PROYECTO DE SOFTWARE

UN POCO MÁS SOBRE GIT

¿Qué veremos en este video?

- ¿Cómo revertimos cambios?
- ¿Cómo resolvemos conflictos?
- Ramas y Tags
- Versionado Semántico
- Gitflow

REVIRTIENDO CAMBIOS

git checkout

- Este comando saca contenido del repositorio y lo pone en el área de trabajo.
- De esta forma permite descartar los cambios no commiteados.
- Además puede utilizarse para "moverse" de rama.
- Este comando **NO** altera la historia de commits.

EJEMPLOS CON CHECKOUT

Descartando cambios en un archivo:

```
$ git checkout file
```

Cambiando de rama:

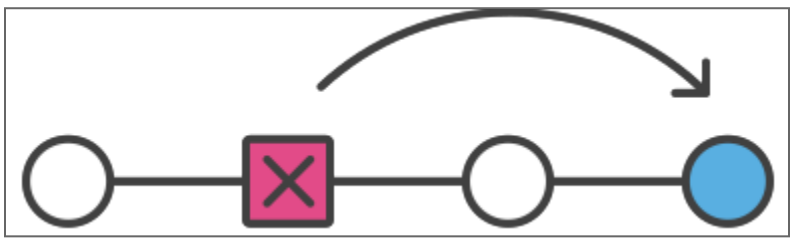
```
$ git checkout rama
```

Crear nueva rama y cambiar a ésta:

```
$ git checkout -b rama
```

git revert

- Este comando crea un nuevo commit que deshace los cambios introducidos por un commit previo.
- Agrega nueva historia al proyecto, **NO** modifica lo ya existente.



EJEMPLOS CON REVERT

- Revertir un commit (produciendo un nuevo commit con los cambios contrarios):

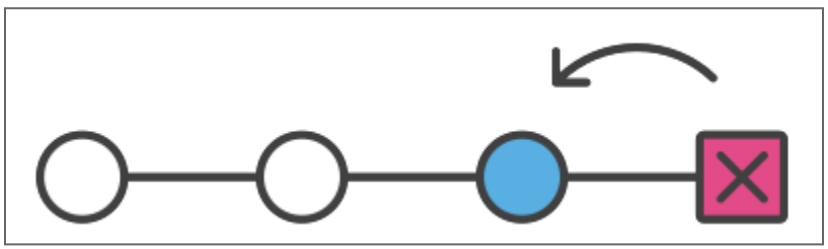
```
$ git revert commit
```

- Aplica los cambios necesarios para revertir el commit sin generar un nuevo commit. Afecta el directorio de trabajo y el área de staging.

```
$ git revert --no-commit commit
```

git reset

- Dependiendo cómo es utilizado este comando realiza operaciones muy distintas.
- Modifica el área de staging.
- Permite modificar qué commit es el último (HEAD) de una rama.



*Este comando **altera la historia del respositorio**.*

EJEMPLOS CON RESET

- Descartar todos los cambios locales en tu directorio de trabajo:

```
$ git reset --hard HEAD
```

- Sacar todos los archivos del área de pruebas (es decir, deshacer el último *git add*):

```
$ git reset HEAD
```

EJEMPLOS CON RESET

- Reestablecer tu puntero HEAD a un commit anterior y descartar todos los cambios desde entonces:

```
$ git reset --hard commit
```

- Reestablecer tu puntero HEAD a un commit anterior y preservar todos los cambios en el área de pruebas (stage area):

```
$ git reset commit
```

- Reestablecer tu puntero HEAD a un commit anterior y preservar los cambios locales sin confirmar (uncommitted changes):

```
$ git reset --keep commit
```

¿ CUÁNDO UTILIZAR CHECKOUT, REVERT O RESET?

- Si modificamos erróneamente un archivo en nuesta área de trabajo sin haber commiteado los cambios, se debe utilizar **git checkout** para traer una copia anterior del archivo desde el repositorio local.
- Si en la historia del proyecto se produjo un commit que luego se decide que fue erróneo, se debe utilizar **git revert**. Va a deshacer los cambios provocados por el commit erróneo, generando un nuevo commit que registra esta corrección.
- Si realizamos un commit erróneo, pero no lo hemos compartido con nadie más (todos los commits son locales). Se puede utilizar **git reset** para reescribir la historia como si nunca se hubiera realizado el commit con errores.

CONFLICTOS

Si al actualizar el repositorio local con el remoto (**git fetch** + **git merge** = **git pull**) no se puede realizar el merge por estar las mismas líneas modificada se produce un **CONFLICTO**.

