

# Gestión de Proyectos con Git

CORE IWEB 2018-2019

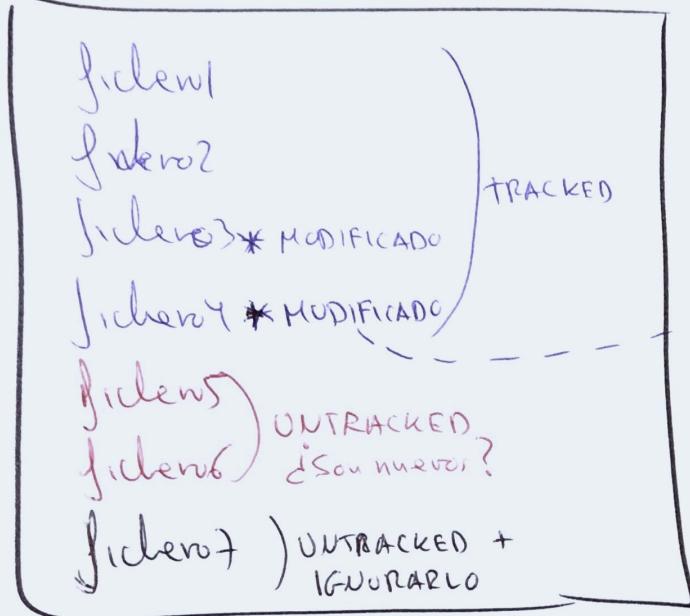
Santiago Pavón

Juan Quemada

Versión: 2019-01-30

# Gestión de Versiones a mano

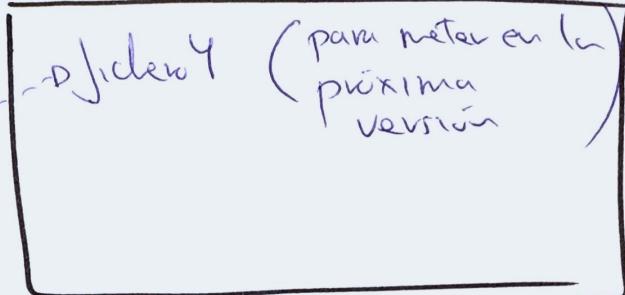
## Directorio de Trabajo



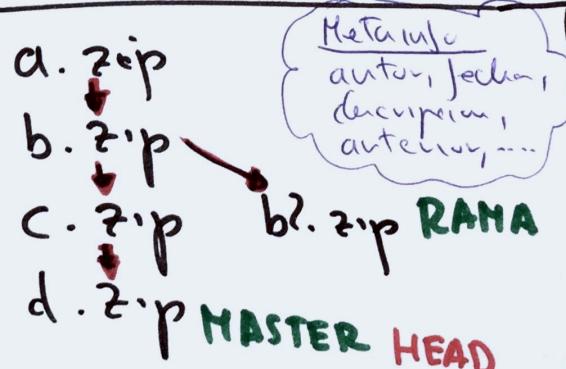
Nueva versión apartir  
de HEAD

## Gestión de Versiones

### STAGING AREA:



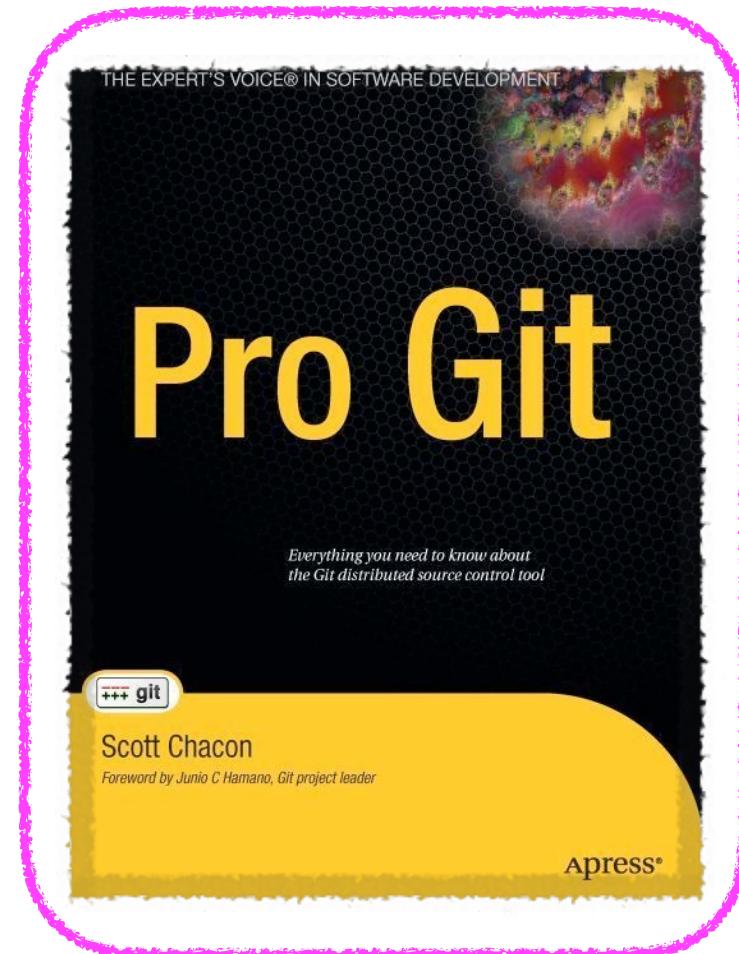
### Versiones congeladas:



# Introducción a GIT

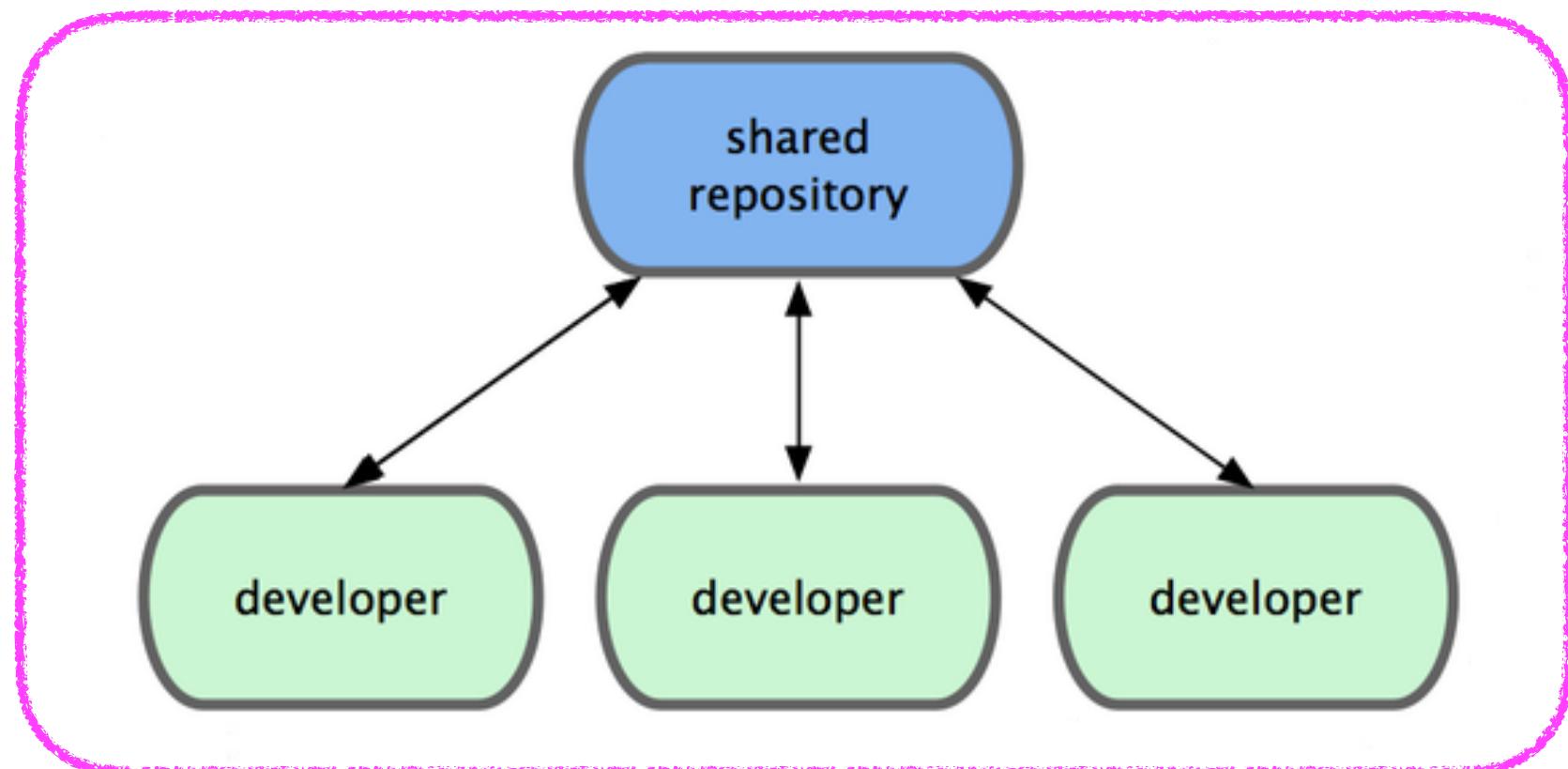
# GIT

- GIT: gestor de versiones
  - Desarrollado por Linus Torwalds para Linux
    - Desarrollo colaborativo de proyectos
  - Muy eficaz con proyectos
    - grandes o pequeños
  - Tutorial Web y eBook gratis
    - <http://git-scm.com/book/en>
    - <http://git-scm.com/book/es>
    - Otros:
      - <http://gitref.org>



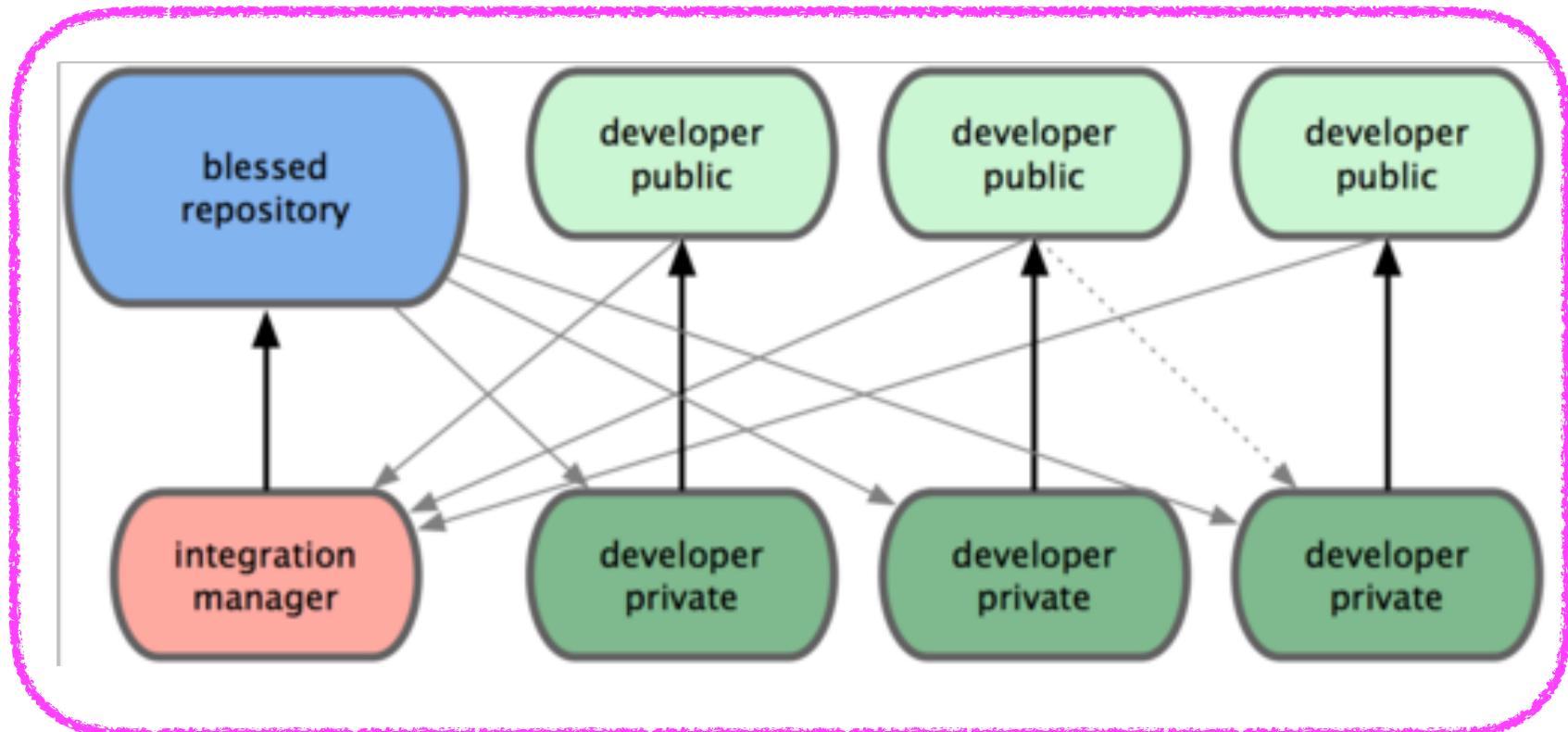
# Políticas de Organización

- Flujo de trabajo centralizado



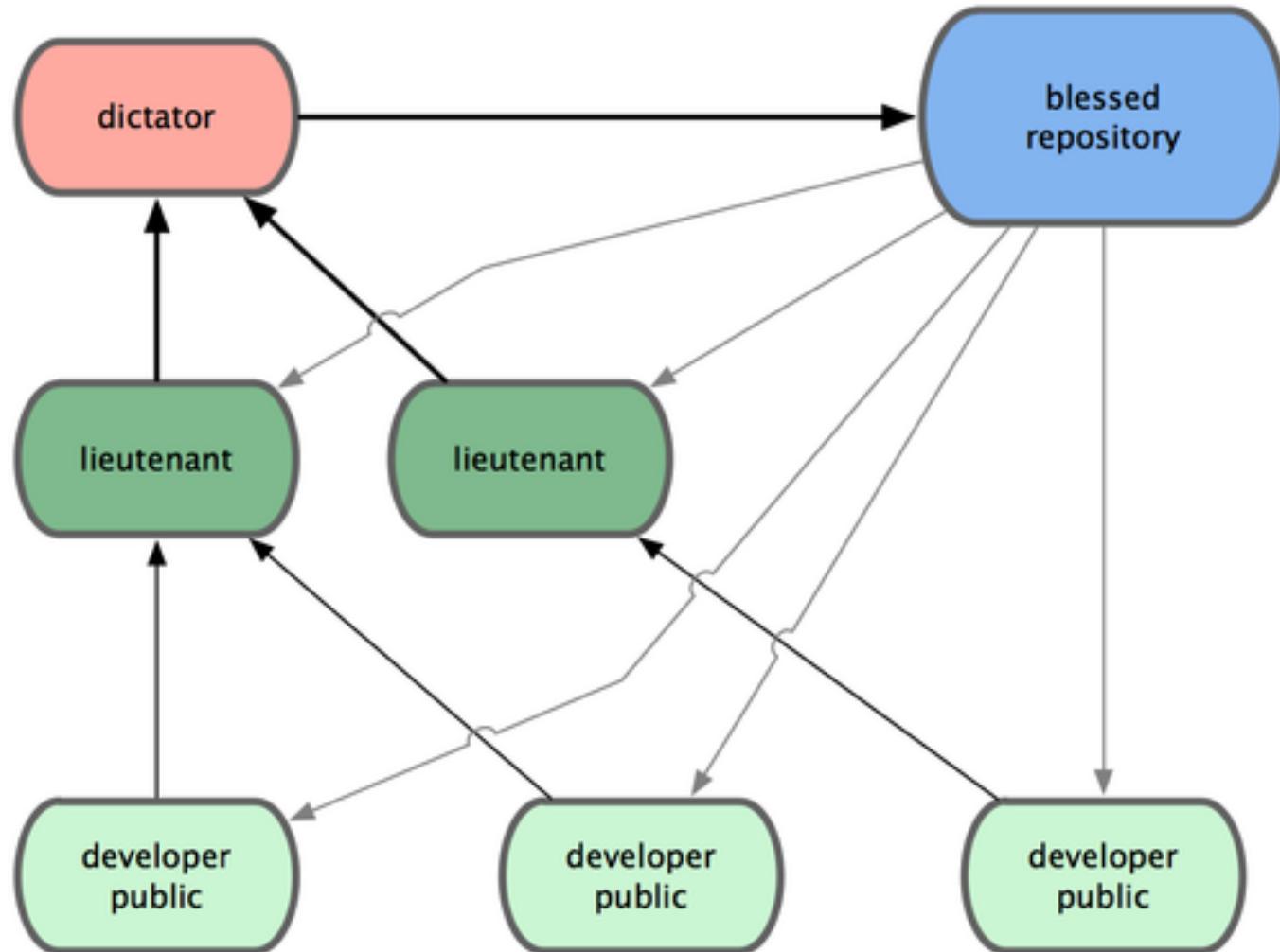
\*de Scott Chanson: <http://git-scm.org/book/>

- Flujo de trabajo Integration-Manager



\*de Scott Chanson: <http://git-scm.org/book/>

- Flujo de trabajo Dictador - Tenientes



\*de Scott Chanson: <http://git-scm.org/book/>

# Servidores

- Los desarrolladores tienen copias de los repositorios en sus ordenadores de trabajo, pero además, los repositorios también suelen estar alojados en ordenadores que hacen el papel de servidores.
- Los hosts servidores tienen que configurarse y mantenerse.
  - Cuentas de usuarios, permisos, acceso anónimo, etc.
  - Configuración de los demonios.
  - Protocolos de acceso.  
`file:///home/juan/demo.git`  
`git@github.com:core/demo.git`  
`ssh://github.com/core/demo`  
`https://github.com/core/demo`
  - Creación de los repositorios.
    - Repositorios git: creados con la opción --bare.
  - etc...
- Los repositorios también pueden alojarse en sitios que se dedican al hosting de proyectos git:
  - GitHub, Gitorious, Assembla, ...

