



**manexware**

## **Gestión de Proyectos Software con Git y Github**

---

- La finalidad de ..... es que los estudiantes, organizados en equipos de trabajo, adquieran las competencias establecidas en el plan de estudios mediante el desarrollo de un proyecto común para todas las asignaturas, trabajando de forma similar a como lo harían en el mundo profesional.
- Dentro de esa espiral de mejora continua, uno de los aspectos de la metodología que ha sufrido diversos cambios es la herramienta empleada para la gestión del trabajo colaborativo.
- Para el desarrollo de un proyecto de este tipo, resulta esencial disponer de una herramienta que permita alojar todo el trabajo realizado por los grupos en sus proyectos y que proporcione mecanismos de gestión y seguimiento tanto para los estudiantes como para el profesorado.
- Y al tratarse de un proyecto de software, uno de los aspectos fundamentales para el desarrollo adecuado del proyecto es disponer de un Sistema de Control de Versiones (SCV) que permita gestionar los cambios que se producen de forma continua en el código de un proyecto (Control de versiones, 2017).
- Por ello, **todos los años** se plantea el uso de una herramienta de este tipo que tenga incorporado un sistema de control de versiones.

- GitHub es una plataforma de desarrollo colaborativo de software para alojar proyectos que usa el sistema de control de versiones Git, uno de los SCV más empleados en la actualidad tanto de forma particular como profesionalmente (GitHub, 2017).
- GitHub emerge como una herramienta para el aprendizaje que te proporciona por un lado un repositorio para el código y herramientas muy útiles para el trabajo en equipo y por otro como una herramienta docente que facilita la monitorización de un proyecto de software (Lopez-Pellicer, Béjar, Latre, Nogueras-Iso y Zarazaga-Soria, 2015).

- El objetivo principal perseguido con el uso de GitHub es el empleo de una herramienta común a todas las asignaturas que permita albergar todo el trabajo realizado en los proyectos y que sea útil para estudiantes y profesores.
- Este objetivo principal se puede dividir en una serie de objetivos más concretos en función de los dos actores principales que participan en el ABP de Multimedia, los estudiantes organizados en equipos de trabajo, y el profesorado.

- Para los estudiantes:
  - Disponer de un repositorio organizado de todo el código fuente desarrollado para su proyecto.
  - Poder emplear el sistema de control de versiones integrado Git para gestionar todos los cambios producidos en el código del proyecto.
  - Facilitar la gestión del trabajo del grupo en el proyecto, en aspectos como la comunicación, reparto de tareas, las entregas, ...

- Para el profesorado:
  - Poder realizar un seguimiento adecuado del desarrollo del proyecto, mediante el acceso sencillo a documentos y código, la visión de gráficas de uso y progreso, ...
  - Disponer de mecanismos para proporcionar una retroalimentación cualitativa y cuantitativa adecuada a los proyectos y grupos.

# Experiencia


---



- Usar esta opción con el propósito de albergar los proyectos del curso 2020. Para ello se debe efectuar los siguientes pasos:
  1. Creación de una organización en el sitio web de GitHub.
  2. Solicitud de registro en GitHub a estudiantes y al profesorado en el caso de no disponer ya de un usuario en la herramienta.
  3. Creación de los equipos de trabajo (Teams en GitHub) de los estudiantes y el correspondiente al profesorado.
  4. Creación de los repositorios correspondientes a los proyectos del curso (Repositories en GitHub), y asignación a cada uno de ellos de su equipo de estudiantes responsable y del equipo de profesores.



# Github en la vida real



## ABPMultimediaUA

Repositorio padre de los proyectos ABP de 4º del Grado en Ingeniería Multimedia Curso 2016/17

Repositories

**lab21** Private

[ABP16] Juego de acción/ciencia ficción que se desarrolla en un laboratorio secreto subterráneo.

C++ Updated 5 hours ago

**Boom-it** Private

[ABP16] Juego estratégico laberíntico multijugador al estilo Bomberman.

C++ ★ 2 Updated a day ago

**Sack-3D** Private

[ABP16] Juego mezcla de plataformas y shooter en 2.5D multijugador frenético.

C++ Updated 4 days ago

**code** Private

[ABP16] Juego con tematica Survival Horror ambientado espacial o futurística

HTML Ψ 1 Updated 2 days ago

Teams 43

☐ **[ABPCE] ASYNC GAMES**  
[ABP] Games Team

☐ **[ABPCE] Dire Wolf Games**  
[ABP] Games Team

☐ **[ABPCE] Ikigai**  
[ABP] Games Team

☐ **[ABPCE] Paradox Studios**  
[ABP] Games Team

☐ **[ABPCE] Rorschach Games**  
[ABP] Games Team

☐ **[ABPCE] SkyScrapers**  
[ABP] Games Team

☐ **[ABPCE] Xplosion Studio**  
[ABP] Games Team

- Desde la perspectiva de los estudiantes organizados como equipo de trabajo, GitHub ofrece muchas prestaciones útiles para el trabajo colaborativo. Entre ellas se destacan a continuación las más empleadas por los grupos en el presente curso:
  - **Código organizado en Repositorio:** cada equipo dispone de un repositorio para su proyecto donde almacena todo el trabajo desarrollado para todas las asignaturas. Para ello, al inicio del curso se estableció una estructura fija formada por tres carpetas: Desarrollo, para guardar todo el código fuente implementado; Documentación, para incluir todos los documentos e informes generados; y Entregas, para contener todos los entregables creados para las asignaturas en las fechas de entrega establecidas.

- **Control de versiones Git:**

- GitHub integra el SCV Git, el más empleado en la actualidad. Git es un sistema distribuido que tiene como característica principal la ramificación. Las ramas se usan para gestionar los cambios de código de forma más fácil. En este sentido, GitHub facilita un visor de ramas donde se pueden comparar los progresos realizados en las distintas ramas de nuestro repositorio.

- **Entrega mediante Issues:**

- Los issues o peticiones en GitHub representan la forma de realizar el seguimiento de tareas, mejoras o errores que suceden en un proyecto. A cada issue se le pueden asociar etiquetas, para categorizarlos y filtrarlos; hitos, para relacionarlos con funcionalidades específicas o fases importantes del proyecto; y comentarios, para que todo aquel con acceso al repositorio pueda proporcionar retroalimentación. El uso principal de los issues ha estado relacionado con las entregas: para cada hito establecido los equipos de trabajo realizaban la entrega mediante un issue, de forma que los entregables fueran fácilmente accesibles por el profesorado.

- Desde el punto de vista del profesorado, GitHub presenta diversas características que facilitan la monitorización del progreso de los proyectos desarrollados por los equipos de trabajo. A continuación se resaltan las más utilizadas en el presente curso:
  - **Seguimiento de código en Repositorio:**
    - El equipo de profesores está asignado a todos los repositorios creados en GitHub y en consecuencia todos sus miembros tienen acceso a todo el código desarrollado, la documentación generada y las entregas realizadas.

- Gráficos del proyecto:
  - GitHub proporciona varios tipos de gráficos que permiten visualizar el trabajo desarrollado en el proyecto por sus diferentes componentes. Los gráficos más interesantes son el de contribuciones, que muestra las aportaciones (commits, issues,...) realizadas al repositorio por cada miembro; el de commits, que presenta el total de commits realizados en el proyecto por semana; y el de tarjeta perforada (punch card), que visualiza los commits hechos por día de la semana y hora.

# Profesorado: GitHub como herramienta de monitorización



ABPMultimediaUA / Boom-it Private

<> Code **Issues 14** Pull requests 0

Filters

☐ **14 Open** ☒ 0 Closed

- ☐ **Entrega Hito 3**  
#14 opened 12 days ago by Ryasu
- ☐ **Evaluación Proyectos Multimedia Hitos 0, 1**  
#13 opened on 2 Mar by villagra-abp
- ☐ **Entrega Hito 2**  
#12 opened on 31 Jan by Ryasu
- ☐ **Entrega Tema 9 PM**  
#11 opened on 12 Jan by Ryasu

## Entrega Hito 3 #14

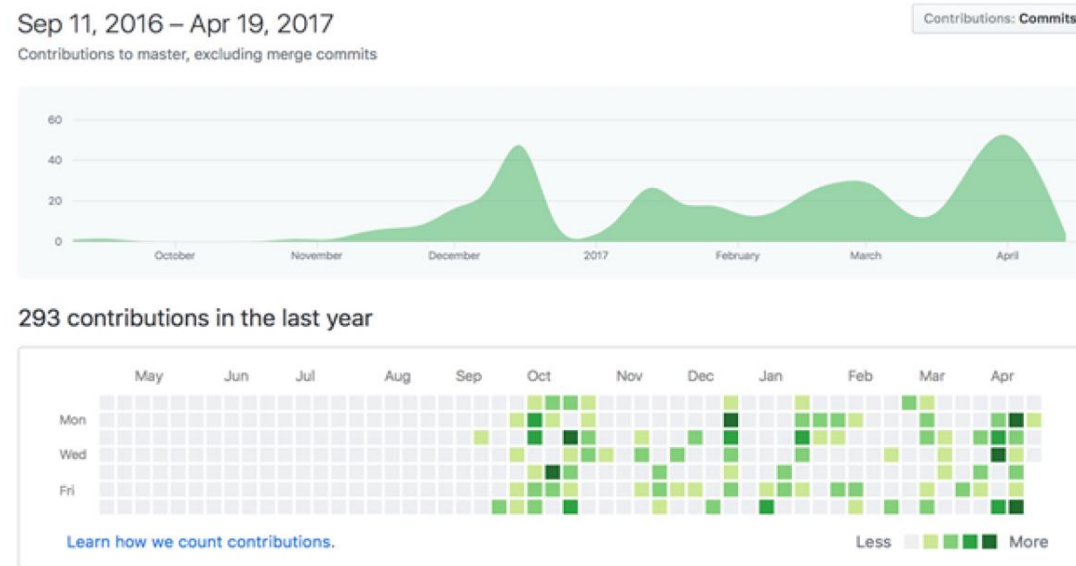
**Open** Ryasu opened this issue 12 days ago · 0 comments



Ryasu commented 12 days ago

- Presupuesto  
[Presupuesto ABP](#)
- Postproducción Digital  
[Crear tráiler del juego](#)
- Videojuegos 1
  - V1.01 - Documento de diseño de mecánicas de los NPCs
  - V1.02 - Documento de diseño de sistemas de toma de decisión
  - V1.03 - Documento de diseño técnico de la arquitectura de la IA
  - V1.07 Sistema de toma de decisión con Árboles de Decisión - No entregado
  - Entrega V1.19 - V1.22 - V1.25
- Videojuegos 2
  - V2.13 - Power-ups y elementos adicionales
  - V2.17 Diseño y creación de niveles
  - V2.25 y V2.26 Implementación de menú e implementación de HUD
  - V2.27 Implementación de clipping (comparativa de FPS con/sin oclusiones) - No entregado
- Proyectos Multimedia  
[Entrega Hito 3 - Proyectos Multimedia](#)
- Realidad Virtual
  - Items 2 y 3: Modelado y texturizado de los personajes
  - Items 8 y 9: Modelado y texturizado del entorno y de los elementos
- Técnicas Avanzadas de Gráficos  
[Entregables 4 al 15 - Técnicas Avanzadas de Gráficos](#)

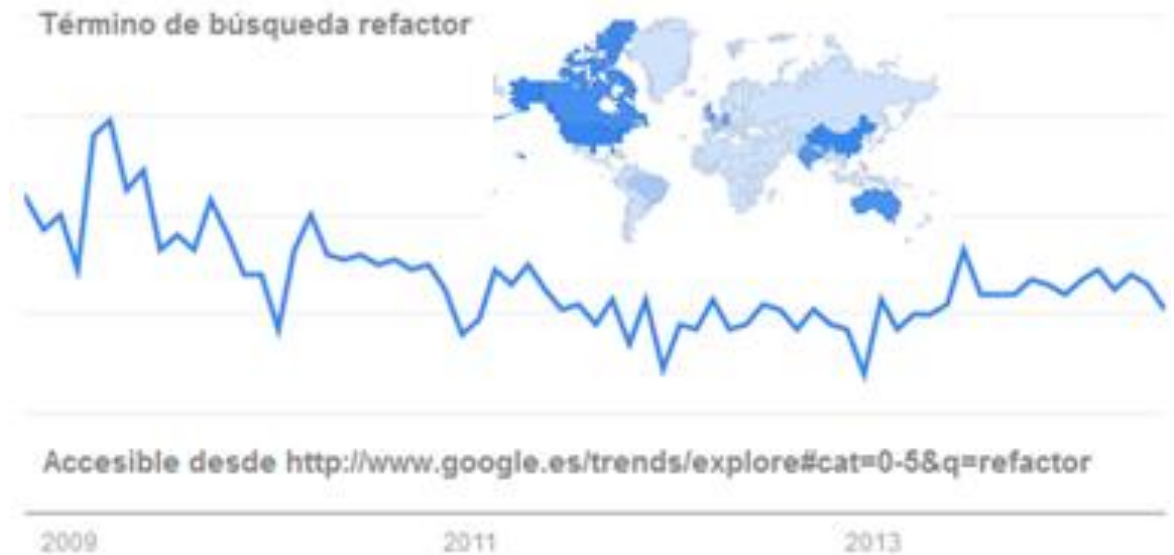
- En la figura 3 se muestra una gráfica ejemplo de las contribuciones realizadas (evolución del número de commits realizados al día desde el inicio del proyecto hasta el momento actual) al repositorio por uno de los grupos del curso 2016/17, y las individuales de uno de sus miembros (número de commits por mes y día de la semana: la mayor tonalidad del color verde es indicativa de un número mayor de contribuciones).





- En el área de conocimiento de la ingeniería del software (Bourque & Fairley 2014) las tareas de refactorización de código se encuentran dentro del sub área de mantenimiento del software.
- La refactorización es un tipo de tarea de corrección que se define como: una transformación del programa aplicada en fase de mantenimiento que no cambia su comportamiento externo (Fowler *et al.*, 1999).
- Los motivos para refactorizar son la mejora de algún atributo de calidad. En el contexto de refactorización también se describe y categoriza el concepto de defecto de código (*bad smell*). Es decir, el proceso de refactorización se inicia para eliminar una instancia concreta de un defecto de código y mejorar así algún atributo de calidad. El resultado de una tarea de refactorización es un cambio en el código.
- Desde un punto de vista de ingeniería del software, la gestión de cambio está definida en el área de conocimiento de gestión y configuración del software.

- Parece interesante poder definir actividades de aprendizaje que tengan en cuenta esta relación. Actualmente las tareas de refactorización son utilizadas en los equipos de desarrollo empresarial, y gozan de una gran popularidad entre los desarrolladores, especialmente en los principales países exportadores de desarrollos software (ver Fig.).



- Además tienen una multitud de documentación, tanto *online* y libre (catálogo de M. Fowler en [refactoring.com](http://refactoring.com)), como bibliográfica (Brown, 1998), (Fowler *et al.*, 1999), (Lippert & Roock, 2006), (Wake, 2004).

- El problema planteado es que ninguna de la bibliografía mencionada tiene un enfoque puramente pedagógico, donde se pueda guiar el proceso de enseñanza y aprendizaje utilizando en menor medida las metodologías centradas en el profesor (caracterizadas como expositivas y pasivas), para ir evolucionando hacia metodologías y actividades centradas en el estudiante (activas, dinámicas y participativas) (Meneses, Fernández & Ballesteros-Regana, 2011). Según (Cabero & Román, 2005), las actividades para el aprendizaje se refieren a las diferentes acciones que los estudiantes llevan a cabo en completa relación con los contenidos e informaciones que les han sido ofrecidos. Si estas actividades son presentadas, realizadas o transferidas a través de la red, entonces se pueden considerar como e-actividades.

- Los repositorios de proyectos, como GitHub, pueden ayudar a desarrollar **e-actividades** de refactorización y favorecer la interacción de los estudiantes.
- La utilización de repositorios de control de versiones en docencia busca la mejora del proceso de enseñanza/aprendizaje.

- Antes de abordar su planificación, es necesario tener claro el conjunto de e-actividades a utilizar en la asignatura.
- A continuación se realiza la propuesta de las mismas, para posteriormente encuadrarlas en el proceso de enseñanza/aprendizaje.

- **Lectura.**

- Las lecturas son los textos y la documentación escrita, tanto de obras completas como fragmentos que se han recogido y editado como fuente de profundización en los contenidos trabajados.

- **Ejemplos.**

- Se trata de la exposición de ciertos elementos que, por analogía, pueden transferirse a otros contenidos o situaciones. Los ejemplos facilitan la comprensión de las informaciones y de los contenidos de aprendizaje, conocimientos, ideas o procedimientos. El abanico de objetivos que permiten trabajar es muy amplio, pero normalmente por sí solos los ejemplos no aseguran la consecución de ninguno de ellos. En este sentido, se caracterizan porque actúan como complementos de otros recursos.



- **Pruebas objetivas.**

- Son instrumentos de medida, elaborados rigurosamente, que permiten evaluar conocimientos, capacidades, destrezas, rendimiento, aptitudes, actitudes, inteligencia, etc. Suelen ser un recurso utilizado para la evaluación diagnóstica, formativa y sumativa. Podemos encontrarlas de naturaleza variada como por ejemplo: de respuesta breve, de completar, de discriminación, de ordenación, de localización o identificación.

