

Git 1

comandi base

risorse.students.cs.unibo.it/lab/2024

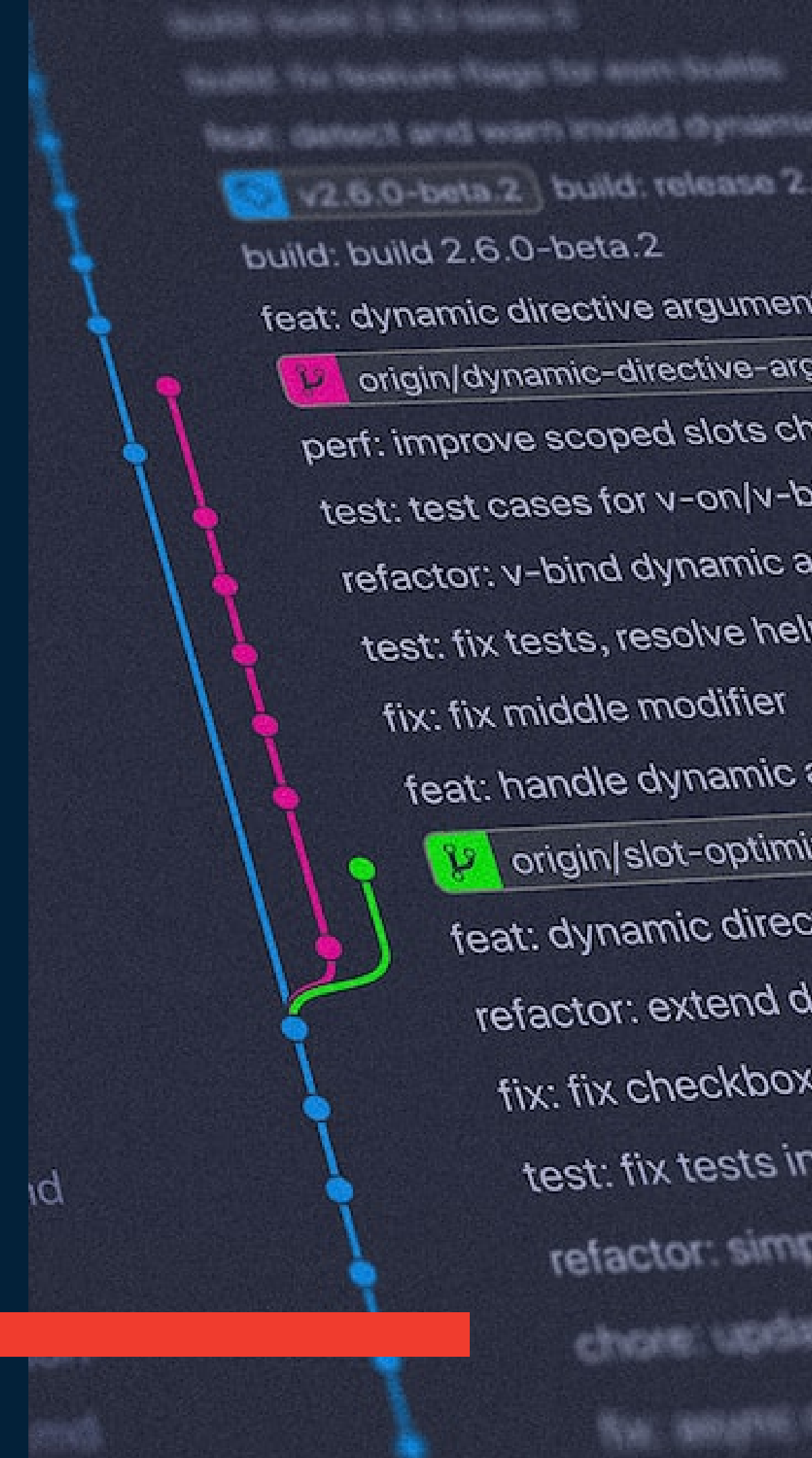
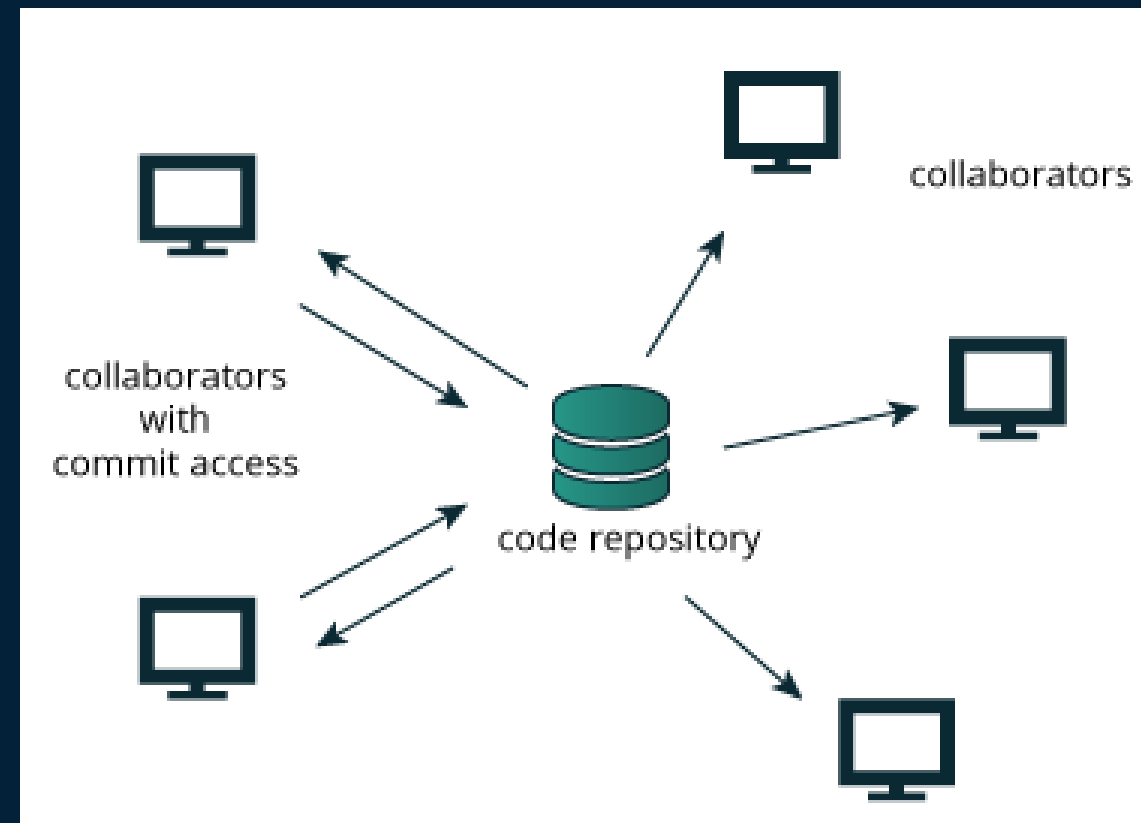
Alice Benatti, Samuele Musiani

