Project Documentation

cASpeR

Riferimento	
Data	22/05/2022
Destinatario	Prof. De Lucia, Dott. Di Nucci, Dott. Manuel De Stefano, Dott. Emanuele Iannone
Presentato da	Lerose Ludovico, Milione Vincent
Approvato da	

Sommario

1. Introduzione e Planning

La documentazione presenta le informazioni raccolte nella fase di reverse engineering e le scelte di analisi compiute per le change request al plugin Intellij cASpeR, illustrate nel documento della project proposal. Eventuali cambiamenti allo scope della change request resteranno confinati a questo documento per ricordare lo scopo originale del progetto. Prima d'iniziar, introduciamo alcuni aspetti legati alla pianificazione delle attività compiute e delle attività di software configuration management. La pianificazione del testing verrà riportata nel documento relativo al testing.

1.2 Pianificazione Reverse engineering

Non avendo a disposizione della documentazione pregressa sul plugin cASpeR, si è deciso di condurre code reverse engineering con lo scopo di fare design recovery del tool. Rappresentiamo attraverso il paradigma GMT il modo in cui intendiamo portare avanti questa attività e gli obbiettivi che la guidano.

Goals:

- Identificazione delle dipendenze alle API di Intellij
- Comprendere cosa fanno le API di Intellij
- Identificare le parti responsabili del processo di analisi statica dei smells
- Identificare le parti responsabili del refactoring di uno smell
- Possibilmente suddividere verticalmente il funzionamento del tool in base allo smell

Method: Utilizzeremo una rappresentazione UML. In particolare, useremo i class diagram per avere un'idea delle relazioni tra componenti e i sequence diagram per dare un'idea del workflow del tool.

Tools: Misto tra i tool di generazione automatica dei diagrammi UML dell'IDE di Intellij e program comprehension.

1.2 Pianificazione Analisi

Dato che l'intervento proposto è un intervento di reengineering, non verrà condotta un impact analysis, ma un'analisi più generica. L'analisi dell'intervento servirà a trovare possibili soluzioni ai problemi delle change request. Ogni soluzione va analizzata e va scelta in considerazione dei suoi possibili rischi e benefici per il progetto.

L'analisi dell'intervento avverrà in seguito all'attività di reverse engineering usando i suoi prodotti. In particolare, andranno usati i class diagram estratti per capire se l'eliminazione di qualche componente potrebbe indirettamente impattare qualche altra.

Scelta la soluzione da adottare, andrà specificato come implementarla.

1.3 Pianificazione Software Configuration Managment

In questa sezione verranno presentati gli elementi sottoposti a Configuration Management e le funzionalità che utilizzeremo.

Identificazione

Oltre che al prodotto software, verranno sottoposti anche i documenti relativi al processo di reingegnerizzazione prodotti. Dunque, il documento prodotto e i documenti project proposal e testing verranno aggiunti.

Tools and Version Control

Dato che il prodotto software è già sottoposto a version control su GitHub, includeremo in una directory docs gli elementi identificati. Dato che usiamo GitHub lo stile di version control sarà di tipo copy-modify-merge. GitHub verrà usato anche per il change management mediante il suo issue tracker.

Lo stile di workspace control che adotteremo sarà di tipo isolato. Il che ci consentirà possibilmente di lavorare indipendentemente e parallelamente sulle due change request, qualora dopo l'intervento di analisi sia giudicato possibile. Entrambi gli studenti nel corso del progetto useranno Intellij 2019.3 Ultimate edition.

Siccome il tool verrà reingegnerizzato in un Maven plugin, il tool di building utilizzato per questa configurazione sarà Maven, anziché Gradle. La versione di Maven utilizzata sarà quella integrata con Intellij.

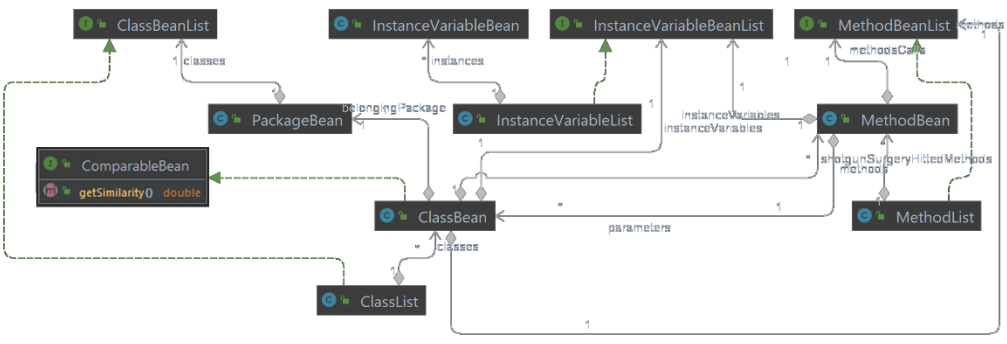
2. Reverse engineering report:

Attraverso un po' di program comprehension e l'aiuto dei tool di Intellij per la generazione dei diagrammi UML automatici abbiamo rilevato le seguenti caratteristiche per i package:

2.1 Logica applicativa

Package storage.beans

Le classi di rilievo nel package beans sono fondamentalmente MethodBean, ClassBean e PackageBean che rappresentano rispettivamente un metodo, una classe e un package all'interno di un progetto software con una lista di eventuali smell che affliggono la componente. Questi bean sono serviti per incapsulare le informazioni necessarie per



l'identificazione di smells che ci vengono passate dai bean psi di intellij.

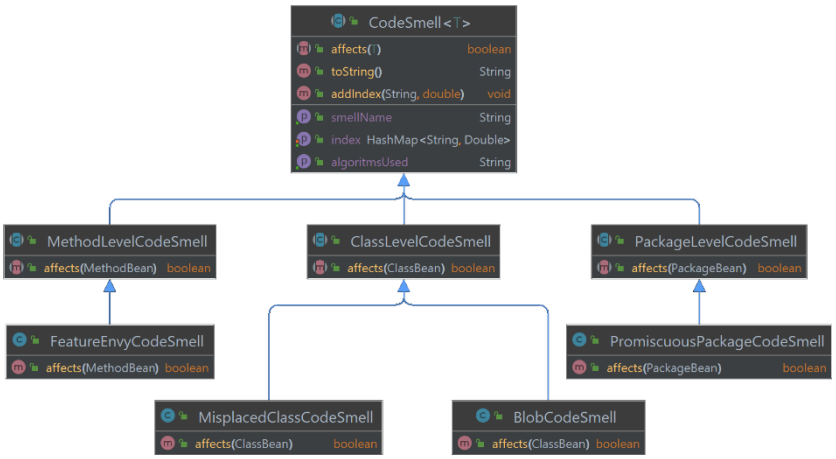
Questi bean implementano l'interfaccia ComparableBean presente nel package analyze.

Package analyze

Il package analyze ha lo scopo di effettuare l'analisi statica per l'individuazione dei code smell all'interno di un progetto software Java. Si suddivide in due package. Li analizziamo uno ad uno in sezioni separate e vedremo in che modo dipendono l'uno dall'altro.