# GITHUB

- Portal Web para alojar repositorios GIT
  - Enfoque social y colaborativo
  - Facilita la comunicación en un grupo y con terceros
- Proyectos open source son gratis, privados de pago
  - Aloja: Linux Kernel, Eclipse, jQuery, Ruby on Rails, ...
- Acceso al portal: <https://github.com>

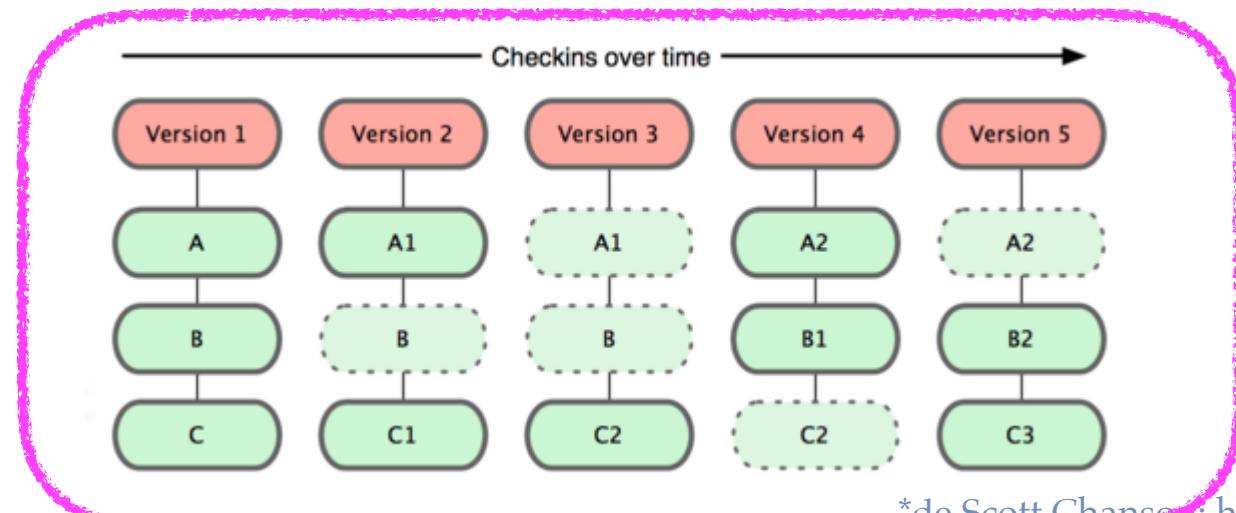
# Instalar GIT



- En **Ubuntu**:
  - Instalar el paquete git ejecutando:  
`$ sudo apt-get install git`
- En **Mac**:
  - Instalar Xcode.
  - Alternativa Home Brew.
- En **Windows**:
  - El instalador está en <http://msysgit.github.com>
    - Al instalar indicar que queremos ejecutar git desde el terminal de comandos.
    - Se instalará cygwin con más comandos unix.

# Historia de un Proyecto

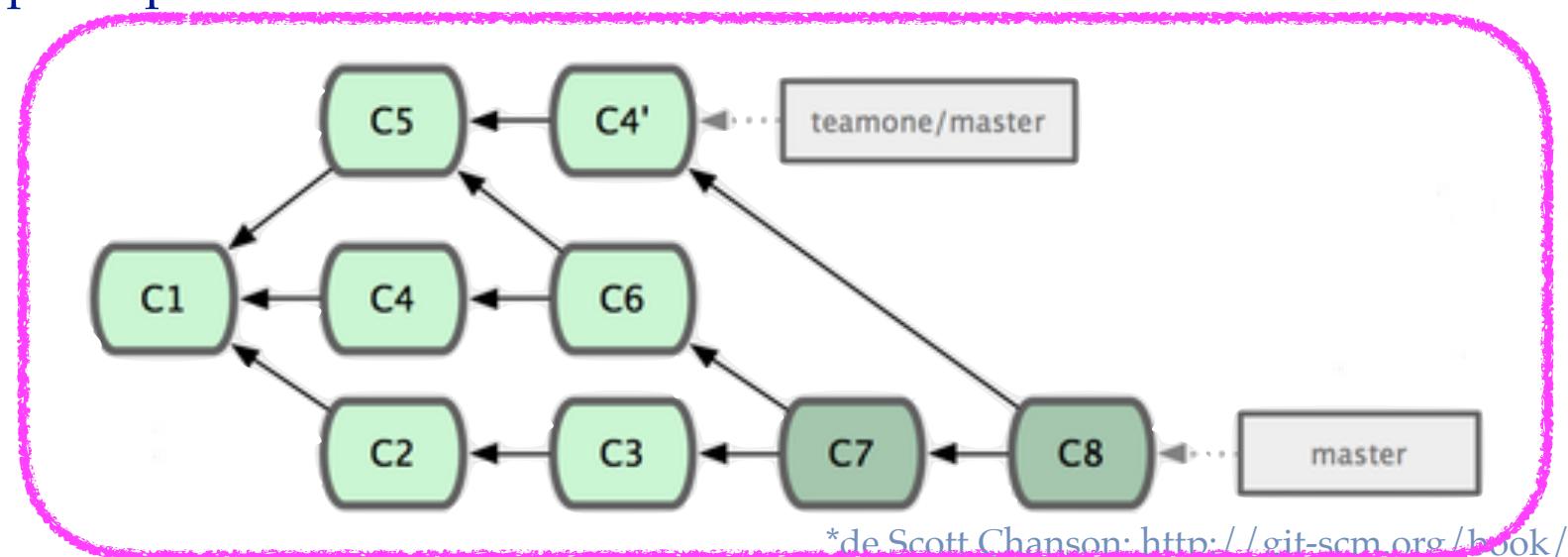
- Durante la vida de un proyecto se hacen cambios, generando nuevas versiones de él.
- Cada versión añade nuevas funcionalidades, o las modifica, borra, ...
  - Conviene consolidar versiones a menudo.
    - Con pocos cambios por versión



\*de Scott Chacon: <http://git-scm.org/book>

# Árbol de desarrollo

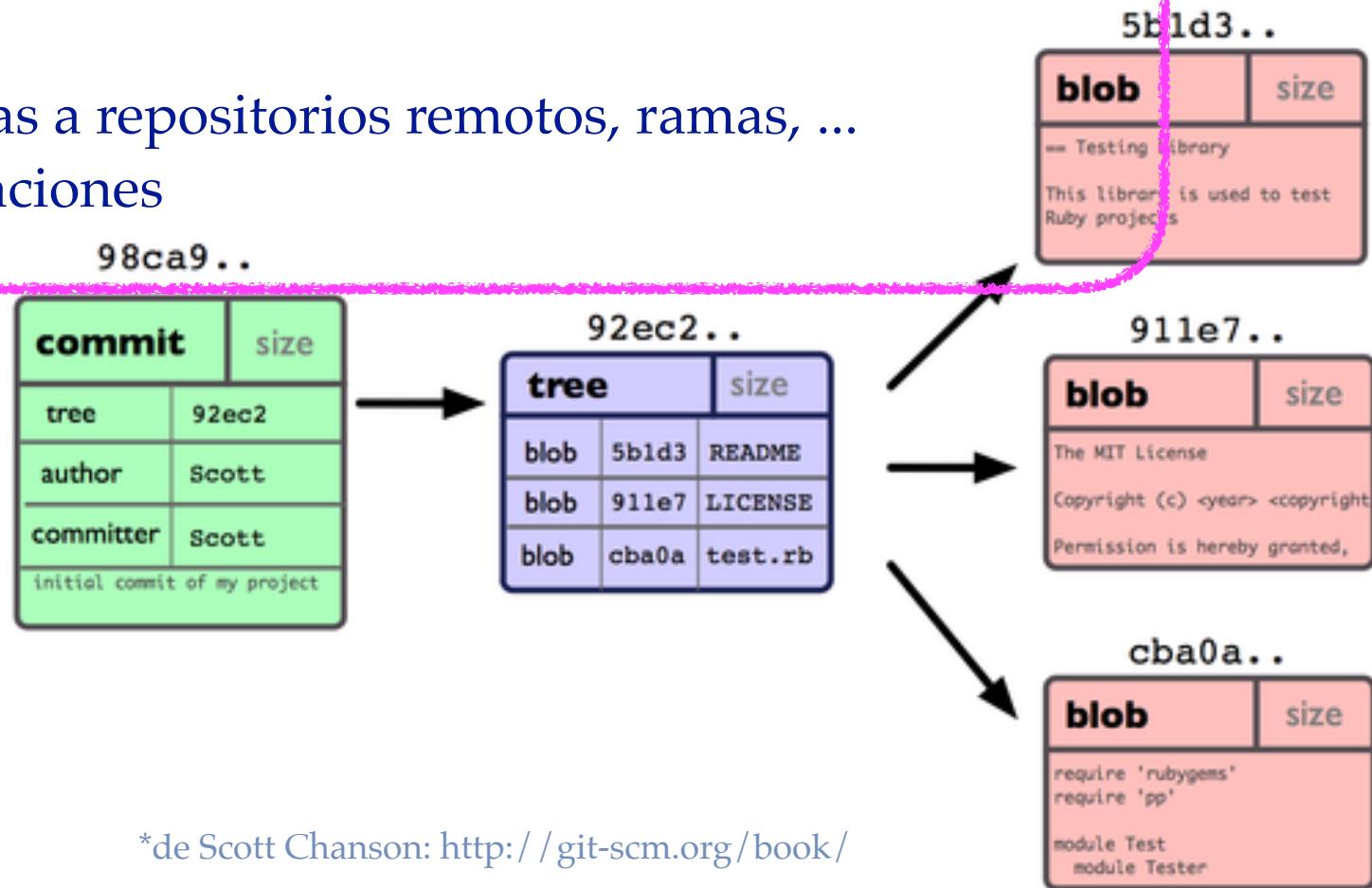
- Proyectos software en equipo son complejos.
- Múltiples desarrollos en paralelo.
  - nuevas funcionalidades, corrección de errores, mejoras, ...
- Cada desarrollo es una rama del árbol.
  - Ramas con cambios estables se integran (mezclan) en la rama principal.



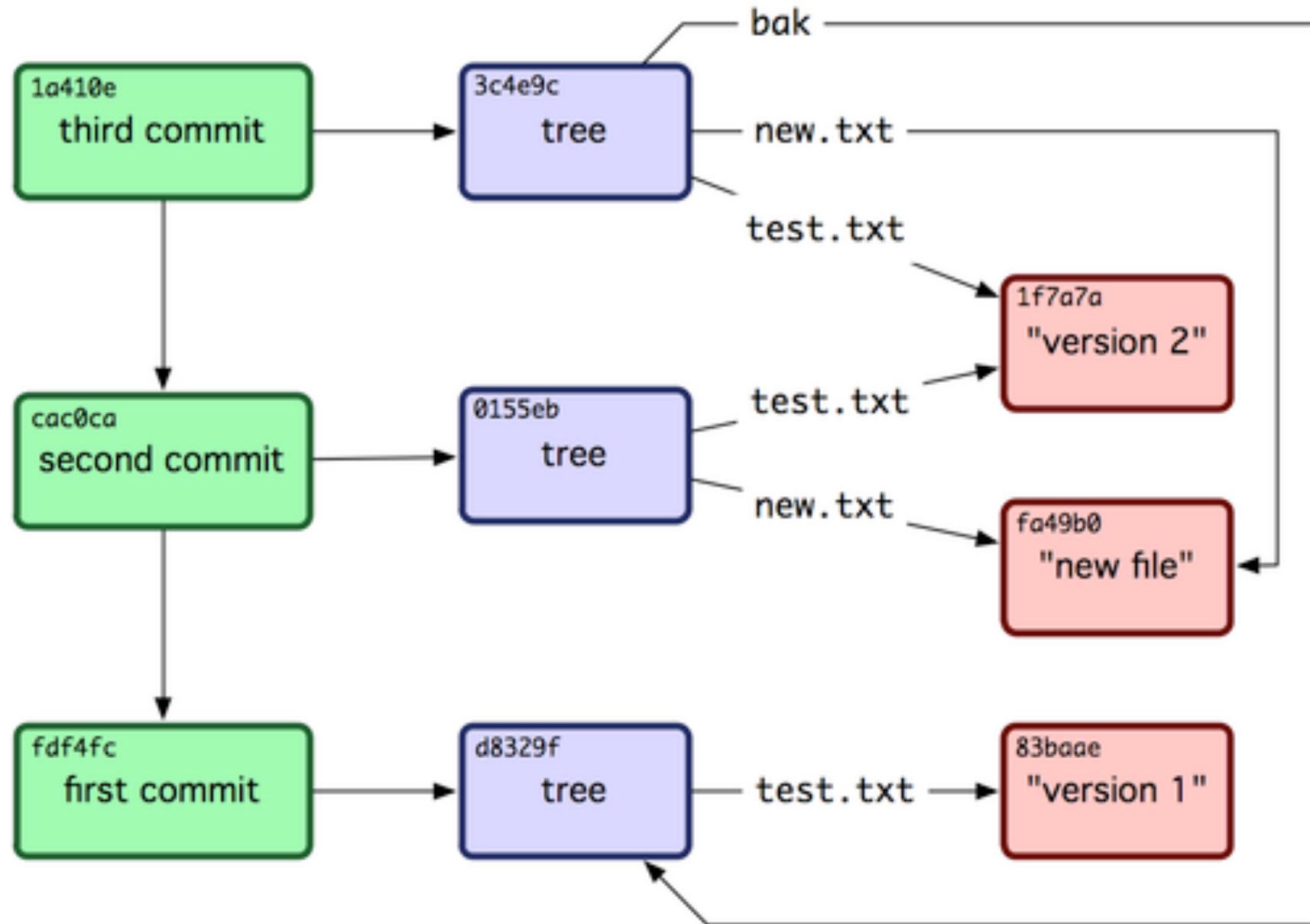
\*de Scott Chanson: <http://git-scm.org/book/>

# Información almacenada en .git

- En el subdirectorio .git se almacena:
  - objetos que representan los ficheros, los directorios, los commit,...
  - Referencias a repositorios remotos, ramas, ...
  - Configuraciones
  - ...



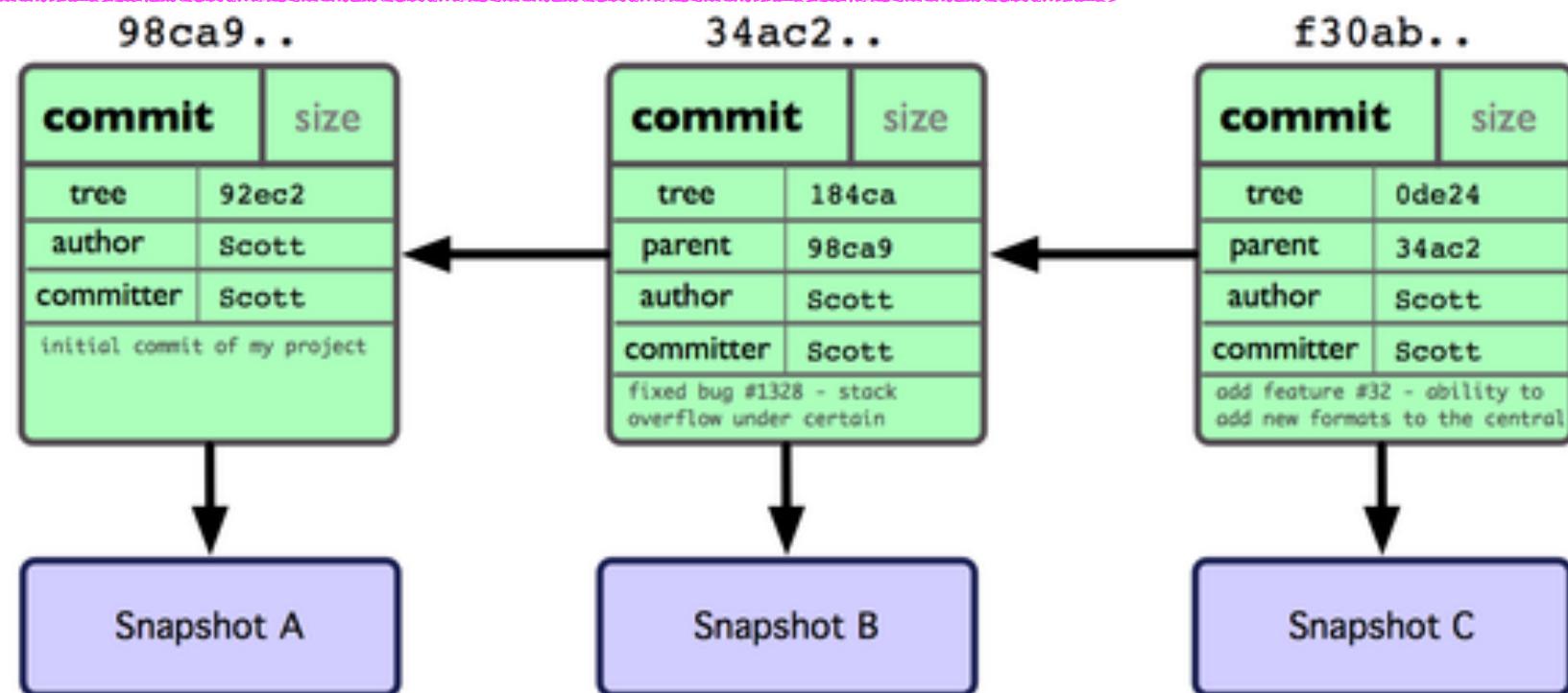
\*de Scott Chanson: <http://git-scm.org/book/>



\*de Scott Chanson: <http://git-scm.org/book/>

# Árbol de Commits

- La historia de versiones (commits) es un árbol.
  - Cada commit apunta a su padre (o padres).