CONFLICTOS

Git nos avisa que debemos resolver nosotros el conflicto a mano:

```
From git.proyecto.linti.unlp.edu.ar:proyecto/grupo_XX
   2b33f0d..ea36b87  master     -> origin/master
Auto-merging index.php
CONFLICT (content): Merge conflict in index.php
Automatic merge failed; fix conflicts and then commit the result.
```

CONFLICTOS

Debemos editar el archivo en conflicto que va a tener marcadas las líneas con problemas:

```
<h1>
<?php
<<<<<<< HEAD
print "hello Proyecto!!!"
?>
</h1>
=======
print "hello World!!!!!!!!!!"
?>
>>>>>>> ea36b870f9a0e1e6439758b6e681bd329a04db3d
```

y luego volverlo a agregar con **git add** y commiterarlo **git commit**.

ARCHIVOS .GITIGNORE

- Sirven para definir, archivos o directorios que serán ignorados por el sistema de control de versiones.
- **No** todos los archivos dentro de nuestro directorio de trabajo deben ser versionados.

¿QUÉ DEBEMOS IGNORAR?

- Archivos con configuraciones o información sensible que no debe ser pública.
- Dependencias de la app.
- Archivos generados por la app como logs, cache, archivos subidos, binarios, dumps, etc.

PATRONES

Típicamente se crea en nuestro directorio de trabajo un archivo **.gitignore** que puede poseer patrones como por ejemplo:

```
*.log
local_settings.py
env/
var/
```

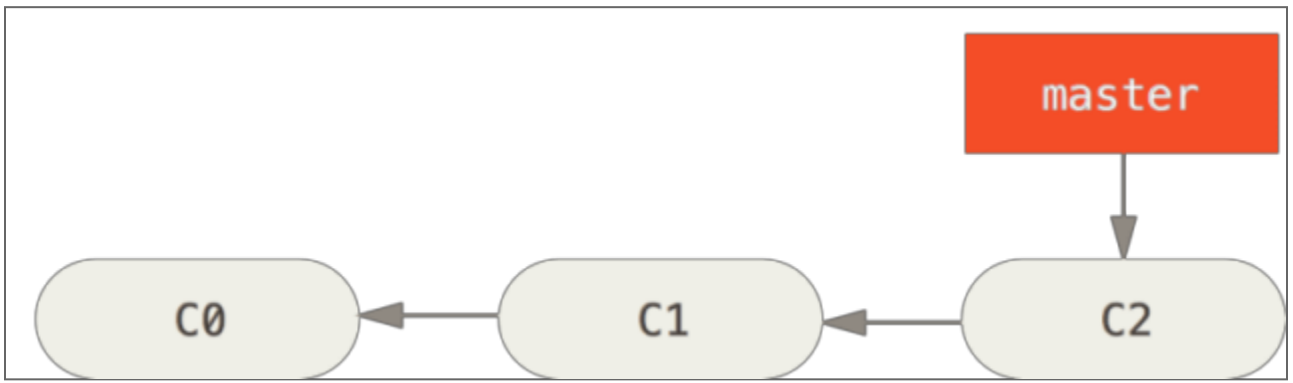
Algunos archivos predefinidos **https://www.gitignore.io/**.

RAMAS (BRANCHES)

- Una rama (branch) es una línea de desarrollo independiente.
- Uno puede desarrollar una nueva funcionalidad independientemente sin interferir con la línea principal.

RAMA MASTER

- Por defecto **git init** crea una rama por defecto para trabajar: **master**.