- **Aprendizaje basado en problemas.**
  - Es una técnica en la cual el estudiante debe resolver una situación problemática concreta (que puede tener más de una posible solución), a partir de los contenidos que se han trabajado.

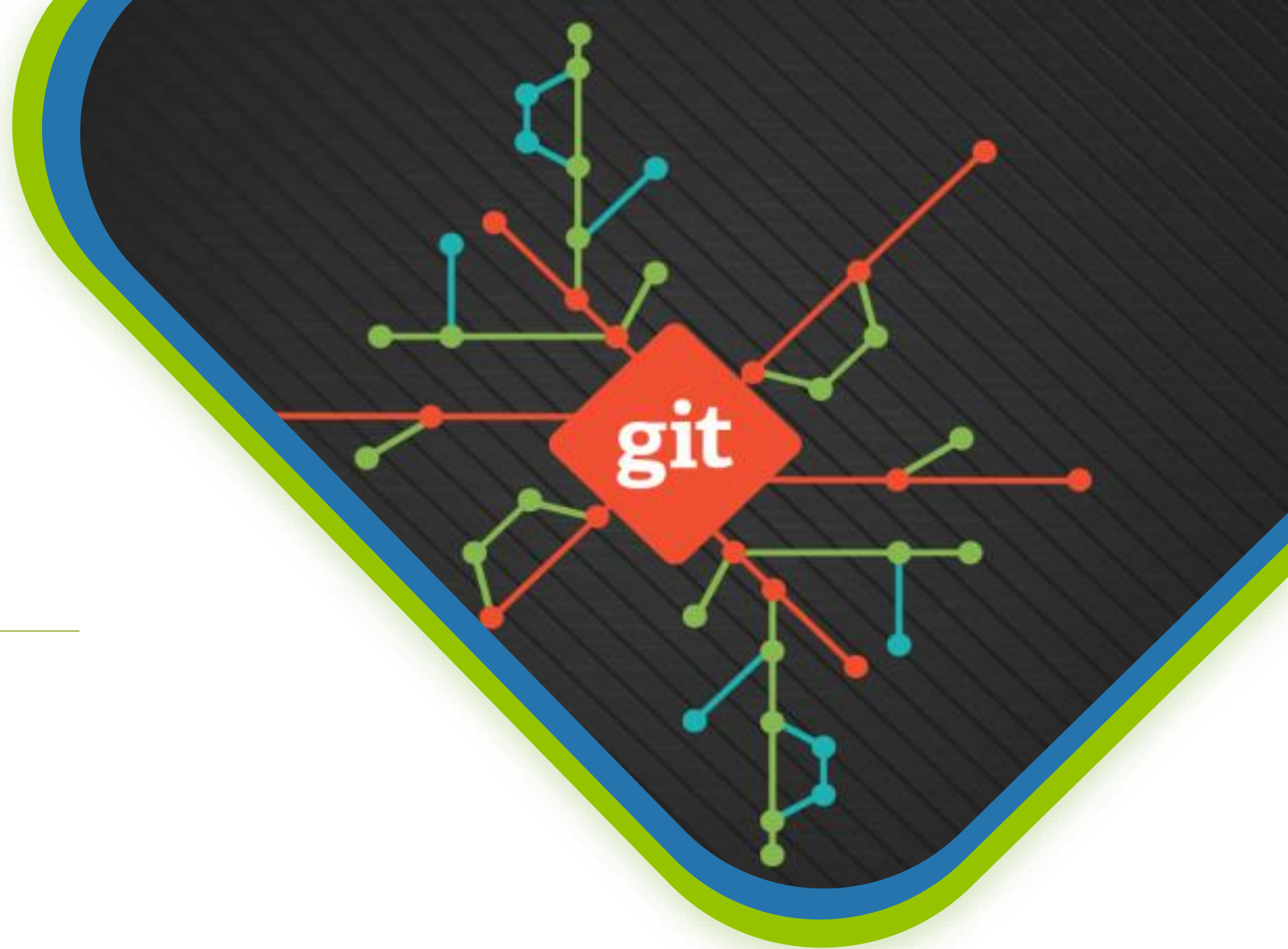
- **Estudio de caso (*Case-Based Learning*).**
  - El estudio de caso se basa en la presentación de una situación problemática, real o ficticia, que el estudiante tiene que resolver. Las competencias sobre las que se trabaja son: identificación y análisis de factores externos e internos, planificación de procesos y toma de decisiones argumentada.

- Seminario virtual.
  - Es una técnica de grupo que promueve el estudio intensivo de un tema. Se caracteriza por la discusión, la participación, la elaboración de documentos y las conclusiones compartidas por todos los componentes del seminario. Su objetivo es explorar sobre un tema concreto, reflexionar sobre un tema específico, transmitir informaciones.

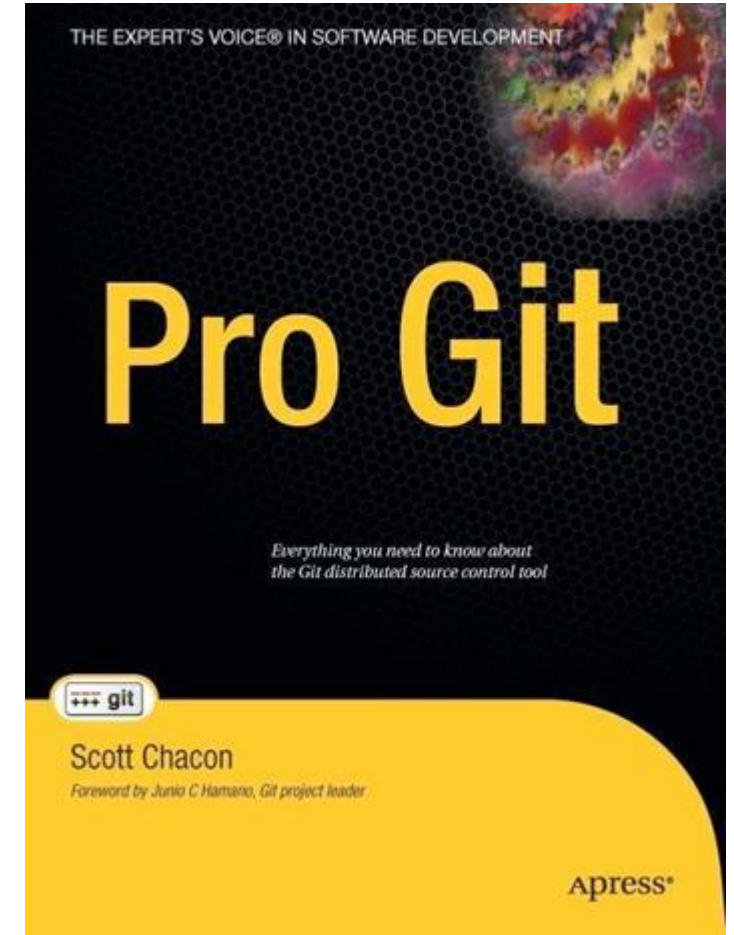
## PARTE 01


### Introducción a 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>



 **git** --everything-is-local

Search entire site...

**About**  
**Documentation**  
Reference  
**Book**  
Videos  
External Links  
**Downloads**  
**Community**


This book is available in [English](#).  
Full translation available in  
[български език](#),  
[Deutsch](#),  
[Español](#),  
[Français](#),  
[Ελληνικά](#),  
[日本語](#),

## Book

The entire Pro Git book, written by Scott Chacon and Ben Straub and published by Apress, is available here. All content is licensed under the [Creative Commons Attribution Non Commercial Share Alike 3.0 license](#). Print versions of the book are available on [Amazon.com](#).



### 1. Getting Started


- [1.1 About Version Control](#)
- [1.2 A Short History of Git](#)
- [1.3 What is Git?](#)
- [1.4 The Command Line](#)
- [1.5 Installing Git](#)
- [1.6 First-Time Git Setup](#)
- [1.7 Getting Help](#)
- [1.8 Summary](#)



2nd Edition (2014)

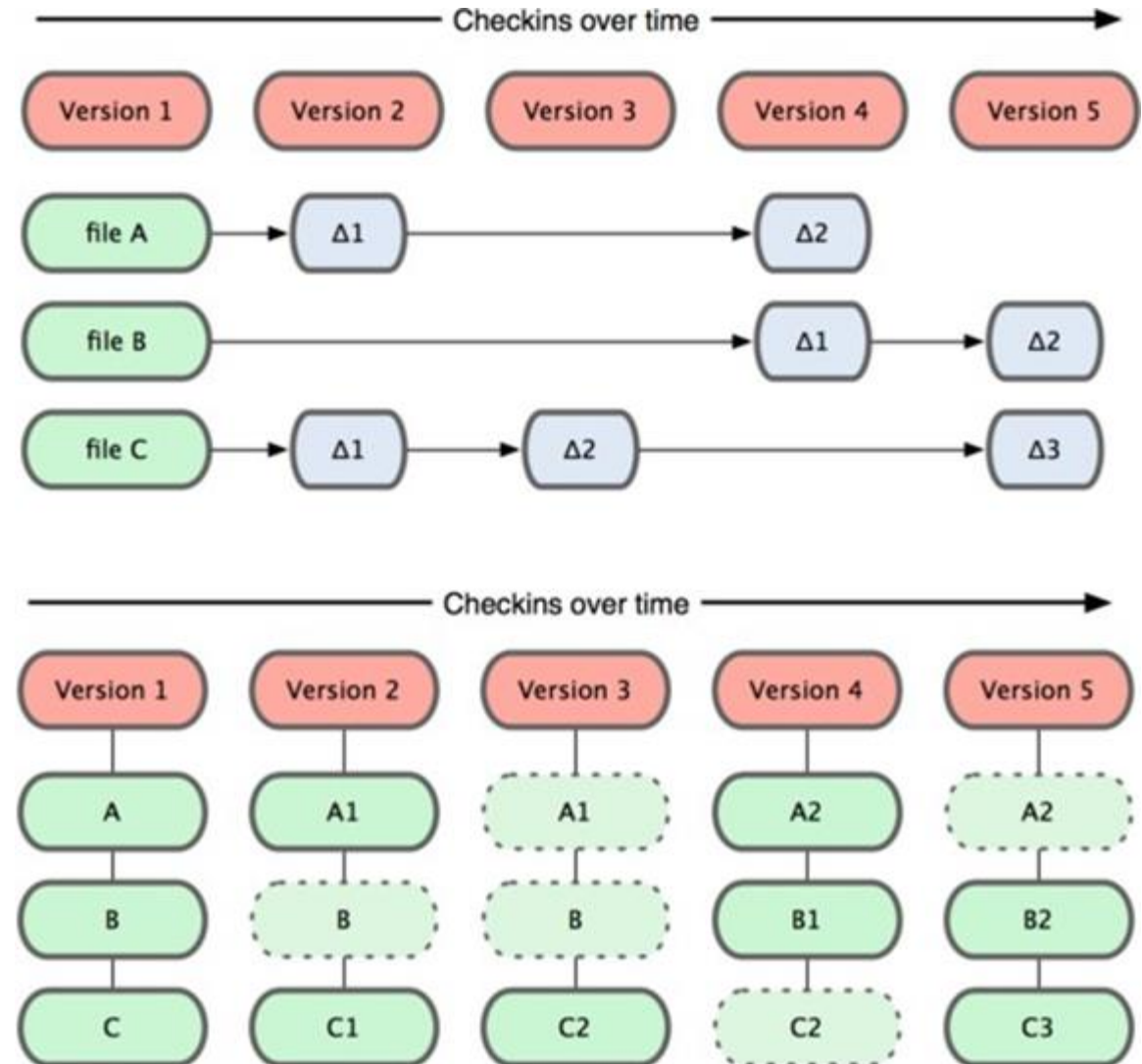
### Download Ebook

 pdf  epub

 mobi

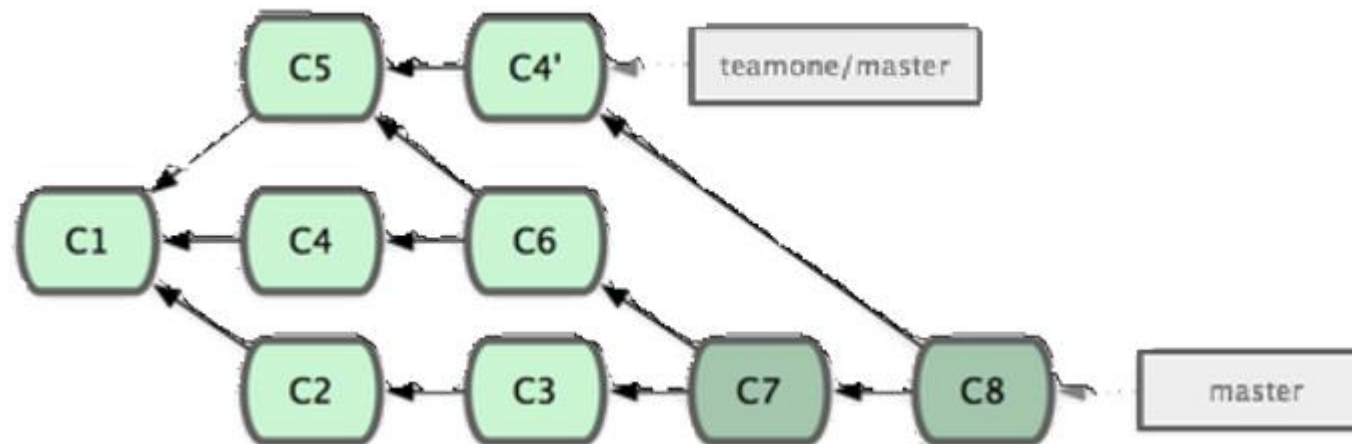
# Historia de un proyecto

- 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



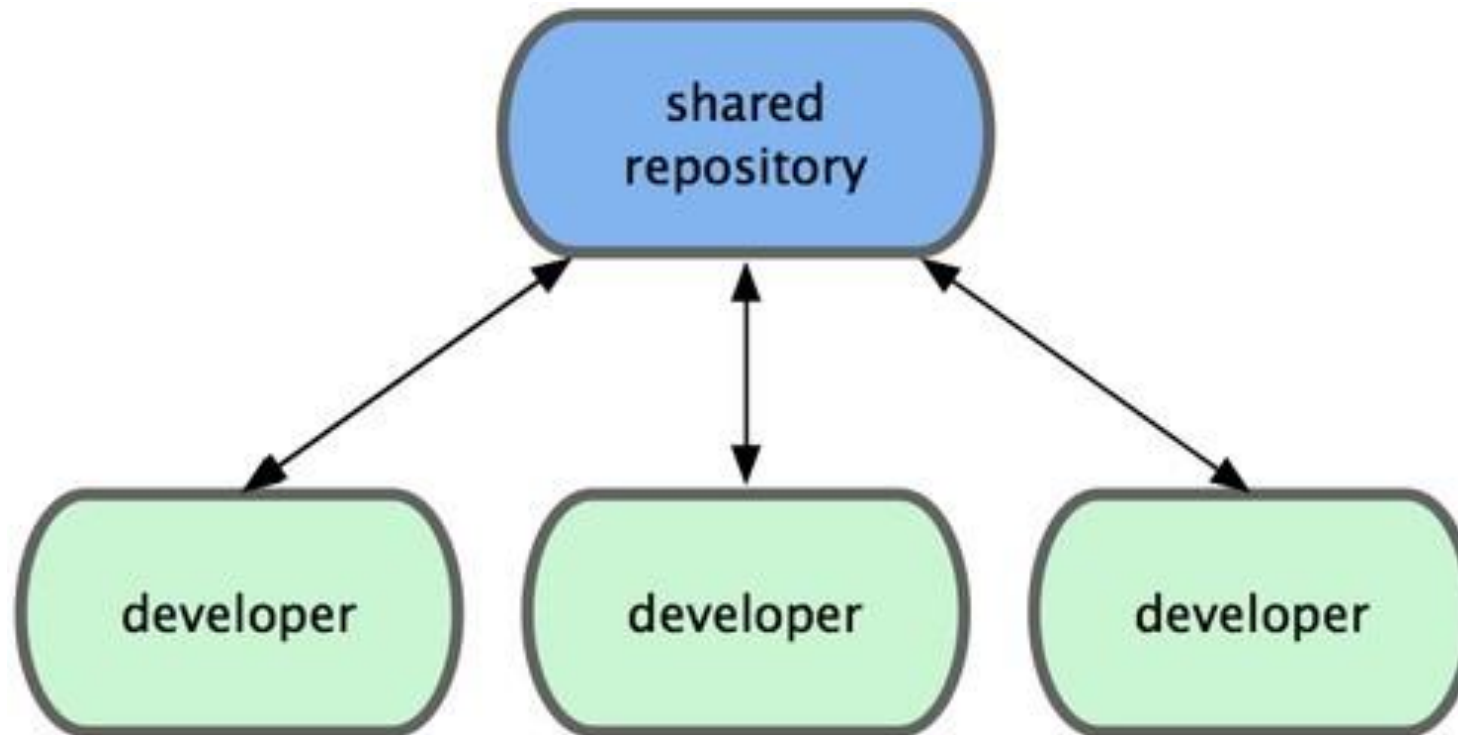


- Proyectos software en equipo son complejos.
- Múltiples desarrollos en paralelo.
  - nuevas funcionalidades, corrección de errores, mejoras, ...
- Cada desarrollo es una rama del árbol.
  - Ramas con cambios estables se integran (mezclan) en la rama principal.



