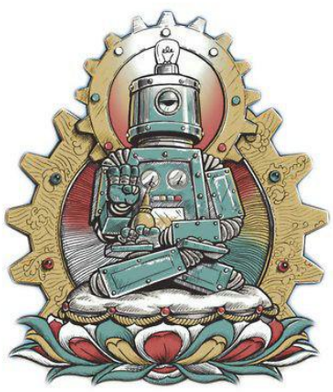
UNIVERSITÀ DEGLI STUDI DI SALERNO
CORSO DI INGEGNERIA DEL SOFTWARE

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, Code Smell Detection,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 Code Smell Detection Questo sottosistema contiene le classi responsabili di effettuare l'analisi per l'individuazione dei code smell presenti nel codice. Esistono anche qui quattro classi, ognuna responsabile di individuare un tipo di smell. Esse sono: *FeatureEnvyRule, BlobRule, MisplacedClassRule, PromiscuousPackageRule*, a cui pertiene di individuare, rispettivamente, i code smell *Feature Envy, Blob, Misplaced Class* e *Promiscuous Package*, attraverso l'algoritmo di analisi testuale TACO.

Sottosistema Parser Sottosistema che contiene le classi responsabili di produrre i beans a partire da un'altra rappresentazione. Nell'attuale implementazione esistono 4 classi, una per Bean, le quali generano il proprio bean a partire da una rappresentazione in PSI (l'AST di Intellij IDEA). Esiste, inoltre, una classe che fa da *Façade* al sottosistema. Questa implementa 4 metodi statici, uno per bean.

Sottosistema Refactoring Sottosistema che contiene le classi che implementano gli algoritmi di Refactoring per gli smell di Feature Envy e Misplaced Class.

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.

Nel sottosistema della GUI va segnalato un particolare package, denominato *Project Analysis*, in cui è contenuta un'unica classe, un singleton, che viene utilizzato come una sorta di sessione tra le varie schermate e a cui viene delegato il compito di invocare gli algoritmi di analisi e di mettere in cache i risultati.

Non essendo presente una documentazione reale del sistema, tali informazioni sono state estrapolate per reverse enineering direttamente dal codice sorgente. Tali informazioni sono state dettagliatamente riportate nelle sezioni "Sistema Corrente" dei documenti di Analisi dei Requisiti e di System Design, alle quali si rimanda.

1.2 Analisi della Modifica Richiesta

La change request 01 richiede una manutenzione adattiva, atta ad aumentare la manutenibilità del sistema in previsione di future modifiche. Dunque i cmabiamenti che verranno apportati non saranno visibili a livello di funzionalità, bensì in termini di requisiti non funzionali.

1.2.1 Individuazione del Problema

Allo stato attuale, il sistema presenta un forte accoppiamento tra i vari sottosistemi: si pensi, ad esempio, che il modulo parser presenta un *façade* i cui metodi sono statici, e dunque immutabili e globali. Sostanzialmente, ogni componente è legata direttamente ad una classe concreta presente in un'altra, ed il più delle volte non solo ad una. Spesso, come nel caso degli algoritmi di Code Smell Detection, esiste solamente una classe concreta che implementa un determinato algoritmo, quando essi potrebbero essere resi interscambiabili mediante l'uso di appositi design patterns (*Strategy*). Le classi bean, in più, presentano un gran numero di attributi, di cui alcuni sono opzionali. Tuttavia, esse implementano solo un costruttore vuoto e viene lasciato ai metodi *setter* il compito di inizializzare questi attributi. Infine, il singleton che viene usato come sessione tra le varie finestre

dell'interfaccia grafica (la classe *ProjectAnalyzer*), siccome viene utilizzato per invocare gli algoritmi di analisi e per mantenerne i risultati, rischia di diventare un Blob nel caso in cui si aggiungano altre procedure di identificazione di smell.

Dunque, nel dettaglio, le problematiche che saranno affrontate nel corso del processo di cambiamento sono le seguenti:

- Riprogettazione della logica di creazione dei Bean;
- Riprogettazione e refactoring del modulo *Parser*;
- Riprogettazione e refactoring del modulo *Code Smell Detection*;
- Riprogettazione e refactoring del modulo *Refactoring*;
- Eliminazione e sostituzione della classe *ProjectAnalyzer*.

