

Hada T1: Control de versiones

git: Un sistema de control de versiones distribuido.

Departamento de Lenguajes y Sistemas Informáticos Universidad de Alicante

Objetivos del tema

- Conocer los conceptos básicos asociados a los *Sistemas de Control de Versiones*.
- Aprender a usar **git** localmente de manera individual.
- Aprender a usar **git** en grupo
- Conocer y saber cómo usar **git** con **github** (en remoto).



Usar un Sistema de Control de Versiones es como tener una máquina del tiempo

para tus documentos/código



**BACK
TO THE PAST**

Mucha gente utiliza un sistema de control de versiones muy básico

- Normalmente la gente guarda diferentes versiones del mismo archivo

miprograma.c / miprogramav2.c / programaFINAL.c / programaFINALfinal.c

- o incluso usa fechas:

miprograma2018-12-02.c / miprograma2018-12-03.c

¿POR QUÉ QUEREMOS GUARDAR VERSIONES ANTERIORES?

EN GRUPOS (1 minuto)

- ¿CUÁLES SON LOS **PROBLEMAS** QUE NOS PODEMOS ENCONTRAR USANDO ESTE MÉTODO?
- ¿qué pasa cuando se trabaja **colaborativamente** en un documento?



Problemas

1. **CUÁLES** son los cambios entre las diferentes versiones
2. **CUÁNDO** se hizo un cambio concreto



Por lo tanto... ¿cómo podemos **recuperar una versión concreta** de un archivo?



Cuando trabajamos colaborativamente...



1. ¿Cómo **comunicamos los cambios** que hemos hecho a las otras personas del grupo?
2. ¿Cómo podemos **mezclar/fusionar diferentes versiones**?
3. ¿Cómo sabemos **QUIÉN** hizo una modificación concreta?

Documentos/Código colaborativo



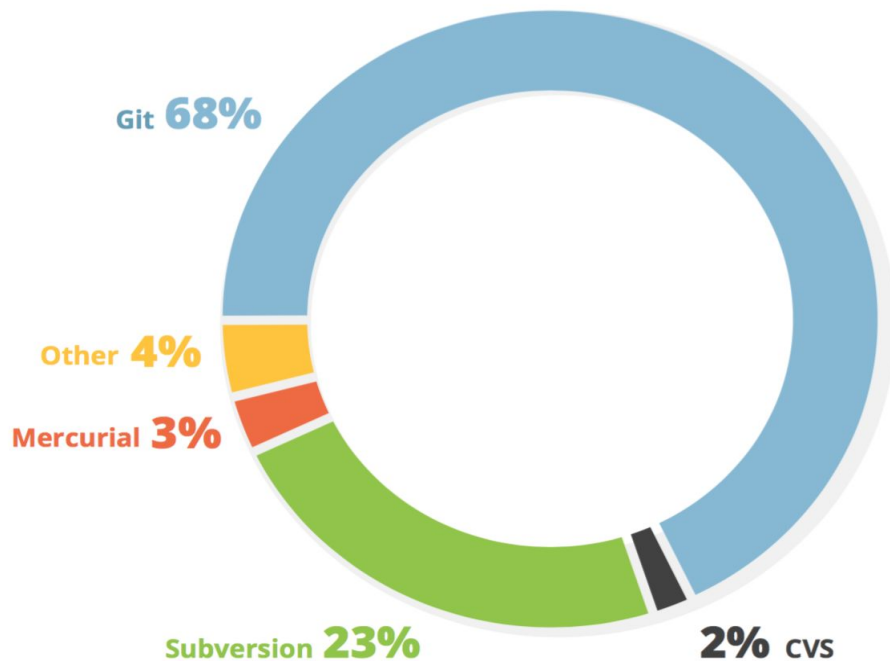
¡NO SON SISTEMAS DE CONTROL DE VERSIONES!

Historial de versiones \Rightarrow temporal (en dropbox los últimos 30),

cambios imprecisos, solo para documentos (no código)

a veces es suficiente

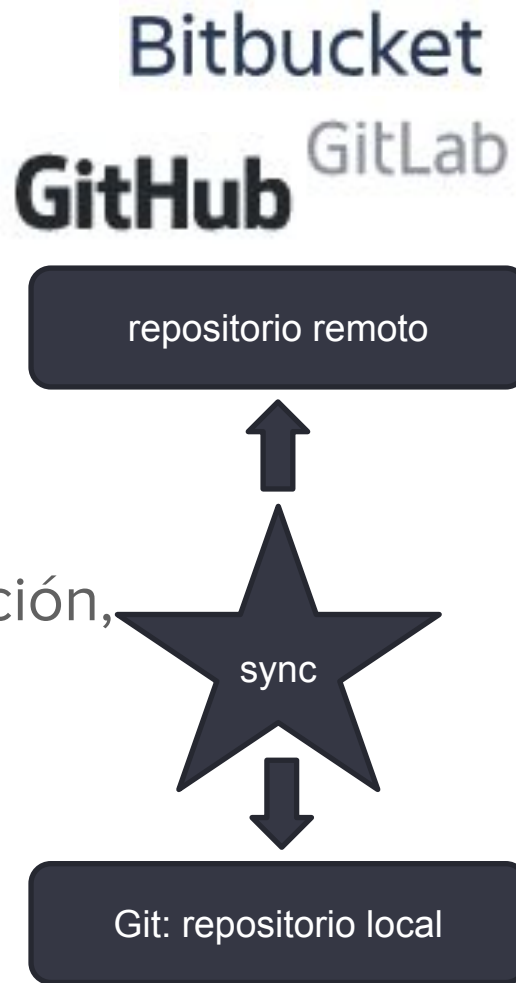
Sistemas de Control de Versiones (SCV) más comunes



SCV



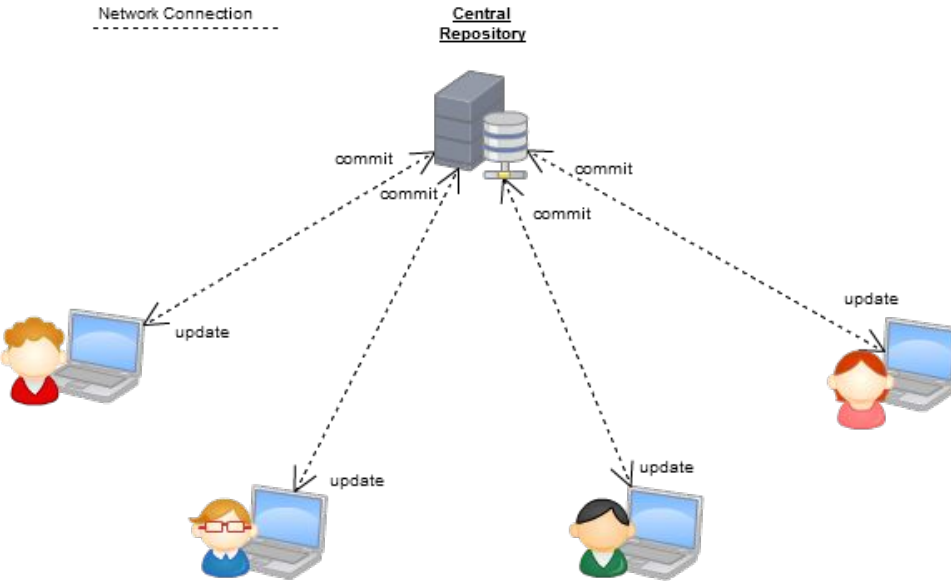
- Documentos
- Proyectos software: lenguajes de programación, conjuntos de datos...
- Más control (commandos específicos).
- Versión de la historia más útil.



Por lo tanto,
¿qué nos proporcionan
los ***sistemas de
control de versiones***
(SCV)?

