

Git 1 comandi base

risorse.students.cs.unibo.it/lab/2023

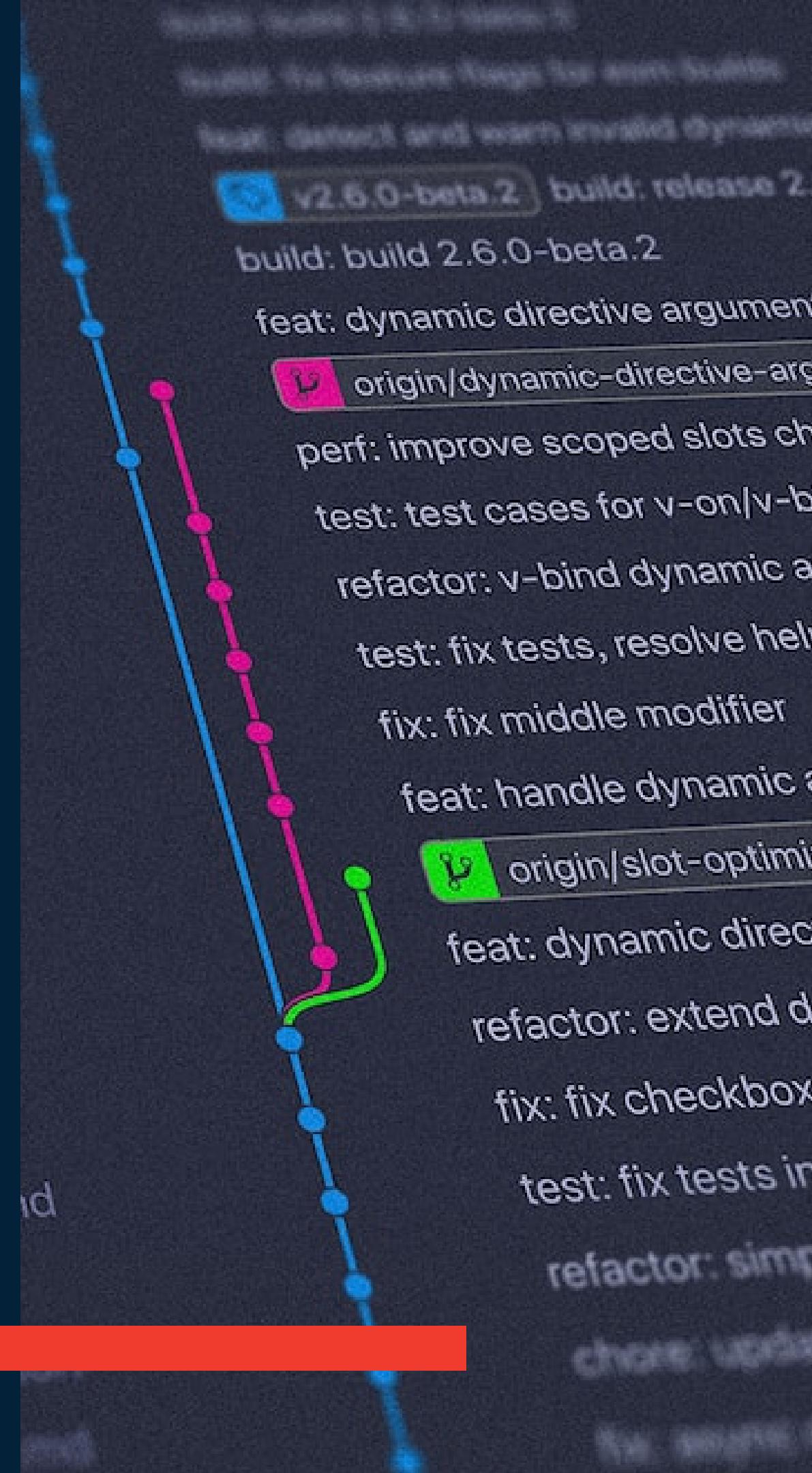
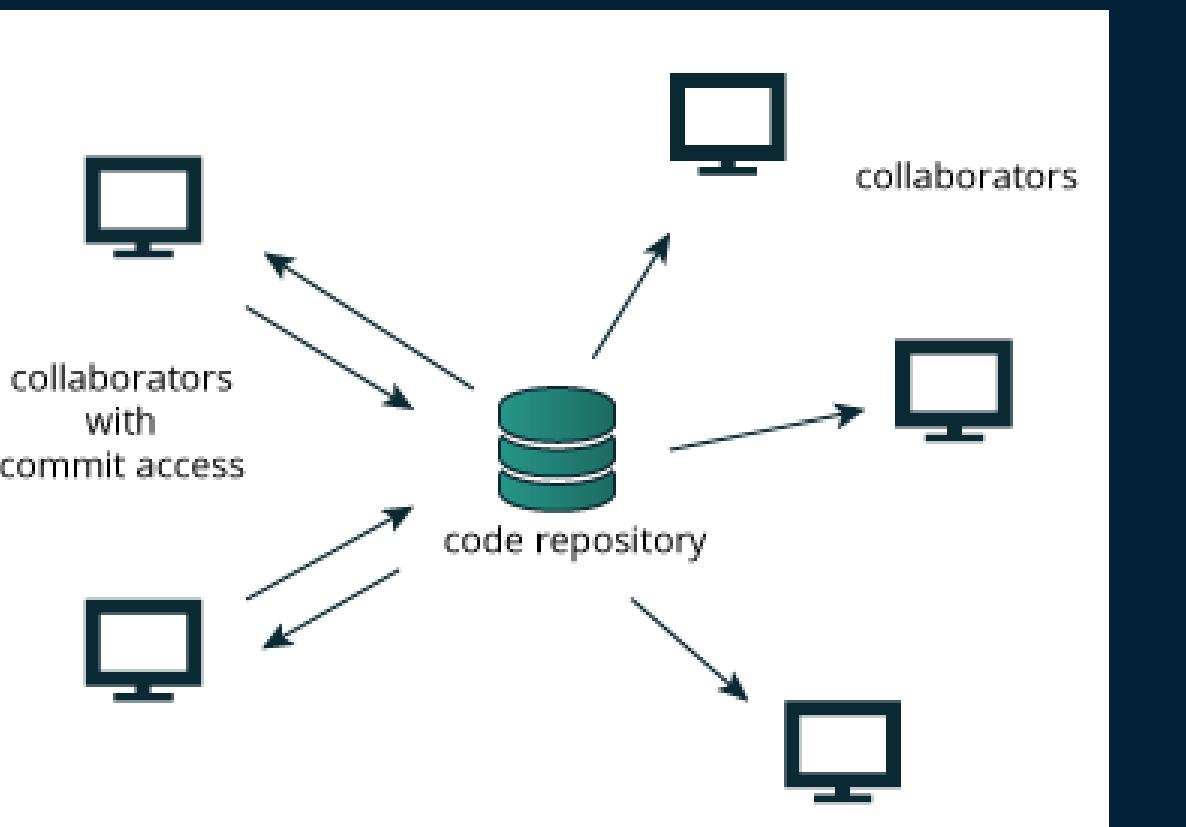
Alice Benatti, Samuele Musiani

slide di riferimento di Stefano Volpe, Luca Tagliavini



Cos'è git?