slide di riferimento di Stefano Volpe, Luca Tagliavini



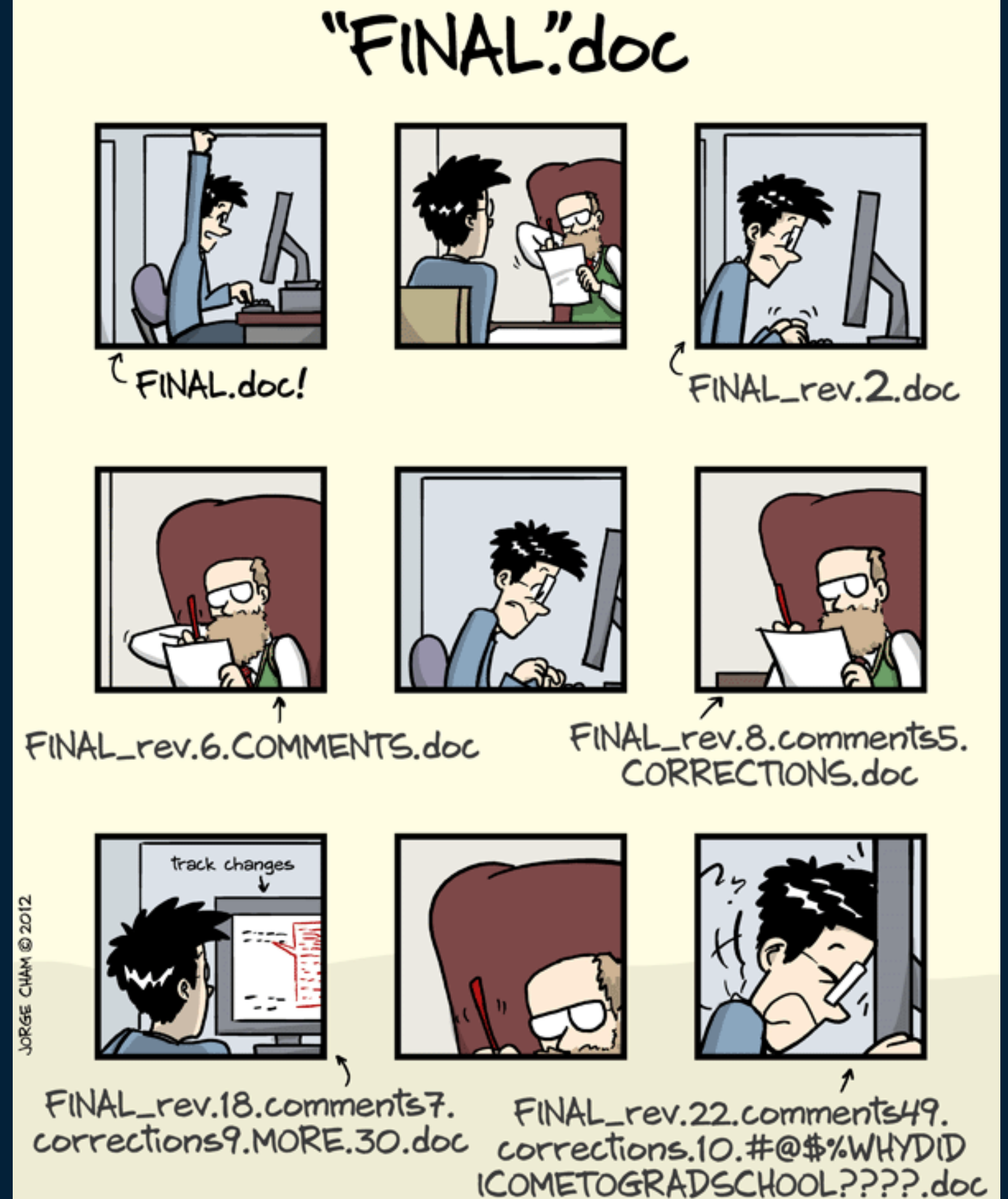
Cos'è git?

- **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 dinamiche diverse dello stesso progetto



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, ...]



Installare git

Linux con Ubuntu/Debian

```
$ sudo apt-get update  
$ sudo apt-get install git
```

[per altre distribuzioni clicca qui]

Windows

- via Command Line con *winget*

```
winget install --id Git.Git -e --source winget
```
- scaricare dal link del sito ufficiale

[Click here to download](#) the latest (2.42.0) 64-bit version of Git for Windows. This is the most recent maintained build. It was released about 2 months ago, on 2023-08-30.

Other Git for Windows downloads



Installare git

MacOS

Ci sono diversi package manager che permettono di installare facilmente git, scegli quello che preferisci:

- **Homebrew**

Installa [homebrew](#) se non lo hai già, e lancia il comando:

```
$ brew install git
```

- **MacPorts**

Installa [MacPorts](#) se non lo hai già, e lancia il comando:

```
$ sudo port install git
```

- **Xcode**

Apple fornisce un pacchetto binario Git con [Xcode](#).

[per ulteriori info seguite la [guida sul sito ufficiale](#)]



Preparazione via ssh

1. scegliere una delle macchine di laboratorio [es: XXX.cs.unibo.it dove XXX = *lily, lucia, morales, edmondo, pancrazio, ...*]
2. generare una chiave ssh (se non l'avete già)

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

3. collegarsi via ssh

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

Per convenzione nelle slide useremo comandi shell (\$):
quando ci saranno *[qualcosa]*, intenderemo che è facoltativo;
quando useremo *< qualcosa >*, intenderemo che è una parte da completare

Preparazione via ssh

1. scegliere una delle macchine di laboratorio [es: XXX.cs.unibo.it dove XXX = *lily, lucia, morales, edmondo, pancrazio, ...*]
3. 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 per tutti

Configurare *git* con i vostri dati:

usate la mail che preferite, sarà poi associata con il vostro user.name alle varie operazioni che svolgerai con git.

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

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

000



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.



cartella autogenerata al momento della creazione di una repository con *git*

È dedicata a tutti i file di git, non deve essere modificata!!

Serve per tenere traccia di:

- commit
- branch
- files
- modifiche
- tag
- worktree

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

git add

02

Aggiunge le modifiche specificate in *<path>* agli *staged files*

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

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
```



git commit

buone abitudini

1. fare commit quando si è alla **fine di una determinata modifica**:

il mio obiettivo: aggiustare un bug

commit quando dopo varie modifiche al progetto riesco a raggiungere il mio obiettivo

[molto probabilmente non sarà l'ultimo commit per il nostro obiettivo, ma dobbiamo cercare di farne il meno possibile]

2. **Convenzione**: dare nomi riconoscibili ai commenti del commit

- add
- feat
- fix
- docs
- style
- [...]



breve descrizione

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 HelloWorld in C++
```

