

Sistemas de control de versiones: git

Rodrigo Lugones

rlugones@df.uba.ar

¿A qué nos referimos con “control de versiones”?

Se llama **control de versiones** a la gestión de los cambios que se realizan sobre documentos, programas o cualquier colección de información.

Ejemplos en la vida real:

- Cuaderno de laboratorio.
- Ediciones y revisiones de un libro.

¿Por qué realizar un control de versiones?

El control de versiones le da **trazabilidad** a nuestro trabajo.

¿Y? ¿Para qué queremos eso?

Curiosidad: ¿cuántas líneas de código manejamos?

- *Script* propio para calcular algunas cosas: ~ 100
- Código grande propio: ~ 1 mil
- Aplicación de celular: ~ 30 mil
- Transbordador espacial: ~ 400 mil

Curiosidad: ¿cuántas líneas de código manejamos?

- *Script* propio para calcular algunas cosas: ~ 100
- Código grande propio: ~ 1 mil
- Aplicación de celular: ~ 30 mil
- Transbordador espacial: ~ 400 mil
- Linux Kernel 2.2.0 (1999): $\sim 1,5$ millones
- Telescopio Hubble: ~ 2 millones
- Google Chrome: $\sim 6,5$ millones
- Mozilla Firefox: $\sim 9,5$ millones
- Android: ~ 12 millones
- Linux Kernel 3.1 (2011): ~ 15 millones

Curiosidad: ¿cuántas líneas de código manejamos?

- *Script* propio para calcular algunas cosas: ~ 100
- Código grande propio: ~ 1 mil
- Aplicación de celular: ~ 30 mil
- Transbordador espacial: ~ 400 mil
- Linux Kernel 2.2.0 (1999): ~ 1,5 millones
- Telescopio Hubble: ~ 2 millones
- Google Chrome: ~ 6,5 millones
- Mozilla Firefox: ~ 9,5 millones
- Android: ~ 12 millones
- Linux Kernel 3.1 (2011): ~ 15 millones
- Windows 7: ~ 40 millones
- Large Hadron Collider: ~ 50 millones
- Facebook: ~ 60 millones
- Software de un auto: ~ 100 millones
- Google: ~ 2000 millones

Fuente

Un ejemplo concreto

- Queremos simular un canal de agua con una columna en el centro.
- Desarrollamos un programa `NS_Solver` que resuelve la ecuación de Navier-Stokes con el método numérico de Euler en el paso temporal. El *output* del programa es, por ejemplo, la energía a cada tiempo.
- Ahora queremos graficar la energía en función del tiempo. ¿Cómo hacemos?

Opción 0: Modificar la carpeta en la que estamos trabajando.

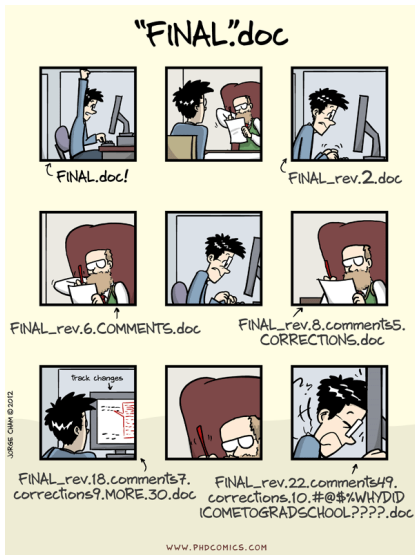
Un ejemplo concreto

Opción 0: Modificar la carpeta en la que estamos trabajando.

Pero si nos equivocamos y rompemos el código, ¿no querríamos poder volver fácilmente a la versión original?

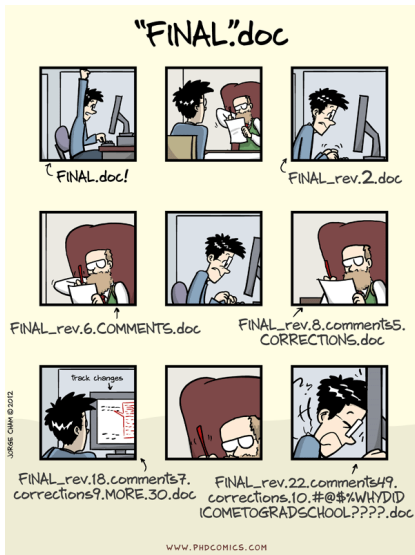
Un ejemplo concreto

Opción 1: Lo más común



Un ejemplo concreto

Opción 1: Lo más común



¿Qué contiene cada versión?