- **sistema di versionamento** = tenere traccia delle versioni dei file, ottenendo una cronologia di tutte le modifiche
- **distribuito** = chiunque ha una copia completamente locale della *repository*



Perché usarlo?

- Cronologia trasparente su tutti i cambiamenti dei file
- Ripristinare subito un qualsiasi **stato precedente**
- Collaborare a copie diverse dello stesso progetto

"FINAL".doc



FINAL.doc!



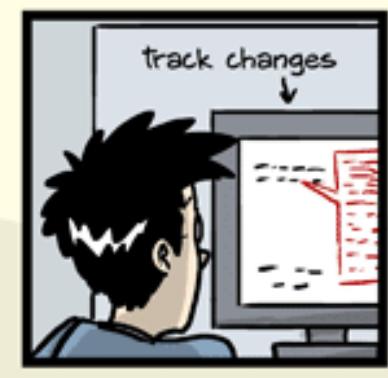
FINAL_rev.2.doc



FINAL_rev.6.COMMENTS.doc



FINAL_rev.8.comments5.
CORRECTIONS.doc



FINAL_rev.18.comments7.
corrections9.MORE.30.doc



FINAL_rev.22.comments49.
corrections.10.#@\$%WHYDID
ICOMETOGRAD SCHOOL????.doc

JORGE CHAM © 2012

Ambiente di lavoro

1.1. Account dipartimentale
a. client ssh

1.2. Git installato sul proprio dispositivo

2. Utente su una piattaforma di hosting per progetti git
[GitHub, GitLab, ...]



Preparazione

1. scegliere una delle macchine di laboratorio [es: XXX.cs.unibo.it dove XXX = *lily, lucia, morales, edmondo, pancrazio, ...*]
2. collegarsi via ssh

```
$ ssh nome.cognome@XXX.cs.unibo.it
```

Preparazione

1. scegliere una delle macchine di laboratorio [es: XXX.cs.unibo.it dove XXX = *lily, lucia, morales, edmondo, pancrazio, ...*]
2. collegarsi via ssh

```
$ ssh nome.cognome@XXX.cs.unibo.it
```

[...]

Are you sure you want to continue connecting
(yes/no/[fingerprint])? yes

[...]

nome.cognome@XXX.cs.unibo.it's password: <password>

Preparazione

3. configurare git con i vostri dati

```
$ git config --global user.name "Nome Cognome"  
$ git config --global user.email \  
"nome.cognome@studio.unibo.it"  
$ git config --global init.defaultBranch main
```

4. generare una chiave ssh

```
$ ssh-keygen -t ed25519 -C \  
"nome.cognome@studio.unibo.it"
```

repository



repository è un progetto mantenuto con git

È come una cartella in cui contenere tutti i file di un progetto gestita con git.

Se ne possono creare tante quante volete.

git init



00

Per inizializzare una *repository* Git vuota in una nuova <directory>;
all'interno di una *directory* lanciamo:

```
$ git init [<directory>]
```

Nota

1. I file interni di git sono nella sottocartella <directory>/.git/
2. Se <directory> non è specificata, viene usata la cartella corrente.

Creiamo un file

01

```
$ nano README.md
```

Scriviamo dentro al file:
“Il nostro primo file”

Come salviamo questo stato della repository?

Vogliamo poter tornare allo stato attuale in un qualsiasi momento futuro

Salviamo lo stato attuale con...

02

git add

Registra le modifiche agli *staged files* commentandole con un <msg>

```
$ git add [<pathspec>...]
```

Per aggiungere il nostro file eseguiamo:

```
$ git add README.md
```

git commit

03

Registra le modifiche agli *staged files* commentandole con un <msg>

```
$ git commit [-m <msg>]
```

Se non uso *-m*, viene aperto un *editor* di testo in cui posso specificare il messaggio,
!! se l'*editor* di testo non è specificato lanciate il comando:

```
$ export EDITOR=nano
```

Creiamo un altro file

00

```
$ nano HelloWorld.cpp
```

```
#include <iostream>

int main() {
    std::cout << "Hello World\n";
}
```

git status

01

mostriamo lo stato attuale della nostra *repository*

\$ git status

Dovreste vedere una situazione del genere:

```
$ git status  
[...]  
Untracked files:  
  HelloWorld.cpp
```

untracked files = tutti i file di cui al momento git non sta tenendo conto

git add

02

Aggiunge i file scelti agli *staged files*

```
$ git add HelloWorld.cpp
```

Dovreste vedere una situazione del genere:

```
$ git add  
[...]  
Changed to be committed:  
  new file: HelloWorld.cpp
```

Abbiamo aggiunto *HelloWorld.cpp* alla **Staging Area**

A questo punto git è a conoscenza del nostro file e monitorerà tutte le sue modifiche future.

