



# 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.....	3
1.1 Pianificazione Training Maven Plugin.....	3
1.2 Pianificazione Reverse engineering.....	3
1.2 Pianificazione Analisi .....	3
1.3 Pianificazione Software Configuration Management.....	4
Identificazione .....	4
Tools and Version Control .....	4
2. Reverse engineering report:.....	4
2.1 Logica applicativa.....	4
Package storage.beans .....	4
Package analyze.....	5
Package code smells .....	5
Package code smell detection .....	6
Strategy Pattern.....	7
Utilities.....	7
Parser Package.....	8
Refactoring Package: .....	9
2.2 Logica d'interazione con la gui .....	9
Package gui e actions.....	9
2.3 Class Diagram .....	13
2.4 Requisiti Estratti .....	14
3. Analysis Operation.....	14
3.1 Analysis Operation: MR1_Refactoring .....	14
3.2 Analysis Operation: MR2_Migrazione_Maven.....	16
Mantenere API di IntelliJ.....	16
Eliminare le API di IntelliJ.....	17
Criterio Selezionato MR2.....	18
4 Progettazione .....	19
4.1 Progettazione MR2.....	19
Progettazione maven_plugin_cli .....	20
Progettazione parser .....	20
4.2 Progettazione MR1 .....	21

# 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'iniziare, 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.

## 1.1 Pianificazione Training Maven Plugin

Il gruppo dedicherà dei giorni per approfondire il tool di building Maven e come costruire un Maven plugin. Per ridurre al minimo indispensabile i tempi di training, oltre che ad usare le documentazioni ufficiali di Maven, useremo anche video tutorial su Internet.

## 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 obiettivi 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 degli 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 Management

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, il documento di 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.

Il gruppo lavorerà in modalità isolata usando come IDE di sviluppo IntelliJ 2019.3 Ultimate edition. Ciò consentirà di lavorare indipendentemente e parallelamente sulle due change request, qualora dopo l'intervento di analisi sia giudicato possibile.

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 – versione 3.6.1.

## 2. Reverse engineering report:

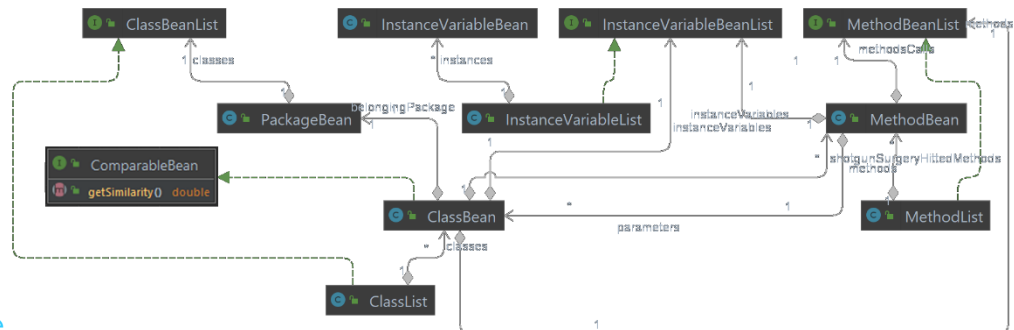
Attraverso la program comprehension e l'aiuto dei tool di IntelliJ per la generazione dei diagrammi UML automatici abbiamo rilevato le seguenti caratteristiche per i package:

### 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.

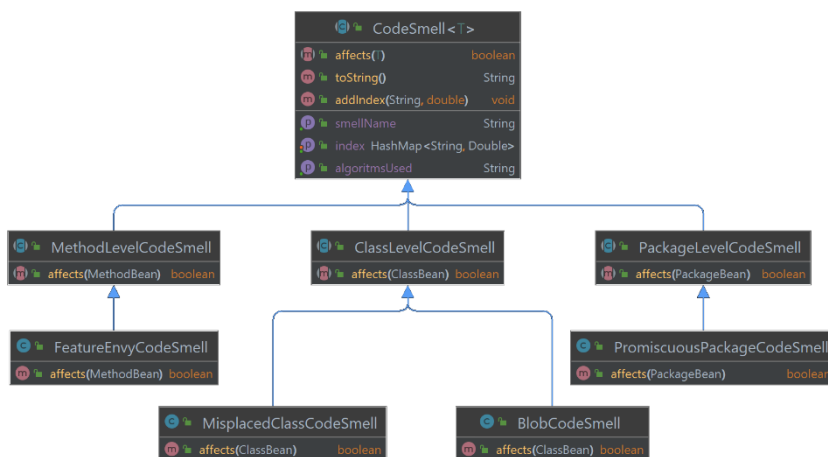


## Package analyze

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 metodo, classe o package (ricordiamo che questi elementi sono modellati dai corrispettivi bean). 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:

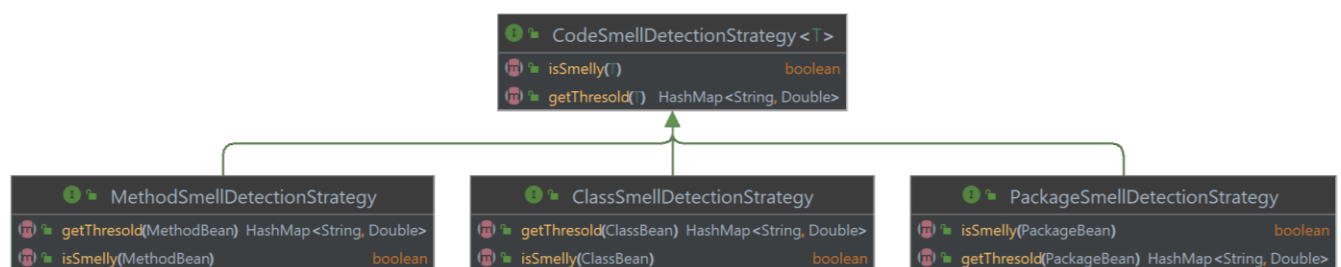
- Livello di metodo
- Livello di classe
- Livello di package

Il metodo astratto affects(T) è responsabile di iniziare l'analisi della componente generica 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 dagli smell concreti identificabili dal sistema: Feature Envy, Misplaced Class, Blob e Promiscuous Package.

