**Il tool SURFACE**

Sommario

[1. Descrizione di massima del tool 1](#_Toc106721132)

[1.1. Metriche Class-level 2](#_Toc106721133)

[1.2. Metriche Project-level 2](#_Toc106721134)

[2. Code Comprehension 3](#_Toc106721135)

[3. Change Requests 8](#_Toc106721136)

[ CR1 8](#_Toc106721137)

[ CR2 8](#_Toc106721138)

[ CR3 9](#_Toc106721139)

[ CR4 9](#_Toc106721140)

[ CR5 9](#_Toc106721141)

[3.1. CR1: Idee per l’implementazione 10](#_Toc106721142)

[3.2. CR2: Idee per l’implementazione 10](#_Toc106721143)

[3.3. CR3: Idee per l’implementazione 10](#_Toc106721144)

[3.4. CR4: Idee per l’implementazione 11](#_Toc106721145)

[3.5. CR5: Idee per l’implementazione 11](#_Toc106721146)

[4. Impact Analysis 11](#_Toc106721147)

[ CR1 - Impact Sets 11](#_Toc106721148)

[ CR2 - Impact Sets 12](#_Toc106721149)

[ CR3 - Impact Sets 12](#_Toc106721150)

[ CR4 - Impact Sets 12](#_Toc106721151)

[ CR5 - Impact Sets 12](#_Toc106721152)

# Descrizione di massima del tool

SURFACE è un tool scritto in Java sviluppato dal dott. Emanuele Iannone. Tale tool consente, mediante un’interazione da linea di comando, di calcolare metriche di sicurezza per codice Object-Oriented fornendo una panoramica del profilo di sicurezza delle classi presenti in un progetto. Può calcolare sia metriche relative a singole classi che ad un intero progetto, che a sua volta può essere sia locale (specificato da un path locale) che remoto (specificato dal link del repository del progetto e dall’hash dell’ultimo commit). I risultati del calcolo delle metriche di sicurezza possono essere mostrati su standard output ed esportati in CSV. Di seguito vengono elencate le metriche attualmente implementate all’interno del progetto, accompagnate da una breve descrizione:

## Metriche Class-level

* **CA (Classified Attributes)** - numero di variabili d’istanza (o attribuiti) che vengono identificati come “riservate” (dall’inglese, *classified*) mediante delle euristiche. Nello specifico, viene verificato il match dei nomi delle variabili con un insieme di parole che suggeriscono che quella variabile contenga un valore riservato;
* **CM (Classified Methods)** - numero di metodi che vengono identificati come “riservati”. Un metodo viene identificato come riservato se interagisce con almeno un *Classified Attribute;*
* **CIVA (Classified Instance Variables Accessibility)** - rapporto tra il numero di *Classified Attributes* che non sono né privati né staticied il numero totale di *Classified Attributes;*
* **CCVA (Classified Class Variables Accessibility)** - rapporto tra il numero di *Classified Attributes* che non sono privati ma sono staticied il numero totale di *Classified Attributes;*
* **CMA (Classified Method Accessibility)** - rapporto tra il numero di *Classified Methods* che non sono privati sul totale dei *Classified Methods;*
* **RP (Reflection Package)** - valore booleano che indica se la classe in esame fa uso del pacchetto *java.reflection;*
* **CMR (Classified Methods Ratio)** - rapporto tra il numero di *Classified Methods* e numero di metodi totali;
* **CAI (Classified Attribute Interactions)** - rapporto tra la somma del numero di *Classified Attributes* di ogni *Classified Methods* che li usa ed il prodotto tra tutti gli usi di *Classified Methods* e *Classified Attributes.*

## Metriche Project-level

* **CC (Critical Classes)** - numero di classi critiche all’interno del progetto. Una classe si definisce critica se e solo se ha almeno un *Classified Attribute* oppure un *Classified Method*;
* **CCR (Critical Classes Ratio)** - rapporto tra il numero di *Critical Classes* ed il numero totale di classi all’interno del progetto;
* **SCCR (Serializable Critical Classes Ratio)** - rapporto tra il numero di *Critical Classes* che implementano l’interfaccia *Serializable* ed il numero di *Critical Classes*;
* **CCE (Critical Classes Extensibility)** - rapporto tra il numero di *Critical Classes* non final (da cui quindi altre classi possono ereditare) ed il numero totale di *Critical Classes*;
* **CME (Classified Methods Extensibility)** - rapporto tra il numero di *Classified Methods* di cui è possibile effettuare l’override ed il numero di *Classfied Methods*;
* **CSCR (Critical Superclasses Ratios)** - media dei rapporti tra numero di *Critical Superclasses* e tutte le loro superclassi.

