

# Proyecto Grupal - Recap 01 - Git

Collapse context

←

Git es un sistema de control de versiones que mantiene registro de los cambios hechos sobre uno o varios archivos y es uno de los mas usados en la industria.

No veremos aqui cosas de como hacer un commit ni como hacer un push ni un release ya que eso esta cubierto en [la guia electronica](#)

Lo que veras en este *recap* es como usar el histórico de Git para poder encontrar las cosas más rápido, incluyendo bugs.

Your answer passed the tests! Your score is 100.0%. [Submission #6525b88d976761c1fb173035]

×

## Question 1:

Clonate el repositorio <https://github.com/Taller-de-Programacion/hands-on-git.git> con `git clone https://github.com/Taller-de-Programacion/hands-on-git.git`

En Git los cambios sobre el repositorio son llamados *commits*. Un *branch* (rama) es una secuencia de commits que representa una línea de desarrollo.

Cambia al branch `good-quality-commits` con corriendo:

```
git checkout good-quality-commits
```

O bien corriendo

```
git switch good-quality-commits
```

Podés verificar en que branch estas actualmente con

```
git status
```

Cual es el mensaje del último commit de este branch?

Podés ver el último commit con `git log -1` o bien con `git show HEAD`

- ☒ "chore: pre-commit hook para cppcheck (linter)"
- ☐ "Documentacion y tests del programa factorial (TDD)"
- ☐ "fix: arreglo bug"
- ☐ "build: pre-commit hook para cpplint"

## Question 2:

Un commit esta compuesto por un mensaje, un diff y un hash (entre otras cosas).

Ya viste el mensaje de un commit y tambien habras visto su hash.

Veras q cada commit en Git tiene un *hash* único, algo de la forma `60b4f8c652b45eddc52a77f0b306ddb79e2678c5`. Muchos comandos reciben un hash como ese aunque es común usar solo los primeros dígitos.

Cual es el hash mensaje del último commit de este branch?


- ☐ `d64c4b87209ab696884c6453ca977db0d6e31dc5`
- ☐ `7a5ab38f4c644710bd3f3eed69d88ddb28116820`
- ☒ `500e41e028db1ed2af38becde2dfc1280ba458a9`

## Information


|                  |                     |
|------------------|---------------------|
| Author(s)        | Martin Di Paola     |
| Deadline         | 24/10/2023 18:00:00 |
| Status           | Succeeded           |
| Grade            | 100%                |
| Grading weight   | 1.0                 |
| Attempts         | 2                   |
| Submission limit | No limitation       |

## Submitting as

➤ AbrahamOsco 102256

 [Classroom : Default classroom](#)

## For evaluation

 Best submission

➤ [10/10/2023 17:48:13 - 100.0%](#)

## Submission history

|                              |
|------------------------------|
| 10/10/2023 17:48:13 - 100.0% |
| 10/10/2023 17:43:23 - 85.71% |

☐ 446c0b84389808c5ed4a379b53740d9b5b4f2392

Question 3:

Como se menciono anteriormente un commit esta compuesto por un mensaje, un diff y un hash (entre otras cosas).

Este *diff* es el cambio que el commit introduce o *patch* (parche).

Por default `git log` no te muestra el diff pero basta con agregar `--patch`

Corre `git log -1 --patch` y mira el patch.

Cual de las siguientes afirmaciones son correctas. Puede haber más de una!

- ☒ Se agrego un nuevo hook para `cppcheck` version `v1.3.5` (especificado con un hash).
- ☐ Se modifiko el archivo `.pre-commit-config.yaml` que esta en la carpeta `b/` del repo.
- ☐ El hook agregado esta configurado para operar sobre C++11.
- ☐ Se agrego un nuevo hook para `cpplint` version `1.6.0`.
- ☐ Se modifiko el archivo `.pre-commit-config.yaml` que esta en la carpeta `a/` del repo.
- ☐ El hook agregado esta configurado para operar sobre C++14.
- ☒ Se modifiko el archivo `.pre-commit-config.yaml` que esta en la carpeta raiz del repo.

Question 4:

Si sabes el hash de un commit podes ver su mensaje y diff con `git show` No es algo que usaras en el dia a dia pero hay veces que sirve.

Corre `git show --patch 612ed061`

Cual de las siguientes afirmaciones son correctas. Puede haber más de una!

Nota: `git show` tambien opera sobre branches y tags, no solo sobre hashes.

- ☐ Se modifiko el archivo `.pre-commit-config.yaml` que esta en la carpeta `a/` del repo.
- ☐ Se modifiko el archivo `.pre-commit-config.yaml` que esta en la carpeta `b/` del repo.
- ☒ Se agrego un nuevo hook para `cpplint` version `1.6.0`.
- ☐ El hook agregado esta configurado para operar sobre C++14.
- ☒ Se modifiko el archivo `.pre-commit-config.yaml` que esta en la carpeta raiz del repo.
- ☐ Se agrego un nuevo hook para `cppcheck` version `v1.3.5` (especificado con un hash).
- ☐ El hook agregado esta configurado para operar sobre C++11.