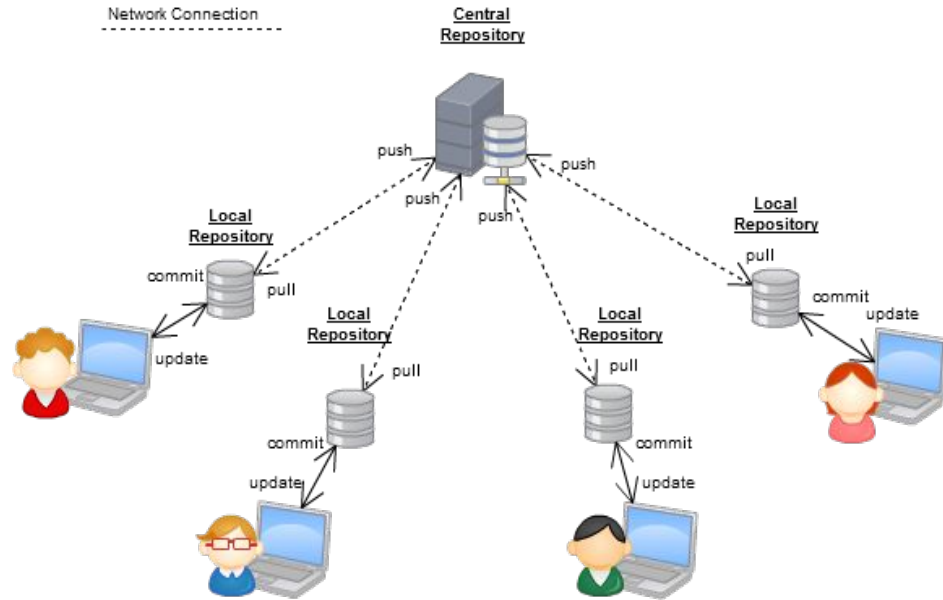
- Gestionar automáticamente los cambios que se realizan sobre uno o varios ficheros de un proyecto.
- Restaurar cada uno de los ficheros de un proyecto a un estado anterior (no solo al inmediatamente anterior).
- Permitir la colaboración de diversos programadores en el desarrollo de un proyecto.

SCV Centralizado



- Repositorio único y centralizado respecto al que se sincronizan los clientes
- Problemas:
 - un único punto de fallo
 - no permite trabajar sin conexión
- Similar a trabajar con Dropbox
 - ¡OJO! Dropbox no es un SCV

SCV Distribuido



- Cada cliente tiene su repositorio local como un copia maestra remota (previniendo la pérdida de datos)
- Puedes trabajar sin conexión (siempre tienes tu repositorio contigo)

Conceptos generales de los SCV I

- **Repositorio**: Es la copia maestra donde se guardan todas las versiones de los archivos de un proyecto. En el caso de git se trata de un directorio. Cada desarrollador tiene su propia copia local de este directorio.
- **Copia de trabajo**: La copia de los ficheros del proyecto que podemos modificar.

Conceptos generales de los SCV II

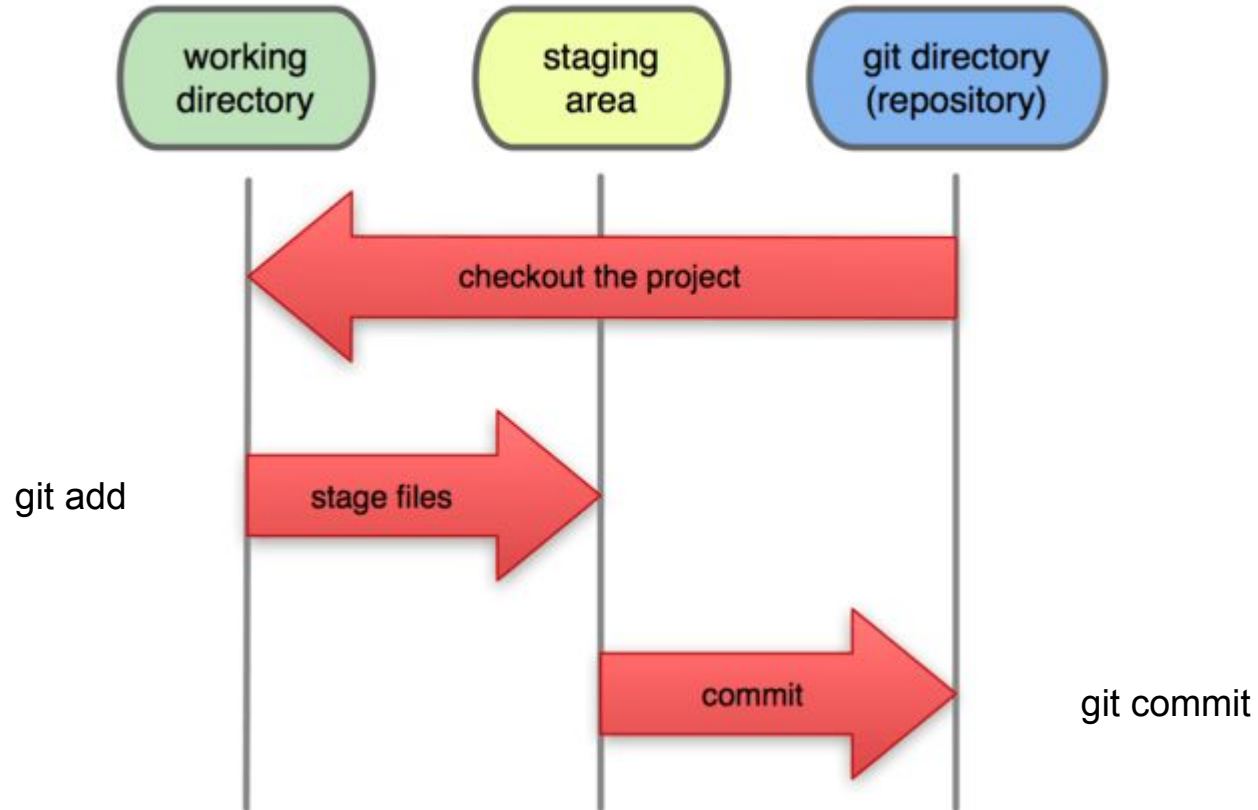
Check Out / Clone: La acción empleada para obtener una copia de trabajo desde el repositorio. En los scv distribuidos -como Git- esta operación se conoce como *clonar* el repositorio porque, además de la copia de trabajo, proporciona a cada programador su copia local del repositorio a partir de la *copia maestra* del mismo.

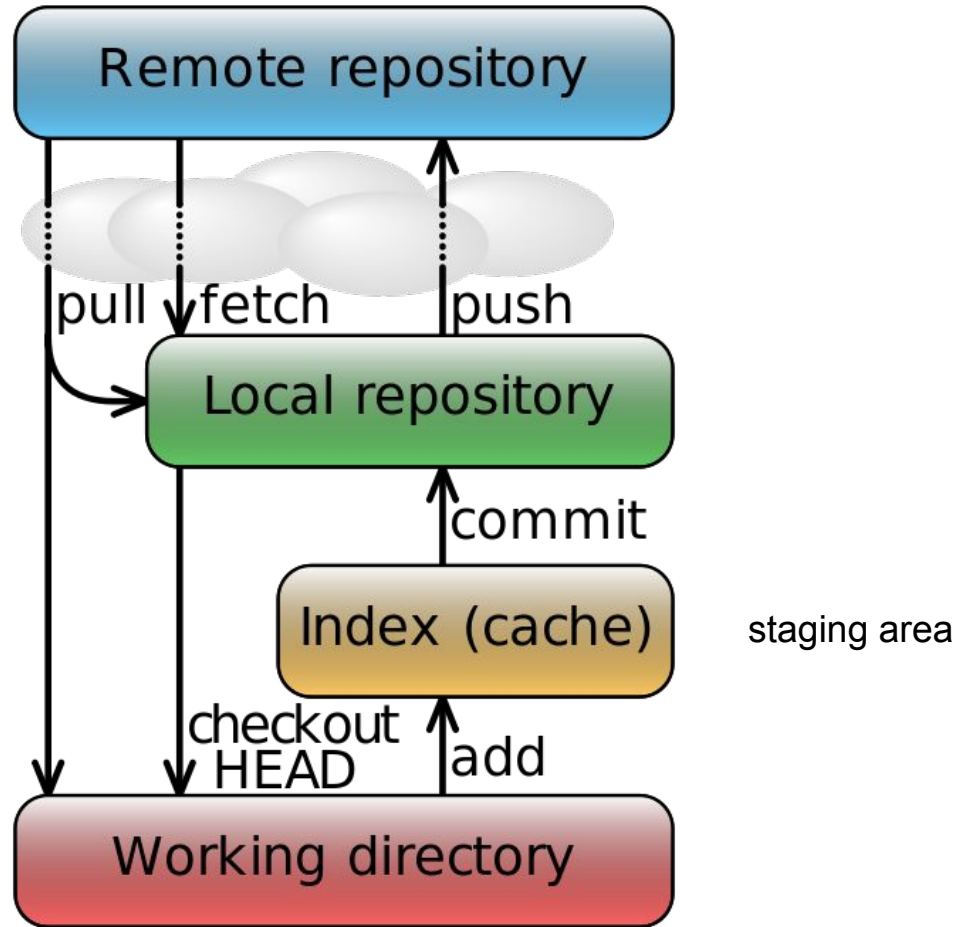
Check In / Commit: La acción empleada para llevar los cambios hechos en la copia de trabajo a la copia local del repositorio. Esto crea una nueva *revisión* de los archivos modificados. Cada commit debe ir acompañado de un “Log Message” el cual es un comentario, una cadena de texto que explica el commit, que añadimos a una revisión cuando hacemos el commit.