\*de Scott Chanson: <http://git-scm.org/book/>

# Comandos Básicos

# Ayuda

- Ayuda en línea de comandos:

\$ git help

\$ git help add

\$ git add --help

\$ man git-add

# Muestra lista con los comandos existentes

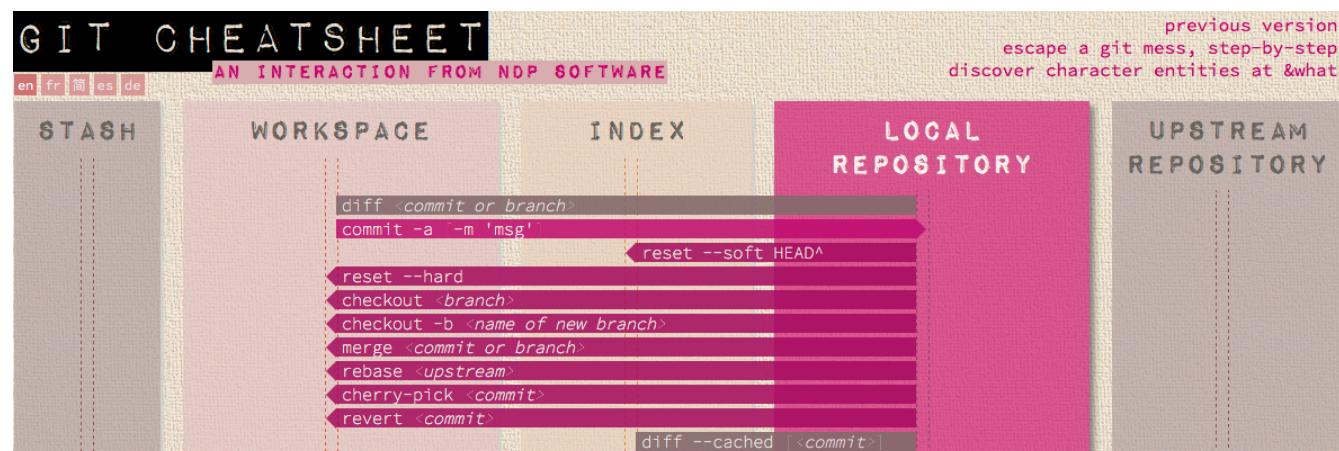
# Ayuda sobre comando especificado

# Equivalente a anterior

# Equivalente a anterior

- Chuleta:

<http://ndpsoftware.com/git-cheatsheet.html>



# Configurar GIT

- Usar el comando **git config** para manejar la configuración de un proyecto.
- Opciones para seleccionar el ámbito al que afecta la configuración:
  - **--system**
    - Para todos los proyectos en el sistema. La configuración se guarda en `/etc/gitconfig`
  - **--global**
    - Para todos los proyectos del usuario. La configuración se guarda en `~/.gitconfig`
  - Sin opción
    - Sólo para el proyecto actual. La configuración se guarda en `./.git/config`
- Consultar todas las opciones existentes: **git help config**
- Dos opciones para identificar a los autores::

```
$ git config --global user.name "Pedro Ramirez"  
$ git config --global user.email pramirez@dit.upm.es
```
- Consultar el valor de todas las opciones configuradas:

```
$ git config --list  
user.name=Pedro Ramirez  
user.email=pramirez@dit.upm.es  
color.ui=true
```
- Consultar el valor de una opción:

```
$ git config user.name  
Pedro Ramirez
```

# Crear Proyecto desde Cero

- El proyecto gestionado por GIT debe estar contenido en un directorio:
  - El directorio de trabajo: contiene los ficheros y subdirectorios del proyecto.
- Los comandos GIT deben invocarse dentro del directorio de trabajo o subdirectorios.

```
$ cd ...../demo      # Moverse a la raíz del directorio de trabajo del proyecto  
  
demo$ git init        # Crea un nuevo proyecto GIT vacío en directorio demo  
                      # Se crea el subdirectorio .git con los ficheros del repositorio  
  
demo$ git add readme.txt    # Añade readme.txt al index  
demo$ git add *.java       # Añade todos los java al index  
demo$ git add .             # Añade todos los ficheros modificados al index  
  
demo$ git commit -m 'Primera versión'    # Congela una versión del proyecto en el repositorio
```

- A partir de este momento:
  - Crear nuevos ficheros, editar ficheros existentes, borrar ficheros, ...
  - Añadir (**add**) los cambios a el index.
  - Congelar (**commit**) una nueva versión con los cambios metidos en el index.

# Clonar un Proyecto Existente

- Para crear un duplicado de un proyecto git existente se usa el comando:

```
git clone <URL>
```

- Se duplica (clona) el repositorio apuntado por URL,
  - Incluyendo toda su historia (todas las versiones congeladas).
  - El URL soporta varios protocolos: http: git: file: ssh:

- Clonar un proyecto ya existente en github:

```
$ git clone http://github.com/ging/demo
```

- Se crea un directorio llamado demo con el clon.
- Podemos indicar cuál es el nombre del directorio a crear  

```
$ git clone http://github.com/ging/demo myDemo
```

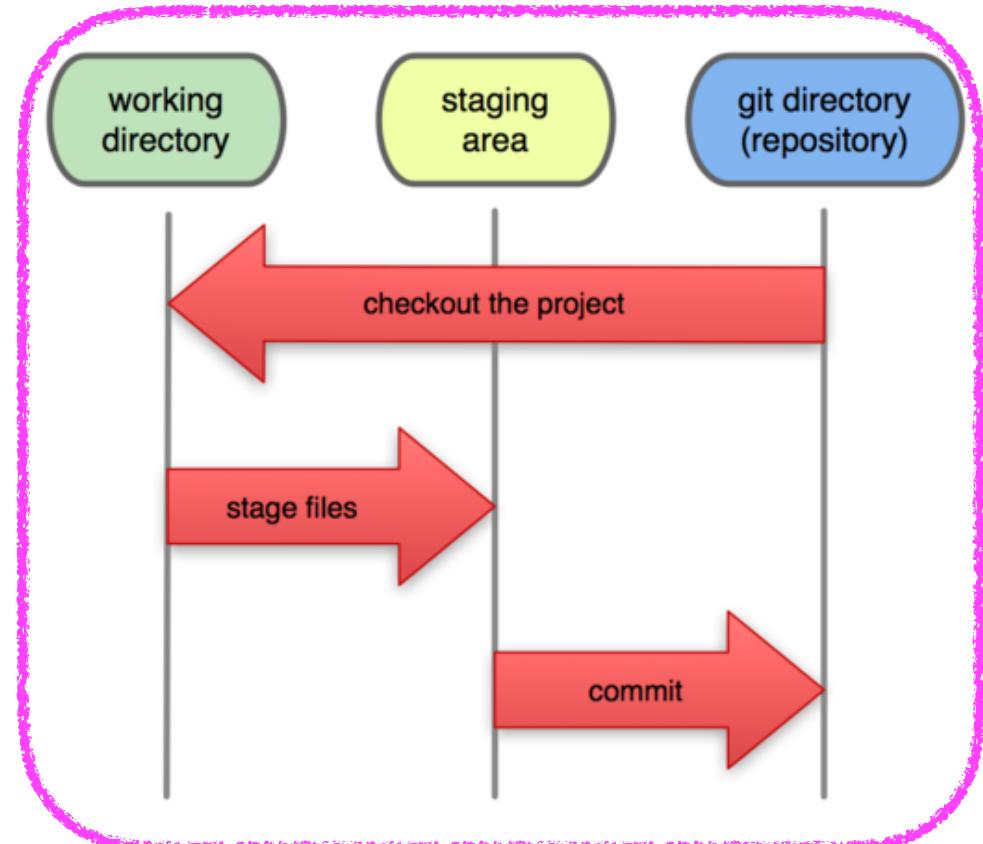
  - El repositorio clonado se creará en el directorio my\_planet

- A partir de este momento:

- Podemos trabajar normalmente en el proyecto clonado: add, commit, ...
  - Y en algunos momentos sincronizaremos los dos repositorios.

# Secciones de un Proyecto GIT

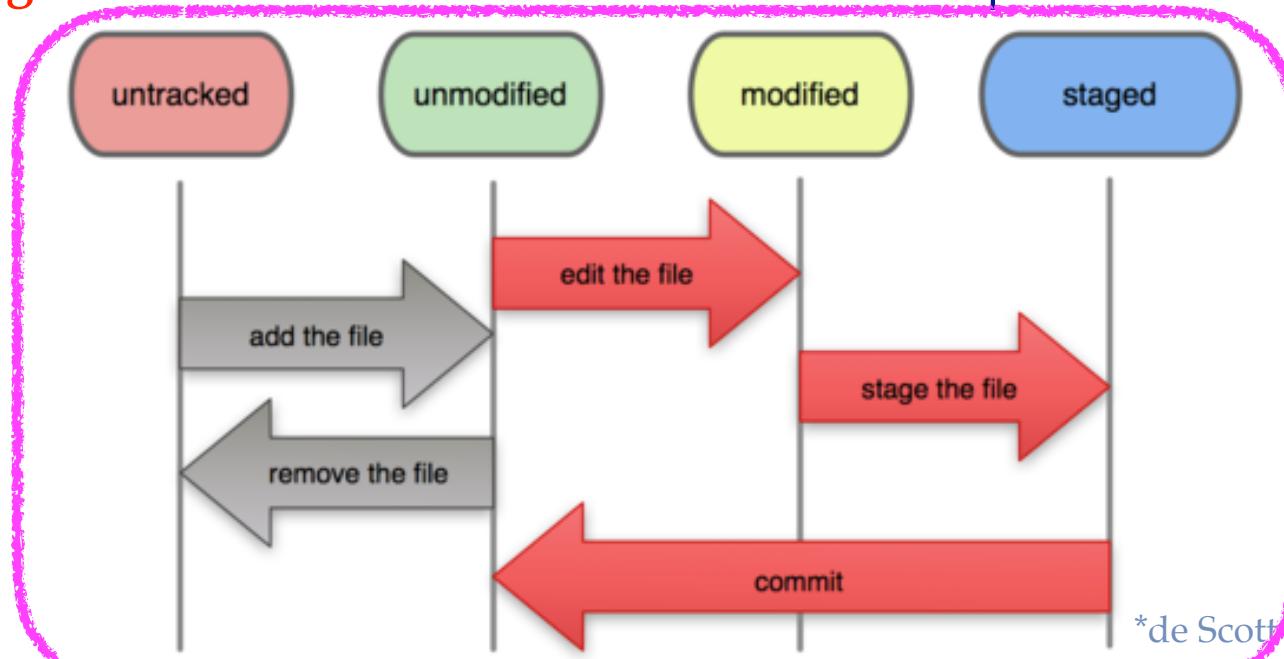
- Directorio de trabajo
  - Contiene los ficheros con los que estamos trabajando.
    - Se habrán copiado (checkout) de una versión del repositorio, o habrán sido creados por nosotros.
- Staging area o index
  - Contiene las modificaciones realizadas en los ficheros de directorio de trabajo que se guardarán en el próximo commit.
- Directorio GIT (.git)
  - Contiene el repositorio con todas las versiones del proyecto, los ficheros, metadatos, ...



\*de Scott Chanson: <http://git-scm.org/book/>

# Estado de los Ficheros

- Estado de los ficheros del directorio de trabajo.
  - **Untracked**. Ficheros que no están bajo el control de versiones.
  - **Tracked**. Ficheros bajo el control de versiones:
    - **Unmodified**: fichero no modificado, idéntico a la versión del repositorio.
    - **Modified**: fichero con modificaciones no incluidas en el staging area.
    - **Staged**: fichero con modificaciones a incluir en próximo commit.



# Consultar el Estado de los Ficheros

- El comando **git status** muestra estado de los ficheros del directorio de trabajo:

```
$ git status
```

On branch **master**

**Changes to be committed:**

(use "git reset HEAD <file>..." to unstage)  
modified: README  
new file: CharIO.java

Ficheros nuevos o con modificaciones incluidos en el staging area

**Changed but not updated:**

(use "git add <file>..." to update what will be committed)  
modified: benchmarks.rb

Ficheros modificados no incluidos en el staging area

**Untracked files:**

(use "git add <file>..." to include what will be committed)  
merge.java  
library/lib.js

Ficheros no gestionados por GIT y no excluidos por .gitignore

# .gitignore

- En el fichero **.gitignore** se añade el nombre de los ficheros que no debe gestionar GIT.
  - **git status** no los presentará como ficheros **untracked**.
  - **git add .** no los añadirá al staging area.
- Pueden crearse ficheros **.gitignore** en cualquier directorio del proyecto,
  - Afectan a ese directorio y a sus subdirectorios.
- Su contenido: líneas con patrones de nombres.
  - Puede usarse los comodines **\*** y **?**.
  - Patrones terminados en **/** indican directorios.
  - Un patrón que empiece con **!** indica negación.
  - Se ignoran líneas en blanco y que comiencen con **#**.
  - **[abc]** indica cualquiera de los caracteres entre corchetes.
  - **[a-z]** indica cualquiera de los caracteres en el rango especificado.
- Ejemplos:

<code>private.txt</code>	excluir los ficheros con nombre <code>private.txt</code>
<code>*.class</code>	excluir los ficheros bytecode de java
<code>*.[oa]</code>	excluir ficheros acabados en <code>.o</code> y <code>.a</code>
<code>!lib.a</code>	no excluir el fichero <code>lib.a</code>
<code>~~</code>	excluir ficheros acabados en <code>~</code>
<code>testing/</code>	excluir los directorios llamados <code>testing</code>

# Consultar Cambios: No Staged

- El comando **git diff** muestra las modificaciones realizadas en los ficheros del área de trabajo que aun no han sido incluidas en el staging área.

\$ **git diff**

```
diff --git a/benchmarks.rb b/benchmarks.rb
index 3cb747f..da65585 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -36,6 +36,10 @@ def main
    @commit.parents[0].parents[0].parents[0]
  end

-  # -> insert new task here
+  run_code(x, 'commits 1') do
+    git.commits.size
+  end

+  run_code(x, 'commits 2') do
    log = git.commits('master', 15)
    log.size

diff --git a/tests.rb b/tests.rb
... ETC .....
```

modificaciones en benchmarks.rb

rango de líneas con cambios

lineas no modificadas

líneas eliminadas empiezan por - menos  
líneas añadidas empiezan por + mas

lineas no modificadas

modificaciones en el fichero tests.rb

# Consultar Cambios: Staged

- El comando `git diff` con la opción `--cached` o la opción `--staged` muestra las modificaciones que han sido staged para añadir en el próximo commit.

```
$ git diff --staged
diff --git a/benchmarks.rb b/benchmarks.rb
index 3cb747f..da65585 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -36,6 +36,10 @@ def main                                rango de líneas con cambios
    @commit.parents[0].parents[0].parents[0]
  end                                         líneas no modificadas

-    # -> insert new task here      líneas eliminadas empiezan por -
+    run_code(x, 'commits 1') do      líneas añadidas empiezan por +
+      git.commits.size
+    end

+    run_code(x, 'commits 2') do      líneas no modificadas
+      log = git.commits('master', 15)
+      log.size
```

# Historia de versiones: git log

- Mostrar la historia de versiones congeladas (commits):

```
$ git log
```

- Opciones:

**-NUMERO**

muestra los últimos NUMERO commits.

