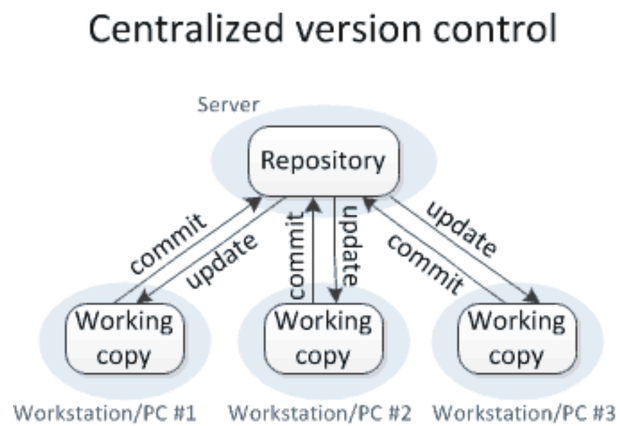
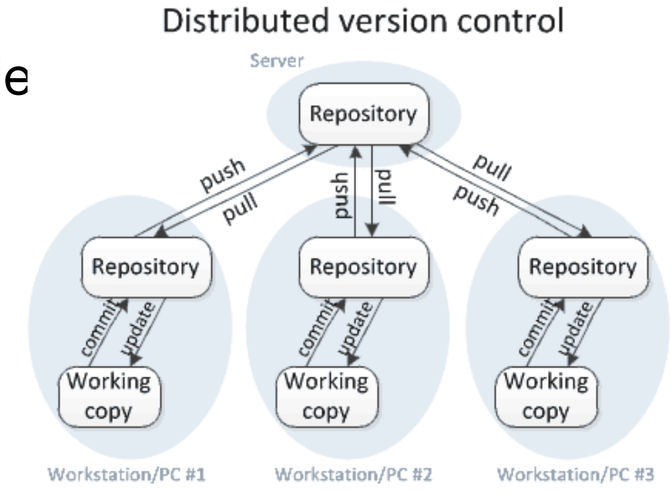
MADS
S10: Flujos de trabajo Git
(Bloque 4 - Entrega continua)

Sistema de Control de Versiones Distribuido - Git

- El primer elemento de un sistema de *continuous delivery* es un sistema de control de versiones
- Git es un **Sistema de Control de Versiones Distribuido** (DVCS en inglés)
- Permite clonar repositorios en distintas máquinas, hacer commits en la versión clonada y publicar los cambios, sincronizando los commits
- Normalmente se define un **repositorio remoto compartido** entre todo el equipo y repositorios locales de cada desarrollador
- Una de las características principales de Git es la facilidad de
- Gran variedad de posibles **flujos de trabajo**





Algunas características de los DVCS (y Git)

- Posibilidad de recuperar cualquier versión previa
  - Un sistema de control de versiones permite guardar todo el código fuente de un proyecto, junto con una historia de los cambios introducidos en su desarrollo
  - Para cada cambio es posible consultar mucha información asociada:
    - Quién ha realizado el cambio
    - Qué ficheros se han modificado
    - Qué líneas de código se han introducido, borrado y modificado
    - Es posible volver a cualquier commit previo para recuperar cualquier fichero o la versión completa del proyecto
- Trabajo independiente que después se integra (o no)
  - Es definir ramas, puntos en los que la historia del proyecto se divide en dos
  - Ramas de corto recorrido (***short-lived branch***): trabajo independiente que se termina integrando en la rama de la que surgió
  - Ramas de largo recorrido (***long-lived branch***): ramas que se mantienen para siempre en el repositorio remoto y que permiten mantener distintas versiones actuales del mismo repositorio (por ejemplo, una versión con el desarrollo actual y otra con la versión en producción)

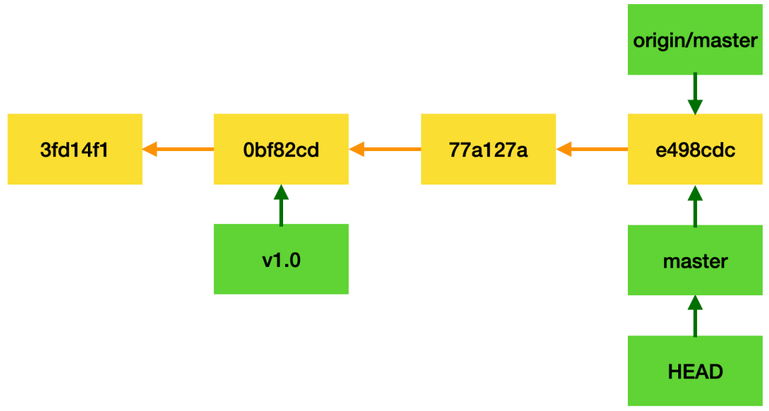
Disclaimer

En las figuras de estas diapositivas aparece como nombre de la rama principal “**master**”.

La forma actual de nombrar la rama principal es “**main**”.

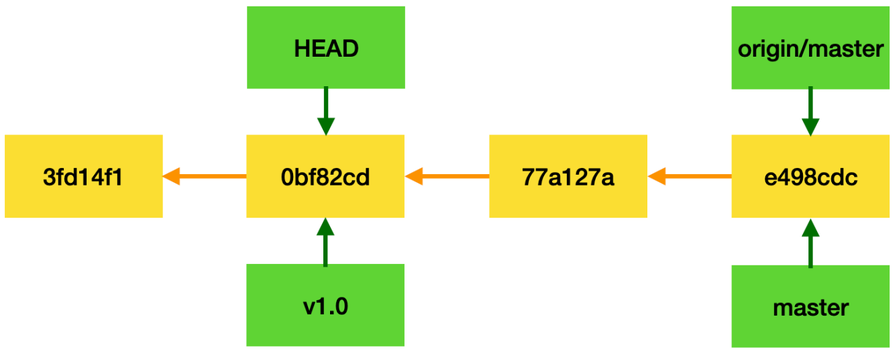
Repaso Git: grafo de commits e índices

1



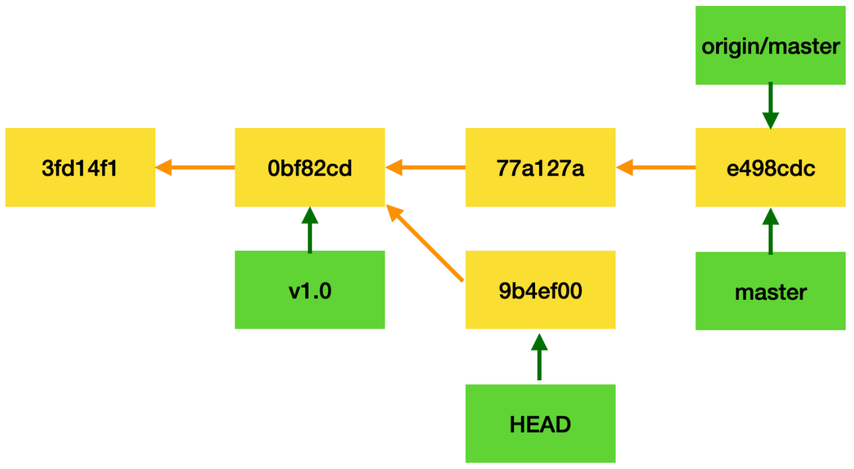

```
git checkout v1.0
```

2



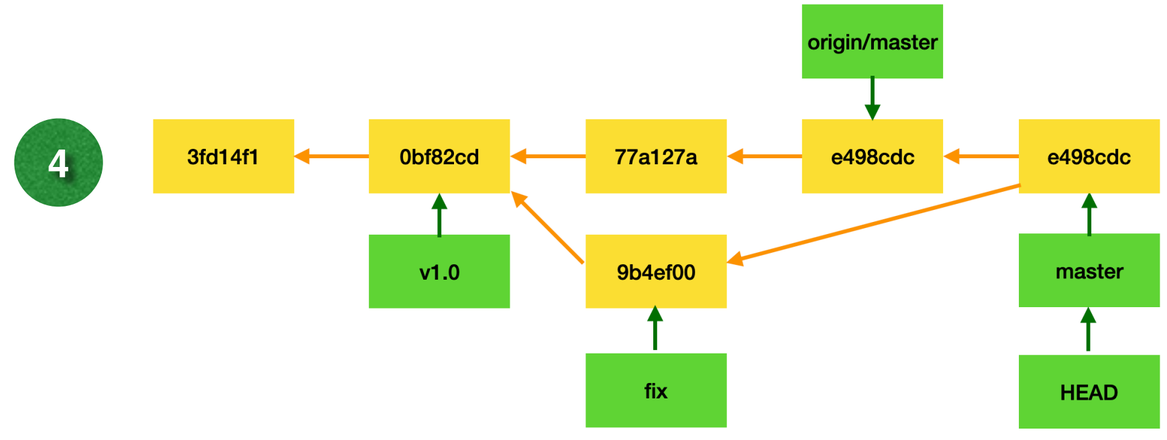

```
$ git add .
$ git commit -m "Cambios sobre un commit pasado"
```

3




```
$ git branch fix
$ git checkout master
$ git merge fix
```

4



Repaso Git: cambiar el último commit en local

Si queremos cambiar sólo el mensaje del commit:

```
$ git commit --amend -m "<nuevo mensaje>"
```

Si queremos deshacer el commit, pero no los cambios introducidos en él:

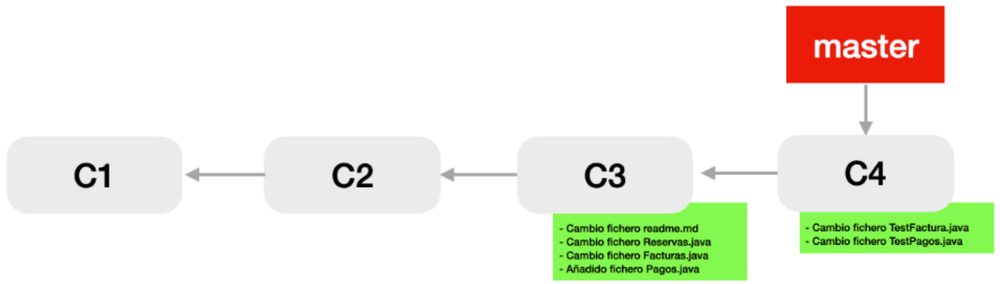
```
$ git reset HEAD^
```

El espacio de trabajo no cambia, pero el commit se ha desecho.

`HEAD^` significa "el commit anterior a HEAD". Es equivalente a poner el número de commit anterior al actual:

```
$ git reset <commit-anterior>
```

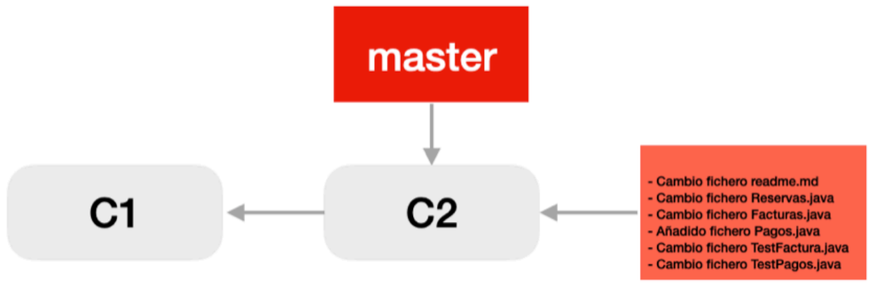
Repaso Git: reset



Si hacemos:

```
$ git reset C2
```

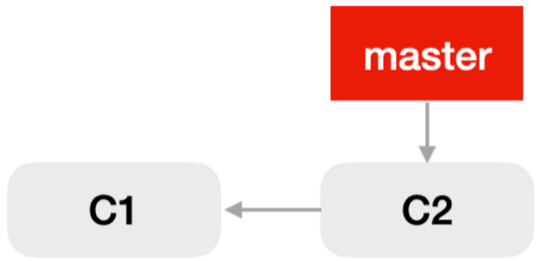
moveremos el índice de la rama actual a `C2` y todos los cambios de los commits intermedios (`C3` y `C4`) se mantienen en el directorio de trabajo, pero sin estar anotados en ningún commit:



Sin embargo, si hacemos

```
$ git reset --hard C2
```

eliminamos todos los cambios y toda la historia. Es equivalente a un `checkout` moviendo el índice de la rama.