Question 5:

Con `git log` podes ver todos los commits del branch `good-quality-commits` y con `git log --patch` podes verlos junto a sus diffs.

Notaras que estos commits tienen *buenos mensajes* descriptivos de cada uno de los cambios agregados.

Buscá en que commit se implementó la función `factorial`. O sea, en que commit aparecio por primera vez la función.

Podrías buscar commit por commit y ver el diff de cada uno de ellos hasta ver cual commit implementó **factorial**.

Pero por suerte en el branch hay **buenos commits**: commits que tienen buenos mensajes que explican no solo **que** cambio introducen sino también el **por que**.

Con **git log** podrás entonces simplemente **leer los mensajes** de los commits y no tendrás que revisar los diffs.

Referencias:

- [Recomendaciones de como escribir un commit](#)

- ☐ 07082a9e02d4ef4b09dcc7d10510d0454c7043ba
- ☐ bbe6cab231dd604a646511106d030adc77925cd0
- ☐ 774d083983ef1bb7ae20e1c099d44152fd62d7c7
- ☐ d8f889aba04a1489fe6db3771e4a1d6de8ce2bed
- ☒ 69eba3d9b3ce72eab23487debab1276d7e5b23a5
- ☐ 4bb695aa1bf4db1e7dcf5019daaa9d9ae7b522af

Question 6:

Proba ahora en ver en que commit se *fixeo* el buffer overflow.

Una vez mas podrás usar **git log** pero he aquí un truquillo: con **git log --grep=<text>** podes buscar los commits que tenga **<text>** en sus mensajes.

Como los commits son de buena calidad es muy probable que el commit que fixea el buffer overflow **diga** que lo fixea (y probablemente diga el **por que**).

Por ejemplo: **git log --grep=overflow**

**git log --grep** es una manera increiblemente rapida de ir y ver donde se cambio algo. Cuando trabajes en equipo (y si escriben buenos commits) esto puede ahorrarte mucho tiempo.

Entonces, que commit es?

- ☐ 612ed0612a42f60a46e520fb3196e3f60aaeccce
- ☐ bbe6cab231dd604a646511106d030adc77925cd0
- ☐ d8f889aba04a1489fe6db3771e4a1d6de8ce2bed
- ☒ 9182fc860c0f1e1e7fe9e6ca81bc1173c1f0ea2b
- ☐ f9d23f19fdd620fec8e65aef6ad0de3c43a41e66
- ☐ 7ce12cc33daf83312359c941e928067e7c45f1a3

Question 7:

Pasemos ahora al branch **poor-quality-commits** con **git switch** (verifica que estas en ese branch con **git status**)

Buscá en que commit se implementó la función **factorial**. O sea, en que commit aparecio por primera vez la función.

Veras que ni **git log** ni **git log --grep** te servirán ya que todos los commits tienen un **muy pobre** mensaje que **no explican** nada.

Por suerte aun te queda una carta bajo la manga. **git log -G<text> --patch** va buscar **<text>** no en los mensajes sino en los diffs.

Probá entonces `git log -Gfactorial --patch` No te salvarás de revisar varios diffs pero al menos no tendrás que revisar absolutamente todos.

`git log -G` es menos preciso que `git log --grep` pero es super util cuando tenes que saber si un patch esta o no en tu branch. No te olvides que cuando trabajes en equipo no siempre sabras que esta o no en un branch compartido como `main`.

Una moraleja: invertir 2 minutos para **escribir commits de buena calidad da sus frutos**.

- ☐ `c8ed07beb09eddddf3536b80e141239bb92ca703`
- ☐ `ad3e254f5da42959bcfe8a223b433329171ec3e5`
- ☐ `5f986d380e82c9287a23ee32c4ab11ad3be3575e`
- ☐ `d352dbec175b91ef346a3545d1801e2db98dbdf7`
- ☐ `4e8c74ba582c1631efe73a3710b5b1aa754a8489`
- ☒ `616a1df9484507a8c65005758cd63c91e23707a8`

Question 8:

La documentación del proyecto (`doctests/factorial.md`) funciona no solo como documentación sino también como tests automáticos.

[byexample](#) es una tool que toma los snippets de código de la documentación (los ejemplos), los ejecuta y compara la salida con lo que el ejemplo muestra.

Si la comparación falla habrás encontrado un bug.

Para poder correr los tests necesitas tener Python 3 instalado y `pip`. En Debian/Ubuntu podes hacer `apt-get install python3 python3-pip`. Luego instalate `byexample` corriendo `pip install byexample`.