Un ejemplo concreto

Opción 2: Nombres descriptivos

```
NS_Solver_Euler  
NS_Solver_Euler_y_grafico_E
```

Opción 2bis: Archivo `versiones.txt` que registre los cambios realizados.

```
NS_Solver_1  
NS_Solver_2  
versiones.txt
```

```
$ cat versiones.txt  
1: Navier-Stokes con metodo de Euler  
2: Navier-Stokes con metodo de Euler y grafico de Energia.
```

Un ejemplo concreto

Este podría considerarse un casero y primitivo sistema de control de versiones.

Un ejemplo concreto

Este podría considerarse un casero y primitivo sistema de control de versiones.

Ahora, ¿qué pasaría si en la versión 19 del programa encontráramos un *bug* que existía desde la versión 1?

Un ejemplo concreto

Este podría considerarse un casero y primitivo sistema de control de versiones.

Ahora, ¿qué pasaría si en la versión 19 del programa encontráramos un *bug* que existía desde la versión 1?

Si bien podríamos pensar soluciones, vemos que los problemas se multiplican a medida que el proyecto crece. La solución es utilizar alguna herramienta especializada en controlar versiones.

Hay una infinidad de posibilidades (Mercurial, Subversion, etc). Nos vamos a concentrar en una, la más extendida: git

La ventaja de utilizar las herramientas adecuadas para el objetivo que uno quiere lograr es que, de alguna forma, favorecen (o hasta imponen) una filosofía de trabajo.

Con `git` vamos a poder:

- 1 Hacer backup de estados consistentes del proyecto
- 2 Documentar cambios
- 3 Seguir los bugs a través de la historia del desarrollo
- 4 Compartir cambios
- 5 Distribuir el desarrollo a muchas personas

Git: Una filosofía de trabajo

¿Y cómo funciona git?

Git: Una filosofía de trabajo

¿Y cómo funciona git?



Entonces, para que no suceda esto, primero definamos algunos conceptos claves.

- **Estados de un repositorio:** Son análogos a las carpetas en el ejemplo del SCV casero. Cuando terminamos de trabajar en un estado y lo *consolidamos* (en nuestra analogía, sería decir que terminamos de desarrollar la funcionalidad que queríamos en la carpeta y, entonces, no modificamos más esa carpeta) lo llamamos *snapshot*. El *snapshot* actual se llama HEAD.
- **Ciclo de vida de los archivos:** En un repositorio de git, cada archivo puede tener tres estados:
 - No-modificado
 - Modificado
 - Actualizado

Git: Una filosofía de trabajo

- Un archivo está en estado **no-modificado** cuando es exactamente igual al archivo que está guardado en el último *snapshot*.
- Modificar un archivo (por ejemplo, cambiar el nombre de una variable) lo transforma, evidentemente, en un archivo **modificado**.

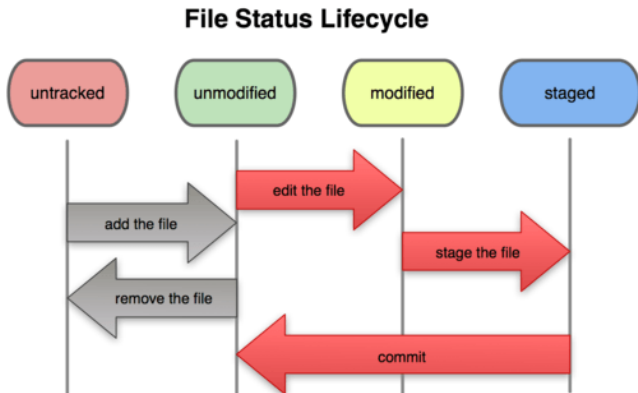
Git: Una filosofía de trabajo

- Un archivo está en estado **no-modificado** cuando es exactamente igual al archivo que está guardado en el último *snapshot*.
- Modificar un archivo (por ejemplo, cambiar el nombre de una variable) lo transforma, evidentemente, en un archivo **modificado**.
- Pero, y esto es muy importante, `git` no hace seguimiento a un archivo sólo porque está en estado modificado. Para que `git` se haga cargo del archivo modificado lo tenemos que **actualizar** (o, el término en inglés, *stage*). Con todos los archivos actualizados, podemos consolidar el cambio y, en consecuencia, tomar un nuevo *snapshot*.