Una vez que hemos modificado el grafo de commits en nuestro repositorio local, podemos subirlo al repositorio global (reemplazando lo que hay) con:

```
$ git push --force
```

¡Cuidado! No hacer esto si algún compañero se ha descargado los últimos cambios del repo global.

Repaso Git: revert

Si queremos modificar unos commits que ya se han descargado del repositorio remoto otros compañeros, podemos **revertir los cambios** con el comando

```
$ git revert
```

El comando introduce un commit con exactamente los cambios contrarios a los commits indicados.

Por ejemplo, el siguiente comando crea un commit que revierte el último commit

```
$ git revert HEAD
```

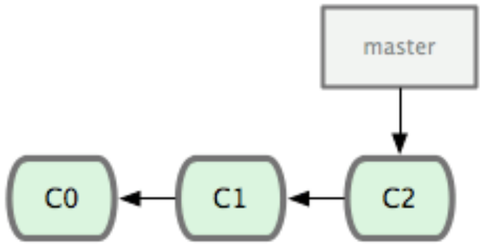
Para revertir los cambios realizados hace 3 commits:

```
$ git revert HEAD~3
```

Para revertir sin commitear los cambios realizados por el quinto último commit en master (incluido) hasta el tercer último commit en master (incluido), usando la opción -n:

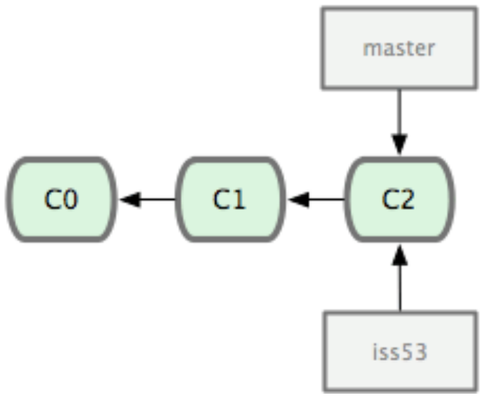
```
$ git revert -n master~5..master~2
```

Recordatorio de ramas en git



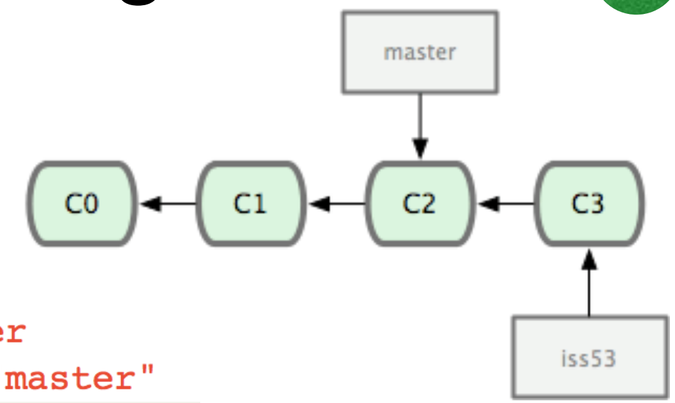
1

```
$ git branch iss53
$ git checkout iss53
$ git checkout -b iss53
Switched to a new branch "iss53"
```



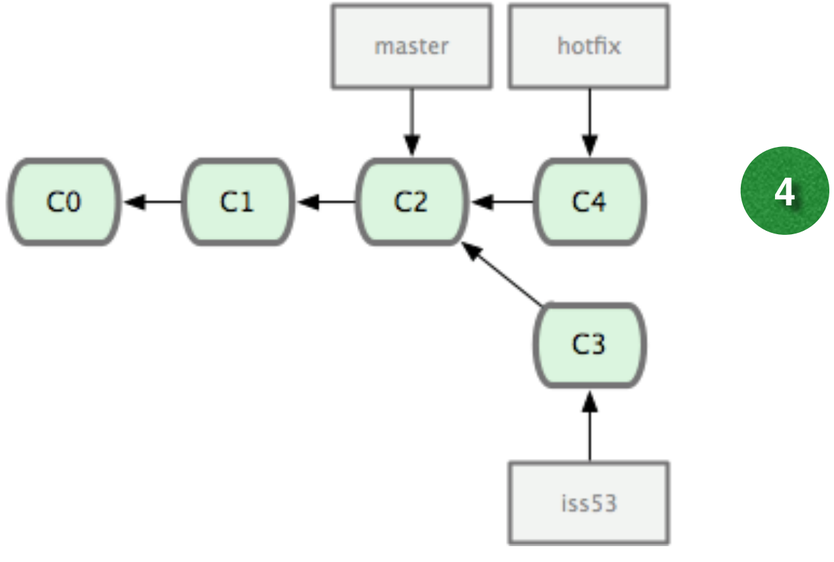
2

```
$ vim index.html
$ git commit -a -m 'added a new footer [issue 53]'
```



3

```
$ git checkout master
Switched to branch "master"
$ git checkout -b hotfix
Switched to a new branch "hotfix"
$ vim index.html
$ git commit -a -m 'fixed the broken email address'
[hotfix]: created 3a0874c: "fixed the broken email address"
 1 files changed, 0 insertions(+), 1 deletions(-)
```



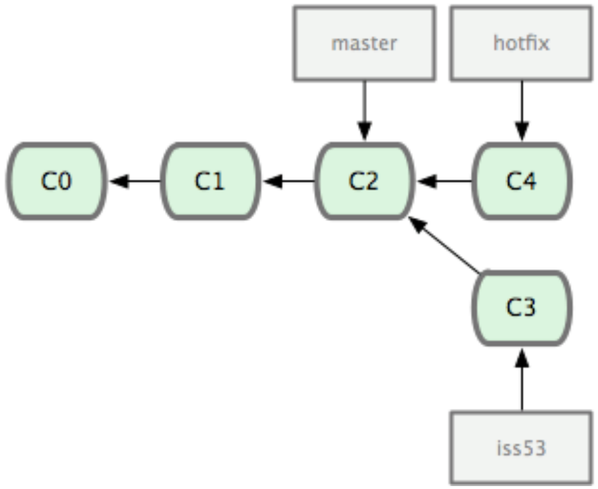
4

[Git Documentation - Basic Branching and Merging]

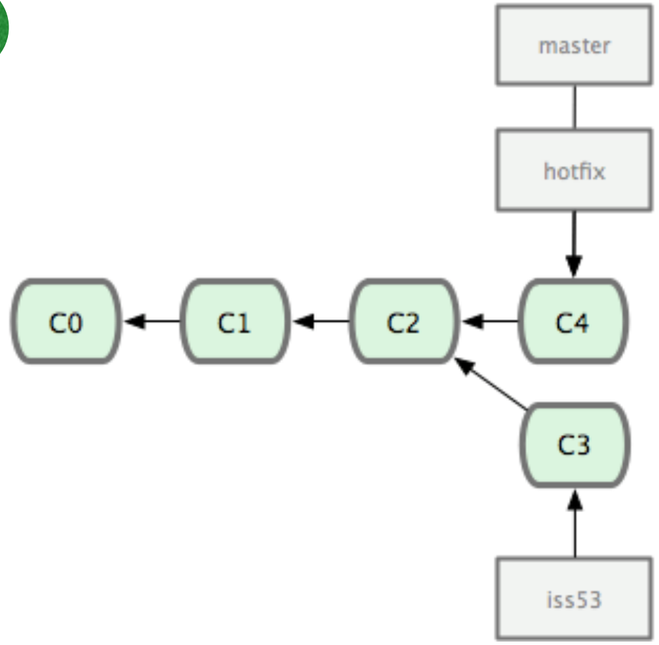
Mezcla de ramas (1)

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast forward
 README |    1 -
 1 files changed, 0 insertions(+), 1 deletions(-)
```

4



5

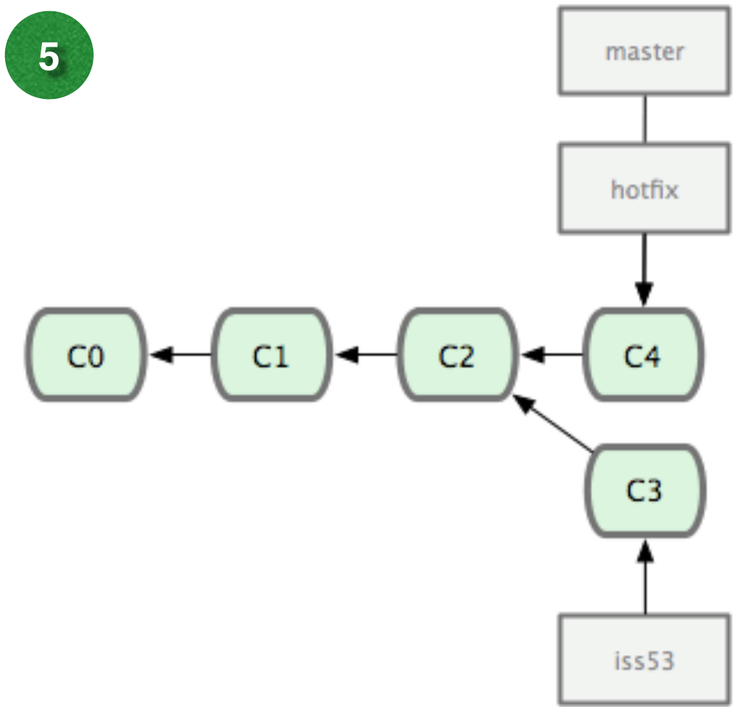


Borramos rama hotfix y añadimos un commit en iss53

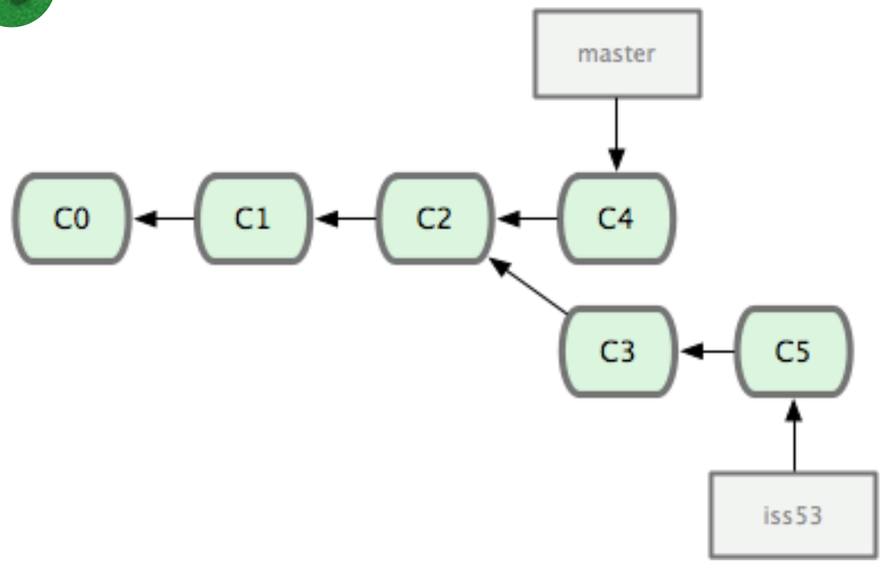
```
$ git branch -d hotfix
Deleted branch hotfix (3a0874c).
```