Asegúrate que estas en el branch `poor-quality-commits` y corre `make tests`

Cual de las siguientes afirmaciones son correctas? Puede haber más de una!

Referencias:

- ☒

El test/ejemplo muestra que correr `./bin/factorial 0` deberia dar 1 (expected) pero dio 0 (got).
- ☐

El make ejecuto el comando `byexample -l cpp factorial/main.cpp`
- ☒

`byexample` dice que en total pasaron 4 tests (PASS) y que 1 fallo (FAIL).
- ☐

`byexample` dice que en total pasaron 6 tests (PASS) y que 1 fallo con error (ERROR).
- ☒

El test/ejemplo que esta fallando esta en la linea 16 de `doctests/factorial.md`.
- ☒

El make ejecuto el comando `byexample -l shell doctests/factorial.md`

Question 9:

Al parecer hay un factorial que no esta bien calculado...

Podrías ponerte a debuggear, si, pero tal vez podemos usar Git a nuestro favor.

Esta es la idea: en algún momento todos los tests pasaban, luego alguien introdujo el bug y a partir de ahí los tests quedaron rotos.

Si es así podríamos ir commit a commit del branch `poor-quality-commits` y correr en cada paso `make tests`: en el momento que los tests dejen de fallar (o empiecen a fallar) ya sabras que **ese commit** metió el bug.

Tal vez no te diga exactamente cual es el bug pero tendrás **mucho mas contexto** ya que el bug tal ves este en el patch y con solo verlo te des cuenta o al menos te de pistas/ideas (lo que ayuda muchísimo al debugging posterior)

Ahora, ir de a un commit a la vez es una búsqueda secuencial y ya sabras que hay algo mucho mejor: **búsqueda binaria** y Git ya la implementa por vos con `git bisect` !

Corre estos 2 comandos para iniciar el `bisect mode` y marcar el commit actual como *"malo"*.

```
git bisect start
git bisect bad
```

Ahora, con este comando le decís a Git que commit es el *"bueno"*.

```
git bisect good <hash>
```

En `<hash>` tenes que poner el hash de algún commit que sepas que los tests pasaban. Por suerte hay un commit así. Fijate con `git log` q hay un commit que dice *"arreglo bug, vamos que los tests pasan!!"*, usa el hash de ese commit. (Tip: el hash empieza con `9c...`)

A partir de acá Git te va a switchear al commit punto medio entre el *"bueno"* y el *"malo"*.

Lo que tenes que hacer es correr `make tests` y si los tests pasan marcas el commit como bueno (`git bisect good`) o de lo contrario como malo (`git bisect bad`).

Así Git sabe cual es el siguiente commit medio y te hace el switch. Continuas así hasta encontrar el commit que introdujo el bug.

Cuando termines tenes que hacer `git bisect reset` para limpiar todo el estado de Git.

Parece complicado pero pensalo. Si alguien rompe algo en un commit pero nadie se da cuenta hasta semanas despues, saber que commit es el malo es super útil!

Habiendo dicho todo esto, que commit introdujo el bug?

- ☐ `446c0b84389808c5ed4a379b53740d9b5b4f2392`
- ☒ `fa9af0155756f1c5a9d3def8d8a7a000e5951786`
- ☐ `d352dbec175b91ef346a3545d1801e2db98dbdf7`
- ☐ `02f5255d4071b4f5c85b63dcc4da50ccee231dda`

Question 10:

Cuando trabajas en equipo es útil ver que cambios están en un branch respecto a otro.

Típicamente vas a querer comparar que features/fixes un branch `foo` esta a punto de introducir al branch `main` haciendo `git log main..foo` o `git diff main foo`.

Con `git log main..foo` ves que commits están en `foo` pero no aun en `main`.

Con `git diff main foo` ves que cambios (patch) `foo` agregaría a `main`.

Proba en ver que cambios hay entre `poor-quality-commits` y `good-quality-commits` con `git diff poor-quality-commits good-quality-commits`

Fijate que hay de diferente en la funcion `factorial` que hace que en `good-quality-commits` pasen los tests pero en `poor-quality-commits`

Podes ayudarte viendo solo los cambios sobre `factorial/main.cpp` corriendo `git diff poor-quality-commits good-quality-commits -- factorial/main.cpp`

Super tip: podes correr `git difftool --tool=<xxx> poor-quality-commits good-quality-commits -- factorial/main.cpp` Funciona exactamente igual que `git diff` solo que `git difftool` usa una herramienta externa (`<xxx>`) para computar el diff y verlo. Yo personalmente uso mucho `meld` para ver los diffs.

Nota: "*en este branch anda pero en este no*", con `git diff` podras saber el por que.