Conceptos generales de los SCV III

- **Push**: La acción que traslada los contenidos de la copia local del repositorio de un programador a la copia maestra del mismo.
- **Update/Pull/Fetch+Merge/Rebase**: Acción empleada para actualizar nuestra copia local del repositorio a partir de la copia maestra del mismo, además de actualizar la copia de trabajo con el contenido actual del repositorio local.
- **Conflicto**: Situación que surge cuando dos desarrolladores hacen un *commit* con cambios en la *misma región del mismo fichero*. El **scv** lo detecta, pero es el programador el que debe corregirlo.

Local Operations





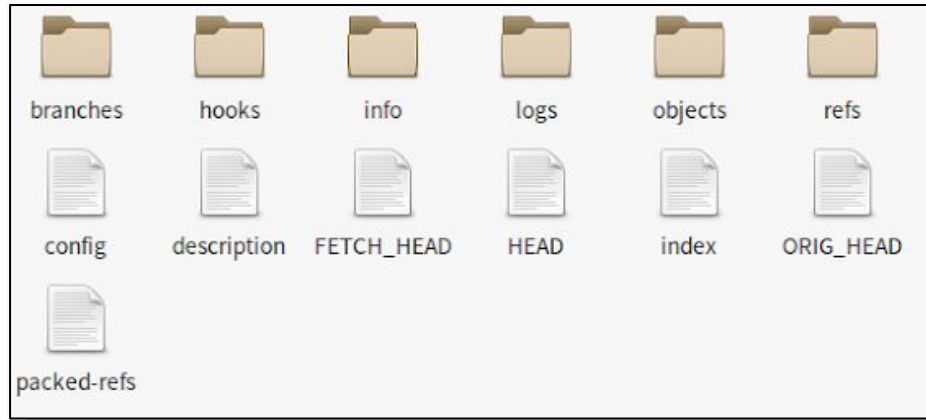
Git: Historia

- Los desarrolladores de *linux* emplean BitKeeper hasta 2005.
- BitKeeper es un **scv** distribuído. Git también lo es, al igual que Darcs, Mercurial, SVK, Bazaar y Monotone.
- Linus comienza el desarrollo de *git* el 3 de abril de 2005, lo anunció el día 6 de abril.
- Git se auto-hospeda el 7 de abril de 2005.
- El primer núcleo de *linux* gestionado con *git* se libera el 16 de junio de 2005, fue el **2.6.12**.

Git: Implementación

- La parte de bajo nivel (plumbing) se puede ver como un sistema de ficheros direccionable por el contenido.
- Por encima incorpora todas las herramientas necesarias que lo convierten en un **scv** más o menos amigable (porcelain).
- Cuenta con aplicaciones escritas en **C** y en **shell-script**. Con el paso del tiempo algunas de estas últimas se han reescrito en **C**.
- Los elementos u objetos en los que *git* almacena su información se identifican por su valor SHA-1.

Git: Directorio **.git**



Uso I

- La orden principal es **git**.
 - Comprobamos qué versión tenemos instalada:

```
> git --version
```

```
git version 2.8.0
```
- Creamos un repositorio, se puede hacer de dos modos:
 1.

```
> mkdir Proyecto; cd Proyecto; git init
```
 2.

```
> mkdir Proyecto; git init Proyecto
```
- Añadimos archivos y guardamos:
 1.

```
> git add .
```
 2.

```
> git status
```
 3.

```
> git commit -m 'Primer commit.' -m "Descripcion detallada."
```
 4.

```
> git commit -a
```

Uso II

- Configuración:
 - archivo “**.git/config**” ⇒ particular del proyecto actual
 1. > **git config user.name "nombre apellidos"**
 2. > **git config user.email "usuario@email.com"**
 - archivo “**~/.gitconfig**” ⇒ general para todos los proyectos del usuario
 3. > **git config --global user.name "nombre apellidos"**
 4. > **git config --global user.email "usuario@email.com"**

Información

- **Git status:**

- muestra el estado del directorio de trabajo y el index/staging area/caché
- estado de los archivos:
 - untracked: no incluidos en el repositorio bajo control de versiones
 - **committed: confirmados** y almacenados en tu copia local del repositorio
 - modified: modificados respecto a la copia local del repositorio
 - (un)staged: (no) preparados para enviar a tu copia local del repositorio

Información II

- **Git log:**
 - muestra el estado del repositorio (confirmaciones - commits - en orden cronológico inverso)
 - -p: diferencias entre confirmaciones (commits)
- **Git show:**
 - muestra el último commit (confirmación) con diferencias detalladas
- **Git diff:**
 - muestra las diferencias entre el directorio de trabajo y el commit (confirmación) más reciente

Etiquetas

- Podemos etiquetar confirmaciones (commits) para marcar puntos importantes
- Por ejemplo, puntos donde se crea una versión nueva (v1.0, v1.1, etc.)

- `> git tag tagname commit` //crear etiqueta
- `> git tag -a -m "mensaje" tagname commit` //etiqueta anotada
- `> git tag -l` //lista todas las etiquetas
- `> git tag -d tagname` //elimina una etiqueta
- `> git show tagname`
//muestra la información de la etiqueta y el objeto de referencia

Descartar cambios

1. Si por error has subido al repositorio algún archivo o directorio que no debería estar, puedes eliminarlo con el siguiente comando, que mantiene la copia local en tu ordenador de ese archivo o directorio:

```
git rm --cached nombre-del-archivo-o-directorio
```

2. O puedes deshacer cambios de archivos con seguimiento desde la última confirmación (el último git commit)

```
git reset --hard
```

3. O puedes traer cambios de archivos y ramas específicas

```
git checkout path-to-file or branch
```

Cómo organizar un repositorio - ¿Qué incluir?

- Archivos de código
- Archivos que permitan construir y ejecutar la solución correctamente (archivos .sln y .csproj).
- Archivos de configuración y propiedades.
 - Prestar especial atención a **NO poner rutas absolutas** en estos ficheros
- Se pueden incluir los scripts de la creación de la base de datos y tablas.

Cómo organizar un repositorio - ¿Qué NO incluir? I

- No incluyas archivos que se puedan generar a partir de otros ya incluidos. Estos archivos incluyen binarios, ejecutables (.exe), etc.
 - En los proyectos de visual studio, este tipo de archivos se almacenan en las carpetas bin y obj
- No incluyas carpetas con resultados de compilación (ej. Debug, release...)
- No incluyas la BBDD

Cómo organizar un repositorio - ¿Qué NO incluir? II

- Archivos binarios grandes
 - Grande con respecto a la cantidad de memoria RAM libre disponible
 - Evita archivos de más de 1GB
- Evita crear repositorios muy grandes (cuando sea posible)
 - Estos repositorios ralentizan muchas operaciones de git por la cantidad de archivos y por el tamaño de los mismos
 - Grande depende del tamaño de la RAM y otros factores.
- Más consejos en:
<https://sethrobertson.github.io/GitBestPractices/#misc>
<https://sethrobertson.github.io/GitBestPractices/#donot>

.gitignore

- Se puede definir en el archivo .gitignore los ficheros y carpetas que queremos NO incluir (y por tanto no rastrear) en el repositorio
- Ejemplo de .gitignore

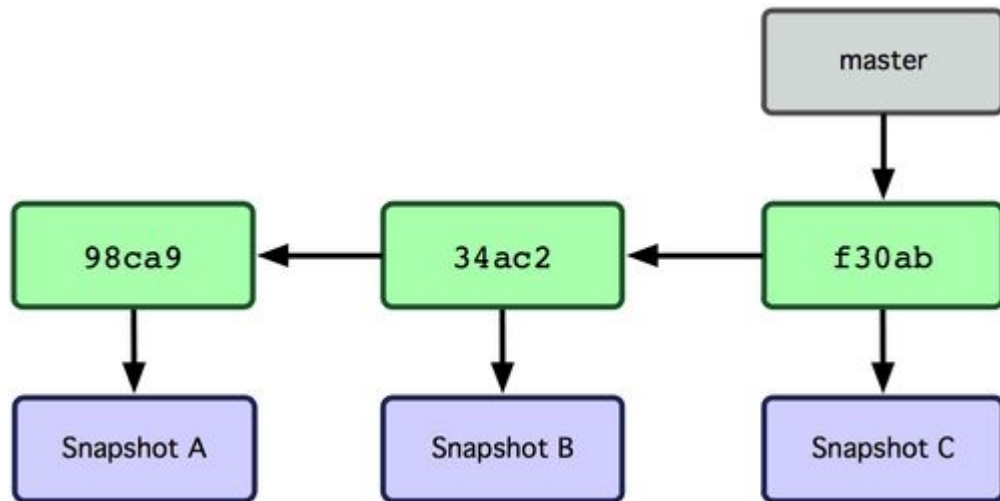
```
#archivos de usuario
.suo
...
#carpeta Debug
[Dd]ebug/

#archivo de BBDD
*.mdf
```