```
$ git checkout iss53
Switched to branch "iss53"
$ vim index.html
$ git commit -a -m 'finished the new footer [issue
```

5

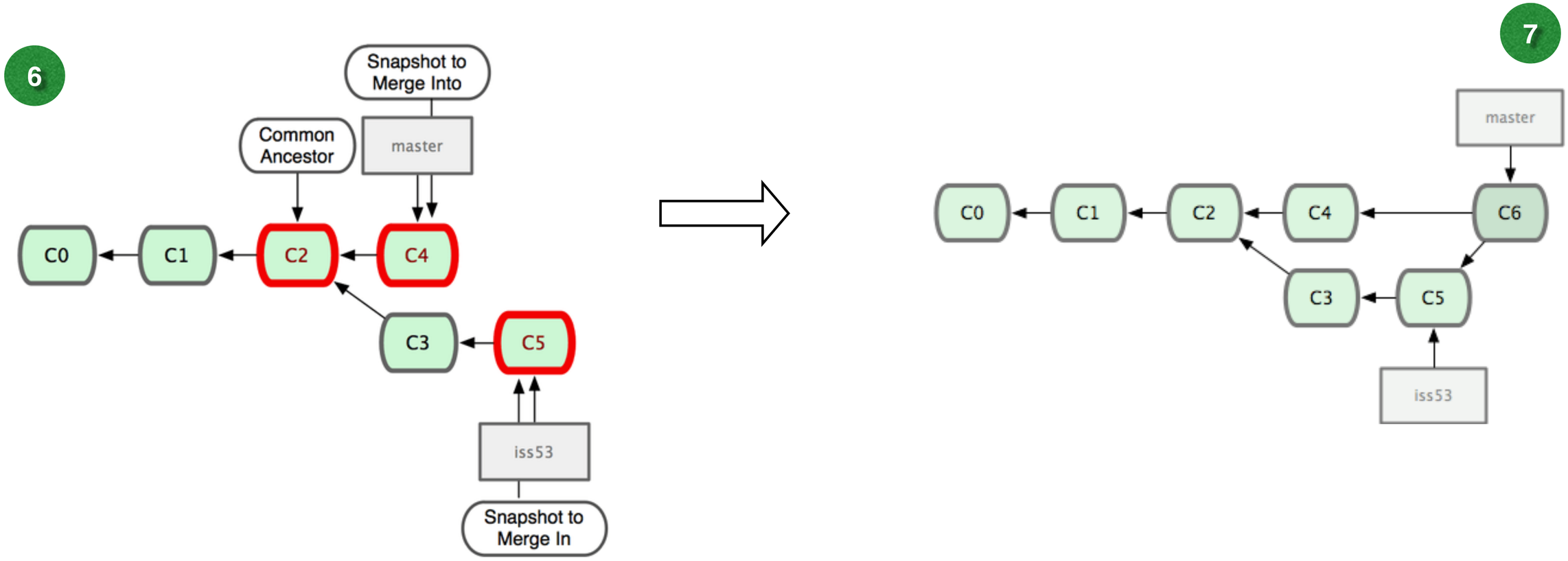


6



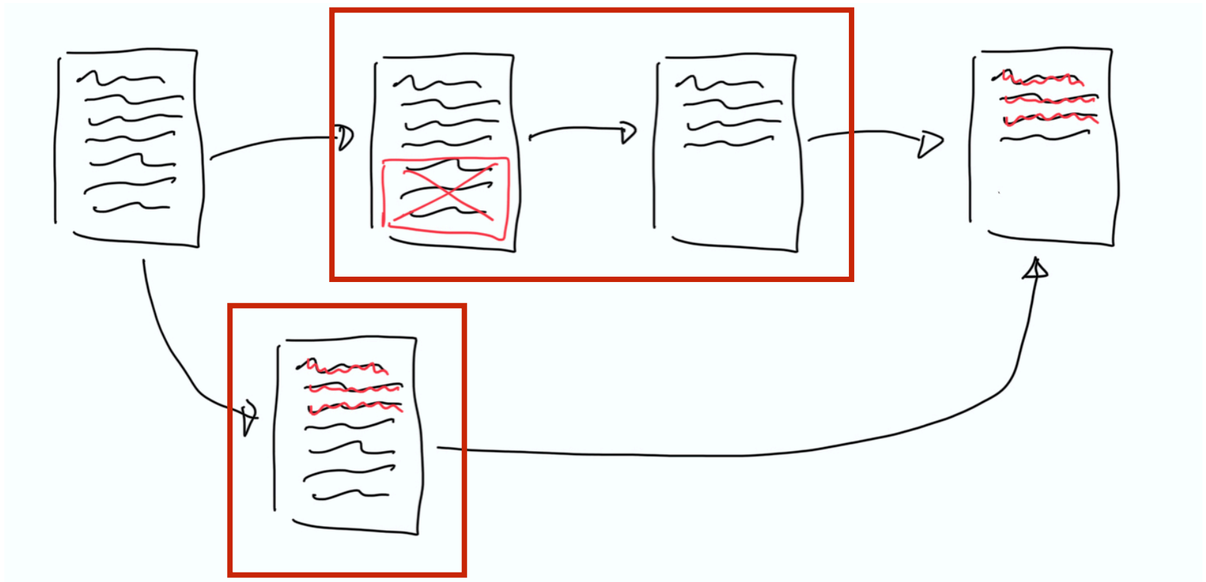
Mezcla de ramas (2)

```
$ git checkout master
$ git merge iss53
Merge made by recursive.
 README |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```



Conflictos en la mezcla de dos ramas

No hay conflicto



No hay conflicto cuando el resultado de aplicar los cambios de la rama 1 y la rama 2 es independiente del orden en que aplicamos las ramas.

Hay conflicto

```
  <ul>
    <li><a href="#">Home</a></li>
<<<<<<< HEAD
    <li><a href="#">Equipo</a></li>
    <li><a href="#">Projectos</a></li>
=======
    <li><a href="#">Servicios</a></li>
    <li><a href="#">Portfolio</a></li>
    <li><a href="#">Equipo</a></li>
>>>>>>> iss56
    <li><a href="#">Contacto</a></li>
  </ul>
```

Hay conflicto cuando una y otra rama tocan las mismas líneas de algún fichero.

Solución de los conflictos

```
  <ul>
    <li><a href="#">Home</a></li>
<<<<<<< HEAD
    <li><a href="#">Equipo</a></li>
    <li><a href="#">Projectos</a></li>
=======
    <li><a href="#">Servicios</a></li>
    <li><a href="#">Portfolio</a></li>
    <li><a href="#">Equipo</a></li>
>>>>>>> iss56
    <li><a href="#">Contacto</a></li>
  </ul>
```

Git detiene el merge y nos informa de los conflictos

```
$ git merge iss56
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the resul
```

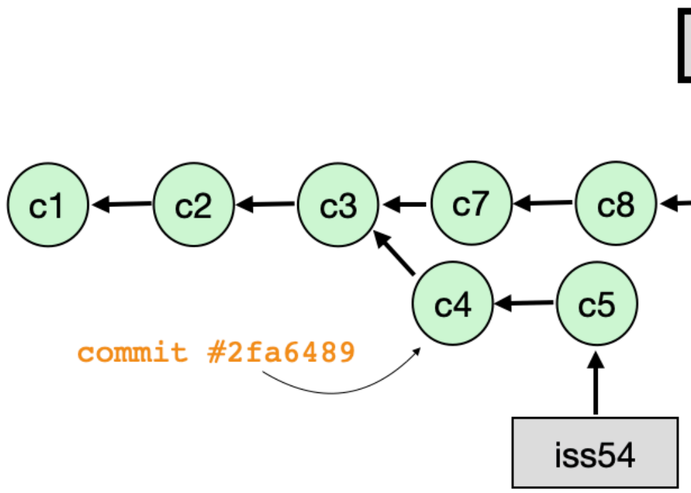
Git modifica el contenido de los ficheros en los que ha detectado el conflicto, de forma que se marcan los cambios que se introducen en una y otra rama.

Tendremos que editar los ficheros y dejar el código como queramos. Hay editores que tienen una interfaz especial que permite seleccionar uno de los cambios o los dos.

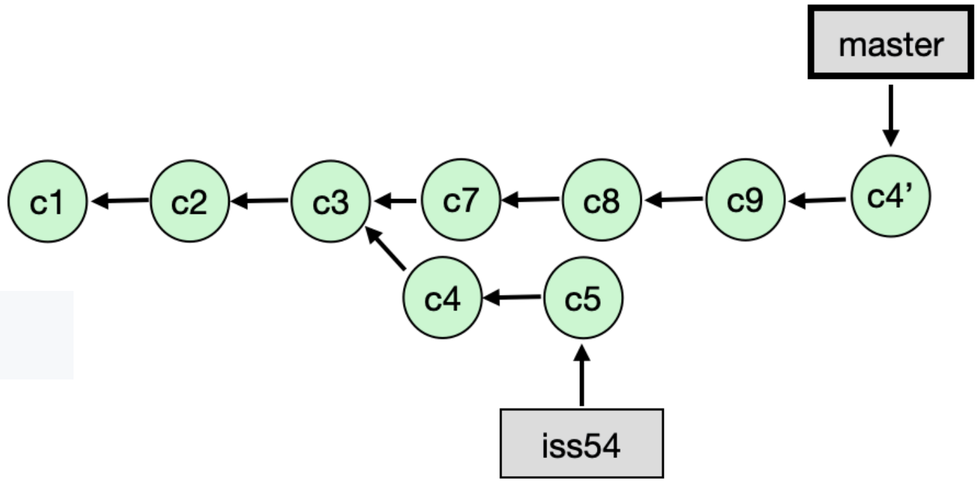
Una vez modificados todos los ficheros en los que hay un error hacer un **add** y un **commit** de los ficheros modificados, lo que confirma el merge.

```
$ git add .
$ git status
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:

    modified:   index.html
$ git commit -m "Merge branch 'iss56' y resueltos conflictos"
[master 2046b52] Merge branch 'iss56' y resueltos conflictos
```

