Documento di manutenzione

VITRuM

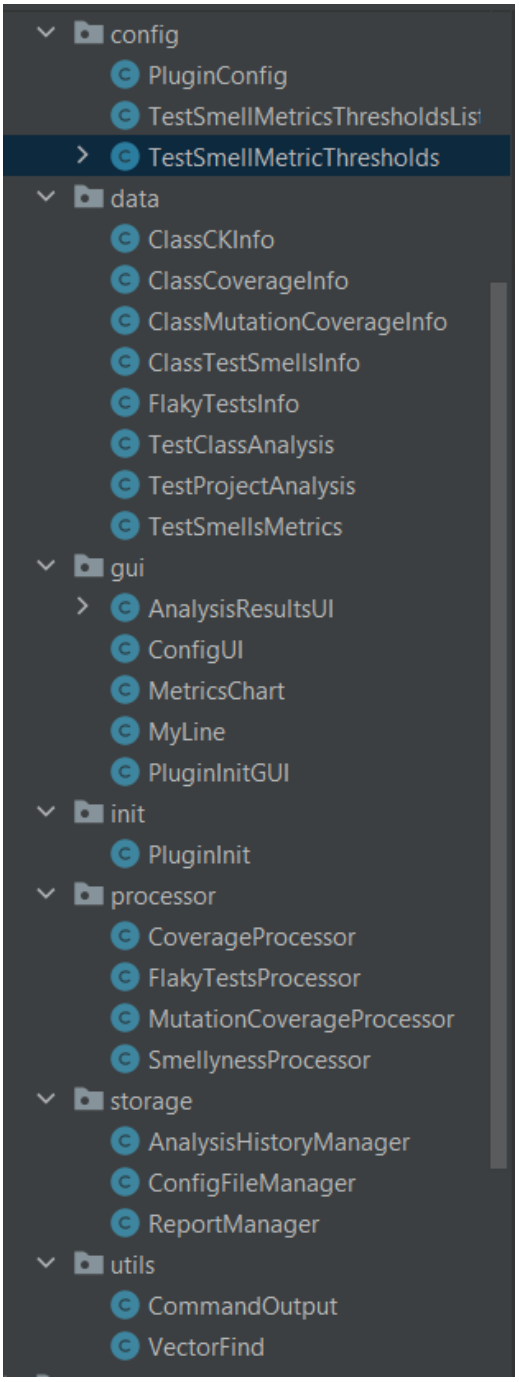
Conte Armando	0522501023
De Stefano Saverio	0522501109
Inglese Mario	0522501024
Pagliaro Vincenzo	0522500968

Tabella dei contenuti

1. Studi preliminari

1.1 Panoramica del sistema

Il sistema in esame si compone sostanzialmente di 6 sottosistemi: *Config, Data, Init, Processor, Storage, Utils*. A questo si deve aggiungere un sottosistema aggiuntivo che contiene tutte le componenti di interfaccia grafica (sottosistema *GUI*). Di seguito verranno descritti nel dettaglio:



Sottosistema config: Questo sottosistema contiene 3 classi: ***PluginConfig, TestSmellMetricThresholdsList e TestSmellMetricThresholds***. Questo sottosistema permette di gestire e modificare le configurazioni.

Sottosistema data: Questo sottosistema contiene le classi che rappresentano le unità dei dati su cui lavora il plugin. Sono presenti 8 classi: ***ClassCKInfo, ClassCoverageInfo, ClassMutationCoverageInfo, ClassTestSmellsInfo, FlakyTestInfo, TestClassAnalysis, TestProjectAnalysis, TestSmellsMetrics***. Ognuno di essi è relativo a una classe di metriche. Essi rappresentano i dati su cui lavorano tutti gli altri sottosistemi.

Sottosistema init: Questo sottosistema contiene la classe ***PluginInit*** che si occupa di estrarre il progetto e salvarlo in una classe di ***TestProjectAnalysis*** estraendo le informazioni di cui ha bisogno per poi passarle alla classe ***initGUI***.

Sottosistema processor: Questo sottosistema contiene le classi responsabili di effettuare l'analisi per il calcolo delle metriche dei test. Il sistema contiene 4 classi corrispondenti alle metriche computate: ***CoveregeProcessor, FlakyTestsProcessor, MutationCoveregeProcessor, SmellynessProcessor***. Le classi di questo package utilizzano classi del sottosistema data per gestire le metriche calcolate.

Sottosistema storage: Questo sottosistema contiene le classi: ***AnalysisHistoryManager, ConfigFileManager e ReportManager***. Sono responsabili dello storage dei dati che sono stati calcolati.

