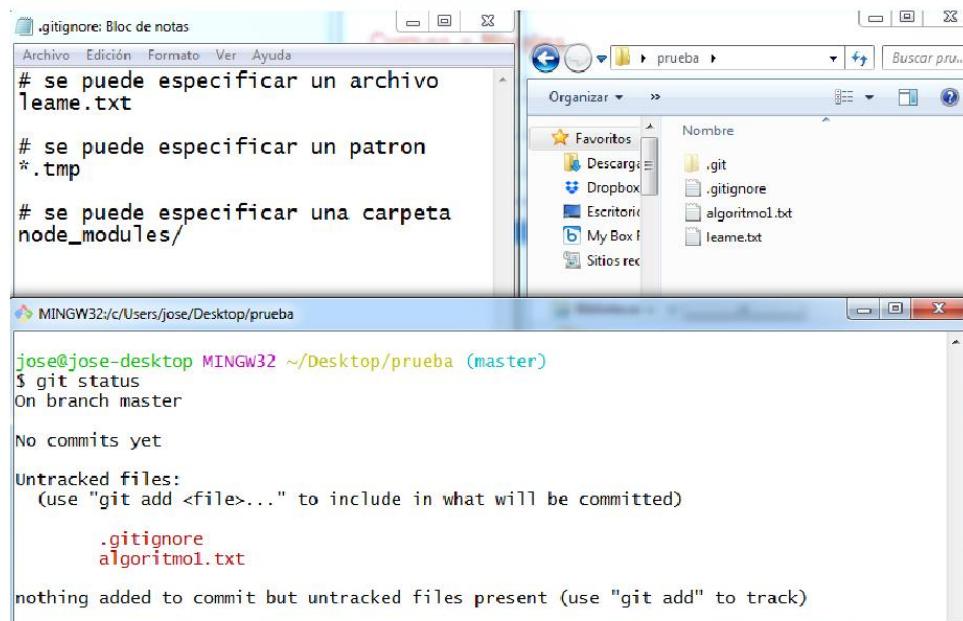


```
# se puede especificar un archivo
teamo.txt

# se puede especificar un patron
*.tmp

# se puede especificar una carpeta
node_modules/
```

Luego de crear y agregar en el archivo .gitignore los archivos que se desea sean ignorados, al ejecutar el comando "git status" se conseguirá que los archivos no aparecen en la lista de archivos "untracked":



Capítulo 3. AGREGANDO AL REPOSITORIO

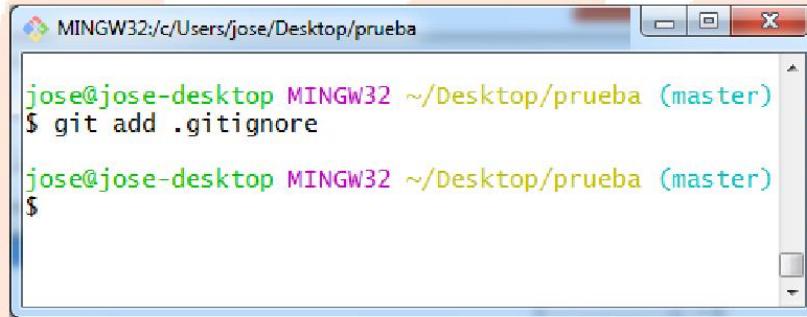
3.1.- Agregar Archivos al Repositorio

Para que un archivo empiece a ser rastreado por Git, es decir, que pase del estado "untracked" a "tracked", debe ser agregado al repositorio. Esto puede ser muy confuso al inicio, porque se puede suponer que el hecho de que el archivo esté en la carpeta implica que ya es parte del repositorio, pero Git no funciona de esa forma.

Un archivo en la carpeta debe ser agregado explícitamente al repositorio Git para que éste pueda empezar a rastrear su comportamiento. El comando para agregar un archivo al repositorio es:

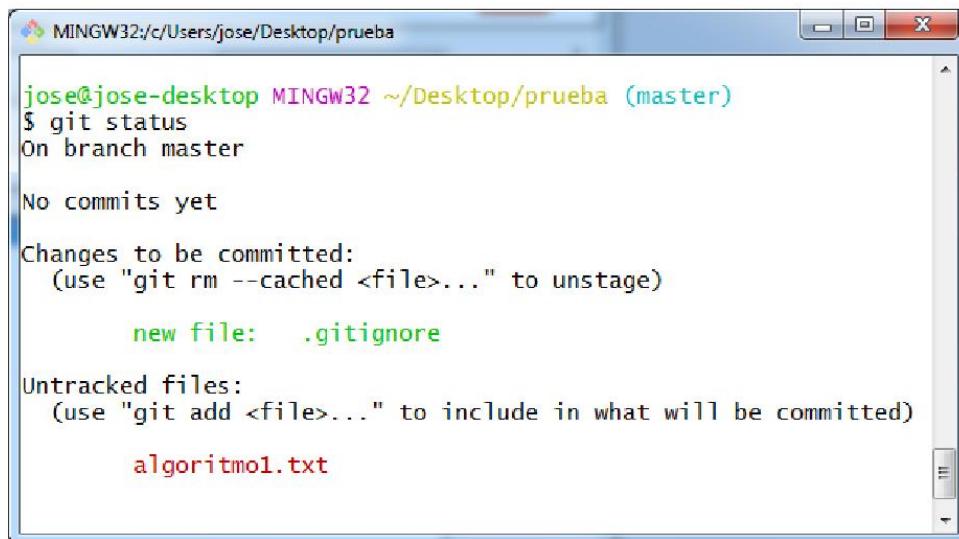
```
git add archivo
```

El siguiente ejemplo muestra como agregar el archivo .gitignore:



A screenshot of a terminal window titled "MINGW32:/c/Users/jose/Desktop/prueba". The window shows the command \$ git add .gitignore being typed by the user. The terminal is running on a Windows operating system, as indicated by the window style and icons.

Luego de ejecutar este comando, al verificar el estatus del repositorio Git se mostrará un mensaje indicando que hay un archivo nuevo. Como se puede visualizar en la siguiente imagen, hay un archivo marcado como "new file" y otro que aún no esta siendo rastreado por Git:



```
jose@jose-desktop MINGW32 ~/Desktop/prueba (master)
$ git status
On branch master

No commits yet

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

    new file:  .gitignore

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

    algoritmo1.txt
```

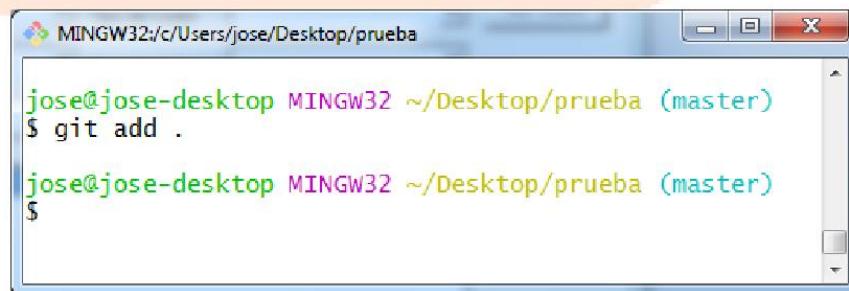
Si es necesario agregar varios archivos al repositorio, se pueden separar el nombre de cada archivo por espacios en blanco, por ejemplo:

```
git add archivo1 archivo2 archivoN
```

Cuando se desea agregar todos los archivos que están en el directorio, se utiliza el comando

```
git add .
```

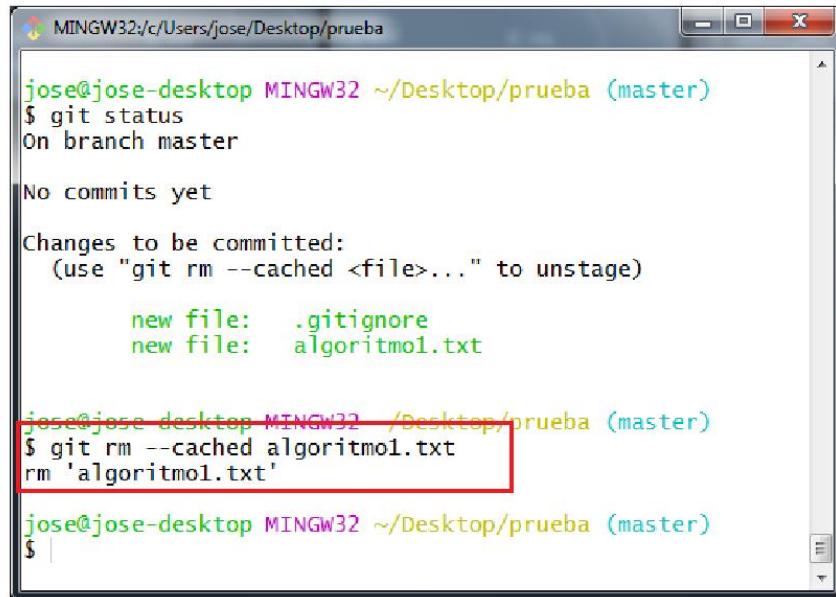
Se debe destacar que el punto "." está separado de la palabra "add". Por ejemplo:



```
jose@jose-desktop MINGW32 ~/Desktop/prueba (master)
$ git add .

jose@jose-desktop MINGW32 ~/Desktop/prueba (master)
$
```





```
jose@jose-desktop MINGW32 ~/Desktop/prueba (master)
$ git status
On branch master

No commits yet

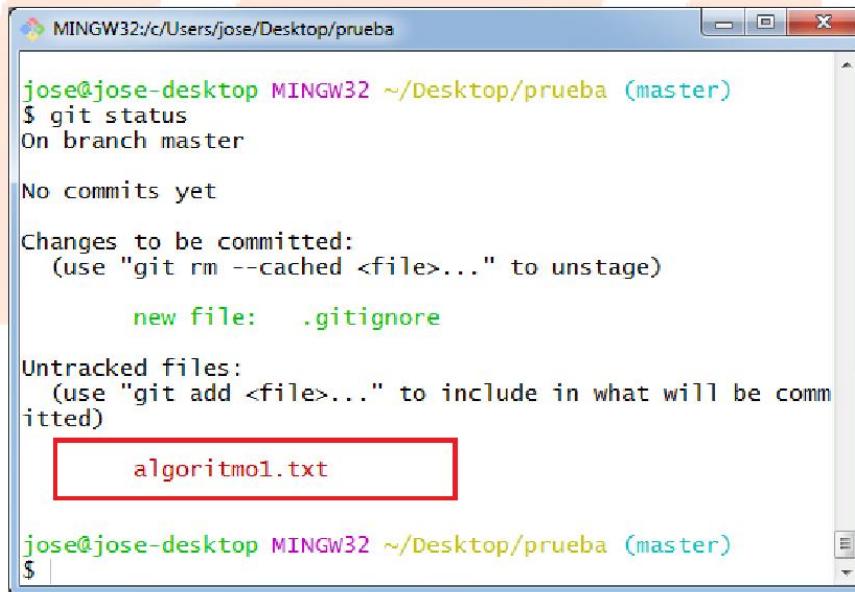
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:  .gitignore
    new file:  algoritmo1.txt

jose@jose desktop MINGW32 ~/Desktop/prueba (master)
$ git rm --cached algoritmo1.txt
rm 'algoritmo1.txt'

jose@jose-desktop MINGW32 ~/Desktop/prueba (master)
$ |
```

Luego de que el archivo ha sido sacado del repositorio, vuelve a aparecer en la lista de archivos "untracked", como se muestra en la siguiente imagen:



```
jose@jose-desktop MINGW32 ~/Desktop/prueba (master)
$ git status
On branch master

No commits yet

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

    new file:  .gitignore

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

    algoritmo1.txt

jose@jose-desktop MINGW32 ~/Desktop/prueba (master)
$ |
```

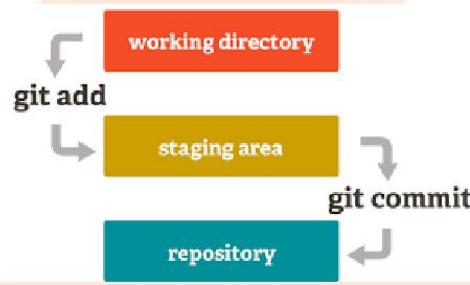
Capítulo 4. CONFIRMACIONES. PARTE 1

4.1.- Áreas Del Repositorio

En un repositorio de Git existen 3 áreas en donde se albergan los archivos. Estas áreas son las siguientes:

- Working Area: es donde se encuentran los archivos que están siendo editados.
- Staging Area: es donde se encuentran archivos que ya han sido modificados y fueron agregados usando el comando git add, por lo tanto, están preparados para la confirmación.
- Local Repository: es donde se encuentran las versiones finales de los archivos que ya han sido confirmados.

