

## Ampliación de Ingeniería del Software

### Tema 3 – Gestión de la configuración del Software

#### Tema 3.2 – Gestión del código fuente



Universidad  
Rey Juan Carlos

**Micael Gallego**

Correo: micael.gallego@urjc.es  
Twitter: @micael\_gallego

**Francisco Gortázar**

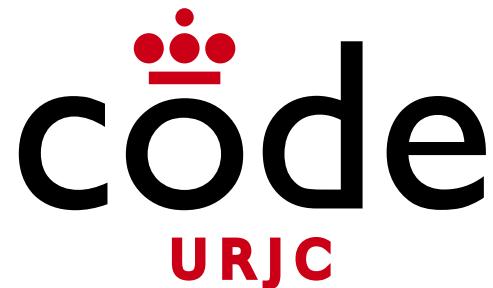
Correo: francisco.gortazar@urjc.es  
Twitter: @fgortazar

**Michel Maes**

michel.maes@urjc.es

**Óscar Soto**

oscar.soto@urjc.es



©2024

Micael Gallego, Francisco Gortázar, Michel Maes, Óscar Soto

Algunos derechos reservados

Este documento se distribuye bajo la licencia  
“Atribución-CompartirIgual 4.0 Internacional”  
de Creative Commons Disponible en

<https://creativecommons.org/licenses/by-sa/4.0/deed.es>

# Gestión del código fuente

- Introducción
- Git: trabajando en local
- Git y GitHub: trabajando en equipo
- Gestionando el repositorio
- Modelos de desarrollo

# Introducción

- Los programadores desarrollan código de forma **colaborativa**
- Las herramientas **populares** para compartir ficheros (email, dropbox, Gdrive, OneDrive) **no son adecuadas** para el código fuente
- Los **sistemas de control de versiones** (*Version control system, VCS*) son herramientas especialmente diseñadas para desarrollo colaborativo

# Introducción

- **Diferencias con Dropbox/GDrive/OneDrive**
  - En desarrollo software, el programador decide **cuándo quiere subir** al servidor su código
  - En desarrollo software, se pueden crear copias de un mismo fichero para que varios programadores trabajen en una copia y se puedan **fusionar**

# Introducción

- Tienen muchos nombres
  - Sistema de control de versiones (VCS)
    - *Version Control System*
  - Gestor de código fuente (SCM)
    - *Source Code Manager*
  - Repositorios de código
    - *Code repository*

# Introducción

- Hay muchos diferentes



# ¿Por qué Git?

- Software libre
- Muy popular
- Múltiples herramientas para trabajar con él
- **Muy rápido** en realizar las operaciones
- Posibilidad de trabajar offline y luego sincronizar (**distribuido**)
- Funcionalidades que facilitan el **trabajo colaborativo**



# Introducción

- **Git** es uno de los más utilizados en la actualidad
- Desarrollado por **Linus Torvalds** en 2005 para el desarrollo del **Kernel de Linux**

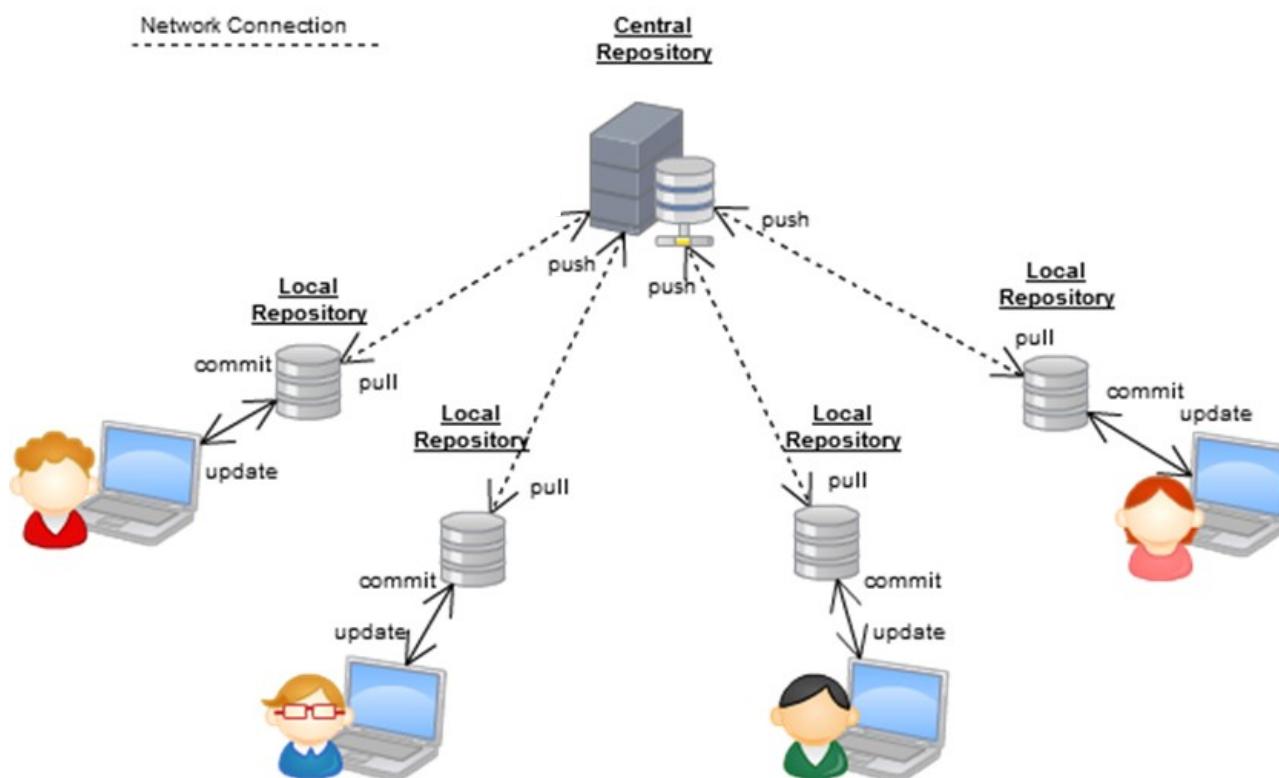


<https://git-scm.com/>

<https://en.wikipedia.org/wiki/Git>

# Introducción

- Git es un **SCM distribuido**
  - Cada desarrollador tiene un **repositorio en local** que se sincroniza con el **repositorio compartido**



# Introducción

## • Clientes

- Existen **múltiples aplicaciones** para trabajar con los repositorios locales (clientes)



Git en Linux / Mac



Git for Windows

Línea de comandos

# Introducción

## • Clientes

- Los **clientes gráficos** es posible que no ofrezcan todas las funcionalidades de Git
- Es buena idea conocer cómo realizar las operaciones por **línea de comandos**
- Se pueden **combinar** sin problemas

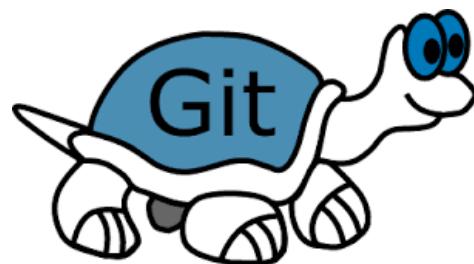


# Introducción

```
mac express: git log --oneline --graph
* 11a77a3 Fix inner numeric indices incorrectly altering parent req.params
* ee90042 Fix infinite loop condition using mergeParams: true
* 97b2d70 4.13.2
* a4fcd91 deps: update example dependencies
* a559ca2 build: istanbul@0.3.17
* c398a99 deps: type-is@~1.6.6
* 1cea9ce deps: accepts@~1.2.12
* e33c503 deps: path-to-regexp@0.1.7
* 9848645 Merge tag '3.21.2'
| \
| * cb59086 3.21.2
| * ce087e5 build: marked@0.3.5
| * 93dd15c deps: connect@2.30.2
| * b53feaa deps: vary@~1.0.1
| * d51d1ea build: ejs@2.3.3
| * fc95112 build: should@7.0.2
* | 659c0b1 deps: update example dependencies
* | 09c80bf deps: array-flatten@1.1.1
* | de7ffca tests: add test for matching route after error
```

# Introducción

- **C**lientes



**Tortoise Git**

<https://tortoisegit.org>

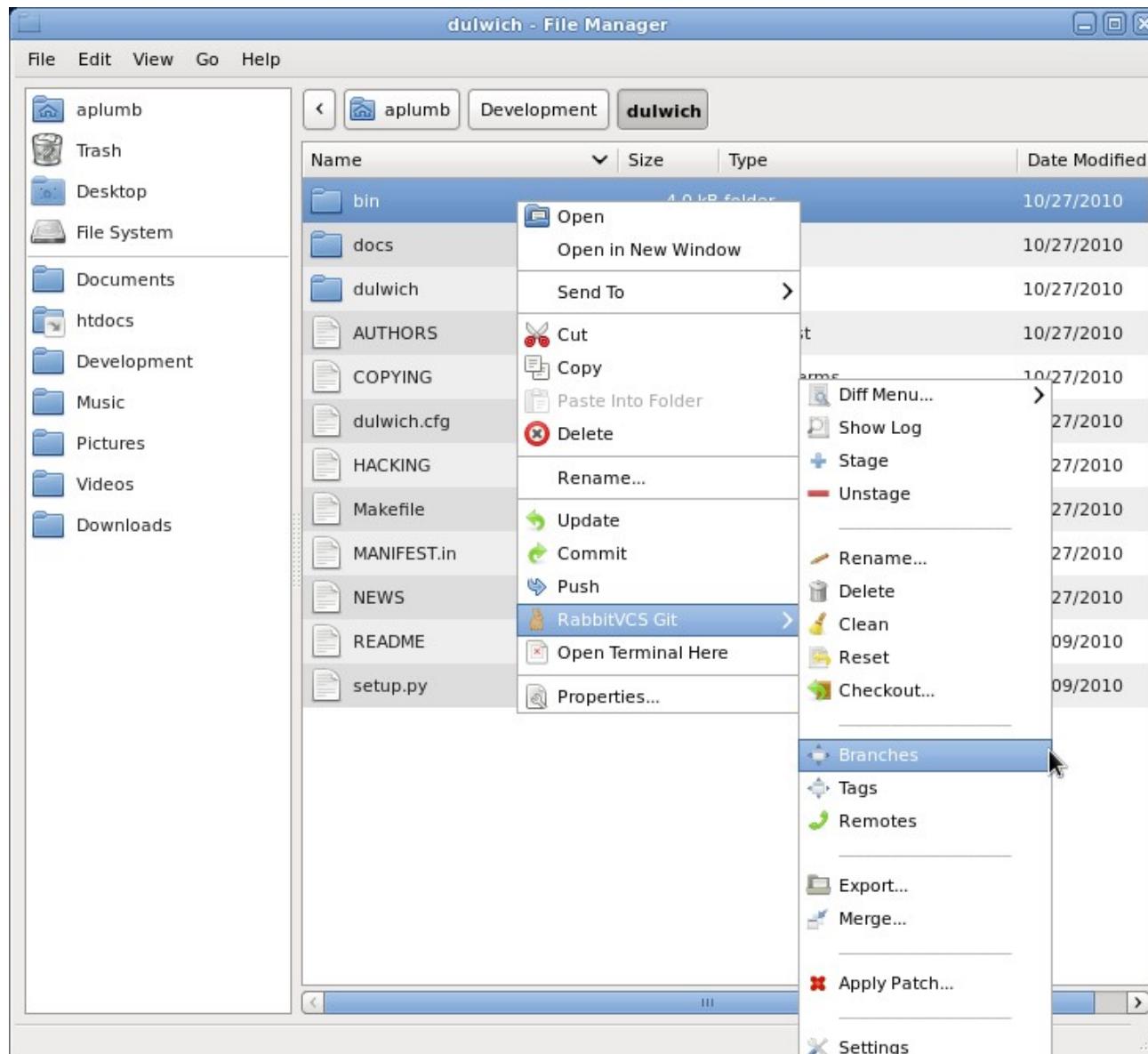


**RabbitVCS**

<http://wiki.rabbitvcs.org/wiki/install/ubuntu>

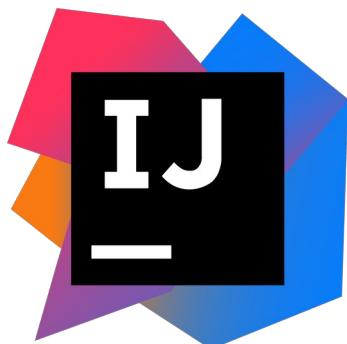
Integrado en el explorador de ficheros

# Introducción



# Introducción

- Clientes



Visual Studio Code



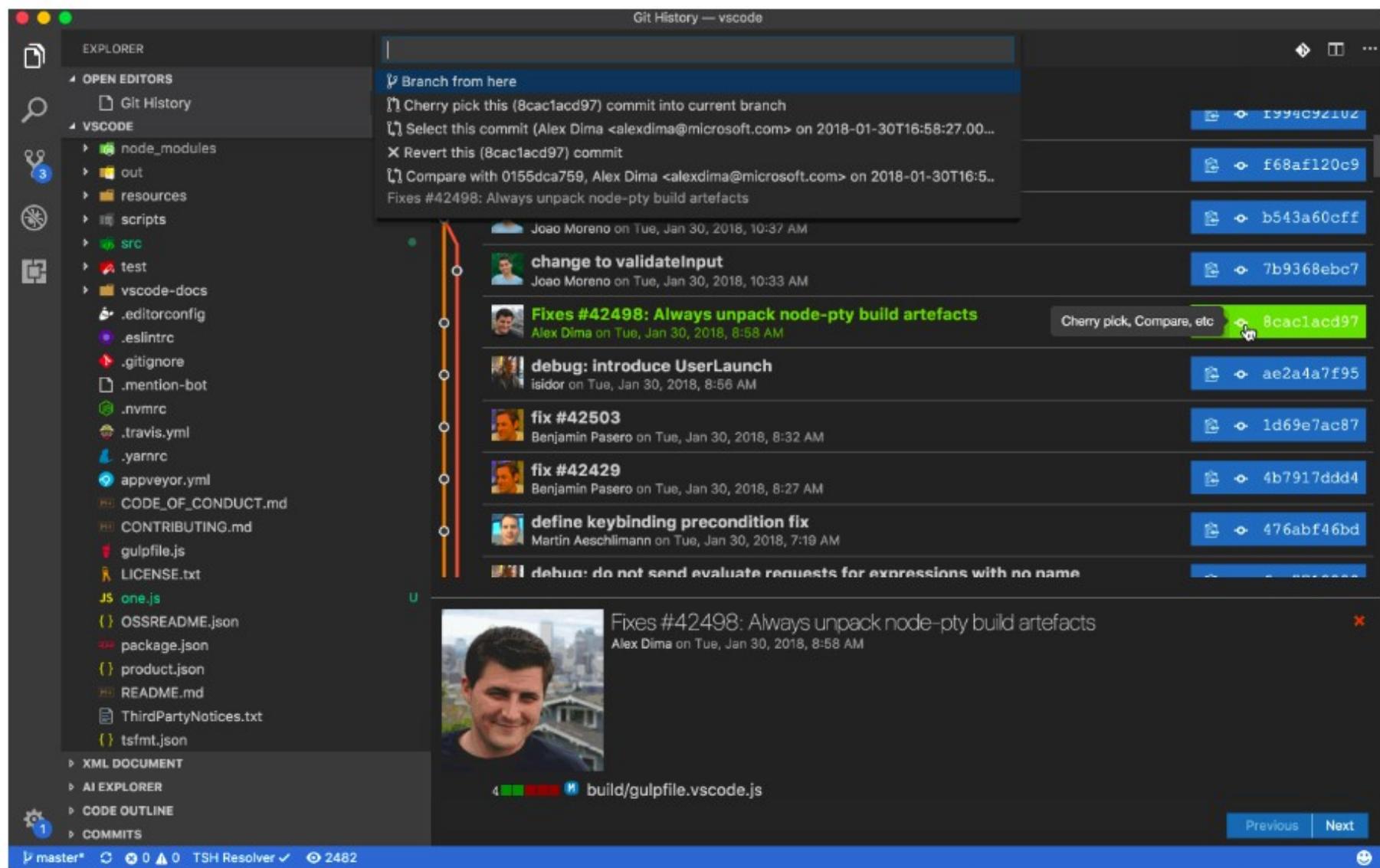
NetBeans



eclipse

Integrado en los editores / IDEs

# Introducción



# Introducción

- Clientes



<https://boostlog.io/@nixus89896/top-10-git-gui-clients-5b3336b244deba0054047685>

Clientes gráficos

# Introducción

sourcetree-website (Git)

Commit Pull Push Branch Merge Shelve Show in Finder Terminal Settings

WORKSPACE File status History Search

BRANCHES

BOOKMARKS

TAGS

REMOTES

SHELVED

SUBREPOSITORIES

All Branches Show Remote Branches Ancestor Order Jump to:

Graph	Commit	Author	Description	Date
b7358c7	Rahul Chhab...	↑ master	origin/master origin/HEAD Removing ol...	Mar 3, 2016, 11:...
bdb8bef	Rahul Chhab...	Merged in update-google-verification (pull request #14)		Feb 18, 2016, 1:3...
dfe975d	Tyler Tadej...	↑ origin/update-google-verification	Update google verificati...	Feb 11, 2016, 2:2...
3bc3290	Tyler Tadej...	Replace outdated Atlassian logo in footer with base-64 en...	Feb 11, 2016, 2:1...	
dba47f9	Tyler Tadej...	Add gitignore		Feb 11, 2016, 1:3...
ff67b45	Mike Minns...	Updated Mac min-spec to 10.10		Feb 15, 2016, 11:...
72d32a8	Michael Min...	Merged in hero_images (pull request #13)		Feb 15, 2016, 10:...
246c4ff	Joel Unger...	↑ origin/hero_images ↑ hero_images Used Tinypng to c...		Feb 11, 2016, 3:3...
9d9438c	Joel Unger...	Replacing hero images with new version of SourceTree		Feb 9, 2016, 2:59...
ce75b63	Michael Min...	Merged in bug/date-https (pull request #12)		Feb 15, 2016, 10:...
85367bb	Patrick Tho...	↑ origin/bug/date-https fixed date and https errors		Jan 7, 2016, 12:2...
4f9b557	Joel Unger...	New Favicon		Feb 8, 2016, 3:55...
384e6d5	Rahul Chhab...	↑ origin/search-console-access search console google ver...		Feb 3, 2016, 2:09...
6fa47a9	Mike Minns...	updated to move supported version to OSX 10.9+		Dec 15, 2015, 2:0...
8dd87bb	Mike Minns...	remove extra , when a line is skipped due to empty server		Nov 23, 2015, 2:2...
faa195e	Mike Minns...	Skip records with empty server/user id as gas rejects them		Nov 23, 2015, 2:1...
0cdfe96	Mike Minns...	corrected paths after merge		Nov 23, 2015, 2:0...
051ab1b	Mike Minns...	corrected column counting		Nov 23, 2015, 1:5...
a723bc2	Mike Minns...	Merge branch 'au2gex'		Nov 23, 2015, 1:5...
65fd580	Mike Minns...	deal with invalid instanceids		Nov 23, 2015, 1:5...
500a892	Michael Min...	Merged in au2gex (pull request #11)		Nov 23, 2015, 1:0...

# Introducción

- Servidores



# Introducción

- GitHub es un servicio para desarrolladores que proporciona repositorios git en sus servidores
  - **Gratis** para repositorios públicos y privados
  - **Funcionalidades adicionales:**
    - Bug reports, wiki, releases, Pull Requests, Integración Continua...



<https://github.com/>

# Introducción

The screenshot shows the GitHub Features page. At the top, there's a navigation bar with links for "Why GitHub?", "Business", "Explore", "Marketplace", and "Pricing". A search bar and "Sign in" and "Sign up" buttons are also present. The main heading is "How developers work" with the subtext "Support your workflow with lightweight tools and features. Then work how you work best—we'll follow your lead." Below this, there's a button labeled "New to GitHub? See how it works" with a play icon. At the bottom, there are seven icons representing different GitHub features: "Code review" (document with checkmark), "Project management" (checklist), "Integrations" (two overlapping boxes), "Team management" (four user icons), "Social coding" (two people talking), "Documentation" (book with plus sign), and "Code hosting" (cloud with server). The URL in the browser is <https://github.com/features>.

Features - The right tools for th x +

← → C 🔒 GitHub, Inc. [US] | https://github.com/features

Search GitHub

Sign in Sign up

Why GitHub? Business Explore Marketplace Pricing

How developers work

Support your workflow with lightweight tools and features. Then work how you work best—we'll follow your lead.

