05
Distributed Version Control Systems I
Introduzione al Java Collections Framework

Danilo Pianini
Giovanni Ciatto, Angelo Croatti, Mirko Viroli

Ingegneria e Scienze Informatiche
Alma Mater Studiorum—Università di Bologna, Cesena

20 ottobre 2017

Preparazione Ambiente di Lavoro

- Accendere il PC
- Loggarsi sul sito del corso
  - `https://bit.ly/oop2017cesena`
- Scaricare dal sito il file `lab05.zip` contenente il materiale dell'esercitazione odierna
- Spostare il file scaricato sul Desktop
- Decomprimere il file usando 7zip (o un programma analogo) sul Desktop
- Importare la cartella `lab05` come progetto all'interno di Eclipse

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:

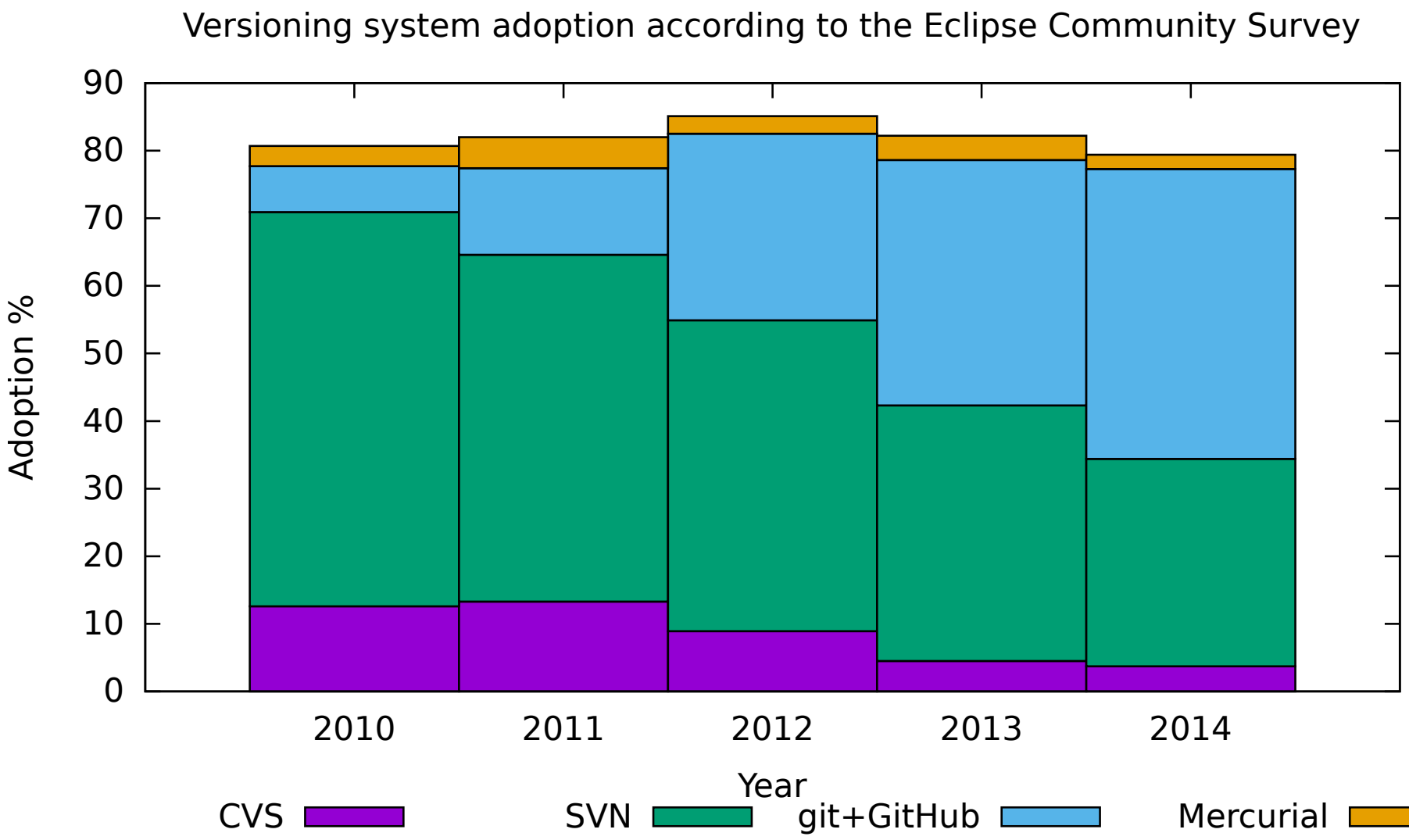
- "Fai da te" — è il sistema che la maggior parte di voi ha usato finora: si fa una copia di tutti i file in una cartella (magari numerata). Costa molto in spazio ed in tempo, rende difficile lo scambio di file e il lavoro parallelo.
- *CVS* — Fu il primo sistema di versioning. Studiato per salvare automaticamente 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 (quasi) tutti i moderni processi di sviluppo software. Un po' di esempi:

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

Diffusione


Versioning system adoption according to the Eclipse Community Survey
Adoption %
90
80
70
60
50
40
30
20
10
0
2010
2011
2012
2013
2014
Year
CVS
SVN
git+GitHub
Mercurial

Mercurial vs. Git

Similarità

- Identici principi di base
- Spesso identici comandi

Principali differenze

- Mercurial è concettualmente più snello, con architettura a plugin
- Git è più veloce, ha un'architettura monolitica
- Git gestisce il branching e gli accessi remoti in modo più semplice
- Git ha una gestione più conservativa dello "stage"
- Git nasce per le esigenze di Linus Torvalds e Linux, e si porta dietro informazioni Unix-specifiche, come i permessi dei file
  - Mercurial è più Windows-friendly...
  - ...chi sviluppa in Unix tende a preferire le feature aggiuntive di Git

Imparato uno dei due sistemi, l'altro si apprende con un investimento di poche ore.

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"

The git README.md file

```
GIT - the stupid content tracker

"git" can mean anything, depending on your mood.

 - random three-letter combination that is pronounceable, and not
   actually used by any common UNIX command.  The fact that it is a
   mispronounciation of "get" may or may not be relevant.
 - stupid. contemptible and despicable. simple. Take your pick from the
   dictionary of slang.
 - "global information tracker": you're in a good mood, and it actually
   works for you. Angels sing, and a light suddenly fills the room.
 - "goddamn idiotic truckload of sh*t": when it breaks
```

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

Commit

Salvataggio dello stato del repository

Concetti basilari e terminologia II

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

Nota: Mercurial differisce da Git in questo particolare. Il concetto di staging area in Mercurial è assente, lo stato corrente del repository è esso stesso lo stage.