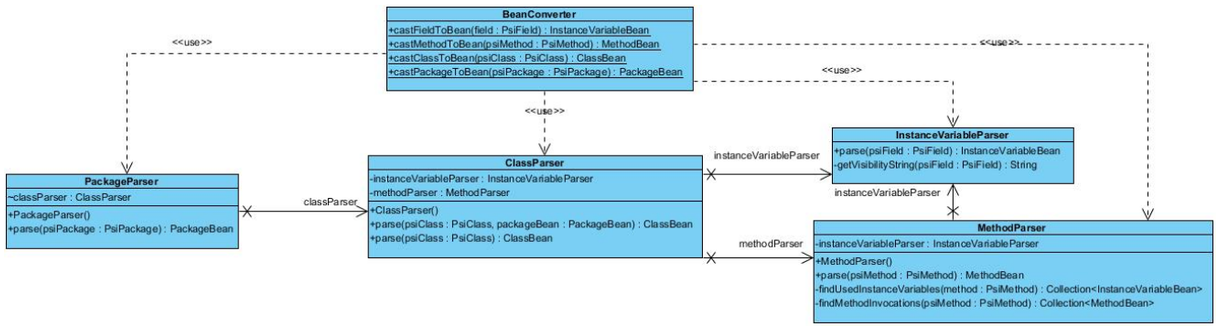
Nella sezione seguente saranno trattate nel dettaglio le problematiche e le soluzione legate ad ognuno dei seguenti punti.

1.2.2 Soluzione proposta

Riprogettazione della logica di creazione dei Bean I beans, come già detto in precedenza, sono le astrazioni fondamentali su cui si basa l'intero plugin. Tuttavia, essi presentano un gran numero di attributi, spesso opzionali, che non vengono inizializzati tramite costruttore, bensì solo attraverso metodi *setter*, mentre il costruttore è lasciato quello di default. Questo può portare alla creazione di un oggetto in maniera incoerente, perché alcuni attributi obbligatori potrebbero essere omessi, ed alcune combinazioni di valori potrebbero essere inconsistenti.

La soluzione applicata è l'introduzione del design pattern *Builder*. Ogni bean avrà un suo *Builder*, implementato come classe interna, che gestirà la logica di creazione, imponendo i parametri obbligatori tramite costruttore, e settando coerentemente gli attributi opzionali, tramite dei metodi semanticamente significativi.

Riprogettazione del modulo Parser Il modulo *Parser*, come già detto, presenta 4 classi contenenti un unico metodo, che converte i 4 nodi PSI (PsiPackage, PsiClass, PsiMethod, PsiField) nel loro corrsipettivo in Bean. Di seguito viene riportato il class diagram per il modulo allo stato attuale.

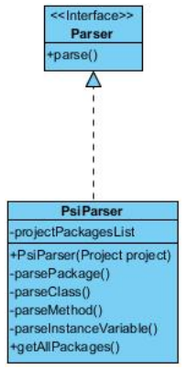


Tale struttura presenta evidenti difetti:

1. Le varie classi (ad eccezione della classe *BeanConverter*) presentano un unico metodo (gli altri sono overload con l'oggetto parent a cui legare l'oggetto su cui effettuare il parsing) che utilizza il metodo parse di un'altra classe per il parsing delle relazioni interne (e.g. Il package parser scorre la lista delle classi all'interno di un package ed invoca il metodo parse del *ClassParser* per ognuna di esse).
2. La classe *BeanConverter*, nei suoi 4 metodi, istanzia staticamente i 4 tipi di parser e li utilizza.

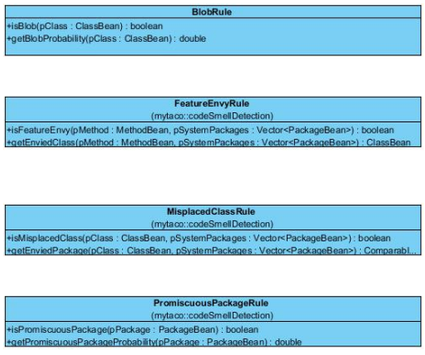
Conseguenza della prima situazione è un alto accoppiamento tra le varie classi. Conseguenza della seconda, invece, è l'impossibilità di poter utilizzare un'altra tecnica di parsing senza dover modificare la classe *façade*.