# Code Comprehension

Questa sezione contiene un resoconto delle principali informazioni estratte dalla Code Comprehension effettuata al fine di svolgere correttamente l’Impact Analysis. Per ogni componente è stata fornita una descrizione ad alto livello, soffermandosi principalmente sulle funzionalità core e sulle relazioni più rilevanti con il resto del sistema.

* La main class è **CLIStarter**. Prende argomenti da linea di comando, li parsa mediante la classe *CLIParser* (in particolare mediante il metodo parse(args)). In caso di errore nel parsing lancia una ParseException. Infine, crea un'istanza di Surface sull'input parsato e la runna;
* La classe **CLIParser** presenta tre variabili statiche:
  + **DEFAULT\_METRICS**: definisce le metriche che vengono usate se l'utente non ne specifica (CA, CM)
  + **DEFAULT\_PROJECT**: definisce come directory di progetto, la current working directory da usare a meno che l'utente non ne specifichi una esplicitamente ("")
  + **DEFAULT\_EXPORT**: definisce il formato di output di default (".csv")
* La classe **CLIParser** presenta solo il metodo parse che si occupa di accedere ai 4 parametri passati da linea di comando e ne effettua il parsing eventualmente sostituendoli a quelli specificati di default dalle variabili statiche. Viene restituita un'istanza di SurfaceInput costruita a partire da:
  + codici delle metriche da considerare
  + path di un progetto remoto
  + path di un progetto locale
  + formato di export
* La classe **CLIOptions** è un'estensione della classe org.apache.commons.cli.Options. Questa classe presenta 4 variabili statiche che rappresentano i nomi delle opzioni e una variabile CLIOptions che rappresenta la singola istanza. Inoltre usa il Design Pattern Singleton, per cui possiede un costruttore privato e un metodo getInstance() che garantisce la creazione di una sola istanza della classe. Il costruttore definisce 4 istanze della classe org.apache.commons.cli.Option (metrics, remoteProjects, project, export) e dopo averle istanziate le aggiunge alla collezione di Options usando il metodo addOption della classe da cui eredita.
* La classe **SurfaceInput** contiene 4 variabili di istanza final che rappresentano i codici delle metriche, il path assoluto del progetto remoto, il path assoluto del progetto locale e il formato di export. Questa classe rappresenta il concetto di input per il tool wrappando i dati necessari. Presenta solo un costruttore e dei metodi getter (non ci sono setter in quanto le variabili di istanza sono final).
* La classe **Surface** funge da runner per computare le metriche. Ha una variabile di istanza di tipo SurfaceInput che contiene i parametri passati dall'utente. Possiede un costruttore che si occupa solo di settare la variabile di istanza SurfaceInput. Possiede un metodo run che serve a stabilire quale runner utilizzare (runner per progetto locale o runner per progetto remoto). Il controllo viene effettuato in base alla presenza o meno del parametro "remoteProjectAbsolutePath".
* La classe **ProjectsControl** è astratta e si occupa di effettuare l'analisi del progetto. Possiede una variabile di istanza che rappresenta i codici delle metriche con un relativo metodo getter. Possiede un singolo metodo chiamato processProject che non può essere chiamato direttamente su questa classe (in quanto astratta) ma può essere chiamata sulle classi che la implementano. Questo metodo istanzia un ProjectAnalyzer che analizza il progetto, ottenendo i risultati sulla base dei quali vengono computate le metriche di sicurezza.
* La classe **RemoteSnapshotProjectsControl** estende la classe ProjectsControl. Possiede diverse variabili di istanza:
  + **remoteProjectsAbsolutePath**: path del file CSV contenente le informazioni sui progetti remoti da analizzare
  + **BASE\_DIR**: rappresenta la directory di base per l'export
  + **exportFormat**: rappresenta il formato di esportazione
* La classe **RemoteSnapshotProjectsControl** inoltre possiede un costruttore che chiama quello della superclasse (ProjectsControl) settando i codici delle metriche e setta localmente il path del progetto remoto e il formato di esportazione. Possiede un metodo run che effettua il parsing del file .csv contenente gli URI dei progetti remoti, li clona facendo riferimento ad un determinato commit specificato nel file .csv e chiama il metodo processProject della superclasse, esportando successivamente i risultati
* La classe **SingleLocalProjectControl** estende la classe ProjectsControl. Possiede una variabile che rappresenta il path del progetto locale. Ha un metodo run che si occupa di chiamare il metodo processProject della superclasse. PERPLESSITA': IL METODO NON ESPORTA I RISULTATI IN NESSUN MODO.
* La classe **CSVSnapshotsImporter** implementa l'interfaccia SnapshotImporter si occupa di parsare il file .csv che descrive i progetti remoti. Nello specifico fornisce un solo metodo chiamato extractSnapshots che prende in input il path di un file e restituisce i suoi contenuti parsati referenziando i campi "projectID", "github" e "commitHash".
* L'interfaccia **SnapshotImporter** dichiara il metodo extractSnapshots.
* La classe **ProjectAnalyzer** permette di costruire un modello strutturato delle classi che compongono un progetto mediante il metodo analyze che pone il risultato dell'analisi di ogni classe all'interno di un oggetto istanza di ClassifiedAnalyzerResults. Tutti i ClassifiedAnalyzerResults prodotti vengono inseriti all'interno di una collezione.
* La classe **ClassifiedAnalyzerResults** conserva i risultati dell'analisi di una classe tenendo traccia di dichiarazioni di metodi, di attributi (variabili di istanza) e ha un flag che indica se la classe faccia o meno uso della reflection. Contiene dei getter e dei setter che consentono di accedere e di modificare i metadati della classe analizzata.
* La classe **ProjectAnalyzerResults** contiene una descrizione della root del progetto con i relativi risultati dell'analisi delle classi rappresentate mediante un Set di ClassifiedAnalyzerResults. Fornisce un metodo per ottenere i risultati di una specifica classe.
* La classe **ProjectMetricsResults** contiene una descrizione della root del progetto, un insieme di risultati dell'estrazione delle metriche delle classi del progetto e una lista di metriche considerate a livello di progetto. Espone solo metodi getter e setter e dei metodi per fare add di metriche di classe e di progetto.