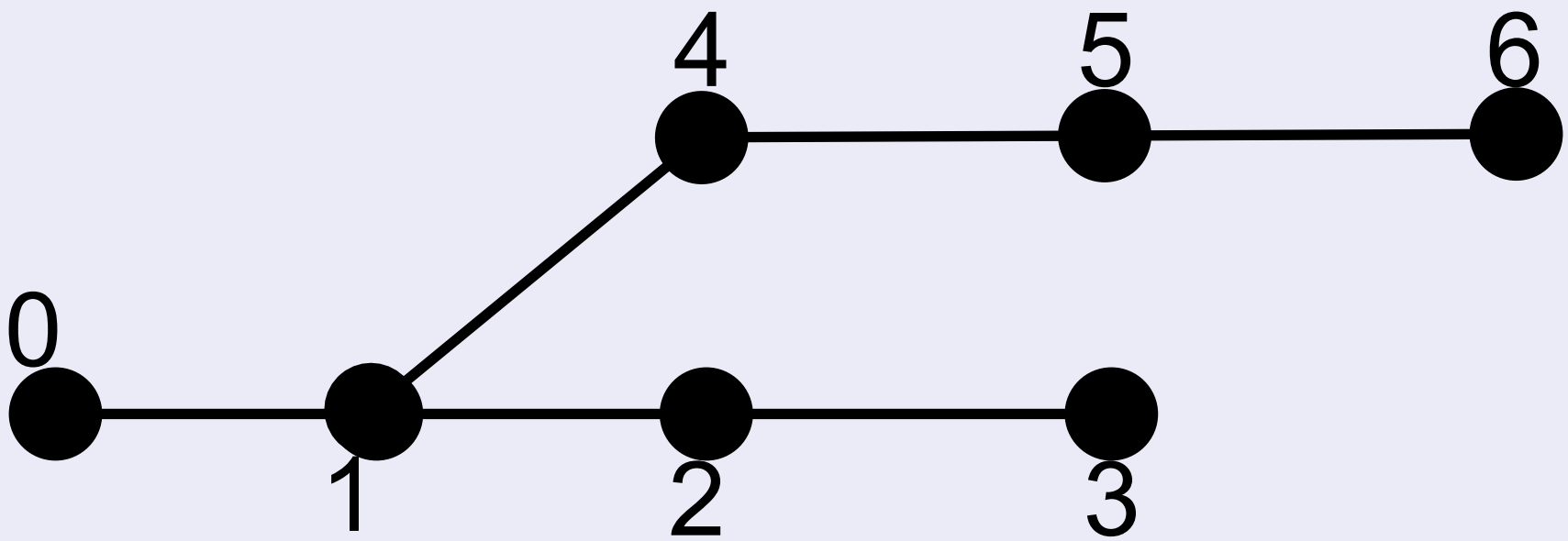
Navigazione della storia

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

Concetti basilari e terminologia III

Branch

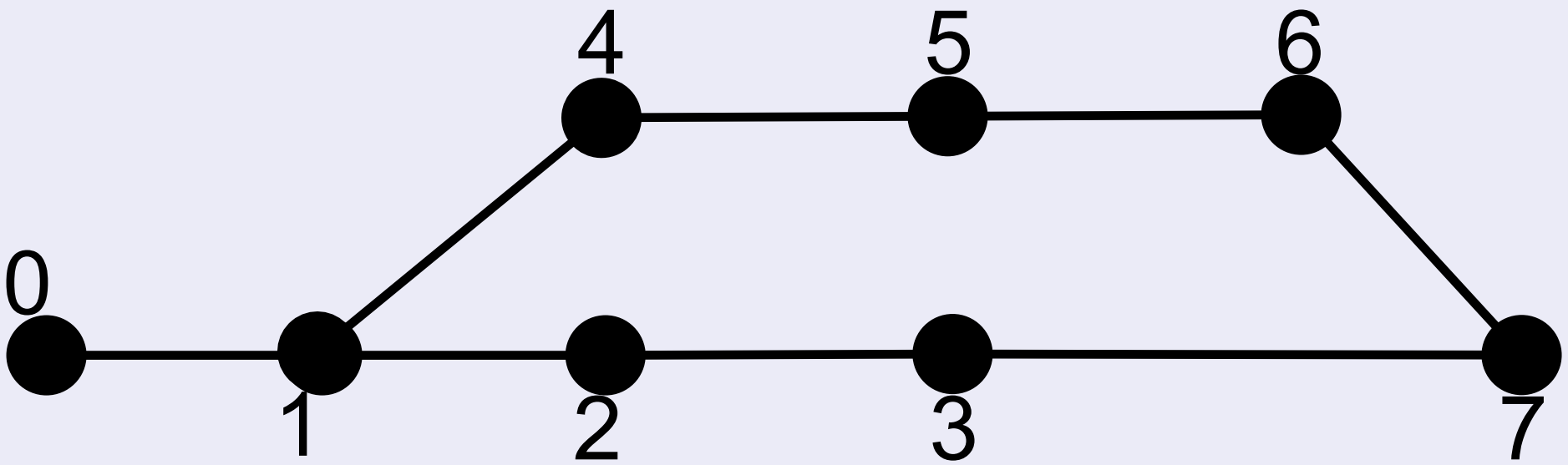
Linea di sviluppo (ossia sequenza di commit successivi). Dal momento in cui si può tornare indietro nella storia dei salvataggi e ripartire a sviluppare, si può creare una nuova linea di sviluppo, che si diparte da quella originale e procede parallelamente.



Concetti basilari e terminologia IV

Merge

Fusione di due branch (linee di sviluppo) in una sola.



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).

In Git

È possibile specificare un nome utente di default utilizzando

- `git config --global user.name "YOUR NAME"`
  - **OVVIAMENTE** al posto di `YOUR NAME` dovrete inserire il vostro nome.