**--oneline**

muestra una línea por commit

**--stat**

muestra estadísticas

**--p**

muestra diferencias entre commits

**--graph**

muestra un gráfico ASCII con ramas y merges

**--decorate=auto**

muestra las referencias (ramas) de los commits listados

**--since=2.weeks**

muestra commits de las últimas 2 semanas

```
$ git log -2
```

```
commit 973751d21c4a71f13a2e729ccf77f3a960885682
Author: Juan Quemada <jquemada@dit.upm.es>
Date:   Mon Nov 21 18:17:13 2011 +0100
```

```
    rails generate controller Planet index contact
```

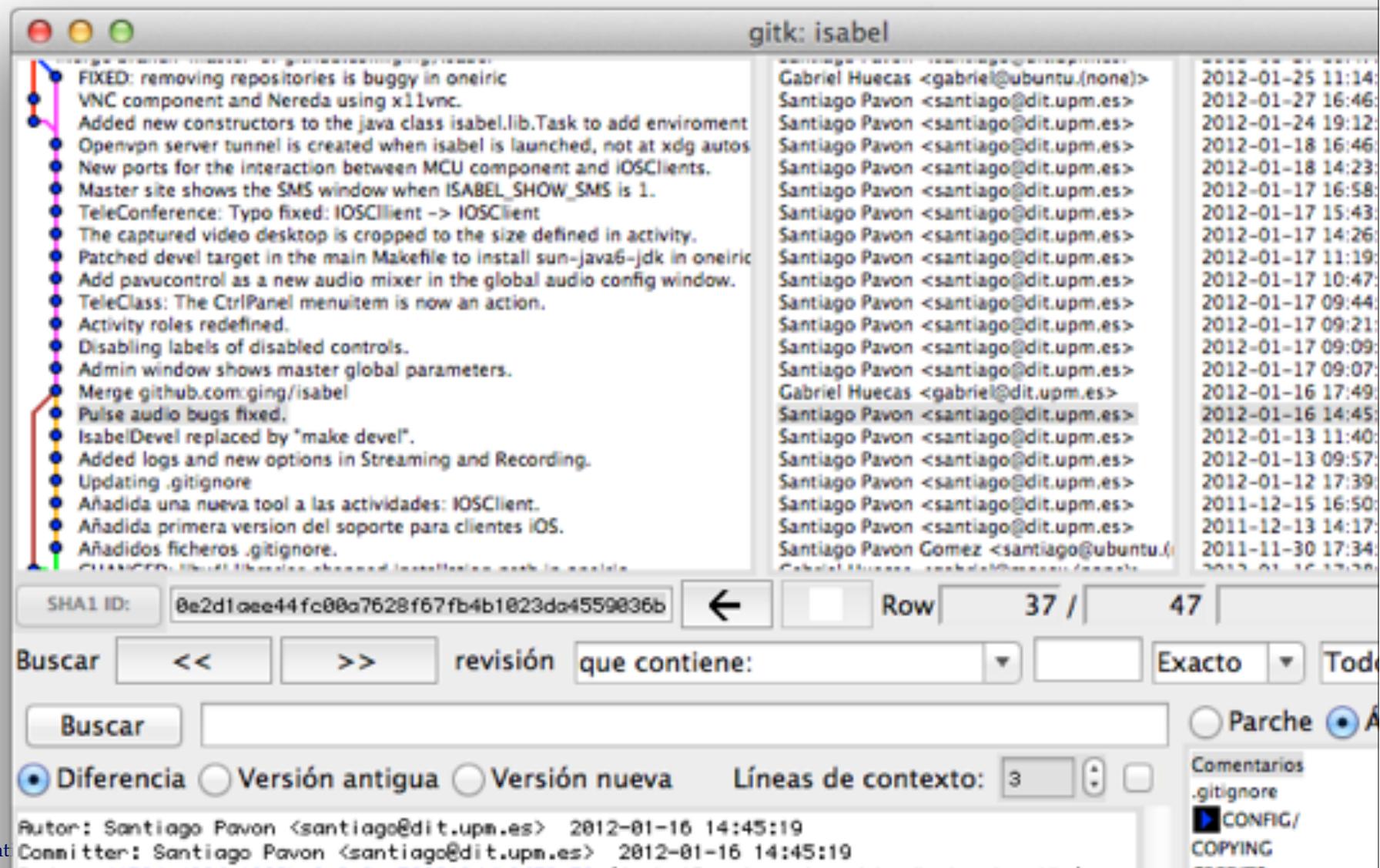
```
commit 27ec17607086f1bffd0695791a8fc263f87d405f
Author: Juan Quemada <jquemada@dit.upm.es>
Date:   Mon Nov 21 18:16:03 2011 +0100
```

```
    new planet creado
```

```
$ git log --oneline
```

```
973751d rails generate controller Planet index contact
27ec176 new planet creado
```

# Ver la historia de versiones gráficamente: gitk



# Git integrado en muchos IDEs

The screenshot shows a Java-based IDE (likely NetBeans) with a Git integration interface. The main window displays two files side-by-side: `main.js` and `c5be60ba06fc... (Read-only)`. The code in both files is identical, showing a readline interface for a command-line application. The IDE's navigation bar includes tabs for 'Side-by-side viewer', 'Ignore whitespaces and empty lines', 'Highlight words', and various icons. On the left, there's a project tree for 'CORE\_2018\_Practicas\_2\_3\_4' and a 'Log' panel showing commit history. The bottom of the screen features tabs for 'TODO', 'Version Control', 'Terminal', and 'Event Log'. A pink oval highlights the 'Version Control' tab.

```
const readline = require('readline');
const {log, biglog, errorlog, colorize} = require("./out");
const cmdss = require("./cmds");

// Mensaje inicial
biglog('CORE Quiz', 'green');

const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout,
    prompt: colorize("quiz > ", 'blue'),
    completer: (line) => {
        const completions = 'h help add delete edit list test p play credits'.split(' ');
        const hits = completions.filter((c) => c.startsWith(line));
        // show all completions if none found
        return [hits.length ? hits : completions, line];
    }
});
rl.prompt();
rl.on('line', (line) => {
    let args = line.split(" ");
    let cmd = args[0].toLowerCase().trim();

    switch (cmd) {
        case '':
            rl.prompt();
            break;
        case 'help':
            rl.prompt();
            break;
    }
});
```

```
const readline = require('readline');
const {log, biglog, errorlog, colorize} = require("./out");
const cmdss = require("./cmds");

// Mensaje inicial
biglog('CORE Quiz', 'green');

// Tabla de comandos:
const commands = {
    'help': cmdss.helpCmd,
    'h': cmdss.helpCmd,
    'quit': cmdss.quitCmd,
    'q': cmdss.quitCmd,
    'add': cmdss.addCmd,
    'list': cmdss.listCmd,
    'show': cmdss.showCmd,
    'test': cmdss.testCmd,
    'play': cmdss.playCmd,
    'p': cmdss.playCmd,
    'delete': cmdss.deleteCmd,
    'edit': cmdss.editCmd,
    'credits': cmdss.creditsCmd,
};

const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout,
    prompt: colorize("quiz > ", 'blue'),
    completer: (line) => {
        const completions = Object.keys(commands);
        const hits = completions.filter((c) => c.startsWith(line));
        // show all completions if none found
        return [hits.length ? hits : completions, line];
    }
});
```

Paso 4.1 - Convertir en un servicio origin & master  
Paso 3.3 - Comandos devuelven promesas.  
Paso 3.2 - Crear tabla de comandos y sustituir switch po if\_elseif\_else.  
Paso 3.1 - Usar BBDD en el modelo.  
Paso 2.15 - Implementar el comando play.  
Paso 2.14 - Implementar el comando test.  
Paso 2.13 - Añadir persistencia al modelo.  
Paso 2.12.2 - Añadir edición en el comando edit para que sea más eficiente.  
Paso 2.12.1 - Implementar el comando edit.  
Paso 2.11 - Implementar el comando delete.  
Paso 2.10 - Implementar el comando add.  
Paso 2.9 - Implementar el comando show.  
Paso 2.8 - Implementar el comando list.  
Paso 2.7.3 - Crear el módulo cmdss.  
Paso 2.7.2 - Crear el módulo out.  
Paso 2.7.1 - Crear el módulo model.

Santiago Pavón 3/2/18 10:06  
Santiago Pavón 3/2/18 15:42  
Santiago Pavón 3/2/18 14:24  
Santiago Pavón 3/2/18 13:28  
Santiago Pavón 2/2/18 17:01  
Santiago Pavón 2/2/18 16:46  
Santiago Pavón 2/2/18 16:38

In 7 branches: HEAD, master, practica\_3, practica\_4, origin/master, origin/practica\_3... [click to show all](#)

# Borrar Ficheros

- El comando **git rm** elimina un fichero en la próxima versión a congelar.

- Añadiendo este cambio al staging area.

```
$ git rm CharIO.java      Borra fichero del directorio de trabajo y del index.  
                           Tras el próximo commit dejará de estar tracked.
```

- La opción **--cached** se usa para no borrarlo del directorio de trabajo.

```
$ git rm --cached CharIO.java  Borra fichero del staging area,  
                           pero no lo borra del directorio de trabajo.  
                           Tras el próximo commit dejará de estar tracked.
```

- El comando /bin/rm borra ficheros del directorio de trabajo, pero no añade este cambio en el staging area.

- Es como hacer una modificación en el contenido del fichero.
- Debe usarse git add o git rm para meter en el staging area esta modificación.

```
$ rm CharIO.java          Borra el fichero de directorio de trabajo,  
                           pero este cambio aun no ha sido staged.
```

- El comando git rm falla si se intenta borrar un fichero con modificaciones en el directorio de trabajo o en el staging area.

- Para no perder de forma accidental modificaciones realizadas.
- Usar la opción -f para forzar el borrado.

# Renombrar Ficheros

- El comando **git mv** se usa para mover o renombrar un fichero en la próxima versión a congelar.

```
$ git mv filename_old filename_new
```

- Se implementa internamente ejecutando **git rm** y **git add**.
- El comando anterior es equivalente a ejecutar:

```
$ mv filename_old filename_new  
$ git rm filename_old  
$ git add filename_new
```

# Deshacer operaciones realizadas: Modificar el último commit

- El comando **git commit --amend** se usa para rehacer el último commit realizado.
  - Para cambiar el mensaje de log.
  - Para añadir una modificación olvidada
  - ...
- Ejemplo:
  - Creamos un commit:  
**\$ git commit -m 'Editr aca bado'**  
- pero hemos olvidado modificar un fichero, y el mensaje de log tiene algún error:
  - Realizamos los cambios olvidados, y los añadimos al staging area:  
**\$ git add forgotten\_file**
  - Ahora repetimos git commit pero con la opción --amend y un nuevo mensaje de log  
**\$ git commit --amend -m "Editor acabado"**
  - Este comando elimina el commit erróneo y se crea un nuevo commit.
- IMPORTANTE:
  - No usar la opción **--amend** sobre un commit que se haya subido a un repositorio público.
  - Podemos incomodar el trabajo realizado por otros desarrolladores que se base en el commit borrado.

# Deshacer operaciones realizadas: Eliminar Modificaciones Staged

- Comando para eliminar del staging area las modificaciones realizadas en un fichero es:

```
git reset HEAD <file>
```

- Ejemplo:
  - Modificamos readme.txt y añadimos los cambios en el staging area:  
`$ vi readme.txt`  
`$ git add readme.txt`  
- Pero nos arrepentimos, y queremos dar marcha atrás
  - Para cambiar el estado de readme.txt a unstaged ejecutamos:  
`$ git reset HEAD readme.txt`
  - Ahora:
    - readme.txt ya no está modificado en el staging area.
    - pero conserva sus modificaciones en el directorio de trabajo.

# Deshacer operaciones realizadas: Eliminar Modificaciones en el Directorio de Trabajo

- El comando para eliminar las modificaciones realizadas en un fichero del directorio de trabajo, y dejarlo igual que la versión del repositorio es:

```
$ git checkout -- <file>
```

- Ejemplo:
  - Modificamos el fichero readme.txt.  

```
$ vi readme.txt
```

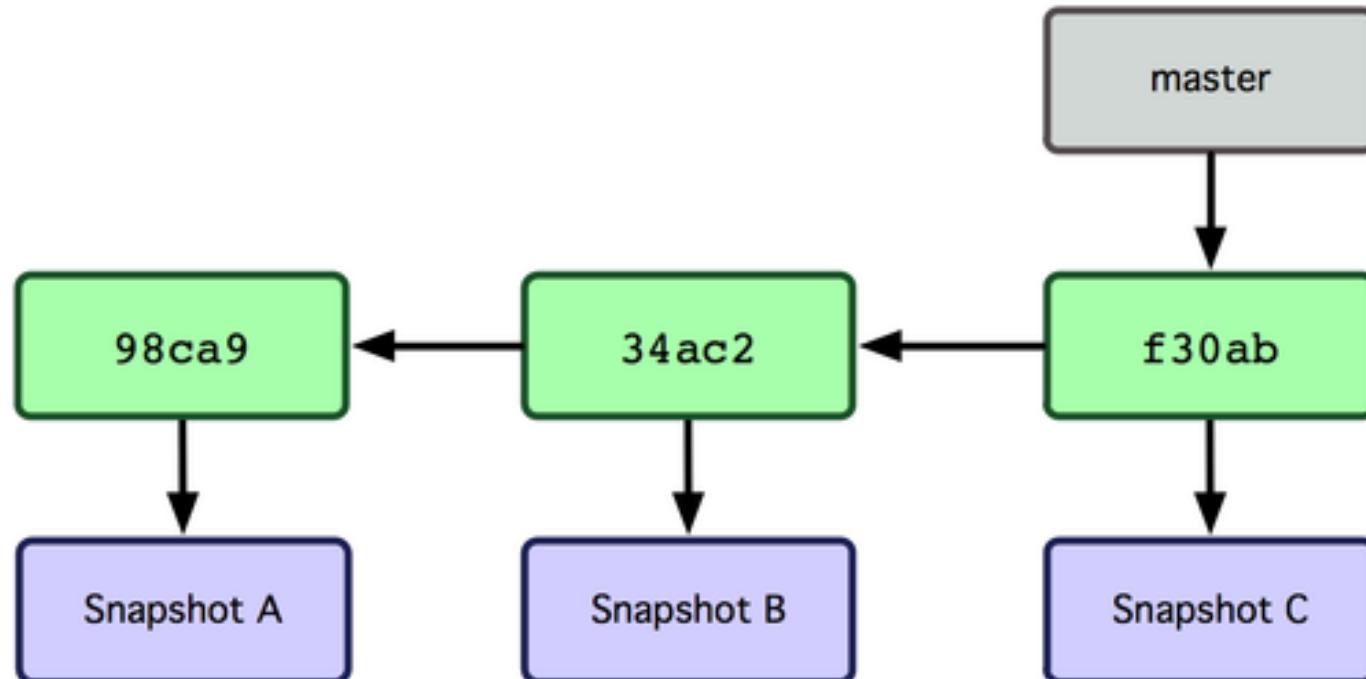
- pero nos arrepentimos de los cambios realizados.
  - Para restaurar el fichero a su estado original ejecutamos:  

```
$ git checkout -- readme.txt
```

# Ramas

# ¿Qué es una Rama?

- Una rama es un puntero a un commit.
  - Este puntero se reasigna al crear nuevos commits.
- En GIT suele existir una rama principal llamada **master**.



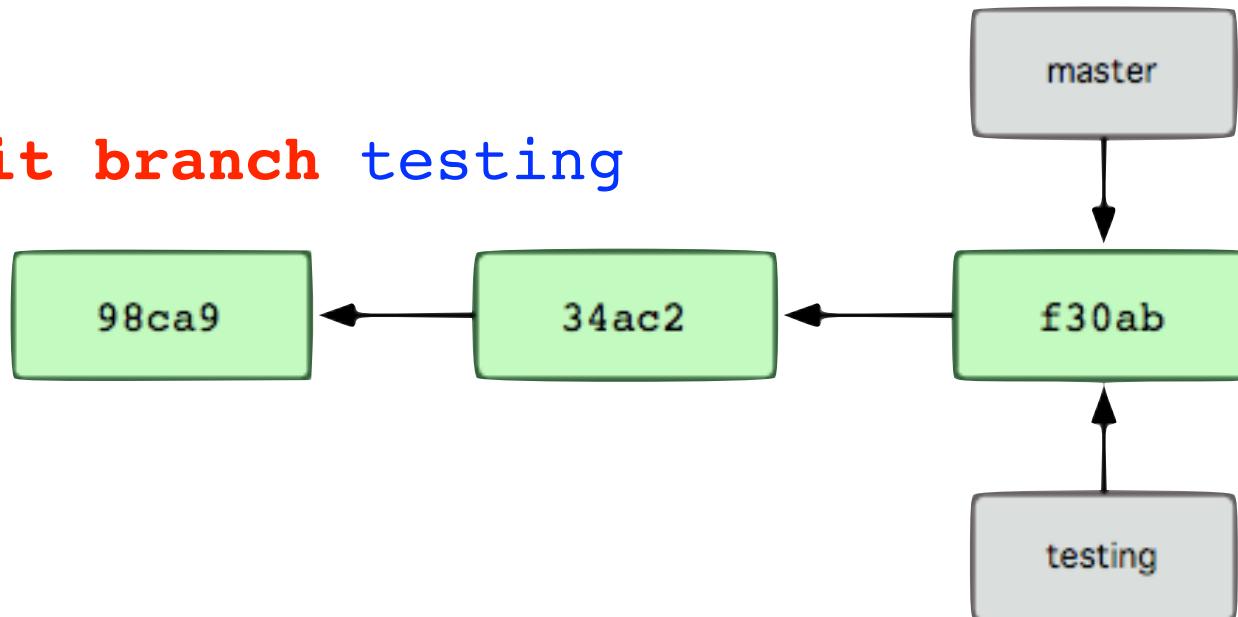
\*de Scott Chanson: <http://git-scm.org/book>

- Para crear nuevas ramas:

```
$ git branch <nombre>
```

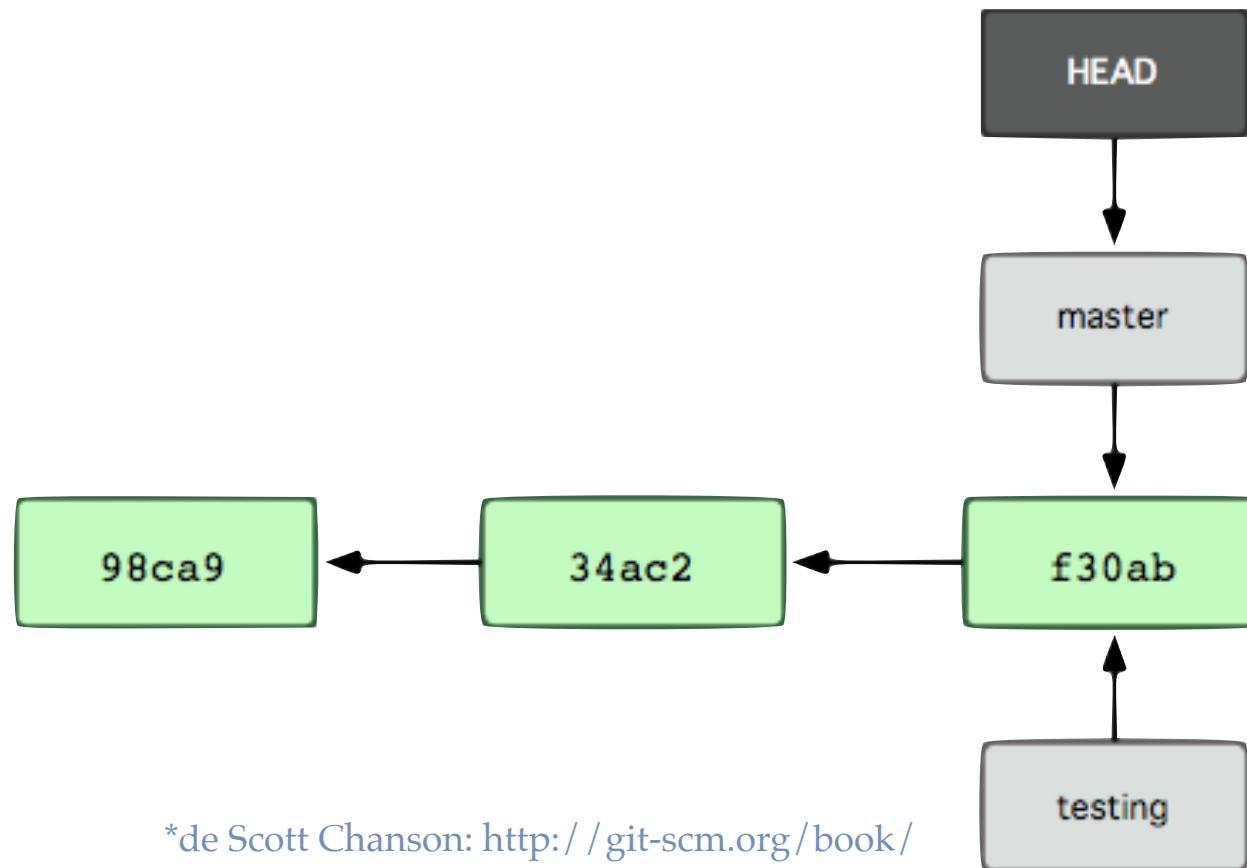
- Se crea un nuevo puntero.

```
$ git branch testing
```



\*de Scott Chanson: <http://git-scm.org/book/>

- Existe una referencia llamada **HEAD** que apunta a la rama actual
  - que apunta al commit sobre el que trabajamos.
    - los ficheros del directorio de trabajo y del staging area están basados en este commit.
  - Los nuevos commits se añaden al commit apuntado por **HEAD**.