<https://github.com/github/gitignore/blob/main/VisualStudio.gitignore>

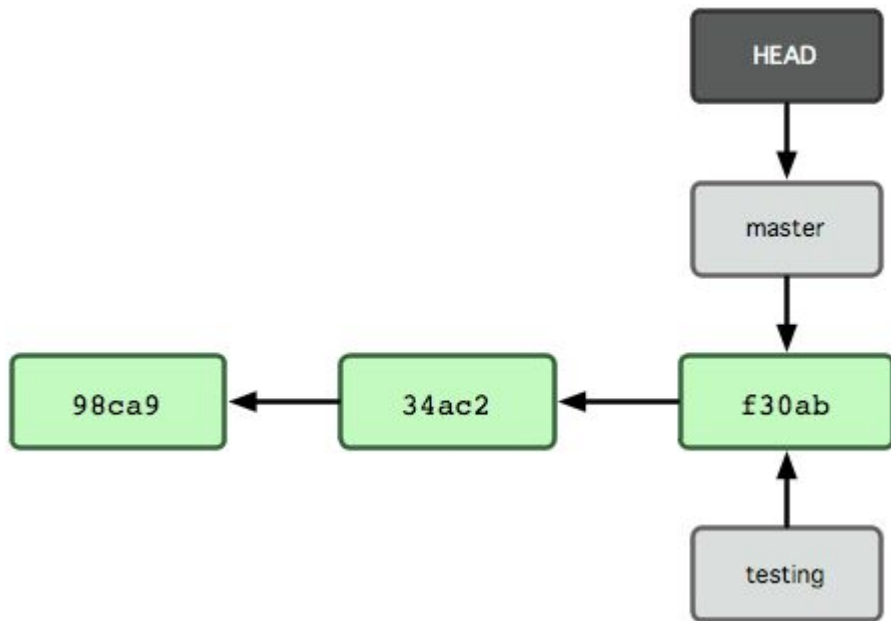
Ramas I

La rama por defecto se llama **master**. Con los primeros cambios confirmados, la rama principal **master** se creará apuntado a esta confirmación (=commit). Por cada commit que hagamos, la rama avanzará automáticamente. La rama **master** siempre apuntará la última confirmación (commit) realizado.



Ramas II

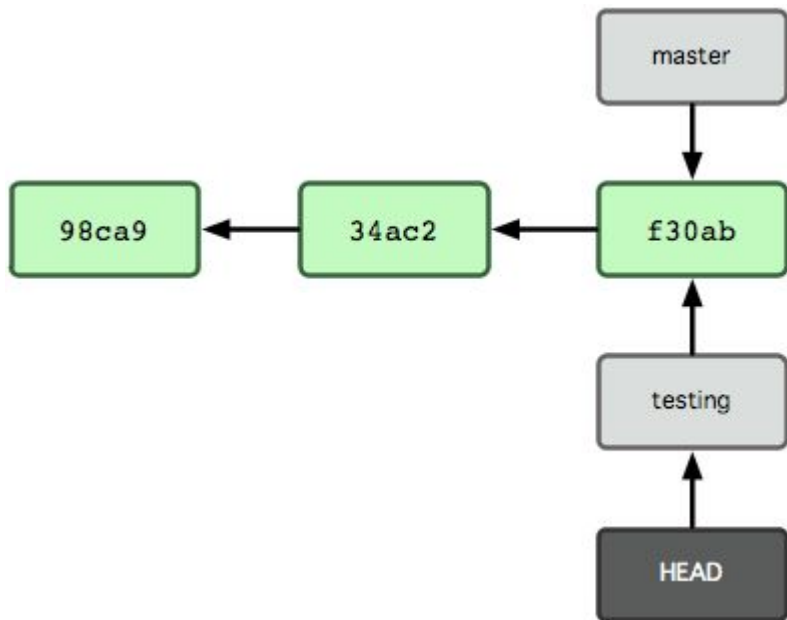
- **git branch testing:** Este comando creará una nueva rama llamada **testing** con un puntero en a la misma confirmación (commit) donde estés actualmente.



¿Cómo sabe Git en qué rama estás en este momento?
Mediante un puntero especial llamado HEAD.

Ramas III

- **git checkout testing:** para saltar de una rama a otra

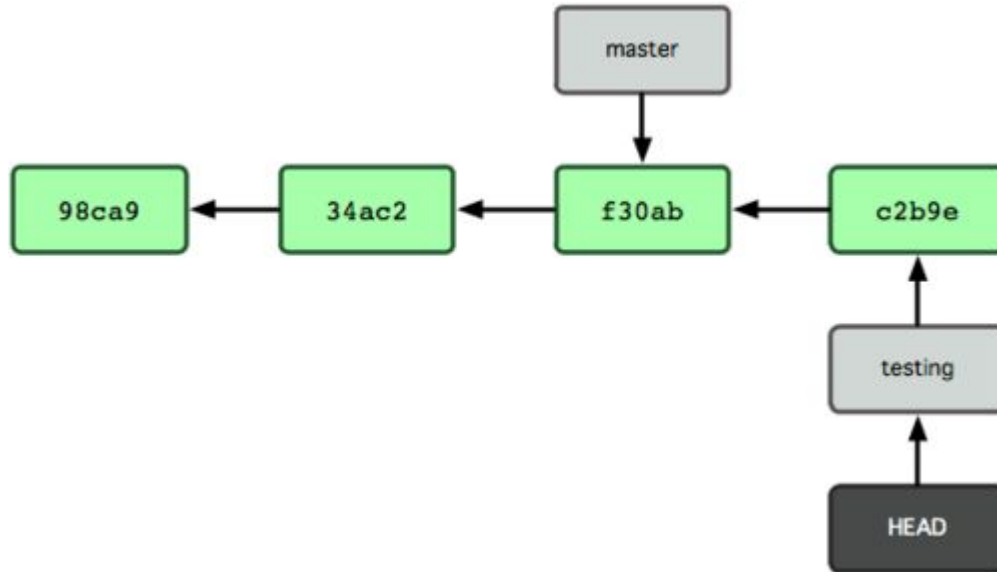


Ahora hemos cambiado a la rama testing, que acabamos de crear.

Por lo tanto, el puntero HEAD apunta a la rama testing.

Ramas IV

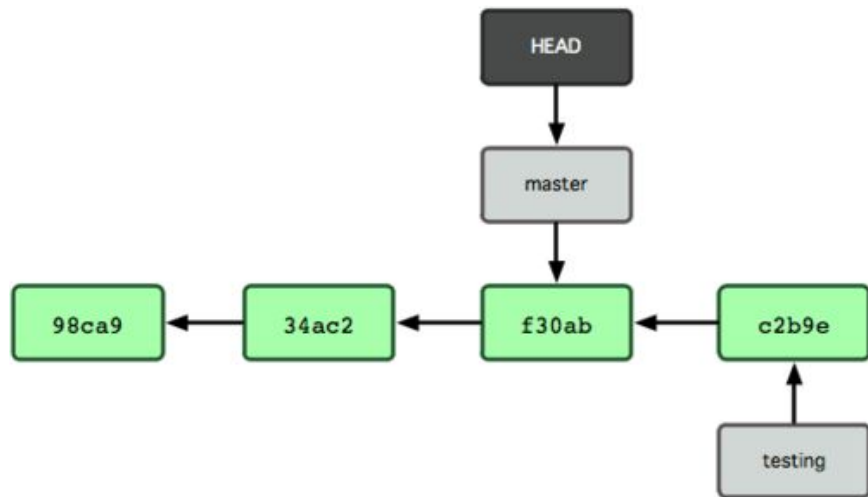
- Si hacemos modificaciones en la rama testing



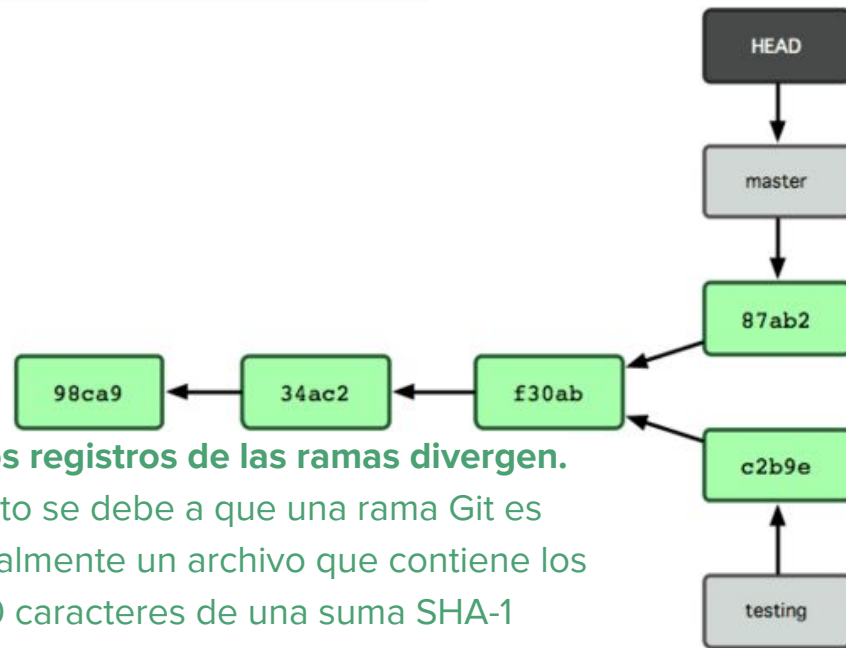
La rama testing avanza
La rama master se mantiene en
el mismo lugar ⇒ confirmación
(commit)