- Un archivo está en estado **no-modificado** cuando es exactamente igual al archivo que está guardado en el último *snapshot*.
- Modificar un archivo (por ejemplo, cambiar el nombre de una variable) lo transforma, evidentemente, en un archivo **modificado**.
- Pero, y esto es muy importante, `git` no hace seguimiento a un archivo sólo porque está en estado modificado. Para que `git` se haga cargo del archivo modificado lo tenemos que **actualizar** (o, el término en inglés, *stage*). Con todos los archivos actualizados, podemos consolidar el cambio y, en consecuencia, tomar un nuevo *snapshot*.
- Al hacer esto, los archivos que estaban actualizados ahora forman parte del nuevo *snapshot*, que pasa a ser el nuevo HEAD del repositorio. Es decir que consolidar cambios actualiza automáticamente el HEAD del repositorio, y de esta manera los archivos que se encontraban en el estado actualizado pasan al estado no-modificado.

Git: Una filosofía de trabajo

- Finalmente, si creamos un archivo nuevo y le queremos hacer seguimiento, tenemos que agregarlo al repositorio. De la misma manera, podemos remover un archivo del repositorio para dejar de seguirlo.



Retomemos nuestro proyecto original, `NS_Solver`, pero desde el comienzo vamos a trabajar dentro de git.

Crear un repositorio

El primer paso es la ardua tarea de crear el repositorio. Para eso creamos una carpeta en la que queremos iniciar nuestro trabajo (y cambiamos el directorio a esa carpeta) y ejecutamos:

```
$ git init .
```

Esta carpeta va a ser considerada un repositorio de git.

¿Y dónde está guardada toda la información del repositorio?

```
$ ls -a  
.  
..  
.git
```

Estado de un repositorio

Para saber en qué estado se encuentra un repositorio:

```
$ git status
On branch master
No commits yet
nothing to commit (create/copy files and use "git add" to track)
```

En este caso, tanto el repositorio como el directorio están vacíos.

Agregar archivos al repositorio

Ahora creamos el archivo `navierstokes.py`, que resuelve el problema con el método de Euler.

```
$ ls
navierstokes.py

$ git status
On branch master
No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be committed)
   navierstokes.py
nothing added to commit but untracked files present (use "git add" to
track)
```

En el directorio aparece ese archivo, pero git no lo reconoce, pues nunca le dijimos que lo *siguiera*.

Repositorios de git: agregar archivos

Es decir, `navierstokes.py` no está siendo seguido. Si agregamos el archivo al repositorio (`git add`), pasa a estar actualizado:

```
$ git add navierstokes.py

$ git status
On branch master
No commits yet
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
   new file:   navierstokes.py
```

Repositorios de git: agregar archivos

Ahora *consolidamos* los archivos actualizados mediante `git commit`.

```
$ git commit -m "Agregado metodo de Euler"  
[master (root-commit) 13aa40c] Agregado metodo de Euler  
1 file changed, 0 insertions(+), 0 deletions(-)  
create mode 100644 navierstokes.py
```

Noten dos cosas:

- El *flag* `-m`, seguido del *mensaje de commit*. El mensaje de commit es **obligatorio**.
- El *commit hash* (número hexadecimal, 13aa40c), con el que podemos referirnos a este *snapshot*.

Repositorios de git: agregar archivos

Si ahora miramos el estado del repositorio,

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

El mensaje *working directory clean* nos dice que no hay archivos en estado *modificado*, ya que el archivo `navierstokes.py` (que no volvimos a tocar) es igual al que está guardado en `HEAD`.