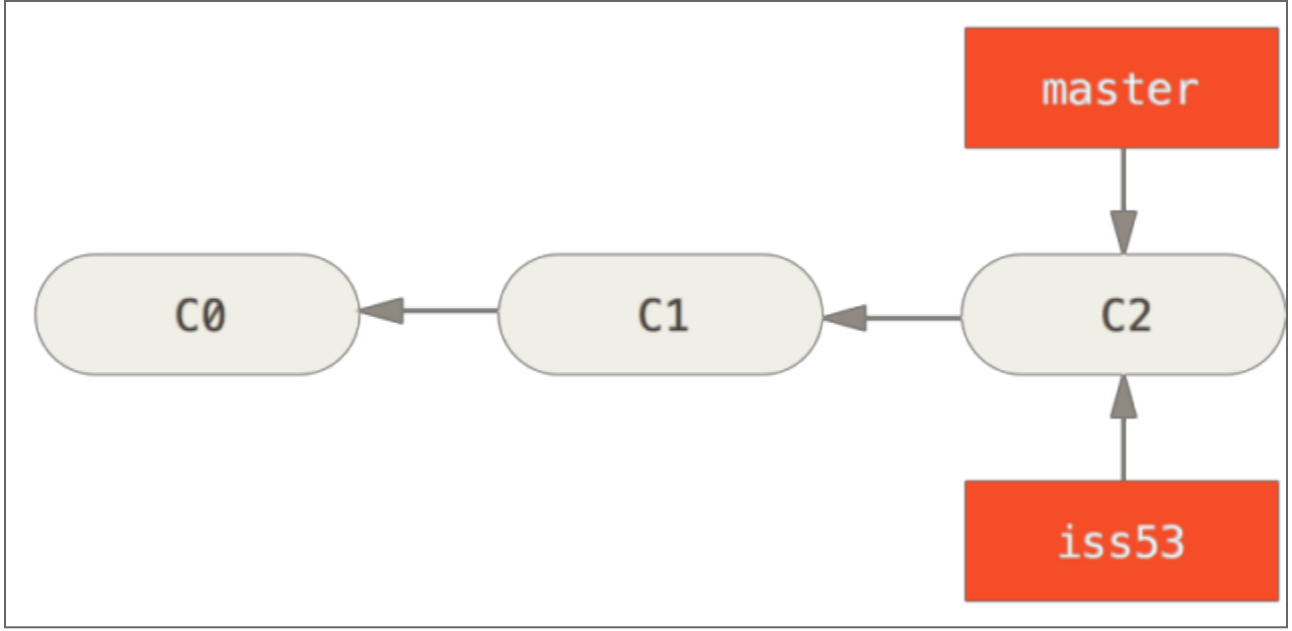
CREANDO UNA RAMA NUEVA

Utilizamos el comando **git branch** para crear una rama nuena y **git checkout** para cambiar el directorio de trabajo a esa rama.

```
$ git branch iss53
$ git checkout iss53
```

Esto se puede realizar en un único comando:

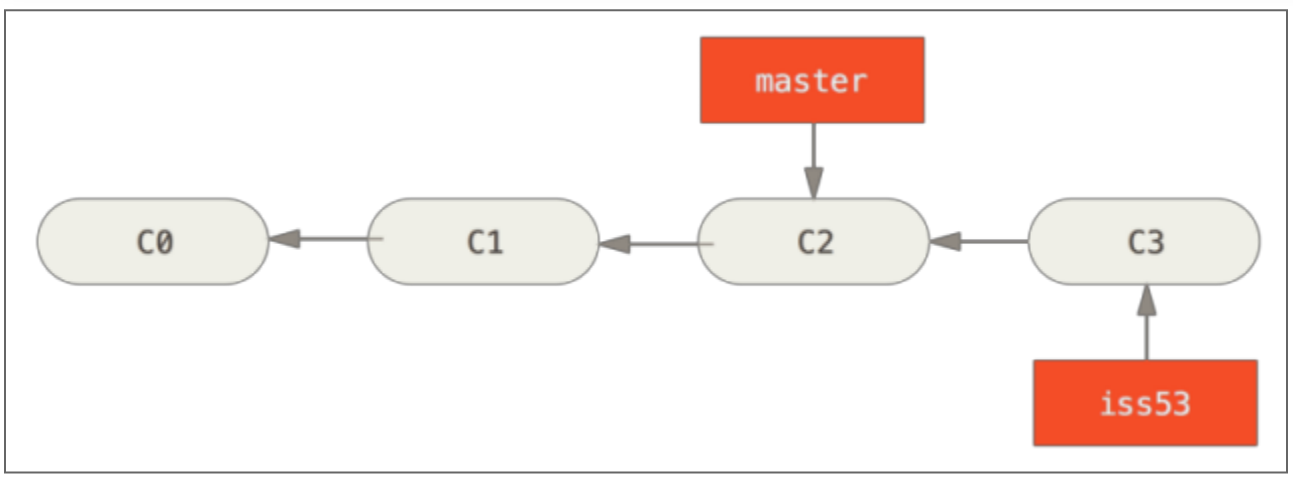
```
$ git checkout -b iss53
```


master
C0
C1
C2
iss53

TRABAJANDO EN LA NUEVA RAMA

Generamos un nuevo commit.

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



VOLVIENDO A MASTER

Necesitamos hacer un arreglo urgente en master, debemos reposicionarnos.

