Introduzione ai Distributed Version Control Systems
Visualizzazione della storia dello sviluppo con git

Programmazione ad Oggetti – Lab04

Docenti: Danilo **Pianini**, Roberto **Casadei**
Tutor: Simone **Costanzi**, Luca **Tremamunno**

C.D.S. Ingegneria e Scienze Informatiche
Alma Mater Studiorum—Università di Bologna, Campus di Cesena

13 ottobre 2021

Outline

Outline

Cosa sono

I DVCS sono software che consentono di:

- Mantenere traccia dei cambiamenti fatti ad un progetto, consentendo di andare "avanti e indietro" nel tempo.
- Consentire e promuovere il lavoro di gruppo, anche in parallelo (lo vedremo nel prossimo lab)

L'esigenza di poter tornare a salvataggi precedenti è sempre stata avvertita dagli sviluppatori (e non solo). L'operazione di salvare più stati del proprio lavoro è detta *versioning*, un software che semplifica il versioning è un *version control system* (o *versioning system*).

Pillole di storia

Sistemi di versioning:

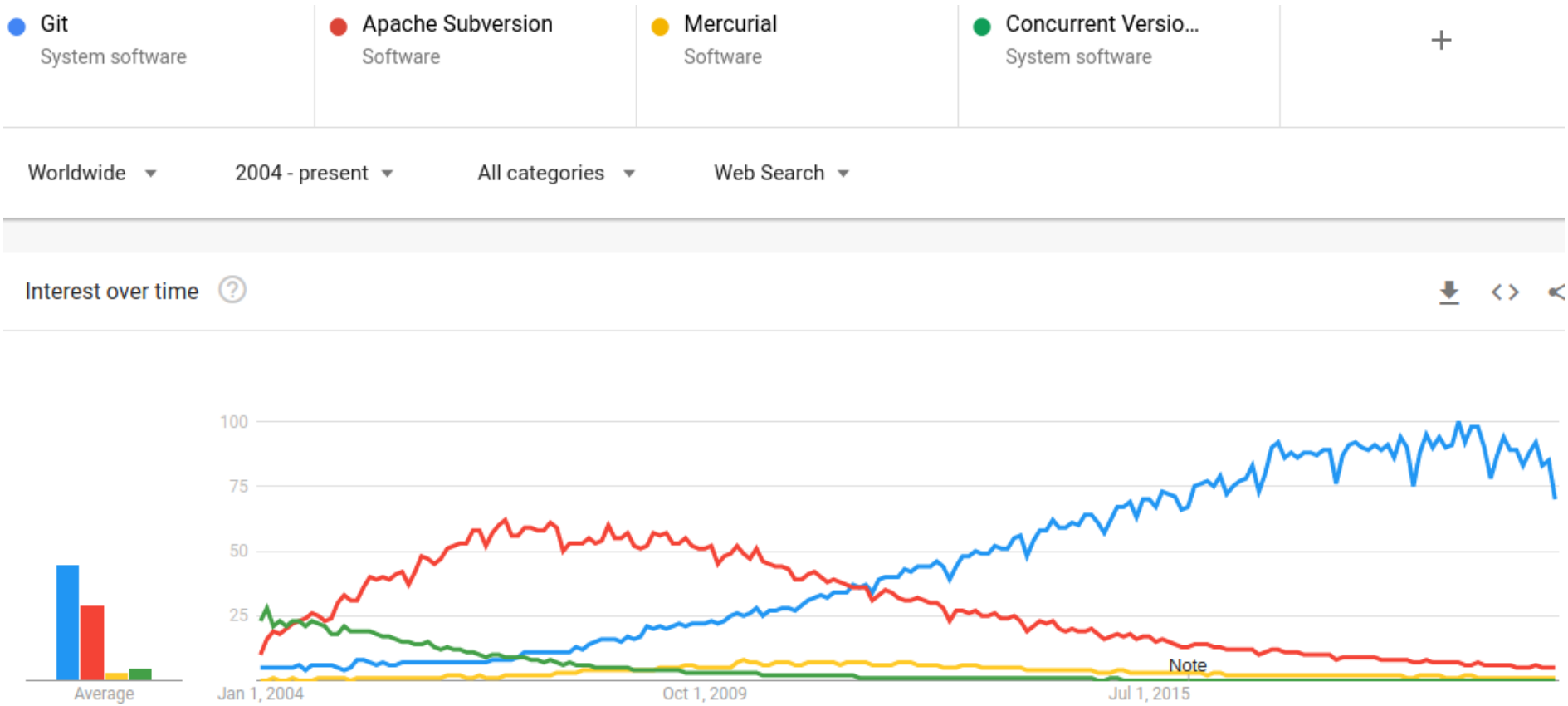
- "Fai da te" — è il sistema che la maggior parte di voi ha usato finora: si fa una copia di tutti i file in una cartella (magari numerata). Costa molto in spazio ed in tempo, rende difficile lo scambio di file e il lavoro parallelo.
- *CVS* — Fu il primo sistema di versioning. Studiato per salvare automaticamente i punti di salvataggio di file di testo. È difficile usarlo per file binari, facilita lo scambio di file rispetto ad inviarsi cartelle.
- *SVN* — Evoluzione di CVS. Molto più veloce e con supporto nativo a file binari. Il lavoro in parallelo è possibile a patto di adottare un flusso di lavoro di squadra molto controllato.
- *Git* e *Mercurial* — Sviluppati parallelamente per superare le limitazioni di SVN, sono nati praticamente identici. Più veloci di SVN e pensati per supportare il lavoro massivamente parallelo di team sparsi per il mondo.

Diffusione

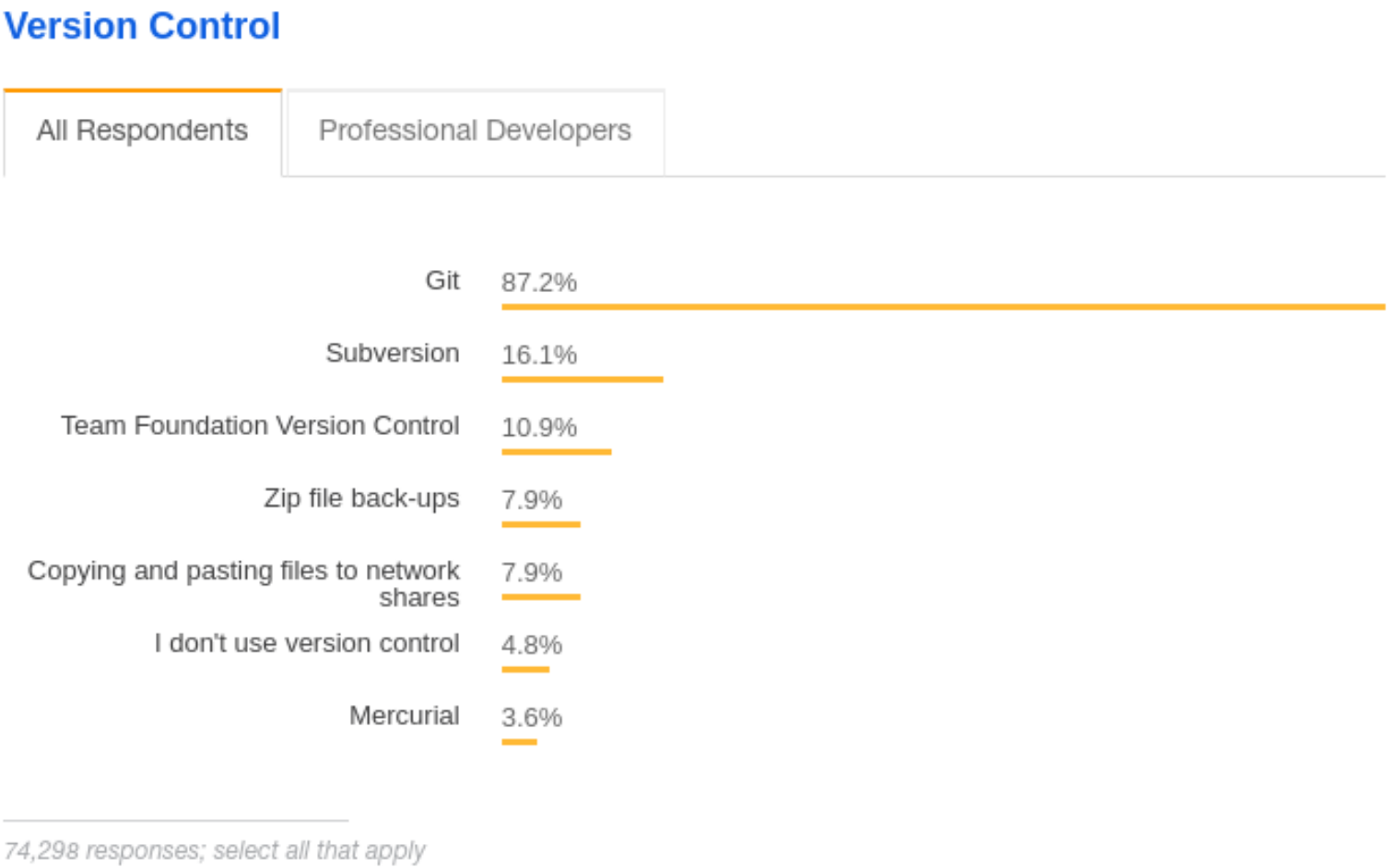
Sono usati per tutti i moderni processi di sviluppo software. Un po' di esempi:

- Android (git)
- Drupal (git)
- Facebook (Mercurial)
- GCC (git)
- Go (git)
- Java JDK (Mercurial)
- Libreoffice (git)
- Linux kernel (git)
- Python (git)
- VLC Video Player (git)
- Wine (git)
- le slides e il laboratorio di OOP! (git)

Diffusione dei sistemi di controllo di versione I

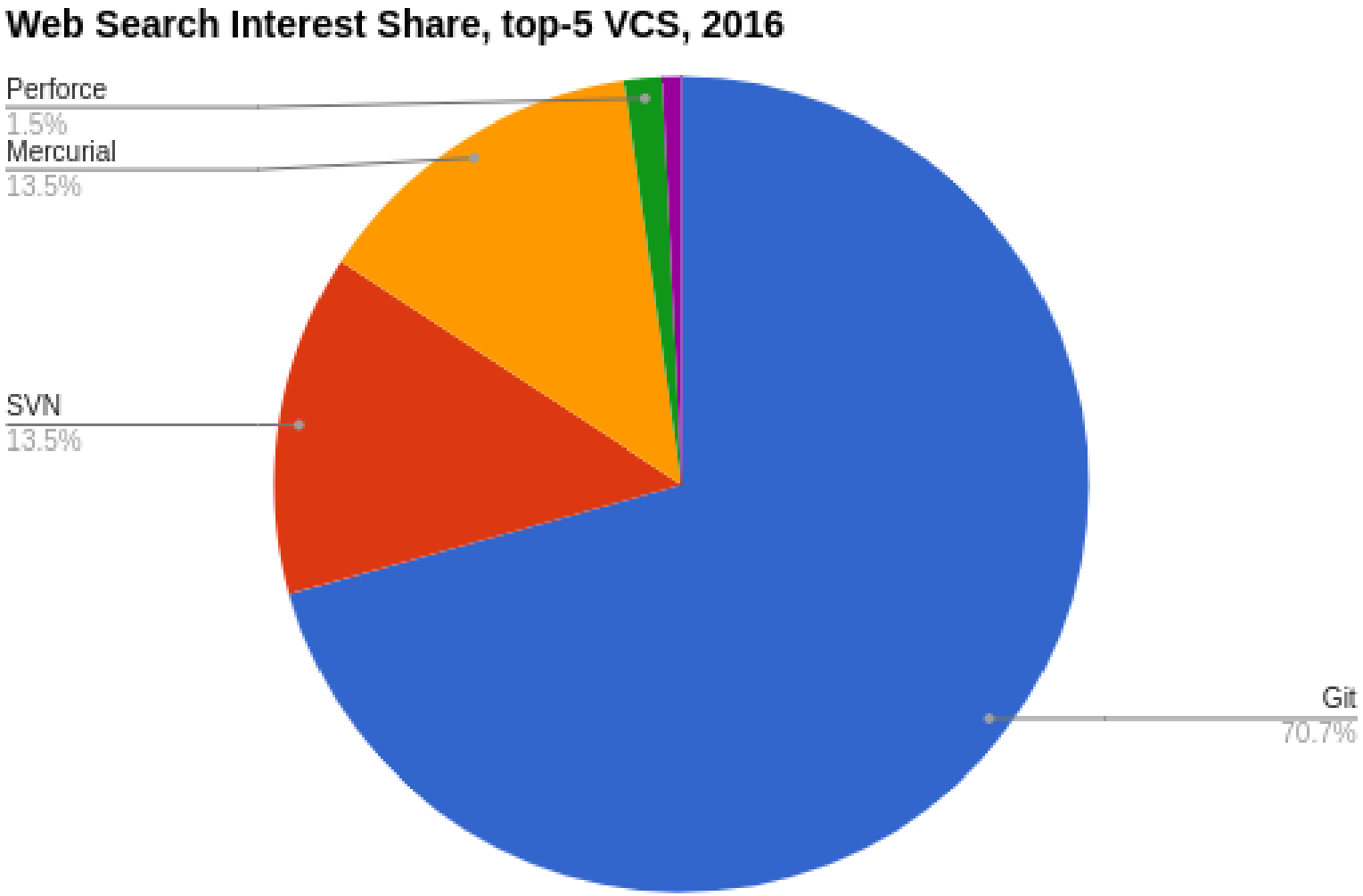


