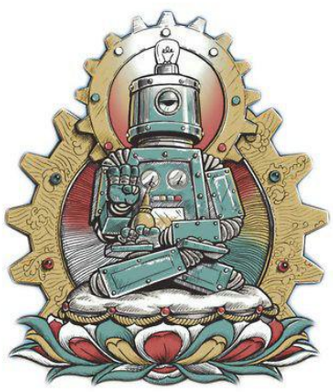
UNIVERSITÀ DEGLI STUDI DI SALERNO
CORSO DI INGEGNERIA DEL SOFTWARE

ASCETIC

Automated Code Smell Identification and Correction

DOCUMENTO DI MANUTENZIONE

VERSION 1.1



30 gennaio 2019

Coordinatore Progetto:

Nome	**Matricola**
Manuel De Stefano	0522500633

Partecipanti:

Nome	**Matricola**
Amoriello Nicola	0512104742
Di Dario Dario	0512104758
Gambardella Michele Simone	0512104502
Iovane Francesco	0512104550
Pascucci Domenico	0512102950
Patierno Sara	0512103460

Revision History:

Data	**Versione**	**Descrizione**	**Autore**
05/10/2018	1.0	Studio di Fattibilità, Reverse Engineering e Individuazione del CIS	Manuel De Stefano
29/01/2019	1.1	Report della modifica e calcolo delle metriche di precisione	Manuel De Stefano

Indice

1 Studi Preliminari

1.1 Panoramica del Sistema

Il sistema in esame si compone sostanzialmente di 6 sottosistemi: *Beans, Analisi,Metriche,Topic, Parser* e *Refactoring*. A questo si deve aggiungere un sottosistema aggiuntivo che contiene tutte le componenti di interfaccia grafica (sottosistema *GUI*). Di seguito verranno descritti nel dettaglio:

Sottosistema Beans Questo sottosistema contiene le classi che rappresentano le unità dei dati su cui lavora il plugin. Sono presenti 4 classi: *PackageBean, ClassBean,MethodBean* ed *InstanceVariableBean*. Ognuno di essi corrisponde ad un'astrazione, rispettivamente, per Package, Classe, Metodo e Variabile d'Istanza. Essi rappresentano i dati su cui lavorano tutti gli altri sottosistemi.

Sottosistema Analisi Questo sottosistema contiene le classi responsabili di effettuare l'analisi per l'individuazione dei code smell presenti nel codice. Il sistema contiene una classe astratta *CodeSmell* e 4 classi concrete corrispondenti ai code smell analizzati: *FeatureEnvy, MisplacedClass, Blob* e *PromiscuousPackage*. Inoltre, è presente una interfaccia *CodeSmellDetectionStrategy* dalla quale derivano le classi concrete per l'individuazione dei vari code smell. Attualmente sono presenti 4 strategy, i quali implementano l'algoritmo di analisi testuale per il riconoscimento del rispettivo smell.

Sottosistema Parser Sottosistema che contiene le classi responsabili di produrre i beans a partire da un'altra rappresentazione. Nell'attuale implementazione esiste una interfaccia, *Parser*, ed una sua realizzazione, *PsiParser*, che realizza il parsing in beans a partire dall'albero PSI di IntelliJ.

Sottosistema Refactoring Sottosistema che contiene le classi che implementano gli algoritmi di Refactoring per gli smell di Feature Envy e Misplaced Class, anch'essi realizzati tramite il design pattern *Strategy*.

Sottosistema Topic Sottosistema che contiene le classi che implementano gli algoritmi di estrazione di topic dal contenuto testuale dei beans.

Sottosistema Metriche Sottosistema che contiene le classi che implementano gli algoritmi per il calcolo delle metriche della suite CK.

Tali informazioni sono state sintetizzate a partire dalla documentazione prodotta a seguito della *Change Request 01*

1.2 Analisi della Modifica Richiesta

La change request 02 richiede una manutenzione evolutiva, atta ad migliorare le prestazioni del sistema, in particolare durante le operazioni più onerose, come l'analisi. Dunque i cambiamenti che verranno apportati non saranno visibili a livello di funzionalità, bensì in termini di requisiti non funzionali.