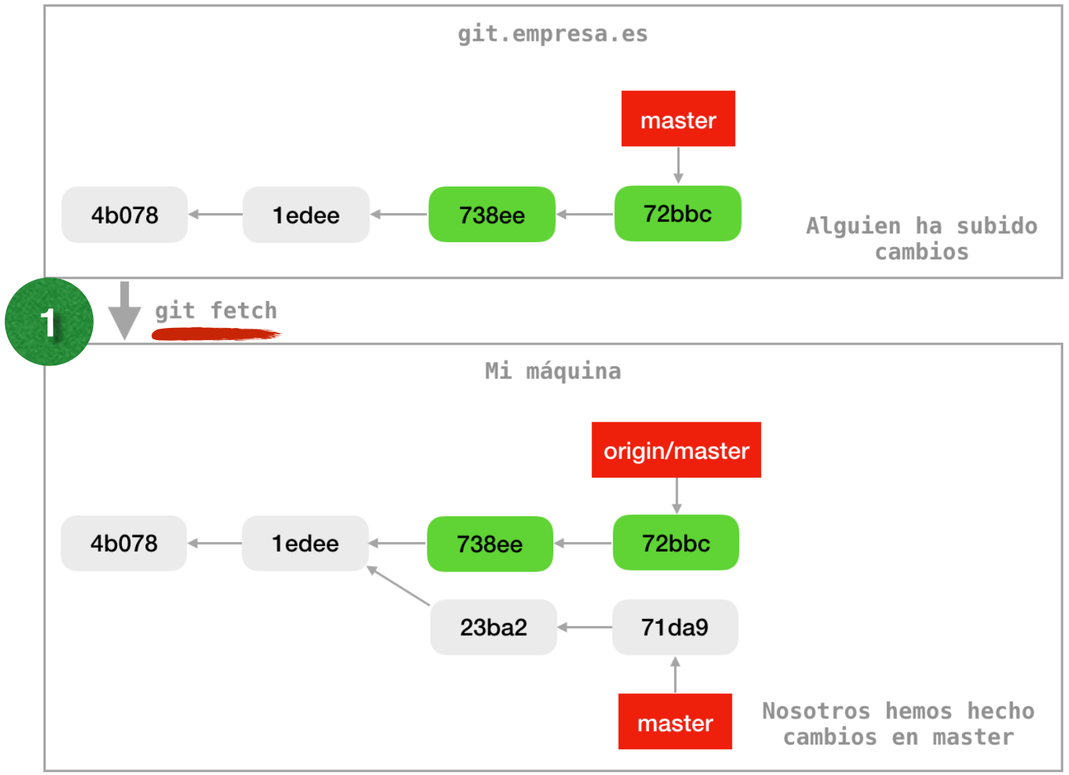
Cherry-pick



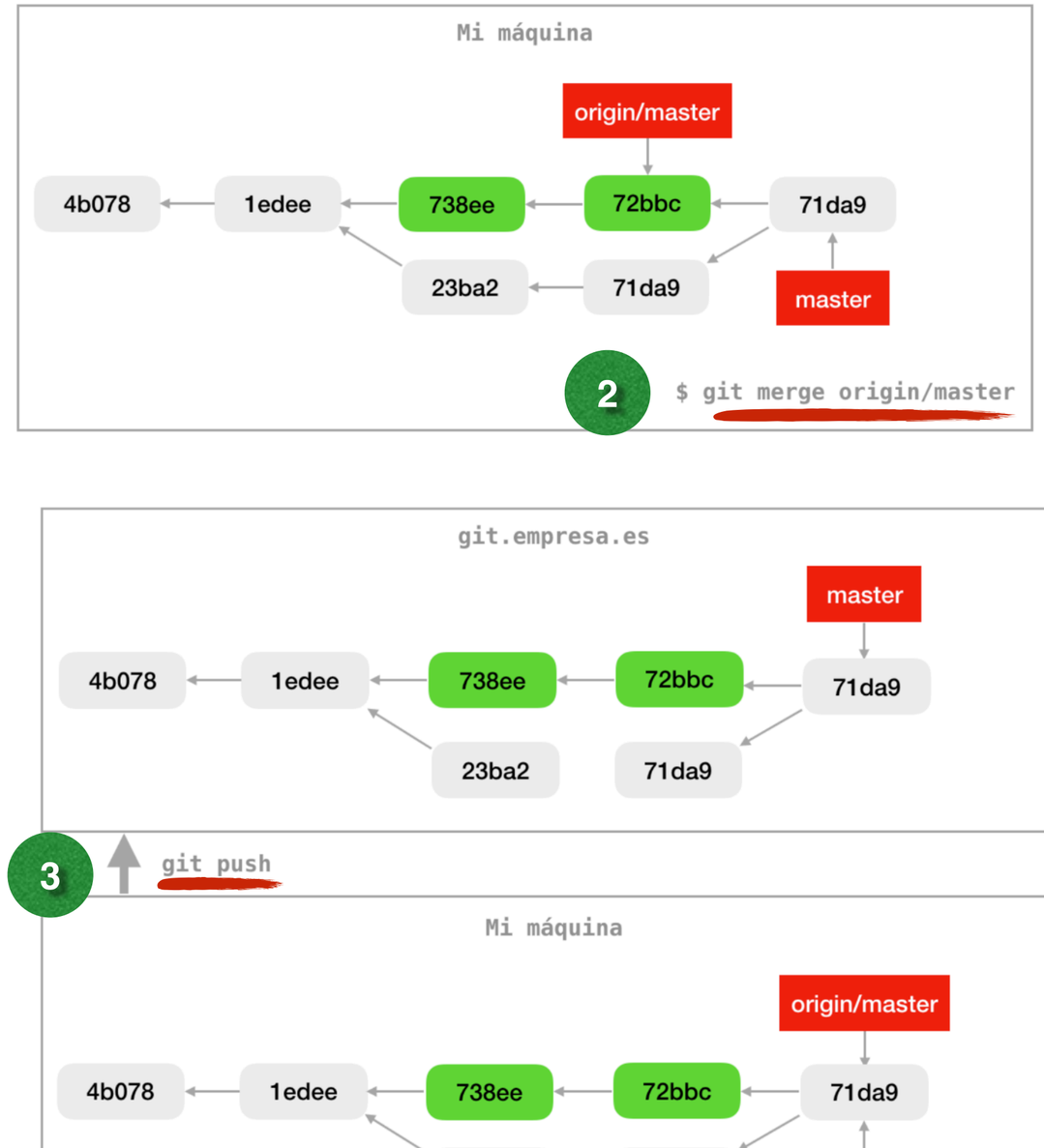

```
$ git cherry-pick 2fa6489
```



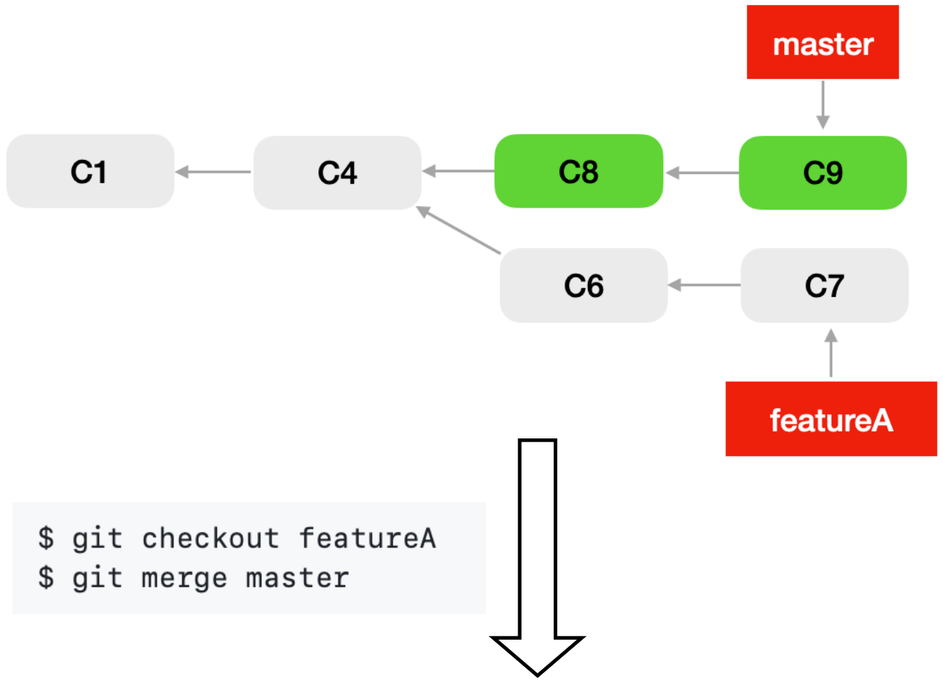
Ramas remotas



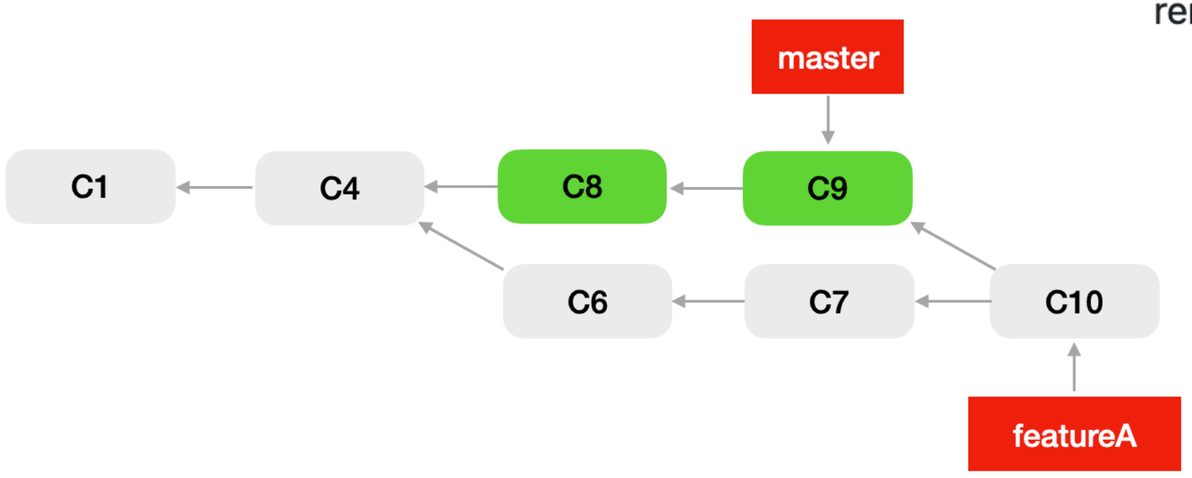
El comando git pull hace automáticamente un git fetch + git merge



Actualizar una rama con cambios en main




```
$ git checkout featureA
$ git merge master
```



El comando diff sigue funcionando correctamente y muestra sólo los cambios de la rama:

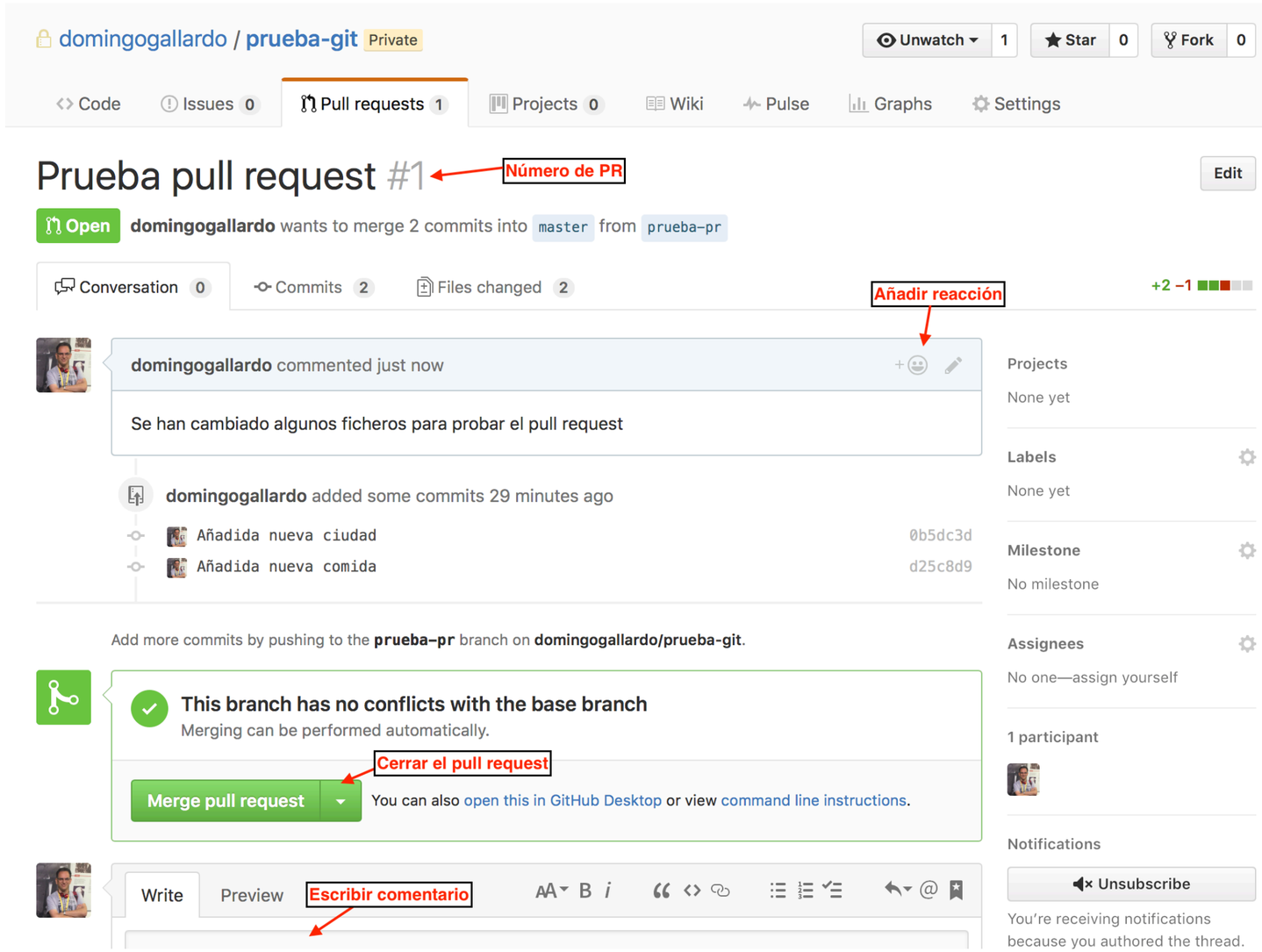
```
$ git diff master...featureA
# Muestra los cambios de los commits C6 y C7
```

