Alma Mater Studiorum · Università di Bologna

Scuola di Ingegneria e Architettura
Dipartimento di Informatica · Scienza e Ingegneria · DISI
Corso di Laurea in Ingegneria Informatica

DATABASE A GRAFI PER SISTEMA DI CONTROLLO DELLE VERSIONI

Relatore:
Prof. Paolo Ciaccia

Presentata da:
Riccardo Barbieri

Anno Accademico 2021/2022

Elenco delle figure

Indice

Introduzione

Un VCS (Version Control System) è un tipo di sistema utile a gestire le versioni di un progetto. Nell'ambito dello sviluppo di progetti software, la funzionalità principale di un sistema di questo tipo è la possibilità di tenere traccia delle modifiche apportate al codice sorgente, ai file di documentazione e ad altri contenuti di un progetto.
Senza l'utilizzo di un VCS, gli sviluppatori di un progetto potrebbero semplicemente mantenere diverse copie dei contenuti, classificandole a seconda della versione e delle modifiche apportate da una versione all'altra; un approccio simile tuttavia diventa ingestibile per progetti di grandi dimensioni, in quanto si verrebbero a creare molteplici copie di file uguali o molto simili tra di loro.

In questo lavoro di tesi verrà illustrato e discusso il processo di sviluppo di un rudimentale VCS, chiamato GGit (ispirato a Git), basato su un database a grafi e implementato come una applicazione CLI in Python 3.10.

I database a grafi sono database che fanno uso di strutture a grafi per immagazzinare informazioni. Questo tipo di struttura è composta da due elementi fondamentali, nodi (o vertici) e archi, applicando all'ambito dei database, i nodi sono utilizzati per rappresentare le entità, mentre gli archi rappresentano le relazioni tra le entità.

Una delle proprietà più vantaggiose dei database a grafi utilizzati per questo progetto, ovvero LPG o labeled-property graph, consiste nel fatto che il tempo di attraversamento di un grafo è costante[1].

In questo lavoro di tesi si illustrerà il processo di sviluppo, partendo dall'illustrazione delle scelte progettuali, discutendo le metodologie di gestione dello stato di una repository e di interazioni con il database, passando poi alla descrizione delle scelte implementative e dei moduli utilizzati.

Si concluderà con una discussione sulle potenzialità di questo progetto e sulle possibili future evoluzioni.

1 Scelte progettuali

Analizzando la scelta dei tool disponibili per il versioning, si evince che Git è il VCS più diffuso e più utilizzato[2] dagli sviluppatori, è facile da utilizzare per i principianti ma allo stesso tempo supporta funzionalità avanzate per gli utenti più esperti. Per questo motivo si è deciso di ispirarsi al suo funzionamento per la realizzazione di `GGit`, mantenendo l'interfaccia utente esposta da Git, seppur in forma ridotta, per creare un'esperienza di utilizzo familiare.

1.1 Strutture dati di GGit

Git è nato come un semplice content-addressable filesystem, ovvero un sistema di storage basato su una relazione chiave-valore, nel quale è possibile archiviare qualsiasi tipo di contenuto digitale. Essenzialmente, l'azione di base che Git esegue è quella di memorizzare i contenuti di un file associandoli ad un hash calcolato su di essi: questa è la chiave che può essere utilizzata per accedere ai dati salvati.

L'hash di ogni oggetto (nomenclatura utilizzata per file e altre entità descritte in seguito), è calcolato usando l'algoritmo SHA-1, una funzione di hashing che genera un hash di 20 byte. Pur essendo un algoritmo di hashing crittografico, SHA-1 non è sicuro, in quanto è stato provato che è possibile generare collisioni[3] (ovvero due input che generano lo stesso hash). Tuttavia, per l'utilizzo al quale è destinato all'interno di un VCS (cioè identificare univocamente il contenuto di un file) la complessità di SHA-1 è sufficiente, in quanto, nel peggior caso trovato, è stato necessario effettuare 2^{61} operazioni per ottenere una collisione[4].

Git si basa su 5 tipi di oggetti:

- blob: oggetti che rappresentano un file;
- tree: oggetti che rappresentano una directory;
- commit: oggetti che pongono un particolare tree (la root directory) nella storia della repository, associando un commit padre, un timestamp e un messaggio che descrive le modifiche apportate;
- tag: un container che riferisce un altro oggetto, usato prevalentemente per dare un nome a un commit;
- packfile: una versione compressa di una serie di oggetti.

Il contenuto che verrà dato come input alla funzione di hash sarà una stringa che racchiude le caratteristiche dello specifico oggetto, fattore che aiuta a ridurre ulteriormente la probabilità di collisioni.

Si è deciso di strutturare il modello del dominio di GGit considerando i primi tre tipi di oggetti: blob, tree e commit. Di seguito viene illustrato come vengono calcolati gli hash di ogni tipo di oggetto.

1.1.1 Blob

Un file è rappresentato da un blob, per calcolarne l'hash si considera la caratteristica principale, ovvero il suo contenuto, che viene considerato in byte sia per file in puro testo ASCII che binari. Viene poi calcolata la lunghezza in byte del contenuto e il valore risultante viene concatenato con la stringa `"blob "`, seguita da un carattere NULL il tutto seguito dal contenuto del file:

```
blob <file_length>\0<file_content>
```

1.1.2 Tree

Un tree è essenzialmente una collezione di blob e tree, la funzione di un tree è associare a blob e tree un nome e una stringa che esprime i permessi sul file.

I permessi sono espressi in modo analogo alla modalità utilizzata dai sistemi UNIX, ovvero tre caratteri ottali che indicano i permessi di lettura, scrittura e esecuzione per il proprietario, il gruppo e gli altri utenti, una cifra ottale che rappresenterebbe i bit setuid, setgid e sticky (ignorati da Git) e altri tre caratteri ottali che indicano la POSIX mode[5] dell'oggetto:

- 04 per directory;
- 10 per file regolari;
- 12 per link simbolico;
- 16 per Git link o sottomoduli, non presente tra le classiche POSIX modes (non supportati da GGit).

L'hash di un tree viene calcolato in modo analogo a quello di un blob:

```
tree <tree_length>\0<tree_content>
```

dove `tree_length` è la lunghezza in byte del `tree_content`, che è una lista di oggetti, in ordine alfabetico per nome, nel seguente formato:

```
<object_mode> <object_name>\0<object_hash>
```

La stringa `object_name` è il nome del file o directory e object_hash l'hash dell'oggetto corrispondente. La `object_mode` è la stringa di sei caratteri ottali che identifica il tipo e i permessi dell'oggetto descritta in precedenza, questa stringa in GGit può assumere i seguenti valori:

- 100644 per file regolari;
- 100755 per file eseguibili;
- 040000 per directory (altri tree);
- 120000 per link simbolici,

nel caso in cui la stringa abbia zeri iniziali, questi vengono troncati (regola applicabile soltanto nel caso delle directory).