git status

04

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
```

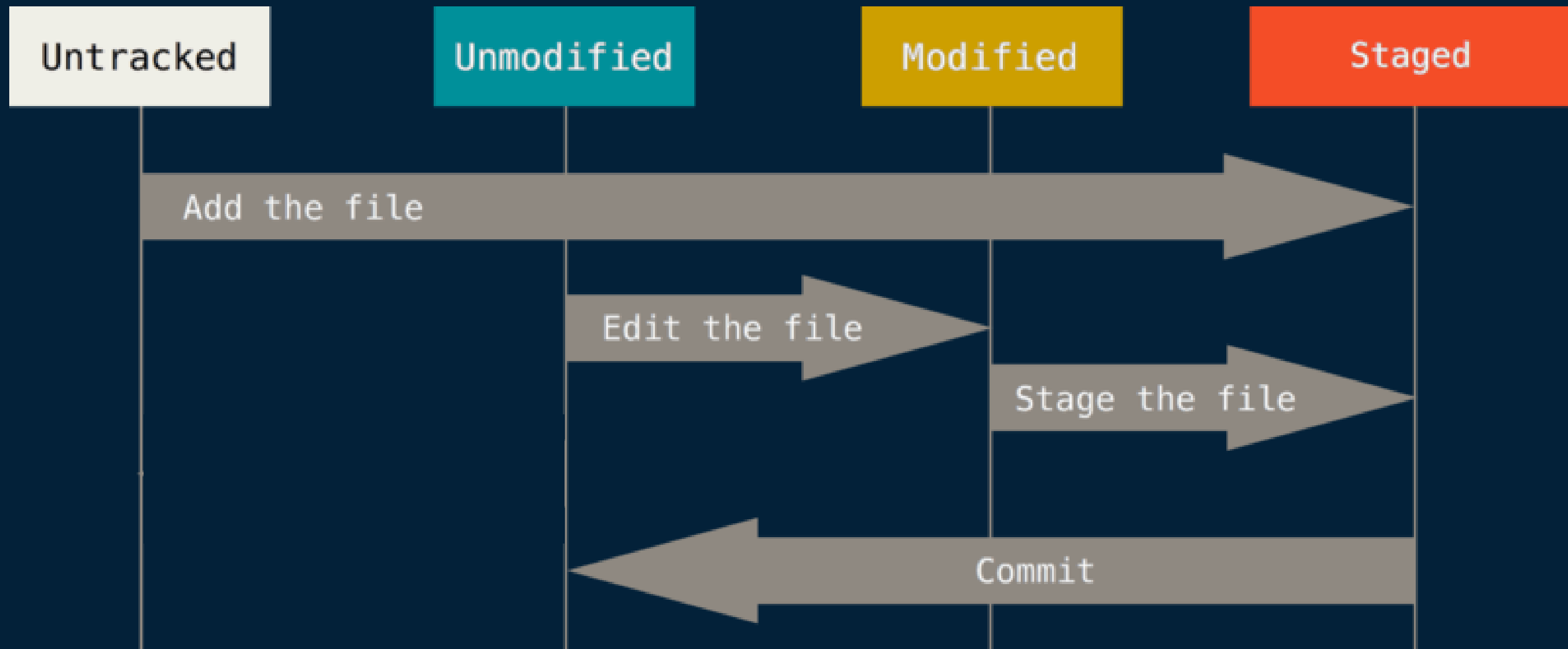
Modified Area = tutti i file che sono stati modificati rispetto al del commit precedente

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 "istantanea" di git
4. ***staged*** modificato rispetto all'ultima "istantanea" di git e pronto ad essere registrato

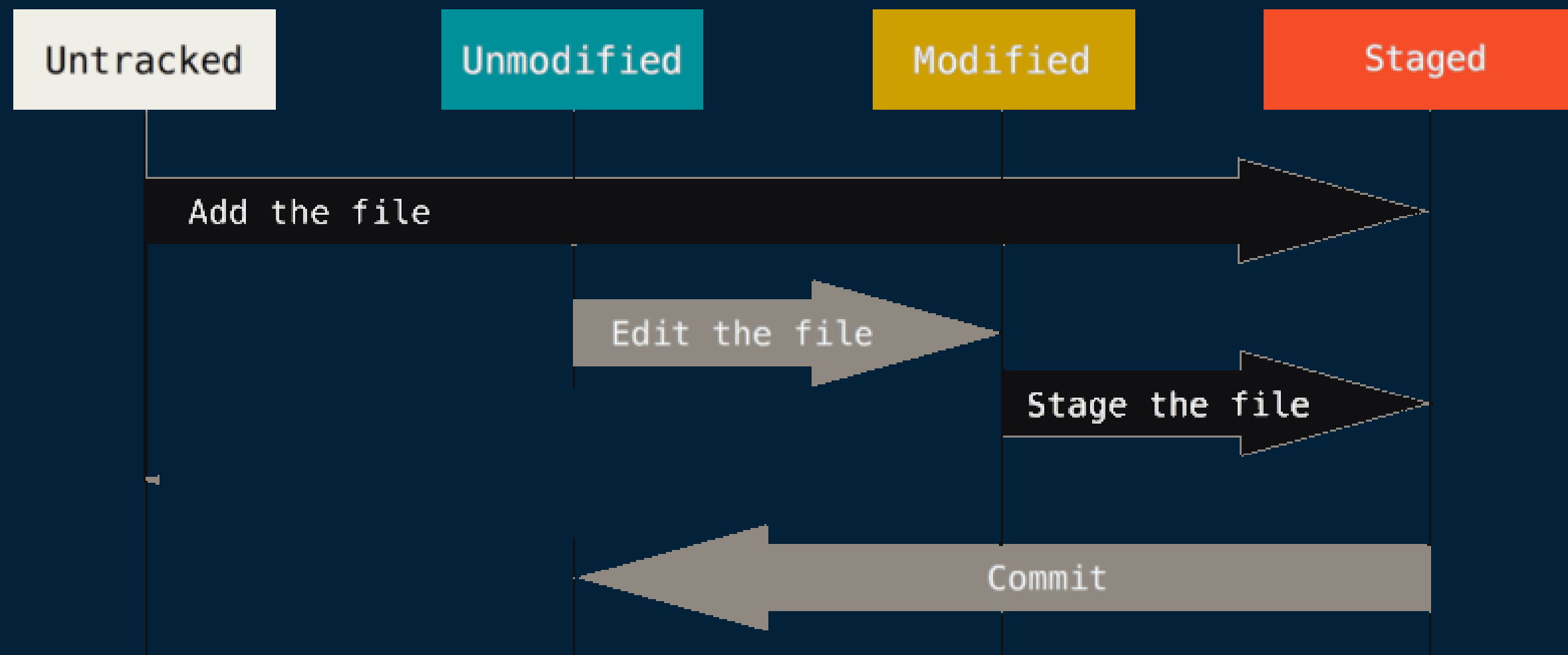
git status



git add

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

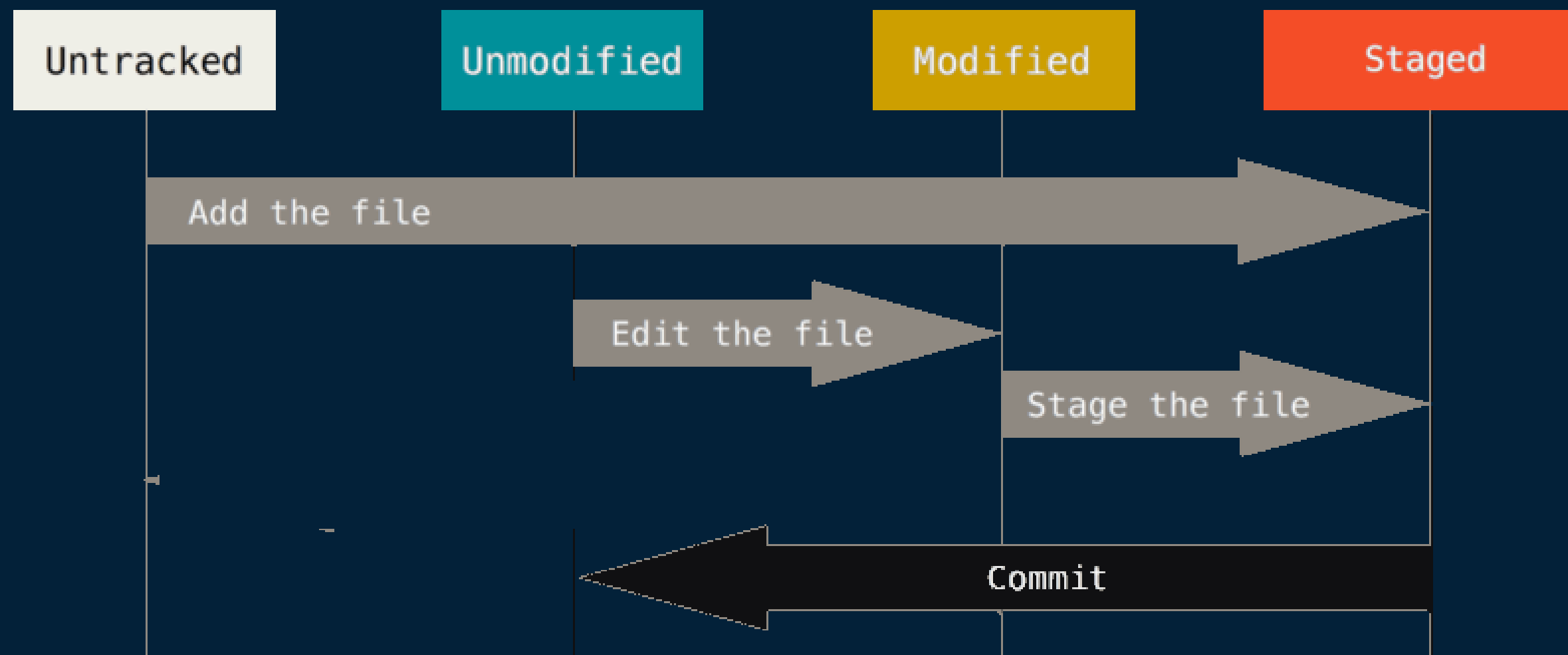
```
$ git add [<pathspec>...]
```



git commit

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

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



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.

git log

006

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.

Nota

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