Diffusione dei sistemi di controllo di versione II

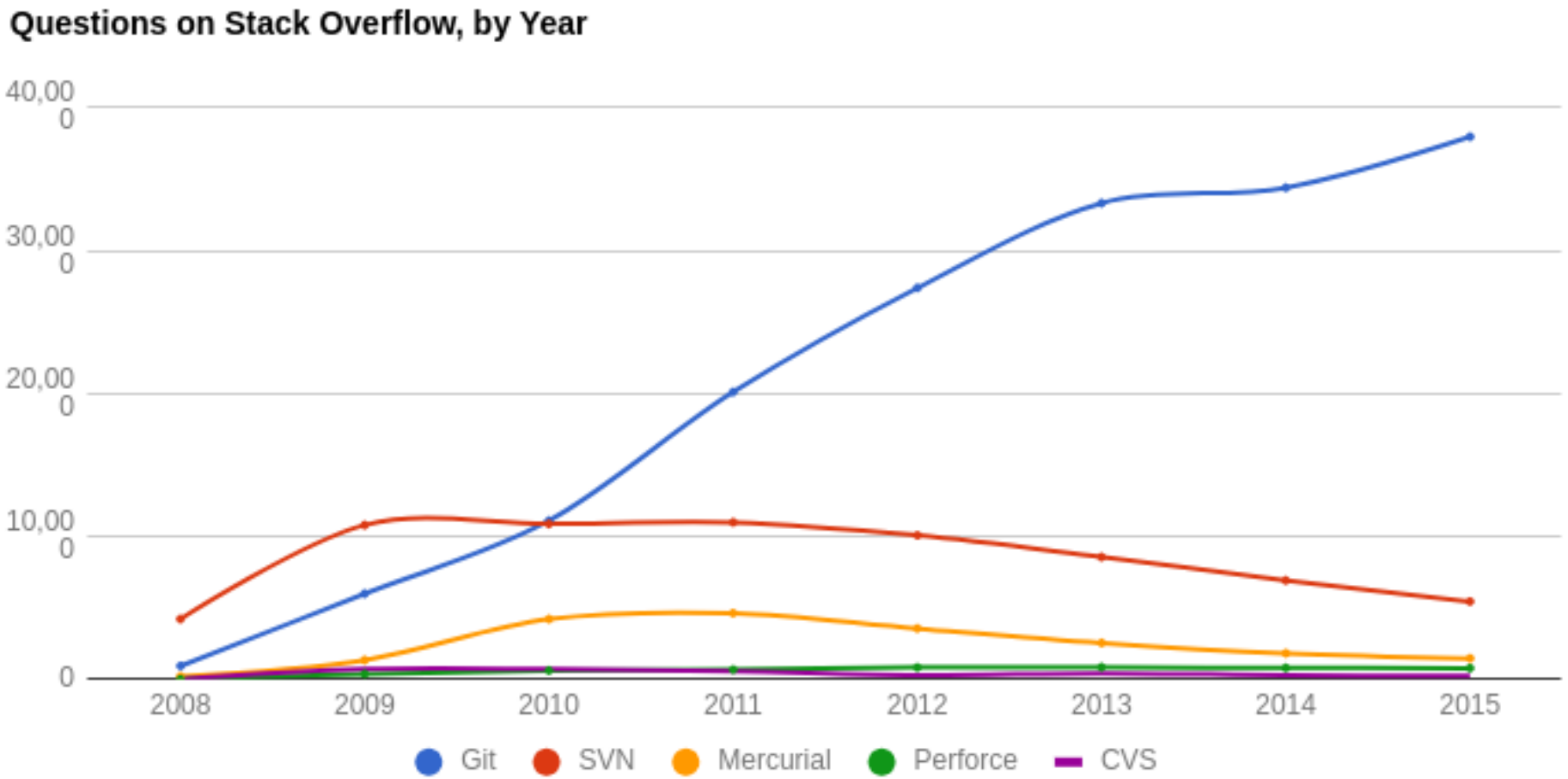


Git is the dominant choice for version control for developers today, with almost 90% of developers checking in their code via Git.

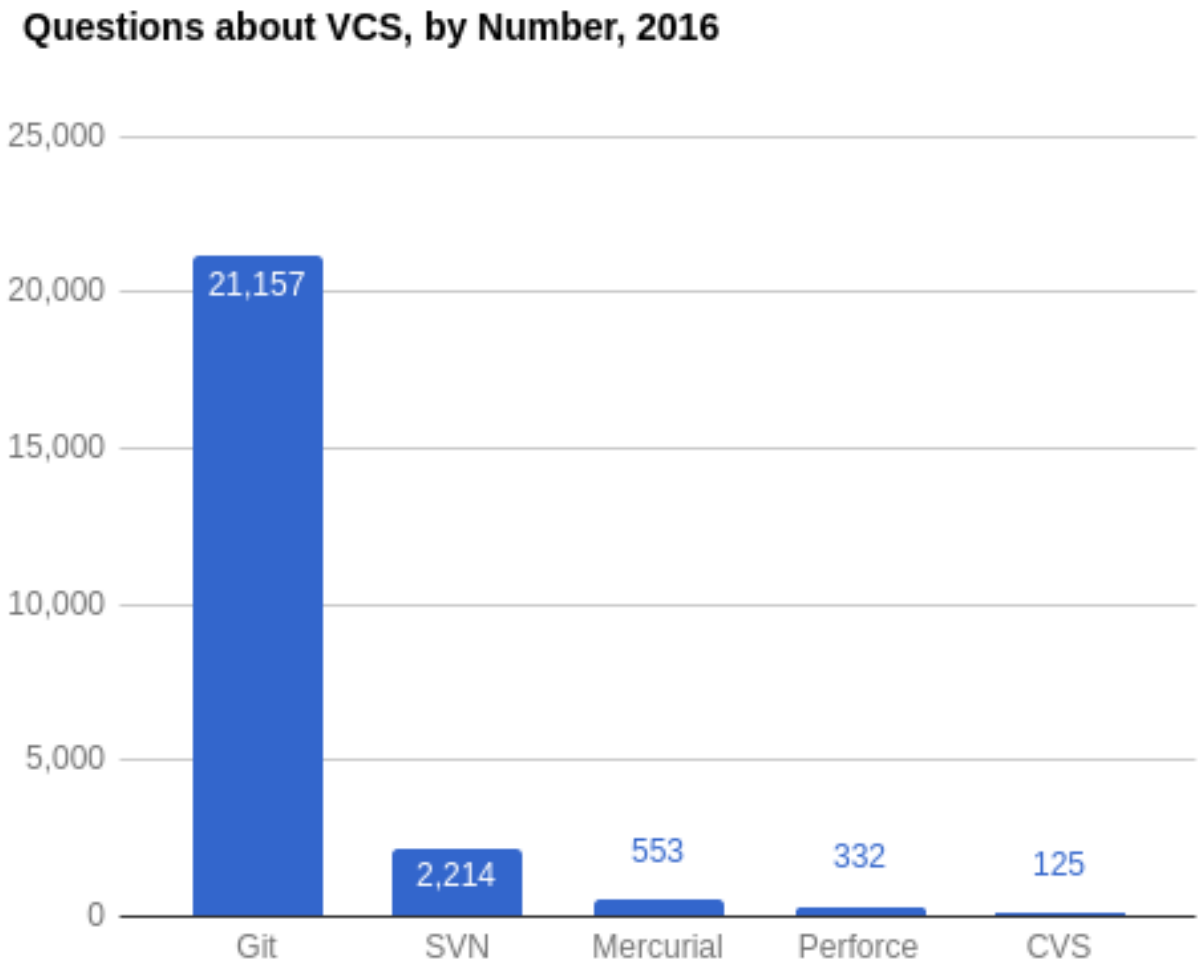
Diffusione dei sistemi di controllo di versione III



