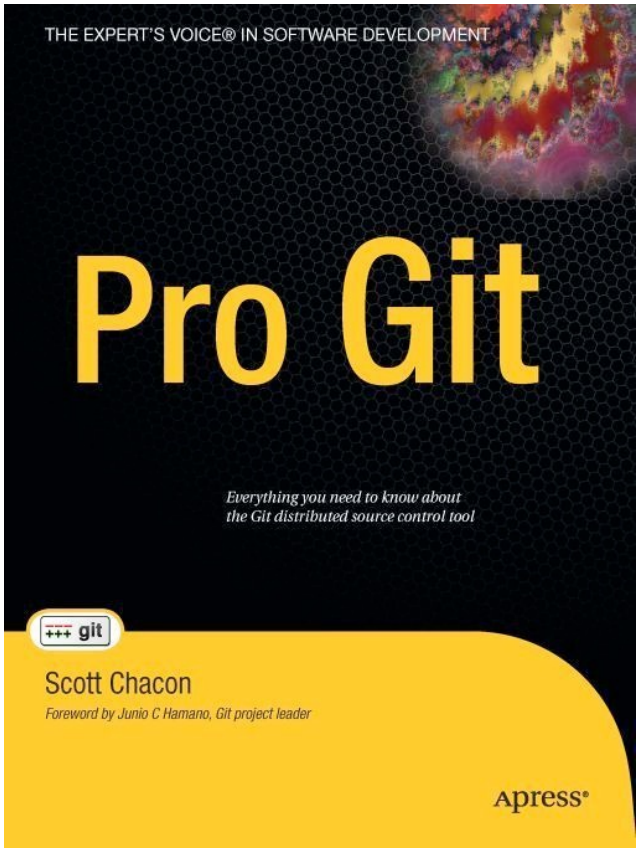
Gestión de Proyectos Software con Git y Github

Versión: 2013.04.06

Parte 1:
Introducción a GIT

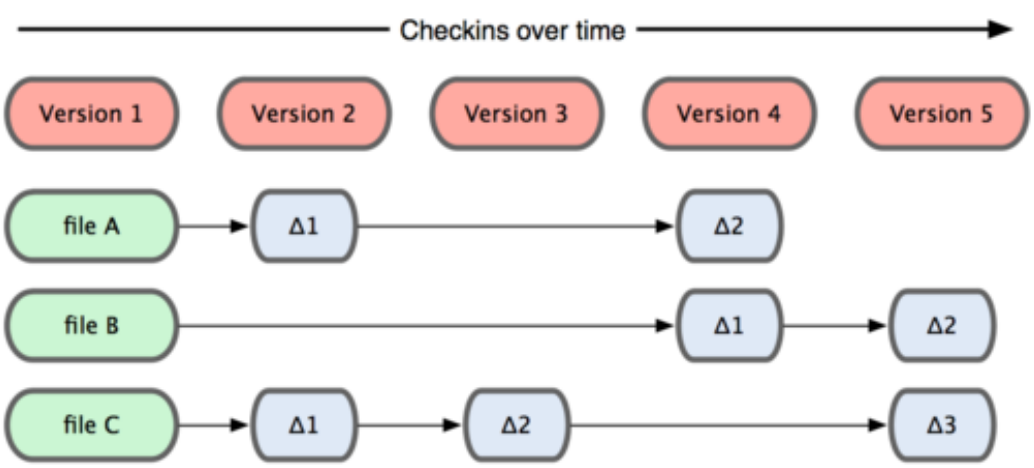
GIT

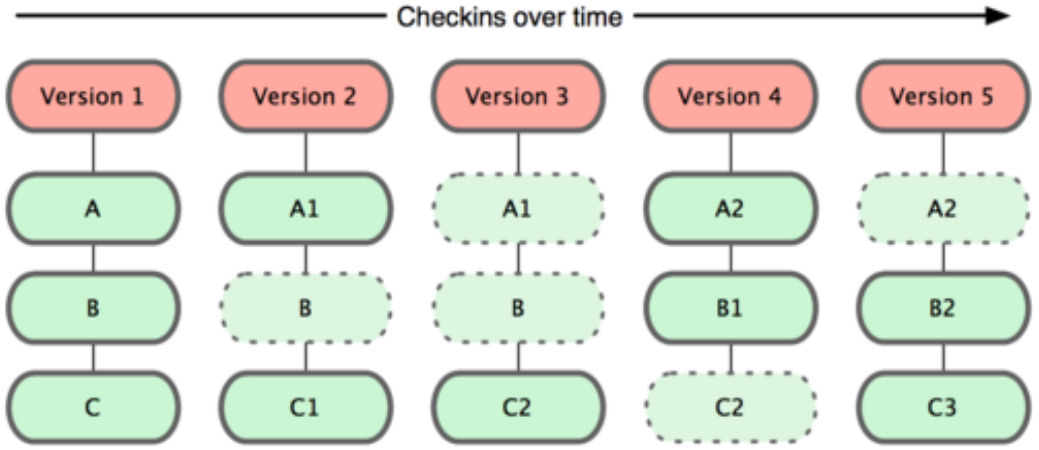
- GIT: gestor de versiones
  - Desarrollado por Linus Torwalds para Linux
    - Desarrollo colaborativo de proyectos
- Muy eficaz con proyectos
  - grandes o pequeños
- Tutorial Web y eBook gratis
  - http://git-scm.org
  - Otros:
    - http://gitref.org



Historia de un proyecto

- La historia de cambios de un proyecto es compleja
  - Se ordena por **versiones**
- Versión (Commit)
  - Punto de sincronización de un proyecto
- Cada versión consolida una nueva función (completa)
  - Conviene consolidar versiones a menudo
    - Con pocos cambios por versión

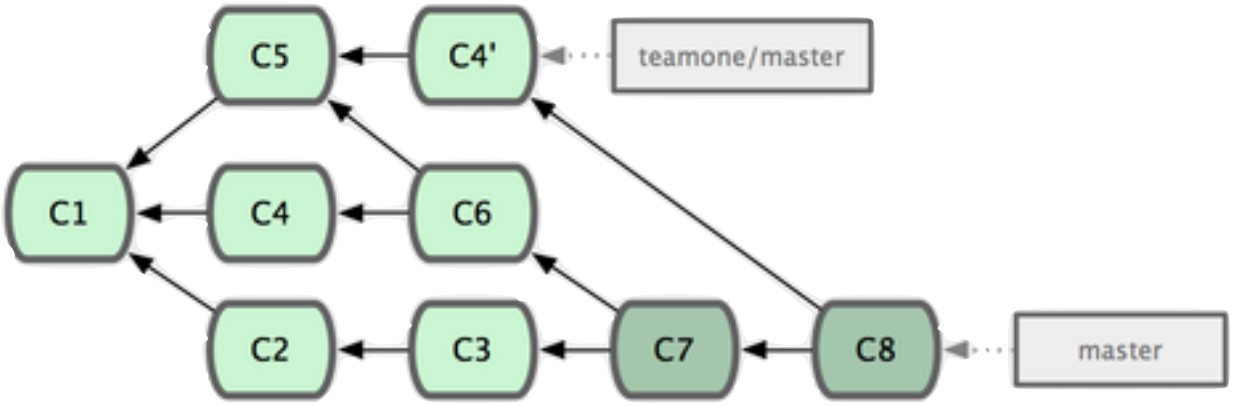




*de Scott Chanson: http://git-scm.org/book/

Árbol de desarrollo

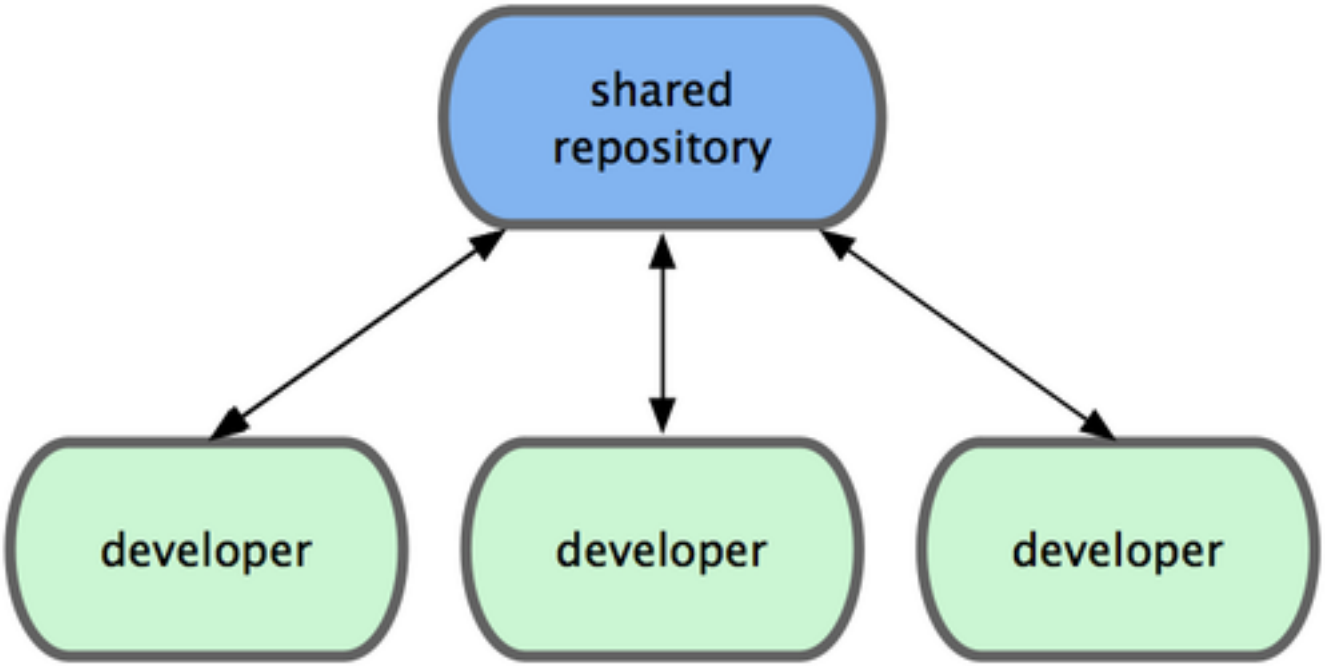
- Proyectos software en equipo son complejos.
- Múltiples desarrollos en paralelo.
  - nuevas funcionalidades, corrección de errores, mejoras, ...
- Cada desarrollo es una rama del árbol.
  - Ramas con cambios estables se integran (mezclan) en la rama principal.



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

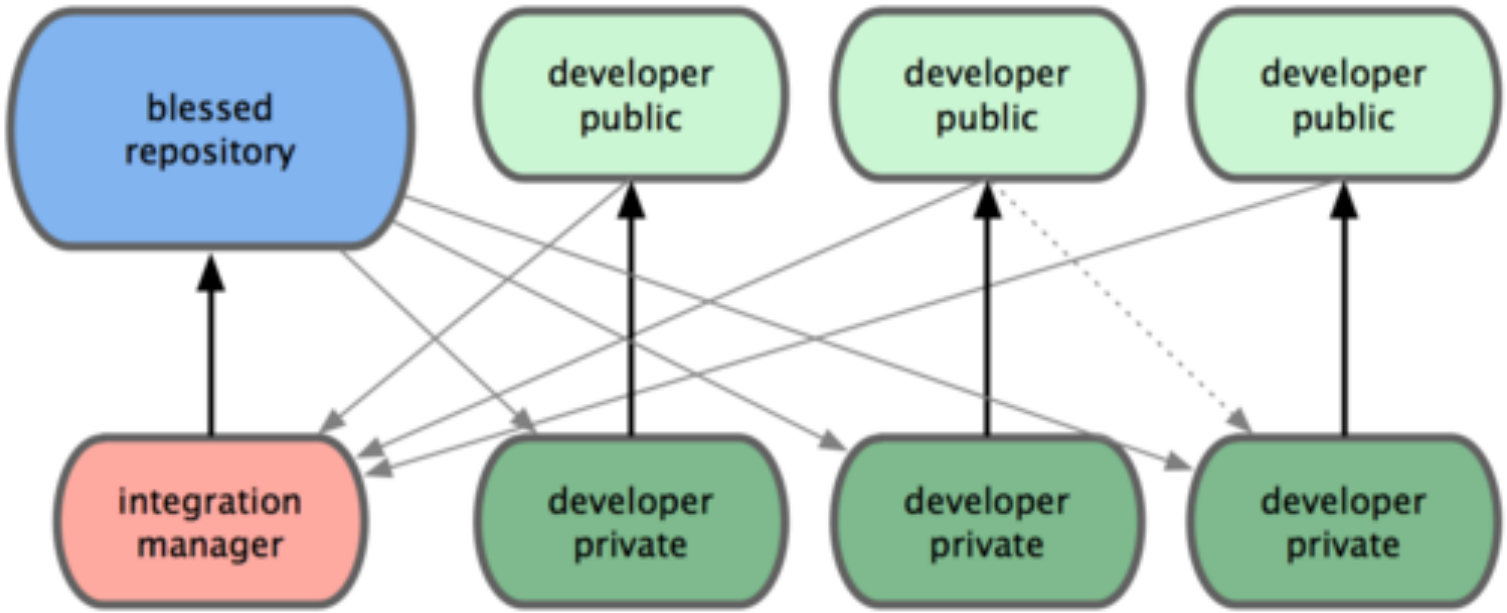
Políticas de Organización

- Flujo de trabajo centralizado



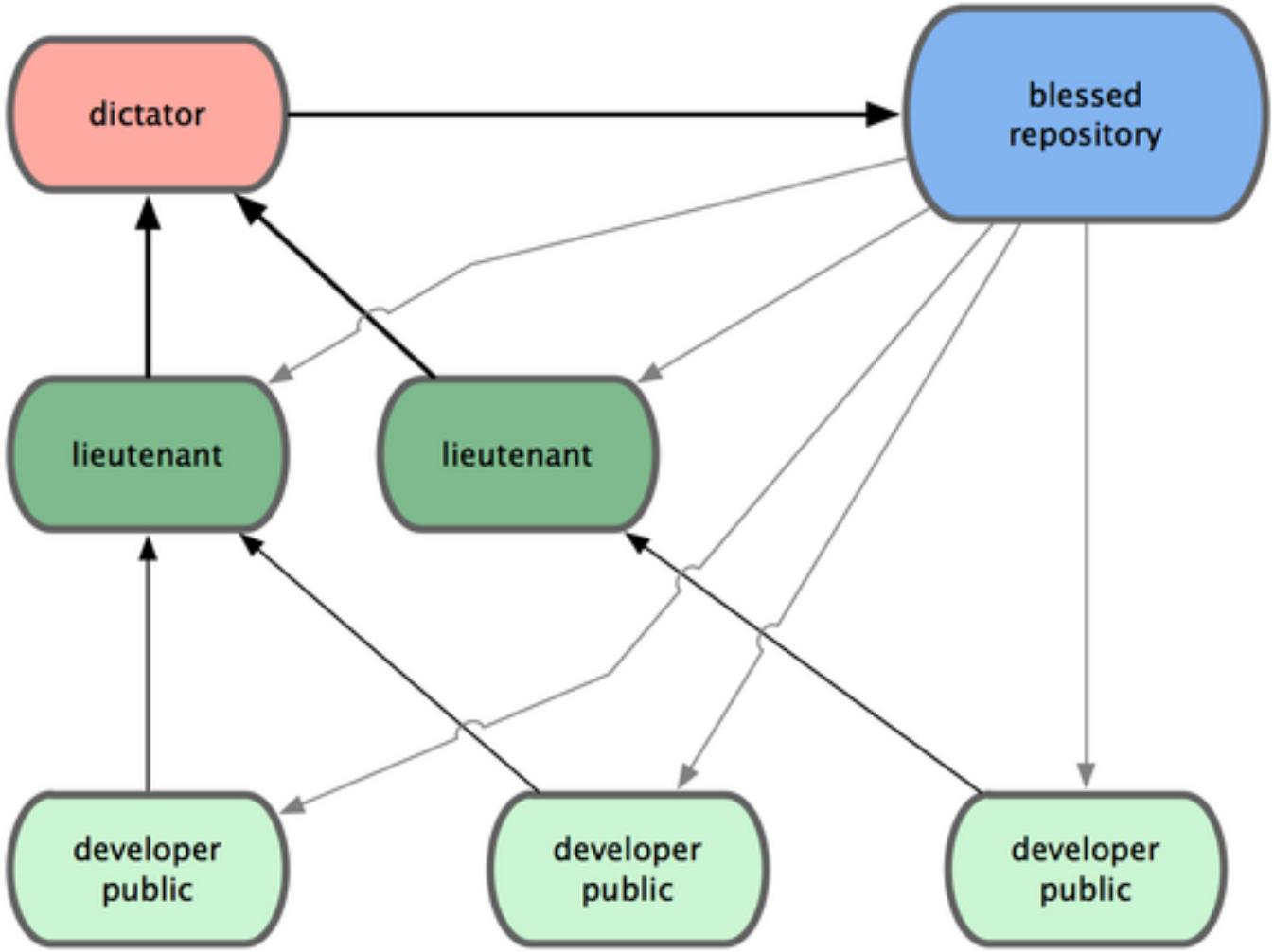
*de Scott Chanson: http://git-scm.org/book/

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 repositorios pueden estar alojados en hosts que tenemos que configurar.
  - ★ Configuración de los demonios.
  - ★ Protocolos de acceso.

    local: file:///home/juan/demo.git

    git: git@github.com:jquemada/swcm2012.git

    ssh: ssh://github.com/jquemada/swcm2012

    http(s): https://github.com/jquemada/swcm2012
  - ★ Cuentas de usuarios, permisos, acceso anónimo,etc.
  - ★ Creación de los repositorios.
- Los repositorios también pueden alojarse en sitios que se dedican al hosting de proyectos git:
  - ★ GitHub, Gitorious, Assembla, ...

