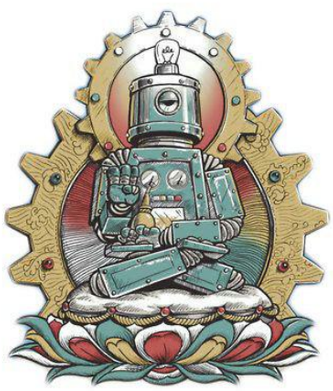
UNIVERSITÀ DEGLI STUDI DI SALERNO
CORSO DI INGEGNERIA DEL SOFTWARE

ASCETIC

Automated Code Smell Identification and Correction

SYSTEM DESIGN DOCUMENT

VERSION 1.1



29 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**
11/12/2018	1.0	Prima stesura	Tutto il Team
27/12/2018	1.1	Ridefinite le interfacce dei sottosistemi	Tutto il Team
09/01/2019	1.2	Ridefinita la struttura del Persistent Data Management.	Tutto il team

Indice

1 Introduction

1.1 Purpose of the system

Durante il ciclo di vita di un software, le modifiche costituiscono una parte essenziale ed inevitabile. La manutenzione, volta alla correzione di bug o all'integrazione di nuove funzionalità, porta ad un graduale deperimento del codice, il quale non inficia il corretto funzionamento del software bensì introduce delle debolezze di progettazione: i cosiddetti Code Smell. Per questo motivo nasce ASCETIC, un plug-in sviluppato per l'IDE intelliJ IDEA che consente di analizzare il progetto, rilevando 4 possibili tipologie di smell(Blob, Promiscuous Package, Feature Envy, Misplaced Class), ed effettuare un'eventuale correzione automatica.
ASCETIC (inizialmente TACOR) è un'opera di reingegnerizzazione, ed offre allo sviluppatore la possibilità di:

- Identificare le quattro sopracitate tipologie di Code Smell.
- Correggere automaticamente i Code Smell trovati dal sistema.
- Far ignorare al sistema determinati Code Smell, selezionati ad hoc dallo sviluppatore, trovati in fase d'analisi.
- Porre nella sezione "To do" di IntelliJ, tramite la funzione Reminder, gli Smell che lo sviluppatore decide di correggere manualmente in un secondo momento.

Allo stato attuale ASCETIC usufruisce di una tecnica di analisi testuale per il rilevamento dei vari Code Smell: il primo step consiste nell'estrazione del contenuto testuale che caratterizza le componenti del codice, facendo una cernita degli elementi necessari per l'analisi, ossia i commenti e gli identificatori. Questi ultimi vengono normalizzati, ed infine le parole normalizzate vengono pesate in base allo schema tf-idf. Le componenti normalizzate vengono analizzate dallo Smell Detector, basato su LSI, che trasforma le componenti del codice in vettori di termini presenti in un dato software. Tali vettori vengono proiettati in un K-spazio , ridotto appositamente per limitare l'effetto del rumore testuale. La dimensione di tale spazio viene determinata usando l'euristica di Kuhn. Infine la somiglianza fra i documenti viene calcolata come il coseno dell'angolo tra i due vettori. I valori vengono calcolati in maniera differente a seconda dello Smell da individuare per ottenere le probabilità che la porzione di codice sia affetta da tale Smell. Tali probabilità vengono convertite in valori booleani per indicare se un componente sia affetto o meno da Code Smell e richieda quindi una correzione. La scelta di questo metodo è dettata sia dalla validità comprovata di tale tecnica, sia dalla mancanza momentanea di alternative valide per effettuare il procedimento svolto dal plug-in in fase d'analisi. Uno degli obiettivi di sviluppo però è rendere ASCETIC flessibile alle modifiche ed all'introduzione di eventuali nuove tecniche per la gestione dell'analisi del codice. Per quanto riguarda il procedimento di refactoring, ASCETIC utilizza le API di IntelliJ per la manipolazione del codice sorgente.

