UNIVERSITÁ DEGLI STUDI DI SALERNO

Dipartimento di Informatica

Corso di Laurea Triennale in Informatica

TESI DI LAUREA

Progettazione ed implementazione di un plug-in IntelliJ per l'identificazione e il refactoring automatico di Code Smell

RELATORE

Prof. **Andrea De Lucia**

Dott. **Fabiano Pecorelli**

Universitá degli studi di Salerno

CANDIDATO

Michele Simone Gambardella

Matricola: 0512104502

Anno Accademico 2018-2019

Non vi sono certezze, solo opportunità

Abstract

Il ciclo di vita di un sistema software è soggetto a continui cambiamenti, i quali possono portare ad un deterioramento del design iniziale, ad una perdita di qualità del codice e all'introduzione di soluzioni inadeguate. Tali problemi spesso si manifestano nel codice in termini di code smell. I code smell non sono (e non rivelano) "bug", cioè veri e propri errori, bensì debolezze di progettazione che riducono la qualità del software, a prescindere dall'effettiva correttezza del suo funzionamento. In letteratura, molti sono gli approcci che puntano all'identificazione e successivo refactoring di code smells, tuttavia, l'individuazione e la correzione di code smell non sono mai definite in termini assoluti, ma richiedono sempre un elemento di giudizio soggettivo da parte del programmatore. L'oggetto di questo lavoro di tesi è il miglioramento delle prestazioni di ASCETIC (Automated Smell CodE idenTIfication and Correction), plug-in per l'identificazione automatica di code smell sviluppato come progetto universitario. Precedentemente a questo lavoro di tesi, ASCETIC era in grado di rilevare 4 tipi di code smell di diversa natura e a diversi livelli di granularità (i.e, Feature Envy, Blob, Misplaced Class e Promiscuous Package) tramite algoritmi basati sull'analisi testuale e proponeva una possibile soluzione di refactoring per rimuovere i code smell identificati. Se richiesto, il plug-in era inoltre in grado di effettuare in automatico le operazioni di refactoring sfruttando le API di IntelliJ IDEA per la manipolazione del codice sorgente.

Il più importante tra i miglioramenti apportati durante lo sviluppo di questo progetto di tesi riguarda l'aggiunta di algoritmi di analisi strutturale per fornire una tecnica alternativa a quella già presenteper la rilevazione di Feature Envy, Blob, Misplaced Class e Promiscuous Package. Collegando l'IDE IntelliJ a GitHub è possibile avviare automaticamente il plug-in al momento del commit, in modo tale da mettere in guardia lo sviluppatore qualora egli stia effettuando delle modifiche che introducono una o più smell. ASCETIC, inoltre, è ora configurabile consentendo agli sviluppatori di effettuare l'analisi tramite delle soglie personalizzabili, settabili da una apposita schermata, e di scegliere la tecnica di analisi da usare tra quelle presenti. Il plug-in, infine, presenta una nuova interfaccia grafica e fornisce suggerimenti sulla priorità di correzione degli smell identificati in base alla loro gravità.

Indice

Glossario

- **IntelliJ IDEA:** IntelliJ IDEA è un ambiente di sviluppo integrato per il linguaggio di programmazione Java. E' stato sviluppato da JetBrains.
- **ASCETIC:** Acronimo per Automated Smell CodE idenTIfication and Correction. E' un plug-in, sviluppato per l'IDE IntelliJ IDEA, creato per risolvere il problema dei "code smell" mediante l'analisi testuale.
- **Feature Envy:** Tipoligia di code smell. E' un Metodo che è maggiormente interessato a variabili e metodi di una classe differente dalla sua.
- **Misplaced Class:** Tipoligia di code smell. E' una Classe che non ha alcuna attinenza con le classi dello stesso package.
- **Blob:** Tipoligia di code smell. E' una Classe che implementa responsabilità molto diverse tra di loro.
- **Promiscuous Package:** Tipoligia di code smell. E' un Package contenente classi che hanno responsabilità diverse.
- **PSI** (Program Structure Interface): è la radice di una struttura che rappresenta il contenuto di un file come una gerarchia di elementi in un particolare linguaggio di programmazione.

Elenco delle figure

Elenco delle tabelle

CAPITOLO 1

Introduzione

1.1 Contesto applicativo

Sviluppare software è un processo alquanto complesso: sono necessarie, infatti, fondamentali e delicate procedure di studio e progettazione che precedono la fase di programmazione vera e propria. Inizialmente, si ha una fase di raccolta e analisi dei requisiti, volta a comprendere appieno le richieste dell'utente che dovrà interagire con esso, e successivamente, vi è una fase di progettazione, la quale ha come scopo quello di definire come il software dovrà essere strutturato e implementato e secondo quale architettura dovrà essere organizzato. Segue la fase di implementazione, durante la quale verrà effettivamente creato il software seguendo quanto stabilito nelle fasi precedenti. Infine, si procede con la delicata e fondamentale fase di testing che ha come obbiettivo quello di verificare se, effettivamente, il software soddisfa i requisiti definiti in fase di analisi. In caso di fallimento dei test, secondo precise tecniche vengono apportate delle correzione al fine di identificare e rimuovere gli errori. Il ciclo di vita del software non coincide, tuttavia, con il suo ciclo di sviluppo, poiché una volta consegnato al cliente sarà necessario, con il tempo, procedere con continue e periodiche modifiche al fine di preservare le sue funzionalità. Riguardo ad una migliore comprensione del fenomeno è possibile ricorrere alla prima legge di Lehman sull'evoluzione del software:

> ***Cambiamento continuo**: Un sistema usato in ambienti reali necessariamente deve cambiare o diventerà progressivamente meno utile in quell'ambiente.*

La necessità del software di evolvere si traduce in interventi di modifica per adeguarlo a nuove esigenze, richieste dall'utente o dall'ambito di esecuzione, identificate durante il processo di manutenzione. Tale fase del software, tuttavia risulta molto delicata e difficile da gestire. In un sistema progettato e sviluppato seguendo il paradigma Object Oriented, attualmente il più utilizzato e su cui ricade questo studio, risulta fondamentale preservare la qualità del software in vista di future e continue modifiche a cui è destinato, in quanto queste ultime spesso lo allontanano dal design originario compromettendone l'efficienza. I suoi problemi dipendono in gran parte dalla mancanza di controllo e disciplina durante l'analisi e la progettazione del software.

Di fatto, più un software è complesso, più risulta onerosa e complessa la sua manutenzione. I principi di buona programmazione vengono posti quindi in secondo piano e si procede ad una implementazione della nuova richiesta di cambiamento nella maniera più diretta. Questo modo di procedere, tuttavia, comporta un aumento della complessità del codice e un progressivo deperimento della sua qualità, rispetto alle linee guida OOP [1]. Ciò si traduce in una progressiva complicazione della struttura del software, come spiega la seconda legge di Lehman sulla manutenzione del software:

> ***Entropia crescente**: Quando un sistema cambia, la sua struttura tende a diventare più complessa. Risorse extra devono essere utilizzate per preservare e semplificare la sua struttura.*

Vengono così introdotti dei debiti tecnici [2] come, ad esempio, i code smell (o più semplicemente smell). I code smell non sono (e non rivelano) "bug", cioè veri e propri errori, bensì debolezze di progettazione che riducono la qualità del software, a prescindere dall'effettiva correttezza del suo funzionamento. L'operazione, usata dai programmatori, che prende il nome di *Refactoring*, identifica l'esecuzione di azioni di ristrutturazione del codice volte a migliorarne la struttura, abbassandone la complessità e aumentandone la qualità senza modificarne le funzionalità [1].

La qualità di un software che segue il paradigma orientato agli oggetti può essere definita in termini di *coesione* e *accoppiamento*. Codice affetto da code smell può presentare quindi una bassa coesione interna ed un alto accoppiamento.

- La *coesione* misura il grado di interdipendenza tra le componenti all'interno di un singolo modulo.

- L'*accoppiamento* indica il grado di interdipendenza tra i moduli.

Per modulo si intende un componente software autonomo e ben identificato, e quindi facilmente riusabile. La programmazione modulare è un paradigma di programmazione nato con lo scopo di favorire la riusabilità dei componenti software. Quindi ad esempio, se una classe ha alta coesione, i suoi metodi realizzano compiti simili e sono collegati gli uni agli altri. Poiché una classe dovrebbe occuparsi di una singola responsabilità, è desiderabile un alto grado di coesione. Se due classi sono scarsamente accoppiate sono relativamente indipendenti, quindi le modifiche apportate ad una avranno un impatto limitato sull'altra. La situazione ideale in tal caso sarebbe quella di avere classi con alta coesione e basso accoppiamento. Una buona decomposizione funzionale e l'identificare e risoluzione dei code smell permette quindi di migliorare le performance del software raggiungendo tale obbiettivo. Tuttavia ciò è molto complesso in quanto, per aumentare la coesione occorre incrementare le responsabilità interne al modulo. Ciò, però, ha l'effetto di aumentare l'interdipendenza delle classi interne, che si traduce in un aumento dell'accoppiamento.

1.2 Motivazioni e Obiettivi

Durante lo sviluppo di un software è importante avere un occhio di riguardo alla qualità del prodotto, preservandola per agevolare futuri cambiamenti e operazioni di manutenzione. Gli sviluppatori devono però solitamente rispettare vari vincoli, quali consegne strette e vicine, e ciò provoca l'impossibilità di trovare una soluzione che consenta di ottenere il massimo guadagno in ogni aspetto. Vari fattori tecnici provocano l'introduzione di code smell: difficoltà nel comprendere codice scritto da altri sviluppatori, mancanza di competenze o di documentazione completa o consistente, vincoli temporali troppo stretti, software non progettato in vista di future modifiche, sono solo alcuni dei motivi per cui il lavoro di manutenzione può essere difficoltoso.

Da anni i code smell sono oggetto di studio, ed è emerso quanto questi siano longevi dopo la loro introduzione e persistano nelle varie versioni dei software, impattando negativamente sui requisiti non funzionali di questi ultimi, in particolare sulla comprensione e la manutenibilità. A seguito di tali rivelazioni sono stati definiti vari metodi di individuazione degli smell e, dove possibile, sono state sviluppate tecniche di refactoring volte a correggerli. Sulla base di questi studi, numerosi sono stati i tool nati con lo scopo di analizzare codice sorgente, rilevare e correggere i code smell presenti. Tali tools prendono in esame varie metriche di

qualità del software, come, ad esempio, le proprietà strutturali dei componenti. Benché questi approcci si sono mostrati molto validi, è stato mostrato come una tecnica basata sull'analisi testuale risulta essere molto più efficace: TACO (Textual Analysis for Code smell detectiOn), uno smell detector basato su Information Retrieval, è stato proposto a supporto di tale ipotesi [2].

ASCETIC (Automated Smell CodE idenTIfication and Correction) è un plug-in per l'identificazione e correzione automatica di code smell sviluppato per l'IDE IntelliJ IDEA. ASCETIC è in grado di riconoscere 4 tipi di code smell: Feature Envy, Misplaced Class, Blob e Promiscuous Package. Attualmente tale plug-in presenta un approccio di analisi unicamente basato su metriche testuali. Lo scopo di questo lavoro di tesi consiste nel perfezionamento delle tecniche di analisi, raffinando la correttezza di quelle già presenti e inglobando anche tecniche di analisi basate su metriche strutturali, permettendo, inoltre, una migliore interazione e maggiore comprensione da parte dell'utente durante le fasi volte alla correzione automatica degli smell individuati.

1.3 Risultati

Come ci si era prefissati, è stato realizzato un plug-in per IntelliJ IDEA che consente la possibilità di analizzare progetti java allo scopo di trovare e correggere, se richiesto, quattro tipi di smell. L'algoritmo di parsing e quello di splitting sono stati perfezionati mentre le interfacce grafiche e gli algoritmi d'analisi hanno ricevuto un importante revisione e ampliamento, poiché oggetto dello studio di tesi, per l'aggiunta dei nuovi requisiti.

Si è scelta la piattaforma IntelliJ, non solo per il suo sempre più diffuso utilizzo, ma anche per la semplicità e la vasta disponibilità di API per lo sviluppo di plug-in.

1.4 Struttura della tesi

Nei prossimi capitoli sono esposti in dettaglio tutti gli argomenti ai quali si è accennato. Questo lavoro di tesi si compone di quattro capitoli, incluso il presente. Di seguito i dettagli relativi a tale suddivisione.

- Capitolo 2: Tecniche di Code Smell Detection e Refactoring: Nell'introduzione del capitolo vengono definiti i quattro tipi di smell trattati in questo studio di tesi. Nella

prima sezione poi vengono descritte le principali tecniche presenti in letteratura che si occupano dell'individuazione di code smell, mentre il loro refactoring viene trattato nella seconda sezione; il tutto soffermandosi in particolare sugli approcci implementati.

- Capitolo 3: Plug-in ASCETIC: Illustra nello specifico il progetto ASCETIC, soffermandosi in particolare sulle scelte implementative effettuate durante il suo sviluppo e sul come utilizzare i metodi di identificazione degli smell e di refactoring automatico presenti.

- Capitolo 4: Conclusioni e sviluppi futuri: In questo capitolo vi sono alcune osservazioni sul lavoro sviluppato e sulle prospettive future.

CAPITOLO 2

Tecniche di Code Smell Detection e Refactoring

In seguito a numerosi studi effettuati nell'ambito dei *code smell*, nel tempo, sono stati sviluppate diverse metodologie volte all'identificazione e correzione automatica di questi. Tutte le metodologie esistenti nascono dalle definizioni presenti in quattro libri riguardanti i difetti di design del codice sorgente e le euristiche per la loro identificazione.

1. Webster [3] descrive i rischi nello sviluppo Object-Oriented, spaziando dalla gestione di un progetto fino a trattare i problemi legati alle diverse scelte implementative. Nel libro ci si pone l'obbiettivo di fornire una descrizione sintetica sul modo di individuare ed eventualmente evitare molti dei potenziali problemi che possono sorgere in ognuna delle fasi del ciclo di sviluppo del software.

2. Nel testo di Riel [4] vengono definite più di sessanta linee guida per verificare l'integrità del design di un software. Sebbene le varie euristiche coprano vari argomenti, non sono da intendersi come leggi infrangibili, dunque, all'occorrenza, qualcuna può essere trascurata.

3. Martin Fowler [1] descrive nel suo libro circa 22 tipi di *code smell* con le relative operazioni di refactoring che sono volte a risolverli. In questo caso, tuttavia, ci si sofferma solamente su difetti di design che hanno a che fare con lo sviluppo software.

4. Brawn et al. [5] espongono 40 smell classificandoli in smell di sviluppo, architetturali e manageriali e, relativamente ai primi, propongono anche delle euristiche per la loro identificazione.

Durante questo studio di tesi sono stati presi in considerazione quattro tipi di smell :

- Il **Feature Envy** è uno smell che si presenta a livello di metodo. Esso compare quando un metodo è maggiormente interessato in un'altra classe (chiamata anche *classe invidiata*) piuttosto che a quella in cui si trova. Spesso è caratterizzato da un gran numero di dipendenze verso la classe invidiata [1]. Solitamente, questa situazione influenza negativamente il grado di coesione e di accoppiamento della classe in cui il metodo è implementato. Il metodo che soffre di Feature Envy, infatti, tende a ridurre la coesione della classe di appartenenza, in quanto implementa responsabilità differenti da questa, e ne aumenta l'accoppiamento, a causa del gran numero di dipendenze con i metodi della classe invidiata. L'operazione che mira a risolvere tale smell è il Move Method Refactoring, che sposta il metodo affetto nella classe più affine.

- Il **Misplaced Class** è l'equivalente a livello di classe del Feature Envy. In maniera analoga al Feature Envy, ma ad un livello di granularità più alto, una Misplaced Class si presenta quando una classe si trova in un package contenete classi semanticamente e strutturalmente ad essa collegate [1]. Come conseguenza diretta di questa situazione si ha una perdita di coesione del package e un aumento dell'accoppiamento tra il package in cui è presente la classe e il cosiddetto package invidiato (in quanto la classe affetta tende ad accedere frequentemente alle classi del package invidiato). Dunque, occorre spostare la classe nel package ad essa più appropriato, applicando una operazione di Move Class Reafctoring. Approcci analoghi a quelli della rimodulazione [6] [7] [8] sono stati proposti per semplificare il processo di identificazione e correzione di questo smell.

- Il **Blob** (conosciuto anche come God Class) è una classe caratterizzata da grandi dimensioni, un gran numero di metodi e di attributi e da un'alta dipendenza con le altre classi [1], con una conseguente scarsa coesione ed alto accoppiamento. Esso si trova spesso in design dove è presente un'unica classe che monopolizza le operazioni, mentre le altre classi hanno l'unico compito di incapsulare dati. Si tratta, dunque, di una classe che implementa troppe responsabilità, spesso molto distante tra loro, ed è il risultato di una allocazione inappropriata di requisiti. Spesso lo si ritrova come conseguenza di uno sviluppo iterativo, dove un abbozzo di codice evolve nel tempo in un prototipo, ed eventualmente, in un sistema in produzione. Una non corretta ripartizione dei requisiti, durante lo sviluppo iterativo, può portare, dunque, un modulo a prevalere sugli altri. Un Blob è spesso accompagnato da codice non necessario, rendendolo difficile da differenziare quali sono le funzionalità utili della classe Blob e qual è il codice che

non viene più usato. Per risolvere questo tipo di smell occorre applicare un Extract Class Refactoring, che mira a dividere la componente affetta in due o più classi con responsabilità più coese.

- Il **Promiscuous Package** è l'equivalente a livello di package del Blob. Generalmente, infatti, il concetto di coesione viene applicato alle classi. I package, infatti, nei software Object-Oriented, raggruppano insieme le classi logicamente e strutturalmente correlate a localizzare i cambiamenti, tra le altre cose. Tali raggruppamenti dovrebbero essere fatti con cura, altrimenti si traducono in moduli difficili da capire e da mantenere. Durante l'evoluzione del software, tuttavia, la coesione dei package, allo stesso modo delle classi, tende ad erodersi. Per risolvere questo tipo di smell occorre applicare un Extract Package Refactoring, che mira a dividere la componente affetta in due o più package con responsabilità più coese. Molti approcci sono stati proposti per aggregare procedure con un alta coesione funzionale. La maggior parte di questi approcci si basano sull'identificazione di sotto grafi fortemente connessi nel grafo delle chiamate che rappresenta il programma.

2.1 Tecniche di Code Smell Detection

Le tecniche di Detection esistenti per l'individuazione dei code smell possono essere classificate in quattro tipologie, di seguito trattate. Tali metodi si basato sull'analizzare il contenuto informativo del codice in esame e cercando di individuare caratteristiche negative legate al design usato, ovvero i *code smell*. Di seguito viene fatta una rassegna delle tecniche di code smell detection presenti in letteratura, soffermandosi con particolare attenzione su quelle relative ai quattro smell trattati in questo studio di tesi. Attualmente in ASCETIC risultano però implementate esclusivamente le prime due tecniche esposte ovvero quella strutturale e quella testuale.

2.1.1 Tecnica di analisi strutturale

Gli approcci euristici identificano i code smell mediante regole di rilevamento basate su metriche software. Questa sottosezione è stata ben approfondita poiché tale approccio di analisi costituisce la più importante aggiunta al plug-in apportata in questo studio.

Il processo generale seguito da tali approcci consiste in due fasi:

- l'identificazione dei sintomi chiave che caratterizzano un code smell che può essere mappato a un insieme di soglie basate su metriche strutturali (ad esempio, se Linee di codice> k);
- la combinazione di questi sintomi, che porta alla regola finale per rilevare lo smell [9] [10] [11] [12].

