GESTIONE DELLE VERSIONI DEL CODICE SORGENTE COL VCS GIT

All'interno di un progetto software, nel caso di progetti di grandi dimensioni a cui collaborano molti sviluppatori, si trovano centinaia o migliaia di file.

Anche in un progetto di piccole dimensioni scritto da un solo sviluppatore si può avere a che fare con decine o centinaia di file.

Durante la scrittura del codice, i file del progetto si modificano in continuazione, sia in fase di sviluppo che durante la manutenzione correttiva e evolutiva.

A volte si scrive un metodo o una funzione con un certo codice, poi durante lo sviluppo ci si rende conto che la funzione si sarebbe potuta scrivere in un modo più efficiente, allora si "torna indietro" e si modifica la funzione. Poi magari ci si rende conto che la nuova versione crea problemi e allora si vorrebbe tornare alla versione precedente della funzione ma ormai essa è stata cancellata e bisogna riscriverla.

A volte l'esigenza di tornare sui propri passi non riguarda solamente le modifiche apportate ad un file del progetto ma a tre o quattro file, o anche di più. Può accadere che vengano scritte delle intere classi e poi decidere di riscrivere completamente quelle classi in modo diverso. Insomma spesso sarebbe utile "poter tornare sui propri passi" durante lo sviluppo di un software. Gestire queste situazioni in maniera manuale, soprattutto se **più collaboratori** partecipano al progetto, è molto complicato. Per questo vi sono degli appositi software che consentono di "**fare una fotografia**" del progetto in un determinato istante, di memorizzare questa fotografia, e di poter, al bisogno, ripristinare il progetto nella stessa identica situazione fotografata. Questa tipologia di software è chiamata **VCS (Version Control System)**. Il nome deriva dal fatto che i VCS identificano ogni "fotografia" come una **versione del software** e consentono, in ogni istante, di ripristinare una versione del software precedente.

In alcuni casi, inoltre, il progetto software potrebbe evolversi *lungo due strade diverse*, magari perché, ad esempio, uno sviluppatore si occupa della parte relativa all'interfaccia e uno sviluppatore delle parte relativa alla gestione dei file, e alla fine potrebbe essere necessario far confluire i due "rami" dello sviluppo in un unico software.

Oppure, ancora, in un ramo dello sviluppo potrebbero essere svolti dei tentativi di implementazione di funzionalità che poi si possono rivelare non utili o non necessari, e quindi quel "ramo" potrebbe poi essere abbandonato.

Una rappresentazione grafica delle situazioni descritte è la seguente:

ESEMPIO

Il diagramma di FIGURA 5 illustra la sequenza temporale dello sviluppo di una ipotetica applicazione software (i valori numerici identificano le versioni).