1.1.3 Commit

La funzione principale di un commit è quella di associare lo stato corrente dell'area di commit a un punto nella storia della repository e facilitare la reperibilità di un particolare stato della repository. L'area di commit, in GGit chiamata `stash`, è essenzialmente una lista di file che verranno aggiunti al prossimo commit. Per calcolare l'hash di un commit si procede in modo analogo a quello degli altri tipi di oggetti:

```
commit <commit_length>\0<commit_content>
```

dove `commit_length` è la lunghezza in byte del `commit_content`, che è una stringa formattata come segue:

```
tree <tree_hash>
parent <parent_hash>
author <author_name> <<author_email>> <author_timestamp> <timezone>
committer <committer_name> <<committer_email>> <committer_timestamp> <timezone>

<commit_message>
```

Il `tree_hash` è l'hash del tree che rappresenta la root directory dell'area di lavoro, il `parent_hash` è l'hash del commit padre, ovvero il commit dal quale è stato creato il commit corrente, se il commit è il primo della storia della repository, questo campo è omesso. Le due linee successive contengono le informazioni che identificano rispettivamente l'utente che ha implementato le modifiche (author) e l'utente che ha creato il commit (committer). A oggi GGit non supporta la possibilità di applicare patch a commit passati come Git[6], è però possibile, al momento della creazione del commit, specificare un autore diverso dal committer, per dare credito all'effettivo autore del codice e non solo all'utente che lo ha accettato all'interno della reposito-

ry. Infine l'ultima riga è il messaggio associato al commit, stringa che idealmente descrive le modifiche che vengono implementate con il commit.

1.2 Database

I tipi di oggetti in gioco hanno un alto livello di interconnessione tra di loro: ogni commit è relativo a un tree, un tree può contenere molteplici blob e tree e un tree diverso può contenere un sottoinsieme di blob e tree di un altro tree. Data questa caratteristica, si è scelto di utilizzare un database a grafi.

La gamma di diversi tipi di database a grafi è ampia (anche se non come nel caso dei database relazionali); si è scelto di utilizzare per questo progetto Neo4j, un database a grafi scalabile orizzontalmente del quale esiste una community edition totalmente opensource[7]. Neo4j è inoltre il database a grafi più utilizzato[8], con una vasta community di sviluppatori, il che aiuta in fase di sviluppo per trovare soluzioni ad eventuali problemi.

Un'altra caratteristica che ha portato alla scelta di questo particolare database sono i tool a supporto dello sviluppo, come ad esempio Neo4j Desktop o Browser, che espongono una interfaccia grafica intuitiva per eseguire query e visualizzare rappresentazioni grafiche dei grafi risultanti; è disponibile inoltre Neo4j AuraDB che permette di effettuare il deployment di un database a grafi in cloud, permettendo quindi di creare repository condivise senza necessità di gestire infrastrutture server.

1.2.1 Sviluppo del modello di dati

Il modello di dati è stato sviluppato per poter rappresentare in modo completo tutti i tipi di oggetti presenti in GGit e grazie alla caratteristica schema-less dei database a grafi, sarà possibile aggiungere nuovi tipi di oggetti o caratteristiche al modello del dominio senza intaccare maggiormente la struttura preesistente.

Anche se il database scelto è schema-less, è comunque buona pratica definire e documentare in modo preciso il modello dei dati, così da evitare in futuro di fare scelte incompatibili con il modello preesistente.

Per sviluppare il modello dei dati è possibile sfruttare l'approccio "whiteboard", che permette di rappresentare il modello in modo grafico per esprimere ad alto livello le entità e le relazioni in gioco per poi formalizzare la struttura senza dover convertire le decisioni prese in tabelle.

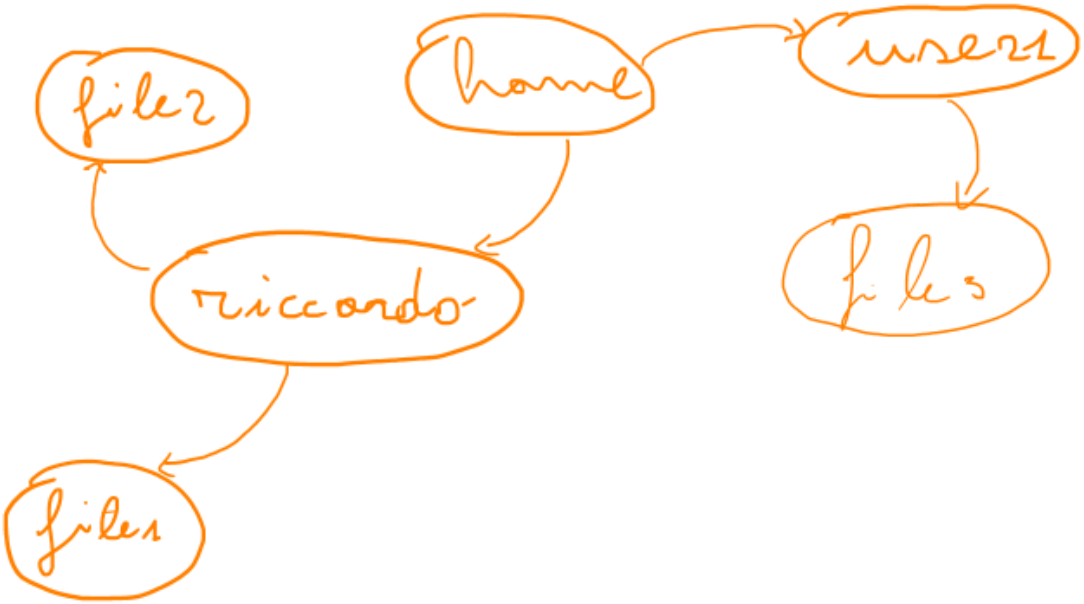


Figura 1.1: Whiteboard sketch della struttura di un'ipotetica repository

Dopo aver creato il cosiddetto "whiteboard model" si procede alla formalizzazione delle entità e relazioni individuate, etichettando con label le entità e relazioni e associandovi tutte le proprietà.