1.2 Design goals

L'operazione di reengineering del plug-in ASCETIC dovrà tenere conto di un'alta manutenibilità, in modo che l'utente possa inserire con facilità nuove funzioni di analisi e correzione.
Il reengineering apporterà anche modifiche sul piano dell'usabilità, con un'interfaccia gradevole e intuitiva.
Il plug-in dovrà effettuare le operazioni richieste in tempi accettabili, con bassi consumi di memoria e in assenza di output errati.
Inoltre i casi di crash o blocchi dovrà essere ridotta al minimo e in caso di errori dovrà essere in grado di segnalare il problema e mostrare all'utente istruzioni per un'eventuale risoluzione.
Il sistema continuerà ad essere implementato completamente in Java e continuerà ad avere un modesto livello di sicurezza, poiché l'esecuzione viene svolta completamente in locale e non memorizza dati sensibili degli utenti.
Di seguito verranno elencati i criteri di qualità ricavati dai requisiti non funzionali.

1.2.1 Performance Criteria

Tempi di risposta	Il sistema deve essere reattivo, ma gran parte di questo dipende dalla mole di dati da elaborare e dalle prestazioni del dispositivo usato.
Throughput	Il sistema tiene conto del tempo impiegato per una singola istruzione piuttosto che del numero di correzioni effettuabili in parallelo
Memoria	Il sistema memorizza i dati persistenti in file distinti su memoria locale. E' implementata una meccanica di caching volta ad accelerare il recupero dei dati, che derivano dall'analisi testuale del codice, la quale è computazionalmente onerosa.

1.2.2 Dependability Criteria

Robustezza	Il sistema restituisce sempre un output che rispecchia le aspettative dell'utente. Cioè, nel caso in cui è possibile restituire una soluzione valida, il sistema la esegue. Nel caso in cui si verifichi un errore oppure non sia possibile restituire una soluzione valida, il sistema notificherà l'insuccesso all'utente.
Affidabilità	Il sistema non deve essere soggetto a frequenti casi di crash. In caso ciò accada il sistema deve essere in grado di notificare all'utente di eventuali input errati.
Disponibilità	Il sistema è disponibile all'uso dell'utente in qualsiasi momento dall'avvio dell'ambiente di sviluppo IntelliJ.
Tolleranza all'errore	Il sistema è in grado di riconoscere un possibile disservizio e gestirlo notificando l'errore all'utente e salvando i progressi fatti fino al guasto.
Sicurezza	Il sistema è usufruibile da qualsiasi utente che utilizza il plug-in perché non vengono memorizzati dati sensibili. Gli unici dati memorizzati sono parti di codice da rielaborare.

1.2.3 Cost Criteria

Costi di sviluppo	È stimato un costo complessivo di 4800 ore per la progettazione e lo sviluppo del sistema (80 ore per ogni membro del team).
Costi di amministrazione	È stimato un costo complessivo di 40 ore per l'amministrazione del sistema (40 ore per ogni PM del team).

1.2.4 Maintenance Criteria

Estensibilità	Il sistema è progettato in modo tale da poter essere esteso con altre funzionalità oppure ampliare la tipologia di smell che possono essere analizzati/corretti con le funzionalità già presenti.
Modificabilità	Il codice è stato scritto secondo paradigmi ingegneristici. Il tutto è stato scritto seguendo una specifica suddivisione in moduli, i quali rendono la struttura leggibile e intuitivamente modificabile.
Adattabilità	Il plug-in funziona su qualsiasi sistema operativo purché sia provvisto dell'applicativo IntelliJ IDEA. Non sarà possibile utilizzarlo su IDE diversi da questo (Eclipse, Visual Studio, NetBeans,...).
Leggibilità	Il sistema sarà fornito di una documentazione esauriente che avvalora la leggibilità della struttura del codice.
Portabilità	La portabilità è garantita dalla scelta implementativa adoperata, giacché Java è per sua natura un linguaggio votato alla portabilità.
Tracciabilità dei requisiti	La tracciabilità dei requisiti sarà possibile, si può retrocedere al requisito associato ad ogni parte del progetto. La tracciabilità sarà garantita dalla fase di progettazione fino al testing.