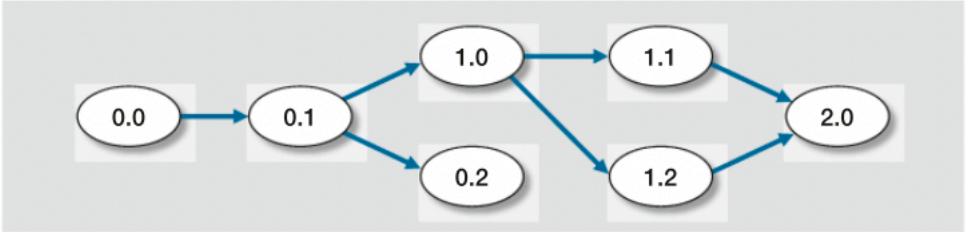


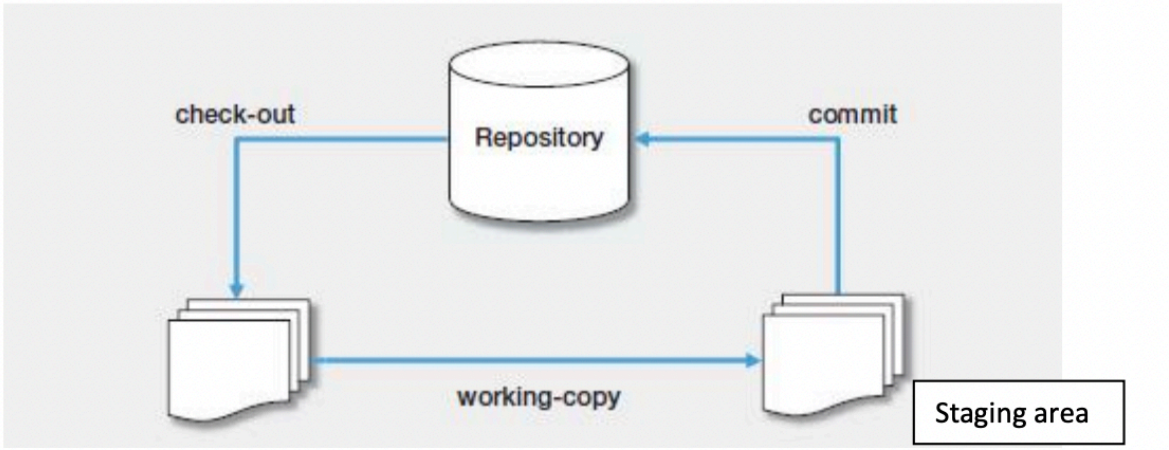
FIGURA 5

Il diagramma illustra una versione di codice che non è stata ulteriormente sviluppata (la versione 0.2) e due distinte versioni originate dallo stesso codice (le versioni 1.1 e 1.2 entrambe evoluzione della versione 1.0) che sono confluite in un'unica versione (la 2.0).

Un VCS consente quindi in ogni istante di "*ritornare*" ad una delle versioni del progetto "*fotografate*".

Una modalità molto utile dell'utilizzo di un VCS è quello di tenere un **ramo** (chiamato **branch**) per la versione stabile, testata, del progetto, e una versione "di sviluppo" su cui si lavora creando di volta in volta le nuove classi, le nuove funzionalità. Ogni volta che una modifica della versione di sviluppo è stata opportunamente testata e può considerarsi stabile, la si può far confluire nel ramo stabile (chiamato solitamente ramo master).

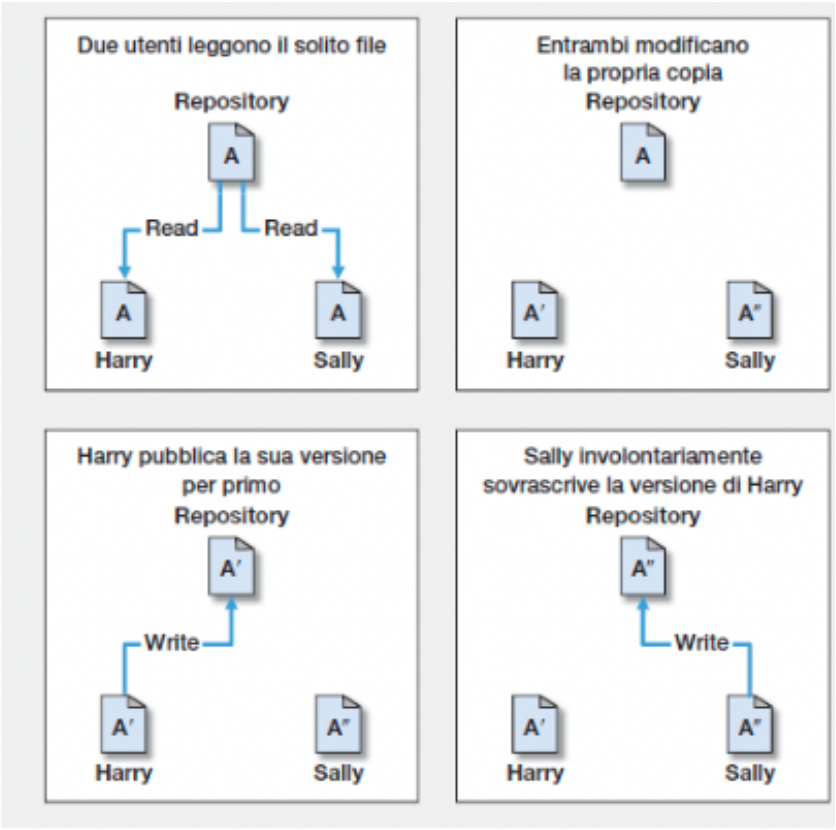
Il funzionamento di un generico VCS è rappresentato dal seguente disegno:



