

Relazione progetto Ingegneria dei sistemi distribuiti e laboratorio

Università degli studi di Catania

Dipartimento di Matematica ed Informatica

Corso di Laurea Magistrale in Informatica

Progetto: Code Clone Refactoring

Studente: Luca Strano

Matricola: 100052662

Anno: 2022/2023

Introduzione

L'obiettivo del progetto è la creazione di uno strumento in grado di analizzare il codice sorgente di un'applicazione software scritta in java ed effettuare un'analisi di similarità tra i metodi utilizzando l'algoritmo della distanza di Levenshtein per verificare la presenza di eventuali parti di codice ridondanti all'interno dell'applicativo.

Per l'analisi del codice si è fatto uso della libreria Javaparser.

Descrizione del problema e soluzione adottata

Nel contesto dello sviluppo software, è fondamentale scrivere del codice pulito, efficiente e facilmente manutenibile. Uno dei principi chiave della programmazione è la ridondanza minimale, il cui scopo è quello di evitare la duplicazione di codice all'interno del nostro applicativo; l'identificazione di metodi ridondanti è quindi un passo importante per migliorare la qualità del codice e ottimizzarne le prestazioni.

L'utilizzo di Javaparser ha consentito di analizzare in maniera capillare il codice sorgente di una classe Java e confrontare i metodi presenti al suo interno al fine di determinarne la similarità tra ogni possibile coppia, questa può essere valutata in base a vari criteri, quello preso in considerazione in questo progetto consiste nell'effettuare un'analisi di similarità tra i metodi basandosi sul loro corpo e quindi estraendo per ognuno una lista che conterrà la tipologia degli statement che ne fanno parte.

Dati 2 metodi quindi, vengono estratte le rispettive liste, queste vengono poi confrontate tramite una rivisitazione dell'algoritmo della distanza di Levenshtein (che viene tipicamente utilizzato per calcolare la distanza tra due stringhe), il risultato di questo algoritmo sarà una percentuale di similarità tra le 2 liste, più alta è la

percentuale più simili ovviamente saranno i corpi dei 2 metodi presi in considerazione.

Una volta identificati i metodi ridondanti, possono essere prese le opportune decisioni di refactoring per eliminare l'eventuale duplicazione di codice. Questo può comportare inoltre la creazione di metodi ausiliari per la condivisione di codice comune o l'applicazione di design pattern appropriati.

Analisi e progettazione

Fasi di sviluppo

La realizzazione del progetto è stata suddivisa in due fasi al fine di garantire una progressione logica nello sviluppo.

Una prima parte è stata dedicata allo studio della libreria Javaparser e di tutti i costrutti che questa mette a disposizione. Dopo aver acquisito una comprensione dettagliata delle sue funzionalità sono state implementate le funzioni che consentono di estrarre la lista degli statement di ciascun metodo della classe analizzata. Gli statement rappresentano le istruzioni individuali che compongono il corpo di un metodo; estraendo e memorizzando tali statement è stata quindi creata una base solida per l'analisi successiva.

La seconda parte del progetto è stata dedicata maggiormente all'analisi di similarità tra i metodi utilizzando i dati estratti nella fase precedente. Questo confronto ha coinvolto criteri come la struttura sintattica e l'ordine degli statement per ogni metodo, è stato quindi implementato l'algoritmo della distanza di Lievenshtein per ottenere una percentuale di similarità tra essi. Tutte le coppie di metodi che presentano una percentuale di similarità al di sopra di una certa soglia, ad esempio, potrebbero essere considerate ridondanti, questo permetterebbe allo sviluppatore di prendere alcune decisioni di refactoring del codice in merito ai risultati ottenuti.

