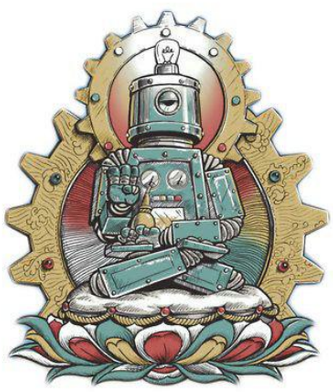
UNIVERSITÀ DEGLI STUDI DI SALERNO
CORSO DI INGEGNERIA DEL SOFTWARE

ASCETIC

Automated Code Smell Identification and Correction

STATEMENT OF WORK

VERSION 1.0



22 luglio 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**
12/10/2018	1.0	Prima stesura	Tutto il Team

Indice

1 Introduzione

1.1 Descrizione Breve

ASCETIC(Acronimo per Automated Code Smell Identification and Correction) è un plug-in, sviluppato per l'IDE IntelliJ IDEA, creato per individuare e correggere "code smell" presenti nel codice mediante l'uso di varie euristiche.

Il sistema è in grado di analizzare il codice, suggerire le possibili soluzioni per la correzione del problema e, per alcuni tipi di code smell, è possibile effettuare un refactoring automatico, sfruttando le librerie per la manipolazione del codice sorgente presenti nell'IDE.

1.2 Obiettivi

Il plug-in ASCETIC vuole essere una versione stabile ed evoluta del suo predecessore TACOR.
Attraverso un reengineering del sistema saranno introdotte nuove features che renderanno l'esperienza dell'utente maggiormente fruibile.

Il prodotto finale implementerà:

- Un nuovo algoritmo per l'analisi del code smell 'BLOB'.
- Un nuovo algoritmo per l'analisi del code smell 'Promiscuous Package'.
- Un'interfaccia grafica che possa rendere più facile ed intuitivo l'approccio con le varie operazioni.
- Maggiore robustezza e controlli sul linguaggio del codice analizzato.
- Sistema predisposto a futuri tipi d'analisi.

2 Background Information

TACOR(Acronimo per Textual Analysis for Code smell detectiOn and Refactoring) è un plug-in, sviluppato per l'IDE IntelliJ IDEA, creato per risolvere il problema dei "code smell" mediante l'analisi testuale. Esso è il progetto su cui è basato ASCETIC.
Presenta più o meno le caratteristiche del suo successore ma alcune funzionalità non sono state sviluppate seguendo un paradigma ingegneristico adeguato.

Il termine *code smell* che viene utilizzato per indicare porzioni di codice sorgente che non genera errori di alcun tipo, ma che impoveriscono la qualità e il design del software.
Le tipologie di *cose smell* trattate sono:

- Blob: Classe che implementa responsabilità molto diverse tra di loro.
- Promiscuous Package: Package contenente classi che hanno responsabilità diverse.
- Feature Envy: Metodo che è maggiormente interessato a variabili e metodi di una classe differente dalla sua.
- Missplaced Class: Classe che non ha alcuna attinenza con le classi dello stesso package.

3 Current Envirorment

3.1 Scenari

Nome scenario	**SC 1: Ricerca Blob**
Istanze di attori partecipanti	**Manuel: Sviluppatore**
Flusso di eventi	1. Manuel, sviluppatore del codice vuole ricercare all'interno del codice Java potenziali code smells di tipo "Blob". 2. All'interno di intelliJ, nella sezione dedicata ai packages, con il tasto destro Manuel seleziona tra i plug-ings "Ascetic", e nella sottovoce sceglie "Blob". 3. Dopo la selezione della voce e un elaborazione, viene visualizzata a schermo una finestra contenente una radar map con i 5 termini maggiormente presenti e a più alta probabilità di smell, oltre ad una tabella riepilogativa. 4. Siccome per questa tipologia di smell non c'è meccanismo di auto-fix, Manuel inizia ad intervenire sul codice in maniera manuale.

