Il controllo delle Versioni

Premessa

Ogni programmatore, soprattutto dopo che ha incidentalmente perso un proprio lavoro almeno una volta, inizia a perfezionare e ad attuare tecniche proprie di salvataggio dei codici sorgenti, rinominando i file o copiando intere directory su dischi esterni, a volte in formato compresso per ridurre lo spazio occupato.

Spesso è necessario effettuare il salvataggio non solo a scopo di backup, ma anche per **storicizzare la versione distribuita** ogni qualvolta la si rilascia al cliente finale; in ogni progetto lo sviluppo delle nuove versioni deve infatti "convivere" con gli aggiornamenti "on fly" dovuti alla correzione di eventuali bug riscontrati in corso d'opera.

Inoltre, lo **sviluppo del software**, che sappiamo essere un processo lungo generalmente fatto in squadra, non termina con la consegna del prodotto: la **manutenzione**, sia adattativa sia migliorativa, segue il prodotto sino alla sua dismissione o alla sua sostituzione con nuove soluzioni alternative.

> Lo sviluppo di applicativi per il Web risulta essere ancora più articolato e, nel caso si voglia conservare un'intera versione, è necessario replicare l'intero dominio; ciò, con l'accumularsi delle release, può diventare molto complesso.

Pertanto, è estremamente raro che una funzionalità del codice rimanga invariata nel tempo, perché si apportano modifiche e migliorie in fase di sviluppo e, soprattutto, in tempi successivi al suo rilascio: è necessario introdurre un sistema di identificazione del codice in modo che sia sempre possibile conoscere le sue funzionalità e l'evoluzione che le stesse hanno avuto nel tempo.

Numerazione delle release/versioni

Spesso, per identificare le diverse release/versioni, si utilizza un sistema di numerazione mediante tre contatori su tre livelli che vengono incrementati a partire da **1.0.0**:

1. **major level** (**release**): indica una modifica sostanziale nel prodotto che può implicare modifiche nell'architettura, per esempio una nuova veste grafica oppure il passaggio a una nuova piattaforma;
2. **minor level** (**versione**): si migliorano le funzionalità esistenti oppure si aggiungono nuove funzionalità alla versione precedente e/o aggiornamenti di natura fiscale;
3. **patch level** (**correzione**): si rimuovono errori/malfunzionamenti.

> Non esiste una regola standard per la numerazione di release e versioni, vi sono molti metodi differenti; di fatto, è sufficiente tenere a mente due regole essenziali:
> 1. ogni programmatore deve assegnare un numero a ogni nuova versione del programma;
> 2. a ogni evoluzione del software il numero deve essere maggiore del precedente.