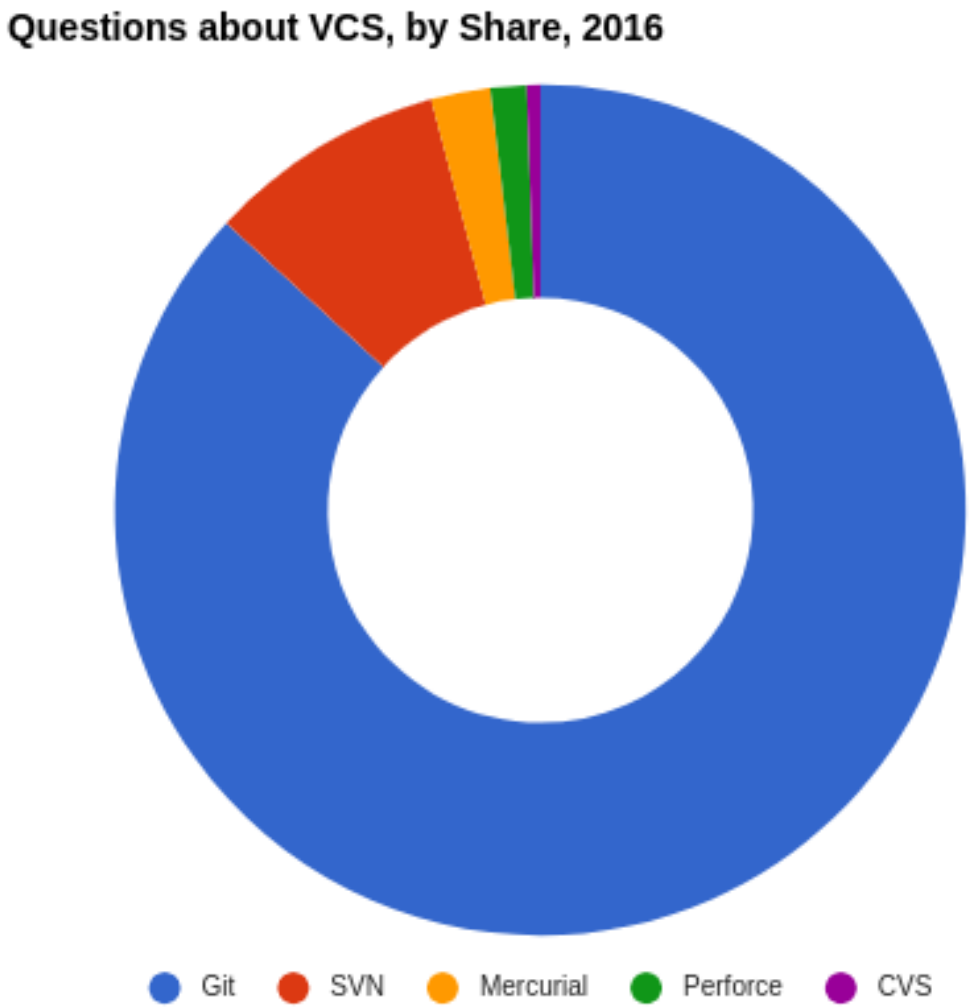
Diffusione dei sistemi di controllo di versione IV



Diffusione dei sistemi di controllo di versione V



Diffusione dei sistemi di controllo di versione VI



Bits of history

- In April 2005, BitKeeper, the SCM Linux was developed with, withdrawn the free (as in beer) use
- No other SCM met the requirements of Torvalds
  - Performance was the *real* issue with such a code base
- Torvalds decided to write his own
- The project was successful, and Torvalds appointed maintenance to Hamano

Why the name

> *I'm an egotistical bastard, and I name all my projects after myself. First 'Linux', now 'git'.* [a]
>
> — Linus Torvalds

[a] From the project Wiki. "git" is slang for "pig headed, think they are always correct, argumentative"

Outline

Concetti basilari e terminologia I

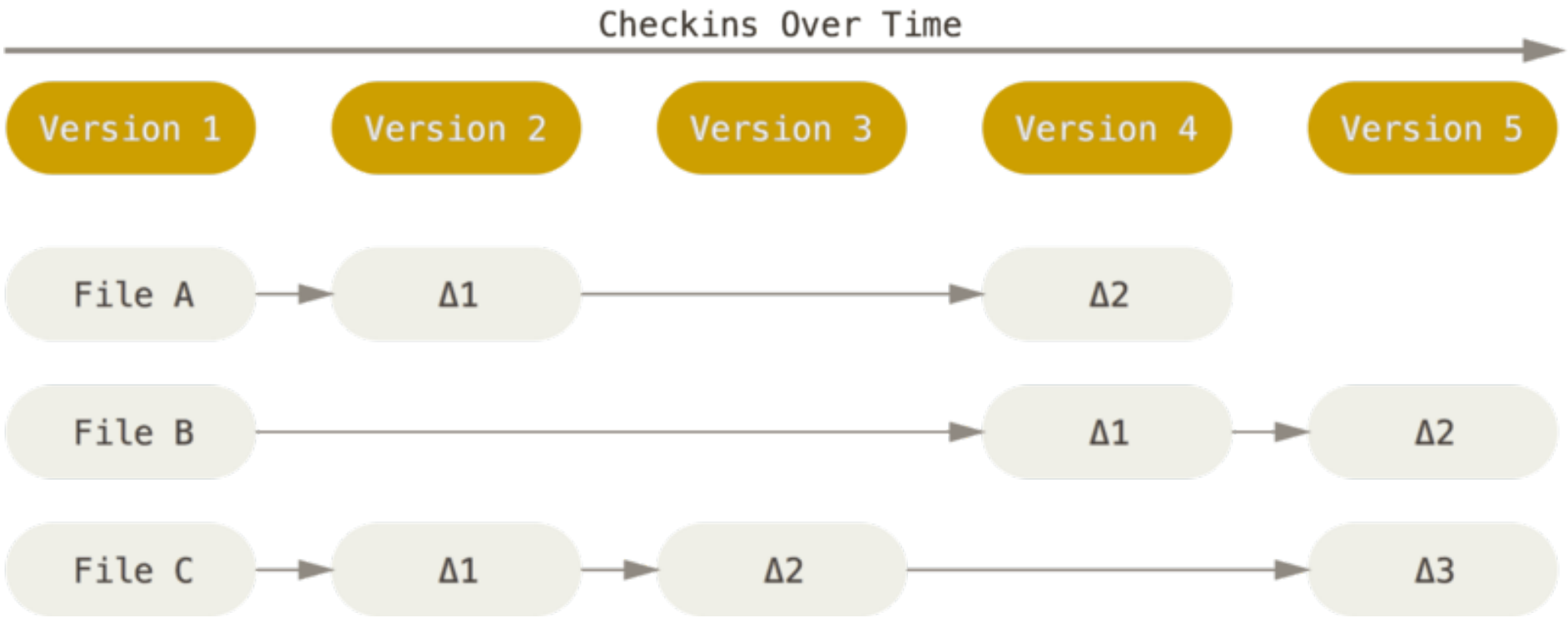
Repository

Il repository è l'insieme dei file che vengono tracciati dal DVCS assieme ai metadati, ossia alle informazioni che servono a ricostruire qualunque stato precedente.

Tracciamento delle differenze

Abilità di registrare le differenze fra diverse versioni di uno o più file. Invece di salvare l'intero stato (tutto il contenuto di un file), vengono salvate solo le informazioni necessarie a ricostruire il file a partire dal salvataggio precedente.

