Source Control Management (SCM)

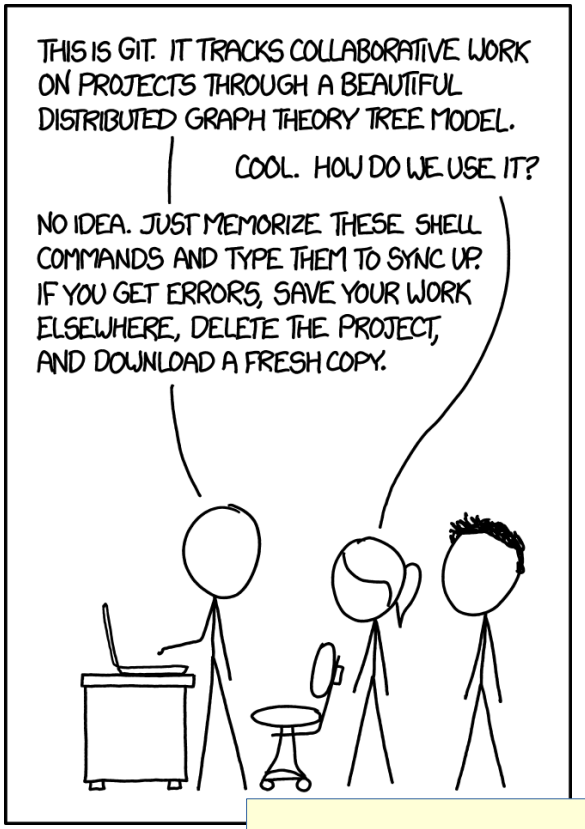
- Sinonimo: Version Control System (**VCS**)
- Obiettivi
  - Mantenere traccia dei cambiamenti nel codice; sincronizzazione del codice tra utenti
  - Cambiamenti di prova senza perdere il codice originale; tornare a versioni precedenti
- **Centralized** → architettura client/server (CVS, Subversion, ...)
  - Repository centralizzato con lo storico completo del progetto
    - (codice sorgente, risorse, configurazioni, documentazione, …)
  - check-out/check-in (lock del file), branch/merge (conflitti)
- **Distributed** → architettura peer-to-peer (**Git**, Mercurial, ...)
  - Repository clonato su tutte le macchine
  - Gran parte del lavoro può essere fatto offline

Modelli di Versioning

- Naive
  - A e B modificano lo stesso file allo stesso tempo; A salva i cambiamenti nel repository; subito dopo B salva i suoi, nascondendo i cambiamenti di A
    - Non accettabile, i cambiamenti di A rischiano di essere persi
- Lock – modify – unlock
  - Il file può essere cambiato solo da un utente per volta.
  - Semplice, ma ha una serie di problemi:
    - Lock dimenticati; serializzazione anche quando non è necessario; gestione dipendenze in altri file …
    - Era il comportamento standard nei vecchi sistemi centralizzati
- Copy – modify – merge
  - Si modifica il file nell'area di lavoro locale, poi si fa il merge con la copia nel repository.
  - Necessita una accurata gestione dei conflitti
    - È il modello comunemente usato nei sistemi moderni

Git

- 2005 by Linus Torvalds et al.
  - Pensato per lo sviluppo di Linux
- Integrato nei principali ambienti di sviluppo
- Client ufficiale *(comodo per uso avanzato)*
  - Scaricabile da https://git-scm.com/
  - Disponibile per i principali sistemi operativi
  - Codice sorgente: https://github.com/git/git
- Il repository remoto può essere su
  - **github.com**, gitlab.com, bitbucket.org, …



https://xkcd.com/1597

Configurazione di Git

- Per verificare la configurazione corrente
  - git **config** --list
- La possiamo specificare a tre livelli, vince la più specifica tra
  - Sistema: Nel folder di installazione del programma
    - *(sconsigliato, non entriamo nei dettagli)*
  - Globale: Nel folder dell'utente corrente, file **.gitconfig**
  - Locale: Nel folder del progetto corrente, file **.git/config**
- È richiesto il nome e l'email dell'utente, qui configurati globalmente
  - git **config** --global user.name "Emanuele Galli"
  - git **config** --global user.email egalli64@gmail.com

Repository locale – area di lavoro

- Attenzione a non creare un repository Git in un altro repository Git!
- Eseguendo il comando git **init** in una directory
  - Si crea una directory **.git** → un nuovo repository Git locale
  - La directory corrente diventa l'area di lavoro del nuovo repository
- Un file nell'area di lavoro può essere, rispetto al repository
  - Sconosciuto (untracked)
  - Modificato – cambiato rispetto all'ultima versione
  - Indicizzato (staged) – segnalato a git per il prossimo commit
  - Committato – nello stato corrente
- Per verificare lo stato dell'area di lavoro e di stage: git **status**