* La classe **ClassMetricsResults** contiene i risultati delle metriche della classe e la lista delle metriche riferite a quella classe. Espone solo dei metodi getter e setter.
* La classe **MetricValue** è astratta ed è estesa dalla classe BooleanMetricValue e NumericMetricValue che a sua volta è estesa da IntMetricValue e DoubleMetricValue. Rappresenta un contenitore per una metrica contenendone il nome, il codice ed il valore ad essa associati.
* La classe astratta **NumericMetricValue** rappresenta una metrica di tipo numerico. Possiede solo un costruttore e non ha variabili di istanza.
* La classe **BooleanMetricValue** è una classe che rappresenta una metrica di tipo booleano. Possiede solo un costruttore e non ha variabili di istanza.
* La classe **IntMetricValue** estende la classe NumericMetricValue e rappresenta una metrica di tipo intero. Possiede solo un costruttore e non ha variabili di istanza.
* La classe **DoubleMetricValue** estende la classe NumericMetricValue e rappresenta una metrica di tipo double. Possiede solo un costruttore e non ha variabili di istanza.
* La classe **ProjectMetric** è una classe astratta che possiede un metodo chiamato compute che prende un ProjectAnalyzerResults come parametro e restituisce un tipo generico. Rappresenta il concetto di metrica riguardante un progetto.
* I package cc, cce, ccr, cme, cscr e sccr contengono ciascuno 3 classi:
  + Una classe astratta **[nome\_metrica]** che rappresenta la metrica (**CC**, **CCE**, **CCR**, **CME**, **CSCR**, **SCCR**)
  + Una classe **[nome\_metrica]Cached** che estende la classe [nome\_metrica] e mantiene il risultato di una specifica istanza di computazione di una metrica. Contiene un metodo compute che restituisce l'istanza precomputata se disponibile, altrimenti ne effettua il calcolo (**CCCached**, **CCECached**, **CCRCached**, **CMECached**, **CSCRCached**, **SCCRCached**).
  + Una classe **[nome\_metrica]Impl** che estende la classe [nome\_metrica] ed effettua il calcolo effettivo della metrica in questione (**CCImpl**, **CCEImpl**, **CCRImpl**, **CMEImpl**, **CSCRImpl**, **SCCRImpl**) mediante un metodo compute
* **ClassMetric** è una classe astratta che possiede un metodo chiamato compute che prende un ClassifiedAnalyzerResults come parametro e restituisce un tipo generico. Rappresenta il concetto di metrica riguardante una classe.
* I package ca, cai, ccva, civa, cm, cma, cmr e rp contengono ciascuno 3 classi:
  + Una classe astratta **[nome\_metrica]** che rappresenta la metrica (**CA**, **CAI**, **CCVA**, **CIVA**, **CM**, **CMA**, **CMR**, **RP**)
  + Una classe **[nome\_metrica]Cached** che estende la classe [nome\_metrica] e mantiene il risultato di una specifica istanza di computazione di una metrica. Contiene un metodo compute che restituisce l'istanza precomputata se disponibile, altrimenti ne effettua il calcolo (**CACached**, **CAICached**, **CCVACached**, **CIVACached**, **CMCached**, **CMACached**, **CMRCached**, **RPCached**).
  + Una classe **[nome\_metrica]Impl** che estende la classe [nome\_metrica] ed effettua il calcolo effettivo della metrica in questione (**CAImpl**, **CAIImpl**, **CCVAImpl**, **CIVAImpl**, **CMImpl**, **CMAImpl**, **CMRImpl**, **RPImpl**) mediante un metodo compute
* La classe **ClassMetricsFactory** implementa l'interfaccia MetricsFactory ed ha una variabile di istanza di tipo MetricsStructure che è una classe interna statica. Questa classe interna statica istanzia le classi che rappresentano le metriche e la relativa versione cached. Possiede dei metodi getter. La classe ClassMetricFactory si occupa di riconoscere in base all'array di stringhe ricevuto in input che rappresenta i codici delle metriche il tipo di metrica "desiderato" e mediante i getter della classe interna va a recuperarne delle istanze per poi restituirle.
* La classe **ProjectMetricsFactory** effettua le stesse operazioni di ClassMetricsFactory ma le applica alle metriche di progetto.
* L'interfaccia **MetricsFactory** è tipizzata con un tipo generico T che estende Metric (ClassMetrics o ProjectMetrics) e dichiara un metodo getMetrics che prende in input un array di Stringhe e restituisce una lista del tipo T con cui la classe è parametrizzata
* La classe **Snapshot** rappresenta la snapshot in un determinato momento di una repository. Possiede 3 variabili di istanza di tipo Stringa che sono: id del progetto, URI della repository e l'hash dell'ultimo commit. Possiede dei metodi getter.
* La classe **CSVSnapshotsExporter** implementa SnapshotExporter. Ha due costanti: una stringa ed un array di stringhe che contiene "projectId" e "commitHash". Possiede un metodo export che prende in input uno Snapshot, un ProjectMetricsResults e un array di stringhe e si occupa di esportare in formato CSV le metriche di progetto (nomi e valori) e lo Snapshot.
* La classe **NullSnapshotExporter** implementa SnapshotExporter e possiede un metodo export che restituisce true (non fa niente)
* La classe **ProjectMetricsResultsExporter** ha 3 variabili di istanza di tipo Snapshot, ProjectMetricsResults e un array di stringhe che contiene i codici delle metriche. Ha un costruttore ed un metodo exportAs che in base al parametro passatogli come stringa sceglie l'exporter e ne chiama il metodo export.
* L'interfaccia **SnapshotExporter** dichiara il metodo export
* La classe **SnapshotExporterFactory** ha un metodo getExporter che prende in input una stringa che identifica il tipo di exporter da usare ed in base alla quale restituisce l'istanza dell'exporter adatto. NOTA IMPORTANTE: ANCHE QUESTA CLASSE SI TROVERA' NEL CANDIDATE IMPACT SET.
* L'interfaccia **AnalyzerResults** è una semplice interfaccia marker.
* L'interfaccia **MetricsResults** è una semplice interfaccia marker.
* La classe **Metric** è astratta e contiene due stringhe: name e code che rappresentano nome e codice della metrica considerata. Possiede un metodo astratto compute e dei getter e setter.
* La classe **ProjectMetricsCalculator** ha un costruttore che prende in input un ProjectAnalyzerResult. Possiede un metodo calculate che prende in input i codici delle metriche (di classe e di progetto) da calcolare per poi calcolarle effettivamente e restituirle.
* Il package core.analysis contiene le classi **ClassAnalyzer**, **ClassifiedAnalyzer**, **ClassifiedPatterns** e **ProjectAnalyzer** che si occupano di estrarre l’Abstract Syntax Tree delle classi in esame.