## Package code smell detection

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

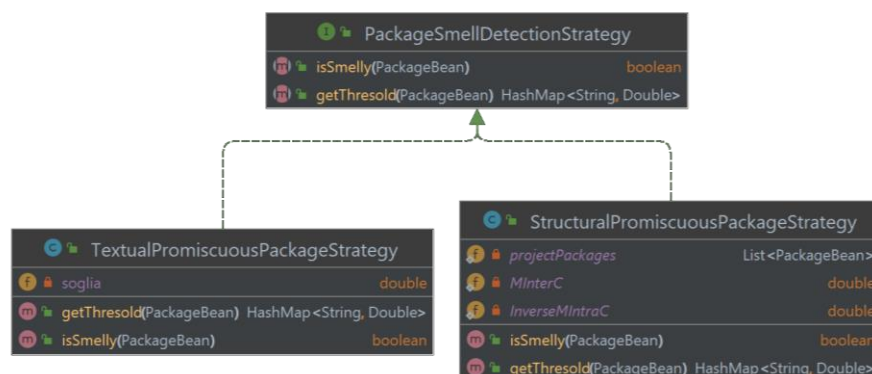


`CodeSmellDetectionStrategy<T>` è un'interfaccia che presenta due metodi:

- `isSmelly(T)` chiede se la componente T generica è o meno smelly
- `getThresold(T)` restituisce una lista di misure estratte dalla 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 rappresentano una soglia. **I valori di soglia vengono usati nel metodo isSmelly(T) e se nel calcolo questi vengono superati, 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 vedere, la gerarchia dei code smell modella il contesto. Mentre la gerarchia delle strategie è rappresentata dalla gerarchia di CodeSmellDetectionStrategy. L'algoritmo che applica la strategia è il metodo isSmelly, che come già menzionato stabilisce attraverso l'uso della strategia se la componente passata è o meno affetta dallo smell.

È importante notare che non è possibile passare una qualunque strategia ad uno smell. Alle classi che modellano un generico code smell a livello di classe, metodo e package viene specificato che può essere passata solo una strategia del medesimo livello. Le concrete implementazioni di questi code smell a loro volta accettano una qualsiasi strategia di identificazione che sia per quella componente. Ciò significa che attraverso l'uso di questo pattern possono verificarsi delle condizioni erronee come passare ad un blob smell un algoritmo strutturale per l'identificazione di un misplaced smell.

## Utilities

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 promiscuous package.

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

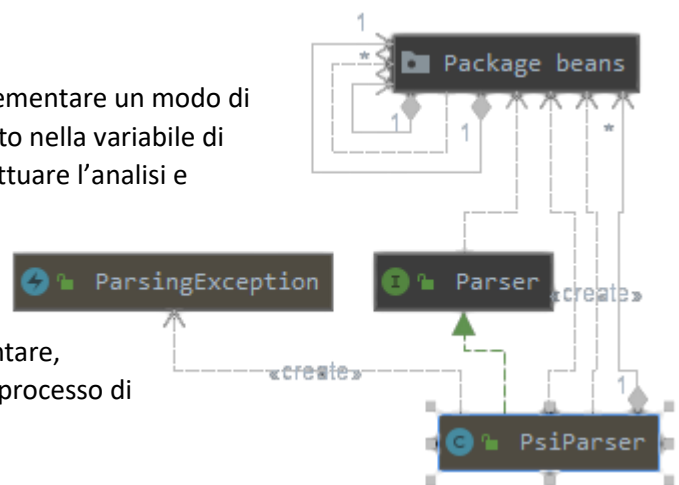
-CKMetrics: E' una classe che racchiude numerosi metodi statici per misurare tramite metriche strutturali determinati aspetti dei bean che gli vengono passati.

-TopicExtractor: 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

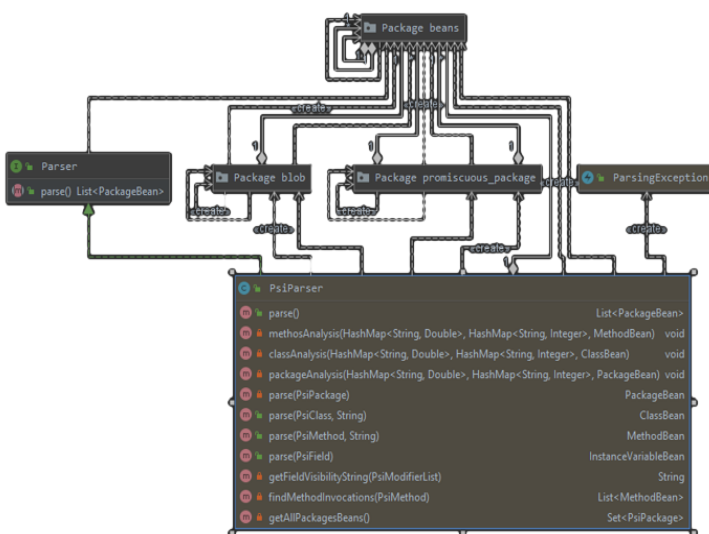
L'interfaccia Parser assegna la responsabilità di implementare un modo di effettuare il parsing del progetto di IntelliJ, incapsulato nella variabile di tipo Project, in classi bean usate dal sistema per effettuare l'analisi e rilevazione di code smell. PsiParser implementa tale comportamento. Nel caso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

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 della sezione, 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 istanziati 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 tale strategia.

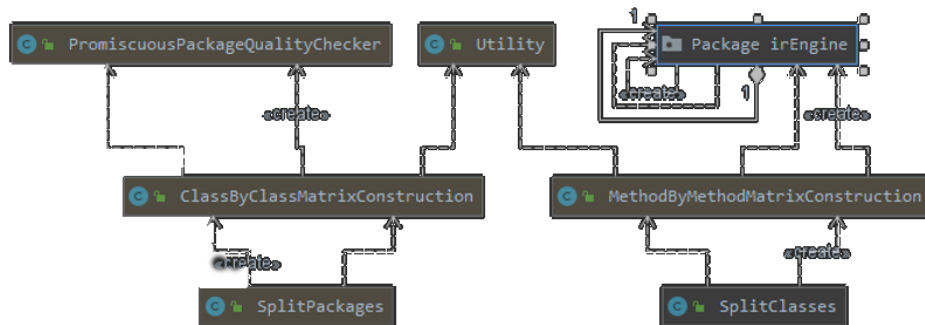
Da un'attenta analisi di program comprehension in questi metodi è possibile già stabilire quali potrebbero essere delle classi da includere nell'operazione di refactoring (MR1), dato che vi sono alcune strategie che sono state commentate e mai utilizzate nel programma. Queste coincidono con quelle che non sono state menzionate nel paper di ricerca.