Le tecniche di rilevamento che rientrano in questa categoria differiscono principalmente nell'insieme delle metriche strutturali sfruttate, che dipendono dal tipo di code smells da rilevare e da come vengono combinati i sintomi chiave identificati. La maggior parte degli approcci proposti ha ottenuto una tale combinazione impiegando operatori AND / OR [10] [11] [12], mentre quelli più recenti hanno adottato metodi di clustering [13].
Durante questo studio di tesi, per quanto riguarda l'identificazione strutturale, sono stati implementati ed aggiunti in ASCETIC gli approcci d'analisi esposti nel dettaglio nella Sezione 3.2.1.

Di seguito sono elencate ulteriori tecniche strutturali presenti in letteratura, di cui molte suddivise in base allo smell trattato, ma non implementate in ASCETIC.
Moha et al. [9] hanno introdotto in DECOR, un metodo per specificare e rilevare code smells utilizzando un DSL (Domain-Specific Language) utilizzando regole, chiamate "rule card", che descrivono le caratteristiche intrinseche di una componente affetta da uno smell.

Tsantalis et al. [13] hanno presentato JDEODORANT, uno strumento la cui prima versione è stata in grado di rilevare gli smells code di Feature Envy e suggerire opportunità di Move Method Refactoring. Successivamente, sono stati supportati altri code smells (ad es. State Checking, Long Method, and Blob) [15] [16]. Le strategie di rilevamento per questi smells si basano su metriche di codice che vengono quindi collegate tra loro mediante algoritmi di clustering supervisionato e soglie per tagliare i dendrogrammi risultanti. La valutazione empirica delle prestazioni di JDEODORANT ha mostrato la sua elevata precisione (in media, circa il 75%). Nonostante le buone prestazioni ottenibili con le tecniche discusse, il lavoro precedente [17] [18] ha messo in evidenza tre importanti limitazioni che potrebbero precludere il loro uso nella pratica:

- soggettività degli sviluppatori rispetto ai code smells rilevati da questi strumenti,

- scarso accordo tra i diversi rivelatori
- difficoltà nel trovare buone soglie da utilizzare per il rilevamento.

Gli autori hanno dimostrato che DECOR è in grado invece di identificare gli smells con una media di successo circa del'80%.

Tecniche per l'identificazione di Feature Envy

Un primo semplice modo per identificare questo smell nel codice è di visitare l'Abstract Syntax tree (AST) di un sistema software allo scopo di identificare, per ogni campo, l'insieme delle classi referenziate [19]. Dunque, usando una soglia è possibile discriminare i campi che hanno troppi riferimenti ad altre classi. Altre tecniche più sofisticate definite in letteratura sono in grado di identificare e richiedere la rimozione dello smell [20] [13].

Tsantalis et al. [13] hanno ideato una tecnica di individuazione direttamente integrata in JDeodorant basata su informazioni strutturali. Tale approccio usa un algoritmo di clustering analysis mostrato nella Figura 2.1. Dato un metodo M, l'approccio forma un insieme T di classi candidate in cui potrebbe essere spostato M (l'insieme T nella Figura 2.1). Questo insieme è ottenuto esaminando le entità (cioè attributi e metodi) delle altre classi a cui M accede (l'insieme S nella Figura 2.1). In particolare, ogni classe nel sistema contenente almeno una delle entità a cui accede M viene aggiunto a T. Successivamente, le classi candidate vengono ordinate (sort(T) nella Figura 2.1) in ordine discendente in base al numero di entità che sono accedute da M. Nei seguenti step, ciascuna classe T_i è analizzata per verificare la sua idoneità per essere la classe suggerita come destinataria del metodo. In particolare T_i deve soddisfare tre condizioni per essere considerata nell'insieme dei suggerimenti [13]:

- T_i non è la classe a cui appartiene M.
- M modifica almeno una struttura dati in T_i.
- Spostando M in T_i si soddisfa un insieme di comportamenti, preservando le precondizioni (ad esempio, la classe obbiettivo non contiene un metodo con la stessa firma di M).

L'insieme delle classi in T che soddisfano tutti e tre i suddetti requisiti vengono inserite nell'insieme dei suggerimenti. Se tale insieme non è vuoto, l'approccio suggerisce di spostare il metodo M nel primo candidato dell'insieme, seguendo l'ordine che era stato dato a T.

```
extractMoveMethodRefactoringSuggestions(Method m)
  T = {}
  S = entity set of m
  for i = 1 to size of S
    entity = S[i]
    T = T U {entity.ownerClass}
  sort(T)
  suggestions = {}
  for i = 1 to size of T
    if(T[i] != m.ownerClass &&
        modifiesDataStructureInTargetClass(m, T[i]) &&
        preconditionsSatisfied(m, T[i]))

          suggestions = suggestions U
                    {moveMethodSuggestions(m => T[i])}

    if(suggestions != {})
      return suggestions
    else
      for i = 1 to size of T
        if(T[i] = m.ownerClass)
          return {}
        else if preconditionsSatisfied(m, T[i])
          return moveMethodSuggestions(m => T[i])
    return {}
```

Figura 2.1: Identificazione implementata in JDeodorant

Al contrario, se l'insieme è vuoto, allora le classi in T vengono riconsiderate usando dei vincoli più tenui. In particolare, se la classe T_i è la proprietaria di M, allora non sarà necessaria alcuna operazione di refactoring, e quindi l'algoritmo termina. Altrimenti, l'approccio verifica se spostare M in T_i soddisfa i comportamenti preservando le precondizioni.

Tecniche per l'identificazione di Misplaced Class

Bavota et al. [21] hanno proposto l'impiego di MethodBook, già precedentemente usato per l'individuazione di Feature Envy [20], ad un livello di granularità più alto, per l'individuazione di questo tipo di difetto. Tale approccio utilizza RTM sull'intero package e calcola i package "amici" di ogni singola classe. Se l' "amico" più vicino corrisponde al package di appartenenza della classe, allora la componente non risulta affetta dallo smell. In caso contrario, viene rilevata la presenza di un Misplaced Class.

Tecniche per l'identificazione di Blob

L'identificazione del Blob è stata affrontata dai ricercatori sotto tre diversi punti di vista. Il primo consiste nell'uso di metriche qualitative e corrisponde a quello implementato in ASCETIC ed esposto in Tabella 2.2. Tale approccio è usato anche in DECOR (DEtection and CORrection of Design Flaws) [22]. Esso definisce una "rule card" (Figura 2.2) che descrive le caratteristiche di una classe affetta da Blob e le regole per l'identificazione.

```
RULE_CARD: Blob {
  RULE: Blob
        {ASSOC: associated FROM: mainClass ONE TO: DataClass MANY}
  RULE: mainClass
        {UNION LargeClassLowCohesion ControllerClass}
  RULE: LargeClassLowCohesion
        {UNION LargeClass LowCohesion}
  RULE: LargeClass
        {(METRIC: NMD + NAD, VERY_HIGH, 20)}
  RULE: LowCohesion
       {(METRIC: LCOM5, VERY_HIGH, 20)}
  RULE: ControllerClass
       {UNION (SEMANTIC: METHODNAME, {Process, Control, Command, Manage, Drive, System}),
           (SEMANTIC: CLASSNAME, {Process, Control, Command, Manage, Drive, System}}
  RULE: DataClass
       {(STRUCT: METHOD_ACCESSOR, 90)} };
```

Figura 2.2: Rule Card per l'identificazione di Blob

Un altro approccio prevede l'identificazione indiretta dei Blob. Fokaefs et al. [16] hanno proposto una tecnica che, dato in input un software, suggerisce di dividere un insieme di classi candidate come Blob in diverse classi, in modo da avere una migliore distribuzione di responsabilità. Per ogni classe, vengono analizzate le dipendenze strutturali tra le entità della classe (attributi e metodi): in particolare, la metrica su cui si fa più affidamento è la distanza di Jaccard, calcolata tra tutte le coppie di insiemi di entità della classe, in modo tale da raggrupparli in gruppi coesi di entità che possono essere estratti come classi separate. La distanza di Jaccard è calcolata come segue:

$$Jaccard(E_i, E_j) = 1 - \frac{|E_i \cap E_j|}{|E_i \cup E_j|}$$

dove E_i e E_j sono due insiemi di entità, il numeratore è il numero di entità in comune tra gli insiemi e il denominatore è il numero totale di entità univoche nei due insiemi. Se dividendo la classe in classi separate migliora la qualità complessiva del sistema, allora è stato identificato un Blob. L'approccio proposto da Fokaefs et al. è stato implementato come un plug-in di Eclipse, chiamato JDeodorant [23]. In maniera simile, Bavota et al. [24] hanno proposto un approccio basato sull'uso di analisi strutturale e concettuale per identificare un Blob.

Bavota et al. [20] hanno inoltre proposto l'uso della teoria dei giochi per individuare un bilanciamento tra la coesione e l'accoppiamento nella divisione delle responsabilità in varie classi. Nello specifico, la sequenza delle operazioni da eseguire è calcolata effettuando un gioco, in cui l'equilibrio di Nash [25] definisce il compromesso tra coesione e accoppiamento.

Tecniche per l'identificazione di Promiscuos Package

Cimitile e Visaggio [25] hanno proposto una tecnica basata su dominance tree per aggregare procedure in moduli riutilizzabili. Un miglioramento di questa tecnica è stata proposta da Shawn et al [26] per supportare una migliore comprensione del codice. Antoniol et al. [27] hanno invece proposto l'uso dell'analisi concettuale a ristrutturare l'organizzazione architettonica dei file di codice sorgente nei sistemi legacy. Si è visto come, però, l'uso combinato di misure semantiche e strutturali hanno portato un maggior successo.

2.1.2 Tecnica di analisi testuale

Per l'identificazione dei code smell presenti nel progetto tramite analisi testuale, uno dei metodi più usati riguardo lo studio della coesione interna. Tale tecnica testuale si basa su una serie di operazioni. 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 (term frequency inverse document frequency). Le componenti normalizzate vengono analizzate dallo Smell Detector, basato su LSI (Latent Semantic Indexing), 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.

Palomba et al. [2] hanno proposto, invece, un approccio basato sull'Information Retrival e l'analisi testuale tramite il calcolo della somiglianza tesuale tra le varie componenti in analisi. Tali tecniche sono state implementate in ASCETIC e vengono esposte nel dettaglio nella Sezione 3.2.1.

2.1.3 Tecniche di analisi dello storico

Mentre la maggior parte delle tecniche di rilevamento si basano solo su informazioni strutturali, molti code smell sono intrinsecamente caratterizzati da come gli elementi del codi-

ce cambiano nel tempo. Il cambiamento è, come già ribadito, una caratteristica fondamentale del ciclo di vita del software, e quanto più questo viene modificato e rimodellato più è facile che contenga smell dovuti ad una manutenzione non molto accurata. Tale tecnica permette una più accurata analisi dei code smell e addirittura l'identificazione di alcuni di questi non rintracciabili con altri metodi.

Palomba et al. [29] hanno proposto HIST (Historical Information for Smell deTection), un approccio, valutato in due studi empirici, che sfrutta le informazioni sulla cronologia dei cambiamenti per rilevare istanze di diversi code smells. Sono state fornite prove del fatto che i dati storici possono essere sfruttati con successo per identificare smell intrinsecamente caratterizzati dalla loro evoluzione nella storia del programma, come Divergent Change, Shotgun Surgery, Parallel Inheritance, Blob and Feature Envy. Il primo studio, condotto su venti progetti open source, mirava a valutare l'accuratezza di HIST nel rilevare casi di code smell sopra menzionati. I risultati indicano che la precisione di HIST varia tra il 72% e l'86%, e il suo richiamo varia tra il 58% e il 100%. Inoltre, i risultati del primo studio indicano che HIST è in grado di identificare code smell che non possono essere identificati da approcci competitivi basati esclusivamente sull'analisi del codice dell'istantanea di un singolo sistema. Quindi, è stato condotto un secondo studio volto a studiare in che misura i code smell rilevati da HIST (e dalle tecniche di analisi del codice della concorrenza) riflettano la percezione degli sviluppatori di scelte di progettazione e implementazione inadeguate. Durante lo studio è stato riconosciuto che oltre il 75% delle istanze di code smell identificate da HIST come reali problemi di progettazione / implementazione.

Ratiu et al. [30] ha proposto di utilizzare le informazioni storiche della sospetta struttura imperfetta per aumentare l'accuratezza del rilevamento automatico dei problemi.

2.1.4 Tecniche basate su machine learning

Al contrario degli approcci euristici, le tecniche appartenerti a questa categoria sfruttano un metodo supervisionato: più specificamente, un set di variabili indipendenti (dette anche "predittori") è usato per prevedere il valore di una variabile dipendente (cioè lo smell di una classe) utilizzando un classificatore di apprendimento automatico (ovvero Regressione Logistica [31]). Il modello può essere addestrato utilizzando una quantità sufficientemente grande di dati disponibili dal progetto in analisi, ovvero una strategia all'interno del progetto o utilizzando dati provenienti da altri progetti software (simili), ovvero una strategia tra più

progetti. Questi approcci differiscono chiaramente da quelli basati sull'euristica, poiché si basano su classificatori per discriminare gli smell delle classi piuttosto che su soglie predefinite su metriche calcolate. Kreimer [32] ha proposto un modello di previsione che, sulla base delle metriche del codice utilizzate come variabili indipendenti, può portare a valori elevati di accuratezza. Adotta gli ALBERI DI DECISIONE per rilevare due code smell (ad es. Blob e Long Method). Più tardi, Amorim et al. [33] hanno confermato i risultati precedenti valutando le prestazioni degli ALBERI DI DECISIONE su quattro progetti open source su media scala. Vaucher et al. [34] hanno studiato l'evoluzione di Blob basandosi su un classificatore NAIVE BAYES, mentre Maiga et al. [35] [36] hanno proposto l'uso di SUPPORT VECTOR MACHINE (SVM) e hanno mostrato che un tale modello può raggiungere una misura F di circa l'80%. L'uso di BAYESIAN BELIEF NETWORKS per rilevare istanze di Blob, Functional Decomposition e Spaghetti Code su programmi open source, proposti da Khomh et al. [37] [38] portano a una F-measure complessiva vicina al 60%. Allo stesso modo, Hassaine et al. [39] hanno definito un approccio di ispirazione immunitaria per la rilevazione degli odori di Blob, mentre Oliveto et al. [40] hanno usato una B-Splines per rilevarli. Più recentemente, alcuni autori hanno studiato la fattibilità dell'apprendimento automatico per rilevare cloni di codice [41] [42].

Khomh et al. [37], infatti, hanno proposto un approccio probabilistico basato su reti Bayesiane per l'identificazione del Blob, il quale fornisce un valore di probabilità per cui una classe possa essere uno smell, piuttosto che un semplice valore booleano. Questa è anche una delle caratteristiche principali dell'approccio basato sulle metriche di qualità e B-spline proposto da Oliveto et al. [40] per identificare le istanze di Blob nel codice. Dato un insieme di classi Blob è possibile ricavare la loro rma (rappresentata da una curva) che sintetizza la qualità della classe. In particolare, ciascun punto della curva è il valore di una specifica metrica di qualità. L'identificazione del Blob si ottiene confrontando la curva della classe analizzata con le curve dei Blob identificati in precedenza. Maggiore è la somiglianza, maggiore è la probabilità che la nuova classe sia un Blob.

Arcelli Fontana et al. hanno compiuto i progressi più rilevanti in questo campo [43] [44] [45]. Nel loro lavoro, hanno raggiunto vari obbiettivi:

- teorizzato che la ML potrebbe portare a una valutazione più obiettiva della pericolosità degli odori di codice [44]

- hanno fornito un metodo ML per valutare l'intensità dell'odore di codice [45]
- confrontato con 16 tecniche ML per il rilevamento di quattro tipi di odore di codice [43] che mostrano che ML può portare a valori di F-Measure vicini al 100%

Tuttavia, recentemente Di Nucci et al. [46] hanno dimostrato che, in uno scenario di casi d'uso reali, i risultati raggiunti da Arcelli Fontana et al. [43] non possono essere generalizzati, contrastando così la reale efficacia dell'apprendimento automatico per il rilevamento dei code smell.

Pecorelli et al. [47] hanno approfondito il confronto empirico tra le prestazioni delle tecniche basate sull'euristica e quelle basate sull'apprendimento automatico. I risultati chiave sottolineano la necessità di ulteriori ricerche volte a migliorare l'efficacia dell'apprendimento automatico e degli approcci euristici per il rilevamento di code smell: mentre DECOR generalmente ottiene prestazioni migliori rispetto a una base di apprendimento automatico, la sua precisione è ancora troppo bassa per renderla utilizzabile nella pratica.

2.2 Tecniche di Refactoring

In ingegneria del software, il refactoring (o code refactoring) è una tecnica strutturata per modificare la struttura interna di porzioni di codice senza modificarne il comportamento esterno [1], applicata per migliorare alcune caratteristiche non funzionali del software quali la leggibilità, la manutenibilità, la riusabilità, l'estendibilità del codice nonché la riduzione della sua complessità, eventualmente attraverso l'introduzione a posteriori di design pattern [48]. Si tratta di un elemento importante delle principali metodologie emergenti di sviluppo del software (soprattutto object-oriented), per esempio delle metodologie agili, dell'extreme programming, e del test driven development. Le tecniche di refactoring argomentate di seguito sono implementate in ASCETIC ed adoperate tramite API di IntelliJ. Esse vengono utilizzate per modificare la componente affetta da smell, rimuovendo quest'ultimo, senza comprometterne il funzionamento.

2.2.1 Move Method Refactoring

La tecnica del Move Method viene usata per eliminare lo smell del Feature Envy, gestendo un metodo che viene utilizzato più in un'altra classe che nella propria. L'operazione eseguita consiste nel creare un nuovo metodo nella classe che utilizza maggiormente il metodo, quindi sposta il codice dal vecchio metodo nella classe invidiata. Il codice del metodo originale

viene invece trasformato in un riferimento al nuovo metodo nell'altra classe oppure rimosso completamente se non vi sono altri richiami nella classe che lo conteneva.
I vantaggi ottenuti sono vari:

- Spostando un metodo in una classe che contiene la maggior parte dei dati utilizzati dal metodo, si rende le classi più coerenti internamente.
- Spostando un metodo si può ridurre o eliminare la dipendenza tra la classe che chiama il metodo e la classe in cui si trova questo. Ciò può essere utile se la classe chiamante dipende già dalla classe in cui si prevede di spostare il metodo.

Meccanica del refactoring:

1. Verificare tutte le funzionalità utilizzate dal vecchio metodo nella sua classe. Potrebbe essere una buona idea anche spostarli. Di norma, se una funzione viene utilizzata solo dal metodo in esame, è consigliabile spostarla su di essa. Se la funzione viene utilizzata anche da altri metodi, è necessario spostare anche questi metodi. A volte è molto più semplice spostare un gran numero di metodi piuttosto che stabilire relazioni tra loro in classi diverse.
2. Assicurarsi che il metodo non sia dichiarato in superclassi e sottoclassi. In tal caso, ci si deve astenere dal trasferirlo oppure implementare una sorta di polimorfismo nella classe del destinatario al fine di garantire le varie funzionalità del metodo suddiviso tra le classi.
3. Dichiarare il nuovo metodo nella classe destinatario. È possibile che si desideri assegnare un nuovo nome al metodo più appropriato per esso nella nuova classe.
4. Decidere come far riferimento alla classe del destinatario. Potrebbe già essere presente un campo o un metodo che restituisce un oggetto appropriato, ma in caso contrario, si deve scrivere un nuovo metodo o campo per memorizzare l'oggetto della classe destinatario.
5. Una volta ottenuto un modo per fare riferimento all'oggetto destinatario e un nuovo metodo nella sua classe, si può trasformare il vecchio metodo in un riferimento al nuovo metodo.
6. Infine, si valuta se eliminare del tutto il vecchio metodo. In tal caso, inserire un riferimento al nuovo metodo in tutti i luoghi che utilizzano quello precedente.