La soluzione a questa problematica prevede la realizzazione di una interfaccia *Parser* che offre un unico metodo, *parse*, il quale restitisce una lista di package beans. Tale interfaccia, allo stato attuale, viene realizzata da una classe, che prende nome *PsiParser*, che implementa, appunto, la responsabilità di effettuare il parsing in beans a partire dalla struttura PSI. Tale classe consiste nella fusione delle 4 classi precedenti, con l'eccezione che i 4 metodi siano privati e vengono invocati a cascata a partire dal metodo *parse*. Il *PsiParser*, poi, prevede un costruttore che prende in input un oggetto *Project* (appartenente alle API di Intellij) dal quale, tramite un apposito metodo privato, ottiene la lista di tutti i package su cui poi effettuare il parsing. Di seguito, il diagramma del nuovo modulo.



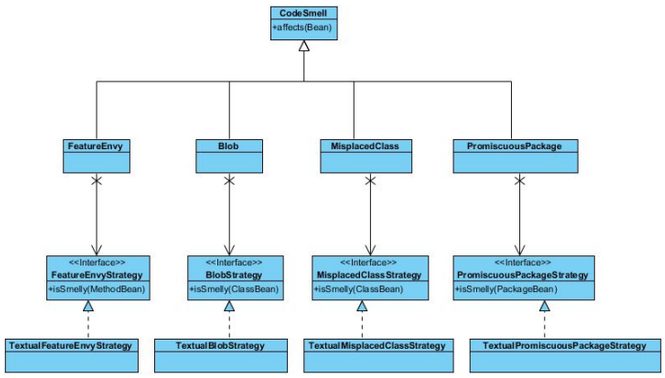
Tale soluzione, non solo aumenta la coesione del modulo, poiché i metodi strettamente collegati si trovano in un'unica classe, piuttosto che in 4 classi diverse, ma diminuisce anche la coesione di questo modulo con i moduli esterni, dato che essi si legano ad una interfaccia piuttosto che ad una classe concreta.

Riprogettazione e refactoring del modulo Code Smell Detection Il modulo di *Code Smell Detection*, oltre a delle clasi di util, presenta 4 classi fondamentali (FeatureEnvyRule, BlobRule, MisplacedClassRule, PromiscuousPackageRule) che individuano rispettivamente i code smell di Feaure Envy, Blob, Misplaced Class e Promiscuous Package, applicando un algoritmo di analisi testuale. Tuttavia, le classi client che le utilizzano sono legate alla specifica implementazione dell'algoritmo. Inoltre, per poter utilizzare tali algoritmi, deve essere istanziata la classe che lo implementa e gli deve essere passato un apposito bean. Di seguito, il diagramma delle classi che rappresenta la situazione attuale.



La soluzione proposta, prevede l'introduzione di un oggetto astratto di tipo *CodeSmell*, dal quale poi derivano i 4 tipi di smell che sono analizzati dal plugin, ovvero *FeaureEnvy, Blob, MisplacedClass e PromiscuousPackage*. Ognuno di essi, poi, applicando lo *strategy pattern*, presenta un metodo (*affects*) che prende in input un bean che può, appunto, presentare lo smell. Lo *strategy* che implementa l'algoritmo è, poi, passato attraverso il costruttore. Ogni *strategy*, dunque, implementa un metodo (*isSmelly*) che prende in input il corrispondente bean. Alle quattro classi precedenti, è stato applicato un refactoring in maniera tale da implementare gli strategy. Così si ottengono 4 nuove classi: *TextualFeatureEnvyStrategy, TextualBlobStrategy, TextualMisplacedClassStrategy* e *TextualPromiscuousPackageStartegy*. Infine, ai vari bean, è stato aggiunto un metodo *isAffected* che prende in input un oggetto di tipo *CodeSmell* e restituisce un valore booleano a seconda se il bean è afflitto o meno dallo smell.