Nome scenario	**SC 2: Ricerca Promiscuous Package**
Istanze di attori partecipanti	**Manuel: Sviluppatore**
Flusso di eventi	1. Manuel, sviluppatore del codice vuole ricercare all'interno del codice Java potenziali code smells di tipo "Promiscuous Package". 2. All'interno di intelliJ, nella sezione dedicata ai packages, con il tasto destro Manuel seleziona tra i plug-ings "Ascetic", e nella sottovoce sceglie "Promiscuous Package". 3. Dopo la selezione della voce e un elaborazione, viene visualizzata a schermo una finestra contenente una radar map con i 5 termini maggiormente presenti e a più alta probabilità di smell, oltre ad una tabella riepilogativa. 4. Fatto tesoro del risultato mostrato, Manuel inizia il fixing manuale del problema, data l'assenza di automatismi di fixing.

Nome scenario	**SC 3: Ricerca Misplaced Class**
Istanze di attori partecipanti	**Manuel: Sviluppatore**
Flusso di eventi	1. Manuel, sviluppatore del codice vuole ricercare all'interno del codice Java potenziali code smells di tipo Misplaced Class. 2. All'interno di intelliJ, nella sezione dedicata ai packages, con il tasto destro Manuel seleziona tra i plug-ings "Ascetic", e nella sottovoce sceglie "Misplaced Class". 3. Dopo la selezione della voce e un elaborazione, viene visualizzata a schermo una finestra,in cui si possono notare tre radar map che mostrano i topic del package attuale (in alto a sinistra), del package invidiato (in alto a destra) e del potenziale candidato a Misplaced Class (in basso a destra); nella parte bassa della schermata è presente anche una tabella contenente tutti i potenziali candidati individuati. Manuel seleziona la voce "Perform Refactoring" e una nuova finestra viene mostrata all'utente. 4. La nuova finestra avrà e due nuove radar map (per un totale di 5),che rappresentano una previsione rispettivamente dei topic implementati dal package attuale post-refactoring e i topic del package invidiato dopo il refactoring. Selezionando il bottone "Refactor Code" verrà applicata l'operazione suggerita alla classe affetta dallo smell che viene spostata nel package invidiato. 5. La schermata si chiude con un avviso che il fix è stato applicato.

Nome scenario	**SC 4: Ricerca FeaturesEnvy**
Istanze di attori partecipanti	**Manuel: Sviluppatore**
Flusso di eventi	1. Manuel, sviluppatore del codice vuole ricercare all'interno del codice Java potenziali code smells di tipo Feature Envy. 2. All'interno di intelliJ, nella sezione dedicata ai packages, con il tasto destro Manuel seleziona tra i plug-ings "Ascetic", e nella sottovoce sceglie "Feature Envy". 3. Dopo la selezione della voce e un elaborazione, viene visualizzata a schermo una finestra, si possono notare le tre radar map che mostrano i topic della classe attuale, della classe invidiata, e del potenziale candidato a Feature Envy. In più sarà presente anche una tabella contenente tutti i potenziali candidati individuati, selezionando la voce Perform Refactoring, una nuova finestra viene mostrata all'utente. 4. La nuova finestra avrà e due nuove radar map,che rappresentano rispettivamente i topic implementati dalla classe attuale dopo il refactoring e i topic della classe invidiata dopo il refactoring, selezionando il bottone Refactor Code verrà, quindi, applicata l'operazione suggerita, e quindi, il metodo affetto dallo smell sarà spostato nella classe invidiata. 5. La schermata si chiude con un avviso che il fix è stato applicato. 6. Manuel clicca su Refactor e dopo una breve elaborazione la finestra del plug-in si chiude e lo smell è stato fixato.

3.2 Requisiti Funzionali

RF 1 - Ricerca code smell

Il sistema ricerca i code smell all'interno del codice java.

