Mosè Giordano, Pietro Giuffrida

```
git commit -m"LaTeX"
```

Git 4 LaTeX
Una guida introduttiva a Git per progetti LaTeX

v.1.0 del 2013/10/29

Licenza d'uso

Presentazione della guida

In questa guida mostreremo come utilizzare Git per tener traccia delle modifiche rilevanti e delle versioni elaborate nel corso dell'elaborazione di un documento scritto con LaTeX, un programma di composizione tipografica di alta qualità. Qui non si spiegherà il funzionamento di LaTeX, per maggiori informazioni su questo programma si possono consultare l'ottima guida di Pantieri e Gordini (2012) e la documentazione in italiano reperibile a partire dal sito del GuIt (`http://www.guitex.org`).

Solo i primi tre capitoli sono indispensabili. Nel capitolo 1 si proverà a convincere il lettore dell'utilità dei programmi per il controllo delle revisioni (VCS), saranno presentati i VCS in generale e verranno trattate le basi del funzionamento di Git in particolare. Nel capitolo 2 sarà brevemente spiegato come installare Git nei vari sistemi operativi. Nel capitolo 3 saranno illustrati i passaggi fondamentali per la creazione di un *repository* locale del proprio progetto, per il salvataggio progressivo delle versioni, e quindi per svolgere eventuali operazioni di ripristino. Il capitolo 4 è dedicato alla gestione di un *repository* remoto, utile per la collaborazione con altre persone su un comune progetto. Infine il capitolo 5 presenta un breve riepilogo schematico di tutti i comandi introdotti nella guida.

Installando Git vengono fornite due interfacce grafiche (GUI), `git-gui` e `gitk`, ma questa guida non si occuperà di descrivere l'uso di alcuna interfaccia grafica. Tutti i comandi e i passaggi illustrati vanno eseguiti tramite terminale. Chi non dovesse avere familiarità con la linea di comando può leggere la guida di Giacomelli (2013). Spesso indicheremo i comandi da eseguire nel seguente modo:

```
~/progetto$ git log -4 -p -- main.tex
```

Il carattere `$` rappresenta il *prompt* del terminale, cioè l'invito a inserire un nuovo comando. Ciò che si trova alla sinistra del *prompt* è il percorso

della cartella in cui il comando viene eseguito. In genere questo percorso è puramente esemplificativo e può essere ignorato. Tutto ciò che si trova alla destra del *prompt* è il comando vero e proprio che dovrà essere eseguito dall'utente. Dunque, nell'esempio precedente il comando va eseguito copiando, o riscrivendo, in un terminale `git log -4 -p -- main.tex` e premendo il tasto Invio. Fra parentesi ad angolo $\langle \dots \rangle$ sono riportate le parti di un comando che non devono essere ricopiate così come sono ma andranno sostituite dall'utente, come spiegato nella guida di volta in volta.

Per maggiori informazioni sui vari comandi Git illustrati, loro opzioni e argomenti è possibile consultare la documentazione presene sul proprio computer eseguendo il comando `man git-`⟨*azione*⟩ e sostituendo ad ⟨*azione*⟩ l'azione Git che si vuole approfondire. Così, per saperne di più su `git commit` si dovrà eseguire `man git-commit`.

Non abbiamo esperienza né di Git né di LaTeX su sistemi operativi non Unix-like. Un minimo di compatibilità con altri sistemi è garantita dal fatto che i comandi tipici della shell Unix sono evidenziati e commentati, in modo che l'utente di altri sistemi operativi o abituato all'uso di GUI possa sostituirli svolgendo altrimenti le medesime attività. I comandi di Git dovrebbero invece restare i medesimi in ogni sistema operativo, per quanto anche in questo caso le medesime attività possano essere svolte mediante GUI.

Non siamo esperti di informatica, ma troviamo bello far le cose, per quanto possibile, con le nostre mani, sapere cosa fa la macchina e avere l'illusione che nel rapporto quasi-simbiotico con il computer siamo noi a decidere.

I lettori possono contribuire a migliorare e ampliare le prossime versioni di questa guida con le loro critiche e i loro suggerimenti contattando gli autori oppure contribuendo direttamente al *repository* presente all'indirizzo https://github.com/GuITeX/guidagit.

Gli autori

Mosè Giordano
giordano dot mose at libero dot it
Pietro Giuffrida
pietro dot giuffri at gmail dot com

Indice

Introduzione ai VCS 1

1.1 Cos'è un VCS?

Un *version control system*, abbreviato in VCS, è un programma che permette di tener traccia di tutte le modifiche e le evoluzione effettuate nel corso della stesura di un codice o di un qualsiasi progetto su supporto digitale.

Un software VCS permette di mantenere una copia del proprio codice sorgente, sia in locale sia in remoto, senza incorrere in un eccessivo dispendio di energie e senza deconcentrarsi eccessivamente dalla stesure del proprio testo. In linea di principio un VCS permette di tenere sotto controllo qualsiasi documento, siano essi foto, documenti realizzati con programma di videoscrittura, fogli di calcolo, ecc... ma un VCS dà il meglio di sé con i file di testo semplice, non formattato, poiché permette di vedere tutte le modifiche apportate di volta in volta al file. Proprio per questo l'uso principale dei VCS è quello di controllare lo sviluppo dei software. Anche il codice dei documenti LaTeX si scrive su file di testo semplice e quindi un software VCS è particolarmente adatto a essere utilizzato in accoppiata con LaTeX.

L'insieme dei file e cartelle facenti parte di un progetto controllato da un VCS, comprendente tutta la cronologia delle modifiche, si chiama *repository*. L'operazione di registrazione di una modifica di uno o più file all'interno *repository* si chiama *commit*.

I VCS si dividono in due grandi classi: *sistemi centralizzati* e *sistemi distribuiti*. I VCS più vecchi sono di tipo centralizzato (*centralized version control system*, CVCS), cioè il *repository* principale si trova su un server remoto centrale a cui tutti gli utenti che vogliono utilizzarlo devono fare riferimento. Con il passare del tempo ci si è resi conto che questo meccanismo pone pesanti limiti, perché ogni operazione legata al VCS (controllare la cronologia, registrare una modifica, ecc...) deve necessariamente essere

fatta in un momento in cui si dispone di una connessione alla rete in cui si trova il server centrale. Inoltre questo comporta dei tempi relativamente lunghi per alcune operazioni semplici e frequenti a causa al fatto che è necessario inviare attraverso la rete la richiesta di una certa operazione al server, il quale elabora la richiesta e invia al richiedente la risposta nuovamente attraverso la rete. Quando negli anni passati le connessioni a Internet non raggiungevano le velocità attuali era spesso tedioso aspettare tanto tempo per ottenere l'output di queste operazioni. Dunque in un CVCS si è completamente dipendenti dal server centrale che contiene il *repository* del progetto, con l'ulteriore conseguenza che se il server remoto viene spento dal suo gestore o compromesso da malintenzionati tutto il lavoro può essere perduto. Per questi e altri motivi si sono diffusi in seguito i sistemi distribuiti (*distributed version control system*, DVCS), nei quali ogni utente del progetto ha una copia dell'intero *repository* sul proprio sistema e, se lo si desidera, è possibile inviare una copia del proprio *repository* su un server remoto per metterlo in condivisione con altre persone. Un DVCS può quindi riprodurre il flusso di lavoro di un CVCS ma senza avere i suoi aspetti negativi legati alle limitazioni di dover lavorare necessariamente con l'unico server centrale.

Fra i più famosi CVCS ricordiamo Concurrent Versions System (abbreviato in CVS) e Subversion (SVN), i DVCS più noti invece sono Bazaar (bzr), Git e Mercurial (hg).

1.2 Perché usare un VCS?

> Supponi che stai lavorando per tutta la notte su un documento importante, come la tesi di laurea o un articolo. Il giorno dopo ti svegli e ti chiedi come mai una scimmia abbia usato la tua tastiera per scrivere parole casuali che non formano un discorso di senso compiuto. A questo punto un VCS ti permetterà di ritornare all'ultima versione corretta del tuo documento.

Questo esempio, tratto da un commento da `hobo` sul sito TeX Stack Exchange,[1] spiega abbastanza bene perché sia utile un VCS. Naturalmente i VCS non sono gli unici software con funzioni simili. Anche i programmi

[1] http://tex.stackexchange.com/questions/1118#comment1442_1118.

di videoscrittura come Libreoffice Writer o Microsoft Word hanno delle funzioni per registrare le modifiche ai documenti, ma i VCS forniscono un controllo maggiore su quali modifiche devono effettivamente essere registrate. I VCS offrono anche la possibilità di aggiungere ai *commit* un messaggio descrittivo, in modo da poter rivedere successivamente la ragione dei cambiamenti apportati in passato. A differenza delle cronologia in Writer o Word, un VCS non è legato a uno specifico editor di testo LaTeX, anche se alcuni editor possono fornire una migliore integrazione con i VCS rispetto ad altri.