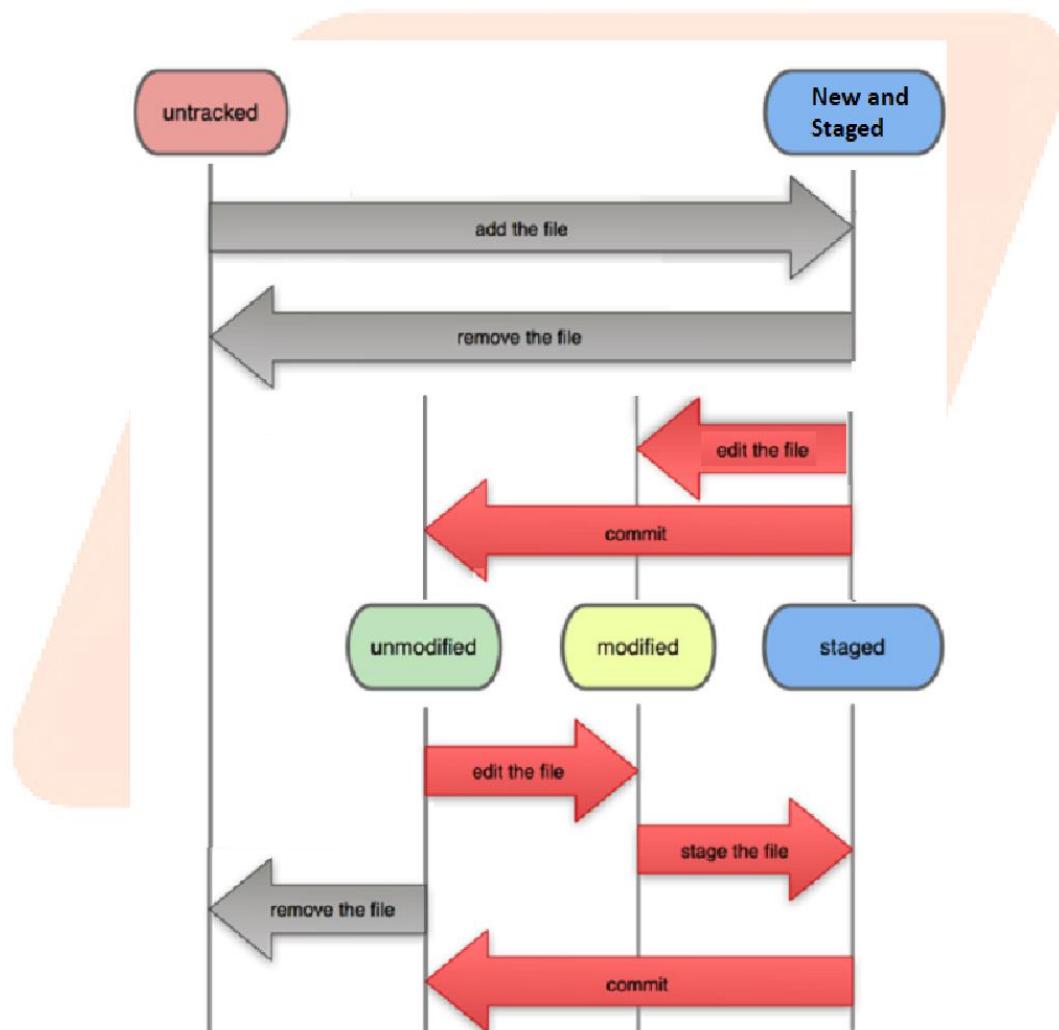
Este ciclo de vida se repite una y otra vez para todo archivo del repositorio que sufra modificaciones. Hasta ahora, se han visto 2 de las 3 áreas



Git estará monitoreando las modificaciones que se realicen sobre los archivos del repositorio, de esta manera en el "Working area" se encuentran los archivos en los que se está trabajando actualmente. Cuando los archivos ya han sido modificados y se quieren registrar los cambios, se deben pasar a "Staging área" para prepararlos para la confirmación de estos cambios.

La confirmación de los cambios ocurre en el momento en que se pasan los archivos al repositorio local, es justo en ese momento en donde termina temporalmente el ciclo de vida de cambios para uno o varios archivos y queda registro de una versión de éstos.

En la siguiente imagen se muestran los estados por los que pasa un archivo almacenado en una carpeta asociada a un repositorio Git, siempre y cuando no esté en la lista de archivos ignorados (.gitignore):



Los estados que pueden tener los archivos son:

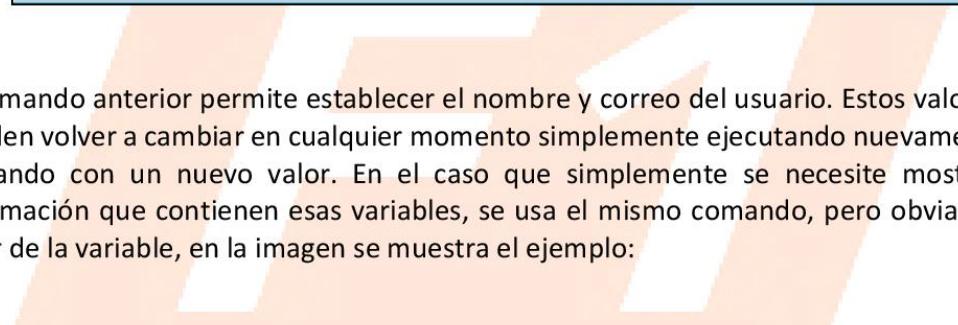
- untracked: es el estado inicial de cualquier archivo o carpeta que está en la carpeta del repositorio o el estado que toma un archivo cuando es removido del repositorio.
- new and staged: el estado que toma el archivo cuando es agregado al repositorio la primera vez con el comando git add.
- unmodified: el estado que toma el archivo cuando es confirmado utilizando el comando git commit.
- modified: es el estado al que pasa un archivo que está en el repositorio y ha sido modificado. Los cambios del archivo están en el "Working Área".
- staged: el estado que toma el archivo cuando es agregado al "Staging area" con el comando git add.

4.2.- Configuración

Antes de hacer una confirmación, lo primero que se debe hacer es establecer un nombre de usuario y dirección de correo electrónico. Esto es importante porque Git usa esta información en las confirmaciones de cambios, que es introducida de manera inmutable en los cambios que se registran. Esta configuración se puede hacer global (para todos los repositorios) o local, para cada repositorio. Si se hace global, se debe hacer una sola vez y todos los repositorios que se creen utilizarán esta configuración. Si se hace local, cada repositorio tendrá su propia configuración y se debe establecer cada vez que se crea un repositorio.

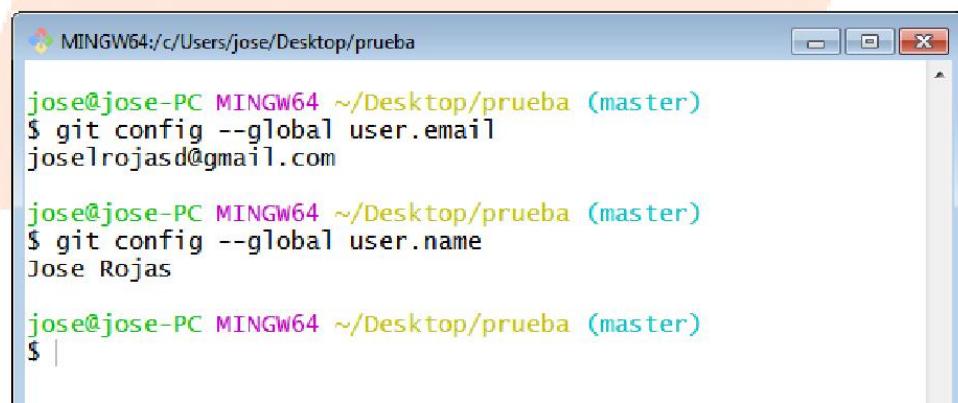
Git trae una herramienta llamada git config que permite obtener y establecer variables de configuración, que controlan el aspecto y funcionamiento de Git. El programa necesita que se coloque información de configuración inicial, como el nombre y el correo electrónico.

El comando de Git utilizado para establecer la configuración inicial es "config". De la siguiente manera se puede configurar el nombre y el email del usuario que está utilizando Git:



```
jose@jose-PC MINGW64 ~/Desktop/prueba (master)
$ git config --global user.name "Jose Rojas"
jose@jose-PC MINGW64 ~/Desktop/prueba (master)
$ git config --global user.email "joselrojasd@gmail.com"
jose@jose-PC MINGW64 ~/Desktop/prueba (master)
$
```

El comando anterior permite establecer el nombre y correo del usuario. Estos valores se pueden volver a cambiar en cualquier momento simplemente ejecutando nuevamente el comando con un nuevo valor. En el caso que simplemente se necesite mostrar la información que contienen esas variables, se usa el mismo comando, pero obviando el valor de la variable, en la imagen se muestra el ejemplo:



```
jose@jose-PC MINGW64 ~/Desktop/prueba (master)
$ git config --global user.email
joselrojasd@gmail.com

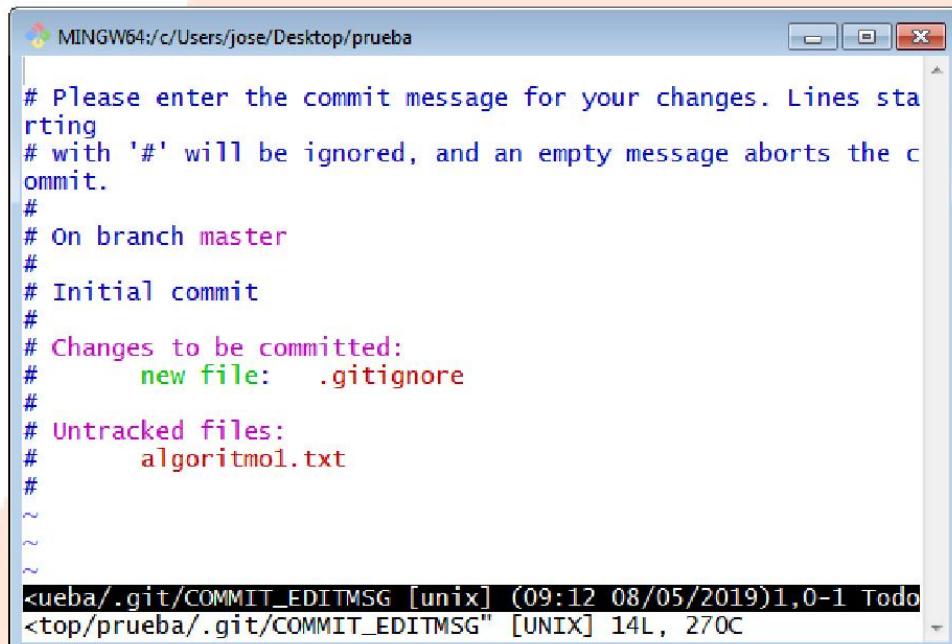
jose@jose-PC MINGW64 ~/Desktop/prueba (master)
$ git config --global user.name
Jose Rojas

jose@jose-PC MINGW64 ~/Desktop/prueba (master)
$ |
```



Una confirmación requiere un mensaje que describa qué incluye la confirmación que se está realizando. Esto permitirá a cualquier persona que revise el historial de cambios del repositorio, con un mensaje lo suficientemente entendible, conocer qué modificaciones se hicieron en cada confirmación.

Al ejecutar el comando "git commit", se abre un editor de texto para que escriba la descripción de la confirmación. El editor aparece con algunas líneas que muestran los archivos que están siendo agregados en la versión. Las líneas que comiencen con # serán ignoradas.



The screenshot shows a terminal window titled "MINGW64:/c/Users/jose/Desktop/prueba". The window contains the following text:

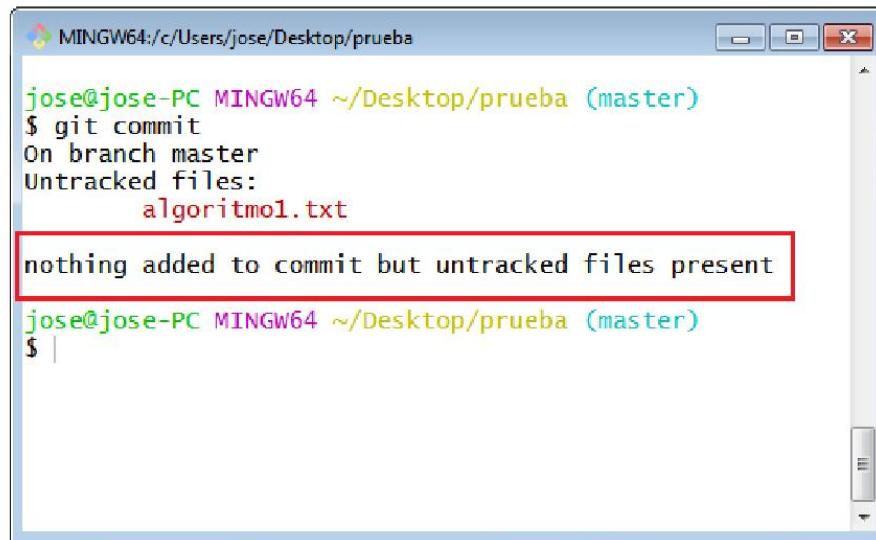
```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the c
ommit.
#
# On branch master
#
# Initial commit
#
# Changes to be committed:
#       new file:  .gitignore
#
# Untracked files:
#       algoritmo1.txt
#
~
```

At the bottom of the terminal window, there is a status bar with the following information:

cueba/.git/COMMIT_EDITMSG [unix] (09:12 08/05/2019)1,0-1 Todo
<top/prueba/.git/COMMIT_EDITMSG" [UNIX] 14L, 270C