9

GITHUB

- Portal Web para alojar repositorios GIT
  - Enfoque social y colaborativo
    - -> “social coding”
  - 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
  - Asignatura: https://github.com/ging/swcm

10

Parte 2:
Comandos básicos de GIT

Instalar GIT

 # GIT suele venir en sistemas UNIX,
 # o puede descargarse de la red e instalarse.

 # - Descargas: http://git-scm.com/downloads

 # - Instrucciones: http://git-scm.com/book sección 1.4 del libro.

 # - En github: https://github.com

12

Ayuda

 # Ayuda en línea de comandos:

$ git help **# Muestra lista con los comandos existentes**

$ git help comando **# Ayuda sobre comando especificado**

$ git help add **# Ayuda sobre el comando add**

$ git add --help **# Equivalente a anterior**
$ man git-add **# Equivalente a anterior**

 # Manual de referencia, chutetas, videos, otros enlaces:

http://git-scm.com/doc

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

https://na1.salesforce.com/help/doc/en/salesforce_git_developer_cheatsheet.pdf

13

Configurar GIT

```
# El comando "git config" permite manejar opciones de configuración.

# Las opciones configuradas pueden afectar a distintos ámbitos:
#    - Para todos los proyectos en el sistema.
#      Usar opción --system. La configuración se guarda en /etc/gitconfig
#    - Para todos los proyectos del usuario.
#      Usar opción --global. La configuración se guarda en ~/.gitconfig
#    - Sólo para el proyecto actual.
#      Sin opción. La configuración se guarda en .git/config

# Consultar todas las opciones existentes: git help config

# Dos opciones importantes para que todo vaya firmado correctamente:
$ 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
```


14

Creación de un 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

$ cd .../planet       # moverse a la raíz del directorio de proyecto (trabajo)

planet$ git init      # init: crea un nuevo proyecto GIT vacío en directorio planet
                      # Se crea el subdirectorio .git con los ficheros del repositorio

planet$ git add readme.txt     # add: añade readme.txt al index
planet$ git add *.java         # add: añade todos los java al index
planet$ git add .          # add: añade todos los ficheros modificados al index

planet$ git commit -m 'Primera version'
                # commit: congela una versión del proyecto en el repositorio

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


15

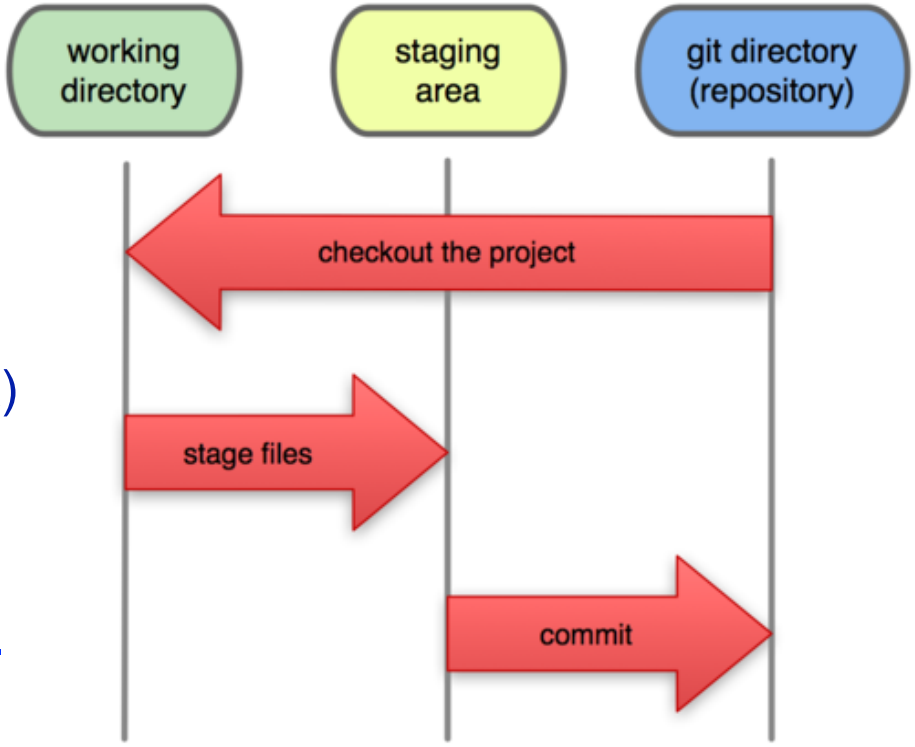
Clonar un proyecto existente

```
# 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).

# Clonar un proyecto ya existente en github:
$ git clone http://github.com/ging/planet
                                  # Se crea un directorio llamado planet con el clon

# Podemos indicar cual es el nombre del directorio a crear
$ git clone http://github.com/ging/planet my_planet
         # El repositorio clonado se creará en el directorio my_planet

# El URL soporta varios protocolos: http: git: file: ssh:
# Clonar el repositorio planet creado en la transparencia anterior:
$ git clone ../el_path_hasta_el_directorio_a_clonar/planet

 # A partir de este momento:
# - podemos trabajar normalmente en el proyecto clonado: add, commit, ...
# Y en algunos momentos sincronizaremos los dos repositorios.
```


16

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
- **Area de cambios** (staging area, index)
  - Contiene información sobre las modificaciones realizadas en los ficheros de directorio de trabajo que **se guardarán** en el próximo commit.
- **Directorio GIT** (el repositorio)
  - Guarda todas las versiones del proyecto.
    - ficheros, metadatos, ...

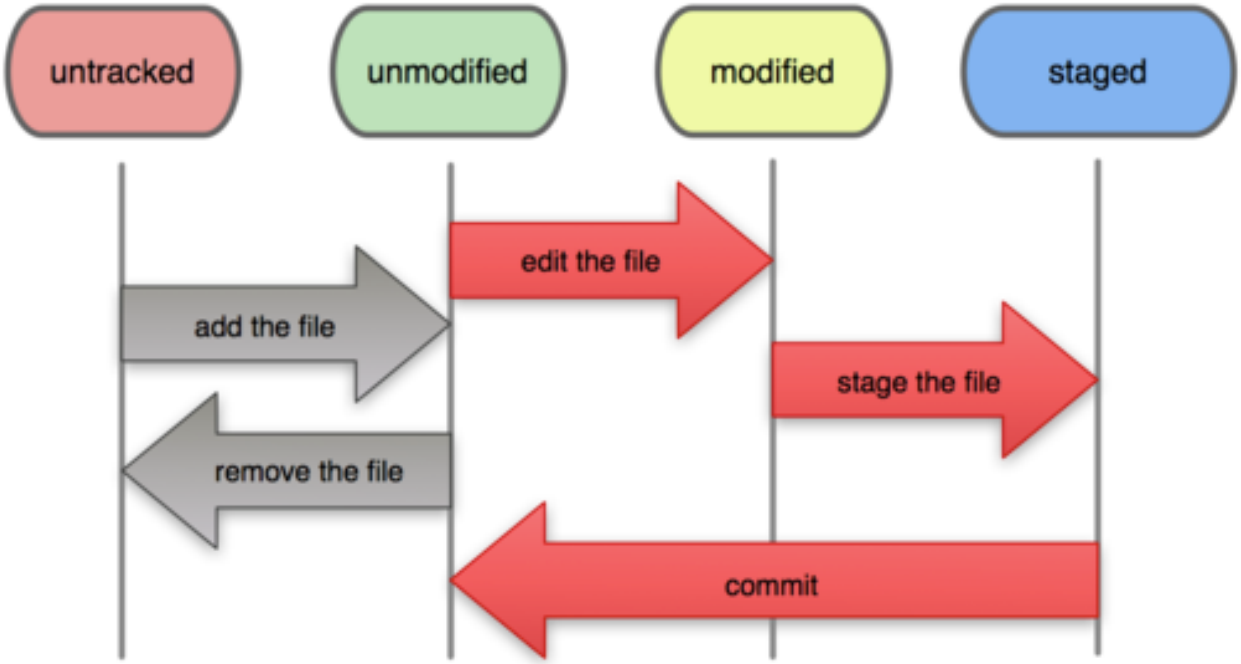


*de Scott Chanson: http://git-scm.org/book/

(El repositorio, el staging area y otros datos se mantienen en el directorio .git)

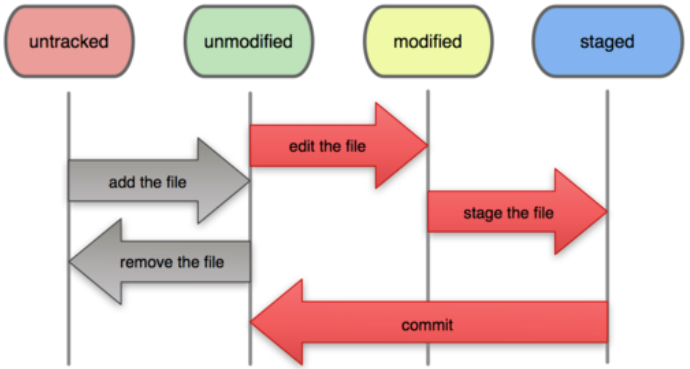
Estado de los ficheros

- 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.



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

Consultar el estado de los ficheros del directorio de trabajo




```
# "git status" muestra estado de los ficheros del directorio de trabajo:
#     1) Changes to be committed.
#     2) Changed but not updated.
#     3) Untracked files.

$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#           modified:  README
#           new file:   CharIO.java
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#           modified:   benchmarks.rb
#
# Untracked files:
#   (use "git add <file>..." to include what will be committed)
#
#           merge.java
#          library/lib.js
```

Ficheros nuevos o con modificaciones incluidos en el staging area

Ficheros modificados no incluidos en el staging area

Ficheros no gestionados por GIT y no excluidos por .gitignore

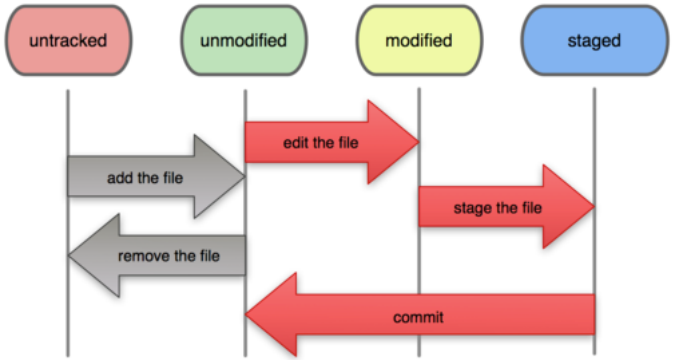
19

.gitignore

```
# .gitignore es un fichero que informa de los ficheros que no debe gestionar GIT.
#  - git status no los presentará como ficheros untracked.
#  - git add . no los añadira al staging area.

# Los ficheros .gitignore pueden crearse en cualquier directorio del proyecto,
#       y 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 patron 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

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

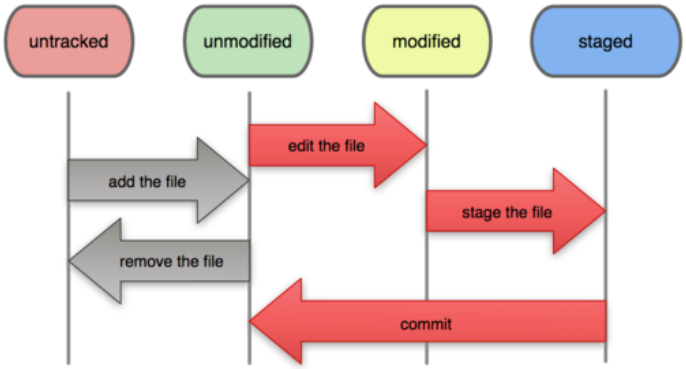

20

Consultar Cambios **no Staged**




```
# "git diff" muestra las modificaciones realizadas en los ficheros del area de trabajo
#              y que aun no han sido incluidas en el staging area.

$ git diff
diff --git a/benchmarks.rb b/benchmarks.rb          # modificaciones en 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                                           # contenido no modificado

-       #  -> 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                   # contenido no modificado
         log = git.commits('master', 15)
         log.size

diff --git a/tests.rb b/tests.rb                    # modificaciones en el fichero tests.rb
.....
```

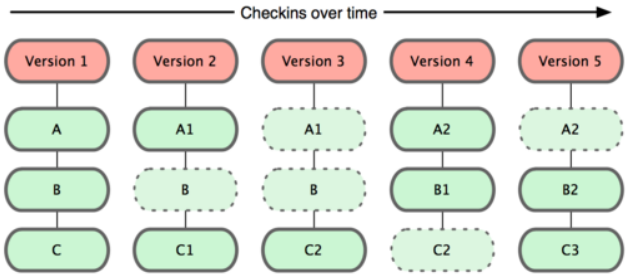
Consultar Cambios **Staged**




```
# "git diff --cached" o "git diff --staged"
# muestra las modificaciones que han sido staged para el proximo commit.

planet$> 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                                 # contenido no modificado

-       #  -> 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         # contenido no modificado
         log = git.commits('master', 15)
         log.size
```

git log: Historia de versiones

```
# La historia de commits (versiones congeladas) de un proyecto se muestra con "git log"
# Existen muchas opciones:
#   "git log --stat"              # muestra estadísticas
#   "git log --graph"             # muestra un gráfico ASCII con ramas y merges
#   "git log --since=2.weeks"     # muestra commits de las últimas 2 semanas