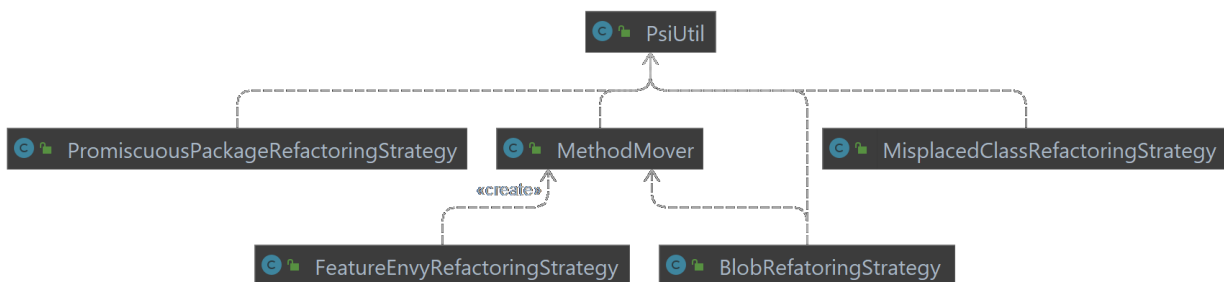


## Refactoring Package:

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.

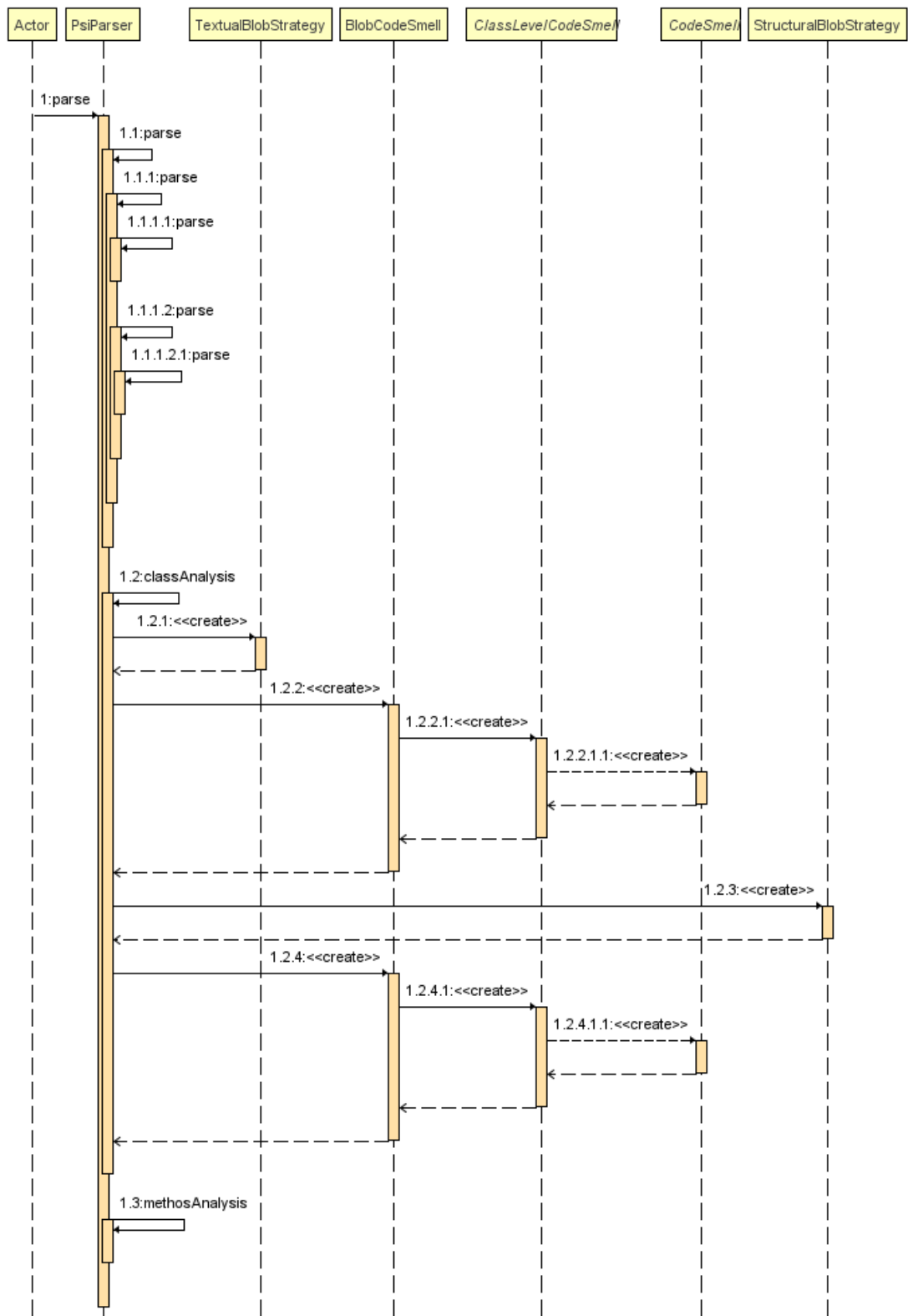


Ne consegue che la parte di refactoring è fortemente accoppiata con le API di IntelliJ.

