Università degli Studi di Salerno

Dipartimento di Informatica
Corso di Laurea Triennale in Informatica

Tesi di Laurea

Identificazione di Tangled Code Changes in Repository Open-Source

Relatore
Prof. **Andrea De Lucia**
Università degli Studi di Salerno

Candidato
Marco Costante
Matricola: 0512105772

Anno Accademico 2020-2021

A chi mi ha sempre sostenuto

Abstract

Gli sviluppatori, durante le varie fasi del ciclo di vita di un software, si trovano frequentemente ad interagire con sistemi di controllo di versione, il cui scopo è consentire di gestire le modifiche apportate ai file di un progetto garantendo reversibilità e lavoro concorrente. Un recente studio ha mostrato che gli sviluppatori tendono spesso, soprattutto in fase di correzione, a raggruppare modifiche poco, o per nulla, correlate tra di loro in un unico commit, dando luogo in tal modo ai cosiddetti Tangled Code Changes. I Tangled Changes non causano di per sé problemi nello sviluppo del progetto software, ma minacciano e compromettono fortemente l'analisi cronologica delle revisioni. L'obiettivo di questo lavoro di Tesi è quello di progettare e sviluppare un software in grado di identificare automaticamente i Tangled Changes in modo da poter in futuro disporre di commit che coinvolgano cambiamenti che siano il più possibile connessi tra di loro.

Indice

Elenco delle figure

Capitolo 1

Introduzione

1.1 Contesto applicativo

L'ingegneria del software è l'applicazione di un approccio sistematico, disciplinato e quantificabile allo sviluppo, funzionamento e manutenzione del software [1].

Ogni sistema software ha un suo ciclo di vita, pertanto è soggetto ad una continua evoluzione nel tempo, poiché subisce costantemente modifiche per introdurre funzionalità o adattarne alcune esistenti.
Risulta evidente, quindi, quanto la collaborazione sia un aspetto fondamentale nel processo di sviluppo di progetti di qualsiasi dimensione.

La gestione di un progetto software, inoltre, include tutte quelle attività di supervisione, che assicurano la consegna di un sistema di alta qualità, in tempo e nei limiti di budget.

In tale contesto, necessariamente ci si imbatte nel cosiddetto "Software Configuration Management", ovvero il processo che mira a monitorare e controllare i cambiamenti nei prodotti che sono oggetto del lavoro. Risulta, quindi, fondamentale adottare tale processo per gestire tutte quelle situazioni in cui ci si trova in un team di sviluppatori, magari distribuito geograficamente, che deve lavorare in modo concorrente su uno stesso artefatto.

Il configuration management consente agli sviluppatori di tener traccia delle modifiche. In particolare, il sistema viene visto come un insieme di configuration items che vengono revisionati in modo indipendente.
Questa organizzazione in versioni permette agli sviluppatori non solo di tenere trac-

cia delle varie modifiche effettuate, ma anche di ripristinare uno stato ben definito del sistema quando esse non riescono.

È facile capire, dunque, l'essenziale utilità di tale metodologia, soprattutto nell'ambito della gestione dei flussi evolutivi dello sviluppo software.

1.2 Motivazioni e obiettivi

Una grande parte del recente lavoro nell'ambito dell'ingegneria del software si basa sull'estrazione dei dati delle diverse versioni software, analizzando quali modifiche sono state apportate a un sistema, da chi, quando, e dove. Tali informazioni possono essere utilizzate per prevedere modifiche correlate [2], per prevenire difetti futuri [3], per analizzare a chi dovrebbe essere assegnato un particolare compito [4], o semplicemente per ottenere approfondimenti su progetti specifici. Tutti questi studi dipendono dall'accuratezza delle informazioni estratte, accuratezza spesso minacciata da una erronea gestione dello sviluppo software.

Secondo una serie di studi, una fonte significativa di alterazioni nella storia dei progetti software risulta essere la presenza dei Tangled Code Changes. Si verifica un Tangled Code Change ogni qualvolta uno sviluppatore raggruppa una quantità consistente di modifiche in un unico commit poco significativo.

Una tipica situazione che porta alla creazione di un Tangled Change è la fase di correzione. Ad esempio, qualora uno sviluppatore si ritrovi a risolvere tre bug differenti, è preferibile avere tre commit separati, ognuno con un messaggio significativo, piuttosto che un unico macro-commit, identificato da un messaggio poco chiaro, che comprometterebbe anche eventuali operazioni di rollback.

La presenza di Tangled Changes non solo compromette una corretta gestione dello sviluppo software, ma ostacola fortemente l'accuratezza delle analisi delle revisioni in svariati approcci di Mining di Repository Software.
Conseguentemente la possibilità di identificare la presenza di tali alterazioni all'interno di una repository, introdurrebbe un utile strumento per verificare la precisione e la finezza delle valutazioni, ai fini di una migliore qualità del software.

1.3 Risultati

In questo lavoro di tesi è stato progettato e sviluppato un software Java, lanciabile da linea di comando, che consenta di identificare la presenza di Tangled Code Changes. Il tool, previa installazione, può essere eseguito dal terminale dello sviluppatore. Una volta eseguito, sarà possibile inserire l'URL di una repository remota e uno specifico commit code, e il programma eseguirà un algoritmo che, a partire dal commit originale, restituirà i file coinvolti negli eventuali Tangled Changes identificati e un'indicazione di quelli che sarebbero dovuti essere i corretti commit da sottomettere.

1.4 Struttura della tesi

Gli argomenti introdotti sono trattati in dettaglio nei seguenti capitoli che compongono la tesi:

- **Capitolo 2:** presenterà un'analisi dettagliata dello stato dell'arte, concentrandosi sul problema in esame e quale punto è stato raggiunto dalla ricerca sul tema;
- **Capitolo 3:** avrà lo scopo di analizzare e approfondire una ricerca che descrive una possibile tecnica per identificare i Tangled Changes, la cui analisi sarà l'oggetto fondamentale di questo lavoro;
- **Capitolo 4:** mostra nel dettaglio le specifiche del software realizzato, le tecnologie utilizzate e saranno descritte nel dettaglio le scelte implementative per la realizzazione dell'algoritmo di detection, descritto nella sezione precedente;
- **Capitolo 5:** trarrà le conclusioni finali del lavoro descritto, proponendo spunti e riflessioni per eventuali sviluppi futuri e approfondimenti sul tema;

Capitolo 2

Background e stato dell'arte sui Tangled Changes

Questo capitolo illustra un approfondimento sullo stato dell'arte e i lavori presenti in letteratura sugli aspetti di ricerca trattati in questo lavoro di tesi.

2.1 Version Control System

Molte ricerche nell'ingegneria del software si sono concentrate sul miglioramento della qualità del software e sull'automazione del processo di manutenzione per ridurre i costi e mitigare le complicazioni associate al processo evolutivo.

Negli ultimi anni le aree di ricerca relative al mining di repository sono state molto attive e hanno attirato una significante attenzione.
Le informazioni contenute all'interno dei commit presenti nei sistemi di versioning Open-Source sono frequentemente usati come supporto alla ricerca nell'ambito dello sviluppo software. A tal proposito molti ricercatori hanno provato a mettere in relazioni determinati commit a determinate attività. Ad esempio, è più probabile che commit di grandi dimensioni siano originati da attività di sviluppo del codice, mentre quelli piccoli siano legati ad attività di manutenzione. Tuttavia, queste distinzioni sono vaghe, perché non esiste una definizione coerente di cosa sia un commit piccolo o grande.

Data la loro natura accessibile, i ricercatori spesso analizzano i log della cronologia dei sistemi di versioning, invece di elaborare tutte le revisioni di ogni singolo file di un sistema [5], è molto importante quindi disporre di commit chiari e definiti.

I sistemi di versioning sono spesso utilizzati come punto di riferimento, poiché ricchi di informazioni che possono aiutare gli sviluppatori ad acquisire conoscenze o principi che saranno utili per le attività di ingegneria del software [6] [7].
Tuttavia, nonostante il principio generalmente accettato che un commit dovrebbe includere solo modifiche per una singola attività, i programmatori sottomettono spesso modifiche intricate (cosiddette tangled), che vanno a inglobare due o più attività differenti [8].

Con sistema di versioning, o VCS (Version Control System), si intende uno strumento software che è in grado di monitorare e gestire le modifiche apportate a un filesystem, nonché di offrire utility collaborative che consentono di condividere e integrare tali modifiche con altri utenti di VCS.

Scopo di un VCS è quello di realizzare una corretta gestione delle modifiche garantendo tre caratteristiche essenziali: reversibilità, concorrenza e annotazione.
La reversibilità è la capacità di un VCS, di poter sempre tornare indietro in un qualsiasi punto della storia del codice sorgente.
La concorrenza permette a più persone di apportare modifiche allo stesso progetto, facilitando il processo di integrazione di artefatti di codice sviluppati da più sviluppatori.
L'annotazione è la funzione che consente di aggiungere spiegazioni e riflessioni ulteriori alle modifiche apportate.