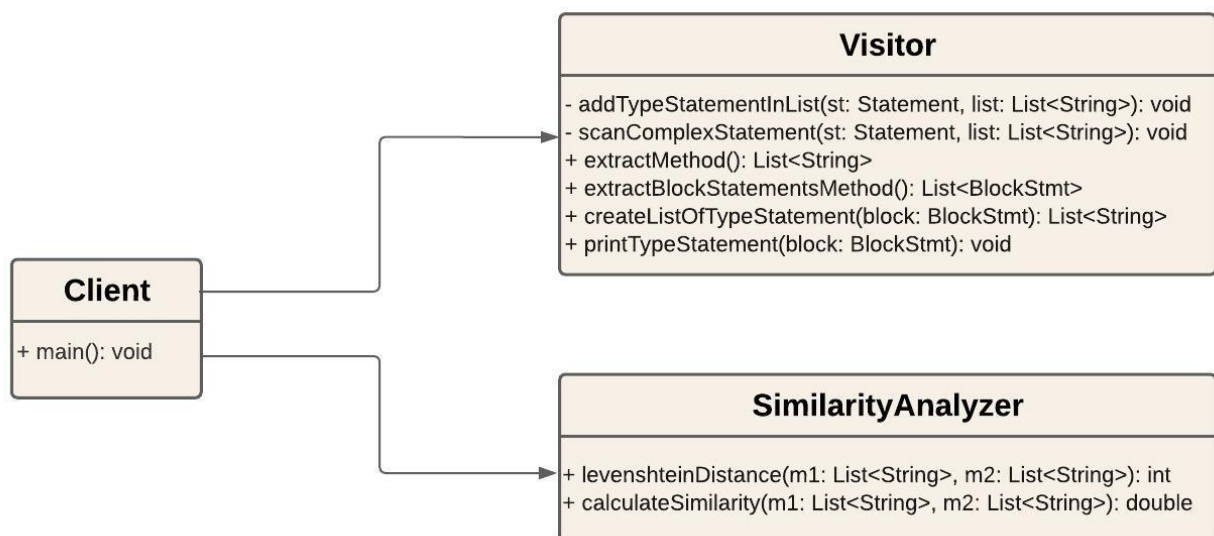
Premessa

Nel progetto è stato utilizzato Maven come strumento di gestione delle dipendenze e di compilazione del codice Java.

Una delle ragioni principali per aver scelto di utilizzare questo tool è stata la sua capacità di gestire le dipendenze in modo efficiente, grazie ad esso infatti è stato possibile specificare le dipendenze dalle varie librerie utilizzate nel file di configurazione `pom.xml` del progetto. Questo ha permesso a Maven di scaricare automaticamente le librerie e le dipendenze necessarie, garantendo che queste fossero presenti e aggiornate senza la necessità di gestirle manualmente. Ciò ha semplificato notevolmente il processo di setup del progetto e ha garantito che il codice avesse accesso alle risorse esterne necessarie.

Utilizzando i comandi di Maven è possibile compilare il codice sorgente Java in modo rapido e affidabile. Inoltre, esso fornisce strumenti per la gestione dei test, la generazione di report e la gestione del ciclo di vita del progetto.

Diagramma UML delle classi



Classi implementate

Visitor: è la classe in cui è stata sfruttata la libreria Javaparser, rappresenta infatti il visitatore principale che ispeziona uno script Java prendendo in input un oggetto di tipo CompilationUnit che conterrà il parse del codice sorgente da analizzare. Al suo interno vengono implementati i seguenti metodi e le seguenti classi private:

- Metodo *extractNameMethod()*: crea un oggetto di tipo *MethodName*, che estende la classe *VoidVisitorAdapter*, e visita il sorgente (*CompilationUnit*) per raccogliere i nomi dei suoi metodi all'interno di una lista.
- Metodo *extractBlockStatementsMethod()*: crea un oggetto di tipo *MethodBlockStmt* che estende *VoidVisitorAdapter* e visita il sorgente per raccogliere i blocchi di statement dei suoi metodi all'interno di una lista.
- Metodo *scanComplexStatement(Statement st, List<String> list)*: esamina in maniera ancora più capillare uno statement complesso (if, for, while, foreach, do, try) riempiendo la lista con tutte le tipologie di statement interni ad essi. Inoltre, vengono gestiti i casi di statement if con blocchi else e statement try con blocchi catch e finally.
- Metodo *addTypeStatementInList(List<String> list, Statement st)*: se lo statement passato come parametro è un'espressione semplice (es. *VariableDeclarationExpr*, *MethodExpr* ecc..) allora aggiunge la sua tipologia all'interno della lista, se è uno Switch invece farà la medesima cosa creando però un oggetto *SwitchEntryVisitor* che sa come ispezionare i vari case di uno Switch, altrimenti vuol dire che ci troviamo di fronte ad uno statement complesso quindi verrà richiamato il metodo *scanComplexStatement*.