# Change Requests

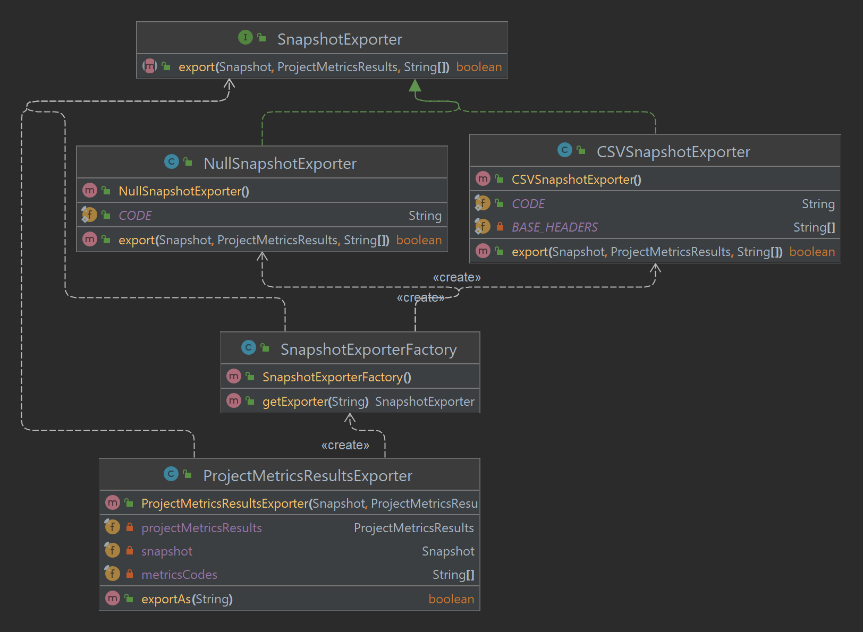
È stato deciso di apportare diverse modifiche al sistema attualmente disponibile. Nello specifico sono state definite 3 change requests:

* **CR1:** Disaccoppiamento della gestione dell’input dall’interfaccia a caratteri. Attualmente il tool prende parametri solo da linea di comando ed il sistema di gestione dell’input non prevede la possibilità di usare altri metodi. Infatti, vi è una classe CliStarter che istanzia staticamente un CliParser che si occupa del parse dei parametri. Si prevede di dare la possibilità di rendere modulare la gestione dell’input. La priorità di tale intervento di modifica è alta.

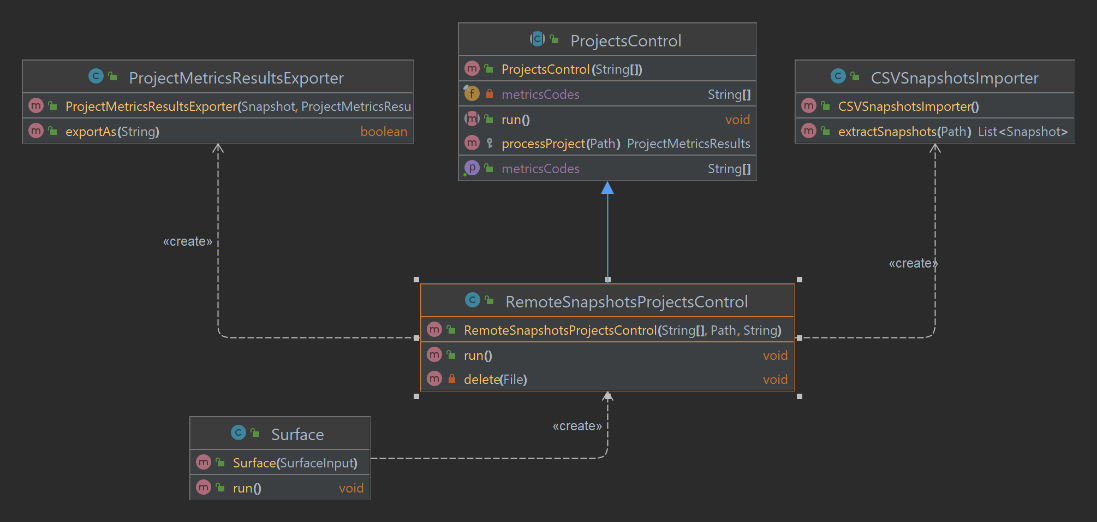
Immagine che contiene testo, targa, metallo, nero

Descrizione generata automaticamente

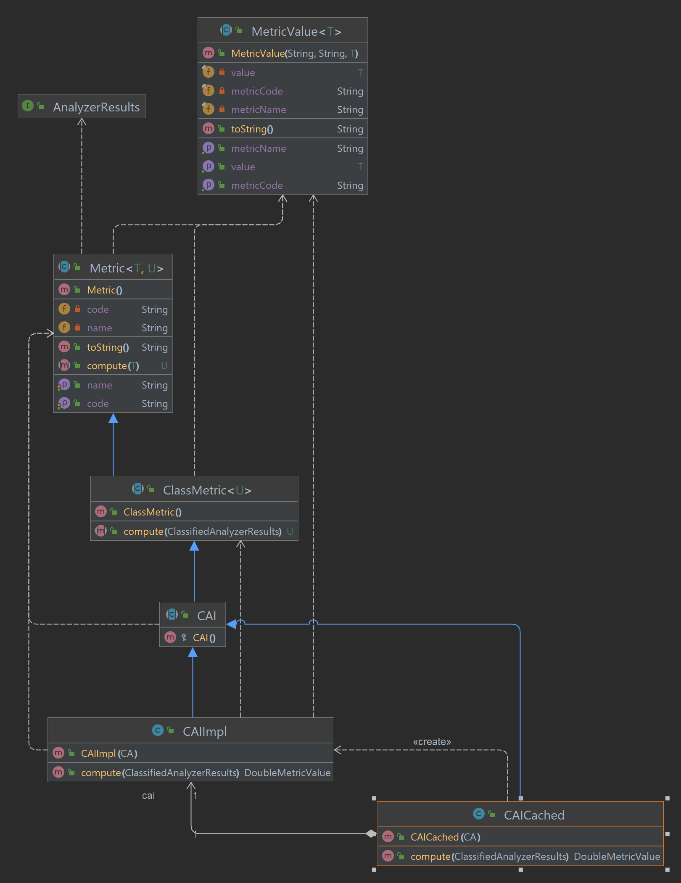
* **CR2:** Rendere la logica applicativa invocabile mediante servizi REST. Al momento l’interazione con l’applicazione avviene solo localmente installando il tool manualmente ed avviando il JAR. Si prevede di intervenire sulla logica di invocazione del tool per superare questo limite e renderlo invocabile anche in maniera remota. È stato deciso di limitare ai soli progetti remoti l’interazione con il tool mediante servizio REST.