La **working copy** è la copia del progetto sul quale si lavora (l'insieme dei file e delle cartelle che costituiscono il progetto sull'HD). Ogni volta che essa viene modificata, si portano (con un apposito comando del software VCS) i file modificati in una "**staging**" area, una specie di "*anticamera*" nella quale si mettono i file modificati prima di essere "*fotografati*" per una nuova versione del progetto. Quando il programmatore ritiene che le modifiche apportate siano sufficienti per poter generare una nuova versione del prodotto, i file modificati vengono caricati dalla **staging area** in un contenitore chiamato **repository** del progetto. Questa operazione è chiamata **commit**. Un commit corrisponde a una fotografia della nuova versione del progetto che verrà conservata nel repository. Ogni volta che il programmatore lo riterrà necessario, potrà estrarre dal repository una versione precedente del progetto (ossia "fare un passo indietro"). L'operazione di estrazione dal repository è detta **check-out**. Una volta effettuato il checkout, il programmatore ha a disposizione nella working copy la versione del progetto estratta e potrà lavorare su di essa.

*N.B.: il **repository** può essere pensato come una cartella di lavoro speciale che mantiene un registro completo delle modifiche apportate ai file nel corso del tempo*

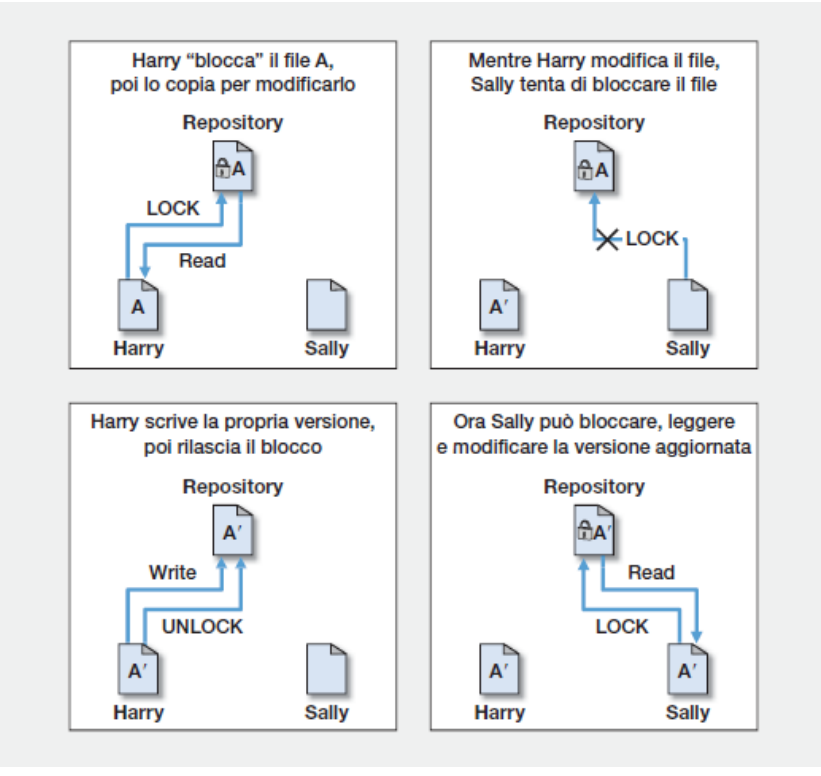
Un VCS non è solamente un sistema di *backup*. Infatti esso consente anche la gestione di file che possono essere modificati da due o più sviluppatori distinti. Quando più sviluppatori lavorano allo stesso progetto potrebbe infatti accadere che la modifica apportata su un file da uno sviluppatore possa venire involontariamente eliminata da un altro sviluppatore, come mostrato nel seguente esempio:



Le soluzioni fornite dai software VCS a questo problema sono le seguenti due tecniche:

1. *Tecnica lock-modify-unlock:*

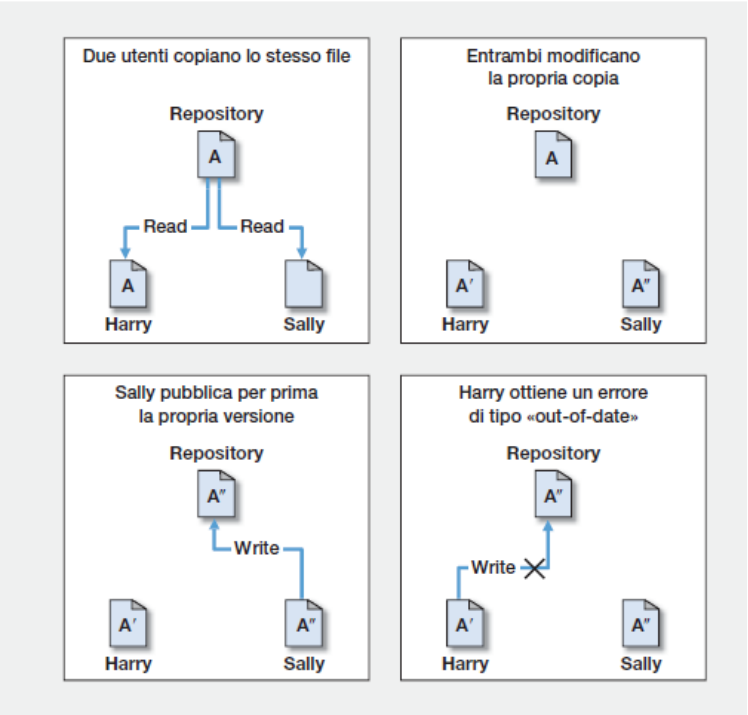
Il primo che accede al repository blocca l'accesso al file per gli altri sviluppatori, il file viene sbloccato solamente quando colui che l'ha bloccato carica le proprie modifiche con un commit.



Svantaggio: per un certo tempo altri sviluppatori non possono accedere ai file bloccati per lavorare. In particolare il problema è dovuto al fatto che quando si modifica il codice sorgente, pur modificando su un solo file la compilazione ha bisogno di molti altri file. Questo fa sì che Harry debba scaricare un'intera cartella di file (bloccandoli) non consentendo a Sally di procedere con il proprio lavoro neppure se essa deve lavorare su un altro file del progetto.

2. *Tecnica copy – modify – merge:*

Tutti gli sviluppatori possono scaricare e modificare i file, ma quando uno sviluppatore effettua un commit (o write), se il file è già stato modificato da qualcun altro, riceve un messaggio di errore (out of date). Per effettuare il commit del proprio file lo sviluppatore che ha ricevuto l'errore dovrà integrare le proprie modifiche con quelle effettuate dagli altri sviluppatori (tale operazione di "unione" delle diverse modifiche apportate è detta **merge**).



La tecnica è **più vantaggiosa** rispetto alla precedente perché spesso i diversi sviluppatori lavorano ad uno stesso progetto su file diversi (ad esempio su classi diverse). Nel caso in cui le modifiche riguardassero un file comune, gli sviluppatori devono concordare un'unica versione del file da commmittare.

Utilizzando la tecnica **copy-modify-merge**, lo sviluppatore segue il seguente *ciclo di lavoro*:

1. Effettua il checkout del progetto dal repository
2. Apporta le proprie modifiche
3. Verifica se le proprie modifiche non sono in conflitto con eventuali altre modifiche effettuate da altri nel repository.
4. Risolve eventuali conflitti in accordo con gli altri sviluppatori
5. Effettua un commit nel repository della propria versione del progetto

Ogni volta che viene effettuato un commit, ad esso viene associato un identificativo che lo distingue da qualunque altro commit. Il modo di identificare un commit cambia a seconda del VCS. Un commit corrisponde ad una versione del progetto in un determinato istante (una "fotografia" del progetto in quell'istante).