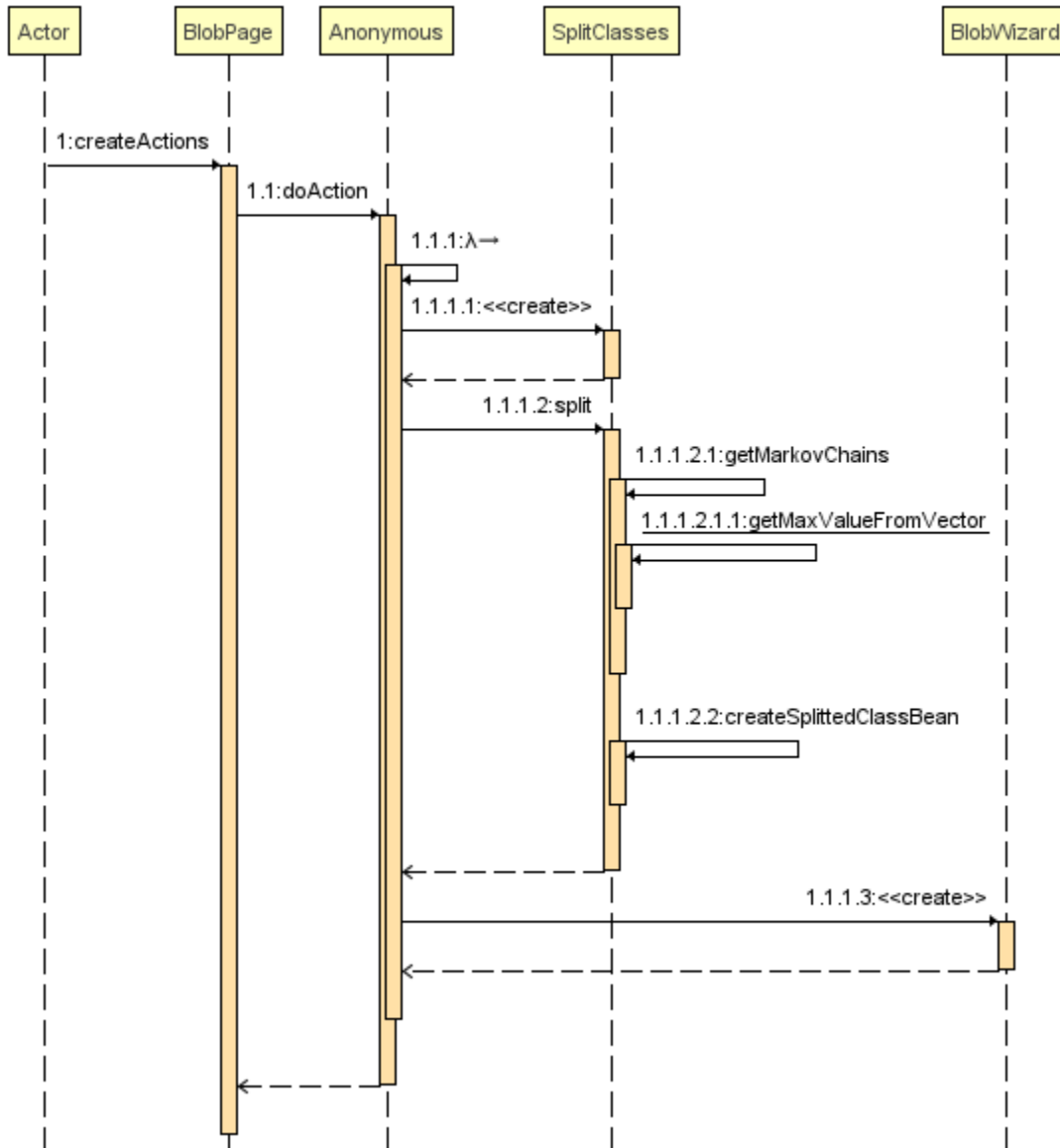
## 2.2 Logica d'interazione con la gui

### Package gui e actions

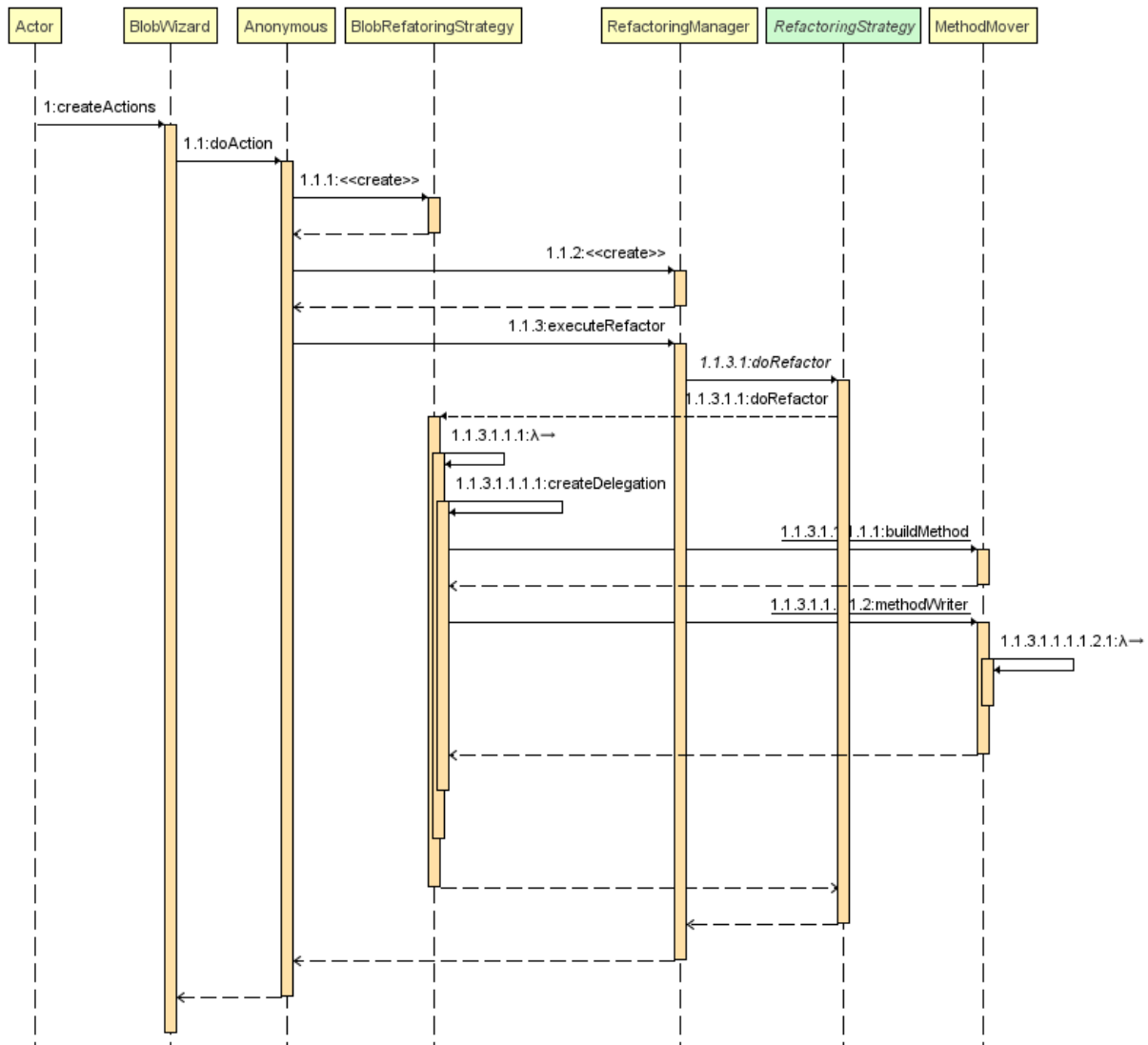
Il package actions contiene ogni ActionListener in ascolto di eventi che richi amino l'uso del plugin. Gli eventi BeginAction e CommitAction vengono lanciati per avviare il tool – il primo dal menu tools e il secondo quando avviene un commit. In entrambi i casi l'applicazione istanzia la main window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



Ogni smell ha una sua 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 aggiuntivo di splitting visto prima.

