Git

Stefano Entatti

1 Cos'è un sistema di controllo di versione

Nella realizzazione di un progetto software potresti esserti accorto di voler tenere traccia dei tuoi progressi, magari registrandoli in modo da sapere cosa hai fatto e quando, o di voler tornare al momento esattamente prima a quella modifica che non fa funzionare più nulla.

Potresti provare, ogni volta che apporti modifiche importanti, a creare una nuova copia del progetto, numerarla e magari annotarti quali cambiamenti ha portato. Man mano che il progetto avanza, le copie possono diventare moltissime e serve comunque confrontare i file riga per riga per capire cosa cambia da una copia all'altra.

Se al progetto lavorano più persone, (e quindi carichi il progetto su drive) non è comunque possibile che più persone modifichino il codice nello stesso momento, e se ognuno modifica una copia diversa, queste vanno poi unite sempre a mano con dei copia e incolla.

Un sistema di controllo di versione tiene traccia di tutte le modifiche effettuate su un insieme di file (che compongono il progetto), ovvero di chi le ha fatte, quando, quali righe sono state tolte e quali aggiunte. Ogni registrazione di modifica riporta l'autore, un titolo e una descrizione che spiega in dettaglio il perchè e stata necessaria o qualche peculiarità della modifica stessa in modo da facilitare la comprensione ad altri sviluppatori o ad una rilettura in futuro. Inoltre in questo modo è possibile navigare e riportarsi in qualsiasi punto della storia del progetto.

Per permettere a più persone di lavorare nello stesso momento ognuno copia in locale il progetto da un punto di riferimento (server comune ad esempio), e dopo aver apportato le modifiche necessarie è possibile unirle tenendo traccia di chi ha modificato cosa ed in quale ordine.

Esistono molti software di controllo versione, tra i più famosi ci sono Git, CVS, Subversion, Mercurial, Bazaar e BitKeeper.

2 Il sistema di controllo versione Git

Git è il sistema di controllo versione più utilizzato, ed è anche uno dei più semplici da usare. é nato nel 2005 ed è stato ideato da Linus Torvalds (il creatore di Linux) per facilitare lo sviluppo del kernel di Linux, uno dei progetti opensource più grandi, a cui lavorano moltissime persone. Anche git è opensource. è scritto principalmente in C e shell.

Git è **distribuito**, ciò significa che l'intero **repository**, ovvero tutti i file che compongono un progetto, tra cui sorgenti, readme e file creati da git per tenere traccia dello storico delle modifiche può essere memorizzato su un server a cui possono accedere tutti i membri del progetto. Allo stesso tempo ognuno può tenere una copia completa del del repository in locale e utilizzare tutte le funzionalità di git anche **offline**. L'interazione col server è limiata a quando lo sviluppatore desidera scaricare le modifiche effettuate da altri o vuole caricare le proprie sul server.

Il server viene spesso fornito da una **piattaforma di hosting**. Le più famose sono GitHub, GitLab, BitBuckets e Gitea. è tuttavia possibile creare un proprio server git.

Git è nato come un programma a riga di comando, e anche se viene considerato più efficiente se utilizzato in questo modo, oggi esistono molti client grafici ed estensioni che permettono di usufruire della maggior parte delle funzionalità di git direttamente dall'IDE.

3 Branching

Git è in grado di gestire contemporaneamente più sviluppi all'interno della stessa copia del progetto, questi si chiamano **branch**. Con più sviluppi si intendono più tipologie di modifiche differenti anche all'interno dello stesso file. Si può saltare da uno sviluppo all'altro senza perdere nessuna modifica evitando di avere più copie del progetto.

Ogni repository git possiede almeno un branch, chiamato **master**. Possono essere creati infiniti branch, ognuno dei quali può essere utilizzato per la realizzazione di una particolare funzionalità, in modo che modifiche su parti molto diverse del codice non possano causare conflitti. Spesso un singolo sviluppatore lavora su un proprio branch, in modo da non dover gestire anche i cambiamenti fatti da altri mentre svolge il suo compito.

Oppure il branch master può essere quello che viene distribuito agli utilizzatori del software perchè considerato stabile, mentre gli altri possono essere sperimentali e dunque contenere bug o modifiche che possono essere scartate senza compromettere tutto il resto.

A volte vari branch servono a separare versioni con funzionalità leggermente diverse dello stesso software.

Un branch viene creato a partire da un altro. Dopo la creazione di un branch "figlio" le modifiche che nel tempo vengono apportate su questo rimangono completamente separate da quelle che eventualmente continuano ad essere apportate sul "padre". Se vengono creati più branch, almeno uno di questi discende da master.

Ovviamente, due branch possono essere uniti tramite un **merge**. Gli algoritmi di merge confrontano due branch riga per riga, manentendo quella appartenente al branch in cui è stata modificata. Se però la stessa riga è stata cambiata in entrambi i branch si crea un conflitto e git richiede all'utente di risolverlo a mano.

Si può passare in qualsiasi momento da un branch all'altro.

4 Comandi principali

Git offre moltissimi comandi, ognuno dei quali svolge un piccolo insieme di funzionalità. Ogni comando (in questa sezione tratteremo i principali) è sempre preceduto da `git`. Ognuno può accettare un diverso numero di argomenti e di opzioni (tratteremo le principali per ogni comando), spesso precedute da un singolo o da un doppio meno. Gli argomenti e le opzioni dei comandi possono essere combinati in moltissimi modi per ottenere il comportamento desiderato da git.

4.1 init, clone

Per creare un nuovo repository locale si utilizza init (stando all'interno della cartella del progetto del quale si vuole tenere traccia):

```
$ git init prova
Inizializzato repository Git vuoto in /home/user/prova/.git/
```

Per copiare un intero repository remoto si utilizza git clone. Viene scaricato solo il branch master, ma una volta clonato si può accedere a tutti gli altri

```
$ git clone https://github.com/torvalds/linux.git
```

Per utilizzare tutti gli altri comandi, occorre posizionarsi all' interno della cartella del repository, che viene creata in automatico da clone e init.

4.2 Commit

Ogni volta che si fa una certa quantità di cambiamenti è utile fare un commit, ovvero segnare un punto nello sviluppo a cui sarà sempre possibile tornare, quindi registrare e descrivere queste modifiche.

```
$ git commit
```

Questo comando aprirà l'editor di default (vedere 6) di git. Nella prima riga va scritto il titolo del commit. è una buona norma che il titolo mantenga una lunghezza massima di 72 caratteri e contenga solo nomi e verbi al presente. La seconda si lascia sempre vuota e dalla terza inizia la descrizione, che può essere molto lunga e dettagliata per spiegare in modo discorsivo cosa si è fatto, perché e se eventualmente ha causato dei problemi.

Un commit rimane sempre associato al proprio autore, riconoscibile da come ha configurato il punto 6, e all'orario in cui è stato fatto.

Tutte queste informazioni sono visibili da git log (4.7).

Se non si necessita di una descrizione si può utilizzare l'opzione -m ("message"):

```
$ git commit -m "add options page"
```

Ci sono varie teorie sulla lunghezza e il contenuto dei messaggi e delle descrizioni dei commit e su *ogni quanto* si debba committare. In genere un commit deve essere relativo ad un solo argomento e non comprendere modifiche totalmente indipendenti tra di loro. I messaggi di commit non devono essere generici (come "fix crash") altrimenti col passare del tempo sarà impossibile capire cosa si era fatto senza controlare il codice. Questo è un buon approfondimento dell' argomento.