I due VCS più noti sono entrambi rilasciati con licenze open source e sono:

- **Subversion** (abbreviato con **SVN**)
- **Git** (sul quale si basa la piattaforma di hosting per progetti software **Github**)

IL VCS Git

E' un VCS che permette lo sviluppo **distribuito** fra più utenti.

VCS **centralizzato** (Subversion):

- C'è una sola copia "centrale" del progetto
- I cambiamenti vengono committati su questa copia
- Bisogna essere connessi a internet per poter committare

VCS **distribuito** (Git):

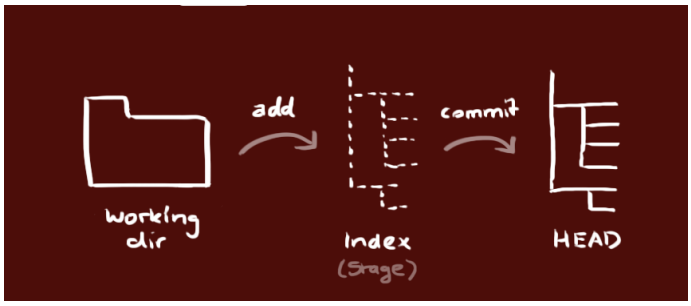
- Ogni copia può essere quella principale
- I cambiamenti vengono committati in locale
- Si può lavorare anche senza internet

Con Git è possibile creare dei repository per qualunque tipo di file.

Come funziona:

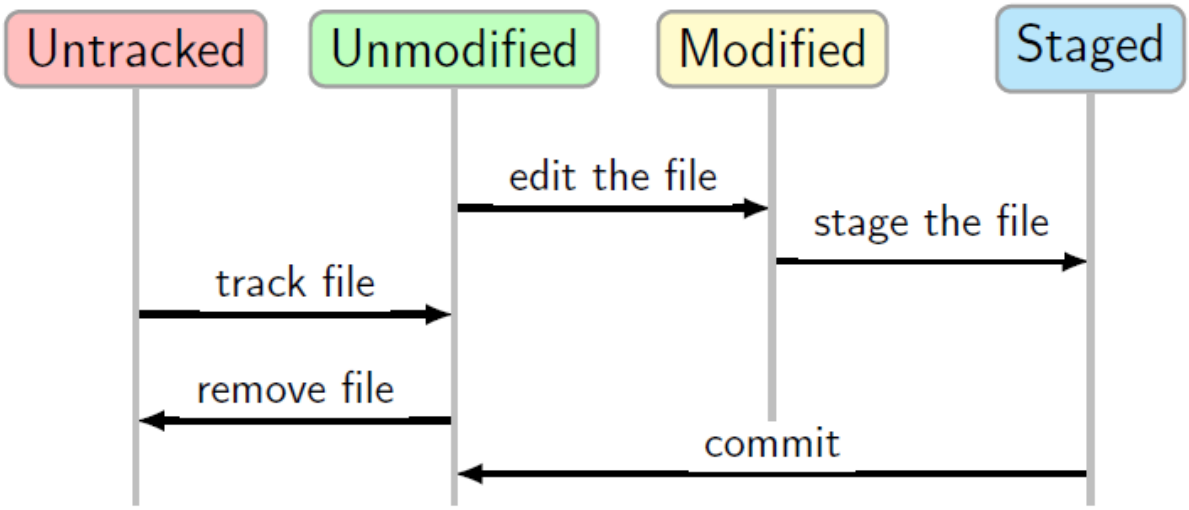
Si lavora ad un progetto per volta. Per ciascun progetto è possibile creare un repository (repo) locale. Ci sono 3 ambienti gestiti da Git:

- La nostra cartella di lavoro (la cartella contenente il progetto) o **working directory**.
- La **staging area** (un'area intermedia in cui possiamo indicare i file che vogliamo tracciare con Git, visto che non sempre sarà necessario tracciare tutti i file di un progetto). La staging area è una specie di area intermedia in cui vengono posti i file da tracciare dopo che sono stati modificati. In questa area dunque si "preparano" i file che costituiranno un nuovo repo.
- Il **repository** in cui le varie versioni vengono memorizzate, o meglio, "commitate".