Modifichiamo il file

03

```
$ nano README.md
```



Il nostro primo file
Progetto HelloWord in C++

git status

mostriamo lo stato attuale della nostra *repository*

\$ git status

Dovreste vedere una situazione del genere:

```
$ git status  
[...]  
Changes not staged for commit:  
modified: README.md
```

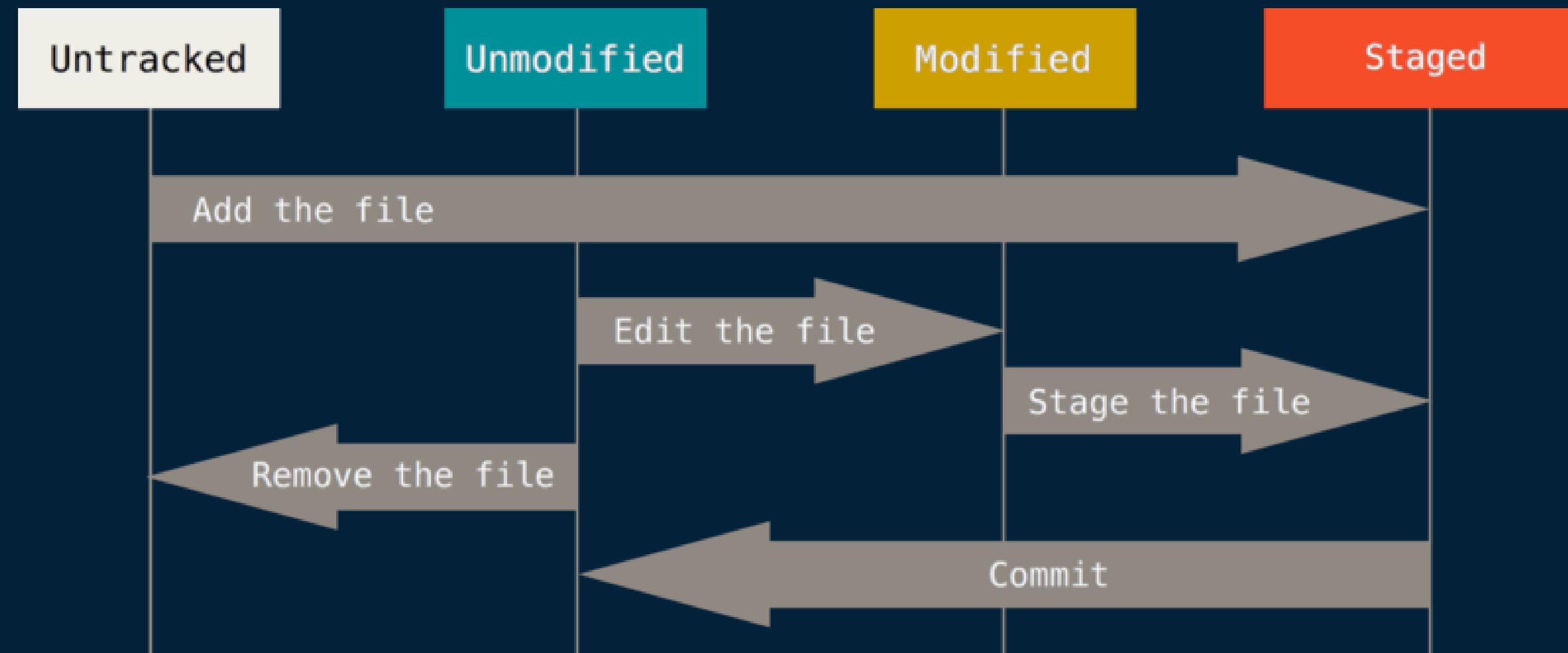
Modified Area = tutti i file che sono stati modificati rispetto al del commit precedente

Stato dei file

I file di una repository sono sempre in esattamente uno dei seguenti quattro stati:

1. ***untracked*** non tracciato da git
2. ***unmodified*** non modificato rispetto all'ultima “istantanea” di git
3. ***modified*** modificato rispetto all'ultima “instantanea” di git
4. ***staged*** modificato rispetto all'ultima “instantanea” di git e pronto ad essere registrato

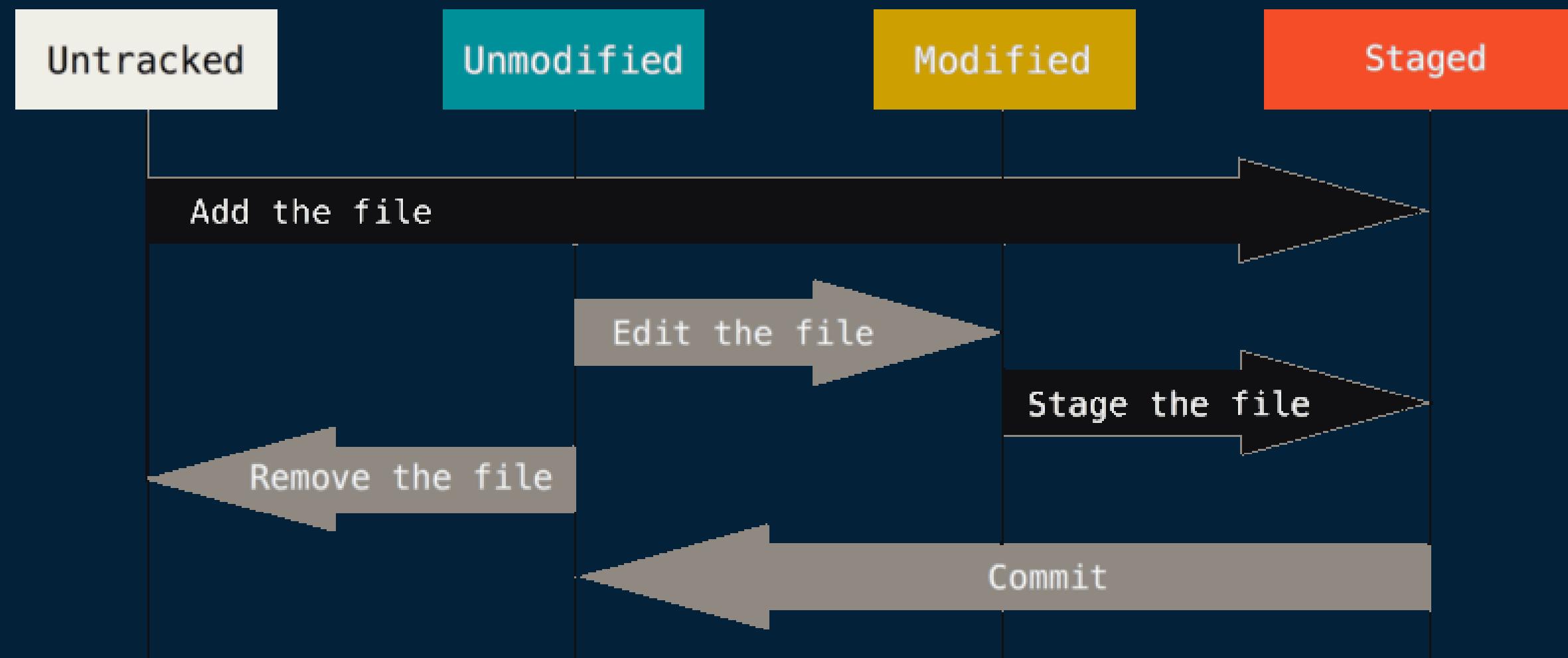
git status



git add

Aggiunge <path_specifico> (non tracciato o modificato) agli *staged files*:

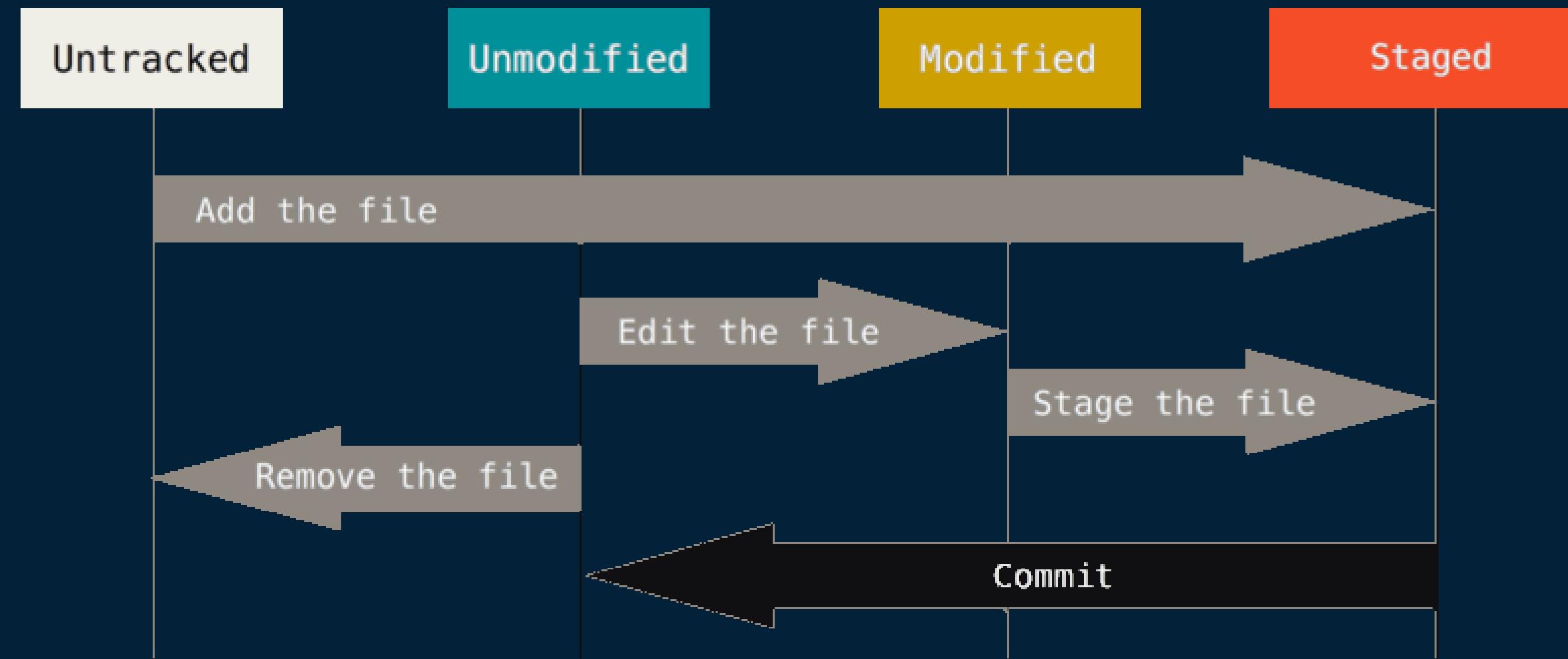
```
$ git add [<pathspec>...]
```



git commit

Registra le modifiche agli *staged files* commentandole con un *<msg>*

```
$ git commit [-m <msg>]
```



05

Aggiungiamo tutte le modifiche alla staging area con:

```
$ git add .
```

Nota

Con “.” aggiunge tutti i file presenti nella directory che sono stati modificati o che sono nell'**untracked area**.

Committiamo:

```
$ git commit -m "add: helloworld.cpp"
```

Verifichiamo lo status:

```
$ git status
```

Dovreste vedere una situazione del genere:

```
$ git status
Nothing to commit, working tree clean.
```

Unmodified Area = avendo committato tutti i file, non abbiamo attualmente altre modifiche sui file.

06

git log

Mostriamo il registro dei commit:

```
$ git log [--graph]
```

Notiamo che sono visibili tutti i commit ordinati per data.

Ogni commit ha un numero identificativo **hash code**, un autore, la data e testo.

```
commit f81e2661ec68130d6627277f47d3b3f73b2c9f0d (HEAD -> main)
Author: John Doe <johndoe@email.com>
Date:   Tue Mar 15 10:00:00 2023 -0400

    Add new feature

commit 8105e5b6fb5f6b166c6de5c9d4d4db4b44f71aa7
Author: Jane Doe <janedoe@email.com>
Date:   Mon Mar 14 10:00:00 2023 -0400

    Fix bug in existing feature

commit b13f6098dd1cdaec24e3c3c3e9d62e7bb56b38ec
Author: John Doe <johndoe@email.com>
Date:   Sun Mar 13 10:00:00 2023 -0400

    Initial commit
```

hash code ↗

Nota

1. usando *--graph*, il registro è rappresentato come un grafo

Modifichiamo il file

07

```
$ nano HelloWorld.cpp
```

```
#include <iostream>

int main() {
    std::cout << "Hello World\n";
    return 0;
}
```

git diff

08

Mostra le differenze presenti nei file rispetto a quelli di git committati precedentemente

\$ git diff

Dovreste vedere una situazione del genere:

```
$ git diff
diff --git a/helloWorld.cpp b/helloWorld.cpp
index df92bc7..8b66388 100644
--- a/helloWorld.cpp
+++ b/helloWorld.cpp
@@ -2,4 +2,5 @@

```

```
int main() {
    std::cout << "Hello World\n";
+    return 0;
}
```

git restore

09

Ripristiniamo un file alla versione dell'ultimo commit

```
$ git restore [--staged]
```

Unmodified

Modified

Staging Area

Attenzione!