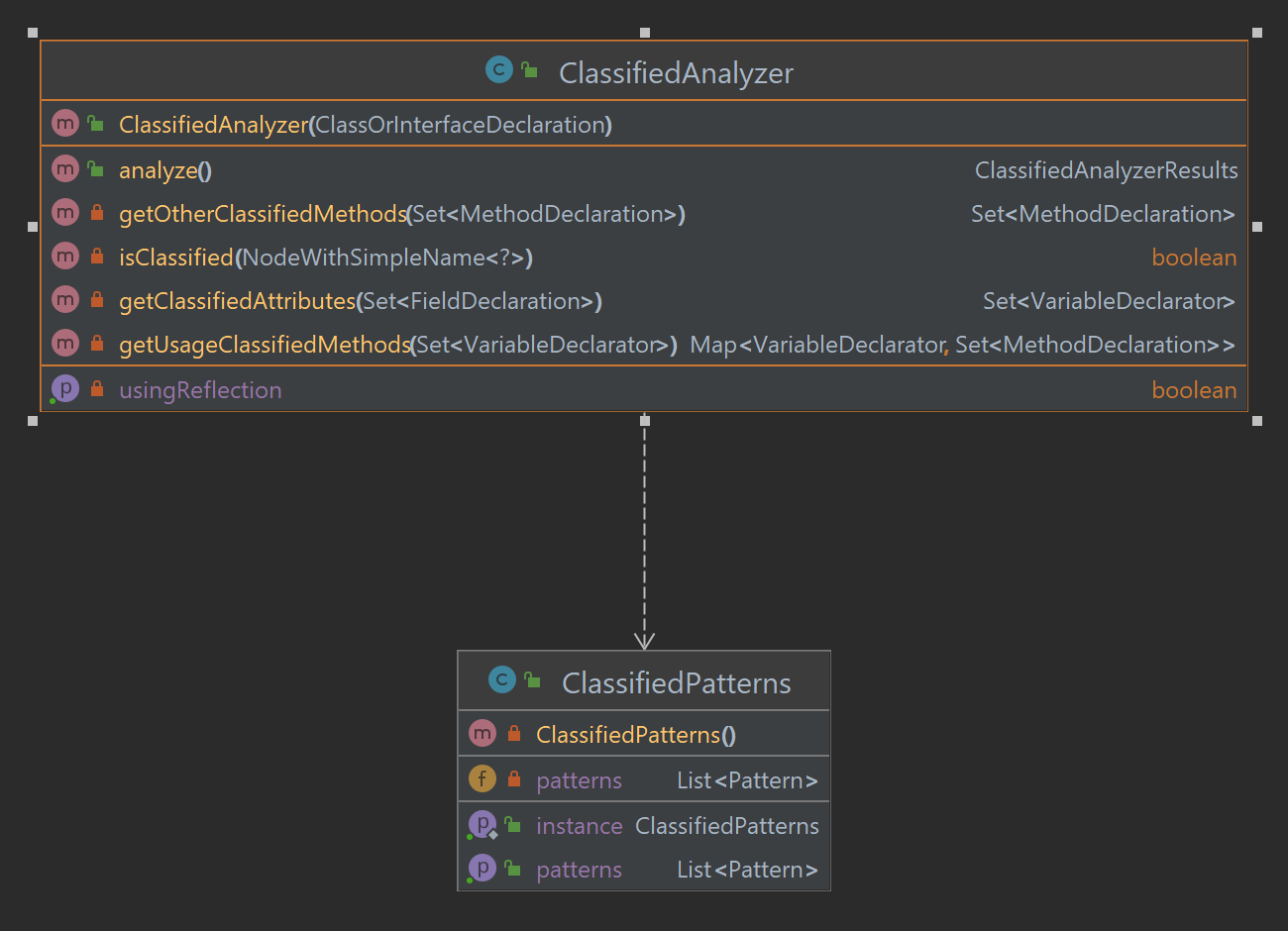
* **CR3:** Implementare un’interfaccia grafica che renda possibile l’utilizzo del tool mediante un Browser Web evitando di dover effettuare manualmente le chiamate agli endpoint REST.
* **CR4:** Correzione della gestione degli URI nella classe RemoteSnapshotsProjectsControl per rendere il tool compatibile anche con Microsoft Windows.