$ git log -2                  # Muestra los 2 últimos commits
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
```

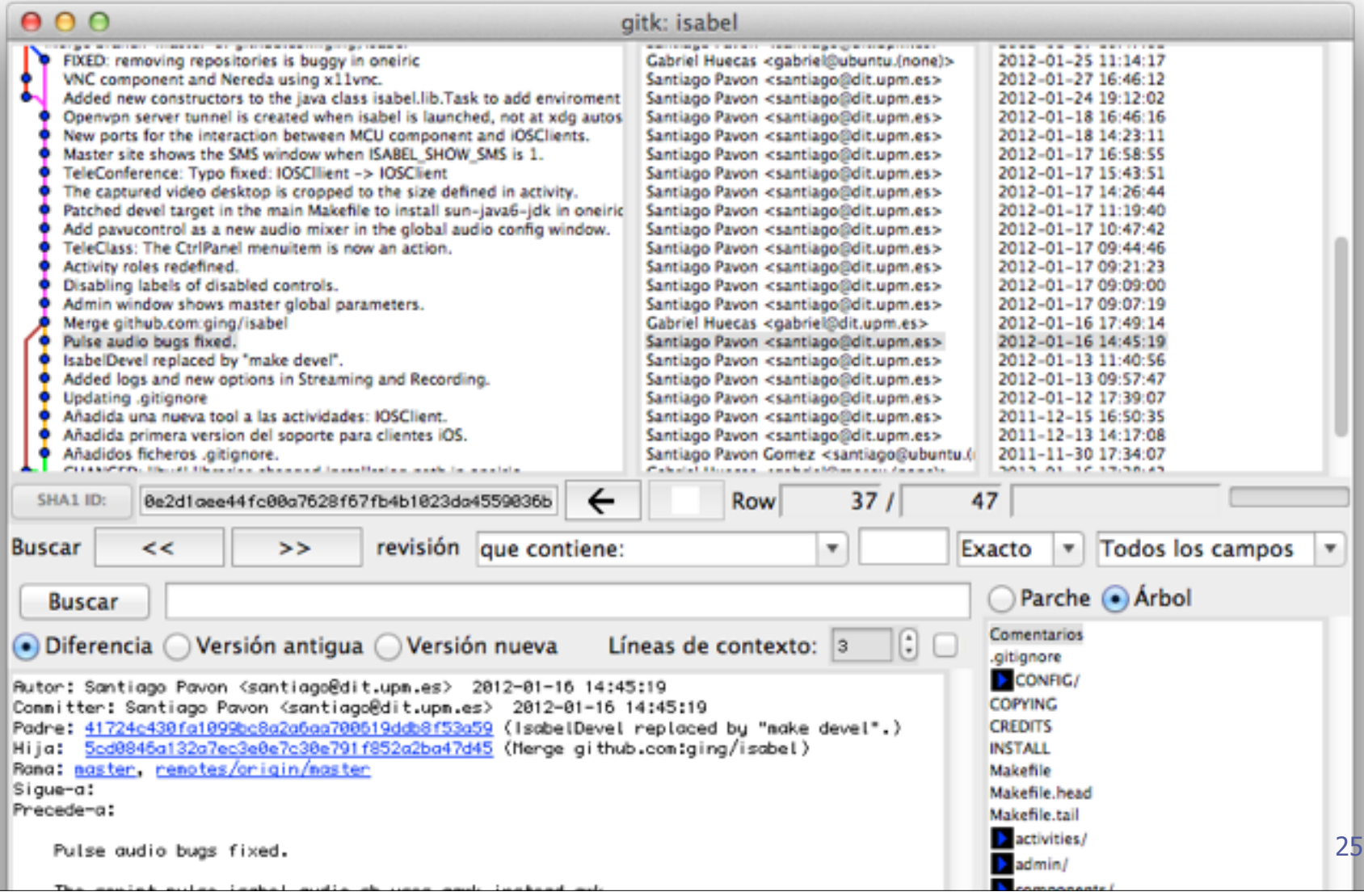



```
$ git log --oneline       # Muestra un log resumido. Una línea por commit

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

```
$ git log -p                                          # Muestra diferencias entre commits
commit 188799e21c4a71f13a2e729ccf77f3a960885682
Author: Juan Quemada <jquemada@dit.upm.es>
Date:   Mon Nov 21 18:17:13 2011 +0100

    migración base de datos

diff --git a/db/schema.rb b/db/schema.rb
index b5e6a79..61dcaab 100644
--- a/db/schema.rb
+++ b/db/schema.rb
@@ -11,6 +11,13 @@
 #
 # It's strongly recommended to check this file into your version control system.

-ActiveRecord::Schema.define(:version => 0) do
+ActiveRecord::Schema.define(:version => 20111121171513) do
+
+  create_table "types", :force => true do |t|
+    t.string   "name"
+    t.text     "description"
+    t.datetime "created_at"
+    t.datetime "updated_at"
+  end

 end

...
```


24

gitk
Ver la historia de versiones gráficamente



25

Borrar ficheros

```
# Eliminar un fichero en la próxima versión a congelar:

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

$ git rm --cached CharIO.java   # Borra fichero del staging area.
                                # 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 los borra del 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.

# 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.

# Mover un fichero. Internamente se implementa ejecutando git rm y git add.
$ git mv file_from file_to          # Renombrar un fichero
```


26

Renombrar ficheros

```
# Mover o renombrar un fichero:

$ git mv filename_old filename_new
$ git mv index.htm index.html



# Internamente se implementa ejecutando los comandos git rm y git add.

$ git mv filename_old filename_new

# es equivalente a ejecutar:

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


27

Deshacer operaciones realizadas: Modificar el último "commit"

```
# Para rehacer el último commit realizado usaremos git commit --amend
#     Para cambiar el mensaje de log.
#     Para añadir una modificación olvidada
#     . . .

$ git commit -m 'editor acabado'  # creamos el commit pero olvidamos
                                  # añadir un fichero, y el mensaje de
                                  # log no esta en ingles

# Realizamos los cambios olvidados:
$ git add forgotten_file

# Repetimos  git commit con la opción --amend y un nuevo mensaje de log
$ git commit --amend -m "Editor finished"

# Se elimina el commit erroneo y se crea un nuevo commit.
```

IMPORTANTE: no realizar --amend sobre un commit que se haya hecho público a otros desarrolladores (publicado en otro repositorio).

28

Deshacer operaciones realizadas: Eliminar Modificaciones Staged

```
# Para eliminar del staging area las modificaciones de un fichero:
#     git reset HEAD <file>

# Ejemplo:
# Modificamos un fichero y registramos los cambios en el staging area:

$ vi readme.txt  # editamos el contenido del fichero readme.txt
$ git add .      # añadimos todos los cambios existentes en todos los
                 # ficheros tracked al staging area.

# Pero no queriamos añadir los cambios de readme.txt.
# Para cambiar el estado de readme.txt a unstaged ejecutamos:

$ git reset HEAD readme.txt

# readme.txt ya no esta modificado en el staging area.
# readme.txt conserva sus modificaciones en el directorio de trabajo.
```


29

Deshacer operaciones realizadas: Eliminar Modificaciones en el Directorio de trabajo

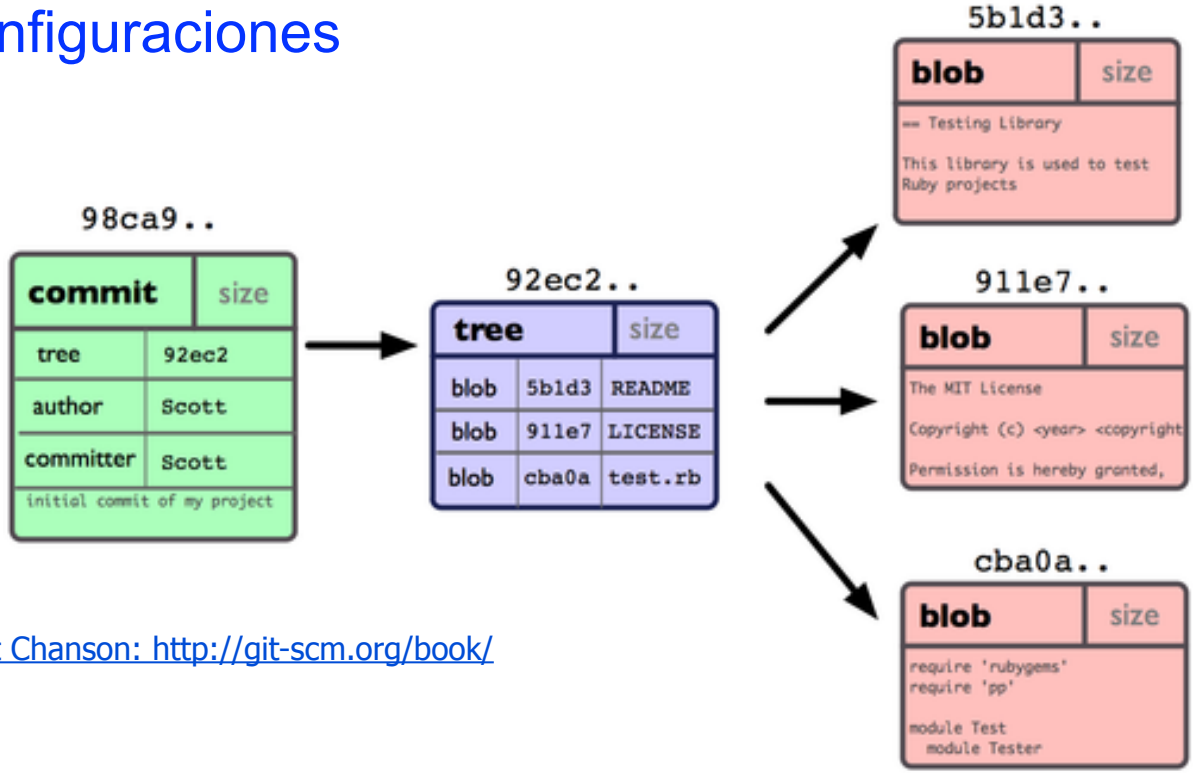
```
# Para eliminar las modificaciones realizadas en un fichero del
# directorio de trabajo, y dejarlo igual que la version del repositorio:
#     git checkout -- <file>