Para empezar a escribir en este editor, se debe presionar la tecla "i" (de insert). Cuando se termine de escribir el mensaje, se debe presionar la tecla escape. Luego se debe presionar las teclas ":wq". Si no se desean guardar los cambios, sólo se debe presionar ":q" (se cierra el editor de texto y el commit se cancelará)



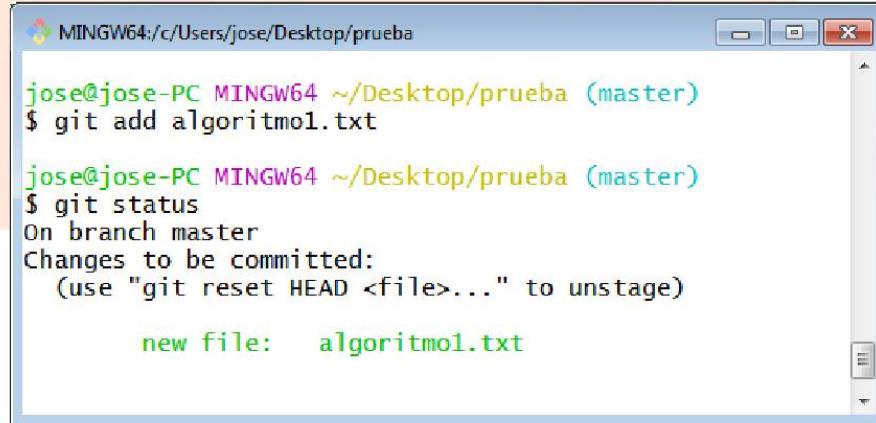


```
jose@jose-PC MINGW64 ~/Desktop/prueba (master)
$ git commit
On branch master
Untracked files:
  algoritmo1.txt

nothing added to commit but untracked files present

jose@jose-PC MINGW64 ~/Desktop/prueba (master)
$ |
```

En el ejemplo anterior, no hay cambios registrados, pero hay un archivo con el nombre "algoritmo.txt" que no está siendo rastreado y puede ser agregado al Staging Area para ser agregado en una siguiente confirmación. Al agregar este archivo, está listo para ser confirmado:



```
jose@jose-PC MINGW64 ~/Desktop/prueba (master)
$ git add algoritmo1.txt

jose@jose-PC MINGW64 ~/Desktop/prueba (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   algoritmo1.txt
```

Capítulo 5. CONFIRMACIONES. PARTE 2

5.1.- Confirmación Resumida

Existe una forma resumida de hacer commit, en la que solo se coloca el mensaje resumen de la confirmación sin necesidad de abrir el editor de texto. La forma de hacerlo es agregando al comando "git commit" el parámetro "-m" y luego entre comillas el mensaje resumen que se desea tenga asociado la confirmación:

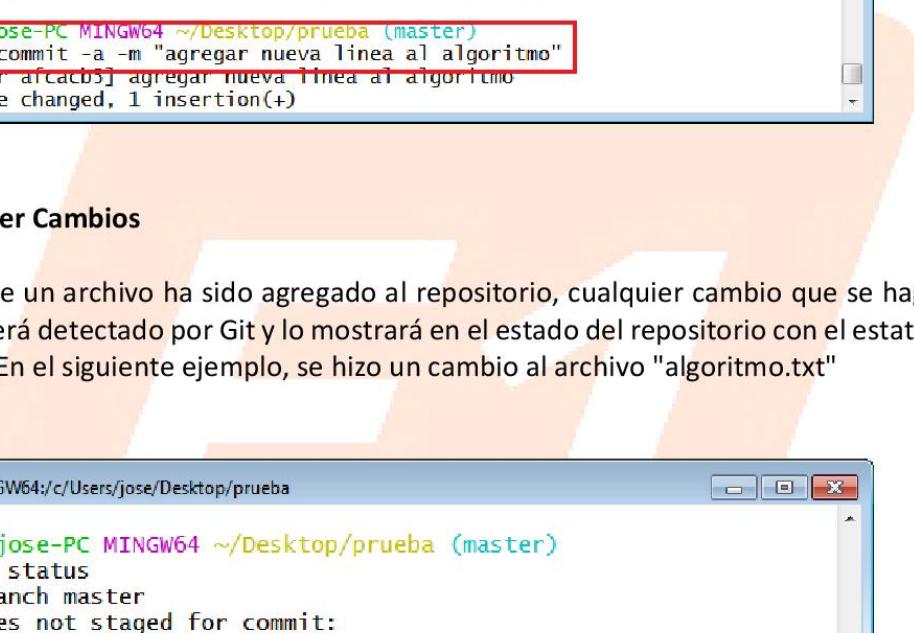


```
jose@jose-PC MINGW64 ~/Desktop/prueba (master)
$ git commit -m "Se agrego el archivo algoritmo.txt"
[master 2e6e2d4] Se agrego el archivo algoritmo.txt
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 algoritmo1.txt

jose@jose-PC MINGW64 ~/Desktop/prueba (master)
$ |
```

Como no se abre el editor de texto, la confirmación se realiza inmediatamente. Una desventaja de hacer confirmaciones de esta forma, es que los mensajes tienden a ser muy cortos, a veces poco detallados y explícitos.

También existe una forma de agregar archivos al Staging Area y hacer la confirmación, todo en un solo paso. La forma de hacerlo es pasando el parámetro "-a" al "git commit". Esto forma de hacer confirmación sólo aplica a archivos que ya hayan estado en el repositorio. Por ejemplo:



```
jose@jose-PC MINGW64 ~/Desktop/prueba (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

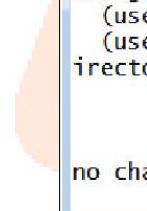
    modified:   algoritmo1.txt

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

jose@jose-PC MINGW64 ~/Desktop/prueba (master)
$ git commit -a -m "agregar nueva linea al algoritmo"
[master afdca05] agregar nueva linea al algoritmo
 1 file changed, 1 insertion(+)
```

5.2.- Deshacer Cambios

Luego de que un archivo ha sido agregado al repositorio, cualquier cambio que se haga sobre éste será detectado por Git y lo mostrará en el estado del repositorio con el estatus "modified". En el siguiente ejemplo, se hizo un cambio al archivo "algoritmo.txt"



```
jose@jose-PC MINGW64 ~/Desktop/prueba (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working d
irectory)

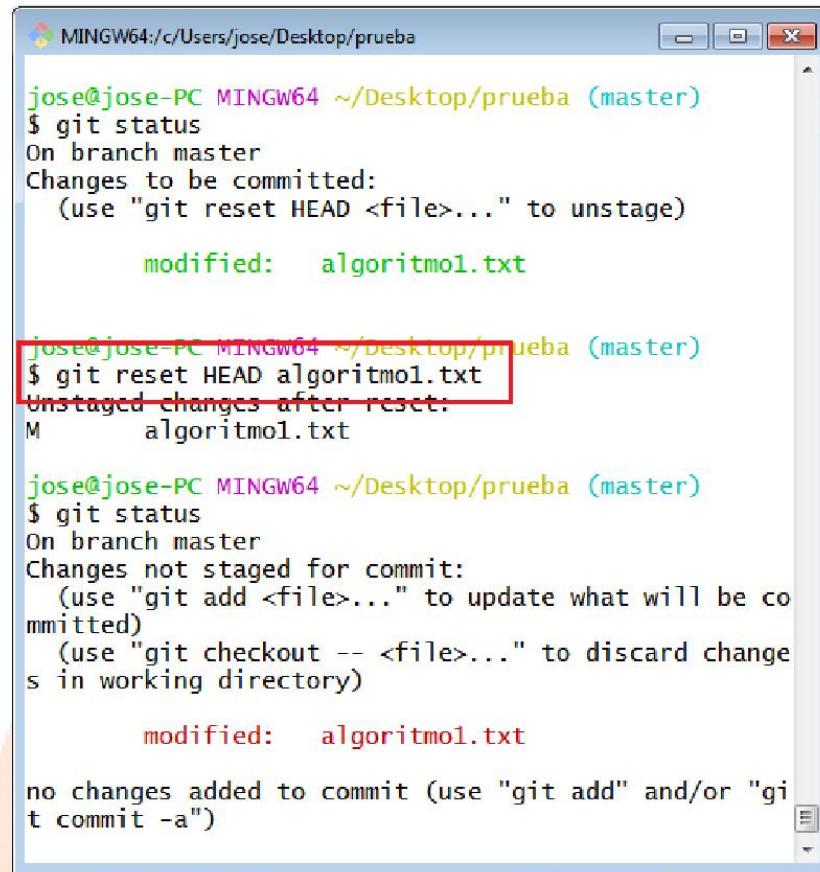
    modified:   algoritmo1.txt

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

Estos cambios pueden ser descartados ejecutando el comando:

```
git checkout -- archivo
```





```
jose@jose-PC MINGW64 ~/Desktop/prueba (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   algoritmo1.txt

jose@jose-PC MINGW64 ~/Desktop/prueba (master)
$ git reset HEAD algoritmo1.txt
Unstaged changes after reset:
M       algoritmo1.txt

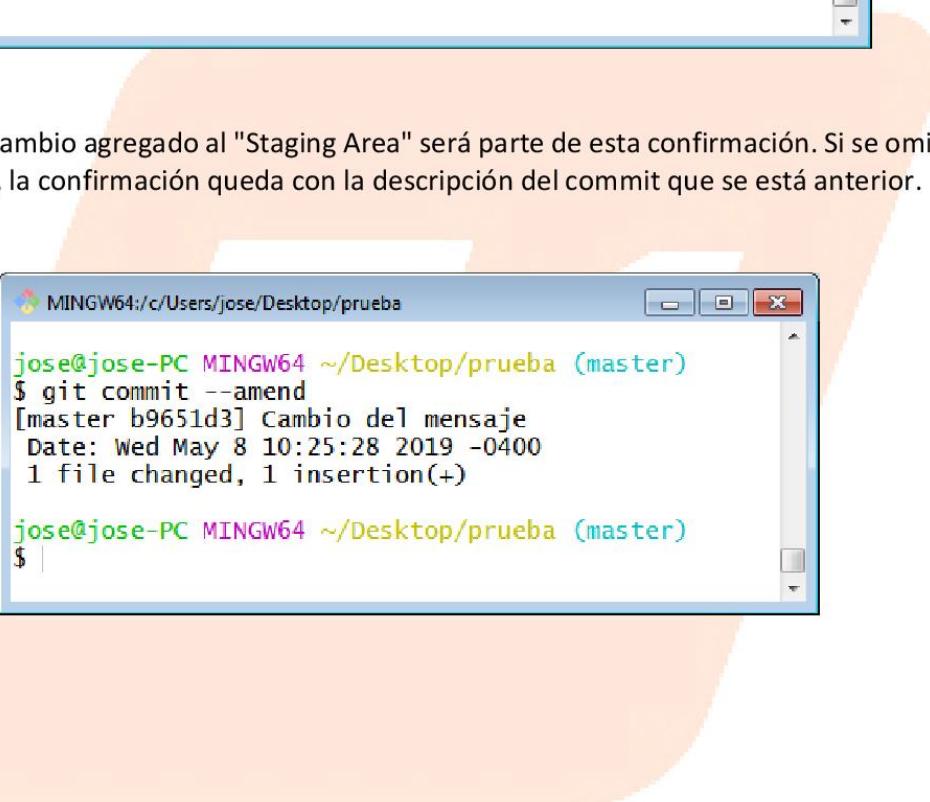
jose@jose-PC MINGW64 ~/Desktop/prueba (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
    (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   algoritmo1.txt

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

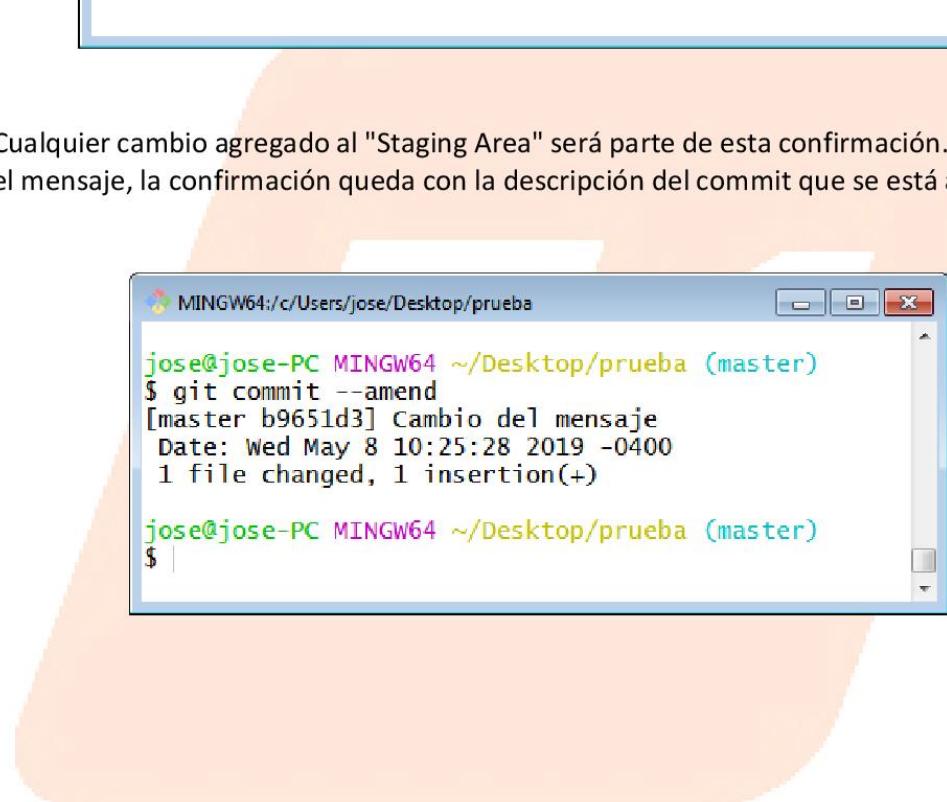
5.3.- Modificar Confirmación

En ocasiones se hace una confirmación habiendo dejado por fuera algún cambio, o se comete algún error en la descripción de la confirmación. Para corregir este problema y evitar crear una confirmación adicional, se utiliza el parámetro "--amend" del comando "commit". La forma de usarlo es la siguiente:



```
MINGW64:/c/Users/jose/Desktop/prueba
jose@jose-PC MINGW64 ~/Desktop/prueba (master)
$ git commit --amend -m "Cambio del mensaje del ultimo"
```