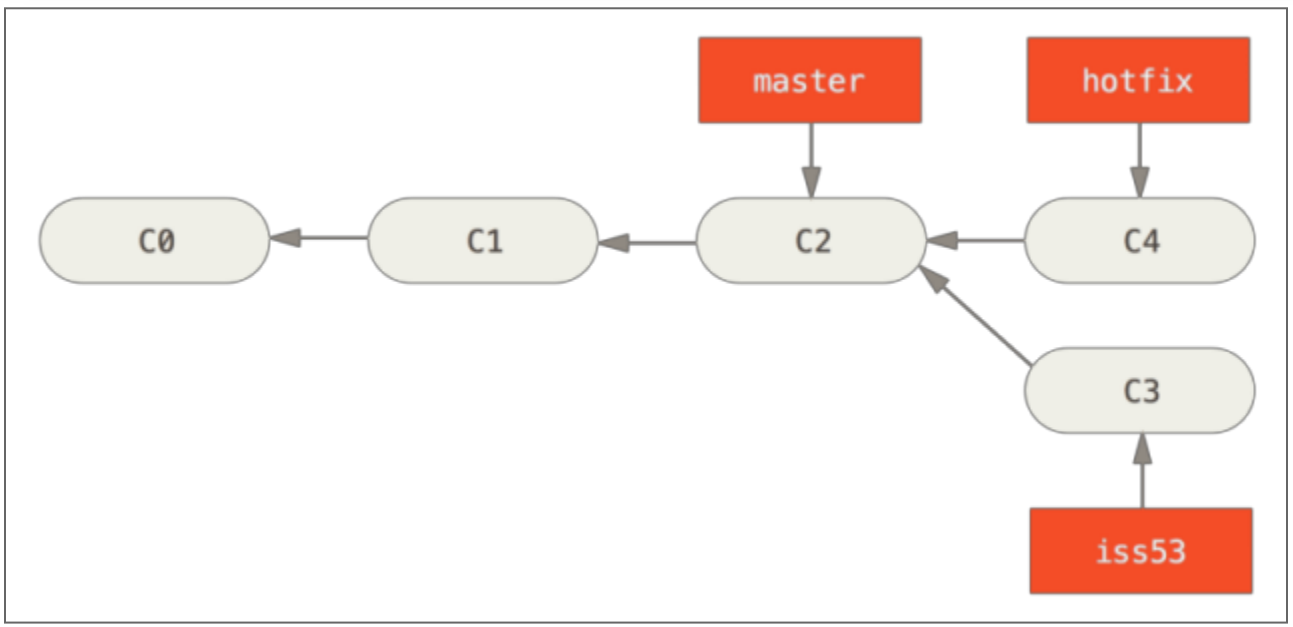
```
$ git checkout master
Switched to branch 'master'
```

Existe un puntero **HEAD** que indica en que lugar nos encontramos situados con el directorio de trabajo.

```
cat .git/HEAD
```

ARREGLAMOS EL ERROR

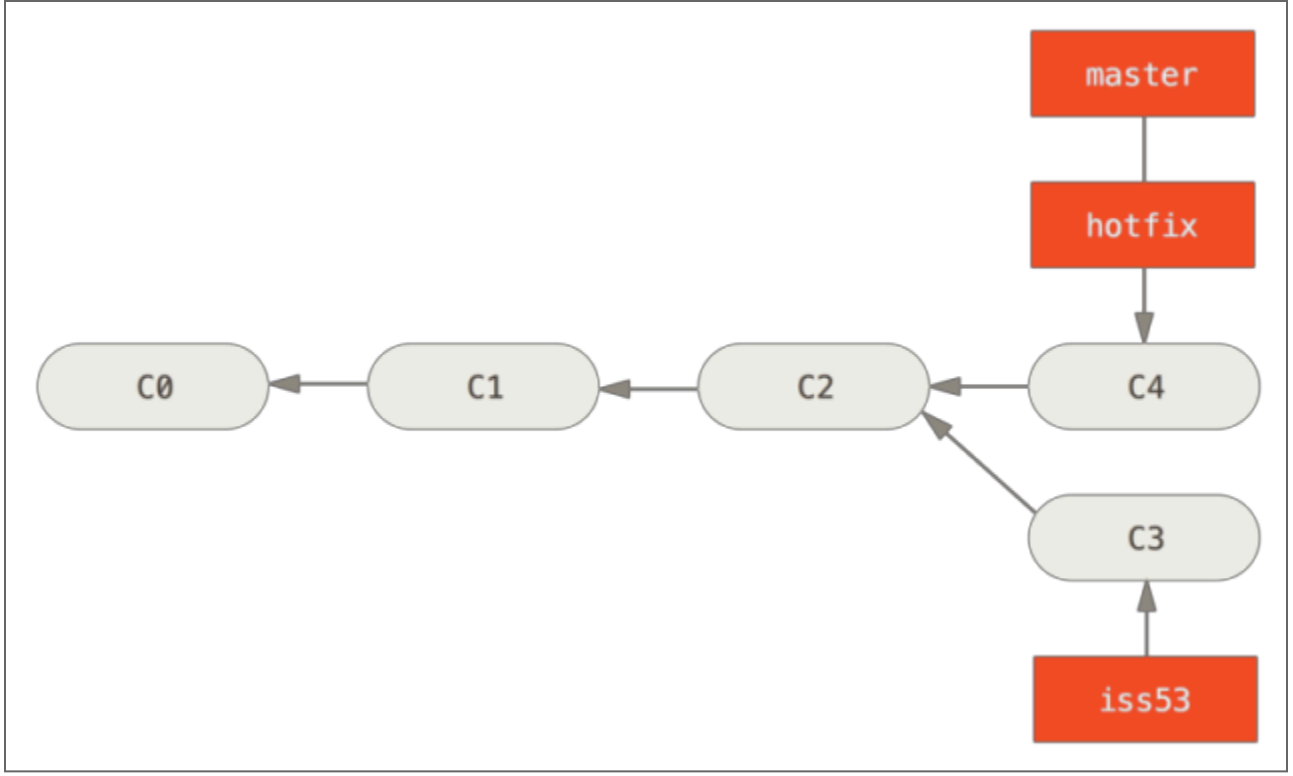
```
$ git checkout -b hotfix
Switched to a new branch 'hotfix'
$ vim index.html
$ git commit -a -m 'fixed the broken email address'
[hotfix 1fb7853] fixed the broken email address
 1 file changed, 2 insertions(+)
```