1.2.5 End user Criteria

Utilità	Il sistema si rivela utile poiché in assenza di questo l'utente avrebbe impiegato sforzi e tempi maggiori, oltre ai possibili errori di distrazione che a un occhio umano potrebbero sfuggire.
Usabilità	L'interazione fra il sistema e l'utente sarà molto semplice anche senza la consultazione della documentazione associata. L'interfaccia risulterà intuitiva e gradevole da usare, sia per novizi che per esperti.

1.2.6 Trade-off

Trade-off	
Manutenibilità vs Performance	Il reenginering del sistema da priorità alla manutenibilità suddividendo il codice del progetto in più moduli coerenti con le proprie finalità. Questa preferenza favorisce la leggibilità del codice per future operazioni di manutenzione, ma può gravare sensibilmente sulle performance.
Performance vs Memoria	Il reenginering del sistema da priorità alla performance. L'algoritmo di analisi viene eseguito una sola volta all'avvio del software per poi memorizzare i risultati in memoria primaria e secondaria tramite una meccanica di caching. Solo in caso di modifiche al codice verrà rieseguito l'algoritmo di analisi per aggiornare la cache e se queste sono effettuate su i file originali verrà rieseguito il caching anche sulla memoria persistente.
Performance vs Sicurezza	Il reenginering del sistema da priorità alle performance a discapito della sicurezza, poiché l'obiettivo del plug-in è quello di ridurre i tempi di lavoro dei vari utenti nella risoluzione di code smell. La sicurezza viene trascurata perché non vengono manipolati dati sensibili.
Manutenibilità vs Sicurezza	Il reenginering del sistema da priorità alla manutenibilità del plug-in. In questo modo l'utente può ampliare l'elenco di smell e aggiungere nuove funzionalità. Siccome sarà possibile mettere mano al codice, il livello di sicurezza può essere messo a rischio.
Usabilità vs Sviluppo rapido	Il reenginering del sistema da priorità all'usabilità del plug-in, realizzando una nuova e intuitiva interfaccia con la quale l'utente può interagire in modo rapido. Tuttavia i costi in termine di tempo di sviluppo aumentano.
Robustezza vs Portabilità	Il reenginering del sistema da priorità alla robustezza del plug-in, il quale dovrà restituire in ogni caso l'output sperato dall'utente. Il sistema è compatibile solo con l'ambiente di sviluppo IntelliJ IDEA.

1.3 Definitions, acronyms, and abbreviations

Code Smell : porzioni di codice scritte in maniera non ottimale, le quali non compromettono il funzionamento del sistema ma introducono debolezze di progettazione, riducendo la qualità complessiva del codice.
TACOR : acronimo per Textual Analysis for Code smell detectiOn and Refactoring.
ASCETIC : acronimo per Automated Code Smell Identification and Correction.
tf-idf : abbreviazione di term frequency inverse document frequency.
LSI : abbreviazione di Latent Semantic Indexing.