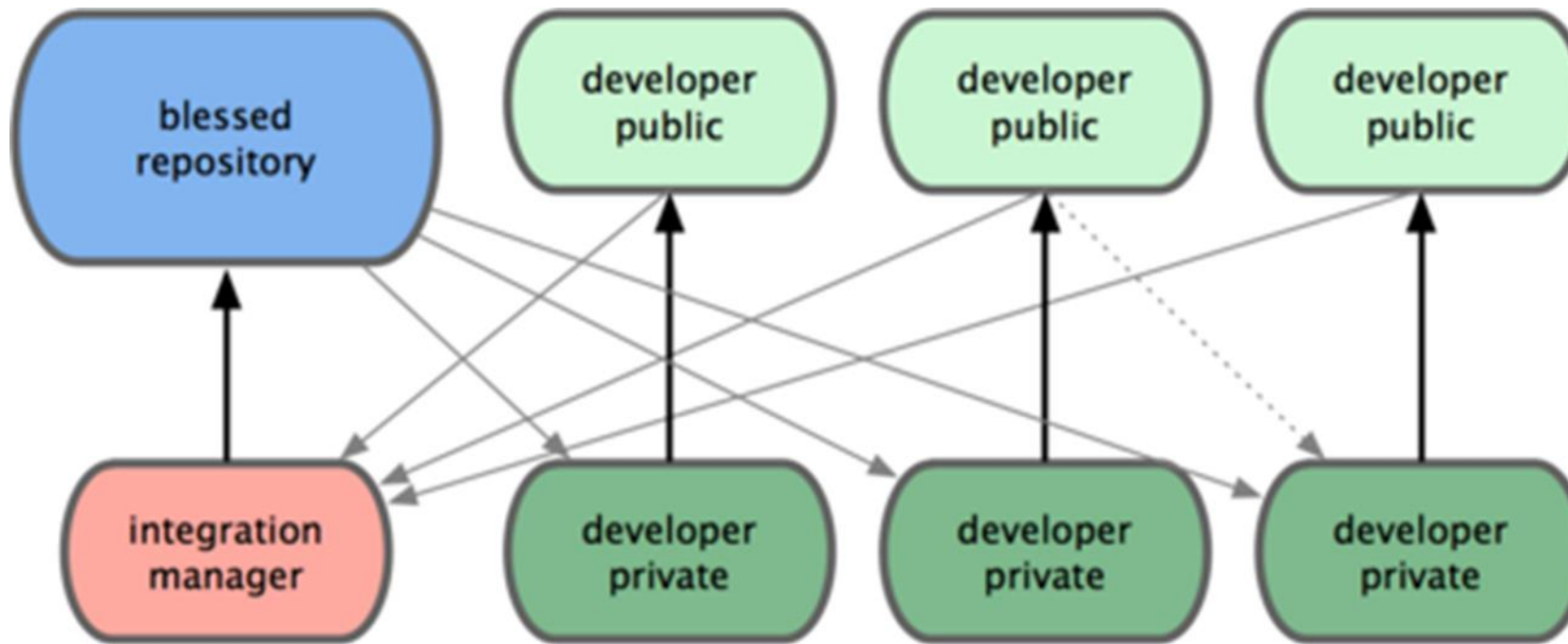
\*de Scott Chanson: <http://git-scm.org/book/>

- Flujo de trabajo centralizado



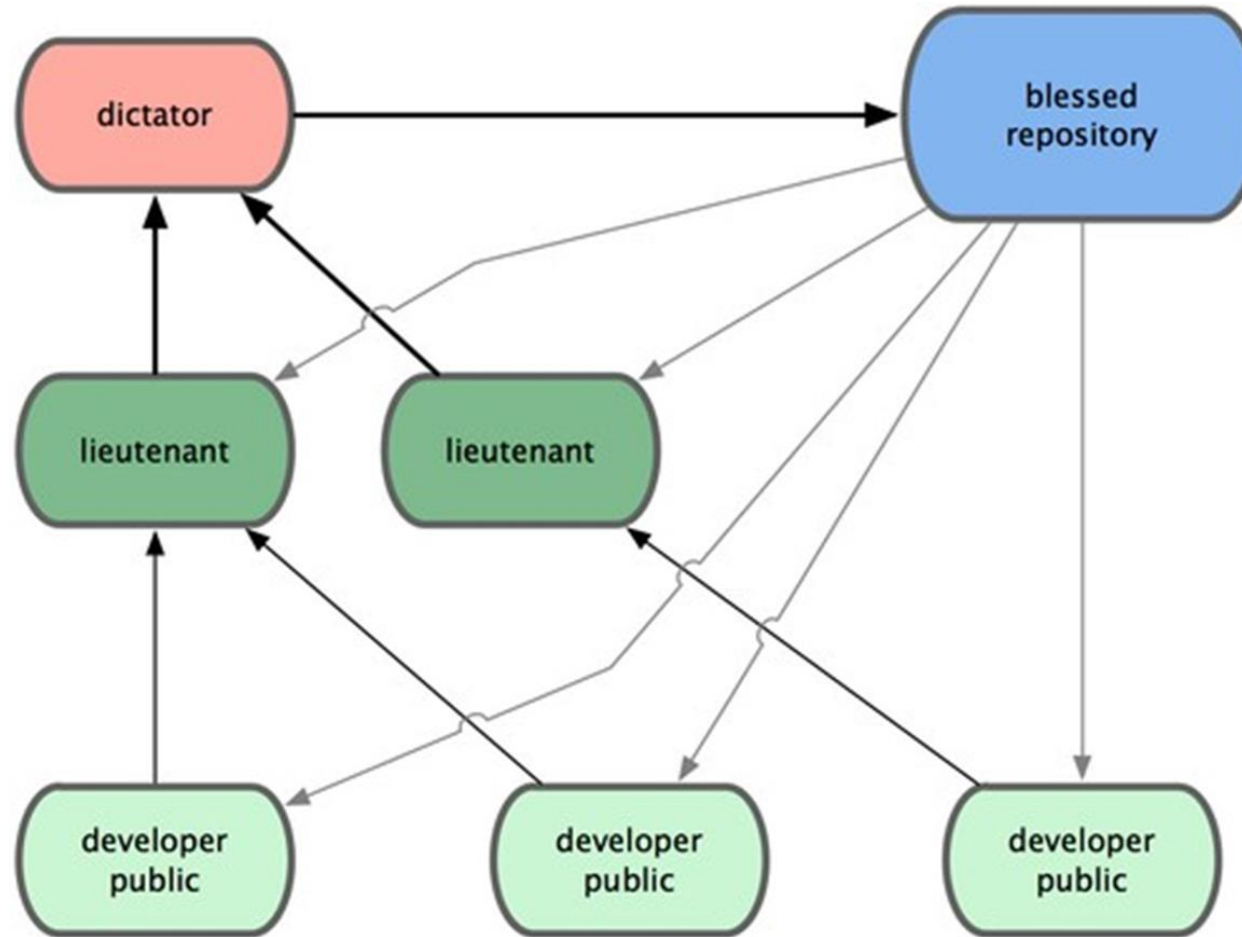
\*de Scott Chanson: <http://git-scm.org/book/>

# 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/](http://git-scm.org/book/)

- Los repositorios pueden estar alojados en hosts que tenemos que configurar.
  - Configuración de los demonios.
  - Protocolos de acceso.
    - local: file:///home/mangrovex/demo.git
    - git: git@github.com:mangrovex/demo.git
    - ssh: ssh://github.com/mangrovex/demo
    - http(s): https://github.com/mangrovex/demo
  - 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, ...

- 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>

# 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>

# Common Git Commands

## PARTE 02

### Comandos básicos de GIT

---



- `$git config`
- `$git init`
- `$git clone <path>`
- `$git add <file_name>`
- `$git commit`
- `$git status`
- `$git remote`
- `$git checkout <branch_name>`
- `$git branch`
- `$git push`
- `$git pull`
- `$git merge <branch_name>`
- `$git diff`
- `$git reset`
- `$git revert`
- `$git tag`
- `$git log`



**# 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>**

## # 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, pollas, 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](https://na1.salesforce.com/help/doc/en/salesforce_git_developer_cheatsheet.pdf)

# 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](mailto: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](mailto:pramirez@dit.upm.es)**

**color.ui=true**

# Consultar el valor de una opción:

\$ **git config user.name**

**Pedro Ramirez**

# 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

# 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.

# 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

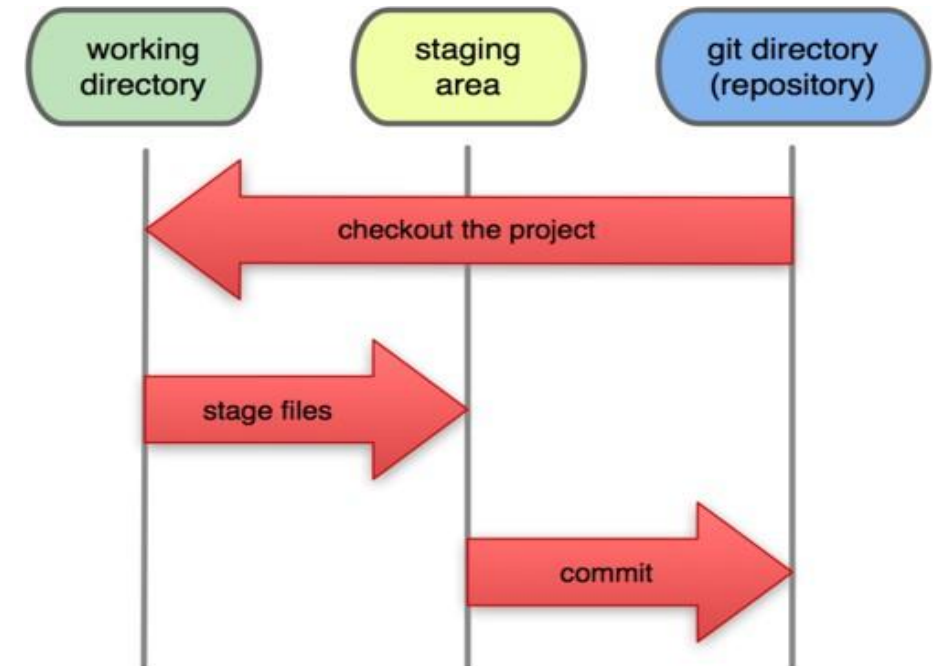
## ◆ Area de cambios (staging area, index)

- ★ 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, ...

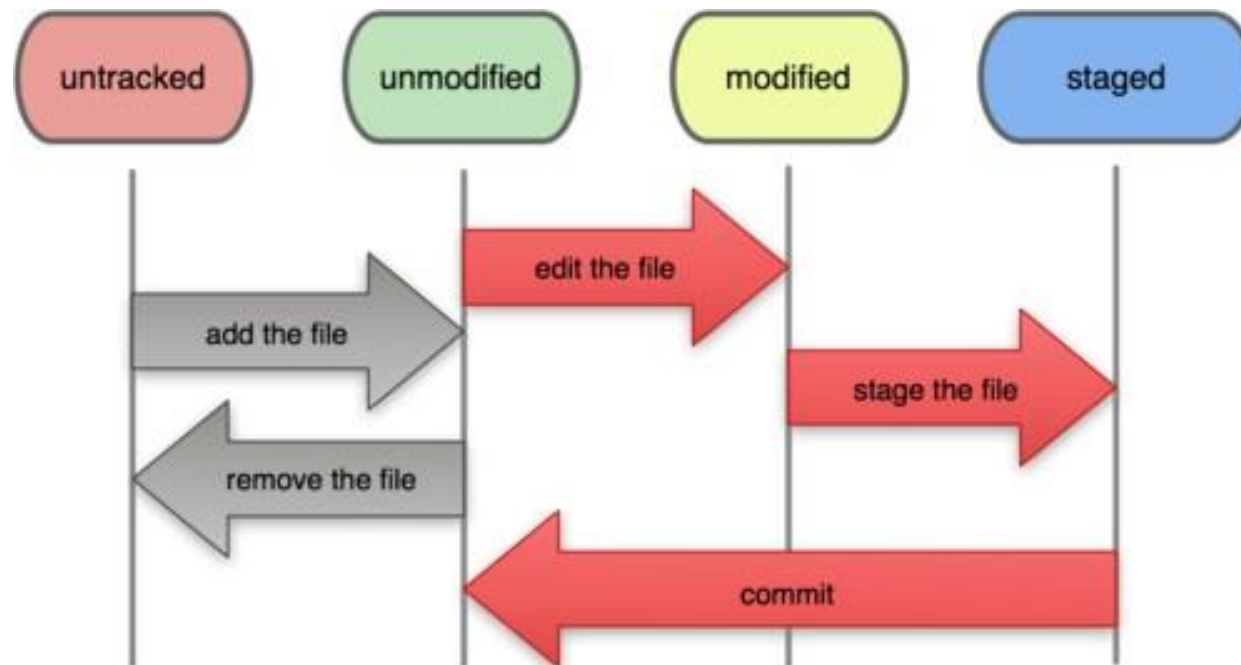
(El repositorio, el staging area y otros datos se mantienen en el directorio .git)



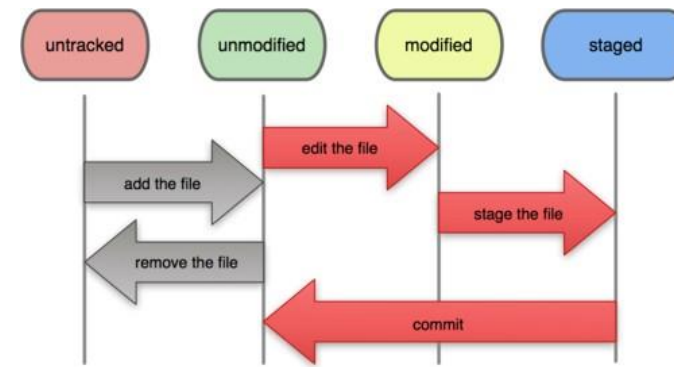
\*de Scott Chanson: <http://git-scm.org/book/>

# 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 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
```

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

# **.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 añadirá al staging area.

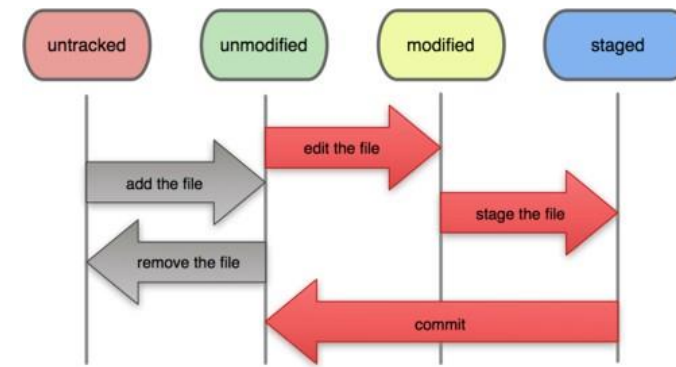
# Los ficheros **.gitignore** pueden crearse en cualquier directorio del proyecto,  
# y afectan a ese directorio y a sus subdirectorios.

# 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

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

# 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
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
```

# modificaciones en benchmarks.rb

# rango de líneas con cambios

# contenido no modificado

# líneas eliminadas empiezan por "-"

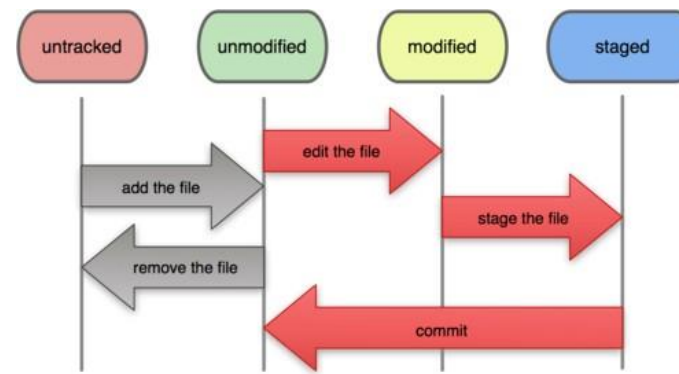
# líneas añadidas empiezan por "+"

# contenido no modificado

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
    @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
```

# rango de líneas con cambios

# contenido no modificado

# líneas eliminadas empiezan por "-"

# líneas añadidas empiezan por "+"

# contenido no modificado

# 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
```

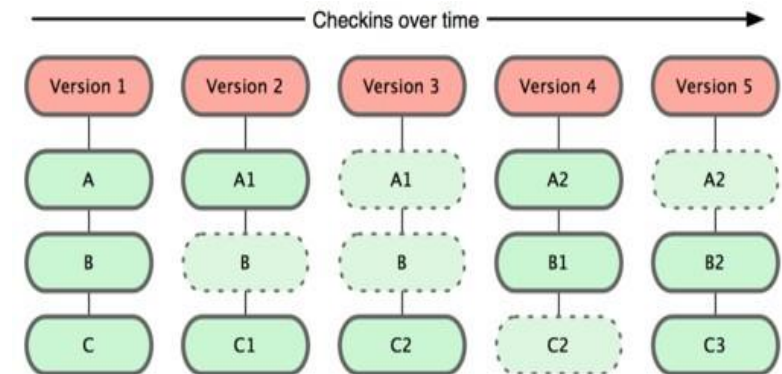
rails generate controller Planet index contact

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

new planet creado

\$ git log --online # Muestra un log resumido. Una línea por commit

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



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

# Muestra diferencias entre commits

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.