```
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 

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 diff

08

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

```
$ git diff
```

Nota

Git diff mostra solo le differenze nei file **non ancora** nella Staging Area,

per mostrare le differenze dei file **nella Staging Area** usiamo: `$ git diff --cached`

Si può specificare su quali file mostrare le differenze aggiungendo i path in fondo al comando:

```
$ git diff [--cached] <path_del_file>
```

```
$ git diff [--cached] <path_del_file1> <path_del_file2>
```

git restore

Ripristiniamo un file alla versione dell'ultimo commit

```
$ git restore [--staged] <path_file>
```

Unmodified

Modified

Staging Area



Attenzione!

facendo `$ git restore`

le modifiche effettuate sui files

verranno perse

Quindi per togliere *HelloWorld.cpp* dalla staging area faremo:

```
$ git restore --staged HelloWorld.cpp
```

Se volessimo ripristinare il file all'ultima versione registrata su git useremo:

```
$ git restore HelloWorld.cpp
```

Nel nostro progetto, noi aggiungiamo le modifiche fatte alla staging area dato che c'era una dimenticanza!

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

e committiamo:

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


git restore

Ripristiniamo un file ad una specifica versione/commit

```
$ git restore --source=<commit/branch> <path_file>
```

Se il file è modificato attenzione che verrà sovrascritto

file_1.md **main**

file_2.md **main**



file_1.md **main**

file_2.md **03f4dbf**

```
$ git restore --source=03f4dbf file_2.md
```

git checkout

Permette di spostarsi su un altro commit, ma non solo...

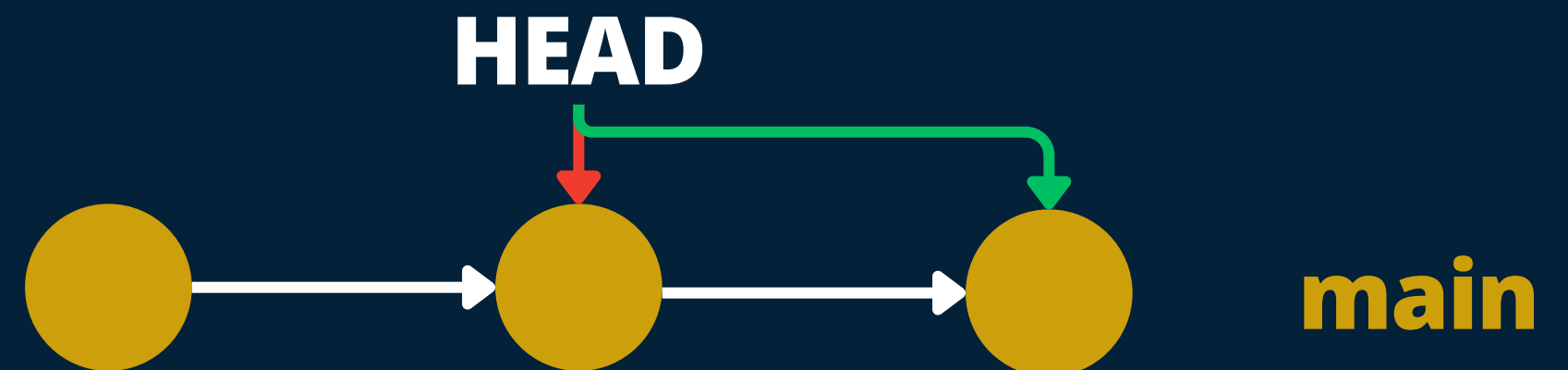
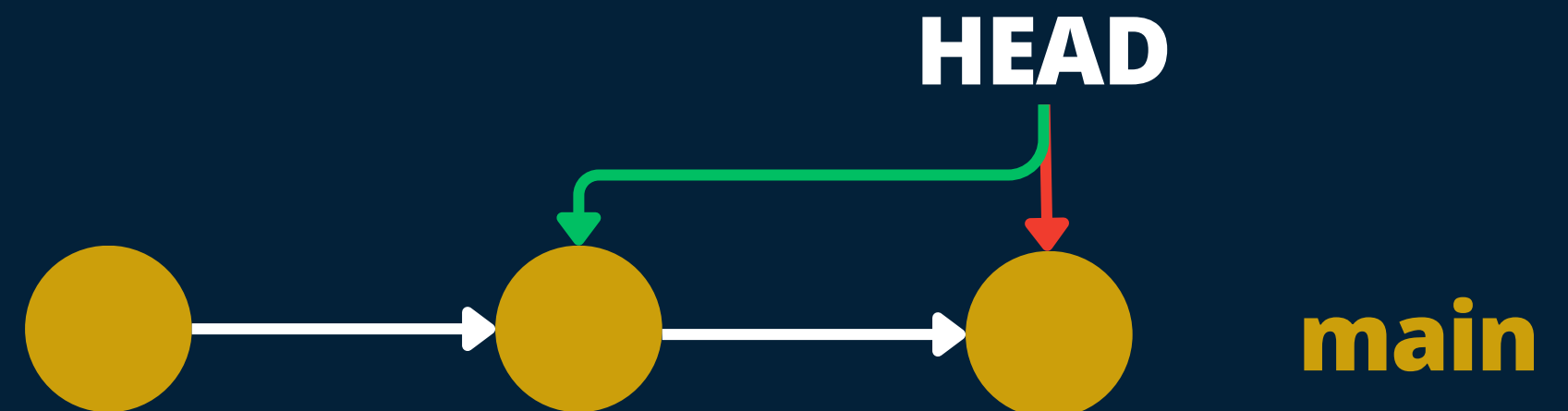
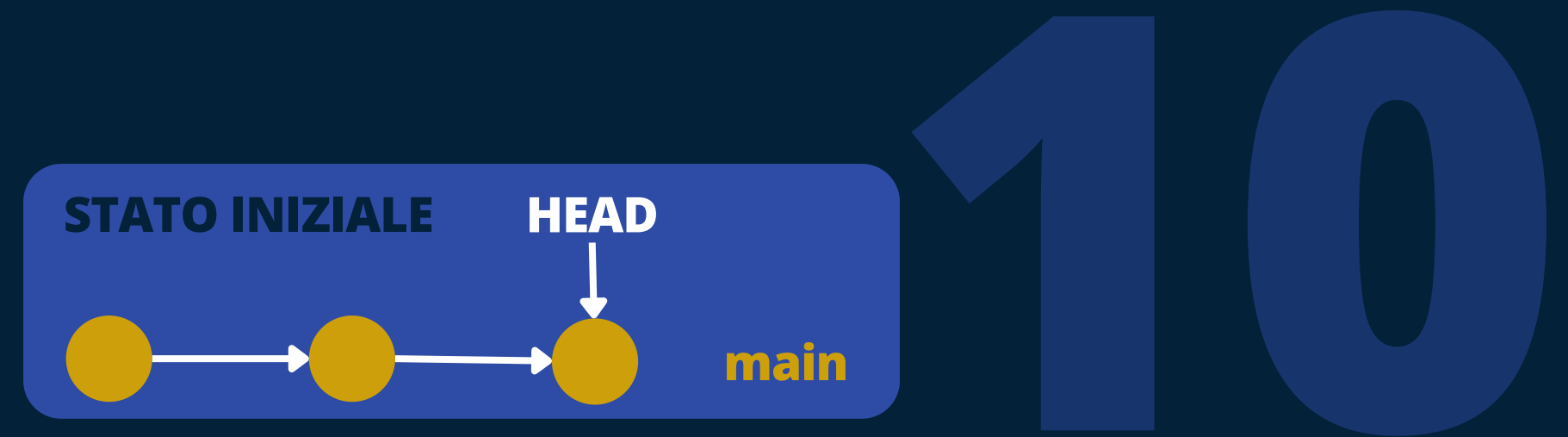
```
$ git checkout <hash_code>
```

