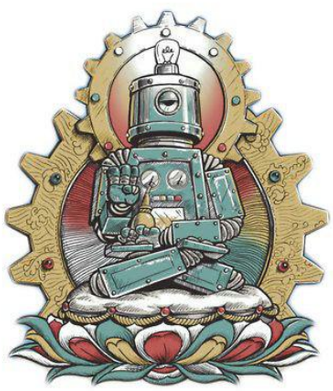
UNIVERSITÀ DEGLI STUDI DI SALERNO
CORSO DI INGEGNERIA DEL SOFTWARE

ASCETIC

Automated Code Smell Identification and Correction

DOCUMENTO DI MANUTENZIONE

VERSION 1.1



1 febbraio 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 03 richiede una manutenzione evolutiva, volta all'aggiunta di due nuove funzionalità, ovvero il refactoring automatico di *Blob* e *Promiscuous Package*.

1.2.1 Individuazione del Problema

Attualmente, il sistema è solamente in grado di individuare componenti affette da questi due smell, tuttavia esso non è in grado di trovarne una soluzione. Questo perché, mentre per *Feature Envy* e *Misplaced Class*, l'individuazione del problema consiste anche nell'individuarne la soluzione, ciò non è altrettanto vero per *Blob* e *Promiscuous Package*, dato che per poter essere corretti necessitano, rispettivamente, operazioni di *Extract Class Refactoring* ed *Extract Package Refactoring*. Occorre, dunque, una euristica per individuare le componenti da estrarre.

1.2.2 Soluzione proposta

La soluzione proposta è quella di implementare tali algoritmi,i quali sono stati teorizzati e pubblicati da Bavota *et al.*[1, 2], ai cui articoli si rimanda. Una volta ottenuti i risultati di tali algoritmi, verranno applicate le procedure di *Extract Class Refactoring* e *Extract Package Refactoring*. Vi è da precisare che, mentre per l' *Extract Package Refactoring* non vi è nessun tipo di problemi, se non quello di aggiornare la dichiarazione del package nella classe spostata e gli import nelle altre classi (cosa che viene eseguita in automatico da IntelliJ), per l'*Exctract Class Refactoring* è necessario l'uso della delegation, in quanto risulta molto difficoltosa la sostituzione dei riferimenti.

Al fine di applicare tali modifiche verranno introdotte 4 nuove classi nel modulo di analisi, e 2 nel modulo di refatoring. Nel primo caso avremo:

1. SplitClasses
2. SplitPackages
3. BuildMethodToMethodMatrix
4. BuildClassToClassMatrix

Esse saranno, appunto, responsabili di applicare gli algoritmi di *splitting* per *Blob* e *PromiscuousPackage*. Nel modulo di refactoring saranno invece introdotte le seguenti classi, che implementeranno l'interfaccia *RefactoringStrategy*:

1. ExtractClassRefactoringStrategy
2. ExtractPackageRefactoringStrategy

Le quali hanno responsabilità che si evincono dal nome. Infine, saranno aggiunte due nuove classi di interfaccia grafica, per permettere all'utente di visionare le soluzioni applicate e di applicare il refactoring:

1. BlobWizard
2. PromiscuousPackageWizard

Per permettere la visualizzazione delle suddette interfacce, tuttavia, sarà necessaria una modifica alle seguenti classi:

1. BlobPage
2. PromiscuousPackagePage

Tale modifica fa si che, una volta individuati i candidati smell, si possa visionare il frame per applicarne la soluzione. Nel pratico, la modifica sarà applicata su degli action listener, i quali avranno il compito di invocare e visualizzare le nuove interfacce.

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:

- BlobPage;
- PromiscuousPackagePage;

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. Poiché nessuna altra classe esistente mostra una dipendenza nei confronti delle classi nello *Starting Set*, il *Candidate Set* sarà composto unicamente da queste due classi.

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:

- BlobPage
- PromiscuousPackagePage

2.2 Calcolo delle Metriche

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

$$Recall = \frac{|CIS \cap AIS|}{|AIS|} = \frac{2}{2} = 1$$

$$Precision = \frac{|CIS \cap AIS|}{|CIS|} = \frac{2}{2} = 1$$

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

3 Conclusioni

Come si può notare, l'impatto della modifica è stato minimo, in quanto la maggior parte del lavoro è stata l'aggiunta delle nuove classi, mentre l'unica modifica necessaria è stata quella dell'integrazione di queste. Gli alti valori di *Precision* e *Recall* ne sono una dimostrazione.

Riferimenti bibliografici

[1] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, *Automating extract class refactoring: An improved method and its evaluation*, Empirical Softw. Engg., vol. 19, no. 6, pp. 1617–1664, Dec. 2014.

[2] G. Bavota, A. De Lucia, and R. Oliveto, *Identifying extract class refactoring opportunities using structural and semantic cohesion measures*, J. Syst. Softw., vol. 84, no. 3, pp. 397–414, Mar. 2011.