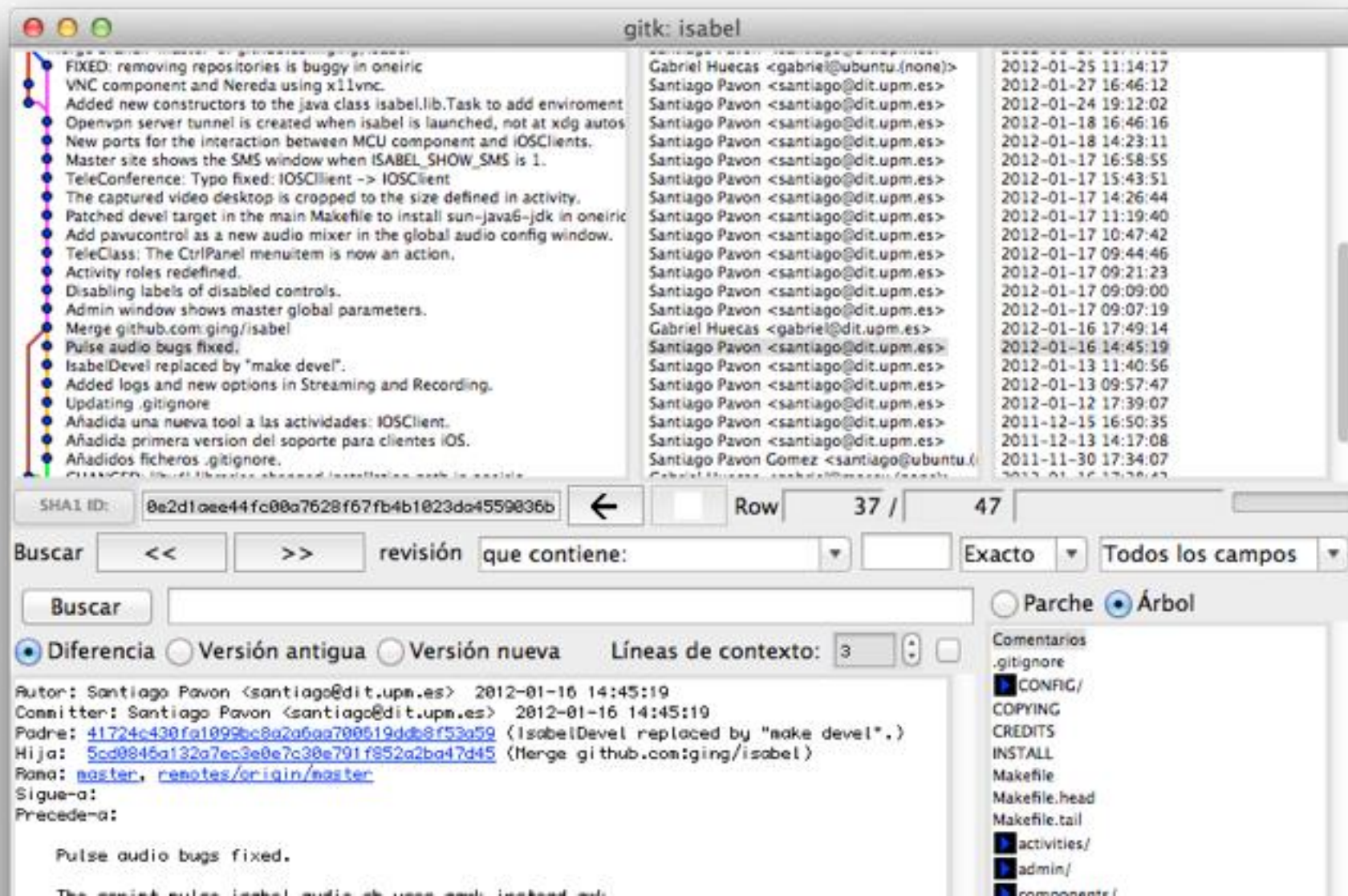
-ActiveRecord::Schema.define(:version => 0) do
+ActiveRecord::Schema.define(:version => 20111121171513) do
+
+  create_table "types", :force => true do |t|
+    t.string "name"
+    t.text "description"
+    t.datetime "created_at"
+    t.datetime "updated_at"
+  end
```

end

...

# gitk

## Ver la historia de versiones gráficamente



# 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

# 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
```



## 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).

## 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.

## 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:

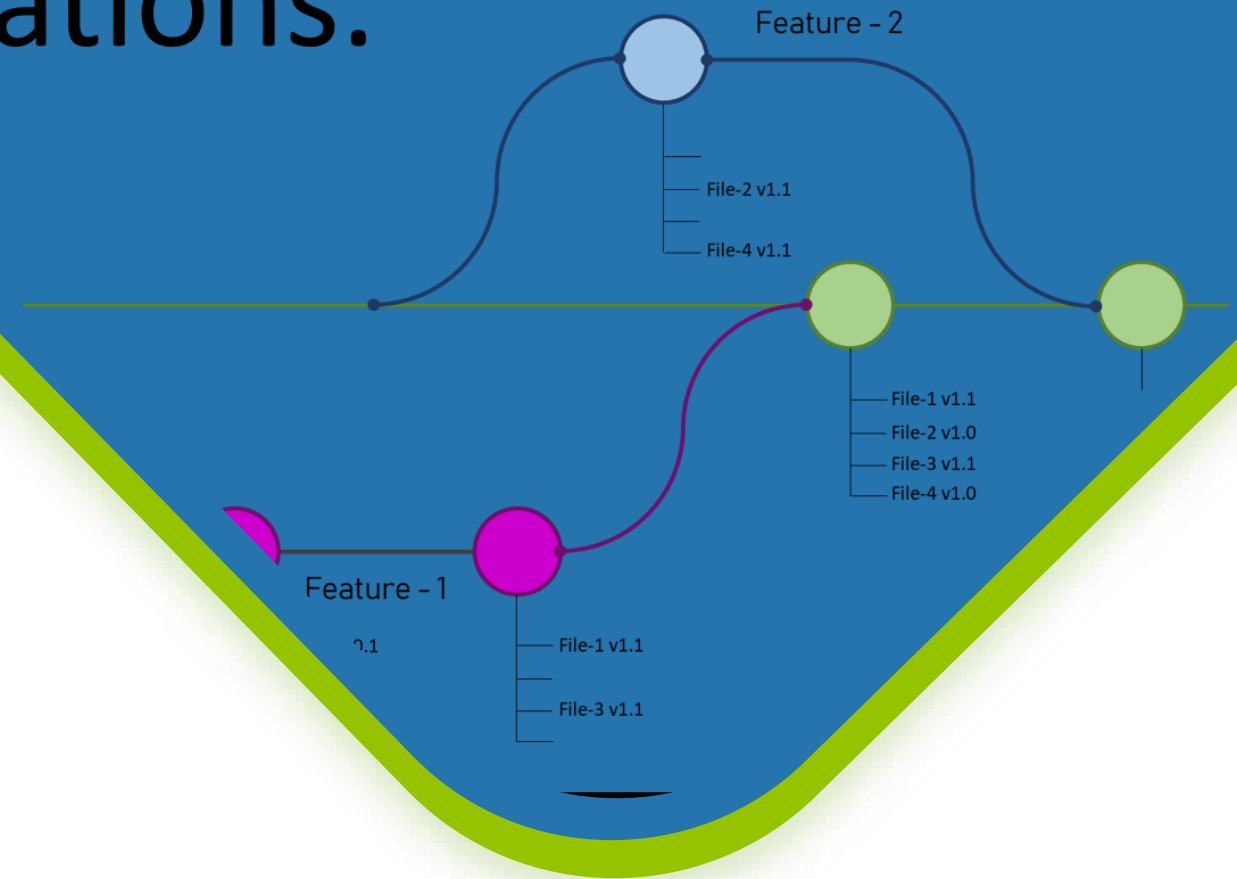
\$ **git checkout -- readme.txt**

# IT Branch and its Operations.



## PARTE 03

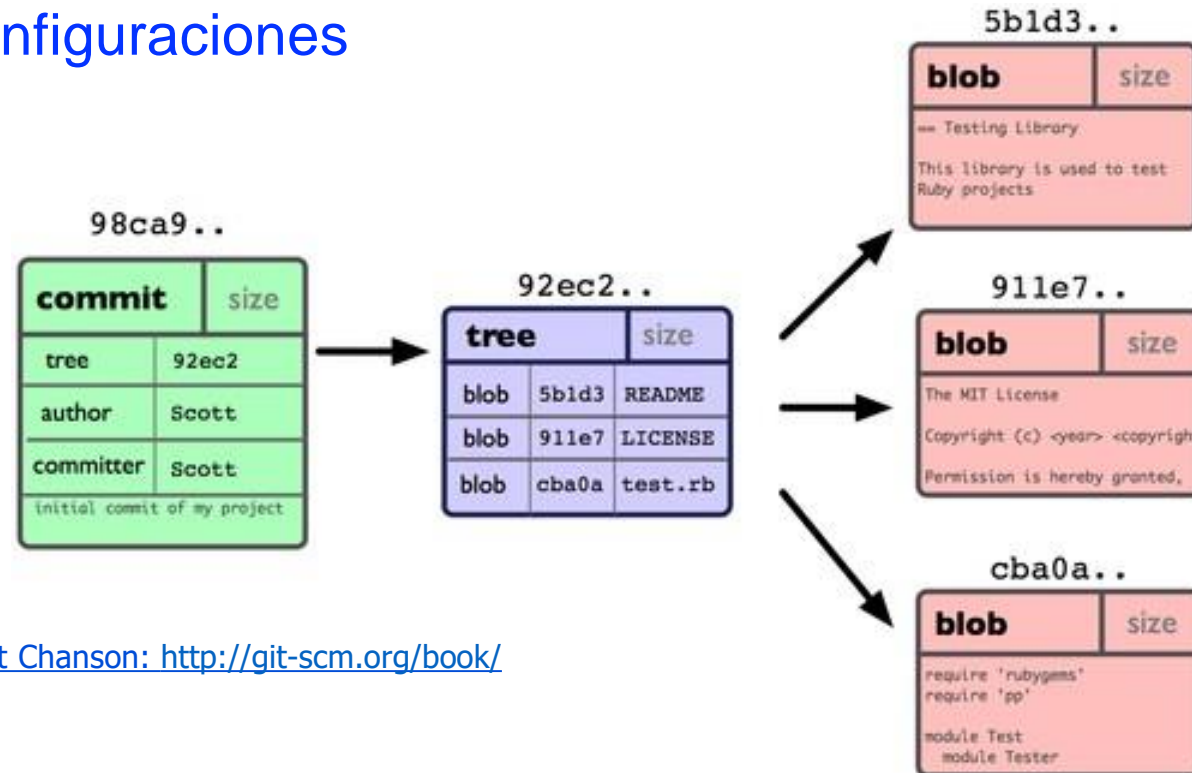
Usando Ramas



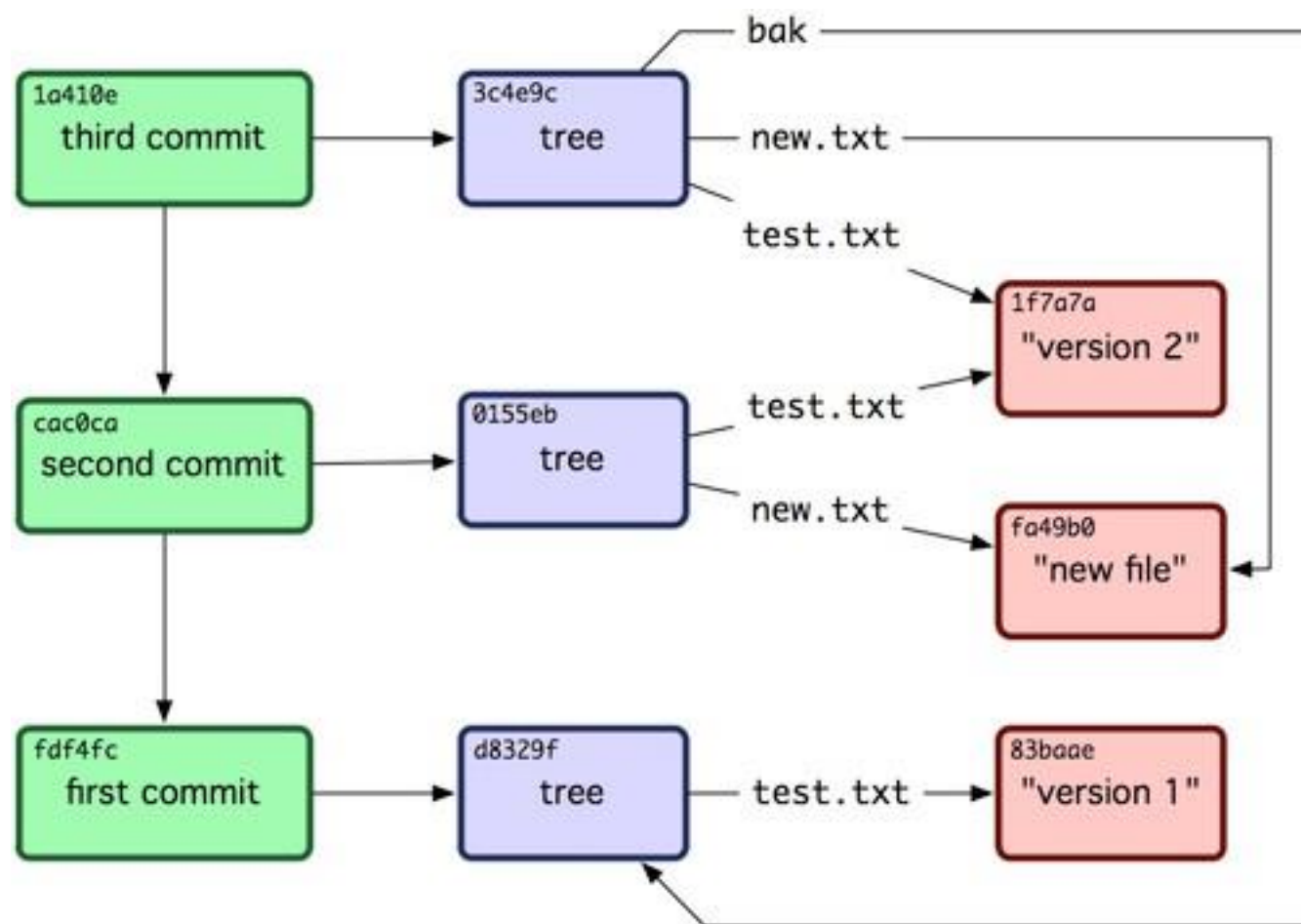
# Información almacenada en .git

## ✦ En .git se almacena:

- ★ objetos que representan los ficheros, los directorios, los commit,...
- ★ Referencias a repositorios remotos, ramas, ...
- ★ Configuraciones
- ★ ...