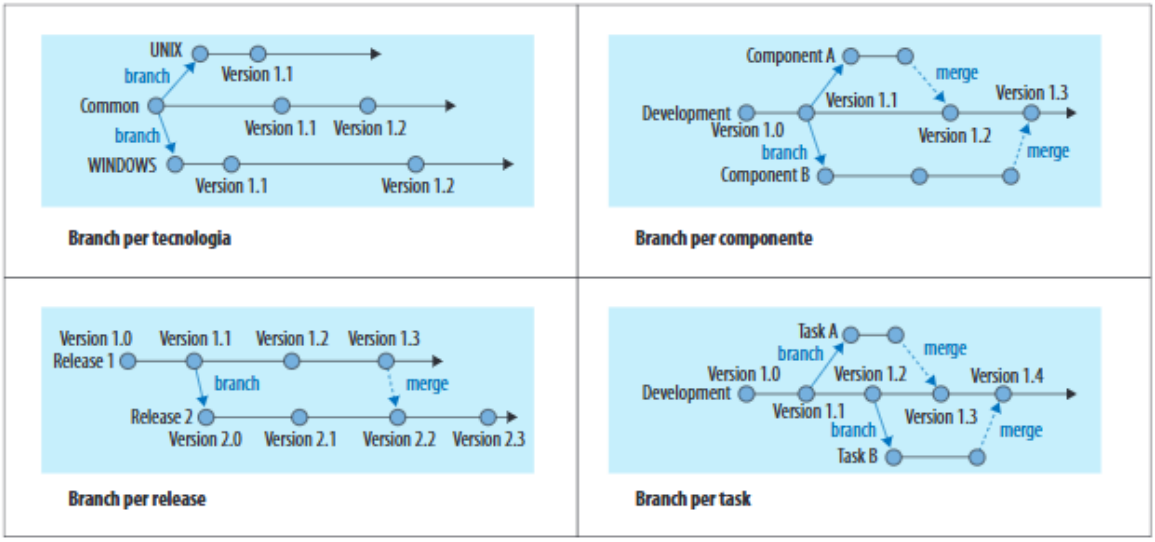
I sistemi di versionamento

Un **sistema di controllo di versione** (**VCS**, *Version Control System*) consente di gestire le modifiche apportate ai file di un progetto su cui tipicamente lavorano più persone.
Scopo di un VCS è di realizzare una corretta gestione delle modifiche garantendo le seguenti caratteristiche: **reversibilità**, **concorrenza**, **annotazione**.
La **reversibilità** è la capacità di un VCS di poter sempre tornare indietro in un qualsiasi punto della storia del codice sorgente: può capitare di accorgersi di aver introdotto un errore (al posto di una miglioria!) ed è necessario ripristinare l'ultima versione stabile del software.
La **concorrenza** è la prerogativa che permette a più persone di apportare modifiche allo stesso progetto, facilitando il processo di integrazione di pezzi di codice sviluppati da due o più sviluppatori.
L'**annotazione** è la funzione che consente di aggiungere spiegazioni e osservazioni alle modifiche apportate; a ogni versione si possono aggiungere delle note che spiegano le modifiche effettuate, le migliorie apportate e ogni altra informazione che può essere utile a tutto il team di lavoro.

Tutti i programmatori sono stati formati con il vecchio dogma: "se qualcosa funziona, non cambiarla"; ora, con i sistemi VCS, è possibile apportare modifiche e migliorie senza il timore di "non poter più tornare indietro".

- **branch**: possiamo pensare a un branch come a una diramazione del percorso di sviluppo: questa funzionalità è di fondamentale importanza e di grande utilità per gli sviluppatori, dato che nella realizzazione di un'applicazione potrebbe essere necessario realizzare degli sviluppi paralleli su un progetto.

ESEMPIO
Riportiamo quattro esempi classici dove il branching viene effettuato per una specifica esigenza.



- **fork**: serve per creare una copia (clone) di un repository per poi effettuare modifiche.

VCS centralizzati e distribuiti

Esistono due tipologie di sistemi VCS:

1. i **sistemi di versionamento centralizzati CVCS**, progettati per avere una singola copia completa del repository, ospitata in uno o più server, dove gli sviluppatori salvano le modifiche che hanno apportato;
2. i **sistemi di versionamento distribuiti DVCS**, dove ogni sviluppatore ha una propria copia locale di tutto il repository e può salvare le modifiche ogni volta che vuole.

> Se un **VCS** dipende completamente da un server centralizzato, si ha come conseguenza che, se il server o il collegamento va offline, gli sviluppatori non saranno in grado di salvare le modifiche: se il repository centrale viene danneggiato, e non esiste un backup, la storia del progetto andrà persa.

Anche in **Git** esiste un server remoto che contiene l'intero repository condiviso da tutti gli sviluppatori ma, se in un certo momento il server che ospita il repository è **offline**, gli sviluppatori possono continuare a lavorare senza alcun problema, rimandando la registrazione delle modifiche nel repository condiviso ad un secondo momento.

> Le differenze tra i due tipi di sistemi possono essere riassunte nella seguente espressione: con un **CVCS** abbiamo una completa dipendenza da un server remoto per svolgere il controllo di versione, mentre con un **DVCS** il server remoto è solo un'opzione per condividere le modifiche.