* **CR5:** Correzione dell’implementazione della metrica CAI per le classi, in modo da rendere il calcolo di questa con la formula corretta presente sui paper che trattano la suddetta metrica.



* **CR6:** Apertura del file "patterns" all'interno della classe ClassifiedPatterns non mediante il suo path assoluto ma tramite un metodo specifico che consente l'esecuzione del tool quando esportato come file JAR



3.1. CR1: Idee per l’implementazioneViene creata una classe astratta chiamata Parser che possiede un singolo metodo astratto parse e lo stato che prima apparteneva esclusivamente a CliParser. CliParser estenderà la classe astratta Parser. CliStarter non istanzierà un CliParser ma un Parser. Verrà usato il design pattern Abstract Factory e verrà introdotta una classe ParserFactory che conterrà un singolo metodo statico getParser che prende in input l’identificativo del parser che si vuole utilizzare e si occuperà di istanziarlo e restituirlo al chiamante. Questo parametro verrà passato alla classe main tramite argomenti CLI.

3.2. CR2: Idee per l’implementazioneViene creato un controller Spring che funge da endpoint REST. Da questo controller verrà richiamata la logica applicativa e l’input verrà fornito come JSON e restituirà, allo stesso modo, il risultato come JSON all’utente. Il JSON restituito viene prelevato da un file (in formato JSON) che verrà generato dal JSONExporter (classe che verrà creata durante l’implementazione di questa change request). Il controller dovrà fornire all’input di Surface “json” come exportFormat in maniera hard-coded. Non viene effettuata l’esportazione del file csv nel caso di progetti locali, ma ci si limita unicamente alla stampa.

3.3. CR3: Idee per l’implementazioneDal momento che con la CR2 verrà implementato un insieme di endpoint REST, è possibile sfruttare tali endpoint per creare una WebApp indipendente dal tool che consenta un’interazione semplificata con quest’ultimo disaccoppiando le tecnologie. Verrà, infatti, utilizzato il Framework React che mediante semplici chiamate HTTP ottiene i risultati dell’analisi del tool e le mostra all’utente.

3.4. CR4: Idee per l’implementazione  
Nella classe *RemoteSnapshotsProjectsControl*, per ricavare l’URI delle repo remote dal file CSV che le specifica, viene utilizzata la classe *Paths*. Se questa classe viene utilizzata su un sistema operativo **Unix-based**, riesce a gestire correttamente la differenza tra path **locale e remoto**. Tuttavia, se utilizzato su sistema Microsoft Windows, quando viene specificato un path **remoto** viene sollevata un’eccezione del FileSystem riguardante un errore di parsing nel percorso (il carattere “:” non può essere utilizzato per i path in questo sistema). Per questa ragione, si può effettuare un controllo sul sistema operativo in uso e, nel caso viene rilevato un sistema Windows, utilizzare una classe dedicata chiamata *URI* per gestire i **path remoti**.

3.5. CR5: Idee per l’implementazione  
Nella classe *CAIImpl* viene modellato il calcolo della metrica **CAI** per le classi in esame dal tool. Il calcolo all’interno della classe, tuttavia, non è corretto perché nella divisione presente all’interno della formula viene utilizzato un metodo errato. Il metodo in questione è *getUsageClassifiedMethods()* che considera solo i metodi **classified** effettivamente utilizzati da altri metodi, al contrario di quanto specificato nella formula fornita dal paper di riferimento (). Per risolvere basta semplicemente sostituire il suddetto metodo con *getAllClassifiedMethods()*, che considera correttamente **tutti** i metodi **classified** consentendo il corretto calcolo in riferimento alla formula sopracitata.

3.5. CR6: Idee per l’implementazione  
Nella classe *ClassifiedPatterns*, viene caricato il file patterns mediante uno scanner che prende in input il path relativo al suddetto file. Tuttavia, quando viene effetuata la build del sistema, e quindi viene generato il jar, il path specificato non é più valido. Per questo motivo, al momento del caricamento del file questo non viene trovato. Per risolvere il problema, si può usare un *InputStream* ottenuto tramite il metodo *getResourceAsStream()* con specificato solo il nome del file senza fare riferimento al path.