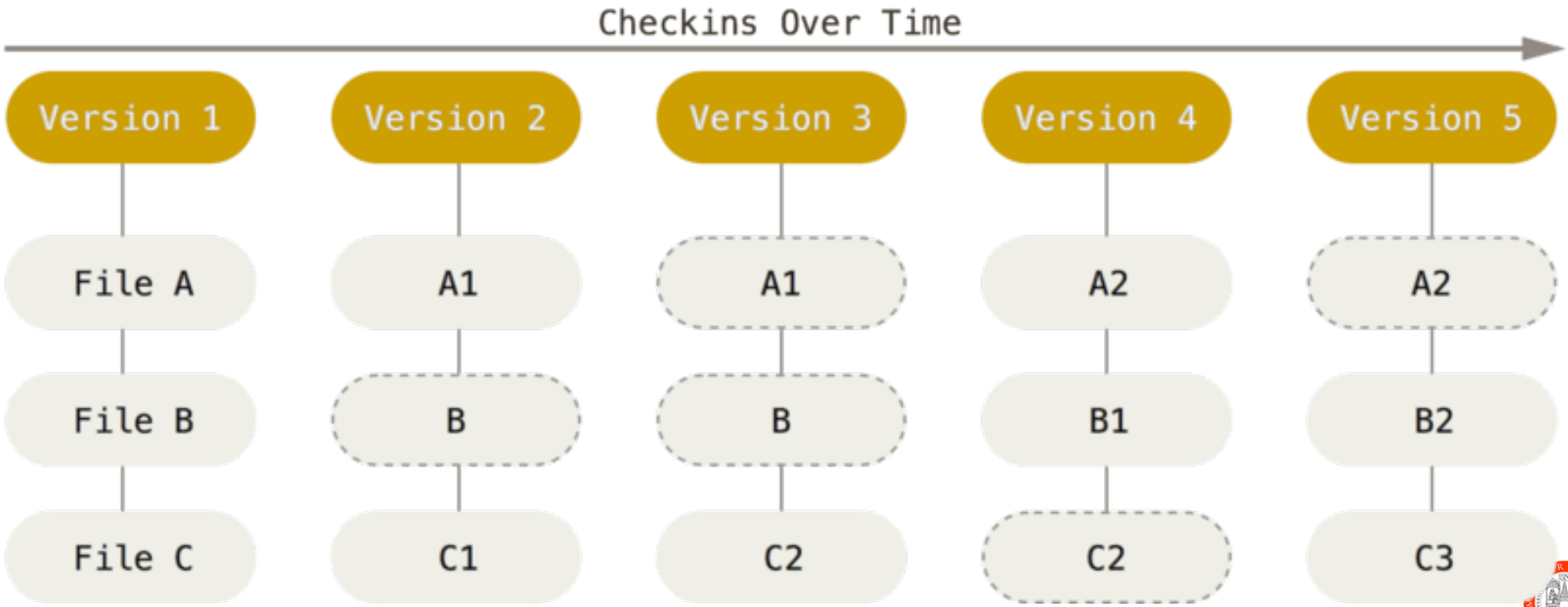
Concetti basilari e terminologia II



Concetti basilari e terminologia III

Snapshotting

Capacità di salvare lo stato di tutti i file che fanno parte del progetto in una sola versione, creando una fotografia del sistema.



Concetti basilari e terminologia IV

Nota: un buon version control system è in grado di esporre sia meccanismi di snapshotting (ad esempio, riporta il repository a tre versioni fa) e meccanismi basati sulle differenze (ad esempio, dimmi cos'è cambiato in questo file dall'anno scorso).

Concetti basilari e terminologia V

Commit

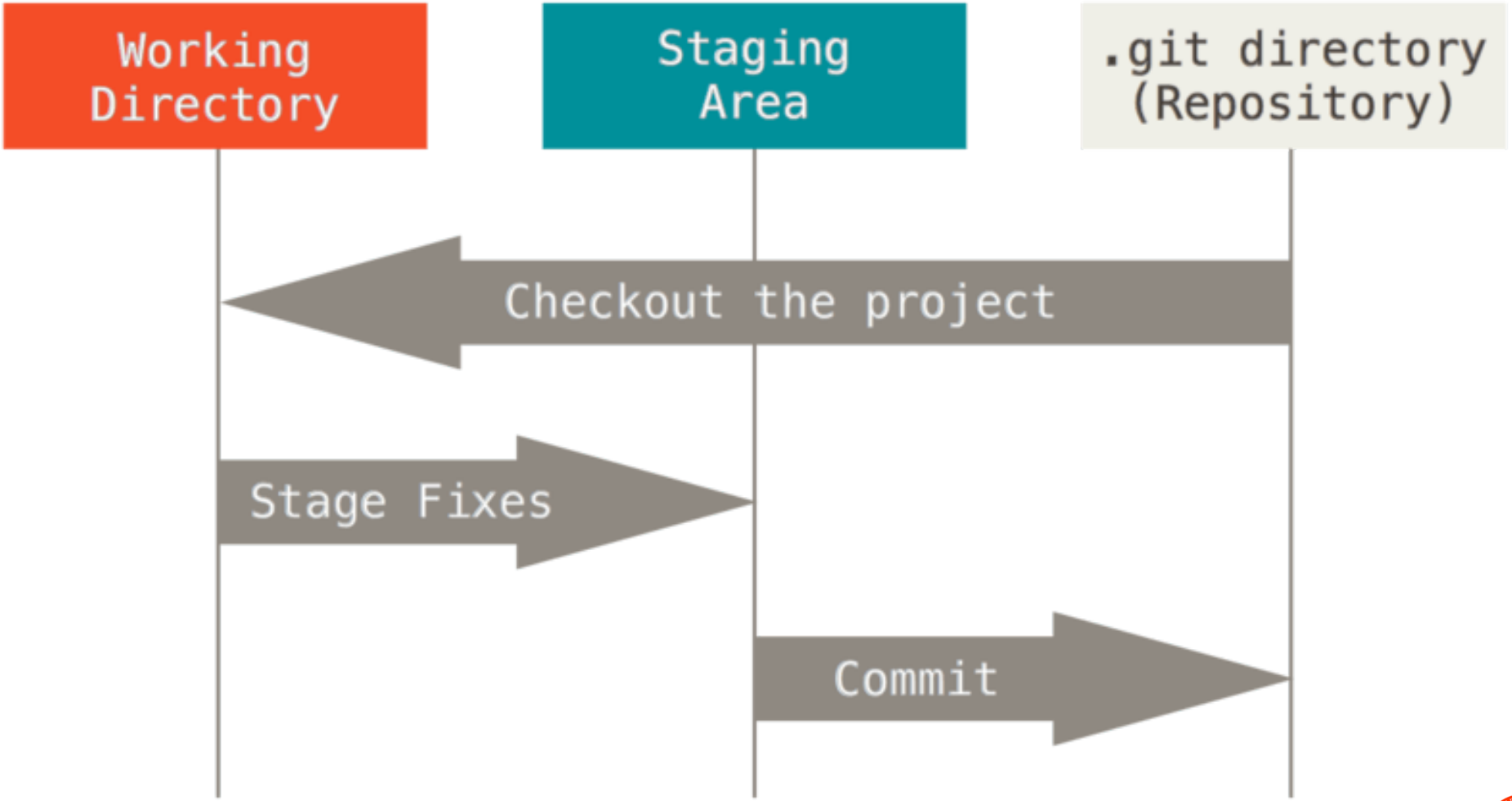
Salvataggio dello stato del repository

Staging area

Insieme delle modifiche accodate per esser salvate al prossimo commit. Il processo di salvataggio si articola infatti in due fasi:

1. **Staging** — Selezione di quali file modificati, aggiunti o rimossi salvare al prossimo commit
2. **Commit** — Effettivo salvataggio delle modifiche presenti nella staging area

Concetti basilari e terminologia VI



Concetti basilari e terminologia VII

Navigazione della storia

Possibilità di tornare ad un qualunque commit (salvataggio) precedente o successivo

Outline

Configurazione globale I

In generale

Come ogni prodotto software, i DVCS necessitano di alcune operazioni di configurazione preliminari. In particolare, richiedono di impostare un nome utente ed una email (in modo da poter capire chi ha apportato modifiche).

Configurazione globale II

In Git

Si specifica un nome utente di default utilizzando

- `git config --global user.name "YOUR NAME"`
  - **OVVIAMENTE** al posto di `YOUR NAME` dovrete inserire il vostro nome.
  - **Configurate il vostro vero nome, non un nickname!**
    - In fase di correzione del progetto usiamo (anche) git per capire chi ha fatto cosa, e vedere roba firmata da `superkikk4z` e `TheDarkDestructor` non è che sia proprio bello (oltre al fatto che rischiamo di sbagliare l'assegnamento nome-persona)
    - Se siete in lab importa poco, ma meglio far le cose bene fin da subito
    - Il nickname fantasioso potrete usarlo su GitHub prossimamente, lo usiamo anche noi (`DanySK`, `metaphori`...)