A volte, al posto di **VCS** vengono utilizzati altri due acronimi, **SCM** (*Source Code Management*) oppure **RCS** (*Revision Control System*), ma sostanzialmente indicano la stessa tipologia di prodotto, cioè un tool specifico per il controllo delle versioni durante lo sviluppo di un applicativo software, che sia un singolo programma sviluppato da un solo programmatore oppure un progetto.

In commercio sono presenti molteplici prodotti per il controllo delle versioni; di seguito riportiamo i più utilizzati:

- Concurrent Versions System (CVS)
- **Subversion SVN**
- **Git** e **GitHub**
- Mercurial
- Monotone

Alcuni di essi sono completamente gratuiti o a utilizzo limitato (per esempio 50 ore al mese gratuite come Git), altri a pagamento: in generale si basano tutti su pochi semplici concetti.
Tra questi noi utilizzeremo **Git**, che è un sistema di versionamento distribuito e si contrappone a quelli centralizzati come SVN.

SVN

SVN, acronimo di *Subversion*, è stato uno dei primi sistemi open source per il controllo di versione a diventare ampiamente utilizzato come naturale successore ed estensione del primo software CVS. Presenta miglioramenti nella realizzazione di molte funzionalità, più veloci, più sicure, più flessibili.
Tra i client SVN ricordiamo:

- Apache Subversion, non più aggiornato dall'aprile 2022, come si può vedere all'indirizzo **http://subversion.apache.org/**;
- TortoiseSVN, standalone scaricabile dall'indirizzo **http://tortoisesvn.net/**;
- Subclipse, plug-in di Eclipse scaricabile dall'indirizzo **http://subclipse.tigris.org/**.

L'architettura di SVN è **centralizzata**, il che significa che tutte le revisioni del codice sorgente vengono archiviate in un unico repository centrale; è molto più lento di Git e non gestisce agevolmente i conflitti: SVN è comunque ancora utilizzato in alcune organizzazioni, soprattutto quelle che hanno una lunga storia con questo sistema di controllo versione e non hanno bisogno di tutte le funzionalità più avanzate offerte da Git.

i singoli sviluppatori lavorano estraendo dal *repository* i file da modificare (questa operazione è denominata *read* o *check-out*) e trasferendoli nuovamente nel *repository* una volta apportate e verificate le modifiche (questa operazione è denominata *write* o *commit*) (FIGURA **6**).

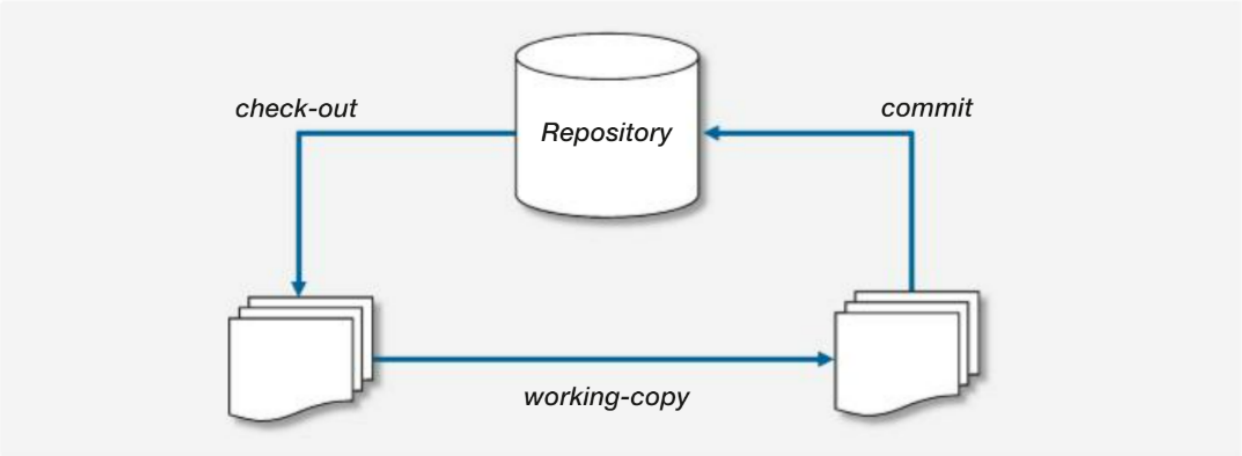
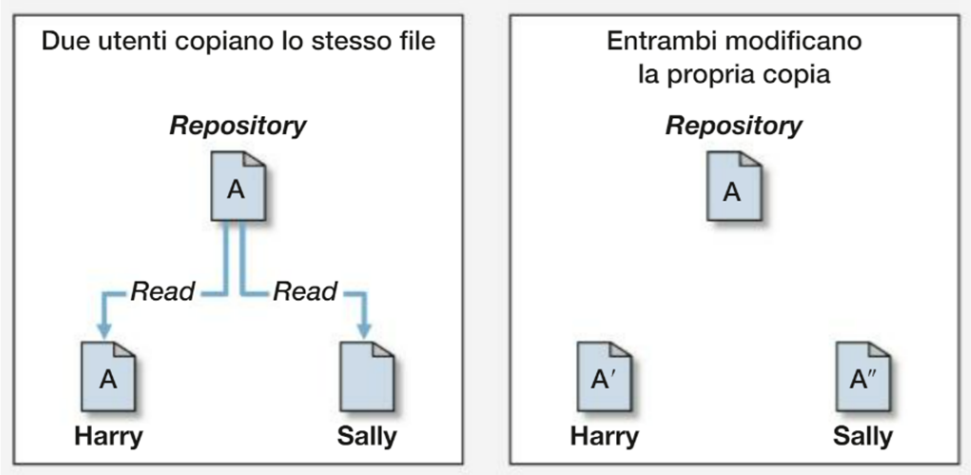