I VCS possono essere usati per tenere una copia di backup dei propri documenti LaTeX. Il primo rozzo metodo che probabilmente molti hanno provato è quello di creare copie del documento principale con nomi complicati e misteriosi del tipo `documento5.tex`, `Copia di documento.tex`, `documento_20090524.tex`, `documento-OK_ieri.tex` e così via. Quando la cartella di lavoro inizia ad affollarsi con decine di documenti simili si capisce che districarsi in questa foresta di file è tutto meno che elegante o utile. I software di backup oppure di *file hosting* online hanno esattamente questo stesso scopo, ma questi programmi generalmente creano una nuova versione dei file ogni volta che viene salvata una modifica nell'editor di testo, anche se la modifica consiste nell'aggiunta di un punto o nella cancellazione di uno spazio superfluo. Come già detto, i VCS forniscono un controllo migliore su cosa deve e cosa non deve essere monitorato e i messaggi associati alle modifiche registrate nel VCS rendono la consultazione della cronologia molto più significativa rispetto al solo elenco di date e orari di modifiche.

Come già detto, i VCS permettono di inviare una copia della propria cartella di lavoro, comprensiva di tutta la cronologia, a un *repository* remoto. Questo può essere l'unica postazione a cui diverse persone che lavorano su un unico documento devono fare riferimento. Il lavoro di gruppo è dunque notevolmente facilitato. Inoltre i DVCS permettono generalmente di impostare più di un *repository* remoto su cui copiare il proprio *repository* locale, creando quindi numerose copie differenti di backup. In realtà, sebbene possa essere utile, è spesso scomodo avere più di uno o due *repository* remoti in quanto tutte le operazioni di sincronizzazione vanno ripetute per ciascun *repository*.

I VCS rendono semplice la condivisione di file su più computer. Chi lavora su un documento dal computer fisso di casa, sul proprio laptop e sul computer del lavoro potrà salvare il *repository* su una penna USB o su un

server in remoto e lavorare su un unico progetto senza correre il rischio di fare confusione fra differenti copie dei documenti.

Tutte le revisioni di un *repository* in un VCS sono identificate con una stringa numerica o alfanumerica, che permette di ritrovarle successivamente. È possibile anche aggiungere un'etichetta (*tag*) con un nome significativo a una determinata versione del *repository* per identificare una specifica copia. Le etichette permettono di ritrovare quella particolare versione del documento più facilmente rispetto all'uso della stringa identificativa standard. Se per esempio si dà al proprio documento un numero di versione come quelli che si assegnano ai software (cosa frequente per i pacchetti e le classi di LaTeX oppure per i manuali), l'etichetta potrebbe essere il numero di versione. Si potrebbe anche etichettare la versione di un articolo che è stato inviato a una rivista. Quando il revisore dell'articolo invia i commenti sull'articolo, che nel frattempo è stato modificato, grazie alle etichette sarà facile ripescare la copia dell'articolo a cui lui fa riferimento;[2] lo stesso si potrebbe fare per identificare la copia di un documento inviata a un proprio amico.

Infine con il meccanismo del *branching* è possibile mantenere delle modifiche sperimentali parallelamente a una versione "stabile". In un documento LaTeX potrebbe trattarsi di una sezione che non si è sicuri si voglia aggiungere al documento finale. Invece nel caso di un pacchetto o di una classe LaTeX il *branching* permette di fare esperimenti TeXnici senza aver paura di perdere la versione stabile del file sulla quale si potrà continuare a lavorare. Un altro possibile utilizzo dei branch è quello di creare una copia del documento in cui andare a salvare solo alcune delle modifiche compiute nella copia principale (*cherry-picking*). Questo può essere utile per i laureandi che vogliono avere una copia personale della propria tesi differente da quella da consegnare al proprio relatore, per assecondare, per esempio, le sue bizzarre richieste riguardanti i margini e l'interlinea del documento.[3]

Riepilogando, un VCS richiede un certo, seppur minimo, lavoro da parte dell'utente, ma l'utente è ripagato dall'avere il controllo totale di quello che viene registrato, la possibilità di rivedere una cronologia commentata che

[2] Applicazione suggerita da Andrew Stacey su TeX Stack Exchange: http://tex.stackexchange.com/a/1124.

[3] Idea proposta da yoda su Stack Overflow: http://stackoverflow.com/a/6190412.

mostra anche le differenze fra ciascuna versione, la capacità di ripristinare in ogni momento il progetto a una delle versioni precedenti se necessario, l'opportunità di creare dei *branch* per sperimentare sui propri file senza alcun timore di perdere il lavoro precedente, la possibilità di condividere un progetto in tempo reale su più postazioni e con più persone. Probabilmente un software di *file hosting* andrebbe ugualmente bene per file semplici, ma un VCS offre un controllo non ancora raggiunto da quei programmi e spesso il desiderio di avere l'ultima parola sul risultato finale è il motivo per il quale un utente passa dai classici software di videoscrittura a LaTeX.

1.3 Git: the stupid content tracker

Questo paragrafo è stato in parte tradotto da http://git-scm.com/book/ch1-3.html.

Questa guida descrive in particolare uno dei più diffusi DVCS in circolazione, sicuramente quello con la comunità più attiva: Git. Git è software libero essendo rilasciato sotto licenza GPL ed è disponibile, oltre che nei *repository* delle varie distribuzioni GNU/Linux, all'indirizzo `http://git-scm.com`. Git è stato creato da Linus Torvalds, inventore del kernel Linux, perché aveva bisogno di un buon VCS per gestire lo sviluppo di Linux.

Git permette di tenere traccia delle modifiche apportate a un progetto registrando una copia dei file del progetto in un database. Sostanzialmente Git scatta una foto dei file presenti nella cartella nel momento in cui si effettua il *commit*. Per essere efficiente, se un file non è stato modificato in un *commit* Git non lo memorizza nuovamente nel *repository* ma crea un semplice collegamento all'ultima versione precedentemente salvata di quello stesso file.

Per controllare continuamente l'integrità dei dati Git utilizza un sistema, detto *checksum*, che associa a ogni stato di un progetto una sequenza di bit che la identifica. Nella fattispecie Git utilizza l'algoritmo SHA-1 che restituisce una stringa esadecimale detta *hash* composta da 40 caratteri alfanumerici (numeri da `0` a `9` e lettere da `a` a `f`) che può apparire così:

```
43c5858e91a7090b834dd9f09ddfae1061901ee4
```

In questo modo è impossibile modificare un file del progetto senza che Git se ne renda conto. Per fare riferimento a una particolare versione del progetto si indica il corrispondente *hash* oppure da una forma abbreviata

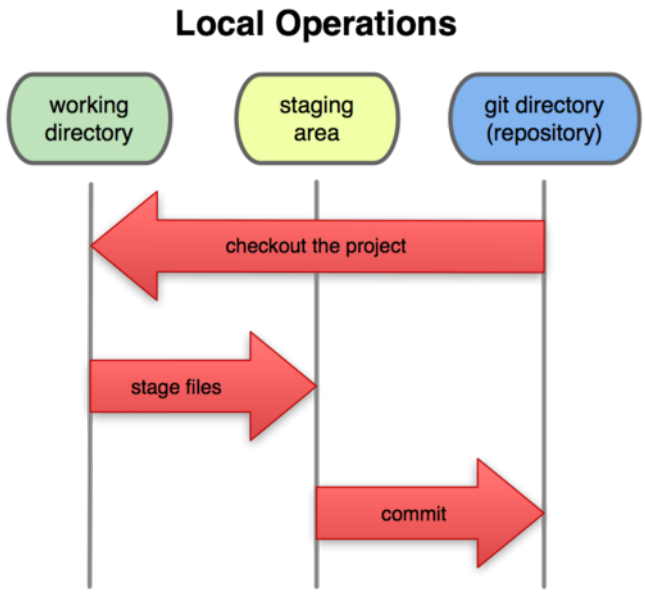

Figura 1.1 Working directory, staging area e git directory. Immagine presa da: http://progit.org/book/ch1-3.html.

dell'hash costituita da un certo numero dei caratteri iniziali, per esempio i primi sette oppure otto.

Prima di iniziare a metterci al lavoro c'è un'ultima cosa da sapere su Git. I file possono trovarsi in tre stati chiamati, in inglese, *committed*, *modified* e *staged*. *Commited* significa che il file è stato salvato nel proprio database locale; *modified* indica che il file è stato modificato ma non ancora salvato nel database con un *commit*; *staged* significa che il file è stato modificato e la sua versione attuale verrà salvata nel database con il *commit* successivo, cioè è preparato per essere aggiunto nella nuova revisione. Un progetto Git può quindi essere suddiviso in tre sezioni principali: la *git directory*, la *working directory* e la *staging area*. La prima è dove Git conserva i metadati e gli oggetti del database del proprio progetto. La *working directory* è, come dice il nome stesso, la "cartella di lavoro", ossia una copia di una versione del progetto a nostra disposizione per l'uso e la modifica dei file. L'ultima sezione, la *staging area*, è un semplice file, generalmente contenuto nella cartella Git, che conserva le informazioni su ciò che dovrà entrare nel

commit successivo.

Con Git si lavora più o meno così:

1. si modifica uno o più file presenti nella *working directory*;
2. si aggiunge un suo *snapshot*, cioè una loro copia, nella *staging area*;
3. si esegue un *commit*, cioè l'operazione con la quale i file vengono copiati così come sono presenti alla *staging area* all'interno della Git *directory* in maniera definitiva. Al *commit* viene associato l'*hash* che identifica univocamente la versione del progetto così salvata.