A livello di filesystem, un VCS tiene traccia delle operazioni di aggiunta, eliminazione e modifica applicate a file e directory. A livello dei singoli file del codice sorgente, invece, tiene traccia delle aggiunte, delle eliminazioni e delle modifiche delle righe di testo all'interno di un determinato file.

I vantaggi offerti da un VCS in termini di ottimizzazione del flusso di lavoro collaborativo all'interno di un team software sono inestimabili. Nessun progetto software che includa più di uno sviluppatore preposto alla gestione dei file del codice sorgente può fare a meno di un VCS. Tuttavia, anche quei progetti che hanno un unico manutentore possono trarre enormi vantaggi dall'uso di un sistema di questo tipo.
Infatti, un VCS, permette di essere integrato con sistemi in cloud, permettendo così di avere un backup sicuro del codice attuale (e di tutte le versioni del tuo codice), confrontarsi con altri sviluppatori e contribuire ad altri progetti.

La figura 2.1 rappresenta uno schema di collaborazione tramite VCS.

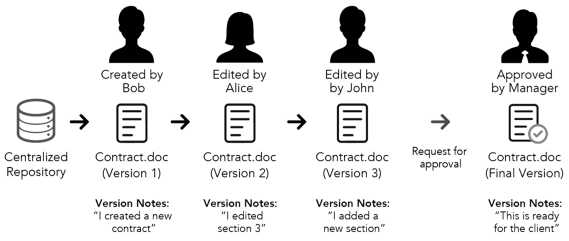

Figura 2.1: Collaborazione tramite Version Control System.

2.1.1 Git

Questo lavoro di tesi è concentrato sull'interazione con il sistema di controllo di versione Git [9], in particolare si farà riferimento a repository remote su GitHub.

Git è un sistema di controllo delle versioni, gratuito e open source progettato per gestire tutto, dai progetti piccoli a quelli molto grandi con velocità ed efficienza. GitHub, invece, è un servizio di hosting Git basato su web, che offre tutto il controllo di revisione distribuito e la funzionalità di gestione del codice sorgente di Git.

Per poter comprendere a fondo le potenzialità di Git è necessario, però, definire una serie di termini e comandi chiave:

- **Repository:** è la cartella che tiene traccia di tutti i cambiamenti fatti al software in esame, costruendo una storia che si evolve nel tempo;
- **Commit:** è l'azione che permette di aggiungere, rimuovere e modificare i file nella repository;
- **Branch:** è un puntatore a un commit, man mano che viene creato un nuovo commit, il branch corrente si sposta e punta al nuovo commit creato. Git usa il puntatore speciale HEAD per capire qual è il branch corrente;
- **Merge:** permette di incorporare uno o più commit di un branch in un altro;

2.2 L'impatto dei Tangled Code Changes

Ci sono diverse ragioni per cui gli sviluppatori tendono ad "aggrovigliare" task non correlati in un singolo commit.
Quando si analizzano insiemi di modifiche così intricati, non è facile determinare quali artefatti di codice sono stati modificati e per quale task. In molti casi, gli sviluppatori stessi potrebbero essere in grado di separare due modifiche non correlate e rivalutarle separatamente.

Secondo K. Herzig e A. Zeller [8] il bias introdotto dai Tangled Changes è significativo e spesso la presenza di Tangled Code Changes è identificabile direttamente effettuando un'analisi del commit message.
In particolare, effettuando un approfondimento mirato ad identificare la presenza di Tangled Code Changes in cinque progetti open source, hanno manualmente controllato tutti quei commit message che indicavano chiaramente che le modifiche eseguite riferivano a più task.
Tali messaggi infatti sono tipicamente caratterizzati dalla presenza di segnalazioni a più problemi (es. "Corretto bug x, aggiunta classe Y, miglioramento metodo Z"), oppure dall'indicazione di lavoro extra impiegato lungo la correzione del problema (es. "Risoluzione problema Y [...] e refactoring "), soprattutto legato ad operazioni di refactoring e cleanup [10].
La figura 2.2 rappresenta i risultati della ricerca condotta su repository Open-Source.

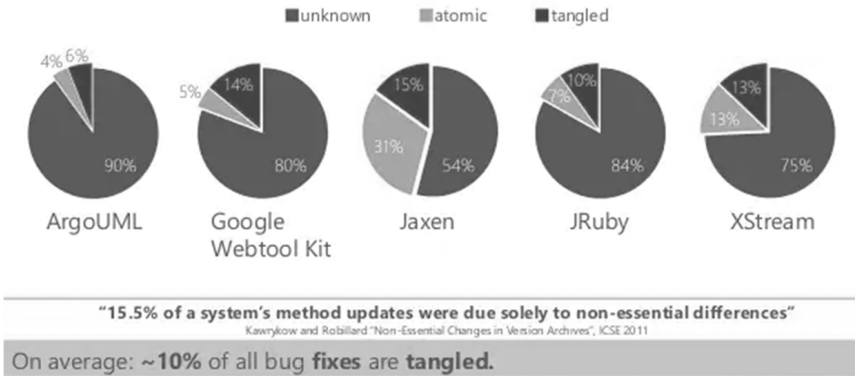

Figura 2.2: Risultati ricerca Tangled Changes in repository Open-Source.

I risultati di questa analisi hanno confermato che quello dei Tangled Changes non è un problema puramente teorico.

Il 6%-12% di tutti i cambiamenti con riferimenti a segnalazioni di problemi, risultavano essere tangled, introducendo cosi un bias in qualsiasi analisi della rispettiva cronologia delle modifiche.
La frazione di cambiamenti riferiti a risoluzioni di bug, invece, risultavano essere tangled nell'11,9% dei casi.

2.3 Ostacoli creati dai Tangled Changes

I commit che includono Tangled Changes possono creare numerosi ostacoli allo sviluppo software [11].
I Tangled Changes, innanzitutto, sono inappropriati per gestire il merging tra il codice di una branch con quello di un'altra. Se un commit include solo quelle modifiche di cui si ha intenzione di effettuare il merge, dovrebbe essere possibile eseguire tale operazione efficientemente. Tuttavia, se un commit contenesse Tangled Changes, risulterebbe complicato effettuare il merging solo di quei cambiamenti legati a specifici task.
In questi casi si dovrebbe, quindi, scegliere tra due strade:

- Effettuare un merging manuale, senza utilizzare le funzionalità apposite messe a disposizione dai Version Control Systems;
- Cancellare manualmente il codice extra, dopo aver effettuato le operazioni di merging;

Per entrambi i casi sarebbe, però, necessario eseguire operazioni manuali.

In secondo luogo, le operazioni di ripristino (es. comando "revert" in Git) per uno specifico task su un commit tangled, risultano essere difficoltose. Quando viene utilizzata una funzione di ripristino delle modifiche in un sistema di controllo di versione, infatti, tutte le modifiche nel commit in esame vengono annullate, andando cosi potenzialmente ad inficiare su una delle caratteristiche fondamentali dei VCS, ovvero la reversibilità.

Infine, è difficile riesaminare il codice di un commit che contiene Tangled Changes, in quanto significherebbe cercare le modifiche specifiche per il task in esame, all'interno di tutto il commit.

2.4 Tecnica di identificazione proposta da Kirinuki et al.

Con lo scopo di evitare problemi relativi ai Tangled Code Changes Kirinuki, Hiroyuki, et al. hanno effettuato una ricerca [12] presso l'università di Osaka, per progettare un tool in grado di avvisare lo sviluppatore riguardo la presenza di Tangled Code Changes nelle repository di codice.

L'idea di base proposta consiste nell'utilizzare un pattern di modifiche al codice. La ricerca ha infatti rivelato che alcuni pattern di cambiamenti appaiono frequentemente in diversi commit.

La tecnica consiste in due fasi separate:

- Creare un database di *template*;
- Verificare se le modifiche effettuate dallo sviluppatore siano tangled;

Per quanto riguarda la creazione del database, le modifiche vengono estratte da ogni coppia di due revisioni consecutive. Il processo di estrazione si compone di diverse fasi:

- STEP 1: i file sorgente modificati tra due revisioni successive vengono analizzati e viene estratto un elenco di istruzioni singole del codice sorgente;
- STEP 2: le due liste di istruzioni vengono confrontate con l'algoritmo della più lunga sotto sequenza comune e si individuano le differenze tra loro. Ognuna delle differenze è definita *pattern*.
- STEP 3: i nomi delle variabili nei pattern vengono sostituiti con dei *token* speciali.

Un insieme di *pattern*, identificato da ogni coppia di due revisioni consecutive, si definisce *tamplate*. Tutti i *template* identificati vengono memorizzati in un database.