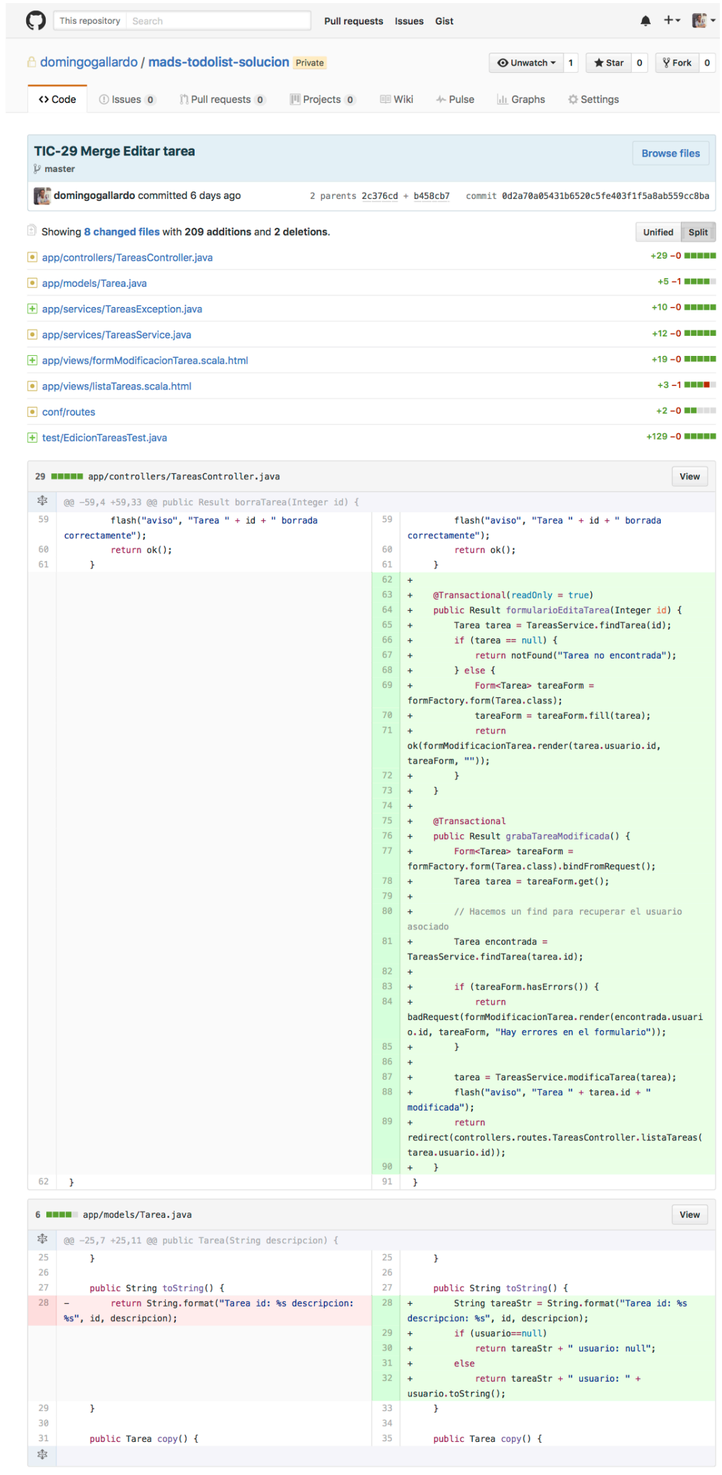
Podemos seguir trabajando en la rama, subirla a repositorio remoto y los compañeros pueden seguir trabajando en ella.

Pull requests

- Funcionalidad proporcionada por los servicios de Git (GitHub, Bitbucket, …) que permite usar la **interfaz web** para solicitar un **merge entre una rama y otra** que residen en el servidor remoto, y permitir interactuar al equipo.
- Las ramas a mezclar pueden estar en repositorios distintos. Por ejemplo, es muy normal en el desarrollo open source que se quiera integrar una rama de un repositorio originado por un **fork**.
- El nombre es equívoco. La funcionalidad debería llamarse mejor **merge request**.
- El servicio permite seleccionar la rama que se desea integrar y la rama en la que queremos hacer la integración. Podemos seguir subiendo commits a cualquiera de las dos ramas y el servicio las mantiene actualizadas.
- Cuando se pulsa el botón, se realiza el merge y el servicio nos da la opción de borrar la rama mezclada.
- En la rama en la que se integran los cambios se crea un commit de merge (similar a si hiciéramos un merge —no-ff).
- Se puede consultar toda la historia de pull-requests.

Pull requests




This repository Search
Pull requests Issues Gist
domingogallardo / mads-todolist-solucion Private
Unwatch 1 Star 0 Fork 0
Code Issues 0 Pull requests 0 Projects 0 Wiki Pulse Graphs Settings
TIC-29 Merge Editar tarea
master
Browse files
domingogallardo committed 6 days ago
2 parents 2c376cd + b458cb7 commit 0d2a70a05431b6520c5fe403f1f5a8ab559cc8ba
Showing 8 changed files with 209 additions and 2 deletions.
Unified Split
app/controllers/TareasController.java +29 −0
app/models/Tarea.java +5 −1
app/services/TareasException.java +10 −0
app/services/TareasService.java +12 −0
app/views/formModificacionTarea.scala.html +19 −0
app/views/listaTareas.scala.html +3 −1
conf/routes +2 −0
test/EdicionTareasTest.java +129 −0
29 app/controllers/TareasController.java
View
@@ −59,4 +59,33 @@ public Result borraTarea(Integer id) {
flash("aviso", "Tarea " + id + " borrada correctamente");
return ok();
}
+
+ @Transactional(readOnly = true)
+ public Result formularioEditaTarea(Integer id) {
+ Tarea tarea = TareasService.findTarea(id);
+ if (tarea == null) {
+ return notFound("Tarea no encontrada");
+ } else {
+ Form<Tarea> tareaForm = formFactory.form(Tarea.class);
+ tareaForm = tareaForm.fill(tarea);
+ return ok(formModificacionTarea.render(tarea.usuario.id, tareaForm, ""));
+ }
+ }
+
+ @Transactional
+ public Result grabaTareaModificada() {
+ Form<Tarea> tareaForm = formFactory.form(Tarea.class).bindFromRequest();
+ Tarea tarea = tareaForm.get();
+
+ // Hacemos un find para recuperar el usuario asociado
+ Tarea encontrada = TareasService.findTarea(tarea.id);
+
+ if (tareaForm.hasErrors()) {
+ return badRequest(formModificacionTarea.render(encontrada.usuario.id, tareaForm, "Hay errores en el formulario"));
+ }
+
+ tarea = TareasService.modificaTarea(tarea);
+ flash("aviso", "Tarea " + tarea.id + " modificada");
+ return redirect(controllers.routes.TareasController.listaTareas(tarea.usuario.id));
+ }
}
6 app/models/Tarea.java
View
@@ −25,7 +25,11 @@ public Tarea(String descripcion) {
}
public String toString() {
- return String.format("Tarea id: %s descripcion: %s", id, descripcion);
+ String tareaStr = String.format("Tarea id: %s descripcion: %s", id, descripcion);
+ if (usuario==null)
+ return tareaStr + " usuario: null";
+ else
+ return tareaStr + " usuario: " + usuario.toString();
}
public Tarea copy() {

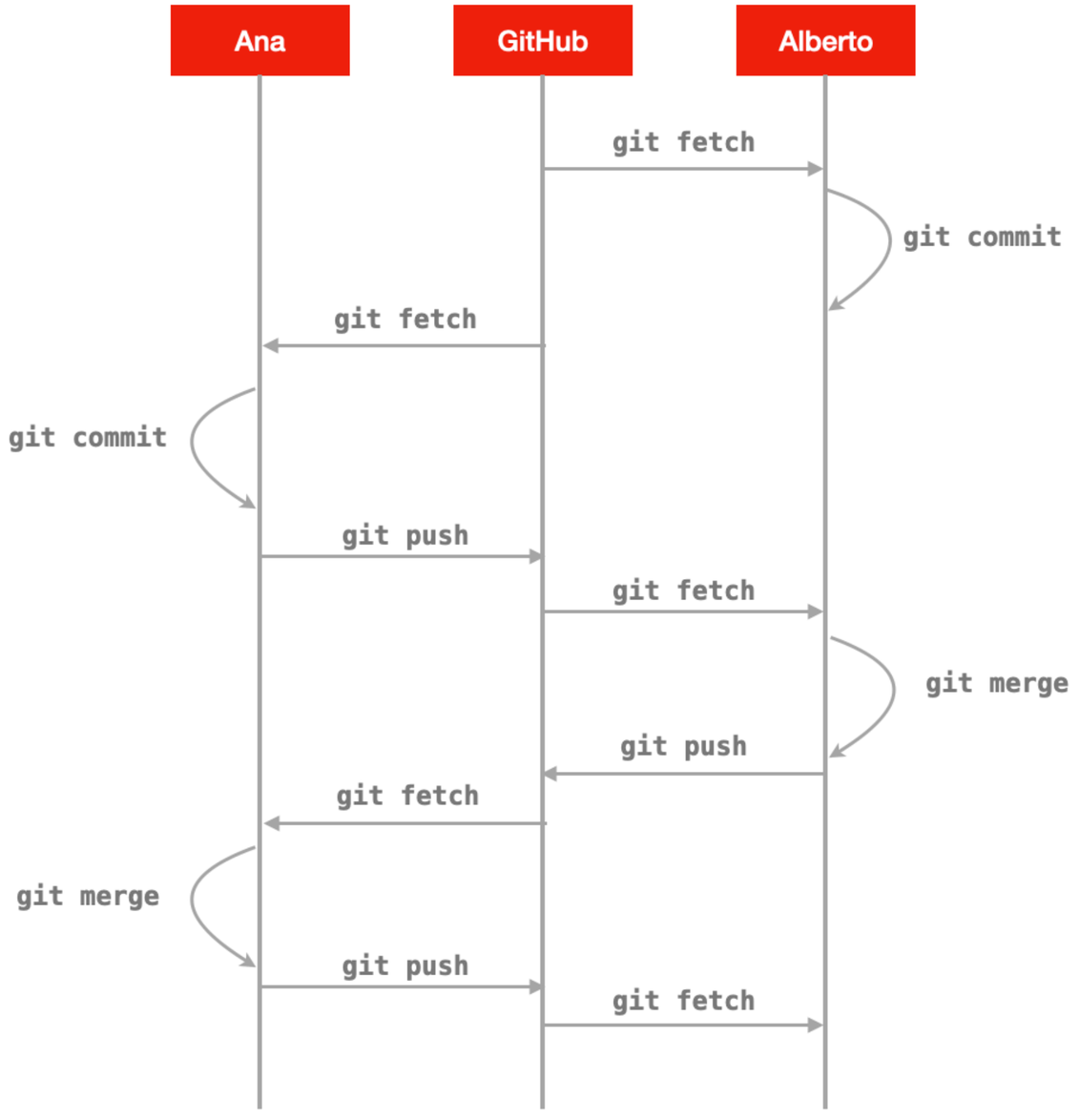
Flujos de trabajo

- Distintas opciones posibles
- En los tutoriales de Atlassian se puede encontrar un documento muy bueno que compara los distintos flujos de trabajo (*Comparing Workflows*)