Se una particolare versione di un file si trova nella cartella git è considerato *committed*. Se è stato modificato e aggiunto alla *staging area* esso è detto *staged*. Se è stato modificato da quando è stata aperta la cartella di lavoro ma non ancora aggiunto alla *staging area* allora il file è detto *modified*.

Installare Git 2

Gli utenti dei sistemi operativi Mac OS e Windows possono scaricare la versione più recente di Git dal sito http://git-scm.com. Dopo aver terminato l'installazione, si ha la possibilità di usare Git sia tramite interfaccia grafica, sia tramite un shell o, per gli utenti Windows, nel più noto Prompt dei Comandi. Altre GUI sono disponibili all'indirizzo http://git-scm.com/downloads/guis.

Gli utenti dei sistemi operativi GNU/Linux o altri sistemi Unix possono compilare il codice sorgente di Git scaricabile all'indirizzo https://github.com/git/git oppure installarlo usando il proprio gestore pacchetti. Insieme a Git verranno installate le interfacce grafiche predefinite `git-gui` e `gitk`. Ecco i comandi da utilizzare per installare Git nelle principali distribuzioni:
Debian/Ubuntu

```
$ apt-get install git
```

Fedora

```
$ yum install git
```

Gentoo

```
$ emerge --ask --verbose dev-vcs/git
```

Arch Linux

```
$ pacman -S git
```

FreeBSD

```
$ cd /usr/ports/devel/git
$ make install
```

Solaris 11 Express

```
$ pkg install developer/versioning/git
```

OpenBSD

```
$ pkg_add git
```

Git e LaTeX 3

Vogliamo sviluppare un documento con LaTeX, utilizzando Git come VCS. Git non funziona in modo particolarmente esotico. Si tratta semplicemente di creare una directory, di posizionare in essa i nostri file `.tex` e di dire a Git di considerare tale directory come un *repository*.

3.1 Creazione e inizializzazione del *repository*

```
~$ mkdir progetto
~$ cd progetto
~/progetto$ touch np_main.tex
~/progetto$ git init
~/progetto$ git add .
~/progetto$ git commit -am "Inizializzazione del nuovo
    progetto"
```

Vediamo con calma i singoli passaggi. I primi tre comandi servono rispettivamente per creare la directory (`mkdir` ⟨*nome directory*⟩), per spostarsi al suo interno (`cd` ⟨*nome directory*⟩) e per creare un file vuoto chiamato `np_main.tex` (`touch` ⟨*nome file*⟩).

I passaggi successivi, eseguiti sempre dati dall'interno della cartella in cui si troverà il progetto, sono quelli specifici di Git:

- con `git init` si crea un nuovo *repository* vuoto, cioè essenzialmente la cartella nascosta chiamata `.git/` che conterrà l'intero database. Questo comando va dato solo al momento della creazione di un nuovo *repository* e poi non sarà più necessario riutilizzarlo;

- `git add .` aggiunge l'argomento (in questo caso '`.`', che è un'abbreviazione del percorso della cartella in cui siamo posizionati) alla *staging area* del *repository* appena creato;

- col comando `commit -am nota di versione` effettuiamo il commit che, come detto, salva una copia dei file contenuti nella *staging area* all'interno del database. Nella cronologia delle modifiche, a questa operazione risulterà associato il messaggio `nota di versione`.

Così facendo saremo già pronti a lavorare con il nostro editor di fiducia sul file `.tex` appena creato.

Per continuare il lavoro sul nostro documento e registrare i suoi cambiamenti dobbiamo eseguire le seguenti operazioni (il seguente elenco puntato deve essere confrontato con quello presente nel paragrafo 1.3 a pagina 7):

1. si modificano i file del codice sorgente presenti nella con il nostro editor di testo di fiducia, oppure se ne aggiungono di nuovi (per esempio possono essere aggiunte nuove figure);

2. si aggiungono i file che vogliamo registrare nel successivo *commit* alla *staging area* con il comando `git add` ⟨*file*⟩, dove al posto di ⟨*file*⟩ va sostituito l'elenco, separato da uno spazio, dei file di interesse. Per maggiori informazioni sull'aggiunta di file a un progetto Git vedi il paragrafo 3.2;

3. si effettua un *commit* con il comando `git commit`. A questo punto si aprirà l'editor di testo predefinito di Git per l'inserimento del messaggio del *commit*. Vedi il paragrafo 3.8 per sapere come configurare l'editor predefinito di Git e il paragrafo 3.1.1 per maggiori informazioni sui messaggi dei *commit*.

I punti 2 e 3 possono essere eseguiti insieme se si vogliono aggiungere alla *staging area* tutti i file che risultano *modified* nella *working directory* utilizzando il comando `git commit -a`.

Prima di eseguire un *commit* si possono modificare più file, se ne possono aggiungere di nuovi o se ne possono cancellare altri. Nel *commit* verranno memorizzati tutti i cambiamenti presenti nella *staging area* e solamente quelli, cioè non verranno considerati i file modificati ma non preparati per la nuova revisione.

3.1.1 Messaggi dei *commit*

In Git, a differenza di altri software VCS, è necessario specificare un messaggio di *commit* non vuoto. Se quando registriamo un nuovo *commit* con il comando `git commit` non specifichiamo un messaggio con l'opzione `-m` si aprirà l'editor di testo associato per default a Git in cui potremo scrivere il messaggio del *commit*. Se si utilizza un editor di testo per scrivere il messaggio di un *commit* è possibile inserire messaggi estesi su più righe. Generalmente si utilizza la seguente convenzione:

- nella prima riga del messaggio, che non deve superare i 50 caratteri, si inserisce una breve e riassuntiva descrizione del *commit* che si sta registrando. Il testo del messaggio in questo primo rigo può non essere correttamente formattato, per esempio può essere assente la punteggiatura o un corretto utilizzo dei caratteri maiuscoli;
- si lascia la seconda riga vuota e a partire dalla terza si scrive un paragrafo contenente una descrizione più dettagliata delle modifiche apportate. Il testo di questo paragrafo deve utilizzare la punteggiatura opportuna e i caratteri maiuscoli dove necessario. Anche le righe di questo paragrafo non dovrebbero superare i 72 caratteri;
- si possono inserire altri paragrafi per descrivere ulteriormente le modifiche lasciando una riga vuota fra un paragrafo e il successivo e seguendo le convenzioni illustrate nel punto precedente.

Nei messaggi il carattere `#` viene interpretato come carattere di commento e ha lo stesso utilizzo del carattere `%` in LaTeX, cioè tutto ciò che in una riga si trova alla destra del carattere `#` verrà ignorato. Si noti che all'apertura dell'editor di testo per la modifica del messaggio sono presenti, alla fine del file, delle righe di commento contenenti delle informazioni relative al *commit* che si sta registrando. Queste righe possono essere ignorate.

3.2 Aggiungere e rimuovere file del progetto

Per controllare lo stato di un *repository* Git esiste il comando `git status`. Questo comando ci permette di monitorare costantemente la situazione di tutti i file e fornisce spesso dei suggerimenti utili (in lingua inglese) sulle

operazioni che possono essere eseguite sui file. Se dopo aver effettuato un *commit* si sono modificati dei file, l'output di questo comando sarà qualcosa di simile a ciò che segue

```
# 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:   TODO
#       modified:   git4latex.tex
#
no changes added to commit (use "git add" and/or "git
     commit -a")
```

In questo esempio nella *working directory* sono presenti due file, tutti e due *modified* ma non *staged*, come ben spiegato da Git. Nell'ultima riga dell'output Git ci suggerisce anche come aggiungere i file alla *staging area*, cioè usando i già visti `git add` ⟨*file*⟩ oppure `git commit -a`. Se aggiungiamo il file `git4latex.tex` alla *staging area* l'output di `git status` diventerà

```
$ git add git4latex.tex
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   git4latex.tex
#
# 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:   TODO
#
```

Adesso `git4latex.tex` compare nell'elenco dei file che faranno parte del prossimo *commit*. Ci viene anche suggerito come rimuovere un file dalla *staging area*, usando `git reset HEAD` ⟨*file*⟩, ma riprenderemo questo discorso nel paragrafo 3.4.1. Non è necessario che tutti i file siano spostati nella *staging area* prima di effettuare un nuovo *commit*, possiamo spostare solo quelli che vogliamo registrare nella revisione successiva.