New to GitHub? See how it works

Code review Project management Integrations Team management Social coding Documentation Code hosting

22

# Conceptos básicos

- **¿Cómo se crea un repositorio?**
  - Se puede crear localmente y luego publicarlo en GitHub
  - Se puede crear en GitHub y descargarlo (**clonar**) (Más sencillo)
- **¿Qué es un repositorio?**
  - Es una carpeta en disco con una subcarpeta llamada .git
- **¿Qué se guarda un repositorio?**
  - Todas las versiones de todos los ficheros
  - Quién modificó cada línea de cada fichero (ficheros de texto)
  - Para ahorrar espacio se comprimen los ficheros del repositorio

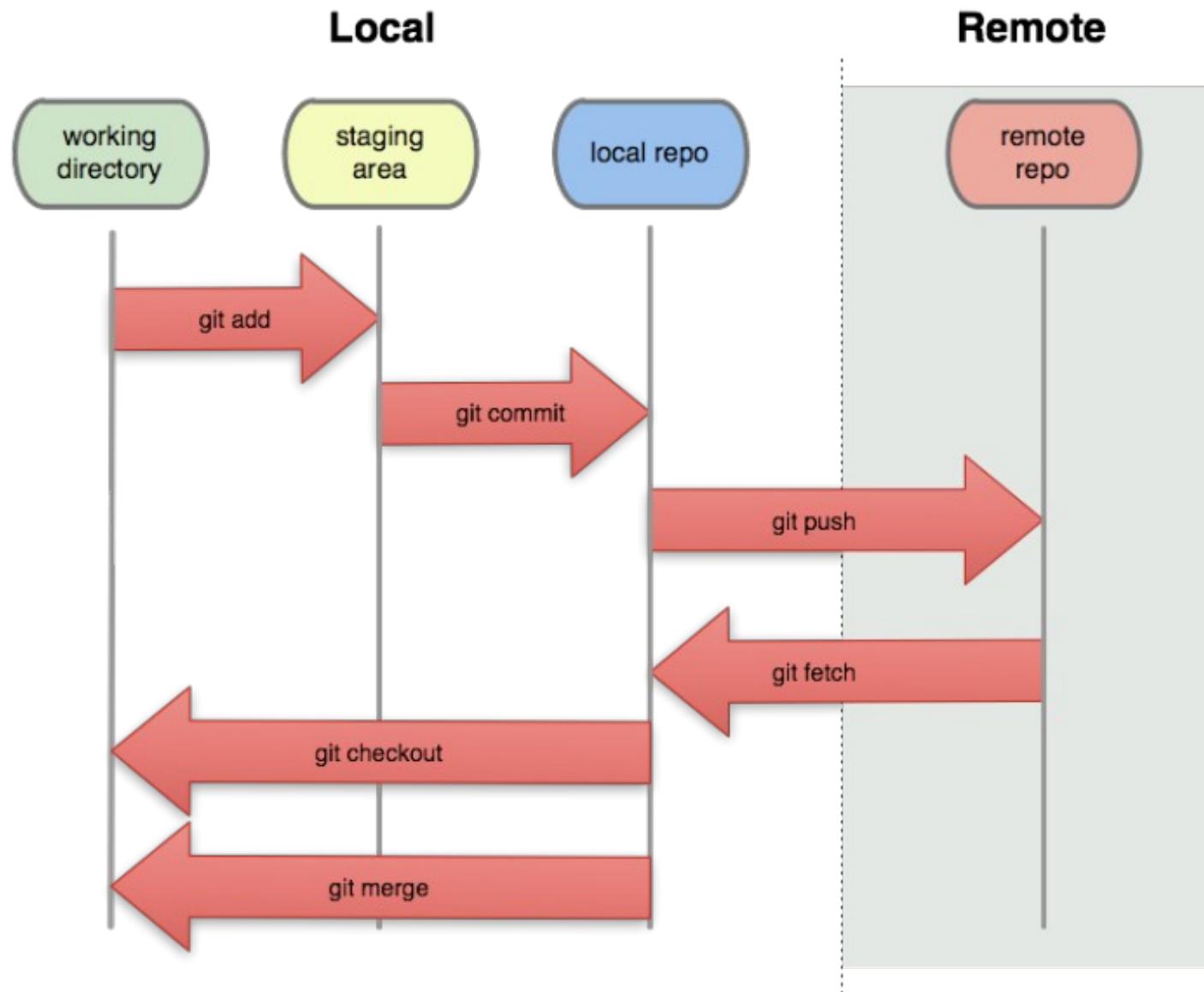
# Conceptos básicos

- **¿Cómo se trabaja con un repositorio?**
  - **Paso 1)** El desarrollador crea y edita los ficheros de forma habitual con editores o cualquier programa (son **ficheros normales** en disco)
  - **Paso 2)** Cuando se completa una funcionalidad, se indican los ficheros nuevos o los que han cambiado (**add**) y se crea la nueva mini-versión (**commit**)
  - **Paso 3)** De vez en cuando se suben las mini-versiones que tenemos en local al servidor compartido (**push**)
  - **Paso 4)** De vez en cuando se descargan los cambios del servidor compartido que han realizado otros programadores al repositorio local (**fetch o pull**)

# Conceptos básicos

- **Zonas en las que está un fichero**
  - Carpeta raíz del proyecto (**working directory**)
    - Es el espacio donde el desarrollador crea y modifica los ficheros
    - Si se va a trabajar con un repositorio existente, los ficheros se obtienen de una mini-versión (**commit**) alojada en el directorio git (usando el comando **checkout**)
  - El directorio .git (**local repo**)
    - Contiene todas las versiones de los ficheros del repositorio y los metadatos
    - Es lo que se copia cuando se clona un repositorio
    - Es el contenido de la carpeta .git
  - **Staging** (no es una carpeta real, es una lista de ficheros)
    - Listado de ficheros que se añadirán en la próxima mini-versión (**commit**)

# Conceptos básicos



# Conceptos básicos

- Los 4 estados (de los ficheros)

Untracked

El fichero se acaba de crear y todavía no se ha añadido al repositorio

Modified

El fichero ha sido modificado desde la última vez que se añadió al repositorio

Staged

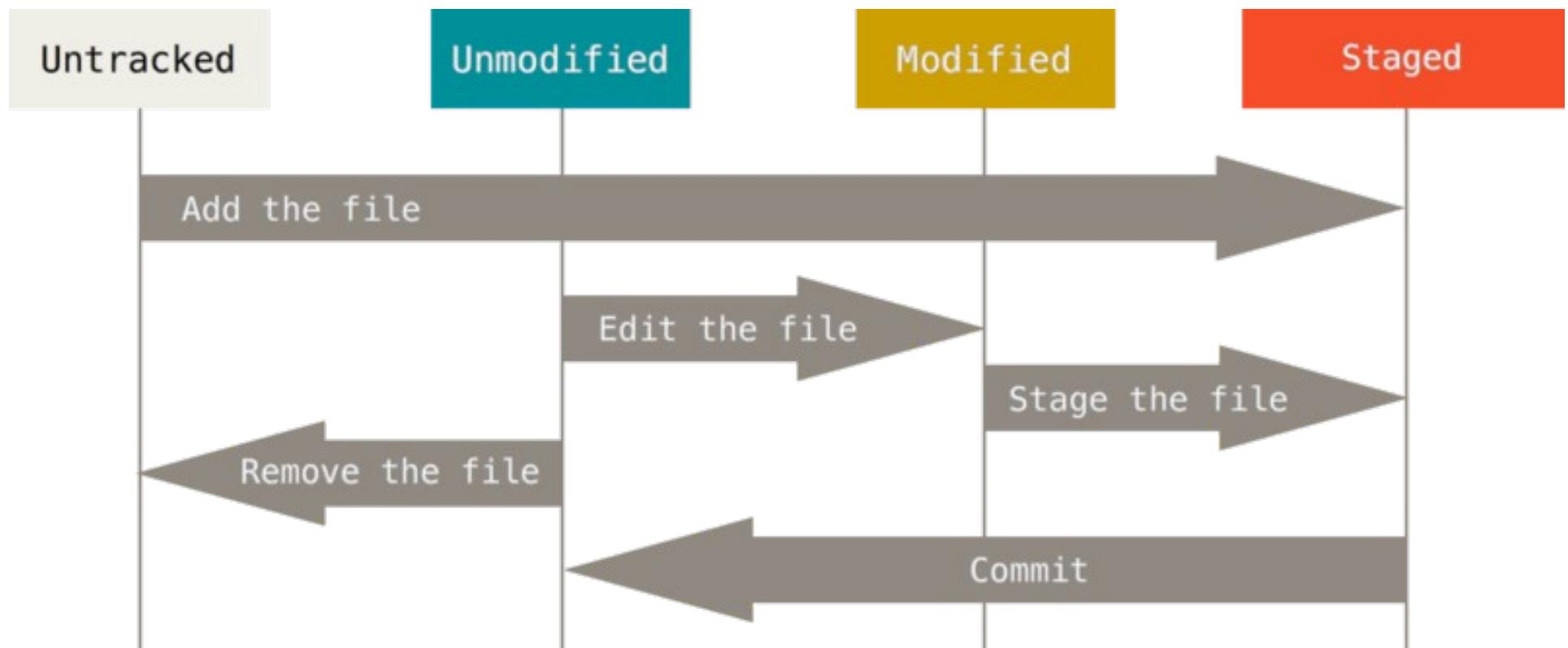
El fichero ha sido modificado y se ha marcado para formar parte de la próxima mini-versión (commit)

Unmodified

El fichero forma parte de una mini-versión (commit) del repositorio y no se ha modificado

# Conceptos básicos

- Los 4 estados (de los ficheros)



# Conceptos básicos

- **Commits (mini-versiones)**

- Cuando se quiere registrar una mini-versión del proyecto se hace un **commit**
- El commit tiene un **mensaje** indicando qué cambios se han introducido en el código en ese commit (ideal para analizar el histórico del proyecto)
- Cada commit se identifica con un **número muy largo**

```
commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

        removed unnecessary test
```

- Como es tan largo, se suelen usar únicamente los primeros 7 caracteres (en este caso 085bb3b)

# Conceptos básicos

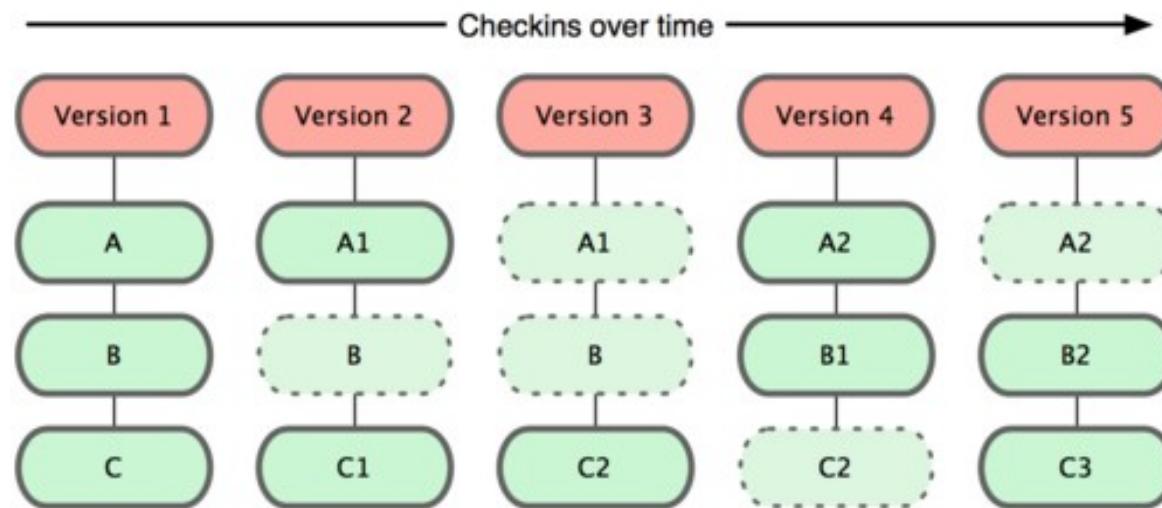
- **Commits (mini-versiones)**

- **¿Por qué usar ese número “tan raro”?**
  - En otros sistemas se usa un número incremental (1, 2, 3 ...) pero eso tiene problemas cuando dos desarrolladores cambian el mismo fichero en su propia máquina (antes de sincronizar con el servidor)
- **¿Cómo se obtiene ese número?**
  - Usando la función Hash SHA1 con la información del commit (autor, mensaje, commit anterior, etc.)
  - Se suele usar la expresión “el hash del commit” en vez de “número del commit” o “id del commit”

# Conceptos básicos

- **Snapshots**

- En general, por cada commit no se guardan las diferencias de los ficheros que han cambiado. Se guardan los ficheros completos (para que sea rápido obtener un commit concreto)



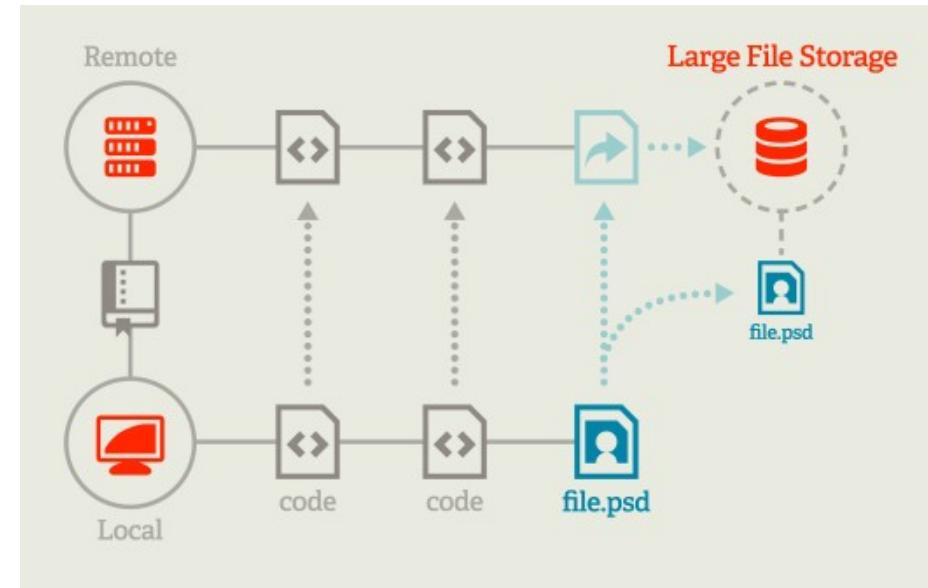
# Conceptos básicos

- **Snapshots**

- Un repositorio git no está diseñado para guardar grandes ficheros binarios que cambian frecuentemente porque se guardan todas las versiones
- Existen extensiones de git para estos casos



**Git Large File Storage**



# Desarrollo colaborativo con Git

- Introducción: Git y GitHub
- **Git: trabajando en local**
- Git y GitHub: trabajando en equipo
- Gestionando el repositorio
- Modelos de desarrollo

# Instalación del cliente git

- Instalación rápida

- Windows:

- <https://git-scm.com/download/win>

- Mantener valores por defecto

- Linux (diferentes sabores)

- <https://git-scm.com/download/linux>

- MacOS

- <https://git-scm.com/download/mac>

# Instalación del cliente git

- Comprobación rápida
  - Windows:
    - Busca la aplicación Git bash y ábrela
    - Es una línea de comandos tipo unix
  - Linux y MacOS: abrir la línea de comandos
  - Comprobar que git está instalado y en el PATH:

```
$ git --version  
git version 2.17.1
```

# Instalación del cliente git

- **La identidad (del desarrollador)**
  - Cada commit lleva asociado la identidad del desarrollador que lo hizo
  - La identidad está formada por nombre y correo electrónico
  - Es importante tener bien configurados estos datos
    - Dos commits realizados desde máquinas diferentes pueden “parecer” que pertenecen a usuarios diferentes
    - Es posible que haya problemas de acceso al repositorio compartido en el servidor

# Instalación del cliente git

- La identidad (del desarrollador)
  - Los datos de identidad se suelen configurar a nivel de usuario en el sistema operativo
    - Fichero .gitconfig en la carpeta HOME del usuario
      - /home/<user>/.gitconfig
      - C:\Users\<user>\.gitconfig
    - El comando “config” permite configurar el fichero
      - git config --global user.name “micaelgallego”
      - git config --global user.email “micael.gallego@gmail.com”
      - cat \$HOME/.gitconfig

# Crear el repositorio en local

```
~$ mkdir repo  
~$ cd repo  
~/repo$ git init  
Initialized empty Git repository in /home/mica/repo/.git/  
~/repo$ git status  
On branch master  
No commits yet  
nothing to commit (create/copy files and use "git add" to track)  
~/repo$ git log  
fatal: your current branch 'master' does not have any commits yet
```

# Trabajando en el repositorio

- **Instalar Visual Studio Code**

- Editor ligero
- Con plugins para multitud de lenguajes de programación
- Tiene plugin de git básico instalado
- Se pueden instalar más plugins adicionales
- Nos permitirá modificar los ficheros sin tener que hacerlo desde la línea de comandos
- Versiones para Win, Mac y Linux
- <https://code.visualstudio.com/>

# Trabajando en el repositorio

The screenshot shows a dark-themed code editor interface. On the left is the Explorer sidebar, which lists the project structure under 'OPEN EDITORS'. The file 'timing.js' is selected in the list. The main editor area displays the code for 'timing.js', which is a script for displaying performance data. The bottom status bar shows the current branch is 'master', there are 6 changes (6 down), 0 errors, 0 warnings, and 0 info. The bottom right corner shows the page number '40'.

```
function alignRight(str, len, ch) {
    return new Array(len - str.length + 1).join(ch || " ") + str;
}

const enabled = !!process.env.TIMING;

const HEADERS = ["Rule", "Time (ms)", "Relative"];
const ALIGN = [alignLeft, alignRight, alignRight];

/* istanbul ignore next */
/** 
 * display the data
 * @param {Object} data Data object to be displayed
 * @returns {string} modified string
 * @private
 */
function display(data) {
    let total = 0;
    const rows = Object.keys(data)
        .map(key => {
            const time = data[key];

            total += time;
            return [key, time];
        })
        .sort((a, b) => b[1] - a[1])
        .slice(0, 10);

    rows.forEach(row => {
        row.push(` ${(row[1] * 100 / total).toFixed(1)}%`);
        row[1] = row[1].toFixed(3);
    });
    rows.unshift(HEADERS);
    const widths = [];

    for (let i = 0; i < rows[0].length; i++) {
        let width = 0;
        for (let j = 0; j < rows.length; j++) {
            if (rows[j][i] === undefined) {
                break;
            }
            width = Math.max(width, rows[j][i].length);
        }
        widths.push(width);
    }

    return rows.map((row, index) => {
        const formattedRow = row.map((value, index) => {
            if (index === 0) {
                return value.padStart(widths[0], " ");
            } else if (index === 1) {
                return value.padEnd(widths[1], " ");
            } else {
                return value.padEnd(widths[2], " ");
            }
        });
        return formattedRow;
    });
}
```

# Git

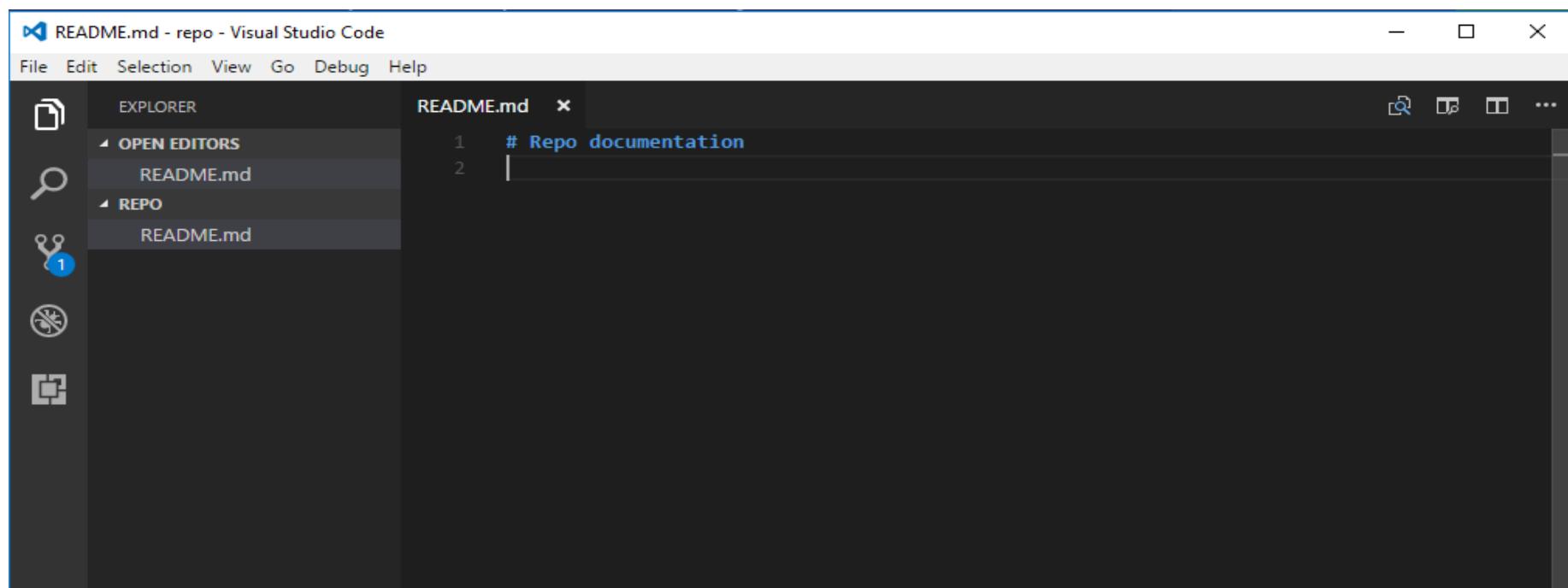
- **Ejemplo 1**

- Crear un fichero README.md con texto
- Añadirlo al repositorio: comitear el fichero README.md
- Verificar que tenemos dos commits en el repo

# Crear un fichero y añadirlo al repositorio

- Crear un fichero README.md en la raiz del repositorio

```
$ code .
```



# Crear un fichero y añadirlo al repositorio

```
~/repo$ git status
```

On branch master

No commits yet

Untracked files:

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

README.md

nothing added to commit but untracked files present (use "git add" to track)

Instrucciones sobre  
cómo proceder a  
continuación



# Crear un fichero y añadirlo al repositorio

```
~/repo$ git add README.md
```

```
~/repo$ git status
```

On branch master

No commits yet

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

new file: README.md

Instrucciones sobre  
cómo revertir los  
cambios

Qué se va a  
añadir al repo

# Crear un fichero y añadirlo al repositorio

```
~/repo$ git commit -m "Add README"  
[master (root-commit) 85fe787] Add README  
 1 file changed, 1 insertion(+)  
 create mode 100644 README.md
```

Working area limpia,  
todos los cambios se  
incluyeron en el repo

```
~/repo$ git status  
On branch master  
nothing to commit, working tree clean
```



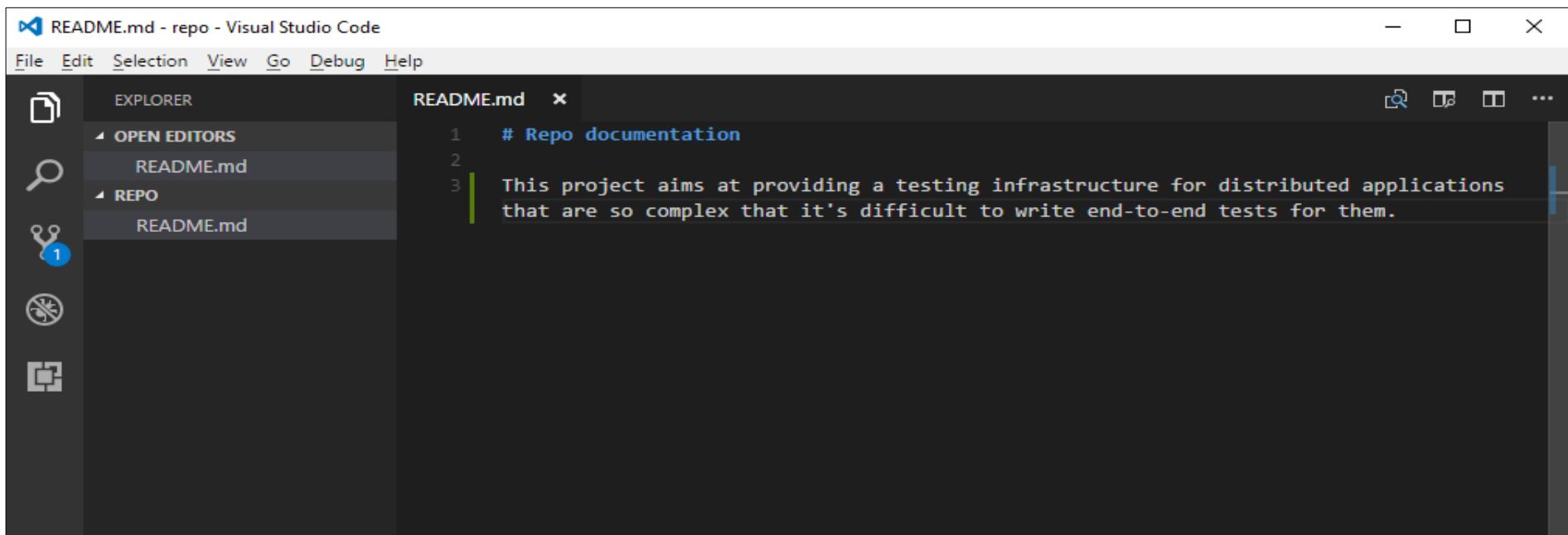
```
~/repo$ git log  
commit 85fe787f225a1df8ea03803d8abf0cc581465372 (HEAD -> master)  
Author: patxigortazar <patxi.gortazar@urjc.es>  
Date:   Mon May 15 01:00:34 2017 -0700  
  
        Add README
```

Lista de commits



# Subir una actualización a un fichero

- Editar el fichero README.md añadiendo contenido



The screenshot shows the Visual Studio Code interface with the following details:

- Title Bar:** README.md - repo - Visual Studio Code
- Menu Bar:** File Edit Selection View Go Debug Help
- Explorer Sidebar:** Shows two entries under "OPEN EDITORS": "README.md" and "REPO". Under "REPO", there is also a "README.md" entry.
- Editor Area:** Displays the content of the README.md file.

```
1 # Repo documentation
2
3 This project aims at providing a testing infrastructure for distributed applications
that are so complex that it's difficult to write end-to-end tests for them.
```

# Subir una actualización a un fichero

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

    modified:   README.md

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

# Trabajo con el repositorio

- **Ejercicio 2**

- Añadir un fichero LICENSE y editar el fichero README.md
- Comitear el fichero LICENSE primero
- Comitear el fichero README.md después

# Trabajo con el repositorio

- Consulta de la historia del proyecto

- El comando “log” muestra por defecto bastante información de los commits
- Existen opciones que permiten un histórico más compacto

```
$ git log --pretty=format:"%h - %an, %ar : %s"
ca82a6d - Scott Chacon, 6 years ago : changed the version number
085bb3b - Scott Chacon, 6 years ago : removed unnecessary test
a11bef0 - Scott Chacon, 6 years ago : first commit
```

# Buenas prácticas en los commits

- **Commits pequeños**

- Cada cambio debería tener su propio commit
- Así es más fácil identificar qué ha cambiado por el mensaje del commit
- Es más sencillo de gestionar el commit para resolver conflictos, revisar su contenido por otros desarrolladores, etc.
- En cada commit el proyecto tiene que compilar y pasar los tests (si tiene)

# Buenas prácticas en los commits

- Buenas prácticas en mensajes de commits

	COMMENT	DATE
O	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
O	ENABLED CONFIG FILE PARSING	9 HOURS AGO
O	MISC BUGFIXES	5 HOURS AGO
O	CODE ADDITIONS/EDITS	4 HOURS AGO
O	MORE CODE	4 HOURS AGO
O	HERE HAVE CODE	4 HOURS AGO
O	AAAAAAA	3 HOURS AGO
O	ADKFJSLKDFJSDFKLJ	3 HOURS AGO
O	MY HANDS ARE TYPING WORDS	2 HOURS AGO
O	HAAAAAAAAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

# Buenas prácticas en los commits

- **Buenas prácticas en mensajes de commits**

1. Separa el asunto del cuerpo con una línea en blanco
2. Limita el asunto a 50 caracteres
3. Inicia el asunto con mayúsculas
4. No finalices el asunto con un punto
5. Usa el verbo imperativo (no el pasado) en el asunto
6. No uses líneas mayores de 72 caracteres
7. Usa el cuerpo para explicar qué y por qué, no cómo (está en el código)

<https://chris.beams.io/posts/git-commit/>

<https://github.com/erlang/otp/wiki/writing-good-commit-messages>

# Buenas prácticas en los commits

## • Buenas prácticas en mensajes de commits

```
commit eb0b56b19017ab5c16c745e6da39c53126924ed6
```

```
Author: Pieter Wuille <pieter.wuille@gmail.com>
```

```
Date: Fri Aug 1 22:57:55 2014 +0200
```

Simplify serialize.h's exception handling

Remove the 'state' and 'exceptmask' from serialize.h's stream implementations, as well as related methods.