2.2.2 Move Class Refactoring

La tecnica del Move Class viene usata per eliminare lo smell del Misplaced Class, gestendo una classe che si trova in un package che contiene altre classi a cui non è correlata nella funzione. L'operazione eseguita consiste nello spostare la classe in un package più pertinente, oppure creare un nuovo package se richiesto per un utilizzo futuro. Le classi vengono spesso create in package vicini al punto in cui vengono utilizzate, questo può avere senso fino a quando la classe inizia a essere riutilizzata da altre parti del prodotto. Il package in questione potrebbe anche essere diventato troppo grande (si preferisce, di norma, che i package non abbiano mai più di circa 10 classi).
I vantaggi ottenuti sono vari:

- Spesso è meglio spostare una classe in un package più correlato ad essa nella forma o nella funzione. Ciò può aiutare a rimuovere dipendenze complesse a livello di package e facilitare agli sviluppatori la ricerca e il riutilizzo delle classi.
- Se ci sono molte dipendenze per la classe all'interno del proprio package, Extract Class (esposto nella sezione 2.2.3) potrebbe essere usato per separare le parti rilevanti.
- Può essere utilizzato per spostare gli oggetti delle risorse String in un sotto-package per semplificare la compilazione.

Meccanica del refactoring:

1. Spostare la classe nella sua nuova cartella nell'albero dei sorgenti.
2. Rimuovere tutti i file di classe generati dalla compilazione di questa classe nella sua posizione precedente.
3. Modificare l'istruzione del package nel file di origine per riflettere il nuovo package.
4. Creare istruzioni di importazione per tutte le classi dipendenti nel package originale.
5. Compilare e testare la classe nel suo nuovo package, aggiornando e spostando i test unitari come richiesto usando questo stesso metodo.
6. Modificare e creare in alcuni casi le istruzioni di importazione su qualsiasi classe di dipendente, più semplice se non vengono utilizzate importazioni estese.
7. Controllare le classi che potrebbero creare un'istanza dinamica di questa classe utilizzando java.lang.reflecto o la ResourceBundleAPI.

8. Compilare e testare tutto il codice dipendente dalla classe originale.

9. Si deve prendere in considerazione l'applicazione della tecnica di Extract Package quando ci sono molte classi di funzionalità diverse in un determinato package.

2.2.3 Extract Class Refactoring

La tecnica del Extract Class viene usata per eliminare lo smell del Blob, gestendo una classe che si occupa di troppe funzionalità indipendenti tra loro. Tale componente viene scorporato in diverse classi seguendo una precisa logica di splitting. Questo metodo di refactoring contribuirà a mantenere l'adesione al principio di responsabilità unica, aumentando inoltre la leggibilità del codice. Le classi a responsabilità singola sono più affidabili e tolleranti nei confronti dei cambiamenti. Ad esempio, supponendo di avere una classe responsabile di dieci funzionalità diverse, la manutenzione di una di queste potrebbe provocare il malfunzionamento di un'altra.

Meccanica del refactoring:

1. Come primo passo è necessario decidere come esattamente dividere le responsabilità della classe e crearne di nuove in base a quanto definito. La classe in esame viene scansionata per poter estrarre una matrice method-by-method, di dimensioni *nxn*, dove n è il numero di metodi della matrice.

2. Una generica entry $c_{i,j}$ della matrice rappresenta la probabilità con cui il metodo m_i e il metodo m_j debbano stare nella stessa classe. Tale probabilità è calcolata come misura dell'accoppiamento tra i metodi, ottenuta attraverso una media ponderata tra somiglianza strutturale tra i metodi (SSM) [49], la dipendenza delle chiamate tra i metodi (CDM) [50] e la somiglianza concettuale tra i metodi (CSM) [51]. Dopo aver costruito la matrice method-by-method, la sua chiusura transitiva è calcolata in modo da estrarre le catene di metodi fortemente correlati tra loro; ogni catena rappresenta l'insieme delle responsabilità, cioè i metodi che dovrebbero essere raggruppati in una nuova classe. Avendo come riferimento tre soglie legate alle metriche prima citate, si decide come raggruppare quindi i vari metodi.

3. Utilizzare Move Class e Move Method per ciascun campo e metodo che si è deciso di spostare nella nuova classe. Per i metodi, iniziare con quelli privati al fine di ridurre il rischio di commettere un gran numero di errori. Riposizionare un pò alla volta le

varie componenti e testare i risultati dopo ogni azione, al fine di evitare un accumulo di errori risolti alla fine.

4. Una volta terminate le varie operazioni di spostamento, è necessario controllare nuovamente la vecchia classe che, avendo responsabilità modificate, può essere rinominata per maggiore chiarezza. Controllare, infine, se è possibile eliminare le relazioni reciproche tra le classe, se presenti.

Durante tali operazioni è importante riflettere anche sull'accessibilità della nuova classe dall'esterno. Si può nascondere la classe completamente dal client rendendola privata, gestendola tramite i campi della vecchia classe. In alternativa, si può renderla pubblica consentendo al client di modificare i valori direttamente. Tale decisione è dovuta alla fiducia risposta sull'efficacia del metodo di refactoring, il quale non dovrebbe alterare il funzionamento del codice.

2.2.4 Extract Package Refactoring

La tecnica del Extract Package viene usata per eliminare lo smell del Promiscuous Package, gestendo un package poco comprensibile che ha troppe classi, le quali ricoprono responsabilità indipendenti tra loro. Generalmente un package con classi debolmente accoppiate sia da un punto di vista semantico che strutturale esibisce un basso livello di coesione. In questo modo, estraendo uno o più package dall'originale ne incrementa il grado di coesione. Tale componente viene suddivisa in diversi package in base alle dipendenze e al loro utilizzo.
I vantaggi ottenuti sono vari:

- Le dipendenze alla fine causeranno problemi in progetti di qualsiasi dimensione, quindi ha senso iniziare il refactoring il prima possibile per chiarire quale parte del codice utilizza cosa ed evitare il propagarsi di errori nel tempo.

- Questo tipo di modifica può rendere il codice più flessibile. Ad esempio se si sta scrivendo del codice e si decide di modificarlo, a meno una strutturazione corretta, avrai difficoltà a riutilizzare determinati componenti. Il packaging è un modo per rendere esplicite le dipendenze.

- Questo refactoring può essere utile quando un package diventa troppo grande per essere facilmente compreso. Ciò semplifica l'identificazione dell'uso di una classe da parte della sua associazione implicita in un package.

Bavota et al. [27] hanno proposto un approccio che analizza le relazioni (strutturali e semantiche) tra classi in un pacchetto per suggerire una possibile ri-modulazione in due o più packages. L'uso combinato di metriche di accoppiamento strutturale e semantico consente loro di analizzare la coesione del pacchetto dal punto di vista delle dipendenze tra il classi e il punto di vista delle responsabilità delle classi dal package.

Meccanica del refactoring:

1. Identificare come raggruppare le varie classi creando una matrice *nxn* chiamata class-by-class, dove n è uguale al numero delle classi nel package. Una generica entri della matrice $m_{i,j}$ rappresenta la probabilità che la classe c_i e la classe c_j debbano stare nello stesso package.

2. Calcolare, per ogni coppia, le metriche ICP (Information-flow-based Coupling) e CCBC (Conceptual Coupling Between Classes). La prima misura l'accoppiamento strutturale contando la quantità di parametri passati e le chiamate a funzione effettuate. La seconda, invece, calcola l'accoppiamento tra due componenti basandosi su informazioni semantiche acquisite da identificatori e commenti nel codice sorgente. Avendo come riferimento certe due soglie legate alle metriche precedentemente esposte, si decide come estrarre catene di classi fortemente connesse. Se necessario è possibile ricorrere all'uso del Extract Superclass per raggruppare prima le classi strutturabili in una gerarchia.

3. Vengono creati tanti nuovi package quante sono le catene. Questi presenteranno una coesione maggiore rispetto a quella del package identificato come smelly. Le classi vengono quindi predisposte per essere distribuite tra i package appena realizzati basandosi sulle catene estratte.

4. Viene eseguito Move Class per ogni file che deve essere spostato. Il tutto avviene senza alterare la struttura interna delle classi. Spesso è efficace spostare gruppi di classi contemporaneamente. Compilare il codice nel package padre e ripetere il test ad ogni ripetizione.

Tale approccio è stato poi implementato nel plug-in per Eclipse AIRES [52].

CAPITOLO 3

Plug-in ASCETIC

3.1 Descrizione Plugin

ASCETIC (Automated Smell CodE idenTIfication and Correction), nato da un'opera di reengineering come progetto universitario, è un plug-in che permette il rilevamento di 4 tipi di code smell di diversa natura e a diversi livelli di granularità (i.e, Feature Envy, Blob, Misplaced Class e Promiscuous Package) e, se richiesto, il plug-in è inoltre in grado di effettuare in automatico le operazioni di refactoring sfruttando le API di IntelliJ IDEA per la manipolazione del codice sorgente.
In questo studio di tesi ci si è concentrati sul raffinamento delle tecniche di analisi, ampliando e migliorando le funzionalità già offerte dalla precedente versione del plug-in. Ulteriore obiettivo di questo lavoro è stato raffinare le tecniche di parsing e di refactoring per garantire una migliore estrazione dei dati e una correzione automatica degli smell più affidabile, al fine di preservare la funzionalità del codice. ASCETIC si rivolge ad un pubblico composto principalmente da sviluppatori Java che utilizzano l'IDE IntelliJ IDEA e si pone l'obiettivo di facilitare la correzione del codice sorgente. Lo sviluppatore ha la possibilità di:

- Gestire singolarmente le soglie usate dagli algoritmi di analisi di ogni smell, permettendone la personalizzazione tramite apposita finestra;
- Analizzare il codice sorgente per rilevare la presenza di 4 code smell;

- Applicare la correzione automatica dei code smell, rilevati durante la fase d'analisi, richiedendo al sistema una soluzione. E' possibile applicare nel caso del Blob e Promiscuous Package una soluzione personalizzata;
- Avviare il plug-in automaticamente al momento del commit.

Di seguito sono riportati i requisiti funzionali e non funzionali derivanti dalla precedente implementazione di ASCETIC. Sono stati aggiunti in questo studio di tesi i requisiti RF 1.2 e 1.3, RF 6, RF 7 e RF 8. Tra questi i più rilevanti sono i primi due, poiché legati all'implementazione delle tecniche strutturali di analisi e alla combinazione di queste con le altre già presenti. I requisiti non funzionali, invece, non sono stati modificati.

Requisiti Funzionali

- **RF 1 - Identificazione code smell**

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

  RF 1.1 - Identificazione testuale Identificazione di Code smell tramite tecniche di analisi testuale: il plug-in consente la identificazione di quattro tipi di code smell (i.e., Feature Envy, Misplaced Class, Blob e Promiscuous Package) attraverso l'utilizzo di tecniche basate sulla somiglianza testuale.

  RF 1.2 - Identificazione strutturale (nuovo) Identificazione di Code smell tramite tecniche di analisi strutturale: il plug-in consente l'identificazione di tre tipi di code smell (i.e., Feature Envy, Misplaced Class, Blob e Promiscuous Package) attraverso l'utilizzo di tecniche basate sulla struttura interna del codice sorgente.

  RF 1.3 - Tecnica combinata (nuovo) Il sistema permette di combinare i risultati ottenuti dall'identificazione testuale e strutturale per una maggiore accuratezza.

- **RF 2 - Proposta di correzione**

  Il sistema consente la creazione di una possibile soluzione ai quattro principali tipi di code smell (i.e., Feature Envy, Misplaced Class, Blob e Promiscuous Package).

RF 2.1: Il sistema permette di modificare la soluzione proposta al Blob, attraverso lo spostamento dei metodi tre le varie classi, e al Promiscuous Package tramite lo spostamento delle classi tra i vari package.

- **RF 3 - Refactoring**

  Il sistema consente di applicare automaticamente la soluzione approvata per risolvere i quattro tipi di code smell (i.e., Feature Envy, Misplaced Class, Blob e Promiscuous Package).

- **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 legati al codice implementato.

- **RF 6 - Priorità di correzione (nuovo)**

  Il sistema consente di valutare il grado di severità degli smell identificati, consigliando la priorità con cui effettuare la correzione.

- **RF 7 - Avvio al commit (nuovo)**

  Il sistema consente di avviare l'analisi del progetto al momento del commit su GitHub, interrompendolo se richiesto dall'utente.

- **RF 8 - Personalizzazione delle soglie (nuovo)**

  Il sistema consente di personalizzare le soglie usate dagli algoritmi di analisi.

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

- **RNF 4 - Manutenibilità**
  Il sistema deve presentare un elevato grado di manutenibilità, in particolare, favorisce l'aggiunta di nuove funzionalità.

- **RNF 5 - Implementazione**
  Il sistema è realizzato interamente in linguaggio Java, sia parte back-end che front-end.

- **RNF 6 - Affidabilità**
  Il sistema assicura un'alta affidabilità, riducendo al minimo i casi di arresto anomalo del sistema.

Nelle sottosezioni seguiti verranno elencati i criteri di qualità ricavati dai requisiti non funzionali.

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.

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.

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.

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.

3.2 Dettagli Tecnici

ASCETIC è un plug-in evoluto nel tempo e nato da un progetto universitario, ne conserva perciò l'architettura indirizzata all'uso ottimale dei Design Pattern. Le funzionalità del sistema sono suddivise in componenti e layer logici in base alle differenti caratteristiche di queste, come riportato in Figura 3.1. In seguito a questo studio di tesi le componenti già presenti sono state ampliate, sfruttando il disaccoppiamento fornito dai pattern, per andare incontro ai nuovi requisiti funzionali.

Le aggiunte effettuate riguardano le componenti:

- **Action** : inserimento dell'avvio del plug-in al momento del commit;
- **GUI** : aggiunta dell'interfaccia per la personalizzazione delle soglie e riadattamento di quelle già presenti;
- **Analisys** : implementazione delle tecniche di analisi strutturale per i quattro tipi di smell analizzati.

Le componenti Code Parser e Refactoring, invece, sono state raffinate correggendo casi non bene gestiti precedentemente legati rispettivamente all'estrazione dei dati e all'esecuzione del refactoring del Feature Envy e del Promiscuous Package.

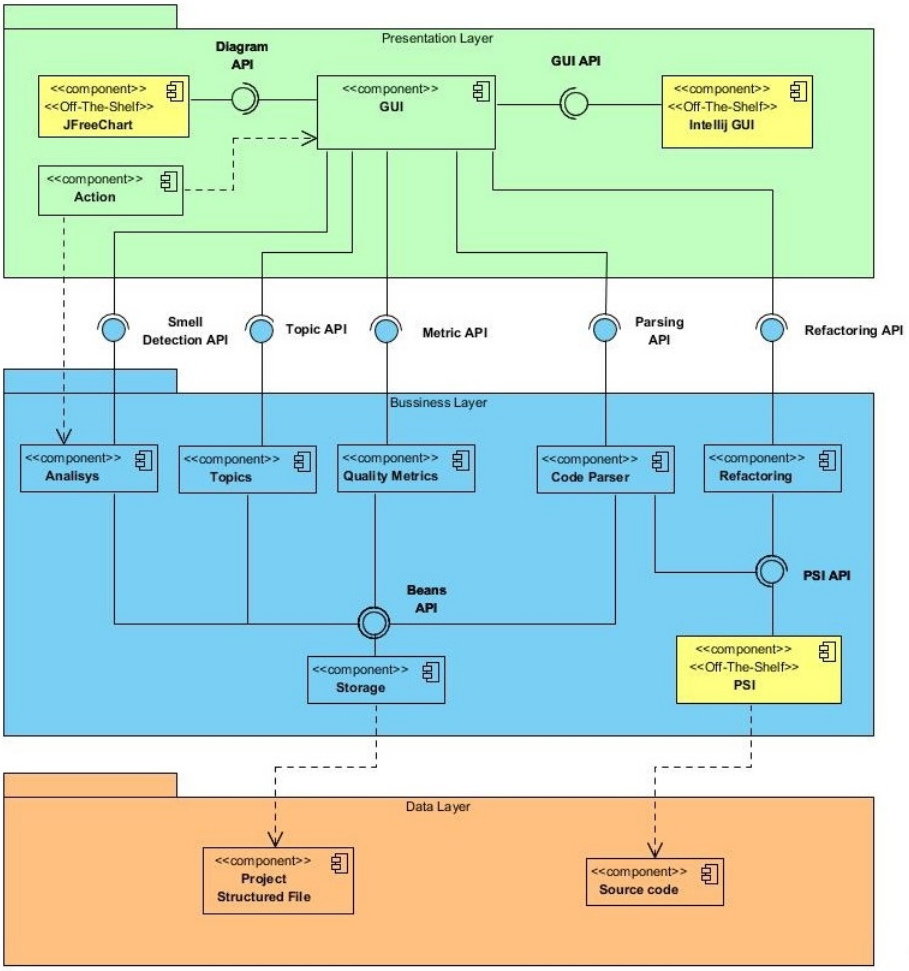

Figura 3.1: Component Diagram

La massima coesione e il minimo accoppiamento tra i sottosistemi sono garantite in modo che i cambiamenti in un sottosistema non influiscano sugli altri. Il sistema usa il modello

architetturale MVC: impianto di memorizzazione (Model), presentazione (View) e business logic (Controller).

- Il presentation layer include le componenti GUI e Action che, opportunamente configurate, eseguono codice in risposta a interazioni dell'utente con componenti grafiche estese dell'IDE.
  - **Action:** Componente che funge da connettori tra IntelliJ e il codice del plug-in. Nel caso specifico permettono di avviare il plug-in quando l'utente lo richiede cliccando nel menù dei "Tool" o intercettando l'invio di un commit su GitHub.
  - **GUI:** Componente che realizza l'interfaccia grafica, la quale si appoggia su IntelliJ API(per componenti grafiche) e JFreeChart(per grafici).
- L' application layer contiene tutto il codice di business del plug-in ed è composto dalle seguenti componenti:
  - **Analisi:** Componente che si occupa di analizzare il codice java identificando, tramite metriche e topic ottenute da altre componenti di seguito riportate, 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 che gestisce l'accesso al file strutturato del progetto in analisi
  - **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 mostrate nelle varie gui(tali metriche saranno approfondite di seguite nella sezione 3.2.1).
  - **Topics:** Componente che si occupa del calcolo dei topic ovvero dei termini più ricorrenti all'interno del codice analizzato.
- Lo storage layer, invece, è costituito dai files contenente il codice sorgente in esame e dalla componente **Project Structured File**, 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). Il source code non viene mai acceduto direttamente dal codice di business del plug-in, ma attraverso le API offerte dalla piattaforma IntelliJ.

Scenario d'uso

In questa sezione viene descritta la successione di eventi generata durante l'uso del plug-in. 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.

1. L'utente vuole effettuare l'identificazioni di eventuali code smell presenti nel progetto Java in analisi.

2. Nel caso lo voglia, lo sviluppatore può visionare le soglie utilizzate dagli algoritmi di analisi tramite la schermata riassuntiva. Nel menù dei tool è presente una voce legata al plug-in tramite cui aprire la suddetta schermata. L'utente può anche procedere alla modifica delle soglie, sciegliendo se ricercare tutti i tipi di smell trattati e se applicare una o entrambe le tecniche proposte.