Questa modifica, seppur incrementando la complessità del modulo, in termini di dimensioni, rende molto più versatile l'analisi. Ora, un qualunque client che necessiti di analizzare un bean, deve unicamente di costruire un oggetto smell, scegliere lo strategy che vuole utilizzare per effettuare l'analisi, ed invocare il metodo *isAffected* direttamente sul bean. Inoltre, se in futuro si volesse implementare un nuovo algoritmo di analisi per individuare un determinato *code smell*, occorre semplicemente implementare un nuovo strategy. Esso potrà poi essere utilizzato da un



client, semplicemente istanziandolo (o nel caso di client vecchi, modificando unicamente la linea in cui lo strategy viene dichiarato). Tutto questo con un minimo sforzo di manutenzione. Infine, il modulo verrà rinominato in *Analisi*.

Riprogettazione del modulo Refactoring La situazione del modulo di refactoring è del tutto analoga a quella del modulo di *Code Smell Detection*: sono presenti due classi concrete che implementano le responsabilità di eseguire il *Move Method Refactoring* e il *Move Class Refactoring*. Dunque, sostanzialmente, presenta le stesse limitazioni che presentava il modulo di *Code Smell Detection*. Le modifiche proposte, quindi, sono analoghe: verrà creata una classe *RefactoringManager*, la quale esporrà un metodo *doRefactor*. Le singole classi, poi, dopo la creazione di una interfaccia, *RefactoringStrategy*, verranno trasformate in realizzazioni della suddetta interfaccia. Dunque, ogni qual volta che si vorrà introdurre un nuovo metodo di refactoring, occorrerà semplicemente creare un nuovo strategy.

Eliminazione e sostituzione della classe ProjectAnalyzer La classe *ProjectAnalyzer* è una classe che implementa il pattern singleton, e che viene utilizzata come "sessione" tra una schermata ed un'altra: in essa, infatti, vengono invocate le operazioni di parsing e analisi in base ai parametri passati dal'utente. I risultati di tali operazioni vengono poi conservati e scambiati tra le varie schermate. I problemi di questa classe sono sostanzialmente 2: il primo è che vi è una dipendenza del flusso di controllo dai valori contenuti in questa classe che, sostanzialmente, può essere paragonata ad una variabile globale; il secondo è che tale classe presenta un metodo per invocare le analisi di ogni tipo di smell e, dunque, all'aumentare del numero di smell trattati, con l'evolversi del software, aumenterebbero i numeri di metodi (e il numero di modifiche da effettuare sulla classe).

La soluzione proposta, complementare alla modifica del modulo di analisi, prevede una sua rimozione. I valori che prima venivano mantenuti tra un frame ed un altro verranno, adesso, scambiati dai vari frame. Le invocazioni al parser e ai metodi di analisi, invece, verranno eseguite direttamente nei metodi che fungono da controllers (e.g. gli *ActionListeners*).

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. Poichè, nel nostro caso, la modifica corrisponde all'unione di 5 sotto-modifiche, lo *Starting Impact Set* della nostra modifica, corrisponderà all'unione dei singoli *Impact Set*. Dunque, lo *Starting Impact Set* per la prima modifica prevederà le seguenti componenti (classi):

- PackageBean;
- ClassBean;
- MethodBean;
- InstanceVariableBean.

Il secondo *Impact Set* sarà composto da:

- BeanConverter;

- PackageParser;
- ClassParser;
- MethodParser;
- InstanceVariableParser.

Il terzo *Impact Set* sarà composto da:

- FeatureEnvyRule;
- BlobRule;
- MisplacedClassRule;
- PromiscuousPackageRule.

Il quarto *Impact Set* sarà composto da:

- RefactoringManager;
- MoveMethodRefactoring;
- MoveClassRefactoring.

Il quinto *Impact Set* sarà composto da:

- ProjectAnalyzer.

Poiché non ci sono elementi in comune tra i sopracitati insiemi, l'elenco delle classi appena enunciate corrisponde allo *Starting Impact Set*.

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.