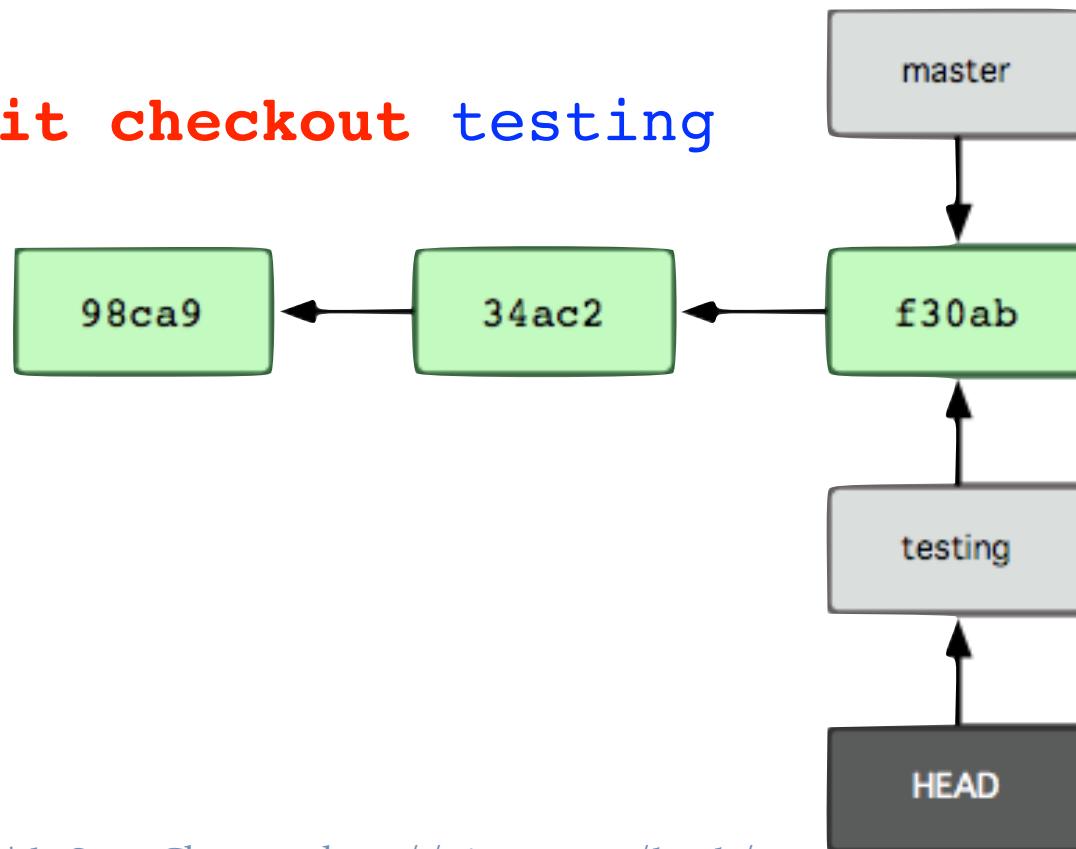


- Para cambiar de rama actual:

```
$ git checkout <nombre_rama>
```

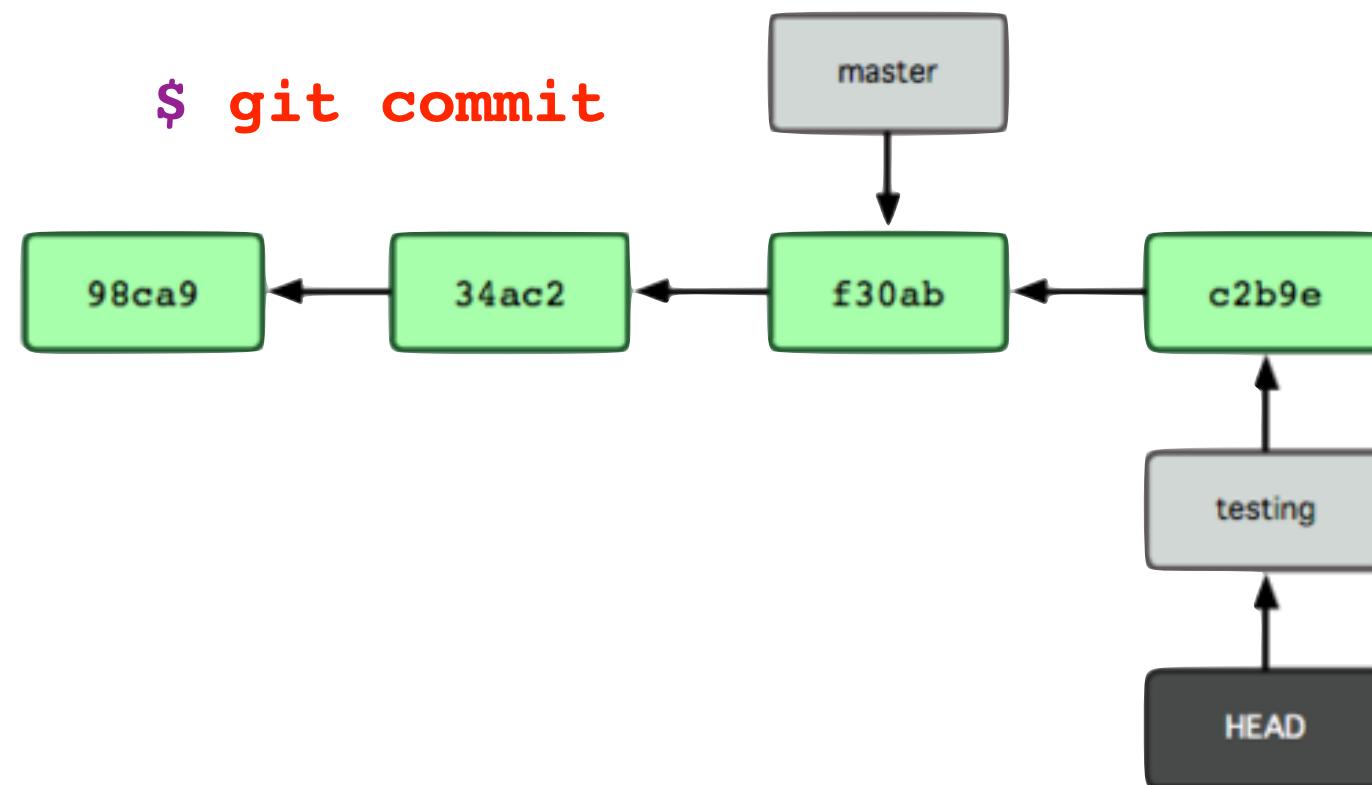
- HEAD apuntará a la nueva rama.
- Se actualiza el directorio de trabajo y el staging area.

```
$ git checkout testing
```



\*de Scott Chanson: <http://git-scm.org/book/>

- Al hacer commit avanza la rama apuntada por HEAD.

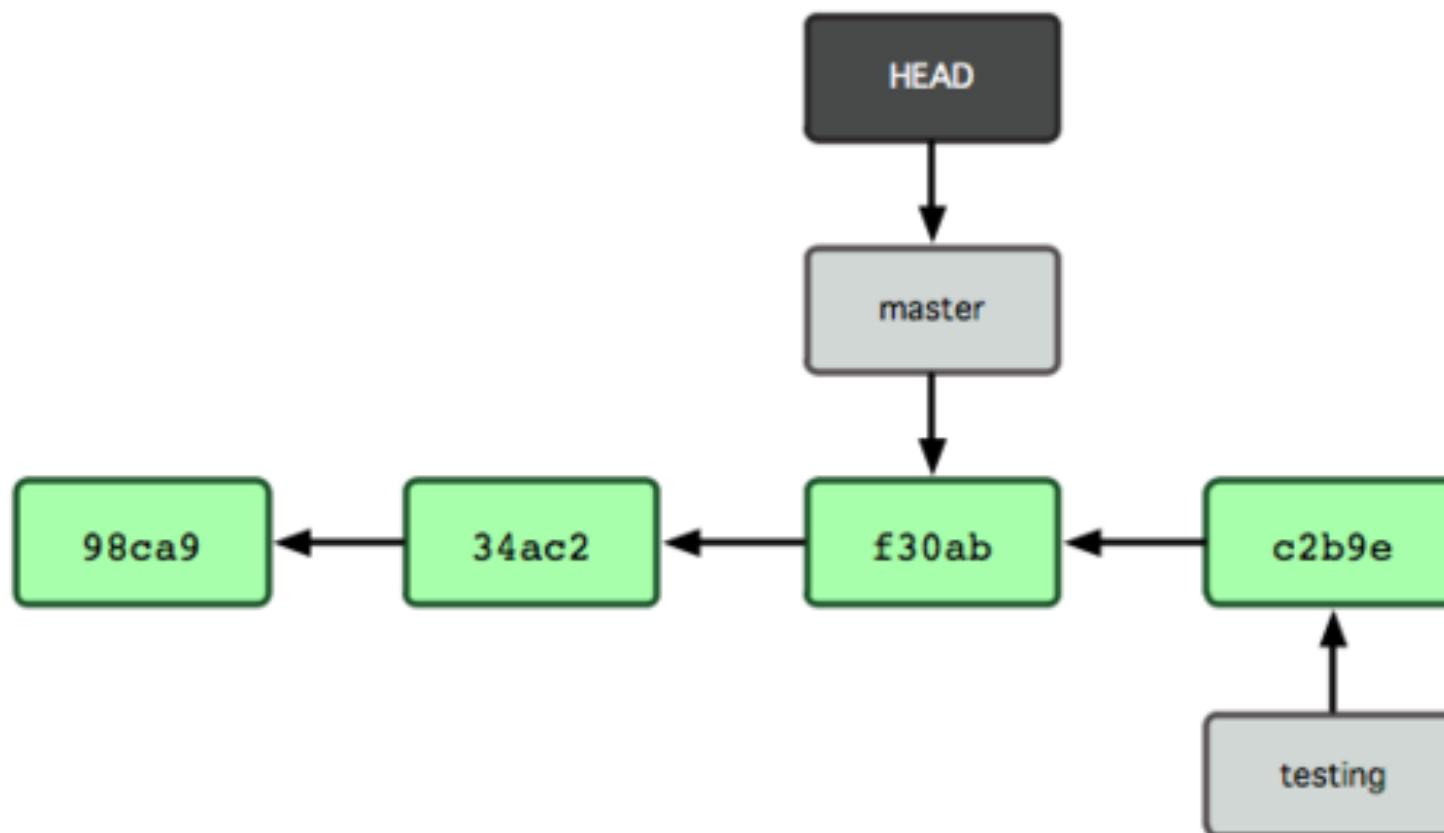


\*de Scott Chanson: <http://git-scm.org/book/>

- Para volver a la rama master:

**\$ git checkout master**

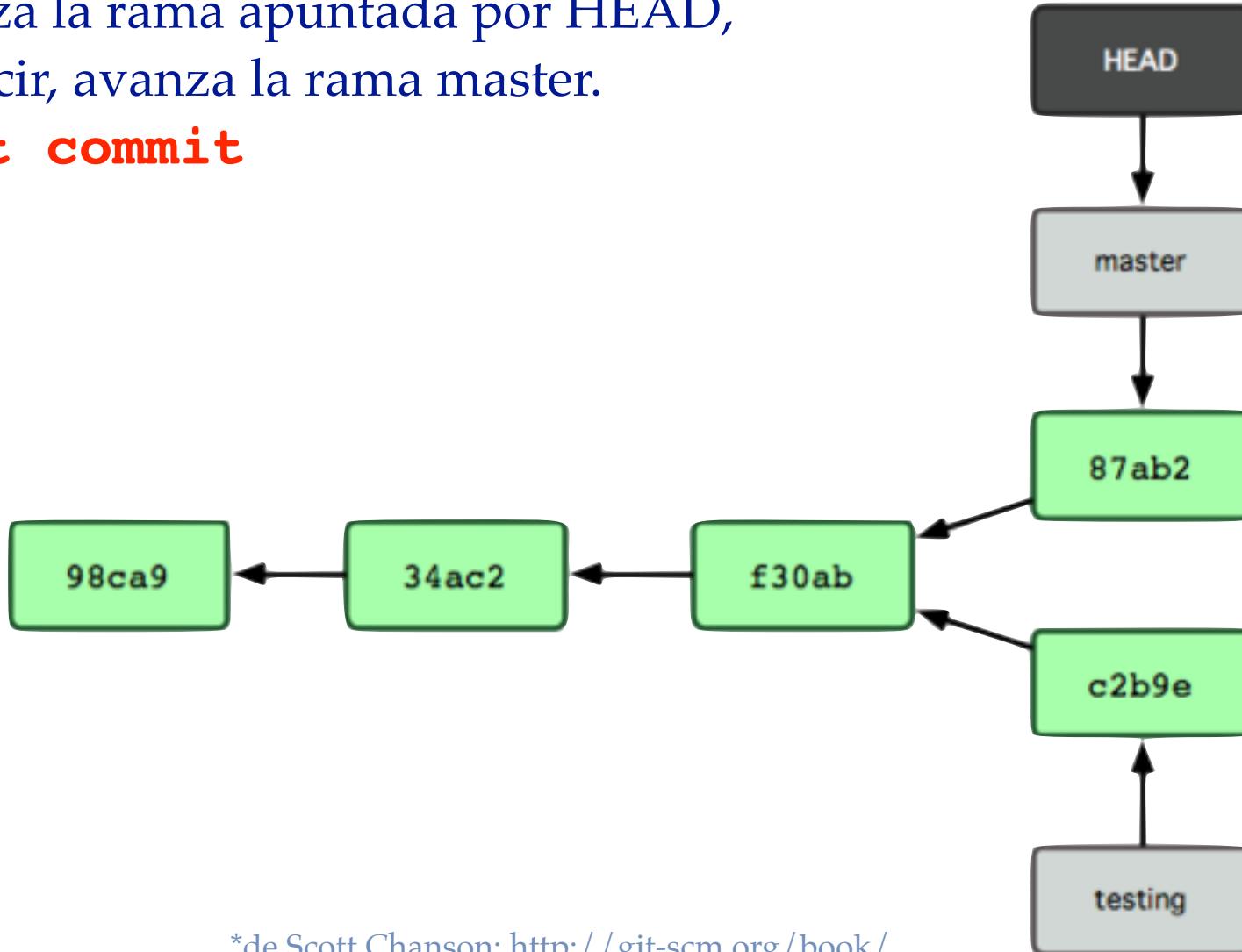
- HEAD apuntará a la nueva rama.
- Se actualiza el directorio de trabajo y el staging area.



\*de Scott Chanson: <http://git-scm.org/book/>

- Si hacemos un nuevo commit:
  - avanza la rama apuntada por HEAD,  
es decir, avanza la rama master.

**\$ git commit**



\*de Scott Chanson: <http://git-scm.org/book/>

# Crear Ramas y Cambiar de Rama

- Para crear una rama nueva en el punto de trabajo actual:  
`$ git branch <nombre>`
- Para crear una rama nueva situada en un commit dado:  
`$ git branch <nombre> <commit>`
- El comando `git checkout -b` permite crear una rama y cambiarse a ella con un solo comando:

```
$ git checkout -b <nombre>
```

# Cambiar de Rama

- Para cambiar de rama:

```
$ git checkout <nombre>
```

- La opción **-b** se usa para crear una rama y cambiarse a ella:

```
$ git checkout -b <nombre>
```

- Checkout falla si existen cambios en los ficheros del directorio de trabajo o en el staging area que no están incluidos en la rama a la que nos queremos cambiar.
  - Podemos forzar el cambio usando la opción **-f**.
    - perderemos los cambios realizados
  - Podemos usar la opción **-m** para que nuestros cambios se mezclen con la rama que queremos sacar
    - Si aparecen conflictos, los tendremos que solucionar.