facendo \$ git restore

le modifiche effettuate sui files
verranno perse



git checkout

Ci spostiamo su un altro commit

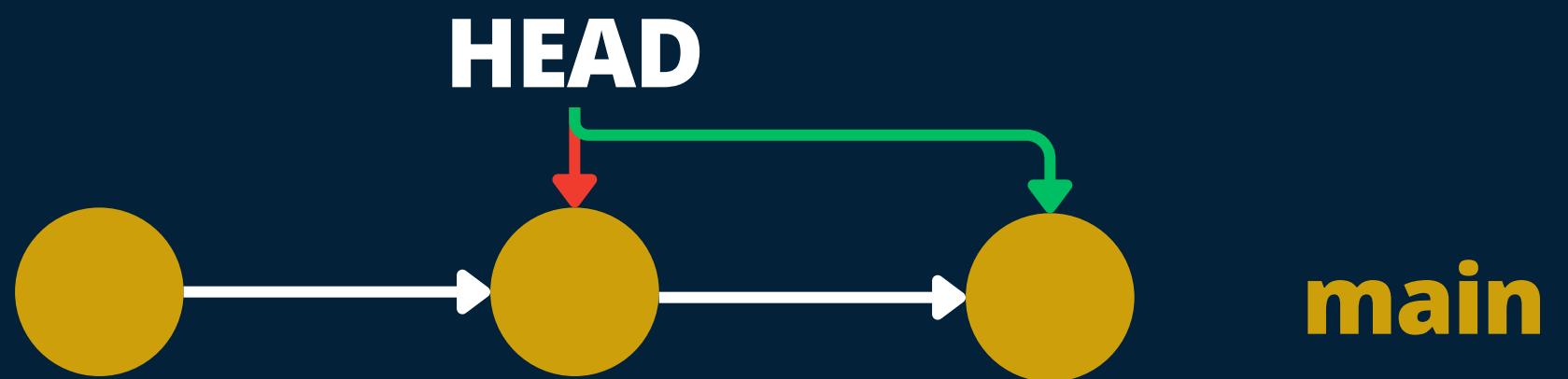
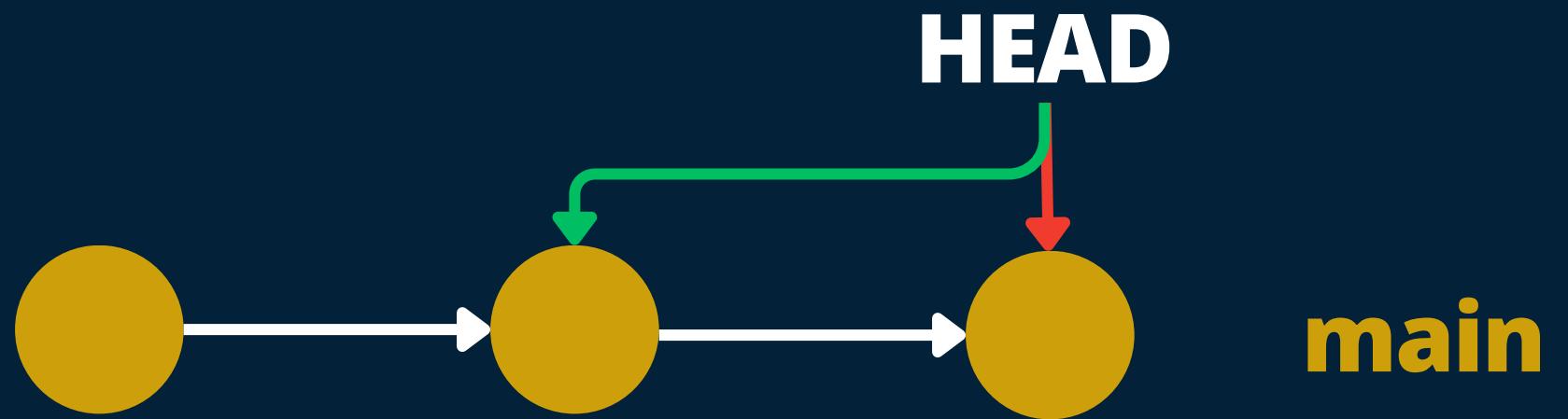
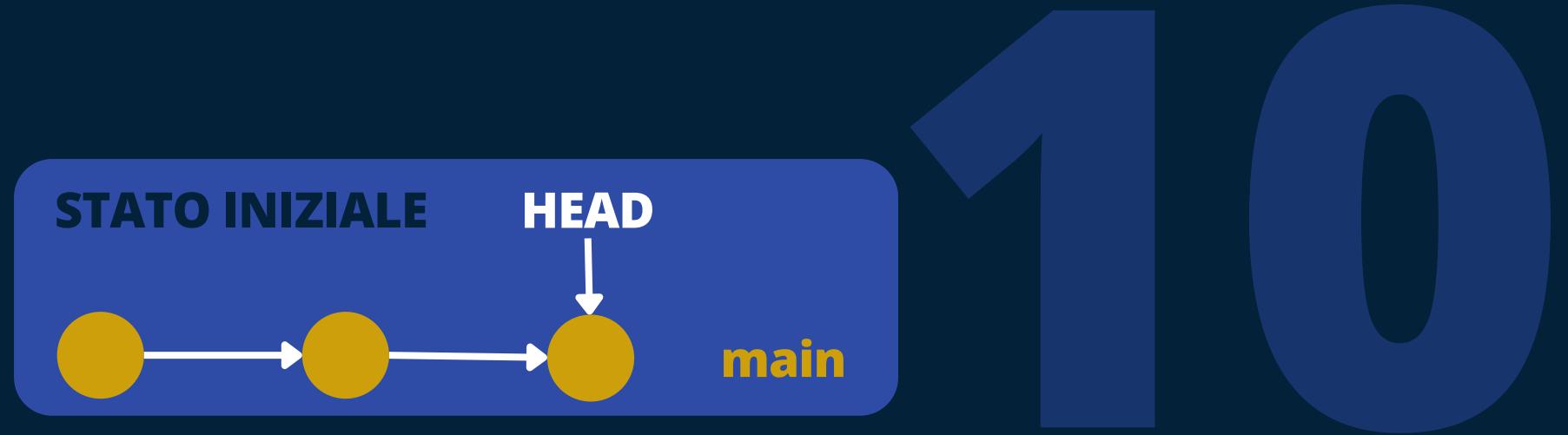
```
$ git checkout <hash_code>
```

HEAD punta sempre al commit corrente

Non si può fare se ci sono file modificati all'interno della directory...