MERGEAMOS LOS CAMBIOS

Con **git merge** podemos incorporar los cambios a master.

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast-forward
 index.html | 2 ++
 1 file changed, 2 insertions(+)
```


master
hotfix
C0
C1
C2
C4
C3
iss53

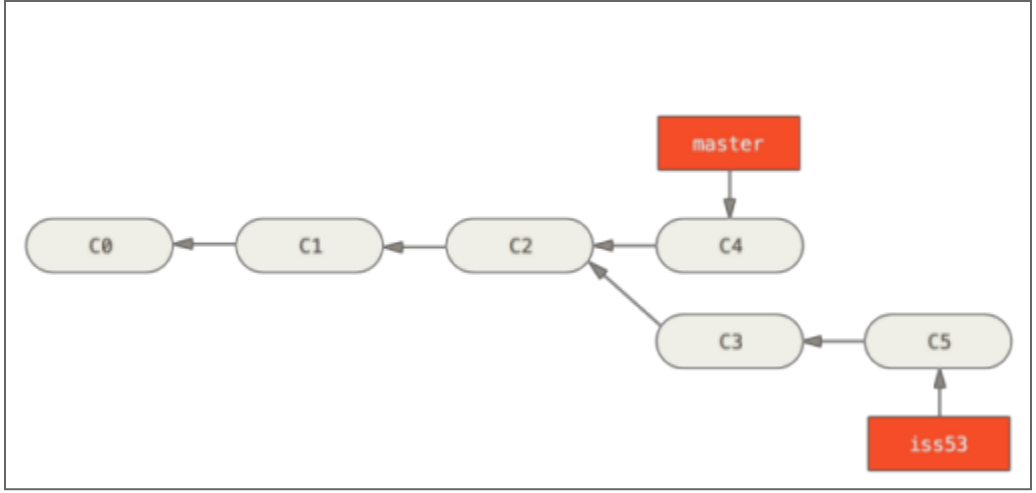
ELIMINANDO UNA RAMA

Las ramas ya mergeadas en master pueden ser eliminadas fácilmente.

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

Seguimos trabajando en la nueva funcionalidad:

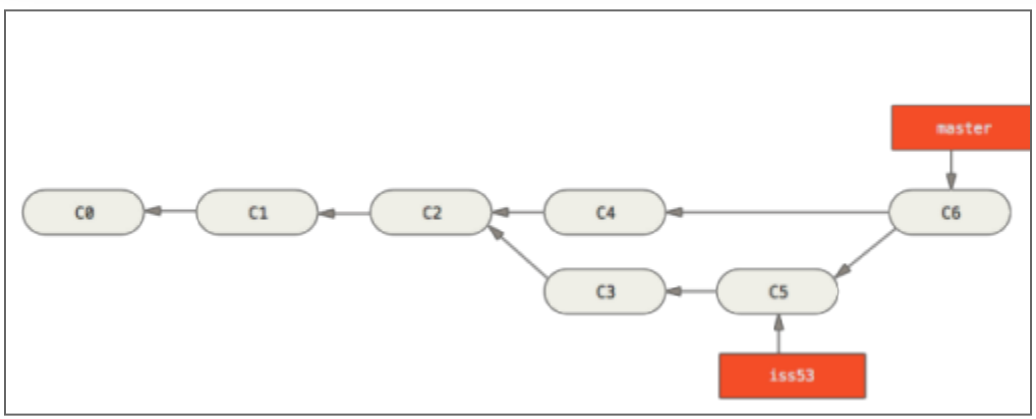
```
$ git checkout iss53
Switched to branch "iss53"
$ vim index.html
$ git commit -a -m 'finished the new footer [issue 53]'
[iss53 ad82d7a] finished the new footer [issue 53]
1 file changed, 1 insertion(+)
```


master
C0
C1
C2
C4
C3
C5
iss53

MERGEAMOS LA NUEVA FUNCIONALIDAD

Como el commit de la rama en la que nos encontramos no es un ancestro directo de la rama que queremos mergear, git debe generar un **nuevo commit** con toda la funcionalidad mergeada.