# Ejemplo:

# Modificamos un fichero.
$ vi readme.txt  # editamos el contenido del fichero readme.txt

# Nos arrepentimos de los cambios realizados.
# Para restaurar el fichero a su estado original ejecutamos:

$ git checkout -- readme.txt
```


30

Parte 3:
Usando Ramas

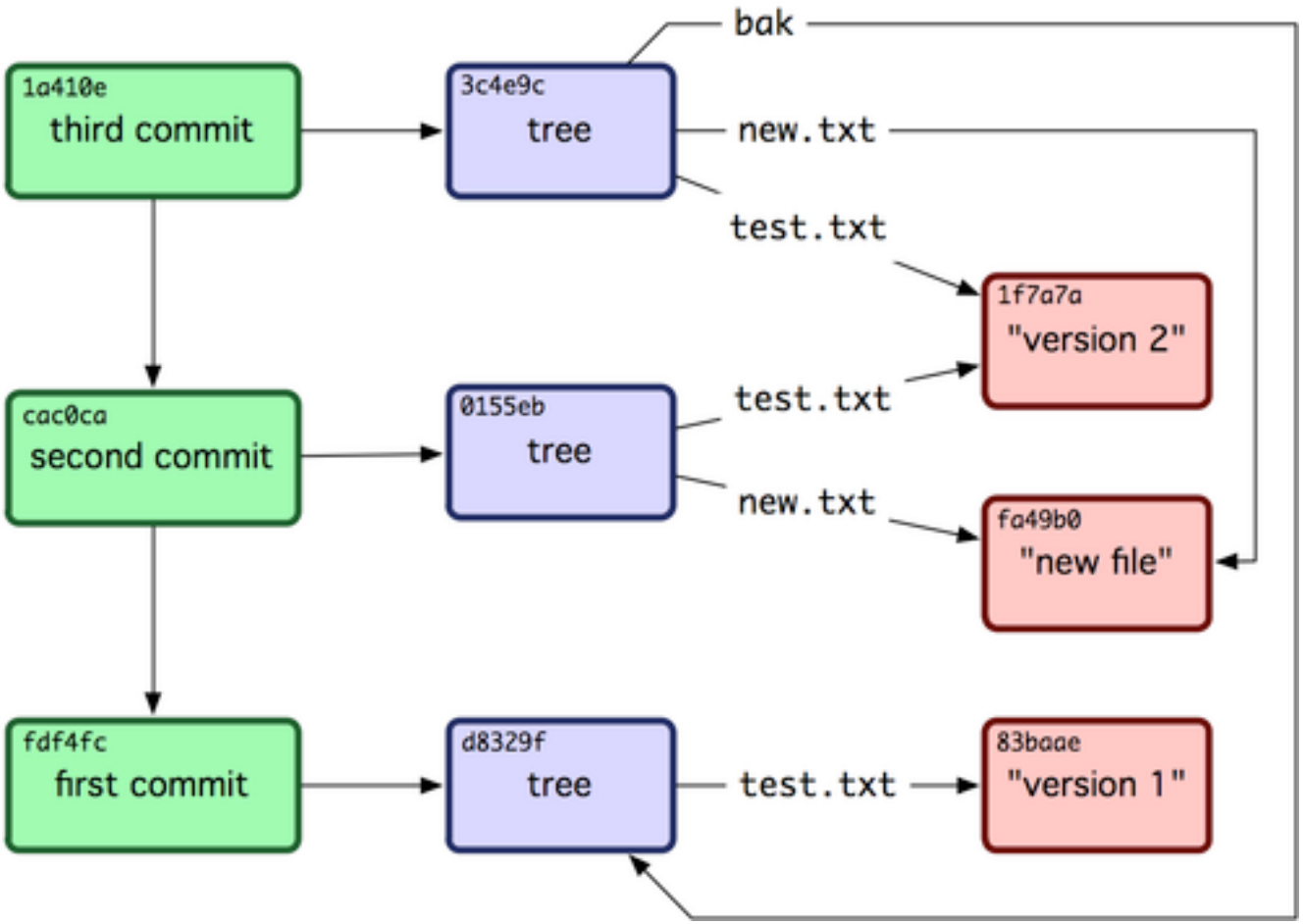
Información almacenada en .git

- En .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/

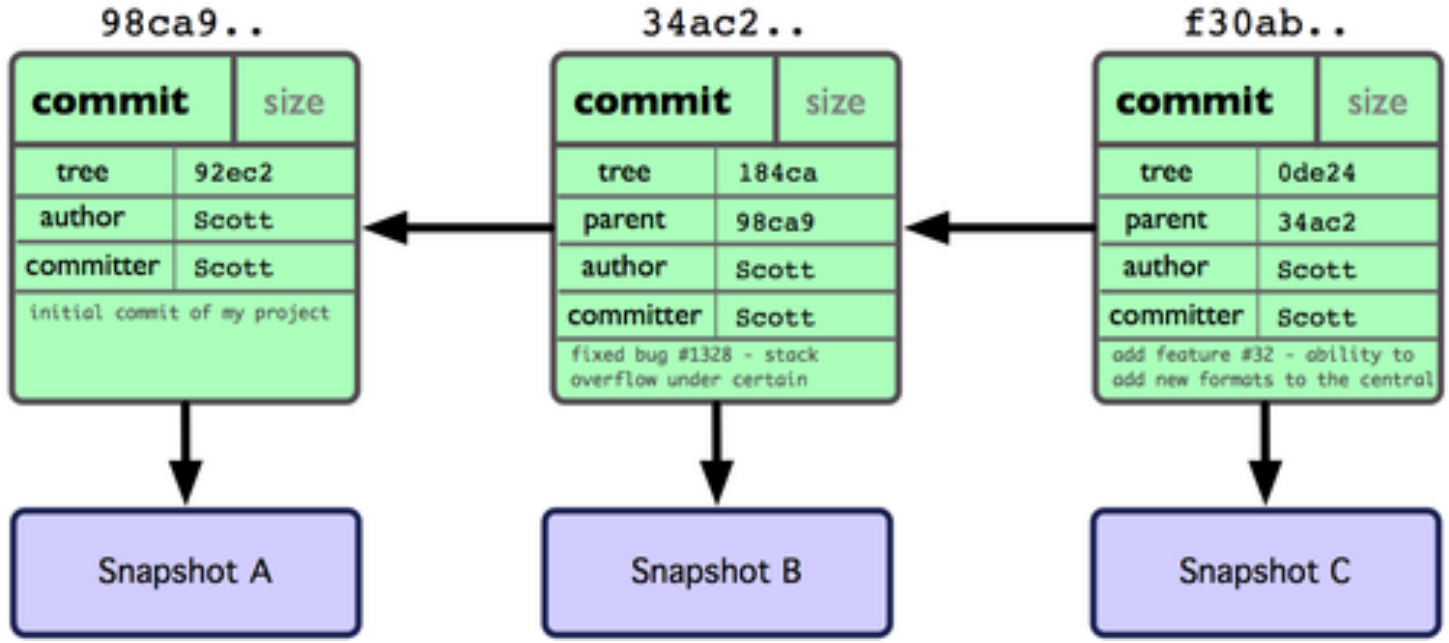
32


bak
1a410e
third commit
3c4e9c
tree
new.txt
test.txt
1f7a7a
"version 2"
cac0ca
second commit
0155eb
tree
test.txt
new.txt
fa49b0
"new file"
fdf4fc
first commit
d8329f
tree
test.txt
83baae
"version 1"

33

Árbol de commits

- La historia commit es un árbol.
  - Cada commit apunta a sus padres

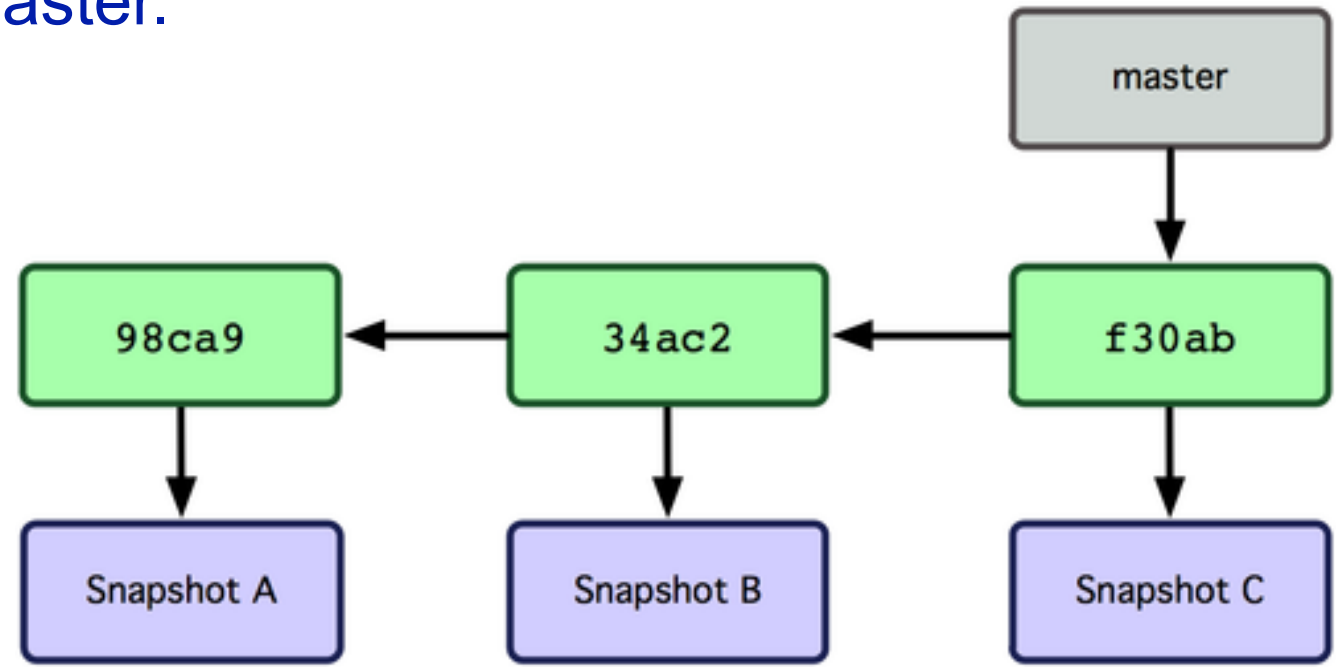


*de Scott Chanson: http://git-scm.org/book/

34

¿Que es una rama?

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

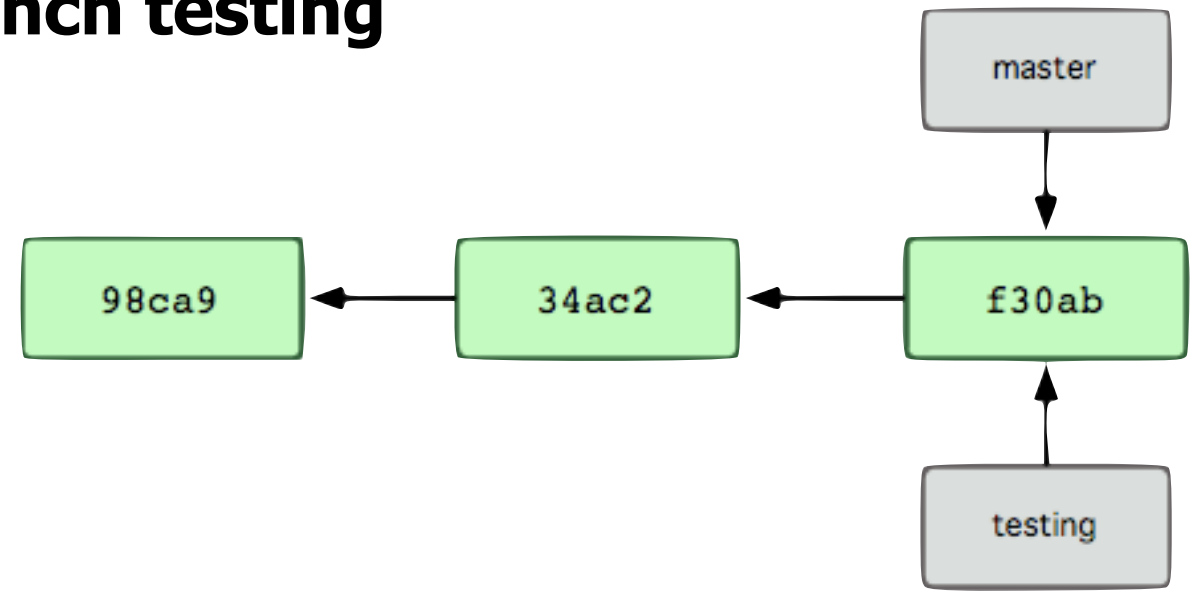


*de Scott Chanson: http://git-scm.org/book/

35

- Para crear nuevas ramas: **git branch <nombre>**
  - Se crea un nuevo puntero.

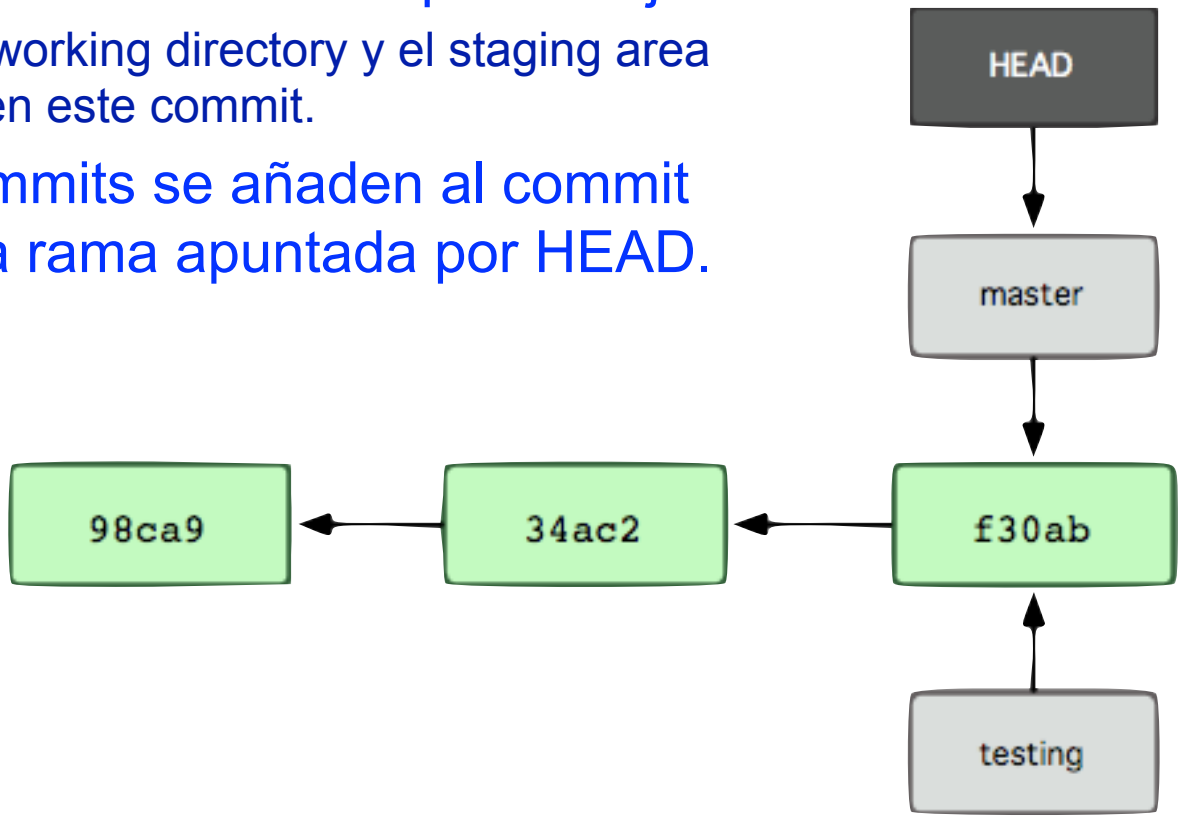
$ git branch testing



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

36

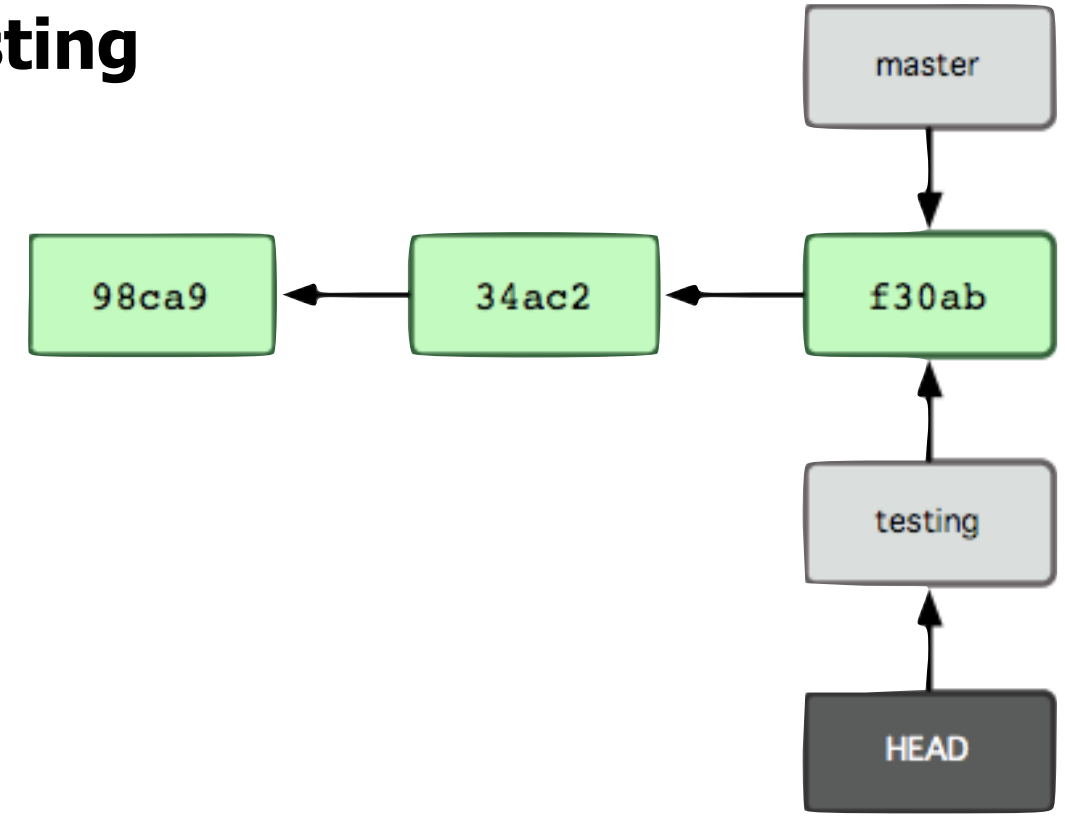
- En GIT existe una referencia llamada **HEAD** que apunta a la rama actual
  - ★ que apunta al commit sobre el que trabajamos.
    - los ficheros del working directory y el staging area están basados en este commit.
  - ★ Los nuevos commits se añaden al commit apuntado por la rama apuntada por HEAD.



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

- Para cambiar de rama: **git checkout <nombre_rama>**
  - **HEAD** apuntará a la nueva rama.
  - Se actualizan el working directory y el staging area.

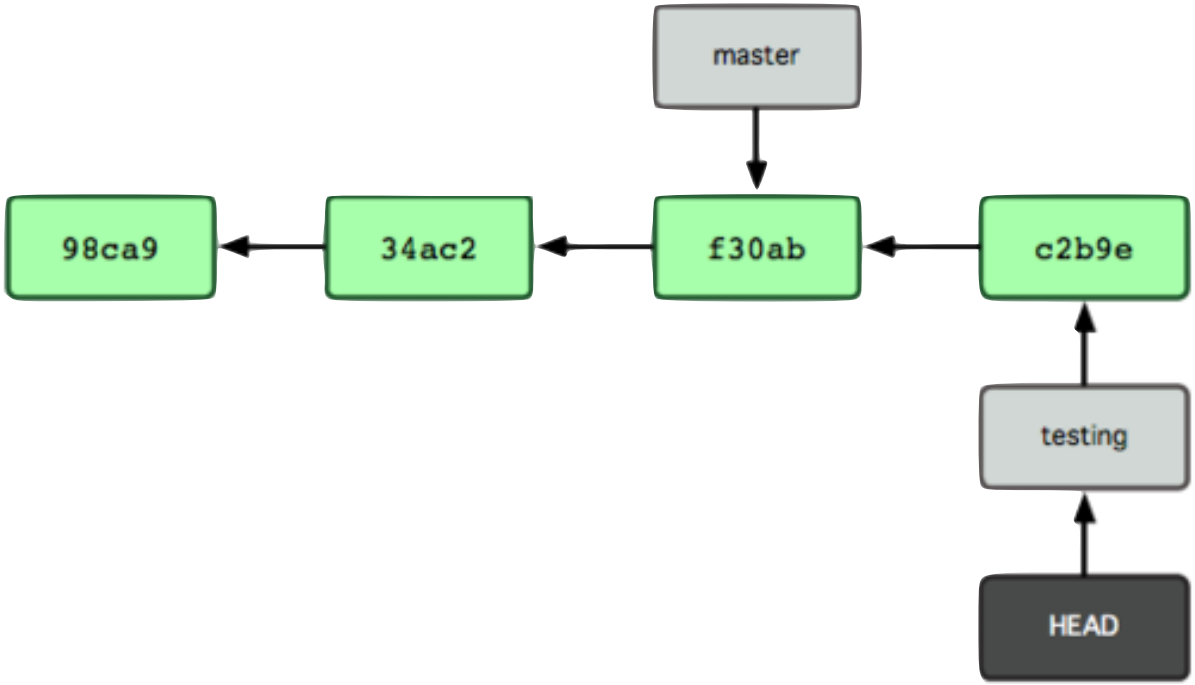
$ git checkout testing



38

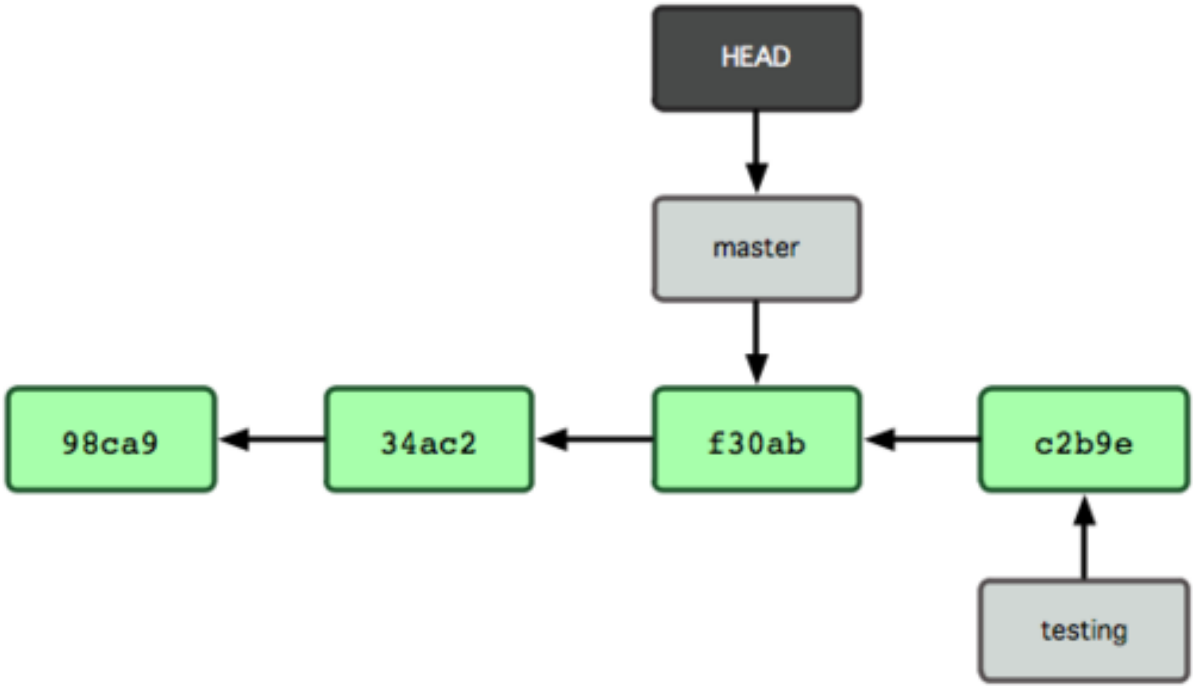
- Al hacer commit avanza la rama apuntada por HEAD.

$ git checkout testing



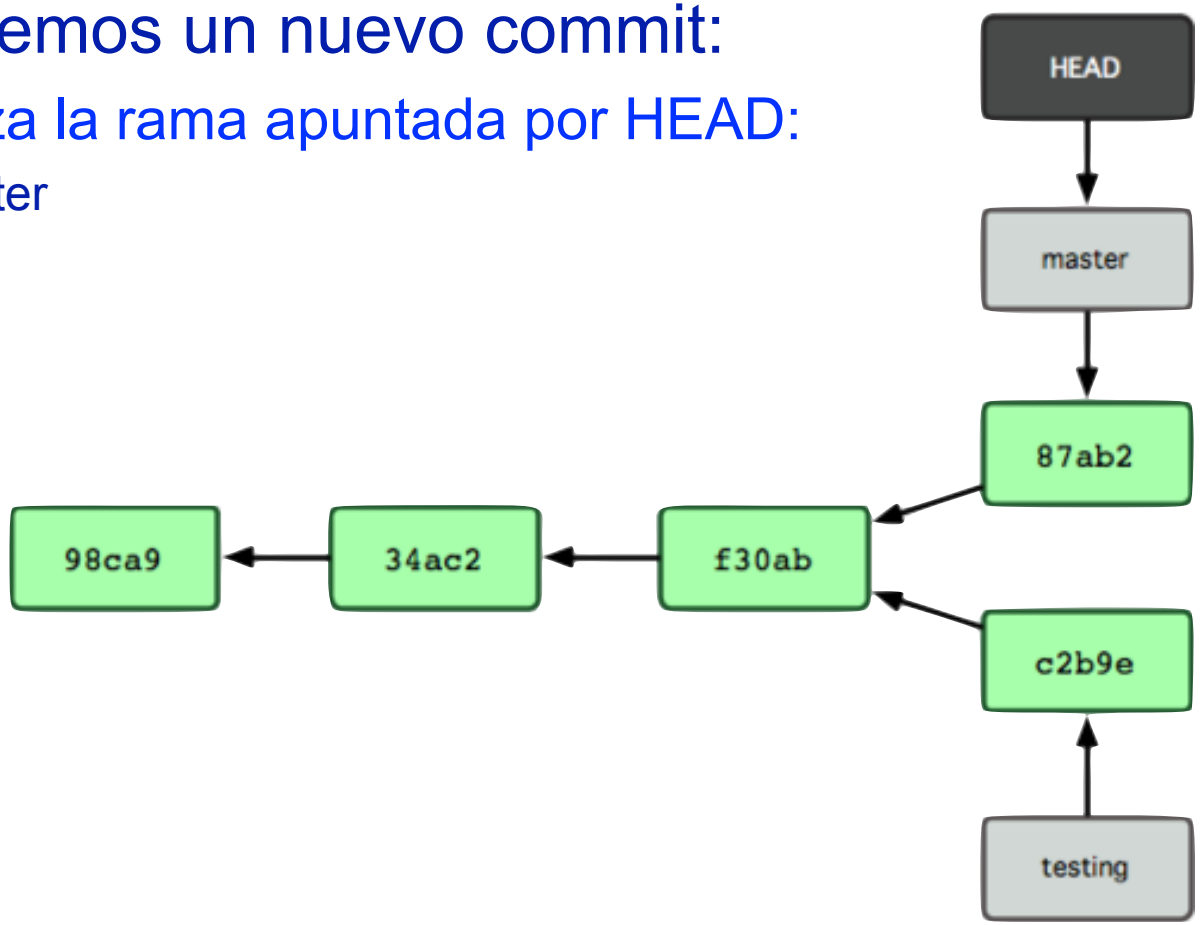
- Para volver a la rama master:

  $ git checkout master



40

- Si hacemos un nuevo commit:
  - avanza la rama apuntada por HEAD:
    - master



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 en un commit dado:

  git branch <nombre> <commit>

- El comando checkout con la opción -b también permite crear una rama y cambiarse a ella:

  git checkout -b <nombre>

42

Cambiar de rama

- Para cambiar de rama:

  git checkout <nombre>

- Para crear una rama y cambiarse a ella:

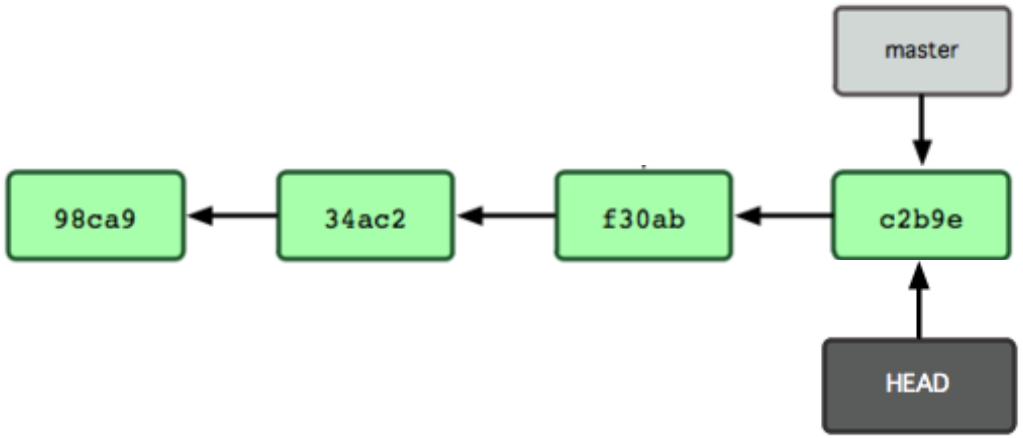
  git checkout -b <nombre>

- Checkout falla si tenemos 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.

43

Más usos de "git checkout"