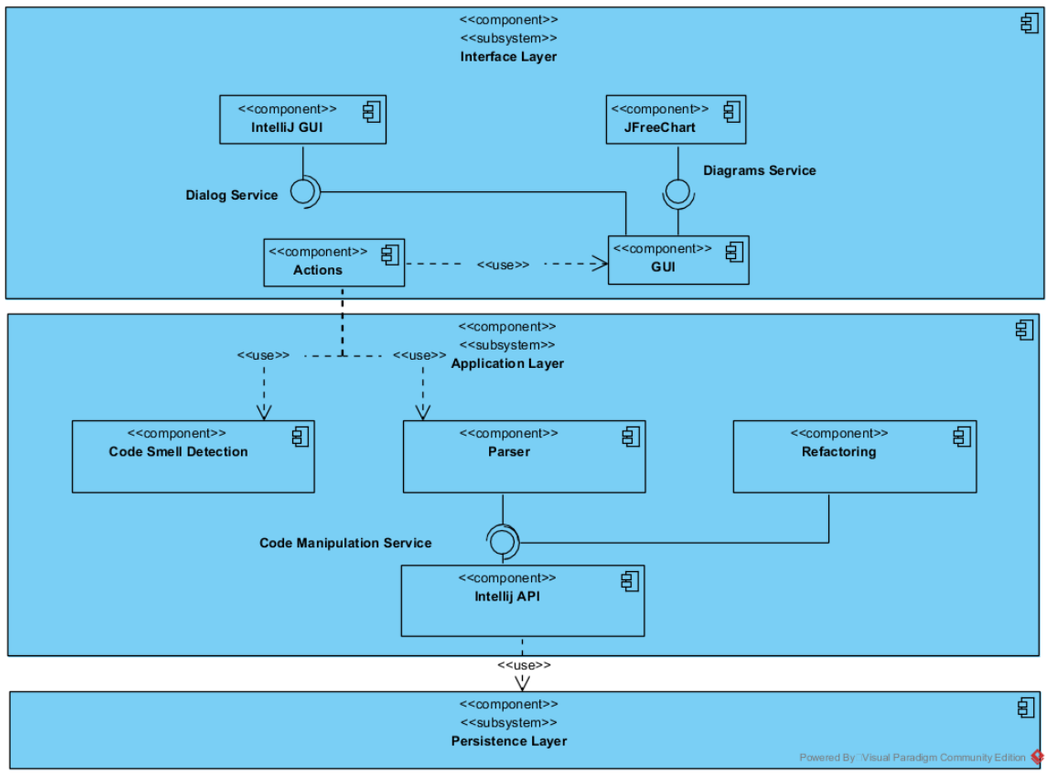
1.4 References

Si fa riferimento :

- al libro "Object-Oriented Software Engineering- using UML, Pattern and Java Bernd Bruegge Allen H.Dutoit"
- ai precedente documentazione riguardante il progetto.

2 Current software architecture

Il sistema corrente presenta una *three-tier architecture* in cui sono presenti i classici 3 layer: *presentation*, *application* e *storage*.



Current Architecture Overview

Il *presentation layer* include tutte le interfacce grafiche e le *actions*, ovvero componenti che, opportunamente configurate, eseguono codice in risposta a interazioni dell'utente con componenti grafiche estese dell'IDE. Queste componenti fungono, dunque, da connettori tra Intellij e il codice del plugin.

L' *application layer* contiene tutto il codice di business del plugin ed è composto da 3 componenti principali: *Code Smell Detection*, *Parser* e *Refactoring*. Il primo si occupa delll'individuazione dei code smell all'interno dei componenti codice sotto analisi, il secondo di ricavare dal codice in esame le informazioni necessarie al *Code Smell Detection* per effettuare le analisi, mentre il terzo si occupa di mettere in atto le operazioni di refactoring correttive dei *code smell*. Tutti e tre sono utilizzati e coordinati dalle classi *actions* e dagli *action listeners* delle interfacce grafiche e quindi utilizzati come se fossero un *Service Layer* che agisce su classi che mantengono solamente dati, i beans. Tutto ciò indica che il plugin presenta un'architettura, come la definisce Martin Fowler, *anemica*.

Lo *storage layer*, invece, è costituito dai files contenente il codice sorgente in esame. Esso non viene mai acceduto direttamente dal codice di business del plugin, ma attraverso le API offerte dalla piattaforma Intellij.