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



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.



Branch

ramificazioni

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!

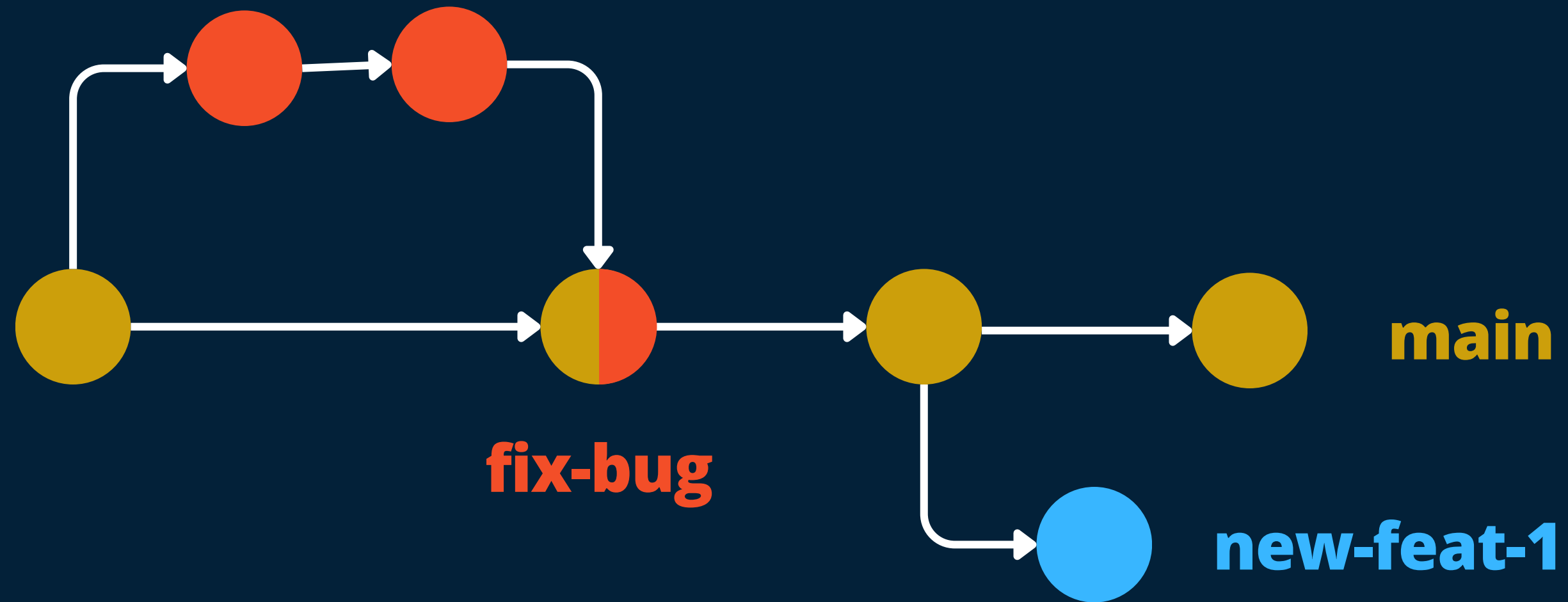
In *git*, ogni versione (*branch*, ramificazione) si alterna nella stessa cartella.

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

```
$ git branch
```

git branch

Esempio di una branch tipo



git branch & git switch

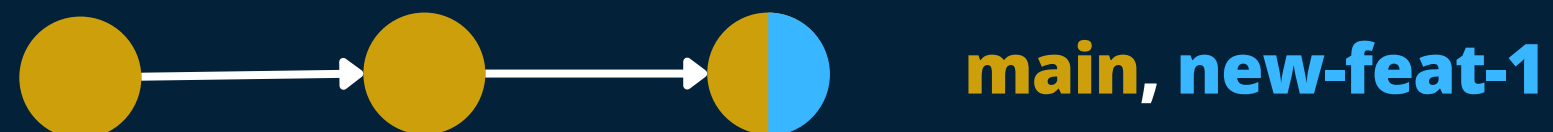
12

Creiamo la branch “*new-feat-1*”