Package code smells

Le classi che modellano code smells, presenti all'interno del package code smell, sono organizzate secondo la seguente gerarchia:



CodeSmell è una classe generica ed astratta. Modella un code smell che potrebbe avere effetto su un bean che modella una componente generica di un progetto. Presenta fra le sue variabili d'istanza delle stringhe per identificare il nome e la tipologia di algoritmo usato ("testuale" o "strutturale") per la detection dello smell in una certa componente e l'algoritmo di detection stesso.

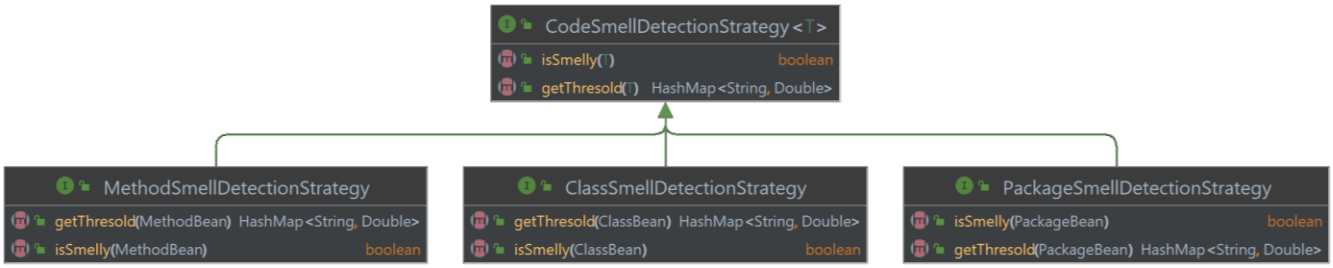
Le tre sottoclassi di CodeSmell, specificano a che livello del progetto lo smell può avere un impatto:

- Livello di metodo
- Livello di classe
- Livello di package

Il metodo astratto affects(T) è responsabile di iniziare l'analisi della componente T avviando l'algoritmo di detection passato. Restituisce l'esito mediante un booleano, true se la componente è affetta dallo smell, false altrimenti. Il metodo è implementato dai smell concreti identificabili dal sistema: Feature Envy, Misplaced Class, Blob e Promiscous Package.

Package code smell detection

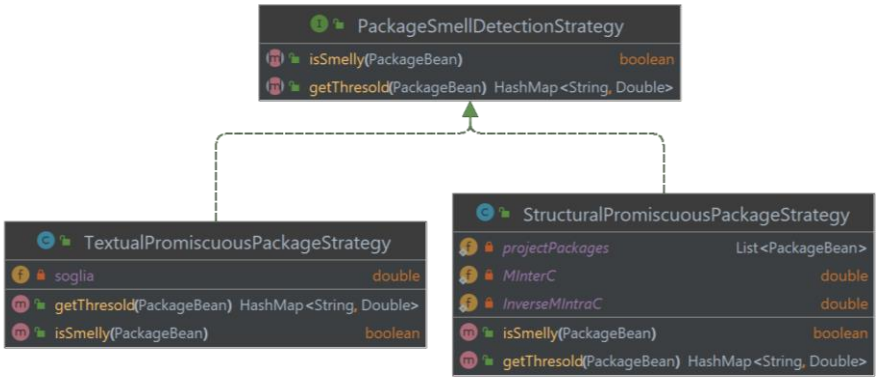
Le strategie di detection, all'interno del package code smell detection, sono organizzate come sopra.



CodeSmellDetectionStrategy<T> è un'interfaccia che presenta due metodi: isSmelly(T) chiede se la componente T generica è o meno smelly, e getThresold(T), che restituisce una lista di misure effettuate sulla componente T.

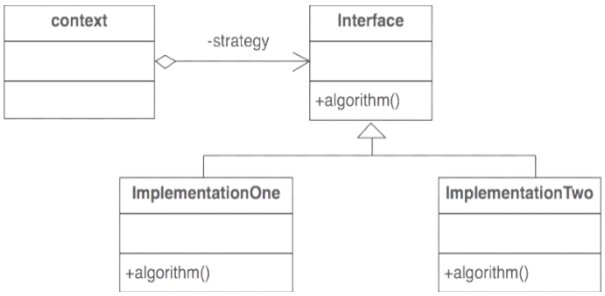
Le tre interfacce ClassSmellDetectionStrategy, MethodSmellDetectionStrategy e PackageSmellDetectionStrategy estendono CodeSmellDetectionStrategy; la componente T è parametrizzata rispettivamente usando le componenti ClassBean, MethodBean e PackageBean.

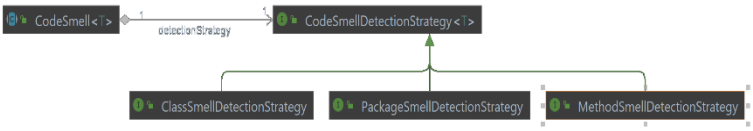
Per ogni smell è presente una coppia di classi, che indica il tipo di algoritmo (strutturale o testuale) di detection da applicare sulla componente. In totale vi sono dunque otto sottoclassi. Per ognuna di queste sottoclassi, vi sono delle variabili d'istanza che mi rappresentano una soglia. I valori di soglia vengono usati nel metodo isSmelly(T) e se nel calcolo viene superata allora la componente soffre di quel particolare smell.



Strategy Pattern

Le due gerarchie espresse finora nel package analyze, formano insieme un design pattern noto come lo strategy pattern.





L'obiettivo del pattern è isolare un algoritmo all'interno di un oggetto, così da poter essere utilizzato nelle situazioni ove si ritiene necessario. Il pattern prevede che gli algoritmi sono dunque intercambiabili - in particolare durante l'esecuzione - in base ad una condizione specificata, in modo completamente trasparente al client che ne fa uso.

Come possiamo prevedere anche dal modo in cui sono state nominate alcune classi e interfacce, La gerarchia della parte context è rappresentata dalla gerarchia dei code smell. Mentre la gerarchia delle strategie è rappresentata dalla gerarchia di CodeSmellDetectionStrategy. L'algoritmo che incapsula la cosidetta stategia è il metodo isSmelly, che come già menzionato stabilisce con il suo algoritmo se la componente passata è o meno affetta dallo smell.

Utilities (da rivedere un po' soprattutto i diagrammi)

Nel package codesmell_detection vi sono anche alcune classi di utility che incapsulano della logica comune tra alcune classi del package.

-BeanComparator: fornisce un metodo di comparazione fra due ComparableBean.

-BeanDetection: è una classe di utilità col metodo statico detection, che prende in input un MethodBean: e restituisce false se il metodo non è un setter, un getter, toString, equals, main, hashcode o un costruttore, true altrimenti. Viene usato per evitare di continuare l'analisi a livello di classe nel caso in cui questa sia composta esclusivamente dai metodi sopracitati, non rilevanti per l'identificazione di smells.