Torniamo indietro nel commit più recente:

```
$ git checkout main
```



git stash

11

Metto da parte delle modifiche che non voglio committare ora

```
$ git stash
```

Per riprendere le nostre modifiche usiamo:

```
$ git stash pop
```

Attenzione

Quando si fa *checkout* è necessario che nella directory corrente non ci siano modifiche, per evitare che vengano sovrascritte.

Così le salviamo in un “commit temporaneo” che è semplice da riprendere.

git branch

Per lavorare su cose diverse basandosi sullo stesso codice mantenendo più versioni separate dello stesso progetto:

- quella principale;
- quella su cui stiamo sviluppando una funzione ancora sperimentale;
- quella su cui stiamo correggendo un problema...

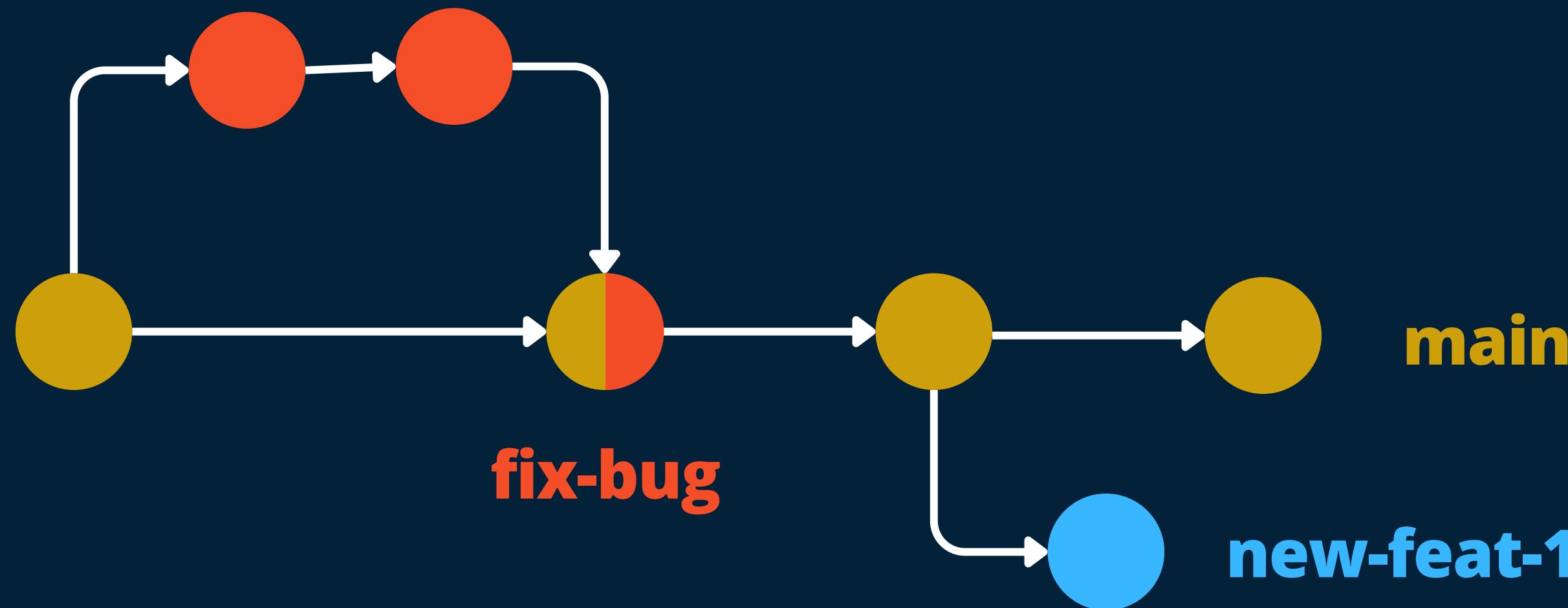
Non dobbiamo copiare tutta la directory della repo!

In git, ogni versione (*branch*, ramificazione) si alterna nella stessa cartella.

Usiamo il seguente comando per mostrare un elenco delle *branch* esistenti nella repository

```
$ git branch
```

git branch



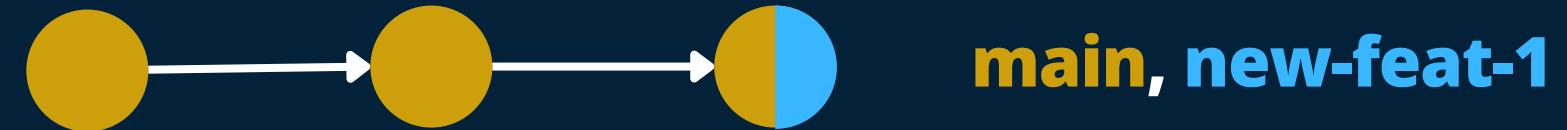
git branch

Creiamo la branch “*new-feat-1*”

```
$ git branch new-feat-1
```

Spostiamoci sulla nuova branch

```
$ git checkout new-feat-1
```



12

Modifichiamo il file

13

```
$ nano HelloWorld.cpp
```

```
#include <iostream>

int main() {
    std::cout << "Hello World\n";
    int input;
    std::cin >> input;
    std::cout << "New feature: " << input;
    return 0;
}
```