```
$ git checkout master
Switched to branch 'master'
$ git merge iss53
Merge made by the 'recursive' strategy.
index.html |    1 +
1 file changed, 1 insertion(+)
```



Esto puede generar conflictos que no se mergeen automáticamente.

INFORMACIÓN SOBRE RAMAS

Listas las ramas locales:

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

El * indica en que rama nos encontramos posicionados (HEAD).
Último commit de cada rama:

```
$ git branch -v
iss53    93b412c fix javascript issue
* master 7a98805 Merge branch 'iss53'
```

INFORMACIÓN SOBRE RAMAS LOCALES Y REMOTAS

Listado de ramas locales:

```
git branch -l
```

Listado de ramas remotas:

```
git branch -r
```

Todas las ramas:

```
git branch -a
```

ETIQUETAS (TAGS)

Git nos da la posibilidad de marcar o **taggear** puntos específicos dentro de la historia de nuestro repositorio. Lo más común es marcar las versiones del desarrollo.
Listado de tags:

```
$ git tag
v1.0
v2.0
```

CREANDO TAGS

Git soporta 2 tipos de tags: **livianos** o **anotados**.

- Los **tags livianos** son como ramas que nunca cambian, punteros a commits.

```
$ git tag v1.4-lw
```

- Los **tasg anotados**, en cambio, son guardados como un objeto completo en git. Se le aplica un checksum conteniendo el nombre del autor, su mail y la fecha, tiene un mensaje y pueden ser firmados y verificados con GNU Privacy Guard (GPG).