Ramas V

- Cambiamos a master



y modificamos

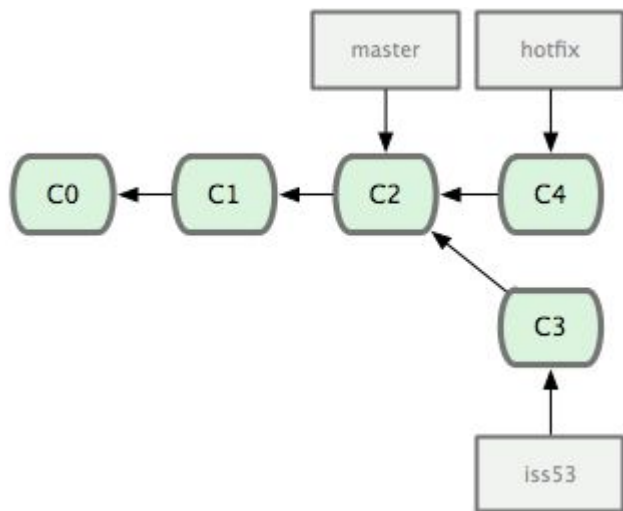


Los registros de las ramas divergen.

Esto se debe a que una rama Git es realmente un archivo que contiene los 40 caracteres de una suma SHA-1 (representado la confirmación de cambios a la que apunta), por eso no cuesta nada crear y eliminar ramas en Git.

Ramas VI

- **git checkout master**
- **git merge hotfix**

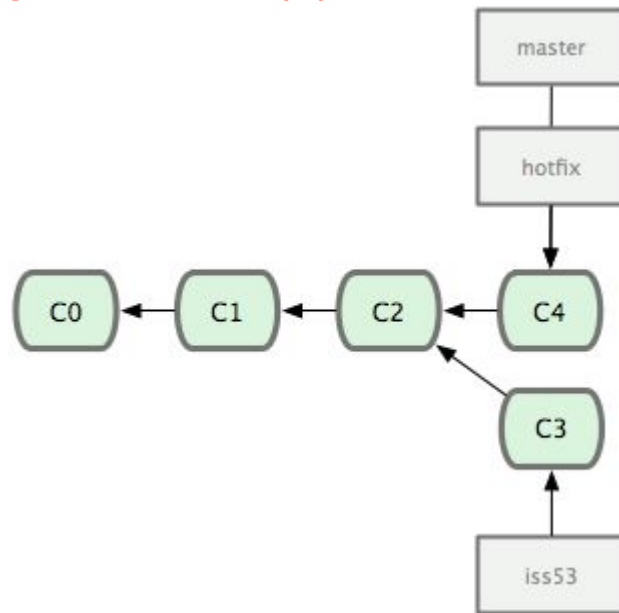


Updating f42c576..3a0874c

Fast forward

README | 1 -

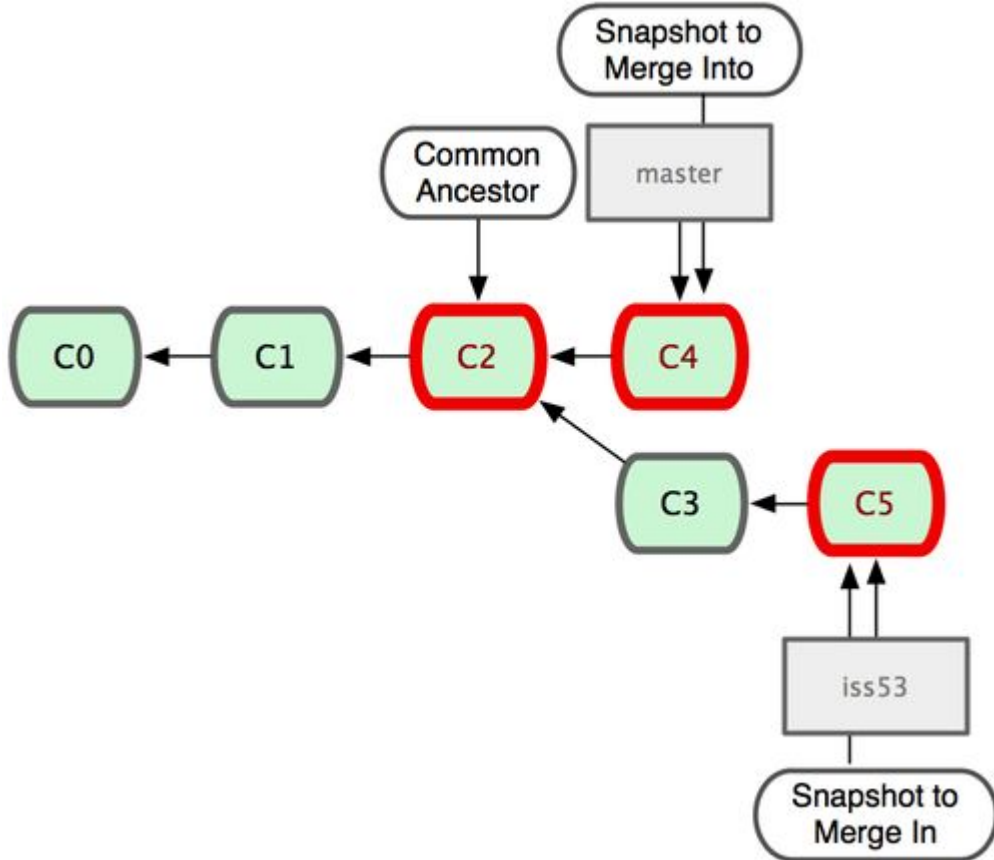
1 files changed, 0 insertions(+), 1 deletions(-)



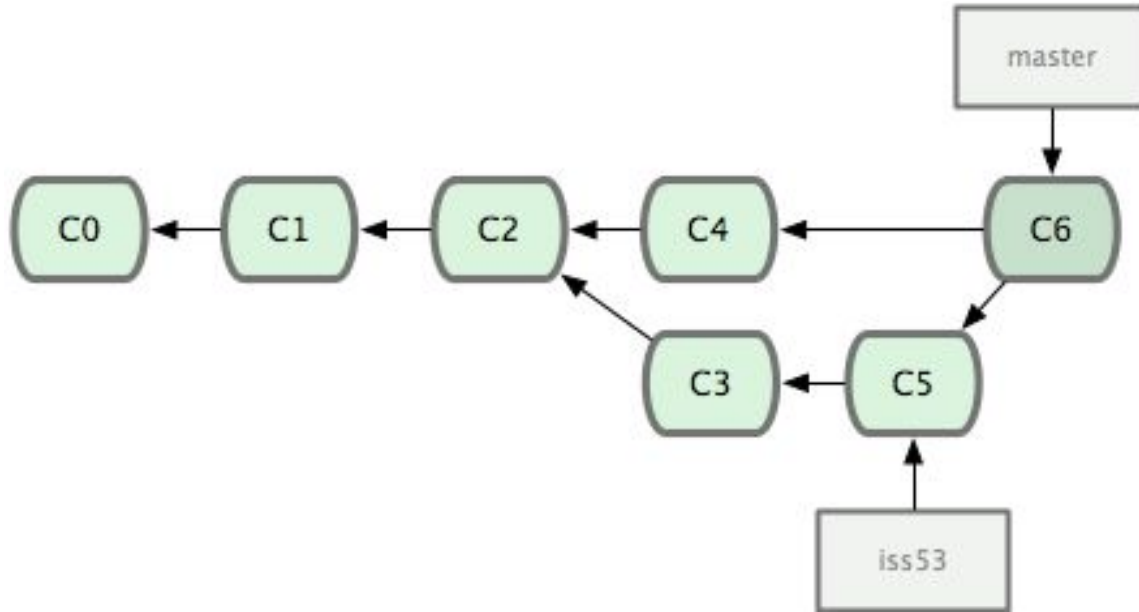
Git ha movido el puntero hacia delante, ya que la confirmación de la rama master (commit C2) apuntaba a la siguiente confirmación (commit C4, rama hotfix)