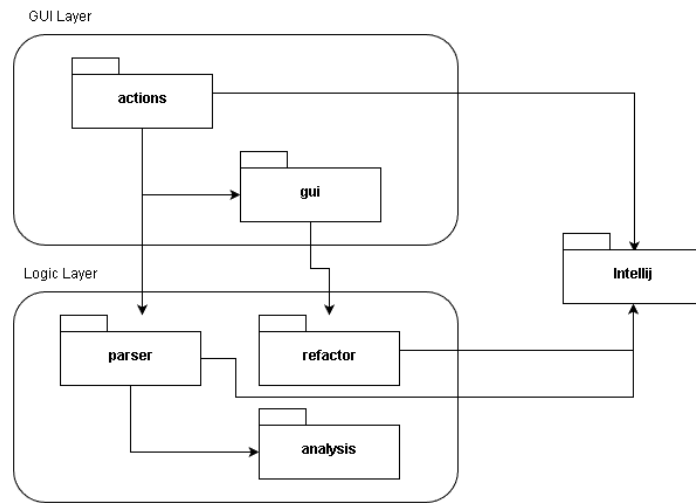


Dopodiché viene istanziato il wizard. Alla pressione del bottone “Refactoring” avvia il processo di refactoring per risolvere lo smell con la relativa strategia. Nel caso di blob e promiscuos smell, l’algoritmo di refactoring usa anche il prodotto degli algoritmi di splitting passati dalla view page.

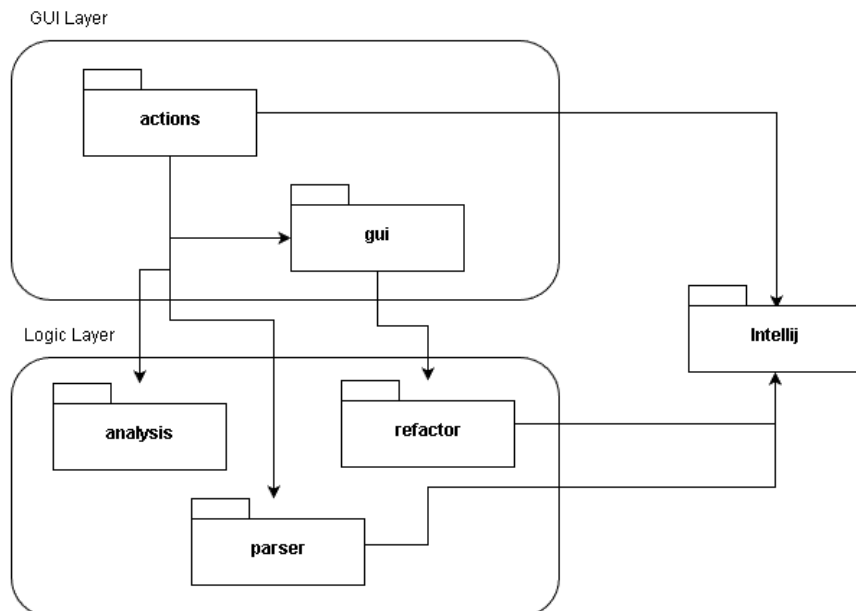


Infine, l’action listener ConfigureAction permette di istanziare una view per configurare i thresholds.

## 2.3 Class Diagram



Nel class diagram non sono inclusi i beans del sistema perché in qualche modo ogni componente è in dipendenza con i beans. Prima di andare avanti, illustriamo un passaggio di alterazione ritenuto necessario. Nel paragrafo dedicato al parser nella sezione 2.1 abbiamo visto che questa componente ha due responsabilità. Di norma, la responsabilità del parser è quella della sola “traduzione”. Dunque, si è pensato di isolare i metodi statici responsabili di avviare l’analisi statica e spostare quest’ultimi in un'altra classe. Il passo di alterazione è lecito in quanto non stiamo cambiando, aggiungendo o rimuovendo funzionalità al sistema, ma stiamo solo modificando la sua architettura. Il prodotto finale è questo:



## 2.4 Requisiti Estratti

A seguito delle 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 realmente usata. Per quanto riguarda MR2 sono stati identificati vari approcci.

### 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 identificate come 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

È importante notare che è necessario modificare le classi della GUI sopracitate esclusivamente per poter buildare il progetto nell’ambito della MR1, dato che in MR2, poichè riguarda la migrazione del tool a Maven, cASpER sarà sprovvisto di interfacce grafiche e diventerà interagibile da linea di comando.

Si è pensato anche di effettuare un’operazione di refactoring riguardante il parser: il metodo parse di quest’ultimo si occupa sia di effettuare il parsing che di lanciare le attività di analisi sul progetto. Si può incapsulare questa seconda operazione in un metodo apposito e diverso da quello adibito al parsing, si sceglie però di farlo durante la MR2 dato che questa coinvolge comunque ampie modifiche al parser.

### 3.2 Analysis Operation: MR2\_Migrazione\_Maven

Per questa MR sono stati considerati vari approcci che comportano l'eliminazione o meno delle dipendenze alle API di IntelliJ. Come già visto nel class diagram estratto, le dipendenze alle API di IntelliJ sono particolarmente critiche per il parsing del progetto software e nella fase di refactoring.

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

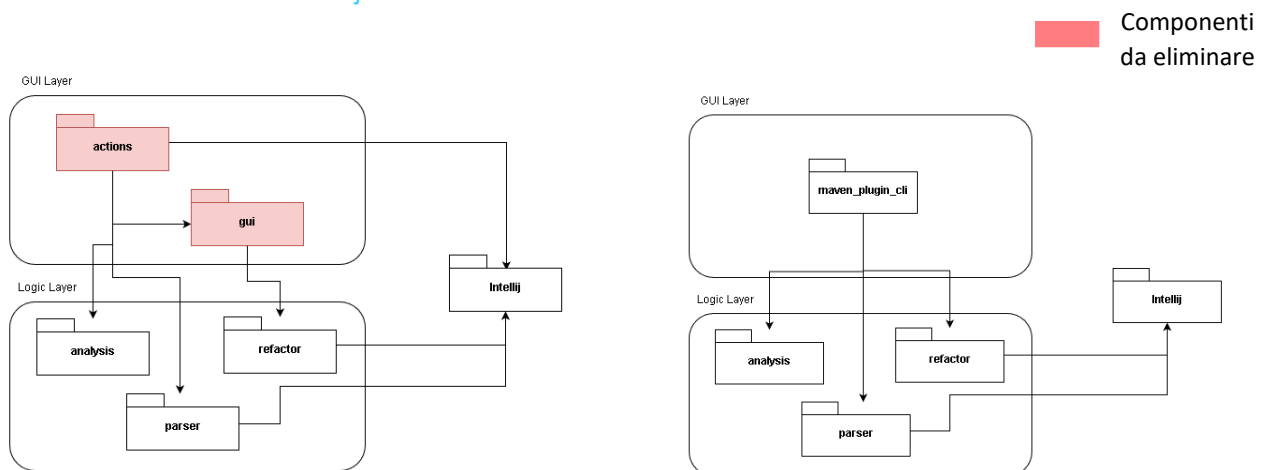
- Usare una third party library o API che permetta di fare le stesse cose che facevano le API di IntelliJ
- Manualmente implementare il comportamento delle API di IntelliJ

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 decida di mantenere le API di IntelliJ e il caso in cui decidiamo invece 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 R20. 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 user del plugin attraverso la realizzazione di un "hook" nel progetto. Ciò rende 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.