- **Trunk based development**
  - Recomendación de Fowler y Humbler: "*Everyone Commits To the Mainline Every Day*"
  - Una única rama de desarrollo principal, en la que todos los desarrolladores hacen *commit* a diario
  - La rama de desarrollo se despliega diariamente en el servidor de integración continua
  - Lo veremos cuando hablemos de Continuous Delivery
- **Branch based development**
  - Se trabajan en ramas y se integran en la rama principal
  - Para ser ágiles, las ramas deben ser *short-lived branches*, ramas en las que se trabaja durante unos pocos días

Trabajo sobre una rama

- Lo hemos probado en la práctica 4.
- El último desarrollador que sube los cambios a la rama es el que deberá hacer el “merge” de los cambios hecho por el compañero.
- Esto promueve un interés en integrar rápidamente los cambios y hacer cambios pequeños y frecuentes.



Línea principal con *short-lived branches*

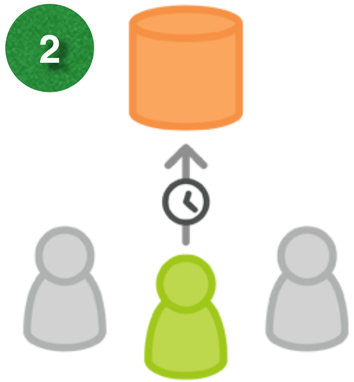
1

Mary begins a new feature

```
git checkout -b marys-feature master
```

```
git status
git add <some-file>
git commit
```

2



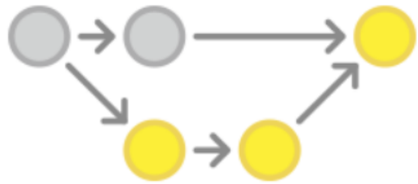
```
git push -u origin marys-feature
```

Cualquier compañero puede bajar la rama y subir cambios:

```
git fetch
git checkout -b marys-feature origin/marys-feature
# realizar cambios
git push -u origin marys-feature
```

3

Mary publishes her feature



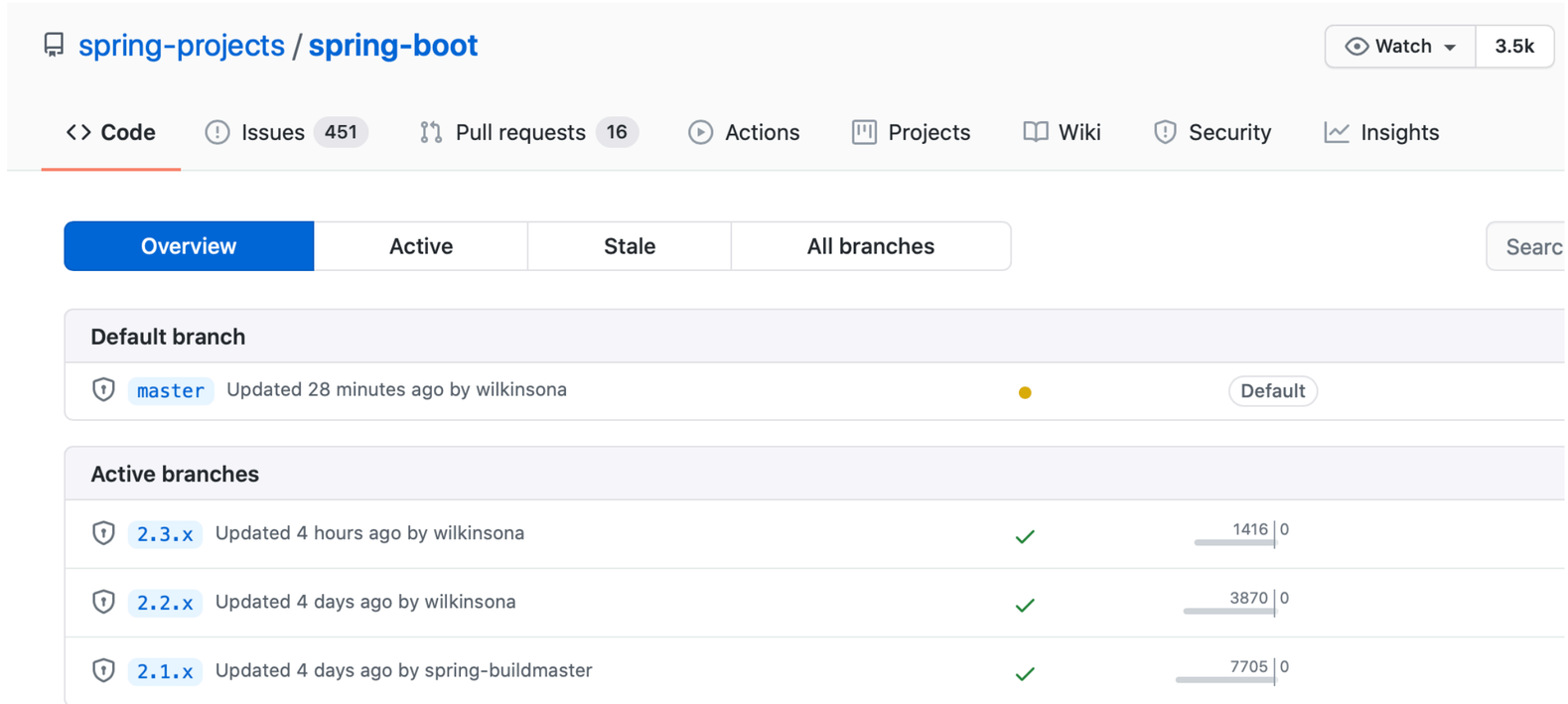
```
git checkout master
git pull
git pull origin marys-feature
git push
```

Hace un merge de la última versión subida a la rama marys-feature. Se puede substituir por un pull request y se denomina **GitHub flow**.

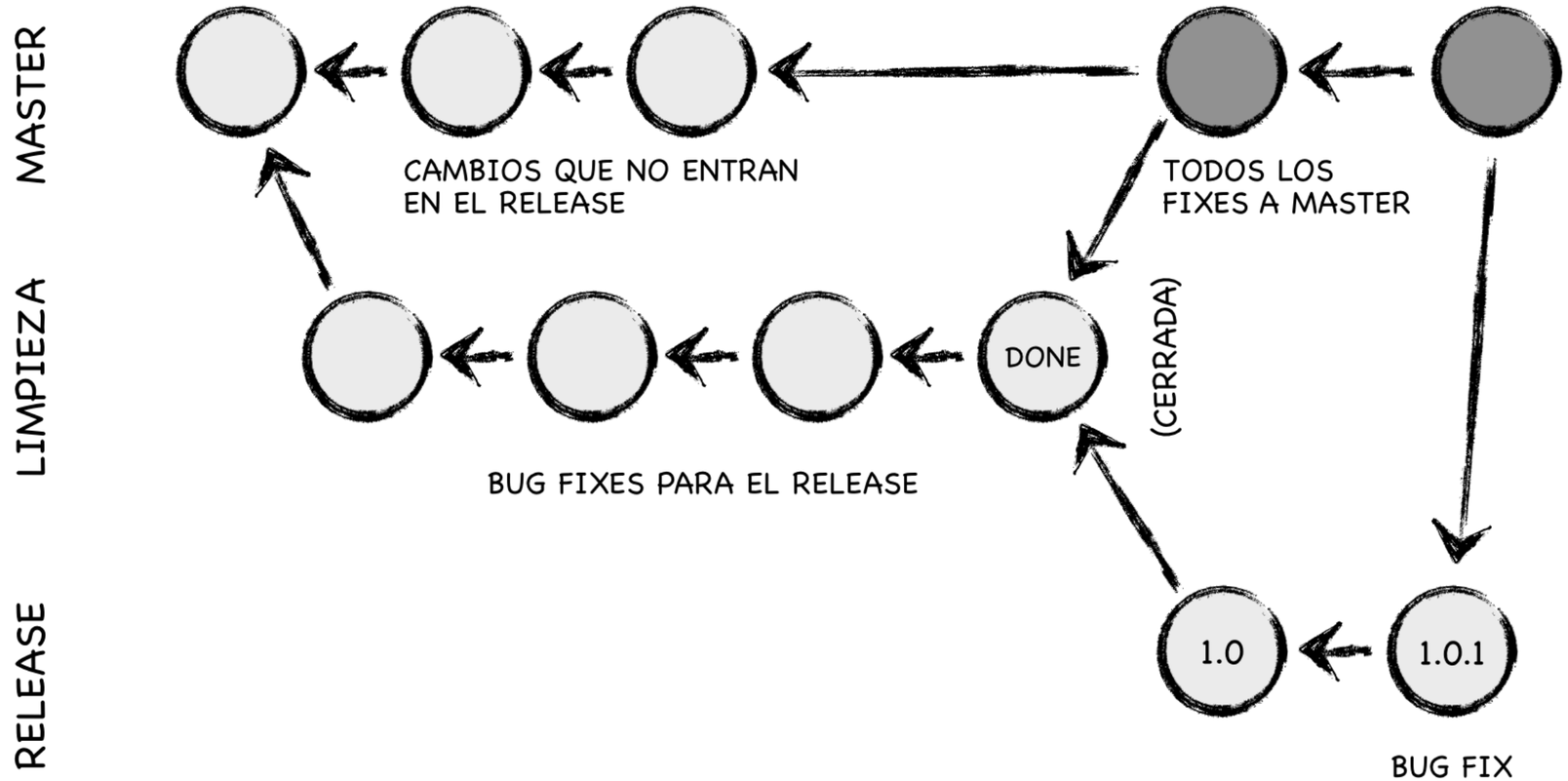
Atlassian Git Tutorials - Feature Branch Workflow