A partir de acá, la tarea para generar un *snapshot* es siempre la misma:

- 1 Modificamos/creamos uno o varios archivos: pasan al *working directory*
- 2 Los agregamos al *staging area* con `git add`
- 3 Los consolidamos en un *snapshot* con `git commit`

Repositorios de git: continuar trabajando

Por ejemplo, creamos el archivo `graficar_E.py` para graficar la energía y modificamos el archivo `navierstokes.py`.

Si queremos ver qué diferencias entre la versión modificada de `navierstokes.py` y la que se consolidó en último *snapshot* (o sea, la versión en `HEAD`), simplemente escribimos

```
$ git diff navierstokes.py
diff --git a/navierstokes.py b/navierstokes.py
index 219e905..eb3f118 100644
--- a/navierstokes.py
+++ b/navierstokes.py
@@ -1,1 @@
-codigo viejo
+codigo nuevo
```


Repositorios de git: continuar trabajando

Y si ahora queremos consolidar los nuevos cambios en un nuevo estado del repositorio (*snapshot*), hacemos:

```
$ git add graficar_E.py navierstokes.py
$ git commit -m "Graficar energia"
[master 9b72f8b] Graficar energia
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 graficar_E.py
```

Repositorios de git: historia de los *snapshots*

Si queremos ver todos los *commits* que hemos hecho en el repositorio, usamos `git log`.

```
$ git log
commit 9b72f8bf4e05dd88ab8d365e712a3d6dfe372ca7 (HEAD -> master)
Author: Rodrigo Lugones <rodrigolugones@gmail.com>
Date: Sun Feb 25 12:28:43 2018 -0300
```

Graficar energia

```
commit 13aa40cbad74d2ecd8bb58452471cd79afb20e83
Author: Rodrigo Lugones <rodrigolugones@gmail.com>
Date: Sun Feb 25 12:26:19 2018 -0300
```

Agregado metodo de Euler

Repositorios de git: visitar *snapshots*

¿Y si queremos ir a un *snapshots* viejo? Simplemente ejecutamos

```
git checkout <commit hash>
```

```
$ git checkout 13aa
```

```
Note: checking out '13aa'.
```

You are **in** 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make **in** this state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may **do** so (now or later) by using **-b** with the checkout **command** again. Example:

```
git checkout -b <new-branch-name>
```

```
HEAD is now at 13aa40c Agregado metodo de Euler
```



Repositorios de git: visitar *snapshots*

Veamos qué hay en la carpeta.

Repositorios de git: visitar *snapshots*

Veamos qué hay en la carpeta.

```
$ ls  
navierstokes.py
```

¿¡CÓMO!? ¿¡Y DÓNDE ESTÁ `graficar_E.py`!? ¡Perdí dos semanas escribiendo el *script* para graficar!

Repositorios de git: visitar *snapshots*

Veamos qué hay en la carpeta.

```
$ ls
navierstokes.py
```

¿¡CÓMO!? ¿¡Y DÓNDE ESTÁ graficar_E.py!? ¡Perdí dos semanas escribiendo el *script* para graficar!

Recordemos que el archivo `graficar_E.py` no existía en este *snapshot*. El HEAD dejó de ser `9b72f8b` y pasó a ser `13aa40c`.

Si ahora queremos volver al *snapshot* donde está implementado el gráfico de la energía, ejecutamos

```
$ git checkout 9b72
Previous HEAD position was 13aa40c Agregado metodo de Euler
HEAD is now at 9b72f8b Graficar energia
$ ls
navierstokes.py  graficar_E.py
```

- **Los *commits* son análogos a nuestro SCV casero:** Desarrollar una nueva funcionalidad (*consolidar* una nueva carpeta en el SCV casero) lleva mucho tiempo. Entonces, tiene sentido hacer *commits* intermedios (no los hacíamos con el SCV casero porque es mucho laburo).