Referencias:

- [meld](#)

- ☐ La funcion `main` tiene un `if (num == 0)` antes de llamar a `factorial`
- ☐ La funcion `factorial` tiene un `if (num == 0)` que maneja correctamente el caso `factorial(0)`
- ☐ La funcion `main` tiene un `if (num <= 1)` antes de llamar a `factorial`
- ☒ La funcion `factorial` tiene un `if (num <= 1)` que maneja correctamente el caso `factorial(0)`

Question 11:

Es correcto `if (num <= 1)` o debería ser `if (num == 0)` ?

No es raro a veces ver un código sospechoso y preguntarse, quien lo escribió y por que?

Cuando queremos ver que commit introdujo una linea de código en particular podemos usar `git blame` (`git log -G` serviría también pero hay veces que `git log -G` es demasiado impreciso).

Anda al branch `good-quality-commits` y corre `git blame factorial/main.cpp`

Que commit metió ese `if (num <= 1)` (línea 43) tan sospechoso?

Gracias a que el commit es de buena calidad podrás hacer luego un `git show <hash>` y saber entonces el **por que si el código es correcto o no** con solo entender el mensaje del commit.

- ☐ `45ae01cfd594b93776e9c5c7f3440f362013ae55`
- ☐ `f9d23f19fdd620fec8e65aef6ad0de3c43a41e66`
- ☒ `7ce12cc33daf83312359c941e928067e7c45f1a3`
- ☐ `bbe6cab231dd604a646511106d030adc77925cd0`

Question 12:

Git permite hookearse y ejecutar un script en distintas etapas del versionado.

[pre-commit](#) es una administrador de hooks que te permite instalar y correrlos fácilmente.



Necesitas tener Python 3 y `pip` y luego correr `pip install pre-commit`

Andate al branch `good-quality-commits` e instala los hooks corriendo:

```
pre-commit install
pre-commit install --hook-type commit-msg
```

El branch `good-quality-commits` fue agregando y configurando varios hooks: `cpplint`, `cppcheck` y `commitizen`.

Para `cppcheck` necesitas tener el programa `cppcheck` instalado. Podes tenerlo con `apt-get install cppcheck`.

Proba en modificar `factorial/main.cpp` dejando alguna variable sin definir o sin usar e intenta commitearlo.

Hacé

```
git add factorial/main.cpp
git commit
```

Si commiteas debería `cppcheck` correr automáticamente y fallarte el commit.

Puede parecerte tedioso pero pensalo, al detectar rápido algo que esta mal, es mucho mas fácil arreglarlo ahí, en el momento, en vez de arreglarlo 2 semanas después.

Nota: `commitizen` es para personas algo obsesivas y es opcional, pero `cpplint` y `cppcheck` son virtualmente obligatorios (y el Sercom hara las mismas verificaciones).

Referencias:

- [pre-commit](#)
- [commitizen](#)
- [cpplint](#)
- [cppcheck](#)

- ☒ Instale y pude ver que el hook `cppcheck` me aviso del error que introduje en el codigo.
- ☐ No instale ningun hook.

Question 13:

Si corres `git commit` te abrirá un editor por default, típicamente `nano`, `emacs` o `vim`.

Puede que te tientes y hagas `git commit -m "mi mensaje aqui"` que es más fácil.

No.

Usar `git commit -m "mi mensaje aqui"` solo lleva a que hagas commits de baja calidad.

Si el editor por default que abre Git no te gusta, cambialo!!

Git permite cambiar el editor al que quieras, solo tendras que buscar en Google como hacerlo exactamente ya que cada editor puede requerir un tweak o 2 para funcionar.

Aquí Google is your friend.

- ☐ No tengo ningun editor y usare `git commit -m "mi mensaje aqui"`.
- ☒ Tengo configurado Git para que use mi editor de preferencia.

Question 14:

En este *hands-on* no te mostre las cosas elementales de Git como `git commit` o `git push`.

Preferí ver algo que no siempre se aprecia y es la forma que Git nos permite explorar el pasado, que cambios metiste vos o tu compañero y que es relevante en el presente.

Te aconsejo aca que te armes un **cheatsheet** de Git con lo visto hasta aqui.

Toma papel y lapiz (o armate un file) y anotá:

- Ver los cambios del branch: `git log` y `git log --patch`
- Buscar un commit: `git log --grep=<text>` (en el mensaje) y `git log -G<text> --patch` (en el patch)
- Buscar bug: (1) `git bisect start`, (2) `git bisect bad`, (3) `git bisect good <hash>`, (4..`n`) `git bisect [good/bad]`, (n+1) `git bisect reset`
- Ver diff entre branches: `git diff A B` o `git difftool --tool=<tool> A B`
- En cada repo configurar: editor para `git commit`, diff para `difftool` y hooks que me sirvan (linters y otros)

- ☒ Me arme el **cheatsheet** y lo tendre a mano siempre (tal vez un sticker en la pared o notebook?).
- ☐ No arme el **cheatsheet**.

Submit