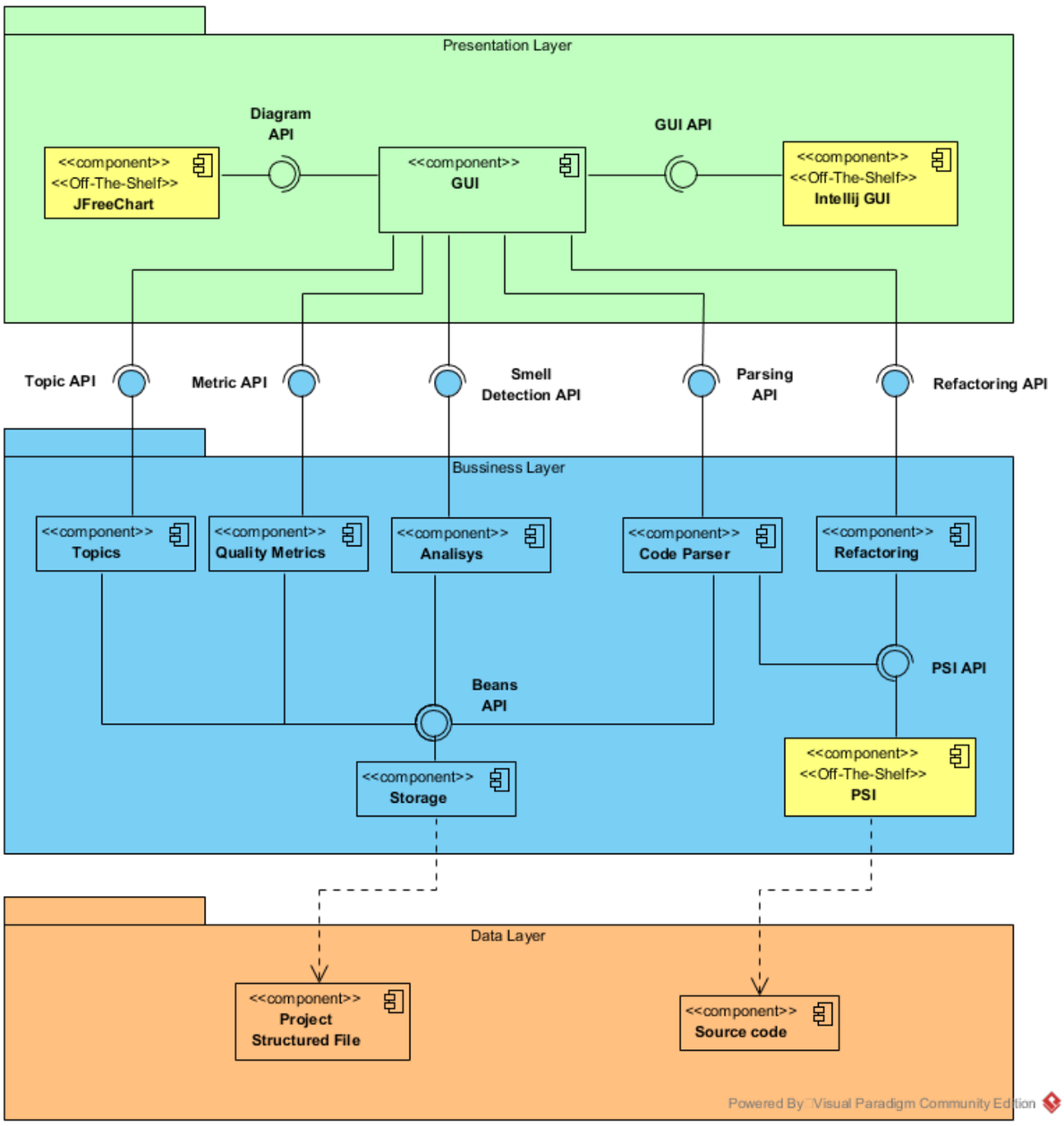
Le modifiche presenti nella sezione 3 consistono in una rimodulazione e ristrutturazione generale dell'architettura, volte a risolvere i problemi di manutenibilità e la mancata adesione all'object-orientation che derivano dall'architettura anemica del sistema. Inoltre, tale ristrutturazione rende più semplice l'aggiunta delle due nuove funzionalità di correzione di *Blob* e *Promiscuous Package*. Per ulteriori dettagli, si fa riferimento al documento di *Change Impact Analysis*.