As exceptmask always included 'failbit', and setstate was always called with bits = failbit, all it did was immediately raise an exception. Get rid of those variables, and replace the setstate with direct exception throwing (which also removes some dead code).

As a result, good() is never reached after a failure (there are only 2 calls, one of which is in tests), and can just be replaced by !eof().

fail(), clear(n) and exceptions() are just never called. Delete them.

# Buenas prácticas en los commits

- **Buenas prácticas en mensajes de commits**

- El cuerpo del mensaje debería responder a:
  - ¿Por qué el cambio es necesario?
  - ¿Cómo resuelve el problema?
  - ¿Qué efectos colaterales (*side effects*) tiene el cambio?
- Si el proyecto usa un sistema para gestión de issues (features o bugs reports) incluye la URL del issue que motiva este cambio

# Buenas prácticas en los commits

- **Buenas prácticas en mensajes de commits**
  - Algunos proyectos define un prefijo para identificar el tipo de commit

## Formato

```
<type>(<scope>): <subject>
<BLANK LINE>
<body>
<BLANK LINE>
<footer>
```

### Type

- **feat:** A new feature
- **fix:** A bug fix
- **docs:** Documentation only changes
- **style:** Changes that do not affect the meaning of the code (white-space, formatting..)
- **refactor:** A code change that neither fixes a bug or adds a feature
- **perf:** A code change that improves performance
- **test:** Adding missing tests
- **chore:** Changes to the build process or auxiliary tools and libraries

### Scope

- The scope could be anything specifying place or module of the commit change

### Footer

- The footer should contain any information about Breaking Changes and is also the place to reference GitHub issues that this commit Closes.

<http://karma-runner.github.io/3.0/dev/git-commit-msg.html>

<https://gist.github.com/brianclements/841ea7bffdbo1346392c>

# Ramas (*branches*)

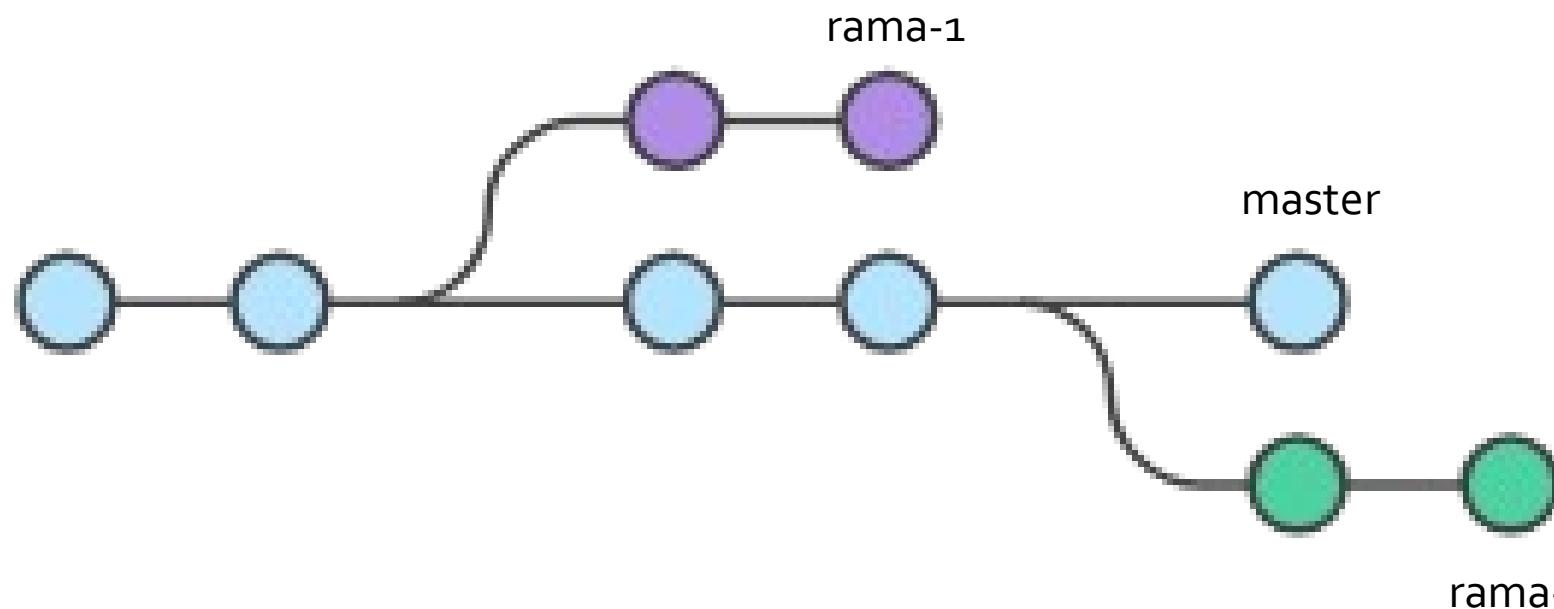
- Hemos visto cómo añadir commits al repositorio (mini-versiones)
- En realidad se añaden a una zona llamada **rama master**
- Cuando el código de la rama master está “estable”, se publicará una **nueva release**



# Ramas (*branches*)

- Creación de una rama

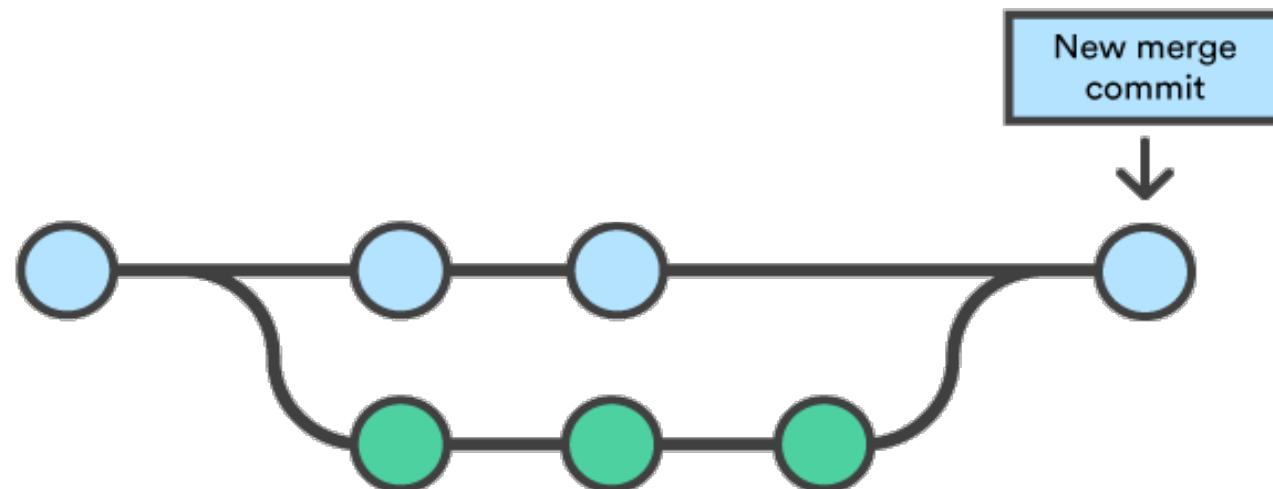
- Una rama se puede crear partiendo de cualquier commit del repositorio
- Es como hacer una copia de un documento para poder trabajar en dos “líneas de trabajo” en paralelo



# Ramas (*branches*)

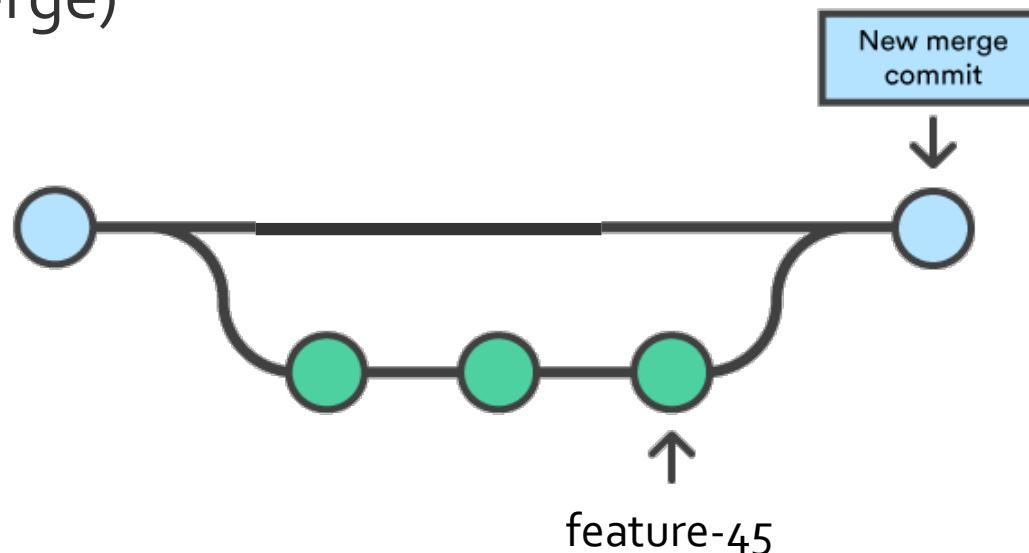
- **Mezcla de una rama**

- En cualquier momento se puede mezclar el código de dos ramas diferentes
- Se suele utilizar para integrar el trabajo de una rama en la rama master



# Ramas (*branches*)

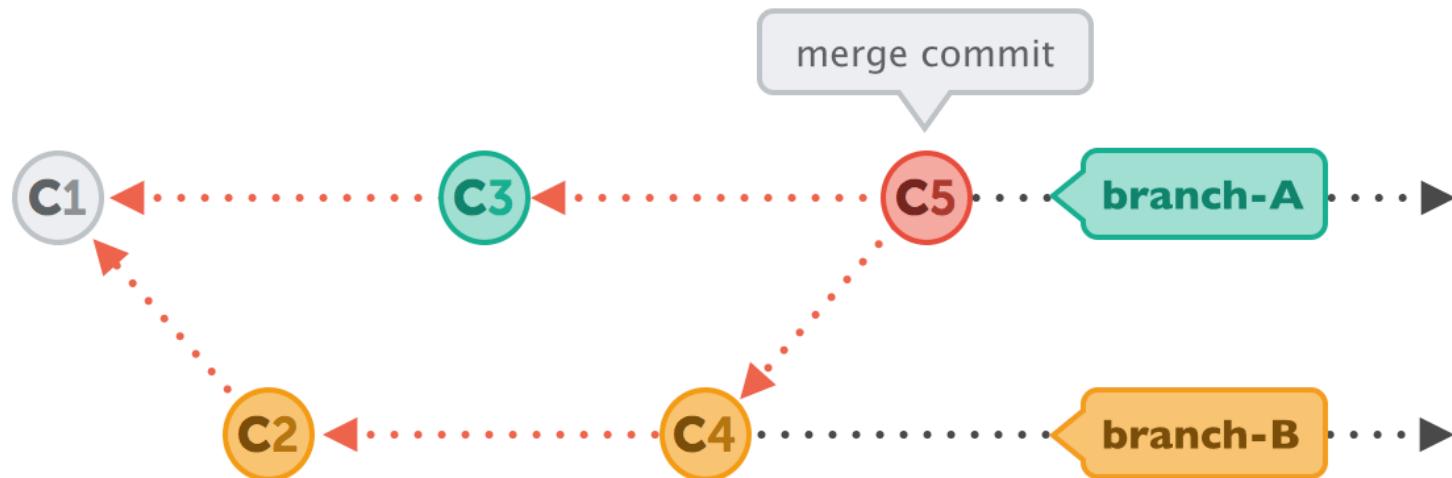
- Cada funcionalidad en una rama (*branch-per-feature*)
  - Existe una estrategia de desarrollo en la que cada nueva funcionalidad se implementa en una nueva rama
  - Cuando la funcionalidad se termina, se mezcla con master
  - En master no se hacen commits “normales”, sólo mezclas de ramas (merge)



# Ramas (*branches*)

- **Ramas y commits**

- Los commits de mezcla (merge) se basan en dos commits anteriores
- El resto de commits, únicamente se basan en un commit anterior que se conoce como *parent commit*



# Ramas (*branches*)

- Comandos para trabajar con ramas

- Crear una rama

- ```
$ git branch feature1
```

- Ver las ramas de un repositorio

- ```
$ git branch --list
```

- Borrar una rama

- ```
$ git branch -d <rama>
```

- Trabajar en una rama

- ```
$ git checkout <rama>
```

- Crear una nueva rama y trabajar en ella

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

# Ramas (*branches*)

- **Ejercicio 3**

- Añadir una rama llamada “documentation”
- En dicha rama:
  - Crear una carpeta docs
  - Movernos a la carpeta docs
  - Añadir los siguientes ficheros (no importa el contenido):
    - index.html
    - styles.css
- Comitear

# Ramas (*branches*)

- **Ejercicio 4**

- Cambiar a master (hacer un checkout sin -b)
  - `git checkout master`
- En master:
  - Modificar el fichero README.md
  - Comitear

# Ramas (*branches*)

- Estado actual del repositorio

```
$ git log --graph --all --oneline
* 1450e35 (HEAD -> master) Add new section to README.md
| * 09d9e74 (documentation) Add documentation
|/
* 52046cc Improve readme again
* 95cb647 Add LICENSE
* e815802 Improve readme
* 85fe787 Add README
```

# Ramas (*branches*)

- Referencias

- HEAD, master, documentation son referencias a commits
- Existe una referencia por cada rama
- HEAD es una referencia que indica el commit del repositorio que ha cargado el directorio de trabajo
- HEAD apunta al commit que será parent commit del nuevo commit que hagamos
- Es posible que HEAD apunte a un commit del pasado (para consulta), pero si hacemos cambios y queremos hacer un nuevo commit con ellos, debemos crear una rama nueva en ese commit

```
$ git checkout <commit>
```

# Ramas (*branches*)

- Volvamos a la rama documentación y añadamos un par de commits más

```
$ git checkout documentation
$ echo "A new file" > file1.txt
$ git status
$ git add file1.txt
$ git commit -m "Add file1"
$ echo "Another file" > file2.txt
$ git add file2.txt
$ git commit -m "Add file2"
```

# Ramas (*branches*)

- Veamos nuestro árbol

```
$ git log --graph --all --oneline
```

```
* 39aec4f (HEAD -> documentation) Add file2  
* 4605118 Add file1  
* 09d9e74 Add documentation
```

```
| * 1450e35 (master) Add new section to README.md
```

```
| /
```

```
* 52046cc Improve readme again  
* 95cb647 Add LICENSE  
* e815802 Improve readme  
* 85fe787 Add README
```

# Ramas (*branches*)

- Vamos a incorporar la documentación a la rama master
- Hay tres estrategias diferentes
  - Merge
  - Merge con squash
  - Fast-forward merge

# Ramas (*branches*)

- **Merge**

- Se crea un nuevo commit basado en el último commit de master y en el último commit de documentation
- La rama master ahora incluye los cambios de la rama documentation
- Nos posicionamos en la rama en la que queremos continuar trabajando (master) y solicitamos el merge de la otra rama (documentation)

# Ramas (*branches*)

- Merge

```
$ git checkout master
$ git merge documentation
Merge made by the 'recursive' strategy.
 docs/index.html | 4 +++
docs/styles.css | 1 +
file1.txt        | 1 +
file2.yxy        | 1 +
4 files changed, 7 insertions(+)
create mode 100644 docs/index.html
create mode 100644 docs/styles.css
create mode 100644 file1.txt
create mode 100644 file2.yxy
```

# Ramas (*branches*)

- Merge

Commit con dos padres  
39aec4f y 1450e35

```
$ git log --graph --all --oneline
*   72c38ef (HEAD -> master) Merge branch 'documentation'
  |\ 
  * 39aec4f (documentation) Add file2
  * 4605118 Add file1
  * 09d9e74 Add documentation
* | 1450e35 Add new section to README.md
  |
* 52046cc Improve readme again
* 95cb647 Add LICENSE
* e815802 Improve readme
* 85fe787 Add README
```

# Ramas (*branches*)

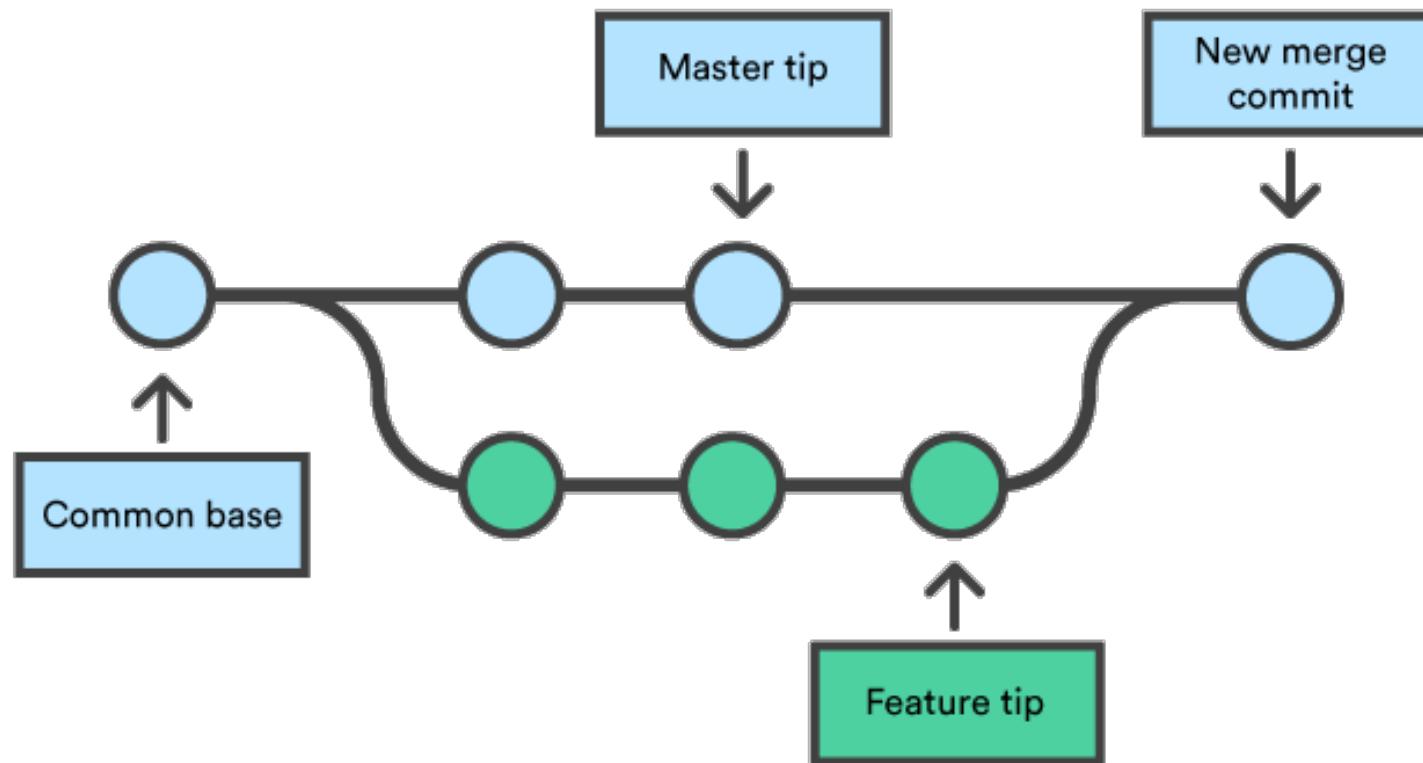
- **Merge**

Todos los commits de ambas ramas forman parte ahora de la historia de master

```
$ git log --oneline  
72c38ef (HEAD -> master) Merge branch 'documentation'  
39aec4f (documentation) Add file2  
4605118 Add file1  
1450e35 Add new section to README.md  
09d9e74 Add documentation  
52046cc Improve readme again  
95cb647 Add LICENSE  
e815802 Improve readme  
85fe787 Add README
```

# Ramas (*branches*)

- Merge



# Ramas (*branches*)

- **Merge**

- Cuando existen commits de merge no hay una evolución “lineal” del código
- Hay bifurcaciones (en los commits de merge) que en un punto del pasado nacen de un commit común
- Hay desarrolladores que prefieren mantener una historia lineal, para ello existen dos estrategias:
  - Merge con squash
  - Fast-Forward merge (con o sin rebase)

# Ramas (*branches*)

- **Merge con squash**

- Todos los commits de una rama se “fusionan” en un único commit sobre la rama destino

```
$ git merge <rama> --squash
```

- El comando deja en el directorio de trabajo la fusión de los commits
- No hace el commit para que el usuario pueda revisar y editar y haga él mismo el commit (con el mensaje que quiera)
- La historia es lineal, pero se pierde trazabilidad
- El repositorio no vincula las ramas master y documentation

# Ramas (*branches*)

- Merge con squash
  - Creamos una nueva rama (ramasquash) y hacemos dos commits en el fichero README.md
    - Add paragraph1
    - Add paragraph2
  - Vamos a master y creamos el fichero README2.md
    - Add README2.md

# Ramas (*branches*)

- Merge con squash

```
~/repo$ git log --graph --all --oneline
* 811983f (HEAD -> master) Add README2.md
| * eef095b (ramasquash) Add paragraph2
| * 3894584 Add paragraph1
|
* c0eacee (addfile) Add nuevo2.txt
* 2c0d8e2 Add nuevo.txt
* b15f345 Merge branch 'documentation'
  \
  * 4c7fab3 (documentation) Add file2
  * 2da7fc5 Add file1
  * 6dd54cc New docs
* | e5d8f72 Add new paragraph to README.md
  /
* c6b5451 Improve documentation
* a8b186e Add license
* b7a1a04 Mi segundo commit
```

# Ramas (*branches*)

- Merge con squash
  - Ejecutamos el merge con squash

```
~/repo$ git merge ramasquash --squash
Squash commit -- not updating HEAD
Automatic merge went well; stopped before committing as requested
```

# Ramas (*branches*)

- Merge con squash
  - Los cambios de la rama ramasquash se aplican al directorio de trabajo

```
~/repo$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to
unstage)

        modified:   README.md
```

- Los comitemos a master

```
~/repo$ git add README.md
~/repo$ git commit -m "Update README.md"
```