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

Spostiamoci sulla nuova branch

```
$ git switch new-feat-1
```



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;
}
```

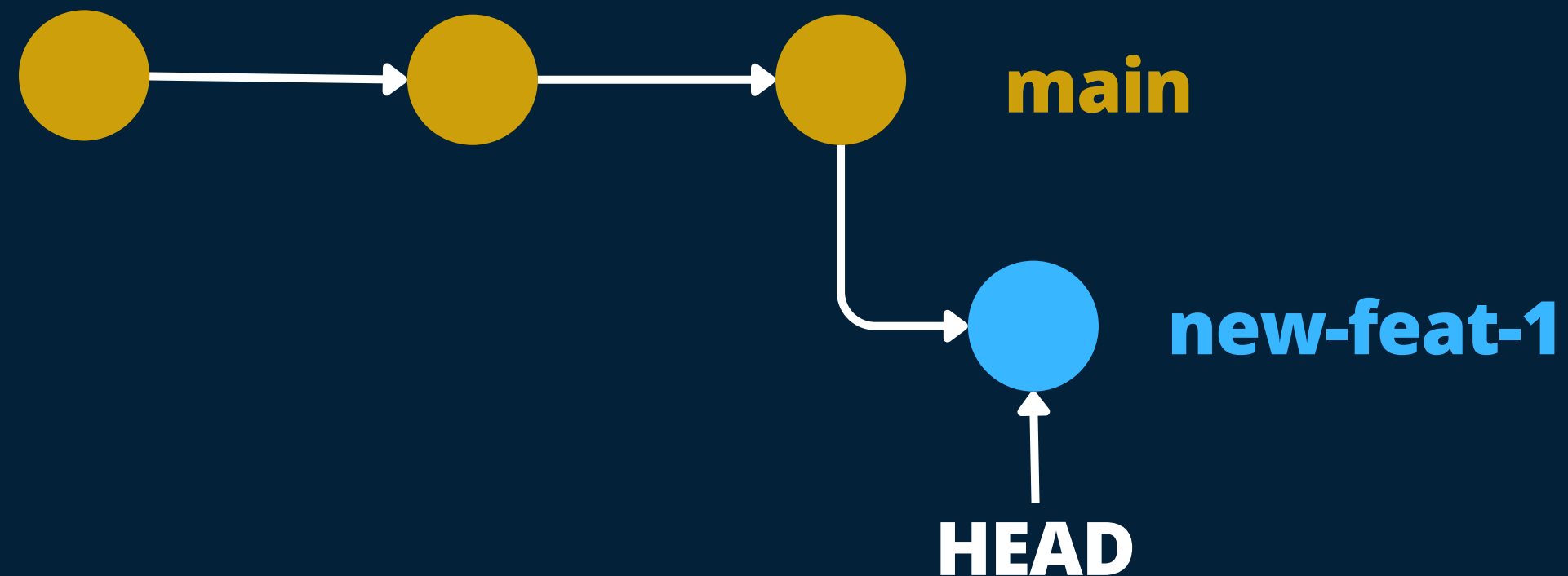

Siamo nella branch giusta? Controlliamo con:

```
$ git branch
```

Aggiungiamo alla staging area il file e committiamo

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

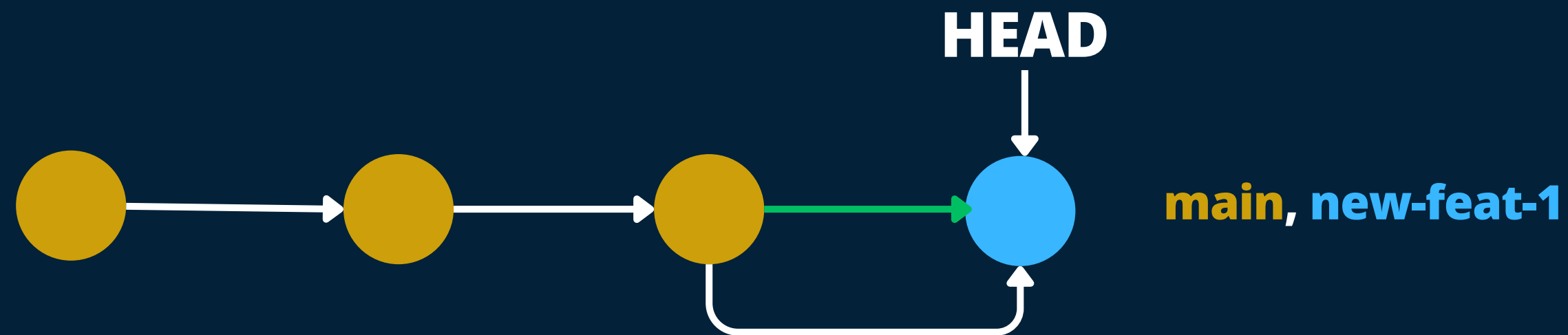
Facendo `git log --graph` possiamo vedere una cosa del genere:



14

git merge

unisce i commit di due branch

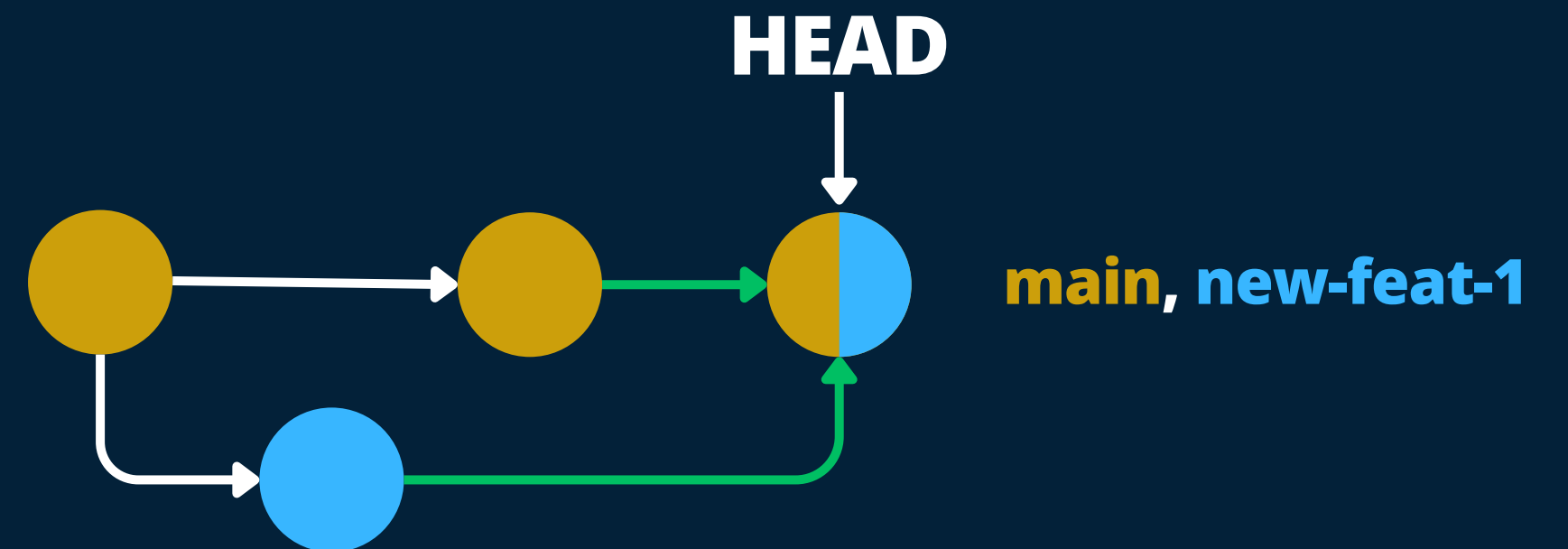