3. L'utente per avviare il plug-in si reca nella sezione dedicata, nel menù dei tool di IntelliJ, e cliccando l'apposito bottone richiede l'inizio dell'analisi.

4. Dopo la selezione della voce e una breve elaborazione, viene visualizzata a schermo una finestra riportante l'elenco degli smell identificati e varie informazioni a loro collegati come nome della componente, la sua priorià di correzione e il contenuto testuale. Anche in questa schermata è presente una il piccola area dedicata alle soglie, in cui è possibile modificarle in modo da filtrare meglio la lista di elementi visibili, risultati affetti da smell.

5. L'utente, cliccando sull'apposito pulsante, prende visione nel dettaglio dei dati legati alla componente affetta, visionando varie tabelle informative e un numero variabile, in base al tipo di smell, di RadarMaps che mostrano i 5 termini più presenti nella componente di riferimento.

6. Lo sviluppatore, nel caso voglia correggere un elemento affetto può, tramite apposito bottone, richiedere la creazione automatica di una soluzione per rimuovere lo smell.

7. L'utente prende visione in una nuova schermata della proposta del sistema. Tramite le tabelle visibili in schermata è possibile vedere come la soluzione andrà a modificare la componente. Nel caso in cui l'elemento sia affetto da Blob o Promiscuous Package, l'utente può personalizzare la soluzione proposta sfruttando l'interfaccia e vedere quali effetti avrà la modifica se applicata.

8. L'utente, se convinto della soluzione ottenuta, cliccando sull'apposito bottone ne richiede l'applicazione.

9. Dopo una breve elaborazione una finestra di notifica avvisa l'utente dell'avvenuta correzione o di eventuali problemi riscontrati nel durante. In caso di successo le schermate del plug-in vengono chiuse.

3.2.1 Casi d'uso

Come visibile in Figura 3.2, tramite la sezione **ASCETIC- Analyze Project**, è fornita all'utente la possibilità di avviare il plug-in tramite la voce "Run plug-in" o di aprire la console per la configurazione delle soglie tramite la voce "Configure Threshold".

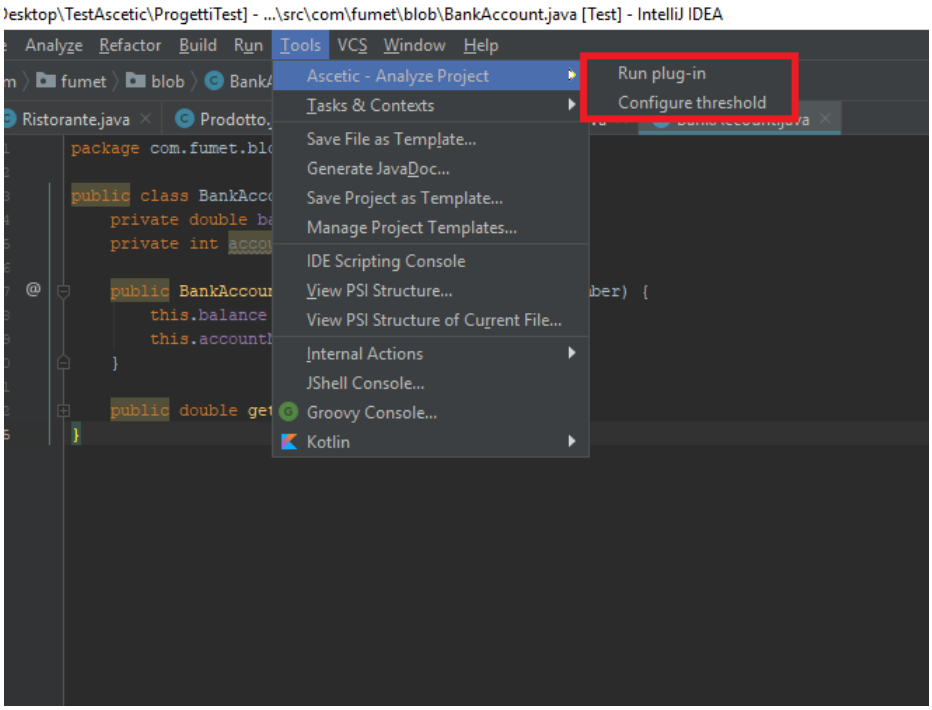

Figura 3.2: Sezione di avvio del plug-in

Prima di esporre però il funzionamento del plug-in è importante soffermarsi sulle tecniche di detection implementate in esso. Gli approcci di analisi strutturale presenti in ASCETIC sono riportati dalla Tabella 3.3 alla Tabella 3.6.

Feature Envy
Per rilevare lo smell di tipo Feature Envy viene usata come metrica di riferimento la dipendenza e l'analisi si suddivide nelle seguenti fasi: 1. viene calcolata la dipendenza interna della classe che presenta il metodo in analisi; 2. si procede successivamente al calcolo delle dipendenze rispetto alle altre classi del package, identificando la classe con dipendenza maggiore come "envied" (invidiata come più adatta a contenere il metodo); 3. se la classe "envied" non corrisponde a quella originale che presenta il metodo, allora la componente può essere considerata smell.

Tabella 3.3: Detection strutturale implementata per Feature Envy

Misplaced Class
Per quanto riguarda l'iter di ricerca dei Misplaced Class, questo risulta molto simile a quello adottato per i Feature Envy con l'unica differenza di far riferimento alla dipendenza tra classe in analisi e il package di appartenenza, ricercando eventuali package "envied" più idonei a contenere la componente sotto analisi.

Tabella 3.4: Detection strutturale implementata per Misplaced Class

Promiscuous Package
Per l'identificazione del Promiscuous Package sono state usate le metriche: • **MIntraC** (MediumIntraComplexity) maggiore di 0.5; calcola le dipendenze interne delle classi, appartenenti al package in analisi, rispetto alle altre classi nello stesso; • **MInterC** (MediumInterComplexity) maggiore di 0.5; calcola le dipendenze esterne delle classi, appartenenti al package in analisi, rispetto alle altre classi presenti negli altri package del progetto.

Tabella 3.5: Detection strutturale implementata per Promiscuous Package

Blob
Il Blob è stato rilevato secondo criteri più complessi ottenuto tramite la combinazione delle seguenti metriche per identificare se la classe in analisi fosse una classe di controllo o una grande classe con

bassa coesione:

- **LCOM5** (Lack of Cohesion Of Methods [14] maggiore di 20;
- **ELOC** (Effective Lines of Code) superiore a 500 linee di codice effettive (si tiene conto solo di righe che non sono commenti, spazi vuoti o parentesi graffe e tonde);
- **featureSum** maggiori di 20; tale valore è dato dalla somma delle metriche WMC (Weighted Methods for Class) e NOA (Number Of Attributes) che rispettivamente indicano la complessità della classe (espressa come la somma dei metodi che presenta) e il numero di attributi posseduti dalla classe;
- un nome che contiene un suffisso nel set {Process, Control, Command, Manage, Drive, System} e ha un'associazione uno-a-molti con le classi di dati.

Tabella 3.6: Detection strutturale implementata per Blob

Le metriche usate, relative alle varie tecniche adottate, saranno riprese ed approfondite di seguito in questa sezione. Inoltre, sono state aggiunte delle euristiche finalizzate all'identificazione di classi "Bean", evitando così falsi positivi da parte degli algoritmi di analisi. Tali classi sono utilizzate per incapsulare più oggetti in un oggetto singolo (il bean), cosicché tali oggetti possano essere passati come un singolo oggetto bean invece che come multipli oggetti individuali. Un bean è definito come tale se presenta determinati metodi con particolari caratteristiche quali: costruttore (senza o con parametri, senza ritorno di valore, ha nome uguale alla classe di appartenenza), getter (senza parametri, valore di ritorno uguale a quello richiesto), setter (con un solo parametro, ritorna void), toString (senza parametri, oggetto "String" di ritorno), hasCode (senza parametri, ritorna oggetto "int") e equals (con parametro un oggetto generico, ritorna valore booleano).

Le tecniche di analisi testuale implementate in ASCETIC sono riportate, invece, dalla Tabella 3.7 alla Tabella 3.10.

Misplaced Class
Nel caso del Misplaced Class, l'analisi testuale si basa sul fatto che una classe affetta da questo smell è semanticamente più correlata ad un package differente piuttosto che a quello in cui si trova [2]. Il loro approccio si pone come obbiettivo quello di individuare il package che contiene la massima similarità testuale con la classe in esame. Nel caso in cui questo corrisponda al package di appartenenza della classe, allora viene identificato lo smell. In caso contrario, la componente è identificata come smelly. Più formalmente, viene calcolata una probabilità che la classe in esame sia affetta da Misplaced Class: $$P_{MC}(C) = sim(C,P_{closest}) - sim(C,P_O)$$

Dove P_{MC}(C) indica la probabilità che la classe C sia affetta da Misplaced Class, $P_{closest}$ indica il package testualmente più vicino alla classe C, P_O indica invece il package di appartenenza di C, e infine sim(X,Y) indica la somiglianza testuale tra la classe X e il package Y. La probabilità, compresa nell'intervallo [0;1], che la componente sia affetta dallo smell è data dalla differenza tra le somiglianze di C con $P_{closest}$ e P_O. Dunque se $P_{closest}$ è diverso da P_O e la probabilità supera la soglia designata allora la classe soffre dello smell.

Tabella 3.7: Detection testuale implementate per Misplaced Class

Feature Envy
Per l'identificazione del Feature Envy può essere sfruttato un ragionamento simile a quello fatto per il Misplaced Class, a diversa granularità, poiché si lavora sul metodo delle classi in analisi identificandone la somiglianza testuale rispetto alle altre classi nello stesso package.

Tabella 3.8: Detection testuale implementate per Feature Envy

Blob
Per quanto riguarda l'identificazione di Blob, ci si basa sull'ipotesi che questi siano caratterizzati da un'alta dispersione semantica. Formalmente, sia C la classe da analizzare, M = {M_1;...;M_n} l'insieme dei metodi in C, la coesione testuale della classe viene calcolata come segue [28]: $$ClassCohesion(C) = \underset{i \neq j}{mean}\ sim(M_i, M_j)$$ dove *n* è il numero di metodi in C e *sim(M_1,M_j)* denota la somiglianza testuale tra i metodi M_1 ed M_j in C. Dunque la probabilità che C sia un Blob viene calcolata nel seguente modo: $$P_B(C) = 1 - ClassCohesion(C)$$ Anche in questo caso il valore di P_B(C) è compreso nell'intervallo [0;1]. Se la probabilità così ottenuta supera la soglia designata allora la componente risulta affetta dalle smell.

Tabella 3.9: Detection testuale implementate per Blob

Promiscuous Package
Per l'identificazione del Promiscuous Package può essere sfruttato un ragionamento simile a quello fatto per il Blob, a diversa granularità, poiché si lavora con il package in analisi e calcolandone la coesione testuale sulla base delle classi contenute, basandosi sull'ipotesi che il package sia caratterizzato da un'alta dispersione semantica.

Tabella 3.10: Detection testuale implementate per Promiscuous Package

Configure Threshold

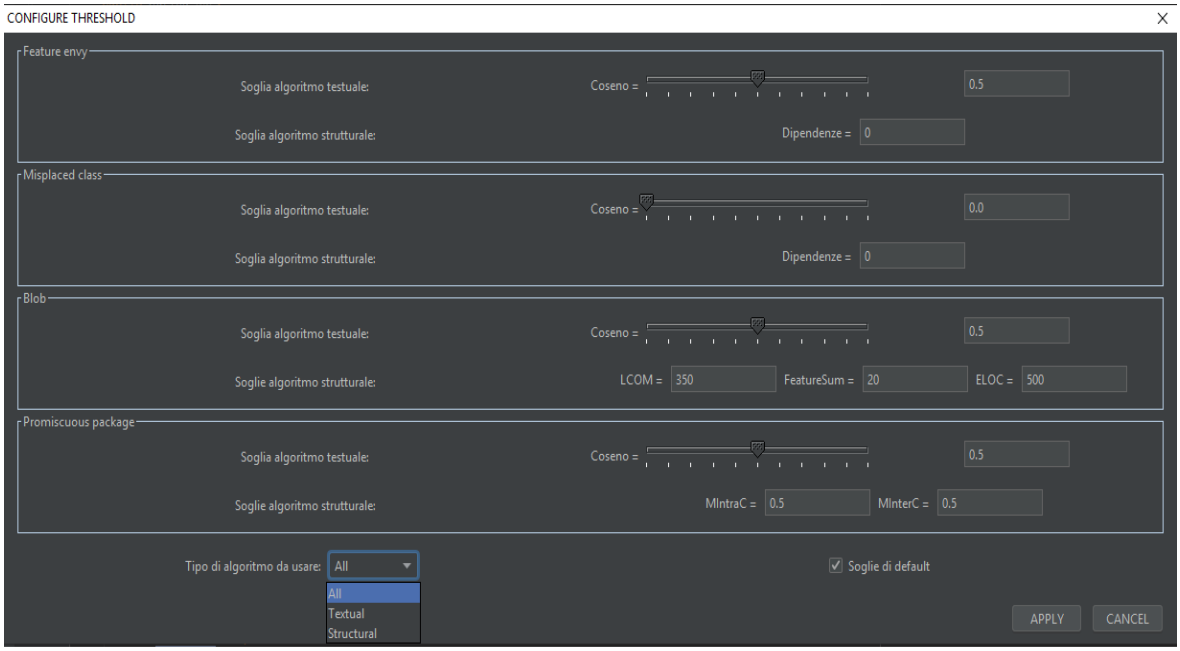

Figura 3.3: Schermata di configurazione delle soglie

ID	**UC 1 - Configurazione soglie**
Participating actors	Sviluppatore
Entry Condition	L'utente avvia con successo IntelliJ.
Flow of events	**Utente** L'utente selezionando la seconda voce del menù "ASCETIC - Analyze Project", visibile in Figura 3.2, richiede la possibilità di visionare ed eventualmente personalizzare le soglie da usare per l'analisi. **Sistema** Il sistema preleva le soglie precedentemente memorizzate mostrando poi una finestra di riepilogo con le varie soglie attualmente in uso, in assenza di precedenti vengono mostrate le soglie di default elencate in Tabella 3.12. **Utente** L'utente visiona la schermata riportata in Figura 3.3, che presenta quattro sezioni orizzontali, ognuna dedicata ad uno smell differente. Per ogni voce sono presenti le metriche, con relative soglie, usate per la rilevazioni testuale o strutturale di tali smell. Vi è la possibilità di personalizzare a piacimento le varie soglie, scegliendo anche se applicare una o entrambe le tecniche tramite un pratico menù, posto nella parte inferiore della schermata. Nel caso si voglia applicare solo una tecnica, selezionando la voce corrispondente, i

	campi legati all'altro metodo di analisi diventeranno invisibili per una migliore comprensione. La modifica del valore del coseno può essere effettuata tramite lo slider o inserendo direttamente il valore nel campo editabile posto alla sua destra, è importante ricordare che tale soglia appartiene all'intervallo [0;1]. Nel caso si voglia modificare un valore e questo esca dal range consentito (nessuna soglia può assumere valore negativo) esso verrà automaticamente settato al limite più vicino. Sempre nella sezione in basso è presente un bottone che permette di resettare tutti i campi con i valori di default. Se esso viene cliccato per errore o se richiesto dallo sviluppatore, ripremendo la check-box, è possibile ripristinare i valori dei campi allo stato precedente. In caso l'utente voglia salvare le soglie modificate è necessario selezionare "APPLY". **Sistema** Il sistema memorizza i dati in un file in locale per poter essere usati alla successiva analisi. **Utente** L'utente viene avvisato tramite un pop-up sull'esito del processo e le schermate vengono chiuse.
Exit condition	La finestra di configurazione si chiude.
Exception condition	
Quality requirements	

I valori di default delle metriche variano in base al tipo di smell, come visibile nella Tabella 3.12.

Code Smell	Metriche testuali	Metriche strutturali
Feature Envy	Coseno = 0.5	Dipendenza = 0;
Misplaced Class	Coseno = 0.0	Dipendenza = 0;
Blob	Coseno = 0.5	LCOM = 350, FeatureSum = 20, ELOC = 500
Promiscuous Package	Coseno = 0.5	MIntraC = 0.5, MInterC = 0.5

Tabella 3.12: Valori di default delle metriche

Run plug-in

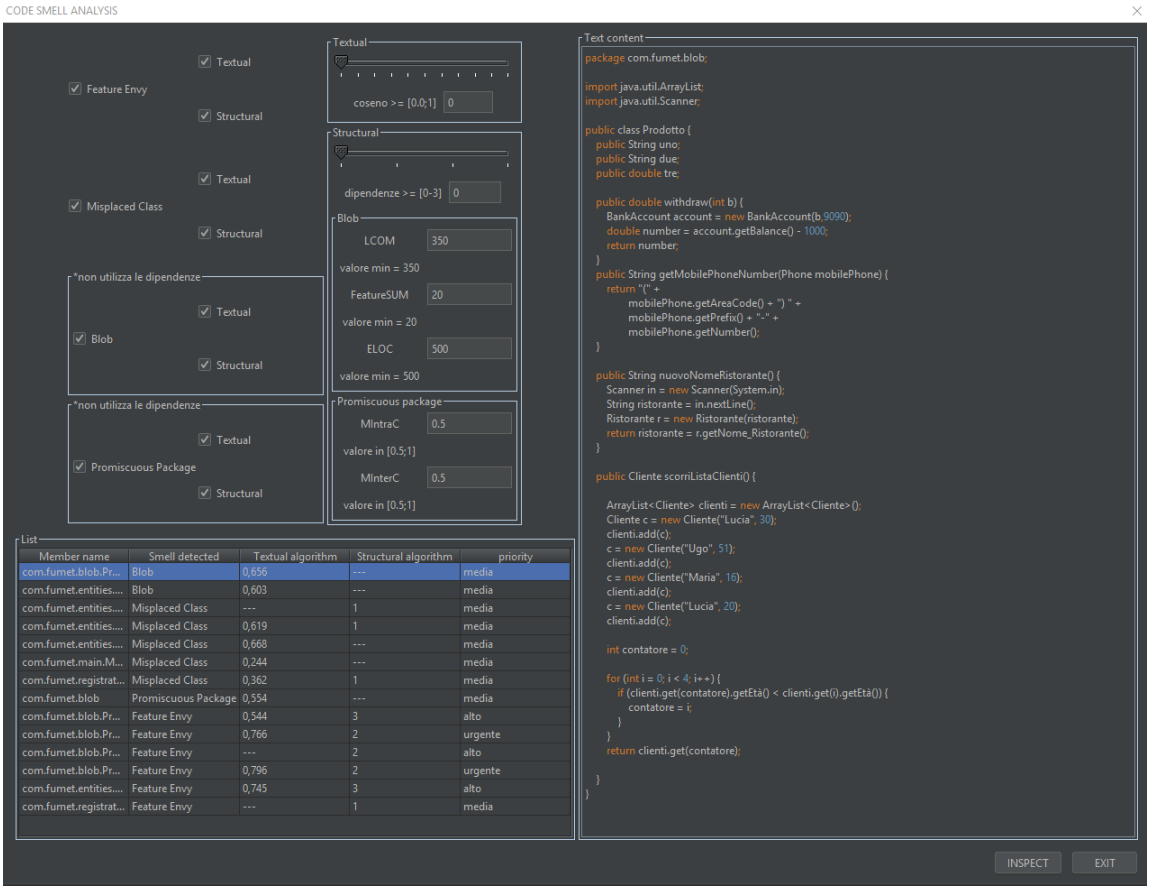

Figura 3.4: Schermata inziale plug-in

ID	**UC 2 - Analisi progetto**
Participating actors	Sviluppatore
Entry Condition	L'utente avvia con successo IntelliJ.
Flow of events	**Utente** L'utente selezionando la prima voce del menù "ASCETIC - Analyze Project", visibile in Figura 3.2, richiede l'analisi del progetto designato. **Sistema** Il sistema preleva le soglie memorizzate e, tramite i parametri specificati dall'utente, crea all'avvio del plug-in un thread di controllo che inizia la sua attività di analisi dell'intero progetto. Nel caso in cui non sia stato effettuato ancora il primo avvio, viene creata la cache. Dopo l'elaborazione, viene mostrata una finestra di riepilogo all'utente. **Utente** L'utente visiona la schermata riportata in Figura 3.4, che presenta due gran-