La figura 2.3 rappresenta un esempio di come vengono creati i pattern che compongono il database.

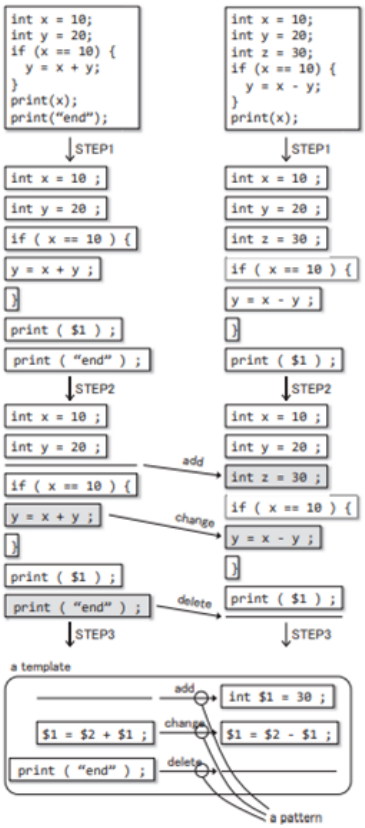

Figura 2.3: Fasi della creazione del database di Kirinuki, Hiroyuki, et al.

La tecnica proposta da Kirinuki, Hiroyuki, et al. mira ad estrarre le modifiche dello sviluppatore, confrontando i file sorgente nella working directory, con la revisione in testa alla repository in esame.
La tecnica verifica poi se qualcuno dei *template* nel database è ricorrente nei suoi ultimi cambiamenti.

Se non viene individuato nessun *tamplate* allora le modifiche non risultano essere tangled e lo sviluppatore può eseguire il commit con la sicurezza che siano legate ad un singolo task.
Se, invece, viene identificato un riscontro nel database, il sistema restituisce un feedback allo sviluppatore, così che possa soffermarsi sulle modifiche che sta per sottomettere e valutare se siano effettivamente tangled.

Capitolo 3

L'algoritmo di identificazione di K. Herzig e A. Zeller

Questo capitolo mira ad approfondire l'articolo "The Impact of Tangled code Changes" di Kim Herzig e Andreas Zeller [8], il quale getta le basi sullo studio dei tangled changes, ed è stata la fonte principale di questo lavoro di tesi, in quanto propone un'idea di algoritmo di identificazione automatica di Tangled Changes.

3.1 Introduzione all'algoritmo

Come detto finora, gli insiemi di Tangled Changes portano a dannosi bias che influenzano negativamente la storia delle versioni. Per ridurre tali bias sarebbe necessario, in un primo momento, identificare tali insiemi e, successivamente, effettuarne l'untangling in modo da ottenere partizioni di modifiche individuali.

Il processo di detection, e conseguentemente quello di risoluzione, è molto complesso. Di conseguenza un algoritmo di untangling dovrebbe fare affidamento su euristiche, che offrano un'idea del perché due modifiche al codice dovrebbero essere separate. Chiaramente, eliminare totalmente il fenomeno dei Tangled Changes è impensabile, in quanto ci si basa sempre su dati approssimativi. Tuttavia, si ha l'obiettivo di ridurre, laddove possibile, i problemi da essi causati.

L'approccio proposto da Herzig e Zeller mira a progettare un algoritmo che sia interamente automatico e semplice allo stesso tempo il quale, a partire da un insieme di modifiche tangled, restituisca una serie di partizioni untangled.

3.2 Confidence voters

Per combinare e valutare i diversi aspetti che riguardano le dipendenze e le relazioni tra le varie operazioni di cambiamento, l'algoritmo, per decidere se due operazioni di cambiamento dovrebbero stare insieme o meno, fa affidamento ad una serie di indicatori definiti ConfVoter.

Ogni ConfVoter prende in input una coppia di operazioni di modifica e restituisce un valore tra 0 e 1, che suggerisce il grado di correlazione tra le due operazioni. Un indicatore di 1 suggerisce che le due operazioni dovrebbero stare nella stessa partizione. Al contrario lo 0 suggerisce che le due operazioni sono poco correlate.
La figura 3.1 rappresenta uno schema di come tali ConfVoter vengono utilizzati ai fini dell'algoritmo.

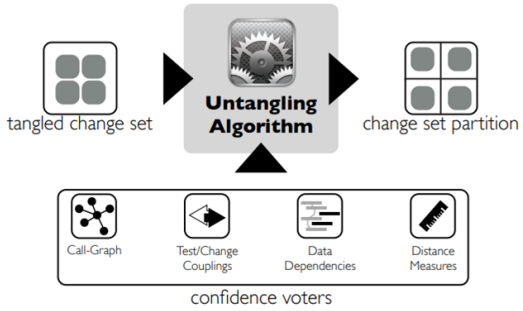

Figura 3.1: Schema del funzionamento dei ConfVoter.

Prima di descrivere i vari ConfVoter teorizzati nel paper, è necessario fare una distinzione fra due categorie.
Da un lato abbiamo una serie di indicatori che mirano a specificare il grado di correlazione tra due operazioni di modifica che avvengono all'interno di uno stesso file. Dall'altro, invece, abbiamo degli indicatori che a partire dal percorso dei file modificati, valutano se questi sarebbero dovuti essere nella stessa partizione o meno.

Ogni voter da indicazione su uno specifico aspetto che riguarda la dipendenza tra operazioni di modifica, in particolare sono stati studiati 5 possibili indicatori:

- **PackageDistance:** se le due operazioni di modifica sono state applicate a file differenti, questo voter restituisce il numero di segmenti di nomi di pacchetto differenti, ottenuto confrontando il percorso dei due file in esame;

- **ChangeCoupling:** si applica a operazioni di modifica tra file separati. Calcola, infatti, il numero di volte in cui i file coinvolti sono stati modificati insieme, tenendo conto di tutta la storia della repository. Tale indicatore segue la metodologia [13] per cui, più spesso due file vengono modificati insieme, più è probabile che sia necessario modificarli insieme.
  La figura 3.2 rappresenta uno schema del funzionamento di questo voter;

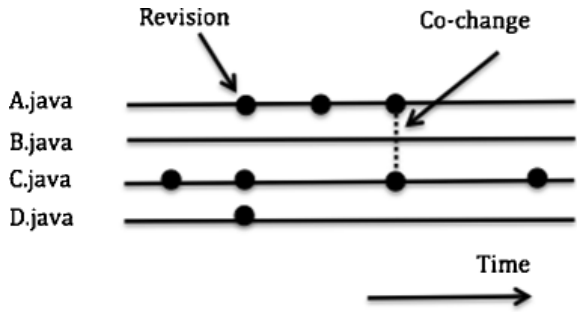

Figura 3.2: Schema del change coupling tra classi.

- **FileDistance:** non viene considerato se le due operazioni sono applicate a file differenti, altrimenti restituisce il numero di righe tra le due operazioni di modifica;
- **DataDependency:** restituisce 1, se entrambe le modifiche leggono o scrivono la stessa variabile, 0 altrimenti;
- **CallGraph:** identifica i metodi modificati all'interno del grafo di chiamata derivato dal progetto e restituisce la distanza tra due nodi del grafo;

Ai fini della realizzazione di una versione software utilizzabile di tale algoritmo, in questo lavoro di tesi è descritta la progettazione di un detector di Tangled Changes che si basi esclusivamente su operazioni di modifica eseguite su file differenti, andando a valutare esclusivamente i primi due ConfVoter descritti.

3.3 Funzionamento dell'algoritmo

Come già anticipato, l'algoritmo realizzato riceve in input un insieme di modifiche, costituito da tutti i file coinvolti nel commit in esame, e restituisce in output un insieme di partizioni indipendenti.

Lo scopo dell'algoritmo è quello di iterare su coppie di operazioni di cambiamento per capire se sono connesse e dovrebbero stare nella stessa partizione.
La figura 3.3 rende l'idea di come esso operi.

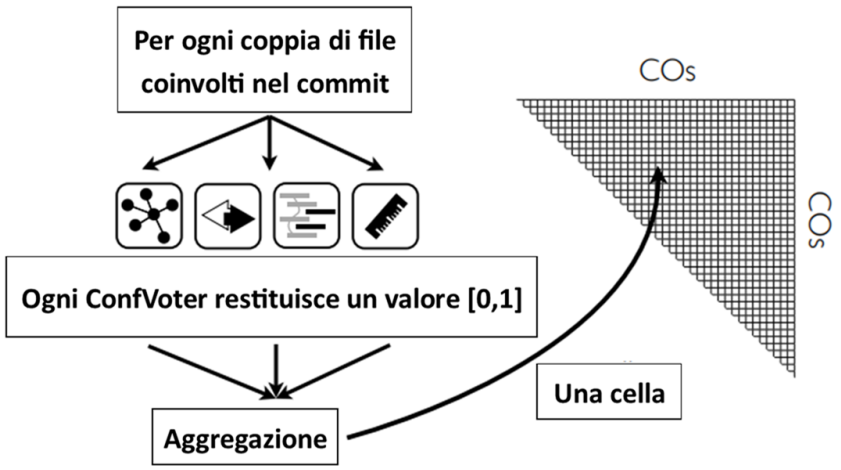

Figura 3.3: Schema del funzionamento dell'algoritmo di detection.