***HEAD** è un puntatore speciale che punta alla versione attuale dell'ambiente di lavoro o al branch (ramo) attualmente selezionato. È essenzialmente un riferimento simbolico al commit più recente nel branch corrente.*

Ogni file del progetto si trova sempre in uno dei seguenti stati e passa da uno stato all'altro con le seguenti operazioni:



Un file sul quale si sta lavorando (un file di testo, un codice) viene generalmente modificato e committato più volte, quindi si trova ciclicamente negli stati **Unmodified**, **Modified** e **Staged**. Nello stato **Untracked** si trova solamente prima che si inizi a tracciarlo (tenerne traccia con Git) o quando si decide di non tenerne più traccia.

Comandi base

- Git è fatto per essere utilizzato principalmente da *linea di comando*
- Nel dubbio digitate **git help**
- Solitamente c'è il man anche dei sottocomandi (e.g. man git clone)
- Esistono anche delle **GUI** (gitg, gitk, git gui), possono avere un'utilità in alcuni casi.

git config user.name serve per visualizzare il nome dell'utente attualmente configurato in Git

git config user.email serve per visualizzare l'email dell'utente attualmente configurato in Git

git init serve a inizializzare un repository Git in locale e va digitato nella cartella che volete inizializzare come repository;

git clone serve per "clonare" un repository Git già esistente. Il repository può essere sia remoto che locale

git clone <url del repo> lo clona in una cartella chiamata come il repository

git clone <url del repo> <nomecartella> per clonarlo in un'altra cartella

Generalmente questo comando esegue il checkout da un server remoto di repository (ad esempio GitHub) consentendo di avere una copia locale di un repository su cui lavorare.

git status mostra una serie di informazioni:

- Su che branch siete
- Se siete "avanti" o "indietro" rispetto al branch remoto che state tracciando
- I file modificati non ancora nella staging area

- I file nuovi non ancora tracciati
- I file tracciati che sono stati rimossi

git add serve per spostare tutti i file modificati (o non ancora tracciati) nella staging area. Nella staging area vi saranno quindi i file che si vuole aggiungere al repository nel successivo commit. Si noti che nella staging area vengono **aggiunte solamente le modifiche** ai file che sono cambiati.

- **git add –A** vengono aggiunte alla staging area le modifiche ai file già tracciati e i file non ancora tracciati
- **git add -a** aggiunge tutte le modifiche sui file già tracciati
- **git add <nome file>** aggiunge solo il file specificato. Se <nome file> è una cartella aggiunge tutte le modifiche dei file compresi nella cartella.

N.B.: *git add non aggiunge il file alla staging area, aggiunge le modifiche fatte al file nel momento in cui viene eseguito il comando*.

.gitignore Per dichiarare a Git i file che vogliamo esplicitamente ignorare, dobbiamo inserirli nel file .gitignore nella root del repository

N.B.: **.gitignore è un file** del vostro repository, quindi va addato e committato.

git commit Crea un commit contenente le modifiche che si trovano nella staging area

- I file che vengono trasferiti nel repository sono quelli presenti nella staging area
- Bisogna aggiungere un messaggio di commento per indicare cosa è stato modificato rispetto al commit precedente: git commit –m <messaggio> (esempio: “aggiunto capitolo 2”)
- Se non viene specificato il messaggio si apre un editor apposito in cui aggiungere il messaggio (per attivare i comandi in basso nel menu dell’editor usare ctrl+tasto)
- E’ consigliato (best practice) fare commit piccoli e frequenti.
- E’ consigliato che un commit di un progetto comprenda tutti i file che consentano la compilazione del progetto

Ogni **commit** è identificato univocamente da un id formato da un hash

Di solito (date le proprietà degli hash) se vogliamo riferirci ad uno specifico commit bastano le prime 6 o 7 cifre dell’hash

git tag Quando ci sono commit particolarmente importanti, ad esempio una versione definitiva del progetto, è possibile assegnare al commit un **tag** (ad esempio 1.0, 1.1 ecc..):

git tag <nome tag>: assegna il tag all’ultimo commit eseguito.
git tag<nometag><idcommit>: assegna un tag al commit specificato, anche se non è l’ultimo.