	di aree. La parte a sinistra presenta delle check-box, ognuna per uno smell e una tecnica di analisi ad esso collegato. Sono presenti, inoltre, dei campi editabili dove è possibile variare le soglie delle metriche usate dagli algoritmi per modificare il contenuto della tabella sottostante, rendendo visibili solo gli elementi compatibili ai valori inseriti. Tali valori sono inizialmente settati con quelli definiti nella schermata di "Configure Threshold". In questa finestra, però, i campi dedicati alla soglia del coseno e a quella della dipendenza sono comuni a tutti gli smell. Pertanto, il loro valore è il minore assunto dalle corrispondenti voci nelle tecniche legate agli smell. Non è quindi possibile modificare le due soglie in modo mirato per un certo smell. Vicino ad ogni nome di smell e tecnica fornita è presente una check-box che permette di eliminare visivamente in tabella ogni riferimento all'elemento deselezionato. La tabella fornisce una serie di informazioni legate alle componenti ritenute smelly dal sistema fornendone: nome, tipologia di smell di cui affetto, valori ottenuti dagli algoritmi di analisi e urgenza di correzione in base alla pericolosità. Nella sezione a destra vi è invece, un'area di testo che mostra il contenuto testuale della componente selezionata in tabella. Tutti i contenuti testuali visibili nelle varie aree di testo del plug-in sono editati in modo da evidenziare, con font color tipici dell'IDE, le key word di java e i valori numerici per una migliore comprensione. Nel caso l'utente volesse proseguire con la correzione di uno smell è necessario selezionare prima l'elemento interessato nella tabella e poi cliccare il bottone "INSPECT". Tale successione di passi può essere effettuata per uno smell alla volta, non sono pertanto accettate selezioni multiple di voci in tabella. In caso contrario l'ispezione viene applicata al primo elemento indicato. **Sistema** Il sistema, in base alla componente selezionata dall'utente, effettua una elaborazione sul codice, mostrando poi una finestra di riepilogo all'utente. Le metriche calcolate variano in base al tipo di smell. **Utente** L'utente visualizza una schermata riassuntiva che riporta varie informazioni, differenti in base al tipo di smell, sulla componente selezionata utili per decidere se proseguire verso il refactoring o meno.
Exit condition	L'utente visualizza nel dettaglio l'elemento affetto da smell.
Exception condition	

Quality requirements	Tempi massimi di elaborazione nell'ordine dei 3 minuti, nel caso di progetti di grandi dimensioni.

Il grado di priorità identificato per ogni componente smelly è calcolato sulla base dei risultati delle tecniche, come specificato in Tabella 3.14.

Priorità	**Motivazione**
bassa	Componente ritenuta smelly da una sola tecnica, con riscontro **vicino** alla soglia
media	Componente ritenuta smelly da entrambe le tecniche con riscontro **vicino** alle soglie o da una sola tecnica con riscontro **né** vicino **né** troppo elevato rispetto ad esse
alta	Componente ritenuta smelly da entrambe le tecniche, con riscontro **né** vicino alle soglie **né** troppo elevato
urgente	Componente ritenuta smelly da entrambe le tecniche implementate, con **almeno un riscontro molto alto** e perciò ritenuto critico

Tabella 3.14: Priorità di correzione

La schermata mostrata dopo la selezione del bottone "INSPECT", nel caso di Feature Envy o Misplaced Class risulta simile ed è rappresentata in Figura 3.5, mentre nel caso di Blob o Promiscuous Package la schermata, anch'essa simile, è mostrata in Figura 3.8. La successione di operazioni da effettuare per giungere al refactoring è affine nelle coppie di smell sopra citate. Esse sono esposte nel dettaglio nelle omonime sezioni successive.

Feature Envy e Misplaced Class

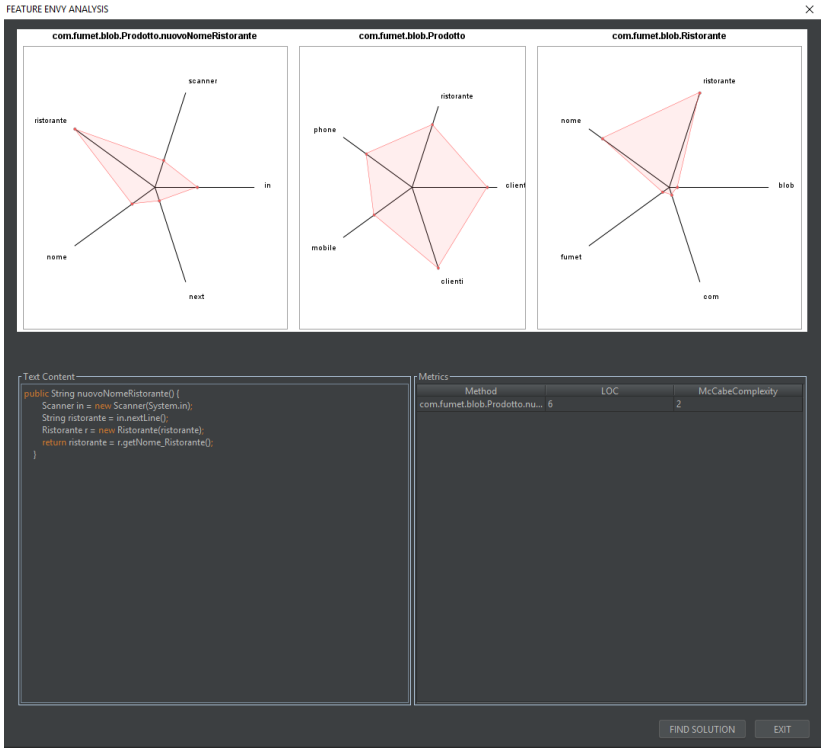

Figura 3.5: Ispezione per Feature Envy e Misplaced Class - Esempio Feature Envy

Come già detto le metriche calcolate variano in base al tipo di smell analizzato e nel caso del Misplaced Class, esse sono riportate in Tabella 3.15.

Nome metrica	**Descrizione**
LOC (Line Of Code)	numero di linee di codice della classe
WMC (Weighted Methods per Class)	grado di complessità del codice legato al numero di percorsi/decisioni linearmente indipendenti tra loro
LCOM (Effective Line Of Code)	effettive linee di codice tenendo conto solo di righe che non sono commenti, spazi vuoti o parentesi
CBO (Coupling Between Objects)	conteggio del numero di classi che sono accoppiate a una classe particolare, ovvero dove i metodi di una classe chiamano i metodi o accedono alle variabili dell'altra
NOM (Number Of Methods)	stima la quantità di metodi all'interno della classe
NOPA (Number Of Public Attributes)	conta il numero di attributi pubblici di una classe

Tabella 3.15: Metriche di qualità per Misplaced Class

Se l'elemento risulta affetto da Feature Envye, le metriche mostrate sono le seguenti.

Nome metrica	**Descrizione**
LOC (Line Of Code)	numero di linee di codice
McCabe-Complexity	grado di complessità del codice legato al numero di percorsi/decisioni linearmente indipendenti tra loro

Tabella 3.16: Metriche di qualità per Feature Envy

ID	**UC 3.1 - Refactoring componente affetta da Feature Envy o Misplaced Class**
Participating actors	Sviluppatore
Entry Condition	L'utente visualizza nel dettaglio l'elemento affetto da smell.
Flow of events	**Utente** L'utente visualizza la schermata in Figura 3.5, che mostra tre RadarMap rappresentanti i 5 termini più noti, legate rispettivamente al metodo/classe affetto da smell, classe/package che contiene la componente smelly e la classe/package più affine alla componente affetta. Nella porzione inferiore è presente il contenuto testuale della componente smelly e una tabella che ne riporta il nome e le metriche di qualità calcolate. Nel caso l'utente voglia richiedere una proposta di soluzione, è necessario cliccare sul bottone "FIND SOLUTION". **Sistema** Il sistema, dopo una breve elaborazione, propone una possibile soluzione, calcolandone anche le metriche e topic per una maggiore comprensione. **Utente** L'utente, tramite la schermata in Figura 3.6, prende visione della soluzone proposta dal sistema. Le RadarMaps presenti in alto rappresentano i topic legati alla vecchia componente corrente (che contiene l'elemento smelly), alla vecchia componente invidiata (componente più affine all'elemento affetto) e alla loro versione dopo aver applicato il refactoring per mostrare i miglioramenti apportati. Nella porzione inferiore sono presenti due aree che mostrano il contenuto testuale della componente prima della correzione e il testo della componente invidiata dopo il refactoring. Nel caso si voglia procedere applicando la soluzione proposta è necessario cliccare sul bottone "REFACTORING". **Sistema** Il sistema effettua il refactoring applicando la soluzione approvata.

	Utente L'utente, alla fine dell'operazione, viene avvisato tramite un pop-up sull'esito del processo, come mostrato in Figura 3.7.
Exit condition	La finestra del plug-in si chiude.
Exception condition	
Quality requirements	Tempi massimi di elaborazione nell'ordine dei secondi.

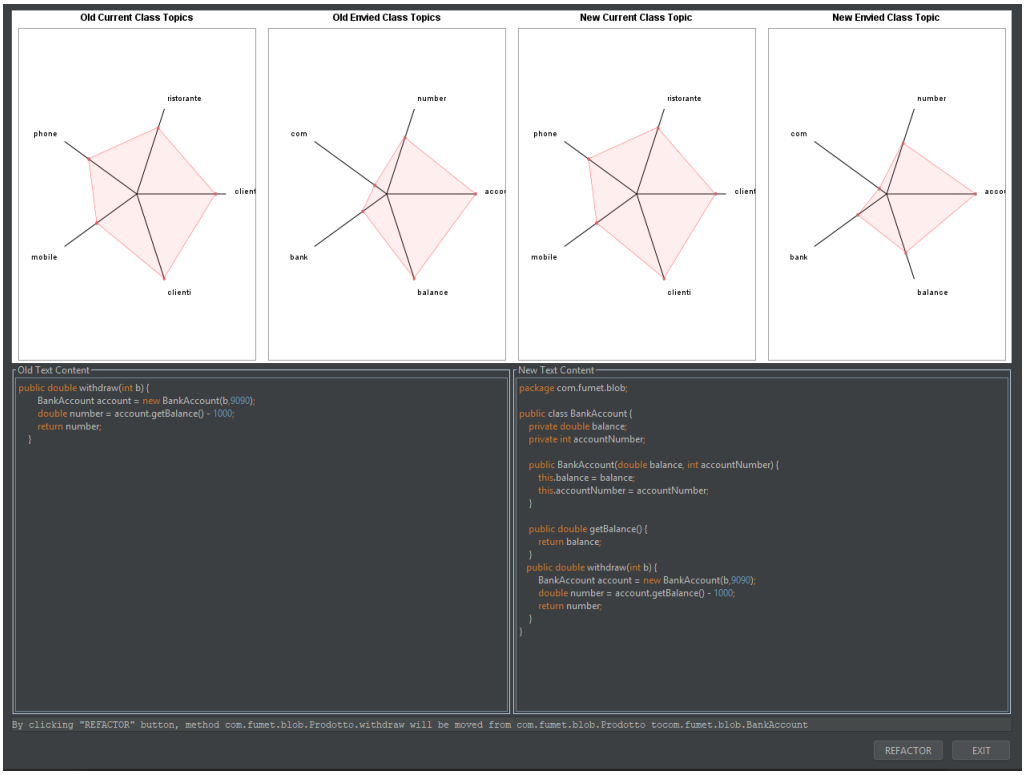

Figura 3.6: Proposta di soluzione per Feature Envy e Misplaced Class - Esempio Feature Envy

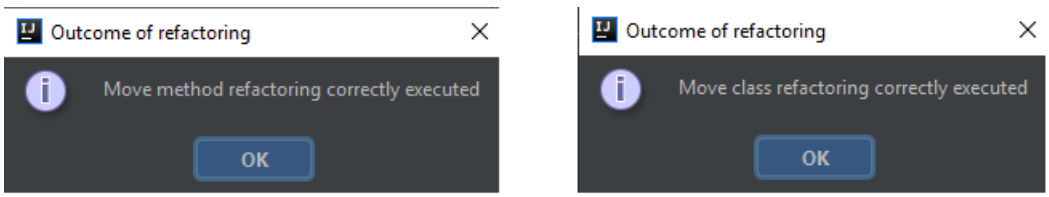

Figura 3.7: Successo del refactoring per Feature Envy e Misplaced Class

Blob e Promiscuous Package

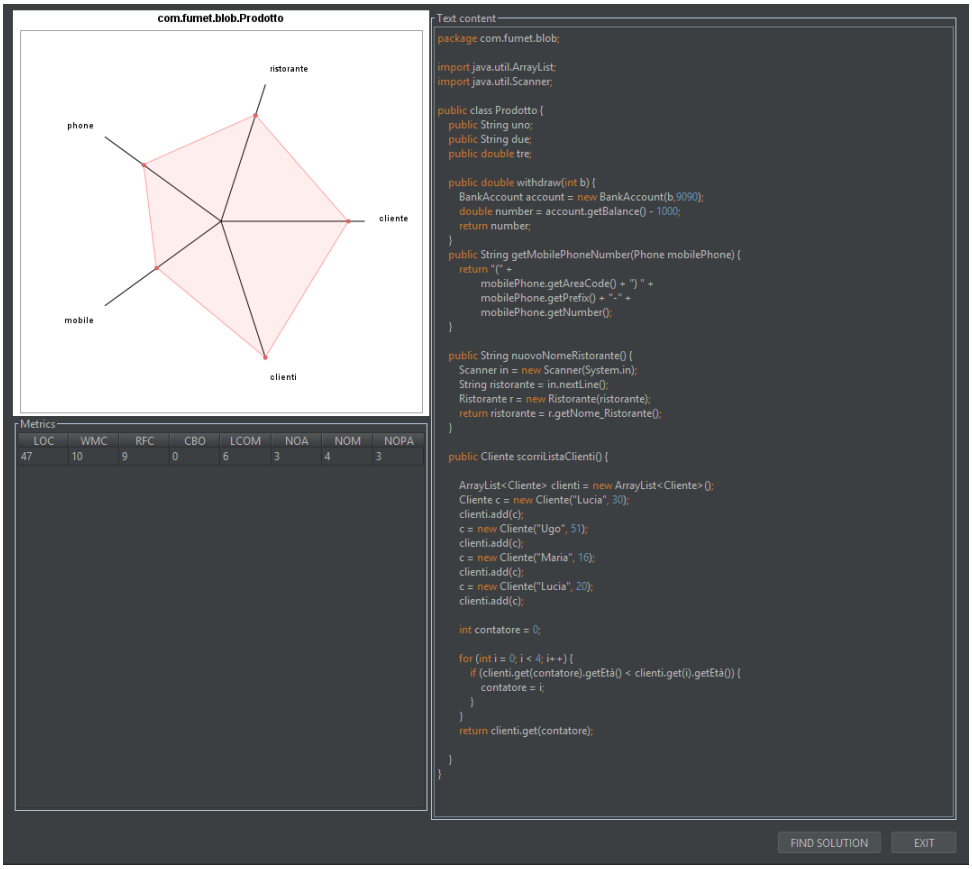

Figura 3.8: Ispezione per Blob e Promiscuous Package - Esempio Blob

La procedura di correzione di smell di tipo Blob o Promiscuous Package risulta differente data la maggiore complessità nella definizione automatica di una soluzione. Le metriche calcolate, relative al package smelly nel caso del Promiscuous Package, sono elencate nella Tabella 3.18.

Nome metrica	**Descrizione**
NOC (Number Of Classes)	tiene conto del numero di classi presenti nel package
MIntraC (Medium Intra Connectivity)	calcola il rapporto, in intervallo [0;1], tra il numero di dipendenze interne delle classi di un dato package e il numero delle classi
MInterC (Medium Inter Connectivity)	calcola il rapporto, in intervallo [0;1], tra il numero di dipendenze esterne delle classi di un dato package e quelle di un insieme di package

Tabella 3.18: Metriche di qualità per Promiscuous Package

Se viene ispezionato un Blob, invece, i valori visualizzati riguardanti alla classe in analisi sono mostrati nella Tabella 3.19.

Nome metrica	**Descrizione**
LOC (Line Of Code)	numero di linee di codice della classe
WMC (Weighted Methods per Class)	grado di complessità del codice legato al numero di percorsi/decisioni linearmente indipendenti tra loro
RFC (Response For Class)	complessità della classe in termini di chiamate al metodo. Viene calcolato aggiungendo il numero di metodi nella classe (esclusi i metodi ereditati) più il numero di chiamate di metodi distinte effettuate dai metodi nella classe (ogni chiamata di metodo viene conteggiata una sola volta anche se viene chiamata da metodi diversi)
CBO (Coupling Between Objects)	conteggio del numero di classi che sono accoppiate a una classe particolare, ovvero dove i metodi di una classe chiamano i metodi o accedono alle variabili dell'altra
LCOM (Effective Line Of Code)	effettive linee di codice tenendo conto solo di righe che non sono commenti, spazi vuoti o parentesi graffe o parentesi
NOA (Number Of Attributes)	numero di attributi usati dalla classe
NOM (Number Of Methods)	stima la quantità di metodi all'interno della classe
NOPA (Number Of Public Attributes)	conta il numero di attributi pubblici di una classe

Tabella 3.19: Metriche di qualità per Blob

ID	**UC 3.2 - Refactoring componente affetta da Blob e Promiscuous Package**
Participating actors	Sviluppatore
Entry Condition	L'utente visualizza nel dettaglio l'elemento affetto da smell.
Flow of events	**Utente** L'utente visualizza la schermata in Figura 3.8, che mostra la RadarMap rappresentante i 5 termini più noti, legata alla classe/package affetto da smell. Nella porzione a destra è presente il contenuto testuale della componente smelly mentre in basso vi è una tabella che ne riporta il nome e le metriche di qualità calcolate. Nel caso l'utente voglia richiedere una proposta di soluzione, è necessario cliccare sul bottone "FIND SOLUTION".