Cualquier cambio agregado al "Staging Area" será parte de esta confirmación. Si se omite el mensaje, la confirmación queda con la descripción del commit que se está anterior.



```
MINGW64:/c/Users/jose/Desktop/prueba
jose@jose-PC MINGW64 ~/Desktop/prueba (master)
$ git commit --amend
[master b9651d3] Cambio del mensaje
Date: Wed May 8 10:25:28 2019 -0400
1 file changed, 1 insertion(+)

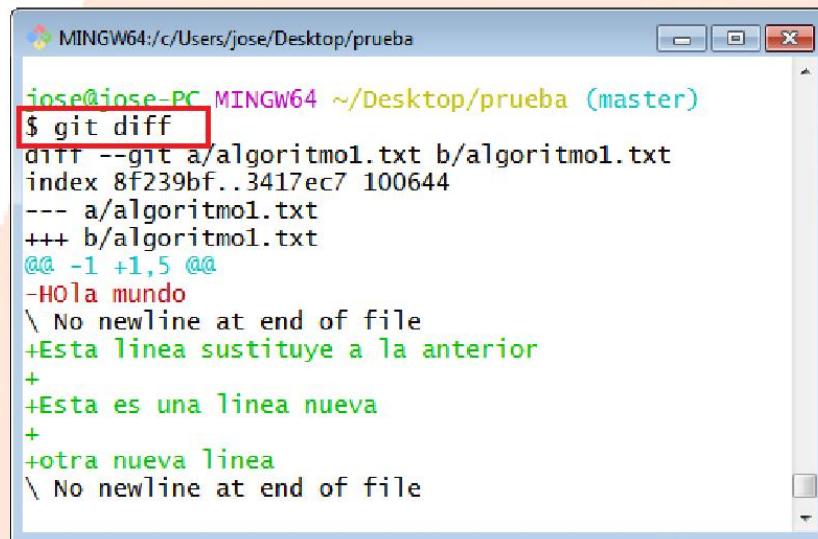
jose@jose-PC MINGW64 ~/Desktop/prueba (master)
$
```

Capítulo 6. HISTORIAL DE CONFIRMACIONES

6.1.- Diferencias

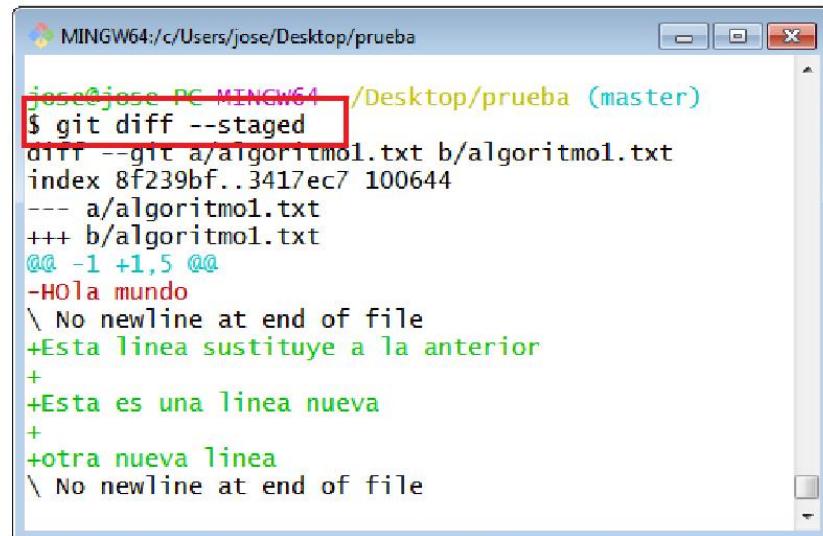
Una de las grandes ventajas de trabajar con un sistema de control de versiones es que se puede visualizar la diferencia que tiene un archivo con su versión anterior. Git muestra los cambios introducidos a un archivo que está en el "Working Area" en comparación con el que está en el repositorio.

Esto se logra con el comando "git diff". En el siguiente ejemplo se muestra como el archivo "algoritmo.txt" tiene líneas nuevas y líneas eliminadas:



```
jose@jose-PC MINGW64 ~/Desktop/prueba (master)
$ git diff
diff --git a/algoritmo1.txt b/algoritmo1.txt
index 8f239bf..3417ec7 100644
--- a/algoritmo1.txt
+++ b/algoritmo1.txt
@@ -1 +1,5 @@
-Hola mundo
\ No newline at end of file
+Esta linea sustituye a la anterior
+
+Esta es una linea nueva
+
+otra nueva linea
\ No newline at end of file
```

Si los archivos ya fueron agregados al "Staging Area" se debe agregar el parámetro "--staged", como se muestra en la siguiente imagen:

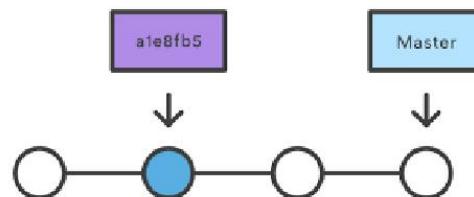


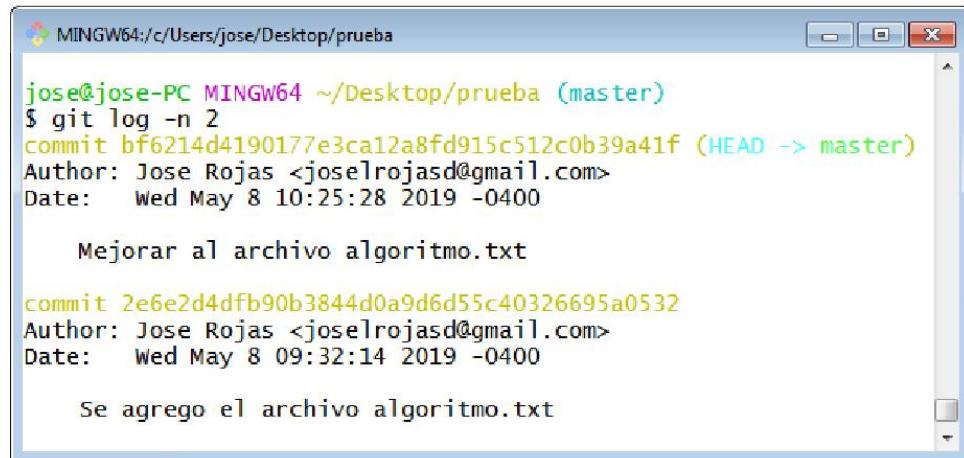
```
jose@jose PC MINGW64 /Desktop/prueba (master)
$ git diff --staged
diff --git a/algoritmo1.txt b/algoritmo1.txt
index 8f239bf..3417ec7 100644
--- a/algoritmo1.txt
+++ b/algoritmo1.txt
@@ -1 +1,5 @@
-Hola mundo
\ No newline at end of file
+Esta linea sustituye a la anterior
+
+Esta es una linea nueva
+
+otra nueva Linea
\ No newline at end of file
```

6.2.- Historial de Confirmaciones

El objetivo de cualquier sistema de control de versiones es registrar los cambios en los archivos que son parte del repositorio. Esto le da la capacidad de revisar el historial de cambios del proyecto para ver quién contribuyó con qué, descubrir dónde se introdujeron ciertos cambios que podrían causar algún error y revertir los cambios problemáticos.

Pero, tener todo este historial disponible es inútil si no se puede visualizar. Ahí es donde entra el comando `git log`. El comando `git log` muestra el historial de todas las confirmaciones realizadas al repositorio, mostrando primera las más recientes y de último la menos reciente, es decir, ordenadas cronológicamente de forma descendente.





```
jose@jose-PC MINGW64 ~/Desktop/prueba (master)
$ git log -n 2
commit bf6214d4190177e3ca12a8fd915c512c0b39a41f (HEAD -> master)
Author: Jose Rojas <joselrojasd@gmail.com>
Date:   Wed May 8 10:25:28 2019 -0400

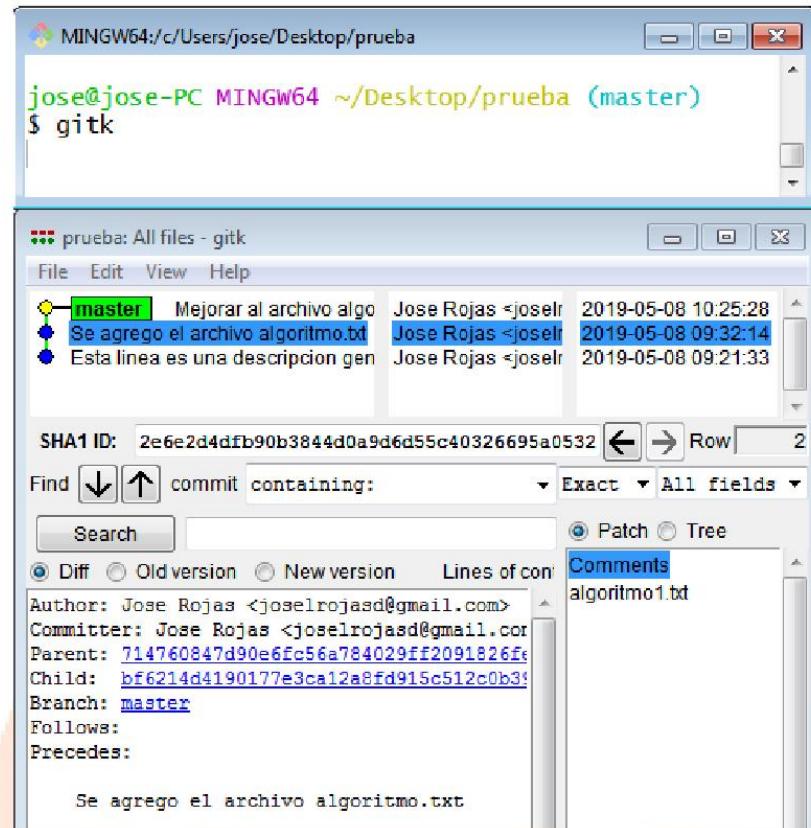
    Mejorar al archivo algoritmo.txt

commit 2e6e2d4dfb90b3844d0a9d6d55c40326695a0532
Author: Jose Rojas <joselrojasd@gmail.com>
Date:   Wed May 8 09:32:14 2019 -0400

    Se agrego el archivo algoritmo.txt
```

6.3.- Historial Gráfico

La instalación básica de Git incluye una herramienta con interfaz gráfica para interactuar con el historial de cambios del repositorio y visualizar las diferencias entre cada confirmación. La herramienta se llama "gitk" y debe ser ejecutada en la línea de comandos, estando posicionado en la carpeta del repositorio. Por ejemplo:



El historial gráfico permite visualizar varios aspectos:

- el historial de cambios, indicando fecha y autor. Cada círculo es una confirmación.
- si hay cambios por guardar.
- los archivos que fueron afectados en cada confirmación.
- la diferencia que existe entre un archivo y su versión anterior (si la tiene).
- las ramas del repositorio (se verá más adelante). Inicialmente, sólo muestra "master".

Cuando hay cambios en el Working Area, gitk los muestra como un punto rojo posterior a la última confirmación.

También permite visualizar qué cambios tiene cada archivo, como se puede notar en la siguiente imagen, en la cual puede verse como al archivo "algoritmo.txt" se le eliminó una línea y se le agregaron 2:

The screenshot shows the gitk application window with the following details:

- Commit History:** A tree view on the left shows a commit on the 'master' branch with the message "agregar nueva linea al algoritmo". A red box highlights this commit with the label "Cambios que estan en el working area".
- Log View:** To the right of the commit history is a log of commits from Jose Rojas, dated from May 8 to May 9, 2019.
- Diff View:** Below the commit history is a diff view for the file 'algoritmo1.txt'. It shows the following changes:
 - A red box highlights a deleted line: "-Hola mundo" with the label "Línea eliminada".
 - A red box highlights two new lines: "+Esta linea sustituye a la anterior" and "+Esta es una linea nueva" with the label "Líneas nuevas".
- Comments:** On the right side, under the 'Comments' section, the text "algoritmo1.txt" is listed.