```
# "git checkout ." deshace cambios staged de area de trabajo

planet$> git checkout .


# "git checkout -- <fichero>" deshace cambios stagged de <fichero>
#  en area de trabajo

planet$> git checkout -- <fichero>

                                  #  !!OJO!! Los cambios se pierden
                                  #  y no podrán volver a ser restaurados
```



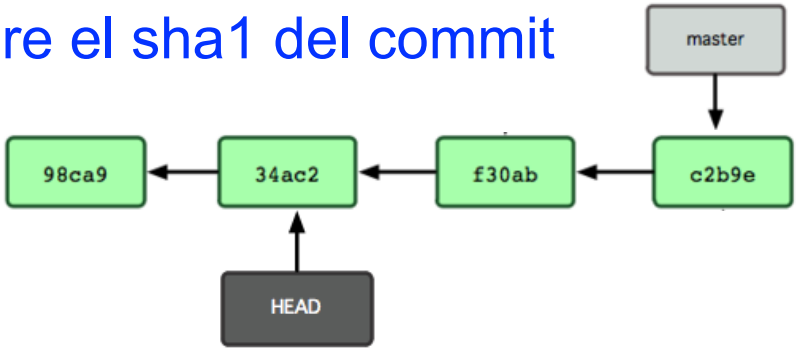
44

Detached HEAD Branch

- Es una rama que se crea sobre un commit que no tiene un nombre de rama apuntándole.

  $ git checkout 34ac2

  - Se ha realizado el checkout sobre el sha1 del commit



- Suele hacerse para inspeccionar los ficheros en un punto de la historia.
- Se creado una rama sin un nombre.
  - En cualquier momento podemos crear una rama en este punto con: **git checkout -b <nombre_rama>**

45

Unir ramas

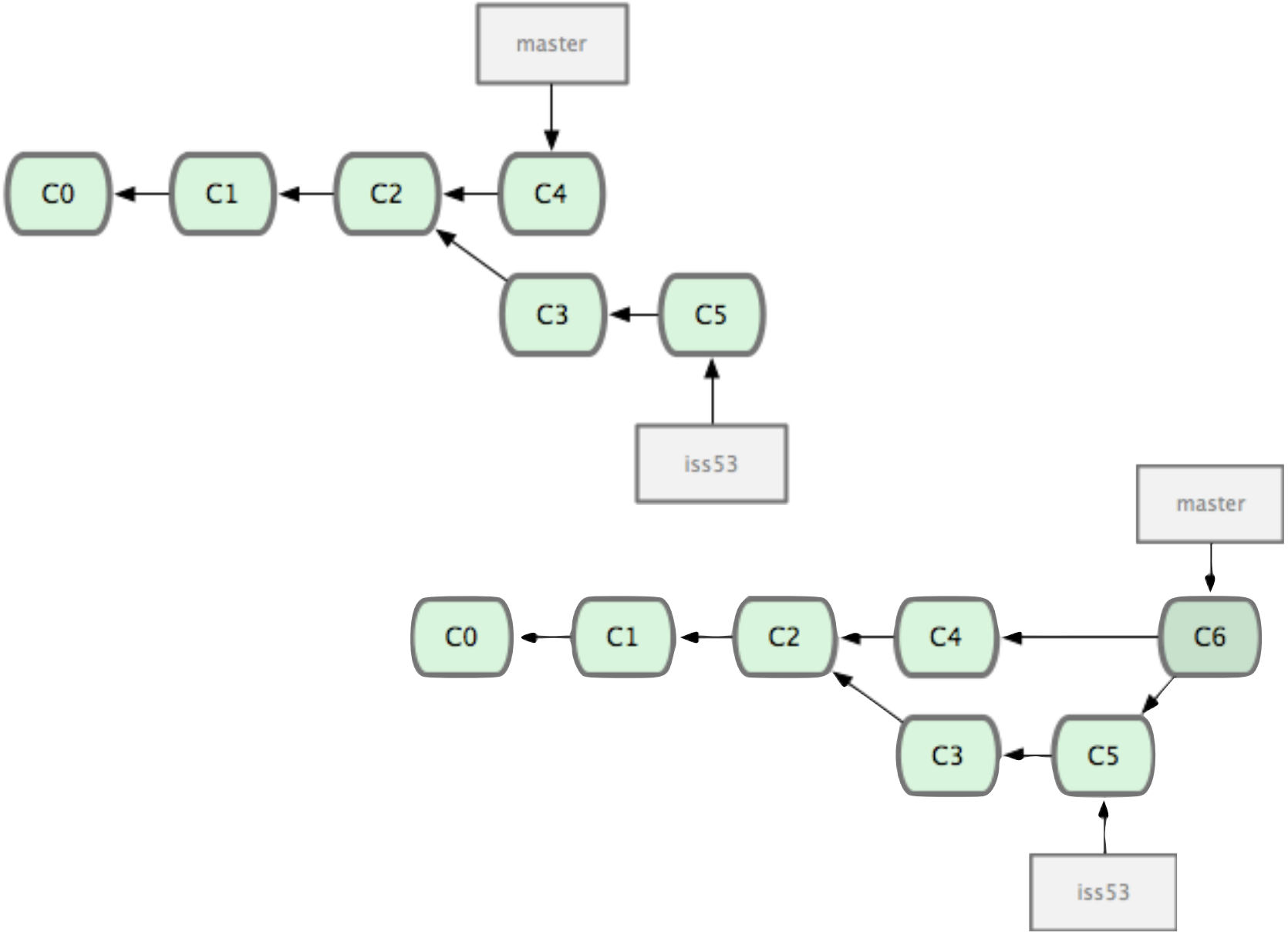
- Para incorporar en la rama actual los cambios realizados en otra rama:

  git merge <rama>

- Internamente GIT analiza la historia de commits para calcular como hacer la mezcla.
  - ⋆ 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
```


46



47

Conflictos

- Al hacer el merge pueden aparecer conflictos
  - 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
```

  - **git status** lista los ficheros con conflictos como **unmerged**.
- Para resolver el conflicto hay que:
  - editar el fichero para resolver el conflicto.
  - y ejecutar **git add** y **git commit**.

48

Borrar ramas

- Una ver terminado el trabajo con una rama
  - la borraremos con **git branch -d <nombre>**
    - Se elimina el puntero al commit.

- Si la rama a borrar no ha sido mezclada con la rama actual
  - se muestra un mensaje de error y no se borra.
  - Para borrar la rama independientemente de si ha sido mezclada o no, usar la opción **-D** en vez de **-d**.

    git branch -D <nombre>

49

Listar ramas

- Para listar las ramas existentes: **git branch**

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

  - Se marca con un asterisco la rama activa.

- 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 la rama actual.
  - **--no-merged** muestra las ramas que aun no se han

50

Parte 4:
Repositorios Remotos (Remotes)

¿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 remotos a su repositorio local.
  - Y suben (push) sus propias contribuciones a los repositorios remotos.
- 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.

52

- 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

53

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



# origin es el remote que se creó automáticamente al clonar el proyecto.
```


54

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

$ git  remote  add  pepito  git://github.com/pepe/planet.git

$ git remote -v
pepito   git://github.com/pepe/planet.git
origin   santiago@github.com:santiago/planet.git

# Ya podemos usar el nombre pepito en vez de la URL del repositorio.
# Ejemplos:
      # Descargar las últimas actualizaciones del repositorio:
      $ git fetch pepito
      # Subir a origin mis cambios en la rama master
      $ git push origin master
```


55

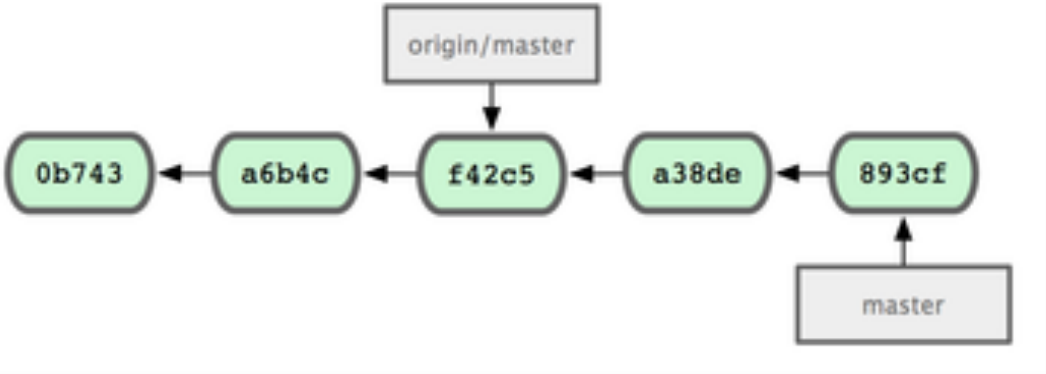
Más comandos sobre remotes

```
# 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 informacion sobre los remotes definidos: git remote update

# Para borrar ramas que ya no existen en el remote: git remote prune
```


56

Ramas Remotas

- Una rama remota es un puntero a un commit.
  - ★ Indica cual 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.

57

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>**

58

- 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.
    - Notese 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.

59

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 un 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
```


60

Descargar datos y Mezclar

```
# Bajarse los datos de un remoto y aplicar merge:
        git pull [nombre_del_remote [refspec]]

# Si la rama actual es una tracking branch:
#    El comando git pull [nombre_del_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 se realiza un pull de origin.

# Este comando ejecuta un fetch con los argumentos dados, y despues realiza
# un merge en la rama actual con los datos descargados.
$ git pull pepito demo # Mezcla en la rama actual la rama demo de pepito.
```