# Ramas (*branches*)

- Merge con squash

```
~/repo$ git log --graph --all --oneline
* 566ec18 (HEAD -> master) Update README.md
* 811983f Add README2.md
| * eef095b (ramasquash) Add paragraph2 ←
| * 3894584 Add paragraph1
|
* c0eacee (addfile) Add nuevo2.txt
* 2c0d8e2 Add nuevo.txt
* b15f345 Merge branch 'documentation'
  \
  * 4c7fab3 (documentation) Add file2
  * 2da7fc5 Add file1
  * 6dd54cc New docs
  * | e5d8f72 Add new paragraph to README.md
  |
* c6b5451 Improve documentation
* a8b186e Add license
```

No hay relación entre la  
rama master y la rama  
ramasquash

# Ramas (*branches*)

- **Fast-forward merge**
  - Cuando no ha habido commits en master desde que se creó la otra rama, no es necesario crear un commit de merge
  - Basta con hacer que la rama master apunte al último commit de la rama que se quiere mezclar
  - Cuando se puede evitar el commit de merge se llama **fast-forward**
  - Se puede forzar la creación del commit merge con la opción **--no-ff**

# Ramas (*branches*)

- Fast-forward merge
  - Creamos una rama addfile y añadimos dos ficheros

```
$ git checkout addfile
$ echo "New file" > new.txt
$ git add new.txt
$ git commit -m "Add new.txt"
$ echo "New file2" > new2.txt
$ git add new2.txt
$ git commit -m "Add new2.txt"
```

# Ramas (*branches*)

- Fast-forward merge

```
$ git log --graph --all --oneline
* c0eacee (HEAD -> addfile) Add new2.txt
* 2c0d8e2 Add new.txt
* b15f345 (master) Merge branch 'documentation'
  \
   * 4c7fab3 (documentation) Add file2
   * 2da7fc5 Add file1
   * 6dd54cc New docs
* | e5d8f72 Add new paragraph to README.md
  /
* c6b5451 Improve documentation
* a8b186e Add license
* b7a1a04 Mi segundo commit
* ed11fcf Add README
```

# Ramas (*branches*)

- **Fast-forward merge**

Se ha detectado que el merge puede hacerse con fast-forward (sin crear commit de merge)

```
$ git checkout master
$ git merge addfile
Updating b15f345..c0eacee
Fast-forward
  new.txt    | 1 +
  new2.txt   | 1 +
  2 files changed, 2 insertions(+)
  create mode 100644 new.txt
  create mode 100644 new2.txt
```

# Ramas (*branches*)

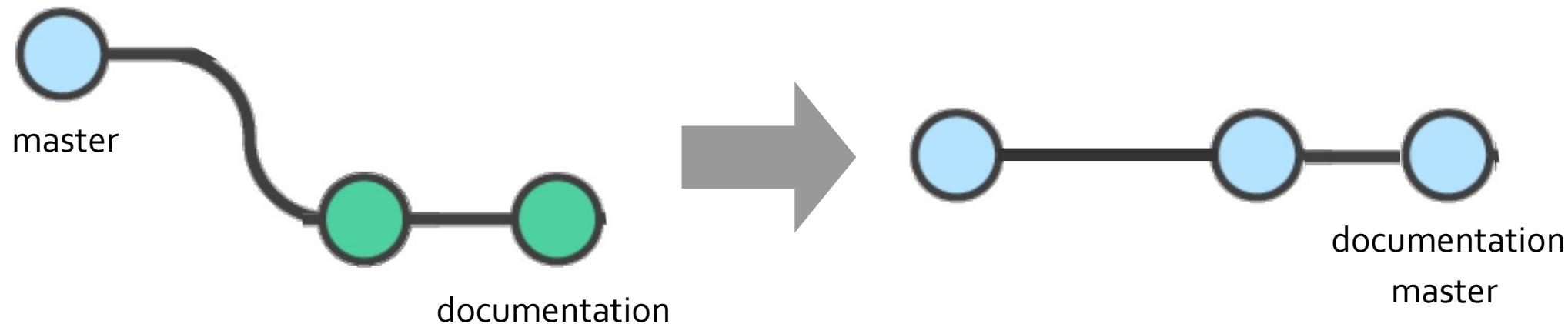
- Fast-forward merge

```
$ git log --graph --all --oneline
* c0eacee (HEAD -> master, addfile) Add new2.txt
* 2c0d8e2 Add new.txt
*   b15f345 Merge branch 'documentation'
  \
  * 4c7fab3 (documentation) Add file2
  * 2da7fc5 Add file1
  * 6dd54cc New docs
* | e5d8f72 Add new paragraph to README.md
  /
* c6b5451 Improve documentation
* a8b186e Add license
* b7a1a04 Mi segundo commit
* ed11fcf Add README
```

La rama master apunta  
al último commit de la  
rama addfile.  
No hay commit de merge

# Ramas (*branches*)

- Fast-forward merge



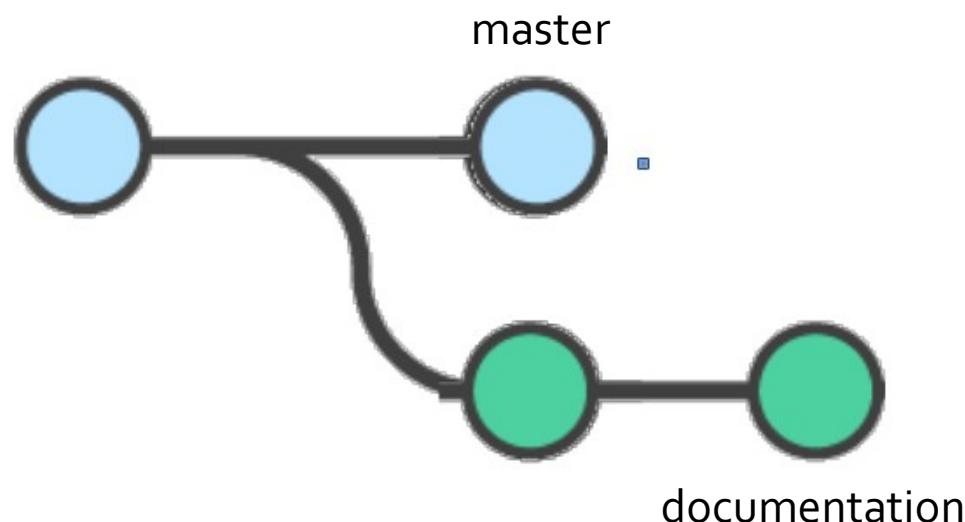
Sólo se cambia la referencia de la rama master

# Ramas (*branches*)

- **Fast-forward merge**
  - El resultado obtenido con el **merge con fast-forward** hace que el repositorio siga una evolución lineal sin perder ningún commit (como con squash)
  - Pero si ha habido commits en master posteriores a la creación de la rama, no se puede hacer
  - Existe una forma de transformar la rama para que “empiece” en el último commit de máster > el “**rebase**”

# Ramas (*branches*)

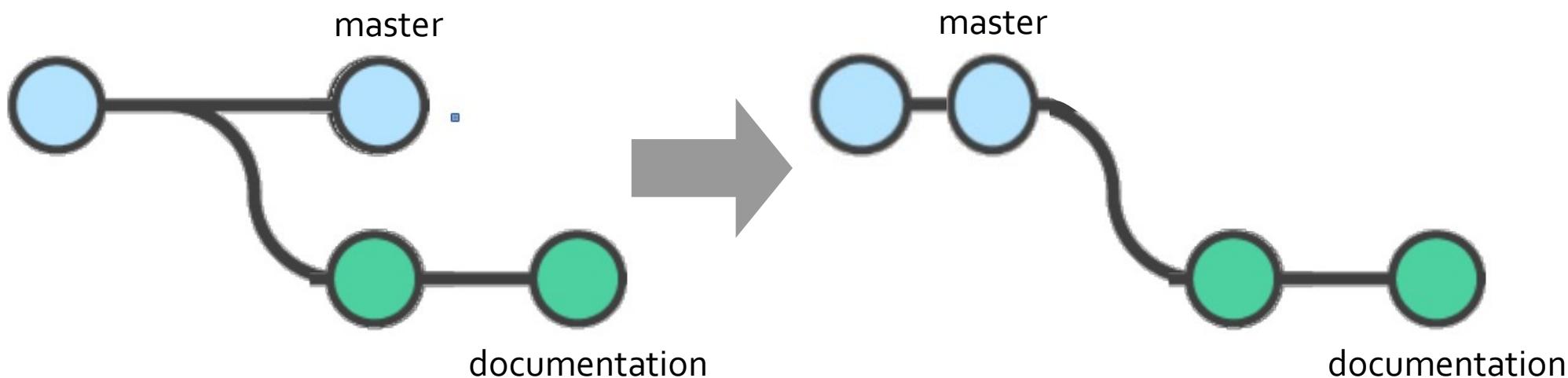
- **Fast-forward merge con rebase**
  - ¿Qué ocurre si se han hecho commits en master después de haber creado la rama que se quiere mezclar?
  - No se permite el fast-forward merge



# Ramas (*branches*)

- **Fast-forward merge con rebase**

- El rebase desplaza la nueva rama para que se base en el último commit de la rama master
- De esa forma se puede aplicar el fast-forward



# Ramas (*branches*)

- **Fast-forward merge con rebase**
  - Vamos a crear la rama newdoc con un commit en README.md
  - Vamos a volver a master y hacer un commit en README2.md
  - Vamos a intentar hacer un merge forzando fast-forward de newdoc a master para que nos detecte que no se puede
  - Rebasamos newdoc sobre master
  - Volvemos a intentar el merge con fast-forward

# Ramas (*branches*)

- Fast-forward merge con rebase

```
~/repo$ git checkout -b newdoc
Switched to a new branch 'newdoc'
~/repo$ echo "New content" >> README.md
~/repo$ git add README.md
~/repo$ git commit -m "Add new line in README.md"
[newdoc 4b75181] Add new line in README.md
 1 file changed, 3 insertions(+), 1 deletion(-)
~/repo$ git checkout master
Switched to branch 'master'
~/repo$ echo "New content" >> README2.md
~/repo$ git add README2.md
~/repo$ git commit -m "Add content to README2.md"
[master 1687778] Add content to README2.md
 1 file changed, 1 insertion(+)
```

# Ramas (*branches*)

- Fast-forward merge con rebase

```
~/repo$ git log --graph --all --oneline
* 1687778 (HEAD -> master) Add content to README2.md
| * 4b75181 (newdoc) Add new line in README.md
|/
* 566ec18 Update README.md
* 811983f Add README2.md
```

# Ramas (*branches*)

- **Fast-forward merge con rebase**
  - Intentamos el merge forzando fast-forward

```
~/repo$ git merge newdoc --ff-only  
fatal: Not possible to fast-forward, aborting.
```

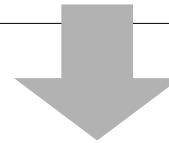
- Rebasamos newdoc sobre master

```
~/repo$ git checkout newdoc  
Switched to branch 'newdoc'  
~/repo$ git rebase master  
First, rewinding head to replay your work on top of it...  
Applying: Add new line in README.md
```

# Ramas (*branches*)

- Fast-forward merge con rebase
  - Ahora se cumplen las condiciones para el fast-forward merge

```
~/repo$ git log --graph --all --oneline
* 1687778 (HEAD -> master) Add content to README2.md
| * 4b75181 (newdoc) Add new line in README.md
|/
* 566ec18 Update README.md
* 811983f Add README2.md
```



```
~/repo$ git log --graph --all --oneline
* 531b278 (HEAD -> newdoc) Add new line in README.md
* 1687778 (master) Add content to README2.md
* 566ec18 Update README.md
* 811983f Add README2.md
```

# Ramas (*branches*)

- Fast-forward merge con rebase
  - Volvemos a intentar el merge

```
~/repo$ git merge newdoc --ff-only  
fatal: Not possible to fast-forward, aborting.
```



```
~/repo$ git merge newdoc --ff-only  
Updating 1687778..531b278  
Fast-forward  
 README.md | 4 +++-  
 1 file changed, 3 insertions(+), 1 deletion(-)
```

# Ramas (*branches*)

- Fast-forward merge con rebase
  - La historia es lineal

```
~/repo$ git log --graph --all --oneline
* 531b278 (HEAD -> master, newdoc) Add new line in README.md
* 1687778 Add content to README2.md
* 566ec18 Update README.md
* 811983f Add README2.md
```

# Ramas (*branches*)

- ¿Rebase o merge?
  - Hay debate al respecto
  - Rebasar una rama con mucho commits es complejo si se trabaja en un código con muchos conflictos
  - Pero los commits de merge, como no tienen trazabilidad con la rama original, pueden perder contexto
  - Pros y Cons de cada uno

<https://www.atlassian.com/blog/git/git-team-workflows-merge-or-rebase>

# Borrar commits

- Podemos **borrar los últimos commits** de la rama
- Al borrarlos posicionamos **HEAD** y la rama en un commit del pasado
- Todavía sería posible **recuperar esos commits** si estuvieran referenciados por otra rama

```
$ git reset
```

# Borrar commits

- Hay que tener cuidado de **no borrar más de la cuenta** y perder información
- Ante la duda, haz una **copia del repositorio “a mano”**
- Si has **subido al servidor los commits** que vas a borrar, es posible que otros compañeros se basen en ellos y es peligroso borrarlos



# Borrar commits

- **\$ git reset --hard <commit>**
  - Borra el contenido actual del area de trabajo y del area de staging (salvo los ficheros *untracked*)
  - Retrocede hasta el commit indicado
  - Saca el contenido del mismo al directorio de trabajo
  - No habrá posibilidad de recuperar los commits posteriores a menos que estén apuntados por otra rama o tag

# Borrar commits

- **\$ git reset --hard HEAD**
  - Borra el contenido actual del area de trabajo y del area de staging (salvo los ficheros *untracked*)
  - Restaura en el directorio de trabajo el contenido del último commit de la rama

# Borrar commits

- Ejercicio 5
  - Crea un fichero nuevo en master
  - Modifica el contenido del README.md
  - Ejecuta: **\$ git reset --hard HEAD**
  - Comprueba que el contenido del README.md se restaura al contenido original pero que el nuevo fichero (*untracked*) sigue intacto

# Borrar commits

- **\$ git reset --soft <commit>**
  - Deja el area de trabajo y el area de staging sin cambios
  - Retrocede hasta el commit que le digamos
  - Como el area de trabajo (y staging) se mantienen, no se pierde contenido de los ficheros si se hace commit
  - El histórico de commits si se pierde con el reset

# Borrar commits

- Ejercicio 6
  - Modifica el contenido de README.md y README2.md
  - Retrocede master dos commits anteriores
  - Revisa que los ficheros README.md y README2.md están intactos (no han perdido información)
  - Haz un nuevo commit

# Borrar commits

- **\$ git reset --mixed <commit>**
  - Deja el area de trabajo sin cambios
  - Borra el area de staging
  - Retrocede hasta el commit que le digamos
  - Como el area de trabajo se mantiene, no se pierde contenido de los ficheros si se hace commit
  - El histórico de commits si se pierde con el reset
  - Es el modo por defecto

# Borrar commits

- **Ejercicio 7**
  - Agrupar los dos últimos commits en un único commit

# Resolviendo conflictos

- En los ejemplos hemos visto que dos ramas se mezclaban de forma automática
- Cuando en dos ramas se editan **ficheros distintos** o el mismo fichero pero **diferentes partes**, no existen conflictos y la mezcla se hace de forma automática

Que no haya conflictos (y git no se queje) no implica que el código resultante de la mezcla **funcione sin problemas** (o incluso compile)



# Resolviendo conflictos

- Cuando en dos ramas se ha modificado **la misma línea del mismo fichero**, se produce un **conflicto al mezclar** y el merge (con commit, squash o rebase) no puede hacerse de forma automática
- Es necesario que el desarrollador decida cómo debe ser la línea de código final cuando en cada commit padre se ha modificado **la misma línea del mismo fichero**

# Resolviendo conflictos

- Crear y posicionarnos en una nueva rama fix-readme
  - Editar README.md
  - Añadir contenido en una línea ya existente (marcado en azul), hacer un commit

```
# Repo documentation

This project aims at providing a testing infrastructure for
distributed applications that are so complex that it's
difficult to write end-to-end tests for them.

New content.

## A new section

With its own content. And some typos fixed.
```

# Resolviendo conflictos

- Volver a master
  - Editar README.md
  - Añadir contenido en la misma línea (marcado en rojo), hacer un commit

```
# Repo documentation
```

```
This project aims at providing a testing infrastructure for  
distributed applications that are so complex that it's  
difficult to write end-to-end tests for them.
```

```
New content.
```

```
## A new section
```

```
This should conflict. With its own content.
```

# Resolviendo conflictos

- Volvemos a la rama fix-readme
- Editar README.md
  - Añadir una nueva sección al final, hacer un commit

```
# Repo documentation
```

```
This project aims at providing a testing infrastructure for  
distributed applications that are so complex that it's  
difficult to write end-to-end tests for them.
```

```
New content.
```

```
## A new section
```

```
With its own content. And some typos fixed.
```

```
## A third section
```

```
Where we explain very important topics.
```

# Resolviendo conflictos

- **Merge con conflictos**

- Queremos integrar los cambios de la rama fix-readme en master
- Vamos a seguir un merge fast-forward
- Tenemos que rebasar fix-readme sobre master (porque se han hecho commits en master desde que se creó la rama fix-readme)

# Resolviendo conflictos

- Rebasamos master...

```
~/repo$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: add "more info" to README.md
Using index info to reconstruct a base tree...
M      README.md
Falling back to patching base and 3-way merge...
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
error: Failed to merge in the changes.
Patch failed at 0001 add "more info" to README.md
Use 'git am --show-current-patch' to see the failed patch

Resolve all conflicts manually, mark them as resolved with
"git add/rm <conflicted_files>", then run "git rebase --continue".
You can instead skip this commit: run "git rebase --skip".
To abort and get back to the state before "git rebase", run "git rebase --abort".
```

Error: failed to merge

# Resolviendo conflictos

- Rebase mueve HEAD hasta el primer ancestro común que encuentra, entonces empieza a aplicar los commits de master, luego los de fix-readme
- El último commit de master cambia una línea de readme que también cambia fix-readme en su primer commit
- El rebase no puede mezclar ambos cambios, ¿qué línea deja/quita?
- El rebase para, y nos deja la opción de solucionar el problema
- Una vez solucionado, ejecutamos

```
$ git add <fichero>
$ git rebase --continue
```

# Resolviendo conflictos

- Veamos el fichero README.md
  - <<<<< HEAD: marca el inicio del contenido de HEAD (situado en ese momento en el último commit de master)
  - =====: separa ambas partes, lo que viene después es el contenido de fix-readme
  - >>>>> fix typos: marca el final del conflicto y el commit

```
## A new section

<<<<<< HEAD

This should conflict. With its own content.

=====

With its own content. And some typos fixed.

>>>>> Fix typos
```

# Resolviendo conflictos

- Visual Studio Code sabe que es un conflicto de git y nos ayuda a solucionarlo

[Accept Current Change](#) | [Accept Incoming Change](#) | [Accept Both Changes](#) | [Compare Changes](#)

<<<<< HEAD (Current Change)

(more info) New paragraph

=====

New paragraph (more info)

>>>>> add "more info" to README.md (Incoming Change)

# Resolviendo conflictos

- Mezclamos ambas líneas manualmente como queramos

```
## A new section
```

```
This should conflict. With its own content. And some typos fixed.
```

- Añadimos el fichero con git add
  - Esto le indica a git que ya lo hemos solucionado
  - Continuamos el rebase con git rebase --continue

# Resolviendo conflictos

- **Ejercicio 8**

- Crea una nueva rama, modifica un fichero y comitea
- Vuelve a master, modifica el mismo fichero en la misma línea y comitea
- Haz merge resolviendo los conflictos

# Desarrollo colaborativo con Git

- Introducción: Git y GitHub
- Git: trabajando en local
- **Git y GitHub: trabajando en equipo**
- Gestionando el repositorio
- Modelos de desarrollo

# Git y GitHub

- La forma recomendada de trabajar con repositorios remotos en git es usar una **clave SSH**
- Técnicamente se deben generar **dos claves**, una **pública** y una **privada**
- La **pública** se configura en el **servidor git** y la **privada** queda siempre en la **máquina del usuario**
- Permite una forma de **seguridad avanzada** porque la clave privada nunca se transmite

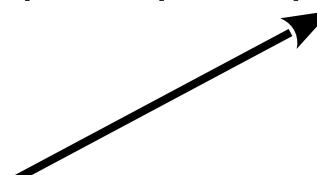
<https://help.github.com/enterprise/2.15/user/articles/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent/>

# Git y GitHub

- Generación en linux

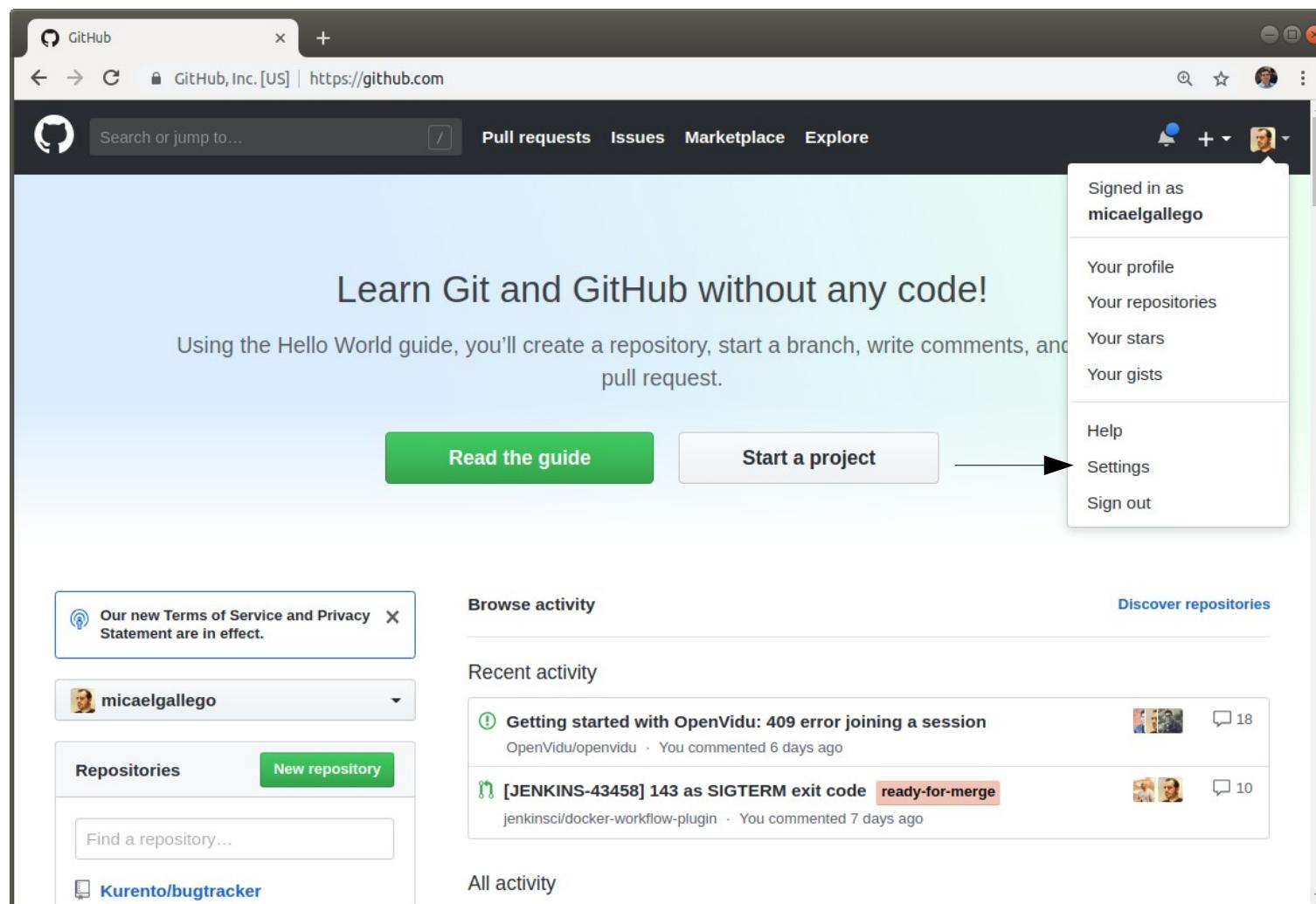
```
$ ssh-keygen -t rsa -b 4096 -C "micael.gallego@gmail.com"
Generating public/private rsa key pair.
Enter file in which to save the key (/home/mica/.ssh/id_rsa):
Created directory '/c/Users/patxi/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/mica/.ssh/id_rsa.
Your public key has been saved in /home/mica/.ssh/id_rsa.pub.
```

Tenemos que añadir la clave pública a GitHub



# Git y GitHub

- Añadimos la clave pública a GitHub



# Git y GitHub

- Añadimos la clave pública a GitHub

The screenshot shows the GitHub 'Personal settings' page with the 'SSH and GPG keys' section highlighted by a red arrow. On the right, there is a green 'New SSH key' button with a red arrow pointing to it. Below the 'SSH keys' heading, a key icon is shown next to the text 'gortazar laptop Ubuntu 14.10'. Underneath this, it says 'Added on Dec 6, 2014' and 'Last used within the last day'. To the right of this entry is a 'Delete' button. At the bottom of the 'SSH keys' section, there is a link to a guide: 'Check out our guide to [generating SSH keys](#) or troubleshoot [common SSH Problems](#)'. The 'GPG keys' section below it indicates 'There are no GPG keys with access to your account.' and provides a link to 'Learn how to [generate a GPG key](#) and add it to your account.'