Ramas de versiones

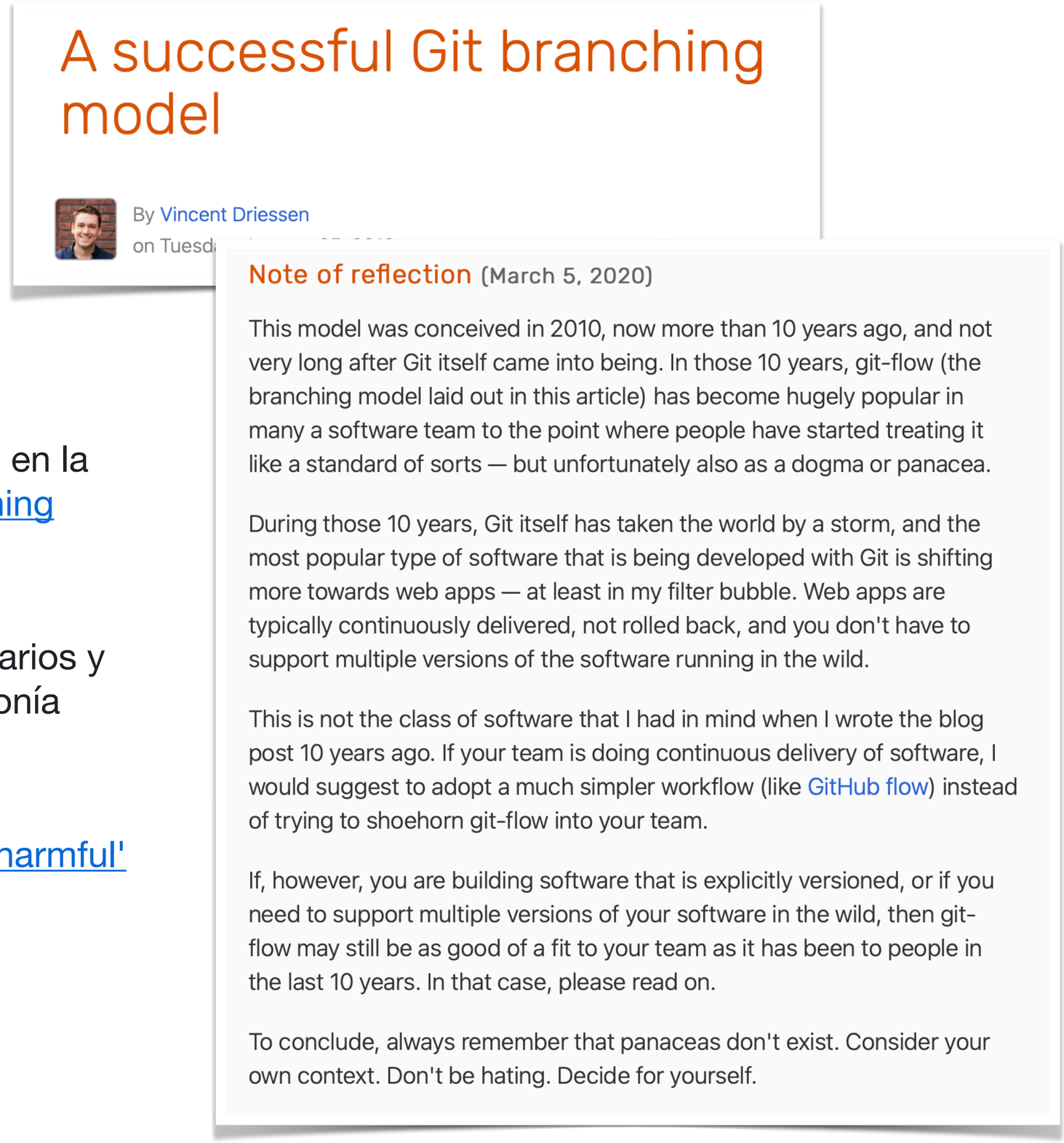
- En la rama main se mantiene la próxima versión.
- Cuando se lanza una versión estable se crea a partir de main una rama con el nombre de la version menor (2.3.x).
- Todos los cambios menores se realizan en la rama y se copian a main si se quiere que se incluyan en futuras versiones.
- Usado en muchos proyectos abiertos: spring-boot, swift, etc.



Congelar el código antes de sacar una versión



GitFlow (1)



A successful Git branching model

By Vincent Driessen

on Tuesd

Note of reflection (March 5, 2020)

This model was conceived in 2010, now more than 10 years ago, and not very long after Git itself came into being. In those 10 years, git-flow (the branching model laid out in this article) has become hugely popular in many a software team to the point where people have started treating it like a standard of sorts — but unfortunately also as a dogma or panacea.

During those 10 years, Git itself has taken the world by a storm, and the most popular type of software that is being developed with Git is shifting more towards web apps — at least in my filter bubble. Web apps are typically continuously delivered, not rolled back, and you don't have to support multiple versions of the software running in the wild.

This is not the class of software that I had in mind when I wrote the blog post 10 years ago. If your team is doing continuous delivery of software, I would suggest to adopt a much simpler workflow (like GitHub flow) instead of trying to shoehorn git-flow into your team.

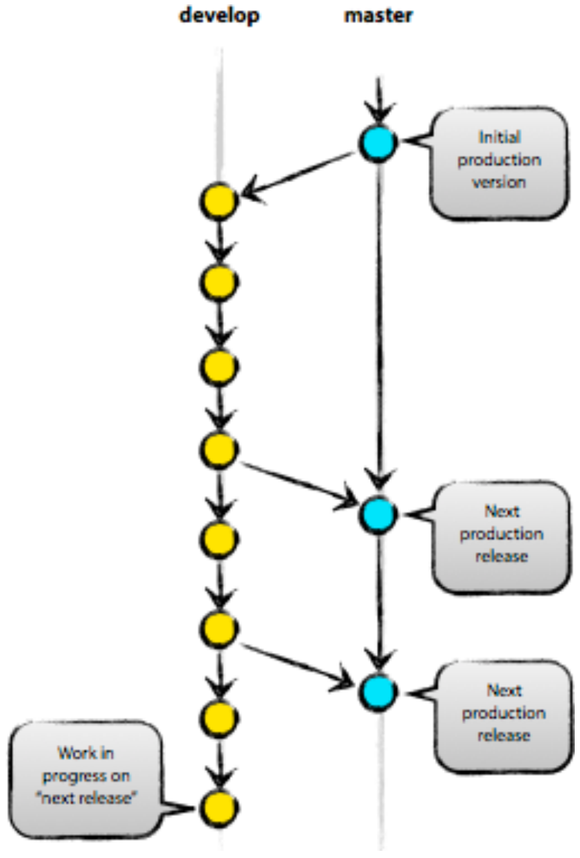
If, however, you are building software that is explicitly versioned, or if you need to support multiple versions of your software in the wild, then git-flow may still be as good of a fit to your team as it has been to people in the last 10 years. In that case, please read on.

To conclude, always remember that panaceas don't exist. Consider your own context. Don't be hating. Decide for yourself.

- Fue propuesto en 2010 por Vincent Driessen en un artículo que publicó en la web titulado A sucessful Git branching model. Se hizo muy popular y fue adoptado por muchas empresas.
- También surgieron muchos comentarios y algunas críticas en las que se proponía flujos de trabajo alternativos:
  - GitFlow considered harmful
  - Follow-up to 'GitFlow considered harmful'
  - A succesful Git branching model considered harmful

GitFlow (2)

Dos ramas *long-lived*:
master (donde van los releases) y
develop (la rama de desarrollo principal)

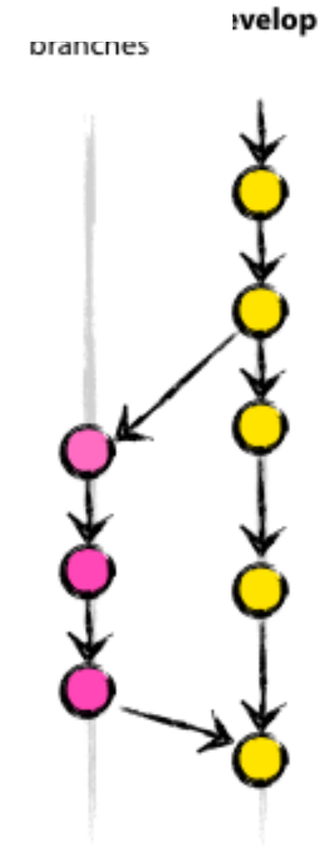


Ramas de ***features***

May branch off from: `develop`
Must merge back into: `develop`
Branch naming convention: anything except `master`, `develop`, `release-*`, or `hotfix-*`

```
$ git checkout -b myfeature develop
Switched to a new branch "myfeature"
```

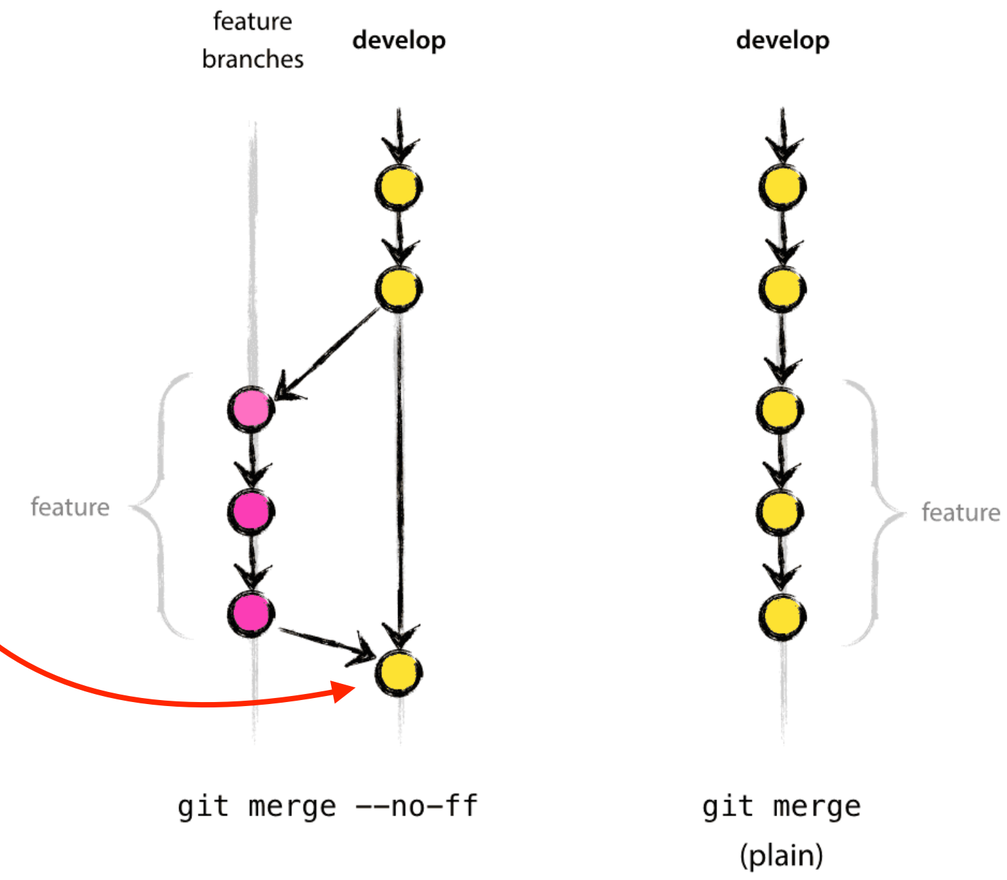
```
$ git checkout develop
Switched to branch 'develop'
$ git merge --no-ff myfeature
Updating ea1b82a..05e9557
(Summary of changes)
$ git branch -d myfeature
Deleted branch myfeature (was 05e9557).
$ git push origin develop
```



GitFlow (3) - git merge --no-ff

- La opción - - no-ff permite crear un commit de merge en la rama en la que se mezcla la feature, incluso en los casos en los que git haría un fast-forward
- Se hace automáticamente cuando mezclamos un pull request. En ese caso, la información del cambio se guarda también en el propio pull request.