- **Los *commits* son análogos a nuestro SCV casero:** Desarrollar una nueva funcionalidad (*consolidar* una nueva carpeta en el SCV casero) lleva mucho tiempo. Entonces, tiene sentido hacer *commits* intermedios (no los hacíamos con el SCV casero porque es mucho laburo).
- **Identifiquen *commits* “interesantes”:** Vamos a tener ciertos *commits* que vamos a querer destacar (por ejemplo, nueva funcionalidad totalmente implementada). Para acceder a ellos fácilmente sin necesidad de recordar el **hash**, podemos *etiquetarlos*:

```
git tag v1.0 9b72
```


- Sean descriptivos con los *commit messages*:

	COMMENT	DATE
○	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
○	ENABLED CONFIG FILE PARSING	9 HOURS AGO
○	MISC BUGFIXES	5 HOURS AGO
○	CODE ADDITIONS/EDITS	4 HOURS AGO
○	MORE CODE	4 HOURS AGO
○	HERE HAVE CODE.	4 HOURS AGO
○	AAAAAAA	3 HOURS AGO
○	ADKFJ\$!LKDFJ\$DKLFJ	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS	2 HOURS AGO
○	HAAAAAAAAAANDS	2 HOURS AGO

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

- Sean descriptivos con los *commit messages*:

	COMMENT	DATE
○	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
○	ENABLED CONFIG FILE PARSING	9 HOURS AGO
○	MISC BUGFIXES	5 HOURS AGO
○	CODE ADDITIONS/EDITS	4 HOURS AGO
○	MORE CODE	4 HOURS AGO
○	HERE HAVE CODE	4 HOURS AGO
○	AAAAAAAAA	3 HOURS AGO
○	ADKFJ\$LKDFJ\$DKLFJ	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS	2 HOURS AGO
○	HAAAAAAAAAANDS	2 HOURS AGO

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

- **No hagan un seguimiento de todos los archivos:** Los archivos que seguimos son los archivos *fuentes* (el código, algún archivo de configuración), y no los archivos que se generan a través de la compilación o ejecución del programa. ¿Y cómo hacemos eso? Mediante la creación de un archivo, `.gitignore`

Repositorios de git: ignorar archivos

Algunas reglas para poner dentro de `.gitignore`:

```
$ cat .gitignore
# Un comentario. Esta linea es ignorada.
# Ningun archivo *.a
*.a

# Pero trackear lib.a, aunque este ignorando los archivos *.a
!lib.a

# Ignorar archivo TODO, pero no los subdir/TODO
/TODO

# Ignorar todos los archivos en el directorio build/
build/

# Ignorar archivos doc/*.txt, pero no los doc/server/arch.txt
doc/*.txt

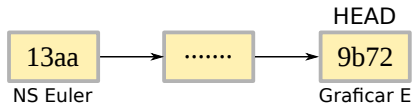
# Ignorar archivos .txt en el directorio doc/ y los subdirectorios
doc/**/.txt
```

Los grafos de git

Repositorios de git: la historia como un grafo

Con estas herramientas y conceptos fundamentales, podemos avanzar un poco más en el entendimiento y la visualización de la historia de un repositorio.

Usualmente, para ver la historia de un repositorio se utilizan grafos. En nuestro ejemplo, tendríamos la siguiente situación



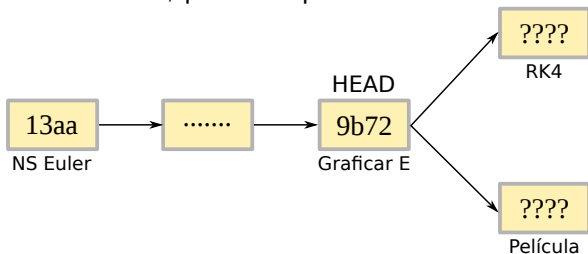
Esta visualización del repositorio nos da una mejor idea del desarrollo del código. Para visualizarla en la consola, escribimos `git log --graph`

Ramas

Complicuemos las cosas. Ahora queremos implementar dos funcionalidades “al mismo tiempo”:

- Crear una película que grafique el paso del agua
- Implementar el método numérico Runge-Kutta de orden 4 (en lugar de Euler)

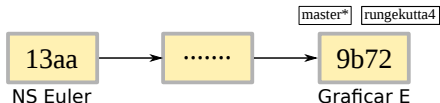
Gráficamente, podemos pensarlo así.