Di seguito, sono descritti nel dettaglio i passi eseguiti dall'algoritmo:

- **PASSO 1:** viene costruita una matrice triangolare $\mathcal{M}$, di dimensione $m \times m$, che contiene una riga e una colonna per ogni partizione del set di modifiche iniziale. Inizialmente la dimensione m sarà pari al numero dei file presenti nel commit in esame;
- **PASSO 2:** per ogni cella $[P_\mathrm{i}, P_\mathrm{j}]$, con $i < j <= m$, di $\mathcal{M}$, si calcola il ConfVoter che indica il grado di correlazione tra le due partizioni;
- **PASSO 3:** viene identificata in $\mathcal{M}$ la coppia $(P_\mathrm{s}, P_\mathrm{t})$ con il più alto confidence value e con $s \neq t$. Vengono allora eliminate le due righe e le due colonne corrispondenti a P_s e P_t, e vengono aggiunte una colonna e una riga per la nuova partizione $P_{\mathrm{m+1}}$, contenente l'unione di P_s e P_t;

- **PASSO 4:** si calcolano i confidence values tra la nuova partizione P_{m+1} e le rimanenti partizioni di $\mathcal{M}$. Per calcolare il valore di una partizione composta si segue la seguente formula:

$$Conf(P_1,P_2) = \text{Max}\{Conf(c_i,c_j) \mid c_i \in P_1 \wedge c_j \in P_2\}$$

- **PASSO 5:** si ritorna al **PASSO 2**;

L'algoritmo termina quando sarà raggiunto un definito numero di partizioni. Questo perché, se non si stabilisse una condizione di stop, l'algoritmo continuerebbe ad iterare sulle partizioni, effettuandone di volta in volta l'unione, fino a che si otterrà nuovamente la partizione originale.

Per definire il numero finale di partizioni, non vengono fornite specifiche indicazioni. Tuttavia, in linea generale sono proposte due tecniche.
La prima è quella di fissare, ogni volta che viene eseguito l'algoritmo, una condizione di stop definita a priori.
Una seconda consiste nel definire il potenziale numero di partizioni finali, effettuando un'analisi del commit message, cercando in esso eventuali indicazioni esplicite riguardo la presenza di più Tangled Changes.

Capitolo 4

ATaCoDe: un tool per l'identificazione di Tangled Code Changes

Dopo aver discusso nei capitoli precedenti di versioning e Tangled Changes, risulta immediatamente evidente come il connubio tra questi due importanti aspetti di Ingegneria del Software possa apportare interessanti conoscenze ai fini del ciclo di vita di un sistema software.

Proprio da questa idea si muove il lavoro che ci si accinge a descrivere, il cui scopo principale è stato realizzare il tool ATaCoDe (Automated Tangled Commit Detector), che potesse identificare automaticamente la presenza di Tangled Code Changes. Più nello specifico, grazie a questo lavoro di tesi, è stato possibile effettuare un analisi delle repository Open-Source disponibili su GitHub, al fine di ricevere un feedback sull'eventuale presenza di commit tangled.

Al fine di raggiungere questo obiettivo è stato necessario sviluppare un sistema che implementi le tecniche descritte dallo studio esposto nel capitolo precedente.
Di seguito saranno, quindi, descritti nello specifico le fasi di sviluppo affrontante e le componenti realizzate.

4.1 Conversione e aggregazione dei confidence values

Come indicato nella sezione antecedente, il funzionamento dell'algoritmo si basa sul calcolo, per ogni partizione, di due confidence values fondamentali: *PackageDistance* e *ChangeCoupling*.

La figura 4.1 rappresenta le classi ConfVoter.

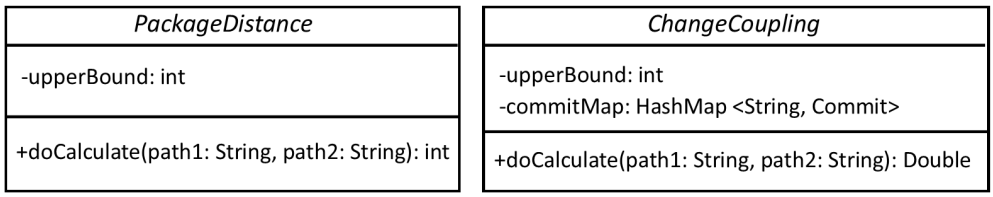

Figura 4.1: Class diagram dei ConfVoter utilizzati.

Tali ConfVoter di base possono potenzialmente assumere valori molto grandi, è quindi necessario attuare un processo di conversione per poter portare i valori reali all'intervallo [0,1].
Una volta ottenuti i valori corretti per i ConfVoter, questi sono stati aggregati in un unico valore, attraverso una media aritmetica.

Nella sezioni successive saranno descritte le varie fasi di conversione dei valori iniziali in quelli effettivamente utilizzabili ai fini del funzionamento dell'algoritmo.

4.1.1 PackageDistance

Il ConfVoter *PackageDistance*, presi in input due percorsi di file in un commit, restituisce il numero dei diversi segmenti di nomi di pacchetto tra i due paths.

Risulta evidente come il numero risultante da questo calcolo sia un intero positivo. Per convertire tale intero in un decimale compreso tra 0 ed 1, si va a dividere il valore calcolato per un limite superiore fissato, definito *UpperBound*. Tale limite, in particolare, è stato ottenuto andando ad individuare la lunghezza del più esteso percorso di file presente all'interno della repository.

InitialPackageDistance(path1,path2)=PackageDistance(path1,path2)/UpperBound

Ricordiamo poi che ai fini dell'algoritmo, il confidence value tra due partizioni deve restituire un'indicazione su quanto queste due siano legate, dato che ad ogni iterazione si individuano ed uniscono le due partizioni maggiormente correlate.
Alla luce di ciò è importante notare che un *InitialPackageDistance*, calcolato come appena descritto, più si avvicina al valore di 1, più da un segnale riguardo il fatto che i due file in esame sono molto distanti, e probabilmente non dovrebbero stare insieme.

È, quindi, necessario attuare un ulteriore conversione per fare in modo che un valore vicino all'uno, dia un'indicazione positiva riguardo la correlazione tra i due file in considerazione. Per far ciò quindi si effettua una semplice differenza andando a sottrarre ad 1 il valore del *InitialPackageDistance*.

FinalPackageDistance(path1,path2)=1-InitialPackageDistance(path1,path2)

. . . . . .

Di seguito è mostrato un esempio dei vari passaggi del calcolo di questo CONF-VOTER:

Supponiamo di avere due percorsi di file contenuti in un commit:

path1: ProjectName/A/B/C/Foo.java, ADD

path2: ProjectName/D/E/Bar.java, MODIFY

allora:

PackageDistance(path1,path2)=3;

Supponendo di avere un UpperBound di 5:

InitialPackageDistance(path1,path2)=3/5=0.6

FinalPackageDistance(path1,path2)=1-0.6=0.4

4.1.2 ChangeCoupling

Il ConfVoter *ChangeCoupling* presi in input due percorsi di file, restituisce un'indicazione su quante volte nella storia della repository i due file vengono modificati insieme.

Anche in questo caso è facile notare come potenzialmente il valore potrebbe assumere valori molto grandi, pertanto è stato necessario attuare una conversione anche in questo caso.

In maniera analoga a quanto fatto per il ConfVoter descritto precedentemente, andremo a dividere il valore dell'*InitialChangeCoupling*, per un determinato *UpperBound*. In particolare tale valore sarà dato dalla dimensione della lista contenente tutti i commit della repository in esame.

ChangeCoupling(path1,path2)=InitialChangeCoupling/UpperBound

. . . . . .

Mostriamo un semplice esempio di calcolo anche in questo caso:

Supponiamo di avere due classi del progetto in esame:

paht1: ProjectName/A/B/Foo.java

path2: ProjectName/C/D/Bar.Java

Ipotizziamo che i due percorsi siano presenti nello stesso commit 15 volte, avremo:

InitialChangeCoupling(path1,path2)=15

UpperBound=50

ChangeCoupling=15/50=0.30

4.2 Requisiti e architettura del sistema realizzato

Come descritto il sistema realizzato ha fondamentalmente due requisiti funzionali fondamentali:

- Identificare, a partire da una repository remota ed un commit in quest'ultima, i Tangled Code Changes presenti in essa;
- Visualizzare gli insiemi di modifiche districati generati dall'algoritmo, rappresentanti i singoli commit che si sarebbero dovuti eseguire, oltre ad una serie di informazioni riguardanti la repository in questione;

É importante tener presente che il tempo di risposta varia in base alle dimensioni del progetto da analizzare.
Il tool, realizzato interamente in linguaggio Java, può essere scaricato dagli utenti sotto forma di file .jar, essere installato ed eseguito da terminale.
Il software è stato realizzato facendo in modo che gli aspetti di input/output rispettassero le convenzioni delle interfacce a linea di comando.