61

Subir datos a un remote

```
# De forma general, el comando para subir cambios a un remote es:
   git push [nombre_remote [refspec]]

# La operación push sólo puede hacerse sobre repositorios bare.
#    Son repositorios donde no se desarrolla. Sólo se suben cambios.
#    No tienen un directorio de trabajo.
#    Los repositorios bare se crean con init o clone, y usando la opcion --bare.

# Si la rama actual es una tracking branch, no suele especificarse un refspec:
#
#    El comando git push [nombre_del_remote], sube los cambios desarrollados 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.

# La operación push sólo puede realizarse si:
#   - se tiene permiso de escritura en el repositorio remoto.
#   - nadie ha subido nuevos cambios al repositorio remoto, es decir,
#     estamos actualizados.
#       * Si en el remote hay actualizaciones que no tenemos, deberemos hacer
#         un pull antes de poder subir nuestros cambios.
#       * No pueden subirse actualizaciones que puedan producir conflictos.
```


62

```
# 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



# refspec permite indicar los nombres distintos para las ramas local y remota:
#         Subir los cambios de rama local uno a la rama dos del repositorio origin:
$ git push origin uno:dos
#         refspec tiene el formato <rama_local>:<rama_remota>



# Para borrar una rama remota :
$ git push origin :dos
#     Usamos un refspec con un nombre de rama local vacío,
#     y con el nombre de la rama remota a borrar.
```

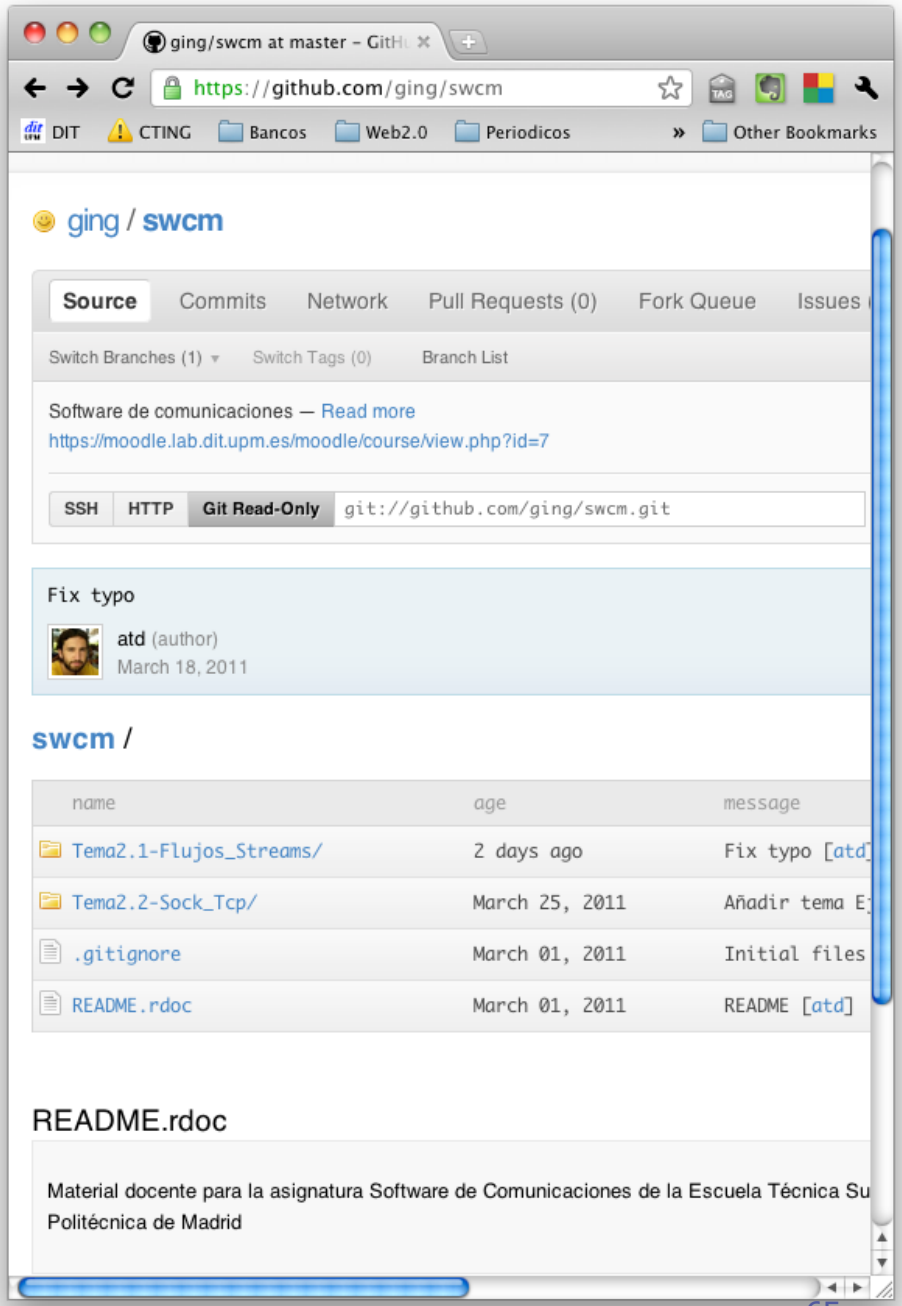

63

Parte 5:
GITHUB

64

GITHUB y SWCM

```
# GITHUB: red social de desarrolladores
# de software para compartir proyectos
# con GIT:   http://github.com
#
#  - Basado en cuentas de personas, no
#    en proyectos
#
# En SWCM distribuimos con GITHUB:
#   - Ejemplos de transparencias
#
# Repositorios SWCM y LSWC están en:
#   http://github.com/ging/swcm
#   http://github.com/ging/planet
#
# Se recomienda:
#
# 1) Hacer una cuenta en GITHUB
#   - Seguir instrucciones online
#            -> crear clave ssh
#
# 2) Bajar ejemplos de repos GITHUB
#   - Tienen permiso de lectura pública
#
```



GITHUB: Disco virtual

swcm /

name
Tema2.1-Flujos_Streams/
Tema2.2-Sock_Tcp/
.gitignore
README.rdoc

swcm / Tema2.1-Flujos_S

name
..
Acceso_a_Dispositivos/
Expresiones_Regulares/
Ficheros/
Internacionalizacion/

swcm / Tema2.1

name
..
CharIO.java
char_io1.rb
char_io2.rb
lineIO.java

swcm / Tema2.1-Flujos_Streams / Acceso_a_Dispositivos / char

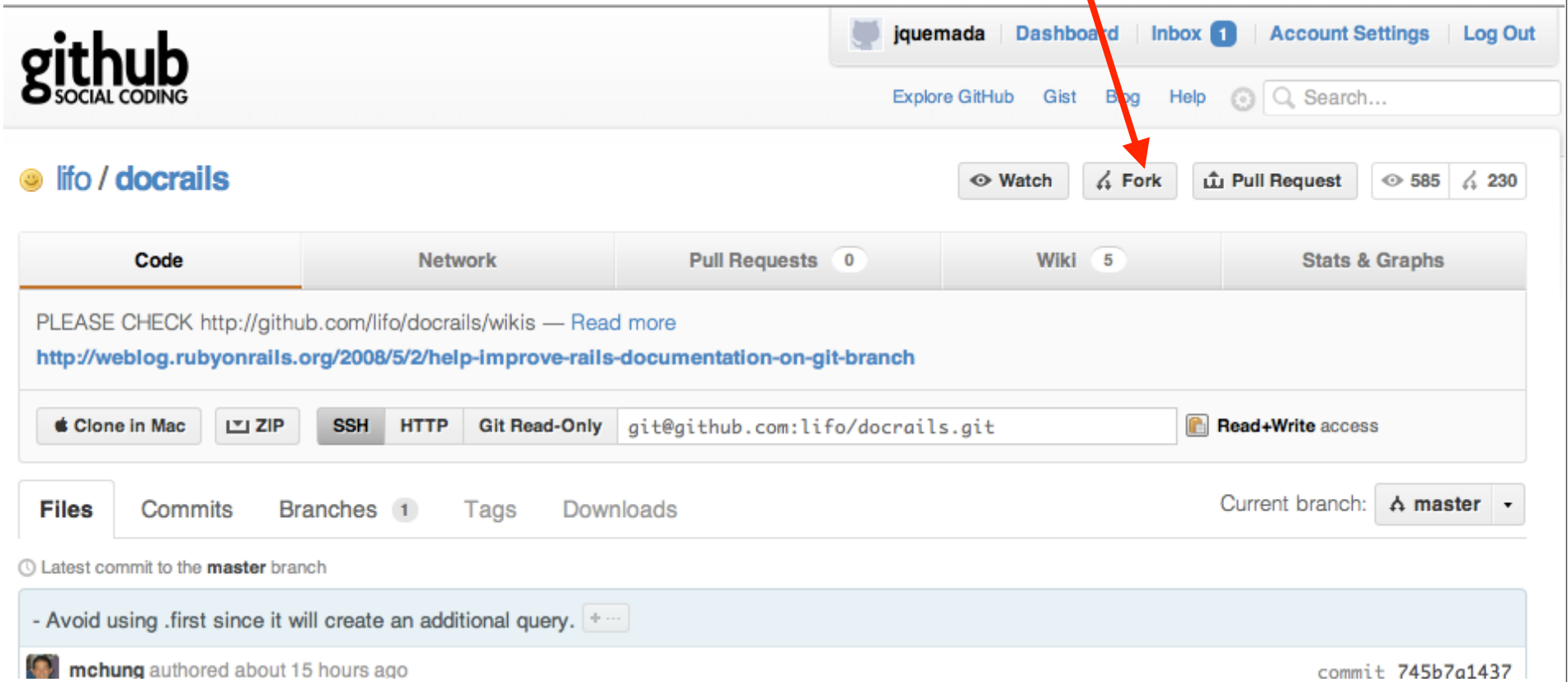
100644 | 8 lines (5 sloc) | 0.285 kb

```
1  # Ejemplo 2 con 'while' como statement modifier y métodos sin paréntesis
2  #  -> Es equivalente a Ejemplo 1, pero con una sintaxis mas concisa
3
4  charRead = "" # Declara variable charRead (antes de usarla)
5  STDOUT.putc charRead while ( (charRead = STDIN.getbyte ) != 46 )
6
7  puts "FIN"
8
```


66

Fork: Bifurcar un proyecto GITHUB

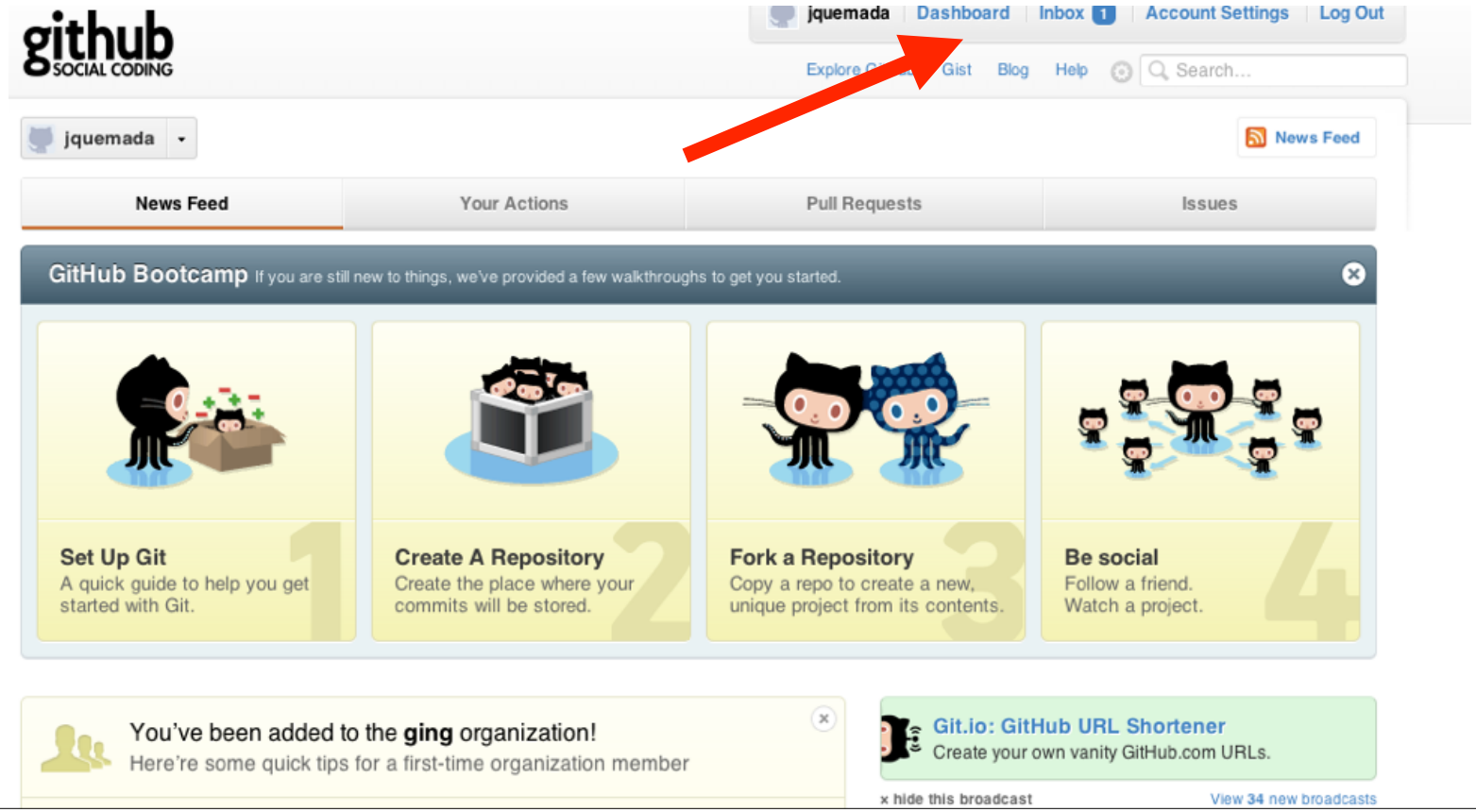
 # GITHUB permite bifurcar un repositorio público en nuestra cuenta
 # -> así podemos seguir trabajando sobre el
#
 # 1) Crear rama de repositorio original en GITHUB con "Fork"



67

GITHUB: creación de repositorios

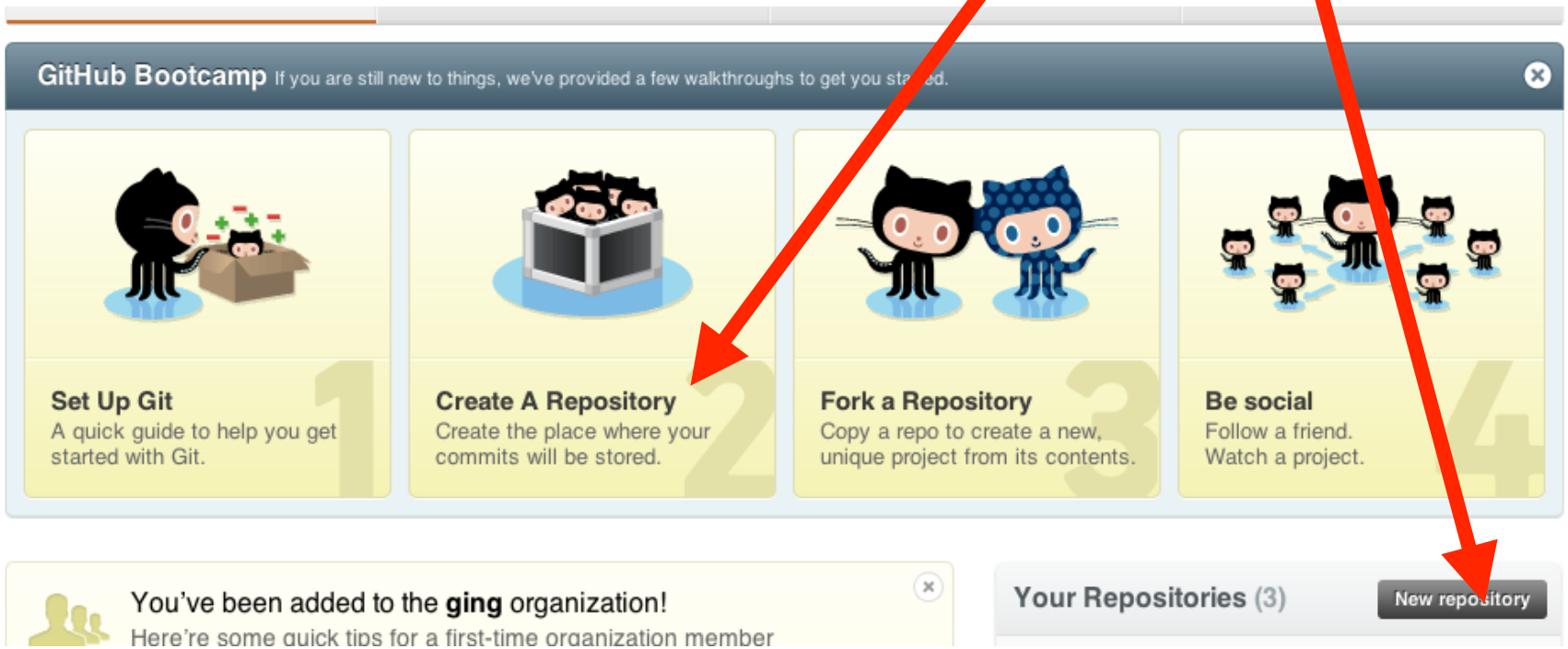
 # GITHUB permite crear repositorios en nuestra cuenta:
 # - Repositorios públicos son gratis, privados de pago
#
 # Un repositorio se crea y gestiona desde el "Dashboard"