1.2.1 Individuazione del Problema

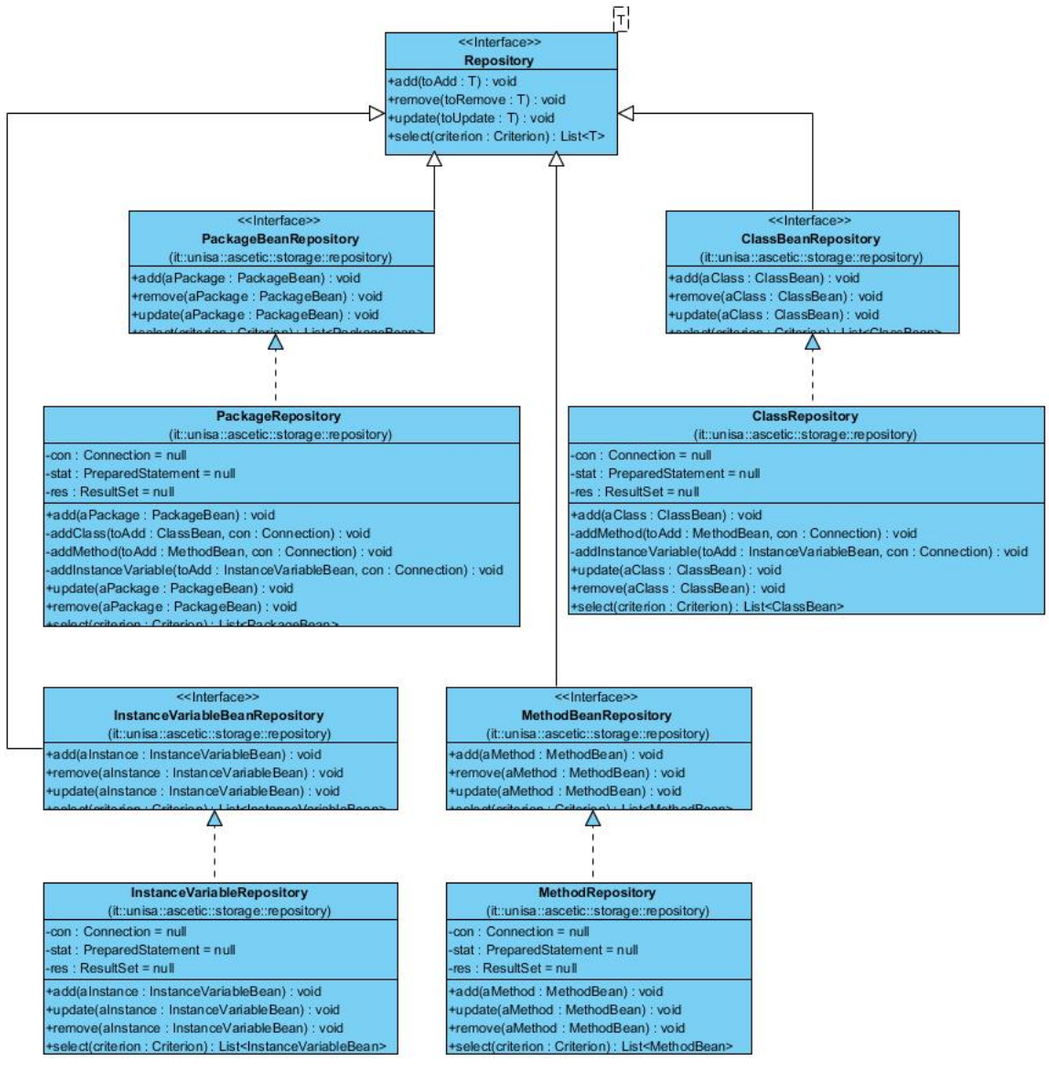
Attualmente, il processo di analisi del codice sorgente risulta essere molto oneroso in termini di tempo e di computazione. Questo è dovuto dal fatto che, essendo stati implementati unicamente gli algoritmi di analisi testuale, occorre effettuare il calcolo della probabilità di *smelliness* confrontando ad una ad una le componenti con tutte le componenti dell'intero progetto. Ad esempio, se volessimo calcolare tutti i metodi affetti da *Feature Envy*, occorrerebbe confrontare tutti i metodi, di tutte le classi, di tutti i package con tutti tutti i metodi, di tutte le classi, di tutti i package del progetto, con un costo computazionale che si aggira intorno a $O(n^2)$, dove n è il numero di metodi presente nell'intero progetto. Apparentemente può sembrare un costo acettabile, ma se si pensa che effettuare un calcolo di somiglianza testuale è già di per sè un'operazione molto onerosa, e che in un progetto di medie dimensioni il numero di metodi si aggira sull'ordine delle centinaia, si ottiene che effettuare questo calcolo possono occorrere diversi minuti.