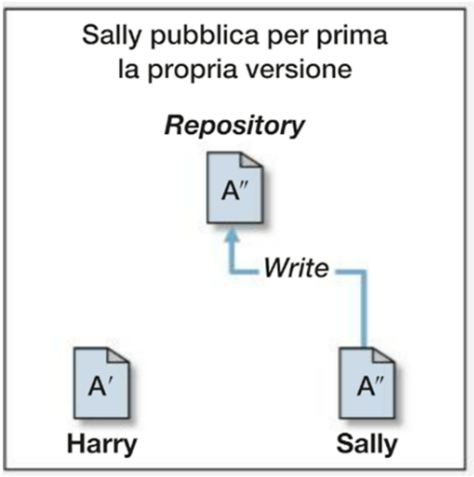
FIGURA **6**

La copia di lavoro (*working-copy*) è una semplice directory del *file-system* del computer utilizzato per lo sviluppo.

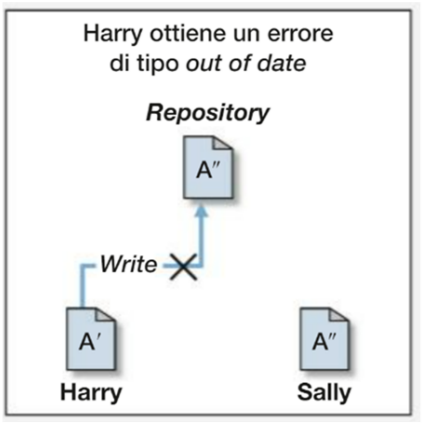
L'operazione di Commit genera automaticamente una nuova revisione identificata da un numero progressivo al quale lo sviluppatore può aggiungere informazioni relative alla modifica fatta

Ma cosa succede quando due sviluppatori modificano il SW contemporaneamente la stessa parte di codice come nell'esempio sotto?