Cuando se han agregado archivos al Staged Area, se muestra el mismo círculo, pero de color verde, como se muestra en la imagen:

The screenshot shows the gitk graphical interface for a repository named 'prueba'. The main window displays a list of commits on the left and a detailed view of the selected commit on the right.

Commits List:

- Local changes checked in to index but not committed
- master agregar nueva linea al algoritmo
- nuevo archivo
- nuevo archivo
- crear carpeta de imagenes y mover
- Se agrego camaras.bmp
- cambio del gitignore
- Eliminar algoritmo
- Cambio del mensaje
- Se agrego el archivo algoritmo.txt
- Esta linea es una descripcion general del commit

Selected Commit:

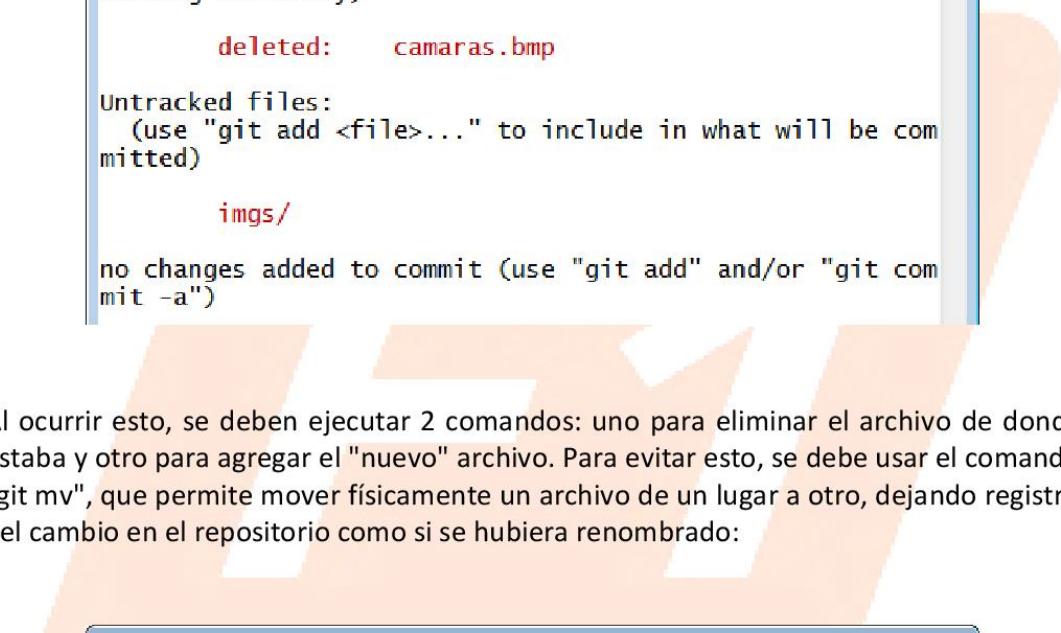
SHA1 ID: 00
Author: Jose Rojas <jose.rojas@correo.ugr.es>
Committer: Jose Rojas <jose.rojas@correo.ugr.es>
Parent: [afcacb5fc2d07f732db1bbec052ee19f8f55cd7f](#) (agreg)
Branch:
Follows:
Precedes:

Patch View:

```
Local changes checked in to index but not committed
----- algoritmo1.txt -----
index 8f239bf..4bc6ab7 100644
@@ -1 +1,3 @@
-Hola mundo
\ No newline at end of file
+Esta linea sustituye a la anterior
+
+Esta es una linea nueva
\ No newline at end of file
```

A red box highlights the commit message "agregar nueva linea al algoritmo" and the patch content. A red arrow points from the commit message to the patch content. Another red box highlights the text "Cambios agregados al Staged Area".





```
jose@jose-PC MINGW64 ~/Desktop/prueba (master)
$ git status
On branch master
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)

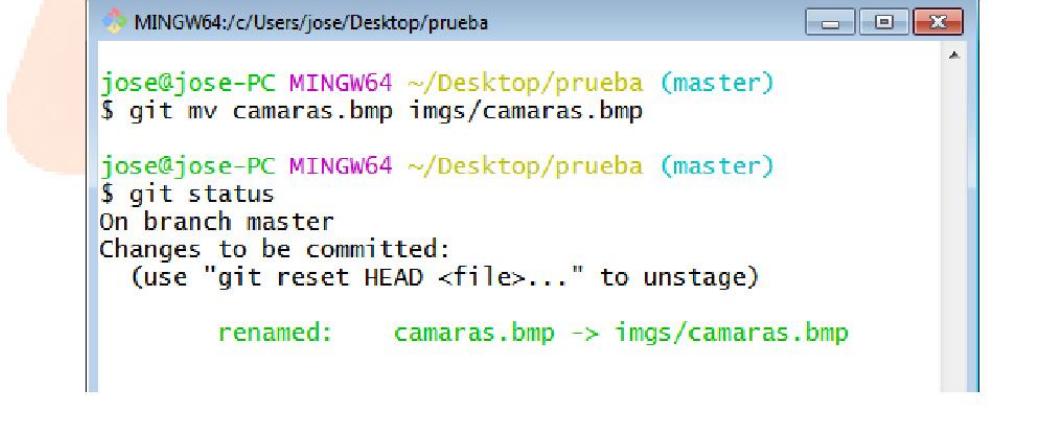
      deleted:   camaras.bmp

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

    imgs/

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

Al ocurrir esto, se deben ejecutar 2 comandos: uno para eliminar el archivo de donde estaba y otro para agregar el "nuevo" archivo. Para evitar esto, se debe usar el comando "git mv", que permite mover físicamente un archivo de un lugar a otro, dejando registro del cambio en el repositorio como si se hubiera renombrado:



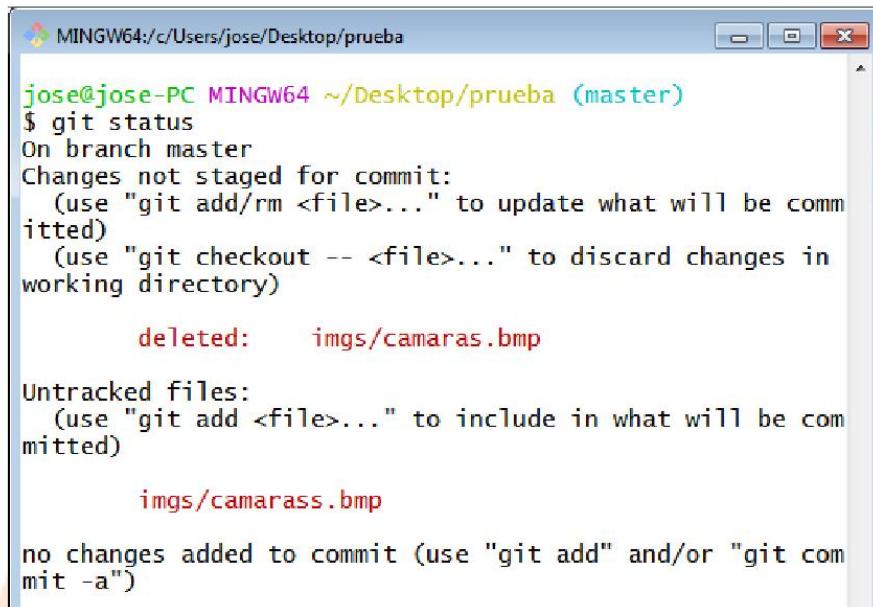
```
jose@jose-PC MINGW64 ~/Desktop/prueba (master)
$ git mv camaras.bmp imgs/camaras.bmp

jose@jose-PC MINGW64 ~/Desktop/prueba (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:   camaras.bmp -> imgs/camaras.bmp
```

7.3.- Cambiar el Nombre

Para cambiar el nombre de un archivo se utiliza el mismo comando "git mv". Si el cambio del nombre se hace por fuera de Git, el detectará como si un archivo fue eliminado y se ha creado uno nuevo.



```
jose@jose-PC MINGW64 ~/Desktop/prueba (master)
$ git status
On branch master
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)

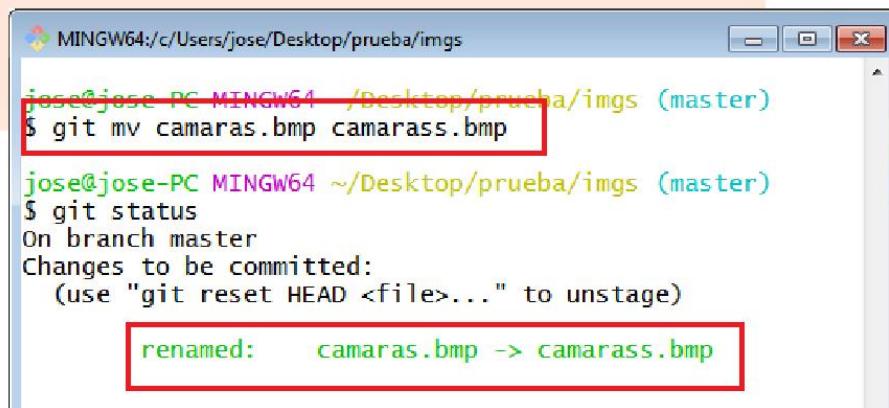
      deleted:    imgs/camaras.bmp

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

    imgs/camarass.bmp

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

Para evitar este comportamiento, es más conveniente utilizar el comando "git mv", como se muestra en la siguiente imagen:



```
jose@jose-PC MINGW64 ~/Desktop/prueba/imgs (master)
$ git mv camaras.bmp camarass.bmp
```

The command above was run and its output is shown below:

```
jose@jose-PC MINGW64 ~/Desktop/prueba/imgs (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    camaras.bmp -> camarass.bmp
```

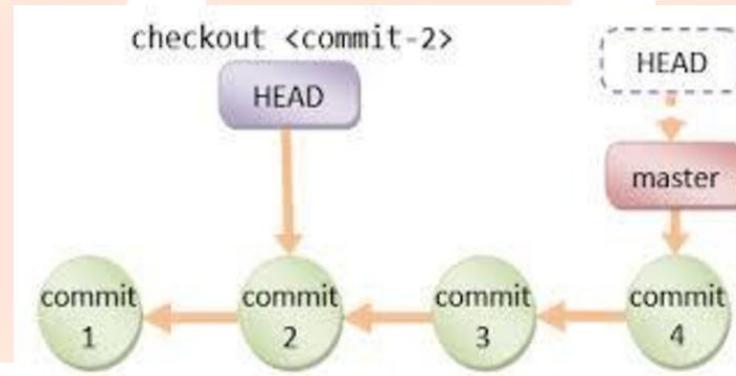
Capítulo 8. VIAJANDO EN EL TIEMPO

8.1.- Checkout

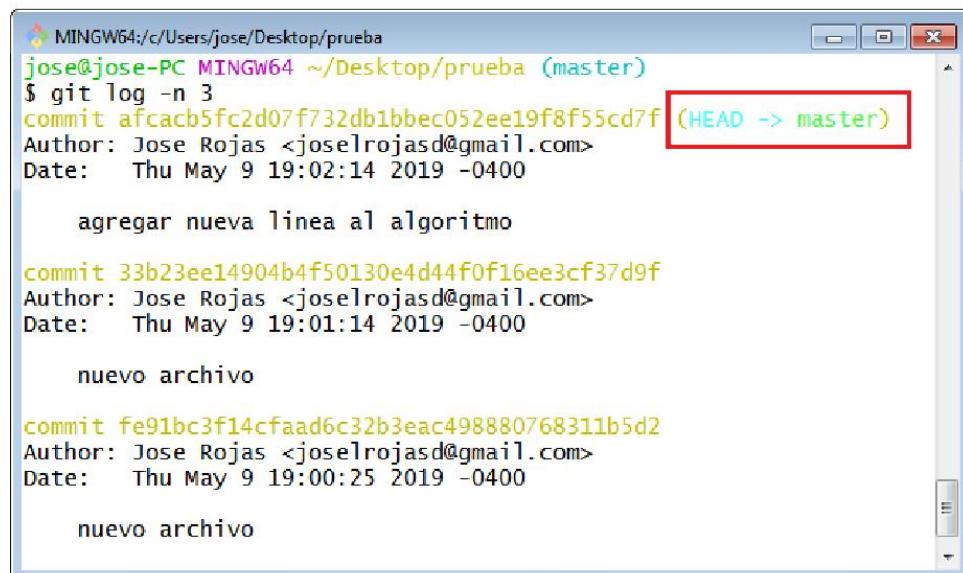
Git provee al usuario la capacidad de literalmente “viajar en el tiempo”. No basta con conocer el historial de todos los cambios que se han realizado en el proyecto, sino que también permite retroceder en el tiempo según sea necesario para beneficio del proyecto.

Para lograr este objetivo es necesario entender:

- cada confirmación tiene un identificador único.
- cada confirmación es parte de un árbol.
- todas las confirmaciones están enlazadas entre sí, de forma tal que se puede navegar de una a otra.
- hay un elemento que permite apuntar la confirmación actual. A este elemento se le denomina "HEAD".