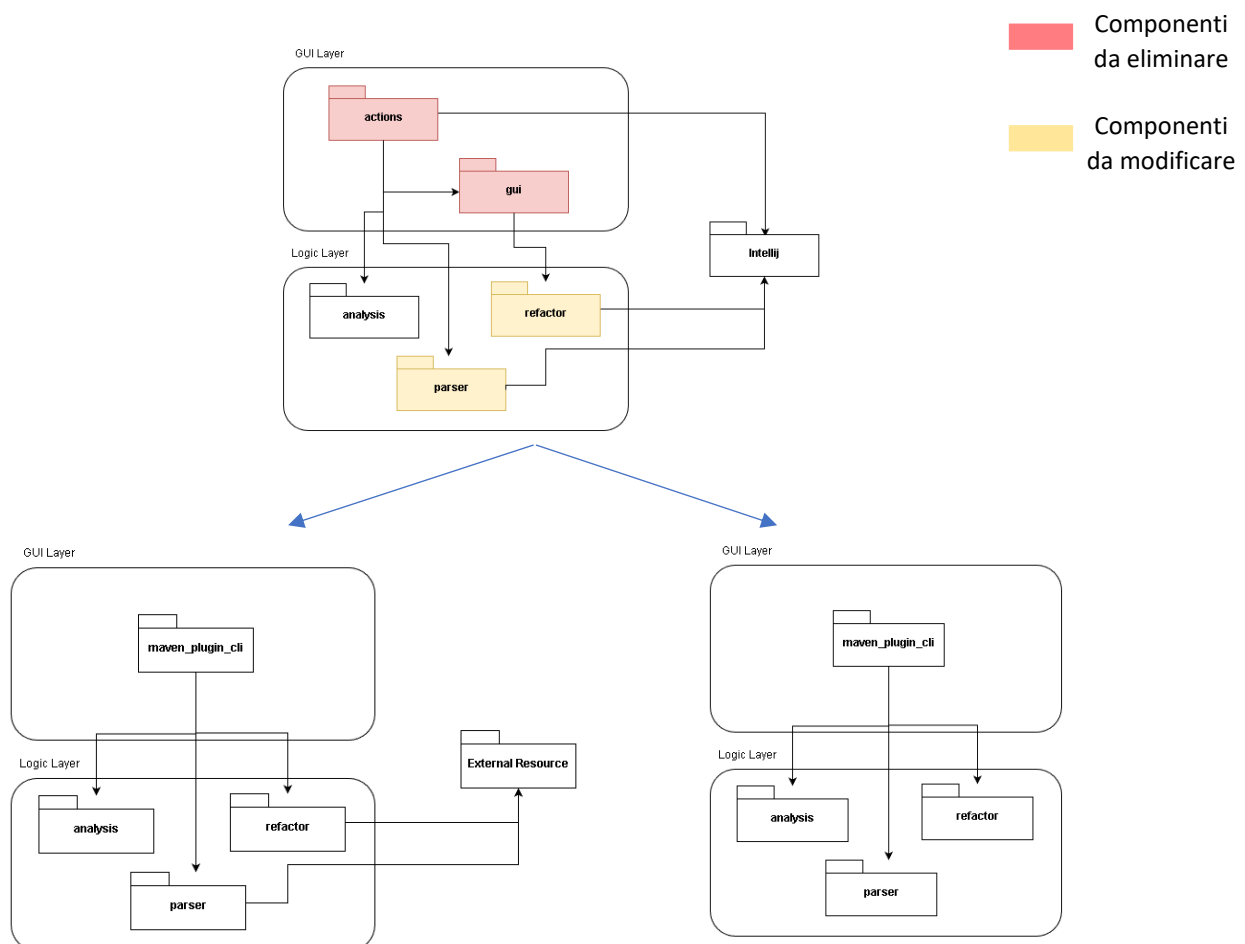


Dato che in questo approccio la logica da implementare è minima, si potrebbe pensare di effettuare una reingegnerizzazione completa. Tuttavia, come anche specificato prima, affinché possa funzionare è necessario trovare le dipendenze necessarie affinché il tool operi come da specificato.

Il rischio di non trovare queste dipendenze è alto. Pure essendoci delle dipendenze disponibili su Maven Central, bisogna capire quali sono quelle essenziali. Inoltre, bisogna vedere se funzionano. IntelliJ non ha nessun supporto al development dei suoi plugin su Maven. Ciò rende il tutto più problematico.

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
- Implementare manualmente 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 precisamente 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. Tenendo in considerazione non solo i tempi di development, testing e di documentazione richiesta diventa necessario restringere lo scope dell'attività di reingegnerizzazione. Di conseguenza verrà applicata una strategia incrementale all'operazione di reengineering.

La parte del refactoring non dipende fisicamente dalla parte di analisi, ma senza di essa non si sa come dover condurre il refactoring. Dunque, ha senso che in questo primo incremento, i nostri sforzi siano concentrati sul reingegnerizzare la parte di analisi. Il set dei requisiti implementati nel primo incremento saranno **R1-R8** includendo anche **R21**.

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 siano elevati. 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 astrazione che potrebbe offrire una third party library, i tempi di development potrebbero ridursi.