L'opzione `-s` ("signed") aggiunge la firma dell'autore nella descrizione:

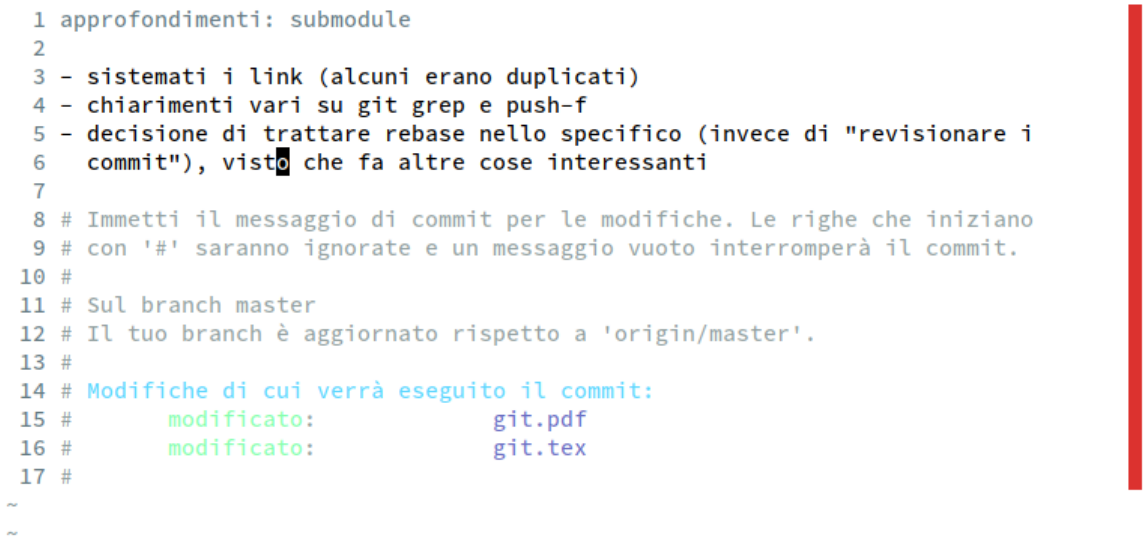

Figure 1: scrittura del testo di un commit in vim

```
Signed-off-by: Stivvo entattis15@itsvinci.com
```

Infine, può capitare di essersi dimenticati di aggiungere un file, di aver effettuato il commit troppo presto o di aver sbagliato la scrittura del messaggio. In questi casi l'opzione `--amend` permette di riscrivere il commit.

4.3 Add, reset, status

I file coinvolti dal commit devono essere prima selezionati con add. In questo modo si entra nella **staging area**. è uno stato intermedio che sta prima di un commit per tracciare le modifiche momentanee in caso di piu' prove.

Se ad esempio si modificano functions.cpp, functions.hpp e main.cpp:

```
$ git add functions.cpp functions.hpp
$ git commit -m "added get function"
```

In questo caso le moficiche di main.cpp non verranno aggiunte al commit `added get function`.

Qualsiasi modifica effettuata su `functions.cpp` o `functions.hpp` dopo l'utilizzo di add verrebbe anch'essa esclusa dal commit.

Git add si comporta in modo indifferente sia per file appena creati che per le modifiche di file già esistenti.

L'opzione -a ("all") passata al comando di commit include automaticamente tutte le modifiche attualmente pendenti.

Alcune opzioni utili per add:

- `-A` aggiunge qualsiasi modifica all'area di staging
- `.` come -A ma non aggiunge la rimozione dei file
- `-u` non aggiunge i nuovi file

Il comando `git reset` fa il contrario di add, rimuovendo dalla staging area i file o cartelle passati per argomento; invocato senza argomento li rimuove tutti. Può essere molto utile se si sono svolte contemporaneamente modifiche che si vuole dividere in commit diversi.

Per vedere quali modifiche sono nella staging area e quali invece non sono ancora state aggiunte con add:

```
$ git status
Sul branch master
Il tuo branch è aggiornato rispetto a 'origin/master'.

Modifiche di cui verrà eseguito il commit:
  (usa "git restore --staged <file>..." per rimuovere gli elementi dall'area di staging)
```

```
modificato:             git.pdf
modificato:             git.tex

Modifiche non nell'area di staging per il commit:
  (usa "git add <file>..." per aggiornare gli elementi di cui sarà eseguito il commit)
  (usa "git restore <file>..." per scartare le modifiche nella directory di lavoro)
modificato:             README.md
```

Questo comando mostra anche informazioni relative al branch su cui si è posizionati e se si è aggiornati rispetto al remote (vedere 5.3).

4.4 Push

Permette di caricare un numero illimitato di commit su un branch di un repository remoto. Le modifiche locali vengono unite a quelle remote.

Git non permette di effettuare un push se la **storia**, intesa come sequenza di commit, del branch remoto non è uguale al branch locale, escludendo i commit appena aggiunti fatti in locale. Se ci si trova in questa situazione occorre effettuare un riallineamento, (pull/rebase).

```
$ git push
Username for 'https://github.com': Stivvo
Password for 'https://Stivvo@github.com':
To https://github.com/Stivvo/msTest.git
 ! [rejected]        master -> master (fetch first)
error: push di alcuni riferimenti su 'https://github.com/Stivvo/msTest.git' non riuscito
Gli aggiornamenti sono stati rifiutati perché il remoto contiene delle
modifiche che non hai localmente. Ciò solitamente è causato da un push
da un altro repository allo stesso riferimento. Potresti voler integrare
le modifiche remote (ad es. con 'git pull ...') prima di eseguire
nuovamente il push.
Vedi la 'Nota sui fast forward' in 'git push --help' per ulteriori
dettagli
```

4.5 Fetch, merge, pull

Fetch permette di aggiornare lo stato dei branch in remoto per controllore se ci sono branch nuovi o magari nuovi commit sul branch al quale si sta lavorando per evitare di rimanere disallineati con il repository di riferimento. Fetch da solo tuttavia non modifica mai alcu file sul repository locale.

Merge permette di fondere le modifiche di due branch, locali o remoti, o commit diversi dello stesso branch. Si usa se ad esempio la versione locale è rimasta indietro rispetto a quella remota. oppure quando bisogna unire su un branch le modifiche fatte su un altro. Tuttavia, merge non preleva mai nulla dal repository remoto, per farlo occorre effettuare prima un fetch.

Pull è in sostanza un fetch seguito da un merge, ed è quello che capita di utilzzare più spesso, anche se la combinazione fetch e merge sarebbe sempre un alternativa più sicura. In generale pull fonde ciò che si trova al momento sul branch del repository remoto con quello locale. Se si effettua un pull come suggerito nel codice della sezione 4.4 avverrà infatti un merge.

Pull e fetch e merge chiamati senza argomenti vanno a prelevare la versione remota del branch su cui si è localmente, ma nel caso di merge è aggiornata alla all' ultimo fetch.