- Metodo *createListOfTypeStatement(BlockStmt block)*: crea e ritorna una lista di stringhe, ciascuna di esse rappresenta i tipi degli statement presenti nel blocco di istruzioni fornito. Richiama il metodo *addTypeStatementInList* per ogni statement nel blocco.
- Metodo *printTypeStatement(BlockStmt block)*: stampa a video i tipi degli statement presenti nel blocco di istruzioni fornito.
- Classe privata *SwitchEntryVisitor*: estende *VoidVisitorAdapter* e gestisce la visita degli statement di uno Switch (case e default).
- Classe privata *MethodName*: estende *VoidVisitorAdapter* e raccoglie i nomi dei metodi in una lista fornita.
- Classe privata *MethodBlockStmt*: estende *VoidVisitorAdapter* e raccoglie i blocchi di statement dei metodi in una lista fornita.

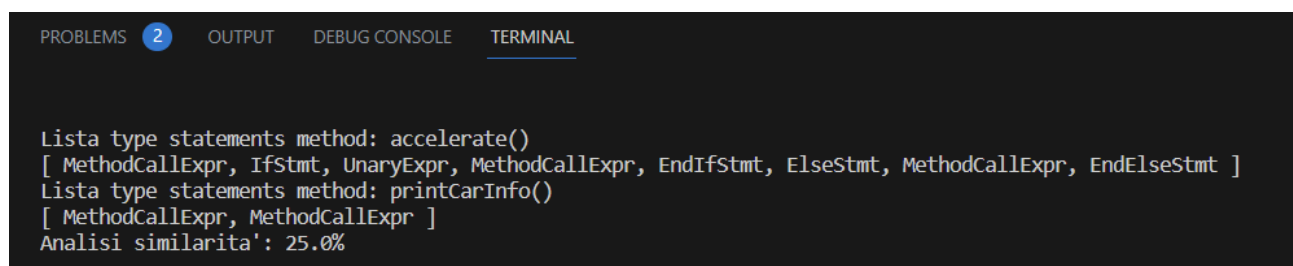
SimilarityAnalyzer: si occupa di effettuare un'analisi di similarità tra due liste di stringhe. Contiene i seguenti metodi:

- Metodo *levenshteinDistance(List<String> m1, List<String> m2)*: calcola la distanza di Levenshtein tra due liste di stringhe che rappresentano i metodi da confrontare. La distanza di Levenshtein misura il numero minimo di operazioni di modifica (inserimento, cancellazioni o sostituzione) necessarie per trasformare una stringa nell'altra (nel nostro caso per trasformare una lista nell'altra). Viene utilizzata una matrice per memorizzare i risultati intermedi dei calcoli.

- Metodo `calculateSimilarity(List<String> m1, List<String> m2)`: calcola la percentuale di similarità tra le due liste di stringhe. Utilizza il metodo `levenshteinDistance` per ottenere la distanza tra le liste. La percentuale di similarità viene poi calcolata come rapporto tra la lunghezza massima delle due liste e la distanza di Levenshtein.

Client: rappresenta il punto d'ingresso dell'applicazione. Utilizza un oggetto di tipo `CompilationUnit` che conterrà il parse del codice che si vuole analizzare (nel nostro caso abbiamo utilizzato una classe di prova **Car** per testare il funzionamento dell'applicazione).

Prende al suo interno un'istanza di tipo `Visitor` e una di tipo `SimilarityAnalyzer`, poi vengono sfruttati i metodi di questi due oggetti per prendere a coppie i metodi della classe di prova e fare un'analisi di similarità tra questi.



```
PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL
Lista type statements method: accelerate()
[ MethodCallExpr, IfStmt, UnaryExpr, MethodCallExpr, EndIfStmt, ElseStmt, MethodCallExpr, EndElseStmt ]
Lista type statements method: printCarInfo()
[ MethodCallExpr, MethodCallExpr ]
Analisi similarita': 25.0%
```

Frammento di esecuzione della classe `Client`, in questo caso vengono confrontati il metodo `accelerate()` e `printCarInfo()` della classe `Car.java`, per ciascun metodo viene stampata una lista dei suoi statement e viene infine fornita una percentuale di similarità tra i 2 .