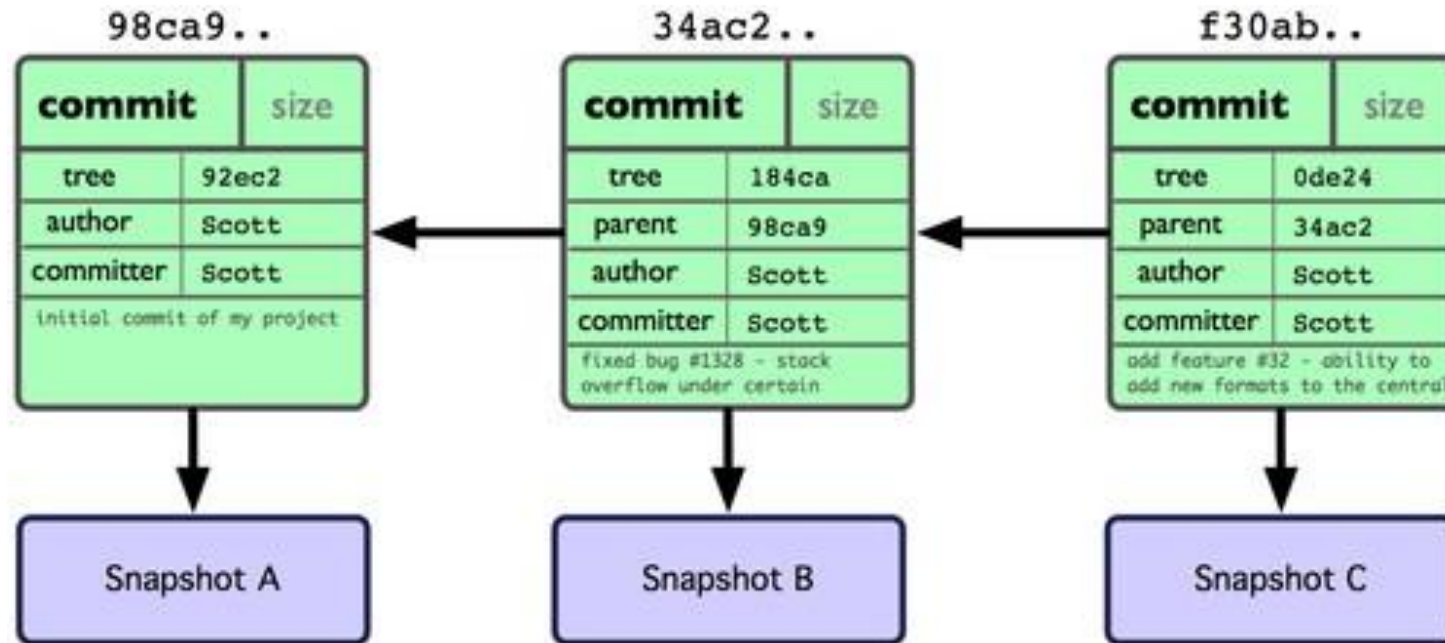


\*de Scott Chanson: <http://git-scm.org/book/>



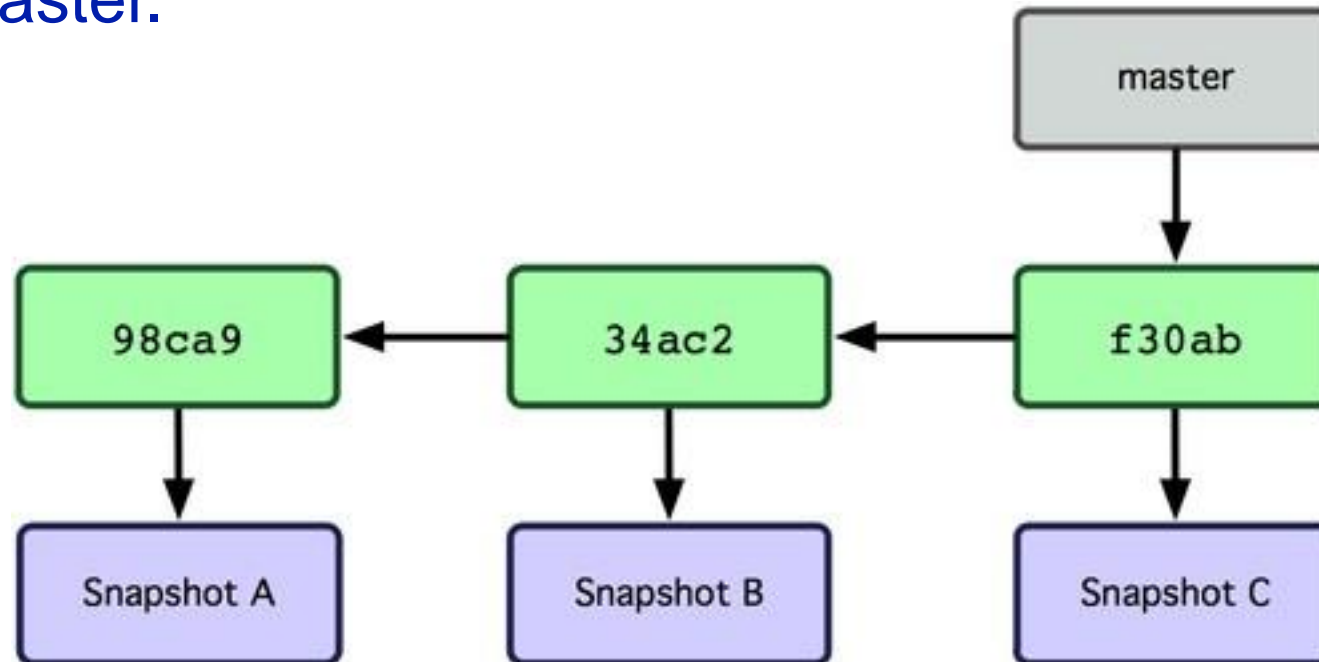
# Árbol de commits

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



## ¿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.

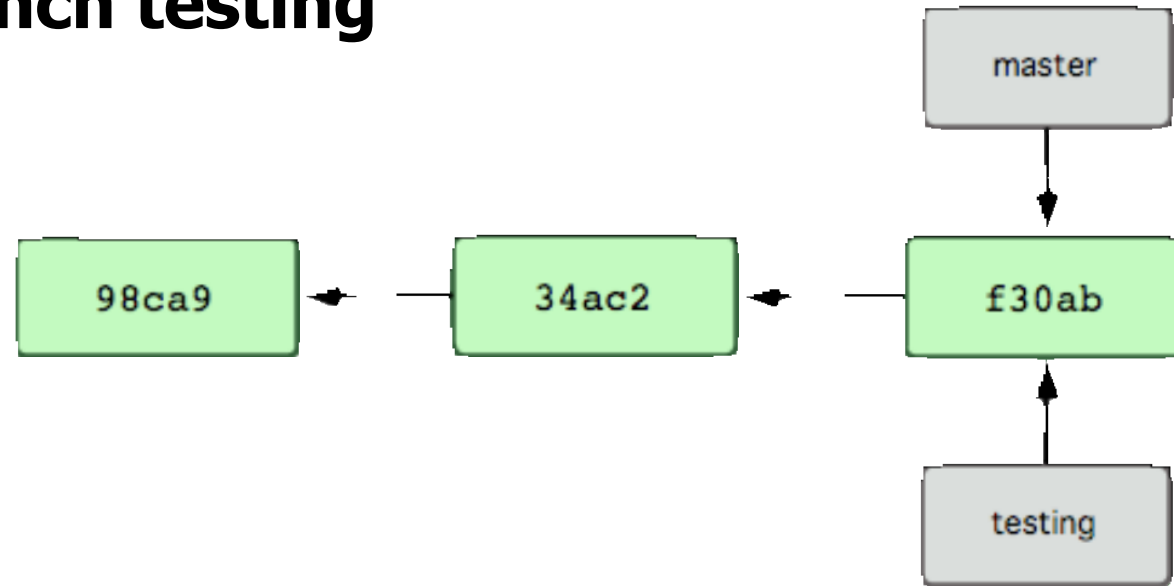




✦ Para crear nuevas ramas: **git branch <nombre>**

★ Se crea un nuevo puntero.

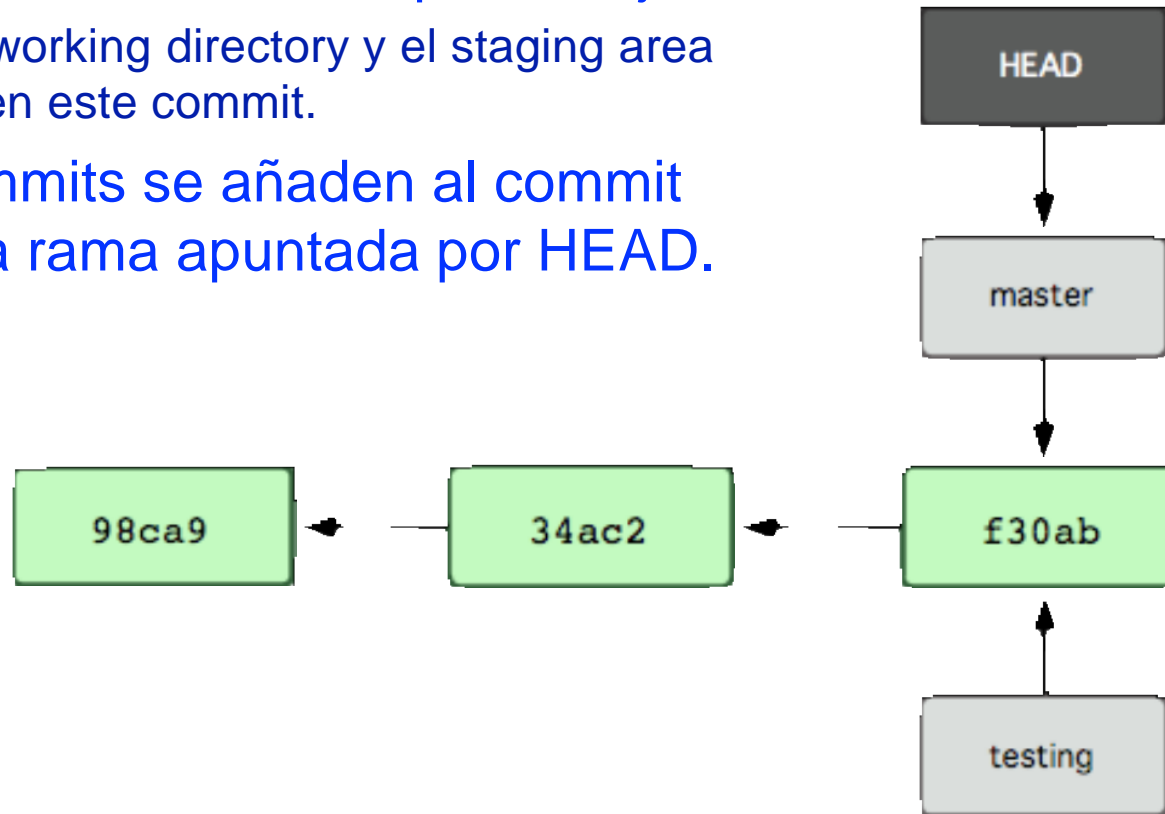
**\$ git branch testing**



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

## ✦ 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.

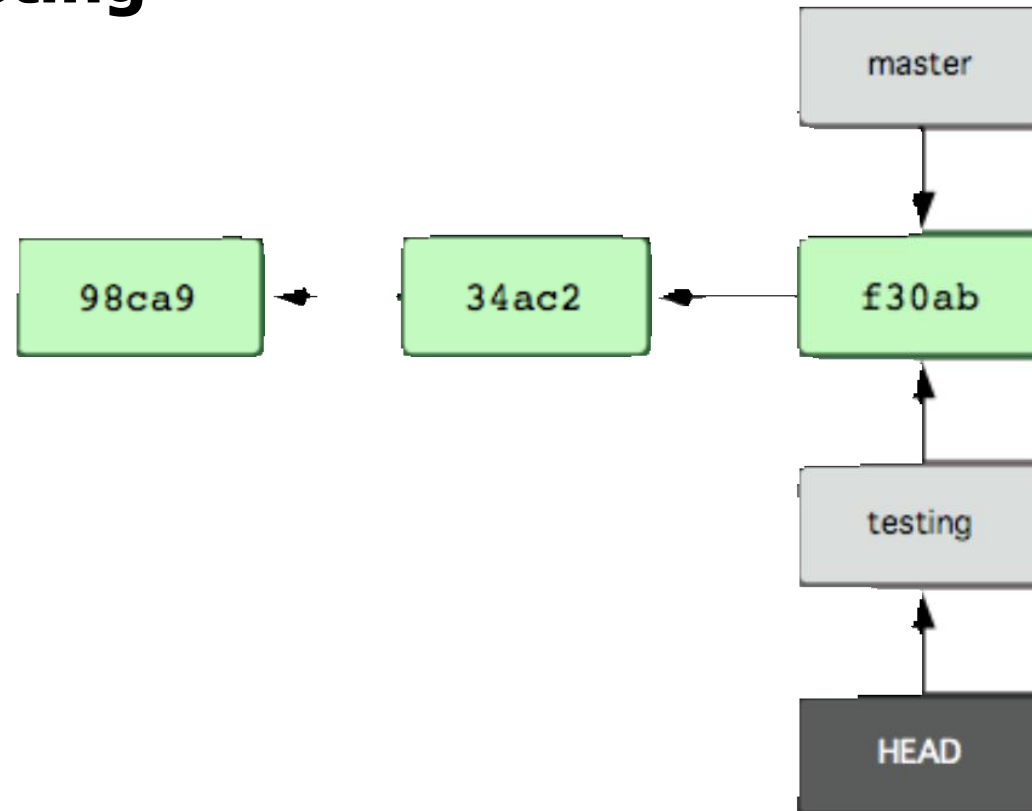


✦ Para cambiar de rama: **git checkout** <nombre\_rama>

★ **HEAD** apuntará a la nueva rama.

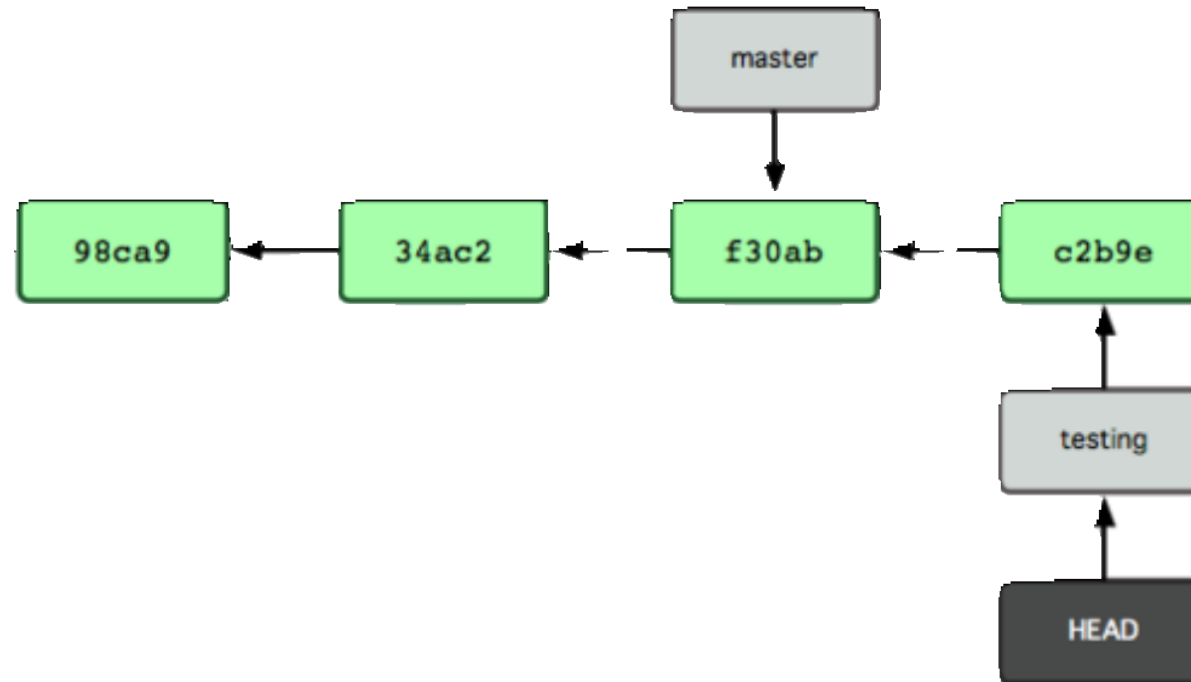
★ Se actualizan el working directory y el staging area.

\$ **git checkout testing**

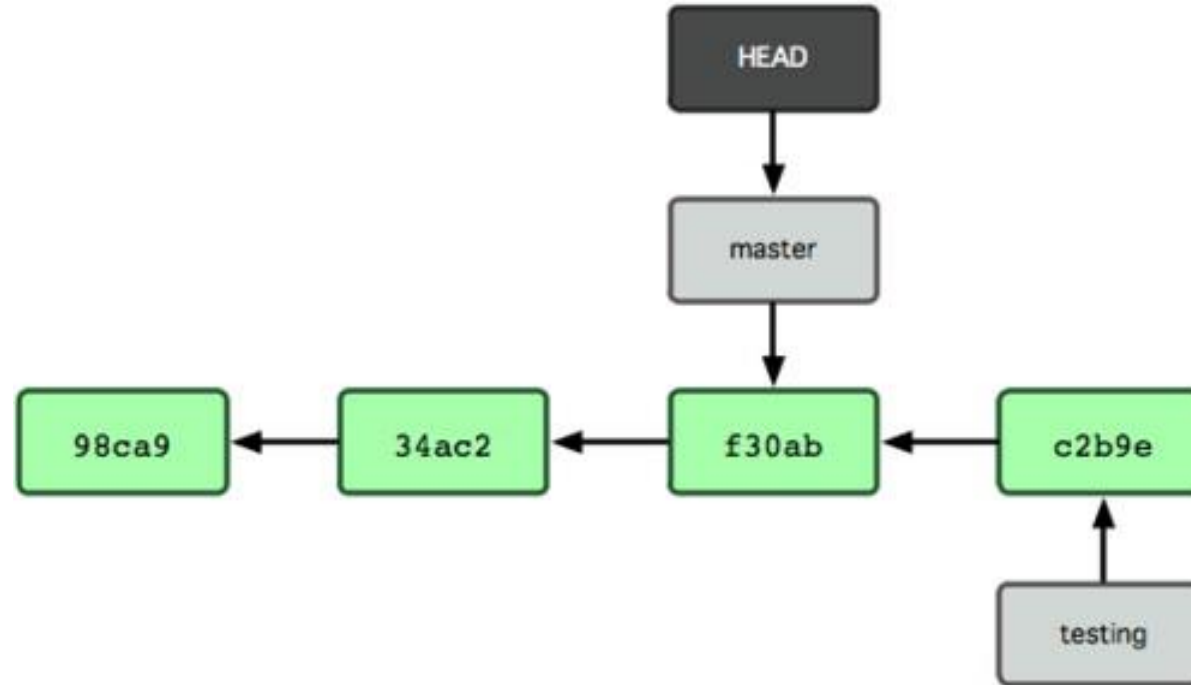


✦ Al hacer commit avanza la rama apuntada por HEAD.