Quindi se si vuole fondere ad esempio le modifiche di develop su master:

```
$ git checkout master
$ git fetch
$ git merge develop
```

Questo permette di vedere prima le nuove modifiche remote: dopo l'utilizzo di fetch, si possono guardare al volo facendo ad esempio `git checkout origin/develop` (si entra in deatached head 5.1). Poi si può decidere se effettuare il merge o no, oppure di effettuare il merge in assenza di internet (in casi estremi le modifiche solitamente prelevate con fetch potrebbero venire da un altro disco).

```
$ git checkout master
$ git pull develop
```

Utilizzando pull si ottiene un risultato identico ma l'operazione di merge non è annullabile. In ogni caso è sempre necessario non avere modifiche non committate quando si effettua pull o merge.

```
<<<<<<< HEAD
class FirstClass {
=======
class SecondClass {
>>>>>>> 4ceb8e7c4fe78b59c00be99418f54280df19078c
```

Questo è il caso in cui mentre il repository locale rimaneva indietro di alcuni commit la classe è stata rinominata in FirstClass da un primo sviluppatore. Nel frattempo un secondo ha fatto un push di un commit in cui l'ha chiamata SecondClass. Quando il primo sviluppatore si trova a dover fare il pull delle modifiche del secondo, si creano dei conflitti di merge, e git obbliga l'utente a risolverli. Per farlo, deve scegliere tra la versione locale (HEAD, vedere 5.1) e quella dell'altro, identificata dal codice hash (vedere 4.7) del commit che ha rinominato la classe in SecondClass. Il prossimo push sarà quello del commit di merge, automaticamente creato da git.

4.6 Checkout, branch

Git branch senza opzioni viene utilizzato per creare un nuovo branch locale. La creazione del branch develop:

```
$ git branch develop
```

Per posizionarsi su develop:

```
$ git checkout develop
M git.pdf
M git.tex
Si è passati al branch 'develop'
```

Se si prova a eseguire un push dal branch appena creato occorre aggiungerlo alla lista dei branch remoti:

```
fatal: Il branch corrente develop non ha alcun branch upstream.
Per eseguire il push del branch corrente ed impostare il remoto come upstream, usa

    git push --set-upstream origin develop
```

L'opzione –all mostra tutti i branch locali e remoti. Il branch seguito dall'asterisco è quello su cui si è posizionati correntemtente.

```
$ git branch --all
* develop
  master
  temp
  remotes/origin/HEAD -> origin/master
  remotes/origin/develop
  remotes/origin/master
```

In questo caso, temp è solamente locale.
L'opzione -d invece elimina un branch. Non è possibile eliminare il branch corrente:

```
$ git branch -d develop
error: Impossibile eliminare il branch 'develop' di cui è stato eseguito
il checkout in '/home/stefano/prog/GitNoob2ProIta'
```

Per eliminare lo stesso branch anche dal repository remoto:

```
$ git push -d origin develop
```

L'opzione -b di checkout crea un branch se quello passato come parametro non esiste, utilizzando quindi prima un git branch e poi un git checkout.
Per fondere due branch si utilizza ovviamente merge (4.5).

```
commit 089be30068ddf7d8b415bf20008f6273d6781b9f (HEAD -> master, origin/master, origin/HEAD)
Author: Stivvo <entattis15@itisvinci.com>
Date:   Thu Feb 27 10:12:40 2020 +0100

    git su windows

commit 4118350ac8ccebffe015be7e207022b8d33dcb55
Author: Stivvo <entattis15@itisvinci.com>
Date:   Wed Feb 26 22:28:05 2020 +0100

    link su revert e reset; fetch e pull

commit 12dab022c373ab6d0f008bb62fdc280b9b8fd400
Author: Stivvo <entattis15@itisvinci.com>
Date:   Wed Feb 26 22:21:02 2020 +0100

    chiarimenti in diff

commit e6c98b42fcb220cdb33c4afb6dbba7465d748bfe
Author: Stivvo <entattis15@itisvinci.com>
Date:   Wed Feb 26 22:15:12 2020 +0100

    commit --amend

commit 64529c1d36ac04cbe750bd9dce9bdb0d11c4f2b8
Author: Stivvo <entattis15@itisvinci.com>
Date:   Wed Feb 26 22:03:18 2020 +0100

    ordine

    - la notazione <branch>~<numero di commit>
    - nella sezione dei branch non si tenta più di spiegare il merge
    - revert non è un pull senza merge ma un merge senza merge

commit 7c807655935a417b0a493b89bf0a74f4a6081a6b
Author: Stivvo <entattis15@itisvinci.com>
Date:   Wed Feb 26 21:17:37 2020 +0100

    Ultimi chiarimenti su fetch, pull e merge

    - fetch viene prima di merge, perchè in merge devo spiegare che guarda i
      dati scaricati dall'ultimo fetch
    - perchè è importante dividere fetch e merge (e l'assenza di internet)
```

Figure 2: l'output del comando git log sul pager less

4.7 Log

Molto di quanto appena spiegato sarebbe inutile se non si potesse vedere la storia dei commit.

Git log mostra l'intera storia dei commit visualizzata nel pager di default (vedere 6). Le lunghe serie di caratteri sono i codici **hash**, univoci per ogni commit. L'output del comando mostra anche a quale commit puntano la HEAD e i repository remoti (se abbiamo dei commit di cui non abbiamo fatto ancora il push è probabile che quest'ultima sia più indietro rispetto ad HEAD).

Questo comando può generare molto output. Sarà più semplice trovare un commit utilizzando l'opzione `pretty=oneline`, che assegna una sola riga ad ogni commit. Dopodichè sarà utile passare l'hash del commit trovato come argomento di log, per vedere informazioni puù dettagliate. L'output di log escluderà semplicemente tutti i commit precedenti a quello.

è anche possibile ricercare il testo del commit interessato passandolo come argomento di log subito dopo `--grep=` (grep è un altro tool unix utilizzato da git):

```
$ git log --grep='git log'
commit 4b64be5218bed736d357d61471b87c4f5363d954 (HEAD -> master, origin/master, origin/HEAD)
Author: Stivvo <entattis15@itisvinci.com>
Date:   Sun Feb 23 17:37:09 2020 +0100

    git log
```

Si può ottenere la lista dei commit in cui è stata aggiunta o rimossa una determinata stringa all'interno dei file del repository passandola come parametro dopo l'opzione `-S` ("string").

Se invece si è interessati a vedere tutti i commit effettuati da una stessa persona:

```
$ git log --author="Stivvo"
```

Se si vuole vedere rapidamente i titoli di tutti i commit senza il loro hash raggruppati per autore può tornare utile `git shortlog`.