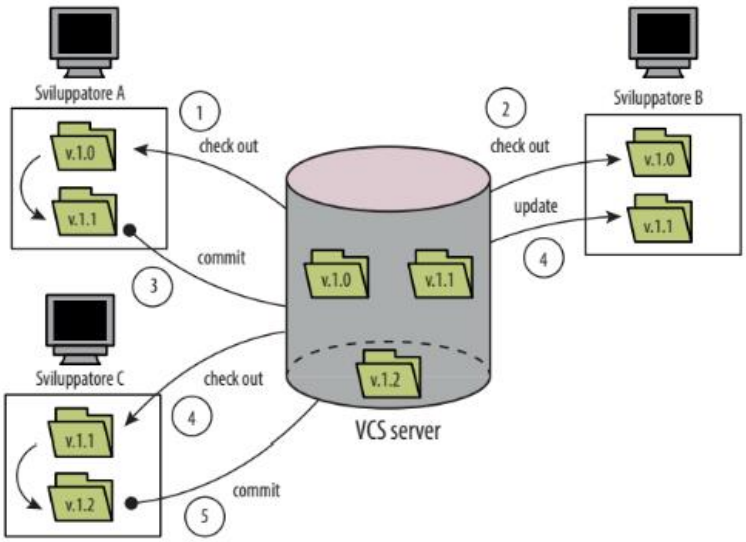




In dipendenza del tipo di modifiche, SVN riesce a fare in autonomia il «merge» delle due versioni. Altrimenti



Harry non potrà fare il commit delle proprie modifiche finchè non avrà definito con Sally come procedere (o più spesso il responsabile del team avrà defintо come procedere)



Questo schema di lavoro prende il nome di **modello copy/modify/merge** e si differenzia notevolmente dal classico modello secondo il quale più programmatori accedevano in concorrenza ai diversi file di un progetto (modello lock/modify/unlock); quest'ultimo garantiva la massima sicurezza da problemi di manomissione contemporanea involontaria ma non ottimizzava nel modo migliore le operazioni.

Lo schema copy/modify/merge si applica a tutti i VCS sia centralizzati che distribuiti.