Repositorios de git: ramas

Ahora bien, ¿cómo lo hacemos? Mediante la utilización de *ramas* (branch). Para crear una rama llamada `rungekutta4`, ejecutamos

```
$ git branch rungekutta4
$ git branch                #esto imprime las ramas existentes
* master
  rungekutta4
```

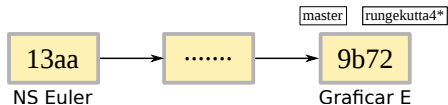


Repositorios de git: ramas

Para posicionarnos en la nueva rama, simplemente ejecutamos

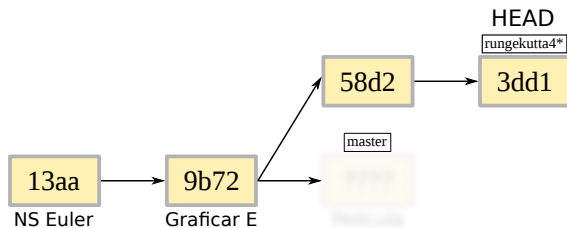
```
$ git checkout rungekutta4
$ git branch

master
* rungekutta4
```



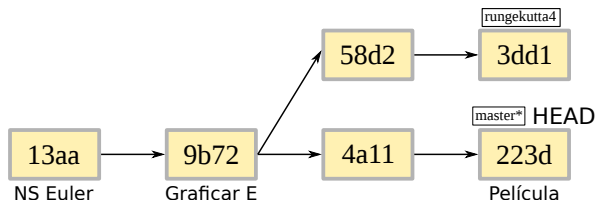
Repositorios de git: ramas

Continuemos con el desarrollo de nuestro código. Creamos el archivo `rk4.py` y empezamos a implementar el método. Observen que todos los *commits* los hemos hecho parados en la rama `rungekutta4`. Mientras, la rama `master` no ha tenido *commits*.



Repositorios de git: ramas

Volvemos a la rama `master` (`git checkout master`) y desarrollamos el código para que genere la película.



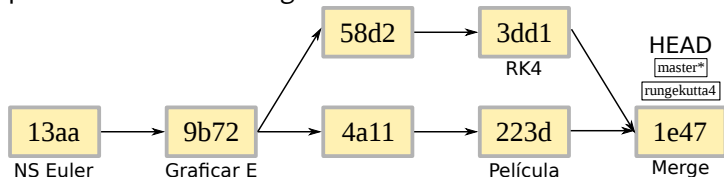
Repositorios de git: unir ramas

Entonces, conseguimos partir el desarrollo del código e implementar RK4, sin romper el desarrollo de la película.

El siguiente paso será juntar estas dos implementaciones (al fin y al cabo, es lo que queremos). Para eso, debemos *unir* las dos ramas.

```
$ git merge master rungekutta4
Merge made by the 'recursive' strategy.
 rk4.py | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 rk4.py
```

Unir las ramas exige un nuevo *snapshot*. O sea, cuando hacemos `merge` nos pide un *commit message*.

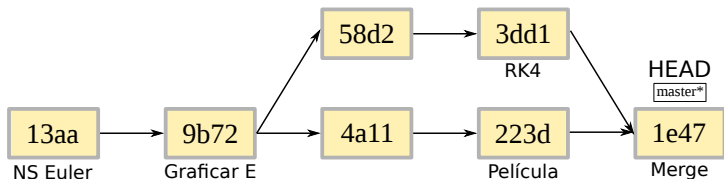


¿Pero cómo supo cómo unir las? Las líneas en las que no hay duda de cómo unir, las une sin preguntar. Si, en cambio, puede llegar a haber conflicto, pregunta.

Repositorios de git: unir ramas

Es buena costumbre borrar las ramas una vez que se dejaron de utilizar:

```
$ git branch -d rungekutta4
```



Veamos rápidamente todo esto en práctica.