-ComponentMutation: è una classe di utilità i cui metodi prendono in input un PackageBean o un ClassBean e restituiscono una stringa col contenuto testuale della componente formattato in un certo modo.

-CosineSimilarity: è una classe di utilità che, tramite il metodo computeSimilarity, prende in input due documenti testuali, sotto forma di vettori di parole, e ne calcola la similarità del coseno.

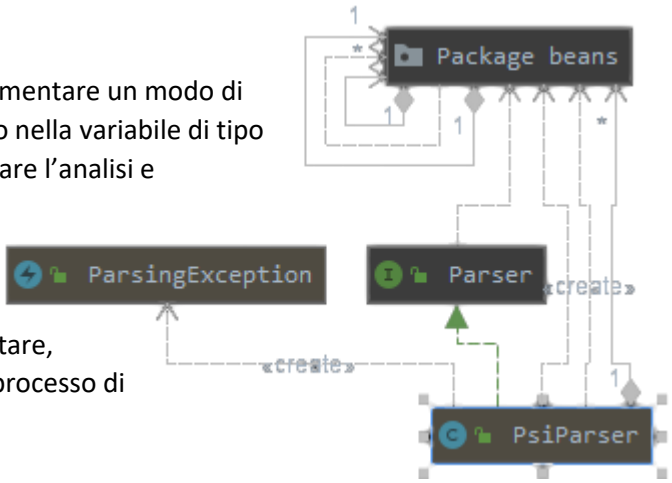
-SmellynessMetric: è una classe che, attraverso il metodo computeSmellyness, prende in input una stringa ottenuta tramite ComponentMutation e restituisce una misura utilizzata nell'analisi testuale per determinare o meno la presenza di blob o promiscous package.

Ulteriori due classi di utilità, CKMetrics e TopicExtracter, sono contenute rispettivamente all'interno dei package structuralMetrics e topic.

-CKMetrics: E' una classe che racchiude numerosi metodi statici per misurare tramite metriche strutturali determinati aspetti dei bean che gli vengono passati.

-TopicExtracter: E' una classe che prende in input un ClassBean, un MethodBean o un PackageBean e restituisce una mappa dei topic, ossia i termini più frequenti all'interno dei contenuti testuali di tali bean.

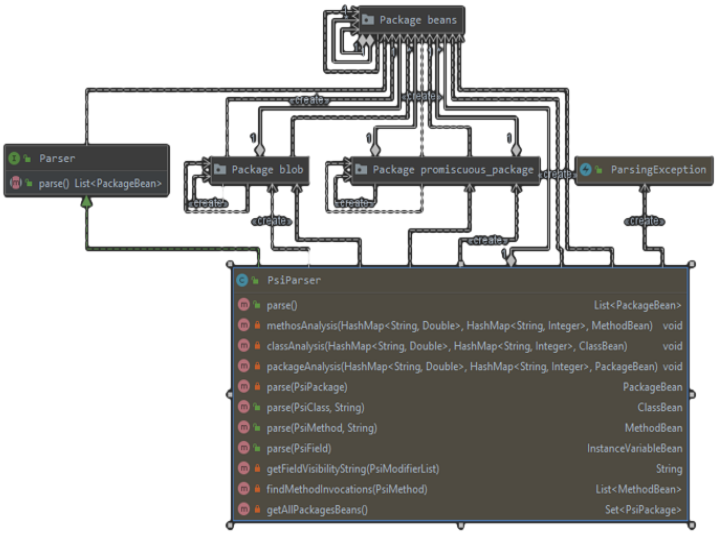
Parser Package



L'interfaccia Parser assegna la responsabilità di implementare un modo di effettuare il parsing del progetto di Intellij incapsulato nella variabile di tipo Project nelle classi bean usate dal sistema per effettuare l'analisi e rilevazione di code smell. PsiParser implementa tale comportamento. Nel caso qualcosa vada storto, lo segnala attraverso una Parsing Exception.

Oltre che a ereditare il comportamento da implementare, PsiParser ha la responsabilità aggiuntiva di iniziare il processo di analisi.

Come possiamo vedere in questo secondo diagramma, vi sono alcuni metodi privati siglati come *Analysis(…). Questi metodi sono responsabili per iniziare il processo di rilevazioni dei code smell nel progetto a livello di package, classe e metodo. Nei metodi analysis vengono istanziate la strategia che si vuole applicare e il relativo code smell. Al code smell istanziato viene passata la strategia da applicare e la relativa componente software su cui applicare la strategia. Attraverso l'invocazione del metodo affects avviamo l'applicazione dell'algoritmo di detection.

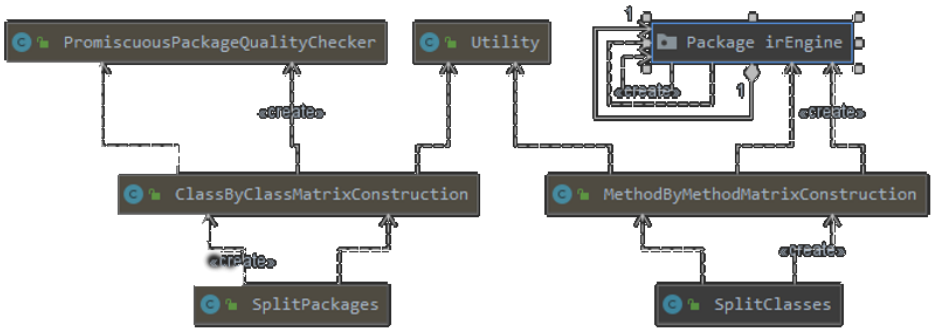


Da un'attenta analisi di program comprehension in questi metodi è possibile già stabilire quali potrebbero essere delle classi da includere nell'operazione di refactoring. Dato che vi sono alcune strategie che sono state commentate e mai utilizzate nel programma.

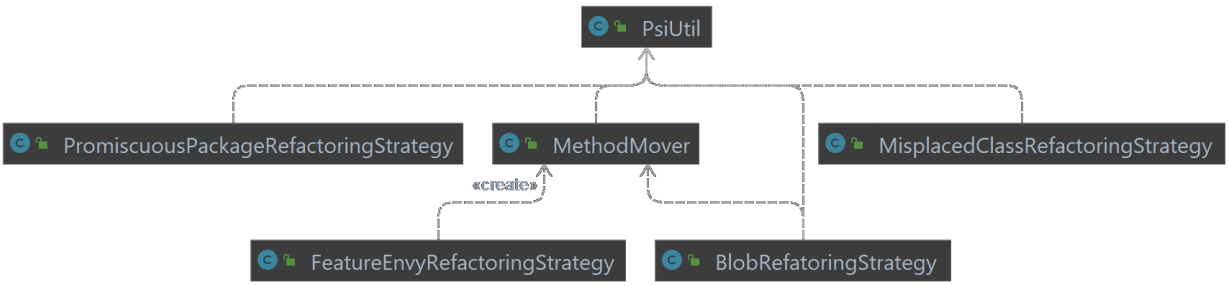
Refactoring Package:

Il refactoring package è diviso in due parti sostanziali. Nel caso fosse necessario per risolvere lo smell, la prima parte si occupa di utilizzare degli algoritmi specifici per spezzare in più parti una classe o metodo.