Por defecto, el HEAD se encuentra ubicado en la última confirmación, como se muestra en la siguiente imagen:



```
MINGW64:/c/Users/jose/Desktop/prueba
jose@jose-PC MINGW64 ~/Desktop/prueba (master)
$ git log -n 3
commit afcacb5fc2d07f732db1bbec052ee19f8f55cd7f (HEAD -> master)
Author: Jose Rojas <joselrojasd@gmail.com>
Date:   Thu May 9 19:02:14 2019 -0400

    agregar nueva linea al algoritmo

commit 33b23ee14904b4f50130e4d44f0f16ee3cf37d9f
Author: Jose Rojas <joselrojasd@gmail.com>
Date:   Thu May 9 19:01:14 2019 -0400

    nuevo archivo

commit fe91bc3f14cfaad6c32b3eac498880768311b5d2
Author: Jose Rojas <joselrojasd@gmail.com>
Date:   Thu May 9 19:00:25 2019 -0400

    nuevo archivo
```

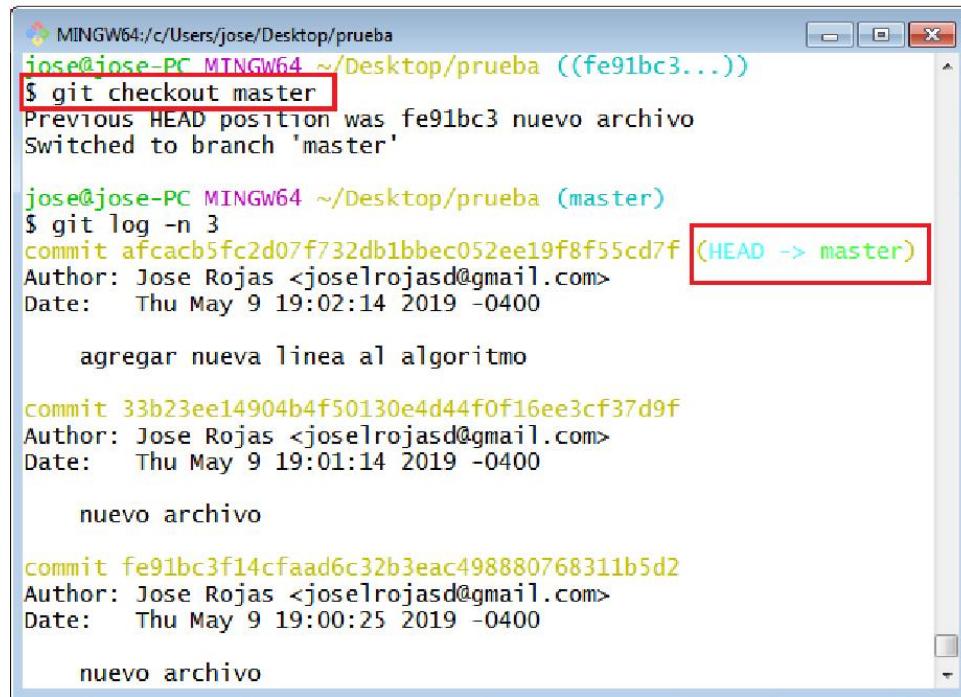
En algún momento se puede hacer que el HEAD se ubique en otra confirmación, de forma tal que se puede visualizar el estado del repositorio en esa confirmación. Esto se logra con el comando "git checkout". La forma general de usarlo es la siguiente:

```
git checkout <codigoCommit>
```

Donde el parámetro "códigoCommit" indica el código del commit al que se desea viajar. Para determinar a cuál confirmación se desea mover, debe averiguar primero el código de la confirmación, utilizando el comando "git log".

En el siguiente ejemplo, se resalta el código de una confirmación a la cual se desea viajar:





The screenshot shows a terminal window titled 'MINGW64:/c/Users/jose/Desktop/prueba'. It displays the output of a 'git log -n 3' command. The first commit is highlighted with a red box around the command line '\$ git checkout master' and the commit message '(HEAD -> master)'. The commit message itself is also highlighted with a red box. The commit details are as follows:

```
jose@jose-PC MINGW64 ~/Desktop/prueba ((fe91bc3...))
$ git checkout master
Previous HEAD position was fe91bc3 nuevo archivo
Switched to branch 'master'

jose@jose-PC MINGW64 ~/Desktop/prueba (master)
$ git log -n 3
commit afcacb5fc2d07f732db1bbec052ee19f8f55cd7f (HEAD -> master)
Author: Jose Rojas <joselrojasd@gmail.com>
Date:   Thu May 9 19:02:14 2019 -0400

    agregar nueva linea al algoritmo

commit 33b23ee14904b4f50130e4d44f0f16ee3cf37d9f
Author: Jose Rojas <joselrojasd@gmail.com>
Date:   Thu May 9 19:01:14 2019 -0400

    nuevo archivo

commit fe91bc3f14cfaad6c32b3eac498880768311b5d2
Author: Jose Rojas <joselrojasd@gmail.com>
Date:   Thu May 9 19:00:25 2019 -0400

    nuevo archivo
```

8.2.- Reset

Los viajes en el tiempo que se han hecho hasta ahora no han realizado cambios ni eliminado ninguna de las confirmaciones anteriores. ¿Qué tal que lo que se desea es retroceder en el tiempo pero a su vez eliminar todas las confirmaciones hasta el punto hasta el cual retrocede?. Esto con el objetivo de no dejar rastros de esas confirmaciones y que no aparezcan en el historial.

Git provee un comando que permite hacer lo anteriormente descrito, es decir, retroceder a algún commit pero con eliminando los commit anteriores. El comando es el siguiente:

```
git reset
```

El comando reset puede ser usado en alguno de estos modos:

```
--soft
```

No toca el archivo de índice o el árbol de trabajo en absoluto (pero restablece el encabezado a <commit>, al igual que todos los modos). Esto deja todos los archivos modificados "Cambios que se comprometerán", como lo indicaría el estado de git.

--mixed

Restablece el índice pero no el árbol de trabajo (es decir, los archivos modificados se conservan pero no se marcan para la confirmación) e informa lo que no se ha actualizado. Esta es la acción por defecto.

--hard

Restablece el índice y el árbol de trabajo. Cualquier cambio en los archivos rastreados en el árbol de trabajo desde <commit> se descarta

En la imagen se muestra el uso del comando reset:

```
vilfer@DESKTOP-DMOLHOS MINGW32 ~/Desktop/cadif1 (master)
$ git reset --soft fc122de2eb57904836960ea4abc8bc06b7cca70e

vilfer@DESKTOP-DMOLHOS MINGW32 ~/Desktop/cadif1 (master)
$ git log
commit fc122de2eb57904836960ea4abc8bc06b7cca70e (HEAD -> master)
Author: vilfer Alvarez <vilferalvarez@gmail.com>
Date:   Sun Apr 8 23:44:24 2018 -0400

    Agregando una linea al archivo REAME.txt

commit 9d617358d9d615a94361ed0c8306fa41dbbf4b23
Author: Vilfer Alvarez <vilferalvarez@gmail.com>
Date:   Sun Apr 8 23:40:29 2018 -0400

    Mi primer commit
```



Es común encontrar nombres de ramas con nombres de participantes del proyecto o cualquier otro identificador no muy específico. Las buenas prácticas indican que se deben tener por lo menos la rama central del proyecto (master), alguna rama de desarrollo (develop) en la cual se hagan las pruebas necesarias para los nuevos cambios que deban integrarse al proyecto y finalmente ramas que respondan a nombres de características que se estén desarrollando en el momento. Por ejemplo, la rama "Validación de inicio de sesión" es una rama en la cual se está desarrollando el inicio de sesión de usuario y que luego será integrada a la rama Develop.

Entre los objetivos de usar ramas está:

- Agilizar el trabajo en equipo
- Separar el ambiente de desarrollo del de producción
- Llevar un mejor control y seguimiento de la realización de tareas dentro del equipo

Esto significa que se pueden hacer cosas como:

- Conmutación de contexto sin fricción. Cree una rama para probar una idea, comience varias veces, cambie de nuevo a donde se ramificó, aplique un parche, cambie de nuevo a donde está experimentando y fíjelo.
- Reglas Basadas en el Rol. Tener una rama que siempre contiene sólo lo que va a la producción, otro que se fusionan en el trabajo para la prueba, y varios más pequeños para el trabajo diario.
- Flujo de trabajo basado en funciones. Cree nuevas ramas para cada nueva función en la que esté trabajando, de modo que pueda pasar sin problemas entre ellas y, a continuación, eliminar cada una de las ramas cuando se fusione con la línea principal.
- Experimentación desecharable. Crear una rama para experimentar, darse cuenta de que no va a funcionar, y sólo eliminarlo - abandonar el trabajo - con nadie más lo vea (incluso si has empujado otras ramas en el ínterin).

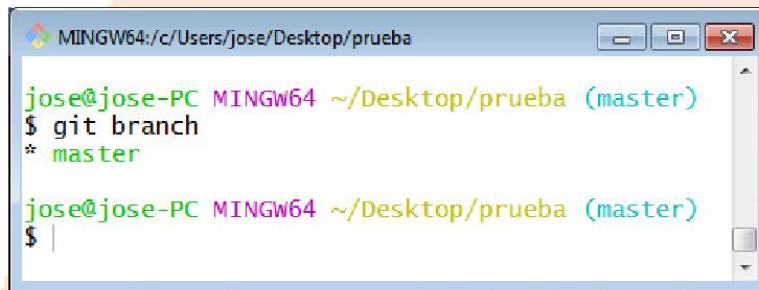
9.2.- Creando Ramas

Cuando se inicializa un repositorio, siempre se crea una rama principal por defecto llamada

"master". Esta rama es la principal del repositorio y es a partir de esta que se podrán crear nuevas ramas cuando sea necesario. Se pueden observar las ramas presentes en el repositorio con el siguiente comando:

```
git branch
```

En la imagen se muestra el resultado de este comando en el repositorio. El símbolo * identifica la rama en la que se encuentra el repositorio actualmente.

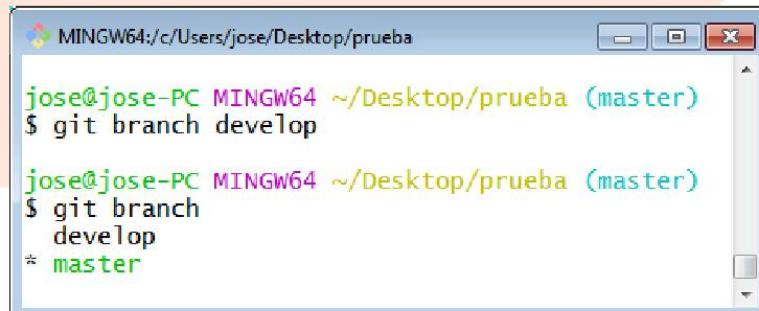


```
jose@jose-PC MINGW64 ~/Desktop/prueba (master)
$ git branch
* master

jose@jose-PC MINGW64 ~/Desktop/prueba (master)
$ |
```

Para crear una nueva rama, se debe usar el comando anterior agregándole el nombre de la nueva rama como sigue:

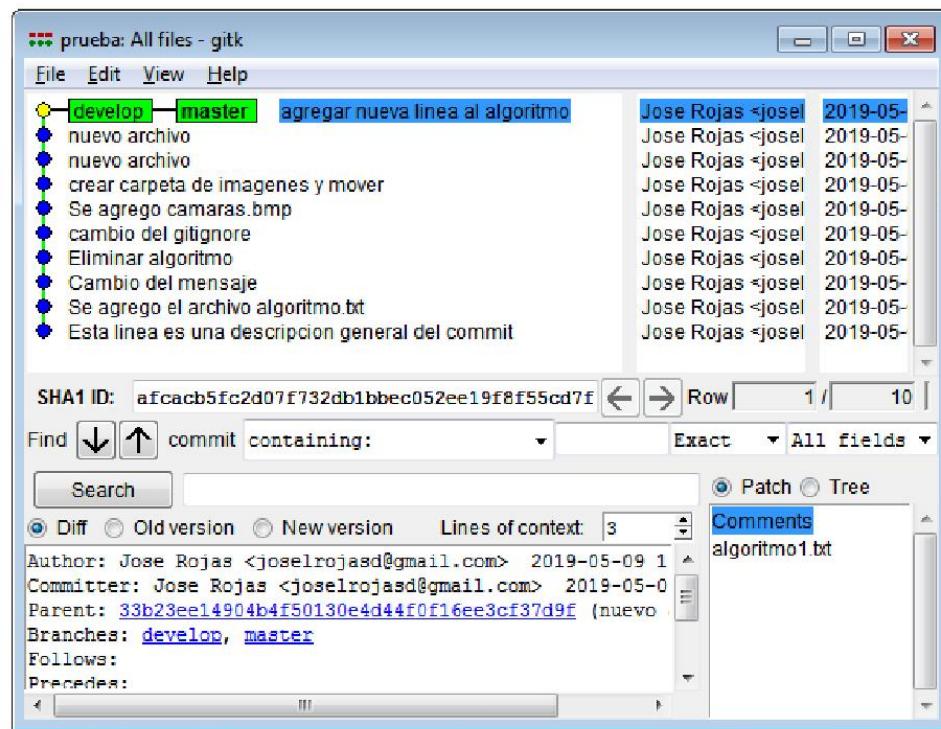
```
git branch nuevarama
```



```
jose@jose-PC MINGW64 ~/Desktop/prueba (master)
$ git branch develop

jose@jose-PC MINGW64 ~/Desktop/prueba (master)
$ git branch
  develop
* master
```

La rama "nuevarama" es un apuntador a la confirmación donde se encuentra el HEAD al momento de crear la rama, como se muestra en la siguiente imagen:



9.3.- Cambiar de Rama

La idea de crear nuevas ramas es el de usarlas para alguna razón. Es por esto que se puede hacer uso del comando "git checkout" para moverse a una de las ramas del repositorio. Para moverse a la rama "desarrollo" se debe usar el siguiente comando:

```
git checkout desarrollo
```

Cuando se pasa a una nueva rama, ahora esta será la rama actual del repositorio y además se estará posicionado en el último commit de esa rama.

```
vilfer@DESKTOP-DMOLHOS MINGW32 ~/Desktop/cadif1 (master)
$ git checkout desarrollo
Switched to branch 'desarrollo'
M      README.txt

vilfer@DESKTOP-DMOLHOS MINGW32 ~/Desktop/cadif1 (desarrollo)
$ git branch
* desarrollo
  master
```

El comando checkout tiene la posibilidad de permitir crear una rama nueva y mover el HEAD a ella en un único paso. Para crear una nueva rama y situar sobre ella se indica el nombre al usar el parámetro -b. De forma general el comando es:

```
git checkout -b otrarama
```

9.4.- Eliminando Ramas

Las ramas en un repositorio son temporales, una vez utilizada y cumplida su función por lo general las ramas son eliminadas para mantener un orden dentro del repositorio. Eliminar una rama simplemente sería quitar la línea de tiempo paralela creada, no afectará en lo absoluto a las demás ramas.