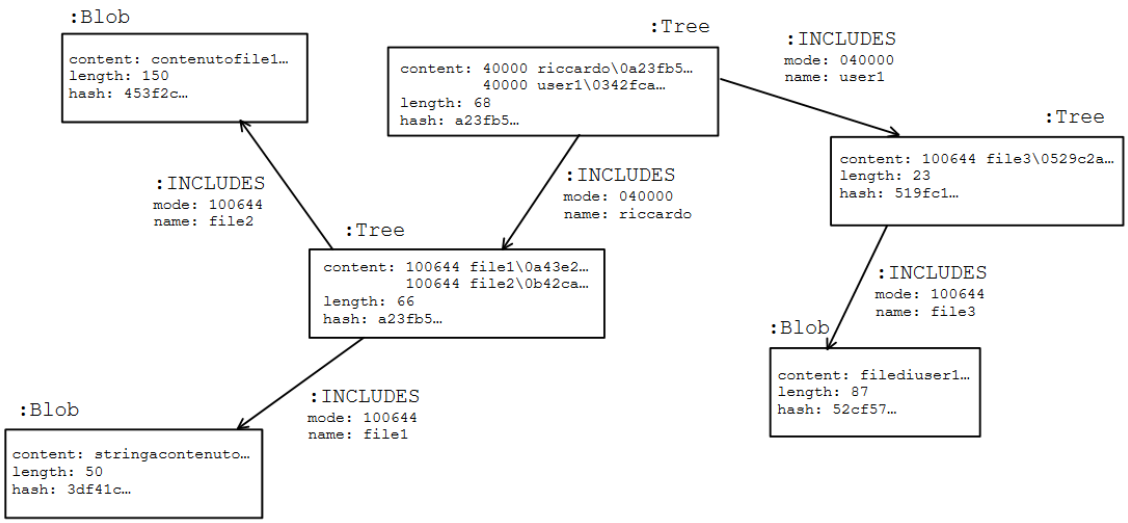


Figura 1.2: Formalizzazione dello sketch con proprietà e label

Come si nota dalla figura precedente si è deciso di assegnare i label (case sensitive) `:Tree` e `:Blob` alle rispettive entità, si individuano inoltre le seguenti proprietà per entrambi i tipi di oggetti:

- `hash`: identificatore univoco dell'oggetto, calcolato con l'algoritmo SHA-1;
- `size`: dimensione in byte del contenuto;
- `content`: contenuto dell'oggetto, come descritti sopra.

Per quanto riguarda le relazioni, in questo sottoinsieme del modello se ne individua soltanto una: il label `:INCLUDES` indica che un oggetto ne contiene un altro. Le relazioni sono ordinate e nel caso di `:INCLUDES` il nodo di partenza è sempre un `:Tree` e il nodo di arrivo può essere sia un `:Tree` che un `:Blob`. Si denotano inoltre due proprietà per questa relazione:

- `name`: nome del file o della directory;
- `mode`: GGit mode dell'oggetto, può assumere il valore "040000" se il nodo di arrivo ha il label `:Tree` e i valori "100644", "100755", "120000" se il nodo di arrivo ha il label `:Blob`.

È stata fatta la scelta di associare le proprietà `name` e `mode` alla relazione e non al nodo in quanto rispecchia correttamente il modello dei dati definito: un oggetto `blob`, così come un `tree`, non ha un nome o una mode associati fino al momento in cui viene inserito all'interno di un `tree`.

Sotto viene mostrato il whiteboard sketch relativo al modello dei `commit` e degli `user`, e come possono essere collegati tra di loro:

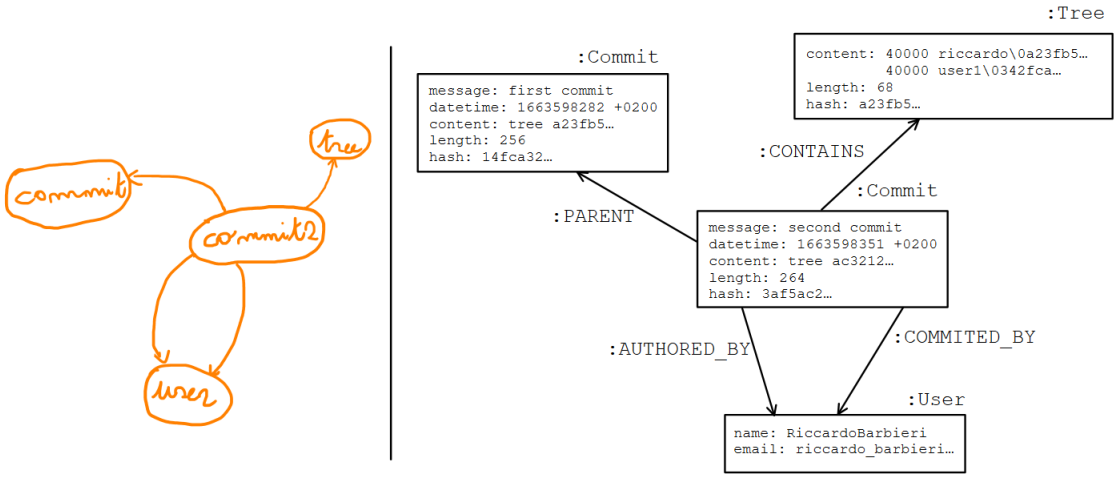


Figura 1.3: Whiteboard sketch del modello dei commit e degli user

In questa sezione del modello si individuano due nuove entità, `:Commit`, che è identificata dalle seguenti proprietà:

- `message`: messaggio del commit;
- `datetime`: momento di creazione del commit (UNIX timestamp) e fuso orario (in ore dal UTC);
- `content`: stringa che rappresenta il commit (quella che viene data come input all'algoritmo di hashing).
- `length`: lunghezza del content.
- `hash`: identificatore univoco del commit, calcolato come descritto sopra,

e `:User`, identificata dalle seguenti proprietà:

- `name`: nome dell'utente;

- `email`: email dell'utente;

Si identificano inoltre nuove relazioni: `:COMMITTED_BY` e `:AUTHORED_BY` che indicano rispettivamente il committer e l'autore di un commit; `:PARENT` che parte da un commit e punta al commit padre e infine `:CONTAINS` che parte da un commit e punta ad un oggetto `tree` che rappresenta lo stato della repository al momento del commit.

Come dimostrato, il modello whiteboard è estremamente utile per formalizzare le entità e relazioni in gioco e per individuare le proprietà da associare ad esse, data la natura ad alto livello di questo approccio è molto semplice passare da un idea, o uno sketch, a una definizione formale del modello dei dati.

1.3 Stato della repository

In questa sezione verranno descritte le modalità di gestione dello stato della repository, che comprende la lista dei file cosiddetti "tracked"; la lista dei file "stashed" e la gestione delle differenze dei file dell'ultima versione.

1.3.1 Stash e tracking

Una delle funzioni fondamentali che GGit utilizza per tenere traccia della storia di una repository è l'area di `stash`, questa è l'area che raccoglie la lista dei file che verranno aggiunti al prossimo commit. Questa area è strettamente legata alla lista dei file tracked, che sono quei file che un qualsiasi punto nella storia della repository sono stati aggiunti all'area di stash.

Si è scelto di mantenere l'area di stash e tracking nella repository stessa, in memoria locale, in modo da evitare di dover sostenere una connessione al database per operazioni che possono essere ripetute frequentemente.

1.3.2 Differenze tra versioni di file

GGit è in grado di riportare quali file sono stati modificati rispetto all'ultima versione, nella versione corrente dell'applicazione questa funzione è utile soltanto ai fini di un particolare comando (descritto successivamente) ma si è deciso di aggiungere questa funzionalità a supporto di future operazioni, come ad esempio la gestione di conflitti durante il merge automatico di versione differenti di uno stesso file.