1.2.2 Soluzione proposta

La soluzione proposta è quella di introdurre un meccanismo di chaching, con lo scopo di memorizzare i risultati delle varie analisi e di aggiornarli ogni qual volta è richiesta esplicitamente un'analisi. Nel dettaglio, vi sarà una prima analisi all'avvio del plugin, la quale, nel caso fosse la prima su un determinato progetto, provvederà anche alla creazione del file in cui verranno memorizzati i risultati, e successivamente, quando l'utente vorrà fare uso del plugin, verranno analizzate e aggiornate solo le componenti che, dalla prima analisi, sono state modificate. Il risultato è un miglioramento delle prestazioni nel tempo, seppur il costo computazionale al caso pessimo rimane invariato.

Tale meccanismo di chaching sarà implementato sfruttando la libreria *SQLite* che permette la realizzazione di un *EmbeddedDataSource* sfruttando le potenzialità di un database relazionale. Verranno aggiunte delle classi, una per bean più l'interfaccia, denominate *Repository* che forniranno l'accesso al *DataSource*. Al momento successivo all'analisi, i risultati verranno memorizzati sul *DataSource*, mentre quando l'utente richiederà di visionare le componenti affette da un determinato smell, verrà effettuata una vera e propria *query*, con cui queste componenti saranno recuperate. A tal proposito, per ottimizzare il caricamento delle lunghe liste di componenti (e.g. quando si carica un package è necessario caricare tutta la lista di classi, e per ognuna di esse la lista delle variabili d'istanza e dei metodi), verrà introdotto un meccanismo di *Lazy Loading*, il quale sarà realizzato tramite il design pattern *proxy*. Ciò implica una modifica sulle classi bean. Sarà introdotta, inoltre, una classe che, sfruttando il *VirtualFileSistem* di IntelliJ, annoterà su uno speciale file CSV quali file, e quindi quali classi, vengono modificate.

Il compito di effettuare l'analisi in background e scrivere i risultati sul *DataSource* sarà affidato al modulo *Parser*, in maniera tale da ottimizzare anche tale procedura e costruire il *DataSource* a mano a mano che viene fatto il parsing di una determinata componente. Di seguito è riportato il class diagram delle classi *Repository*.



1.3 Individuazione dell'Impact Set

Il passo immediatamente successivo alla valutazione di problemi e soluzioni, consiste nell'individuazione degli *impact set*, al fine di verificare la fattibilità della modifica.

1.3.1 Individuazione dello Starting Impact Set

Lo *Starting Impact Set* conterrà tutte quelle componenti che sono direttamente impattate dalla modifica. Lo *Starting Impact Set* per prevederà le seguenti classi:

- PackageBean;
- ClassBean;
- MethodBean;
- InstanceVariableBean.
- PsiParser
- BlobAction;
- FeatureEnvyAction;
- MisplacedClassAction;
- PromiscuousPackageAction;

Saranno, inoltre, create *ex novo* le seguenti classi:

- Repository;
- PackageBeanRepository;
- ClassBeanRepository;
- MethodBeanRepository;
- InstanceVariableBeanRepository;
- SqlitePackageBeanRepository;
- SqliteClassBeanRepository;
- SqliteMethodBeanRepository;
- SqliteInstanceVariableBeanRepository;
- Criterion;
- SqliteCriterion;
- ClassBeanList;
- ClassBeanListProxy;
- ConcreteClassBeanList;
- MethodBeanList;
- MethodBeanListProxy;
- ConcreteMethodBeanList;
- InstanceVariableBeanList;
- InstanceVariableBeanListProxy;
- ConcreteInstanceVariableBeanList;

1.3.2 Individuazione del Candidate Impact Set

Il *Candidate Impact Set* è stato l'uso della *Matrice di Dipendenza*, ovvero una matrice $n \times n$, dove n è il numero di classi, in cui ogni cella indica quante dipendenze ha una classe da un'altra (in termini di variabili di quel tipo usate). L'euristica usata prevede che, per ogni elemento nello *Starting Impact Set* venisse aggiunta nel *Candidate Impact Set*, oltre all'elemento in questione, anche una classe che avesse una dipendenza diretta verso tale elemento, e che quindi fosse riportato dalla matrice come dipendente dall'elemento in questione. Il *Candidate Impact Set* per la modifica prevederà le seguenti classi:

- BlobAction;
- FeatureEnvyAction;
- MisplacedClassAction;
- PromiscuousPackageAction;
- BlobFrame;
- MisplacedClassFrame;
- FeatureEnvyFrame;
- MisplacedClassWizard;
- FeatureEnvyWizard;
- PsiParser;
- TextualFeatureEnvyStartegy;
- TextualMisplacedClassStrategy;
- TextualBlobStrategy;
- TextualPromiscuousPackageStrategy;
- MoveMethodStrategy;
- MoveClassStrategy;
- TopicExtracter;
- RadarMapsUtil;
- MisplacedComponentsUtitlities;
- ComponentMutation;
- CKMetrics.

2 Report Post-Modifica

In questa sezione vedremo come la stima sull'entità fatta nella sezione precedente sia stata precisa. Andremo, innanzitutto, a confrontare *Actual Impact Set* con il *Candidate* e verificheremo se ci sono stati dei falsi positivi oppure delle componenti che non sono state considerate.

2.1 Analisi dell'Actual Impact Set

Dopo aver effettuato le modifiche, l'*Actual Impact Set* contiene le seguenti classi:

- PackageBean;
- ClassBean;
- MethodBean;
- InstanceVariableBean.
- PsiParser
- BlobAction;

- FeatureEnvyAction;
- MisplacedClassAction;
- PromiscuousPackageAction;

Come è evidente, mancano diverse classi. Tali classi sono state inserite in quanto dipendenti dalle classi bean. Tuttavia esse non hanno risentito delle modifiche, dato che il meccanismo di *lazy loading* è stato completamente mascherato e wrappato con metodi già esistenti nei beans stessi. Tali classi sono, dunque, da considerarsi falsi positivi. Non sono presenti, inoltre, classi che non erano state considerate nel *Candidate Set* e quindi il *Discovered Impact Set* è vuoto.

2.2 Calcolo delle Metriche

Di seguito, calcoleremo alcune metriche per verificare l'accuratezza dell'*ImpactAnalysis*.

$$Recall = \frac{|CIS \cap AIS|}{|AIS|} = \frac{9}{9} = 1$$

$$Precision = \frac{|CIS \cap AIS|}{|CIS|} = \frac{9}{21} = 0.42$$

$$Inclusiveness = \begin{cases} 1 & se\ AIS \subseteq CIS \\ 0 & altrimenti \end{cases} = 1$$

3 Conclusioni

In conclusione, possiamo dire che la modifica sia stata portata a termine con successo. Tuttavia, il processo di *ImpactAnalysis* ha leggermente sovrastimato l'entità della modifica, come dimostra il basso valore di *Precision*. Ciò nonostante, la *Recall* è pari ad 1, e quindi nessuna classe realmente modificata è stata tralasciata dalla stima.