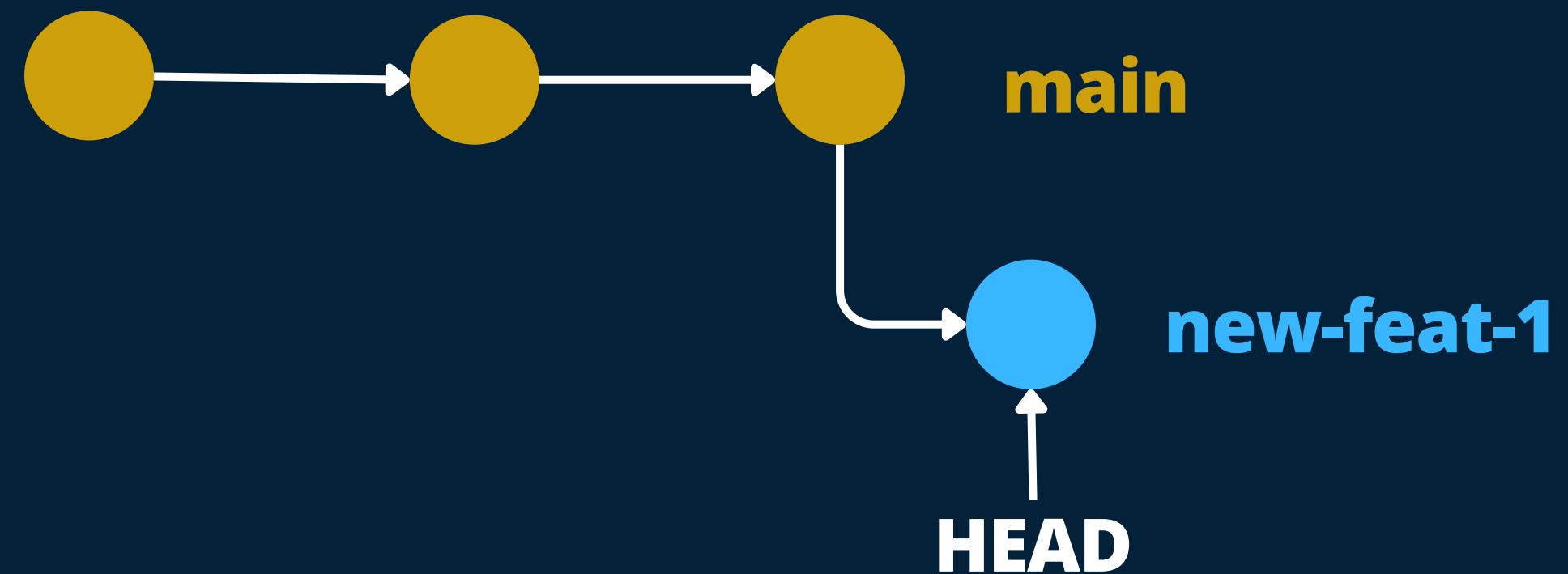
Siamo nella branch giusta? Controlliamo con:

```
$ git branch
```

Aggiungiamo alla staging area il file e committiamo

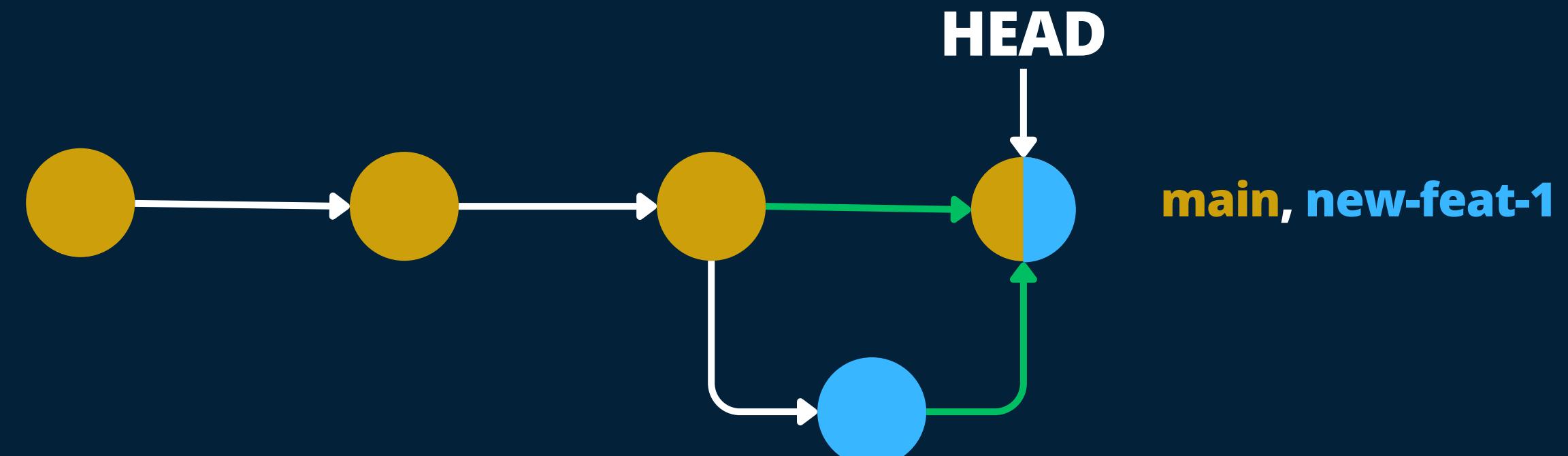
```
$ git add HelloWorld.cpp  
$ git commit -m "add: new feature input"
```

Facendo *git log --graph* possiamo vedere una cosa del genere:



git merge

unisce i commit di due branch



Viene creato un nuovo commit nella branch di destinazione, che unisce le due branch.

git merge

unisce i commit di due branch

15

```
$ git merge <branch_da_mergiare>
```

Per fare *merge* della nostra branch “new-feat-1” dobbiamo spostarci sulla branch di destinazione [“main” nel nostro caso] e da lì eseguire il comando.

Per spostarci come facciamo?

git merge

unisce i commit di due branch

15

```
$ git merge <branch_da_mergiare>
```

Per fare *merge* della nostra branch “new-feat-1” dobbiamo spostarci sulla branch di destinazione [“main” nel nostro caso] e da lì eseguire il comando.