4.8 Diff

Diff è un programma presente in tutti i sistemi unix-like che ha il semplice compito di dare in output tutte le linee che differiscono tra due file, precedute da un ¡ se quella determinata riga è presente solo file passato come primo argomento, altrimenti ¿.

```
$ cat file1
prima
seconda
$ cat file2
prima
terza
$ diff file1.txt file2.txt
< seconda
> terza
```

Git utilizza una propria versione di diff, ad esempio per effettuare i merge, che mette anche a disposizione dell'utente. Aggiunge la capacità di non trattare un file rinominato o spostato come un nuovo file e di comparare la differenze tra i commit.

```
$ git diff 1fd15c30db68c1d9826204f571e4053a5ed89b49 9b004eae46dca7525156f57b9cf048ab147dd67d
```

Visualizza le modifiche apportate tra due commit qualsiasi (compresi quelli in mezzo). Se si specifica un solo commit, si compara ad HEAD di default.

```
$ git diff --staged
```

`--staged` mostra tutte le modifiche aggiunte alla staging area rispetto all' ultimo commit.

```
$ git diff HEAD
```

Questo mostra anche le modifiche che non sono neanche entrate nella staging area.
Un modo comodo per vedere quali modifiche si sono introdotte con i commit che si stanno per pushare su master:

```
$ git diff origin/master
```

L'output di diff può essere ristretto ad uno o più file passati sempre come ultimi argomento.
Diff diventa ancora più utile quando utilizzato insieme a log:

```
$ git log -p
```

Mostra tutti possibili output di diff, separandoli per commit. L'output può essere ovviamente ristretto ad uno o più file passati come argomento.
Infine il comando `git show` mostra tutte le mofidiche introdotte con un commit passato come parametro (se non presente mostra HEAD di default), i file modificati utilizzando diff e il testo del commit utilizzando log.

5 la cartella .git

La cartella .git si trova nella root del repository e contiene tutti i file utilizzati da git, tra cui informazioni sui branch, sui commit. Nei sistemi operativi unix una cartella preceduta da un punto è nascosta e quindi occorre utilizzare il parametro -a di ls per poterla vedere.

```
$ ls .git/
branches/  COMMIT_EDITMSG  config  description  FETCH_HEAD  HEAD  hooks/  index
info/  logs/  objects/  ORIG_HEAD  packed-refs  refs/
```

è molto importante la cartella refs:

```
$ ls .git/refs
heads/  remotes/  tags/
```

Dietro a questi tre nomi ci sono concetti importanti di git che tornano utili con la maggior parte dei comandi

5.1 heads

In git una head è un riferimento ad un branch o ad un commit di un determinato branch, locale o remoto. Una lista delle head disponibili:

```
$ ls .git/refs/heads
master/ develop/
```

HEAD è un file che punta all'ultimo commit del branch in cui si è attualmente posizionati nel repository locale.

```
$ cat .git/HEAD
ref: refs/heads/master
```

Nel caso in cui si voglia ritornare ad un commit precedente si entra nello stato di ***deatached head***, ovvero facendo il checkout su uno specifico commit (identificato con il suo codice hash).

```
$ git checkout 8b10ce361a08e03179d46bab5d691148805bf8d8
Nota: eseguo il checkout di '8b10ce361a08e03179d46bab5d691148805bf8d8'.

Sei nello stato 'HEAD scollegato'. Puoi dare un'occhiata, apportare modifiche
sperimentali ed eseguirne il commit, e puoi scartare qualunque commit eseguito
in questo stato senza che ciò abbia alcuna influenza sugli altri branch tornando
su un branch.

Se vuoi creare un nuovo branch per mantenere i commit creati, puoi farlo
(ora o in seguito) usando l'opzione -c con il comando switch. Ad esempio:
```

```
  git switch -c <nome nuovo branch>

Oppure puoi annullare quest'operazione con:

  git switch -

HEAD si trova ora a b0451d9 immagine scrittura commit
```

Al posto di andare a recuperare l'hash del commit:

```
$ git checkout master~2
```

Il numero dopo il caratterie `~` indica di quanti commit si deve tornare indietro per trovare il commit su cui effettaure il checkout. è un eccezzione il fatto che `~` e `~1` siano equivalenti.

Se si vuole mantenere i commit fatti in questo stato è buona cosa spostarsi su un nuovo branch come suggerito.

Se si sceglie di rimanere sullo stesso, non si può effettuare direttamente il push dei commit effettuati in questo stato, perché non si è di fatto posizionati su nessun branch:

```
$ git push
fatal: Attualmente non sei su un branch.
Per eseguire ora il push della cronologia che ha condotto
allo stato corrente (HEAD scollegato), usa

    git push origin HEAD:<nome del branch remoto>
```

Il comando suggerito da git serve per caricare le modifiche effettuate in deatached head direttamente sul branch remoto, come spiegato nella sezione 5.3. è molto probabile che non funzioni, perchè andrebbe ad eliminare delle modifiche remote successive al commit in cui ci si è posizionati; è necessario aggiungere l'opzione -f ("force") a push se si vuole eliminarle.

Questo non risolve lo stato di deatached head. Occorre infatti ritornare sul proprio branch (in questo caso master), che però contiene ancora i commit che vogliamo eliminare: un push li riporterebbe sul repository remoto. Quindi dopo aver fatto `git checkout master`, tornando sul branch da cui ci si era distaccati, si può utilizzare **reset**, che è come un merge forzato che invece di fondere le moficiche sovrascrive il branch remoto su quello locale:

```
$ git reset --hard origin/master
```

Oppure si può sempre clonare nuovamente il progetto, ma è sempre la soluzione peggiore. Sia in questo modo che utilizzando reset c'è sempre il pericolo di eliminare qualcosa che invece si voleva tenere, perchè si cancellano dei commit o delle modifiche non committate.

Per questo esiste un modo migliore per ritornare a un commit precedente, senza modificare i commit già effettuati:

```
$ git revert 0552dd1c6e3c11c8c5246836e9994e6fcd431a0f..HEAD
$ git commit -m "torno al commit precedente"
```

In questo modo il commit `torno al commit precedente` conterrà delle moficiche che riportano allo stato del commit di cui si è specificato il codice come argomento di **revert**. `..HEAD` indica che si ripristinano le modifiche effettuate da quel commit fino a HEAD (questo intervallo può dunque comprendere diversi commit), ovvero lo stato corrente del branch.

5.2 Tags

Possono essere assegnati ad un commit in cui si è raggiunto un traguardo nel progetto (ad esempio 1.3.5).

Per visualizzare i tag:

```
$ git tag
v1.0
v2.0
```

Oppure:

```
$ ls .git/refs/tags
v1.0 v2.0
```

Per creare un tag basta dare al comando un argomento, che sarà il nome del tag:

```
$ git tag v2.1
```

I tag possono essere utilizzati in questo modo per descrivere piccoli progressi nello sviluppo. Per segnare il punto di una release è bene utilizzare l'opzione -a ("annotated"):

```
$ git tag -a v3.0
```

In questo modo, verrà aperto l'editor di default per poter inserire informazioni su ciò che le novità portate da quella release. Come per i commit, l'opzione -m permette di scrivere un breve titolo senza aprire l'editor.

Per visualzzare queste informazioni:

```
$ git show v3.0
commit ca82a6dff817ec66f44342007202690a93763949
Author: User <user@email.com>
Date:   Mon Mar 17 21:52:11 2020 -0700

    new release 3.0!
```

L'output è molto simile a quello di `git log`. Se si vuole vedere le descrizioni di tutti i tag basta chiamare `git show` senza argomenti.

Si può aggiungere un tag ad un qualsiasi commit precedente specificando il suo codice:

```
$ git tag -a v1.2 9fceb02
```

I tag possono anche essere eliminati con l'opzione -d e si può fare il checkout su uno specifico tag come si fa con i commit. Anche sull' assegnazione dei nomi alle versioni ci sono norme da seguire.