Come per lo *Starting Set*, anche in questo caso il *Candidate Impact Set* è dato dall'unione dei sottoinsiemi delle 5 sotto-modifiche.Dunque, lo *Candidate Impact Set* per la prima modifica prevederà le seguenti classi:

- BlobAction;
- FeatureEnvyAction;
- MisplacedClassAction;
- PromiscuousPackageAction;
- BlobFrame;
- MisplacedClassFrame;
- FeatureEnvyFrame;
- PromiscuousPackageFrame;
- MisplacedClassWizard;
- FeatureEnvyWizard;
- PromiscuousPackageWizard;
- ProjectAnalyzer;
- BeanConverter;
- PackageParser;

- ClassParser;
- MethodParser;
- InstanceVariableParser;
- FeatureEnvyRule;
- BlobRule;
- MisplacedClassRule;
- PromiscuousPackageRule;
- TopicExtracter;
- RadarMapsUtil;
- MisplacedComponentsUtitlities;
- ComponentMutation;
- CKMetrics.

Il secondo *Impact Set* sarà composto da:

- BlobAction;
- FeatureEnvyAction;
- MisplacedClassAction;
- ProjectAnalyzer.

Il terzo *Impact Set* sarà composto da:

- ProjectAnalyzer.

Il quarto *Impact Set* sarà composto da:

- MisplacedClassWizard;
- FeatureEnvyWizard.

Il quinto *Impact Set* sarà composto da:

- BlobAction;
- FeatureEnvyAction;
- MisplacedClassAction;
- BlobFrame;
- MisplacedClassFrame;
- FeatureEnvyFrame;
- MisplacedClassWizard;
- FeatureEnvyWizard;

Essendo tutti questi sottoinsiemi, contenuti nel primo sottoinsieme, il *Candidate Impact Set* corrisponderà proprio al primo sottoinsieme enunciato, unito dello *StartingSet*, e dunque avremo:

- BlobAction;
- FeatureEnvyAction;
- MisplacedClassAction;
- PromiscuousPackageAction;
- BlobFrame;
- MisplacedClassFrame;

- FeatureEnvyFrame;
- PromiscuousPackageFrame;
- MisplacedClassWizard;
- FeatureEnvyWizard;
- PromiscuousPackageWizard;
- ProjectAnalyzer;
- BeanConverter;
- PackageParser;
- ClassParser;
- MethodParser;
- InstanceVariableParser;
- FeatureEnvyRule;
- BlobRule;
- MisplacedClassRule;
- PromiscuousPackageRule;
- TopicExtracter;
- RadarMapsUtil;
- MisplacedComponentsUtitlities;
- ComponentMutation;
- CKMetrics
- PackageBean;
- ClassBean;
- MethodBean;
- InstanceVariableBean.

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:

- BlobAction;
- FeatureEnvyAction;
- MisplacedClassAction;
- PromiscuousPackageAction;
- BlobFrame;
- MisplacedClassFrame;
- FeatureEnvyFrame;
- PromiscuousPackageFrame;
- MisplacedClassWizard;

- FeatureEnvyWizard;
- PromiscuousPackageWizard;
- ProjectAnalyzer;
- BeanConverter;
- PackageParser;
- ClassParser;
- MethodParser;
- InstanceVariableParser;
- FeatureEnvyRule;
- BlobRule;
- MisplacedClassRule;
- PromiscuousPackageRule;
- PackageBean;
- ClassBean;
- MethodBean;
- InstanceVariableBean.

Come è evidente, mancano alcune classi che sono: *TopicExtracter*, *RadarMapUtil*, *MisplacedComponentsUtilities*, *ComponentMutation* e *CKMetrics*. Tali classi sono state inserite in quanto dipendenti dalle classi bean. Tuttavia esse non hanno risentito delle modifiche, dato che in esse non era contenuta logica di creazione, ma solo utilizzi. 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*.

$$Recall = \frac{|CIS \cap AIS|}{|AIS|} = \frac{25}{25} = 1$$

$$Precision = \frac{|CIS \cap AIS|}{|CIS|} = \frac{25}{30} = 0.83$$

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

3 Conclusioni

In conclusione, possiamo dire che la modifica sia stata portata a termine con successo e il processo di *ImpactAnalysis* sia stato abbastanza accurato, come mostrano gli alti valori di *Precision* e *Recall*.