Per spostarci come facciamo?

```
$ git checkout main
```

Verifichiamo con *git log --graph*

Creiamo una nuova branch “docs”

```
$ git branch docs
```

Spostiamoci sulla nuova branch

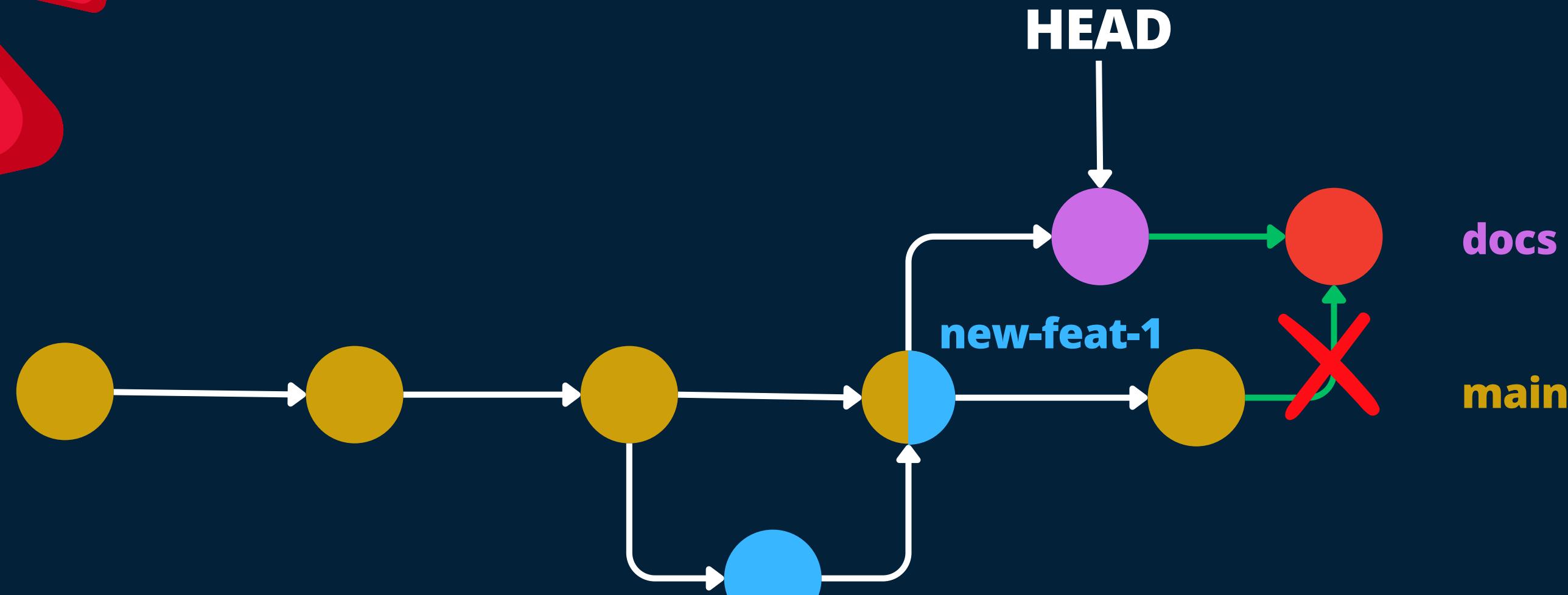
```
$ git checkout docs
```

Creiamo un nuovo file e inseriamo qualcosa

```
$ nano TODO.md
```

Aggiungiamo alla staging area il file e committiamo

```
$ git add TODO.md  
$ git commit -m "add: new docs"
```



Non è buona pratica fare il merge della branch `main` in altre branch “secondarie”,
in quanto la `main` nasce per essere la principale su cui tutto converge.

Spostiamoci nella branch main

```
$ git checkout main
```

Rinominiamo il file *HelloWorld.cpp* in ***HiUniverse.cpp***, lo apriamo e editiamo la stringa di cout

```
$ nano HiUniverse.cpp
```

Aggiungiamo alla staging area il file e committiamo

```
$ git add HiUniverse.cpp  
$ git commit -m "feat: change project"
```

Mostriamo il registro dei commit:

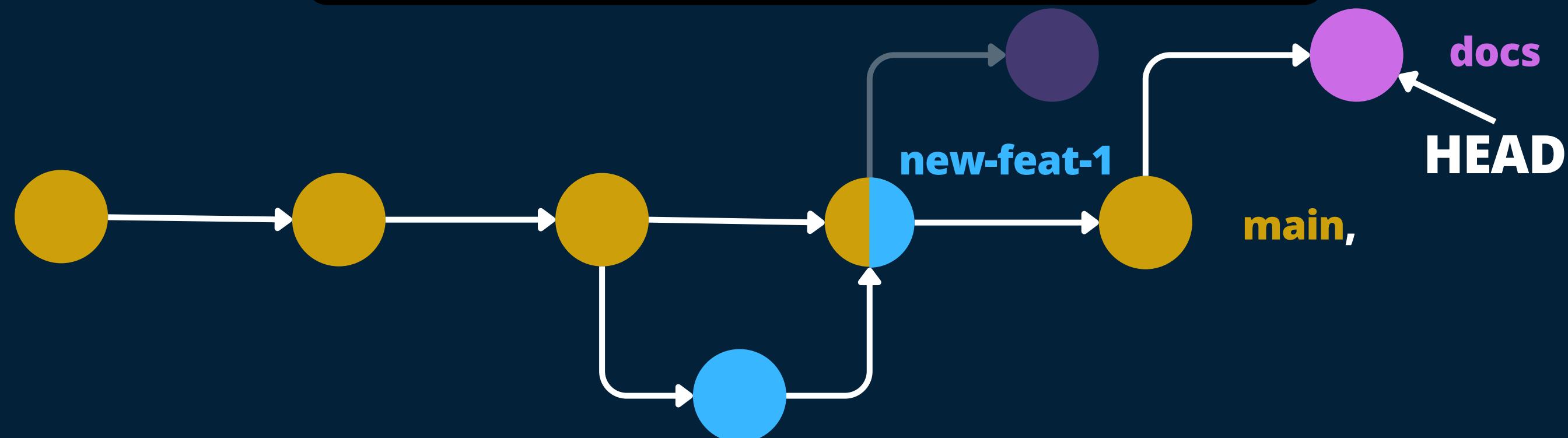
```
$ git log --graph
```

git rebase

(È buona pratica invece:)

sposto il commit di origine della branch allo stato più recente della branch di origine

```
$ git rebase <branch_di_origine>
```



Dalla branch da spostare eseguiamo il comando.



git show

Possiamo mostrare dettagli su uno o più commit a scelta dando un **<object>** univoco di riferimento

```
$ git show [<object> ...]
```

Un modo di riferirsi a un **<object>** è un prefisso univoco del codice sha1 del commit.

Se non si specifica nulla, viene usato *HEAD* (il commit corrente)

git reset

riporta la **branch** allo stato del commit specificato

20

```
$ git reset <hash_code_commit>
```

Dalla branch attuale, elimina i commit fino ad *<hash_code_commit>* (escluso).

Le modifiche effettuate sui commit cancellati, vengono raggruppate in un'unica modifica riportate con stato del file **modified**.

Flag speciali

Scopri tu cosa fanno di magico ✨

```
$ git add -p [<pathspec>...]
```

```
$ git rebase -i <hash_code_commit>
```

```
$ git cherry-pick <hash_code_commit>
```

```
$ git reset --hard <hash_code_commit>
```

```
$ git reset --soft <hash_code_commit>
```

```
$ git merge --squash <branch_da_mergiare>
```