5.3 Remotes

Un remote è il percorso di un repository remoto, di solito è quello del repository sul server. Il primo remote che viene utilizzato in un repository viene chiamato di default **origin** (non è obbligatorio). I remote infatti possono essere aggiunti, rinominati o rimossi:

```
$ git remote add amanjot https://github.com/samanjot/GitNoob2Pro
$ git remote rename amanjot samanjot
$ git remote remove samanjot
$ git remote add amanjot https://github.com/samanjot/GitNoob2Pro
```

Con l'opzione `-v` ("verbose") otteniamo una lista dei remote disponibili. Di default le operazioni come pull sottointendono che si voglia utilizzare il remote origin, ma l'operazione può essere eseguita su qualsiasi altro remote (ad esempio `git pull amanjot master`).

```
$ git remote -v
amanjot https://github.com/samanjot/GitNoob2Pro (fetch)
amanjot https://github.com/samanjot/GitNoob2Pro (push)
origin https://github.com/Stivvo/GitNoob2Pro.git (fetch)
origin https://github.com/Stivvo/GitNoob2Pro.git (push)
```

oppure:

```
$ ls .git/refs/remotes
origin/ amanjot/
```

Il remote amanjot è un fork (7.2) del repository di questa dispensa.

6 Installazione e configurazione

L'installazione di Git su Linux avviene dal package manager della propria distribuzione eed è un processo che richiede pochi secondi. Su Windows, occorre scaricare un wizard che installa un emulatore di una versione minimale di linux che comprende tool come bash e openssh, da cui Git dipende.

```
$ git config --global user.name Stivvo
$ git config --global user.email entattis15@itsvinci.com
```

In questo modo si imposta lo username e l'email che verranno associati a tutti i futuri commit. Non sono da confondere con le credenziali di Github 7. Per impostare editor e pager utilizzati git (di solito vim e less sono già impostati di default):

```
$ git config --global core.editor vim
$ git config --global core.pager less
```

Un pager è un programma a linea di comando pensato per visualizzare in modo comodo l'output di qualsiasi programma o il contenuto di un file, permettendo di effettuare lo scroll avanti e indietro e ricerche sul testo. Se invece si preferisce visualizzare sempre l'output sul terminale:

```
$ git config --global core.pager ""
```

Per utilizzare comunque less per visualizzare l'output di git log sarebbe necessario:

```
$ git log --color=always | less -r
```

Scrivere `git` davanti ad ogni comando può risultare scomodo. Esiste un modo per evitarlo, sfruttando le funzionalità di alias messe a disposizione dalla shell. Per impostarli occorre modificare il file di configurazione della propria shell, `~/.bashrc` se si utilizza bash. Questo è un esempio.

7 Github

Github è la più famosa piattaforma di hosting Git al mondo. è estremamente utilizzato da software **opensource** ma si rivolge anche al mondo closed source attraverso github enterprise, che è ovviamente a pagamento, con il quale ci si può affidare ai server di Github oppure installarlo su uno proprio.

Per utilizzare Github è necessario registrarsi. Utente e password scelti verranno utilizzati richiesti al push su un qualsiasi repository Github e anche al pull o clone di un repository privato.

Conoscendo Git, l' interfaccia del sito diventa presto molto facile da usare. A volte può risultare più comodo svolgere alcune operazioni sul sito piuttosto che sul terminale. Ci sono comunque alcune funzionalità extra che vanno comprese, perchè si basano su concetti importanti spesso collegati a Git che non sono stati ancora trattati.

7.1 Readme

Il readme è la prima cosa con cui un visitatore di un repository pubblico viene a contatto. Nel readme si scrive che cosa contiene il repository, come si utilizza il software, si forniscono istruzioni per chi vuole contribuire o compilare il software da sorgente. è scritto in markdown, un metalinguaggio utilizzato per scrivere testi che viene spesso compilato in file .html.

7.2 Fork

Un gruppo di sviluppatori effettua il fork di un software quando ne crea una copia per continuarne lo sviluppo in modo indipendente dal team originale. Il concetto è simile a quello della creazione di un branch, che diventa però indipendente da tutti gli altri e viene sviluppato da persone diverse. Spesso un fork porta un nome diverso dal progetto originale.

Come si potrà immaginare, i fork nascono, la maggior parte delle volte, da progetti opensource, di cui è perfettamente lecito creare quanti fork si desiderino. Molte volte i fork avvengono quando il contributo di un gruppo di persone ad un progetto non viene accettato dal team originale, ad esempio perchè questi hanno obiettivi molto diversi per stesso software o perchè lo vogliono adattare a specifiche esigenze. Anche i fork di fork non sono così inusuali nel mondo opensource.

Un fork può anche essere effettuato da uno sviluppatore che vuole contribuire ad un progetto pur non essendo collaboratore. Non può quindi apportare direttamente delle modifiche e dunque nemmeno creare un proprio branch. Quando all'interno del fork ha terminato di effettuare le proprie modifiche, può ricongiungersi col progetto principale attraverso una pull request (7.3).

Due fork possono in teoria essere sottoposti ad un merge. A volte però la differenze aumentano tanto che diventa possibile effettuarlo solamente sulle parti che sono state poco modificate dagli autori del fork.

README.md

Qt Creator

Qt Creator is a cross-platform IDE for development with the Qt framework.

Supported Platforms

The standalone binary packages support the following platforms:

- Windows 7 or later
- (K)Ubuntu Linux 16.04 (64-bit) or later
- macOS 10.13 or later

Contributing

For instructions on how to set up the Qt Creator repository to contribute patches back to Qt Creator, please check:

https://wiki.qt.io/Setting_up_Gerrit

See the following page for information about our coding standard:

https://doc-snapshots.qt.io/qtcreator-extending/coding-style.html

Compiling Qt Creator

Prerequisites:

- Qt 5.11.0 or later
- Qt WebEngine module for QtWebEngine based help viewer
- On Windows:
  - ActiveState Active Perl
  - MinGW with g++ 5.3 or Visual Studio 2017 or later
  - jom
  - Python 3.5 or later (optional, needed for the python enabled debug helper)
- On Mac OS X: latest Xcode
- On Linux: g++ 5.3 or later