1.4 Gestione del server Neo4j

Inizialmente si era pensato di includere una distribuzione del server Neo4j all'interno del package dell'applicazione, in modo da non dover richiedere all'utente di installare

il server autonomamente, tuttavia si è scelto di istruire l'utente a installare una versione del server Neo4j, la directory dell'installazione andrà poi specificata all'interno di una variabile d'ambiente chiamata `"NEO4J_HOME"`, variabile che verrà utilizzata per reperire l'installazione e copiarla all'interno di ciascuna repository; il database copiato viene poi inizializzato.

È stata fatta questa scelta per permettere di utilizzare la versione community del server di Neo4j per repository locali, versione che non permette di creare multipli database associati allo stesso server, è comunque possibile utilizzare una versione enterprise del server, ma richiede configurazione aggiuntiva (discussa nelle Impostazioni di configurazione). Inoltre è possibile connettersi ad un server Neo4j remoto, in questo caso è necessario specificare l'indirizzo del server e le credenziali di accesso attraverso le impostazioni di configurazione.

1.5 Interfaccia utente

L'interfaccia utente di `GGit` è ispirata a quella di `Git`, del quale si è deciso di implementare un sottoinsieme ridotto di comandi, ovvero quelli essenziali al funzionamento di base di un VCS:

- `init`;
- `add`;
- `mv`;
- `rm`;
- `commit`;
- `status`;
- `config`;
- `log`.

1.5.1 Gestione della repository

Questi sono i comandi per gestire la repository.

Init

Il comando `ggit init` crea una nuova repository, in particolare viene create la directory `.ggit` all'interno della directory nella quale viene chiamato il comando, si è però deciso di permettere all'utente di specificare la directory da inizializzare con il comando `ggit init <path>`, in questo modo è possibile creare una repository all'interno di una directory diversa dalla corrente.

Dopo aver creato la directory, vengono inizializzati i file di configurazione e dello stato e vengono impostate alcune configurazioni di base, viene inoltre richiesto in-

terattivamente all'utente di inserire username e email, che verranno utilizzati per i commit. Viene poi inizializzato lo stato della repository, reperita l'installazione di Neo4j e impostate le impostazioni relative a username, password e nome, versione e posizione del database.

Config

Il comando `ggit config` permette di gestire le impostazioni di configurazione, in particolare è possibile visualizzare le impostazioni correnti, impostare nuove impostazioni e rimuovere impostazioni esistenti.

Questo comando offre i seguenti parametri:

- `--list` per visualizzare tutte le impostazioni correnti;
- `--get <key>` per visualizzare il valore di una impostazione;
- `--set <key> <value>` per impostare il valora di una impostazione, se non esiste viene aggiunta;
- `--unset <key>` per rimuovere una impostazione.

1.5.2 Gestione dei file

I seguenti comandi sono disponibili all'utente per modificare lo stato della repository, aggiungendo rimuovendo e spostando file.

Tutti i comandi in questa categoria supportano l'utilizzo da qualsiasi sottodirectory della repository, tenendo conto della relatività degli eventuali path specificati.

Add

Il comando `ggit add` permette di aggiungere un lista di file e/o directory all'area di `stash` della repository; ogni volta che un file viene aggiunto, viene anche inserito nella lista dei file tracked, nel caso delle directory vengono aggiunti tutti i file contenuti nella directory stessa e in tutte le sottodirectory.

Il comando offre un solo parametro, obbligatorio, che accetta valori multipli sotto forma di lista separata da spazi, questi sono le stringhe che verranno interpretate come path.

Mv

Il comando `ggit mv` permette di spostare un file o una directory da una posizione all'altra, in questo caso il file viene effettivamente spostato nel file system e viene aggiornato il path del file nella lista dei file tracked e dello stash.

Il comando accetta due parametri, obbligatori, che accettano rispettivamente un solo valore che verrà interpretato come path, il primo è il file o la directory di partenza e il secondo la destinazione.

Rm

Il comando `ggit rm` permette di rimuovere un file o una directory, specificati come parametri, dalla lista dei file tracked e dello stash, in questo caso il file viene rimosso dal file system.

Così come il comando `add`, anche questo comando supporta l'utilizzo di valori multipli, separati da spazi, per specificare più file o directory da rimuovere.

1.5.3 Gestione della storia

Commit

Il comando `ggit commit` permette di creare un nuovo commit partendo dallo stato corrente dello stash della repository.

Inizialmente viene creato un nuovo oggetto tree, partendo dallo stash della repository. Dopodiché vengono interpretate le opzioni passate al comando, in particolare è possibile specificare:

- `-m` o `--message`: specifica il messaggio del commit;
- `-f` o `--file`: specifica il nome del file dal quale leggere il messaggio di commit, mutualmente esclusivo con `-m`;
- `--author`: specifica l'autore del commit, nel formato `nome,email` per effettuare l'override dell'autore di default;
- `--date`: specifica la data, nel formato `YYYY-MM-DDTHH:mm:SS+HH:mm` per effettuare l'override della data corrente.

Dopo aver valutato e interpretato le opzioni, viene recuperato dalle impostazioni di configurazione l'hash dell'ultimo commit, con il quale si ottiene dal database l'oggetto commit associato che viene assegnato al nuovo commit, se questa impostazione è vuota o non presente, significa che il commit che viene effettuato è il primo e non necessita un parent.

Infine viene effettivamente aggiunto il commit al database, aggiornata l'impostazione relativa all'ultimo commit e viene svuotato lo stash.

1.5.4 Visualizzazione dello stato

Status

Il comando `ggit status` permette di visualizzare un riassunto dello stato corrente della directory, in particolare vengono mostrate tre liste di file. La prima lista elenca i file che sono al momento nell'area di `stash` e quindi verranno aggiunti al prossimo commit. La seconda lista mostra i file che sono stati modificati rispetto al commit precedente, sono stati aggiunti, a un certo punto nella storia della repository, all'area di `tracking` e non sono ancora nell'area di `stash`. Infine la terza lista contiene i file che sono stati creati ma non sono mai stati aggiunti all'area di `stash` e quindi non sono ancora nel tracking della repository.

Log

Il comando `ggit log` permette di visualizzare lo storico dei commit della repository, stampa una lista dei commit più recenti, mostrando il loro hash, l'autore, la data di creazione e il messaggio di commit.

Il comando supporta una opzione `-d` o `--depth` che permette di specificare il numero di commit da visualizzare, il valore di default è impostato a 5.

1.6 Impostazioni di configurazione

Le impostazioni di configurazione sono una serie di entry chiave-valore, vengono utilizzate dall'applicazione per salvare e ottenere informazioni relative alla configurazione della repository e del database.

Tutte queste impostazioni possono essere aggiunte o modificate manualmente attraverso il comando `ggit config`.