Questi algoritmi vengono applicati nel caso della rilevazione di blob o promiscuos package smell.



La seconda parte effettua il refactoring vero e proprio e potrebbe usare l'output del processo di splitting per completare le sue operazioni. Nella fase di refactoring avviene poi la traduzione dei nostri bean in oggetti nativi di intellij, il che è necessario per generare eventualmente dei nuovi file o ristrutturare il file esistente nel progetto java Intellij.

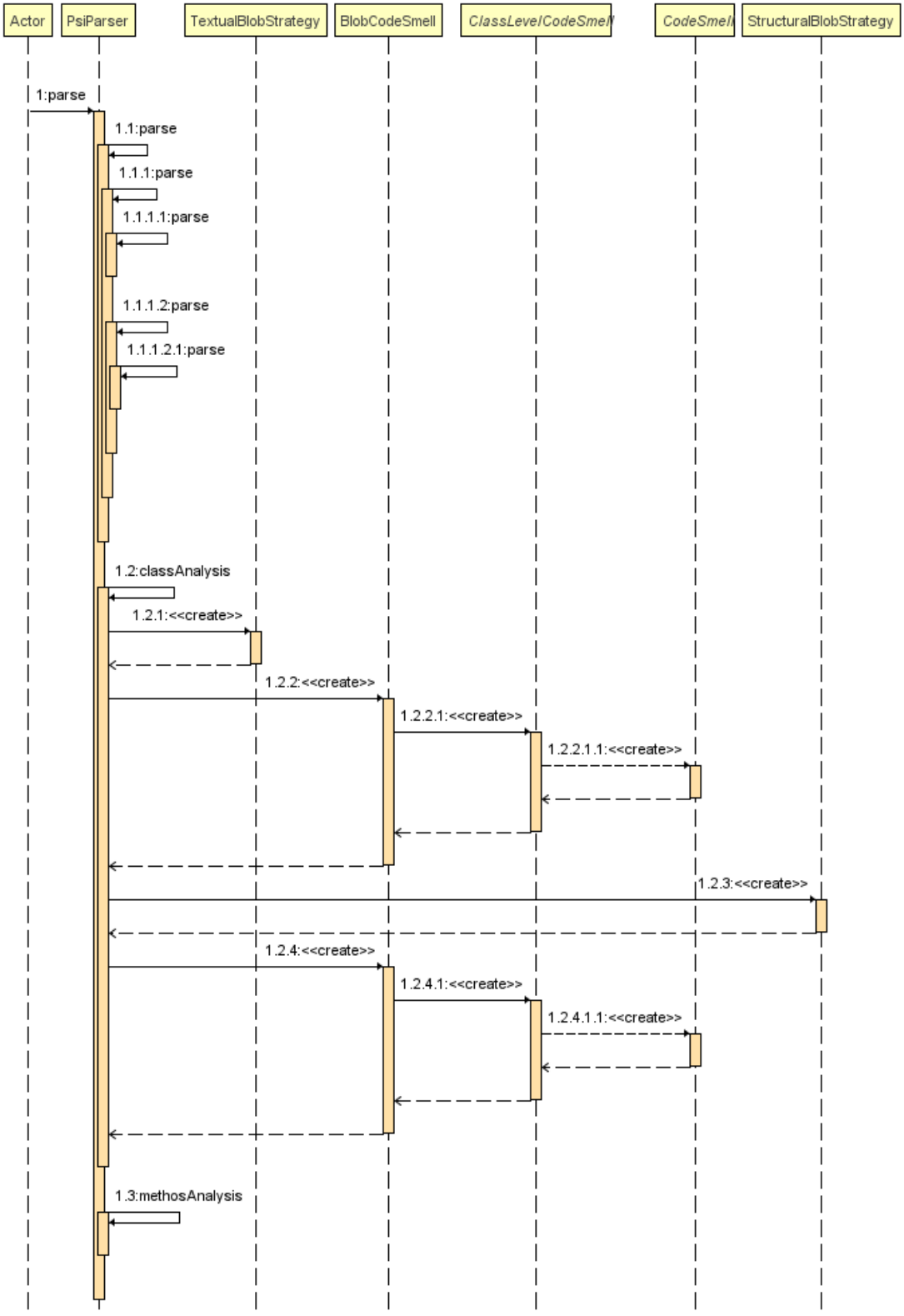


Né consengue che la parte di refactoring è fortemente accoppiata con le API di Intellij.