La figura 4.2 fornisce una rappresentazione immediata di come funzionino le dinamiche del sistema.

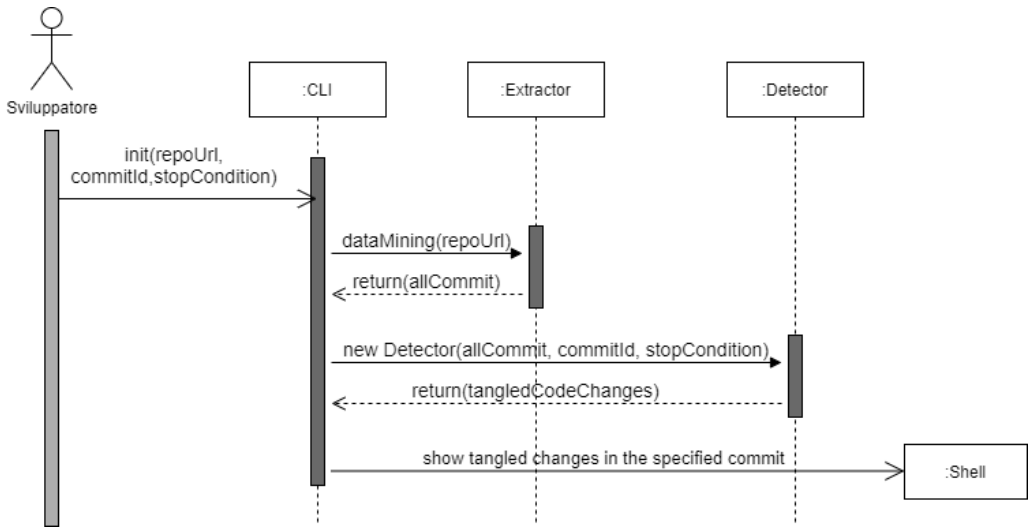

Figura 4.2: Sequence diagram del sistema.

- Lo sviluppatore, che è l'attore principale del sistema, inizia l'interazione inviando i dati necessari all'esecuzione;
- Il main, durante la fase di mining, raccoglie tutte le informazioni necessarie affinché possano essere utilizzate dall'identificatore di Tangled Changes.
- Vengono quindi inviati i dati di partenza per l'esecuzione dell'algoritmo di detection, il quale restituisce i Tangled Code Changes che verranno poi mostrati sul terminale.

Il sistema realizzato è suddiviso fondamentalmente in due layer, rappresentati nella figura 4.3.

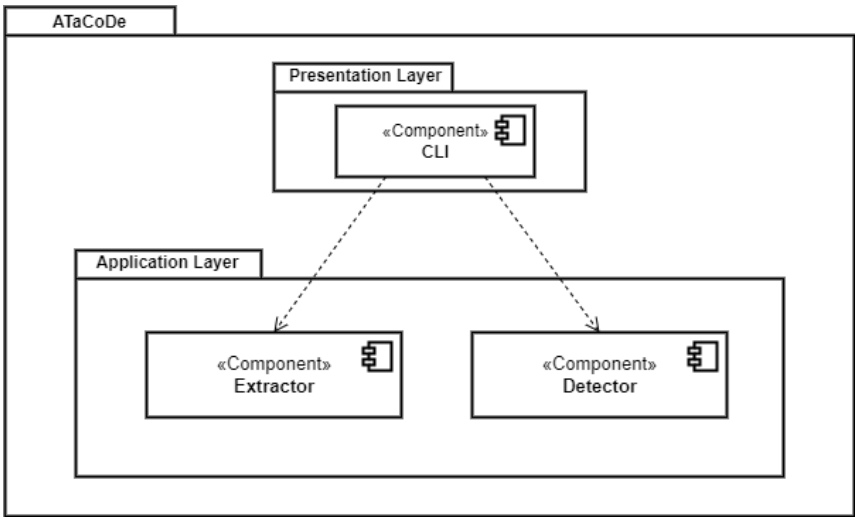

Figura 4.3: Rappresentazione grafica dei layer del sistema.

Presentation Layer: Rappresenta il livello più alto dell'applicazione, si occupa principalmente dell'interazione con l'utente e della visualizzazione dei contenuti. Esso è sostanzialmente costituito da un componente:

- **CLI:** si occupa della funzionalità più importante del tool, ovvero gestire l'interazione sviluppatore-sistema, attraverso un'interfaccia a linea di comando. Questo componente dipende sia dall'extractor, fondamentale per ottenere le informazioni da analizzare, che dal detector, necessario per lo sopo ultimo del software. Nella sezione successiva sarà descritta nel dettaglio la fase di sviluppo riguardante l'aspetto GUI.

Application Layer: Rappresenta la logica del sistema. Esso contiene i sottosistemi che si occupano di ricavare le informazioni ed analizzarle. In particolare questo livello è costituito da due componenti fondamentali:

- **Extractor:** è l'unita logica del sistema che si occupa di interagire con GitHub per ottenere tutti i dati necessari;
- **Detector:** è la componente fondamentale che si occupa di rilevare i Tangled Code Changes;

Per quanto riguarda i pacchetti che costituiscono il progetto, visualizzabili in figura 4.4 abbiamo due moduli fondamentali:

- **cli:** contiene la classe *Main.java*, il cui scopo è eseguire i comandi definiti da terminale;
- **core:** contiene tutta la logica applicativa del sistema e si compone dei package:
  - **beans:** include tutte quelle classi necessarie per incapsulare le informazioni riguardanti gli oggetti di base per lo sviluppo del progetto;
  - **confVoters:** contiene le classi che rappresentano i ConfVoter descritti nella sezione precedente e su cui l'algoritmo di detection fa affidamento;
  - **executor:** contiene le classi contenti la logica di mining, necessaria per estrarre le informazioni dalla repository in esame;
  - **untangler:** rappresenta il centro nevralgico di tutto il progetto, dato che include l'implementazione dell'algoritmo fondamentale, utilizzando tutte le strutture precedenti;

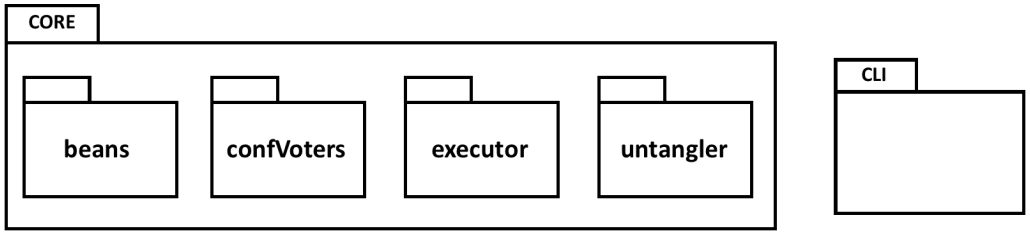

Figura 4.4: Schema della divisione in package.

Come già anticipato nelle sezioni precedenti, per poter eseguire l'algoritmo implementato è necessario interagire con un Version Control System, in modo da poter estrarre, da questo, le informazioni necessarie.
Questo progetto di tesi farà riferimento a repository remote open-source su GitHub.

Sono state quindi definite delle classi che potessero raccogliere le informazioni estratte. In particolare abbiamo:

- **core.beans.Commit:** contiene tutte le informazioni riguardanti il commit in esame.
  Si compone di:
  - Una variabile di tipo *String* che ne rappresenta l'id. Questo perché ogni commit, eseguito tramite GIT, viene identificato univocamente da un codice definito "commit hash". Mantenere tale codice è stato fondamentale per la memorizzazione di tutti i commit estratti dalla repository in esame, nonché ai fini dell'identificazione e analisi delle relazioni tra le classi JAVA presenti in essi;
  - Una variabile che memorizza l'autore del commit;
  - Una lista di *CommitChange*;
- **core.beans.CommitChange:** contiene le informazioni riguardanti tutti i file modificati in un determinato commit.
  Un singolo *CommitChange* è costituito da un path, rappresentate il percorso del file considerato e un'azione che ha lo scopo di memorizzare il tipo di operazione eseguita su quel file. Questo perché il VCS a livello di file-system mantiene traccia di tutte le aggiunte, rimozioni o modifiche che vengono eseguite;

La figura 4.5 rappresenta un class diagram per le classi che inglobano le informazioni di ogni commit tracciato da GIT.