È possibile specificare una email di default utilizzando

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

Configurazione globale II

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 fa più danno che utile.
- Noi useremo una precisa configurazione di Eclipse
- Vogliamo che i nostri file abbiano un terminatore preciso, e vogliamo che tutti i membri del team lo usino correttamente
  - Evitando di fare mega-salvataggi solo perché cambiano i caratteri di fine linea
- Linux e MacOS: usa il carattere presente nel testo
  - `git config --global core.autocrlf input`
- Windows: disattiva la conversione automatica a CLRF
  - `git config --global core.autocrlf false`

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.

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. Fintanto che la cartella `.git` è integra, sarà possibile ripristinare lo stato dei file del repository a qualunque versione.
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"
```

Inizializzazione di un repository III

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

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 jar copiate nel progetti (in realtà, vedrete in altri corsi, esistono sistemi migliori)
- 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 javadoc (rigenerabile dai sorgenti)
- I jar della 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 JEdit o Notepad++), 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

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

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
- I file contenenti la Javadoc
- I nostri jar

Ignorare file indesiderati II

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 III

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 IV

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 V

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 VI

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 VII

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 VIII

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 IX

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

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

Visualizzazione della storia II

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
```

Visualizzazione delle differenze I

In generale

Vogliamo poter controllare quali modifiche sono state introdotte da un commit

Visualizzazione delle differenze II

In Git

Git può mostrare le modifiche fra intercorse fra due commit col sottocomando `diff`

- `git diff`
  - Mostra le differenze fra l'ultimo commit effettuato e il contenuto della staging area
- `git diff FROM`
  - Dove `FROM` è lo hash di un commit
  - Mostra le differenze fra `FROM` e la staging area
- `git diff FROM TO`
  - Dove `FROM` e `TO` sono gli hash di due commit
  - Mostra le differenze fra `FROM` e `TO`

Il formato dell'output è lo stesso del comando Unix `diff`, è interpretabile da quest'ultimo e può essere utilizzato per creare delle "patch".

Visualizzazione delle differenze III

HEAD

Per semplificare l'accesso agli ultimi commit effettuati, Git mette a disposizione la keyword `HEAD`

- `HEAD` fa riferimento all'ultimo commit effettuato
- `HEAD~1` fa riferimento al penultimo commit effettuato
- `HEAD~2` fa riferimento al terz'ultimo commit effettuato
- `HEAD~N` fa riferimento al N-esimo commit effettuato prima dell'ultimo

È possibile quindi, ad esempio, richiedere di visualizzare le modifiche introdotte negli ultimi tre commit prima dell'ultimo usando:

- `git diff HEAD~3 HEAD`

Visualizzazione delle differenze IV

Esercizio

- Osservare le differenze fra la staging area e i tre commit precedenti (`HEAD~2`)

Output atteso

```
diff --git a/junk.txt b/junk.txt
deleted file mode 100644
index de0a8c8..0000000
--- a/junk.txt
+++ /dev/null
@@ -1 +0,0 @@
-some trash
```

Visualizzazione delle differenze V

Spiegazione dell'output

- È stato cancellato un file con permessi ottali 644
- è stato spostato `junk.txt` dentro `/dev/null`
  - Ossia cancellato
  - `/dev/null` è uno speciale device file Unix (Io vedrete in Sistemi Operativi)
- Una modifica ha rimosso una linea a partire dalla riga 0, colonna 0
- Il contenuto della riga rimossa è `some trash`

Outline

Navigazione della linea di sviluppo I

In generale

Vogliamo poter ritornare a qualunque salvataggio della nostra linea di sviluppo

Navigazione della linea di sviluppo II

In Git

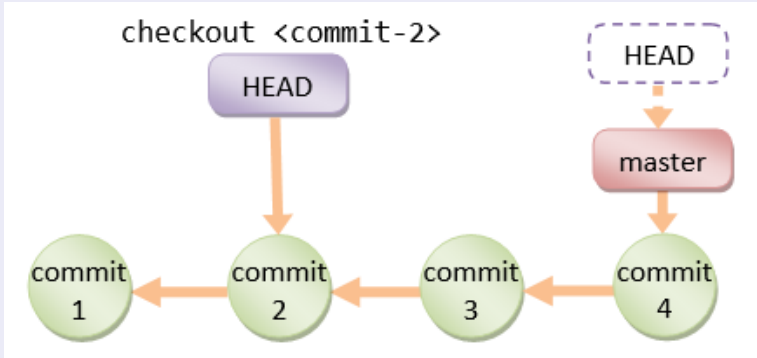
Il sottocomando `checkout` consente di ripristinare una versione precedente di un file o dell'intero repository