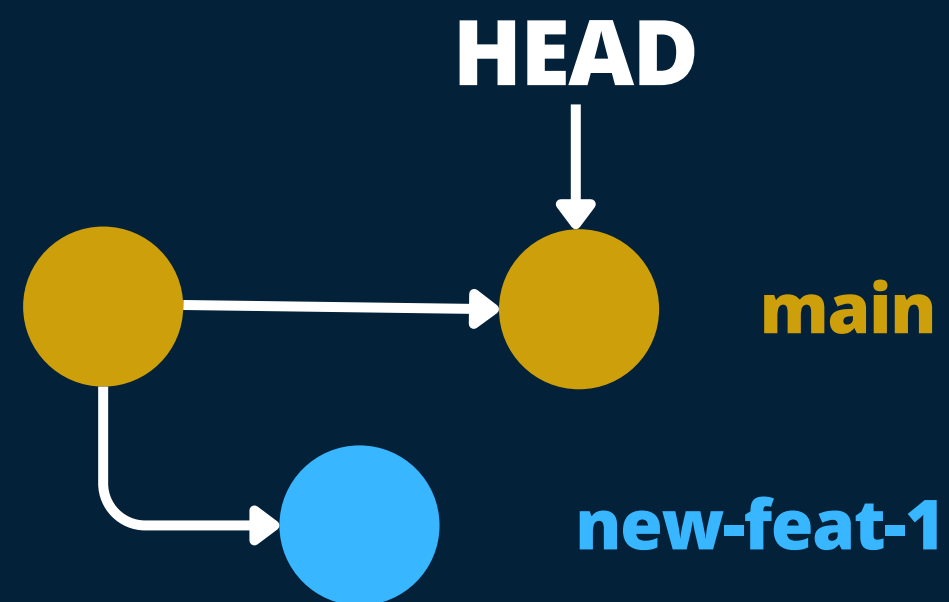


Se è possibile viene fatto il *fast-forward*, ovvero viene spostata la branch **main** allo stesso commit di **new-feat-1**

Questo è possibile solo se gli unici commit di differenza tra le due branch sono sulla branch che si sta mergiando (nel nostro caso **new-feat-1**)

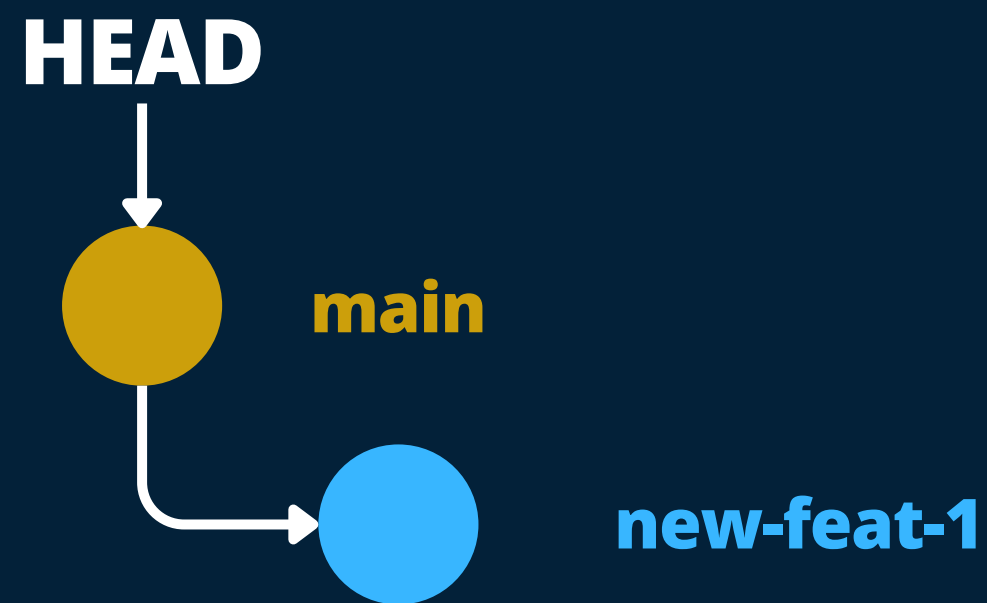
git merge

Se non è possibile fare *fast-forward*, viene creato un commit di merge.



git merge

È possibile forzare il commit di merge con: `--no-ff`



git merge

15

unisce i commit di due branch