Rischi	Benefici
<ul style="list-style-type: none"><li>- Tempi aggiuntivi per il training</li><li>- Adattamenti possono essere complessi</li></ul>	<ul style="list-style-type: none"><li>- Astrazioni maggiori</li></ul>

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
<ul style="list-style-type: none"><li>- Adattamenti possono essere molto complessi</li><li>- Tempi di development più lunghi</li></ul>	<ul style="list-style-type: none"><li>- Portabilità maggiore della logica applicativa</li></ul>

## 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 sono estratti 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 classi interne; nel refactoring sarebbe necessario lavorare con API

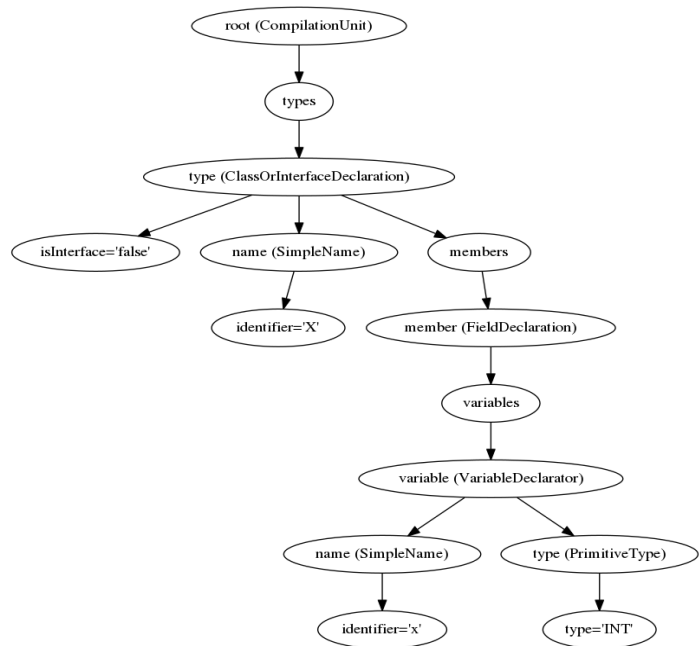
di basso livello come `java.io.File` per apportare le modifiche. Il secondo criterio è stato favorito anche perché è stato trovato molto rapidamente una library che potesse rimpiazzare il ruolo delle API di IntelliJ.

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

Come ci si aspetterebbe da un parser, viene preso in input un file java e restituisce una versione del file java basata su AST nota come Compilation Unit. È possibile tramite alcuni metodi estrarre i nodi che sono di interesse per la costruzione dei bean dell'applicazione, almeno per classi, metodi e variabili d'istanza.

Un altro punto di forza di Javaparser è la possibilità di poter modificare dinamicamente l'AST in modo agevole tramite la sua API. Seppure le modifiche all'AST non si propaghino sul file parsato, comunque è possibile estrarre rapidamente il contenuto in formato testuale e sovrascrivere il file esistente.

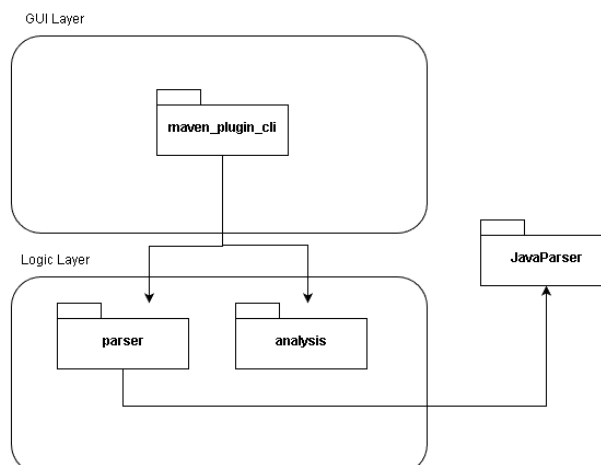
Ciò rende possibile il suo utilizzo anche per la parte di refactoring, nel secondo incremento.



## 4 Progettazione

Qui riportiamo la progettazione delle modifiche. Come si è potuto evincere nella fase di analisi vi sono delle dipendenze tra le due modifiche da effettuare. In particolare, il refactoring (MR1) coinvolge modifiche nelle UI, nel modulo di analisi e refactoring dell'originale. A sopravvivere dopo l'intervento di reingegnerizzazione (MR2) sarà solo il modulo di analisi. Di seguito riportiamo le scelte progettuali per le due MR.

### 4.1 Progettazione MR2



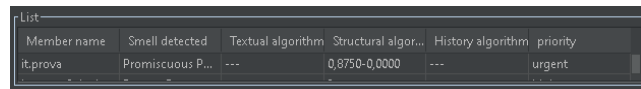
Come è stato illustrato nella fase di analisi, l'intervento, seguendo il criterio selezionato, richiede la completa reingegnerizzazione della parte interattiva del tool. In particolare, questa modifica porta il tool ad essere un tool a linea di comando. Inoltre diventa necessario modificare il parser per accomodare l'uso della libreria JavaParser.

## Progettazione maven\_plugin\_cli

Per realizzare un plugin Maven è necessario realizzare una classe Mojo (Maven plain Old Java Object). Tale classe contiene il metodo execute() il quale permette a Maven di eseguire il plugin. Per individuare il plugin da eseguire, Maven ha inoltre bisogno di sapere il nome del goal che il plugin implementa e in quale fase del ciclo di vita di Maven è necessario eseguirlo. Noi abbiamo scelto come nome del goal "goaldetection" e come fase del ciclo di vita in cui avviare il tool quella di compilazione.

cASpER Maven deve restituire in output i risultati dell'analisi statica come avviene nel tool originale. Il tool li comunica istanziando una tabella nella

sezione "List" della view CheckProjectView.java. Ogni riga contiene uno smell e ci dice da dove esso proviene, la priorità del refactoring e il valore calcolato con l'algoritmo strutturale e testuale. Se clicchiamo su una delle righe, ci viene visualizzato il contenuto testuale della componente affetta dallo smell.



Member name	Smell detected	Textual algorithm	Structural algorithm	History algorithm	priority
it.prova	Promiscuous P...	---	0.8750-0.0000	---	urgent

Inoltre, l'utente ha la facoltà di scegliere di non visualizzare delle tipologie di smell deselectando la spunta del corrispettivo checkbox nella view.