```
$ git tag -a v1.4 -m "my version 1.4"
```

VERSIONADO SEMÁNTICO

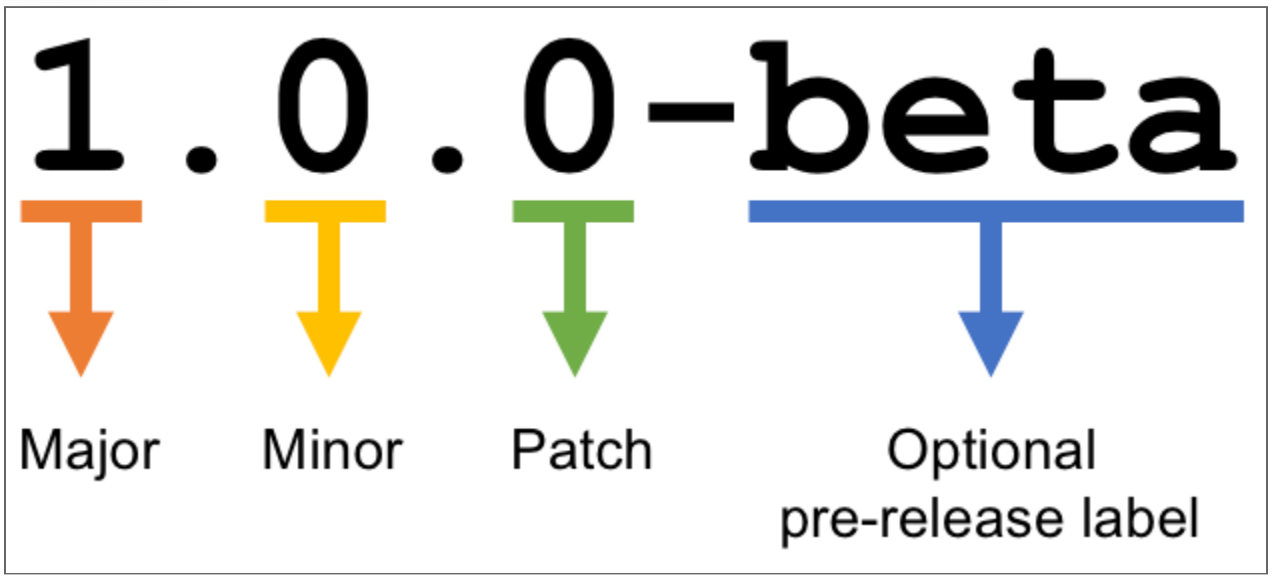
¿QUÉ ES?

*El **versionamiento semántico** es un convenio o estándar a la hora de definir la versión de tu código, dependiendo de la naturaleza del cambio que estás introduciendo.*

VERSIONES

- **Major**: cambio drástico en el software. No es compatible con código hecho en versiones anteriores.
- **Minor**: cambio que añade alguna característica nueva al software o modifica alguna ya existente, pero que sigue siendo compatible con código existente. También cuando marcamos algo como obsoleto.
- **Patch**: cuando arreglamos un bug siendo el cambio retrocompatible.
- **Identificadores de estabilidad**: además se suelen añadir unos identificadores que ayudan a marcar versiones específicas que quieres diferenciar, indicando la estabilidad de esa versión.

EJEMPLO



REFERENCIAS VERSIONADO SEMÁNTICO

- Especificación: **https://semver.org/lang/es/**
- Artículo: **https://blog.armesto.net/que-es-el-versionamiento-semantico-y-por-que-es-importante/**

GITFLOW

- Gitflow es un flujo de trabajo basado en ramas (branches) propuesto por Vincent Driessen en 2010.
- Propone una serie de "reglas" para organizar el trabajo del equipo.

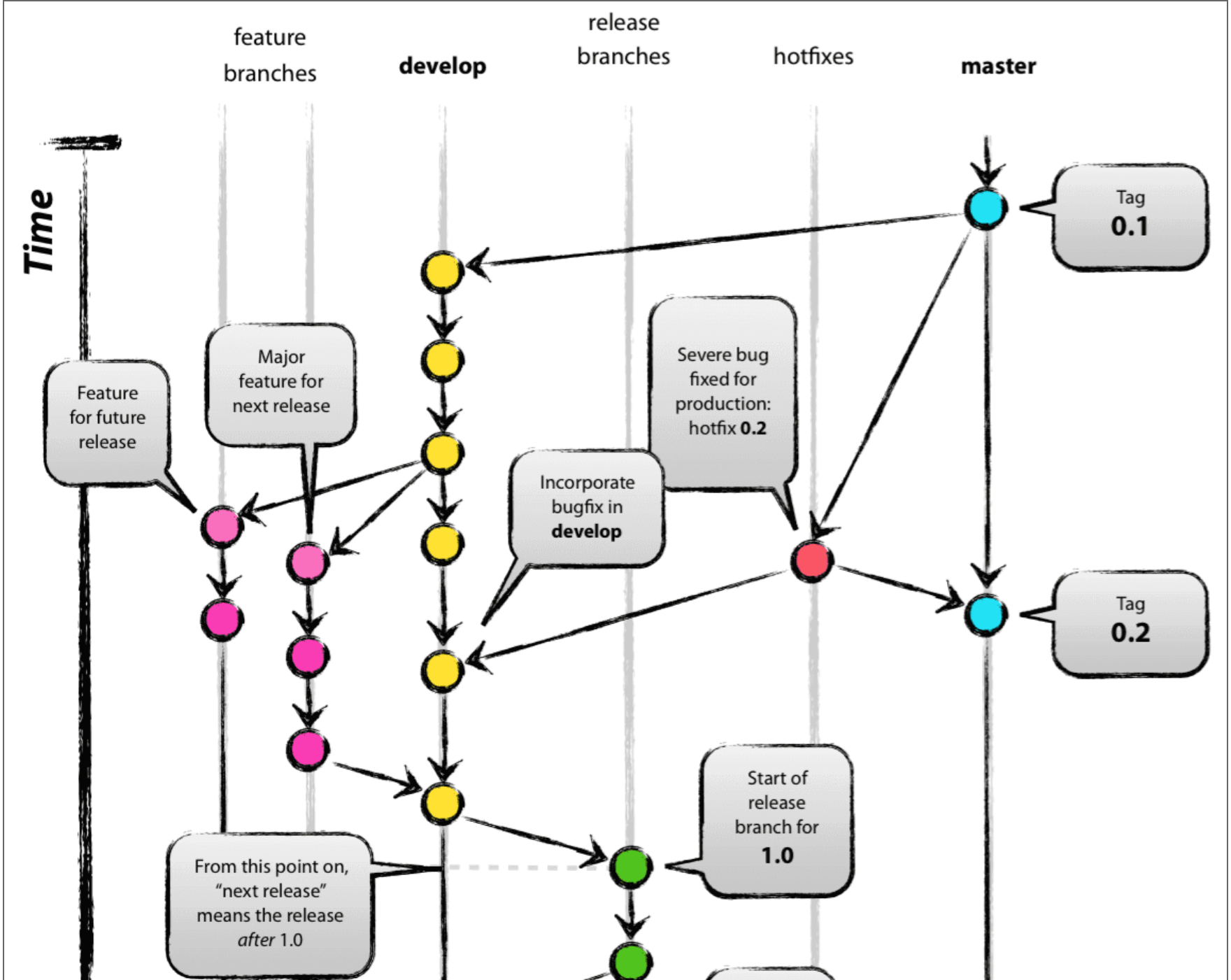