L'email di default is setta con

- `git config --global user.email "your.email@provider"`

Configurazione globale III

Esercizio

- Si apra un terminale
- Si settino username ed email di default usando nome e cognome ed email istituzionale

Outline

Outline

Inizializzazione di un repository I

In generale

È necessario esplicitare che, da un certo punto del file system, si desidera utilizzare il DVCS per tener traccia dei cambiamenti dei file contenuti da quel punto del file system.

- Il DVCS dovrà salvare dei metadati che consentano di ricostruire gli stati precedenti del sistema
- Bisognerà prestare attenzione a quale cartella si inizializza: inizializzare il punto sbagliato implicherà repository enormi ed ingestibili!

Inizializzazione di un repository II

In Git

- `git init`

Marca la cartella corrente come repository Git. Crea una sottocartella nascosta `.git` nella quale saranno salvati i metadati.
È possibile settare username ed email personalizzati per ogni repository, usando i comandi visti prima privati dell'argomento `--global`, e.g.:

```
git config user.email "your.second@email"
```

Errori comuni

- Bisogna posizionarsi dentro la cartella che ospiterà il nostro repository **prima** di dare il comando `git init`
- Dare il comando dentro la home folder (dove dovrebbe aprirsi il terminale di default) marcherà tutta la home folder come repository Git

Inizializzazione di un repository III

Esercizio

- Si crei una cartella di nome `dvcstest` (comando `mkdir`)
- Si entri nella cartella col terminale
- Si inizializzi un repository git
- Si setti l'email utente **per il repository** al vostro indirizzo personale (quindi non quello `@studio.unibo.it`)
  - Nota: chi non volesse usare l'indirizzo personale per ragioni di privacy, usi una email fittizia.

Ispezionare lo stato del repository I

In generale

È necessario sapere quale sia lo stato del repository e della staging area, per conoscere:

- Quali file sono stati modificati
- Quali file sono stati aggiunti allo stage

In Git

È possibile ispezionare lo stato usando:

- `git status`

Ispezionare lo stato del repository II

Suggerimenti

- Bisogna controllare spesso lo stato del repository
- Idealmente, prima di ogni operazione che potrebbe modificare lo stato del repository

Esercizio

- Si ispezioni lo stato corrente del repository

Output atteso

```
On branch master

Initial commit

nothing to commit (create/copy files and use "git add" to track)
```

Ispezionare lo stato del repository III

Spiegazione dell'output

- Prima linea: il branch su cui ci si trova. Non ne è stato creato nessuno esplicitamente, quindi Git assume che di default si lavori su un branch di nome `master`
- Seconda linea: Git ci segnala che non abbiamo ancora effettuato alcun commit
- Terza linea: Git mostra lo stato della staging area. In questo momento non c'è nulla di cui far commit (difatti, non ci sono file nel nostro repository)

Aggiungere files alla staging area I

In generale

È necessario segnalare esplicitamente quali file dovranno essere inclusi nel prossimo salvataggio.

- molti dei file potrebbero essere rigenerabili a partire da altri
- Il tracking differenziale è efficiente con file testuali...
- ...ma inefficiente con i binari!
- tracciare file generabili è uno spreco di risorse!
- I file selezionati saranno aggiunti alla "staging area"

Aggiungere files alla staging area II

In un progetto Java

Vanno tracciati:

- I sorgenti
- Le risorse (icone, file di configurazione...)
- Librerie copiate nel progetti
- Eventuali file esterni, ad esempio un README.md, un file per la licenza, il file .project di Eclipse per facilitare l'import...

Non vanno tracciati:

- I binari (rigenerabile dai sorgenti)
- La documentazione rigenerabile dai sorgenti
- Archivi con la vostra applicazione (rigenerabili)

Aggiungere files alla staging area III

In Git

Il sottocomando add aggiunge delle modifiche alla staging area

- `git add PATH_TO_FILE`
  - Aggiunge il file indicato alla staging area. Il file deve essere all'interno del repository.
  - Il file deve essere cambiato rispetto allo stato precedente (perché nuovo, modificato, o cancellato)
  - Il file può essere un file che esisteva ma è stato cancellato!
  - In questo caso, viene registrata nella staging area la cancellazione
- `git add PATH_TO_FILE_1 PATH_TO_FILE_2 PATH_TO_FILE_N`
  - Aggiunge tutti i file indicati alla staging area.

Aggiungere files alla staging area IV

Esercizio

- All'interno della cartella dove abbiamo inizializzato il repository git vuoto, si creino due cartelle: `src` e `bin`
- Si crei dentro src un file `HelloWorld.java` (ad esempio con Visual Studio Code), contenente un semplice main con una stampa
- Si visualizzi lo stato del repository con `git status`
  - Si noti che ci sono nuove informazioni!
  - Git ci informa che ci sono dei file non tracciati dentro la cartella `src/`
- Si utilizzi in modo appropriato il comando `git add` per aggiungere al tracking il file `HelloWorld.java`
- Se il comando viene eseguito correttamente, non viene dato alcun output all'utente
- Si visualizzi lo stato del repository

Aggiungere files alla staging area V

Output atteso

```
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   src/HelloWorld.java
```

Spiegazione dell'output

- La parte relativa alla staging area è cambiata: git ci segnala che le modifiche ad `src/HelloWorld.java` saranno salvate al prossimo commit

Creare punti di salvataggio I

In generale

L'operazione fondamentale che vogliamo eseguire è quella di creare un punto di salvataggio, che registri i nostri progressi e al quale potremo sempre tornare.

Assieme al salvataggio, vogliamo registrare alcuni metadati:

- Nome dell'autore del commit
- Informazioni per identificare e contattare l'autore (email)
- Un **messaggio** che riassuma quali modifiche sono state fatte in quel commit
- Data e ora del commit (acquisite automaticamente)
- Un identificativo univoco (hash) (generato automaticamente)

Il commit non salverà lo stato del progetto, ma il set di modifiche necessarie per portare i file dallo stato precedente a quello del nuovo commit.

Creare punti di salvataggio II

Buone pratiche di commit

È molto importante specificare un messaggio di commit sensato.

- Deve essere un breve riassunto di quanto è stato fatto dal salvataggio precedente
- Chi vede la storia del progetto deve capire immediatamente quali siano le differenze che il commit introduce
- È buona norma scrivere in inglese usando il simple present

È molto importante fare commit piccoli e frequenti

- Idealmente, uno ad ogni modifica di cui sia possibile fornire una descrizione organica nel commit message
- Non importa se le modifiche sono minimali codice un solo file aggiunta di diversi file sorgente

Creare punti di salvataggio III

Cattive pratiche da evitare

- Messaggi non chiari, generici e/o troppo brevi, ad esempio:
  - Fix bug
  - Fix project
  - Add files
  - Commit
- Commit giganteschi con molte modifiche, magari non strettamente correlate fra loro

Creare punti di salvataggio IV

In Git

Il sottocomando `commit` crea il salvataggio

- `git commit -m "A message"`
  - Esegue un commit di tutte le modifiche aggiunte alla staging area
  - Utilizza `A message` come commit message
- `git commit FILE1 FILE2 FILEN -m "A message"`
  - Esegue un commit salvando tutte le modifiche ai file elencati
  - Utilizza `A message` come commit message