	Sistema Il sistema, dopo una breve elaborazione, propone una possibile soluzione, calcolandone anche le metriche e topic per una maggiore comprensione. **Utente** L'utente, tramite la schermata in Figura 3.9, prende visione della soluzone proposta dal sistema. Questa finestra di proposta di soluzione risulta molto più complessa rispetto all'altra vista precedentemente poiché oltre alla visione dei vari dati estrapolati dalle componenti, è possibile rielaborare la soluzione ottenuta. La tabella in alto a destra della schermata presenta il nome dell'elemento e le stesse metriche sopra citate (differenziate in base al tipo di smell) ma calcolate non solo sulla vecchia componente ma anche su tutte quelle ottenute dall'algoritmo come possibile soluzione. Le metriche in tabella fanno riferimento alla vecchia componente in analisi e a quelle nella soluzione proposta. La RadarMap in alto a sinistra mostra i topic dell'elemento affetto mentre quelle nella porzione centrale della finestra fanno riferimento alle componenti della soluzione proposta. L'elemento più importante in questa schermata risiede però nella parte inferiore dove sono presenti tutti gli elementi appartenenti alla soluzione proposta. Vi è quindi la possibilità di spostare, tramite appositi bottoni ("->" o "<-"), gli elementi a piacimento modificando così la correzione. Nel caso del Blob, ad esempio, sarà possibile spostare i metodi da un classe all'altra rispetto alla disposizione proposta. Stesso ragionamento può essere fatto per il Promiscuous Package con la differenza di lavorare con le classi dei package. Ad ogni modifica verranno ricalcolate le RadarMaps e i valori in tabella legati alle metriche. Nel caso si voglia visionare come la componente affetta è stata suddivisa, è possibile navigare tra i vari elementi tramite menù posti sopra le liste. Nel caso l'utente voglia procedere applicando la soluzione ottenuta è necessario cliccare sul bottone "REFACTORING". **Sistema** Il sistema applica le modifiche approvate dall'utente, effettuando il refactoring. **Utente** L'utente, alla fine dell'operazione, viene avvisato tramite un pop-up sull'esito del processo, come mostrato in Figura 3.10.
Exit condition	La finestra del plug-in si chiude.
Quality requirements	Tempi massimi di elaborazione nell'ordine dei secondi.

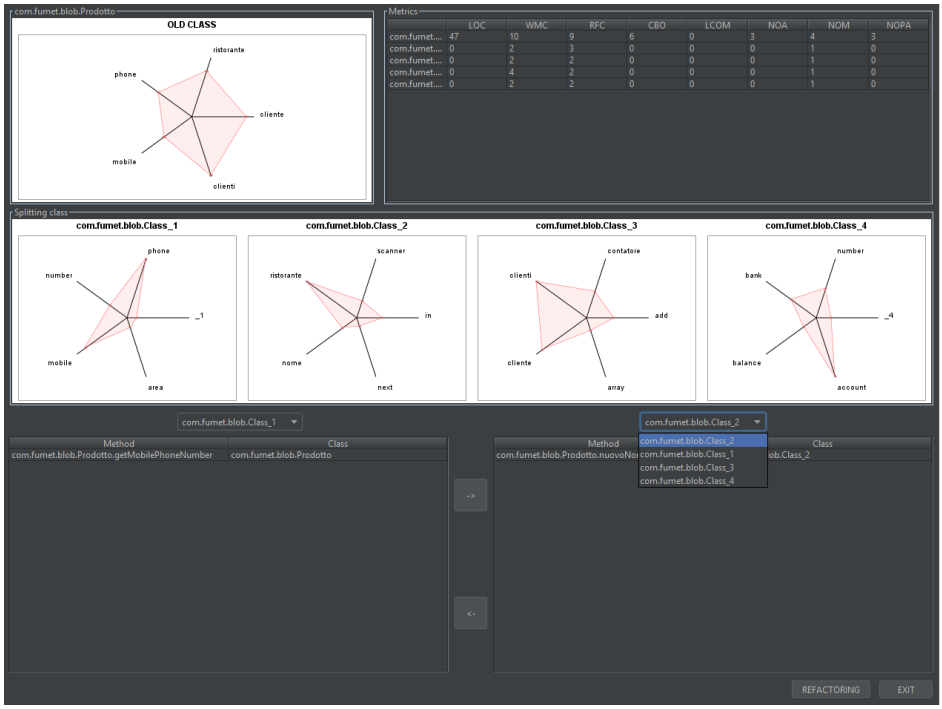

Figura 3.9: Proposta di soluzione per Blob e Promiscuous Package - Esempio Blob

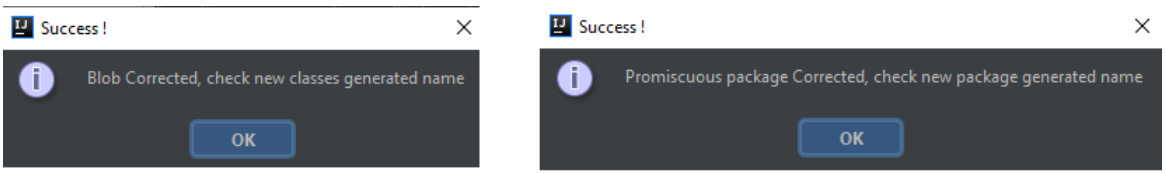

Figura 3.10: Successo del refactoring per Blob e Promiscuous Package

3.2.2 Hardware/software mapping

Il sistema è installato sull'ambiente di sviluppo IntelliJ IDEA e utilizza la libreria SQLite. Poichè il sistema è embedded il collegamento con gli altri sottosistemi è gestito dall'IDE nel quale il plug-in viene eseguito. Come mostrato in Figura 3.11 il sistema è stand-alone e per tanto tutte le componenti software si trovano sullo stesso nodo hardware. Da questo punto di vista quindi non sono state apportate modifiche rispetto alla versione di partenza.

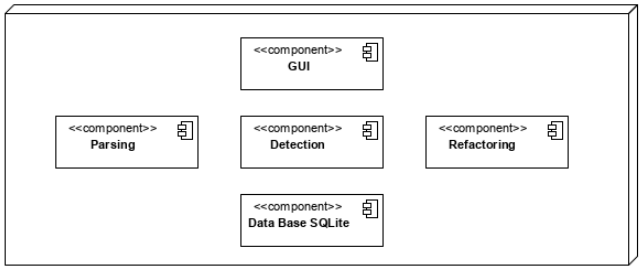

Figura 3.11: Deployment Diagram

3.2.3 Component Off-The-Shelf

Le componenti "off-the shelf" usate, e di seguito elencate, sono state scelte per garantire una maggiore performance e manutenibilità a discapito della memoria e sicurezza, favorendo future aggiunte e manutenzioni al codice. In questo studio di tesi non sono state apportate modifiche circa le componenti adottate, che quindi risultano le stesse. Data la natura locale del sistema non è necessario prevedere sistemi di protezione dall'esterno perché non vengono manipolati dati sensibili.

- **IntelliJ GUI** : Gli elementi della GUI si appoggiano su API di Sistema e API di IntelliJ.
- **PSI (Program Structure Interface)** :
  - **Code Manipulation API** : Il modulo di Refactor tramite queste API fornite da IntelliJ, effettua l'operazione di Extract e di 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). Tale componente è utilizzata per la realizzazione delle RadarMap che rappresenteranno le topic implementate dalle varie componenti analizzate.

3.2.4 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. Il diagramma riportato in Figura 3.12 costituisce uno schema logico rappresentativo del come i file verranno strutturati mediante SQLite. Ogni entità individuata è poi descritta di seguito dalla Tabella 3.21 alla Tabella 3.33.

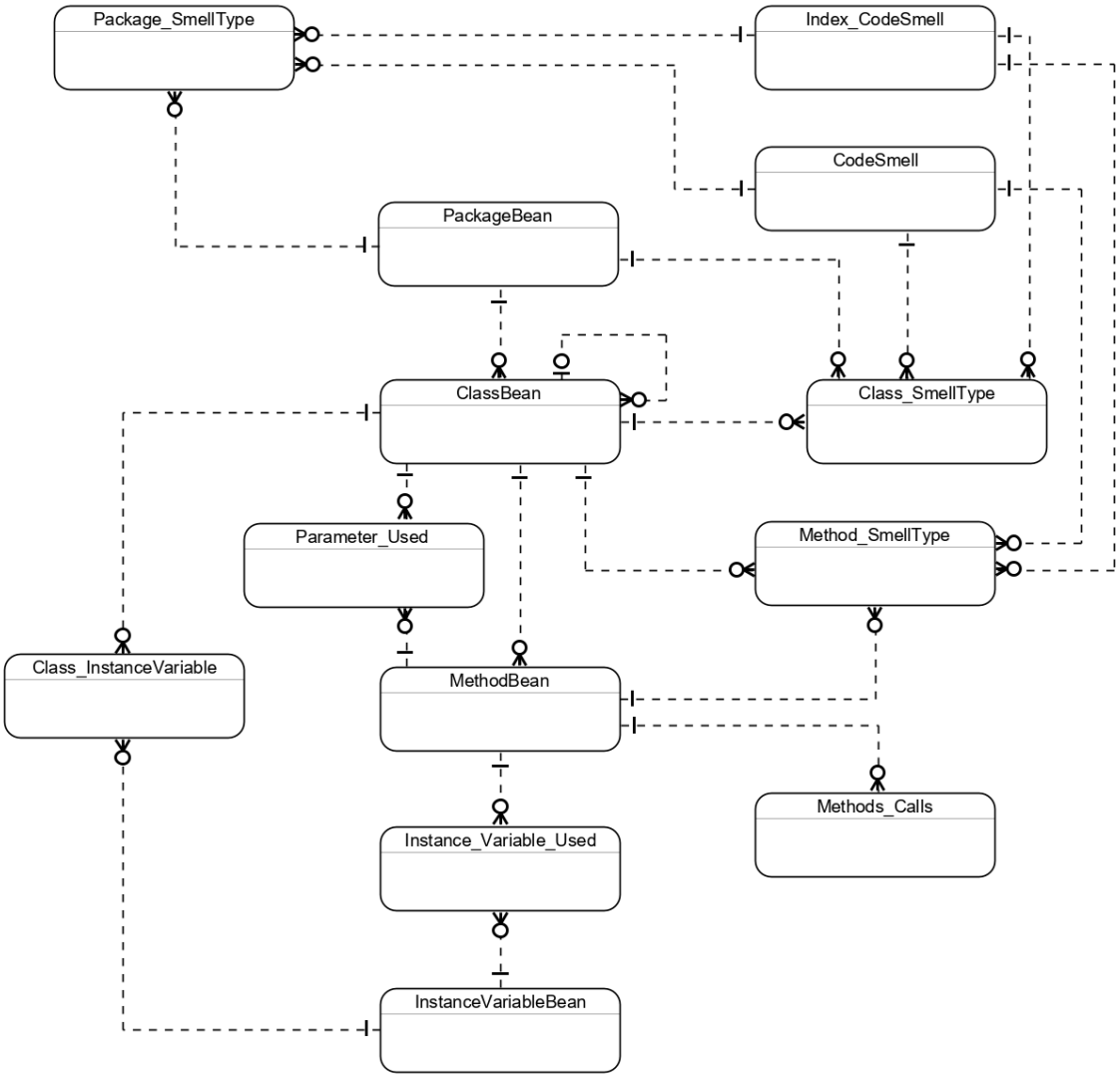

Figura 3.12: Persistence Diagram

In questo studio di tesi è stata aggiunta l'entità Index_Code smell, riportata in Tabella 3.29, in seguito all'aggiunta delle quattro nuove tecniche strutturali di analisi per garantire l'integrità dei dati e una migliore gestione delle soglie calcolate.

PackageBean	Descrive i package presenti nel progetto		
Attributi	**Tipo**	**Ruolo**	**Descrizione**
fullQualifiedName	VARCHAR(255)	PRIMARY KEY	nome che identifica il package in modo univoco
textContent	TEXT NOT NULL		contenuto testuale delle classi presenti nel package

Tabella 3.21: Entità PackageBean

ClassBean	Descrive le classi presenti in un package del progetto		
Attributi	**Tipo**	**Ruolo**	**Descrizione**
fullQualifiedName	VARCHAR(255)	PRIMARY KEY	nome che identifica la classe in modo univoco
textContent	TEXT NOT NULL		contenuto testuale della classe
belongingPackage	VARCHAR(255) NOT NULL	FOREIGN KEY PackageBean (fullQualifiedName)	nome del package al quale la classe appartiene
LOC	INTEGER(10) NOT NULL		numero di linee di codice della calsse
superclass	VARCHAR(255)	FOREIGN KEY ClassBean (fullQualifiedName)	nome della eventuale super classe
entityClassUsage	INTEGER(10)		numero di getter e setter presenti nella classe
pathToFile	VARCHAR(255)		path del file della classe

Tabella 3.22: Entità ClassBean

MethodBean	Descrive i metodi presenti in una classe del progetto		
Attributi	**Tipo**	**Ruolo**	**Descrizione**
fullQualifiedName	VARCHAR(255)	PRIMARY KEY	nome che identifica il metodo in modo univoco
textContent	TEXT NOT NULL		contenuto testuale del metodo
return_type	VARCHAR(255)		tipo di ritorno del metodo
staticMethod	BINARY(1) NOT NULL		identifica se il metodo è statico
isDefaultConstructor	BINARY(1) NOT NULL		identifica se il costruttore utilizzato è quello di default
belongingClass	VARCHAR(255) NOT NULL	FOREIGN KEY ClassBean (fullQualifiedName)	nome della classe alla quale il metodo appartiene
visibility	VARCHAR(255) NOT NULL		visibilità del metodo

Tabella 3.23: Entità MethodBean

InstanceVariable Bean	Descrive le variabile d'istanza usate in un classe del progetto		
Attributi	**Tipo**	**Ruolo**	**Descrizione**
fullQualifiedName	VARCHAR(255)	PRIMARY KEY	nome che identifica la variabile d'istanza in modo univoco
tipo	VARCHAR(255) NOT NULL		tipo della variabile
initialization	VARCHAR(255) NOT NULL		valore d'inizializzazione della variabile
visibility	VARCHAR(255) NOT NULL		visibilità della variabile d'istanza

Tabella 3.24: Entità InstanceVariableBean

CodeSmell	Descrive le tipologie degli smell ricercati		
Attributi	**Tipo**	**Ruolo**	**Descrizione**
fullQualifiedName	VARCHAR(255)	PRIMARY KEY	nome che identifica lo smell
detectionStrategy	VARCHAR(255) NOT NULL		nome della tecnica di detection usata

Tabella 3.25: Entità CodeSmell

Package_SmellType	Elenco degli smell rilevati di cui sono affetti i package		
Attributi	**Tipo**	**Ruolo**	**Descrizione**
packageBeanFull-QualifiedName	VARCHAR(255) L	PRIMARY KEY FOREIGN KEY PackageBean (full-QualifiedName)	nome del package affetto dallo smell
codeSmellFull-QualifiedName	VARCHAR(255)	PRIMARY KEY FOREIGN KEY CodeSmell (full-QualifiedName)	tipologia di smell identificato
algorithmUsed	VARCHAR(255)	PRIMARY KEY	indica il tipo di tecnica usata per rilevare lo smell
indice	VARCHAR(255) NOT NULL	FOREIGN KEY Index_CodeSmell (indexId)	nome per ricercare i valori ottenuti dall'applicazione della tecnica di analisi

Tabella 3.26: Entità Package_SmellType

Class_SmellType	Elenco degli smell rilevati di cui sono affette le classi		
Attributi	**Tipo**	**Ruolo**	**Descrizione**
classBeanFull-QualifiedName	VARCHAR(255)	PRIMARY KEY FOREIGN KEY ClassBean (fullQualifiedName)	nome della classe affetta dallo smell
codeSmellFull-QualifiedName	VARCHAR(255)	PRIMARY KEY FOREIGN KEY CodeSmell (full-QualifiedName)	tipologia di smell identificato
fqn_envied_package	VARCHAR(255)	FOREIGN KEY PackageBean (full-QualifiedName)	nome del package invidiato identificato dall'algoritmo di analisi
algorithmUsed	VARCHAR(255)	PRIMARY KEY	indica il tipo di tecnica usata per rilevare lo smell
indice	VARCHAR(255)	FOREIGN KEY Index_CodeSmell (indexId)	nome per ricercare i valori ottenuti dall'applicazione della tecnica di analisi

Tabella 3.27: Entità Class_SmellType

Method_SmellType	Elenco degli smell rilevati di cui sono affetti i metodi		
Attributi	**Tipo**	**Ruolo**	**Descrizione**
methodBeanFull-QualifiedName	VARCHAR(255)	PRIMARY KEY FOREIGN KEY MethodBean (full-QualifiedName)	nome della classe affetta dallo smell
codeSmellFull-QualifiedName	VARCHAR(255)	PRIMARY KEY FOREIGN KEY CodeSmell (full-QualifiedName)	tipologia di smell identificato
fqn_envied_class	VARCHAR(255)	FOREIGN KEY ClassBean (fullQualifiedName)	nome della classe invidiata identificata dall'algoritmo di analisi
algorithmUsed	VARCHAR(255)	PRIMARY KEY	indica il tipo di tecnica usata per rilevare lo smell

indice	VARCHAR(255)	FOREIGN KEY Index_CodeSmell (indexId)	nome per ricercare i valori ottenuti dall'applicazione della tecnica di analisi

Tabella 3.28: Entità Method_ SmellType

Index_CodeSmell	Elenco delle soglie degli smell identificati		
Attributi	**Tipo**	**Ruolo**	**Descrizione**
indexId	VARCHAR(255)	PRIMARY KEY	concatenazione del nome dello smell e del nome della componente affetta
indice	INTEGER(5)	PRIMARY KEY	valore ottenuto dall'applicazione della tecnica
name	VARCHAR(255)	NOT NULL	nom della soglia calcolata

Tabella 3.29: Entità Index_CodeSmell

Parameter_Used	Elenco dei parametri usati per richiamare un metodo di una classe		
Attributi	**Tipo**	**Ruolo**	**Descrizione**
methodBeanFull-QualifiedName	VARCHAR(255)	PRIMARY KEY FOREIGN KEY MethodBean (full-QualifiedName)	nome del metodo richiamato che usa il parametro
parameterClassFull-QualifiedName	VARCHAR(255)	PRIMARY KEY	nome che identifica il parametro
typeParameter	VARCHAR(255) NOT NULL		nome della classe che identifica il tipo del parametro
classBeanFull-QualifiedName	VARCHAR(255) NOT NULL	FOREIGN KEY ClassBean (fullQualifiedName)	nome della classe che contiene il metodo che usa il parametro

Tabella 3.30: Entità Parameter_Used

Class_Instance-Variable	Elenco delle variabili d'istanza presenti nelle classi		
Attributi	**Tipo**	**Ruolo**	**Descrizione**
classBeanFull-QualifiedName	VARCHAR(255)	PRIMARY KEY FOREIGN KEY ClassBean (fullQualifiedName)	nome della classe a cui appartiene la variabile d'istanza
instanceVariable-BeanFullQualified-Name	VARCHAR(255)	PRIMARY KEY FOREIGN KEY InstanceVariableBean (fullQualifiedName)	nome della variabile d'istanza

Tabella 3.31: Entità Class_InstanceVariable

Instance_Variable_Used	Elenco delle variabili d'istanza usate dai metodi		
Attributi	**Tipo**	**Ruolo**	**Descrizione**
methodBeanFull-QualifiedName	VARCHAR(255)	PRIMARY KEY FOREIGN KEY MethodBean (fullQualifiedName)	nome del metodo che usa la variabile d'istanza
instanceVariable-BeanFullQualified-Name	VARCHAR(255)	PRIMARY KEY FOREIGN KEY InstanceVariableBean (fullQualifiedName)	nome della variabile d'istanza usata

Tabella 3.32: Entità Instance_Variable_Used

Methods_Calls	Elenco metodi richiamati da altri metodi		
Attributi	**Tipo**	**Ruolo**	**Descrizione**
methodCallerFull-QualifiedName	VARCHAR(255)	PRIMARY KEY FOREIGN KEY MethodBean (full-QualifiedName)	nome metodo chiamante

methodCalledFull-QualifiedName	VARCHAR(255)	PRIMARY KEY FOREIGN KEY MethodBean (full-QualifiedName)	nome del metodo chiamato

Tabella 3.33: Entità Methods_Calls

3.2.5 Buondary Conditions

Le condizioni limite, rimaste identiche rispetto alla versione iniziale del plug-in, 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. Il caso di crash del sistema (dovuto a qualche malfunzionamento del plug-in) viene gestito in modo tale che l'utente, dopo aver visionato il log, ne effettui la distruzione insieme alla cache invalidata.