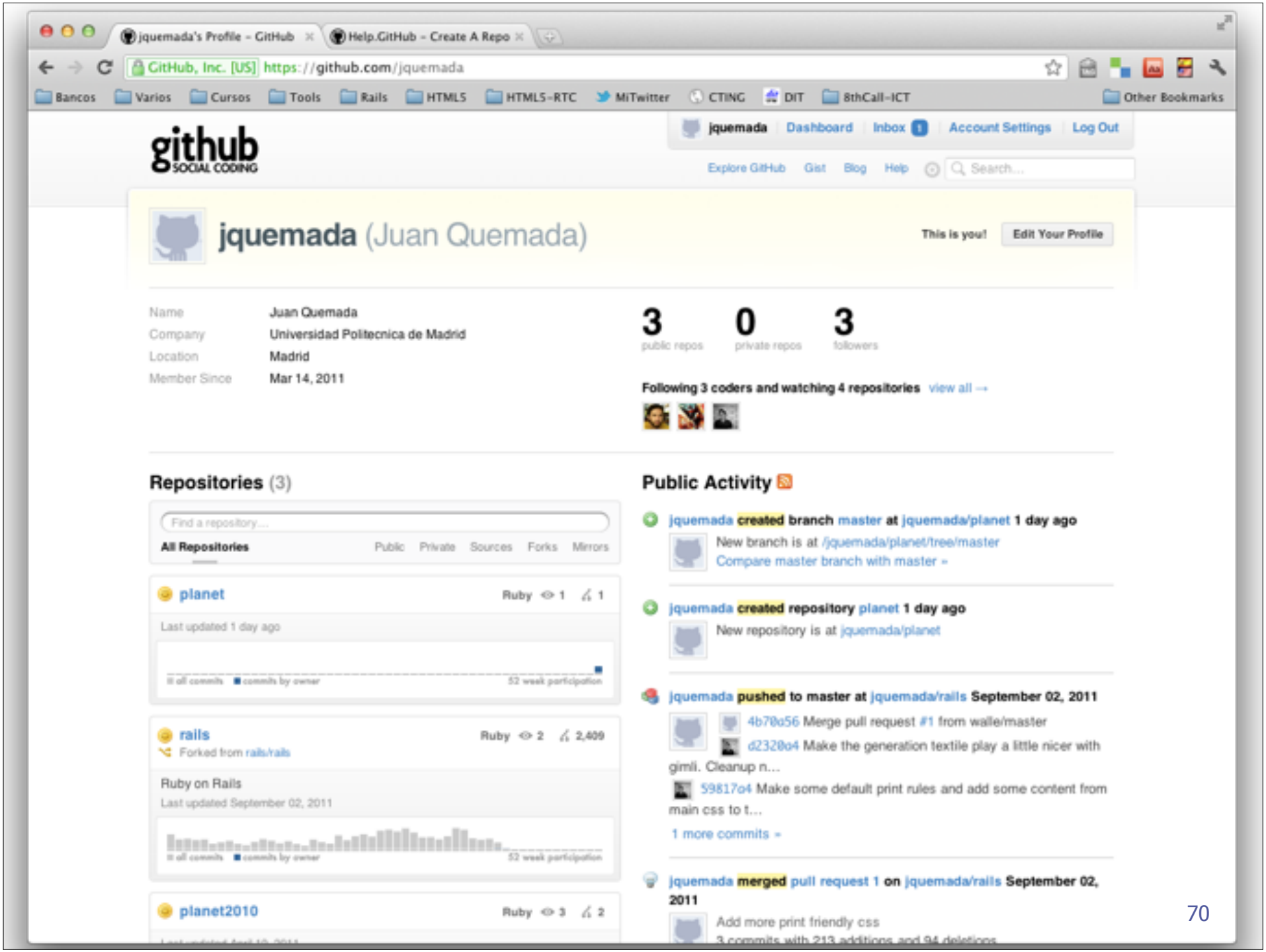
68

GITHUB: subir un repositorio local

```
# Para crear un proyecto nuevo en GITHUB hay que
#
#   1) Crear un repositorio vacio en GITHUB con    ->    "New Repository"
#
#   2) Hacer push de proyecto local (instrucciones GITHUB)

p1> git remote add origin git@github.com:usr/proy.git
p1> git push -u origin master   # -u "tracking reference"
.........
# Cualquiera podra:      ¡¡CLONAR EL PROYECTO!!
```



69


jquemada's Profile - GitHub
Help.GitHub - Create A Repo
GitHub, Inc. [US] https://github.com/jquemada
Bancos Varios Cursos Tools Rails HTML5 HTML5-RTC MiTwitter CTING DIT 8thCall-ICT Other Bookmarks
github SOCIAL CODING
jquemada Dashboard Inbox 1 Account Settings Log Out
Explore GitHub Gist Blog Help Search...
jquemada (Juan Quemada)
This is you! Edit Your Profile
Name Juan Quemada
Company Universidad Politecnica de Madrid
Location Madrid
Member Since Mar 14, 2011
3 public repos
0 private repos
3 followers
Following 3 coders and watching 4 repositories view all →
Repositories (3)
Find a repository...
All Repositories Public Private Sources Forks Mirrors
planet Ruby 1 1
Last updated 1 day ago
all commits commits by owner 52 week participation
rails Ruby 2 2,409
Forked from rails/rails
Ruby on Rails
Last updated September 02, 2011
all commits commits by owner 52 week participation
planet2010 Ruby 3 2
Public Activity
jquemada created branch master at jquemada/planet 1 day ago
New branch is at /jquemada/planet/tree/master
Compare master branch with master »
jquemada created repository planet 1 day ago
New repository is at jquemada/planet
jquemada pushed to master at jquemada/rails September 02, 2011
4b70a56 Merge pull request #1 from walle/master
d2320a4 Make the generation textile play a little nicer with gimli. Cleanup n...
59817a4 Make some default print rules and add some content from main css to t...
1 more commits »
jquemada merged pull request 1 on jquemada/rails September 02, 2011
Add more print friendly css
3 commits with 213 additions and 94 deletions

70

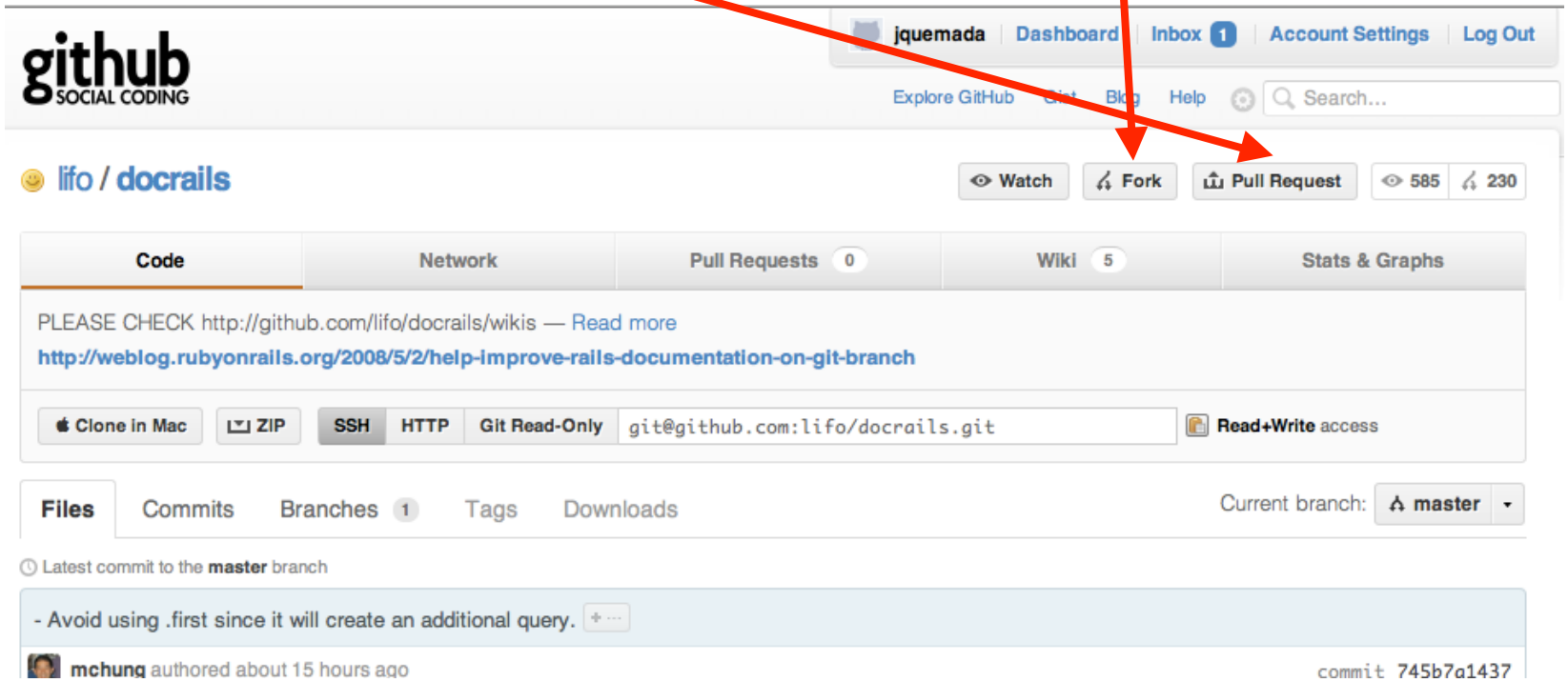
Contribuir a un proyecto GITHUB

 # 1) Crear rama de repositorio original en GITHUB con "Fork" en la cuenta propia

 # 2) Clonar la rama creada en nuestro repositorio GITHUB en ordenador local
p1> git clone git@github.com:usuario/proy.git

 # 3) Introducir contribución en proyecto local, crear commit y hacer "push" a GITHUB
p1> .....
p1> git add ...
p1> git commit -m '.....'
p1> git push origin master

 # 4) Se hace un "Pull Request" al repositorio original



71

Parte 6:
Temas Avanzados

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 coincida con otro SHA1 existente
  - un tag que apunte al comit.
  - el nombre de una rama.
  - referencia a un antepasado.
  - expresiones de rangos.
  - valores guardados en reflog.

Referencia a antepasados y rangos

- 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 commmit C:
  - C^ o C^1 commit padre en su misma rama cuando se creo
  - 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~1 commit padre
  - C~2 commit abuelo
  - C~3 commit bisabuelo
- Así, C~3^2 es el primer tio del bisabuelo del commit.

74

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 especificase varios commits
    - Indican el rango de commit accesibles por cualquiera de los commits especificados.
    - Pueden excluirse commits añadiendo **^C** o **--not C** pr excluir todos los commits accesibles desde C.

75

- 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.

76

Tags

Tags

- Se usan para etiquetar commits importantes de la historia (o ficheros importantes).
  - Ejemplo de uso:
    - Poner etiquetas con un número de version 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 consultar los tags existentes:
  - **git tag** **# lista todos los tags existentes**
  - **git tag -l <patron>** **# lista los tags que encajan con el patron dado.**
  - Ejemplo:

```
$ git tag -l v1.*
v1.0
v1.1
v1.2
```


78

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 345abac
      - Si no especificamos la opción -m se lanzará un editor.
- También se pueden crear tags firmados con GPG.

79

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

80

Modificación de los Commits

- Existen varios comandos que GIT que permiter 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
  - amend

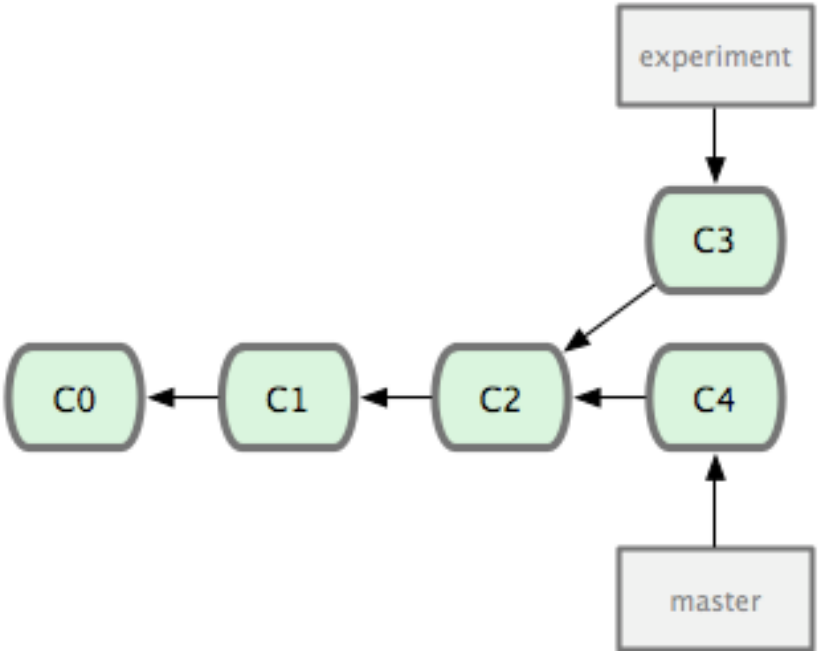
81

Rebasing

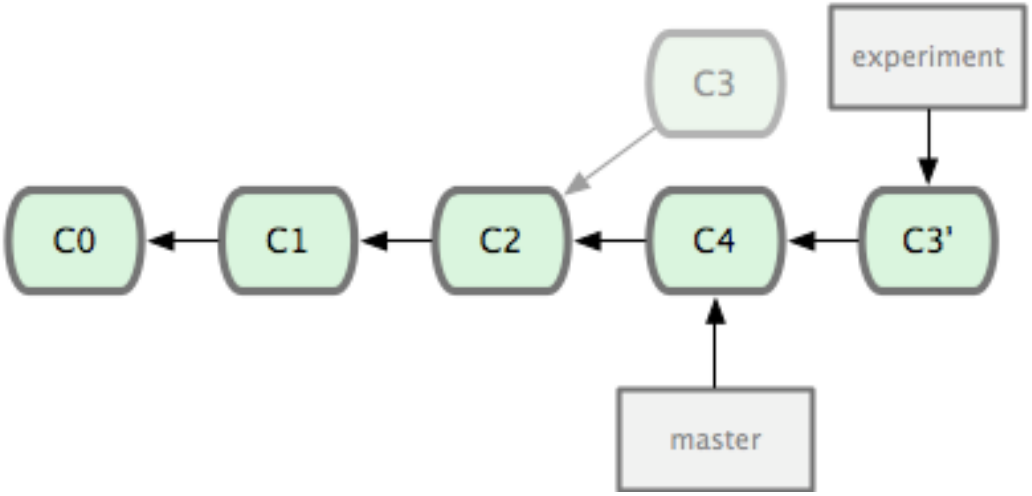
- Antes de nada: No hacer un rebase de un commmit que ya se haya subido a un repositorio público.
  - Cambiar commits que ya pueden tener otros desarrolladores causará problemas
- Rebase permite mover una rama de commits a otro sitio.