Add e commit

- Dato un file nell'area di lavoro, sconosciuto a git o modificato
  - Si aggiunge la prima (o una nuova) versione del file nell'indice: git **add** <filename>
    - Oppure, git **restore** <filename> per ritornare alla versione più recente nel repository
  - Si aggiorna il branch corrente del repository locale: git **commit -m** <message>
    - <message>: un messaggio *(tra doppi apici)* che spieghi perché si cambia il repository, es: "fix ticket #42"
  - Si può fondere *add e commit* in un unico comando: git **commit -am** <message>
    - Aggiunge all'indice e commita tutti i file modificati – ma solo quelli già noti a Git!
- Se vogliamo controllare i cambiamenti nell'area di lavoro prima di commitare: git **diff**
  - Con l'opzione --staged si opera sull'indice (area di staging)
  - Per togliere un file aggiunto all'indice: git **reset** <filename> o git **restore** --staged <filename>
- Il comando git **log** mostra lo storico dei commit, con relativi messaggi
  - git **log -p** (patch) mostra il log con i cambiamenti tra i commit
  - HEAD indica qual è la versione a cui fa correntemente riferimento l'area di lavoro

Da locale a remoto (GitHub)

- Un utente registrato e loggato su GitHub
  - Può creare un repository remoto da questa pagina → https://github.com/new
    - Occorre specificare il repository name, ad es: xyz
    - Bottone “create repository” → nel mio caso, viene creato qui: https://github.com/egalli64/xyz
- Condivisione di un nostro repository locale
  - git **remote add** <alias> <URL> → ad es: https://github.com/egalli64/xyz.git
    - In genere si usa *“**origin**”* come alias all’URL del repository remoto
  - git **push -u** <alias> <branch>
    - Upload dei file da locale a remoto, sul branch principale (convenzionalmente “***master***”, in transizione a “***main***”)
    - L’opzione -u (o --set-upstream) va usata solo la prima volta, poi sarà implicito il branch corrente
    - Come password GitHub richiede un Personal Access Token https://github.com/settings/tokens *(scope → repo)*
- Verifica del repository remoto associato via git **remote -v**
  - Se si vogliono informazioni più dettagliate: git **remote show** *origin*

Da remoto (GitHub) a locale

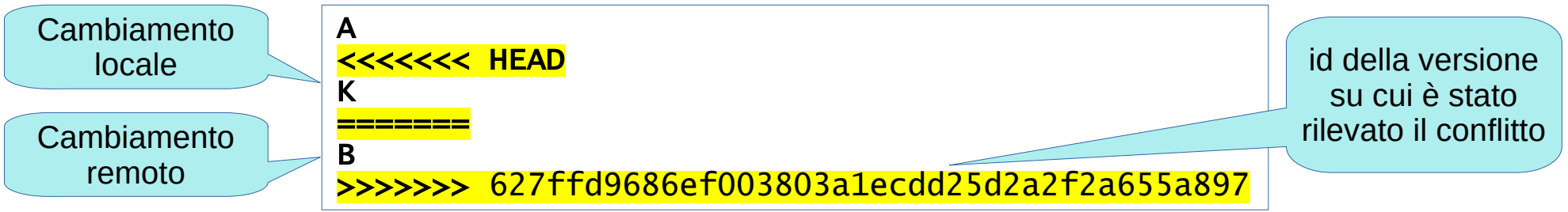
- Solo il proprietario e gli utenti invitati possono modificare un repository
- Ma, se il repository è pubblico, se ne può fare il “fork”
  - Bottone “Fork” sulla destra della pagina del repository
  - Crea una nostra copia indipendente
    - In questo modo abbiamo tutti i permessi sul repository
- Clonazione in locale di un repository
  - GitHub riporta l’URL HTTPS del repository nel tab “Code”, pulsante verde “Code”
  - Il comando git **clone** <URL> crea una subdirectory con il nome del repository
    - Al suo interno c’è la cartella .git con la copia (clone) del repository
    - È pratica comune clonare i repository nella cartella git nella home del nostro utente

Pull e push

- Aggiornamento del branch corrente locale (conviene farlo spesso)
  - git **pull** – abbreviazione di due comandi
    - git fetch – download dei cambiamenti del branch corrente da remoto (origin) a locale
    - git merge – fusione, quando necessario, nei file locali dei cambiamenti rilevati
- Per aggiornare il branch corrente su origin
  - dopo aver eseguito **commit** in locale
  - git **push**
- Ma, se nel frattempo origin è cambiato, git ci blocca!
  - Dobbiamo fare **prima** una pull, verificare che non ci siano conflitti, nel caso risolverli
  - E solo dopo (e se solo se è tutto a posto) eseguire una push