Se dopo aver spostato un file nella *staging area* lo modifichiamo nuovamente, questo apparirà sia nell'elenco dei file che faranno parte del nuovo *commit* sia in quello dei file *modified* ma non *staged*

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   git4latex.tex
#
# 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:   TODO
#       modified:   git4latex.tex
#
```

Succede questo perché Git non ricorda semplicemente l'elenco dei file che sono pronti per entrare nella nuova revisione, ma memorizza nella *staging area* una copia del file così com'era al momento dell'esecuzione del comando `git add` ⟨*file*⟩. Prima di effettuare il *commit* quindi è bene controllare che il file sia solo fra quelli *to be committed* e non anche fra i file *not staged for commit*, per evitare di inserire nel *commit* un file alla versione sbagliata. Un metodo semplice per rimediare a questo pericolo è quello di usare `git commit -a`, ma questo è possibile solo se si vogliono aggiungere al *commit* tutti i file che risultano *modified*.

Il comando `git add` permette di aggiungere al *repository* qualsiasi file presente nella *working directory*, anche quelli non ancora conosciuti da

Git.[1] Per esempio, se abbiamo creato un file per la bibliografia, chiamato `bibliografia.bib`, del nostro documento la situazione sarà la seguente

```
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be
      committed)
#
#       bibliografia.bib
nothing added to commit but untracked files present (
     use "git add" to track)
```

I file *untracked* sono quelli non ancora seguiti da Git poiché non ancora conosciuti. Come avrete probabilmente già capito, e come suggerito anche dall'output del comando, per rendere noto a Git il file dovremo usare `git add bibliografia.bib`. Se avessimo provato a effettuare direttamente un *commit* con `git commit -a` avremmo ottenuto la seguente risposta

```
$ git commit -am "aggiungo bibliografia"
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be
      committed)
#
#       bibliografia.bib
nothing added to commit but untracked files present (
     use "git add" to track)
```

Git si sarebbe accorto della presenza del nuovo file, ma non lo avrebbe aggiunto automaticamente al progetto. D'altra parte, se avesse rilevato delle modifiche ai file precedentemente aggiunti al progetto, non si sarebbe nemmeno curato di comunicarci che il nuovo file non è ancora stato aggiunto al progetto. Si sarebbe infatti limitato a salvare le modifiche ai file che gli abbiamo precedentemente detto di gestire. Solo con il comando `git status` possiamo controllare con certezza lo stato di tutti i file del *repository*.

Oltre ad aggiungere file a un *repository* è possibile, naturalmente, anche rimuoverli. La sintassi è la seguente: `git rm` ⟨*file*⟩, in cui a ⟨*file*⟩ va

[1] In effetti subito dopo aver creato il *repository* con `git init`, Git non conosceva nessun file, noi glieli abbiamo presentati con `git add` ⟨*file*⟩.

sostituito, come al solito, l'elenco dei file che si vogliono cancellare. Questo comando elimina il file dalla *working directory* e aggiunge la cancellazione del file alla *staging area*, ma non crea un nuovo *commit* che dovrà quindi essere eseguito esplicitamente, senza però aver bisogno di fare altro. Poiché la cancellazione è memorizzata nella *staging area*, per rimuovere l'eliminazione accidentale di un file, prima di effettuare un *commit*, eseguita con `git rm` ⟨*file*⟩, possiamo usare `git reset HEAD` ⟨*file*⟩. Adesso la modifica è presente solo nella *working directory* (cioè il file è assente dalla cartella di lavoro), per ripristinarlo useremo `git checkout --` ⟨*file*⟩. Non c'è bisogno di ricordare a memoria tutte le operazioni per il ripristino del file,[2] queste infatti sono suggerite dall'output di `git status`.

```
$ git rm bibliografia.bib
rm 'bibliografia.bib'
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       deleted:    bibliografia.bib
#
$ git reset HEAD bibliografia.bib
Unstaged changes after reset:
M       bibliografia.bib
$ git status
# 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:    bibliografia.bib
#
no changes added to commit (use "git add" and/or "git
     commit -a")
$ git checkout -- bibliografia.bib
$ git status
# On branch master
```

[2] Queste operazioni dovrebbero risultare più chiare dopo avere letto il paragrafo 3.4.1.

```
nothing to commit (working directory clean)
```

3.2.1 Ignorare file del progetto: `.gitignore`

Come ormai sappiamo bene, per aggiungere dei file alla *staging area* dobbiamo usare il comando `git add` ⟨*file*⟩, in cui a ⟨*file*⟩ andrà sostituito l'elenco dei file, separati da uno spazio, che si vogliono includere nel *commit* successivo. Si può scegliere di procedere in due modi: aggiungere indiscriminatamente tutti i file della cartella corrente al progetto che stiamo sviluppando; oppure aggiungere un singolo file. Nel primo caso dovremmo ripetere il comando già usato in fase di inizializzazione:

```
$ git add .
```

Nel secondo caso aggiungiamo un singolo file:

```
$ git add np_secondary.tex
```

Per quanto possa sembrare eccessivo, io preferisco usare il secondo comando. Mi accade con estrema frequenza di dover aggiungere dei file a un progetto, e spesso sono fin troppo distratto da quel che sto scrivendo per occuparmi di quel che Git si aspetta da me. L'aggiunta indiscriminata di ogni file nella directory al *repository* presenta però delle controindicazioni. La più ovvia per chi lavora con LaTeX è la seguente: nel corso dell'elaborazione di un testo inevitabilmente si procederà alla generazione del documento in pdf a partire dai sorgenti; LaTeX provvederà quindi alla generazioni di una serie di file ausiliari (`.toc`, `.out`, ...), nonché di un PDF più o meno inutile. Se eseguissimo il comando `git add .` subito dopo una compilazione, evidentemente Git aggiungerebbe alla *staging area* anche i vari file di lavoro, dal pdf, ai file di log. Per ovviare a questo inconveniente, e continuare pigramente a eseguire `git add .`, è utile creare nella nostra directory di un file denominato `.gitignore` con il seguente contenuto:

```
*.log
*.pdf
*.blg
*.bbl
*.aux
*-blx.bib
*.out
```

```
*~
```

Il metacarattere `*` rappresenta una qualsiasi sequenza di caratteri, quindi, per esempio, `*.log` indica qualsiasi file che termini con l'estensione `.log`. In un file `.gitignore` si possono anche inserire dei commenti con il carattere `#`.

Mediante questo file, istruiamo Git a proposito di tutti i file che non fanno effettivamente parte del progetto, pur essendo presenti nella cartella. Da questo momento in poi Git si limiterà a ignorarli, il che ci permette di eseguire prudenzialmente il comando `git add .` prima di ogni *commit*. Generalmente Git è usato per gestire solo il codice sorgente di un progetto, sia esso un programma o un documento LaTeX, quindi è buona norma aggiungere all'elenco dei file ignorati tutti quei file, ausiliari o di output, che vengono generati nella fase di compilazione, del programma o documento, ma che non fanno parte del codice sorgente propriamente detto. Nel caso di un documento LaTeX saranno i file con estensione, per esempio, `.log`, `.aux`, `.toc` (per i file ausiliari temporanei) e `.pdf` (per il file di output). L'aggiunta del PDF finale comporterebbe un inutile duplicato nella cronologia dato che è già presente l'intero codice sorgente. Inoltre Git è ottimizzato per lavorare con file di testo semplice e non ha buone prestazioni se nel *repository* sono presenti file binari che sono modificati frequentemente, quindi la presenza del PDF appesantisce notevolmente l'intero *repository*. Se si vuole distribuire il documento finale in formato PDF è preferibile caricarlo su un apposito sito di *file hosting*.[3]

Non è necessario, ma è preferibile aggiungere anche il file `.gitignore` al *repository* in modo che tutti gli utenti che condividono il progetto possano avere le stesse configurazioni e non debbano crearsi ciascuno il proprio `.gitignore` in locale.

3.3 Consultare la cronologia

In Git è possibile, naturalmente, consultare la cronologia dei *commit* registrati nel *repository* in uso. Il comando da usare è `git log`. L'output di questo comando, senza ulteriori opzioni, sarà qualcosa di questo tipo

[3] Alcuni siti, come https://bitbucket.org e http://sourceforge.net, che offrono la possibilità di ospitare *repository* git remoti permettono inoltre di caricare dei file esterni al *repository* proprio come un classico sito di *file hosting*.

```
commit 9e1691276ba70443c56c214a52088814a5f25ec5
Author: Pietro Giuffrida <pietro.giuffri@gmail.com>
Date:   Sat Oct 2 19:32:12 2010 +0200

    un po' di pulizia

commit 86ee0a0525dd759bb49308b58cf3e761471a04cd
Author: Pietro Giuffrida <pietro.giuffri@gmail.com>
Date:   Sat Oct 2 19:31:24 2010 +0200

    prime due sezioni

commit 43c5858e91a7090b834dd9f09ddfae1061901ee4
Author: Pietro Giuffrida <pietro.giuffri@gmail.com>
Date:   Sat Oct 2 18:53:30 2010 +0200