# Git y GitHub

- Añadimos la clave pública a GitHub

```
$ cat /home/mica/.ssh/id_rsa.pub
```

The screenshot shows the GitHub user interface for managing SSH keys. The top navigation bar includes the GitHub logo, a search bar, and links for Pull requests, Issues, and Gist. The user's profile picture and account dropdown are also visible.

The left sidebar contains a navigation menu with options: Personal settings, Profile, Account, Emails, Notifications, Billing, **SSH and GPG keys** (which is currently selected), Security, Blocked users, Repositories, Organizations, Saved replies, Authorized OAuth Apps, Authorized Integrations, Installed integrations, and Developer settings.

The main content area is titled "SSH keys". It displays a list of existing keys and a form for adding a new key.

**Existing Keys:**

- A key entry for "gortazar laptop Ubuntu 14.10" was added on Dec 6, 2014, and was last used within the last day. It includes a "Delete" button.

**New Key Form:**

- Title:** git-bash|key
- Key:** ssh-rsa [REDACTED]
- Add SSH key** button

# Crear un repositorio en GitHub

- Hay dos formas de comenzar a trabajar en GitHub
  - **Creamos un repositorio vacío**
    - Esto nos permite subir nuestros cambios si ya tenemos contenido
    - Es un poco más engorroso, pero GitHub nos indica cómo hacerlo
  - **Crear un repositorio con un commit inicial**
    - Normalmente se incluye un README.md y el fichero de licencia (LICENSE)
    - Este repositorio se puede clonar directamente
    - Al clonar el repositorio, se crea la copia local y se configura para que esté conectado al repositorio remoto

# Crear un repositorio en GitHub

- Crear un repositorio con un commit inicial

The screenshot shows a GitHub user profile for 'gortazar'. The main area displays three recent commits from 'j1elo' to 'Kurento/kms-jsonrpc', 'Kurento/kurento-media-server', and 'Kurento/kms-filters'. A red arrow points to a context menu that has appeared over the top right corner of the interface. The menu options are: 'New repository' (highlighted), 'Import repository', 'New gist', and 'New organization'. Below the menu, a section titled 'Repositories you contribute to' lists several repositories with their names, descriptions, and star counts.

New repository

Import repository

New gist

New organization

Repositories you contribute to (54)

- nubomedia/nubomedia-media-s... 0 ★
- nubomedia/nubomedia-sdks 1 ★
- Kurento/adm-scripts 0 ★
- jromeroRatedP... /pvdesign 0 ★
- nubomedia/nubomedia-docume... 0 ★

Show 49 more repositories...

# Crear un repositorio en GitHub

- Crear un repositorio con un commit inicial

The screenshot shows the GitHub interface for creating a new repository. At the top, there's a navigation bar with the GitHub logo, a search bar, and links for 'Pull requests', 'Issues', and 'Gist'. On the right side of the header, there are icons for notifications, a plus sign, and a user profile.

The main section is titled 'Create a new repository'. It explains that a repository contains all the files for a project, including the revision history. Below this, there are fields for 'Owner' (set to 'gortazar') and 'Repository name' ('newrepo'). A green checkmark icon is next to the repository name field.

A note says, 'Great repository names are short and memorable. Need inspiration? How about [crispy-disco](#).'

The 'Description (optional)' field is empty.

Below that, there are two radio button options: 'Public' (selected) and 'Private'. The 'Public' option is described as 'Anyone can see this repository. You choose who can commit.' The 'Private' option is described as 'You choose who can see and commit to this repository.'

A large red arrow points to the 'Initialize this repository with a README' checkbox, which is checked. A explanatory text below it says, 'This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.'

At the bottom, there are buttons for 'Add .gitignore: None' and 'Add a license: Apache License 2.0'. There's also a small info icon (i).

Finally, a large green 'Create repository' button is at the bottom center.

# Trabajo con un repositorio remoto

- Clonar el repositorio

The screenshot shows a GitHub repository page for 'gortazar / newrepo'. The top navigation bar includes 'This repository', 'Search', 'Pull requests', 'Issues', and 'Gist'. Below the header, there's a section for 'Code' with links to 'Issues (0)', 'Pull requests (0)', 'Projects (0)', 'Wiki', 'Pulse', 'Graphs', and 'Settings'. A note says 'No description, website, or topics provided.' with an 'Edit' button. Below this, stats show '1 commit', '1 branch', '0 releases', and '1 contributor'. A red arrow points from the 'Clone or download' button to the 'Clone with SSH' field, which contains the URL 'git@gitHub.com:gortazar/newrepo.git'. Another red arrow points to the 'Initial commit' entry in the commit list, which has a timestamp of 'Initial commit'.

gortazar / newrepo

This repository Search

Pull requests Issues Gist

Unwatch 1 Star 0 Fork 0

Code Issues 0 Pull requests 0 Projects 0 Wiki Pulse Graphs Settings

No description, website, or topics provided. Edit

Add topics

1 commit 1 branch 0 releases 1 contributor Apache-2.0

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

gortazar Initial commit

LICENSE README.md

Initial commit

Initial commit

Clone with SSH Use HTTPS

git@gitHub.com:gortazar/newrepo.git

Download ZIP

README.md

newrepo

# Trabajo con un repositorio remoto

- Clonar el repositorio

```
$ git clone git@github.com:gortazar/newrepo.git
Cloning into 'newrepo'...
The authenticity of host 'github.com (192.30.253.112)' can't be established.
RSA key fingerprint is SHA256:nThbg6kXUpJWGl7E1IG0CspRomTxdCARLviKw6E5SY8.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'github.com,192.30.253.112' (RSA) to the list of
known hosts.
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (4/4), 4.13 KiB | 0 bytes/s, done.
```

# Trabajo con un repositorio remoto

- El repositorio local tiene automáticamente configurado el repositorio de GitHub como un repositorio remoto
  - Lo habitual es tener un único repositorio remoto (el del servidor)
  - Pero podríamos tener varios repositorios remotos para casos de uso avanzados (p.e. sincronizar repositorios)
  - Cada uno tiene un nombre, el repositorio de GitHub se llama **origin**
- Dentro de cada repositorio remoto, podemos configurar una o varias ramas

# Trabajo con un repositorio remoto

```
$ git remote show origin
* remote origin
  Fetch URL: git@github.com:gortazar/newrepo.git
  Push  URL: git@github.com:gortazar/newrepo.git
  HEAD branch: master
  Remote branch:
    master tracked
  Local branch configured for 'git pull':
    master merges with remote master
  Local ref configured for 'git push':
    master pushes to master (up to date)
```

Ahora mismo la rama master está sincronizada con la rama master del remoto (up to date)

# Trabajo con un repositorio remoto

- Vamos a hacer un cambio

```
$ echo "Some update" >> README.md
```

```
$ git status
```

```
On branch master
```

```
Your branch is up-to-date with 'origin/master'.
```

```
Changes not staged for commit:
```

```
  (use "git add <file>..." to update what will be committed)
```

```
  (use "git checkout -- <file>..." to discard changes in working directory)
```

```
modified:   README.md
```

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

git status ahora nos informa de si estamos sincronizados con la versión remota de esta rama



# Trabajo con un repositorio remoto

- Comiteamos

```
$ git add README.md
```

```
$ git commit -m "Updated README"
```

```
[master 444bf6a] Updated README
 1 file changed, 1 insertion(+), 1 deletion(-)
```

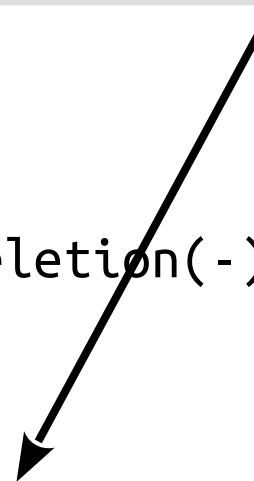
```
$ git status
```

```
On branch master
```

```
Your branch is ahead of 'origin/master' by 1 commit.
(use "git push" to publish your local commits)
```

```
nothing to commit, working tree clean
```

git status nos indica que el repositorio local está 1 commit más avanzado que el remoto



# Trabajo con un repositorio remoto

- Subamos los cambios al repositorio remoto

```
$ git push origin master ←  
Counting objects: 3, done.  
Delta compression using up to 2 threads.  
Compressing objects: 100% (2/2), done.  
Writing objects: 100% (3/3), 305 bytes | 0 bytes/s,  
done.  
Total 3 (delta 0), reused 0 (delta 0)  
To github.com:gortazar/newrepo.git  
 a53fc64..444bf6a master -> master
```

El repositorio remoto al que queremos subir los cambios y la rama dentro del repositorio

```
$ git status  
On branch master  
Your branch is up to date with 'origin/master'.  
  
nothing to commit, working tree clean
```

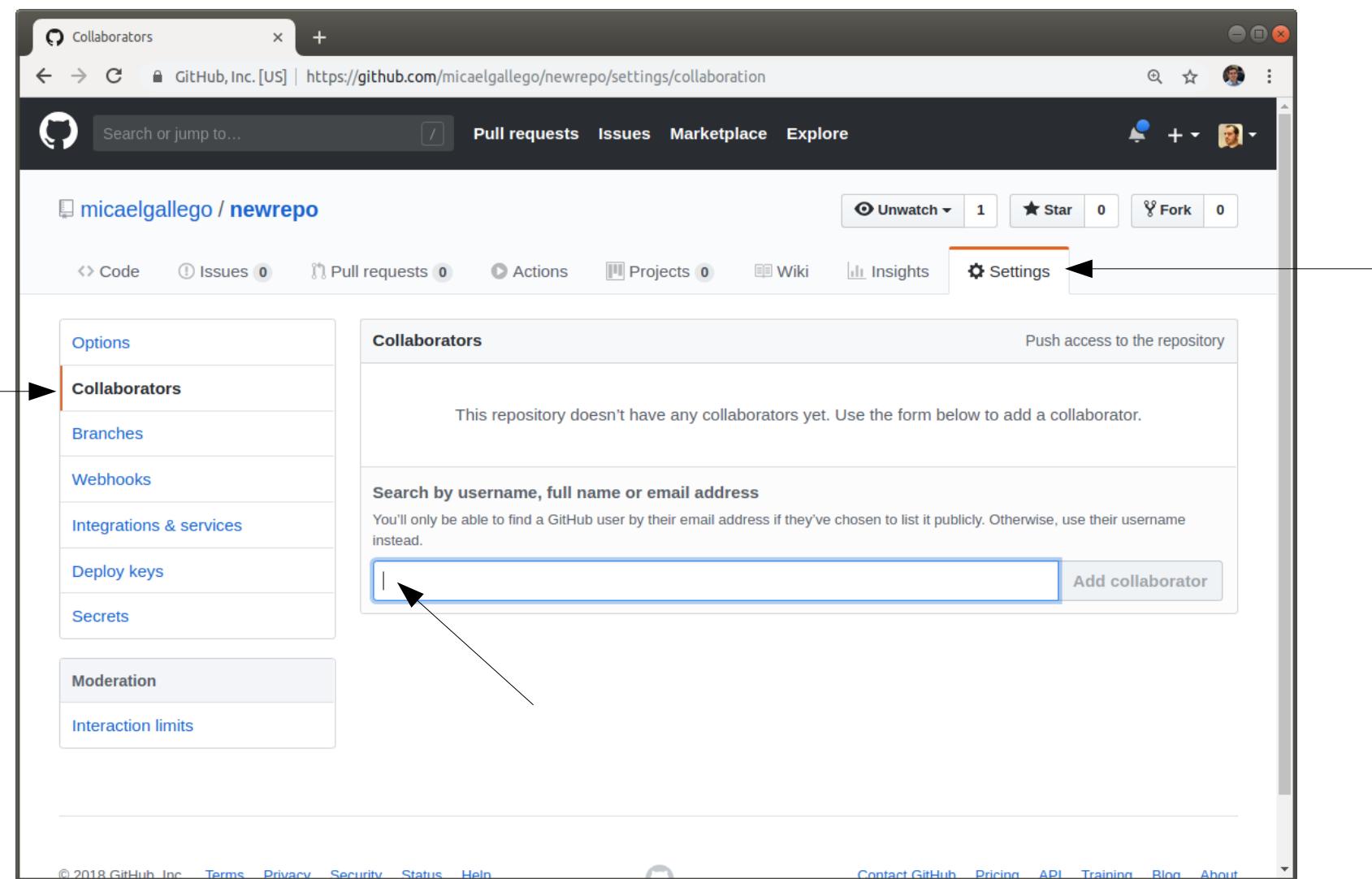
# Trabajo colaborativo

- Normalmente se utiliza **GitHub** como repositorio centralizado de confianza
- Cada desarrollador **tendrá una copia** de este repositorio en local
- Los cambios, además de comitearlos a nuestro repositorio local tenemos que “empujarlos” (**push**) al repositorio remoto
- Periódicamente tendremos que traernos los cambios de otros desarrolladores “tirando” (**pull/fetch**) de ellos

# Trabajo colaborativo

- Podemos dar permisos a otros usuarios a nuestro repositorio GitHub
- Esos usuarios se pueden clonar el repositorio y hacer commits al mismo
- Para colaborar, podemos descargar los commits de otros usuarios al repositorio

# Trabajo colaborativo



# Trabajo colaborativo

- **Simulación de dos usuarios**
  - Vamos a ver cómo descargar commits del repositorio duplicando el repositorio local en otra carpeta (newrepo2)
  - Así podemos crear un commit en un repositorio y descargarlo en otro

# Trabajo colaborativo

- Creamos README2.md en newrepo2

```
~/newrepo2$ echo "Other README" >> "README2.md"
~/newrepo2$ git add .
~/newrepo2$ git commit -m "Add README2 file"
[master 8b7fb4d] Add README2 file
 1 file changed, 1 insertion(+)
 create mode 100644 README2.md

$ git push origin master
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 288 bytes | 288.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To github.com:micaelgallego/newrepo.git
 dc3d426..8b7fb4d  master -> master
```

# Trabajo colaborativo

- Volvemos a newrepo y descargamos el repo remoto
  - **git fetch** descarga el servidor remoto, pero no integra de ninguna forma su contenido en la rama master que tenemos en local

```
~/newrepo$ git fetch
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From github.com:micaelgallego/newrepo
dc3d426..8b7fb4d  master      -> origin/master
```

# Trabajo colaborativo

- Vemos en qué estado se encuentra el repo local

```
~/newrepo$ git status
On branch master
Your branch is behind 'origin/master' by 1 commit, and can be fast-
forwarded.
  (use "git pull" to update your local branch)

nothing to commit, working tree clean
```

- Estamos 1 commit por detrás de origin/master
- El master local se puede sincronizar con origin/master con **fast-forward**, lo que quiere decir que se puede integrar simplemente integrando los commits y haciendo que master apunte al último commit

# Trabajo colaborativo

- **git merge** sirve para integrar los cambios de remoto a local

```
~/newrepo$ git merge
Updating dc3d426..8b7fb4d
Fast-forward
 README2.md | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 README2.md

~/newrepo$ git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

- **git pull** es equivalente a un **git fetch** seguido de **git merge**<sup>142</sup>

# Trabajo colaborativo

- ¿Qué ocurre si dos usuarios hacen commits a master?
  - El primero que los suba lo hace sin problemas

```
~/newrepo$ echo "More content" >> README.md
~/newrepo$ git add .
~/newrepo$ git commit -m "Add more content in README"
[master 87fa529] Add more content in README
 1 file changed, 1 insertion(+)

~/newrepo$ git push
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 318 bytes | 318.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To github.com:micaelgallego/newrepo.git
 8b7fb4d..87fa529  master -> master
```

# Trabajo colaborativo

- ¿Qué ocurre si dos usuarios hacen commits a master?
  - El segundo tiene problemas al subir

```
~/newrepo2$ echo "More content" >> README2.md
~/newrepo2$ git add .
~/newrepo2$ git commit -m "Add more content in README2"
[master 09f7403] Add more content in README2
 1 file changed, 1 insertion(+)
```

Nos dice los problemas que tenemos y cómo solucionarlo

```
~/newrepo2$ git push
To github.com:micaelgallego/newrepo.git
 ! [rejected]      master -> master (fetch first)
error: failed to push some refs to 'git@github.com:micaelgallego/newrepo.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

# Trabajo colaborativo

- ¿Qué ocurre si dos usuarios hacen commits a master?
  - El comando **git push** intenta hacer una mezcla de las ramas con **fast-forward**
  - Si no puede hacerlo, da **error** y recomienda primero integrar los cambios remotos en la rama local antes de hacer el push
  - Esa integración se puede hacer:
    - Rebasando los commits locales sobre los commits remotos (**git rebase**)
    - Creando un nuevo commit de mezcla (merge) que integra los cambios locales y los remotos (**git merge**)
  - Cuando la integración se ha hecho, se puede hacer push

# Trabajo colaborativo

- ¿Qué ocurre si dos usuarios hacen commits a master?
  - Hacemos un **git fetch** para descargar el repositorio remoto

```
~/newrepo2$ git fetch
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From github.com:micaelgallego/newrepo
  8b7fb4d..87fa529  master      -> origin/master
```

```
~/newrepo2$ git status
On branch master
Your branch and 'origin/master' have diverged,
and have 1 and 1 different commits each, respectively.
(use "git pull" to merge the remote branch into yours)
```

nothing to commit, working tree clean

# Trabajo colaborativo

- ¿Qué ocurre si dos usuarios hacen commits a master?
  - Vemos en el histórico que efectivamente la rama master ha divergido en local y en remoto

```
~/newrepo2$ git log --graph --all --oneline
* 09f7403 (HEAD -> master) Add more content in README2
| * 87fa529 (origin/master, origin/HEAD) Add more content in README
|/
* 8b7fb4d Add README2 file
* dc3d426 Update README
* 9c431a9 Initial commit
```

# Trabajo colaborativo

- ¿Qué ocurre si dos usuarios hacen commits a master?
  - Hacemos un **git merge** (o **git pull**) para mezclar ambas ramas

```
~/newrepo2$ git merge
Merge made by the 'recursive' strategy.
 README.md | 1 +
 1 file changed, 1 insertion(+)

~/newrepo2$ git log --graph --all --oneline
 *   da4c96b (HEAD -> master) Merge remote-tracking branch
 'refs/remotes/origin/master'
 |\ 
 | * 87fa529 (origin/master, origin/HEAD) Add more content in README
 * | 09f7403 Add more content in README2
 |/
 * 8b7fb4d Add README2 file
 * dc3d426 Update README
 * 9c431a9 Initial commit
```

# Trabajo colaborativo

- ¿Qué ocurre si dos usuarios hacen commits a master?
  - Ahora ya podemos subir nuestro cambio al servidor (y además también subimos el commit de merge)

```
~/newrepo2$ git push
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (5/5), 601 bytes | 300.00 KiB/s, done.
Total 5 (delta 0), reused 0 (delta 0)
To github.com:micaelgallego/newrepo.git
  87fa529..da4c96b  master -> master

~/newrepo2$ git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

# Trabajo colaborativo

- ¿Qué ocurre si dos usuarios hacen commits a master?
  - Podemos hacer lo mismo rebasando los cambios locales sobre los cambios remotos
    - Vamos a hacer que el usuario en newrepo haga un commit nuevo
    - Intentará hacer push y obtenga un error
    - Se rebasará sobre la rama remota
    - Finalmente podrá hacer el push sin problemas

# Trabajo colaborativo

- ¿Qué ocurre si dos usuarios hacen commits a master?
  - Nuevo commit en newrepo e intento de push

```
~/newrepo$ echo "More Content" >> README.md
~/newrepo$ git add .
~/newrepo$ git commit -m "Add more content (again) in README"
[master 4b2540f] Add more content (again) in README
 1 file changed, 1 insertion(+)

~/newrepo$ git push
To github.com:micaelgallego/newrepo.git
 ! [rejected]      master -> master (fetch first)
error: failed to push some refs to
'git@github.com:micaelgallego/newrepo.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

# Trabajo colaborativo

- ¿Qué ocurre si dos usuarios hacen commits a master?
  - Descarga

```
~/newrepo$ git fetch
remote: Enumerating objects: 9, done.
remote: Counting objects: 100% (8/8), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 5 (delta 0), reused 5 (delta 0), pack-reused 0
Unpacking objects: 100% (5/5), done.
From github.com:micaelgallego/newrepo
  87fa529..da4c96b  master      -> origin/master

~/newrepo$ git status
On branch master
Your branch and 'origin/master' have diverged,
and have 1 and 2 different commits each, respectively.
(use "git pull" to merge the remote branch into yours)

nothing to commit, working tree clean
```

# Trabajo colaborativo

- ¿Qué ocurre si dos usuarios hacen commits a master?
  - Rebase (**git rebase** o **git pull --rebase**)

```
~/newrepo$ git rebase
```

First, rewinding head to replay your work on top of it...