```
$ 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

15

unisce i commit di due branch

```
$ 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 switch main
```

Verifichiamo con `git log --graph`

Creiamo una nuova branch “docs”

```
$ git branch docs
```

Spostiamoci sulla nuova branch

```
$ git switch 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”
```

Spostiamoci nella branch main

```
$ git switch main
```

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

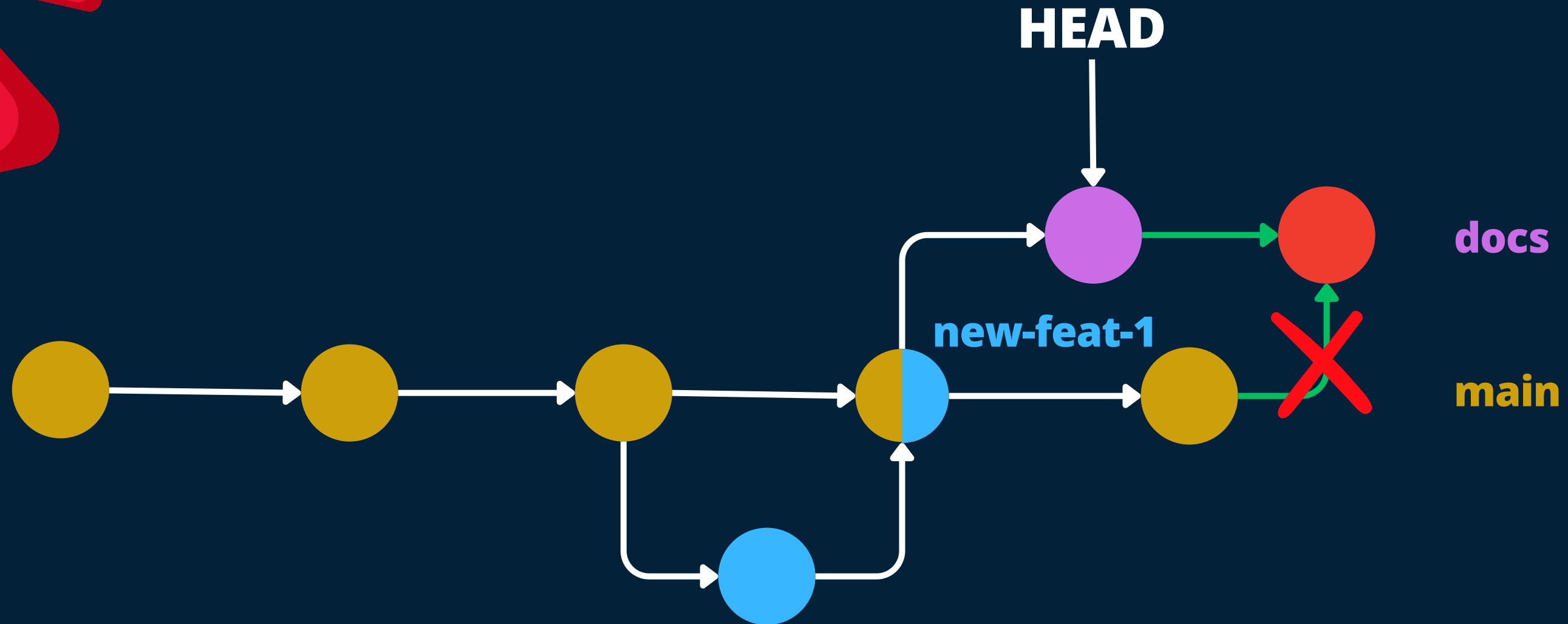
```
$ nano HiUniverse.cpp
```

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
```

Se è possibile evitiamo di mergiare main su branch secondarie.

Casi particolari in cui è “accettabile“:

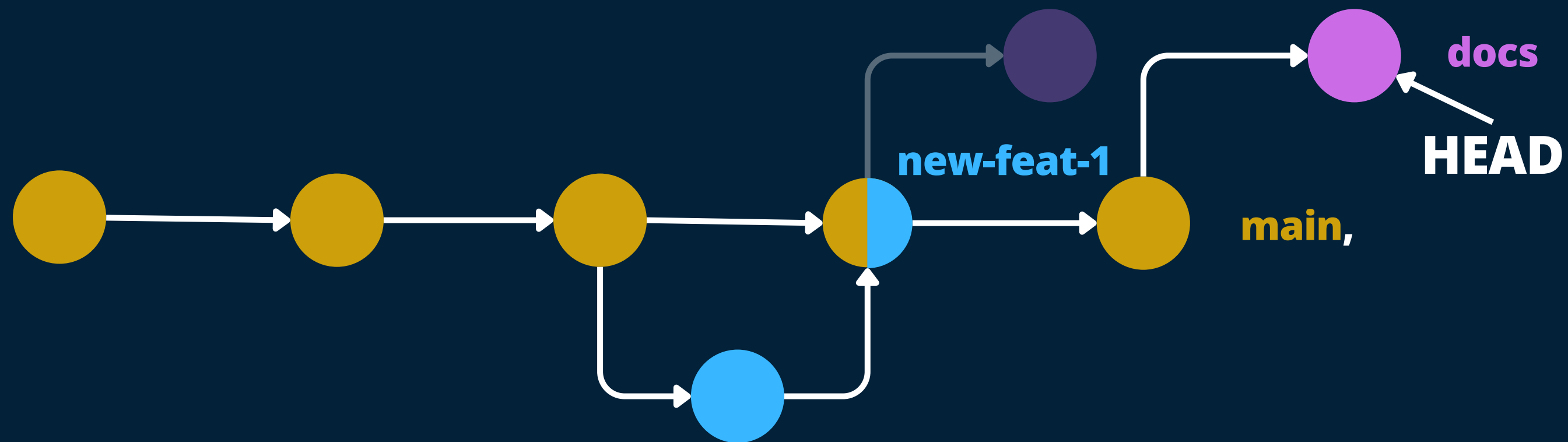
- main ha molti commit avanti rispetto alla branch che si vuole mergiare
- è una branch remota su cui ci stanno lavorando altri

git rebase

18

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

19

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

```
$ git show [<hash_code_commit>]
```

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

```
$ 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**.

20



.gitignore

file di git in cui specificare pattern (estensioni) da ignorare

Crea un file “.gitignore” se non già esistente

```
$ nano .gitignore
```

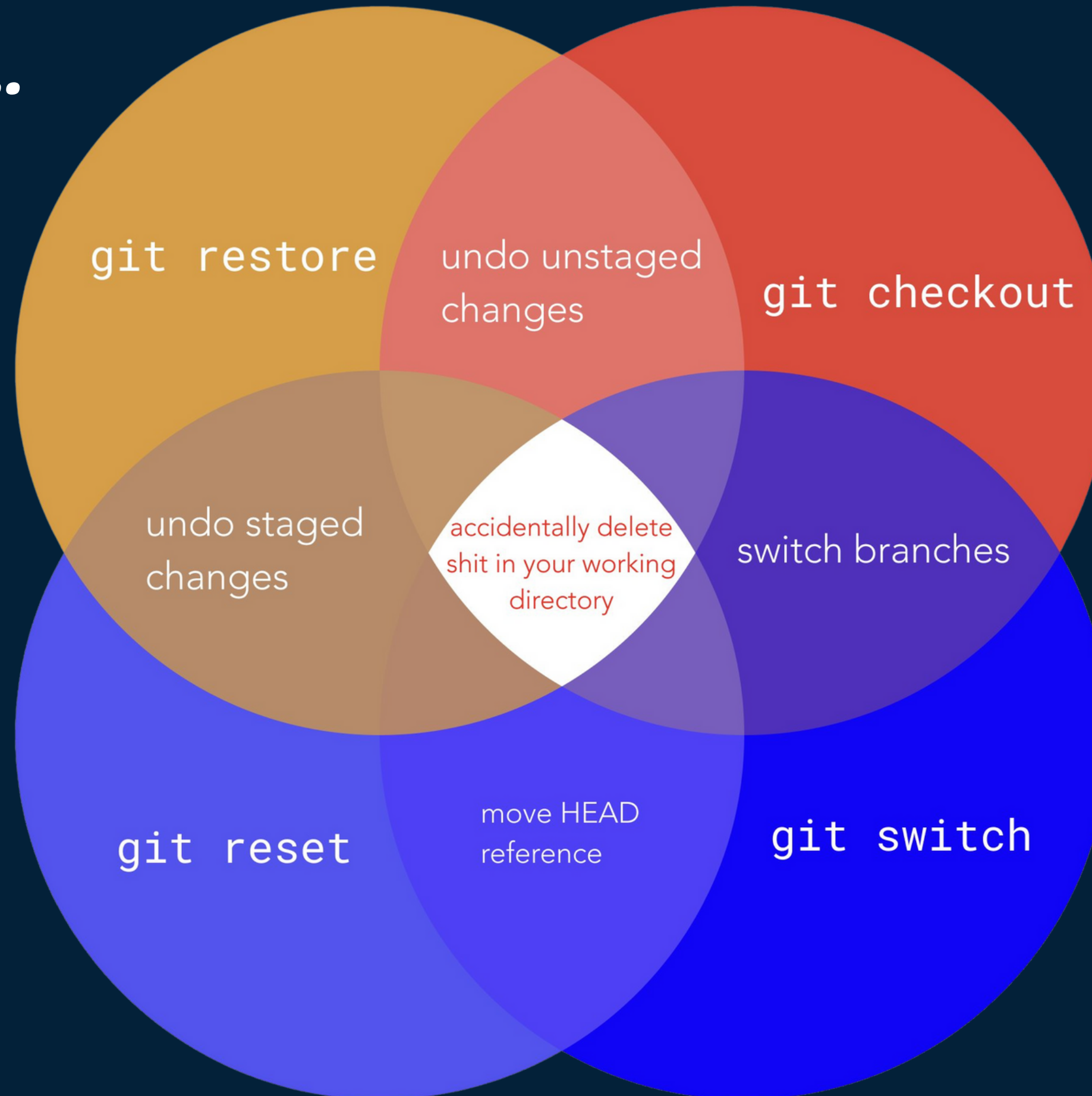
Nel nostro esempio è consigliabile usare un *.gitignore* di questo tipo:

```
# commento  
*.out
```

In questo modo git ignorerà tutti i file con estensione “.out”:

- non li tratterà
- non considererà alcuna modifica/creazione se già esistente il .gitignore

Riassumendo...



Flag speciali

Scopri tu cosa fanno di magico ✨

```
$ git add -p [<path>]
```

```
$ 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>
```