3 Proposed software architecture

3.1 Overview

Le funzionalità del sistema verranno suddivise in component e layer logici in base alle differenti caratteristiche di queste. Sarà garantita massima coesione e minimo accoppiamento tra i sottosistemi in modo che i cambiamenti in un sottosistema non influiscano sugli altri.
Il sistema userà il modello architetturale MVC: impianto di memorizzazione (Model), presentazione (View) e business logic (Controller).

3.2 Subsystem decomposition



Component Diagram

Name	Descrizione
Analisi	Componente che si occupa di analizzare il codice java identificando, tramite metriche e topic ottenute da altre componenti, la presenza di code smell.
Refactoring	Componente che si occupa di generare una possibile soluzione ad un code smell selezionato presente all'interno del codice java in analisi, procedendo poi all'eventuale applicazione di quest'ultima risolvendo così il code smell identificato.
Storage	Componente gestisce l'accesso al file strutturato del progetto in analisi
GUI	Componente che realizza l'interfaccia grafica, la quale si appoggia su Intellij API(per componenti grafiche) e JFreeChart(per grafici).
Code Parser	Componente che prende la struttura elaborata dal PSI come input ed, estrapolando informazioni da questo, costruisce i Beans fornendo così una diversa rappresentazione strutturata dell'input.
Quality Metrics	Componente che si occupa del calcolo delle metriche qualitative quali LOC, WNC, RFC, CBO, LCOM, NOA, NOM, NOPA e NOP.
Topics	Componente che si occupa del calcolo dei topic ovvero dei termini più ricorrenti all'interno del codice analizzato.
Project Structured File	Componente che implementa una meccanica di cache realizzata in SQLite come singolo DB di cache per ogni progetto (descritta nel dettaglio nella sezione del Persistent data management).

3.3 Hardware/software mapping

Il sistema sottoposto a reenginering sarà installato sull'ambiente di sviluppo IntelliJ IDEA e utilizzerà la libreria SQLite. Poichè il sistema è embedded il collegamento con gli altri sottosistemi è gestito dall'IDE nel quale in plug-in viene eseguito.