RF 1.1: Il sistema permette l'analisi di code smell di tipo Blob,

RF 1.2: Il sistema permette l'analisi di code smell di tipo Misplaced Class

RF 1.3: Il sistema permette l'analisi di code smell di tipo Feature Envy

RF 1.4: Il sistema permette l'analisi di code smell di tipo Promiscuos Package

RF 2 - Correzione

Il sistema mostra una possibile soluzione al code smell.

RF 2.1: Il sistema permette di corregere code smell di tipo Feature Envy.

RF 2.2: Il sistema permette di corregere code smell di tipo Misplaced Class.

RF 3 - Refactoring

Il sistema permette allo sviluppatore di risolvere il code smell con degli automatismi del software.

RF 3.1: Il sistema permette di risolvere code smell di tipo Feature Envy.

RF 3.2: Il sistema permette di risolvere code smell di tipo Misplaced Class.

RF 4 - Metriche di Qualità

Il sistema permette di calcolare metriche di qualità.

RF 4.1: Il sistema permette di calcolare metriche di qualità per i metodi.

RF 4.2: Il sistema permette di calcolare metriche di qualità per le classi.

RF 4.3: Il sistema permette di calcolare metriche di qualità per i package.

RF 5 - Estrazione dei Topic

Il sistema consente di estrarre i topic implementati.

3.3 Requisiti Non Funzionali

- **RNF 1-Performance**
  Il sistema è concepito come utilizzabile da singolo utente, con un'interazione diretta, il tempo di risposta varia in base all'analisi proposta.

- **RNF 2-Robustezza**
  Il sistema allo stato attuale non soddisfa il requisito di robustezza.

- **RNF 3-Sicurezza**
  Il sistema si presenta sufficientemente sicuro in quanto il suo ambiente d'azione è locale, quindi non vi è la necessità di protezione da minacce esterne.

- **RNF 4-Usabilità**
  Il sistema presenta bassi criteri di usabilità, data una scarna e poco intuitiva GUI.

- **RNF 5-Manutenibilità**
  Il sistema non presenta alcun tipo di documentazione pregressa, oltre ad adottare scelte implementative poco adatte a tale scopo.

- **RNF 6-Implementazione**
  Il sistema è realizzato interamente in linguaggio Java, sia parte back-end che front-end.

4 Proposed Envirorment

4.1 Scenari

I seguenti scenari, appartenenti al Current Envirorment, non sono stati modificati: SC3 e SC4.

Gli scenari SC1 e SC2 sono modificati in seguito alle nuove funzionalità apportate al sistema:

Nome scenario	**SC1 - Ricerca BLOB**
Istanze di attori partecipanti	**Manuel: Sviluppatore**
Flusso di eventi	1. Manuel, sviluppatore del codice, vuole ricercare all'interno del codice Java potenziali code smells di tipo BLOB. 2. All'interno di IntelliJ, nella sezione dedicata ai packages, con il tasto destro Manuel seleziona tra i plug-ings "Ascetic", e nella sottovoce sceglie "Blob". 3. Dopo la selezione della voce e un'elaborazione, viene visualizzata a schermo una finestra con una Radar-Map grafica che mostra i 5 termini più frequenti della classe attuale e una tabella che elenca i potenziali BLOB individuati all'interno del package. 4. Manuel clicca su "Check Solution" e viene aperta una nuova schermata con 3 Radar-Maps, una identica a quella della precedente schermata, e due che mostrano i 5 termini più frequenti delle due classi ottenute dalla possibile soluzione applicata. 5. Manuel clicca su "Fix Code" per applicare la soluzione che il sistema ha elaborato ed eseguire un refactoring sul codice. 6. La schermata precedente è chiusa e il codice viene corretto. Una finestra di notifica avvisa dell'avvenuta correzione.