Test e risultati

Sono state implementate delle classi di test **TestVisitor** e **TestSimilarityAnalyzer** per verificare il corretto funzionamento dei metodi presenti rispettivamente nelle due classi `Visitor` e `SimilarityAnalyzer`, ottenendo una percentuale di code coverage rispettivamente del 90% e del 100%.

JUnit

Per la realizzazione dei test si è fatto uso del framework JUnit che fornisce una serie di utilità per scrivere ed eseguire test automatici in modo efficiente, tra queste quelle utilizzate sono state:

- Annotazioni: per identificare i metodi di test (alcune annotazioni più comuni sono `@Test`, `@Before`, `@After` ...).
- Assert: che consentono di verificare le asserzioni all'interno dei test. Questi metodi permettono di confrontare i valori attesi con i risultati ottenuti durante l'esecuzione del test.

JUnit è integrato in molti ambienti di sviluppo e strumenti di build come Maven.

Jacoco

Oltre a JUnit, per verificare la quantità di codice coperto dai test è stato utilizzato il plugin Jacoco: uno strumento che consente di ottenere informazioni dettagliate sulla code coverage per quanto riguarda progetti Java.

Dopo l'esecuzione dei test, Jacoco genera dei report che mostrano quali parti del codice sono state coperte dai test, e indicano eventuali aree non testate o insufficientemente testate. I report possono essere generati in diversi formati come HTML o XML.

CodeCloneRefactoring > com.example

com.example

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed Cxty	Missed Lines	Missed Methods	Missed Classes
Car	<div><div></div></div>	0%	<div><div></div></div>	0%	24 24	73 73	10 10	1 1
Client	<div><div></div></div>	0%	<div><div></div></div>	0%	4 4	19 19	2 2	1 1
Visitor	<div><div></div></div>	90%	<div><div></div></div>	83%	7 31	7 71	1 7	0 1
SimilarityAnalyzer	<div><div></div></div>	100%	<div><div></div></div>	100%	0 8	0 14	0 3	0 1
Visitor.SwitchEntryVisitor	<div><div></div></div>	100%	<div><div></div></div>	87%	1 7	0 16	0 3	0 1
Visitor.MethodBlockStmt	<div><div></div></div>	100%		n/a	0 1	0 2	0 1	0 1
Visitor.MethodName	<div><div></div></div>	100%		n/a	0 1	0 2	0 1	0 1
Total	422 of 948	55%	39 of 96	59%	36 76	99 197	13 27	2 7

Report di Jacoco in formato HTML

Conclusioni

Il progetto è stato portato a termine con successo. Sono state implementate tutte le funzionalità necessarie per effettuare l'analisi di similarità tra i metodi di una classe utilizzando la libreria `JavaParser`.

La suddivisione del progetto in due parti, concentrandosi prima sullo studio della libreria e sull'implementazione dei metodi per la manipolazione dei costrutti di una classe, e successivamente sull'analisi di similarità, ha garantito una struttura organizzata e una progressione logica nello sviluppo.

L'utilizzo di `JavaParser` è stata un'ottima scelta per l'analisi del codice sorgente Java. La libreria ha fornito gli strumenti necessari per esaminare e manipolare i costrutti dei metodi, e ha facilitato l'estrazione delle informazioni di interesse come i nomi dei metodi e i blocchi di statement.

Il tipo di analisi di similarità effettuata può rivelarsi utile per identificare i metodi ridondanti o potenzialmente sostituibili all'interno di una classe, quindi il progetto ha raggiunto gli obiettivi prefissati; tuttavia, è importante sottolineare che tale analisi potrebbe essere ulteriormente approfondita in futuro. Ci sono molte altre metriche e algoritmi disponibili per valutare la similarità tra i metodi, e l'aggiunta di tali potrebbe portare a risultati più accurati e approfonditi.