Figure 3: il readme di qt creator, uno dei maggiori progetti opensource 7.1

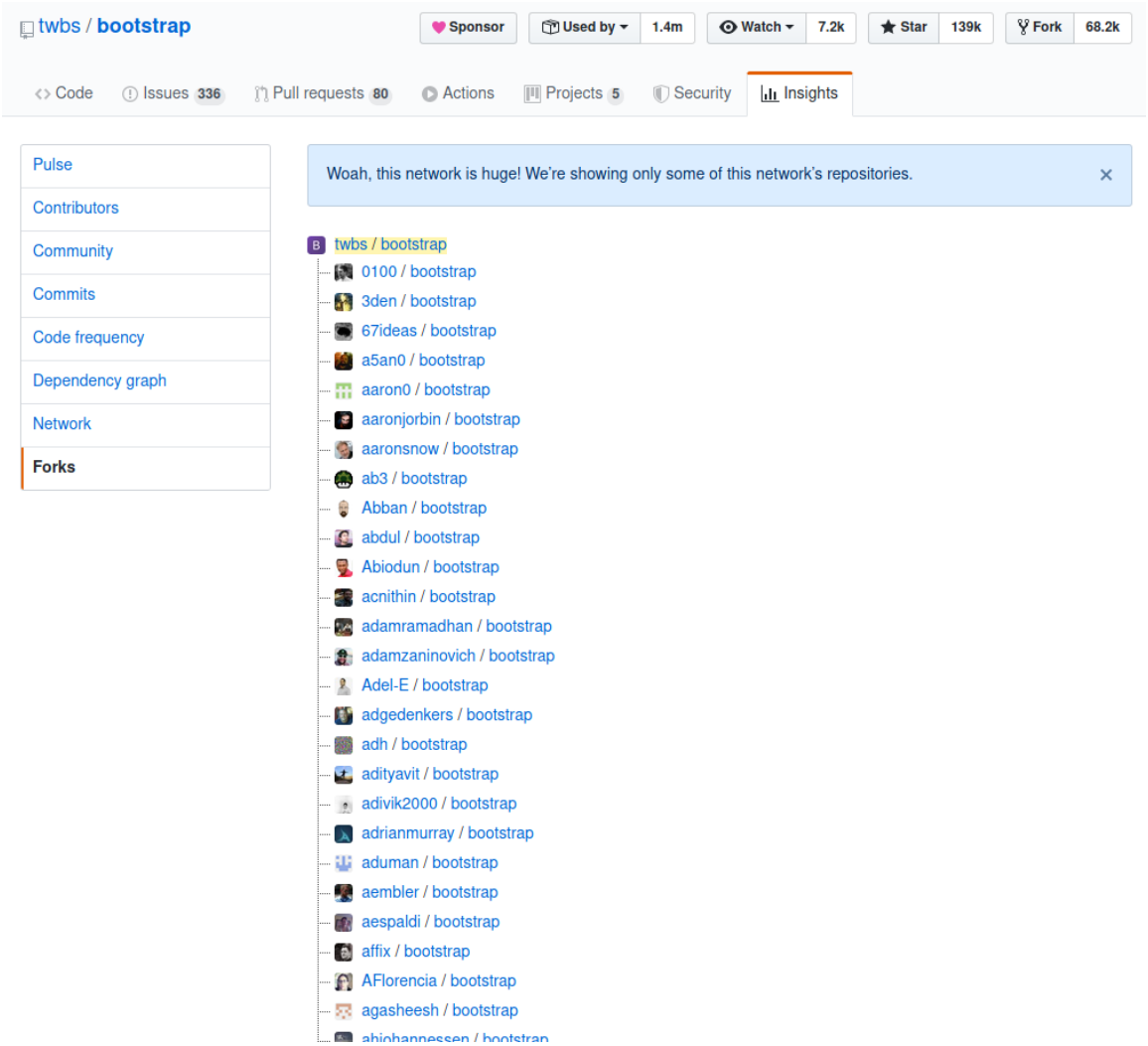

Figure 4: l'albero dei fork di Boost, libreria per HTML/CSS/JS, uno dei repository attualmente più forkati su GitHub 7.2

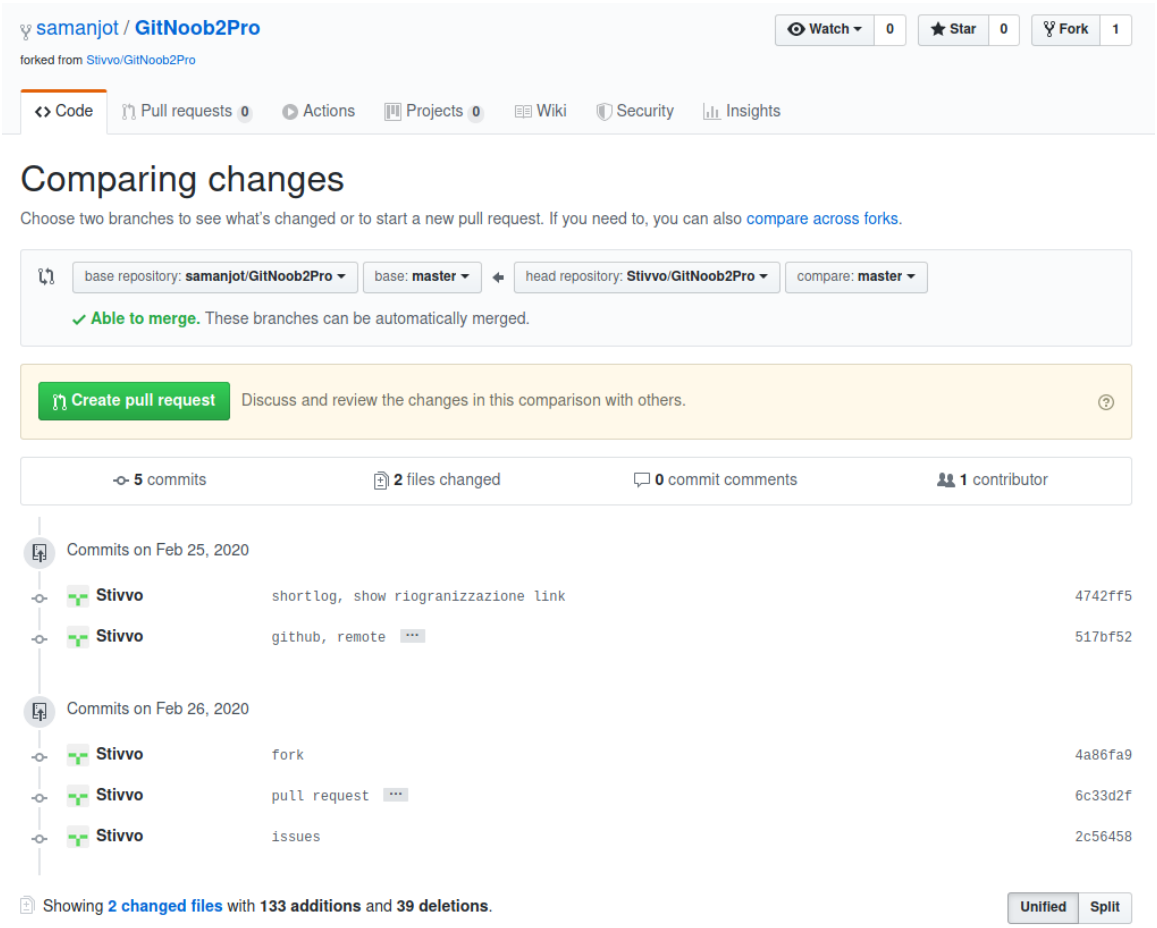

Figure 5: una pullrequest per aggiornare il fork di questa dispensa 7.3

7.3 Pull request

Una pull request è di base un merge tra branch, accompagnato da un testo che spiega quali sono i cambiamenti che da un determinato branch si vogliono unire a ad un altro, dello stesso repository o di un fork. Riporta anche tutti i commit che coinvolge, in modo che oltre al testo sia facilmente comprensibile quali modifiche verranno integrate. La differenza principale rispetto ad un merge è che una pull request deve essere **approvata** da un collaboratore del progetto del repository su cui viene poi effettuato il merge.