1.6.1 Configurazione della repository

Alla versione corrente, le impostazioni di configurazione della repository sono le seguenti:

- `"repository.path"`: path che punta alla root directory della repository;
- `"HEAD"`: contiene l'hash dell'ultimo commit effettuato, se non è presente o è impostato a `"None"` significa che non è ancora stato effettuato nessun commit;
- `"user.name"` e `"user.email"`: nome dell'utente e email utilizzati per i commit.

1.6.2 Configurazione del database

- `"database.username"` e `"database.password"`, che contengono le credenziali di accesso al database di default;
- `"database.name"`, contiene il nome di default del database nella community edition;
- `"database.path"`, salva il path alla directory di installazione di Neo4j, ottenuto tramite la variabile d'ambiente `"NEO4J_HOME"`.

Le impostazioni del database vengono inizializzate automaticamente ai valori utili per la community edition, per utilizzarne una enterprise è necessario modificare manualmente le variabile`"database.username"` e `"database.password"` con le credenziali personalizzate del database, mentre la variabile `"database.name"` deve essere impostata al nome del database che si vuole utilizzare.

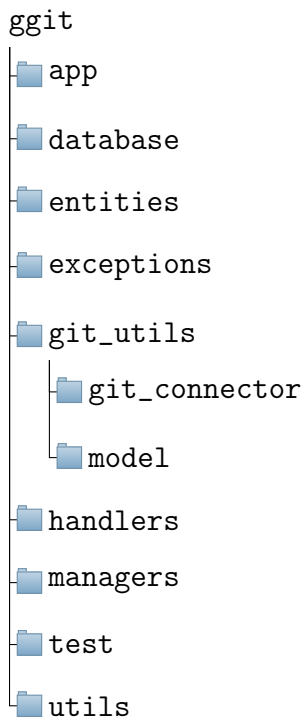
2 | Implementazione

In questo capitolo verrà illustrato il processo di implementazione di `GGit`, facendo riferimento alle scelte progettuali discusse e ponendo particolare attenzione alle problematiche riscontrate durante lo sviluppo.

Come già accennato, si è scelto di sviluppare `GGit` utilizzando Python 3.10, decisione derivata dal fatto che Python è un linguaggio molto flessibile e che permette uno sviluppo veloce sin dai primi passi di un progetto; un altro vantaggio non indifferente è la disponibilità di una quantità disarmante di librerie potenti e ben documentate, che semplificano molto alcune fasi dello sviluppo.

Si inizierà con una breve descrizione delle decisioni prese in merito alla suddivisione in package del codice, per poi passare a una analisi più approfondita delle porzioni di codice più interessanti per ogni modulo. Per precisare, in Python, per package si intende un contenitore per moduli e sub-package (solitamente una directory), mentre un modulo è semplicemente un file che contiene codice Python.

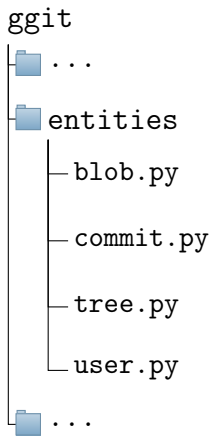
Suddivisione in package



La figura sopra mostra la struttura di package adottata, si evidenziano particolarmente le cartelle `database`, `entities`, `handlers` e `managers`, che contengono i moduli più importanti per il funzionamento di `GGit`.

2.1 Entità

Il package `entities` contiene i moduli che espongono le classi che rappresentano le entità del dominio di `GGit`. In particolare, si trovano le classi `Blob`, `Commit`, `Tree` e `User`. Ogni classe è implementata in un file separato.



La classe `ggit.entities.user.User` contiene gli attributi `name` e `email` e espone metodi accessor tramite l'uso del docoratore `property`.

Le classi

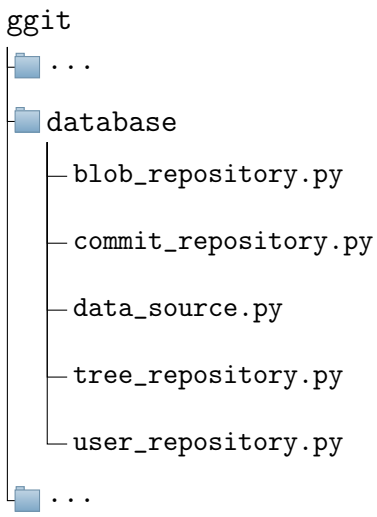
- `ggit.entities.blob.Blob`;
- `ggit.entities.tree.Tree`;
- `ggit.entities.commit.Commit`;

espongono metodi per ottenere l'hash dell'oggetto che rappresentano, così come altri metodi per accedere alle informazioni in esse contenuti (contenuto del file per i blob, autore, committer, data etc. per i commit etc.).

Le proprietà calcolate, come il content per tree e commit o la sua lunghezza, sono state progettate per non essere modificate dall'utente, quindi non è stata implementata la funzione setter decorata con `{attribute}.setter`

2.2 Database

Le classi che governano l'interazione con il database sono contenute nei moduli all'interno del package `database`.



2.2.1 Connessione al database

Le connessioni al database sono gestite attraverso l'utilizzo della classe DataSource, contenuta nel modulo `ggit.database.data_source`.

Un oggetto di questa classe, quando inizializzato, ottiene dalle impostazioni di configurazione le credenziali del database e salva come suo attributo il driver di connessione, inizializzandolo con l'uri fornito e le credenziali ottenute. L'uri del database può essere passato come parametro all'inizializzazione, se non viene specificato viene utilizzato il valore di default `"bolt://localhost:7687"`.

La classe espone il metodo `new_session()`, creato per restituire un oggetto di tipo `neo4j.Session` legato al driver creato all'inizializzazione, che può essere utilizzato per interagire con il database; l'utilizzo consigliato di questo metodo è insieme allo statement `with` per creare un context manager che si occupa di chiudere correttamente la sessione una volta terminate le operazioni. È inoltre esposto il metodo `close()`, che chiude la connessione al database insieme a tutte le sessioni associate.

Questa classe è stata implementata come singleton tramite l'uso di una metaclass chiamata `SingletonMeta` (dettagliata in seguito), in modo da utilizzare lo stesso driver durante una singola esecuzione per semplificare la gestione delle sessioni.

2.2.2 Mapping entità database

Per mappare le entità in gioco al database si utilizza una metodologia CRUD (Create, Read, Update, Delete) senza però implementare i metodi di Update, dato che nel VCS non è prevista la modifica di un commit, o dei dati ad esso associati, una volta creato.

Per ogni entità è stata implementata una classe "repository", che espone i metodi necessari per aggiungere, rimuovere e ottenere uno o tutti gli oggetti di quel tipo dal database:

- `add_{entity}` per aggiungere un oggetto nel database;
- `delete_{entity}` per rimuovere un oggetto dal database;
- `get_{entity}` per ottenere un oggetto dal database;
- `get_all_{entity}` per ottenere tutti gli oggetti di un certo tipo dal database.

Queste classi fanno uso della classe `ggit.database.data_source.DataSource` per ottenere una sessione con il database e per eseguire le query scritte in Cypher, linguaggio di query di Neo4j.