ID	**UC 4 - Crash**
Participating actors	Sviluppatore
Entry Condition	L'utente sta utilizzando ASCETIC su IntelliJ
Flow of events	**Utente** L'utente, durante l'utilizzo del plug-in, riscontra un crash del sistema dovuto ad un errore casuale. **Sistema** Il sistema mostra un messaggio d'errore. Successivamente, alla ripresa del normale funzionamento, mostra i log degli errori riscontrati per poi procede all'invalidazione delle cache. **Utente** 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	

3.3 Design Patterns

Strategy Pattern

Lo Strategy Pattern consente di isolare un algoritmo al di fuori di una classe, per far sì che quest'ultima possa variare dinamicamente il suo comportamento, rendendo così gli algoritmi intercambiabili a runtime. Grazie allo Strategy Pattern è possibile utilizzare una qualsiasi implementazione (che viene genericamente chiamata Strategy o Strategia), scegliendo fra quelle disponibili, che si rende più opportuna in un determinato contesto, in quanto tutte le implementazioni facenti parte della stessa famiglia hanno interfaccia comune. Questo pattern è stato usato per implementare le tecniche di refactoring e di analisi poiché per entrambe le funzionalità, il metodo da eseguire varia da smell a smell.

Per il refactoring vi è un'interfaccia "RefactoringStrategy" che viene implementata dalle seguenti classi: BlobRefactoringStrategy, FeatureEnvyRefactoringStrategy, MisplacedClassRefactoringStrategy, PromiscuousPackageRefactoringStrategy. La struttura non è stata modificata nel corso dello studio di tesi anche se la classe FeatureEnvyRefactoringStrategy è stata raffinata per coprire alcuni casi non gestiti. Il diagramma corrispondente è mostrato in Figura 3.13.

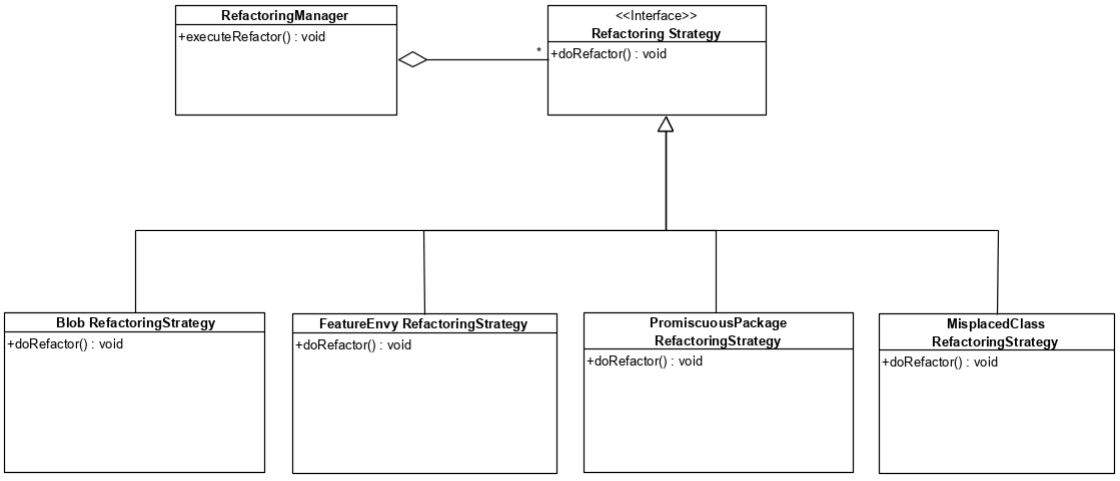

Figura 3.13: Strategy pattern del refactoring

Per l'analisi è presente un'interfaccia "CodeSmellDetectionStrategy" che viene estesa dalle seguenti interfacce: ClassSmellDetectionStrategy, MethodSmellDetectionStrategy, PackageSmellDetectionStrategy. A differenza dello strategy prima esposto, in questo caso vi è stata l'aggiunta delle classi StructuralBlobStrategy, StructuralPromiscuousPackageStrategy, StructuralMisplacedClassStrateg e StructuralFeatureEnvyStrategy in seguito al nuovo RF 1.2 circa l'identificazione strutturale. Il diagramma corrispondente è mostrato in Figura 3.14.

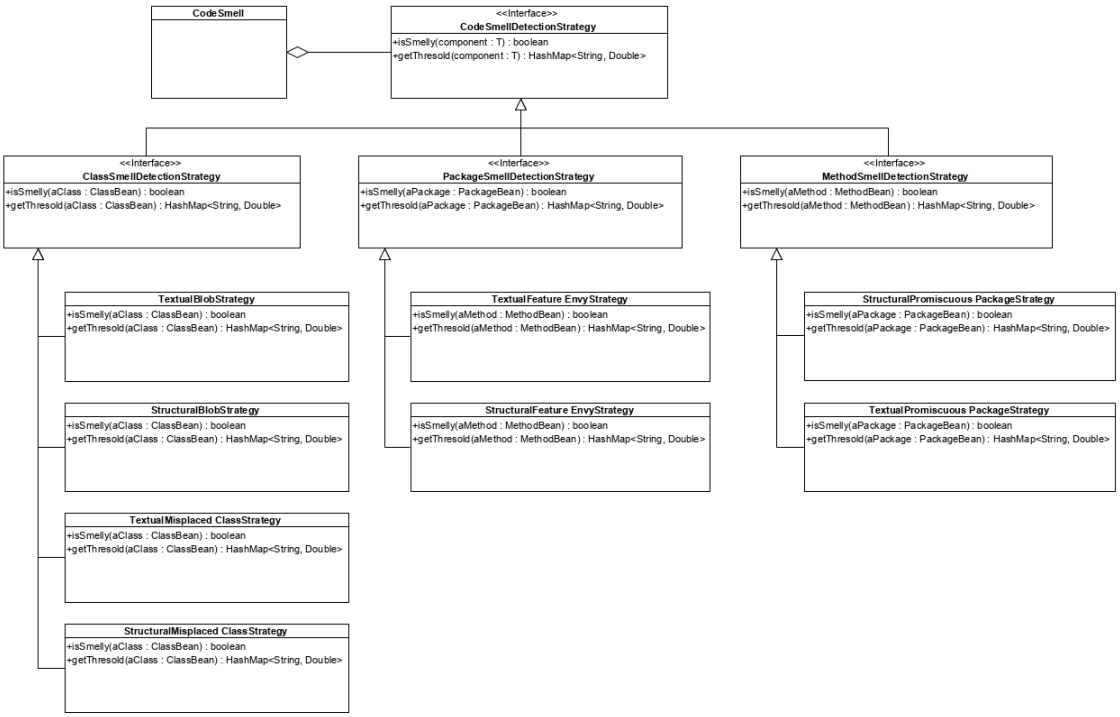

Figura 3.14: Strategy pattern dell'analisi

- L'interfaccia ClassSmellDetectionStrategy è implementata da quattro classi concrete, ossia TextBlobStrategy, StructuralBlobStrategy, StructuralMisplacedClassStrategy e TextualMisplacedClassStrategy.
- L'interfaccia MethodSmellDetectionStrategy è implementata da due classi concrete, ossia StructuralFeatureEnvyStrategy e TextualFeatureEnvyStrategy.
- L'interfaccia PackageSmellDetectionStrategy è implementata da due classe concrete, ossia TextualPromiscuousPackageStrategy e StructuralPromiscuousPackageStrategy.

Repository Pattern

I repository sono classi o componenti che incapsulano la logica necessaria per accedere ai dati sorgente. Centralizzano le funzionalità comuni di accesso ai dati, fornendo una migliore manutenibilità e disaccoppiando l'infrastruttura o la tecnologia utilizzata per accedere ai database dal layer del modello di dominio.

Questo pattern viene utilizzato per lo storage di oggetti di tipo InstanceVariableBean, MethodBean, ClassBean, PackageBean. Per implementare questo pattern viene usata l'interfaccia "Repository" che viene poi estesa dalle seguenti interfacce: InstanceVariableBeanRepository, MethodBeanRepository, ClassBeanRepository, PackageBeanRepository. La struttura non ha ricevuto modifiche durante questo studio di tesi e il diagramma corrispondente è mostrato in Figura 3.15.

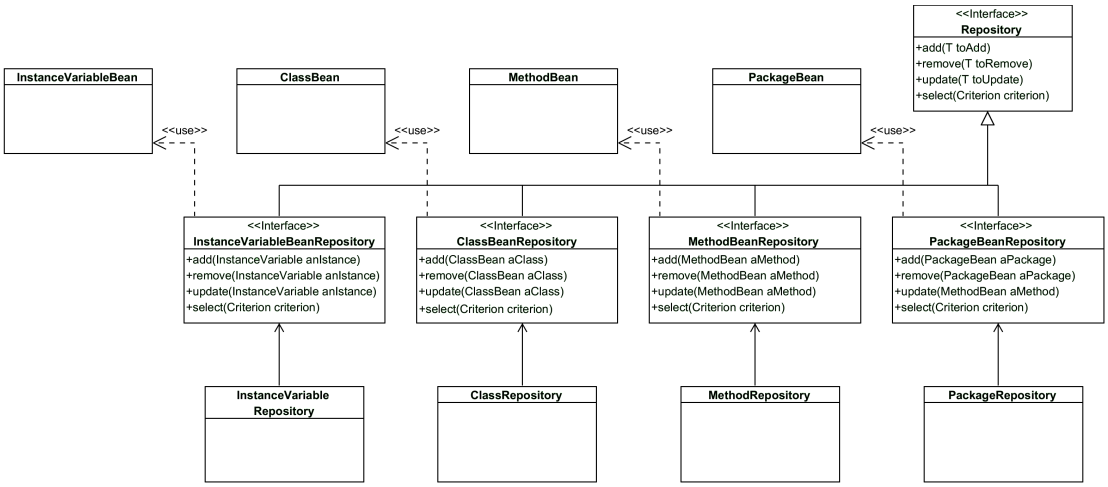

Figura 3.15: Repository Pattern

- L'interfaccia InstanceVariableBeanRepository è implementata dalla classe InstanceVariableRepository.
- L'interfaccia MethodBeanRepository è implementata dalla classe MethodRepository.
- L'interfaccia ClassBeanRepository è implementata dalla classe ClassRepository.
- L'interfaccia PackageBeanRepository è implementata dalla classe PackageRepository.

Proxy Pattern

Si tratta di un pattern strutturale basato su oggetti che viene utilizzato per gestire l'accesso ad un oggetto complesso tramite un oggetto semplice. Permette di ridurre i costi di accesso fornendo i dati solo se necessari per una certa operazione.

Questo pattern viene utilizzato per caricare una versione semplificata degli oggetti dei seguenti tipi:

- InstanceVariableList tramite la classe proxy "InstanceVariableListProxy" per ottenere la lista di variabili d'istanza presenti in una classe, oppure tramite "UsedInstanceVariableListProxy" per avere l'elenco delle variabili usate da un certo metodo;
- MethodList tramite la classe proxy "MethodListProxy" per ottenere la lista di metodi appartenenti ad una classe, oppure tramite "CalledMethodsListProxy" per avere l'elenco dei metodi richiamati da un certo metodo;
- ClassList (lista di classi appartenenti ad un package) tramite la classe proxy "ClassListProxy".

Il client può accedere a queste classi tramite interfacce "InstanceVariableBeanList", "MethodBeanList", "ClassBeanList". La struttura non ha ricevuto modifiche durante questo studio di tesi e il diagramma corrispondente è mostrato in Figura 3.16 .

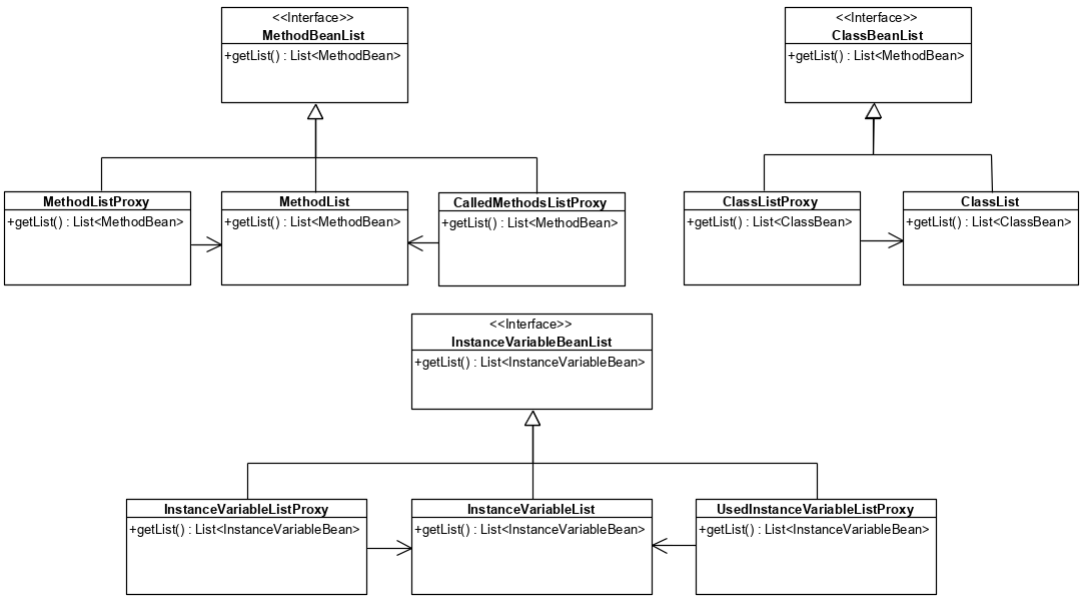

Figura 3.16: Proxy Pattern

Builder Pattern

Si tratta di un pattern creazionale basato su oggetti e viene utilizzato per creare un oggetto senza doverne conoscere i suoi dettagli implementativi. Questo pattern consente di utilizzare un Client che non debba essere a conoscenza dei passi necessari al fine della creazione di un oggetto ma tali passaggi vengono delegati ad un Director (classe che si occupa effettivamente di costruire l'oggetto) che sa cosa e come operare.

Questo pattern è usato nel sistema per costruire gli oggetti di tipo ClassBean, PackageBean, MethodBean. La struttura non ha ricevuto modifiche durante questo studio di tesi e il diagramma corrispondente è mostrato in Figura 3.17.

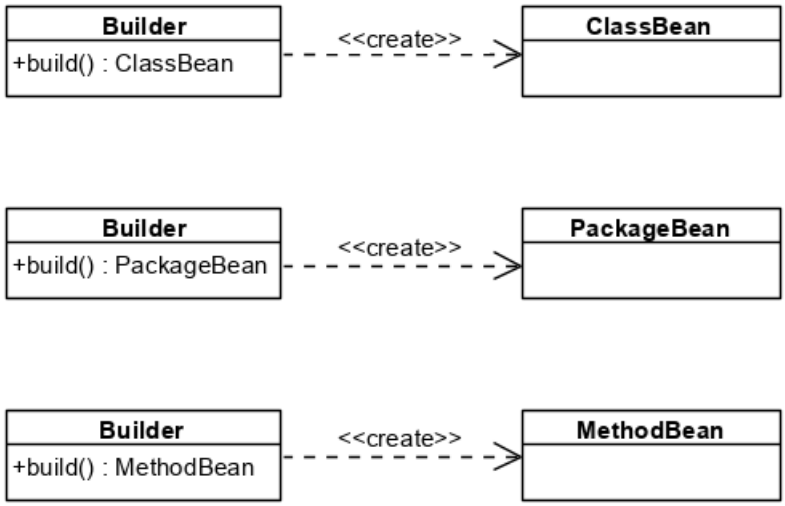

Figura 3.17: Builder Pattern dei Bean

Adapter Pattern

L'Adapter è un pattern strutturale che può essere basato sia su classi (Class Adapter) che su oggetti (Object Adapter) e il suo scopo è convertire le interfacce di una classe in altre interfacce, attese dai client, per far sì che classi con interfacce differenti e incompatibili possano comunque poter comunicare tra loro.

Questo pattern viene utilizzato per costruire le RadarMap di ogni Bean. Per questo pattern è stata definita l'interfaccia "RadarMapUtils", che viene implementata dalla classe RadarMapUtilsAdapter. La struttura non ha ricevuto modifiche durante questo studio di tesi e il diagramma corrispondente è mostrato in Figura 3.18.

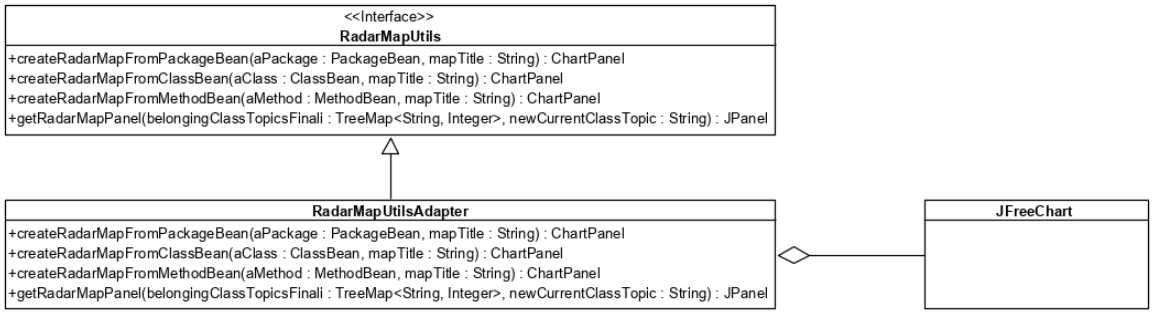

Figura 3.18: Adapter pattern delle RadarMaps

3.3.1 Manuale d'installazione

Per poter procedere all'installazione di ASCETIC è essenziale avere sul proprio device:

- IntelliJ IDEA (a partire dalla versione 2019.1.3)
- il file .zip del plug-in, scaricabile da GitHub dal seguente link: `https://github.com/simgam/ASCETIC3.0/blob/master/Code/build/distributions/ascetic-1.0-SNAPSHOT.zip`

Una volta terminati i preparativi è necessario aprire la console di Settings (tramite menu "File" o shortcut Ctlr+Alt+s) e selezionare la sottovoce "Plugins".