Le pull request possono essere effettuate ad esempio da un singolo sviluppatore per chiedere ai propri collaboratori se può integrare su master le modifiche che ha effettuato sul proprio branch. Se, mentre viene valutata, continua a lavorare sullo stesso branch, la pull request già fatta non viene aggiornata (è necessario crearne un'altra), in questo modo chi controlla le pull request può prendersi molto più tempo per valutarle, essendo sicuro che nel frattempo non possono cambiare.

Le pull request potrebbero sostituire i merge tra branch, tuttavia se si tratta di un' operazione che non richiede l' approvazione di altri è meglio utilizzare solamente i comandi di git perchè si andrebbe a creare confusione nello storico delle pull request.

7.4 Issue

Una issue non ha a che vedere direttamente con i comandi di Git, è invece un servizio aggiunto dalle piattaforme di hosting. Contengono un testo che spiega un problema o una feature che si vorrebbe introdurre e possono essere aperte da chiunque. A volte vengono aperte dagli stessi sviluppatori del progetto, per cui diventano una specie di todolist, o anche semplici utenti che vogliono segnalare un bug. Quindi, se avete delle proposte o trovate degli errori su questa dispensa non esitate.

La grande utilità delle issue (questo è specifico per GitHub) è che possono essere collegate a commit o pull request. Se ad esempio un commit risolve definitivamente i problemi evidenziati da una specifica issue, può essere collegato alla sua chiusura. Inserendo alcune parole chiave nel titolo del commit, seguite da # e poi dal numero della issue, la chiude in automatico.

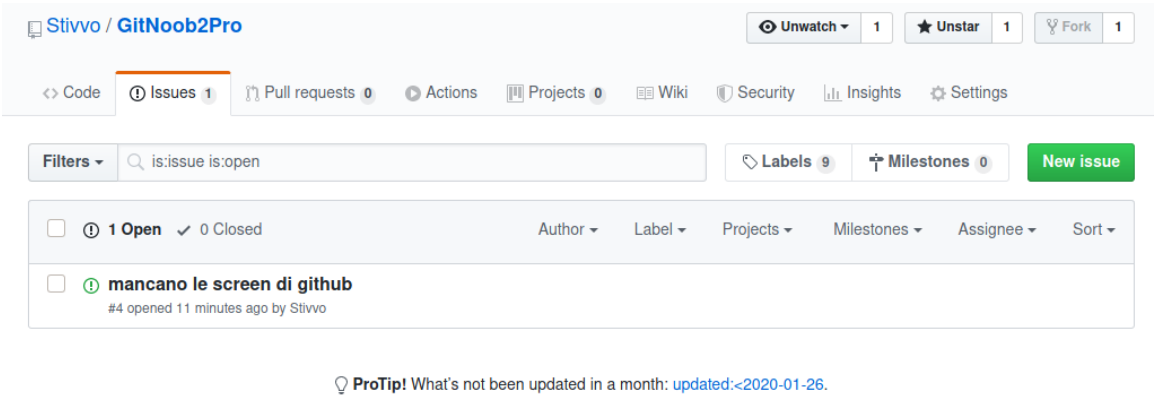

Figure 6: lista delle issue attualmente aperte su questo repository 7.4

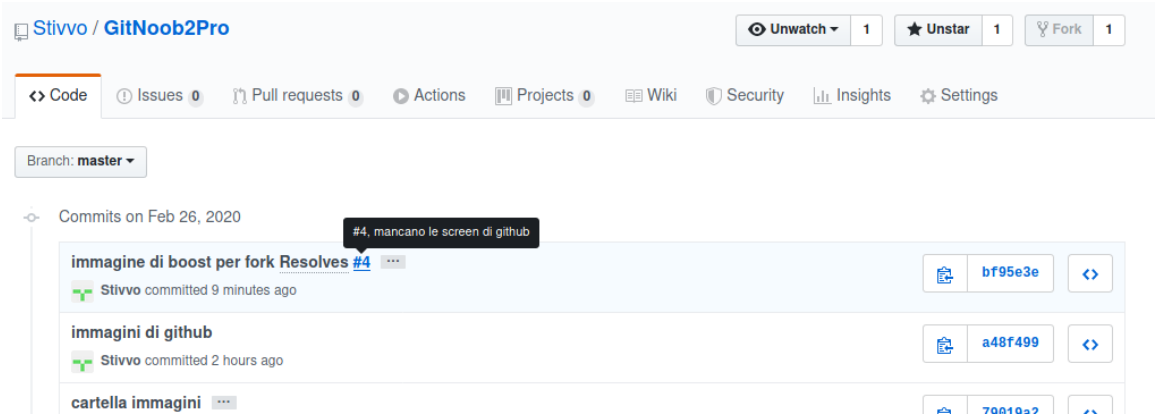

Figure 7: commit che chiude una issue 7.4

Inoltre, nella storia dei commit di github quel commit riporterà un link alla pullrequest che chiude. Una pullrequest chiusa infatti viene solo marcata come tale, ma resta visibile per preservare la memoria dello sviluppo.

8 Approfondimenti

8.1 I submodule

Un caso d'uso dei submodule è quello in cui si vuole utilizzare nel proprio progetto una libreria opensource. Si potrebbe copiare sigolarmente i file necessari oppure installare la libreria sul proprio computer ma spesso l'utilizzo dei submodule risulta più appropriato. L'utilizzo dei submodule permette di integrare un intero repository git all' interno di un altro. In questo modo c'è una separazione netta di quali file appartengono ad uno e quali all' altro, è possibile avere facilmente gli aggiornamenti del submodule e quando una persona qualsiasi effettua il clone del repository non si deve preoccupare di installare la libreria a parte. Infine, è un modo per dare il giusto credito a chi ha creato il repository che si integra come submodule (github ad esempio genera automaticamente il link che riporta a quel repository, rendendo noto a chiunque chi ha fatto che cosa).

Per clonare un repository e tutti i suoi submodule:

```
$ git clone --recursive
```

Per aggiungere questa dispensa come submodule:

```
$ git submodule add https://github.com/Stivvo/GitNoob2Pro
```

Per aggiornare i submodule occorre andare nella loro cartella e aggiornarli alle modifiche remote come se il proprio repository (quindi con pull ecc). è bene eseguire un commit in cui si effettua solamente l'aggiornamento dei submodule per non confonderlo con le effettive modifiche del proprio repository.

Da notare come viene aggiornato il file `.gitmodules`, da cui è possibile fare ulteriori modifiche ai submodule.

8.2 Rebase

Rebase ha lo stesso compito di merge, ma lo svolge in modo molto diverso. Quando si effettua un merge tra due branch, i commit vengono ordinati per data. Se sono state fatte modifiche in entrambi i branch su cui si effettua il merge, si avrà come risultato che i commit di uno e dell'altro saranno mischiati. Rebase risolve questo problema, perchè sposta semplicemente un intervallo di commit da un punto all'altro.

Per esempio, si vuole portare i commit di develop su master:

```
-A-B-C-D master
   \
    G-H-I develop
```

Il risultato utilizzando merge:

```
-A-B-C-D- master
   \    \
    G-H-I
```

E con rebase:

```
-A-B-C-D
       \
        G-H-I master
```

In questo caso si sposta la *base* dei commit da G a I, che non è più B ma D, proprio perchè è stato cambiato il loro posto all'interno dello storico per copiarli su un nuovo branch.