    Ho solo iniziato a lavorare
```

È possibile muoversi nella cronologia con i tasti freccia. Nella prima riga di ogni revisione (che inizia con `commit`) è riportato l'*hash* esteso corrispondente, nella seconda l'autore (`Author`) del *commit*, quindi la data di salvataggio (`Date`) e il messaggio descrittivo.

È possibile mostrare solo l'elenco degli ultimi n *commit* aggiungendo l'opzione `-`$\langle n \rangle$. Dunque per mostrare solo l'ultimo *commit* possiamo usare il comando `git commit -1`. Si può consultare la cronologia relativa a uno o più specifici file e non all'intero *repository* passando come argomento del comando l'elenco dei file di interesse, separati da uno spazio: `git log --` $\langle$*file*$\rangle$. Se si vuole visualizzare solo l'elenco degli ultimi `n` *commit* relativi a un determinato elenco di file si possono utilizzare contemporaneamente l'opzione `-`$\langle n \rangle$ e l'argomento $\langle$*file*$\rangle$. Per esempio il comando `git log -3 -- main.tex capitolo2.tex` mostrerà gli ultimi tre *commit* che hanno modificato i file `main.tex` e `capitolo2.tex`. Il doppio trattino `--` serve per far capire a Git che tutto ciò che segue è l'elenco dei file. In generale non è obbligatorio utilizzarlo ma diventa necessario per evitare ambiguità nel caso in cui ci siano rami o etichette con lo stesso nome del file.

Come abbiamo visto, in maniera predefinita `git log` mostra solo alcune informazioni dei *commit*. Per mostrare anche le modifiche apportate in ciascun *commit* nel formato `diff` si può utilizzare l'opzione `-p`. Per esempio,

l'esecuzione del comando `git log -1 -p` potrà dare un output di questo tipo:

```
commit 5a5f793b0780d2c3352239beca3fc8de432a71b9
Author: Paolino Paperino <paolino.paperino@example.org>
Date:   Fri Oct 8 22:32:51 2010 +0200

    rimuovo pacchetto 'xcolor'

    Ora che l'ambiente 'lstlisting' non e' piu'
         colorato
    non e' piu' necessario.

diff --git a/git4latex.tex b/git4latex.tex
index 74a1474..a84fb27 100644
--- a/git4latex.tex
+++ b/git4latex.tex
@@ -26,9 +26,6 @@
 \usepackage[font=small,format=hang]{caption}
 \usepackage{graphicx}

-\usepackage[dvipsnames, usenames]{xcolor}
-\definecolor{GY}{named}{GreenYellow} % SkyBlue
-
 \usepackage{listings}
 \usepackage{fourier}
```

Segnaliamo infine un'altra utile opzione che serve per mostrare come informazioni del *commit* solo l'*hash* in formato breve e la prima riga del messaggio. Questa opzione si chiama `--oneline` ed è utile se si vuole trovare rapidamente l'*hash* di un *commit* sfogliando le prime righe dei messaggi di tutti i commit. Per esempio, il primo output di `git log` riportato all'inizio di questo paragrafo, con l'opzione `--oneline` apparirebbe così:

```
$ git log --oneline
9e16912 un po' di pulizia
86ee0a0 prime due sezioni
43c5858 Ho solo iniziato a lavorare
```

Per maggiori informazioni è possibile consultare la documentazione di `git log` con il comando `man git-log`.

3.4 Annullare e cambiare modifiche precedenti

Ci sono vari modi per cambiare le modifiche effettuate in un *repository* Git, ma per scegliere quella adatta bisogna sapere che tipo di modifica si desidera annullare. In questo paragrafo presenteremo alcune delle situazioni più frequenti che si possono incontrare.

3.4.1 Annullare modifiche non ancora salvate

Se si vogliono annullare tutte le modifiche effettuate a dei file ma non ancora registrate in un *commit* si può usare il comando

```
$ git reset --hard HEAD
```

In questo modo l'intera *working directory* viene ripristinata alla stato corrispondente all'ultimo *commit* della cronologia, indicato con `HEAD`.

Se si vogliono riportare allo stato dell'ultimo *commit* registrato solo determinati file che sono *modified* ma non ancora *staged*, adottando la terminologia vista all'inizio, senza toccare la restante *working directory* si può utilizzare il comando

```
$ git checkout -- ⟨file⟩
```

in cui a ⟨*file*⟩ va sostituito l'elenco dei file, separati da uno spazio, che si vogliono ripristinare. Le opzioni e gli argomenti che il comando `git commit` può accettare sono numerosi e le operazioni che verranno eseguite possono essere diverse a seconda delle istruzioni fornite (ma non è scopo della presente guida spiegare nei dettagli il completo funzionamento di Git), il doppio trattino `--` serve per far capire a Git, senza possibilità di ambiguità, che tutto ciò che si trova dopo è l'elenco dei file e l'operazione da eseguire è quella qui descritta. Il doppio trattino può essere omesso se non ci sono possibili ambiguità, per esempio se il file da passare come argomento non ha lo stesso nome di uno dei *branch* (vedi paragrafo 3.6) del *repository*. Il comando `git checkout --` ⟨*file*⟩ può anche essere usato per ripristinare file accidentalmente cancellati prima effettuare un nuovo *commit*. In questo caso l'uso del doppio trattino è necessario dal momento che il file cancellato non si trova nella *working directory* e Git non capirebbe che ⟨*file*⟩ è con certezza un elenco di file cancellati. Altri modi per ripristinare file cancellati saranno visti nel paragrafo 3.5.

Se invece si vogliono solo togliere uno o più file dalla *staging area* ma lasciare intatta la *working directory* si può usare il comando

```
$ git reset HEAD ⟨file⟩
```

Come al solito, a ⟨*file*⟩ va sostituito l'elenco dei file di interesse, separati da uno spazio. I file tolti dalla *staging area* possono poi anche essere ripristinati allo stato del *commit* precedente usando il comando `git checkout --` ⟨*file*⟩ illustrato qui sopra.

3.4.2 Modificare l'ultimo *commit* non inviato in remoto

L'opzione `--amend` del comando `git commit` permette di modificare l'ultimo *commit*. Se dopo aver salvato un *commit* ci accorgiamo che dobbiamo correggerlo (per esempio perché ci siamo dimenticati apportare una modifica o di aggiungere un file), possiamo normalmente modificare i file di interesse, aggiungerli alla *staging area* con il comando `git add` ⟨*file*⟩[4] e poi, invece di effettuare un nuovo *commit* con `git commit`, eseguire il comando `git commit --amend` che modificherà l'ultimo *commit* della cronologia. A questo punto si aprirà la solita interfaccia che compare al momento del salvataggio di un *commit*: è possibile modificare il messaggio, altrimenti è sufficiente uscire direttamente dall'interfaccia.

Se bene sia in realtà possibile, è altamente sconsigliato modificare un *commit* già inviato in remoto con `git push` (vedi il paragrafo 4.2) se il *repository* è condiviso con altre persone, poiché i loro *repository* non potranno più continuare a essere sincronizzati con il *repository* centrale. Quindi, se si condivide il progetto con altri utenti è preferibile riscrivere la cronologia con `git commit --amend` solo per *commit* non ancora inviati in remoto.

3.4.3 Annullare un *commit*

Per annullare le modifiche apportate con uno specifico *commit*, creando un nuovo *commit* si può usare il comando

```
$ git revert ⟨hash commit⟩
```

[4] Se ci si era dimenticati di aggiungere un file nuovo o modificato alla *staging area* prima di effettuare il *commit* precedente, sarà sufficiente eseguire `git add` ⟨*file*⟩ senza dover modificare ulteriormente alcun file.

in cui a ⟨*hash commit*⟩ va sostituito l'*hash* del *commit* che si vuole annullare (vedi il paragrafo 3.3 per sapere come recuperare l'*hash* abbreviato sfogliando la cronologia delle modifiche). Se per esempio la forma abbreviata dell'*hash* corrispondente al *commit* che si vuole annullare è `d4fcdbd` il comando da usare è

```
$ git revert d4fcdbd
```

Dopo aver eseguito il comando si aprirà l'interfaccia per la modifica del messaggio del nuovo *commit*. Se il *commit* da annullare è l'ultimo al posto dell'*hash* si può usare il riferimento `HEAD`: `git revert HEAD`.

3.5 Ripristinare file

In questo paragrafo vedremo come ripristinare file cancellati dal progetto oppure riportare file ancora presenti nel *repository* a una versione precedente.

Ipotizziamo a titolo esemplificativo il seguente scenario: abbiamo creato un file, `np_secondary.tex`, l'abbiamo aggiunto al *repository* Git con il comando `git add np_secondary.tex` e abbiamo eseguito il commit. Dopo di ciò abbiamo accidentalmente cancellato il file con il comando `rm np_secondary.tex` e abbiamo effettuato un altro *commit* che ha registrato la cancellazione del file:

```
$ echo "pippo" >> np_secondary.tex
$ git add np_secondary.tex
$ git commit -am "Ho solo iniziato a lavorare"
$ rm np_secondary.tex
$ git commit -am "Ma ho gia' perso tutto"
```

Ora chiediamo conto a Git della sua capacità di ripristinare una versione precedentemente salvata. La prima cosa da fare è consultare la cronologia dei *commit* precedentemente effettuati, per ottenere l'*hash* della versione che vogliamo ripristinare. Il comando appropriato quindi, per quanto visto nel paragrafo 3.3, è `git log --` ⟨*file*⟩. Poiché è sufficiente conoscere l'*hash* in forma abbreviata dell'ultima revisione in cui il file era ancora presente nel progetto usiamo le opzioni `--oneline` e `-2` (l'ultima revisione in cui comparirà il file `np_secondary.tex` è quella in cui è stato cancellato, ma a noi interessa quella precedente)