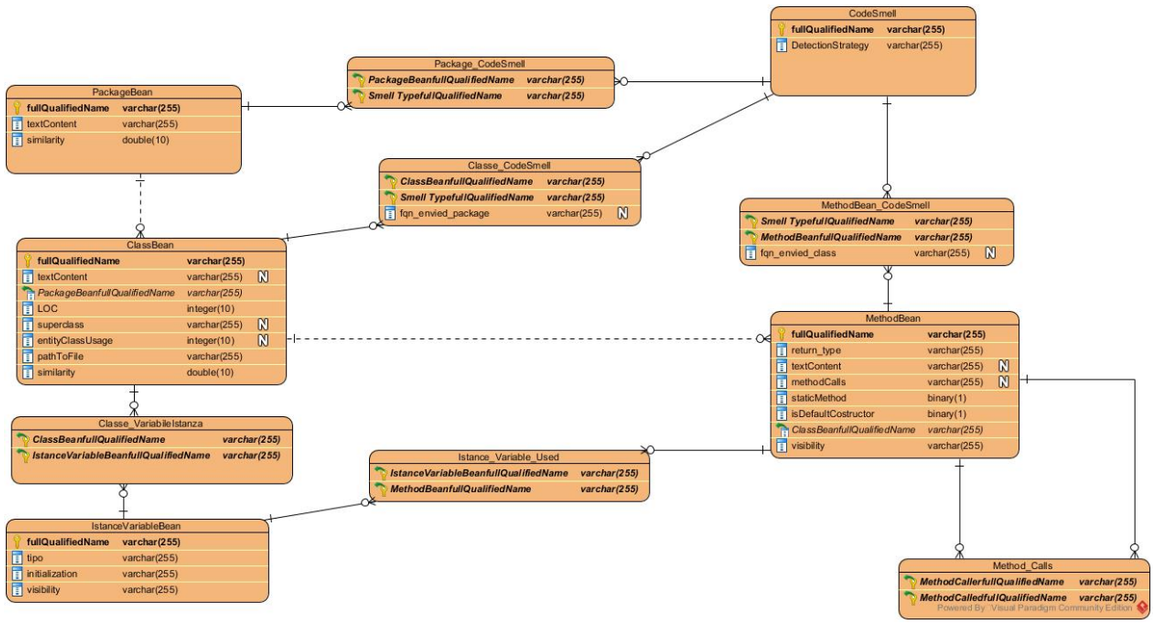
3.4 Component Off-The-Shelf

I componenti "off-the shelf" che useremo sono elencati di seguito, per i dettagli implementativi e, i motivi delle scelte si rimanda alla sezione API-Engineer e Trade-Off.

- **IntelliJ GUI** : La GUI che realizzeremo si appoggerà su API di Sistema a mezzo API di IntelliJ.
- **PSI (Program Structure Interface)** :
  - **Code Manipulation API** : Il modulo di Refactor tramite queste API fornite da IntelliJ, effettua l'Extract e il Move per l'elaborazione delle smells sulle classi o sui package.
  - **Compiler Services** : Il codice preso in esame dovrà sottoporsi al test di compilazione in modo tale da verificarne la correttezza lessicale, semantica e sintattica.
- **JFreeChart (DiagramAPI)** : Libreria Java che offre servizi grafici (grafici a barre, Istogrammi, a torta, grafico a radar). Nel nostro caso sarà utilizzato per rappresentare le RadarMap che rappresenteranno le topic implementate dalle varie componenti analizzate.

3.5 Persistent data management

In ASCETIC la persistenza dei dati è necessaria soprattutto per tenere traccia di informazioni importanti ai fini delle funzioni offerte dal sistema. ASCETIC infatti deve essere in grado di: memorizzare le informazioni temporali relative all'ultima modifica del progetto, ricordare la struttura di quest'ultimo, ossia memorizzare quali e quanti smell vi sono all'interno. Queste caratteristiche rendono più semplice ed immediato il chaching che il sistema sfrutta per mantenere un certo standard in materia di performance. La principale Data Source di ASCETIC è costituita dunque dal codice sorgente del programma preso in esame su IntelliJ: il sistema non sfrutta quindi un vero e proprio DBMS, nonostante l'utilizzo di SQLite (scelto per motivi di ottimizzazione implementativa), ma associa ad ogni progetto un file contenente le informazioni di cui sopra. Lo schema di seguito riportato non costituisce un diagramma ER, bensì uno schema logico rappresentativo del come i file verranno strutturati mediante SQLite.



3.6 Access control and security

ASCETIC è un sistema basato su singolo utente. Chiunque abbia accesso alla macchina su cui è installato il plug-in può usufruire di tutte le sue funzionalità, che nello specifico sono:

- Analisi
- Correzione
- Ignora correzione
- Ricorda azione

Non viene adottata alcuna tipologia di politica di sicurezza, poiché il plug-in è accessibile da un solo tipo di attore e non sono presenti dati personali che possano essere danneggiati.

3.7 Global software control

All'avvio del plug-in, un thread di controllo inizia la sua attività di analisi dell'intero progetto, creando, nel caso in cui sia stato effettuato il primo avvio, la cache. Per avvii successivi del plug-in, la cache viene aggiornata dal thread.
Il controllo del flusso è gestito da classi di tipo controller poiché il sistema è basato su architettura MVC. Il sistema utilizzerà un controllo di flusso event-driven perché le azioni saranno innestate da eventi tramite interazioni dell'utente con l'interfaccia grafica (button, menù, ...). Quindi la gestione del flusso converge tutta su un'entità controller, la quale smisterà le differenti azioni al componente adatto a svolgerle.