Creare punti di salvataggio V

`commit` senza specificare il messaggio

- Non è possibile non specificare un messaggio.
- Se l'opzione `-m` viene omessa, viene aperto un editor per l'inserimento del messaggio
- Se il messaggio viene lasciato vuoto, il commit viene rigettato
- `git commit`
  - Esegue un commit salvando tutte le modifiche nella staging area
  - Apre l'editor di testo di sistema per l'inserimento del commit message
- `git commit PATH_TO_FILE_1 PATH_TO_FILE_2 PATH_TO_FILE_N`
  - Esegue un commit salvando tutte le modifiche ai file elencati
  - Apre l'editor di testo di sistema per l'inserimento del commit message
  - Equivalente ad eseguire `git add FILES` seguito da `git commit` a partire da una staging area vuota

Preferite sempre commit con opzione `-m`, a meno che non abbiate già confidenza con l'editor di sistema

Creare punti di salvataggio VI

Configurare l'editor di testo

Sistemi diversi hanno editor diversi

- Di default MacOS X utilizza `vim`
- In Linux dipende dalla distribuzione, solitamente è uno fra `vim`, `nano`, o `emacs`
- In Windows *dovrebbe* essere `Notepad.exe` (in laboratorio è `vim`)

Non sempre l'editor di default è quello che preferite: è bene configurarlo

- `git config --global core.editor "editorcommand"`
  - Setta `editorcommand` come comando da invocare per aprire l'editor
  - Va da sé che il comando debba essere disponibile sul vostro sistema...

Vi consiglio di:

- Se conoscete già un editor a command line, usate quello
- Se non lo avete, in ambiente Linux o Mac, di usare `nano`

Creare punti di salvataggio VII

Esercizio

- Se si sta usando Linux o MacOS X, si configuri adeguatamente l'editor di testo
- Si verifichi lo stato del repository
- Si effettui il commit delle modifiche presenti nella staging area
  - Inserendo un commit message SENSATO!
- Si visualizzi lo stato del repository

Output atteso dopo il commit

```
[master (root-commit) 19aa252] Create HelloWorld
 1 file changed, 5 insertions(+)
 create mode 100644 src/HelloWorld.java
```

Creare punti di salvataggio VIII

Spiegazione dell'output

- Ci troviamo nel branch master
- Questo è il primo commit (radice)
- L'hash del nostro commit (una parte, in realtà) è `19aa252`
  - Ovviamente il vostro potrebbe differire
- Il messaggio inserito è "Create HelloWorld"
  - Ovviamente questo è il mio, il vostro potrebbe essere diverso, dipende da che messaggio avete inserito
- È stato modificato un file, in totale sono state inserite 5 righe di codice
- È stato creato un nuovo file `src/HelloWorld.java`
  - Si noti che sono stati tracciati i permessi unix (644 in ottale)

Rimuovere files dalla staging area I

In generale

Vogliamo poter togliere dalla staging area dei file che abbiamo aggiunto

- Ad esempio perché li abbiamo aggiunti a seguito dell'uso di un comando con la wildcard
- Oppure perché abbiamo deciso di salvare le modifiche in più commit

Rimuovere files dalla staging area II

In Git

Il sottocomando `reset` rimuove delle modifiche *dalla staging area* (non dai files, a meno di non specificare apposite opzioni)

- `git reset PATH_TO_FILE`
  - Rimuove dalla staging area le modifiche fatte al file indicato (non dal tracking!)
  - Il file potrebbe anche non esistere, ad esempio se abbiamo cancellato un file e aggiunto la modifica alla staging area.
- `git reset PATH_TO_FILE_1 PATH_TO_FILE_2 PATH_TO_FILE_N`
  - Rimuove dalla staging area le modifiche fatte a tutti i file indicati.

Outline

Gestire caratteri newline I

Gestione dei caratteri di newline

- Git prova ad essere "smart" nella configurazione dei caratteri che rappresentano una nuova linea di testo, che differiscono per piattaforma
  - Come spesso capita, nel tentativo di essere smart si fallisce clamorosamente.
- È meglio configurare Git per gestire correttamente i file di linea, specie se il team lavora con OS diversi!
  - Evitando di fare mega-commit solo perché cambiano i caratteri di fine linea

Gestire caratteri newline II

Gestione dei caratteri di newline

- Configuriamo Git per usare il fine linea Unix, eccetto che sugli script Windows-specifici
- Ci sono vari modi, il più portabile è la creazione di un file `.gitattributes` nella radice del repository
  - È possibile specificare quali tipi di file sono testuali, e quali convenzioni usare
  - È bene che il file `.gitattributes` venga aggiunto al tracker!
  - Nota: il file si chiama **esattamente** `.gitattributes`, **non** `ALTRO.gitattributes`, `.gitattributes.txt`, `gitattributes.gitattributes`, o altre stravaganti forme.

Gestire caratteri newline III

Contenuto del file .gitattributes

```
# Very simple .gitattributes working well in most projects
* text=auto eol=lf
*.[cC][mM][dD] text eol=crlf
*.[bB][aA][tT] text eol=crlf
*.[pP][sS]1 text eol=crlf
```

Gestire caratteri newline IV

Esercizio

- Si crei il file .gitattributes
  - Può essere scaricato da `https://bit.ly/oop-gitattributes`
  - Per chi ha una shell Unix:
    - `curl -L -o ".gitattributes" "https://bit.ly/oop-gitattributes"`
- Si osservi lo stato del repository
- Si aggiunga `.gitattributes` alla staging area
- Si osservi lo stato del repository
- Si effettui il commit
- Si osservi lo stato del repository

Ignorare file indesiderati I

In generale

In molti casi, vorremmo poter dire al DVCS di ignorare alcuni file o cartelle, che sappiamo essere rigenerabili o che riteniamo non utili

- I file compilati
- Eventuali archivi o documentazione *rigenerabile*

In Git

È possibile creare, nella radice del repository, un file `.gitignore`

- I file elencati dentro `.gitignore` saranno invisibili a Git
- È bene che il file `.gitignore` venga aggiunto al tracker!
- Nota: il file si chiama **esattamente** `.gitignore`, **non** `ALTRO.gitignore`, `.gitignore.txt`, `gitignore.gitignore`, o altre stravaganti forme.

Ignorare file indesiderati II

Sintassi di .gitignore

```
bin/
doc/
*.log
*.pdf
!myImportantFile.pdf
```

Stiamo dicendo che git deve:

- Ignorare la cartella `bin`, e tutto il suo contenuto
- Ignorare la cartella `doc`, e tutto il suo contenuto
- Ignorare tutti i file di con estensione `.log`
- Ignorare tutti i file di con estensione `.pdf`
- Non ignorare il file `myImportantFile.pdf`
  - È possibile usare `!` per creare eccezioni ad una regola
  - È molto più comodo che elencare tutti i file da escludere uno per uno!

Ignorare file indesiderati III

Note sulla creazione di file il cui nome inizia per .

- Windows ha l'abitudine di aggiungere autonomamente l'estensione ai file che vengono creati...
- ...per poi nasconderla
- Verificate **sempre con il terminale** che il file sia esattamente quello che vi aspettate, ossia `.gitignore`
- Se il file viene chiamato `.gitignore.txt`, o in qualunque modo diverso da `.gitignore`, non sarà considerato un ignore file valido da Git!

Il problema non si pone su MacOS X e Linux.