```
$ git log -2 --oneline -- np_secondary.tex
3802287 Ma ho gia' perso tutto
c21d825 Ho solo iniziato a lavorare
```

Per ripristinare il file cancellato all'ultima versione in cui ancora era presente nel progetto possiamo usare il comando `git checkout` ⟨*hash del commit*⟩ `--` ⟨*file*⟩ in cui ad ⟨*hash del commit*⟩ dovremo sostituire l'*hash* (eventualmente in forma abbreviata) del *commit* a cui vogliamo ripristinare i file elencati in ⟨*file*⟩. Quindi nel nostro esempio per ripristinare il file `np_secondary.tex` alla versione presente nel *commit* `c21d825` useremo il comando

```
$ git checkout c21d825 -- np_secondary.tex
```

Il comando `git checkout` ⟨*hash del commit*⟩ `--` ⟨*file*⟩ permette di riportare qualsiasi file elencato in ⟨*file*⟩ alla versione ⟨*hash del commit*⟩, anche se non cancellato e ancora presente nel progetto, sempre utilizzando la stessa sintassi. Chiaramente, Git permette il ripristino esclusivamente dei file aggiunti al *repository* ed esclusivamente di versioni esplicitamente salvate mediante il comando `git commit`. I salvataggi effettuati tra un *commit* e l'altro non sono quindi ricostruibili.

L'uso di `git checkout --` ⟨*file*⟩ visto nel paragrafo 3.4.1 è un caso particolare di quello qui spiegato, in cui si ripristina un file alla versione presente nell'ultimo *commit*, nel caso in cui non ne viene indicato esplicitamente uno differente.

È inoltre possibile visionare nel terminale una precedente versione di un file con la sintassi

```
$ git show ⟨hash del commit⟩ ⟨file⟩
```

3.6 Gestione dei *branch*

Con il comando `git status` possiamo controllare lo stato del progetto, per esempio se ci sono dei file che sono stati modificati ma non ancora salvati nel *commit* e così via. Un tipico output di questo comando è:

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

L'ultima riga del messaggio significa che non sono presenti file modificati (a parte eventualmente quelli ignorati con il file `.gitignore`) dopo l'ultimo *commit*. Ma cosa significa `On branch master`? *Branch* in inglese significa "ramo" e Git ci dà la possibilità di creare una linea di sviluppo per il nostro progetto parallela a quella iniziale (chiamata *master* in Git) ma con delle modifiche che divergono da questa proprio come il ramo di un albero diverge dal tronco centrale (e per completare questa analogia botanica la linea di sviluppo centrale è a volte chiamata proprio *trunk*, cioè "tronco").

3.6.1 Perché creare un *branch*?

Le risposte a questa domanda possono essere diverse. Una potrebbe essere, per esempio perché abbiamo intenzione di scrivere un capitolo per il nostro documento LaTeX ma non siamo sicuri che sia veramente necessario: possiamo allora creare un *branch* nel quale scriveremo solo questo capitolo mentre nel ramo principale *master* continueremo normalmente a modificare il documento. Quando il capitolo sarà pronto, se il risultato sarà soddisfacente potremo procedere alla funsione del ramo "sperimentale" in quello principale, operazione chiamata *merge* in gergo, e nel nostro documento "comparirà magicamente" il capitolo che abbiamo sviluppato nel *branch*, in caso contrario possiamo semplicemente cancellare il *branch*, come se avessimo potato il ramo indesiderato dell'albero.

Un altro caso in cui può essere utile creare un *branch* è quello in cui si lavori a più mani su un documento: mentre voi scrivete il vostro documento un vostro amico vi chiede di collaborarvi e anch'egli userà Git per tenere traccia delle sue modifiche. Lui creerà un suo ramo di sviluppo in modo che voi potrete continuare a redigere il vostro documento senza che le sue modifiche vadano a confliggere con le vostre. Alla fine il vostro amico vi farà vedere il risultato delle sue modifiche: se le apprezzate potete effettuare il *merge* del suo ramo nel vostro, altrimenti potete dirgli che le sue modifiche non vi piacciono e il suo *branch* sarà cancellato.

Nella figura 3.1 è rappresentato un semplice grafico dello sviluppo di un progetto in cui è stato effettuato un *branching*. Nel ramo `master` abbiamo effettuato le modifiche indicate con `A` e `B`. A questo punto creiamo il ramo `branch`. D'ora in poi le modifiche effettuate su un ramo avranno valore solo su quello: continuiamo a lavorare normalmente su `master` apportando le modifiche indicate con `C` e `D`, mentre in `branch` avremo effettuato le

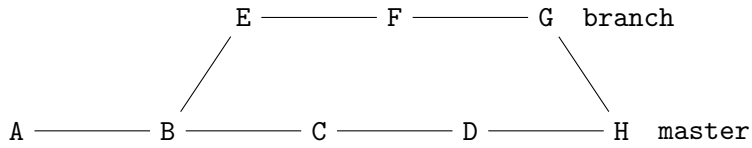

Figura 3.1 Flusso delle modifiche effettuato nel ramo principale `master` e nel ramo secondario `branch`. Il ramo `branch` viene fuso nel ramo `master` con il *commit* `H`.

modifiche `E`, `F` e `G` che apparterranno solo a questo ramo. Decidiamo quindi di fondere i due rami perché le modifiche fatte in `branch` ci soddisfano: con la modifica `H` tutte le modifiche effettuate in `branch` verranno importate nel ramo principale e potremo continuare lì il nostro lavoro.

3.6.2 Creare e cancellare un *branch*

Per creare un *branch* dobbiamo usare il comando `git branch` ⟨*nome branch*⟩. Se il comando va a buon fine non ci sarà nessun output, possiamo però controllare l'elenco dei rami esistenti con il comando `git branch` senza alcun ulteriore argomento:

```
$ git branch foo
$ git branch
  foo
* master
```

L'asterisco vicino a `master` sta a indicare che al momento ci troviamo nel *branch* `master`. Per spostarci in un altro *branch* usiamo il comando `git checkout` ⟨*nome branch*⟩:

```
$ git checkout foo
Switched to branch 'foo'
$ git branch
* foo
  master
```

Che significa che ci siamo spostati in un altro *branch*? Abbiamo cambiato cartella? No, siamo sempre nella stessa cartella (possiamo controllarlo con il comand `pwd`), ma è stato Git che ha modificato la nostra cartella di

lavoro, la *working directory*, andando a prendere dal suo database l'ultimo *snapshot*, l'ultima "fotografia", del *branch* che gli abbiamo richiesto. In alternativa a `git branch` ⟨*nome branch*⟩, per creare un nuovo ramo si può usare il comando `git checkout -b` ⟨*nome branch*⟩ che in più effettua direttamente il *checkout* del nuovo *branch*:

```
$ git checkout -b foo
Switched to a new branch 'foo'
```

Un *branch* appena creato e non modificato è uguale al ramo da cui è stato creato, contiene i suoi stessi file e la stessa cronologia. Lo sviluppo del progetto può continuare in maniera identica a quella vista prima (cioè si usano i soliti comandi `git add` e `git commit`), però le modifiche apportate in questo *branch* non compariranno nel registro di quello principale. Possiamo verificarlo controllando i rispettivi log:

```
$ git branch
* foo
  master
$ git log
commit c71da4fe9bf764274e67a2326ec1dc3911691928
Author: Pietro Giuffrida <pietro.giuffri@gmail.com>
Date:   Sat Oct 9 19:09:56 2010 +0200

    primo commit nel branch 'foo'

commit e17384278acadbfefefebf0ced4e7103005a597c
Author: Pietro Giuffrida <pietro.giuffri@gmail.com>
Date:   Sat Oct 9 19:08:46 2010 +0200

    iniziamo il nuovo progetto
$ git checkout master
Switched to branch 'master'
$ git log
commit e17384278acadbfefefebf0ced4e7103005a597c
Author: Pietro Giuffrida <pietro.giuffri@gmail.com>
Date:   Sat Oct 9 19:08:46 2010 +0200

    iniziamo il nuovo progetto
```

Per cancellare un ramo di cui si è già effettuato il *merge* con il ramo principale (vedremo più avanti come si fa) si usa l'opzione `-d` del comando `git branch`:

```
$ git branch -d foo
Deleted branch foo (was c71da4f).
```

Se proviamo a usare l'opzione `-d` con un ramo di cui non è stato ancora effettuato il *merge* Git ci avverte:

```
$ git branch -d foo
error: The branch foo' is not fully merged.
If you are sure you want to delete it, run 'git branch
    -D foo'.
```

Come al solito Git ci suggerisce i comandi che ci possono tornare utili: se vogliamo ugualmente cancellare il *branch* che non è stato fuso in quello principale dobbiamo usare l'opzione `-D` al posto di `-d`:

```
$ git branch -D foo
Deleted branch foo (was c71da4f).
```

3.6.3 Effettuare un *merge*

Per effettuare la fusione, *merge*, di due rami si deve utilizzare il comando `git merge` ⟨*nome branch*⟩, da dare nel ramo in cui si desidera importare il ramo chiamato ⟨*nome branch*⟩. Se tutto va bene otterremo un output simile al seguente (supponiamo, per esempio, che il ramo da importare si chiami `foo`):