**\$ git checkout testing**



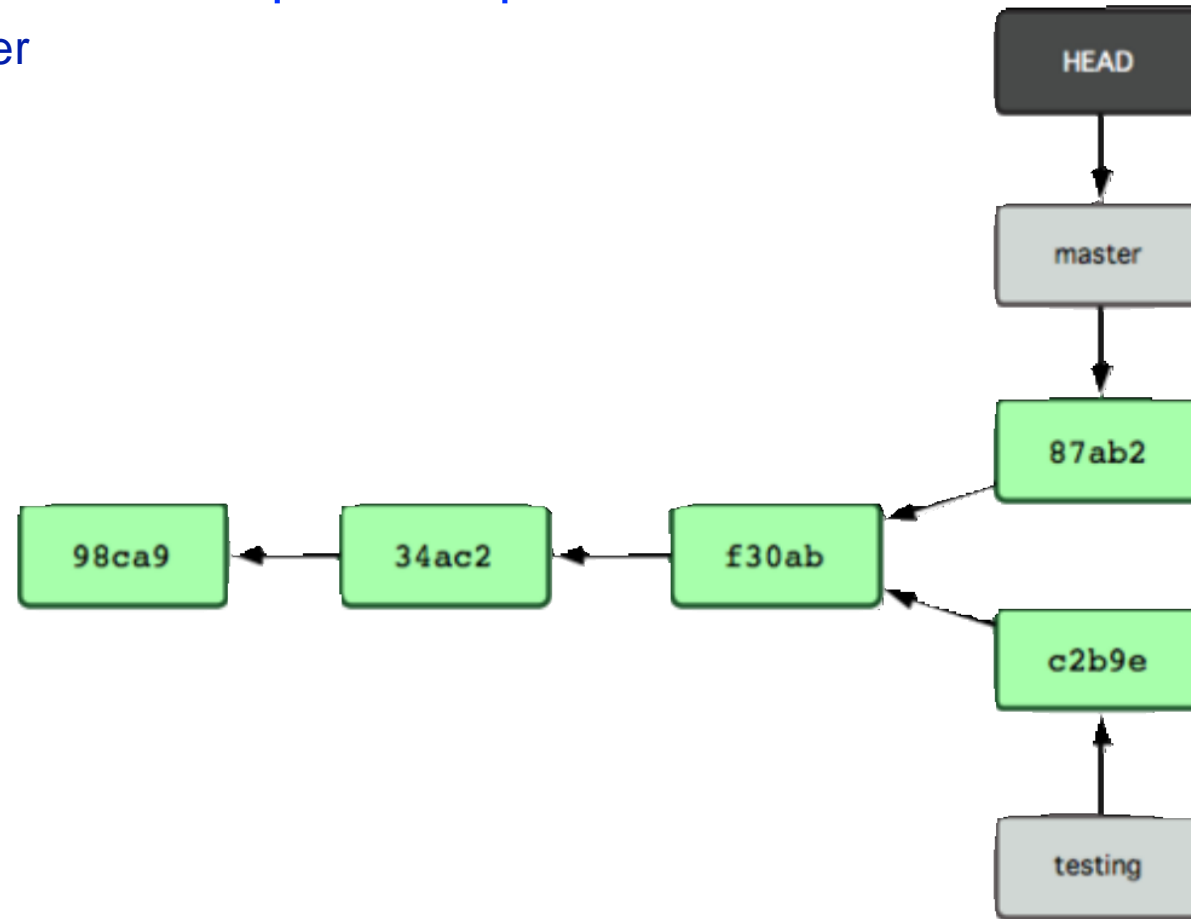
✦ Para volver a la rama master:  
\$ **git checkout master**



## ✦ 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>

# 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.



## Más usos de “git checkout”

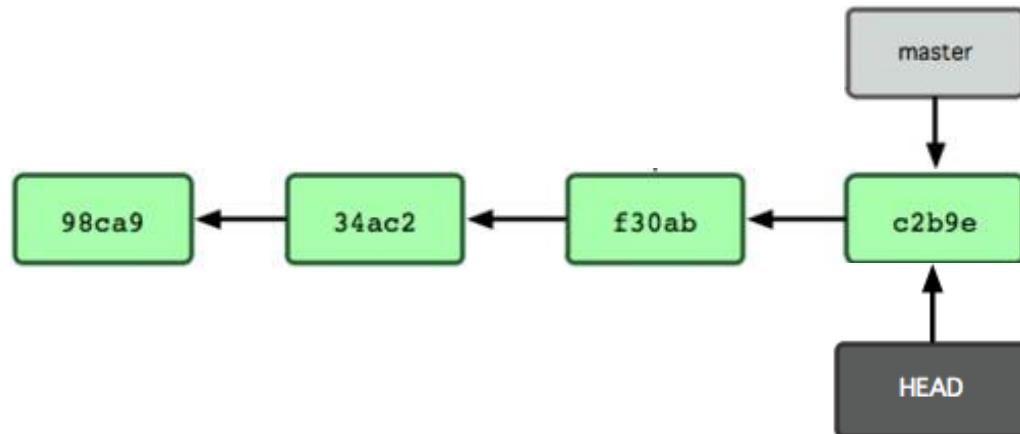
# “git checkout .” deshace cambios staged de area de trabajo

planet\$> git checkout .

# “git checkout -- <fichero>” deshace cambios staged de <fichero>  
# en area de trabajo

planet\$> git checkout -- <fichero>

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



## 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>**

## Unir ramas

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

**git merge <rama>**

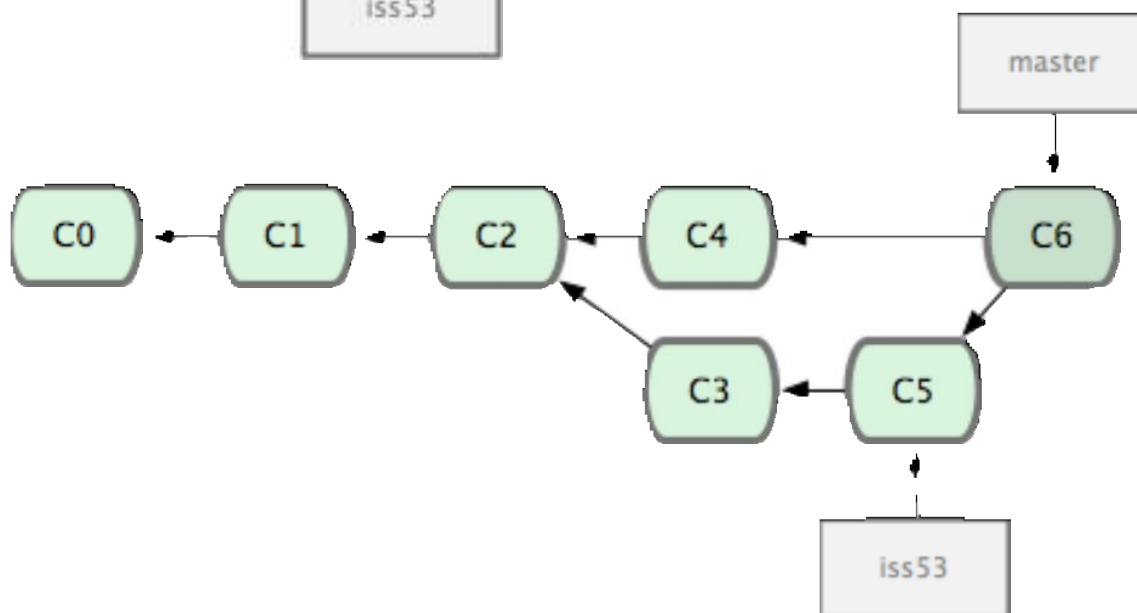
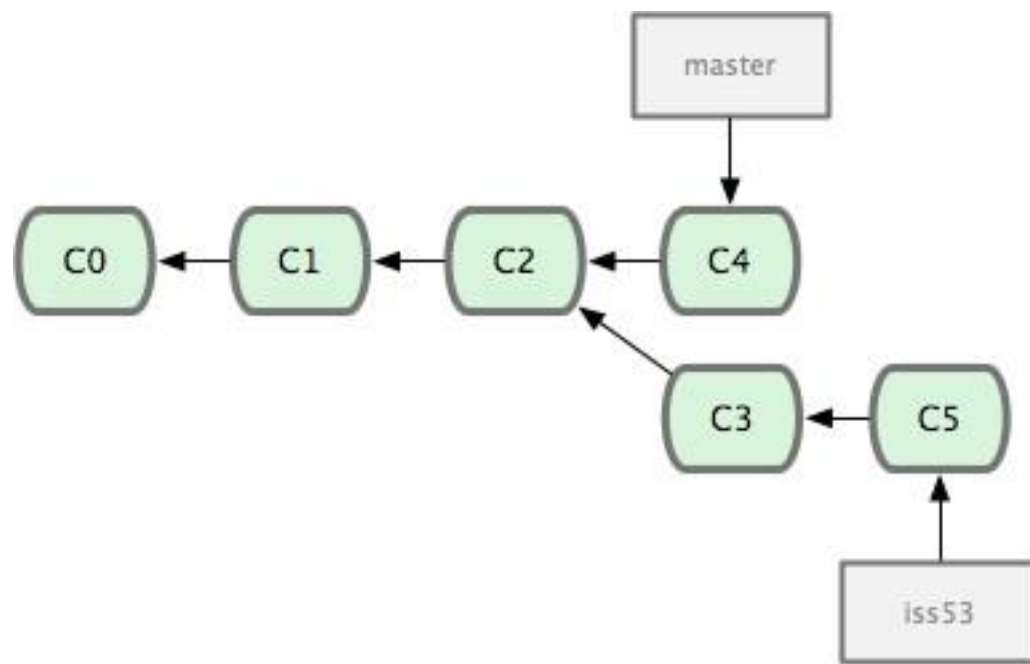
- ✦ Internamente GIT analiza la historia de commits para calcular como hacer la mezcla.

★ 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**



# 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**.

# Borrar ramas

- ✦ Una vez 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>**

# 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

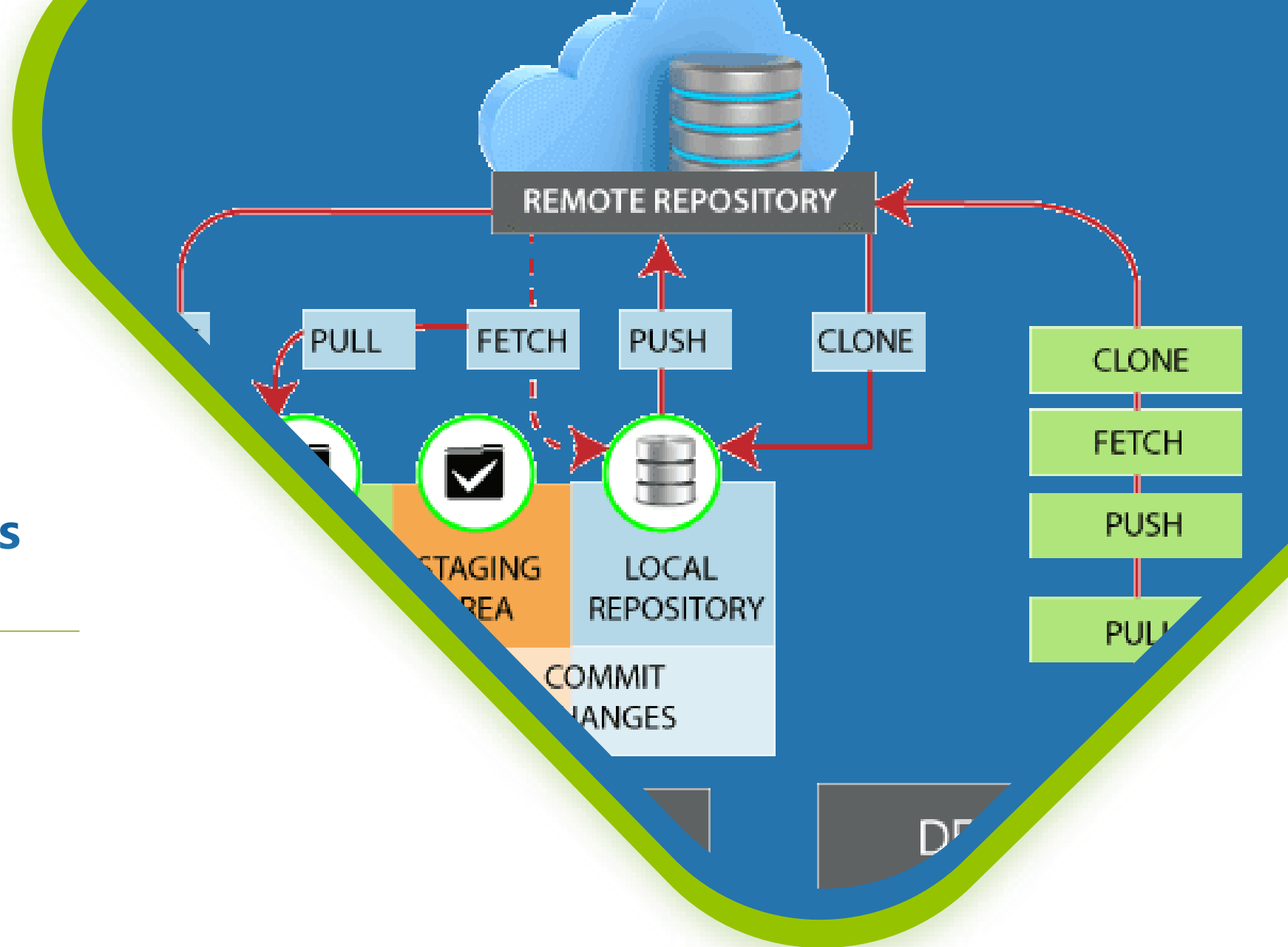
★ **-a** muestra todas las ramas (locales y remotas)

★ **-v** muestra el último commit de la rama.

★ **--merged** muestra las ramas que ya se han mezclado con la rama actual.

★ **--no-merged** muestra las ramas que aun no se han

## PARTE 04





## ¿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.

- ✦ 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
```

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

# 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
```

## Más comandos sobre remotes

# Inspeccionar detalles de un remote: **git remote show** [nombre\_del\_remote]

# Muestra el URL del remote, información sobre las ramas remotas,  
# las ramas tracking, etc.

# Renombrar un remote: **git remote rename** nombre\_viejo nombre\_nuevo

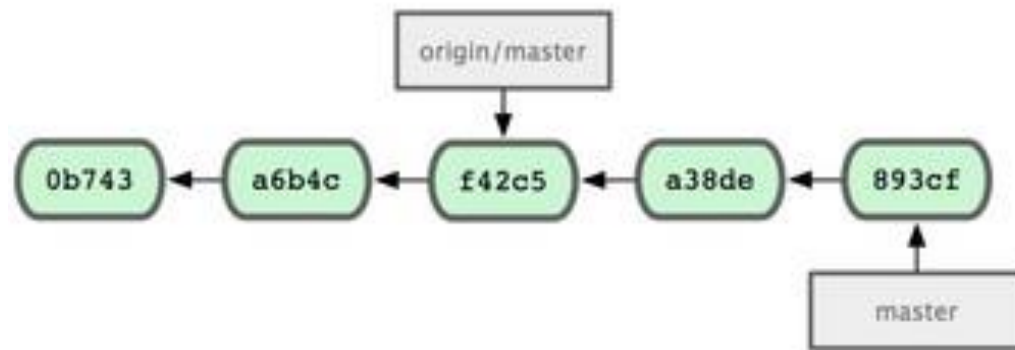
# Borrar un remote: **git remote rm** nombre\_del\_remote

# 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 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.

# 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.

- 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.

✦ 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 un nombre de una rama (tanto local como remota)

# - Si no es específica 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 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.