- `git checkout COMMITHASH`
  - Dove `COMMITHASH` è lo hash di un commit o un suo riferimento, ad esempio:
    - `HEAD`
    - `master`, o altro nome di branch
  - Se non ci sono modifiche che rischiano di essere perse, torna al salvataggio `COMMITHASH`
- `git checkout COMMITHASH -- FILENAME`
  - Ripristina il file `FILENAME` prendendolo dal commit `COMMITHASH`

Navigazione della linea di sviluppo III

La modalità detached HEAD

Quando si torna indietro nella storia, Git entra in modalità "detached HEAD": la "testa", ossia il commit a cui ci troviamo, è staccato dalla "cima" della linea di sviluppo.



- I commit effettuati in questa modalità verranno scartati
- Per tornare alla modalità "attached", è necessario effettuare un checkout con il nome del branch
  - Il nome del branch punta sempre all'ultimo commit su quella linea di sviluppo
  - Ad esempio: `git checkout master`

Navigazione della linea di sviluppo IV

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 recuperi il file `junk.txt` da tre commit fa (`HEAD~2`)
- Si vada al primo commit, usando il suo hash
  - Potete ottenerlo usando `git log`
- Si torni alla cima della linea di sviluppo (`master`)
- Si rimuova il file `junk.txt` dall'area di staging (con `reset`)
- Si elimini il file `junk.txt`

Output atteso

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

Navigazione della linea di sviluppo V

Output atteso

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

        new file:   junk.txt
```

Navigazione della linea di sviluppo VI

Output atteso

```
A       junk.txt
Note: checking out '19aa252373d1e44897233bf5b733cf82019cd5bf'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

HEAD is now at 19aa252... Create HelloWorld
```

Navigazione della linea di sviluppo VII

Output atteso

```
HEAD detached at 19aa252
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   junk.txt

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

        bin/
```

Output atteso

```
A       junk.txt
Previous HEAD position was 19aa252... Create HelloWorld
Switched to branch 'master
```

Navigazione della linea di sviluppo VIII

Output atteso

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

        new file:   junk.txt
```

Output atteso

```
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)
```

Navigazione della linea di sviluppo IX

Output atteso

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

Spiegazione dell'output

- Si noti che Git cerca di non cancellare le modifiche non salvate in un commit quando si naviga la storia (il file `junk.txt` non viene cancellato)
- In caso di conflitti, si rifiuterebbe di cambiare commit
- Si risolve cancellando le modifiche fatte o creando un nuovo commit, in modo da tornare allo stato di "`working tree clean`"

Outline

Creazione di nuove linee di sviluppo (branching) I

In generale

Vogliamo poter sviluppare su più linee

- Ad esempio perché stiamo per sviluppare una funzionalità che non sapremo se e quando completeremo, ma nel frattempo il nostro software va comunque manutenuto
- Ad esempio perché vogliamo sviluppare qualcosa a partire da una versione più vecchia (ossia, salvare i commit effettuati in modalità "detached `HEAD`"

Creazione di nuove linee di sviluppo (branching) II

In Git

Il sottocomando `checkout` consente di creare passare di branch in branch, e di crearne di nuovi tramite l'opzione `-b`

- `git checkout -b branchname`
  - Crea un nuovo branch di nome `branchname`
  - I nuovi commit apparterranno a quel branch
- `git checkout branchname`
  - Passa dal branch corrente al branch `branchname`
  - Questo in realtà è un ripasso di quanto detto un attimo fa...

Il sottocomando `branch` consente di visualizzare i branch

- `git branch`
  - Stampa i branch, mostrando con `*` quello corrente.