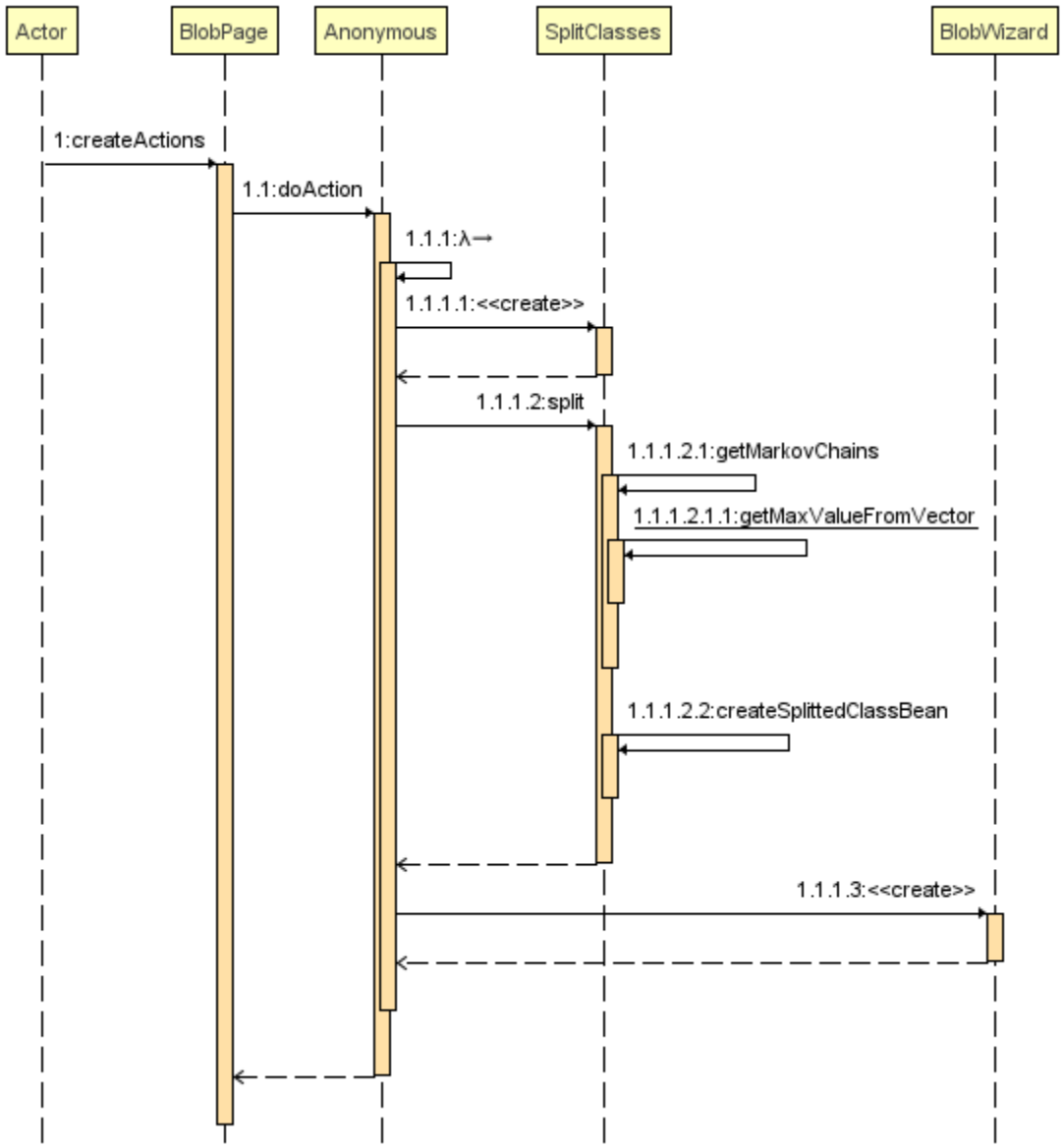
2.2 Logica d'interazione con la gui

Package gui e actions

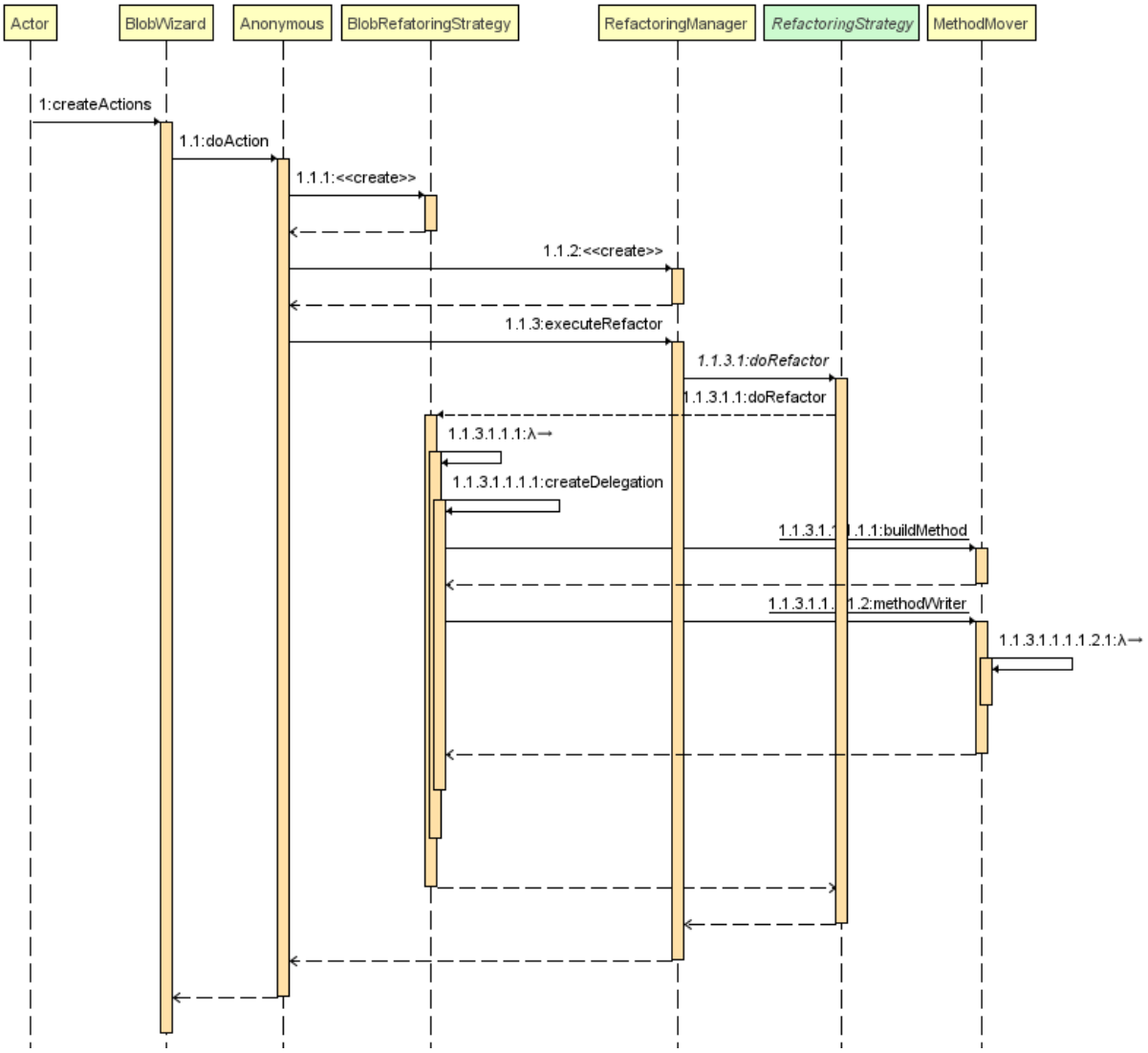
Il package actions contiene ogni ActionListener in ascolto di eventi che richiamo l'uso del plugin. Gli eventi BeginAction e CommitAction vengono lanciati per avviare il tool – il primo dal menu tools e il secondo quando avviene un commit. In entrambi i casi l'applicazione istanzia la main window CheckProjectPage da cui l'utente vede i risultati dell'analisi statica avviata a priori dall'action listener. L'utente poi sceglie tra i smell evidenziate quale refattorizzare, selezionandolo dalla tabella nella view di CheckProjectPage.


Actor
PsiParser
TextualBlobStrategy
BlobCodeSmell
ClassLevelCodeSmell
CodeSmell
StructuralBlobStrategy
1:parse
1.1:parse
1.1.1:parse
1.1.1.1:parse
1.1.1.2:parse
1.1.1.2.1:parse
1.2:classAnalysis
1.2.1:<<create>>
1.2.2:<<create>>
1.2.2.1:<<create>>
1.2.2.1.1:<<create>>
1.2.3:<<create>>
1.2.4:<<create>>
1.2.4.1:<<create>>
1.2.4.1.1:<<create>>
1.3:methosAnalysis

Ogni smell ha un suo corrispondente classe page e wizard. Viene istanziata la view page per fare un resoconto del perché il metodo, la classe o il package è affetto dallo smell. In particolare, nel caso di Promiscuos Package e blob, la view avvia un processo in background addizionale per applicare gli algoritmi di splitting visti prima.