Cypher è un linguaggio progettato per essere espressivo ed efficiente, permette di esprimere con una sintassi concisa query anche molto complesse, permette quindi di concentrare l'attenzione sul dominio in considerazione. inizialmente è nato come linguaggio ad uso esclusivo dei database Neo4j ma nel 2015 è stato scelto come standard per il progetto openCypher[9], che mira a creare un linguaggio di query comune a tutti i database a grafi.

Di seguito verranno illustrate alcune delle query e modalità di creazione delle entità nel database.

Gestione di blob e utenti

Le repository che si occupano della gestione dei blob sono le più semplici, in quanto quando vengono create o eliminate non devono prendersi cura delle relazioni con gli altri oggetti, compito lasciato alle repository delle entità composte.

Esempio di query per creare un oggetto blob:

```
MERGE (b:Blob {id: $blob_id, content: $content})
```

Questa query è composta da due sezioni principali: il pattern che segue la parola `MERGE` e la clausola `MERGE` che indica come comportarsi con il pattern specificato.

La keyword `MERGE` indica che se il pattern specificato non è presente nel database, deve essere creato, è anche possibile usarla insieme alle clausole `ON MATCH` e `ON CREATE` per specificare come comportarsi nel caso in cui il pattern sia già presente nel database o se deve essere creato, tuttavia in questa query non tornano utili.

La sintassi dei pattern è progettata per essere simile a come si disegnerebbe un grafo su una lavagna: cerchi per i nodi e freccie per i collegamenti.

Un altro componente della query sono i parametri, indicati con il simbolo `$`, che vengono sostituiti con i valori passati come argomento alla funzione che esegue la query.

Per la gestione degli user e dei blob vengono eseguite transazioni autogestite, attraverso l'uso del metodo `neo4j.Session.run()` che permette di eseguire una query, impostandone opportunamente i parametri, e effettuare automatica il rollback in caso di errore.

Gestione di commit e tree

Le repository che si occupano della gestione dei commit e dei tree sono più complesse, in quanto devono creare anche i nodi relativi agli oggetti che li compongono e le relazioni tra di essi, per questo motivo vengono utilizzate transazioni esplicite, che permettono di gestire manualmente il commit o il rollback delle operazioni: in caso di una qualsiasi eccezione durante l'esecuzione delle query per inserire un oggetto complesso, viene effettuato il rollback.

Il metodo `add_tree()`, utilizzato per creare un nuovo oggetto tree nel database, come prima cosa crea il nodo principale che rappresenta il tree, si itera poi sulla lista di item contenuti nel tree, creando opportunamente blob attraverso una istanza di `BlobRepository` e subtree usando il metodo stesso in modo ricorsivo.

Esempio di query utilizzate per creare la relazione tra un tree e un suo subtree:

```
MATCH (t:Tree {hash: $hash}) MATCH (t2:Tree {hash: $hash2})
MERGE (t)-[:INCLUDES {mode: $mode, name: $name}]->(t2)
```

Inizialmente vengono identificati i due tree tramite il loro hash nelle clausole `MATCH`, e successivamente viene utilizzata una `MERGE` per creare la relazione.

Altre due query interessanti nell'ambito dei tree sono quella per cancellare un tree e per crearne uno nuovo.

```
MATCH (tree:Tree {hash: $hash})-[r*1..]->(n)
DETACH DELETE tree, n
```

Con questa query viene cancellato un tree: il pattern identifica il tree principale, tutti i percorsi, di lunghezza almeno 1 (sintassi `r*1..`), in uscita verso un qualsiasi nodo e i nodi terminali, viene poi utilizzata la clausola `DETACH DELETE` per eliminare i nodi identificati e tutte le relazioni che li collegano.

```
MATCH relation = (tree:Tree {hash: $hash})-[:INCLUDES*1..]->(item)
RETURN tree, relation, item
```

Questa query, eseguita dal metodo `get_tree()`, identifica tutti i percorsi composti da relazioni di tipo `:INCLUDES` di lunghezza almeno 1 in uscita dal tree e tutti i nodi terminali e li restituisce come risultato. Il risultato viene poi utilizzato dal metodo, sotto forma di oggetto `neo4j.Graph`, per ricostruire l'entità tree.

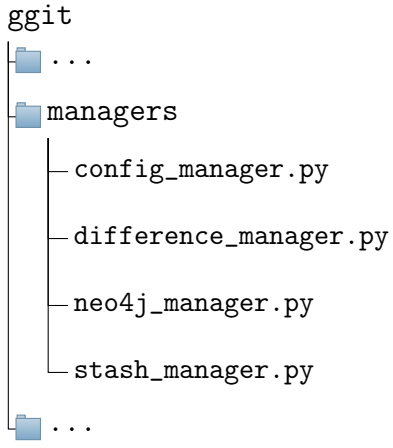
L'oggetto `neo4j.Graph` è una funzionalità sperimentale del driver Neo4j per Python che permette di rappresentare un grafo come un oggetto container di oggetti `neo4j.Node` e `neo4j.Relationship`, oggetto che viene passato come parametro alla funzione di utility `parse_tree()` che si occupa di ricostruire l'entità tree.

Per creare nuovi commit invece, si procede con l'apposito metodo `add_commit()` che si assicura, tramite una `UserRepository`, che esistano gli utenti necessari, per poi passare alla creazione del nodo commit, legandolo contemporaneamente ad autore e committer. Se il commit ha un parent, viene creata la relazione tra i due commit, viene successivamente aggiunto il main tree del commit usando una `TreeRepository`, infine viene creata la relazione `:CONTAINS` tra il commit e il main tree.

L'eliminazione di un commit non è ancora supportata dal sistema, ma è comunque implementato il metodo utile a cancellare il nodo e tutte le relazioni che lo collegano ad altri nodi.

2.3 Managers

Nel package `managers` sono contenuti i moduli per la gestione dello stato della repository, delle impostazioni di configurazione e del server del database.



2.3.1 Gestione della configurazione

La classe ConfigManager permette di aggiungere, rimuovere e modificare le impostazioni di configurazione della repository, è stata implementata come singleton, in modo da utilizzare la stessa istanza in tutte le parti del programma.

Quando viene inizializzata vengono caricati i dati dal file `.ggit/config.json`, se non esiste viene creato un file vuoto.

La classe espone metodi per ottenere il path della repository che sta gestendo e il dizionario di valori che rappresenta la configurazione. Sono inoltre implementati i

metodi `__{get, set, del}item__`, per permettere alla classi di comportarsi come un dizionario python per ottenere e modificare i valori della configurazione.

2.3.2 Gestione dello stash e delle differenze

La classe `StashManager` permette di gestire lo stash, ovvero effettuare operazioni per aggiungere un file o una directory allo stash, rimuoverlo e spostarlo, così come pulire l'area di stash, operazione utilizzata quando viene effettuato un commit.