Commit con la información de los cambios de la feature



GitFlow (4) - Ramas de release

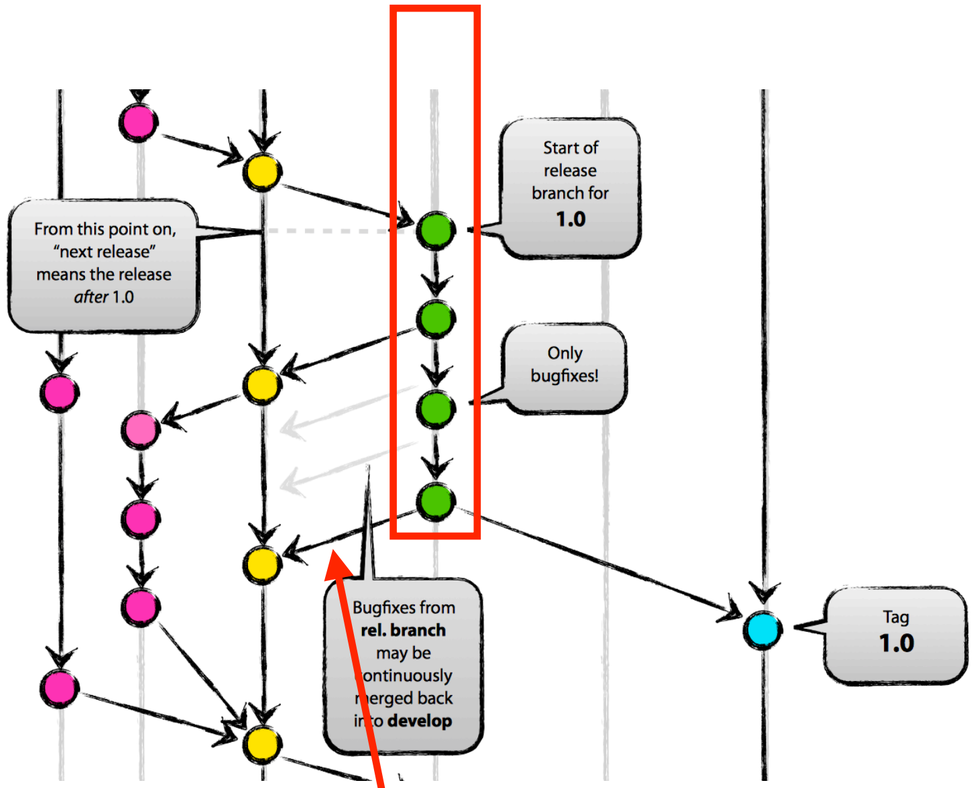
Ramas de ***release***

May branch off from: `develop`
Must merge back into: `develop` and `master`
Branch naming convention: `release-*`

```
$ git checkout -b release-1.2 develop
Switched to a new branch "release-1.2"
$ ./bump-version.sh 1.2
Files modified successfully, version bumped to 1.2.
$ git commit -a -m "Bumped version number to 1.2"
[release-1.2 74d9424] Bumped version number to 1.2
1 files changed, 1 insertions(+), 1 deletions(-)
```

```
$ git checkout master
Switched to branch 'master'
$ git merge --no-ff release-1.2
Merge made by recursive.
(Summary of changes)
$ git tag -a 1.2
```

```
$ git branch -d release-1.2
Deleted branch release-1.2 (was ff452fe).
```



También es posible (algunos lo prefieren) no volver a integrar la rama de release en develop, sino sólo los bugfixes. Se puede crear una rama con cada bugfix, que se integra después en release y en develop.

GitFlow (5) - Ramas hotfixes

Ramas de ***hotfix***

May branch off from:

`master`

Must merge back into:

`develop` and `master`

Branch naming convention:

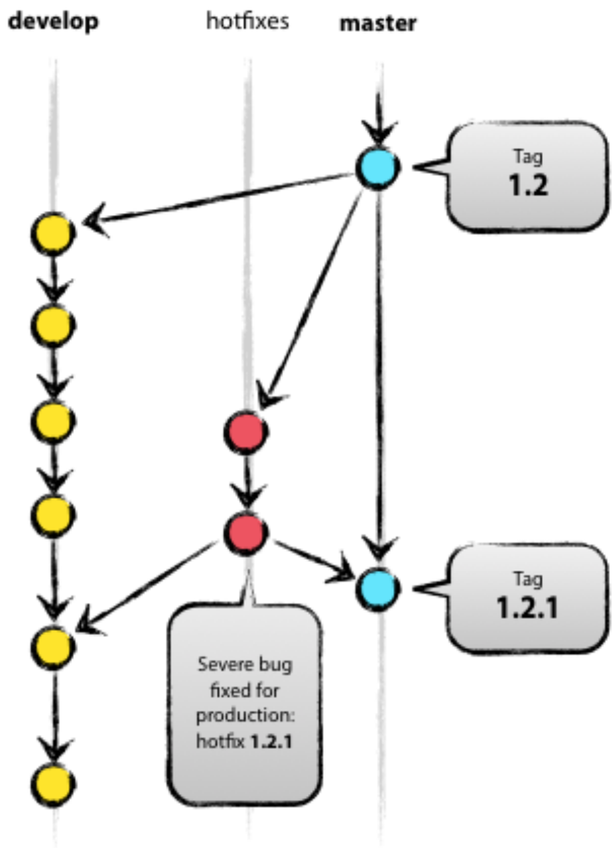
`hotfix-*`

```
$ git checkout -b hotfix-1.2.1 master
Switched to a new branch "hotfix-1.2.1"
$ ./bump-version.sh 1.2.1
Files modified successfully, version bumped to 1.2.1.
$ git commit -a -m "Bumped version number to 1.2.1"
[hotfix-1.2.1 41e61bb] Bumped version number to 1.2.1
1 files changed, 1 insertions(+), 1 deletions(-)
```

```
$ git checkout master
Switched to branch 'master'
$ git merge --no-ff hotfix-1.2.1
Merge made by recursive.
(Summary of changes)
$ git tag -a 1.2.1
```

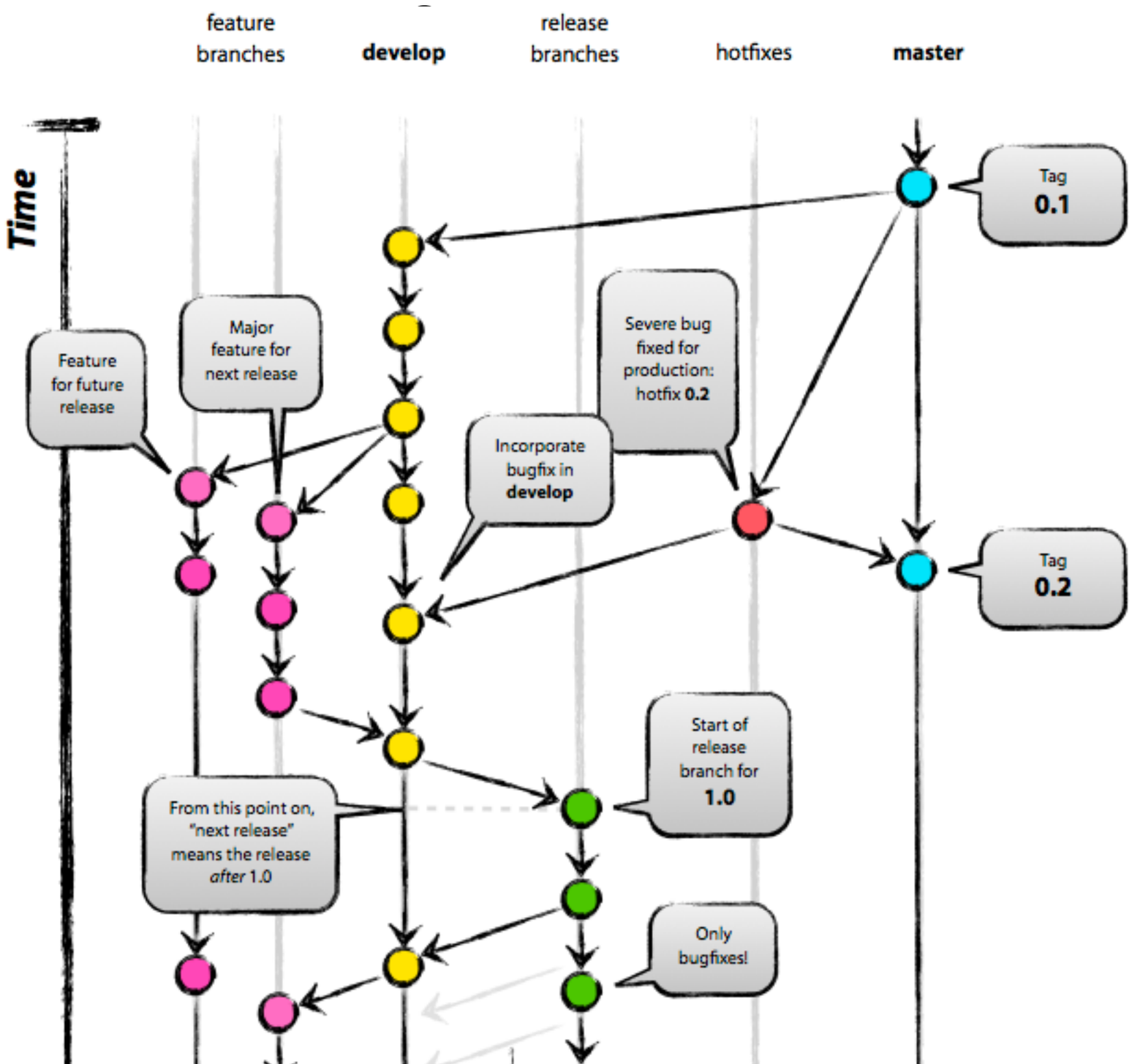
```
$ git checkout develop
Switched to branch 'develop'
$ git merge --no-ff hotfix-1.2.1
Merge made by recursive.
(Summary of changes)
```

```
$ git branch -d hotfix-1.2.1
Deleted branch hotfix-1.2.1 (was abbe5d6).
```



GitFlow (6) - Hotfixes en versiones antiguas

- El flujo de trabajo Git-flow no aborda de forma explícita la gestión de bugs en versiones antiguas, pero el proceso generalmente aceptado es el siguiente:
    1. Crear una rama de mantenimiento a partir de la etiqueta de la versión afectada (por ejemplo, **`maintenance/1.0.x`**)
    2. Corregir el bug en esa rama de mantenimiento
    3. Fusionar la corrección en la rama de mantenimiento y etiquetar una nueva sub-versión (por ejemplo, **`1.0.1`**)
    4. Propagar la corrección a versiones más recientes
- Las ramas de mantenimiento serían también ramas *long-lived* que no habría que eliminar.


feature branches
develop
release branches
hotfixes
master
Time
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
Tag 0.1
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Only bugfixes!

Git-flow es sólo una propuesta

- Existen scripts que incorporan git-flow en forma de comandos que llaman a su vez a comandos más básicos de git
  - **No es recomendable usarl estos scripts** porque perdemos la flexibilidad de adaptar el flujo a nuestras necesidades
- **Nuestra versión** de git-flow
  - Desarrollamos en **master** (en lugar de en develop)
  - Llamamos **production** a la rama con las versiones finales
  - Hacemos un **rebase** cada vez que integramos una feature en master para dejar más limpia la historia del proyecto y siempre añadir features en la cabeza de master
  - Usamos **pull requests** en lugar de merges
- Recomendable leer críticas y contemplar flujos de trabajo alternativos
  - *GitFlow considered harmful*
  - *Follow-up to 'GitFlow considered harmful'*
  - *A succesful Git branching model considered harmful*

Lecturas

- Atlassian Tutorials - *Syncing*
- Atlassian Tutorials - *Comparing Workflows*
- Vincent Driessen - *A successful Git branching model*