3.8 Boundary conditions

Le condizioni limite riguardano solo ed esclusivamente il caso in cui il sistema dovesse andare in crash, poiché i casi relativi all'accensione ed allo spegnimento dipendono dall'avvio e dall'arresto di IntelliJ.

3.8.1 Crash Sistema

Il caso di crash del sistema (dovuto a qualche malfunzionamento del plug-in) viene gestito in modo tale che il sistema mostri i log all'utente e successivamente ne effettui la distruzione insieme alla cache . Così facendo l'utente è costretto a rieffettuare le operazioni precedenti al crash.

ID	**CRASH**	
Participating actors	Sviluppatore	
Entry Condition	Lo Sviluppatore sta utilizzando ASCETIC su IntelliJ	
Flow of events	**UTENTE**	**SISTEMA**
	1. L'utente, durante l'utilizzo del plug-in, riscontra un crash del sistema dovuto ad un errore casuale.	
		2. Il sistema mostra un messaggio d'errore. Successivamente, alla ripresa del normale funzionamento, mostra i log degli errori riscontrati dopodiché procede all'invalidazione delle cache.
	3. L'utente prende visione delle notifiche del sistema e si appresta ad utilizzare di nuovo il plug-in, avendo cura di rieffettuare le operazioni precedenti al crash non soggette a salvataggio.	
Exit condition	Il sistema riprende a funzionare correttamente e sono state invalidate le cache.	
Exception condition		
Quality requirements		

4 Subsystem services

Name	**Servizi offerti**	**Servizi richiesti**	**Dipendenze**
Analisi	Smell Detectors API	Persistence API Beans API	
Refactoring	Refactoring API	PSI API	
GUI		GUI API Diagram API Metric API Topic API Smell Detectors API Parsing API Refactoring API	
Code Parser	Parsing API	Beans API PSI API	
Quality Metrics	Metric API	Beans API	
Topics	Topic API	Beans API	
Storage	Beans API		Project Structured File
Project Structured File			

- **Beans API** : Permette la gestione dei Beans, fornendo così una rappresentazione ulteriormente strutturata del codice in analisi.
- **Smell Detectors API** : Analizza testualmente il codice strutturato in Bean, individuando i vari tipi di code smell (Feature envy, Misplaced class, Blob, Promiscue package).
- **Parsing API** : Esegue l'operazione di avvio dell'analisi con il conseguente riempimento del file.
- **Refactoring API** : Esegue l'operazione di refactoring del codice analizzato in seguito alla generezione di una possibile soluzione, da parte del sistema o dell'utente dove concesso.
- **Metric API** : Implementeremo una meccanica di calcolo e rilevamento di varie metriche qualitative utili all'analisi della porzione di codice designata.
- **Topic API** : Implementeremo una meccanica di rilevamento di topic quali termini più ricorrenti utili all'analisi della porzione di codice designata.

5 Glossary

- **Blob:** Tipoligia di code smell. E' una Classe che implementa responsabilità molto diverse tra di loro.
- **Feature Envy:** Tipoligia di code smell. E' un Metodo che è maggiormente interessato a variabili e metodi di una classe differente dalla sua.
- **IntelliJ IDEA:** IntelliJ IDEA è un ambiente di sviluppo integrato per il linguaggio di programmazione Java. E' stato sviluppato da JetBrains.
- **Misplaced Class:** Tipoligia di code smell. E' una Classe che non ha alcuna attinenza con le classi dello stesso package.
- **Promiscuous Package:** Tipoligia di code smell. E' un Package contenente classi che hanno responsabilità diverse.
- **TACOR:** Acronimo per Textual Analysis for Code smell detectiOn and Refactoring. E' 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.