REGLAS DE GITFLOW: DOS RAMAS PRINCIPALES

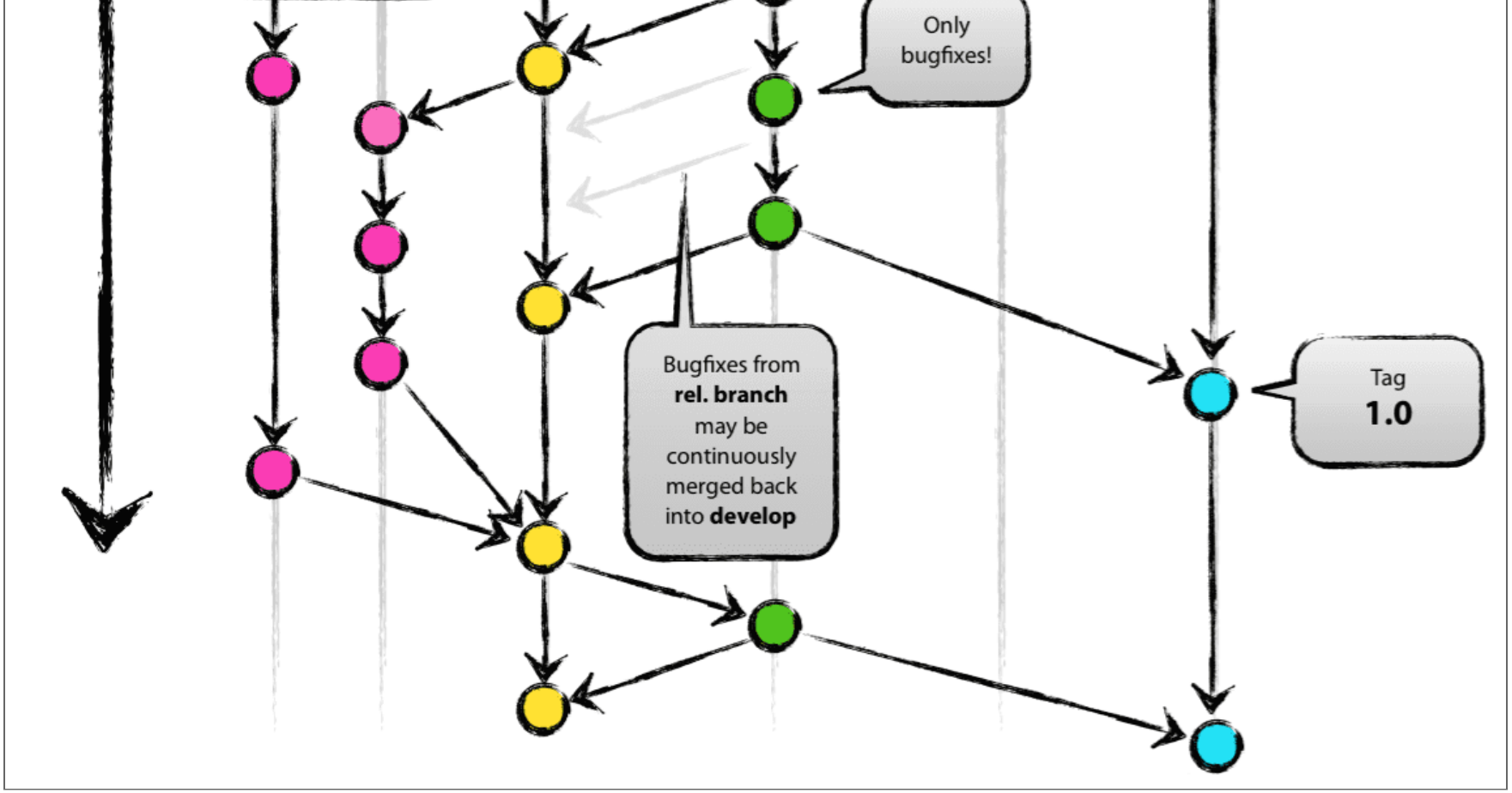
- **master**: cualquier commit que pongamos en esta rama debe estar preparado para subir a producción.
- **develop**: rama en la que está el código que conformará la siguiente versión planificada del proyecto.
  Cada vez que se incorpora código a master, tenemos una nueva versión.

REGLAS DE GITFLOW: RAMAS AUXILIARES

- **Feature**: se originan e incorporan siempre a **develop** , son las nuevas características de la app.
- **Release**: se originan en **develop** y se incorporan a **master** y **develop**. Se utilizan para preparar el siguiente código en producción.
- **Hotfix**: se originan en **master** y se incorporan a **master** y **develop**. Se utilizan para corregir errores y bugs en el código en producción.

> *Estas ramas auxiliares suelen desaparecer una vez incorporadas.*


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


Only
bugfixes!
Bugfixes from
rel. branch
may be
continuously
merged back
into **develop**
Tag
1.0

REFERENCIAS DE GIT-FLOW

- Blog de Vincent Driessen: **https://nvie.com/posts/a-successful-git-branching-model/**
- Extensiones de git: **https://github.com/nvie/gitflow**
- Git Flow Cheatsheet: **https://danielkummer.github.io/git-flow-cheatsheet/index.es_ES.html**

Speaker notes