Repositorios de git: Resumen

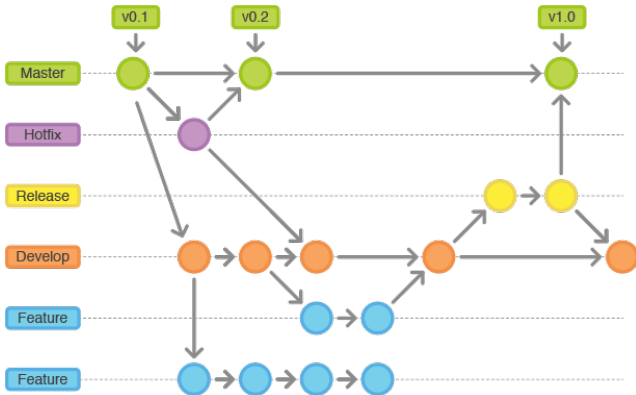
- `git init` # crea repositorio
- `git status` # estado del repositorio
- `git add` # agrega a stage
- `git commit` # guarda en repositorio lo que está en stage
- `git branch <rama>` # crea nueva rama
- `git branch` # lista las ramas existentes
- `git checkout <id>` # visita un determinado snapshot
- `git diff` # indica diferencias
- `git log` # muestra los commits hechos
- `git tag <new_tag> <hash>` # permite cambiar tags de los snapshots

... o cómo pensar el desarrollo del software

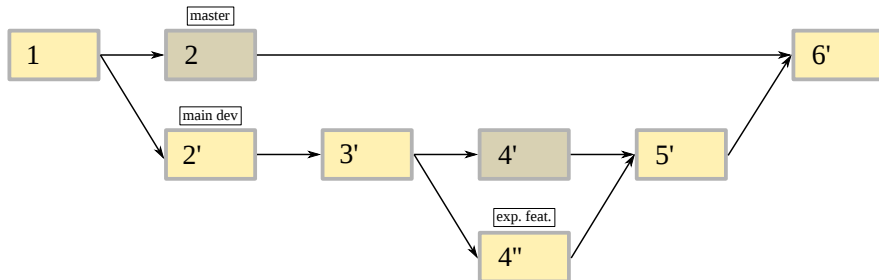


Flujo de trabajo

El ciclo de vida de una *release* de un programa se asemeja bastante al siguiente grafo. Analicémoslo un poco



¿Y por qué se trabaja de esa manera? Simplificadamente, porque los merge son complicados: puede surgir bugs al unir, y es más fácil encontrarlos y arreglarlos si los dos códigos que *mergeamos* son parecidos.

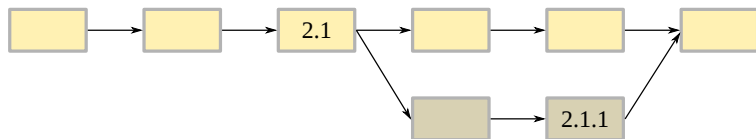


Flujo de trabajo



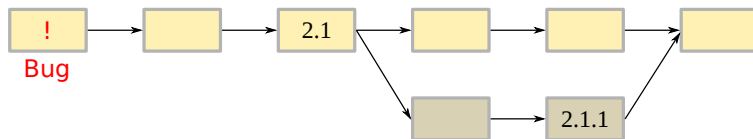
Flujo de trabajo: bugs

¿Y si en la versión 19 del programa nos damos cuenta de que arrastramos un bug desde la versión 1?

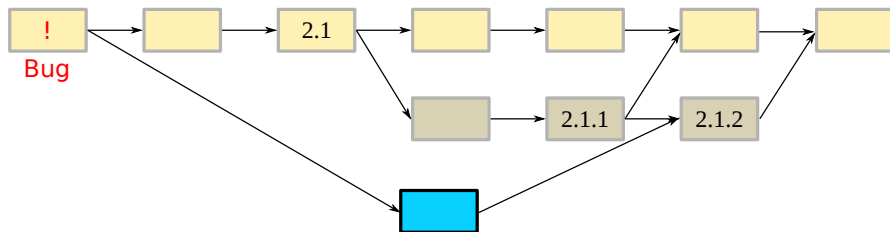


Flujo de trabajo: bugs

¿Y si en la versión 19 del programa nos damos cuenta de que arrastramos un bug desde la versión 1?



¿Y si en la versión 19 del programa nos damos cuenta de que arrastramos un bug desde la versión 1?