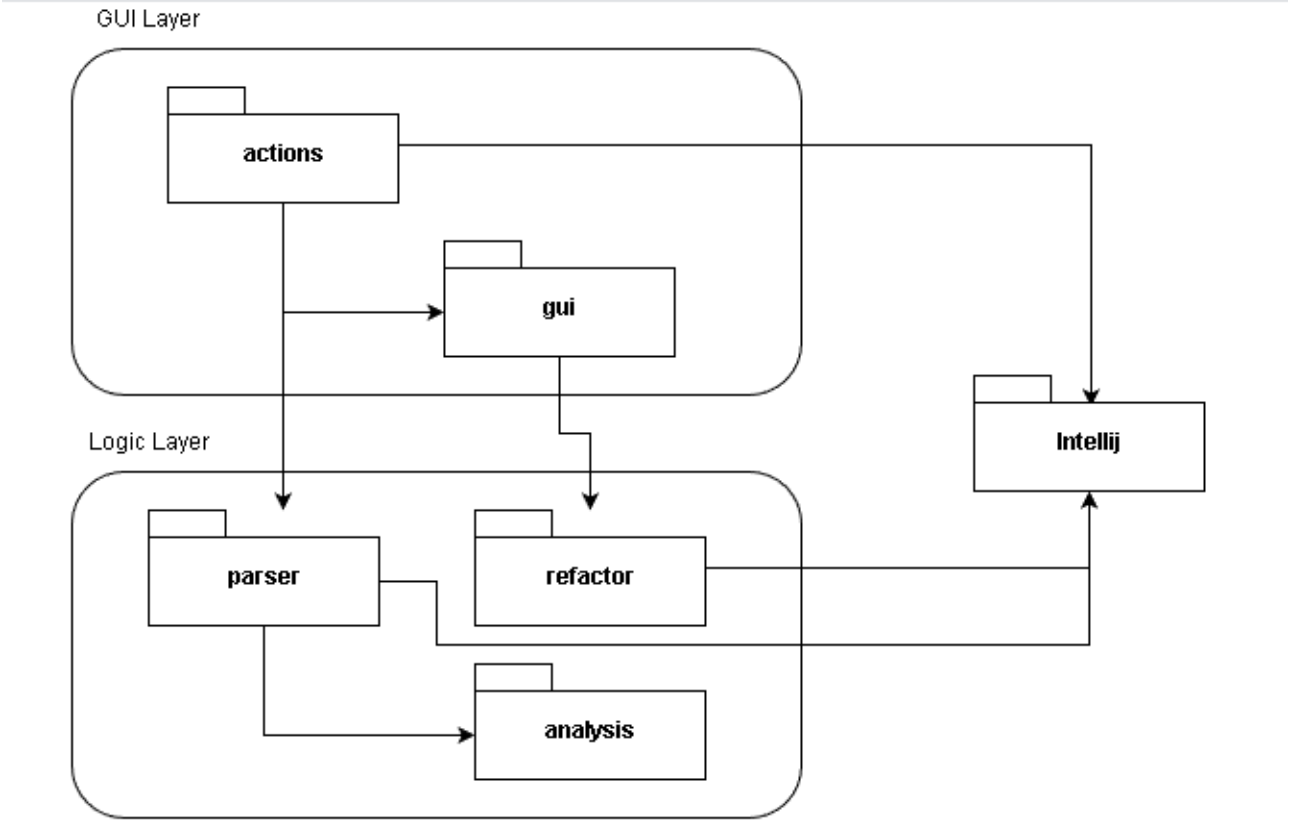


Dopodiché viene istanziato il wizard. Alla pressione del bottone “Find Solution” avvia il processo di refactoring per risolvere lo smell con la relativa strategia di refactoring.



Infine, l’action listener ConfigureAction permette di istanziare una view per i configurare i threshholds per la similarità delle classi.

2.3 Class Diagram Completo



2.4 Requisiti Estratti

Dopo l'attività di program comprehension e l'uso dei tool abbiamo ricavato i seguenti requisiti funzionali:

R1: Detection testuale del blob smell a livello di classe
R2: Detection con metriche strutturali del blob smell a livello di classe
R3: Detection testuale del Promiscuos Package smell a livello di package
R4: Detection con metriche strutturali del Promiscuos Package smell a livello di package
R5: Detection testuale del Feature Envy smell a livello di metodo
R6: Detection con metriche strutturali del Feature Envy smell a livello di metodo
R7: Detection testuale del Misplaced Class smell a livello di classe
R8: Detection con metriche strutturali del Misplaced Class smell a livello di classe
R5: Applicazione della strategia di refactoring blob smell
R6: Applicazione della strategia di refactoring di Promiscuos Package smell
R7: Applicazione della strategia di refactoring di Feature Envy smell
R8: Applicazione della strategia di refactoring di Misplaced Class smell
R9: Detection testuale del blob smell a livello di classe durante un commit
R10: Detection con metriche strutturali del blob smell a livello di classe durante un commit
R11: Detection testuale del Promiscuos Package smell a livello di package durante un commit
R12: Detection con metriche strutturali del Promiscuos Package smell a livello di package durante un commit
R13: Detection testuale del Feature Envy smell a livello di metodo durante un commit
R14: Detection con metriche strutturali del Feature Envy smell a livello di metodo durante un commit
R15: Detection testuale del Misplaced Class smell a livello di classe durante un commit

R16: Detection con metriche strutturali del Misplaced Class smell a livello di classe durante un commit
R17: Applicazione della strategia di refactoring blob smell durante un commit
R18: Applicazione della strategia di refactoring di Promiscuos Package smell durante un commit
R19: Applicazione della strategia di refactoring di Feature Envy smell durante un commit
R20: Applicazione della strategia di refactoring di Misplaced Class smell durante un commit
R21: Configurazione delle soglie dei valori di similarità

3. Analysis Operation

Prima di procedere discutiamo di diversi approcci. Per la MR1, non sono stati identificati altri approcci alla modifica se non quello di analizzare gli usi di una classe e stabilire se questa venga usato o meno. Per quanto riguarda MR2 invece abbiamo identificati queste suddette scelte:

1) Mantenere la UI

2) Non Mantenere la UI

Di seguito riportiamo i risultati della nostra impact analysis:

3.1 Analysis Operation: MR1_Refactoring

Grazie alle informazioni sul tool descritte nel paper e all'attività di reverse engineering dell'intero sistema abbiamo notato che vi è del codice non utilizzato. Queste classi non sono state infatti riportate nei diagrammi estratti dal sistema precisamente perché non svolgevano alcun ruolo. Le classi di cui parliamo sono fondamentalmente relative a code smell non implementati e le classi legate alla parte di history analysis. Dunque, le classi da eliminare dal sistema sono:

-DivergentChangeCodeSmell

-ParallelInheritanceCodeSmell

-ShotgunSurgeryCodeSmell

-HistoryBlobStrategy

-HistoryDivergentChangeStrategy

-HistoryFeatureEnvyStrategy

-HistoryParallelInheritanceStrategy

-HistoryShotgunStrategy

-AnalyzerThread

-HistoryAnalysisStartup

-PythonExeSingleton

-RepositorySingleton

-DivergentChangeRefactoryStrategy

-ParallelInheritanceStrategy

-ShotgunSurgeryRefactoringStrategy

-DivergentChangePage

-DivergentChangeWizard

-ParallelInheritancePage

-ParallelInheritanceWizard

-ShotgunSurgeryPage

-ShotgunSurgeryWizard

Nonostante le classi HistoryAnalysisStartup, DivergentChangePage, ShotgunSurgeryPage e ParallelInheritancePage non siano di fatto utilizzate nel tool finale, si sono trovati comunque dei riferimenti ad esse in altre classi tramite lo strumento “Find usages” di Intellij. Pertanto, oltre ad eliminare le classi precedentemente citate, bisogna anche modificare le seguenti classi:

-CheckProjectPage

-ConfigureThreshold

3.2 Analysis Operation: MR2_Migrazione_Maven

Per questa MR sono state considerate vari approcci che comportano l’eliminazione o meno delle dipendenze alle API di Intellij. Come gia visto nel class diagram estratto, le dipendenze alle API di Intellij sono particolarmente critiche nella fase di parsing dei bean PSI nei corrispondenti bean del pacchetto storage e nella fase di refactoring – dove avviene il viceversa.

Nel caso in cui si decide di intervenire ed eliminare queste dipendenze alle API di Intellij, si potrebbe scegliere:

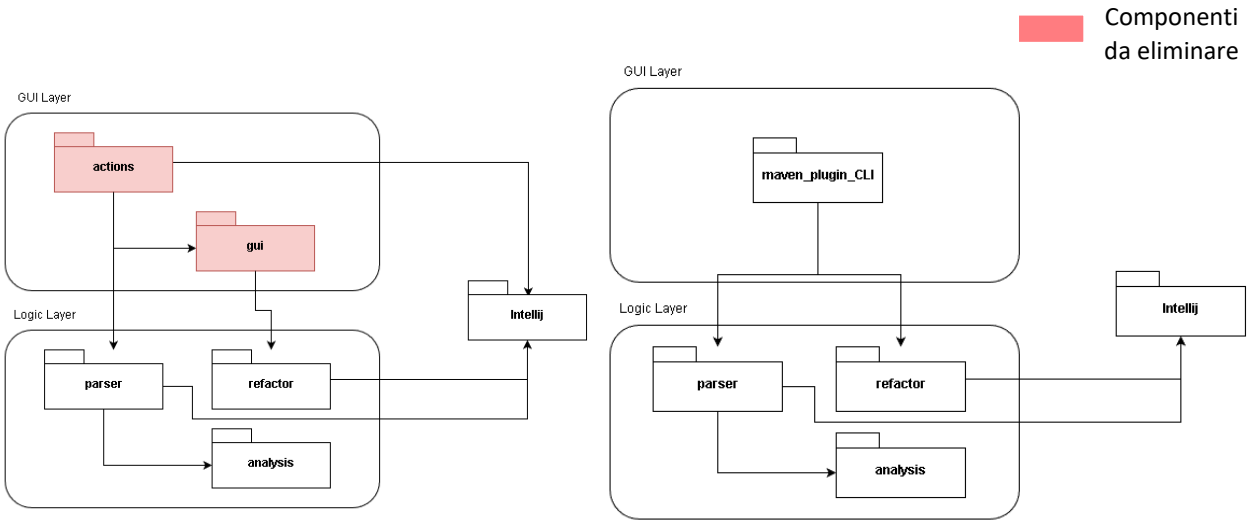
- Usare una third party library o API che permetta di fare le stesse cose che facevano le API di Intellij
- Manualmente implementare il comportamento delle API di Intellij

Sebbene i due approcci elencati siano molto diversi in termini di sforzo, comunque richiedono lo stesso tipo di cambiamento. Dunque, qui riportiamo le analisi fatte nel caso in cui si decide di mantenere le API di Intellij e il caso in cui decidiamo di eliminarle.

Prima di iniziare, vorremo premettere che a seguito di questa operazione di reengineering, tutti i requisiti funzionali del sistema sono impattati. In particolare, facciamo riferimento al set dei requisiti R9 fino a R21. Questo è dovuto a delle limitazioni di carattere tecnologico. Infatti, non è possibile implementare un Maven Plugin che si possa azionare al commit. Queste funzionalità potrebbero essere recuperate dallo sviluppatore. Attraverso la realizzazione di un “hook” nel progetto, è possibile avviare il plugin quando avviene un commit.

Inoltre, odiernamente il progetto è in Gradle, ma **per sviluppare un Maven Plugin è necessario innanzitutto trasferire il progetto in un ambiente Maven**. Ciò ha un notevole impatto riguardo le possibilità di mantenere le dipendenze con le api di intellij. La prima cosa da fare sarà cambiare il tool di building.

Mantenere API di Intellij



Nel caso in cui si trovasse un modo di conservare le dipendenze alle Api di Intellij e data la forte decomponibilità del sistema, l'intervento di reingegnerizzazione si limiterebbe a un semplice intervento di modifica del presentation layer.