N.B.: *Il tag può essere usato al posto dell’id del commit per individuare un commit*.

git tag –a <nome tag>: aggiunge un tag annotato, che include chi ha creato il tag, una data e un commento.

git log Mostra la history dei commit.

git reset Toglie tutto dalla staging area ma non modifica i file sulla working directory, quindi le modifiche effettuate sui file dell’HD rimangono.

git reset --hard <id del commit>

Con questo comando i file nel progetto nella cartella di lavoro vengono rimpiazzati da quelli del commit selezionato. ATTENZIONE: TUTTE LE EVENTUALI MODIFICHE SVOLTE NELLA CARTELLA DI LAVORO VERRANNO PERSE. Comunque sarà possibile ritornare sempre al commit “avanti” conoscendone l’ Id. L’Id dei commit “avanti”

li vedo con la utility grafica **gitk**, che posso invocare dal prompt dei comandi semplicemente digitando: **gitk**.

git checkout <nomefile> Recupera un singolo file che è stato modificato nella working directory e lo riporta alla situazione dell'ultimo commit.

git rm <nome file> Rimuove un file dal progetto, anche dalla working directory sull' HD (si può eventualmente ripristinare con un reset --hard di un commit precedente).

Solitamente si usa per rimuovere la tracciatura di un file quando esso viene eliminato dal progetto. Quando il file viene eliminato dal progetto senza "avvisare" Git della rimozione, Git continua ad avvisare l'utente che non trova più il file tracciato. Per evitare questo bisogna quindi eseguire un git rm oppure un git add (anche con git add Git si accorge del cambiamento dato dall'eliminazione del file)

git rm --cached <nome file> Toglie la tracciatura di un file, ma esso sarà ancora presente nella working directory nell' HD, infatti il file rimosso in questo modo viene indicato da git status come non tracciato

git rm --cached <nome cartella> -r Rimuove una intera cartella dalla tracciatura.

git show <id commit> Mostra informazioni sul commit. Chi l'ha eseguito, quando, il messaggio, le differenze con il commit precedente.

git mv <nome1 file> <nome2 file> Sposta o rinomina un file tracciato.

git blame <nome file> Mostra il file riga per riga indicando, per ogni riga, chi l'ha aggiunta e in quale commit.

git diff Mostra le differenze fra ciò che è sull'HD nella Working directory e ciò che è nella staging area. Vengono mostrati, per ogni file di testo, le righe eliminate (con accanto dei -) e le righe aggiunte (con accanto dei +)

git diff --staged: mostra la differenza fra l'ultimo commit e la staging area

I branch

Con il termine **branch** indichiamo una ramificazione del progetto. Il ramo principale, creato di default da Git è chiamato **MASTER**.

In Git ogni commit contiene un puntatore al commit precedente, è grazie a questo che Git è in grado di ricostruire la history di un repository.

Un puntatore apposito chiamato **HEAD**, punta sempre al commit sul quale si sta attualmente lavorando.

*Un **branch** non è altro che un puntatore che punta ad un particolare **commit**.*

Quando si vuole modificare un progetto ma non si è certi se le modifiche che si intende apportare saranno quelle giuste, è opportuno generare un altro ramo del progetto: un nuovo branch (che possiamo chiamare, ad esempio SVILUPPO, oppure EXPERIMENT). Generalmente gli sviluppatori mantengono sempre un branch stabile (master) e un branch su cui eseguire e testare le modifiche. Quando poi le modifiche funzionano i due branch vengono fatti confluire (con una operazione chiamata merge)

git branch verifica quanti branch sono presenti

git branch <nome branch> In questo modo si crea un branch, che in realtà non è altro che un puntatore, che punta al commit attualmente in uso.

git checkout <nome branch> Per associare la working directory ad un altro branch (*ossia per spostarmi su un altro ramo*)

Scorciatoia: per creare un nuovo branch e contemporaneamente associare la cartella di lavoro a questo nuovo branch (insieme dei due comandi precedenti): **git checkout –b <nome branch>**

git branch –d <nome branch> Per eliminare un branch che non serve più

git merge <nome branch> Per unificare nuovamente i due percorsi (*operazione di merge*)