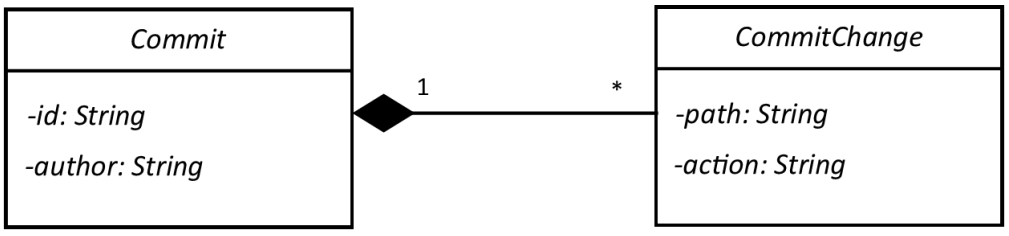

Figura 4.5: Class diagram per Commit.

La fase di estrazione dei dati iniziale è stata resa possibile attraverso lo strumento REPODRILLER.

RepoDriller[1] è un framework Java, creato con lo scopo di aiutare gli sviluppatori ad estrarre una serie di informazioni da qualsiasi repository Git, come commits, info sugli sviluppatori, modifiche, diff e codici sorgente.
Permette inoltre di esportare velocemente le informazioni su file di tipo CSV.

I motivi per cui è stato scelto questo tool sono molteplici. Innanzitutto RepoDriller è un framework estremamente intuitivo, offre tutte le API necessarie, non prevede significative fasi di pre-processing, o grandi databases, ed è molto flessibile. Per utilizzare RepoDriller è necessario specificare l'URL della repository remota sulla quale si vuole operare e il range di commit che si vuole analizzare, specificando uno o più filtri da applicare.
I filtri possono riguardare il tipo di file da escludere, quale branch considerare, escludere i commit coinvolti in operazioni di merging e cosi via.

All'interno della classe **core.executor.Extractor** è stato implementato il processo di mining, rappresentato nell'estratto di codice di seguito.
Dopo aver inserito le dipendenze Maven necessarie per l'integrazione di RepoDriller, è stato possibile implementare il metodo che consente l'estrazione, attraverso un *new RepositoryMining()*, che permette di configurare l'analisi che si intende effettuare specificando la repository remota, i commit da considerare e il tipo di *process*.

```
1  public HashMap<String, Commit> doExtract(){
2    new RepositoryMining()
3    .GitRemoteRepository.hostedOn(repoUrl).buildAsSCMRepository()
4    .through(Commits.all())
5    .process(new RepoVisitor(this))
6    .mine();
7    return commitList;
8  }
```

Listing 4.1: Configurazione del processo di mining.

Ciò che avviene in pratica è che RepoDriller aprirà la repository Git remota ed estrarrà le informazioni in essa. A questo punto il framework passerà ogni commit al *process* il quale, iterando sulla lista di tutte le modifiche estratte, costruirà i vari oggetti necessari per le fasi successive dell'esecuzione.

[1]https://github.com/mauricioaniche/repodriller.git

Una volta ottenute tutte le informazioni riguardanti la repository in esame e quelle relative allo specifico commit che si intende analizzare è stato possibile implementare l'algoritmo descritto nel capitolo precedente.

Come già spiegato, l'algoritmo opera in funzione di una matrice che per ogni coppia di indici valuta quanto i file ad essi associati siano correlati.
Ai fini implementativi, per simulare la costruzione della matrice sono state utilizzate delle classi: *Partition* e *PartitionItem*.
Più nello specifico un oggetto di tipo *Partition* può essere immaginato come la rappresentazione di una riga (dualmente di una colonna) della matrice, mentre un oggetto di tipo *PartitionItem* come una cella della matrice. Quindi risulterà che un oggetto di tipo *Partition* sarà costituito da una serie di *PartitionItem*, così come una riga (o una colonna) di una matrice è costituita da una serie di celle.
In particolare:

- **core.beans.Partition:** ha lo scopo di rappresentare una specifica partizione di modifiche, ovvero un determinato insieme di file coinvolti in un commit. La classe contiene:

  - Una variabile intera che rappresenta l'id, ovvero un identificativo per una specifica partizione;
  - Una lista di interi che hanno lo scopo di rappresentare gli indici a partire dai quali una specifica partizione è stata generata;
  - Una lista di stringhe costituita da tutti i path di file che costituiscono la partizione;
  - Un booleano che indica se la partizione è attiva o meno. La definizione di tale variabile è dovuta al fatto che ad ogni iterazione vengono eliminate due partizioni, ovvero quelle che risultano essere le più correlate, dalla matrice e ne viene aggiunta un'altra che ne costituisce l'unione. Considerando che vi è necessità di conservare il percorso dei file che costituiscono un insieme, piuttosto che cancellare e aggiungere partizioni si è optato per l'utilizzo di questo indicatore e booleano;
  - Una lista di *PartitionItem*;

  Tra i metodi definiti all'interno di questa classe, particolare rilevanza è data dal *findMax()*. Scopo di questo metodo è di identificare, tra tutti i *PartitionItem* presenti in una *Partition*, quello con il ConfVoter più elevato.

- **core.beans.PartitionItem:** rappresenta una singola cella della matrice costruita dall'algoritmo e contiene:
  - Due variabili intere che rappresentano l'indice di riga e di colonna del *PartitionItem* di riferimento;
  - Una variabile di tipo *Double* che contiene il ConfVoter relativo ad uno specifico *PartitionItem*;
  - La lista dei path che costituiscono un *PartitionItem*;
  - Un booleano che indica se il *PartitionItem* è attivo o meno;

La figura 4.6 rappresenta un class diagram delle classi Partition e PartitionItem.

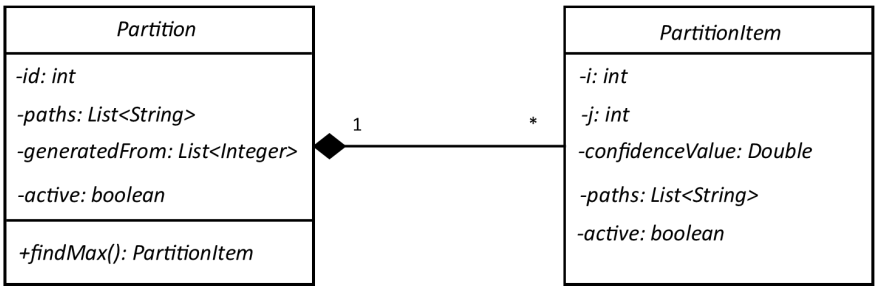

Figura 4.6: Class diagram per Partition.

Utilizzando queste classi, quindi, l'algoritmo realizzato inizia con il costruire una matrice iniziale. Per ogni file i-esimo presente all'interno della lista di tutti i cambiamenti del commit in esame, l'algoritmo crea una nuova *Partition* aggiungendo in essa il file corrente.
Fissata quindi una partizione, si creano i diversi j-esimi elementi che la compongono. Ogni *PartitionItem*, di riga i e colonna j, contiene il confidence value rappresentante il grado di correlazione tra l'i-esimo e il j-esimo file.

L'algoritmo esclude tutte quelle celle per le quali risulta che i>=j, in quanto il confidence value tra il file i-esimo e quello j-esimo risulta essere equivalente anche invertendone l'ordine.
Al termine di questa fase, definita nell'estratto di codice di seguito, si ottiene quindi una matrice triangolare, di dimensione pari al numero di file nel commit estratto, che conterrà i confidence value iniziali a partire dai quali opera l'algoritmo.

```
1    public void buildPartitionMatrix(){
2     for(int i=0; i<commit.getChanges().size(); i++){
3       Partition currentPartition=new Partition();
4       currentPartition.setId(i);
5       currentPartition.getGeneratedFrom().add(i);
6       currentPartition.getPaths().add(commit.getChanges().get(i).getPath());
7       for(int j=0; j<commit.getChanges().size(); j++){
8         PartitionItem currentPartitionItem= new PartitionItem();
9         currentPartitionItem.setPartitionIndex(i);
10        currentPartitionItem.setI(i);
11        currentPartitionItem.setJ(j);
12        String path1=commit.getChanges().get(i).getPath();
13        String path2=commit.getChanges().get(j).getPath();
14        double pd=packageDistance.doCalculate(path1,path2);
15        double cc=changeCoupling.doCalculate(path1,path2);
16        double confidence=(pdValue+ccValue)/2;
17        if(i>=j){currentPartitionItem.setConfidenceValue(-1);}
18        else {currentPartitionItem.setConfidenceValue(confidence);}
19        currentPartitionItem.getPaths().add(paht1);
20        currentPartitionItem.getPaths().add(path2);
21        currentPartition.getPartitionItemList().add(currentPartitionItem);
22      }
23      currentPartition.setActive(true);
24      partitionList.add(currentPartition);
25    }
26    startingMatrixSize=partitionList.size();
27 . }
```

Listing 4.2: Costruzione della matrice iniziale.