Ramas VII

Git automáticamente identifica el ancestro común para realizar la fusión de ramas



Ramas VIII



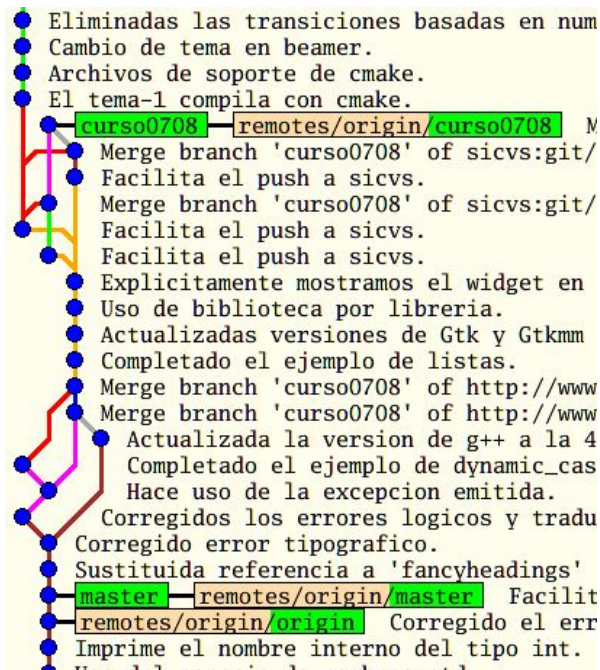
Ahora que todo tu trabajo está ya fusionado con la rama principal, ya no necesitas la rama iss53. Por lo que puedes borrarla.

En lugar de simplemente avanzar el puntero de la rama, **Git crea una nueva instantánea (snapshot) resultado de la fusión a tres bandas; y automáticamente crea una nueva confirmación de cambios (commit) que apunta a ella.** Nos referimos a este proceso como "fusión confirmada". Y se diferencia en que tiene más de un padre.

Ramas IX

- Comandos

1. `> git branch [-a] [-r]`
2. `> git show-branch`
3. `> git checkout [-b] [rama-de-partida]`
4. `> gitk --all`



Ramas X

- Repositorios remotos
 1. > `git remote add nombre protocolo`
 2. > `git remote add origin IP:ruta/hasta/repo`
 3. > `git clone IP:ruta/hasta/repo`
- Operaciones con repositorios remotos
 1. > `git pull [origin] [rama]`
 2. > `git push [repo] [rama]`
 3. > `git checkout -b rama origin/rama-remota`
 4. > `git fetch`
 5. > `git merge`
 6. > `git pull = git fetch + git merge`
 7. > `git rebase otra-rama`

Uso V

- Stash

1. `> git stash [list | show | drop | ...]`

Echa un vistazo a este tutorial sobre [git stash](#).

- Bisect

1. `> git bisect [help | start | bad | good | ...]`

Echa un vistazo a este tutorial sobre [git bisect](#).

- Herramientas gráficas

1. **gitk**
2. **git gui**
3. **git view**
4. **gitg**
5. **giggle**
6. **[gource](#)**

- Cualquier IDE o editor actual (Atom, Sublime, VisualStudio, VisualStudioCode, etc...) dispone de un plug-in para git.

Interacción con otros SCVs

- Git puede interactuar con otros scvs. Además puede hacerlo en ambos sentidos (recuperar y guardar información).
- Por ejemplo, en los casos de CVS y Subversion echa un vistazo a:
 1. > `git cvsimport --help`
 2. > `git svn --help`

Casos de uso I

- ¿Cómo creo una rama local que *siga* los cambios en una remota al hacer ``pull'`?
 - `git branch --track ramalocal origin/ramaremota`
- ¿Se puede crear una rama que no parta del último commit de otra?...**sí**
 - `git branch --no-track feature3 HEAD~4`
- ¿Quién hizo *qué commit* en un fichero del proyecto?:
 - `git blame fichero`
- ¿Cómo creo una rama para resolver un bug y lo integro de nuevo en la rama principal?:
 - `git checkout -b fixes`
hack...hack...hack
 - `git commit -a -m "Crashing bug solved."`
 - `git checkout master`
 - `git merge fixes`

Casos de uso II

- He modificado localmente el fichero “src/main.cs” y no me gustan los cambios hechos. ¿Cómo lo devuelvo a la última versión bajo control de versiones?:
 - `git checkout -- src/main.cs`
- ¿Y un directorio completo, p.e. a la *penúltima* versión de la rama “test”?:
 - `git checkout test~1 -- src/`
- ¿Y si he modificado varios ficheros y quiero dejar todo como estaba antes de la modificación?...tenemos varias maneras:
 1. `git checkout -f`

o también:

 2. `git reset --HARD`

Casos de uso III

- ¿Se puede deshacer un `commit` que es una mezcla (*merge*) de varios “commits”?...**sí**, hay que elegir cuál o cuáles de los commits que forman la mezcla así:
 1. **git revert HEAD~1 -m 1**
En este ejemplo estaríamos deshaciendo sólo el primero de los “commits” que formaban este “merge”.
- ¿Cómo puedo obtener un archivo tal y como se encontraba en una versión determinada del proyecto?... de varias maneras:
 1. **git show HEAD~4:index.html > oldIndex.html**
o también así:
 2. **git checkout HEAD~4 -- index.html**

Casos de uso IV

- ¿De qué maneras distintas puedo ver los cambios que ha habido en el repositorio?:
 1. `git diff`
 2. `git log --stat`
 3. `git whatchanged`
- ¿Cómo puedo saber cuántos commits ha hecho cada miembro del proyecto en la rama actual?:
 1. `git shortlog -s -n`
- ¿Y en todas las ramas?:
 1. `git shortlog -s -n --all`
- ¿Cómo puedo corregir el mensaje de explicación del último commit que he hecho?:
 1. `git commit --amend`
Abre el editor por defecto y nos permite modificarlo.

Ejercicio sencillo

Lo mejor es que pruebes git con algún código tuyo, p.e. de alguna práctica de otra asignatura que ya tengas hecha. Sigue estos pasos:

1. Elige un directorio que contenga el código de esa práctica. Cambiate a él.
2. Inicia el repositorio en este directorio.
3. Añade los archivos que haya previamente en él.
4. Haz el primer “*commit*” de los archivos recién importados.
5. Haz una modificación a uno o varios de ellos. Comprueba cuáles han cambiado, cómo lo han hecho. Añádelos al siguiente commit.
6. Contribuye los cambios creando el “commit”.
7. Crea una rama en el proyecto y cámbiate a ella automáticamente (mira las opciones de *checkout*).
8. Haz cambios y commits en esta rama.
9. Vuelve a la rama “master”.

Repositorios remotos para git

- Existen varios servicios de repositorio remoto para SCV Git
- Permiten sincronizar el repositorio local con el remoto
- Proveen una interfaz web que trabaja sobre git
- Son una plataforma social para compartir trabajo y conocimiento
- En esta asignatura nos centraremos en GitHub



Github I

- Github es una plataforma de desarrollo colaborativo para alojar proyectos software haciendo uso del SCV Git.
- Podemos crear proyectos en ella y almacenarlos de manera gratuita si son:
 - **públicos** ⇒ son visibles para todos y *todo el mundo puede clonarlos*
 - **privados** ⇒ son visibles solo para tí y para los colaboradores que tú elijas, pero únicamente haciendo uso de una *cuenta de pago*.

Github II

- Desde hace un tiempo Github permite hacer uso gratuito de ciertos recursos que normalmente forman parte de la *cuenta de pago*, p.e. poder crear proyectos “privados”.
- Sólo necesitas una cuenta de email institucional de la UA (@alu.ua.es) asociada a la cuenta de Github empleada y darte de alta en [Github-Education](#).

Conflictos I

- Los SCV permiten gestionar las contribuciones uno o varios desarrolladores
- Si los desarrolladores editan el mismo contenido, pueden ocurrir conflictos
 - Cuando trabajas solo, los conflictos pueden aparecer al trabajar en varios ordenadores.
 - Cuando trabajas con varias personas, los conflictos pueden surgir al editar el mismo archivo.
- Para minimizar la aparición de conflictos, cada desarrollador suele trabajar en una rama separada
 - Por eso combinar ramas y lidiar con conflictos son tareas comunes de git

Conflictos II

- La mayor parte del tiempo, el comando `git merge` sabe cómo integrar los cambios entre ramas de forma automática (y sencilla para tí)
- Pero si surge un conflicto, git no sabe automáticamente qué cambio elegir
- El comando `git merge` marcará el archivo como conflictivo e interrumpirá el proceso. Es tu responsabilidad como desarrollador resolver ese conflicto.
- Los conflictos se pueden producir al inicio de o durante el proceso de combinar ramas (`git merge`).