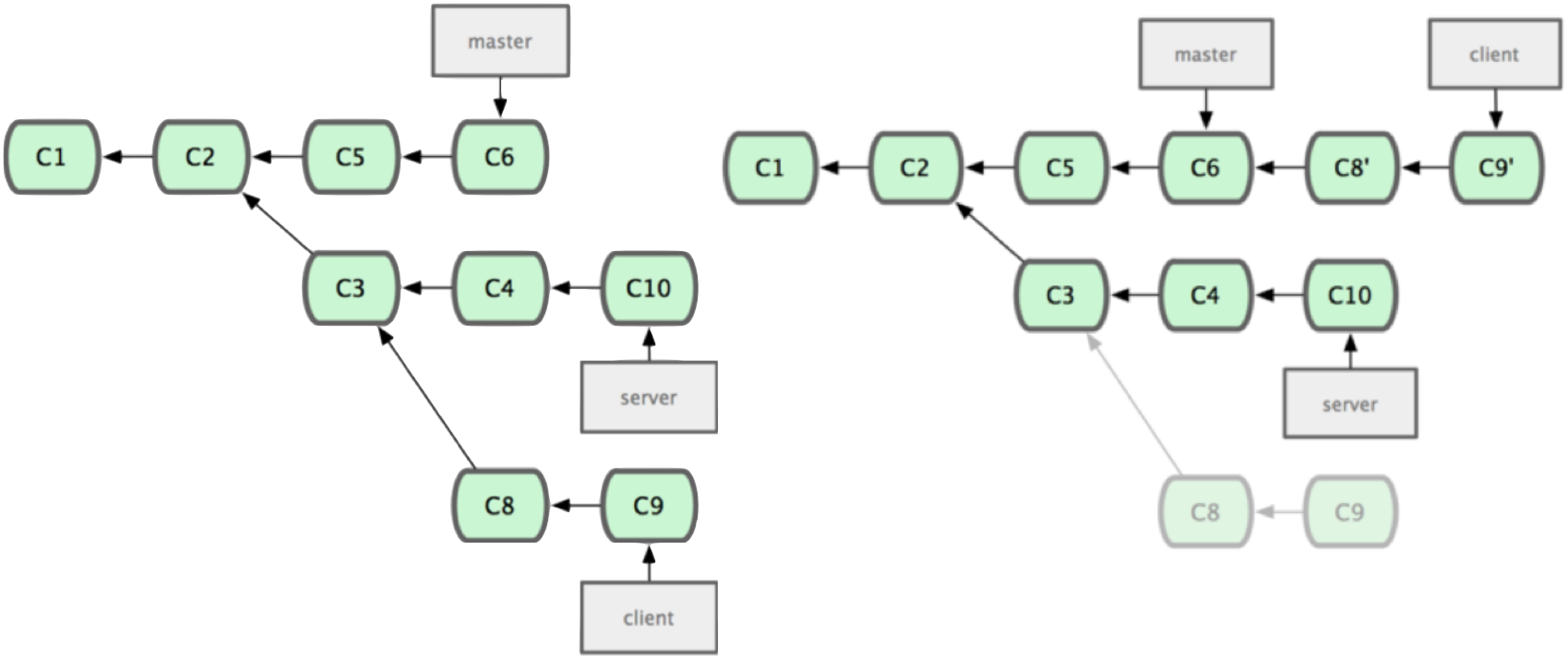
  git rebase [basebranch] [topicbranch]
  - Mueve los commits de la rama *topicbranch* a *basebranch*.
    - Busca un commit común entre basebranch y topicbranch, y partiendo de él, aplica los mismos cambios realizados en los commits topicbranch a basebranch.
  - Si se omite topicbranch, se usa la rama actual.
- Rebase también se usa para editar commits de forma interactiva.

82



$ 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 commmits uno a uno en la nueva localizacion.
  - 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, ejecuta:
    - git rebase --continue
      - Este comando aplica el commit pendiente y continua con el resto commits del rebase.
  - 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, ejecutaremos:
    - git rebase --abort
      - deshace todo lo hecho y deja el repositorio como estaba antes de hacer el “git rebase” original

85

- Otras operaciones que pueden hacerse con el comando rebase:
  - rebase interactivo: **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...
  - reposicionamientos de ramas más complicados.
    - Mover una subrama de una rama a una tercera rama.
      - ver opción --onto

86

git reset

- **git reset** se usa para restaurar HEAD a otro estado.
- **git reset <commit>**
  - Cambia HEAD para que apunte al commit dado.
  - Se restaura el staging area al nuevo estado.
    - Se pierden los cambios del staging area.
  - Los ficheros del directorio de trabajo no se modifican.
    - No se pierden las modificaciones existentes en ellos.
- **git reset --soft <commit>**
  - Se cambia HEAD para que apunte al commit dado.
  - No se borran los cambios creados en el staging area.
    - El staging area tambien reflejara los cambios existentes entre las versiones apuntadas por el HEAD anterior y el nuevo HEAD.
  - No se modifican el directorio de trabajo.
- **git reset --hard <commit>**
  - Cambia HEAD para que apunte al commit dado, restaura el staging area y restaura el directorio de trabajo eliminando todos los cambios existentes.

87

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 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.

- 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.

88

- git reset también se usa restaurar en el staging area la versión de un fichero(s) según estaba en un determinado commit:

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

  - Copia los paths del commit indicado en el staging area.

- Ejemplo:

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

    $ git reset HEAD -- readme.txt

89

git cherry-pick

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

  git cherry-pick <commit>

  - El commit especificado pertenecerá a alguna rama distinta de la mía.

- 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 o en los del stage.
  - Si exiten cambios sin congelar, el comando se queja y no hace nada.

90

git revert

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

  git revert <commit>

  - Permite deshacer alguna modificación que se hizo 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.

91

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 envian 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 recien incorporadas contribuciones estarán disponibles para que todo el mundo se las descargue.

- **git format-patch <commit>** crea un parche para cada commit de la rama actual posterior al commit dado.
  - 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.

93

- **git am <mbox>** aplica los parches incluidos en mbox a la rama actual.
  - El fichero mbox pueden contener varios mensajes, cada uno con un parche.
  - Si el parche se aplica con exito,
    - 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 afectado 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 atras y no aplicar ningun parche:
      - Ejecutar **git am --abort**. Se vuelve al estado inicial.

94

git archive

- El objetivo de este comando es crear un fichero con el contenido del repositorio.
- Ejemplo: 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.
- Puede especificarse el formato de compresión usando la opción **--format**

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

95

git describe

- El comando **git describe <commit>** se usa para generar números de versión.
- 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 numero 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

96

add interactivo

- Si hemos editado varios cambios
  - que pertenecen a contribuciones diferentes
  - y no queremos hacer un commit conjunto
- podemos hacer un add interactivo (**git add -i**)
  - nos 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>
```


97

Stashing

- Normalmente, mientras se está trabajando en una contribución, los ficheros del working directorio y el staging area tienen las modificaciones realizadas hasta el momento.
- Si fuera necesario hacer una modificación urgente relacionada con otra tarea, habría que sacar (chekout) los ficheros relacionados con la nueva tarea, hacer los cambios necesarios y congelar una versión.
  - Pero sin que se mezclen los cambios de ambas actividades.
- ¿Borramos antes nuestros cambios iniciales para evitar que todo se mezcle?
  - No es necesario.
  - Existen soluciones a este problema.

- Solución 1:
  - Hacer un nuevo clon del repositorio, y realizar la modificación urgente sobre el nuevo clon.
  - Esto puede ser poco aceptable si el proyecto es muy grande.
- Solución 2:
  - Usar el comando **git stash** 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.

99

- Para meter un stash en la pila, es decir, para salvar las todas modificaciones existentes en la pila, 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 se identifica por el nombre stash@{#}.

100

- Aplicar el último stash a la rama actual

  git stash apply

- 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 reactualiza 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}

102

Submódulos

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

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.
- Buscar un string
  - Para buscar hacia atrás en todas las diferencias de un fichero en busca de un string dado:
  - git log -S<string> <filename>

104

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
  - ...

105

Temas pedientes

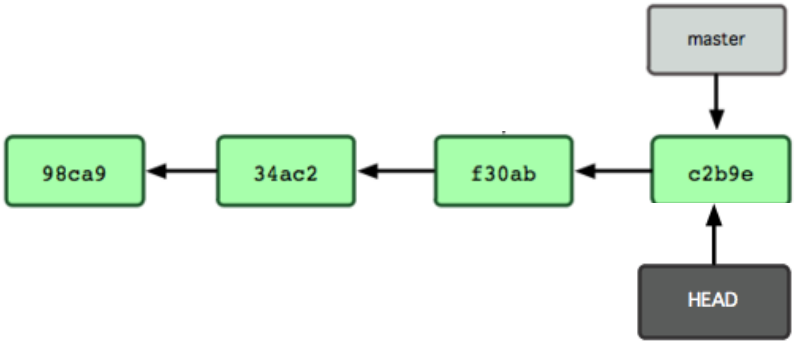
- Reflog
- Atributos
- Hooks
- ...

106

Checkout: volver a commit anterior

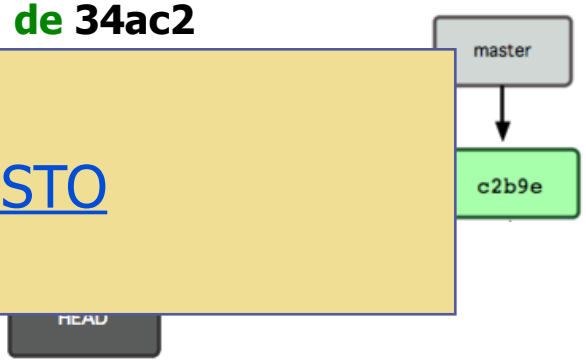
 # "git checkout <commit_id>" restaura el área de trabajo de commit anterior
#
 # los 5-6 primeros dígitos de <commit_id> son suficientes para identificarlo

$ git log --oneline # lista de commits: 5-6 primeros dígitos de <commit_id>
c2b9e migración base de datos
f30ab creación de scaffold Type
34ac2 añadir ejemplo
98ca9 vistas index y contact



$ git checkout 34ac2 # restaura 34ac2 'añadir ejemplo', con
 # 1) Puntero HEAD a 34ac2
 # 2) Working-area de 34ac2

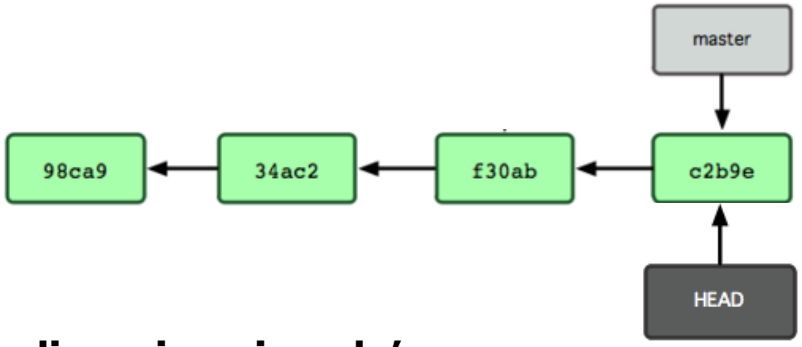
DETACHED HEAD - NO CONTAR ESTO



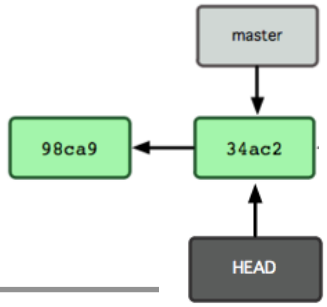
107

Reset: Eliminar commits

```
# "git reset <commit_id>"
#    -> elimina rama del árbol y deja cambios en staging-area
#    -> opción --hard elimina cambios completamente

planet$> git log --oneline    # lista commits
c2b9e migración base de datos
f30ab creación de scaffold Type
34ac2 añadir ejemplo
98ca9 vistas index y contact
```




```
planet$> git reset 34ac2   # restaura 34ac2 'añadir accion ejemplo', con
                           #   1) Punteros HEAD y master a 34ac2
                           #   2) Working-area de 34ac2
                           #   3) Staging-area con cambios de commits hasta master

planet$> git reset --hard 34ac2  # restaura 34ac2 'añadir ejemplo', con
                                 #  1) Punteros HEAD y master a 34ac2
                                 #  2) Working-area de 8b511bd con staging-area vacía

                                 #      ¡OJO! con  "commit reset --hard ..."
                                 # se pierden los cambios de commits eliminado
```




```
planet$>
```


108

Gestion del directorio de trabajo

```
# Otros comandos de interés para gestionar el directorio de trabajo

planet$> rm CharIO.java                  # borra fichero de directorio de trabajo,
                                         # pero sigue en versión

planet$> git rm CharIO.java              # rm: borra fichero de la próxima versión

planet$> git rm --cached CharIO.java     # rm: borra fichero de staging area

planet$> git mv file_from file_to        # mv: mueve o cambia de nombre un fichero

planet$> git commit -a -m 'prueba 1 e/s'   # Opción -a: realiza commit con todos
                                           # los ficheros modificados que están en estado modified
                                           # Equivalente a hacer "add" de lo modificado + "commit"

# "git reset HEAD <fichero>" elimina <fichero> de staging área

planet$> git reset HEAD benchmarks.rb

planet$>
```


109

Repositorios Remotos

- GIT permite manejar **repositorios remotos**
  - Son la base para la colaboración entre los miembros de un equipo
    - Un repositorio remoto se puede
      - clone: Clonar
      - fetch, merge: integrar con nuestro repositorio local
      - pull: contribuir a un repositorio remoto
      - .....
  - GITHUB: es una red social de gestores web de repositorios remotos

- Los repositorios remotos se acceden con 3 protocolos
  - git: git@github.com:jquemada/swcm2012.git
  - ssh: ssh://github.com/jquemada/swcm2012
  - https: https://github.com/jquemada/swcm2012

110

GITHUB: clonar un repositorio remoto

```
# GITHUB es un servidor de repositorios de grupo remotos, están por ejemplo:
#
#    -> Ruby on Rails:  https://github.com/rails/rails
#    -> Linux:          https://github.com/torvalds/linux
#    -> Eclipse:        https://github.com/eclipse
#    -> SWCM:           https://github.com/ging
#    ............
#
# Clonar un repositorio remoto es trivial:

xx> git clone https://github.com/ging/swcm
                        # clone: crea rama de proyecto en directorio local swcm
.......
xx> cd swcm             # Entramos en directorio clonado swcm

swcm> git remote        # remote:ver repositorios remotos definidos
origin                  # "origin" referencia el repositorio clonado

swcm> git remote -v  # con -v muestra tambien el URL del repositorio remoto
origin  https://github.com/ging/swcm
.......

swcm> git clone https://github.com/ging/swcm   my_swcm  # copia repositorio
.......                          # ging/swcm  de GITHUB a directorio local my_swcm
```


111

Repositorios remotos

```
# Otros comandos sobre repositorios remotos de interés son

swcm> git remote rename jq juan # cambia nombre de repositorio remoto

swcm> git remote rm  juan          # elimina referencia a repositorio remoto

swcm> git push origin master       # push: actualiza en "origin" últimos cambios
                                   # de master si no hay conflicto

swcm> git fetch origin    # fetch: traer todos los commit de un repositorio
                          # remoto a nuestro repositorio, los integra como ramas


swcm> git merge origin  # merge: unir 2 o mas ramas, si no hay conflicto


swcm> git pull origin     # pull: hacer un fetch de otro repositorio remoto
                          # e integrar los cambios de la rama master
                  # si hay conflictos: los integra en los ficheros del
                  # working directory p(ara resolverlos con editor)


swcm> git cherry-pick <commit_id>  # cherry-pick: unir una rama y un commit
```


112