Ciascuna operazione ha un metodo pubblico dedicato all'interno della classe, che si occupa di effettuare l'operazione richiesta e di aggiornare il contenuto dei file `.ggit/tracked_files.json` e `.ggit/stash.json` attraverso un dump json delle strutture dati interne alla classe.

Come in molte altre operazioni di questa applicazione, i metodi sono stati implementati in modo ricorsivo, ad esempio il metodo per aggiungere un file allo stash richiede un parametro `path` che rappresenta il path del file o directory da aggiungere e a seconda della sua natura viene utilizzato il metodo privato `__stash_folder()` o `__stash_file()`.

Sono inoltre esposti i metodi getter per ottenere il contenuto dello stash, sotto forma di dizionario con chiave le stringhe che rappresentano il path del file e valore l'hash relativo, e la lista dei file tracked, sotto forma di lista di stringhe che rappresentano i path dei file.

La classe `DifferenceManager` è utilizzata per tenere traccia dei cambiamenti effettuati su un file, in modo da poterli visualizzare tramite il comando `ggit status`. Nella versione corrente dell'applicazione la classe tiene solo traccia di quali file sono stati modificati, non delle modifiche effettuate.

Il funzionamento di questa classe si basa sul file `.ggit/current_state.json`, che contiene un dizionario con chiave i path dei file contenuti nella repository e valore il loro hash. Quando la classe viene inizializzata viene caricato il contenuto del file e ogni volta che viene utilizzato il metodo getter `different_files` viene creato un dizionario formato come descritto sopra contenente i path dei file che hanno un hash diverso da quello salvato nel file `.ggit/current_state.json`.

È possibile aggiornare lo stato attraverso il metodo `update_current_state()`, che viene utilizzato quando viene effettuato un commit, in modo da aggiornare lo stato della repository.

La classe fa uso di un oggetto di tipo `utils.folder.Folder`, che costituisce una astrazione di una directory con qualche funzionalità aggiunta, descritta nella sezione 2.5.

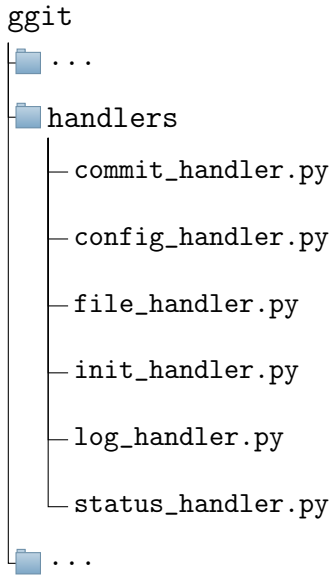
2.3.3 Gestione del server del database

Il modulo `neo4j_manager` contiene gli strumenti utilizzati per gestire lo stato del database del server, nella versione corrente implementa la funzione che permette di avviare il server del database, `start_neo4j_instance()`.

Questa funzione si occupa inoltre di salvare il pid del processo del server all'interno del file `.ggit/neo4j.pid` per poterlo utilizzare in seguito per interigire con il processo, se necessario. Questa funzione viene chiamata all'inizio di ogni handler che necessita di interagire con il database, per assicurarsi che il server sia in esecuzione.

2.4 Handlers

Nel package `handlers` sono raccolti i moduli che implementano funzioni per la gestione dei comandi eseguiti dagli utenti.



2.4.1 init_handler

Questo modulo si occupa di gestire l'inizializzazione della repository: dopo aver controllato che la directory corrente non sia già una repository, viene creata la directory `.ggit` e vengono creati i file di configurazione e tracking dello stato.

Vengono poi inizializzate le impostazioni di base, ovvero `repository.path`, `user.name`, `user.email` e `HEAD`. Successivamente viene ricercata la posizione dell'installazione di Neo4j e viene creata una copia del database di default nella directory `.ggit/neo4j-version`, infine viene assegnato un valore alle impostazioni `database.path`, `database.version`, `database.name`, `database.username` e `database.password`.

2.4.2 file_handler

Il modulo `file_handler` contiene le funzioni per attuare i sottocomandi relativi alla gestione dei file, ovvero `add`, `rm` e `mv`.

Questi metodi fanno uso della funzione di utilità `parse_paths()`, che prende in input una lista di argomenti passati al comando e li interpreta come path, controllando che puntino a file esistenti interni alla repository e li restituiscono come oggetti `pathlib.Path`.

La funzione `add_handler()` si occupa di aggiungere i file specificati allo stash; la funzione `rm_handler()` rimuove i file dallo stash (e dal file system) e la funzione `mv_handler()` si occupa di spostare i file specificati.

Queste tre funzioni fanno uso dei rispettivi metodi `stash()`, `unstash()` e `move()` del manager `StashManager`.

2.4.3 commit_handler

Il modulo `commit_handler` contiene la funzione `commit_handler()`, che si occupa di effettuare un commit.

La funzione crea un oggetto commit a partire dallo stato corrente della repository, utilizzando una istanza di `StashManager`, e elabora i parametri forniti dall'utente, in particolare il messaggio o file di commit, la data e l'autore, se specificati.

Viene utilizzata una istanza della classe `CommitRepository` per ottenere l'entità `Commit` padre, partendo dall'hash salvato nella impostazione di configurazione `HEAD` e per aggiungere il commit creato al database.

Dopo aver aggiunto il commit, viene aggiornato la variabile `HEAD` con l'hash del commit appena creato e viene aggiornato lo stato della repository, utilizzando una istanza di `DifferenceManager`, e svuotato lo stash, utilizzando una istanza di `StashManager`.

2.4.4 log_handler, status_handler, config_handler

In modo similare agli altri handler vengono implementate le funzioni per gestire i comandi `log`, `status` e `config`.

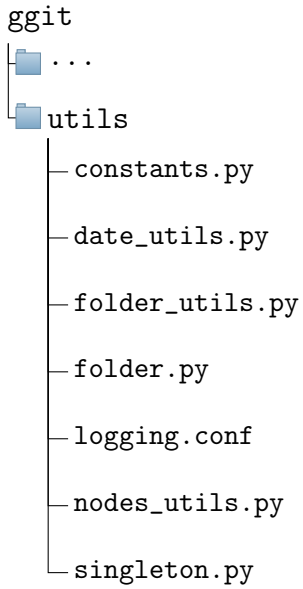
La funzione `log_handler()` si occupa di stampare l'elenco dei commit effettuati, utilizzando una istanza di `CommitRepository` per ottenere l'entità `Commit` padre, partendo dall'hash salvato nella impostazione di configurazione `HEAD` e per ottenere l'elenco di un certo numero di commit padre, quantità specificata attraverso il parametro apposito.

La funzione `status_handler()` si occupa di stampare lo stato corrente della repository, utilizzando una istanza di `DifferenceManager` per ottenere l'elenco dei file modificati, aggiunti e rimossi.