A questo punto comincia la fase cruciale dell'esecuzione, durante la quale le varie partizioni vengono accorpate per determinare gli insiemi di modifica tangled.

In particolare, finché le partizioni attive sono più di quelle che ci si prefissa di ottenere, l'algoritmo identifica il *PartitionItem* con valore più alto, disattiva le partizioni con gli indici ad esso associati e attiva un'ulteriore partizione contenente l'unione delle due precedenti.

Il valore associato a tale partizione è dato dal metodo *compositeValue(Partition, Partition)*, il cui ruolo è computare il valore definito nel capitolo 3:

$$Conf(P_1,P_2) = \text{Max}\{Conf(c_i,c_j) \mid c_i \in P_1 \wedge c_j \in P_2\}$$

Al termine restituisce le partizioni attive, ovvero gli insiemi di file che risultavano essere tangled, attraverso il metodo *getActivePartitions()*.

4.3 Realizzazione della Command Line Interface

Per gestire l'interazione con il sistema è stata utilizzata la libreria Apache Commons CLI[2], la quale fornisce un API[3] per effettuare il parsing degli elementi passati da linea di comando. In particolare, ci sono tre fasi per l'elaborazione della riga di comando: definizione, parsing e interrogazione.

- **Fase di definizione**
  Ogni riga di comando deve definire l'insieme di opzioni che verranno utilizzate per definire l'interfaccia dell'applicazione. La CLI utilizza la classe *Options*, come contenitore per le istanze *Option*.

- **Fase di parsing**
  La fase di parsing è quella in cui viene elaborato il testo passato all'applicazione tramite la riga di comando. Il testo viene elaborato secondo le regole definite dall'implementazione del parser.
  Il metodo fondamentale di questa fase è il *parse* che, definito su un *CommandLineParser*, accetta un'istanza di *Options* e un array di stringhe di argomenti e restituisce un oggetto di tipo *CommandLine*.

- **Fase di interrogazione**
  La fase di interrogazione è quella in cui l'applicazione interroga la *CommandLine* per decidere quale ramo eseguire in base alle opzioni booleane specificate e usa i valori delle opzioni per fornire i dati dell'applicazione.
  Il risultato della fase di interrogazione è che il codice utente è pienamente informato di tutto il testo che è stato fornito sulla riga di comando ed elaborato secondo le regole del parser e delle opzioni.

Seguendo le linee guida delle interfacce a linea di comando, tra le varie opzioni è stata innanzitutto definita quella d'aiuto. Viene mostrato il manuale d'aiuto quando sulla riga di comando non vengono specificate opzioni, viene specificato il flag $-h$ oppure il flag $--help$.

Il testo d'aiuto definito è molto conciso e include una descrizione di cosa fa l'applicazione, alcuni esempi di invocazione e una descrizione di quali flag sono disponibili e come possono essere invocati.

[2]https://commons.apache.org/proper/commons-cli/
[3]Application Programming Interface

La figura 4.7 rappresenta un esempio di manuale d'aiuto Git a linea di comando.

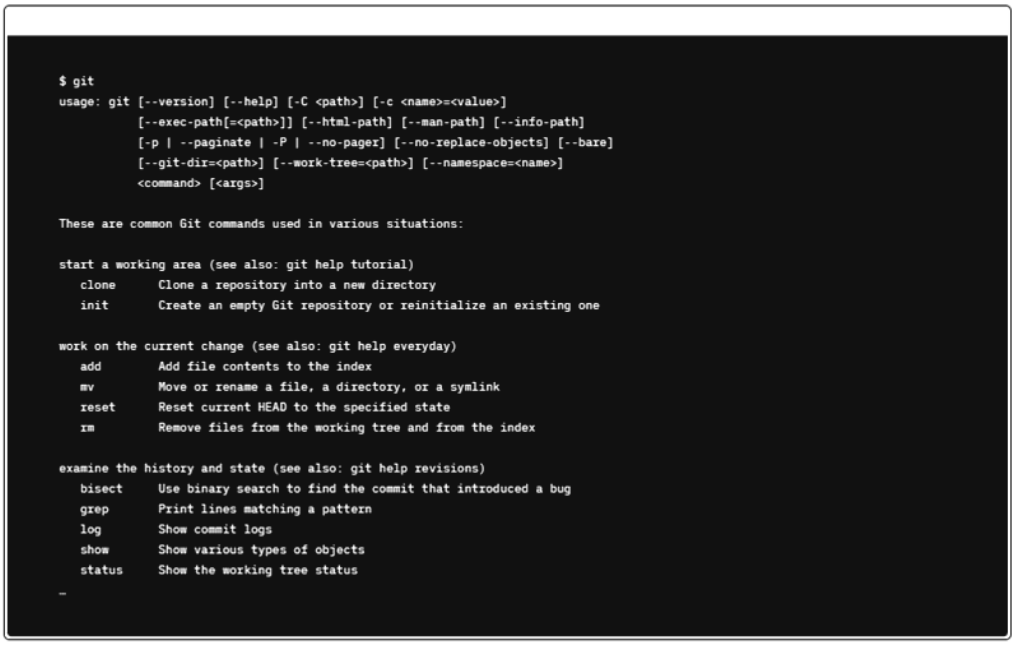

Figura 4.7: Comando -help su terminale Git.

Al fine di garantire un uso rapido del sistema e ottenere un output facilmente leggibile, sono state definite due ulteriori opzioni: $-s$ e $-d$.

L'opzione $-s$ (alternativamente $--show$) seguita dall'URL di una repo remota e un commit code, ha lo scopo di mostrare una serie di informazioni preliminari che riguardano le dimensioni di questi ultimi.

L'opzione $-d$ (alternativamente $--detect$) seguita dall'URL di una repo remota, un commit code e una condizione di stop, ha lo scopo di eseguire l'algoritmo di detection e mostrare il numero e i Tangled Changes rilevati.

Capitolo 5

Esempio di utilizzo

Nel capitolo precedente sono state analizzate nel dettaglio tutte le operazioni che hanno permesso la realizzazione di **ATaCoDe**[4]. A tal scopo è stato descritto ciò che è stato implementato, su quali studi ci si è basati e quali framework sono stati utilizzati. All'interno di questo capitolo verrà mostrato un esempio di utilizzo del tool a linea di comando.
Nella sezione a seguire sarà descritto infatti lo scenario relativo al rilevamento di Tangled Code Changes all'interno di un commit del progetto che si sta analizzando.

5.1 Identificazione di Tangled Code Changes

Lo sviluppatore, per utilizzare ATaCoDe. deve avere a disposizione:

- L'URL remoto di una repository pubblica su GitHub;
- Il *commit hash* di un qualsiasi commit eseguito sulla repository in esame;

Aperto il terminale, è possibile eseguire il tool con il comando $--help$, oppure $-h$, per comprenderne a fondo il funzionamento. La figura 5.1 rappresenta il risultato dell'invocazione di suddetto comando.

```
Usage: $ATaCoDe [-h] [-s repositoryUrl commitId] [-d repositoryUrl commitId stopCondition]
                    -h --help                                          to show help
                    -s --show      repositoryUrl commitId              to show generic information about repositoryUrl
                    -d --detect    repositoryUrl commitId stopCondition     to start detection

BUILD SUCCESSFUL in 469ms
```

Figura 5.1: Risultato comando -h di ATaCoDe.

[4]https://github.com/sambrosio17/TangledChangesUltimate - ATaCoDe.jar

Utilizzando il comando $--show$, come nella figura 5.2, vengono fornite delle informazioni preliminari.

```
Your repository has: 78 commit(s)
Commit: 1ed94bc1a7b6fedb88b8d5daebbae9e075af0c1a has: 5 file(s)

BUILD SUCCESSFUL in 55s
```

Figura 5.2: Risultato comando -s di ATaCoDe.

É possibile, infine, eseguire il comando fondamentale $--detect$, oppure $-d$, accompagnato dall'URL della repository del progetto, l'hash code del commit in esame e un numero intero di partizioni, ottenendo il risultato in figura 5.3.

```
Your repository has: 78 commit(s)
Commit: 1ed94bc1a7b6fedb88b8d5daebbae9e075af0c1a has: 5 file(s)
Created untangled commmit(s): 2
Proposed untangled commit(s):
**************************
Untangled Commit #0
Paths:
    RistoManager/src/it/RistoManager/Control/AggiungiProdotto.java
    RistoManager/src/it/RistoManager/Control/EliminaUtente.java
**************************
Untangled Commit #1
Paths:
    RistoManager/src/it/RistoManager/Control/VisualizzaDati.java
    RistoManager/src/it/RistoManager/Control/ModificaProdotto.java
    RistoManager/src/it/RistoManager/Control/RimuoviProdotto.java
**************************

BUILD SUCCESSFUL in 43s
```

Figura 5.3: Risultato comando -d di ATaCoDe.