Ignorare file indesiderati IV

Creare il file da terminale

- È conveniente usare direttamente il terminale per creare il file `.gitignore`
- `echo > .gitignore`
  - Crea un file di nome `.gitignore` contenente solo una newline
  - Funziona su tutti i sistemi!
  - Evita di creare file manualmente
- `echo WHAT_TO_IGNORE >> .gitignore`
  - Aggiunge una linea con scritto `WHAT_TO_IGNORE` in coda al file `.gitignore`
  - Funziona su tutti i sistemi!
  - Consente di popolare il file `.gitignore` senza dover usare editor esterni
  - e.g. `echo bin/ >> .gitignore` — aggiunge alla lista degli ignore la cartella `bin` e tutto il suo contenuto

Ignorare file indesiderati V

Esercizio

- Si compili dentro bin il file HelloWorld.java che avete creato
  - Spero che vi ricordiate come si compila un file con `javac`
  - Se il file non compilasse, sistematelo, quindi aggiungetelo alla staging area ed eseguite un commit
  - Già che ci siamo, eseguite e verificate che funzioni
- Si osservi lo stato del repository
- Si crei un file `.gitignore` che ignori la cartella bin e tutto il suo contenuto
- Si osservi lo stato del repository
- Si aggiunga `.gitignore` alla staging area
- Si osservi lo stato del repository
- Si effettui il commit
- Si osservi lo stato del repository

Ignorare file indesiderati VI

Output atteso: primo git status

```
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        bin/

nothing added to commit but untracked files present (use "git add" to track)
```

Output atteso: secondo git status

```
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .gitignore

nothing added to commit but untracked files present (use "git add" to track)
```

Ignorare file indesiderati VII

Output atteso: terzo `git status`

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   .gitignore
```

Output atteso: commit

```
[master 3ae8422] Create .gitignore
 1 file changed, 1 insertion(+)
 create mode 100644 .gitignore
```

Output atteso: quarto `git status`

```
On branch master
nothing to commit, working tree clean
```

Ignorare file indesiderati VIII

Spiegazione dell'output

- Si noti come `bin/` sparisca dai file presenti nell'area di lavoro una volta che il file `.gitignore` è stato creato

Rimozione e rinominazione I

In generale

Vogliamo poter cancellare e rinominare files, e tracciare il fatto che lo abbiamo fatto

In Git

Git è in grado di capire da solo quando qualcosa è stato modificato o eliminato (in questo è molto più semplice di Mercurial)

- Git tratta tutte le modifiche allo stesso modo
- A fronte della cancellazione di un file, basta aggiungerlo alla staging area perché la modifica venga registrata al successivo commit
- A fronte di una rinominazione, si aggiungono sia il file col vecchio nome che quello col nuovo
  - Diversamente, verrà trattata come una rimozione o un'aggiunta, a seconda di quale delle due modifiche aggiungete all'area di staging.

Rimozione e rinominazione II

Esercizio

Premessa: si osservi lo stato del repository con `git status` **prima e dopo ogni operazione**, assicurandosi di capire appieno l'output fornito da Git

- Si crei un file `junk.txt`, con un contenuto casuale
- Si aggiunga `junk.txt` alla staging area
- Si effettui il commit
- Si rinomini `junk.txt` in `trash.txt`
- Si aggiungano tutte le modifiche alla staging area
- Si effettui il commit
- Si elimini `trash.txt`
- Si aggiunga `trash.txt` alla staging area
- Si effettui il commit

Rimozione e rinominazione III

Output atteso 1

```
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        junk.txt

nothing added to commit but untracked files present (use "git add" to track)
```

Output atteso 2

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   junk.txt
```

Rimozione e rinominazione IV

Output atteso 3

```
[master 844aebd] Add junk
 1 file changed, 1 insertion(+)
 create mode 100644 junk.txt
```

Output atteso 4

```
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        deleted:    junk.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        trash.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

Rimozione e rinominazione V

Output atteso 5

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        renamed:    junk.txt -> trash.txt
```

Output atteso 6

```
[master 4d086a9] move junk to trash
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename junk.txt => trash.txt (100%)
```

Rimozione e rinominazione VI

Output atteso 7

```
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        deleted:    trash.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

Output atteso 8

```
[master 47b5f2f] Remove the trash
 1 file changed, 1 deletion(-)
 delete mode 100644 trash.txt
```

Rimozione e rinominazione VII

Output atteso 9

```
On branch master
nothing to commit, working tree clean
```

Outline

Visualizzazione della storia I

In generale

Visualizzare l'elenco dei commit effettuati, chi li ha eseguiti, quando, ed il loro message commit

Visualizzazione della storia II

In Git

Git offre il sottocomando `log`

- `git log`
  - Visualizza tutti i commit della linea di sviluppo corrente
  - Se l'output è troppo lungo crea una visualizzazione scorrevole (si vedano i comandi Unix `less` e `more`)
  - Per uscire dalla visualizzazione scorrevole, si usa il tasto Q
- `git log --graph`
  - Come sopra, con visualizzazione grafica dell'evoluzione sulla sinistra
- `git log --graph --oneline`
  - Come sopra, con visualizzazione compatta

Esercizio

- Si visualizzi l'attuale storia del repository, corredata di grafico

Visualizzazione della storia III

Output atteso

```
* commit 47b5f2fb9f5300dc8bc530ce45d37a86a0436755
| Author: Danilo Pianini <danilo.pianini@unibo.it>
| Date:   Wed Oct 19 16:46:21 2016 +0200
|
|     Remove the trash
|
* commit 4d086a9b0d2139f0cd300d329f532a2c464304c7
| Author: Danilo Pianini <danilo.pianini@unibo.it>
| Date:   Wed Oct 19 16:45:28 2016 +0200
|
|     move junk to trash
|
* commit 844aebd840e6f3d2b034312e9fa37677f64b9a15
| Author: Danilo Pianini <danilo.pianini@unibo.it>
| Date:   Wed Oct 19 16:43:48 2016 +0200
|
|     Add junk
|
* commit 3ae84225f45afdfa02c268c6079e0f6c96695c1f
| Author: Danilo Pianini <danilo.pianini@unibo.it>
| Date:   Wed Oct 19 16:26:30 2016 +0200
|
|     Create .gitignore
|
* commit 19aa252373d1e44897233bf5b733cf82019cd5bf
  Author: Danilo Pianini <danilo.pianini@unibo.it>
  Date:   Wed Oct 19 15:51:10 2016 +0200

      Create HelloWorld
```

Nell'esercitazione di oggi

- Fate in modo che i vostri esercizi siano in repository git
- Ossia, inizializzate come repository git la cartella che contiene gli esercizi del laboratorio
- Aggiungete al repository i file forniti da noi
  - Facendo attenzione a cosa tracciare
  - Creando un apposito `.gitignore`
- Prima di chiamarci per le correzioni, effettuate un commit
- Se lo desiderate, potete anche farne di mano in mano (anzi, sarebbe preferibile)

Outline

Design di una applicazione

- Come ingegneri, dovrete essere in grado di tradurre una descrizione in linguaggio naturale ("a parole") di una applicazione in termini di astrazioni adatte ad essere scritte in forma di software (nel nostro caso, in elementi di linguaggio di programmazione orientato agli oggetti)
- Si tratta dell'esercizio **più difficile** e **più importante** che dovrete fare quando realizzerete il progetto
- Dalle vostre capacità di analisi del problema e di design della soluzione dipende il successo o meno del vostro software
- Nell'ultimo esercizio, vi sarà data una descrizione in linguaggio naturale, e starà a voi realizzare il software autonomamente
- **SUGGERIMENTO** — Prima di iniziare a "sporcarsi le mani" col codice, prendete carta e penna e cercate di ottenere un design chiaro dei componenti che realizzerete: nessuno costruisce un'automobile partendo con l'assemblare i bulloni, ma decide prima quali saranno i componenti e come andranno connessi fra loro.