La funzione `config_handler()` esegue tre tipi di azioni, a seconda del parametro specificato, e aggiunge, rimuove o modifica il valore di una impostazione di configurazione. Si occupa anche di stampare la lista di tutte le impostazioni, se viene fornito il parametro apposito.

2.5 Moduli di utilità

Nel package utils, sono contenuti una varietà di moduli contenenti funzioni di utilità e classi di supporto, nonché un file di configurazione per il logger utilizzato dall'applicazione.



Si proseguirà ora con una descrizione sommaria dei moduli più importanti.

`constants` e `date_utils`

Il modulo `date_utils` contiene le funzioni per la gestione delle date, utilizzata dalle classi relative ai commit, fa uso del modulo `constants` che ospita una serie di costanti utili nell'applicazione, come stringhe di regex e nomi di file.

`folder_utils`

Questo modulo contiene funzioni per navigare directory, come `walk_folder_flat()` e `walk_folder_rec_flat()` che permettono di ottenere i file contenuti in una repository, ricorsivamente o no.

È inoltre dichiarata una funzione, `find_repo_root()` per ottenere la root di una repository partendo da un path, che restituisce `None` nel caso in cui nessuna delle directory superiori sia una repository `ggit`.

`nodes_utils`

Il modulo `nodes_utils` contiene funzioni per analizzare e filtrare liste di nodi, come `get_node()`, che individua un nodo dato un hash.

Contiene inoltre la funzione `parse_tree()` che, a partire da una lista di nodi e una di relazioni, come restituite da una query al database, crea un oggetto `Tree`, partendo dall'hash del nodo radice.

`singleton`

Questo modulo contiene la metaclasse `SingletonMeta`, che permette di creare classi singleton, per assicurarsi che una classe abbia una sola istanza.

La classe ridefinisce il metodo `__call__()` per restituire sempre la stessa istanza delle classi marcate con questa metaclasse.

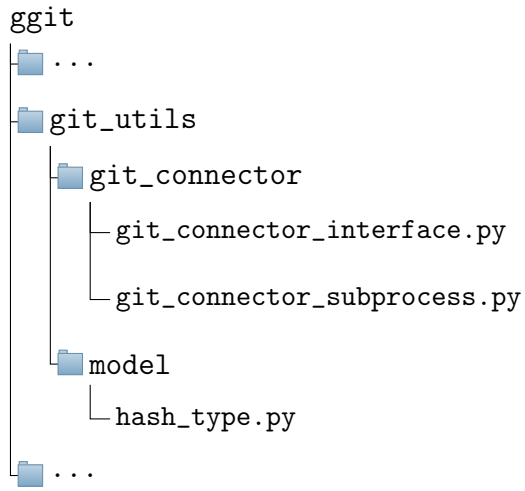
`folder`

La classe `Folder` è fondamentale per il funzionamento del `DifferenceManager`.

Quando inizializzata con un path, viene caricata la struttura della directory e i file contenuti, in modo da poter effettuare le operazioni di confronto tra lo stato corrente e quello precedente.

La funzionalità principale che la classe offre è la possibilità di considerare contemporaneamente i contenuti del file `.ggitignore`, se presente, ignorando i file che corrispondono ai pattern specificati e i contenuti di una whitelist, per poter limitare quali file considerare al momento della creazione di un tree per un commmit.

2.6 Package test e git_utils



In questo package sono contenuti i moduli che implementano test per le funzionalità critiche al funzionamento dell'applicazione.

In alcuni test, ad esempio quelli che controllano se la generazione degli hash è corretta, fanno uso del modulo package `git_utils`, che contiene classi per interagire con repository `git` attraverso la sua CLI e per poter astrarre alcune sue entità per confrontare i risultati del calcolo di hash e identificazione del tipo di oggetto.

2.7 Main applicazione

Lo script principale che avvia l'applicazione è contenuto nel modulo `app`.

In questo modulo è contenuta la classe `GGitAppParser` che estende la classe `ArgumentParser` della libreria `argparse`, usata per creare l'interfaccia da linea di comando. Questa classe permette di aggiungere i sottocomandi e relativi parametri, generando automaticamente messaggi di errore e di help e gestendo l'eventuale mutua esclusione tra i parametri.

Nel main di questo modulo viene effettuato il parse degli argomenti forniti e avviato l'handler corrispondente al comando specificato, passando gli opportunamente gli argomenti come parametri; viene inoltre creato il logger e passato come parametro a tutti gli handler che lo utilizzano.

Sviluppi futuri

Questo progetto di tesi implementa le funzionalità di base di un VCS, tuttavia esistono funzionalità interessanti che in futuro verranno implementate, qui di seguito verranno descritte alcune di queste funzionalità.

`GGit` in futuro permetterà di effettuare il pull dal database di una versione arbitraria di una repository, per poter visualizzare i cambiamenti apportati da una versione all'altra.

Un'altra caratteristica estremamente utile di un VCS è la possibilità di creare delle cosidette branch, ovvero la crezione di versioni sviluppate in parallelo dello stesso progetto, ad esempio per implementare nuove funzionalità o correggere bug, senza dover modificare il codice sorgente della versione principale del progetto; questa funzionalità verrà implementata da `GGit`, attraverso l'espansione delle funzionalità della classe `DifferenceManager`, che verrà estesa con l'ausilio di altre classi per permette di confrontare versioni differenti dello stesso file per unire le modifiche apportate in parallelo una volta terminato lo sviluppo all'interno di una branch.

Ringraziamenti

Ringrazio infinitamente i miei genitori per avermi supportato durante questo percorso e per aver creduto in me, a loro dedico questa tesi.

Bibliografia

[1] Christian Theil Have e Lars Juhl Jensen. «Are graph databases ready for bioinformatics?» In: *Bioinformatics* 29.24 (ott. 2013), pp. 3107–3108. DOI: 10.1093/bioinformatics/btt549. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3842757/.

[2] Stackoverflow. *Developer Survey*. Mag. 2022. URL: https://survey.stackoverflow.co/2022/#section-version-control-version-control-systems.

[3] Marc Stevens. *Attacks on Hash Functions and Applications*. 19 Giu. 2012.

[4] Aron Toponce. *The Reality of SHA1*. 6 Mar. 2014. URL: https://pthree.org/2014/03/06/the-reality-of-sha1/.

[5] Git. *Git Documentation*. 1 Mag. 2021. Cap. The Git index file. URL: https://git-scm.com/docs/index-format.

[6] Git. en. 2ª ed. Berlin, Germany: APress, nov. 2014. Cap. 2.3. URL: https://git-scm.com/book/en/v2/Git-Basics-Viewing-the-Commit-History.

[7] *Neo4j*. URL: https://github.com/neo4j/neo4j.

[8] *DB-Engines Ranking of Graph DBMS*. 2022. URL: https://db-engines.com/en/ranking/graph+dbms.

[9] Emil Eifrem. *Meet openCypher: The Open Source SQL for Graphs*. https://neo4j.com/blog/open-cypher-sql-for-graphs/. 21 Ott. 2015.