Applying: Add more content (again) in README

```
~/newrepo$ git log --graph --all --oneline
```

```
* 7b86c28 (HEAD -> master) Add more content (again) in README
*   da4c96b (origin/master, origin/HEAD) Merge remote-tracking branch
  'refs/remotes/origin/master'
  |
  | * 87fa529 Add more content in README
  * | 09f7403 Add more content in README2
  |
  * 8b7fb4d Add README2 file
  * dc3d426 Update README
  * 9c431a9 Initial commit
```

# Trabajo colaborativo

- ¿Qué ocurre si dos usuarios hacen commits a master?
  - Push

```
~/newrepo$ git push
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 335 bytes | 335.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To github.com:micaelgallego/newrepo.git
 da4c96b..7b86c28  master -> master
```

# Trabajo colaborativo

- ¿Qué ocurre si dos usuarios hacen commits a master?
  - Push

```
~/newrepo$ git log --graph --all --oneline
* 7b86c28 (HEAD -> master, origin/master, origin/HEAD) Add
more content (again) in README
*   da4c96b Merge remote-tracking branch
'refs/remotes/origin/master'
|\ 
| * 87fa529 Add more content in README
* | 09f7403 Add more content in README2
|/
* 8b7fb4d Add README2 file
* dc3d426 Update README
* 9c431a9 Initial commit
```

# Trabajo colaborativo - Resumen

- Cada usuario añade commits a su rama master local
- Cuando se van a subir los cambios al repositorio
  - El primero puede subir sin problemas (**git push**)
  - El segundo tiene que mezclar en local y luego subir
    - **Merge:** crea commit de merge entre los dos commits

**git pull**

**git fetch + git merge**

- **Rebase:** Rebasa sus cambios locales encima de los cambios remotos

**git pull --rebase**

**git fetch + git rebase**

# Pull Requests de GitHub

- Cuando se trabaja sobre **master**, los miembros del equipo sólo pueden **conocer** lo que hace un compañero cuando ya **está integrado**
- Sería interesante que otros miembros del equipo pudieran **revisar** los cambios **antes** de que finalmente se **integren** y para, si es necesario, solicitar **mejoras**
- Es especialmente útil con:
  - **Desarrolladores con poca experiencia**
  - **Proyectos de software libre** en los que desarrolladores ajenos al proyecto hacen contribuciones al mismo

# Pull Requests de GitHub

- Los servidores de git han incorporado un mecanismo para que los **desarrolladores puedan solicitar revisión** antes de que los cambios se integren en master
  - Pull Requests (GitHub y Bitbucket)
  - Merge Request (GitLab)
  - Change (Gerrit)



# Pull Requests de GitHub

- **Desarrollo basado en Pull Request de GitHub**
  - Convertimos la rama en un Pull Request: una petición de integrar ciertos cambios en la rama master
  - Asociado al Pull Request (PR) tenemos un hilo de comentarios
  - Otros desarrolladores/as pueden darnos feedback
  - Es posible que tengamos que actualizar/cambiar cosas en nuestra rama
  - Automáticamente se reflejarán en el PR cuando hagamos push de la rama de nuevo

# Pull Requests de GitHub

## • Funcionamiento

- El desarrollador crea una rama en local (generalmente asociada a una funcionalidad, *branch-per-feature*)
- Hace cambios en la rama (commits) y sube la rama a GitHub
- Desde la interfaz web de GitHub (o un plugin o CLI), solicita integrar la rama en master (se crea el Pull Request)
- El Pull Request tiene un hilo de comentarios asociado para dar feedback sobre sus commits
- Se pueden hacer más commits sobre la rama en respuesta al feedback
- Cuando el código está listo, un desarrollador con permisos “Acepta el Pull Request” y su contenido se integra en master (con merge o fast-forward)

# Pull Requests de GitHub

- Implementamos una nueva funcionalidad
  - Creamos una rama feature-12
  - Hacemos commits en ella

```
~/newrepo$ git checkout -b feature-12
Switched to a new branch 'feature-12'
~/newrepo$ echo "code" >> code.src
~/newrepo$ git add .
~/newrepo$ git commit -m "Create first code file"
[feature-12 3f03103] Create first code file
 1 file changed, 1 insertion(+)
 create mode 100644 code.src
```

# Pull Requests de GitHub

- Subimos la rama a GitHub

```
~/newrepo$ git push origin feature-12
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 313 bytes | 313.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
remote:
remote: Create a pull request for 'feature-12' on GitHub by visiting:
remote:     https://github.com/micaelgallego/newrepo/pull/new/feature-12
remote:
To github.com:micaelgallego/newrepo.git
 * [new branch]      feature-12 -> feature-12
```

# Pull Requests de GitHub

- Cuando consideramos que está terminada, la convertimos en Pull Request

A screenshot of a GitHub repository page for 'micaelgallego / newrepo'. The page shows basic repository statistics: 7 commits, 2 branches, 0 releases, and 1 contributor. A prominent green button labeled 'Compare & pull request' is highlighted with a yellow box and a black arrow pointing to it from the right side of the image. Below the stats, a section titled 'Your recently pushed branches:' lists 'feature-12' (pushed 5 minutes ago). At the bottom, there's a commit history with two entries:

- micaelgallego Add more content (again) in README
- README.md Add more content (again) in README 5 hours ago
- README2.md Add more content in README2 6 hours ago

At the very bottom right of the screenshot, the number '163' is visible.

# Pull Requests de GitHub

- Podemos crear un título y una descripción

The screenshot shows the GitHub interface for creating a pull request. At the top, it says "Comparing master...feature-12". Below that, there are dropdown menus for "base: master" and "compare: feature-12", with a note that they are "Able to merge". A large text input field is labeled "Create first code file". Below it, tabs for "Write" and "Preview" are visible, along with a rich text editor toolbar. A large text area for the pull request description is present, with the placeholder "This is the description of Pull Request. Other developers can see this description". Below this is a file attachment area with the instruction "Attach files by dragging & dropping, selecting them, or pasting from the clipboard". At the bottom right is a green button labeled "Create pull request". To the right of the main form, there are optional settings sections for "Reviewers" (no reviews), "Assignees" (no one assigned), "Labels" (none yet), "Projects" (none yet), and "Milestone" (no milestone).

Comparing master...feature-12

base: master ▾ compare: feature-12 ✓ Able to merge. These branches can be automatically merged.

Create first code file

Write Preview

This is the description of Pull Request. Other developers can see this description

Attach files by dragging & dropping, selecting them, or pasting from the clipboard.

Styling with Markdown is supported

Create pull request

Reviewers

No reviews

Assignees

No one—assign yourself

Labels

None yet

Projects

None yet

Milestone

No milestone

# Pull Requests de GitHub

- Subimos otro commit a la rama

```
mica@mica-PR0214:~/newrepo$ echo "More code" >> code.src
mica@mica-PR0214:~/newrepo$ git add .
mica@mica-PR0214:~/newrepo$ git commit -m "More code"
[feature-12 e74fece] More code
 1 file changed, 1 insertion(+)

mica@mica-PR0214:~/newrepo$ git push origin feature-12
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 261 bytes | 261.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To github.com:micaelgallego/newrepo.git
 3f03103..e74fece  feature-12 -> feature-12
```

# Pull Requests de GitHub

The screenshot shows a GitHub pull request interface. At the top, it says "Create first code file by micaelg" and "GitHub, Inc. [US] | https://github.com/micaelgallego/newrepo/pull/1". The title is "Create first code file #1". A green "Open" button indicates the pull request is open. It shows "micaelgallego wants to merge 2 commits into master from feature-12". Below this, there are tabs for "Conversation 0", "Commits 2", "Checks 0", and "Files changed 1".  
  
The main content area shows a comment from "micaelgallego" (Owner) 5 minutes ago: "This is the description of Pull Request. Other developers can see this description".  
  
A commit from "micaelgallego" 13 minutes ago adds two files: "Create first code file" (commit hash 3f03103) and "More code" (commit hash e74fce).  
  
A note at the bottom says "Add more commits by pushing to the feature-12 branch on micaelgallego/newrepo."  
  
A green sidebar on the right contains the following information:

- Reviewers**: No reviews
- Assignees**: No one—assign yourself
- Labels**: None yet
- Projects**: None yet
- Milestone**: No milestone
- Notifications**: Unsubscribe (button)

At the bottom of the main content area, there is a green "Merge pull request" button and a link to "command line instructions".

# Pull Requests de GitHub

- Otro usuario puede integrar los cambios en master
  - Usando la web:
    - Creando commit de merge
    - Fusionando todos los commits (squash)
    - Rebasando (si no hay conflictos) y mezclando
  - Descargando la rama remota y usando comandos git

```
$ git fetch origin
$ git checkout -b feature-12 origin/feature-12
$ git merge master

$ git checkout master
$ git merge --no-ff feature-12
$ git push origin master
```

# Pull Requests de GitHub

The screenshot shows a GitHub pull request page for a repository named 'Create first code file by micaelg'. The pull request has been merged 2 commits from the 'feature-12' branch into the 'master' branch 36 seconds ago. The description of the pull request is: "This is the description of Pull Request. Other developers can see this description". The commit history shows two commits: "Create first code file" (commit 3f03103) and "More code" (commit e74fce). The pull request was merged by micaelgallego 26 seconds ago. A message at the bottom indicates that the pull request has been successfully merged and closed, and the 'feature-12' branch can be safely deleted. The right sidebar contains settings for reviewers, assignees, labels, projects, milestones, and notifications, with an 'Unsubscribe' button.

Create first code file by micaelg | GitHub, Inc. [US] | https://github.com/micaelgallego/newrepo/pull/1

## Create first code file #1

**Merged** micaelgallego merged 2 commits into master from feature-12 36 seconds ago

Conversation 0 Commits 2 Checks 0 Files changed 1 +2 -0

micaelgallego commented 16 minutes ago

This is the description of Pull Request. Other developers can see this description

micaelgallego added some commits 23 minutes ago

- Create first code file 3f03103
- More code e74fce

micaelgallego merged commit e74fce into master 26 seconds ago

**Pull request successfully merged and closed**

You're all set—the feature-12 branch can be safely deleted.

Delete branch

Write Preview

Leave a comment

Reviewers

No reviews

Assignees

No one—assign yourself

Labels

None yet

Projects

None yet

Milestone

No milestone

Notifications

Unsubscribe

You're receiving notifications because you modified the open/close state.

# Pull Requests de GitHub

- Para que los cambios en el servidor se reflejen en local, es necesario hacer un pull (fetch y merge o rebase)

# Pull Requests de GitHub

```
$ git pull
```

```
Updating 444bf6a..0412ad4
```

```
Fast-forward
```

```
code.src | 1 +
```

```
1 file changed, 1 insertion(+)
```

```
create mode 100644 code/src
```

```
$ git status
```

```
On branch master
```

```
Your branch is up-to-date with 'origin/master'.
```

```
nothing to commit, working tree clean
```

# Pull Requests de GitHub

- Ejercicio 9
  - Implementa una nueva feature en una nueva rama
  - Acepta esa rama en master con un commit que fusioné los cambios

# Pull Requests de GitHub

- **Recordatorio sobre git pull**

- git pull intentará hacer un fast-forward
- Si eso no es posible, intentará hacer un merge
  - Aparece un commit resultado del merge
  - Este commit habría que subirlo a la rama remota en algún momento
- Si el merge tiene conflictos, nos dejará los conflictos en el working dir para que los arreglemos y hagamos nosotros el commit
- Si se usa git pull --rebase en vez de un merge se intentará hacer un rebase de los commits locales sobre los remotos, parando si hay conflictos

# Pull Requests de GitHub

- Podemos hacer rebases a una rama remota con referencias del tipo <remoto>/<rama>:
  - git rebase origin/master
    - Busca el primer ancestro común entre origin/master y nuestra rama
    - Aplica los cambios de origin/master
    - Aplica nuestros cambios encima

# Pull Requests de GitHub

- **Ejercicio 10**

- Por parejas (Alice y Bob)
- Crear un repositorio con README.md en una de las cuentas
- Dar permiso al compañero/a
- Clonar el repositorio ambos
- Alice arrancará una rama donde modificará el README.md
- Bob hará lo propio en otra rama
- Ambos comitearán, subirán la rama y abrirán un PR
- Uno de los dos aceptará el PR primero
- El otro no podrá
- Debe conseguir subir sus cambios a master a través del PR

# Pull Requests de GitHub

- **Ejercicio 11**

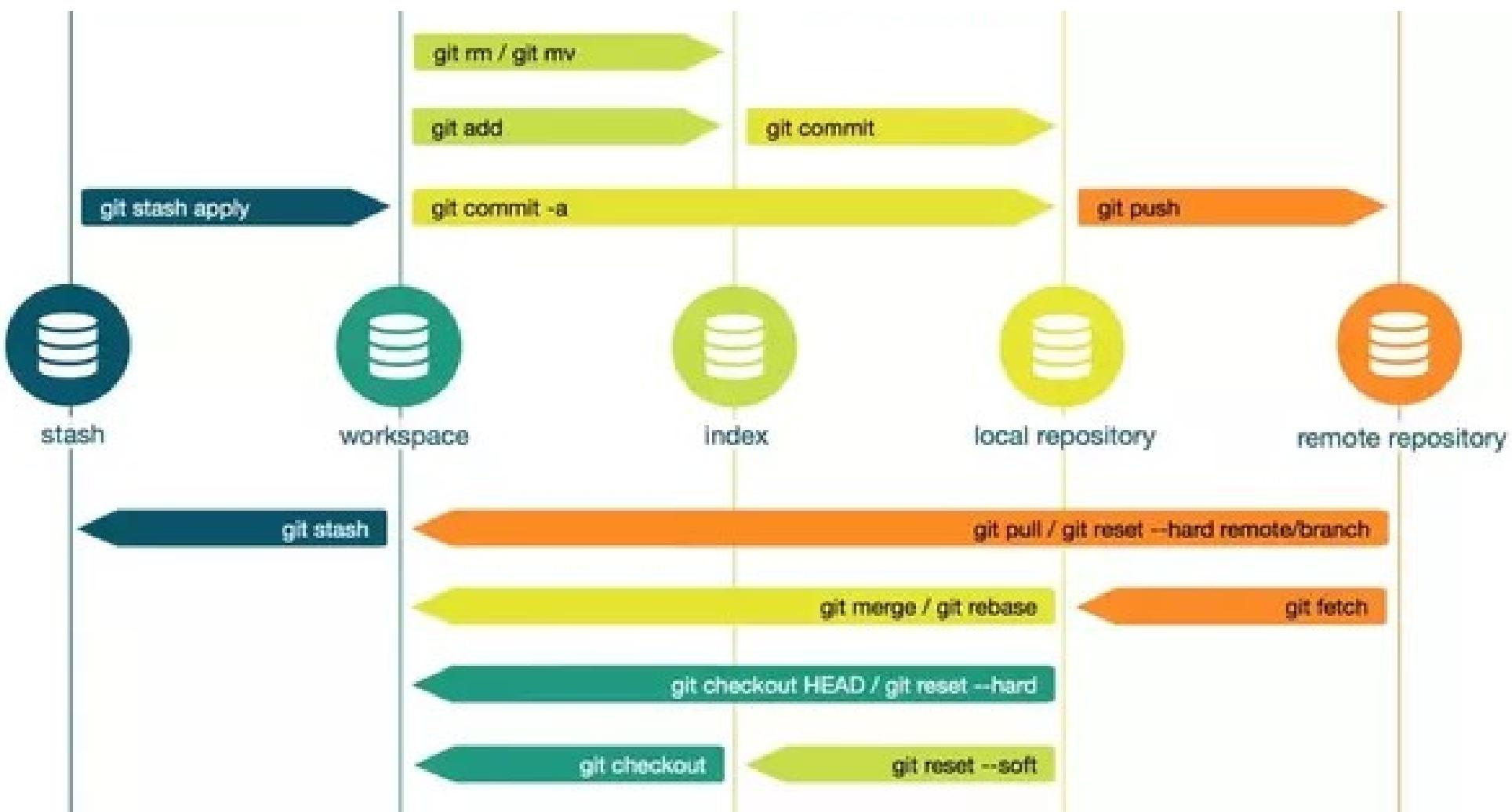
- Por parejas (Alice y Bob)
- Continuando con el ejercicio 10
- Bob creará una nueva rama, la empujará a GitHub y creará un PR
- Alice debe sacar la rama de Bob e incorporar nuevos commits empujándolos después a GitHub
- Bob debe rebasar los cambios de Alice, incluir nuevos commits y empujarlos a GitHub
- Bob debe aceptar el PR

# Desarrollo colaborativo con Git

- Introducción: Git y GitHub
- Git: trabajando en local
- Git y GitHub: trabajando en equipo
- **Gestionando el repositorio**
- Modelos de desarrollo

# Gestionando el repositorio

- **git stash**



# Gestionando el repositorio

- **git stash**

- Stash es un área especial
- Imaginemos un escenario donde, estando trabajando en una feature, nos llega un informe de un bug en la última release
- Nos toca arreglarlo, pero no hemos terminado la feature
- Tampoco está el código como para hacer un commit
- Podemos guardar los cambios desde el último commit en el área de stash
- Nuestra área de trabajo queda limpia
  - Podemos cambiar de rama, arreglar el bug y volver a la feature
- Aplicamos git stash apply y recuperamos el working area en el punto en que la guardamos

# Gestionando el repositorio

- **git revert**

- git reset nos permite volver a un commit anterior en la historia, generando commits que revierten el proyecto a un estado previo
- git revert nos permite eliminar un commit de la historia que no sea el último

- **git rm <file>**

- Marca un fichero en el índice para eliminar
- El commit incluirá la eliminación de dicho fichero

# Gestionando el repositorio

- **git commit --amend**

- Nos permite modificar el último commit
- Permite arreglar algún olvido/despiste cuando hicimos commit por última vez
  - Incluir un nuevo fichero
  - Modificar uno existente
  - ...
- Sólo tiene sentido si no hemos hecho push al repositorio remoto

# Gestionando el repositorio

- **git tag <tag name>**
  - Nos permite anotar un commit
  - Se utiliza para marcar releases, hitos, etc
  - Para subir el tag al repositorio remoto:
    - `git push --tags`
- **git tag**
  - Lista los tags del repositorio

# Gestionando el repositorio

- **Ejercicio 12**

- Bob anota el último PR como v1.0
- Alice hace un checkout de este tag v1.0

# Desarrollo colaborativo con Git

- Introducción: Git y GitHub
- Git: trabajando en local
- Git y GitHub: trabajando en equipo
- Gestionando el repositorio
- **Modelos de desarrollo**

# Modelos de desarrollo

- Trabajamos en equipo y tenemos múltiples entornos: dev, CI, QA, pre, pro...
- ¿Cómo debemos usar nuestros repositorios?
  - ¿En qué rama hacemos los commits del “día a día”?
  - ¿Sobre qué rama corre CI los tests?
  - ¿De dónde sacamos el entorno de QA?
  - ¿Qué rama se saca a producción?
- Es necesario definir el **modelo de desarrollo**, cómo vamos a gestionar sus ramas

# Modelos de desarrollo

- A lo largo de los años diferentes modelos de desarrollo han sido propuestos
- Entre los más comúnmente aceptados están:
  - Git flow
  - GitHub flow
  - GitLab flow
  - Trunk-based development

# Modelos de desarrollo

- Todos se basan en los mismos conceptos:
  - Ramas (en mayor o menor medida)
  - Gestión de versiones
    - Marcar alguna rama como la fuente única de verdad de la que salen las versiones de producción
    - Tags para marcar versiones
  - Mecanismos de resolución de bugs (hotfixes)
  - Integración con releases en mantenimiento y desarrollo

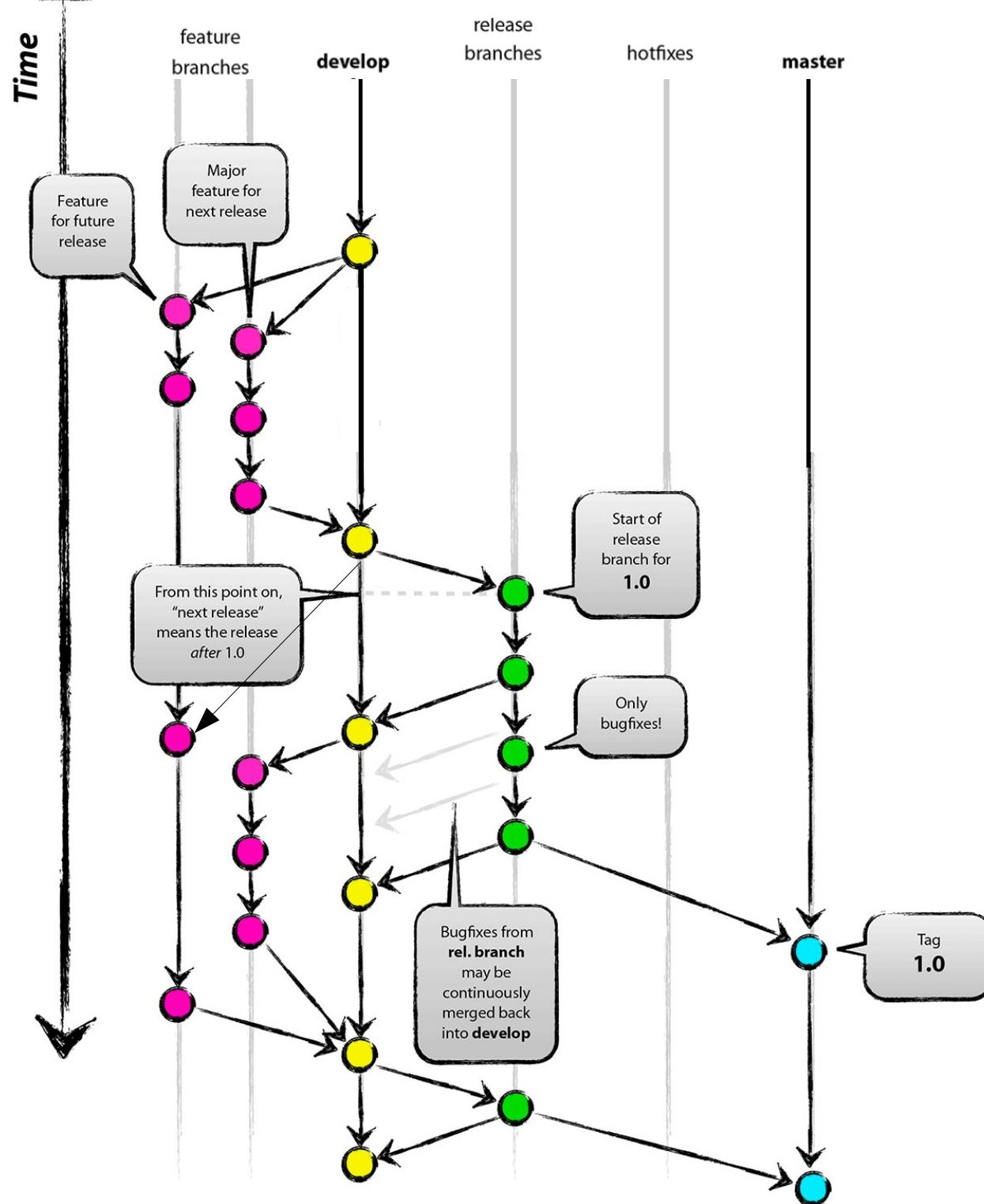
# Desarrollo colaborativo con Git

- Introducción: Git y GitHub
- Git: trabajando en local
- Git y GitHub: trabajando en equipo
- Gestionando el repositorio
- Modelos de desarrollo
  - **Git flow**
  - Trunk based development
  - GitHub flow y GitLab flow

# Git flow

- Creado por Vincent Driessen en 2010
  - <https://nvie.com/posts/a-successful-git-branching-model/>
  - <https://jeffkreeftmeijer.com/git-flow/>
- Modelo estricto en el uso de ciertas ramas
  - master
  - develop
  - feature (varias)
  - release (varias)
  - hotfix (varias)

# Git flow



# Git flow

- La rama **master** siempre refleja código listo para desplegarse en **producción**
  - Sólo contiene releases: se crea un commit por cada versión. Cada uno de ellos tiene el tag con la versión
  - Por definición cada vez que se **mezcla en master** se produce una **release**
  - Todos los commits acaban llegando a máster para ser desplegados en producción (en una release)

# Git flow

- La rama **develop** es la rama principal de **desarrollo**
  - Sus commits reflejan las últimas funcionalidades desarrolladas
  - **Varias** de estas **funcionalidades** acabarán componiendo la siguiente **release**
  - De esta rama se construye el software que será probado en el servidor de integración continua y se **desplegará** en el **entorno de desarrollo**
  - También se la denomina **rama de integración**

# Git flow

- Ramas de funcionalidades (*feature branches*)
  - Se utilizan para **desarrollar las funcionalidades** de forma aislada para no interferir con otros desarrollos
  - Se crean desde la rama **develop**
  - Cuando la funcionalidad está lista, la rama se integra de nuevo en **develop** (con **merge o rebase**)
  - Normalmente se nombran **feature-feat** o **feature-id** con el id de la tarea en la que gestiona esa feature

# Git flow

- Ramas de funcionalidades (*feature branches*)
  - Antes de “integrar” el código en la rama `develop` se pasan unos controles de calidad
    - La build construye
    - Los tests pasan
    - ...
  - Aunque esas verificaciones se puedan hacer en **local**, el servidor de **integración continua (CI)** suele ejecutar los controles antes de permitir la mezcla con `develop`

# Git flow

- **Ramas de release (*release branches*)**
  - Son ramas de estabilización del código de develop
    - No se integran nuevas features
    - Se pasan más controles de calidad
    - Se arreglan bugs
    - Se prepara el código para la release: número de versión, fecha, ...
  - Son ramas que se bifurcan desde **develop**