Para proceder a la eliminación de una rama, se usa el siguiente comando:

```
git branch -d nombrerama
```

Si la rama tiene cambios sin fusionar dará un error como el siguiente:

```
error: The branch desarrollo is not an ancestor of your current HEAD.
If you are sure you want to delete it, run git branch -D desarrollo.
```

```
vilfer@DESKTOP-DMOLHOS MINGW32 ~/Desktop/cadif1 (master)
$ git branch -d desarrollo
Deleted branch desarrollo (was fc122de).

vilfer@DESKTOP-DMOLHOS MINGW32 ~/Desktop/cadif1 (master)
$
```



```
vilfer@DESKTOP-DMOLHOS MINGW32 ~/Desktop/cadif1 (desarrollo)
$ git log
commit 7da7e71a40d307be483d4469c0bdfeee5d2aed88 (HEAD -> desarrollo)
Author: Vilfer Alvarez <vilferalvarez@gmail.com>
Date:   Wed Apr 18 01:40:02 2018 -0400

    Agregando una tercera linea al archivo README.txt

commit fc122de2eb57904836960ea4abc8bc06b7cca70e (master)
Author: Vilfer Alvarez <vilferalvarez@gmail.com>
Date:   Sun Apr 8 23:44:24 2018 -0400

    Agregando una linea al archivo REAME.txt

commit 9d617358d9d615a94361ed0c8306fa41dbbf4b23
Author: Vilfer Alvarez <vilferalvarez@gmail.com>
Date:   Sun Apr 8 23:40:29 2018 -0400

    Mi primer commit

vilfer@DESKTOP-DMOLHOS MINGW32 ~/Desktop/cadif1 (desarrollo)
$ |
```

¿Cómo unir los cambios de una rama con otra? En Git se pueden unir ramas con el comando merge como sigue:

Git merge ramaAFusionar

La unión de la ramaAFusionar se va a realizar con la rama en la cual se está posicionado actualmente.

En la fusión de ramas generalmente ocurre uno de los siguientes casos

- Fast-Forward: No existen conflictos de cambios similares entre las ramas.
- Manual Merge: Existen conflictos en cambios hechos en ambas ramas sobre los mismos archivos.

El Fast-Forward ocurre cuando la rama que se desea fusionar es hija de la rama actual, la cual no ha sufrido cambios. En este caso simplemente se fusionan las ramas, se agregan los nuevos commit a la rama actual y se posiciona el HEAD en el último nuevo commit. Es

decir, la rama actual queda ahora con los mismos commit que la rama que se fusionó. Siguiendo con el ejemplo de la rama desarrollo:

```
vilfer@DESKTOP-DMOLHOS MINGW32 ~/Desktop/cadif1 (master)
$ git merge desarrollo
Updating fc122de..7da7e71
Fast-forward
 README.txt | 4 +---
 1 file changed, 3 insertions(+), 1 deletion(-)

vilfer@DESKTOP-DMOLHOS MINGW32 ~/Desktop/cadif1 (master)
$
```

10.2.- Resolviendo Conflictos

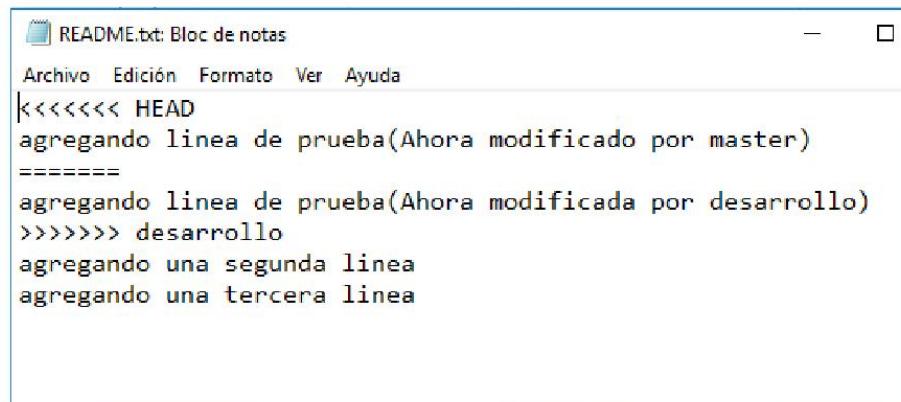
El segundo caso que puede ocurrir al hacer una fusión entre ramas, es que exista confirmaciones de cambios en ambas ramas que hayan afectado a las mismas líneas de uno o más archivos. En este caso Git muestra la advertencia y las líneas en las cuales existen conflictos, así que se debe proceder manualmente a solventar el problema para luego hacer un nuevo commit.

Ejemplo: en la rama master y en la rama desarrollo se modificó la primera línea del archivo README.txt. Al tratar de hacer una unión de desarrollo con master sucede lo que muestra la imagen:

```
vilfer@DESKTOP-DMOLHOS MINGW32 ~/Desktop/cadif1 (master)
$ git merge desarrollo
Auto-merging README.txt
CONFLICT (content): Merge conflict in README.txt
Automatic merge failed; fix conflicts and then commit the result.

vilfer@DESKTOP-DMOLHOS MINGW32 ~/Desktop/cadif1 (master|MERGING)
$ |
```

Git escribe en las líneas de los archivos donde existe conflicto. Es por eso que se deben revisar esos archivos, modificarlos y dejar lo que se conservará y confirmar cambios. La imagen muestra el archivo README.txt de ejemplo:



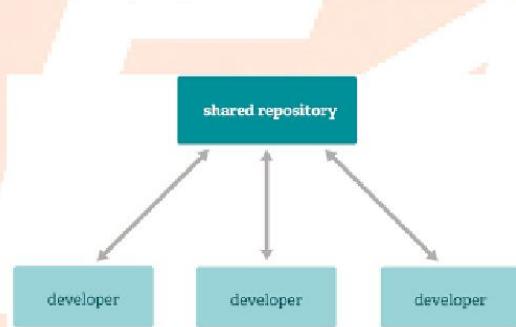
```
 README.txt: Bloc de notas
Archivo Edición Formato Ver Ayuda
<<<<< HEAD
agregando linea de prueba(Ahora modificada por master)
=====
agregando linea de prueba(Ahora modificada por desarrollo)
>>>>> desarrollo
agregando una segunda linea
agregando una tercera linea
```

Capítulo 12. GITHUB. PARTE 1

12.1.- Plataformas Para el Trabajo en Equipo

El trabajo en equipo es fundamental para lograr los objetivos de algún proyecto de manera efectiva, además, que cada vez es más común encontrar equipos de trabajo donde existe una separación física de sus integrantes, lo que se convierte en un reto para las nuevas organizaciones. Es por ello que se ha pensado en el desarrollo de ciertas plataformas que apoyen la integración del trabajo en equipo, facilitando a sus miembros la realización de sus actividades.

Estas plataformas permiten manejar repositorios Git pero ahora de manera compartida con múltiples usuarios, donde todos ellos pueden hacer cambios simultáneamente en el repositorio de manera ordenada y segura.



Entre las plataformas más utilizadas hoy en día que usan git como controlador de versiones están:

- Github
- Bitbucket
- Gitlab
- Coding



GitHub es una forja (plataforma de desarrollo colaborativo) para alojar proyectos utilizando el sistema de control de versiones Git. Se utiliza principalmente para la creación de código fuente de programas de computadora. El software que opera GitHub fue escrito en Ruby on Rails. Desde enero de 2010, GitHub opera bajo el nombre de GitHub, Inc. Anteriormente era conocida como Logical Awesome LLC. El código de los proyectos alojados en GitHub se almacena típicamente de forma pública, aunque utilizando una cuenta de pago, también permite hospedar repositorios privados.



GitHub es el host más grande para los repositorios de Git, y es el punto central de colaboración para millones de desarrolladores y proyectos. Un gran porcentaje de todos los repositorios de Git están alojados en GitHub, y muchos proyectos de código abierto lo usan para el alojamiento de Git, el seguimiento de problemas, la revisión de códigos y otras cosas. Entonces, si bien no es una parte directa del proyecto de código abierto de Git, hay muchas posibilidades de que desee o necesite interactuar con GitHub en algún momento mientras usa Git profesionalmente.

12.2.- Usando Github

Lo primero es crear una cuenta en GitHub. Para eso debe ir al sitio oficial <https://github.com/> donde podrá encontrar un formulario para crear una nueva cuenta. Se le solicitará información personal como nombre, email y password.

Username
Pick a username

Email
you@example.com

Password
Create a password

Use at least one letter, one numeral, and seven characters.

Sign up for GitHub

By clicking "Sign up for GitHub", you agree to our [terms of service](#) and [privacy statement](#). We'll occasionally send you account related emails.

En la sección de “mi perfil” se puede culminar de añadir la información personal relevante que le sea solicitada. Ya en el dashboard de GitHub se pueden observar características como una barra la búsqueda tanto de personas como de proyectos en los cuales se deseé colaborar.

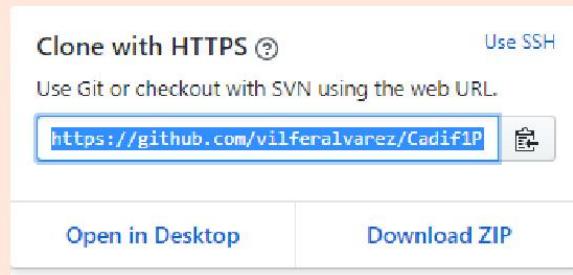


```
vilfer@DESKTOP-DM0LH0S MINGW32 ~/Desktop/cadif1 (master)
$ git remote -v

vilfer@DESKTOP-DM0LH0S MINGW32 ~/Desktop/cadif1 (master)
$ |
```

El repositorio remoto tiene asociado una url dentro de la plataforma con el siguiente patrón:

<https://github.com/vilferalvarez/Cadif1Project.git>



Para establecer la conexión entre los 2 repositorios, se debe usar una variante del comando remote como sigue:

```
git remote add nombre url
```

Donde “nombre” es el nombre para identificar el repositorio remoto y la “url” para especificar la dirección exacta de ese repositorio en la web. En la imagen siguiente se muestra la conexión con el repositorio creado en prácticas anteriores:

```
vilfer@DESKTOP-DM0LH0S MINGW32 ~/Desktop/cadif1 (master)
$ git remote add origin https://github.com/vilferalvarez/Cadif1Project.git

vilfer@DESKTOP-DM0LH0S MINGW32 ~/Desktop/cadif1 (master)
$ git remote -v
origin  https://github.com/vilferalvarez/Cadif1Project.git (fetch)
origin  https://github.com/vilferalvarez/Cadif1Project.git (push)
```

El nombre del repositorio remoto es arbitrario. En cualquier momento se puede cambiar el nombre con el comando "git remote rename". La forma general del comando es:

```
git remote rename [antiguo-nombre] [nuevo-nombre]
```

Donde "antiguo-nombre" es el nombre original y "nuevo-nombre" el nombre que se desea asignar.

También se puede eliminar un repositorio remoto con el comando "git remote rm". La forma general del comando es:

```
git remote rm [nombre-remoto]
```

Donde "nombre-remoto" es el nombre del repositorio remoto que se desea eliminar. Por ejemplo:

```
git remote rm origin
```

Eliminará el repositorio remoto que tiene el nombre "origin".