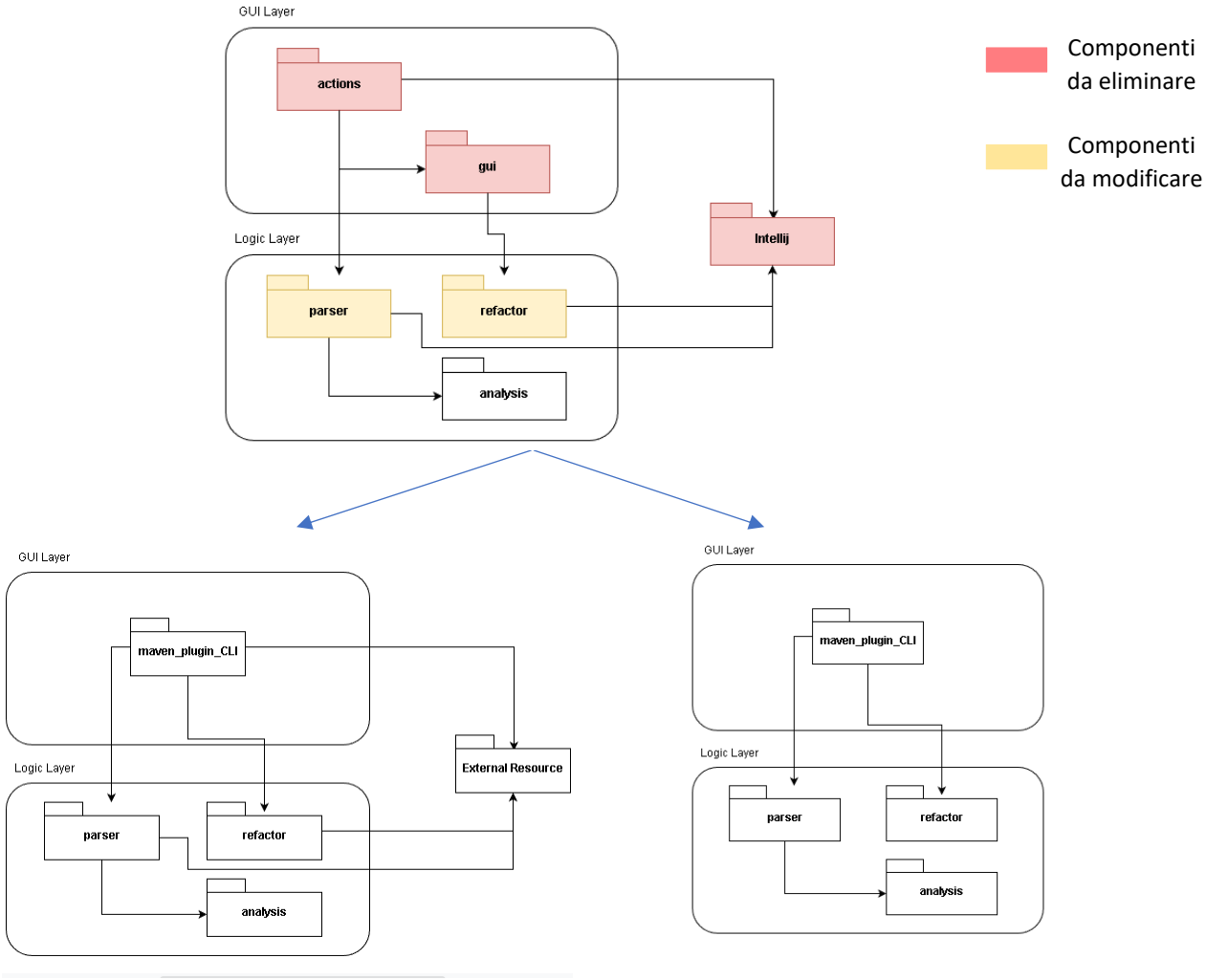
Come vediamo in figura, si prevede che i due package che formano il presentation layer (actions e gui) verranno sostituiti dal package maven_plugin_CLI. Quest'ultimo dovrà diventare responsabile di avviare il processo di analisi statica istanziando il parser e di risoluzione degli smell istanziando le varie strategie di refactoring.

Dato che in questo approccio la logica da implementare è minima, si potrebbe pensare di effettuare una reingegnerizzazione completa. Tuttavia, come anche specificato prima, in modo da poter funzionare è necessario trovare le dipendenze necessarie affinché il tool operi come da specificato.

Il rischio di non trovare queste dipendenze è alto. Pure essendoci delle dipendenze disponibili su Maven Central, bisogna capire quali sono quelle essenziali. La poca realizzabilità del criterio potrebbe portare al fallimento del progetto.

Rischi	Benefici
- Fallimento del progetto (alto)	- Numero di modifiche basso - Semplicità

Eliminare le API di Intellij



Questo caso prende in esame non solo la necessità di sostituire il livello interattivo, ma anche di eliminare le dipendenze alle api di Intellij. Come già anticipato, si potrebbe scegliere:

- Usare una third party library o API che permetta di fare le stesse cose che facevano le API di Intellij
- Manualmente implementare il comportamento delle API di Intellij

In entrambi i casi, le dipendenze alle API d'Intellij non sono per niente banali. L'uso delle API avviene sia nella fase di analisi – più precisamene nella fase di parsing dei bean psi – sia nella fase di refactoring. La strategia di approccio all'intervento potrebbe dover cambiare dato che le parti da modificare sono aumentate in numero, conseguentemente aumentando la complessità dell'intervento. Potrebbe dover essere necessario restringere lo scope dell'attività di reingegnerizzazione per far rientrare nei tempi previsti l'intervento.

Nella prima soluzione, sarebbe necessario cercare una third party library o API che ricopra il ruolo finora svolto da IntelliJ. I rischi correlati a una soluzione di questo tipo sono che i tempi per la ricerca e anche per imparare ad usare queste librerie. Inoltre, non sono da trascurare i possibili adattamenti da dover fare alle classi dipendenti da Intellij per poter usare questa nuova libreria. Viceversa, grazie ai livelli di astrazioni che potrebbe offrire una third party library, i tempi di development potrebbero ridursi.

Rischi	Benefici
- Tempi addizionali per il training - Adattamenti possono essere complessi	- Astrazioni maggiori

La seconda soluzione richiederebbe di lavorare con delle API di basso livello che consentono la manipolazione di file. Il parser costruirà direttamente a partire dai file i beans e il refactoring package dovrà essere modificato per lavorare con le API di basso livello.

Rischi e benefici per la seconda soluzione sono invece quasi il contrario.

Rischi	Benefici
- Adattamenti possono essere molto complessi - Tempi di development più lunghi	- Portabilità maggiore della logica applicativa

3.3 Criterio Adottato

Qui riportiamo il criterio adottato per MR2 in relazione ai rischi e benefici identificati nella sezione precedente. MR1 non viene dettagliata in quanto è stato identificato un unico criterio possibile.

Criterio Selezionato MR2

Seppure il primo criterio presentato sembri la strada più semplice da intraprendere, non è stato possibile attuarlo.

Come descritto nel paragrafo del package parser, il plugin ottiene dall'IDE un oggetto di tipo Project che modella il progetto attualmente aperto. Da questa variabile project estraiamo tutti i PsiBean contenenti tutte le informazioni relative ai package, classi e metodi per iniziare l'analisi statica. Senza questa variabile project, saremmo costretti a costruire manualmente i bean psi. Se così fosse allora la soluzione non sarebbe poi così diversa dal terzo criterio presentato.

Dunque, si è pensato di scaricare le dipendenze alle librerie di Intellij responsabili per l'importazione di un progetto maven. È stato tentato più di una volta di scaricare le dipendenze alle librerie di Intellij tramite il pom.xml, ma vi sono sempre delle dipendenze a delle librerie di terze parti che mancano.

Data l'impraticabilità del primo approccio, abbiamo analizzato i pro e i contro di quelli rimanenti. Siamo giunti alla conclusione che quello meno rischioso e praticabile è il secondo approccio. Il terzo approccio richiede di lavorare con dei problemi di basso livello fin troppo complessi da risolvere. Ad esempio, nella detection sarebbe necessario identificare funzioni lambda e classi interne; nel refactoring sarebbe necessario lavorare con API di basso livello come java.io.File per apportare le modifiche. Il secondo criterio è stato favorito anche perché è stato trovato molto rapidamente una library che potesse rimpiazzare il ruolo delle API di Intellij.

Questa library è JavaParser (https://github.com/javaparser). JavaParser contiene un set di libreria che implementano – come suggerisce il nome – il parser di Java dalla sua versione 1.0 alla versione 15. Inoltre, arrichisce il parser con delle nuove funzionalità basate su analisi statica. Quest'ultime non verrano sfruttate ora, ma potrebbero avere qualche uso in futuro.

Come ci si aspetterebbe da un parser, prende in input un file java e restituisce una versione del file java basata su AST.