Conflitti su pull

- Il file hello.txt ha una sola riga: “**A**”
- L’utente X aggiunge una riga “**K**” e committa
- L’utente Y fa una pull, aggiunge la riga “**B**”, committa e fa un push
- Ora, il pull di X causa un auto-merging di hello.txt con un conflitto
  - Git chiede aiuto solo se non riesce a decidere cosa fare → conflitto sulla stessa riga
- Va risolto editando il file + add/commit del risultato



File ignorati da Git

- File che ***non*** vogliamo mettere nel repository
  - File di configurazione
  - File generati dal compilatore
  - …
- Nel file di testo semplice "**.gitignore**"
  - Su ogni riga possiamo mettere
    - Nome di un file
      - Se prefissato con uno slash si intende relativo alla home del repository
    - Nome di un folder *(terminato da '/' per leggibilità)*
    - Un pattern

```
example.txt
bin/
*.tmp
```

Esempio di file .gitignore

Checkout e reset

- File modificato, vogliamo tornare alla versione nel repository
  - git **checkout** <filename>
- Se si vuole portare nell'area di lavoro una versione specifica
  - git **checkout** <version> **--** <filename>
    - Se non si specifica la versione, si intende la più recente
- Abbiamo già visto l'uso di reset per togliere un file dall'area di staging
  - git **reset** <filename>
- Possiamo usare reset anche per cambiare la "HEAD" del repository locale
  - git **reset** <version> – i file nell'area di lavoro non sono modificati
  - git **reset --hard** <version> – i file nell'area di lavoro corrispondono alla versione selezionata
- Per vedere il log completo dopo un reset → git **log @{1}**

Branching del repository

- Lista dei branch locali, evidenziando quello corrente: git **branch**
  - Per i branch remoti, opzione **-r**, per la lista completa, locali e remoti, **-a**
  - Il comando git **status** riporta anche l'informazione sul branch corrente
- Creazione un nuovo branch: git **branch** <name> copia del branch principale
  - Il nome segue convenzioni es: feature/abc, fix/xyz, …
  - Il primo push del nuovo branch deve creare un upstream branch: git **push -u** origin <name>
- Scelta del branch corrente: git **checkout** <name>
  - Creazione e selezione di un nuovo branch: git **checkout -b** <name>
- Portare un singolo commit nel branch corrente: git **cherry-pick** <commit-hash>
- Fusione dell'intero branch corrente con quello indicato: git **merge** <name>
  - git **rebase** <name> può aiutare a semplificare il lavoro (ma va fatto con maggiore attenzione)
- Eliminazione di un branch locale: git **branch -d** <name>
  - Remoto: git **push --delete** origin <name>

Altri comandi

- Per eliminare / rinominare un file
  - git **rm** <filename> / git **mv** <filename> <newname>
- Per vedere le differenze nei file
  - Tra un file nell'area di lavoro e nel repository in una data versione: git **diff** <version> <filename>
  - In due diversi branch: git **diff** <branch1> <branch2>
- Per informazioni su chi e quando ha committato ogni riga di un file
  - git **blame** <filename> tra le opzioni, -L <first>,<last>, mostra solo l'intervallo indicato
- Varianti per il logging
  - Versione estesa: git **log** --graph --oneline --all
  - Versione compatta: git **log** --pretty=oneline
  - Su più branch: git **log** <branch1> <branch2>
- Per cambiare la URL di remote
  - git **remote** set-url <remote-name> <URL> il *remote name* di solito è *origin*

Suggerimenti

- A inizio giornata, dopo una pausa, prima di un push …: pull
- Meglio tanti piccoli commit *(con messaggi significativi)* e push, invece di uno riassuntivo
- Creare un branch specifico per ogni feature / bug fix
  - Dal branch principale, accertarsi di avere la versione più recente del codice: git pull
  - Creare un nuovo branch, es.: git checkout -b feature/42
  - Commit-pull-push sul feature branch *(con pull anche dal branch principale)*
    - Ricordarsi che il primo push deve specificare il branch remoto es: git push -u origin feature/42
  - Quando la feature è implementata: pull request
    - Si richiede l'approvazione e poi si procede al merge nel branch principale
    - In GitHub, tasto "Compare & pull request"
- Tenere la versione stabile del codice su un suo branch specifico (es: *stable*)
  - Il branch principale è quello di quella che sarà la nuova versione stabile