L'opzione `--all` di `git log` visualizza la storia per tutti i branch

- `git log --all --graph`

Creazione di nuove linee di sviluppo (branching) III

Esercizio

Premessa: si osservi e comprenda lo stato del repository con `git status` **prima e dopo ogni operazione** di modifica di file o `commit`

- Si visualizzino i branch disponibili
- Si crei un nuovo branch di nome `feature-readme`
- Si visualizzino i branch disponibili
- Si crei un nuovo file `README.md`, con del testo semplice
- Si aggiunga `README.md` alla staging area
- Si effettui il commit
- Si passi al branch `master`
- Si modifichi la stampa di `HelloWorld.java`
- Si aggiunga `HelloWorld.java` alla staging area
- Si effettui il commit
- Si visualizzi la storia dei commit su tutti i branch

Creazione di nuove linee di sviluppo (branching) IV

Output atteso

```
* master
```

Output atteso

```
Switched to a new branch 'feature-readme'
```

Output atteso

```
* feature-readme
  master
```

Output atteso

```
On branch feature-readme
nothing to commit, working tree clean
```

Creazione di nuove linee di sviluppo (branching) V

Output atteso

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

        README.md

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

Output atteso

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

        new file:   README.md
```

Creazione di nuove linee di sviluppo (branching) VI

Output atteso

```
[feature-readme 9261ec2] Add README.md file
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
```

Output atteso

```
On branch feature-readme
nothing to commit, working tree clean
```

Output atteso

```
Switched to branch 'master'
```

Output atteso

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

Creazione di nuove linee di sviluppo (branching) VII

Output atteso

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

        modified:   src/HelloWorld.java

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

Output atteso

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

        modified:   src/HelloWorld.java
```

Creazione di nuove linee di sviluppo (branching) VIII

Output atteso

```
[master 56aa7aa] Modify HelloWorld
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Output atteso

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

Creazione di nuove linee di sviluppo (branching) IX

Output atteso

```
* commit 56aa7aaad47026911c2aa2b026f33293c3b4fe31
| Author: Danilo Pianini <danilo.pianini@unibo.it>
| Date:   Thu Oct 20 15:47:57 2016 +0200
|
|     Modify HelloWorld
|
| * commit 9261ec24cfb56d7cd7ecc67d4eaa8add9c44b3ff
|/  Author: Danilo Pianini <danilo.pianini@unibo.it>
|   Date:   Thu Oct 20 15:43:50 2016 +0200
|
|       Add README.md file
|
* 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

... CONTINUA!
```

Unione di più linee di sviluppo (merge) I

In generale

Vogliamo poter unire due linee di sviluppo in una sola

- Abbiamo sviluppato in un branch separato una nuova funzionalità, ora è pronta e vogliamo unirla al resto

Unione di più linee di sviluppo (merge) II

In Git

Il sottocomando `merge` consente di unire due branch

- `git merge branchname`
  - Tenta di unire le modifiche di `branchname` al branch corrente
  - Prima di effettuare il merge, è necessario spostarsi sul branch destinazione (con `checkout`)
  - Se non ci sono conflitti, tutti i commit di `branchname` vengono aggiunti al branch corrente
  - Viene creato un nuovo commit (merge commit)
  - È buona norma non cambiare il messaggio di commit predefinito
  - La risoluzione dei conflitti sarà uno dei temi del prossimo laboratorio
- `git branch -d branchname`
  - Elimina il branch `branchname`
  - Se un branch non dovesse più servire, ad esempio perché tutte le sue modifiche sono state merse in un altro branch, è possibile rimuoverlo.

Unione di più linee di sviluppo (merge) III

Esercizio

Premessa: si osservi e comprenda lo stato del repository con `git status` **prima e dopo ogni operazione** di modifica di file o `merge`

- Si visualizzino i branch disponibili, ci si assicuri di essere su `master`
- Si faccia il merge di `feature-readme`
- Si visualizzino i file del repository con `ls -ahl` (Unix) o `dir` (Windows)
- Si visualizzi la storia dei commit su tutti i branch
- Si elimini il branch `feature-readme`
- Si visualizzino i branch disponibili
- Si visualizzi la storia dei commit su tutti i branch

Output atteso

```
  feature-readme
* master
```

Unione di più linee di sviluppo (merge) IV

Output atteso

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

Output atteso