Resolver conflictos I

- Git falla al inicio del proceso de combinar ramas (merge)
 - El fallo se produce cuando Git ve diferencias entre el directorio de trabajo y el staging area del proyecto.
 - No se pueden combinar las ramas porque esos cambios pendientes podrían ser sobrescritos por los commits que se van a combinar.
 - Cuando esto pasa, no hay conflictos con otros desarrolladores, si no conflictos con cambios locales pendientes.
 - Para resolverlo deberás usar: git stash, git checkout, git commit or git reset.
 - Este fallo produce mensaje de un error similar a:

```
error: Entry '<fileName>' not uptodate. Cannot merge.  
(Changes in working directory)
```

Resolver conflictos II

- Git falla durante el proceso de combinar ramas (merge)
 - En este caso el conflicto se produce entre la rama local y la rama remota a combinar
 - Indica un conflicto con el código remoto
 - Son los más habituales
 - Git tratará de combinar ambas ramas automáticamente pero necesitará que tú resuelvas los archivos conflictivos
 - Para resolverlo, deberás usar los comandos:
 - `git merge --abort`: saldrá del proceso de combinación y volverá a la rama al estado anterior, como si nada hubiera pasado
 - `git reset`: resetea los archivos conflictivos
 - Este fallo produce mensaje de un error similar a:

`error: Entry '<fileName>' would be overwritten by merge.`

Ejemplo de conflicto I

- Creamos un nuevo repositorio con una rama master y un archivo merge.txt

```
$ mkdir git-merge-test
$ cd git-merge-test
$ git init .
$ echo "this is some content to mess with" > merge.txt
$ git add merge.txt
$ git commit -m "we are committing the initial content"
[master (root-commit) d48e74c] we are committing the initial content
1 file changed, 1 insertion(+)
create mode 100644 merge.txt
```

Ejemplo de conflicto II

- Creamos una nueva rama conflictiva: new_branch_to_merge_later
- Sobreescribimos el contenido del archivo merge.txt
- Confirmamos los cambios

```
$ git checkout -b new_branch_to_merge_later
$ echo "totally different content to merge later" > merge.txt
$ git add merge.txt
$ git commit -m"edited the content of merge.txt to cause a conflict"
[new_branch_to_merge_later 6282319] edited the content of merge.txt to cause a
conflict
1 file changed, 1 insertion(+), 1 deletion(-)
```

Ejemplo de conflicto III

- Cambiamos a la rama master, añadimos nuevo contenido al archivo txt y hacemos commit.

```
$ git checkout master
Switched to branch 'master'
$ echo "content to append" >> merge.txt
$ git add merge.txt
$ git commit -m"appended content to merge.txt"
[master 24fbe3c] appended content to merge.tx
1 file changed, 1 insertion(+)
```

- ¿Qué pasa si ejecutamos `$ git merge new_branch_to_merge_later` ?

-  **Tenemos un conflicto**

Auto-merging merge.txt

CONFLICT (content): Merge conflict in merge.txt

Automatic merge failed; fix conflicts and then commit the result.

Resolución de conflicto I

Git ofrece un mensaje descriptivo para avisar del conflicto ocurrido y cómo solucionarlo. Podemos obtener más información con el comando:

```
$ git status
On branch master
You have unmerged paths.
(fix conflicts and run "git commit")
(use "git merge --abort" to abort the merge)
Unmerged paths:
(use "git add <file>..." to mark resolution)
both modified:   merge.txt
```

El proceso de combinación no ha finalizado por el conflicto.

Resolución de conflicto II

El archivo conflictivo ha modificado su estado

```
$ cat merge.txt
```

```
<<<<<< HEAD
```

```
this is some content to mess with  
content to append
```

```
=====
```

```
totally different content to merge later
```

```
>>>>>> new_branch_to_merge_later
```

} rama actual ⇒ master

} rama a combinar ⇒ new_branch_to_merge_later

Resolución de conflicto III

Para resolver el conflicto, simplemente modifica el archivo

```
$ cat merge.txt
this is some content to mess with
content to append
totally different content to merge later
```

} rama actual ⇒ master
conflicto resuelto

Incluye estos cambios en el staging area:

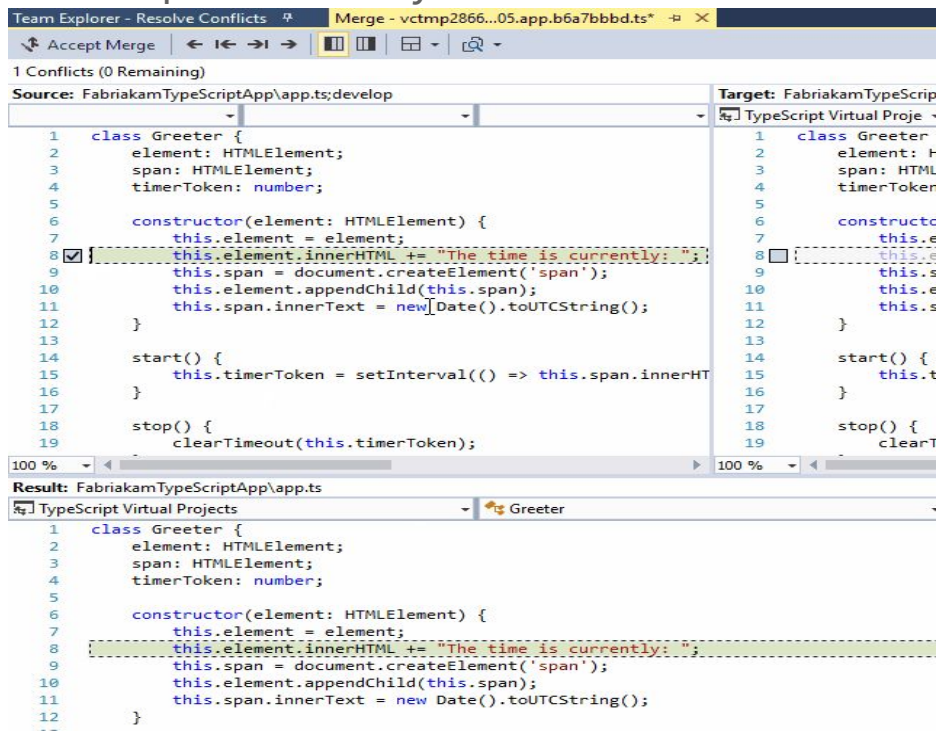
```
$git add merge.txt
$git commit -m "merged and resolved the conflict in merge.txt"
```

¿Qué pasa si ejecutamos `$ git merge new_branch_to_merge_later` ?

- Ya no es necesario, ahora no tenemos conflicto y la rama está actualizada
Already up to date.
- Ahora podríamos enviarlo al repositorio remoto

Resolución de conflicto IV

Los conflictos también los puedes ver y resolver en IDEs como VisualStudio



The screenshot shows the Visual Studio Team Explorer interface for resolving a conflict in the file `vctmp2866...05.app.b6a7bbbd.ts`. The interface is divided into three main sections: Source, Target, and Result.

Source: FabrikamTypeScriptApp\app.ts;develop

```
1 class Greeter {
2   element: HTMLElement;
3   span: HTMLElement;
4   timerToken: number;
5
6   constructor(element: HTMLElement) {
7     this.element = element;
8     this.element.innerHTML += "The time is currently: ";
9     this.span = document.createElement('span');
10    this.element.appendChild(this.span);
11    this.span.innerText = new Date().toUTCString();
12  }
13
14  start() {
15    this.timerToken = setInterval(() => this.span.innerHT
16  }
17
18  stop() {
19    clearTimeout(this.timerToken);
20  }
```

Target: FabrikamTypeScriptApp\app.ts;develop

```
1 class Greeter {
2   element: HTMLElement;
3   span: HTMLElement;
4   timerToken: number;
5
6   constructor(element: HTMLElement) {
7     this.element = element;
8     this.element.innerHTML += "The time is currently: ";
9     this.span = document.createElement('span');
10    this.element.appendChild(this.span);
11    this.span.innerText = new Date().toUTCString();
12  }
13
14  start() {
15    this.timerToken = setInterval(() => this.span.innerHT
16  }
17
18  stop() {
19    clearTimeout(this.timerToken);
20  }
```

Result: FabrikamTypeScriptApp\app.ts

```
1 class Greeter {
2   element: HTMLElement;
3   span: HTMLElement;
4   timerToken: number;
5
6   constructor(element: HTMLElement) {
7     this.element = element;
8     this.element.innerHTML += "The time is currently: ";
9     this.span = document.createElement('span');
10    this.element.appendChild(this.span);
11    this.span.innerText = new Date().toUTCString();
12  }
```

Flujo de trabajo en Git I

Un flujo de trabajo git es un protocolo que se debe seguir para trabajar de forma consistente y productiva

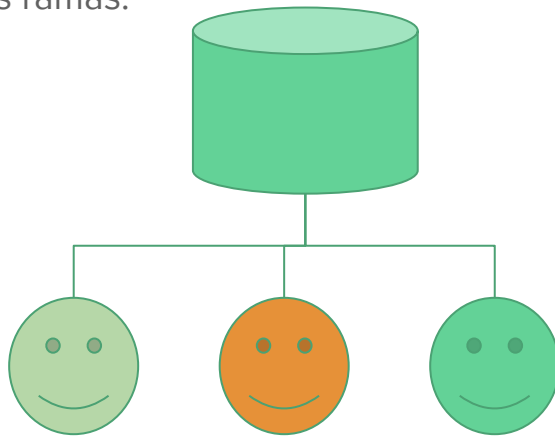
Git proporciona mucha flexibilidad para trabajar y no hay un proceso estandarizado al que atenerse. Es recomendable que cada organización diseñe el suyo.