*Il risultato di un'operazione di **merge** potrà andare a buon fine oppure generare dei conflitti da risolvere, questo dipende da quali file sono stati modificati nei commit dei due branch. Vediamo i casi possibili*:

Caso A: FAST FORWARD MERGE
Si verifica nel seguente caso:

- dal branch master si crea un nuovo branch
- si eseguono dei commit sul nuovo branch (per comodità lo chiameremo ***experiment***) mentre il branch master non viene modificato: *Poiché **experiment** non è altro che **master** con l'aggiunta di alcune modifiche, il merge dei due branch non può generare conflitti (ossia non ci sono modifiche diverse sullo stesso file apportate nei due branch*.

Poiché master non è stato modificato, la richiesta di merge diventa semplicemente "fai diventare **master** come **experiment**", quello che accade è che *dopo il merge i due puntatori master ed experiment puntano allo stesso commit*.
Il merge è di tipo **fast forward**, non ci sono conflitti ed infatti viene eseguito correttamente. Il merge fast forward da un branch verso un altro può sempre avvenire quando esiste un percorso orientato che consente al branch "più indietro" di raggiungere quello "più avanti".

CASO B: 3-WAY MERGE

Avviene quando sui due branch vengono eseguiti dei commit, quindi entrambi i branch hanno committato delle modifiche sui file di progetto:

Cosa accade quando si esegue il merge? Non essendoci percorso orientato fra **experiment** e **master** non è possibile eseguire un merge fast forward. In questo caso possono accadere 3 cose, in base alle modifiche introdotte nei due branch. Ipotizziamo di essere su **master** e di voler fare il merge con **experiment**:

1. I due branch hanno introdotto la stessa modifica nella stessa sezione del file
2. i due branch hanno introdotto modifiche diverse in sezioni diverse del file

In questi due casi non c'è alcun conflitto, le modifiche vengono apportate e si genera automaticamente un nuovo commit

3. I due branch hanno introdotto, nella stessa sezione del file, modifiche diverse. In tal caso si generano dei conflitti. Sarà necessario risolvere i conflitti dopo di chè sarà necessario eseguire il commit del file modificato.

Una volta eseguito il merge, **master** punterà al nuovo commit generato automaticamente. Il nuovo commit è particolare perchè ha **due puntatori**, uno per ciascun branch del merge.

A questo punto sarà possibile sia continuare ad operare in parallelo sul branch "experiment" oppure far confluire anche "experiment" sul "master" eseguendo un merge fast forward (ora è possibile farlo perché c'è il percorso orientato)

Ora vediamo cosa succede quando si presentano dei conflitti e come si possono risolvere:

git avvisa che non è possibile effettuare il merge perché i due rami hanno modifiche diverse sulla stessa sezione quindi non sa quale delle due versioni usare per fare il merge. Chiede quindi di risolvere il conflitto e poi svolgere il commit su master.

I conflitti si possono risolvere semplicemente apportando modifiche con un editor di testo, oppure esistono opportuni software da installare. Facendo commit piccoli è sufficiente usare l'editor di testo.

- risolviamo il conflitto visualizzando il file "da unire" e apportando le opportune modifiche.
- Dopo aver eseguito la correzione del conflitto, è necessario eseguire un git add –A e un git commit per creare il nuovo commit contenente il file corretto dopo il merge

Utilizzo di GitHub e collegamento di un repository locale

Clonare un repository:

git clone <url_del_repository_remoto>

Aggiornare il repository locale:

git pull origin main

Creare un repository su GitHub e collegarlo ad un repository locale:

1. Crea un nuovo repository su GitHub
2. Collega il tuo repository locale al repository su GitHub utilizzando il comando:

git remote add origin <URL del repository su GitHub> (git remote remove origin per scollegarsi dal repository remoto**)**

3. Rinomina il branch master in main:

git branch -M main

4. Invio dei Commit al Repository su GitHub:

git push -u origin main oppure **git push origin <nome del branch>**

per inviare a GitHub solo uno specifico branch, si può poi eseguire il merge su Github