```
Merge made by the 'recursive' strategy.
 README.md | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
```

Output atteso

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

Unione di più linee di sviluppo (merge) V

Output atteso

```
total 2.0M
drwxr-xr-x 5 danysk users 4.0K Oct 20 16:15 .
drwxr-xr-x 6 danysk users 2.0M Oct 20 15:56 ..
drwxr-xr-x 2 danysk users 4.0K Oct 19 16:22 bin
drwxr-xr-x 8 danysk users 4.0K Oct 20 16:15 .git
-rw-r--r-- 1 danysk users    5 Oct 19 18:55 .gitignore
-rw-r--r-- 1 danysk users   25 Oct 20 16:15 README.md
drwxr-xr-x 2 danysk users 4.0K Oct 19 16:00 src
```

Unione di più linee di sviluppo (merge) VI

Output atteso

```
*   commit b68c8a48dc24bd1d948c2ac4409d8d91a000b8b6
|\  Merge: 56aa7aa 9261ec2
| | Author: Danilo Pianini <danilo.pianini@unibo.it>
| | Date:   Thu Oct 20 16:15:47 2016 +0200
| |
| |     Merge branch 'feature-readme'
| |
| * commit 9261ec24cfb56d7cd7ecc67d4eaa8add9c44b3ff
| | Author: Danilo Pianini <danilo.pianini@unibo.it>
| | Date:   Thu Oct 20 15:43:50 2016 +0200
| |
| |     Add README.md file
| |
* | commit 56aa7aaad47026911c2aa2b026f33293c3b4fe31
|/  Author: Danilo Pianini <danilo.pianini@unibo.it>
|   Date:   Thu Oct 20 15:47:57 2016 +0200
|
|       Modify HelloWorld
|
* commit 47b5f2fb9f5300dc8bc530ce45d37a86a0436755
| Author: Danilo Pianini <danilo.pianini@unibo.it>
| Date:   Wed Oct 19 16:46:21 2016 +0200
|
|     Remove the trash
|

...CONTINUA!
```

Unione di più linee di sviluppo (merge) VII

Output atteso

```
Deleted branch feature-readme (was 9261ec2).
```

Output atteso

```
* master
```

Unione di più linee di sviluppo (merge) VIII

Output atteso

```
*   commit b68c8a48dc24bd1d948c2ac4409d8d91a000b8b6
|\  Merge: 56aa7aa 9261ec2
| | Author: Danilo Pianini <danilo.pianini@unibo.it>
| | Date:   Thu Oct 20 16:15:47 2016 +0200
| |
| |     Merge branch 'feature-readme'
| |
| * commit 9261ec24cfb56d7cd7ecc67d4eaa8add9c44b3ff
| | Author: Danilo Pianini <danilo.pianini@unibo.it>
| | Date:   Thu Oct 20 15:43:50 2016 +0200
| |
| |     Add README.md file
| |
* | commit 56aa7aaad47026911c2aa2b026f33293c3b4fe31
|/  Author: Danilo Pianini <danilo.pianini@unibo.it>
|   Date:   Thu Oct 20 15:47:57 2016 +0200
|
|       Modify HelloWorld
|
* commit 47b5f2fb9f5300dc8bc530ce45d37a86a0436755
| Author: Danilo Pianini <danilo.pianini@unibo.it>
| Date:   Wed Oct 19 16:46:21 2016 +0200
|
|     Remove the trash
|