# Más usos de git checkout

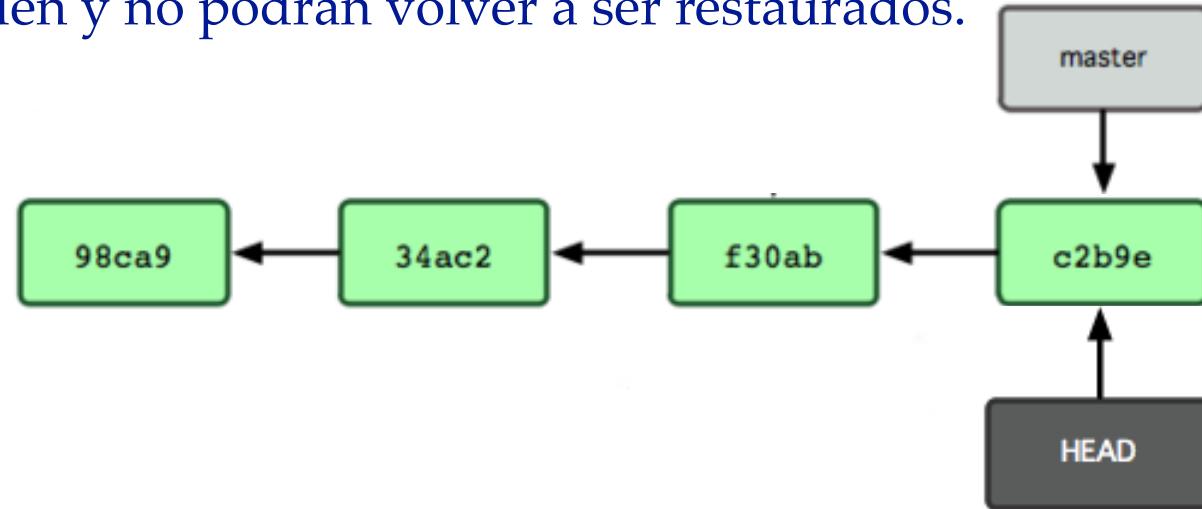
- Deshacer todos los cambios staged en el directorio de trabajo:

```
$ git checkout .
```

- Deshace los cambios staged de <fichero> en directorio de trabajo:

```
$ git checkout -- <fichero>
```

- Los cambios se pierden y no podrán volver a ser restaurados.

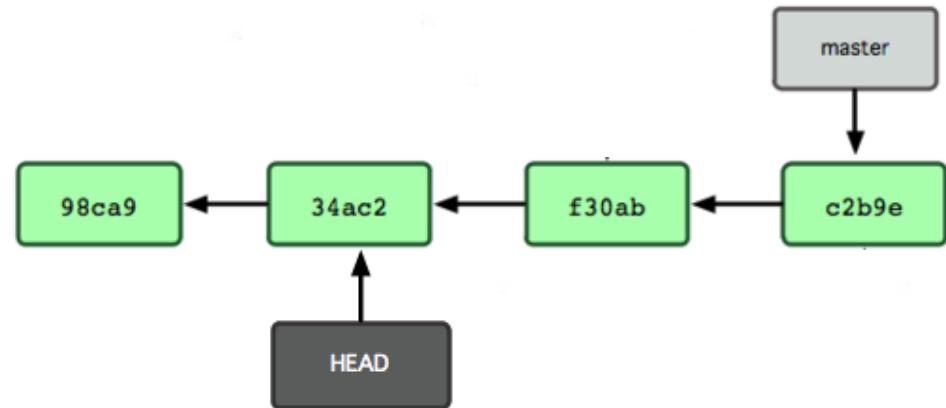


\*de Scott Chanson: <http://git-scm.org/book>

# Detached HEAD Branch

- Es una rama apuntada por HEAD, pero que no tiene un nombre de rama apuntándole.
- Aparecen cuando se realiza checkout directamente sobre el identificador sha1 de un commit.

```
$ git checkout 34ac2
```



- Suele hacerse para inspeccionar en un momento dado los ficheros en un punto de la historia, pero no queremos crear una rama que no vamos a usar.
- En cualquier momento podemos crear una rama con nombre ejecutando:

```
$ git checkout -b <nombre_rama>
```

# Integrar Ramas

- Para incorporar en la rama actual los cambios realizados en otra rama:

```
$ git merge <rama>
```

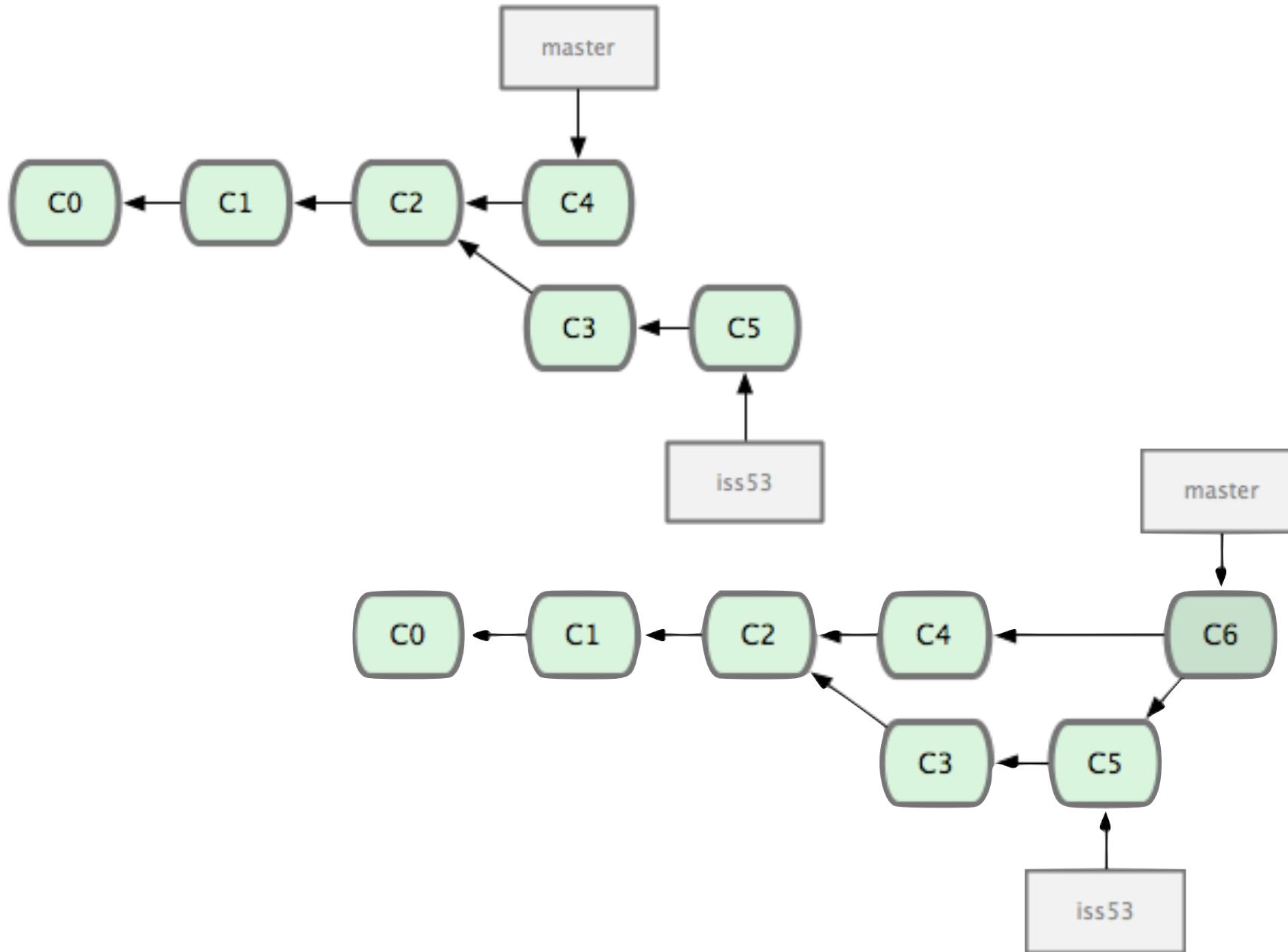
- Internamente GIT analiza la historia de commits para calcular como hacer la integración de las ramas.
  - Puede hacer un fast forward, una mezcla recursiva, ...
- Ejemplo:

```
$ git checkout master
```

- Estamos en la rama master.

```
$ git merge iss53
```

- Incorporamos los cambios hechos en la rama iss53 en la rama master.



\*de Scott Chanson: <http://git-scm.org/book/>

# Conflictos

- Al hacer el **merge** pueden aparecer conflictos
  - si las dos ramas han modificado las mismas líneas de un fichero.
- Si hay conflictos:
  - No se realiza el commit.
  - Las zonas conflictivas se marcan:

```
<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=====
<div id="footer">contact us at support@github.com</div>
>>>>> iss53:index.html
```
  - El comando **git status** lista los ficheros con conflictos como **unmerged**.
- Para resolver el conflicto hay que:
  - editar el fichero para resolver el conflicto.
  - y después ejecutar **git add** y **git commit**.

# Borrar Ramas

- Una vez terminado el trabajo con una rama
  - se borrar con el comando:  
**\$ git branch -d <nombre>**
    - Lo que se elimina es el puntero al commit.
- Si la rama a borrar no ha sido integrada con otra rama
  - se muestra un mensaje de error y no se borra.
  - Para borrar la rama independientemente de si ha sido integrada o no, usar la opción **-D** en vez de **-d**.

**\$ git branch -D <nombre>**

# Listar Ramas

- Para listar las ramas existentes:

```
$ git branch
```

- Ejemplo:

```
$ git branch  
iss53  
* master  
testing
```

- La rama activa se marca con un asterisco.
- Opciones:
  - **-r** muestra ramas remotas
  - **-a** muestra todas las ramas (locales y remotas)
  - **-v** muestra el último commit de la rama.
  - **--merged** muestra las ramas que ya se han mezclado con la rama actual.
  - **--no-merged** muestra las ramas que aun no se han mezclado con la rama actual.

# Identificar Commits

# Identificar Commits

- Los objetos almacenados por GIT se identifican con un SHA1
  - commits, trees, blob
- Para referirse a los commits puede usarse:
  - su valor SHA1
  - un prefijo del valor SHA1
    - que no coincide con otro SHA1 existente
  - un tag que apunte al commit.
  - el nombre de una rama.
  - referencia a un antepasado.
  - expresiones de rangos.
  - valores guardados en reflog.

# Referencias a Antepasados

- Los objetos commits apuntan a los commits en los que se basan.
  - El primer commit no tiene padre. Los demás commits tienen un commit padre, excepto si se han creado con una operación merge.
- Para acceder a los commits a los que apunta un commit C:
  - $C^$  o  $C^1$  commit padre en su misma rama cuando se creó.
  - $C^2$  primer commit usado en el merge.
  - $C^3$  siguiente commit usado en el merge.
- Para retroceder más en la cadena de commits:
  - $C\sim 1$  commit padre.
  - $C\sim 2$  commit abuelo.
  - $C\sim 3$  commit bisabuelo.
- Así,  $C\sim 3^2$  es el primer tío del bisabuelo del commit.

# Rangos de Commits

- Para especificar un rango de commit pueden usarse las siguientes notaciones:
  - **C1..C2** (*dos puntos*)
    - Todos los commits accesibles desde **C2** eliminando los que son accesibles desde **C1**.
  - **C1...C2** (*tres puntos, diferencia simétrica*)
    - Commits accesibles desde **C1** o desde **C2**, pero no desde ambos.
    - La opción **--left-right** marca con < y > los commits de cada lado.
      - En los casos anteriores, si no es especifica uno de los commits del rango se usa **HEAD**.
  - También pueden especificarse varios commits
    - Indican el rango de commit accesibles por cualquiera de los commits especificados.
    - Pueden excluirse todos los commits accesibles desde C añadiendo **^C** o **--not C**.

- Ejemplos:

```
$ git log master..prueba
```

```
$ git log ^master prueba
```

```
$ git log --not master prueba
```

- muestra un log de los commits de la rama prueba hasta que esta parte de master.

```
$ git log origin/master..HEAD
```

- logs con los cambios que he realizado desde la última vez que sincronicé con origin. Son los cambios que se subirán al ejecutar push.

# Tags

- Se usan para etiquetar commits importantes de la historia (o ficheros importantes).
  - Ejemplo de uso:
    - Poner etiquetas con un número de versión a los commits asociados a las versiones públicas del producto desarrollado.
- Podemos usar el nombre de un tag en todos los sitios donde se acepta un identificador de commit.
- Para listar todos los tags existentes:  
**\$ git tag**
- Para listar los tags que encajan con el patrón dado:  
**\$ git tag -l <patron>**
  - Ejemplo:  
**\$ git tag -l v1.\***  
v1.0  
v1.1  
v1.2

# Crear Tags

- Crear un tag ligero:

```
$ git tag <nombre> [<commit>]
```

- Asigna al commit dado un nombre de tag
  - Si no se proporciona un commit, se asigna al último commit.
- Se suele usar para etiquetar temporalmente un commit.

- Crear tag anotado:

```
$ git tag -a <nombre> [<commit>]
```

- Se crea un nuevo commit para el tag con toda la información que tienen los commit: mensaje, autor del tag, fecha, checksum, etc.
- Ejemplo:

```
$ git tag -a v1.4 -m "Version 1.4 de la aplicacion" 345ab1ac
```

- Crea un tag anotado con el nombre v1.4 y el mensaje dado para el commit 345ab1ac
  - Si no especificamos la opción -m se lanzará un editor.

- También se pueden crear tags firmados con GPG.

# Compartir Tags

- El comando **git push** no transfiere automáticamente los tag creados localmente a los repositorios remotos.
  - Para copiar un tag local en un repositorio remoto hay que indicarlo explícitamente en el comando push:

```
$ git push origin [tagname]
```

- Usando la opción **--tags** se transfieren todos los tags que existen en nuestro repositorio local.

```
$ git push origin --tags
```

# Repositorios Remotos

# ¿Qué es un Remote?

- En un proyecto GIT real suelen existir varias copias o clones del repositorio.
  - Cada desarrollador tendrá sus clones en los que trabajará.
- Un repositorio remoto es una de estas copias del repositorio alojada en algún sitio de la red.
  - Los desarrolladores bajan (**pull**) las contribuciones publicadas por otros desarrolladores en un repositorio remoto a su repositorio local.
  - Y suben (**push**) sus propias contribuciones a los repositorios remotos (de tipo bare).
- Para referirse a un repositorio remoto puede usarse:
  - su URL
  - o un remote: es un nombre corto que referencia la URL del repositorio remoto.
    - Es más cómodo que usar la URL.

- Cuando clonamos un repositorio se crea automáticamente un **remote** llamado **origin** que apunta a la URL del repositorio que hemos clonado.
- Podemos crear en nuestro repositorio local un **remote** llamado **estable** que apunte a la URL del repositorio donde se publican las versiones estables del proyecto.
- Podemos crearnos tres **remotes** llamados **pepito**, **luisito** y **jaimito**, que apunten a las URL de los repositorios donde **Pepe**, **Luis** y **Jaime** (tres becarios) publican sus desarrollos para que nosotros los integremos.

# Ver los Remotes Existentes

- El comando **git remote** lista los nombres de los remotes definidos.

```
$ git remote
```

```
bakkdoor
```

```
cho45
```

```
defunkt
```

```
koke
```

```
origin
```

- Usar la opción **-v** para ver las URL de los repositorios remotos.

```
$ git remote -v
```

```
bakkdoor  git://github.com/bakkdoor/grit.git
```

```
cho45      git://github.com/cho45/grit.git
```

```
defunkt    git://github.com/defunkt/grit.git
```

```
koke       git://github.com/koke/grit.git
```

```
origin     git@github.com:mojombo/grit.git
```

- Al clonar un proyecto se creó automáticamente un **remote** llamado **origin**.

# Crear Remotes

- Para crear un remote se usa el comando:

```
$ git remote add <shortname> <URL>
```

- shortname es el nombre corto que damos al remote
- URL es la URL del repositorio remoto
- Ejemplos:

- Crear un remote:

```
$ git remote add pepito git://github.com/pepe/demo.git
```

- Consultar los remotes existentes:

```
$ git remote -v
```

```
pepito    git://github.com/pepe/demo.git
```

```
origin    santiago@github.com:santiago/demo.git
```

- Ahora podemos usar el nombre pepito en vez de la URL del repositorio.
- Para descargar las últimas actualizaciones del repositorio:

```
$ git fetch pepito
```

- Para subir a origin mis cambios en la rama master:

```
$ git push origin master
```

# Más comandos sobre remotes

- Inspeccionar detalles de un remote:

```
$ git remote show [nombre_del_remote]
```

- Muestra el URL del remote, informacion sobre las ramas remotas, las ramas tracking, etc.

- Renombrar un remote:

```
$ git remote rename nombre_viejo nombre_nuevo
```

- Borrar un remote:

```
$ git remote rm nombre_del_remote
```

- Para actualizar la información sobre los remotes definidos:

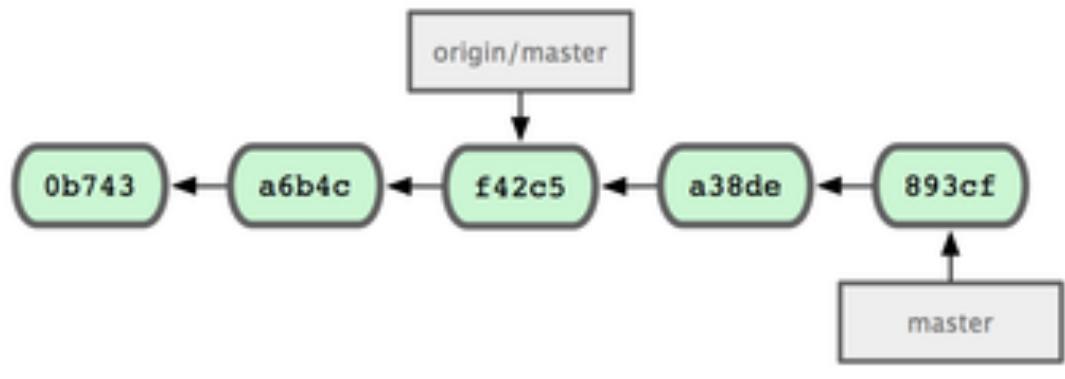
```
$ git remote update
```

- Para borrar ramas que ya no existen en el remote:

```
$ git remote prune
```

# Ramas Remotas

- Una rama remota es un puntero a un commit.
  - Indica cuál era la posición de una rama en un repositorio remoto la última vez que nos conectamos con él.
- Se nombran como: <remote>/<rama>.
- La figura muestra donde estaba la rama **master** en el repositorio **origin** la última vez que nos actualizamos.