# 4. Impact Analysis

In seguito alla definizione delle Change Requests è stata effettuata un’Impact Analysis iniziale volta all’identificazione dello **Starting Impact Set** e del **Candidate Impact Set**. Data l’assenza di documentazione, l’unico strumento disponibile per effettuare tale analisi è stato il codice, sul quale è stata effettuata in primo luogo una fase di code comprehension (il cui resoconto è consultabile nella sezione *docs* della Repository) e dal quale, successivamente, è stato ottenuto un Class Diagram utilizzando il tool di reverse engineering integrato nell’IDE IntelliJ IDEA. In una fase successiva, a partire dal modello precedentemente estrapolato sono stati ottenuti i risultati di seguito riportati:

* **CR1 - Impact Sets**  
  SIS: CliStarter (rinominarla in “Starter” e aggiungere l’istanziazione del tipo di parser generico)  
  CIS: CliParser (deve implementare un’interfaccia “parser” specifica)
* **CR2 - Impact Sets**  
  SIS: SnapshotExporterFactory (aggiungere l’istanziazione di un JSONExporter)  
  CIS: ParserFactory (aggiungere l’istanziazione di un JSONParser)
* **CR3 - Impact Sets**SIS:   
  CIS:
* **CR4 - Impact Sets**SIS: RemoteSnapshotsProjectsControl  
  CIS:   
  DIS:   
  AIS: RemoteSnapshotsProjectsControl  
  FPIS:
* **CR5 - Impact Sets**SIS: CAIImpl  
  CIS:   
  DIS:   
  AIS: CAIImpl  
  FPIS:
* **CR6 - Impact Sets**SIS: ClassifiedPatterns  
  CIS:   
  DIS:   
  AIS: ClassifiedPatterns  
  FPIS:

# Change requests implementate

Le change requests che si è deciso di implementare sono quelle considerate a priorità alta. Due di queste consentono il corretto funzionamento del tool rispetto al calcolo di una delle metriche supportate dal tool e una riguardante l’esecuzione su Microsoft Windows. Una descrizione ad alto livello del lavoro svolto è riportata di seguito:

## CR4

La prima Change Request implementata è stata la “CR4” in quanto permette la corretta esecuzione sul sistema Microsoft Windows e dunque l’implementazione e il testing nell’ambiente di lavoro utilizzato dal team. La modifica ha riguardato la gestione dei path remoti dei repositories da analizzare. Nello specifico all’interno del metodo *run()* della classe *RemoteSnapshotsProjectsControl* veniva usata la classe Paths per gestire sia URI locali che remoti. Nonostante questa classe funzioni in entrambi i casi su sistemi Unix, in sistemi Microsoft Windows viene lanciata una eccezione di tipo *InvalidPathException* poiché all’interno di un path non può comparire il carattere “:”, presente ovviamente negli URL. Per ovviare al problema è stato sufficiente effettuare innanzitutto un controllo sulla piattaforma in cui il tool è correntemente in esecuzione e, nel caso ci si trovasse in ambiente Windows, viene utilizzata la classe *URI*, specializzata nella gestione degli URL.

## CR5

La modifica successivamente implementata è stata la CR5 in quanto il tool calcolava in modo errato la metrica CAI (Classified Attributes Interactions). Il problema era localizzato all’interno del metodo *compute(…)* della classe *CAIImpl* e consisteva nell’utilizzo di un metodo scorretto nel calcolo della metrica: “*getUsageClassifiedMethods()*”. Per l’implementazione della change request, quindi, è stato sufficiente sostituire il metodo precedente con “*getAllClassifiedMethods()*”.

## CR1

La modifica successivamente implementata è la CR1, e permette al tool di costruire il suo input non solo a partire da parametri provenienti dalla riga di comando, come avveniva in precedenza, ma anche a partire da altre fonti (ad esempio un JSON). Ciò è stato realizzato utilizzando il design pattern **factory** introducendo la classe astratta *Parser* e la classe *ParserFactory* che, sulla base del tipo di parametri desiderati, seleziona il parser adatto. Le tipologie di parser supportate sono elencate all’interno della enum *ParserID*.

## CR6

L’ultima change request implementata é la CR6, per risolvere un problema riscontrato nella fase di building, dove il file patterns non veniva trovato all’interno del jar. Ciò é stato risolto tramite l’utilizzo della classe *InputStream* utilizzando il metodo getResourceAsStream() che prende in input solo il nome del file senza specificare il path, permettendo di caricare correttamente il file a prescindere dal suo path. L’istanza della classe *InputStream* associata a questo file è stata fornita in input allo *Scanner* che si occupa di leggere il file.

# Testing di Sistema

Il testing di Sistema sarà effettuato utilizzando uno script in Bash/Batch che richiamerà il tool da linea di comando e si occuperà di richiamare i casi di test