...CONTINUA!
```

Conclusioni

- Avete in mano uno strumento molto potente
- Usatelo sempre d'ora in poi
- A partire da questo laboratorio!
  - Effettuate almeno un commit alla fine di ogni esercizio prima di chiamarci per correggere
  - Se ve ne servono di più, fatene di più!

Nei prossimi laboratori

- Impareremo ad usare git per lavorare in modo efficace con un team di persone
- Impareremo a scambiarci commit via Internet
- Lo useremo per ottenere le esercitazioni (bye bye zip files)

Nel progetto d'esame

- Andrà consegnato sotto forma di repository Git o Mercurial
- L'abilità nell'uso del DVCS sarà parte della valutazione

Outline

Modalità di Lavoro

1. Gli esercizi sono divisi in package con nomi progressivi
2. Troverete un commento con le istruzioni per ciascun esercizio
3. Risolvere l'esercizio in autonomia
4. Cercare di risolvere autonomamente eventuali piccoli problemi che possono verificarsi durante lo svolgimento degli esercizi
5. Utilizzare le funzioni di test presenti nei sorgenti per il testing dell'esercizio
6. Contattare i docenti nel caso vi troviate a lungo bloccati nella risoluzione di uno specifico esercizio
7. **Una volta ultimato l'esercizio chiamare i docenti per un controllo della soluzione**
8. Scrivere il Javadoc per l'esercizio svolto
9. Proseguire con l'esercizio seguente

Esercizio 1: note importanti I

Consegna

1. Leggere **bene** le slides riguardanti l'esercizio
2. Analizzare ed eseguire il programma definito in `UseCollection.java`
3. Seguire i commenti presenti in `UseSet.java` per realizzare un programma che crei, popoli e manipoli un oggetto di tipo `TreeSet<String>`

Esercizio 1: note importanti II

Ordinamento naturale e comparatori

- Gli elementi inseriti in un `TreeSet` sono ordinati secondo l'*ordine naturale*, ovvero il tipo specificato per gli oggetti contenuti nel `TreeSet` deve implementare l'interfaccia `Comparable`. Quindi, ad esempio:
  - un set di tipo `TreeSet<String>` consente di aggiungere elementi al set attraverso il metodo `add` in quanto il tipo `String` implementa l'interfaccia `Comparable`
  - viceversa, se si tenta di aggiungere un qualunque elemento ad un set di tipo `TreeSet<MyClass>` – dove `MyClass` rappresenta una classe che NON implementa l'interfaccia `Comparable` – sarà generato un errore run-time di tipo `CastClassException` (i dettagli sulle eccezioni saranno descritti in una prossima lezione)

Esercizio 1: note importanti III

Modifiche concorrenti (modifiche all'`Iterable` con `Iterator` in uso)

- Tutti gli iteratori creati/richiesti per istanze di oggetti della classe `TreeSet` sono detti *fail-fast*:
  - Se il l'istanza di `TreeSet` è sottoposta a modifica dopo la creazione dell'iteratore (prima della rimozione dell'iteratore stesso), l'iteratore produrrà un errore run-time (nello specifico, sarà lanciata l'eccezione `ConcurrentModificationException` — i dettagli sulle eccezioni saranno descritti in una prossima lezione)
  - **Provare a scrivere un ciclo foreach che itera sull'istanza di** `TreeSet` **– che quindi crea internamente un iteratore – tentando di eseguire sull'istanza il metodo** `remove` **all'interno del ciclo.** Funziona? No, perché si sta tentando di modificare il set dopo aver creato un iteratore sulla stessa istanza.

Esercizio 2

1. Seguire i commenti presenti in `UseSetWithOrder.java` per realizzare un programma che crei e ordini un oggetto di tipo `TreeSet<String>` avvalendosi di un oggetto da creare ad hoc che sia istanza di `Comparator<String>`
2. Si faccia riferimento alla documentazione di Java per verificare come poter utilizzare un Comparator per una specifica istanza di un TreeSet.

Esercizio 3

1. Implementare le classi `WarehouseImpl.java` e `ProductImpl.java` secondo i contratti definiti nelle rispettive interfacce.
2. Si faccia riferimento alla Javadoc inserita per descrive le interfacce per i dettagli circa l'implementazione dei diversi metodi.
3. Si esegua la classe `UseWarehouse.java` e si verifichi il corretto funzionamento del programma.

Esercizio 4

1. L'esercizio è una variazione di quanto fatto nell'esercizio precedente.
2. Si riutilizzi quanto più possibile il codice (funzionante) prodotto per completare l'esercizio 3
3. Per testare il buon funzionamento del codice prodotto, si completino – nel metodo `main` della classe presente in `UseWarehouse.java` – le inizializzazioni delle variabili di tipo `Product` e `Warehouse`.

Esercizio 5

1. Si implementino le funzioni descritte nella classe `Utilities.java`
2. Si verifichi il buon funzionamento di quanto implementato eseguendo il programma descritto in `UseUtilities.java`