- Este puntero no lo podemos mover manualmente.
  - Se mueve cuando actualizamos con el repositorio remoto.

# Tracking Branch

- Es una rama local emparejada con una rama remota para que estén sincronizadas.
  - Hacer un seguimiento de los cambios realizados en ellas
    - Al hacer **git push** en una **tracking branch** se suben los cambios locales y se actualiza la rama remota emparejada.
    - Al hacer **git pull** se actualiza la rama local con los cambios existentes de la rama remota emparejada.
- La rama **master** que se crea al clonar un repositorio es una **tracking branch** de **origin/master**.
- Para listar las tracking branches existentes:

```
$ git branch -vv
```

```
$ git remote show <remote_name>
```

- Para crear una tracking branch, ejecutaremos:

```
$ git checkout -b <branchname> <remotename>/<branchname>
```

- Se crea una rama local que hace el seguimiento de la rama remota indicada.
  - Nótese que el nombre local y remoto de la rama puede ser distinto.
- También podemos crear una tracking branch ejecutando:

```
$ git checkout --track <remotename>/<branchname>
```

- Se crea una rama local que hace el seguimiento de la rama remota indicada, usando el mismo nombre.

- Para subir una rama local a un remoto, y configurarla como una tracking branch, ejecutaremos:

```
$ git push -u <remotename> <branchname>
```

- La rama local copia en el remote usando el mismo nombre, y además se hace un seguimiento de ella.

# Descargar datos de un remote

- Bajarse los datos de un remoto:

```
$ git fetch [nombre_del_remote [refspec]]
```

- **refspec**:

- Indica las ramas local y remota entre las que se hará la bajada de datos.
- Puede ser el nombre de una rama (tanto local como remota).
- Si no es especifica este parámetro, se bajan las actualizaciones de todas las ramas locales que también existan en el repositorio remoto.

- Este comando actualiza el repositorio con los datos existentes en el remote, pero no modifica los ficheros del directorio de trabajo.
- Las operaciones de merge las deberemos invocar explícitamente.

- Ejemplo:

- Bajarse los datos que aun no tengo del repositorio del que me cloné:

```
$ git fetch origin
```

- Ahora mezclo mi rama actual con la rama demo de origin:

```
$ git merge origin/demo
```

# Descargar Versiones e Integrarlas

- Bajarse versiones de un remoto y aplicar merge:

```
$ git pull [nombre_remote [refspec]]
```

- Ejemplo:

- Descarga las versiones de la rama **demo** del remote **pepito** y las integra en nuestra rama actual:

```
$ git pull pepito demo
```

- Si la rama actual es una **tracking branch**, suele omitirse **refspec**:

- El comando **git pull nombre\_remote** actualiza la rama actual con los cambios realizados en la rama asociada del repositorio remoto.

```
$ git pull origin
```

(Actualiza rama actual con los cambios en origin)

```
$ git pull
```

(Por defecto realiza un pull de origin)

# Subir Versiones a un Remote

- Para subir nuestras versiones a un remote:

```
$ git push [nombre_remote [refspec]]
```

- La operación **push** sólo puede hacerse hacia repositorios **bare**.
  - Son repositorios donde no se desarrolla. Sólo se suben cambios. No tienen un directorio de trabajo.
  - Se crean con init o clone, y usando la opción **--bare**.
- Si la rama actual es una **tracking branch**, suele omitirse **refspec**:
  - El comando git push [nombre\_remote] sube las versiones desarrolladas en la rama local actual a la rama asociada del repositorio remoto.

```
$ git push origin master
```

(Subir los cambios en la rama master local a origin)

```
$ git push origin
```

(Subir los cambios de la rama local actual o origin)

```
$ git push
```

(Por defecto el remote es origin)

- Push sólo puede realizarse si se tienen permisos en el repositorio remoto.
- Solo se puede hacer push si estamos actualizados con los cambios ya existentes en el repositorio remoto.
  - Si en el remote hay actualizaciones que no tenemos, deberemos hacer un pull antes de poder subir nuestros cambios.
    - Para evitar subir actualizaciones que puedan producir conflictos.
- Si no es especifica un valor para refspec, se suben las actualizaciones de todas las ramas locales que también existan con el mismo nombre en el repositorio remoto.
- Si se crea una rama local, y se quiere subir al repositorio remoto, debe ejecutarse el comando push con el nombre de la rama como valor de refspec:

```
$ git push origin prueba
```

- El formato del parámetro refspec es <rama\_local>:<rama\_remota>
  - Permite usar nombres distintos para las ramas local y remota.
  - Por ejemplo, para subir los cambios de rama local **uno** a la rama **dos** del repositorio origin:
- Para borrar una rama remota, se usa un refspec con un nombre de rama local vacío, y con el nombre de la rama remota a borrar.:

```
$ git push origin :dos
```

# Modificar Commits

# Modificación de los Commits

- Comandos que permiten cambiar los commits existentes, aplicar los cambios de un commit en otro punto, eliminar los cambios introducidos en un commit, etc.

git **rebase**

git **reset**

git **cherry-pick**

git **revert**

git commit **--amend**

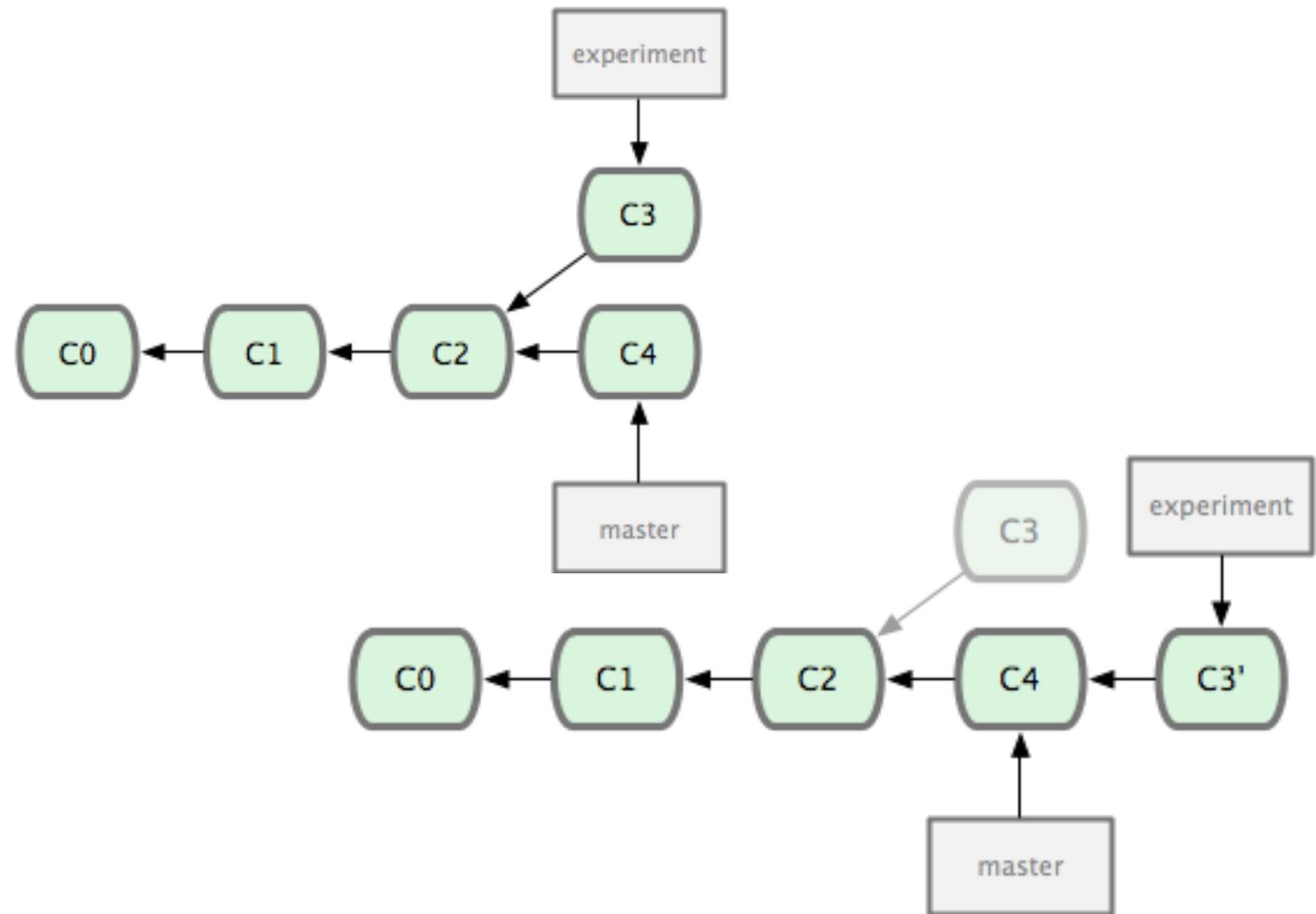
# Mover ramas con Rebasing

- Antes de nada: No hacer un rebase de un commit que ya se haya subido a un repositorio público.
  - Cambiar commits con los que ya están trabajando otros desarrolladores puede causarles problemas.
- Rebasing permite mover una rama de commits a otro sitio.

```
$ git rebase [basebranch] [topicbranch]
```

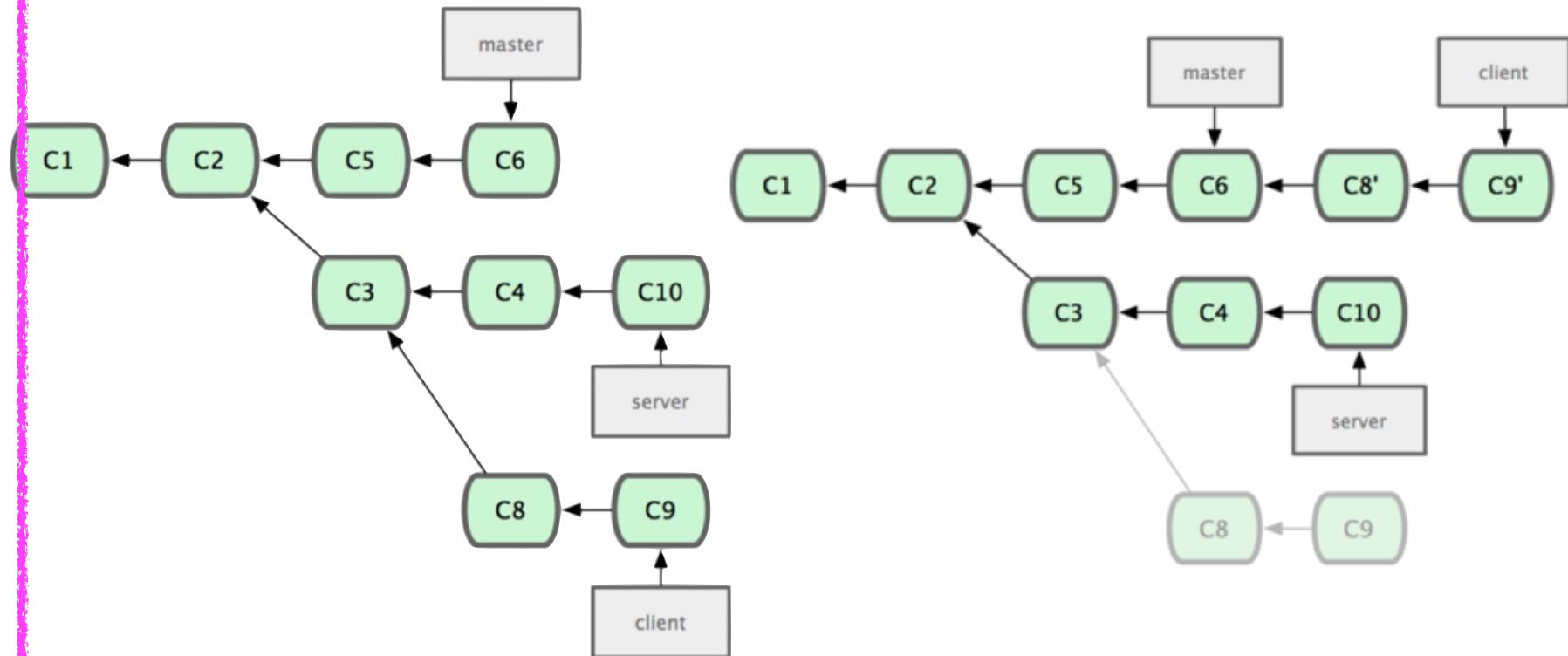
- Mueve los commits de la rama **topicbranch** a **basebranch**.
  - Busca un commit común entre **basebranch** y **topicbranch**, y partiendo de él, aplica los mismos cambios realizados en los commits a **topicbranch** en **basebranch**.
- Si se omite **topicbranch**, se usa la rama actual.

```
$ git rebase master experiment
```



```
$ git rebase --onto master server client
```

- Los commits de un rebase también pueden aplicarse sobre la rama que se desee usando la opción **--onto**.



- C8 y C9 son los commit realizados en la rama client que se creó partiendo de la rama base server.
- El comando anterior mueve esos commits a la rama master.

- Solucionar **conflictos** al hacer un rebase:

- Rebase aplica los commits uno a uno en la nueva localización.
- Si aparece un conflicto al intentar aplicar un commit, se detiene el rebase y debemos solucionar el conflicto manualmente.
  - Editamos los ficheros conflictivos para eliminar los conflictos.
  - Ejecutamos "**git add <ficheros conflictivos>**"
- Una vez solucionado el conflicto, ejecutamos:

**\$ git rebase --continue**

- Este comando aplica el commit pendiente y continua con los commits pendientes.
- Si al examinar el conflicto, decidimos que no queremos aplicar este commit, ejecutaremos:

**\$ git rebase --skip**

- se ignora el commit y se salta al siguiente commit del rebase.
- Si al final llegamos a un estado catastrófico en el que decidimos que no queremos hacer nada del rebase, y volver al estado inicial, ejecutaremos:

**\$ git rebase --abort**

- deshace todo lo hecho y deja el repositorio como estaba antes de invocar el **git rebase** inicial.

# Rebase Interactivo

- Antes de nada: **No hacer un rebase de un commit que ya se haya subido a un repositorio público.**
  - Cambiar commits con los que ya están trabajando otros desarrolladores puede causarles problemas.
- Rebase también se usa para editar commits de forma interactiva.

**\$ git rebase -i**

- Permite:
  - cambiar el orden de aplicación de los commits,
  - eliminar un commit,
  - unir varios commits en uno sólo,
  - partir un commit en varios, etc...

# Cambiar HEAD con git reset

- Para hacer que HEAD apunte al commit pasado como argumento:
- Opciones:

**\$ git reset --soft <commit>**

- No se borran los cambios registrados en el staging area.
  - El staging area también reflejará los cambios existentes entre las versiones apuntadas por el HEAD anterior y el nuevo HEAD.
- No se modifican los ficheros del directorio de trabajo.

**\$ git reset <commit>**

- Se restaura el staging area al nuevo estado.
  - Se pierden los cambios del staging area.
- No se modifican los ficheros del directorio de trabajo.

**\$ git reset --hard <commit>**

- Se restaura el staging area al nuevo estado.
- Se restaura el directorio de trabajo eliminando todos los cambios existentes.

- Ejemplos:
  - Volver al commit anterior (que HEAD apunte al ultimo commit realizado) pero sin perder ninguno de los cambios realizados

```
$ git reset --soft HEAD^
```

- Descartar todo los cambios metidos en el staging area.

```
$ git reset HEAD
```

- HEAD no cambia, se restaura el staging area y no se toca el directorio de trabajo.

- Descartar todos los cambios realizados desde el último commit.

```
$ git reset --hard HEAD
```

- HEAD no cambia, pero se restaura el staging area y el directorio de trabajo.

# Restaurar Index con git reset

- Restaurar en el staging area la versión de un fichero (o ficheros) según estaba en un determinado commit:

```
$ git reset <commit> -- <paths>...
```

- Copia los ficheros (paths) del commit indicado en el staging area.
- Es lo contrario que hace **git add**.

- Ejemplo:

- Eliminar los cambios del fichero **readme.txt** añadidos al staging area:

```
$ git reset HEAD -- readme.txt
```

# git cherry-pick

- Aplicar los cambios realizados en el commit indicado en mi rama, y crear un nuevo commit:

```
$ git cherry-pick <commit>
```

- Si aparecen conflictos, no se realiza el commit.
  - Hay que resolver los conflictos y hacer el commit a mano.
- Para aplicar este comando no deben existir cambios en los ficheros del directorio de trabajo ni en el staging area.
  - Si existen cambios sin congelar, el comando se queja y no hace nada.

# git revert

- Deshacer los cambios realizados en el commit indicado, y crear un nuevo commit:

```
$ git revert <commit>
```

- Es decir, deshace las modificaciones que se hicieron en un commit antiguo.
- Si aparecen conflictos, no se realiza el commit.
  - Hay que resolver los conflictos y hacer el commit a mano.
- Este comando es el inverso de **cherry-pick**.