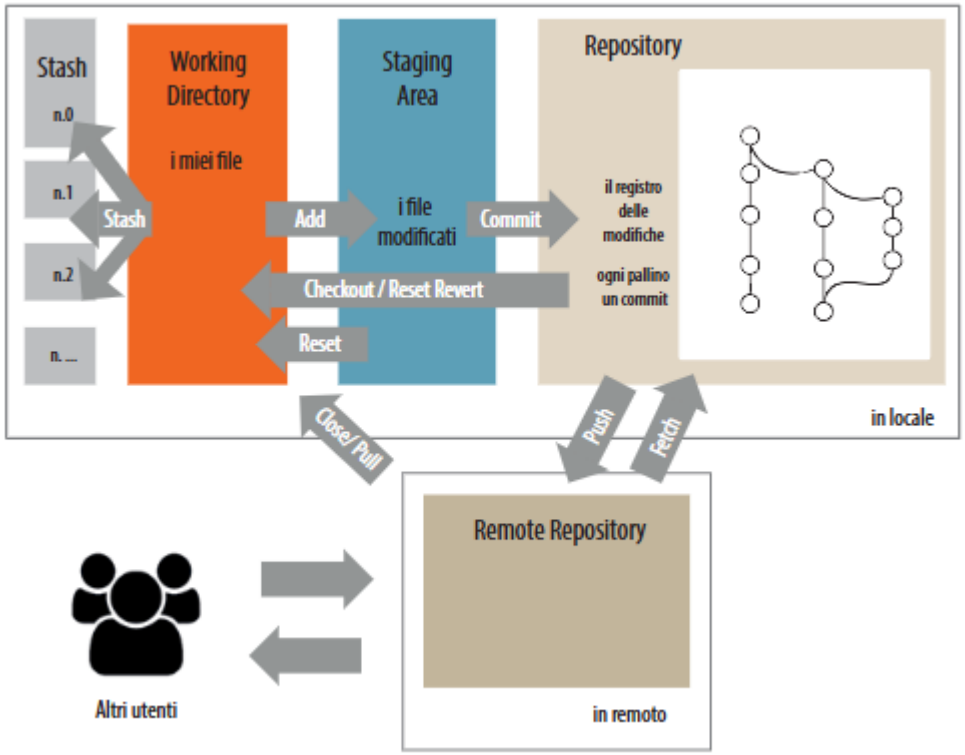
Che cos'è Git

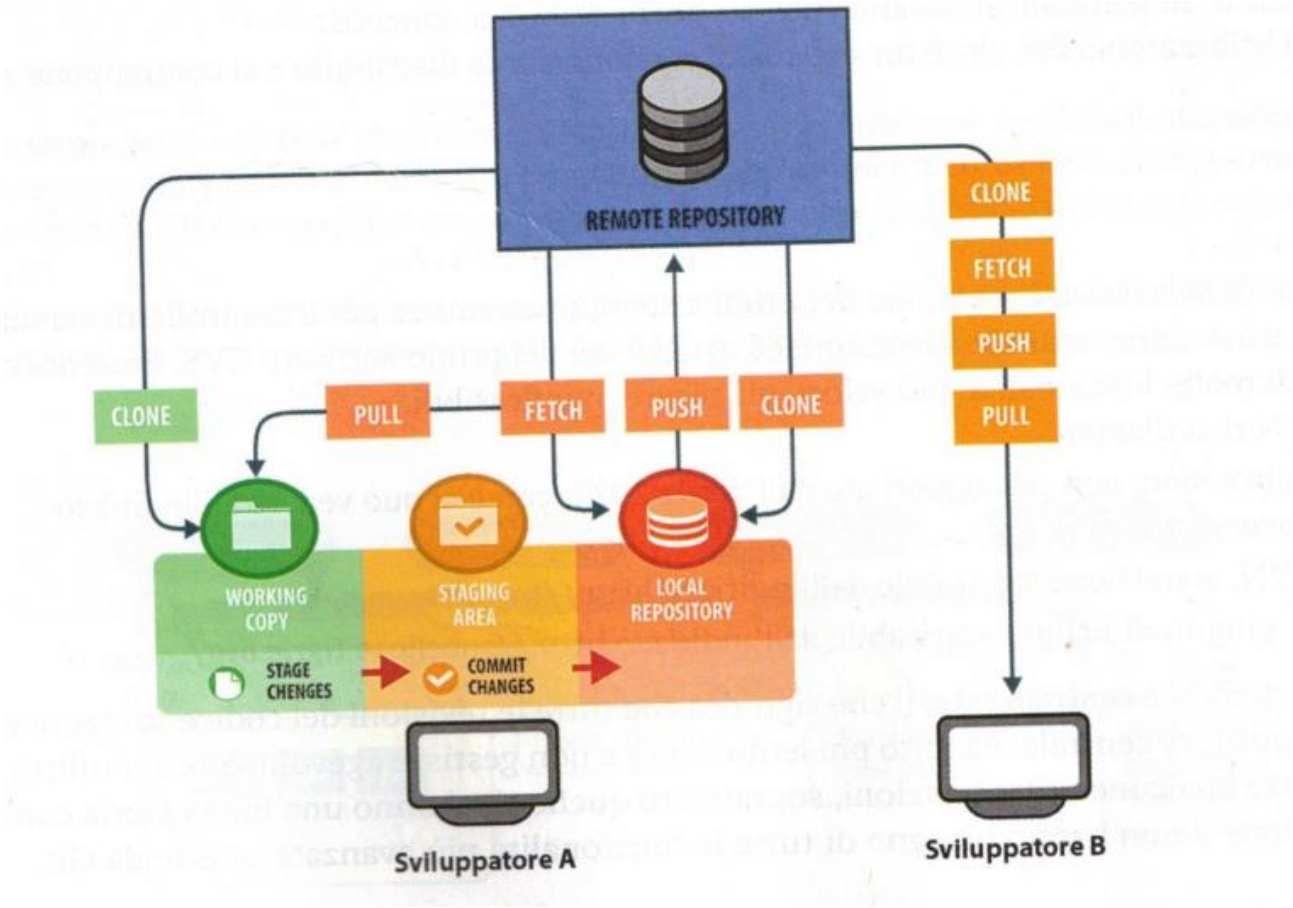
Git è un sistema controllo di versione distribuito che permette di effettuare le operazioni di gestione delle versioni "quasi" automaticamente, grazie anche alla possibilità di utilizzare una semplice interfaccia grafica che consente di gestire intere directory: permette cioè di salvare, ogni qualvolta lo si ritenga necessario, lo stato del lavoro e di caricare/ripristinare ognuno degli stati precedentemente salvati.