# 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.

# 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.

## PARTE 05

### GITHUB

---



# GitHub



# 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**

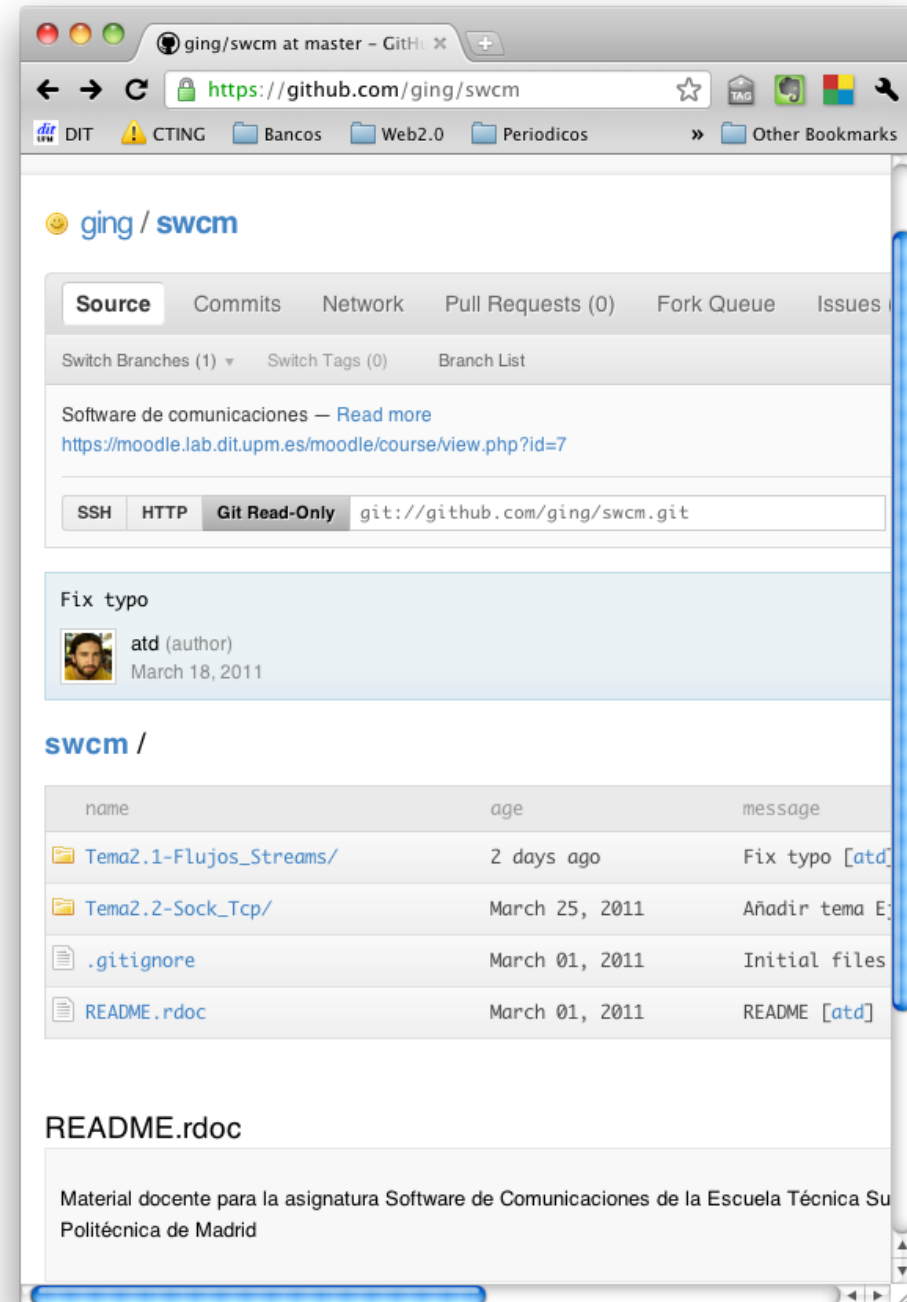
#  
# **En SWCM distribuimos con GITHUB:**  
# **- Ejemplos de transparencias**

#  
# **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
<a href="#">Tema2.1-Flujos_Streams/</a>
<a href="#">Tema2.2-Sock_Tcp/</a>
<a href="#">.gitignore</a>
<a href="#">README.rdoc</a>

swcm / Tema2.1-Flujos\_Streams

name
..
<a href="#">Acceso_a_Dispositivos/</a>
<a href="#">Expresiones_Regulares/</a>
<a href="#">Ficheros/</a>
<a href="#">Internacionalizacion/</a>

swcm / Tema2.1

name
..
<a href="#">CharIO.java</a>
<a href="#">char_io1.rb</a>
<a href="#">char_io2.rb</a>
<a href="#">lineIO.java</a>

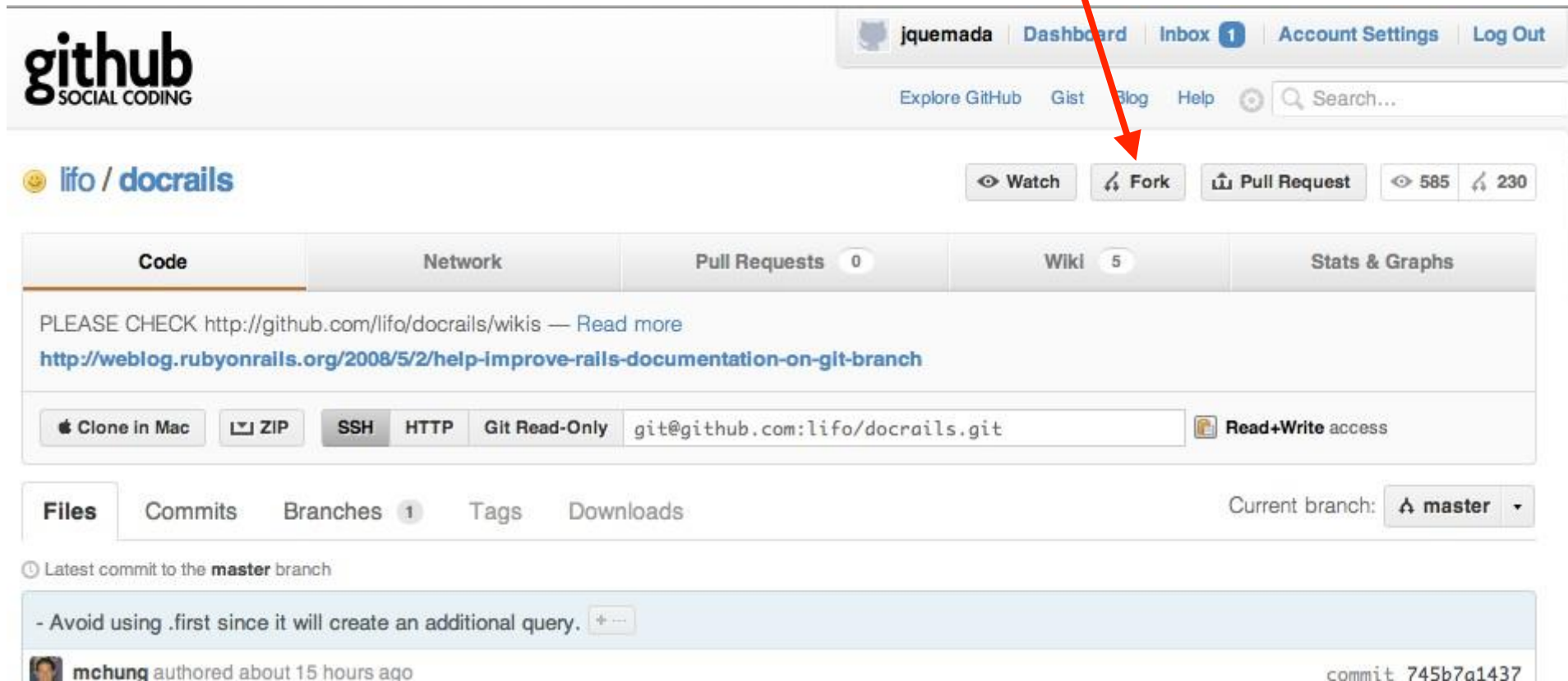
swcm / Tema2.1-Flujos\_Streams / Acceso\_a\_Dispositivos / char\_io1.rb

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
```

# 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"

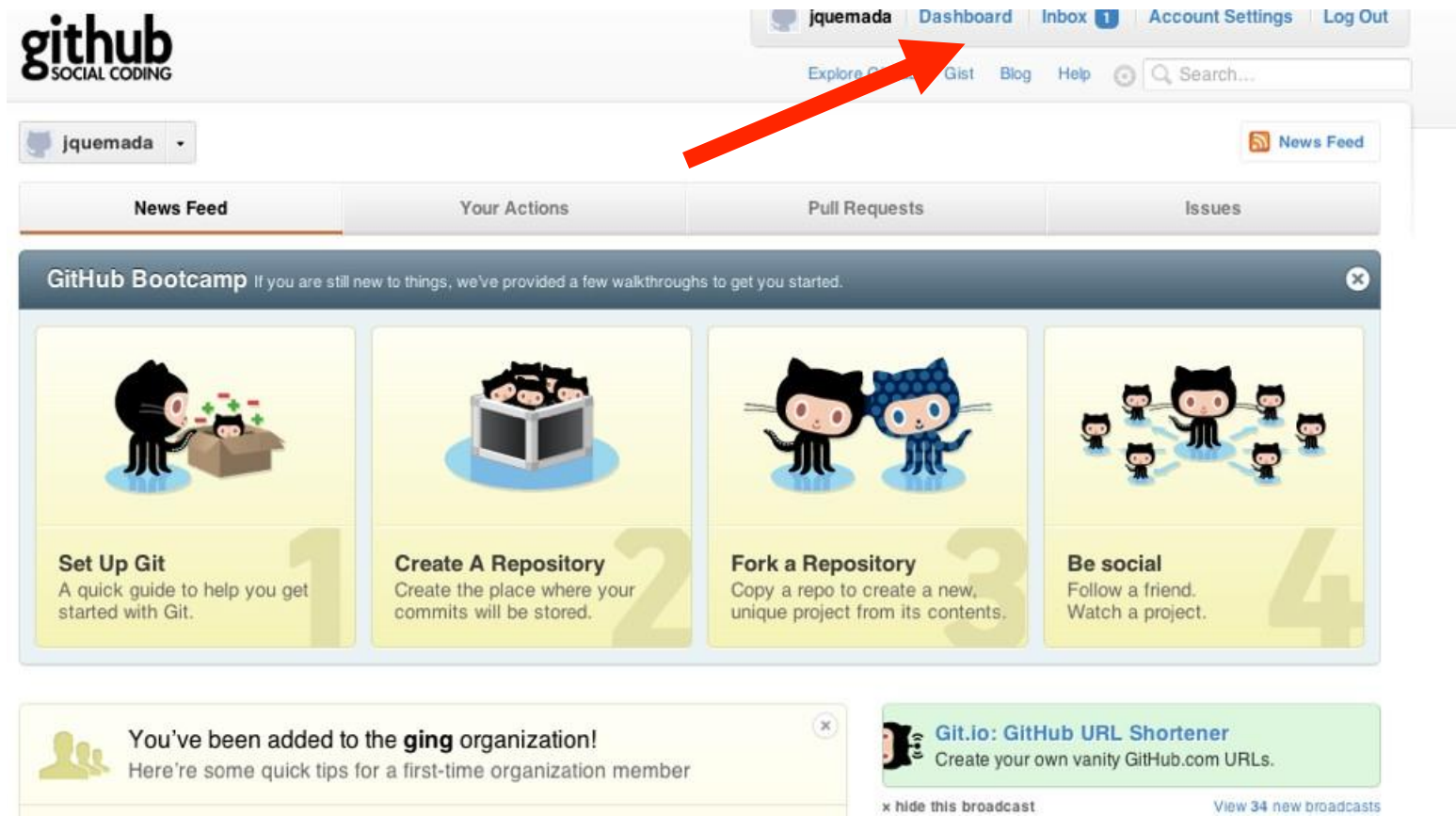


The screenshot shows the GitHub interface for the repository `lifo/docrails`. The user `jquemada` is logged in, with navigation links for `Dashboard`, `Inbox 1`, `Account Settings`, and `Log Out`. Below the repository name, there are buttons for `Watch`, `Fork` (highlighted with a red arrow), and `Pull Request`, along with statistics `585` and `230`. The repository has `0` Pull Requests and `5` Wikis. The `Code` tab is selected, showing a message to check the wiki and a link to a blog post. Below this, there are buttons for `Clone in Mac`, `ZIP`, `SSH`, `HTTP`, and `Git Read-Only`, with the SSH URL `git@github.com:lifo/docrails.git` and `Read+Write` access. The `Files` tab is selected, showing the `master` branch. The latest commit is by `mchung` about 15 hours ago, with the commit hash `745b7a1437`.



# 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”**



# 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!!**





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

52 week participation

rails Ruby 2 2,409

Forked from rails/rails

Ruby on Rails

Last updated September 02, 2011

52 week participation

planet2010 Ruby 3 2

Last updated Aug 15, 2011

## Public Activity

jquemada created branch master at [jqemada/planet](#) 1 day ago

New branch is at [jqemada/planet/tree/master](#)  
[Compare master branch with master](#)

jquemada created repository [planet](#) 1 day ago

New repository is at [jqemada/planet](#)

jquemada pushed to master at [jqemada/rails](#) September 02, 2011

4b7056 Merge pull request #1 from walle/master  
d2320a4 Make the generation textile play a little nicer with  
gmi. Cleanup n...  
5981704 Make some default print rules and add some content from  
main css to 1...  
[1 more commits](#)

jquemada merged pull request 1 on [jqemada/rails](#) September 02, 2011

Add more print friendly css  
3 commits with 213 additions and 94 deletions

# 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

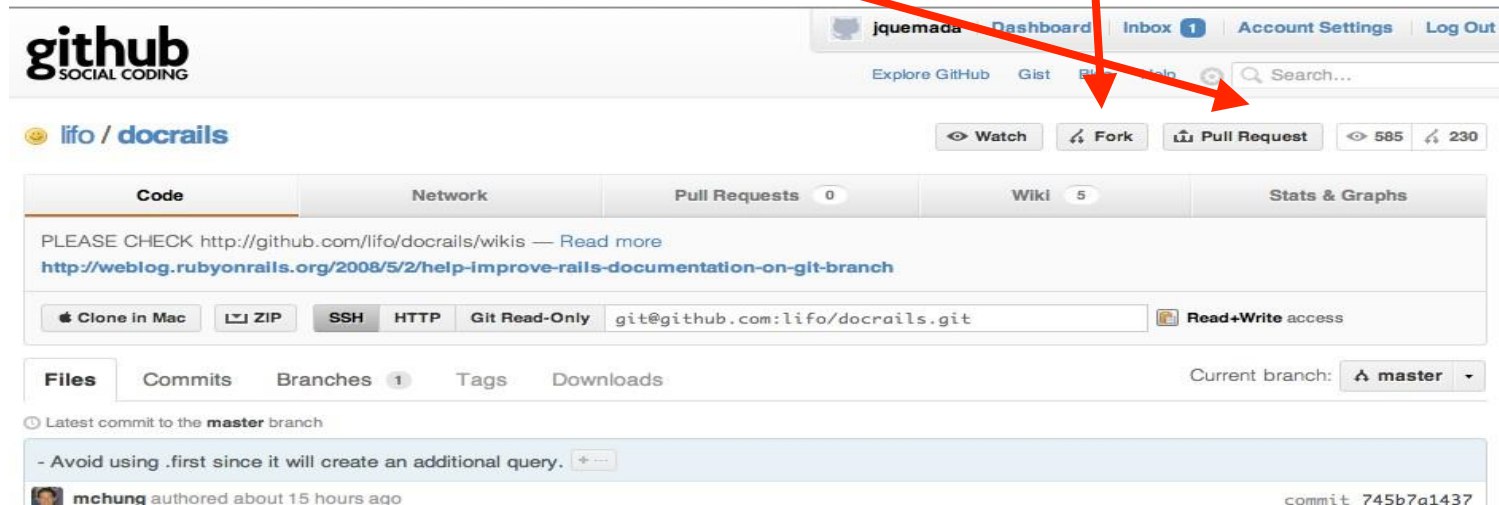
```
p1> .....
```

```
p1> git add ...
```

```
p1> git commit -m '.....'
```

```
p1> git push origin master
```

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



## PARTE 06

### 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 commit C:
  - ★  $C^0$  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:
  - ★  $C_{-1}$  commit padre
  - ★  $C_{-2}$  commit abuelo
  - ★  $C_{-3}$  commit bisabuelo
- ♦ Así,  $C_{-3}^2$  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 commits añadiendo **^C** o **--not C** pr excluir todos los commits accesibles desde 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 sincronice con origin. Son los cambios que se subirán al ejecutar push.

Releases

Tags

Tags

## Tags

6b405e5 zip tar.gz

zip tar.gz

# 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
```