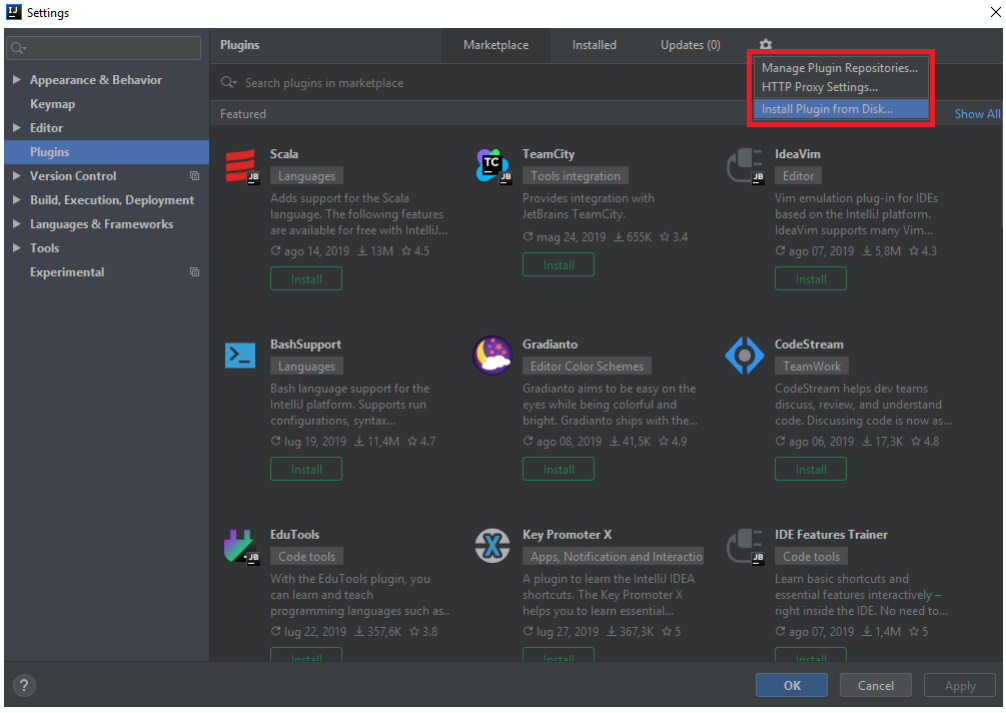

Figura 3.19: Schermata Settings

Selezionare, come mostrato in Figura 3.19, la voce "Install Plugin from Disk" e ricercare il file .zip del plug-in. Fare clic su OK per applicare le modifiche e riavviare l'IDE, se richiesto, per completare l'installazione di ASCETIC. Se il processo ha avuto esito positivo, ritornando nella finestra di controllo, sarà visibile nel menu "Plugins" il tool tra i plug-in installati come mostrato in Figura 3.20.

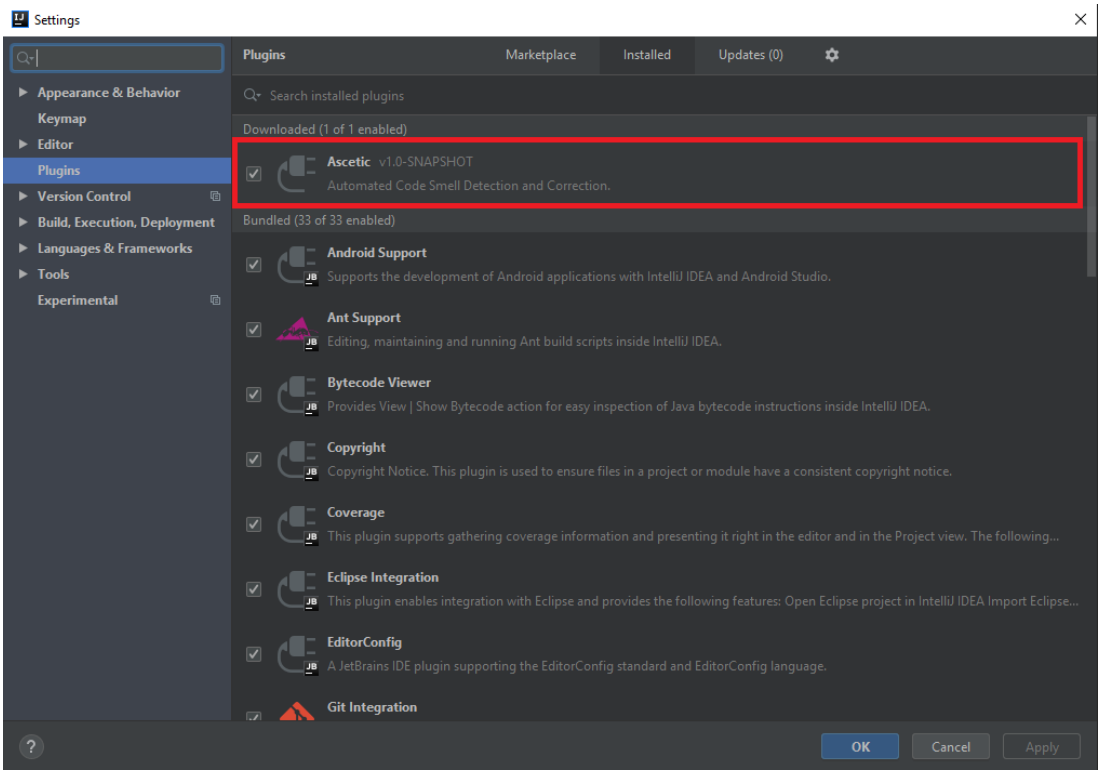

Figura 3.20: Installazione terminata con successo

Terminata l'installazione comparirà nel menu a tendina dei tool la sezione **ASCETIC- Analyze Project**, fornendo all'utente la possibilità di accedere alle funzionalità del plug-in tramite la apposite voci come descritto nella Sezione 3.2.1.

CAPITOLO 4

Conclusioni e Sviluppi Futuri

4.1 Conclusioni

Mantenere un'alta qualità del codice risulta cruciale per lo sviluppo software, poiché ciò comporta molti vantaggi in termini di comprensione e manutenzione. Raggiungere tale obbiettivo però non è facile e richiede un arduo sforzo e competenze tecniche da parte del programmatore durante gli inevitabili cambiamenti che un software subisce nel tempo. A supporto dell'utente, per l'analisi di codice per ricercare smell, vi sono molti plug-in, tra cui ASCETIC.

Allo stato attuale ASCETIC vanta l'uso di ben due tecniche differenti di analisi e la possibilità di rilevare quattro tipi differenti di smell, fornendo così agli sviluppatori un ulteriore supporto nella difficile attività di risoluzione degli smell. Sono state estese le funzionalità di ASCETIC allo scopo di consentire una migliore identificazione con l'aggiunta delle tecniche strutturali per Blob, Feature Envy, Misplaced Class e Promiscuous Package. La GUI ha subito modifiche per una maggiore comprensione e per poter sfruttare al meglio le aggiunte fatte. Il tool ora supporta l'approccio singolo o combinato dell'analisi testuale e strutturale per quanto concerne l'identificazione e permette il refactoring dei quattro tipi dei smell sopra citati. Viene fornita inoltre, la possibilità all'utente di modificare a piacimento i parametri di analisi per effettuare una più precisa identificazione e le soglie da applicare agli algoritmi di analisi.

4.2 Sviluppi Futuri

Una delle caratteristiche che ha contrassegnato lo sviluppo del plug-in riguarda la sua flessibilità alle modifiche future. Pertanto, si potrebbe facilmente estendere l'approccio di individuazione anche ad altri tipi di smell finora tralasciati, introducendo le rispettive operazioni di refactoring e eventuali nuove tecniche per la gestione dell'analisi del codice (come quelle esposte nella sezione 2.1.3 e 2.1.4 ma non implementate). Come possibile estensione potrebbe essere inserito inoltre un controllo sui test legati al codice manipolato in modo da verificare che, successivamente al refactoring subito, tali test diano ancora lo stesso esito riportato nella precedente esecuzione.

ASCETIC utilizza le API di IntelliJ per la manipolazione del codice sorgente ma necessita di ulteriori perfezionamenti circa la definizione della proposta di soluzione nel caso del Blob e del Promiscuous Package, poiché tra le più complesse, per poter competere con gli altri plug-in. L'algoritmo di analisi, inoltre, viene eseguito all'avvio del software per poi memorizzare i risultati in memoria primaria e secondaria tramite una meccanica di caching. Il tempo di esecuzione variabile, in base alla grandezza del progetto, e la quantità di dati da memorizzare può rappresentare un collo di bottiglia, provocando per grandi progetti un calo delle prestazioni. Una possibile soluzione per risolvere analisi troppo lunghe sarebbe quella di eliminare le repository sfruttando al loro posto API di IntelliJ.

Ringraziamenti

Mentre la prima parte di questo lungo percorso sta per concludersi, la successiva già è iniziata a pieno ritmo e di nuovo mi ritrovo tra i banchi a studiare e acquisire nozioni per il mio futuro. Questi tre anni sono stati pieni di sacrifici e nottate insonni ma non sono mancate le soddisfazioni. Ho potuto conoscere varie realtà, e ringrazio i professori che ogni giorno portano in aula il loro sapere sperando che qualche studente volenteroso sia pronto ad accogliere tali conoscenze, ed è stata proprio questa volontà a spingermi fin qui venendo trascinato dalla passione per questo campo. Ho fatto molte scelte importanti durante la mia carriera studentesca, che mi hanno portato via molto. Ringrazio la mia famiglia che mi ha supportato per questi lunghi anni spingendomi a non demordere, i miei parenti e amici a cui non ho dedicato il giusto tempo e attenzioni per concentrarmi sullo studio. Spero che questo piccolo traguardo possa in parte colmare queste mancanze e il loro sforzo, ma so già che tutto questo si ripeterà tra due anni quindi chiedo ancora un po' di pazienza. Termino la mia prima tappa con uno studio di tesi nella materia che più mi ha appassionato in questa triennale, e dopo il molto sforzo e tempo speso posso dire di esserne orgoglioso. Un ringraziamento va in particolar modo al mio relatore e al mio dottorando senza cui non avrei mai portato a termine questo lavoro e che mi hanno seguito con molta attenzione in quest'ultimo periodo. Ringrazio i miei coinquilini e anche tutti i miei compagni di corso per ogni risata, ogni partita fatta, ogni pizza o panino mangiato assieme, ogni posto conservato, ogni progetto portato a termine e ogni sessioni di studio su Discord. Un ringraziamento va anche a tutti coloro che da tutor, PM o semplicemente studenti della magistrale sono diventati miei amici, aiutandomi con le scelte future, concedendomi il loro tempo, con la speranza un giorno di poterli chiamare colleghi, di raggiungere il loro livello. Ho passato gli ultimi 8 mesi tra "cavalli morti", code smell ed esami da preparare in poco tempo per rispettare le scadenze di ASCETIC, diventato il più grande progetto che abbia mai portato a termine sperando sia il primo di una lunga serie.

Infine, un ringraziamento speciale va ad una persona che dall'inizio mi è sempre stata vicina in varie vesti, che mi ha sopportato ad ogni esame, ad ogni errore compiuto diventando la mia psicologa, la mia infermiera, la mia spalla, la mia fidanzata, il mio amore. Cercherò di cogliere tutte le opportunità che mi si proporranno in futuro anche se dovessi indossare una machera per nascondere i sacrifici, perché le coincidenze non esistono.

Bibliografia

[1] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018. (Citato alle pagine 2, 6, 7 e 16)

[2] F. Palomba, A. Panichella, A. De Lucia, R. Oliveto, and A. Zaidman, "A textual-based technique for smell detection," in *2016 IEEE 24th international conference on program comprehension (ICPC)*, pp. 1–10, IEEE, 2016. (Citato alle pagine 2, 4, 13 e 33)

[3] B. F. Webster, *Pitfalls of object-oriented development*. M and T books, 1995. (Citato a pagina 6)

[4] A. J. Riel, *Object-oriented design heuristics*, vol. 335. Addison-Wesley Publishing Company, 1996. (Citato a pagina 6)

[5] W. H. Brown, R. C. Malveau, H. W. McCormick, and T. J. Mowbray, *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., 1998. (Citato a pagina 6)

[6] A. Cimitile and G. Visaggio, "Software salvaging and the call dominance tree," *Journal of Systems and Software*, vol. 28, no. 2, pp. 117–127, 1995. (Citato a pagina 7)

[7] S. C. Shaw, M. Goldstein, M. Munro, and E. Burd, "Moral dominance relations for program comprehension," *IEEE Transactions on Software Engineering*, vol. 29, no. 9, pp. 851–863, 2003. (Citato a pagina 7)

[8] G. Antoniol, G. Casazza, M. Di Penta, and E. Merlo, "A method to re-organize legacy systems via concept analysis," in *Proceedings 9th International Workshop on Program Comprehension. IWPC 2001*, pp. 281–290, IEEE, 2001. (Citato a pagina 7)

[9] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2009. (Citato a pagina 9)

[10] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pp. 350–359, IEEE, 2004. (Citato a pagina 9)

[11] M. Lanza and R. Marinescu, *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007. (Citato a pagina 9)

[12] M. J. Munro, "Product metrics for automatic identification of" bad smell" design problems in java source-code," in *11th IEEE International Software Metrics Symposium (METRICS'05)*, pp. 15–15, IEEE, 2005. (Citato a pagina 9)

[13] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, 2009. (Citato alle pagine 9 e 10)

[14] B. Henderson-Sellers, *Object-oriented metrics: measures of complexity*. Prentice-Hall, Inc., 1995. (Citato a pagina 33)

[15] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of type-checking bad smells," in *2008 12th European Conference on Software Maintenance and Reengineering*, pp. 329–331, IEEE, 2008. (Citato a pagina 9)

[16] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Identification and application of extract class refactorings in object-oriented systems," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2241–2260, 2012. (Citato alle pagine 9 e 12)

[17] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo, "A review-based comparative study of bad smell detection tools," in *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, p. 18, ACM, 2016. (Citato a pagina 9)

[18] M. Zhang, T. Hall, and N. Baddoo, "Code bad smells: a review of current knowledge," *Journal of Software Maintenance and Evolution: research and practice*, vol. 23, no. 3, pp. 179–202, 2011. (Citato a pagina 9)

[19] D. C. Atkinson and T. King, "Lightweight detection of program refactorings," in *12th Asia-Pacific Software Engineering Conference (APSEC'05)*, pp. 8–pp, IEEE, 2005. (Citato a pagina 10)

[20] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, "Methodbook: Recommending move method refactorings via relational topic models," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 671–694, 2013. (Citato alle pagine 10, 11 e 12)

[21] G. Bavota, M. Gethers, R. Oliveto, D. Poshyvanyk, and A. d. Lucia, "Improving software modularization via automated analysis of latent topics and dependencies," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 1, p. 4, 2014. (Citato a pagina 11)

[22] N. Moha, "Detection and correction of design defects in object-oriented designs," in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pp. 949–950, ACM, 2007. (Citato a pagina 11)

[23] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Jdeodorant: identification and application of extract class refactorings," in *2011 33rd International Conference on Software Engineering (ICSE)*, pp. 1037–1039, IEEE, 2011. (Citato a pagina 12)

[24] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, "Automating extract class refactoring: an improved method and its evaluation," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1617–1664, 2014. (Citato a pagina 12)

[25] J. F. Nash *et al.*, "Equilibrium points in n-person games," *Proceedings of the national academy of sciences*, vol. 36, no. 1, pp. 48–49, 1950. (Citato alle pagine 12 e 13)

[26] F. Simon, F. Steinbruckner, and C. Lewerentz, "Metrics based refactoring," in *Proceedings Fifth European Conference on Software Maintenance and Reengineering*, pp. 30–38, IEEE, 2001. (Citato a pagina 13)

[27] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, "Using structural and semantic measures to improve software modularization," *Empirical Software Engineering*, vol. 18, no. 5, pp. 901–932, 2013. (Citato alle pagine 13 e 21)

[28] A. Marcus and D. Poshyvanyk, "The conceptual cohesion of classes," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pp. 133–142, IEEE, 2005. (Citato a pagina 34)

[29] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Mining version histories for detecting code smells," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 462–489, 2014. (Citato a pagina 14)

[30] D. Ratiu, S. Ducasse, T. Gîrba, and R. Marinescu, "Using history information to improve design flaws detection," pp. 223–232, 2004. (Citato a pagina 14)

[31] E. Alpaydin, *Introduction to machine learning*. MIT press, 2014. (Citato a pagina 14)

[32] J. Kreimer, "Adaptive detection of design flaws," *Electronic Notes in Theoretical Computer Science*, vol. 141, no. 4, pp. 117–136, 2005. (Citato a pagina 15)

[33] L. Amorim, E. Costa, N. Antunes, B. Fonseca, and M. Ribeiro, "Experience report: Evaluating the effectiveness of decision trees for detecting code smells," in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 261–269, IEEE, 2015. (Citato a pagina 15)

[34] S. Vaucher, F. Khomh, N. Moha, and Y.-G. Guéhéneuc, "Tracking design smells: Lessons from a study of god classes," in *2009 16th Working Conference on Reverse Engineering*, pp. 145–154, IEEE, 2009. (Citato a pagina 15)

[35] A. Maiga, N. Ali, N. Bhattacharya, A. Sabane, Y.-G. Gueheneuc, and E. Aimeur, "Smurf: A svm-based incremental anti-pattern detection approach," in *2012 19th Working Conference on Reverse Engineering*, pp. 466–475, IEEE, 2012. (Citato a pagina 15)

[36] A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y.-G. Guéhéneuc, G. Antoniol, and E. Aïmeur, "Support vector machines for anti-pattern detection," in *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pp. 278–281, IEEE, 2012. (Citato a pagina 15)

[37] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," in *2009 Ninth International Conference on Quality Software*, pp. 305–314, IEEE, 2009. (Citato a pagina 15)

[38] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "Bdtex: A gqm-based bayesian approach for the detection of antipatterns," *Journal of Systems and Software*, vol. 84, no. 4, pp. 559–572, 2011. (Citato a pagina 15)

[39] S. Hassaine, F. Khomh, Y.-G. Guéhéneuc, and S. Hamel, "Ids: An immune-inspired approach for the detection of software design smells," in *2010 Seventh International*

Conference on the Quality of Information and Communications Technology, pp. 343–348, IEEE, 2010. (Citato a pagina 15)

[40] R. Oliveto, F. Khomh, G. Antoniol, and Y.-G. Guéhéneuc, "Numerical signatures of antipatterns: An approach based on b-splines," in *2010 14th European Conference on Software Maintenance and Reengineering*, pp. 248–251, IEEE, 2010. (Citato a pagina 15)

[41] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pp. 87–98, ACM, 2016. (Citato a pagina 15)

[42] J. Yang, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, "Classification model for code clones based on machine learning," *Empirical Software Engineering*, vol. 20, no. 4, pp. 1095–1125, 2015. (Citato a pagina 15)

[43] F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1143–1191, 2016. (Citato alle pagine 15 e 16)

[44] F. A. Fontana, M. Zanoni, A. Marino, and M. V. Mäntylä, "Code smell detection: Towards a machine learning-based approach," in *2013 IEEE International Conference on Software Maintenance*, pp. 396–399, IEEE, 2013. (Citato a pagina 15)

[45] F. A. Fontana and M. Zanoni, "Code smell severity classification using machine learning techniques," *Knowledge-Based Systems*, vol. 128, pp. 43–58, 2017. (Citato alle pagine 15 e 16)

[46] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia, "Detecting code smells using machine learning techniques: are we there yet?," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 612–621, IEEE, 2018. (Citato a pagina 16)

[47] F. Pecorelli, F. Palomba, D. Di Nucci, and A. De Lucia, "Comparing heuristic and machine learning approaches for metric-based code smell detection," pp. 93–104, 2019. (Citato a pagina 16)

[48] J. Kerievsky, *Refactoring to patterns*. Pearson Deutschland GmbH, 2005. (Citato a pagina 16)

[49] G. Gui and P. D. Scott, "Coupling and cohesion measures for evaluation of component reusability," in *Proceedings of the 2006 international workshop on Mining software repositories*, pp. 18–21, ACM, 2006. (Citato a pagina 19)

[50] G. Bavota, A. De Lucia, and R. Oliveto, "Identifying extract class refactoring opportunities using structural and semantic cohesion measures," *Journal of Systems and Software*, vol. 84, no. 3, pp. 397–414, 2011. (Citato a pagina 19)

[51] A. Marcus, D. Poshyvanyk, and R. Ferenc, "Using the conceptual cohesion of classes for fault prediction in object-oriented systems," *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 287–300, 2008. (Citato a pagina 19)

[52] F. Palomba, M. Tufano, G. Bavota, R. Oliveto, A. Marcus, D. Poshyvanyk, and A. De Lucia, "Extract package refactoring in aries," in *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pp. 669–672, IEEE Press, 2015. (Citato a pagina 21)