> Essendo un software distribuito, ogni sviluppatore ha una **copia locale** del repository Git sul proprio computer e può apportare modifiche al codice sorgente in modo indipendente: al momento che un membro del team raggiunge una situazione "significativa" può inviare la sua modifica al repository centrale e fare in modo che altri membri del team possano accedere e "fare proprie" le modifiche da lui apportate, integrandole nel loro sviluppo del progetto.

Le azioni che possono/devono essere fatte dallo sviluppatore che lavora in collaborazione con Git sono illustrate nella figura seguente.

Si noti che le azioni di commit/checkout già descritte per SVN in questo caso non agiscono sul progetto condiviso, ma sulla copia in locale.





Quali sono i passi da fare per lavorare in team con Git

- La prima volta che dobbiamo lavorare con il progetto si fa il "**clone**" che permette di avere la copia locale di un repository remoto
- In ogni caso, prima di iniziare a lavorare è consigliato fare un "**pull**" (vedi sotto per i dettagli)
- Si apportano le modifiche al progetto
- Ogni volta che si ritiene opportuno (quindi siamo confidenti che le modifiche siano corrette e funzionanti), si aggiungono "**add**" eventuali nuovi file, si rimuovono "**rm**" alcuni file non più necessari e si fa il "**commit**" sul repository locale
- Il "**commit**" si può annullare con un "**revert**"
- Se vogliamo valutare lo stato del nostro repository rispetto a quello remoto, in quanto potrebbero essere nel frattempo state inserite modifiche da altri sviluppatori, si può usare il comando "**fetch**" che evidenzia le modifiche senza toccare il repository locale
- Se vogliamo direttamente aggiornare il proprio repository locale (con le modifiche eventualmente fatte nel frattempo da altri sul repository remoto) usiamo il comando "**pull**"
- In questo secondo caso durante il merge delle due versioni potrebbero emergere dei conflitti che git può risolvere autonomamente o segnalare per una risoluzione manuale
- Risolti eventuali conflitti si può fare "**push**" sul server. Se seguito il processo precedente, difficilmente potranno emergere conflitti (che comunque git evidenzierà e supporterà nella risoluzione)

GitLab e GitHub

GitLab e **GitHub** sono piattaforme di gestione del codice sorgente basate su **Git**, utilizzate per lo sviluppo collaborativo di software. Entrambe offrono funzionalità di controllo versione, gestione dei repository e strumenti per DevOps, ma hanno caratteristiche distintive.

L'acronimo **DevOps** sta per:

DEV = **Development** (Sviluppo)
OPS = **Operations** (Operazioni IT)

Indica l'unione tra i team di **sviluppo software** e **gestione delle operazioni IT**, con l'obiettivo di migliorare la collaborazione, automatizzare i processi e accelerare il ciclo di vita delle applicazioni, dalla progettazione al rilascio in produzione.

GitHub: Piattaforma di Hosting Git

GitHub è una piattaforma di sviluppo e collaborazione per progetti open source e privati. È noto per la sua vasta comunità di sviluppatori e repository pubblici.

GitLab: Piattaforma DevOps Completa

GitLab è una piattaforma DevOps open source che supporta l'intero ciclo di vita del software, dall'hosting del codice alla distribuzione. Offre anche una versione **self-hosted**, utile per aziende con esigenze specifiche di sicurezza e personalizzazione.

Una delle funzioni utili offerta da GitLab è la “Merge Request”.

Se le politiche aziendali richiedono che ogni sviluppatore e/o team debba lavorare su branch diversi, il processo da seguire è:

- Occorre fare una “**fork**”, cioè una copia, del branch da cui si vuole partire e per cui non si hanno i diritti di scrittura
- Si sviluppano le modifiche in un proprio branch e si fanno gli add ed i commit descritti precedentemente
- Se si vogliono inserire le modifiche nel branch su cui non si ha diritto di scrittura occorre fare una **“merge request”**
- Se il proprietario del branch accetta, si può effettuare il merge