# Git flow

- **Ramas de release (*release branches*)**
  - Cuando se termina la funcionalidad se integra de nuevo en:
    - **develop:** para trabajar en la siguiente release
    - **master:** toda release está en master
  - Normalmente se nombran **release-\***
  - Mientras se estabiliza el código para la release, la rama develop puede seguir admitiendo nuevas features

# Git flow

- **Ramas de Hotfix (*hotfix branches*)**

- Se utilizan para solucionar problemas en una release
- Son ramas que se bifurcan desde **master**
- Cuando se termina la funcionalidad se integra de nuevo en:
  - **master**: una nueva release
  - **develop**: de forma que el fix esté en posteriores releases también
- Normalmente se nombran **hotfix-\***
- Suelen incrementar el minor version de la release

# Git flow

- El flujo es tan complejo que hay una herramienta para ayudarnos a ser disciplinados
  - Inicializa el repositorio creando las ramas master y develop
  - Nos ayuda a empezar con el desarrollo de una feature creando la rama y posicionándonos en ella
  - Nos ayuda a mezclar una rama de vuelta a develop
  - Incluye funcionalidades para publicar ramas
  - También nos ayuda a mezclar releases en master

<https://github.com/nvie/gitflow/wiki>

# Git flow

- Inicializar un repositorio git con git flow:

```
$ git flow init -d
```

- Nos crea un commit inicial en master y la rama develop
- Nos sitúa en develop

# Git flow

- No podemos empezar a meter commits en develop, debemos crear una rama para la feature

```
$ git flow feature start createapp
```

- Nos crea la rama feature/createapp
- Nos situa en ella
- Nos indica cómo proceder cuando terminemos la feature
- Realizar un par de commits en la rama

# Git flow

- Cuando terminemos la feature:

```
$ git flow feature finish createapp
```

- Se mueve a la rama master
- Mezcla haciendo un commit de merge en develop desde la rama feature/createapp
  - Se puede forzar rebase (-r) o squash (-S)
- Elimina la rama feature/createapp
  - Se puede mantener (-k)

# Git flow

- Ejercicio 13
  - Repetir el proceso con dos features
    - Se crea la feature f<sub>1</sub>, se introducen un par de commits
    - Sin cerrar f<sub>1</sub>, se crea la feature f<sub>2</sub>
    - Se introducen un par de commits en f<sub>2</sub> que toquen el mismo fichero que f<sub>1</sub>
    - Se finaliza f<sub>2</sub>
    - Se finaliza f<sub>1</sub>

# Git flow

- Una vez que tenemos en develop las features que queremos para la próxima release podemos preparar la release

```
$ git flow release start 0.1
```

- Se crea la rama release/0.1 partiendo de develop
- Nos sitúa en ella
- Nos da instrucciones de cómo terminar la release

# Git flow

- Añadimos “0.1” a un fichero version.txt

```
$ echo '0.1' > version.txt
```

- Terminamos la release

```
$ git flow release finish 0.1
```

- Mezcla la rama release/0.1 con master generando un commit (se nos pide el mensaje de commit)
- Genera un tag en master para esta release (se nos pide el mensaje de tag)
- Mezcla la rama master con develop (se nos pide el mensaje del commit de merge en develop)
- Nos deja en develop

# Git flow

- Surge un problema en prod y hay que arreglar un bug
- Iniciamos el proceso de hotfix

```
$ git flow hotfix start 0.1.1
```

- Nos saca una rama hotfix/0.1.1 desde el último commit de master
- Cambiamos la versión en el fichero version.txt y comiteamos

# Git flow

- Terminamos el proceso de hotfix
  - \$ **git flow hotfix finish 0.1.1**
- Mezcla la rama hotfix/0.1.1 con master generando un commit (se nos pide el mensaje de commit)
- Genera un tag en master para esta release (se nos pide el mensaje de tag)
- Mezcla la rama master con develop (se nos pide el mensaje del commit de merge en develop)
- Nos deja en develop

# Desarrollo colaborativo con Git

- Introducción: Git y GitHub
- Git: trabajando en local
- Git y GitHub: trabajando en equipo
- Gestionando el repositorio
- Modelos de desarrollo
  - Git flow
  - **Trunk based development**
  - GitHub flow y GitLab flow

# Trunk-based development

- Muchos modelos de desarrollo se basan en ramas
- Las ramas con una duración larga (superior a 2 días) son consideradas problemáticas
  - Llevan a merges más grandes (aka merge de la muerte)
  - Mucho trabajo para resolverlos
- Aunque las ramas cortas (1 o 2 días) son mejores, también tienen limitaciones
  - Sólo se integran con el código del resto del equipo cuando se hace el Pull Request
  - Fácilmente se convierten en ramas de duración larga...

# Trunk-based development

- Las bases de TBD

- Todo el desarrollo sucede en *trunk* (*master* en git)
- Se suben los cambios al menos una vez al día
- *Trunk* siempre está en un estado listo para hacer release y desplegar a producción: la *build* pasa
- Código que no está listo se oculta mediante *feature flags* (aka *feature toggles*)
- Soluciona problemas de refactoring con la técnica de *branch by abstraction*

<https://trunkbaseddevelopment.com/>

# Trunk-based development

- **Las bases de TBD**

- Los desarrolladores se actualizan a menudo desde *trunk*
- Los desarrolladores (preferiblemente haciendo pair-programming) suben pequeños cambios a menudo a trunk
- Antes de subir el código el único requisito es **ejecutar la *build*** del proyecto primero
  - Debe poder ejecutarse sin errores
  - Incluye construcción + pruebas

<https://trunkbaseddevelopment.com/>

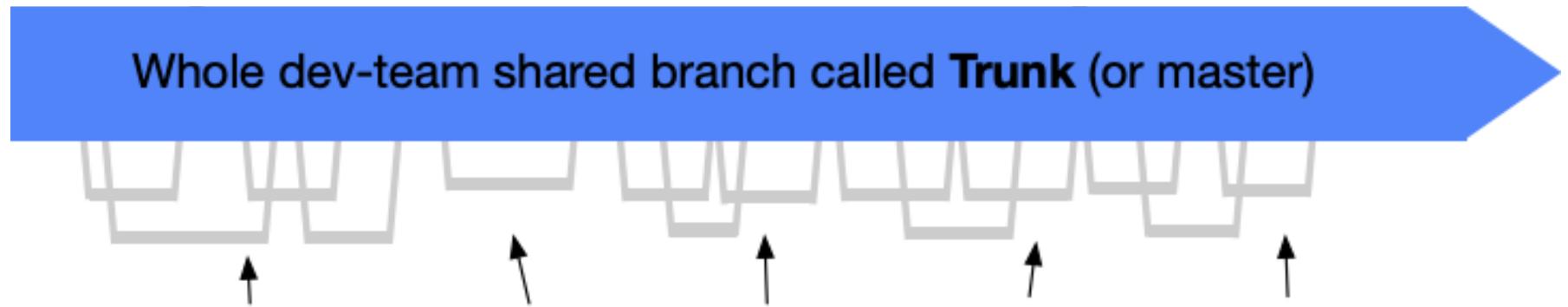
# Trunk-based development

- TBD para equipos grandes
  - El desarrollo se realiza en ramas de vida corta (2 días máximo)
  - De una única persona
  - Se integra en trunk mediante **pull requests** que permiten introducir en este punto:
    - Revisión de código
    - Ejecutar la build

<https://trunkbaseddevelopment.com/>

# Trunk-based development

- TBD para equipos grandes



Trunk-Based Development at scale is best done with **short-lived feature branches**: one person over a couple of days (max) and **flowing through Pull-Request style code-review & build automation** before “integrating” (merging) into the trunk (or master)

# Trunk-based development

- Automatización de la *build* es necesaria para verificar que las integraciones con *trunk*
  - Compilan
  - Pasan al menos pruebas unitarias y de integración
  - Pruebas más complejas pueden no poder ejecutarse en la máquina de desarrollo en el modelo para equipos pequeños aunque el modelo establece que deberían poder ejecutarse incluso pruebas funcionales
  - En el modelo de escala todas las pruebas pueden ejecutarse en integración continua gracias al *pull request*

# Trunk-based development

- **Se puede realizar despliegue continuo**

- Por definición el código siempre está listo para ser desplegado en producción
- Con git flow pueden pasar días o semanas hasta que hay un nuevo commit en master para ser desplegado

- ***Lean experiments***

- Disponiendo de despliegue continuo (CD) es más sencillo poder realizar experimentos en producción
- Permiten evaluar soluciones basadas en análisis de mercado

# Trunk-based development

- **Prerequisitos**
  - Una sólida infraestructura de desarrollo
    - Desarrolladores deben poder ejecutar la aplicación localmente para verificar la *build* (incluyendo ciertas pruebas)
    - No tiene que comportarse a nivel no-funcional como producción
  - Requiere dominar ciertas técnicas de diseño
    - Técnicas para implementar *feature toggles* en nuestra aplicación
    - Técnicas para implementar *branch by abstraction*
  - Requiere dominar ciertas técnicas de nuestro VCS
    - Integraciones de *commits* específicos en ramas de *release* con *cherry pick*

# Trunk-based development

- ¿Cómo es el proceso de desarrollo?
  - *Trunk* siempre está en situación de ser desplegado
    - *Trunk* compila y pasa los tests (regla de no romper nunca la *build*)
    - Las *releases* pueden salir de *trunk* o de una rama de *release* creada en el último momento
    - Menos de una hora en salir a producción
  - *Checkout / clone*
    - Todo el equipo hace *checkout* y *clone* de *trunk*
    - *Update / pull / sync* varias veces al día de *trunk* con seguridad (la *build* pasa)
    - Integración de *commits* del resto del equipo **cada hora**

# Trunk-based development

- ¿Cómo es el proceso de desarrollo?
  - *Commits*
    - Cuando se completa una unidad de trabajo se hace un *commit* a *trunk* (remoto) si no rompe la *build*
    - El tamaño del *commit* debería ser lo más **pequeño posible** sin romper la build
    - Se debe poder **probar que no rompe la build** antes de mezclar con *trunk* (ejecutar en local o CI)
    - Es posible que haya que hacer un *pull* adicional antes de hacer el *commit* (puede haber un *push* de otro desarrollador a *trunk* antes del nuestro): se debe volver a **ejecutar la build**

# Trunk-based development

- ¿Cómo es el proceso de desarrollo?
  - Revisión de código
    - El *commit* debe ser revisado
    - Se puede saltar si ha sido desarrollado trabajando en pares
    - Algunos equipos usan *pull requests* para esto
    - Las ramas de PR deben ser muy cortas en tiempo
    - Las ramas de PR deben borrarse tras integrar en trunk

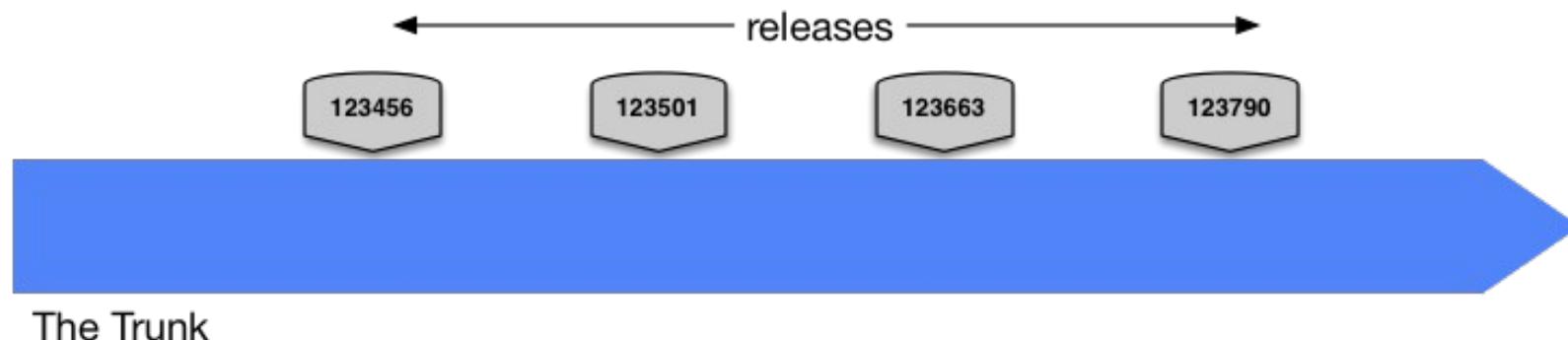
# Trunk-based development

- Opciones para publicar la release
- **Situación ideal:** Equipos que despliegan con frecuencia lo hacen directamente desde *trunk*
- **Situación transitoria:** Equipos que despliegan con menos frecuencia hacen una rama de *release*

# Publicación de release

- **Releases desde *trunk***

- Equipos con una cadencia muy alta de *releases*
- Probablemente se usen los *commits* o fecha y hora para las versiones (en lugar de *semantic versioning*)
- Los *bugs* se arreglan en *trunk* y se despliega una nueva versión (*roll forward*)



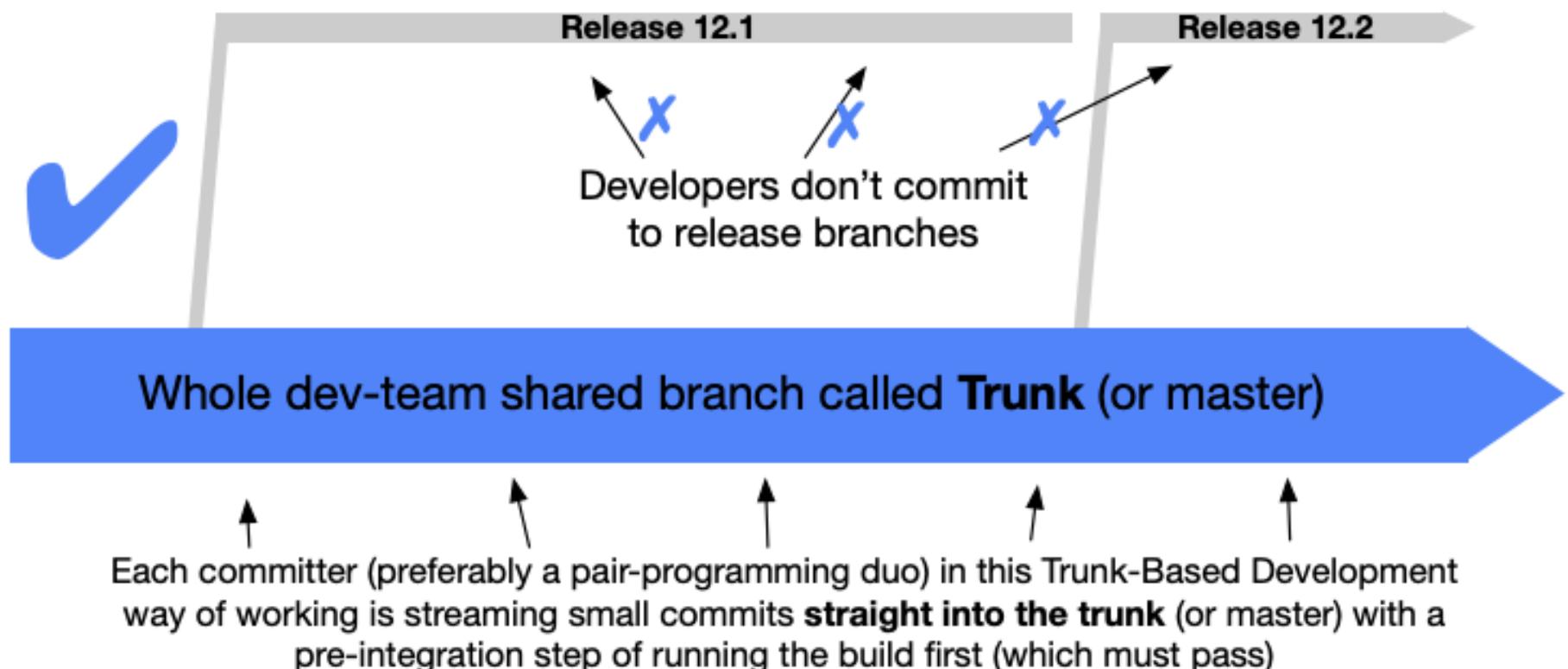
# Publicación de release

- **Releases desde ramas**

- No se hacen *commits* a la rama de *release*
- Técnicamente la rama es un solo *commit* (al que apunta el tag con la versión)
- Los *bugs* se arreglan en *trunk* y sus *commits* se copian en la rama de *release* (*cherry pick*)

# Publicación de release

- Releases desde ramas



# Trunk-based development

- Ejemplo

- Inicializamos un repositorio git en una carpeta

```
$ git init
```

- Creamos un par de features en master, cada una compuesta de un par de commits

```
$ echo "f1.1" > f1.txt
```

```
$ git add .
```

```
$ git commit -a -m "Start f1"
```

```
$ echo "f1.2" >> f1.txt
```

```
$ git commit -a -m "Finish f1"
```

# Trunk-based development

- Ejemplo

- Comenzamos el desarrollo de f3 (dos commits), a mitad sacamos una release

```
$ echo "f3.1" > f3.txt  
$ git add .  
$ git commit -a -m "Start f3"  
$ git checkout -b release-1.0  
$ git tag "v1.0"  
$ git push origin release-1.0  
$ git push --tags
```

# Trunk-based development

- Ejemplo

- Se termina f3 y se corrige un bug en f2 (que afecta a la release 1.0)

```
$ echo "f3.2" >> f3.txt  
$ git add .  
$ git commit -a -m "Finish f3"  
$ echo "bug fix" >> f2.txt  
$ git checkout -b release-1.0  
$ git cherry pick <commit con el bug fix>  
$ git tag "v1.1"  
$ git push --tags
```

# Trunk-based development

- **Integración Continua**

- CI es una parte integral de TBD
- El servidor de CI debe ejecutarse con cada commit de trunk (y de las ramas si usamos PR)
- Debe vigilar que la build no tiene fallos
- En caso contrario notificar inmediatamente
  - Algunos equipos hacen **roll back** manual del commit
  - Otros permiten que CI lo haga automáticamente (*revert*)

# Trunk-based development

- Consideraciones adicionales

- Factores que afectan a la cadencia de releases
  - <https://trunkbaseddevelopment.com/deciding-factors/#release-cadence>
- Hábitos en TBD
  - <https://trunkbaseddevelopment.com/observed-habits/>
- No estamos haciendo TBD
  - <https://trunkbaseddevelopment.com/youre-doing-it-wrong/>

# Trunk-based development

- Funcionalidades que tardan días en implementarse
  - Master/trunk debe ser estable
  - El desarrollo sucede en master principalmente
    - Commits en master a diario
    - Ramas de menos de dos días
  - Se puede desplegar master en producción en cualquier momento
  - ¿Cómo abordar la implementación de una funcionalidad que dura más de dos días?
  - ¿Cómo mantener master estable subiendo commits relativos a una funcionalidad “sin terminar”?

# Trunk-based development

- Solución más sencilla (que no es TBD)
  - Feature branches de varios días
  - La rama durará más de dos días sin integraciones a master
  - Sincronización (rebase o merge) de master a la rama se pueden hacer a diario para minimizar el impacto del merge
- Problemas
  - Rama larga > merge problemático
  - El resto del equipo no ve los cambios hasta el merge o PR

# Trunk-based development

- Soluciones que permiten mantener el ritmo de releases y la estabilidad de master
  - Feature toggles
  - Branch by abstraction
- Prácticamente todos los modelos de desarrollo se apoyan en estas técnicas
- Para algunos modelos como TBD son críticas

# Feature toggles

- Una feature flag es un interruptor que **permite activar y desactivar una feature**
- Esta activación/desactivación puede ser **estática** (durante el despliegue en base a una configuración) o **dinámica** (en ejecución)
- Es un mecanismo que posibilita que sólo **algunos usuarios puedan usar ciertas funcionalidades** (A/B testing, canary releases, dark launches..., deployment vs release...)

<https://www.martinfowler.com/articles/feature-toggles.html>

# Feature toggles

- Cambio dinámico

```
        {
if ( one.click.checkout ,
    key: "bob@example.com",
    groups: ["gold", "google"],
    inBeta: true
)
then
    /* show the one-click checkout feature */
else
    /* show the old feature*/
```

<https://featureflags.io/>

<https://trunkbaseddevelopment.com/feature-flags/>

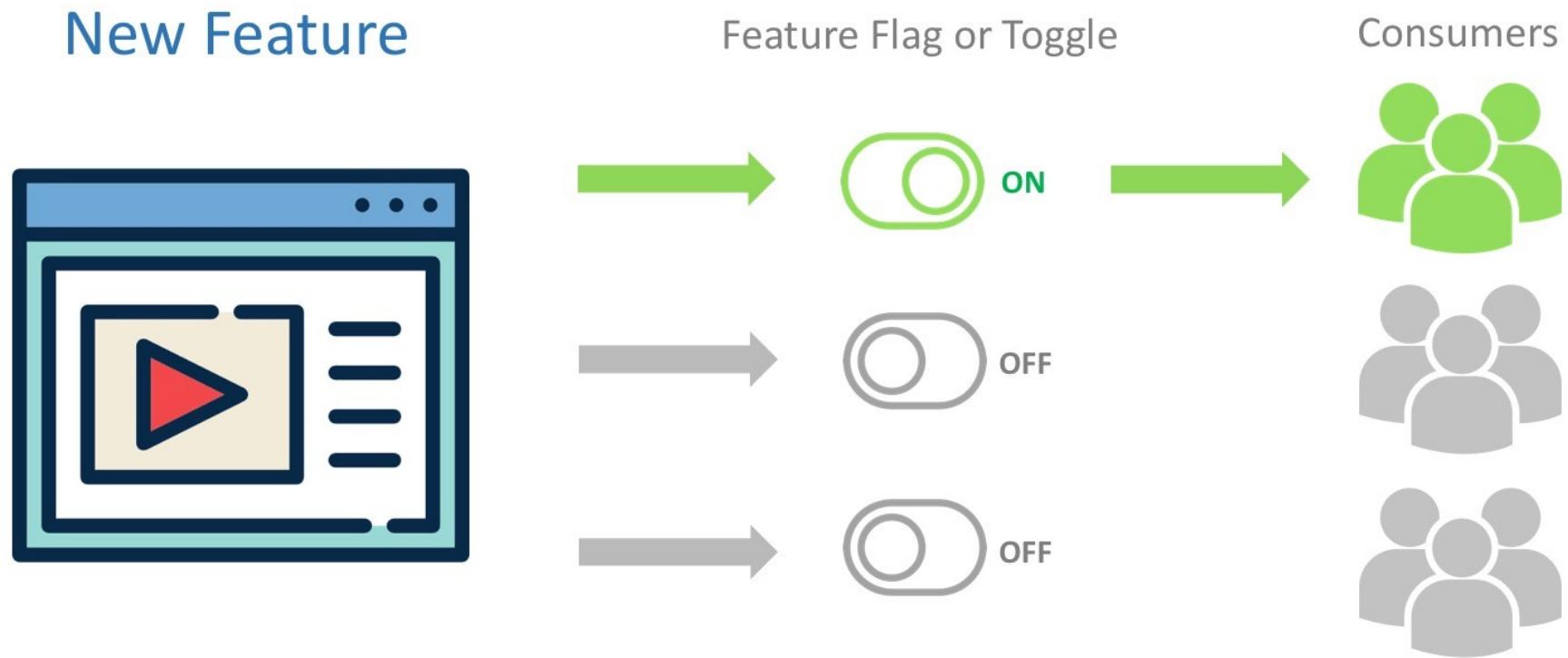
<https://www.baeldung.com/spring-feature-flags>

<https://medium.com/@thysniu/coding-with-feature-flags-how-to-guide-and-best-practices-3f9637f51265>