13.3.- Clonando Repositorios

Es muy común tener la necesidad de obtener algún proyecto de un repositorio remoto que ya esté adelantado, solo para agregarlo al ambiente local. Para obtener un repositorio remoto de GitHub sin tener ningún repositorio local, el usuario debe colocarse en la carpeta donde necesita que se cree el repositorio y usar el comando "clone" de git.

Para clonar un repositorio se necesita la URL del repositorio. La estructura del comando clone es la siguiente:

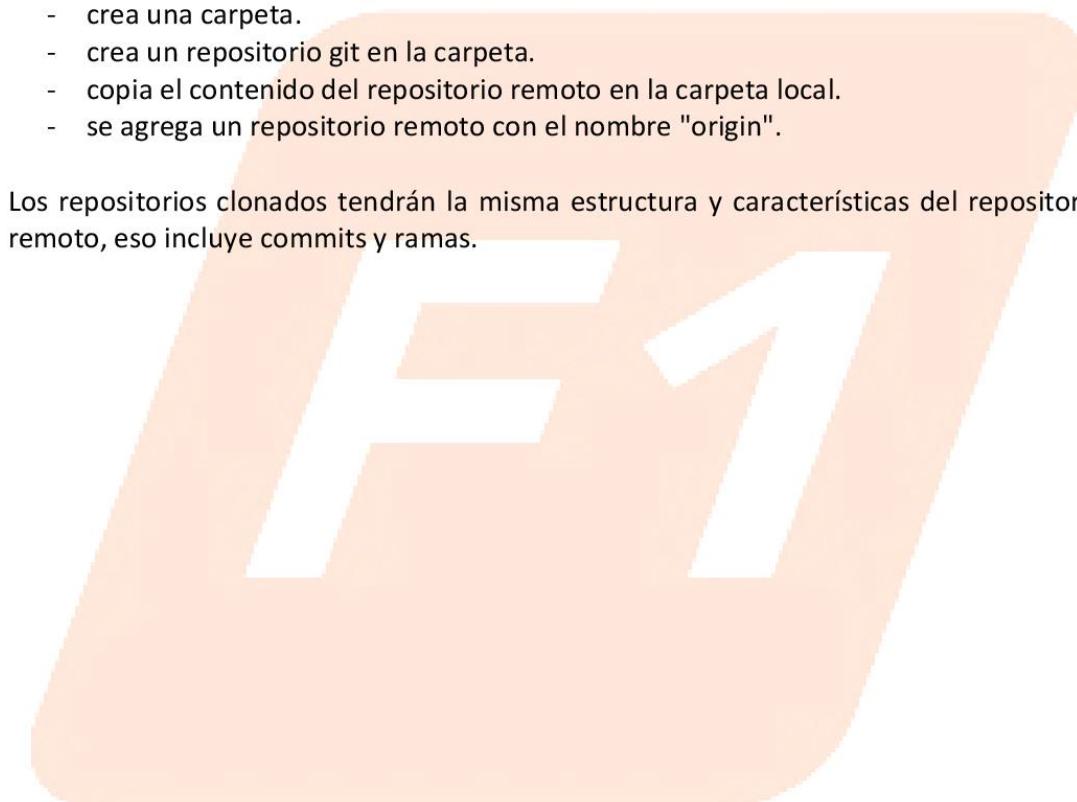
```
Git clone url
```

```
vilfer@DESKTOP-DM0LH0S MINGW32 ~/Desktop/proyecto2
$ git clone https://github.com/vilferalvarez/Cadif1Project.git
Cloning into 'Cadif1Project'...
```

El comando git clone realiza las siguientes acciones:

- crea una carpeta.
- crea un repositorio git en la carpeta.
- copia el contenido del repositorio remoto en la carpeta local.
- se agrega un repositorio remoto con el nombre "origin".

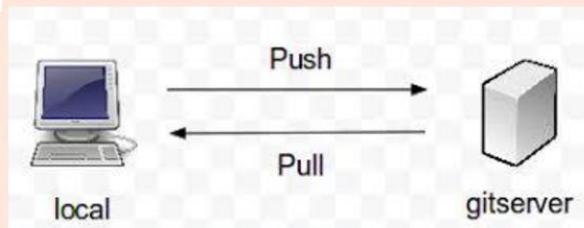
Los repositorios clonados tendrán la misma estructura y características del repositorio remoto, eso incluye commits y ramas.



Capítulo 14. OPERACIONES CON REPOSITORIOS REMOTOS

14.1.- Operaciones

Una vez se dispone de un repositorio local y uno remoto en los que trabajar, se debe aprender a subir los cambios que se han hecho en el repositorio local al repositorio remoto. Se deben asociar los repositorios para que la plataforma redireccione hacia el repositorio remoto los cambios que se hacen en local.



14.2.- Descargar Cambios

Para traer cambios desde el servidor se utiliza el comando "git fetch". La forma general del comando es:

```
git fetch [remoto]
```

Donde "remoto" es el nombre del repositorio remoto de los que están asociados al repositorio local. Si el repositorio fue clonado,

14.3.- Subir Cambios

Para subir los cambios al repositorio remoto solo se debe tener alguna confirmación preparada localmente y proceder a usar el comando: "git push". El comando push es el encargado de enviar los cambios de Local al repositorio remoto. La forma general del comando es la siguiente:

```
git push [nombre-remoto] [nombre-rama]
```

Se debe especificar la rama que se desea subir. Si se desea enviar la rama maestra (master) al servidor origen (origin), se debe usar el siguiente comando:

```
git push origin master
```

Este comando solo funciona si se clonó un repositorio de un servidor sobre el que se tiene permisos de escritura y si nadie más ha enviado datos antes. Si alguien más clona el mismo repositorio y envía información antes, el envío será rechazado.



Capítulo 15. GITHUB. PARTE 2

15.1.- Contribuyendo en Proyectos de Terceros

Uno de los motivos de la creación de GitHub era la de crear un ambiente de colaboración entre desarrolladores de todo el mundo. GitHub le da la posibilidad al usuario de revisar repositorios de terceros y contribuir con piezas de código que sean necesarias. Hasta el momento en este material se ha trabajado con un repositorio local (propio del usuario). Pero ¿Qué tal si lo que se desea es contribuir en un proyecto en el cual no se tiene permisos para modificarlo ? Es allí donde aparece el término “Fork”.

La palabra fork se traduce al castellano, dentro del contexto del software, como bifurcación. Cuando se hace un fork de un repositorio, se hace una copia exacta en crudo (en inglés “bare”) del repositorio original que se puede utilizar como un repositorio git cualquiera. Despues de hacer Fork se tendrán dos repositorios git idénticos, pero con distinta URL. Justo despues de hacer el Fork, estos dos repositorios tienen exactamente la misma historia, son una copia idéntica. Finalizado el proceso, se tendrán dos repositorios independientes que pueden cada uno evolucionar de forma totalmente autónoma. De hecho, los cambios que se hacen en el repositorio original NO se transmiten automáticamente a la copia (Fork). Esto tampoco ocurre a la inversa: las modificaciones que se hagan en la copia (Fork) NO se transmiten automáticamente al repositorio original.



Cuando se realiza el Fork de un repositorio, se crea un repositorio idéntico en la cuenta del usuario que ha hecho el Fork. Para hacer un Fork se debe dirigir al repositorio al que desea contribuir y hacer click en el icono de Fork de la parte superior derecha. Automáticamente se creará en su cuenta un repositorio con el mismo nombre

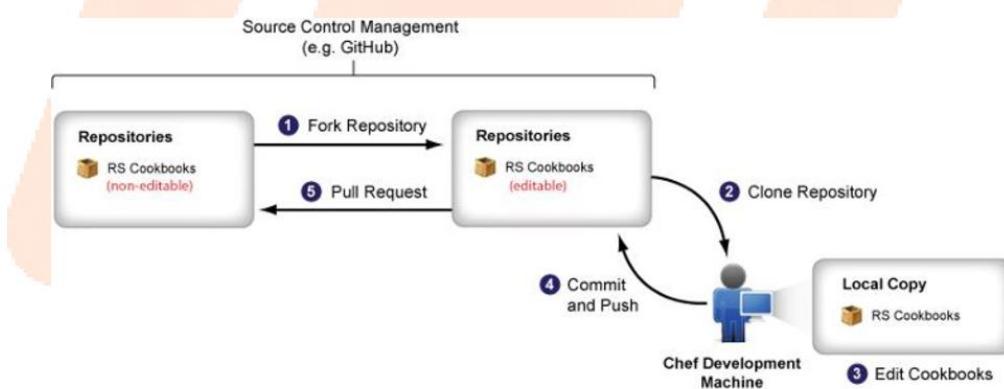


Los repositorios los cuales son producto de un Fork y que ahora están en la cuenta del usuario contribuyente con el proyecto, será un repositorio más en su cuenta, es decir, puede clonarlo y comenzar a trabajar sobre el, posteriormente subir cambios a este proyecto copia del original.

15.2.- Pull Request

Los cambios hechos en los proyectos bifurcados de otros repositorios, solo serán visibles en la copia del repositorio y no en la versión original del mismo, esto quiere decir que hasta ahora se ha colaborado propiamente con el proyecto. Esta es una de las diferencias notables con el procedimiento de clonado, ya que al hacer Fork no se pueden hacer cambios directamente sobre el proyecto original, lo que da más seguridad a la integridad del proyecto.

La imagen muestra el flujo de trabajo para contribuir con un repositorio de un tercero:



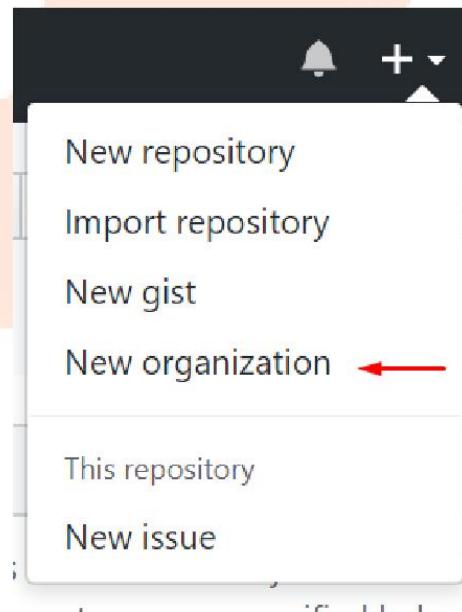
Los Pull Request serán peticiones que se hacen al dueño del proyecto para que integre los cambios que ha realizado un tercero sobre una copia de su repositorio. Será decisión del dueño del repositorio si integra los cambios o no. En el momento en que se hagan

cambios locales y se suban al repositorio copia(Fork), GitHub habilita un botón identificado “Pull Request” el cual será el encargado de enviar la petición de integración de cambios al dueño del repositorio original.

15.3.- Organizaciones

Además de las cuentas de usuario, GitHub tiene Organizaciones. Al igual que las cuentas de usuario, las cuentas de organización tienen un espacio donde se guardarán los proyectos, pero en otras cosas son diferentes. Estas cuentas representan un grupo de personas que comparte la propiedad de los proyectos, y además se pueden gestionar estos miembros en subgrupos. Normalmente, estas cuentas se usan en equipos de desarrollo de código abierto (por ejemplo, un grupo para “perl” o para “rails”) o empresas (como sería “google” o “twitter”).

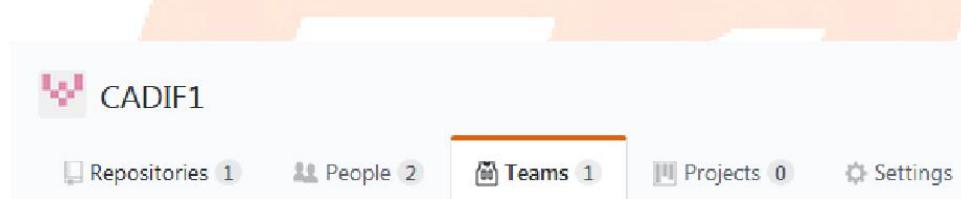
Para crear una nueva organización, simplemente se debe pulsar en el icono “+” en el lado superior derecho y selecciona “New organization”. En primer lugar, tienes que decidir el nombre de la organización y una dirección de correo que será el punto principal de contacto del grupo. A continuación, puedes invitar a otros usuarios a que se unan como co-propietarios de la cuenta.



En las organizaciones serán creados los repositorios en los cuales trabajarán los miembros de la organización. Los miembros podrán realizar cambios directamente en los repositorios según los permisos que se les sean asignados, demostrando una vez más la seguridad a nivel de acceso que posee la plataforma.

15.4.- Equipos

Los Team (equipos) dentro de una organización ayudan a dividir de manera organizada las responsabilidades que pueden tener los miembros de la organización para el aporte a los repositorios. En una organización se pueden crear los equipos que se consideren necesarios y además agregar en cada equipo a los miembros de la organización que se consideren. En la imagen se muestra la organización CADIF1 en la cual existe un Team.



Para crear un nuevo equipo dentro de la organización, se debe ir a la pestaña “Team” y hacer click en el botón “New Team”. Se debe colocar un nombre y una breve descripción al equipo

Create new team

Team name

What is the name of this team?

You'll use this name to mention this team in conversations.

Description

What is this team all about?

Parent team

Select parent team ▾

Team visibility

Visible Recommended

A visible team can be seen and @mentioned by every member of this organization.

Secret

A secret team can only be seen by its members.

Create team