Sin embargo, existen diversos flujos disponibles públicamente de los que partir como:

- El más simple: Centralized Workflow
- El más usado en Github: Gitflow Workflow

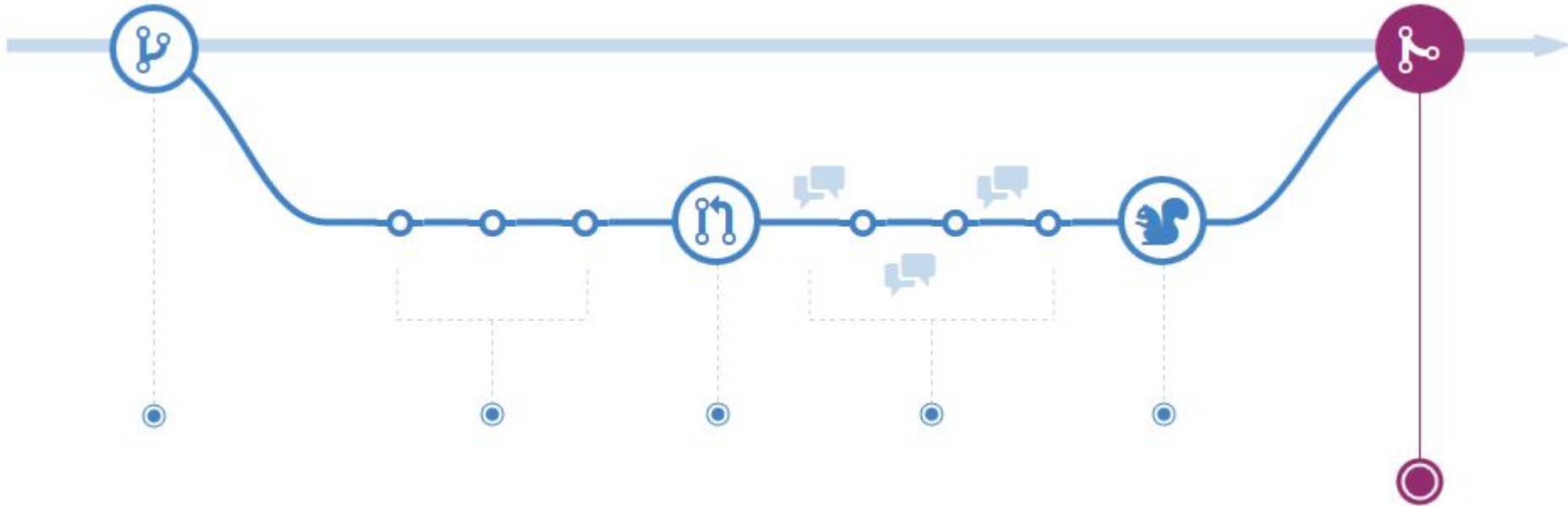
Flujo de trabajo en Git II

- Centralized Workflow:
 - Similar al protocolo usado en SCV centralizados.
 - Usa un repositorio central como punto de entrada único para todos los cambios del proyecto.
 - Todos los cambios se incluyen en la rama master.
 - No se necesitan otras ramas.



Flujo de trabajo en Git III

- Gitflow Workflow:



Flujo de trabajo en Git III

- Gitflow Workflow:

1. **Crea una rama**

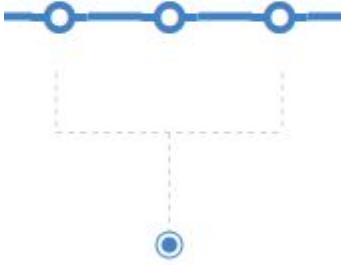
- Crear una rama para cada nueva idea a implementar.
- Los cambios de esta nueva rama no afectan a la rama master
- Puedes experimentar y confirmar cambios (commit), sabiendo que la nueva rama no se combinará hasta que esté lista



Flujo de trabajo en Git III

- Gitflow Workflow:

2. Añadir commits

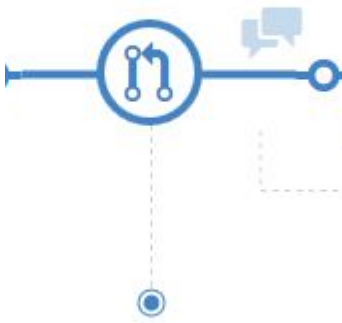


- Cuando la rama está creada, puedes hacer cambios.
- Cada vez que añadas, edites o elimines un archivo, debes hacer un commit.
- Así registras tu progreso sobre una nueva funcionalidad en esta rama y para tus compañeros es transparente por qué haces cada cosa.
- Cada commit tiene un mensaje que describe un único cambio para permitir volver atrás fácilmente, si por ejemplo encuentras un bug

Flujo de trabajo en Git III

- Gitflow Workflow:

3. Abrir una Pull Request



- Los Pull Requests iniciar una discusión sobre tus commits ya que todo el mundo puede ver tus cambios exactos, que serán aceptados si aceptan la petición de combinación (merge)
- Se puede abrir un Pull Request en cualquier momento del desarrollo para iniciar una conversación
- Son útiles para contribuir en proyectos de código abierto

Flujo de trabajo en Git III

- Gitflow Workflow:

4. Discute y revisa tu código



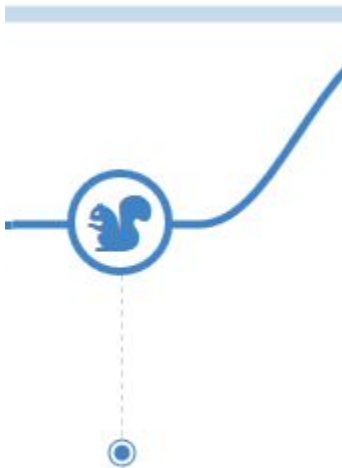
- El encargado de revisar tus cambios puede hacer preguntas o comentarios sobre tus cambios hasta que todo esté en orden (no faltan test, el código no tiene errores, etc.).
- Puedes continuar subiendo cambios a tu rama según los comentarios sobre tus commits
- GitHub muestra tus nuevos commits y cualquier comentario adicional.

Flujo de trabajo en Git III

- Gitflow Workflow:

5. Despliegue

- Github te permite desplegar tu rama en el sistema de producción antes de combina la nueva rama con master.
- Si tu rama causa problemas en producción, puedes volver atrás y desplegar la rama máster en producción.



Flujo de trabajo en Git III

- Gitflow Workflow:



6. Combina (Merge)

- Cuando tus cambios funcionan en producción, es el momento de combinar la rama master con la nueva rama.
- Una vez combinadas, cada Pull Request preserva los cambios históricos de tu código de manera que aunque pase el tiempo, cualquiera puede consultarlos y entender por qué se tomaron las decisión.

Puedes ampliar información sobre este proceso en:

<https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow>

<https://nvie.com/posts/a-successful-git-branching-model/>

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

Flujo de trabajo en Git IV

- HADA Workflow:
 - Para la práctica en grupo:
 - la asignatura tiene su propio flujo de trabajo para que varios desarrolladores colaboren
 - diferente según si eres el responsable del grupo o no

Webs de interés

1. [git](#)
2. [git guide](#)
3. [Carl's Worth tutorial](#)
4. [git-for-computer-scientists](#)
5. [gitmagic](#)
6. [freedesktop](#)
7. [gitready](#)
8. [progit](#)
9. [winehq](#)
10. Presentación en [vídeo de git](#) hecha por *Linus Torvalds*
11. Vídeo de [uso de git/github desde Windows](#) con el intérprete de órdenes y con [Visual Studio Code](#) y Visual Studio 2015.