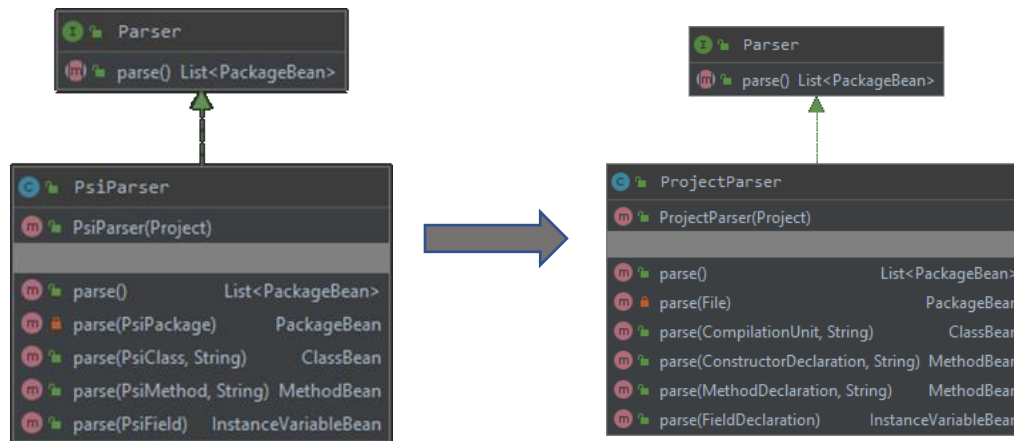
Tutte queste funzionalità a livello di presentazione devono essere in un qualche modo replicate dal nostro tool. Si è deciso che per default l'applicazione restituisce il numero di smell di determinate tipologie identificate e una tabella equivalente a quella prodotta da cASpER su linea di comando. Il comportamento del tool può essere modificato in base all'uso di particolari parametri modificabili per linea di comando. Questi saranno:

- display [m][b][p][f]: Il flag display prende in input le variabili m (Misplaced class smell), b (Blob smell), p (Promiscuous smell), f (Feature Envy) per indicare quale tipologia di smell va considerato nel processo di analisi statica. Se non si specifica alcuna tipologia di smell oppure viene inserito qualsiasi altro input, il plugin restituirà eccezione.
- dump [filename]: il flag permette di esportare le informazioni prodotte per default su linea di comando in un file. Per default, se il filename non è specificato allora verrà prodotto un file col nome "log.txt".
- dump-verbose [filename]: il flag permette di esportare le informazioni prodotte per default su linea di comando in un file. Inoltre, viene esportato anche il text content di ciascun elemento trovato. Per default, se il filename non è specificato allora verrà prodotto un file col nome "log.txt".

Nel caso della sua versione a plugin, i threshold venivano persistiti in un file in modo tale da poter consentire all'utente di modificarli (R21). Per questo fine, si è pensato di tramutare i threshold in parametri del plugin. In questo modo, sarà possibile modificarli direttamente dal pom.xml.

## Progettazione parser

Per poter operare in questo nuovo contesto, il parser richiede delle modifiche nella sua interfaccia. Nei class diagram sottostanti rappresentiamo le modifiche ad alto livello necessarie per conformare il parser al nuovo contesto.



Non sono indicati nei diagrammi UML i metodi privati della classe. Anche questi subiranno modifiche, ma in questo momento non si è in grado di prevedere come. In conclusione, quando bisogna realizzare il modulo “maven\_plugin\_cli” si deve stare attenti ad istanziare correttamente il parser. Per il resto il parser si dovrebbe comportare in modo analogo alla versione originale di cASpER.

## 4.2 Progettazione MR1

A seguito di MR2, le modifiche preventivate in fase di analisi (sezione 3) sono confinate al modulo di analisi. Quindi delle classi inizialmente identificate, restano solo:

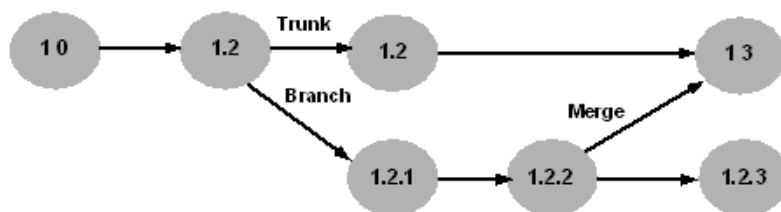
- DivergentChangeCodeSmell
- ParallelInheritanceCodeSmell
- ShotgunSurgeryCodeSmell
- HistoryBlobStrategy
- HistoryDivergentChangeStrategy
- HistoryFeatureEnvyStrategy
- HistoryParallelInheritanceStrategy
- HistoryShotgunStrategy
- AnalyzerThread
- HistoryAnalysisStartup**
- PythonExeSingleton
- RepositorySingleton

L’intervento di refactoring potrebbe essere eseguito sia prima che dopo l’intervento di reingegnerizzazione. Nella tabella seguente riportiamo i vantaggi e svantaggi nei due casi:

Prima		Dopo	
Pro	Contro	Pro	Contro
Ridotti i tempi per l'esecuzione dell'intervento e testing	Nel caso in cui il testing rileva un malfunzionamento è più difficile stabilire a cosa è dovuto	Nel caso in cui il testing rileva un malfunzionamento è più facile capire a cosa è dovuto	Maggiori tempi d'esecuzione dell'intervento e testing (si deve ritestare a valle del refactoring)

Ricordiamo che le parti da re-fattorizzare sono in gran parte code smell e le loro relative strategie. Grazie all'applicazione dello strategy pattern, rimuovere un code smell con le sue relative strategie (o delle strategie in particolare) è piuttosto semplice e non impatta gli altri code smell, i loro algoritmi di detection e il loro uso nella fase di analisi statica. Dunque, la rimozione di un code smell non dovrebbe originare alcun malfunzionamento nel modulo di analisi, ma in chi usa il modulo di analisi. Questo è vero anche per le modifiche da condurre nell'HistoryAnalysisStartup: infatti i metodi per la determinazione della priorità dei code smells, definiti in HistoryAnalysisStartup, erano richiamati da CheckProjectPage e pertanto saranno estratti in una nuova classe affinché HistoryAnalysisStartUp possa essere cancellata e tali metodi possano essere invocati da una classe di maven\_plugin\_cli.

Per questi motivi, si è deciso di condurre MR1 prima ed insieme a MR2.



La baseline di partenza viene realizzata mediante un fork dal progetto principale del sesa lab di Fisciano. Dopodiché verrà realizzato un nuovo branch per effettuare le modifiche. La prima azione da eseguire alla creazione del nuovo branch è l'introduzione di una cartella nominata "casper-maven-plugin". In questa cartella trasferiremo il progetto da un ambiente di building Gradle ad uno basato su Maven. La cartella contenente il codice per il plugin di IntelliJ rimarrà intatta in quanto è necessario nella fase di testing.

Introduciamo una cartella contenente il progetto per il Maven plugin nominata "casper-maven-plugin" nel quale verranno inizialmente copiati i moduli che entreranno a far parte della build finale del progetto.

## 5. Sviluppi futuri

Dato che in questo progetto non verrà effettuato l'incremento per includere la parte di refactoring, interventi futuri possono includere l'integrazione del suddetto modulo. Altre possibilità includono l'integrazione di altre strategie di detection textual, structural o hist based o di altri code smell in generale.