Vengono quindi mostrate le due partizioni identificate, all'interno delle quali sono presenti i vari percorsi di file presenti nel commit tangled originale.

Capitolo 6

Conclusioni e sviluppi futuri

In quest'ultimo capitolo vi sono alcune considerazioni finali riguardo il lavoro di tesi svolto. Verranno inoltre esposti alcuni spunti di riflessione importanti sui possibili sviluppi futuri che possono essere apportati.

6.1 Conclusioni

Mantenere una buona qualità di sviluppo software e contemporaneamente rispettare le scadenze prefissate per le consegne sono obiettivi che, senza l'aiuto di strumenti esterni, risultano molto difficilmente raggiungibili.
Da ciò si evince che esiste il bisogno di poter controllare in modo semplice la qualità dello sviluppo, tenendo sotto osservazione i fattori che sono stati esposti nei capitoli precedenti.
L'obiettivo del tool sviluppato è proprio questo, fornire agli sviluppatori uno strumento utile e facile da utilizzare, che consenta di fornire un'indicazione quanto più possibile affidabile riguardo la qualità delle versioni software che si rilasciano nel tempo.
In definitiva, i contributi principali offerti da questo lavoro di tesi sono:

- Un tool eseguibile, che fornisce una visualizzazione semplice e intuitiva riguardo la qualità della storia dei propri progetti software;
- Una descrizione dettagliata del lavoro svolto, con esempi e scenari d'uso, che supporti gli sviluppatori, e i futuri fruitori del sistema, nella comprensione del funzionamento del sistema e li aiuti ad interfacciarsi con esso;
- Un approfondimento riguardo un miglior approccio allo sviluppo software, ottenibile ponendo attenzione al frequente fenomeno dei Tangled Code Changes;

6.2 Sviluppi futuri

I risultati ottenuti attraverso questo lavoro, non sono sicuramente definitivi.

Durante la descrizione del lavoro svolto si è più volte sottolineato che ATaCoDe può rilevare i presunti Tangled Code Changes presenti in una repository GitHub.
Da qui è possibile definire una prima banale evoluzione del sistema, rendendolo compatibile con altri VCS.

Altro aspetto interessante riguarda le definizione della condizione di stop dell'algoritmo. Attualmente essa viene definita a priori, forzando in un certo senso la ricerca dei Tangled Changes. Si potrebbe effettuare uno studio, e realizzare un'estensione per il sistema, che effettui un parsing del commit message per riscontrare in esso anomalie riguardo le modifiche realmente effettuate.

Un altro spunto di riflessione riguarda il tipo di modifiche considerate. Nel capitolo 3 è stato descritto l'algoritmo fondamentale su cui tutto il lavoro si basa ed è stato spiegato come, oltre a valutare le relazioni tra coppie di file, fosse possibile analizzare modifiche all'interno di uno stesso file.
Sarebbe possibile estendere il funzionamento del sistema a quelle modifiche che riguardano specifiche coppie di cambiamenti all'interno di un singolo file, per valutare se anche queste ultime siano tangled.

Nel lavoro di tesi di S. Ambrosio [14], è stato realizzato PangeaUntangler.
Un software che attraverso ATaCoDe non si limita solo a verificare la presenza di Tangled Changes, ma che ne intercetta la presenza al momento effettivo del commit e che offre una risoluzione Just-In-Time del problema, prima che i cambiamenti siano definitivi.

Capitolo 7

Ringraziamenti

Vorrei dedicare questo spazio a chi, con dedizione e pazienza, ha contribuito in qualche modo al raggiungimento di questo traguardo.

Ringrazio innanzitutto i miei genitori, non solo per avermi supportato economicamente in questi tre anni (e spero per qualche altro ancora), ma soprattutto per avermi permesso di concludere questo traguardo e di appoggiarmi in ogni mia scelta di vita.
Grazie perché, nonostante gli ormai consueti "se non vuò coglie li pummarore, viri di sturia" oppure "se nun tieni capo, la zappa sa andò staie", mi avete fatto capire il valore e l'importanza delle cose, e se oggi sono il *Dottore* che sono, in parte è anche merito loro.

Ringrazio mia sorella Marina, mia complice e regalo più grande che potessi ricevere nella vita. La ringrazio per aver sempre creduto in me e nelle mie capacità, perché anche nei momenti in cui pensavo di non essere abbastanza, è sempre stata li pronta con un "daje che sei bravissimo, siamo fieri di te, chi se ne frega di un 21".
Ringrazio anche il mio storico cognato Valerio, perché con i suoi immancabili e continui "un altro anno... dentro o fuori!", è sempre riuscito a tirarmi su l'umore, anche quando di umore positivo ne avevo poco.

Ringrazio i miei nonni, anche chi non è qui ma sono sicuro stia facendo il tifo per me, per essere sempre stati orgogliosi di me e dei miei traguardi.
Grazie per l'affetto e il sostegno che non mi hanno fatto mai mancare.

Ringrazio Flor, un'amica, una spalla, un pilastro e tanti altri aggettivi positivi, perché è sempre, e sottolineo sempre, stata lì a sopportare e ad ascoltare ogni mio mental-breakdown, ogni mio sfogo e ogni mia insicurezza.
Grazie perché, anche quando immagino sia stato tremendamente difficile non insultarmi per tutte le volte che "ho preso 28, ma meritavo 30", era li a gioire dei miei successi come se fossero anche i suoi.

Ringrazio i miei colleghi e amici Salvatore, Maria Giovanna, Alessia e Raffaele, che hanno reso unici, indimenticabili e pieni di "malattie" questi anni.
Salvatore, per aver affrontato con me questi ultimi step del percorso universitario, per aver retto tutte le mie ansie, i miei complessi e la mia connessione discutibile.
Maria Giovanna che riesce sempre a farmi ridere, essere l'anima della festa e che dopotutto ha il merito di averci unito in un unica piccola famiglia deviata.
Alessia, che per ogni esigenza sa sempre cosa dire e fare e per ricordarmi sempre di essere meno "l'ansia fatta persona".
Raffaele, che nonostante sia l'ultimo arrivato, ci ha permesso di condividere momenti divertenti e stupefacenti.

Ringrazio tutti quelli che in qualche modo hanno sempre avuto un pensiero per me: tutti i miei fidati amici, i miei amorevoli zii, i miei preziosi cugini e i nuovi piccoli ultimi arrivati, per ricordarmi da dove sono partito.

Ecco, ci siamo, questo è il momento in cui concludo dicendo quanto questa laurea sia anche vostra e del vostro affetto, ma non lo farò, perché diciamocelo gli esami li ho fatti io. Pertanto con la modestia che da sempre mi contraddistingue faccio un ultimo affettuoso, caloroso e doveroso ringraziamento alla più meritevole delle persone... ME STESSO.

Bibliografia

[1] IEEE Standards Committee et al. Ieee std 610.12-1990 ieee standard glossary of software engineering terminology. *online] http://st-dards. ieee. org/reading/ieee/stdpublic/description/se/610.12-1990 desc. html*, 1990.

[2] Thomas Zimmermann, Andreas Zeller, Peter Weissgerber, and Stephan Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.

[3] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse. In *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*, pages 9–9. IEEE, 2007.

[4] Lyndon Hiew, Gail C Murphy, and John Anvik. Who should fix this bug? In *Software Engineering, International Conference on*, pages 361–370. IEEE Computer Society, 2006.

[5] Lile P Hattori and Michele Lanza. On the nature of commits. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering-Workshops*, pages 63–71. IEEE, 2008.

[6] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th international conference on Software engineering*, pages 181–190, 2008.

[7] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. Bug prediction based on fine-grained module histories. In *2012 34th international conference on software engineering (ICSE)*, pages 200–210. IEEE, 2012.

[8] Kim Herzig and Andreas Zeller. The impact of tangled code changes. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 121–130. IEEE, 2013.

[9] Scott Chacon and Ben Straub. *Pro git*. Apress, 2014.

[10] Sunghun Kim, E James Whitehead, and Yi Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on software engineering*, 34(2):181–196, 2008.

[11] Shinpei Hayashi and Motoshi Saeki. Recording finer-grained software evolution with ide: An annotation-based approach. In *Proceedings of the Joint ER-CIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, pages 8–12, 2010.

[12] Hiroyuki Kirinuki, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. Hey! are you committing tangled changes? In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 262–265, 2014.

[13] Gustavo Ansaldi Oliva and Marco Aurélio Gerosa. Change coupling between software artifacts: learning from past changes. In *The art and science of analyzing software data*, pages 285–323. Elsevier, 2015.

[14] Salvatore Ambrosio. *PangeaUntangler: una Tecnica Just-In-Time per la Risoluzione di Tangled Commit*. UNISA, 2021.