# Stashing

# Stashing

- Cuando se está trabajando en una contribución, los ficheros del directorio de trabajo y los del staging area tienen las modificaciones realizadas hasta el momento.
- Si por alguna razón, necesitáramos hacer una modificación urgente relacionada con otra tarea, habría que sacar (checkout) los ficheros relacionados con la nueva tarea, hacer los cambios necesarios y congelar una versión.
  - Pero hay que evitar que se mezclen los cambios de ambas actividades.
- Stashing permite apartar nuestros fichero mientras hacemos la modificación, y recuperarlos más tarde, evitando mezclas indeseadas.
- Otra solución es clonar el repositorio en un directorio aparte, y realizar la modificación urgente en el nuevo clon.
  - Esto puede ser poco aceptable si el proyecto es muy grande.

- El comando **git stash** se usa para guardar todas nuestras modificaciones en una pila, y dejar limpios el directorio de trabajo y el staging area.
  - Una vez finalizada la modificación urgente, podemos aplicar todas las modificaciones guardadas en la pila de stash sobre la rama actual, recuperando así nuestro estado inicial.
    - Si el contenido de los ficheros de la rama actual ha cambiado, podrían aparecer conflictos al recuperar los cambios almacenados en la pila de stash.

- Para meter un stash en la pila,
  - es decir, para salvar en la pila las todas modificaciones existentes, y dejar limpios el directorio de trabajo y el staging area:

```
$ git stash
```

- Tras ejecutar este comando, **git status** informaría de que no existen modificaciones.
- Para listar el contenido de la pila de stashes:

```
$ git stash list
```

```
stash@{0}: WIP on master: 12bd442... Creado el defecto
```

```
stash@{1}: WIP on master: 9cab589... Optimiza busqueda
```

```
stash@{2}: WIP on master: 9cab589... Mejorar docs
```

- Cada stash guardado en la pila se identifica por el nombre **stash@{#}**.

- Para aplicar el último stash de la pila a la rama actual:

```
$ git stash apply
```

- Para aplicar un determinado stash a la rama actual:

```
$ git stash apply stash@{1}
```

- Observaciones:
  - El comando apply aplica los cambios del stash sobre la rama actual, que puede tener nuevos cambios sin congelar.
    - Pueden aparecer conflictos al aplicar un stash.
  - Por defecto, apply aplica sus cambios modificando sólo los ficheros del directorio del trabajo. No actualiza el staging area con los cambios que había en él.
    - Para que apply reinserte en el staging area los mismos cambios que este tenía originalmente, debe usarse la opción --index.
  - apply no elimina el stash aplicado de la pila.
- Para eliminar un stash de la pila:

```
$ git stash drop
```

```
$ git stash drop stash@{1}
```

- Para aplicar un stash y eliminarlo de la pila

```
$ git stash pop
```

```
$ git stash pop stash@{1}
```

# Servidor GITHUB



# GitHub

## ◆ Github → lema "Social coding"

- Red social donde programadores comparten repositorios remotos Git
  - Nos da acceso a ellos a través del navegador Web (además de Git)

## ◆ Repositorios **públicos son gratis**, los privados de pago

- Algunos proyectos libres en Github: Linux, Eclipse, jQuery, RoR, ...

## ◆ Este curso requiere tener cuenta en GitHub: <https://github.com>

- Al crearla nos da instrucciones claras y precisas sobre uso de GitHub y **Git**

## ◆ Otro repositorio se identifica en un repositorio local con un **URL**, p. e.

- <https://github.com/jquemada/cal> URL del rep. jquemada/cal en GitHub
- <https://github.com/jquemada/cal.git> con extensión .git explicita (equivalente)
- <git@github.com:jquemada/cal.git> URL Git (equivalente, poco utilizado)



# Funciones principales de GitHub

- ◆ La función principal de **GitHub** es **compartir** repositorios con terceros
- ◆ Las operaciones principales de un **usuario registrado** son
  - **Crear repositorio remoto** inicial nuevo para albergar un proyecto
    - Utilizando el botón: **New repository**
  - **Copia** un repositorio albergado en GitHub a otra cuenta (para contribuir)
    - Utilizando el botón: **Fork**
  - **Importa** un repositorio identificado por su URL a GitHub, incluso en otro formato
    - Utilizando el botón: **Import repository**
      - Equivale a crear repositorio vacío (New\_repository) e importar en él otro repositorio con un URL
  - **Crear una organización** para albergar múltiples proyectos relacionados
    - Utilizando el botón: **New organisation**
      - Organización de asignatura CORE: <https://github.com/CORE-UPM>
    - Y otras operaciones de compartición, gestión y mantenimiento
- ◆ Permite operaciones Git de **sincronización de repositorios bare**
  - **push** (subir rama), **clone** (clonar repositorio), **fetch** (traer rama), **pull** ..

# Tres formas de crear repositorios en GitHub

## ◆ Crear un repositorio vacío con **New\_repository**:

- <https://github.com/jquemada/cal>
  - GitHub lo crea en sus servidores invocando: `git init --bare`



## ◆ Copiar un repositorio a través de su URL con **Import\_repository**:

- [https://github.com/jquemada/cal\\_2com](https://github.com/jquemada/cal_2com)
  - El repositorio puede importarse de otro servidor en Internet o de GitHub, incluso cambiar el formato

## ◆ Copiar un repositorio con **Fork** a otra cuenta u organización:

- <https://github.com/CORE-UPM/cal>
  - se copia de la cuenta **jquemada** a la organización **CORE-UPM**

The screenshot shows a web browser window with the GitHub URL <https://github.com/jquemada/cal>. The browser's address bar and various bookmarks are visible at the top. A context menu is open over the repository name "jquemada / cal". The menu items shown are: New repository, Import repository, New gist, New organization, This repository, and New issue. The "New repository" item is highlighted with a pink box. The GitHub interface includes a header with "Pull requests", "Issues", and "Gist" tabs, and a main area with a message "No description, website, or topics provided." and buttons for "New" and "Add topics".

# Crear jquemada/cal vacío en GitHub

The screenshot shows the GitHub interface for creating a new repository. A callout box at the top left says "Crear nuevo repositorio vacío en GitHub". A dashed arrow points from the "New repository" button in the sidebar to the "Repository name" field. Another dashed arrow points from the "Create repository" button at the bottom right to the "Create repository" button in the main form.

**Crear nuevo repositorio vacío en GitHub**

New repository Import repository New gist New organization

Create a new repository

A repository contains all the files for your project, including

Owner Repository name

jquemada / cal

Nombre del repositorio

Repository público sin .gitignore, LICENSE y README

Public Anyone can see this repository. You choose who can commit to it.

Private You choose who can see and commit to this repository.

Initialize this repository with a README This will let you immediately clone the repository to your computer or repository.

Add .gitignore: None Add a license: None

Create repository

© Juan Quemada,

The screenshot shows the GitHub repository page for "jquemada/cal". A callout box at the top right says "Nuevo repositorio creado con URL: https://github.com/jquemada/cal". Below it, another callout box says "...or create a new repository on the command line". A third callout box at the bottom right says "Instrucciones de uso del repositorio.".

jquemada/cal

This repository Search

jquemada / cal

Nuevo repositorio creado con URL:  
<https://github.com/jquemada/cal>

Quick setup — if you've done this kind of thing before

Set up in Desktop or HTTPS SSH https://github.com/jquemada/cal.git

We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

...or create a new repository on the command line

```
echo "# cal" >> README.md  
git init  
git add README.md  
git commit -m "first commit"  
git remote add origin https://github.com/jquemada/cal.git  
git push -u origin master
```

Instrucciones de uso del repositorio.

...or push an existing repository from the command line

```
git remote add origin https://github.com/jquemada/cal.git  
git push -u origin master
```

...or import code from another repository

You can initialize this repository with code from a Subversion, Mercurial, or TFS

Import code

**Repository en GitHub I**

jquemada / cal

Code Issues 0

No description, website, or topics provided.

New Add topics

2 commits 1 branch 0 releases 1 contributor MIT

Branch: master New pull request

jquemada x^2 button

LICENSE README.md calculator.html

El proyecto: último commit de la rama master con los 3 ficheros indicados.

Hay 2 commits (versiones)

Clonar o descargar el repositorio en nuestro ordenador:  
-> crear un repositorio local.

new file Upload files Find file Clone or download

Latest commit b0e63ad 5 days ago

5 days ago 5 days ago 5 days ago

README.md

cal

Educational Git project. Creates a simple calculator in HTML and JavaScript in short steps.

Fichero README.md se ve aquí. Es muy conveniente incluirlo en un fichero en GitHub describiendo el proyecto o repositorio.

97

**Repository en GitHub II**

**jquemada / cal**

No description, website, or topics provided.

New Add topics

2 commits

Branch: master New pull request

jquemada x<sup>2</sup> button

LICENSE README.md calculator.html

calculator.html

Último commit de la rama master con los 3 ficheros indicados.

Readme Readme x<sup>2</sup> button

README.md

cal

Educational Git project. Creates a simple calculator in HTML

This repository Search Pull requests Issues

jquemada / cal

Code Issues Pull requests Projects

Branch: master cal / calculator.html

jquemada x<sup>2</sup> button

1 contributor

18 lines (15 sloc) 350 Bytes

```

1 <!DOCTYPE html><html><head>
2 <title>Calculator</title><meta charset="utf-8">
3 <script type="text/javascript">
4
5 function square() {
6   var num = document.getElementById("n1");
7   num.value = num.value * num.value;
8 }
9 </script>
10 </head>
11 <body>
12   Number:
13   <input type="text" id="n1"><p>
14
15   <button onclick="square()"> x<sup>2</sup> </button>
16 </body>
17 </html>

```

**Repository en GitHub III**

jquemada / cal

No description, website, or topics provided.

New Add topics

2 commits 1 branch 0 releases

Branch: master New pull request Create new file Upload files Find file Clone or download

jquemada x<sup>2</sup> button

LICENSE README.md calculator.html

README.md

cal

Educational Git project. Creates a simple calculator in HTML and JavaScript in short steps.

Branch: master

Commits on Feb 12, 2017

x<sup>2</sup> button jquemada committed 3 hours ago

Commits on Feb 9, 2017

Readme & License jquemada committed 3 days ago

Number: 2 x<sup>2</sup>

Fork Edit

Latest commit b0e63ad 5 days ago

5 days ago 5 days ago 5 days ago

99

© Juan Quemada, DIT, UPM

**Repository en GitHub III**

jquemada / cal

No description, website, or topics provided.

New Add topics

2 commits 1 branch

Branch: master New pull request

jquemada x<sup>2</sup> button

LICENSE README.md calculator.html

Verde: código añadido

No hay rojo, ni negro porque al ser un fichero nuevo solo se añade.

cal

Educational Git project. Creates a simple calculator in

Branch: master

Commits on Feb 12, 2017

x<sup>2</sup> button jquemada committed 3 hours ago

Commits on Feb 9, 2017

Readme & License jquemada committed 3 days ago

jquemada committed an hour ago 1 parent

Showing 1 changed file with 17 additions and 0 deletions.

```

17  calculator.html
...
@@ -0,0 +1,17 @@
1 +<!DOCTYPE html><html><head>
2 +<title>Calculator</title><meta charset="utf-8">
3 +<script type="text/javascript">
4 +
5 +function square() {
6 +  var num = document.getElementById("n1");
7 +  num.value = num.value * num.value;
8 +
9 +</script>
10 +</head>
11 +<body>
12 +  Number:
13 +  <input type="text" id="n1"><p>
14 +
15 +  <button onclick="square()"> x2 </button>
16 +</body>
17 +</html>

```

0 comments on commit 0e9f90a

Number: 2 x<sup>2</sup>

# Fork: clonar un proyecto de GitHub



Repositorio cal del usuario jquemada en:  
<https://github.com/jquemada/cal>

Copia el repositorio a otra cuenta u organización del usuario en GitHub pulsando el **botón Fork**.

Where should we fork this repository?

GING @ging

@CORE-UP...

Can't find what you're looking for?

You already have a fork of this repository:  
jquemada/cal\_2com

Unwatch 1 Star 0 Fork 0

No description, website, or topics

New Add topics

2 commits

Branch: master New pull request

jquemada x^2 button

Commits on Feb 12, 2017

x^2 button jquemada committed 3 hours ago

Commits on Feb 9, 2017

Readme & License jquemada committed 3 days ago

CORE-UPM / cal  
forked from jquemada/cal

Pull requests Issues Gist

Unwatch 6 Star 0 Fork 0

No description, website, or topics

New Add topics

2 commits

1 branch 0 releases

101

En el momento del **Fork** el repositorio **jquemada/cal** tiene estos **2 commits**. A partir de este momento cada repositorio evolucionara por separado a partir de estos 2 commits

Copia de jquemada/cal creada en la organización CORE-UPM. El nuevo repositorio estará accesible en: <https://github.com/CORE-UPM/cal>

# Otros Temas

# Enviar contribuciones usando Parches

- En algunos modelos de trabajo, los desarrolladores no tienen permiso para integrar sus contribuciones en el repositorio remoto principal.
- En estos casos, un escenario de trabajo podría ser así:
  - Los desarrolladores envían su contribuciones al integrador en forma de parches.
    - por e-mail, u otro medio.
  - El integrador aplica los parches recibidos en una rama de prueba, los prueba, y si son aceptados, los integra en el repositorio principal.
  - En este momento, las recién incorporadas contribuciones estarán disponibles para que todo el mundo se las descargue.

- Crea un parche para cada commit de la rama actual posterior al commit dado.

```
$ git format-patch <commit>
```

- Los parches son ficheros en formato mbox.
  - Se nombran con un número de secuencia y el texto del mensaje de log.
  - Estos ficheros pueden editarse:
    - para modificar los mensaje de log
    - para añadir instrucciones privadas
      - editar justo después de la primera línea ---
- Ejemplo:

```
$ git format-path origin/master
0001-Incluido-control-de-errores.patch
0002-Arreglada-la-documentacion.patch
```

- Se han creado dos parches. Desde la última vez que se sincronizó con origin se han realizado dos commits.

- Aplicar los parches incluidos en mbox a la rama actual.

```
$ git am <mbox>
```

- El fichero mbox pueden contener varios mensajes, cada uno con un parche.
- Si el parche se aplica con éxito,
  - automáticamente se hace un commit usando los datos contenidos en mbox para asignar el autor, la fecha y el mensaje de log.
- Si el parche falla, los ficheros afectados se habrán modificando señalando los conflictos existentes de la forma habitual.
  - En este punto hay que:
    - Editar los ficheros con conflictos para resolverlos.
    - Añadir los ficheros modificados al staging area usando **git add**.
    - Y ejecutar **git am --resolved**
  - Si decidimos que no queremos aplicar este parche:
    - Ejecutar **git am --skip** para saltarnos este parche y pasar al siguiente.
    - Si decidimos que queremos dar marcha atrás y no aplicar ningún parche:
      - Ejecutar **git am --abort**. Se vuelve al estado inicial.

# git archive

- Crear un fichero con el contenido del repositorio.
- Ejemplos:
  - Crear un fichero gzip con el contenido de la rama master

```
$ git archive master --prefix='proy' | gzip > pm.tgz
```

    - Se genera el fichero **pm.tgz**.
    - Al descomprimirlo los ficheros se encuentran bajo el directorio **proy**.
  - Especificar el formato de compresión usando la opción **--format**

```
$ git archive master --prefix='proy'  
--format=zip > pm.zip
```

# git describe

- Generar números de versión:

```
$ git describe <commit>
```

- El número de versión generado se basa en buscar el tag más cercano alcanzable desde el commit dado.
  - Si el tag encontrado apunta al commit dado, se muestra el valor del tag.
    - El tag existente sirve como número de versión.
  - Si el tag encontrado apunta a otro commit, se muestra el nombre del tag, el número de commits hasta el tag, y un prefijo SHA-1 del commit dado.
  - Si no se encuentra un tag, indica que no se encontró.
- Ejemplo:

```
$ git describe master  
v1.3-5-ab34ga33
```

# add interactivo

- Si hemos hecho varios cambios en los ficheros del directorio de trabajo
  - que pertenecen a contribuciones diferentes
  - y no queremos hacer un commit conjunto
- podemos hacer un add interactivo usando la opción **-i**:

```
$ git add -i
```

- Este comando muestra un menú que permite
  - meter y sacar los ficheros del staging area
  - ver cada uno de los cambios existentes en un fichero, y meter sólo alguno de ellos en el staging area.

```
$ git add -i
      staged      unstaged path
 1: unchanged          +0/-1 TODO
 2: unchanged          +1/-1 index.html
 3: unchanged          +5/-1 lib/simplegit.rb

*** Commands ***
 1: status      2: update      3: revert      4: add untracked
 5: patch       6: diff        7: quit        8: help
What now>
```

# Buscar un Commit

**\$ git bisect**

- Para buscar un commit en el que se metió un error. Se especifica un commit en el que el error no estaba, un commit en el que el error si estaba, y se procede a reducir ese rango hasta encontrar en que commit se introdujo el error.

**\$ git blame**

- Informa sobre quien fue el último en modificar cada línea de un fichero, y en que commit lo hizo.

**\$ git log -S<string> <filename>**

- Para buscar el string dado hacia atrás en todas las diferencias de un fichero.

# Submódulos

- Permite asociar subdirectorios con repositorios remotos de proyectos diferentes.
- Ver documentación para más detalles.

# Fontanería

- Comandos de bajo nivel:
  - **git cat-file**: Ver el contenido o el tipo y tamaño de un objeto.
  - **git ls-files**: Ver objetos en el index, directorio de trabajo, modificados, etc.
  - **git write-tree**: Crear un objeto index con el contenido actual del index.
  - **git rev-parse**: Dado un prefijo ID, u otro tipo identificador de revisión, calcula el ID completo al que se refiere.
  - **git hash-object**: Calcula el ID (y puede crear el blog) de un fichero.
  - **git merge-base**: se usa para buscar la base para hacer un merge
  - ...

# Temas Pendientes

# Temas pendientes

- Reflog
- Atributos
- Hooks
- ...