Creo una nueva rama para corregir el bug, y lo corrijo en donde aparece por primera vez. De esta forma, cuando uno ve la historia del repositorio, sabe entre qué versiones estaba presente el bug.

Trabajando a distancia

Repositorios remotos

Antes de hablar propiamente de los repositorios remotos, hablemos de cómo *forkear* un repositorio.

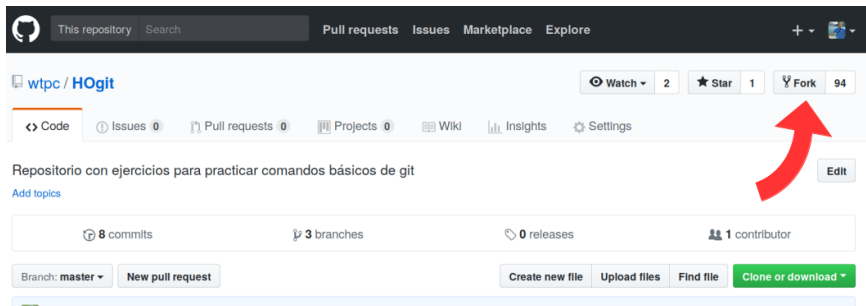
Un *fork* es una copia de un repositorio. *Forkear* un repositorio nos permite experimentar libremente con él sin afectar el proyecto original.

Generalmente, los *forks* se utilizan para proponer cambios en el repositorio de otra persona o para utilizar el proyecto de otro como punto de partida para una idea propia.



Repositorios remotos

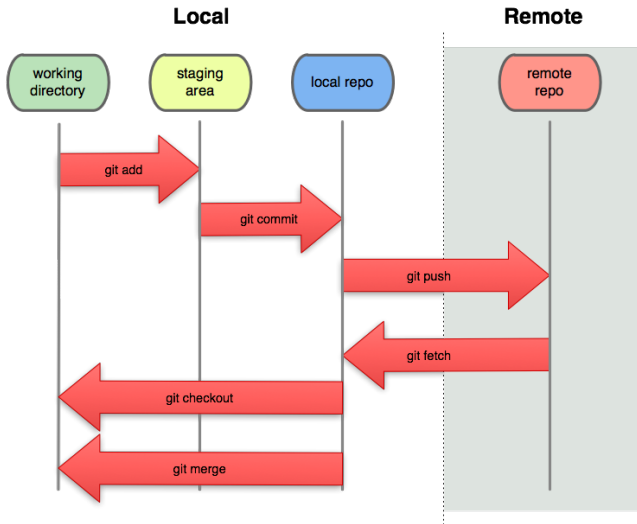
¿Y cómo se hacen?



The screenshot shows the GitHub interface for the repository 'wtpc / HOgit'. At the top, there are navigation tabs: 'This repository', 'Search', 'Pull requests', 'Issues', 'Marketplace', and 'Explore'. Below these, the repository name 'wtpc / HOgit' is displayed, followed by statistics: 'Watch 2', 'Star 1', and 'Fork 94'. A red arrow points to the 'Fork' button. Below the repository name, there are tabs for 'Code', 'Issues 0', 'Pull requests 0', 'Projects 0', 'Wiki', 'Insights', and 'Settings'. The repository description reads: 'Repositorio con ejercicios para practicar comandos básicos de git'. Below this, there are statistics: '8 commits', '3 branches', '0 releases', and '1 contributor'. At the bottom, there are buttons for 'Branch: master', 'New pull request', 'Create new file', 'Upload files', 'Find file', and 'Clone or download'.

Repositorios remotos

Flujo de trabajo



Repositorios remotos

Los comandos para comunicarse con un repositorio remoto son simples:

```
$ git clone https://github.com/<USUARIO>/H0git.git
```

```
$ git push
```

```
$ git pull
```

Una observación: `git pull` incorpora cambios de un repositorio remoto en la rama actual. En su implementación por defecto, `git pull` es simplemente `git fetch` seguido de `git merge FETCH_HEAD`

Sistemas de control de versiones: git

Rodrigo Lugones

rlugones@df.uba.ar