```
$ git merge foo
Updating da45d11..729129c
Fast-forward
 capitolo5.tex |   18 ++++++++++++++++++
 1 files changed, 18 insertions(+), 0 deletions(-)
 create mode 100644 capitolo5.tex
```

Dopo di ciò possiamo normalmente fare il *commit* con `git add .` e `git commit -am messaggio di commit`.

Può verificarsi un problema se uno o più file sono stati modificati nello stesso punto in tutti e due i rami che si vogliono fondere, dopo che è stato

creato il *branch*. Git, infatti, non è in grado di gestire automaticamente questa situazione (è uno strumento potente ma non può certo leggere nel nostro pensiero, non può sapere perché il file è stato modificato diversamente nei due rami) e ci avverte con un messaggio di questo tipo:

```
$ git merge foo
Auto-merging main.tex
CONFLICT (content): Merge conflict in main.tex
Automatic merge failed; fix conflicts and then commit
     the result.
```

Nel presente esempio il file in cui si sono verificati i conflitti si chiama `main.tex`. Aprendo con il nostro editor di testo troveremo nel punto in cui si è verificato il conflitto qualcosa di questo tipo:

```
<<<<<<< HEAD
\usepackage{hyperref}
=======
\usepackage[bookmarks=false,colorlinks=true]{hyperref}
>>>>>>> foo
```

La zona compresa fra `<<<<<<< HEAD` e `=======` indica il contenuto del file presente nel ramo corrente, mentre ciò che è scritto fra `=======` e `>>>>>>> foo` è ciò che si trova nel corrispondente punto dello stesso file `main.tex` ma nel ramo `foo` che stiamo provando a fondere in `master`. Dopo aver corretto il file come desideriamo che risulti, possiamo finalmente salvarlo ed effettuare come al solito il *commit*.

3.6.4 Copiare un singolo *commit* da un *branch* a un altro

Mentre sviluppiamo un *branch* potremmo accorgerci che il *commit* appena effettuato apporta delle modifiche che possono essere utili anche in un altro *branch* (come per esempio la correzione di un errore di ortografia o una modifica al preambolo del documento). Non è necessario effettuare il *merge* solo per un importare un *commit* o modificare manualmente i file dell'altro *branch* perché è possibile effettuare un'operazione detta *cherry-picking* (che significa "raccolta di ciliege"): essa permette di importare singoli *commit* da un *branch* a un altro. La sintassi è semplice (prima di dare il seguente comando bisogna posizionarsi nel *branch* in cui si intende importare il

commit): `git cherry-pick` ⟨*hash del commit*⟩. Inoltre l'oggetto, l'autore (nel caso in cui ci siano più collaboratori che partecipano al progetto) e la data e orario del *commit* saranno gli stessi di quello importato. Al posto dell'intero *hash* del *commit* si può utilizzare la forma abbreviata composta dai primi sette caratteri.

Se dovesse verificarsi qualche problema durante il *cherry-picking*, come può succedere durante il *merge*, Git ci avverte, ci consiglia di risolvere i conflitti, usare il solito `git add` come opportuno e poi, per effettuare il *commit*, di dare il comando `git commit -c` ⟨*hash del commit*⟩, in modo da utilizzare le stesse informazioni del *commit* a cui ci riferiamo (autore, data e orario, oggetto):

```
$ git cherry-pick 97b8c9b
Automatic cherry-pick failed.  After resolving the
     conflicts,
mark the corrected paths with 'git add <paths>' or 'git
      rm <paths>'
and commit the result with:

         git commit -c 97b8c9b
```

3.7 Etichette

In Git è possibile creare delle etichette (*tag*) per identificare una particolare revisione. L'utilità delle etichette è che hanno dei nomi molti più significativi e facili da ricordare rispetto all'*hash* del corrispondente *commit*. L'uso delle etichette è molto vario, si possono usare per indicare la revisione corrispondente a una particolare versione del documento, qualora si adotti un sistema di numerazione delle versioni, oppure per ricordarsi quale sia la revisione della tesi che è stata fatta leggere al proprio relatore. Per creare una nuova etichetta si può usare il comando `git tag` ⟨*etichetta*⟩ ⟨*hash*⟩. L'argomento obbligatorio ⟨*etichetta*⟩ è il nome con cui sarà etichettato il *commit* corrispondente ad ⟨*hash*⟩. L'argomento ⟨*hash*⟩ è opzionale, se omesso verrà etichettato l'ultimo commit. Quindi, per etichettare l'ultimo *commit* con la stringa `v1.0`, indicante che la versione 1.0 del documento, si dovrà usare il comando

```
$ git tag v1.0
```

Un *commit* etichettato potrà in seguito essere richiamato indifferentemente con il suo *hash* oppure con la corrispondente etichetta.

3.8 Configurazioni basilari di Git

Per quanto non strettamente indispensabile per un uso individuale di Git sul proprio pc, segnalo alcune configurazioni elementari necessarie per un uso ottimale di Git.

Le impostazioni di Git possono essere impostate con il comando `config` di Git. Usando l'opzione `--system` le configurazioni così impostate saranno valide per tutti gli utenti che hanno accesso al sistema. Con l'opzione `--global` le impostazioni saranno valide solo per il proprio utente e verranno salvate nel file `~/.gitconfig`. Infine non usando alcuna opzione le configurazioni avranno valore solo per il *repository* in cui vengono impostate.

In primo luogo occorre passare a Git qualche informazione circa l'utente:

```
git config --global user.name "Pietro Giuffrida"
git config --global user.email pietro.giuffri@gmail.com
```

In secondo luogo è utile dire a Git di colorare i log, in modo da renderli più leggibili:

```
git config --global color.branch auto
git config --global color.diff auto
git config --global color.interactive auto
git config --global color.status auto
```

Possiamo poi impostare l'editor di testo predefinito da associre a Git con l'opzione `core.editor`. Se per esempio vogliamo usare Gedit daremo il comando

```
git config --global core.editor gedit
```

Repository remoti 4

Finora abbiamo visto come gestire un *repository* locale. In questa sezione vedremo quali sono i comandi per mantenere il proprio lavoro in locale sincronizzato con un *repository* remoto.

4.1 Gestione di un *repository* remoto

Per copiare in locale un *repository* remoto si deve utilizzare il comando `git clone` ⟨*indirizzo del repository*⟩. L'⟨*indirizzo del repository*⟩ da inserire sarà fornito da chi gestisce il progetto. Per i *repository* online questo indirizzo è spesso ben visibile nella loro pagina Internet. Per esempio, per copiare il *repository* di questa guida si deve eseguire il comando

```
git clone https://github.com/GuITeX/guidagit
```

Il comando `git remote` permette di gestire i diversi *repository* remoti. Per far riferimento a un *repository* remoto si utilizza una nome, che in maniera predefinita è `origin`. È possibile cambiare il nome con il comando `git remote rename` ⟨*vecchio nome*⟩ ⟨*nuovo nome*⟩. Per aggiungere un nuovo *repository* bisogna utilizzare sempre il comando `git remote`, ma con un'azione differente: `git remote add` ⟨*nome del repository*⟩ ⟨*indirizzo del repository*⟩. Se si copia un *repository* con `git clone`, quel *repository* è automaticamente aggiunto con il nome `origin`. Per cambiare l'indirizzo di un *repository* l'azione da eseguire è `set-url`: `git remote set-url` ⟨*nome del repository*⟩ ⟨*nuovo indirizzo*⟩. Infine per eliminare dall'elenco dei *repository* remoti usati si dovrà eseguire `git remote rm` ⟨*nome del repository*⟩.

Se si possiede l'autorizzazione a scrivere in un *repository* remoto precedentemente configurato, è possibile inviarvi i nuovi *commit* eseguiti con

il comando `git push` ⟨*nome remoto*⟩ ⟨*ramo*⟩. Per esempio, per inviare a `origin` il ramo `master` il comando da utilizzare è

```
git push origin master
```

Per associare a un ramo *repository* remoto predefinito a cui inviare le nuove modifiche, bisogna usare l'opzione `-u` in questo modo: `git push -u` ⟨*nome remoto*⟩ ⟨*ramo*⟩. Quindi, per rendere `origin` il *repository* predefinito del ramo `master` si eseguirà