Nome scenario	**SC2 - Ricerca Promiscuous Package**
Istanze di attori partecipanti	**Manuel: Sviluppatore**
Flusso di eventi	1. Manuel, sviluppatore del codice, vuole ricercare all'interno del codice Java potenziali code smells di tipo Promiscuous Package. 2. All'interno di IntelliJ, nella sezione dedicata ai packages, con il tasto destro Manuel seleziona tra i plug-ings "Ascetic", e nella sottovoce sceglie "Promiscuous Package". 3. Dopo la selezione della voce e un'elaborazione, viene visualizzata a schermo una finestra con una Radar-Map grafica che mostra i 5 termini più presenti del package attuale e una tabella che elenca i potenziali Promiscuous Package individuati. 4. Manuel clicca su "Check Solution" e viene aperta una nuova schermata con 3 RadarMaps dove una mostra sempre i 5 termini più presenti del package attuale, in più vengono mostrate altre 2 Radar-Maps che mostrano i 5 termini più frequenti dei due nuovi package ottenuti dalla possibile soluzione applicata. 5. Manuel clicca su "Fix Code" per applicare la soluzione che il sistema ha elaborato ed eseguire un refactoring sul codice. 6. La schermata precedente è chiusa e il codice viene corretto. Una finestra di notifica avvisa dell'avvenuta correzione.

4.2 Requisiti Funzionali

- I seguenti requisiti funzionali, appartenenti al Current Envirorment, non sono stati modificati:

  RF 1:
  RF 1.1
  RF 2:
  RF 2.1
  RF 2.2
  RF 3:
  RF 3.1
  RF 3.2
  RF 4:
  RF 4.1
  RF 4.2
  RF 5

- **RF 2 - Correzione**

  Il sistema consente la creazione di una possibile soluzione ai quattro principalei tipi di code smell.

  RF 2.3: Il sistema fornisce una possibile soluzione al code smell di tipo Blob.
  RF 2.4: Il sistema fornisce una possibile soluzione al code smell di tipo Promiscuous Package.

- **RF 3 - Refactoring:**

  Il sistema consente la creazione di una possibile soluzione ai quattro principali tipi di code smell.

  RF 3.3: Il sistema permette allo sviluppatore di risolvere il code smell di tipo Blob.
  RF 3.4: Il sistema permette allo sviluppatore di risolvere il code smell di tipo Promiscuous Package.

4.3 Requisiti Non Funzionali

- **RNF 1 - Performance**
  Il sistema deve garantire brevi tempi di risposta, in particolare nelle operazioni di analisi del codice.

- **RNF 2 - Robustezza**
  Il sistema è in grado di funzionare correttamente anche in situazioni anomale o in caso di uso scorretto, notificando l'utente della situazione erronea rilevata, ma senza terminare la propria esecuzione.

- **RNF 3 - Sicurezza**
  Il sistema si presenta sufficientemente sicuro in quanto il suo ambiente d'azione è locale, quindi non vi è la necessità di protezione da minacce esterne.

- **RNF 4 - Usabilità**
  Il sistema risulta essere di facile utilizzo. L'interfaccia grafica risulta essere intuitiva, agevolando il lavoro dello sviluppatore.

- **RNF 5 - Manutenibilità**
  Il sistema presenta un elevato grado di manutenibilità, in particolare, favorisce l'aggiunta di nuove funzionalità.

- **RNF 6 - Implementazione**
  Il sistema è realizzato interamente in linguaggio Java, sia parte back-end che front-end.

5 Deliverables

- State of work.
- Requirements Analysis Document (RAD).
- System Design Document.
- Object Design Document.
- Test Plan.
- Implementation.

6 Milestones

DOCUMENTI	SCADENZE
State of work	15 Ottobre 2018
Requirements and Use-Case Model	26 Ottobre 2018
Requirements Analysis Document	9 Novembre 2018
System Design Document	30 Novembre 2018
Object Design Document	14 Dicembre 2018
Test Plan	14 Dicembre 2018
Implementation	15 Gennaio 2019