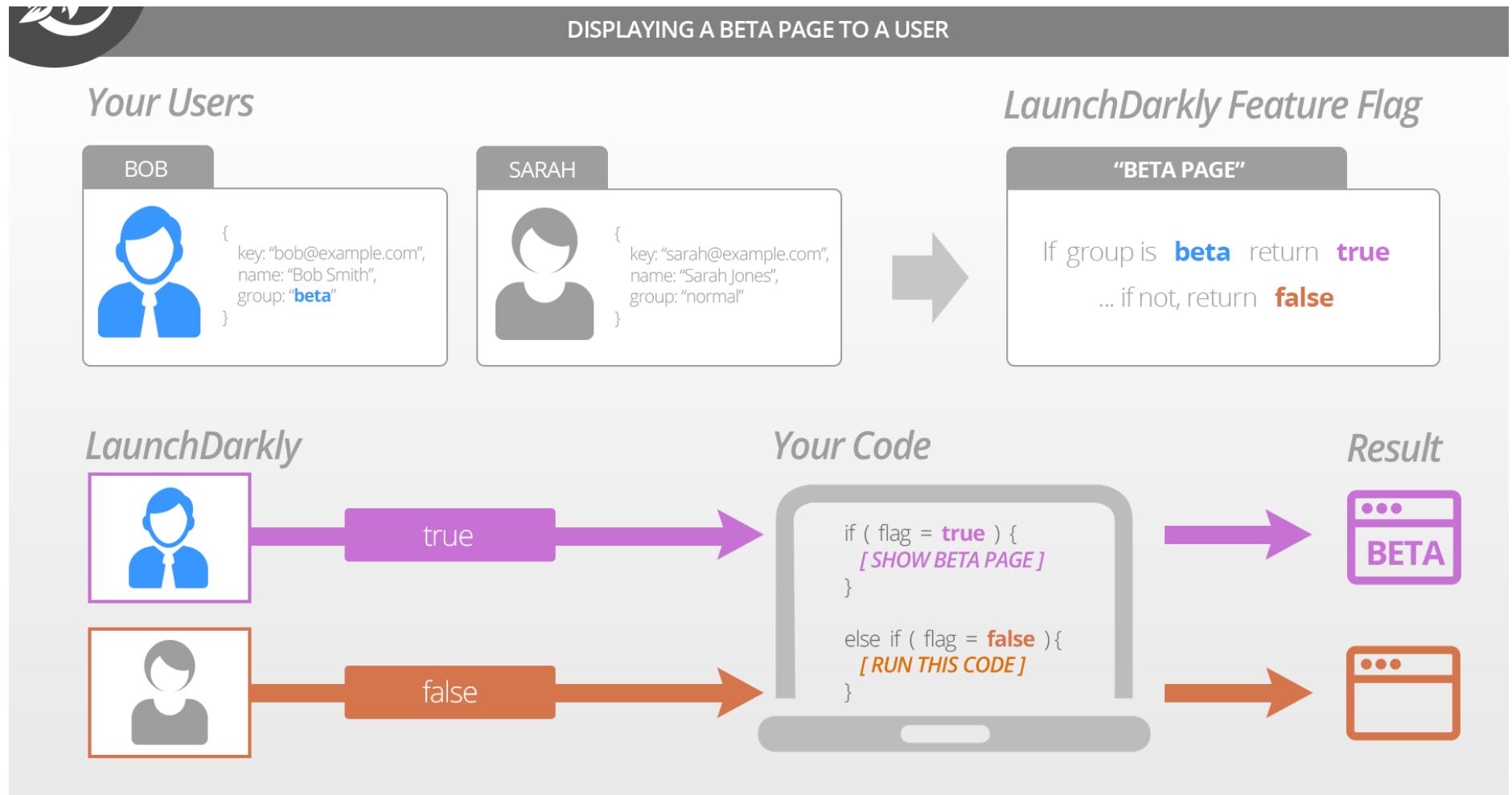
# Feature toggles

- Casos de uso
  - Releases con funcionalidades incompletas
    - Hacer release en cualquier momento sin importar el estado de otras funcionalidades
  - Canary releases
    - Desplegar funcionalidades y activarlas sólo a un subconjunto de los usuarios
    - Según el feedback se pueden desactivar o activar para el resto
  - Lean experiments
    - Desplegar algunas funcionalidades experimentales para usuarios específicos

# Feature toggles



# Feature toggles



# Feature toggles

- Casos de uso

- Dark launches
  - Una funcionalidad terminada se despliega sin ser visible a los usuarios
  - Se utiliza para verificar que funciona correctamente de forma interna antes de hacerla visible
- Incremental roll outs
  - Una funcionalidad se va abriendo paulatinamente a más usuarios según se comprueba que funciona correctamente
- Early access
  - Se proporciona una funcionalidad a un subconjunto de usuarios que solicitan a probarla

# Feature toggles

- **Buenas prácticas**

- Si un toggle está siempre en el mismo estado eliminarlo
- Es difícil saber la granularidad de los toggles, depende de la experiencia
- Establecer una nomenclatura para los toggles
- Auditoría: quién activa o desactiva un toggle
- Proporcionar acceso a diferente personal (QA) incluyendo no técnico (producto)
- No hacer depender toggles de otros toggles
  - La complejidad se dispara
  - Difícil de entender por qué un toggle está en un determinado estado
  - Visibilidad del estado de los toggles para el equipo: dashboard

# Feature toggles

- **Implementación**

- Usando propiedades de configuración de la tecnología
- Usando herramientas/servicios de gestión



<https://launchdarkly.com/>



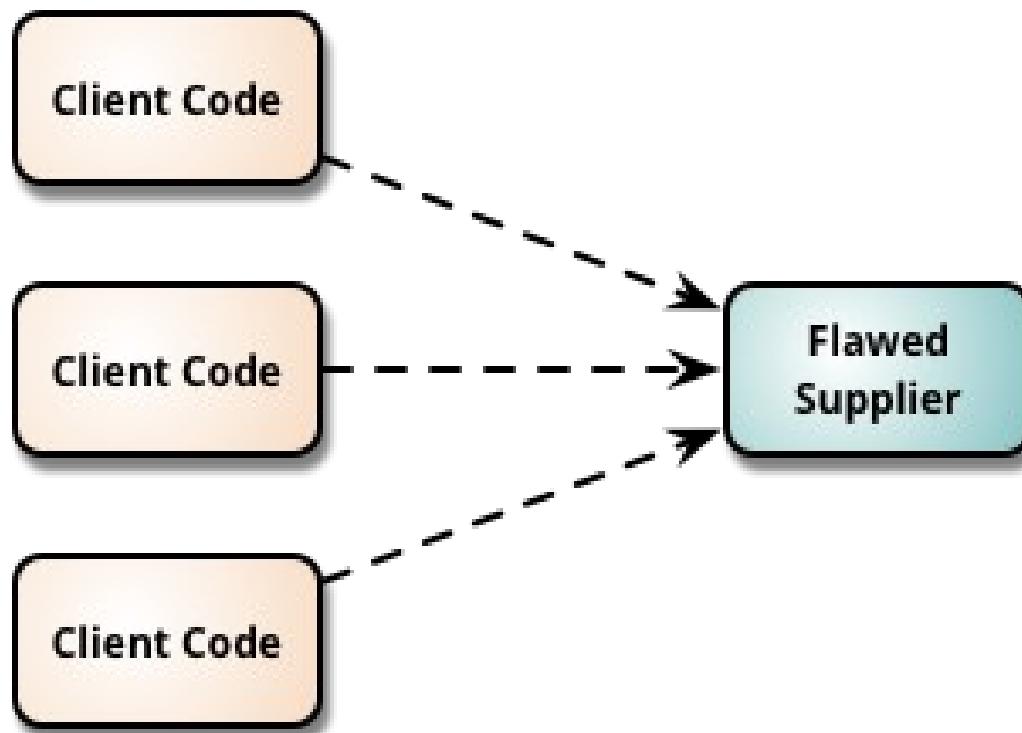
<https://www.togglz.org/>

# Branch by abstraction

- Técnica que permite introducir una nueva implementación de una funcionalidad mientras la anterior funcionalidad sigue activa y usándose por los usuarios
- Se divide el cambio en fases:
  - Abstracción del código que se desea cambiar
  - Inclusión de la nueva implementación dejando la anterior todavía vigente
  - Cuando está implementada completamente, activación de la nuevo implementación
  - Eliminación de la abstracción

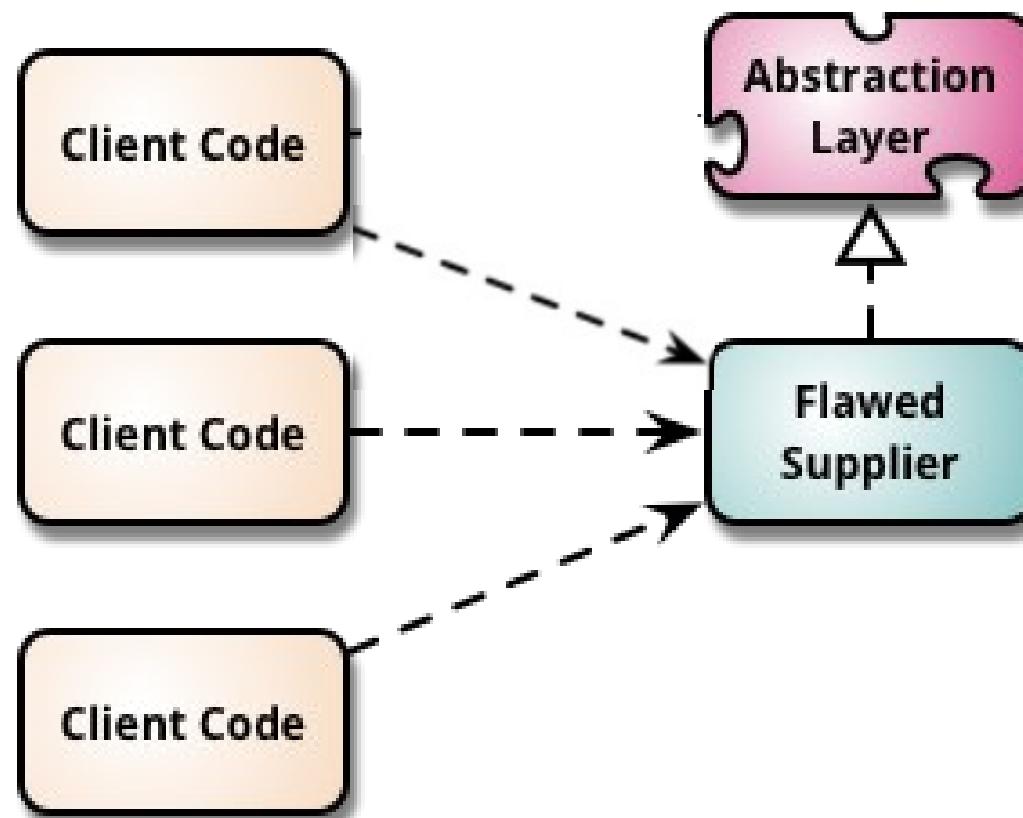
# Branch by abstraction

- Código original



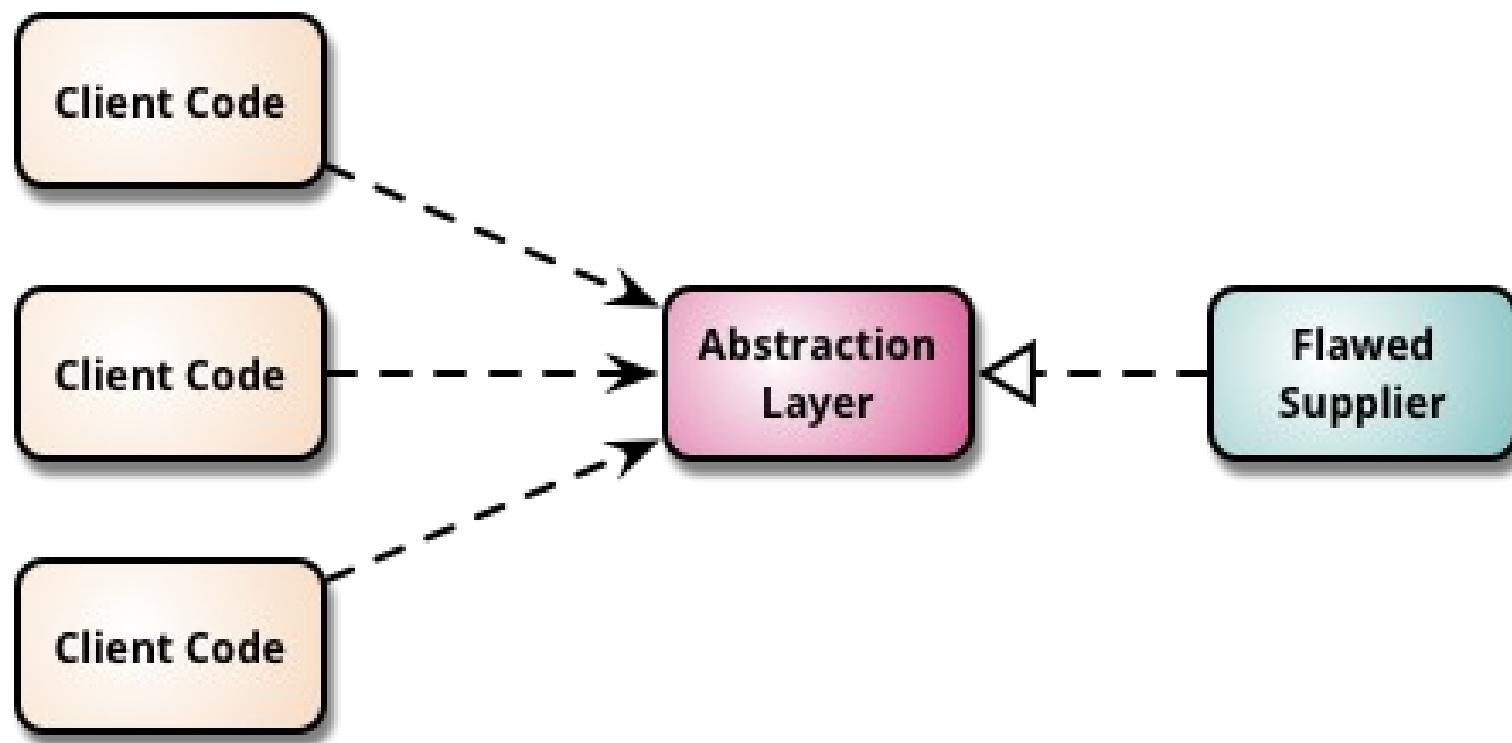
# Branch by abstraction

- Se crea una capa de abstracción (fachada)



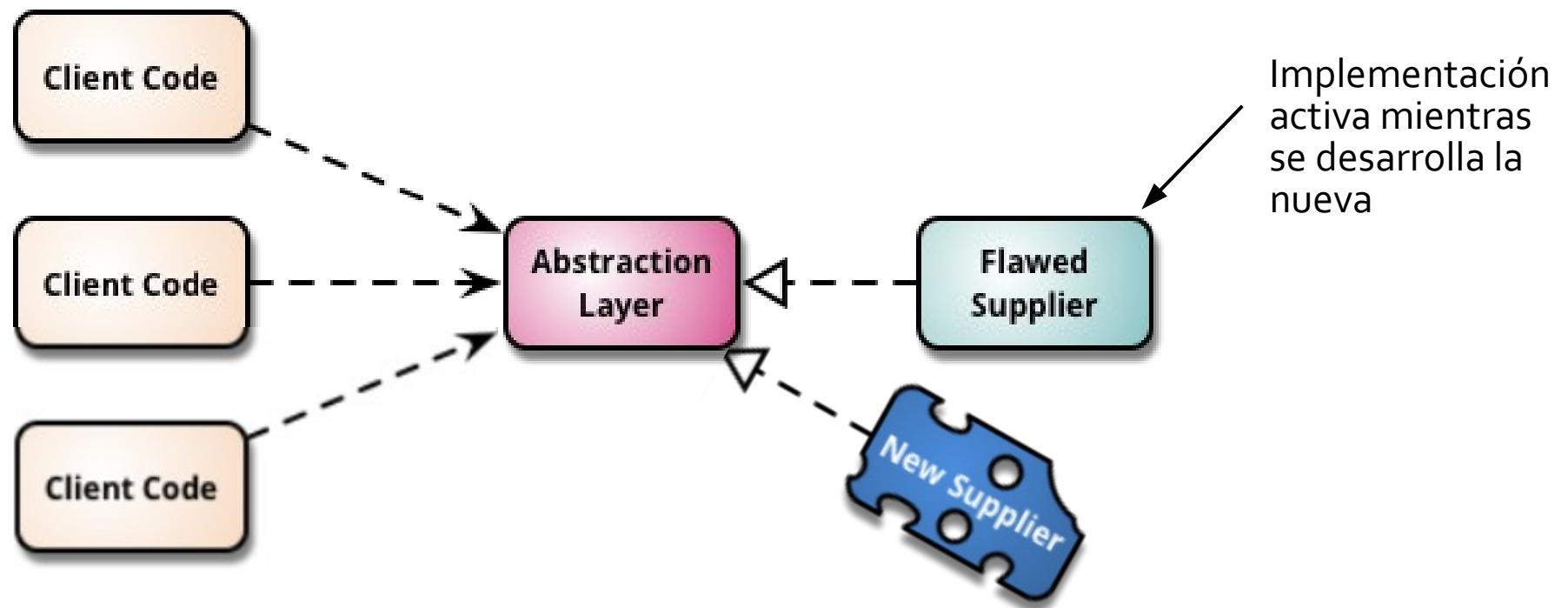
# Branch by abstraction

- Todo el código usa la capa de abstracción



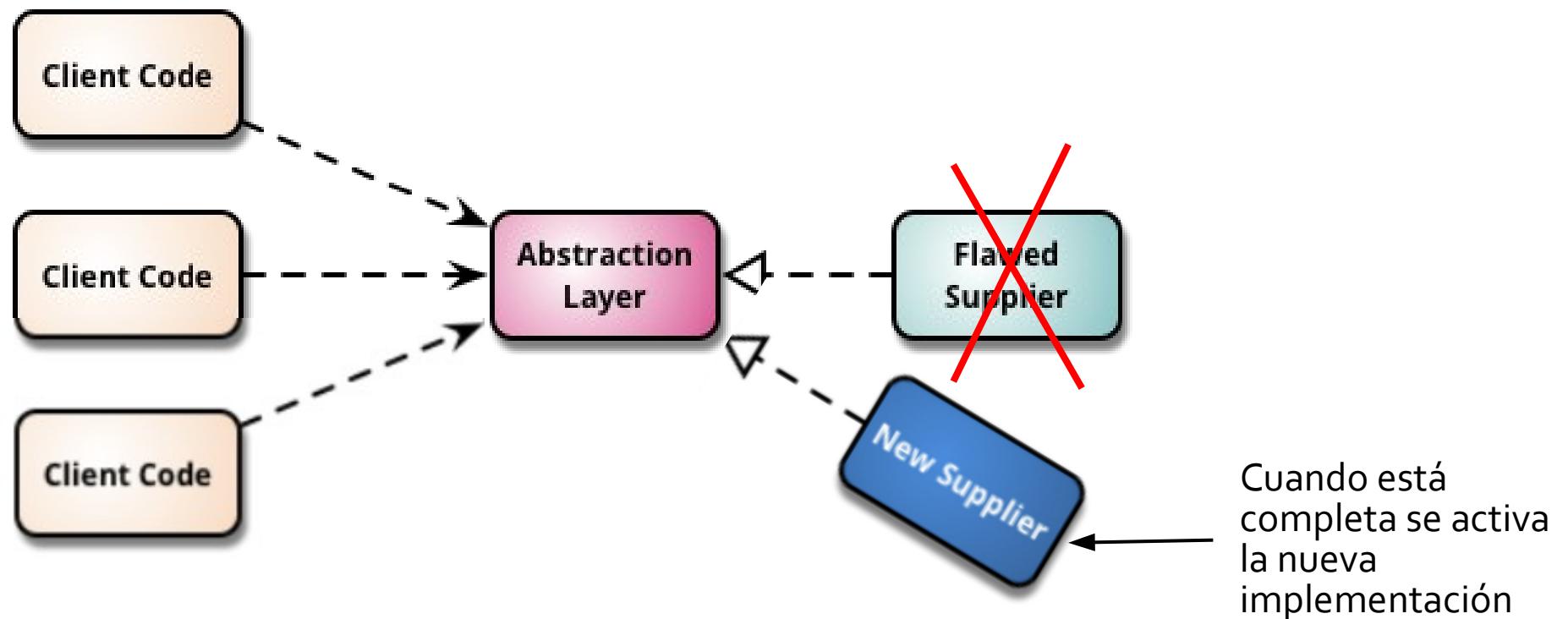
# Branch by abstraction

- Todo el código usa la capa de abstracción



# Branch by abstraction

- Todo el código usa la capa de abstracción



# Branch by abstraction

- Estas fases no impactan en el código que se ejecuta actualmente (el nuevo código está deshabilitado)
- Pero permiten que todo el equipo pueda comentar sobre los cambios
- Master puede ser liberado y desplegado en cualquier momento

<https://martinfowler.com/bliki/BranchByAbstraction.html>

<https://trunkbaseddevelopment.com/branch-by-abstraction/>

# Desarrollo colaborativo con Git

- Introducción: Git y GitHub
- Git: trabajando en local
- Git y GitHub: trabajando en equipo
- Gestionando el repositorio
- Modelos de desarrollo
  - Git flow
  - Trunk based development
  - **GitHub Flow**

# GitHub flow

- **Git flow** es un modelo de desarrollo muy pesado con mucho protocolo
  - Abundantes ramas
  - Propenso a errores
  - El histórico es difícil de seguir
- **Trunk-based development** es un modelo muy ligero pero con ciertos problemas
  - Fuerza a ramas muy cortas
  - Necesidad de dominar técnicas como **feature toggles** y **branch by abstraction**

# GitHub flow

- GitHub flow pretende ser un término intermedio entre Git flow y TBD
  - La rama de integración es master
    - Master está siempre lista para ser versionada y desplegada
  - Features en ramas (y cualquier otra cosa: experimentos, refactorings...)
    - Las ramas no se mezclan con master hasta que están listas
  - Pull requests para mezclar una feature en master
    - Revisión de código
    - Otros desarrolladores pueden ver lo que se va a mezclar
    - CI <https://githubflow.github.io/>

<https://guides.github.com/introduction/flow/>

# GitHub flow

- GitHub tiene dos modelos de Pull Requests
  - Fork & Pull Request
    - Cuando la persona que contribuye al proyecto no pertenece al equipo que lo mantiene y por tanto no tiene permisos en el repositorio del proyecto
    - Útil en proyectos open source
  - Pull Request en un repositorio compartido
    - Cuando la persona que contribuye trabaja sobre el repositorio del proyecto
    - Se empuja la rama local al repositorio y se crea un PR desde la rama remota

# GitHub flow

- **Revisión de código con Pull Requests**

- Asociado al PR se pueden incluir mensajes
  - Hilo de conversación
  - Puede incluir referencias a issues, menciones a otros miembros del proyecto, ...
- Se pueden hacer anotaciones en el código de los commits
- Se pueden seguir haciendo commits a la rama y se reflejan
  - Las notificaciones de nuevos commits en el PR se envían a CI y se pueden pasar los tests

# GitHub flow

- **Releases**

- Una vez el PR ha sido revisado y pasa los tests se despliega en producción
- Si todo va bien
  - Se acepta el PR
  - Se mezcla en master
- Si algo va mal
  - Se retira la versión
  - Se despliega master
  - Se arregla el PR mediante nuevos commits y se repite el proceso

# GitHub flow

- **Releases**

- Se puede aceptar el PR y mergear en master primero pero inmediatamente hay que desplegar a producción
- Objetivo: evitar poner cambios encima de otros que no se han probado todavía en producción

# Comparativa

- **Git flow**

- El más complejo
- Requiere mucha disciplina: merges desde la rama apropiada

- **Trunk-based development**

- El más sencillo
- Requiere dominar ciertas técnicas de diseño para cambios grandes/features

- **GitHub flow**

- Muy sencillo: master + feature branches con PRs
- Muchas cosas quedan en el aire