```
git push -u origin master
```

Se in `git push` ⟨*nome remoto*⟩ ⟨*ramo*⟩ non si specifica ⟨*ramo*⟩, verranno inviati tutti i rami che hanno ⟨*nome remoto*⟩ come *repository* remoto predefinito; se non si specifica ⟨*nome remoto*⟩ il ramo verrà inviato nel *repository* remoto predefinito per il ramo attuale. In definitiva, la prima volta che si devono inviare le modifiche di un ramo (per esempio, `master`) a un *repository* remoto (per esempio, `origin`) si deve eseguire il comando `git push -u origin master`, mentre per tutte le operazioni successive di invio in remoto sarà sufficiente dare `git push` senza ulteriori argomenti. Per inviare in remoto anche tutte le etichette con `git tag` bisogna aggiungere all'azione `push` l'opzione `--tags`: `git push --tags`.

L'ultima azione da menzionare in questo paragrafo è il *push*, vale a dire lo scaricamento di un ramo da un *repository* remoto, quindi l'operazione opposta di un *push*: `git pull` ⟨*remoto*⟩ ⟨*ramo*⟩ aggiorna il ramo chiamato ⟨*ramo*⟩ usando il *repository* ⟨*remoto*⟩. Se non si specificano gli argomenti, i comportamenti predefiniti sono gli stessi descritti sopra per `git push`.

4.2 *Repository* online

Fra i principali siti che permettono di ospitare progetti Git ricordiamo Bitbucket (`https://bitbucket.org`), GitHub (`https://github.com`), Gitorious (`https://gitorious.org`), Google Code (`code.google.com`) e SourceForge (`https://sourceforge.net`). Su ciascuno di questi siti, dopo aver creato un nuovo progetto sarà fornito l'indirizzo del *repository* remoto. Sarà quindi possibile sincronizzare il proprio *repository* locale con quello remoto usando i comandi descritti nel paragrafo precedente.

Riepilogo dei comandi più comuni 5

In questo capitolo ricapitoliamo tutti i principali comandi incontrati nella presente guida, descrivendo i loro utilizzi e alcune opzioni. I dettagli non saranno qui ripetuti.

`git init` crea un nuovo *repository* vuoto nella cartella attuale;

`git status` mostra lo stato del ramo attualmente usato, in particolare le modifiche aggiunte alla *staging area* (*changes to be committed*), i file modificati ma non aggiunti alla *staging area* (*changes not staged for commit*) e i file nuovi non ignorati (*untracked files*);

`git add` ⟨*file*⟩ aggiunge tutte le modifiche apportate a ⟨*file*⟩, un file singolo o un elenco di file separati da spazi, alla *staging area*;

`git commit` crea un nuovo *commit*. Verranno registrate solo le modifiche esplicitamente aggiunte alla *staging area*. Eventuali modifiche non presenti nella *staging area* saranno ignorate, anche se eseguite su file già presenti nel *repository*. Opzioni da ricordare:

- `git commit --amend` modifica l'ultimo *commit*del ramo attuale, non ne crea uno nuovo;
- `git commit -a` aggiunge alla *staging area* tutte le modifiche apportate ai file già presenti nel *repository* e aggiunge anche tutti i file non ancora presenti e non ignorati, infine crea un nuovo *commit*;
- `git commit -m"`⟨*oggetto*⟩`"` crea un nuovo *commit* usando ⟨*oggetto*⟩ come testo descrittivo, senza aprire un editor di testo per inserirlo.

Le opzioni possono essere usate insieme, per esempio: `git commit -am"`⟨*oggetto*⟩`"`;

`git rm` ⟨*file*⟩ rimuove ⟨*file*⟩, un file singolo o un elenco di file separati da spazi, dal *repository* e inoltre lo cancella effettivamente dalla cartella attuale;

`git log` mostra la cronologia, del ramo attuale in maniera predefinita. Se il primo argomento è il nome di un ramo (`git log` ⟨*ramo*⟩), mostra la cronologia di quel ramo; se gli ultimi argomenti sono dei file presenti nel *repository* (`git log` ⟨*file*⟩) mostra la cronologia solo di quei file. Si può visualizzare la cronologia di alcuni file in un ramo specifico con `git log` ⟨*ramo*⟩ ⟨*file*⟩. Opzioni utili:

- `git log -`⟨*n*⟩ mostra la cronologia degli ultimi ⟨*n*⟩ *commit*;
- `git log --oneline` la descrizione dei *commit* compare su una sola riga ed è composta dall'*hash* abbreviato e dalla sola prima riga del messaggio descrittivo;
- `git log -p` mostra non solo il messaggio che descrive il *commit* ma anche il `diff` delle modifiche.

`git reset` ⟨*commit*⟩ riporta lo stato del ramo attuale al ⟨*commit*⟩ specificato. Se non specificato, ⟨*commit*⟩ assume come valore predefinito `HEAD`, che corrisponde all'ultimo *commit* eseguito nel ramo. In particolare, `git reset --hard HEAD` può essere usato per rimuovere dalla *staging area* tutte le modifiche presenti, senza però cambiare i file. `git reset --hard HEAD` ⟨*file*⟩ rimuove solo ⟨*file*⟩ dalla *staging area*;

`git revert` ⟨*commit*⟩ crea un nuovo *commit* che annulla le modifiche apportate in ⟨*commit*⟩;

`git show` ⟨*commit*⟩ ⟨*file*⟩ mostra nel terminale il contenuto di ⟨*file*⟩ nella revisione corrispondente a ⟨*commit*⟩;

`git checkout` svolge alcune operazioni tecnicamente analoghe ma che è bene differenziare:

`git checkout` ⟨*commit*⟩ `--` ⟨*file*⟩ riporta il contenuto di ⟨*file*⟩, un file singolo o un elenco di file separati da spazi, alla situazione corrispondente a ⟨*commit*⟩. In particolare, `git checkout` ⟨*commit*⟩ `-- .` riporta tutti i file della cartella attuale a ⟨*commit*⟩. Quando il *commit* non è specificato, assume come valore predefinito `HEAD`, quindi `git checkout --` ⟨*file*⟩ permette di annullare tutte le modifiche apportate a ⟨*file*⟩ dopo l'ultimo *commit* e non ancora aggiunte alla *staging area*;

`git checkout` ⟨*ramo*⟩ visita il ramo chiamato ⟨*ramo*⟩. Il ramo deve essere già stato creato, per creare un nuovo ramo prima di visitarlo si può usare l'opzione `-b`: `git checkout -b` ⟨*ramo*⟩. Per far partire il nuovo ramo da una revisione precedente all'ultima bisogna aggiungere come ulteriore argomento il *commit* corrispondente: `git checkout -b` ⟨*ramo*⟩ ⟨*commit*⟩.

`git branch` ⟨*ramo*⟩ crea un nuovo ramo chiamato ⟨*ramo*⟩, ma senza visitarlo, usare `git checkout` ⟨*ramo*⟩ per fare questo. Alcune opzioni utili:

`git branch -d` ⟨*ramo*⟩ cancella il ramo chiamato ⟨*ramo*⟩ se è già stato fuso con un altro ramo;

`git branch -D` ⟨*ramo*⟩ cancella il ramo chiamato ⟨*ramo*⟩ anche se non è stato fuso con un altro ramo;

`git merge` ⟨*ramo*⟩ fonde ⟨*ramo*⟩ nel ramo attuale;

`git cherry-pick` ⟨*commit*⟩ crea nel ramo corrente un *commit* uguale a ⟨*commit*⟩, eseguito in un altro ramo;

`git tag` ⟨*etichetta*⟩ ⟨*commit*⟩ etichetta ⟨*commit*⟩ con ⟨*etichetta*⟩. L'argomento ⟨*commit*⟩ è opzionale, se assente verrà etichettata l'ultimo *commit*;

`git remote` permette di gestire i *repository* remoti. Azioni più comuni:

`git remote add` ⟨*nome*⟩ ⟨*URL*⟩ aggiunge un *repository* identificato con ⟨*nome*⟩ e con indirizzo ⟨*URL*⟩;

`git remote rename` ⟨*vecchio nome*⟩ ⟨*nuovo nome*⟩ rinomina il *repository* ⟨*vecchio nome*⟩ in ⟨*nuovo nome*⟩;

`git remote rm` ⟨*nome*⟩ rimuove il *repository* chiamato ⟨*nome*⟩;

`git remote set-url` ⟨*nome*⟩ ⟨*URL*⟩ cambia l'indirizzo del *repository* chiamato ⟨*nome*⟩ in ⟨*URL*⟩;

`git push` ⟨*remoto*⟩ ⟨*ramo*⟩ invia al *repository* ⟨*remoto*⟩ il ramo chiamato ⟨*ramo*⟩. Se non si specifica ⟨*ramo*⟩ verranno inviati tutti i rami che hanno ⟨*remoto*⟩ come *repository* remoto predefinito; se non si specifica ⟨*remoto*⟩ il ramo verrà inviato nel *repository* remoto predefinito per il ramo attuale. Le eventuali etichette create non saranno inviate in remoto se non esplicitamente richiesto con l'opzione `--tags`.

`git push -u` ⟨*remoto*⟩ ⟨*ramo*⟩ invia ⟨*ramo*⟩ al *repository* ⟨*remoto*⟩ e imposta questo come predefinito per quel ramo;

`git push --tags` ⟨*remoto*⟩ ⟨*ramo*⟩ invia ⟨*ramo*⟩ al *repository* ⟨*remoto*⟩ insieme a tutte le etichette create. Gli argomenti ⟨*remoto*⟩ e ⟨*ramo*⟩ sono sempre opzionali;

`git pull` ⟨*remoto*⟩ ⟨*ramo*⟩ aggiorna il ramo chiamato ⟨*ramo*⟩ usando il *repository* ⟨*remoto*⟩. Se non si specificano gli argomenti, i comportamenti predefiniti sono gli stessi descritti sopra per `git push`.

Bibliografia

Giacomelli, R. (2013). *Guida tematica alla riga di comando. Utilizzo della console a riga di comando*. URL http://www.guitex.org/home/images/doc/GuideGuIT/guidaconsole.pdf.

Pantieri, L. e Gordini, T. (2012). *L'arte di scrivere con LaTeX*. URL http://www.lorenzopantieri.net/LaTeX_files/ArteLaTeX.pdf.