Rebase offre anche la possibilità di riscrivere interamente la storia dei commit quando si uniscono due branch, utilizzando l'opzione `-i` ("interactive"'). In questo modo si aprirà l' editor di default con la lsita di tutti i commit interessati dal rebase (figura 8), per ognuno di questi si può scegliere ad esempio di cambiare il contenuto del suo messaggio o unirlo a quello precedente. Dopodichè vengono analizzati uno per volta, a partire dal meno recente. Se sono stati selezionati viene aperto l'editor di default con cui è possibile modificare il messaggio di commit. Quest'ultima opzione è molto utile perchè permette di ripulire lo storico dei commit.

```
$ git checkout master
$ git rebase -i develop
```

Questo comando tenta di spostare la *base* di develop, ovvero il punto in cui si è staccato da master, sull'ultimo commit di master, il branch in cui si è attualmente posizionati. Nel processo, i commit presi in considerazione dal rebase vengono analizati uno ad uno a partire dal meno recente e possono essere modificati, uniti ecc prima di essere effettivamente copiati su master perchè è stata aggiunta l'opzione `-i`.

Se si tiene ad avere uno storico dei commit ordinato, capiterà più spesso di utilizzare questo comando su un solo branch per modificare il suo storico dei commit.

```
$ git rebase -i HEAD~4
```

Quello che si ottiene in questo modo è prendere gli ultimi 4 commit di master e riposizionarli nello stesso punto infatti lo stesso comando senza `-i` non farebbe assolutamente nulla. Mettere il nome del branch corrente al posto di HEAD sarebbe indifferente. Si può utilizzare anche con branch diversi.

Alla fine di ogni rebase non si potrà effettuare direttamente il push: alcuni commit remoti che non sono più presenti in locale perchè modificati o eliminati andrebbero persi per sempre. `push -f` forza la loro cancellazione.

In conclusione, merge rappresenta la vera storia dello sviluppo, anche se questa è spesso confusionaria perchè fatta dai commit appartenenti a vari branch che si incrociano, scritti male o inutili (compresi quelli di merge). Rebase permette invece di effettuare intere revisioni della storia in modo da renderla più chiara e leggibile.

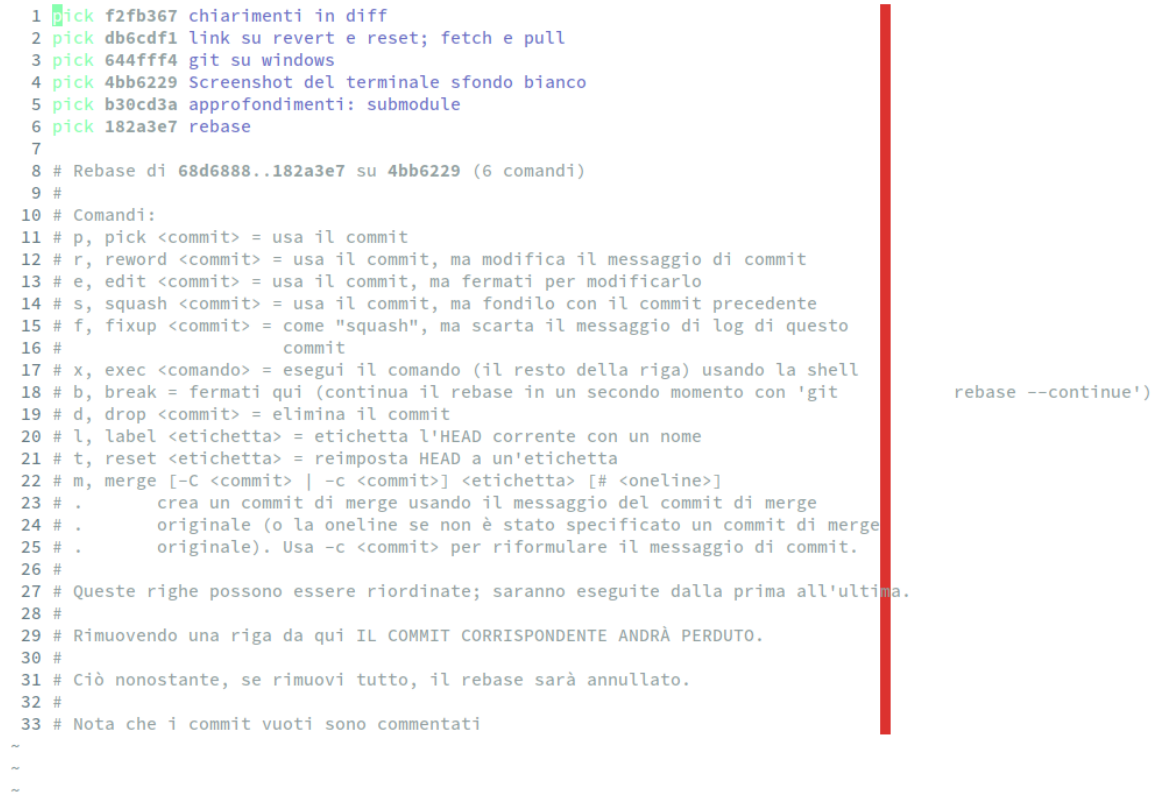

Figure 8: il file per selezionare i commit da modificare prima di un rebase

8.3 Gitignore

Il file .gitignore, posizionato nella root del repository, permette di selezionare file o cartelle di cui git non deve tenere traccia. Esiste una raccolta di file .gitignore per la magggior parte di linguaggi e IDE. Alcuni esempi sono:

- i file .swp, utilizzati da vim come cache per i file aperti
- i file .user, creati dall'IDE qtcreator all'apertura di un progetto per memorizzare quali compilatori sono disponibili sulla macchina
- ogni tipo di eseguibile, vengono ricavati con la compliazione

Il gitignore del repository di questa dispensa esclude file generati dal compilatore latex. Così facendo, si ottiene uno storico dei commit più pulito perchè questi file verrebbero costantemente segnati come modificati.
un ulteriore esempio:

```
*.exe
*.dll
cartellaDaEsclidere/*
Makefile
```

9 fonti, link utuli

9.1 generale

- questa stessa dispensa su Github
- benefici dei version controllo
- vantaggi di git

- pro git (libro completo)
- tutti i comandi

9.2 commit, push, add

- differenza tra pull e fetch
- annullare add
- parametri di add
- sovrascrivere le modifiche locali con quelle remote
- differenza tra fetch e pull
- tornare a commit precedenti
- differenza tra revert e reset

9.3 head, remotes, branch

- checkout di un branch remoto
- eliminare un branch
- rinominare un branch
- cos'è head
- che cos'è origin
- deatached head
- risolvere una deatached head
- differenza tra head, master e origin
- tipi di head
- push origin head
- uscire da deatached head
- diff HEAD vs diff –staged
- che cos'è un remote
- fare il merge di due branch
- differenza tra origin master e origin/master

9.4 merge, rebase

- modificare commit esistenti
- differenza tra merge e pull
- git rebase
- merge vs rebase
- differenza tra merge e rebase
- aggiornare i submodule

9.5 log, diff

- guida a git log
- cercare il messaggio dei commit
- cercare il testo modificato dai commit
- grep del testo modificato dai commit
- git log sempre colorato
- ottenere tutti i commit di uno stesso autore
- storia di un file. log -p
- diff relativo ad un file
- non utilizzare un pager