## 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.

## 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
```

# Modificación de los Commits

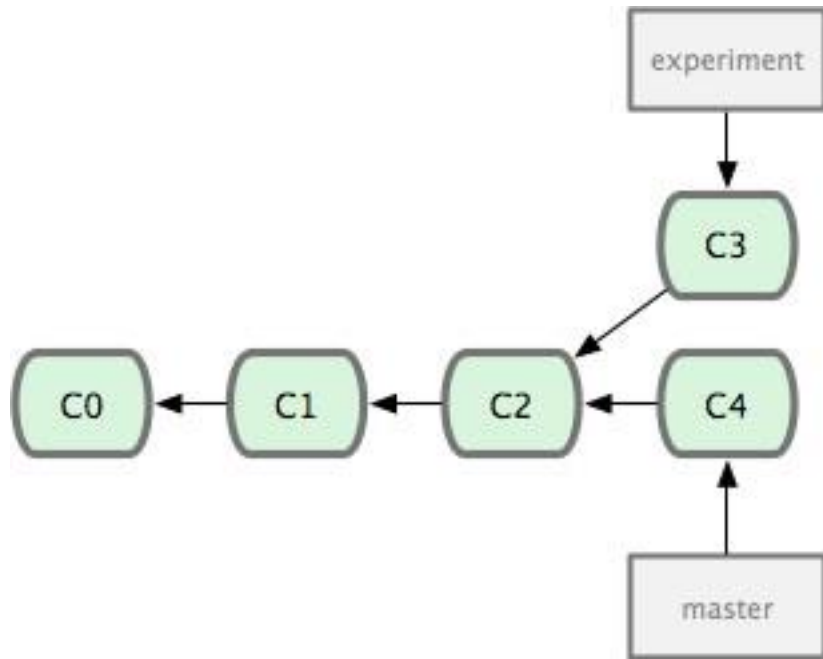
- ✦ Existen varios comandos de GIT que permiten cambiar los commits existentes, aplicar los cambios de un commit en otro punto, eliminar los cambios introducidos en un commit, etc.
  - ★ git rebase
  - ★ git reset
  - ★ git cherry-pick
  - ★ git revert
  - ★ amend

# Rebasing

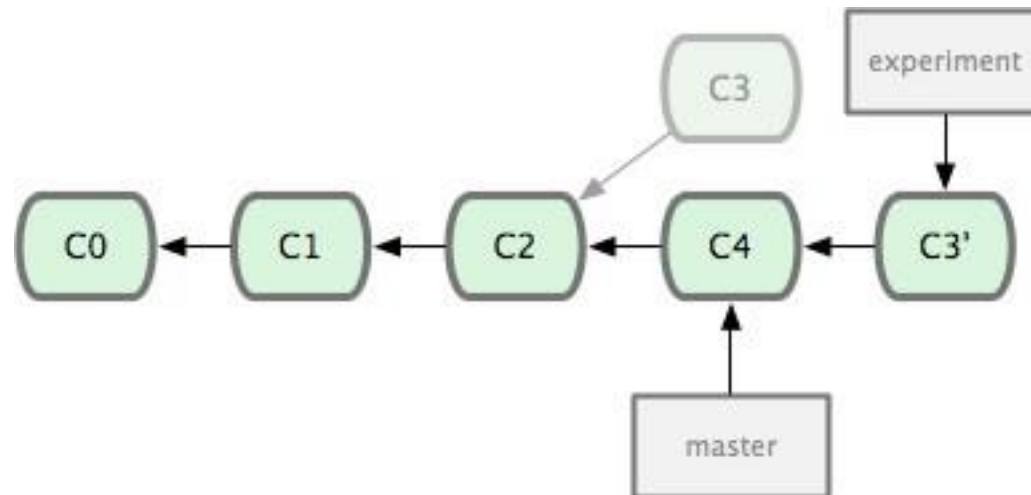
- ✦ Antes de nada: **No hacer un rebase de un commit que ya se haya subido a un repositorio público.**
  - Cambiar commits que ya pueden tener otros desarrolladores causará problemas
- ✦ Rebase permite mover una rama de commits a otro sitio.

## **git rebase [basebranch] [topicbranch]**

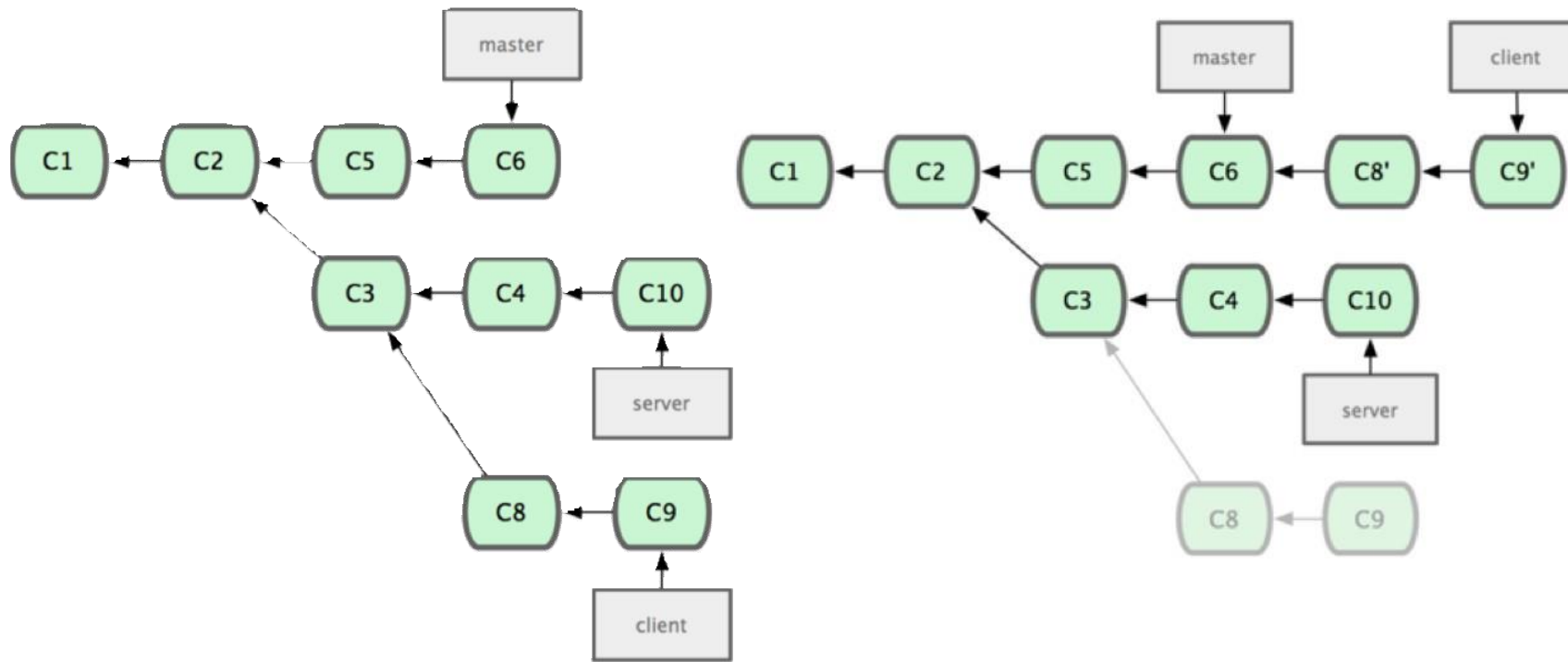
- ★ Mueve los commits de la rama *topicbranch* a *basebranch*.
  - Busca un commit común entre *basebranch* y *topicbranch*, y partiendo de él, aplica los mismos cambios realizados en los commits *topicbranch* a *basebranch*.
- ★ Si se omite *topicbranch*, se usa la rama actual.
- ✦ Rebase también se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



**\$ 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

- ★ Si al hacer commit los comandos no funcionan en una 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, 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

✦ 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

## 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 también 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.

## ♦ 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.

✦ 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**

## 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 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>**

- ★ 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.



## 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 sus 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.

✦ **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.

✦ **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 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

- ✦ 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**

## 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**

## 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>
```

# 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 (checkout) 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.



✦ 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@{#}**.

✦ 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}**

# 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>`

# 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 blob) de un fichero.
- ★ git merge-base : se usa para buscar la base para hacer un merge
- ★ ...

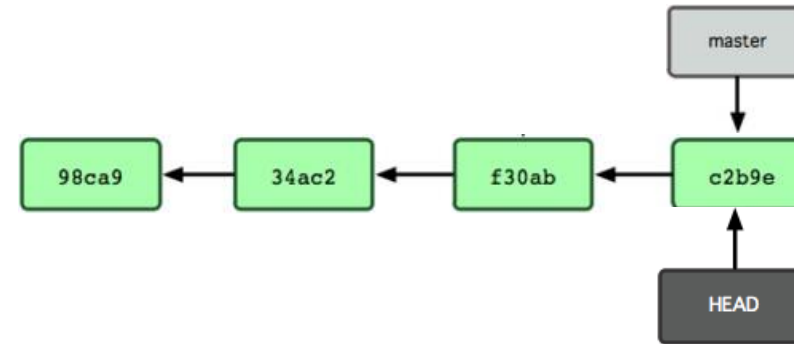
# Temas pendientes

- ◆ Reflog
- ◆ Atributos
- ◆ Hooks
- ◆ ...

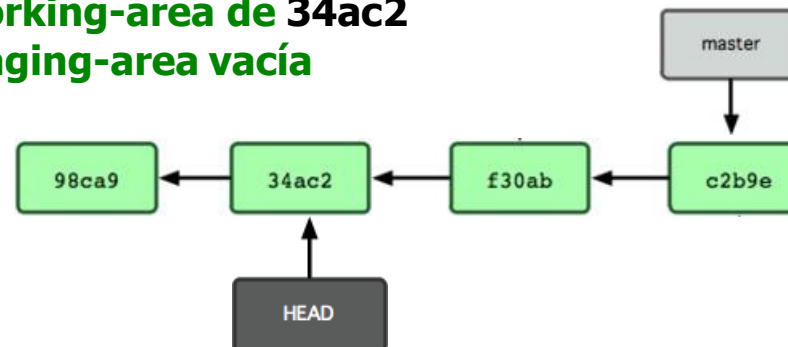
# Checkout: volver a commit anterior

# "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  
# 3) Staging-area vacía



DETACHED HEAD - NO CONTAR ESTO



# Reset: Eliminar commits

# "git reset <commit\_id>"

# -> elimina rama del árbol y deja cambios en staging-area

# -> opción --hard elimina cambios completamente

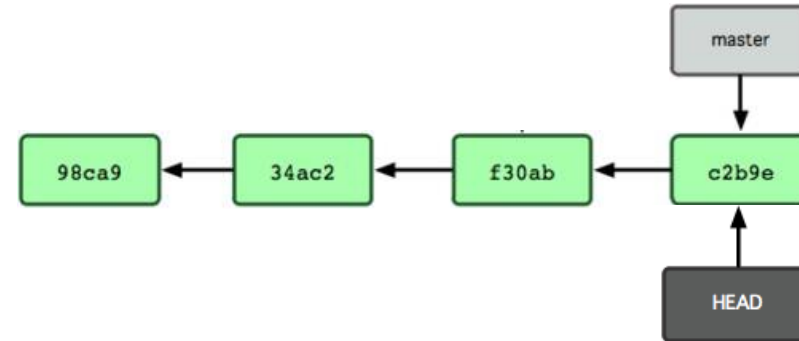
planet\$> git log --oneline # lista commits

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

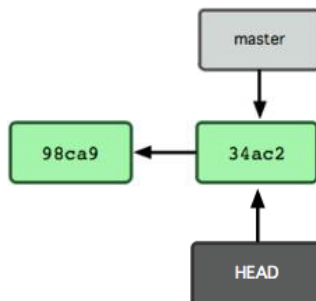
# 2) Working-area de 34ac2

# 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\$>

# 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\$>

# 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`

# 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 también 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

# 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**

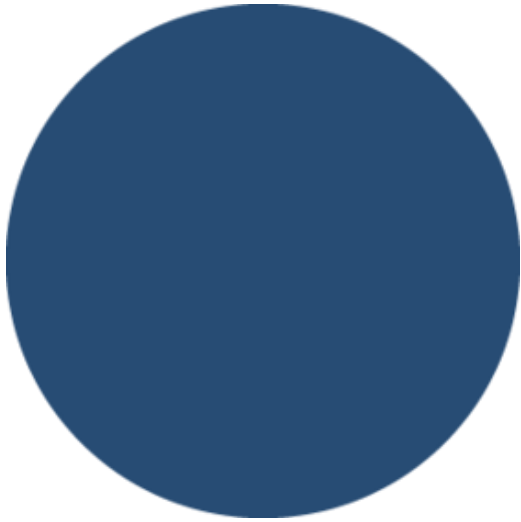
**swcm> git fetch origin # fetch: traer todos los commit de un repositorio  
# remoto a nuestro repositorio, los integra como ramas**

**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**

# MW Blues



## MW Blue

RGB **40-75-115**

# **284b73**

CMYK **90-65-30-20**

Pantone **647**

Sikkens **U0.30.30**

## Usage:

- MW Blue - Used for the Open-Xchange logo when on a white background
- MW Dark Blue - Used for general backgrounds
- MW Medium Blue - Used for generating the spotlight on "MW Dark Blue"
- MW Light Blue - Used for product names



## MW Dark Blue

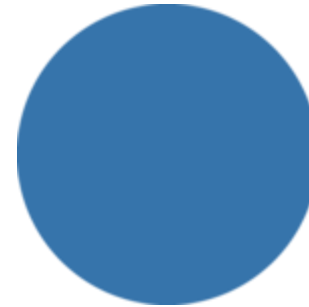
RGB **5-45-75**

# **052d4b**

CMYK **100-80-40-40**

Pantone **533**

Sikkens **U2.21.09**



## MW Medium Blue

RGB **60-115-170**

# **3c73aa**

CMYK **80-50-15-0**

Pantone **653**

Sikkens **U0.30.40**



## MW Light Blue

RGB **120-170-210**

# **78aad2**

CMYK **55-20-0-5**

Pantone **652**

Sikkens **T4.18.59**

# MW GREY

## Usage:

All Manexware greys can be used for backgrounds, gradients, lines and for form fields.

MW Grey and MW Silver can also be used as text colors in specific cases.



**MW Graphite Black**  
RGB 45-45-45  
# 2d2d2d  
CMYK 0-0-0-90  
Pantone **black 7**  
Sikkens **ON.00.15**



**MW Grey**  
RGB 135-135-135  
# 878787  
CMYK 0-0-0-60  
Pantone **Cool grey 10**  
Sikkens **ON.00.40**



**MW Silver Grey**  
RGB 220-220-220  
# dcdcdc  
CMYK 0-0-0-15  
Pantone **Cool grey 2**  
Sikkens **ON.00.76**

# MW HIGHLIGHT COLORS

## Usage:

As the name suggests, these colors are reserved for highlighting elements. This important job can only be achieved successfully if they are used as sparingly as possible. For example: using them to highlight starbursts, buttons and/or similar elements that warrant a color highlight.

Please note that the use of the highlight colors mandarin and mango should be restricted and only be used when all the other highlight colors (ice blue, lime green and raspberry) have already been used.



**MW Raspberry Red**  
RGB **180-35-105**  
# **b42369**  
CMYK **25-95-25-10**  
Pantone **227**  
Sikkens **A0.41.28**



**MW Lime Green**  
RGB **150-195-0**  
# **96c300**  
CMYK **50-0-100-0**  
Pantone **376**  
Sikkens **H2.50.60**



**MW Ice Blue**  
RGB **0-160-225**  
# **00a0e1**  
CMYK **90-20-0-0**  
Pantone **7461**  
Sikkens **R5.52.49**



**MW Graphite Black**  
RGB **45-45-45**  
# **2d2d2d**  
CMYK **0-0-0-90**  
Pantone **black 7**  
Sikkens **ON.00.15**



**MW Grey**  
RGB **135-135-135**  
# **878787**  
CMYK **0-0-0-60**  
Pantone **Cool grey 10**  
Sikkens **ON.00.40**



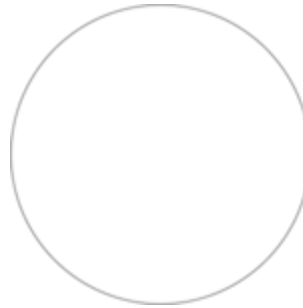
# MW TEXT COLORS

## Usage:

The use of black on light backgrounds or the use of white on dark backgrounds is equally effective, but remember that white can also be used as a spotlight color in the silver gradient or as a form field on a light background.



**Black**  
RGB 0-0-0-0  
# 000000  
CMYK 0-0-0-100  
Pantone **Process black**  
Sikkens **ON.00.10**



**White**  
RGB 255-255-255  
# ffffff  
CMYK 0-0-0-0



**Extra Text Colors**  
**MW Ice Blue, MW Grey and MW Silver Grey** can be used as extra text colors in specific cases

# MW GRADIENTS

## Usage:

Radial gradients can be used to focus attention on something. Like a spotlight behind an isolated visual a radial gradient attracts the viewer's attention. It can also be used to emphasize a piece of text or a headline. Finally it can be used to simply make a flat background more appealing.

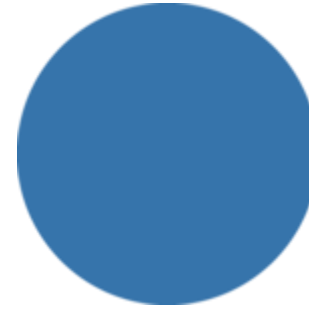
The creation of gradients, and their placement, is quite simple, once you know how. Simply use the pure Open-Xchange colors for each side in the gradient tool and you are done (e.g. MW Dark Blue and MW Medium Blue to create the MW Blue Gradient).



**MW Dark Blue**



**MW Blue Gradient**

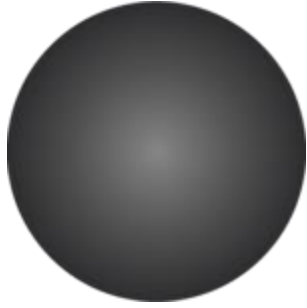


**MW Medium Blue**

# MW GRADIENTS



**MW Graphite Black**



**MW Grey Gradient**



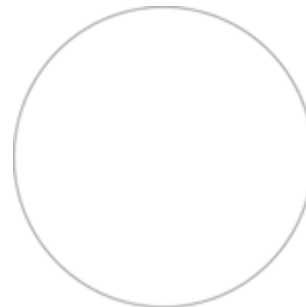
**MW Grey**



**MW Silver Grey**



**MW Silver Gradient**



**White**

# Manexware Corporate Font

Open Sans is a sans serif typeface designed by Steve Matteson. Open Sans has an upright stress and a neutral, yet friendly appearance. It was optimized for print, web, and mobile interfaces, and has excellent legibility characteristics.

We use four selected types of Open Sans and their Italic versions. This allows us to comfortably tackle anything using this font.

Open Sans Light, *Italic*  
Open Sans Regular, *Italic*  
**Open Sans Bold, *Italic***  
**Open Sans Extrabold, *Italic***

- ✓ Get your slice **of the App Pie**
- ✓ What **our Partners** say
- ✓ Take a **Test Drive**

## Font Usage

Using a narrow style in combination with bold is a very good way to make things stand out. Also by using Open Sans Extrabold within an important statement, or headline, that is in Open Sans Light really makes it 'pop'. See examples on the left.