Sottosistema utils: Questo sottosistema contiene 2 classi: ***CommandOutput*** *e* ***VectorFind.*** Dove la prima si occupa di eseguire da linea di comando una istruzione passata come parametro. Mentre la seconda classe si occupa di fare delle ricerche all'interno di vettori di metriche.

1.2 Change request effettuate

1.2.1 Change request 01

La *change request 01* richiede una manutenzione evolutiva, atta a convertire il progetto in un progetto Gradle. Il tool segue la struttura di progetto IntelliJ Platform Plugin, una struttura e sistema di build proprio di IntelliJ adibito ai soli plugin. Questa struttura è adesso sconsigliata, e JetBrains consiglia di migrare ad una struttura Gradle, noto sistema di build per Java simile a Maven e già usato per le app native Android. L'obiettivo di questa change request è quindi quello di convertire il progetto secondo Gradle. Dunque i cambiamenti che verranno apportati non saranno visibili a livello di funzionalità, bensì a livello di struttura del sistema.

Analisi della Modifica Richiesta

Il passo immediatamente successivo alla valutazione di problemi e soluzioni, consiste nell'individuazione degli impact set.

Individuazione dello Starting Impact Set e Candidate Impact Set

Lo Starting Impact Set conterrà tutte quelle componenti che sono direttamente impattate dalla modifica. Lo Starting Impact Set per questa change request prevederà tutte le seguenti classi.

Sia lo Starting Impact Set che il Candidate Impact Set conterranno tutte le classi in quanto ne andremo a modificare la struttura poiché attualmente è la seguente:

```
src
    ├── config
    ├── data
    ├── gui
    ├── init
    ├── processor
    ├── storage
    ├── utils
```

Che si passerà ad una struttura che segue quella di Gradle.

Individuazione della Actual Impact Set

In questa sezione vedremo come la stima sull'entità fatta nella sezione precedente sia stata precisa. Andremo, innanzitutto, a confrontare Actual Impact Set con il Candidate e verificheremo se ci sono stati dei falsi positivi oppure delle componenti che non sono state considerate.

Dopo aver effettuato le modifiche, l'Actual Impact Set conterrà anch'esso tutte le classi in quanto sono tutte impattate dalla modifica.

La struttura attuale seguirà quella di Gradle e sarà la seguente:

```
└── src
    ├── main
    │   ├── java
    │   │   └── config
    │   │   └── data
    │   │   └── gui
    │   │   └── init
    │   │   └── processor
    │   │   └── storage
    │   │   └── utils
    │   └── resources
    │   │   └── META-INF
    │   │       └── plugin.xml
```

Calcolo delle metriche

Di seguito, calcoliamo alcune metriche per verificare l'accuratezza dell Impact Analysis.

$$\text{Recall} = \frac{|CIS \cap AIS|}{|AIS|} = \frac{26}{26} = 1$$

$$\text{Precision} = \frac{|CIS \cap AIS|}{|CIS|} = \frac{26}{26} = 1$$

$$\text{Inclusiveness} = 1 \text{ perchè AIS} \subseteq \text{CIS}$$

La modifica è stata portata a termine con successo. Il processo di Impact Analysis ha stimato correttamente le classi che venivano impattate dalla modifica, infatti si ha avuto sia la Recall che la Precision pari ad 1, quindi non si sono avuti dei falsi positivi e nessuna classe realmente modificata è stata tralasciata dalla stima. Inoltre si ha la Inclusiveness pari 1 perchè AIS $\subseteq$ CIS.

1.2.2 Change request 02

La *change request 02* richiede una manutenzione evolutiva, atta a separare il progetto in due moduli ben distinti:

- uno per la sola logica legata all'infrastruttura plugin IntelliJ;
- uno contenente il cuore del tool, che è indipendente dal fatto di essere un plugin per un IDE;

L'obiettivo di questa change request è quindi quello di avere due moduli ben distinti. Dunque i cambiamenti che verranno apportati non saranno visibili a livello di funzionalità, bensì a livello di struttura del sistema.

Analisi della Modifica Richiesta

La change request 02 ha come obiettivo dunque quello di migliorare la flessibilità e la manutenibilità dell'intero sistema. Questo perché dividendo il sistema in due moduli ben distinti si avrà che la parte "core", ovvero il cuore del sistema, potrà essere utilizzato come una libreria separata utilizzabile da qualsiasi applicazione java ed inoltre essendo separato sarà più manutenibile in quanto non dipende da librerie specifiche del plugin.
Il passo immediatamente successivo alla valutazione di problemi e soluzioni, consiste nell'individuazione degli impact set.

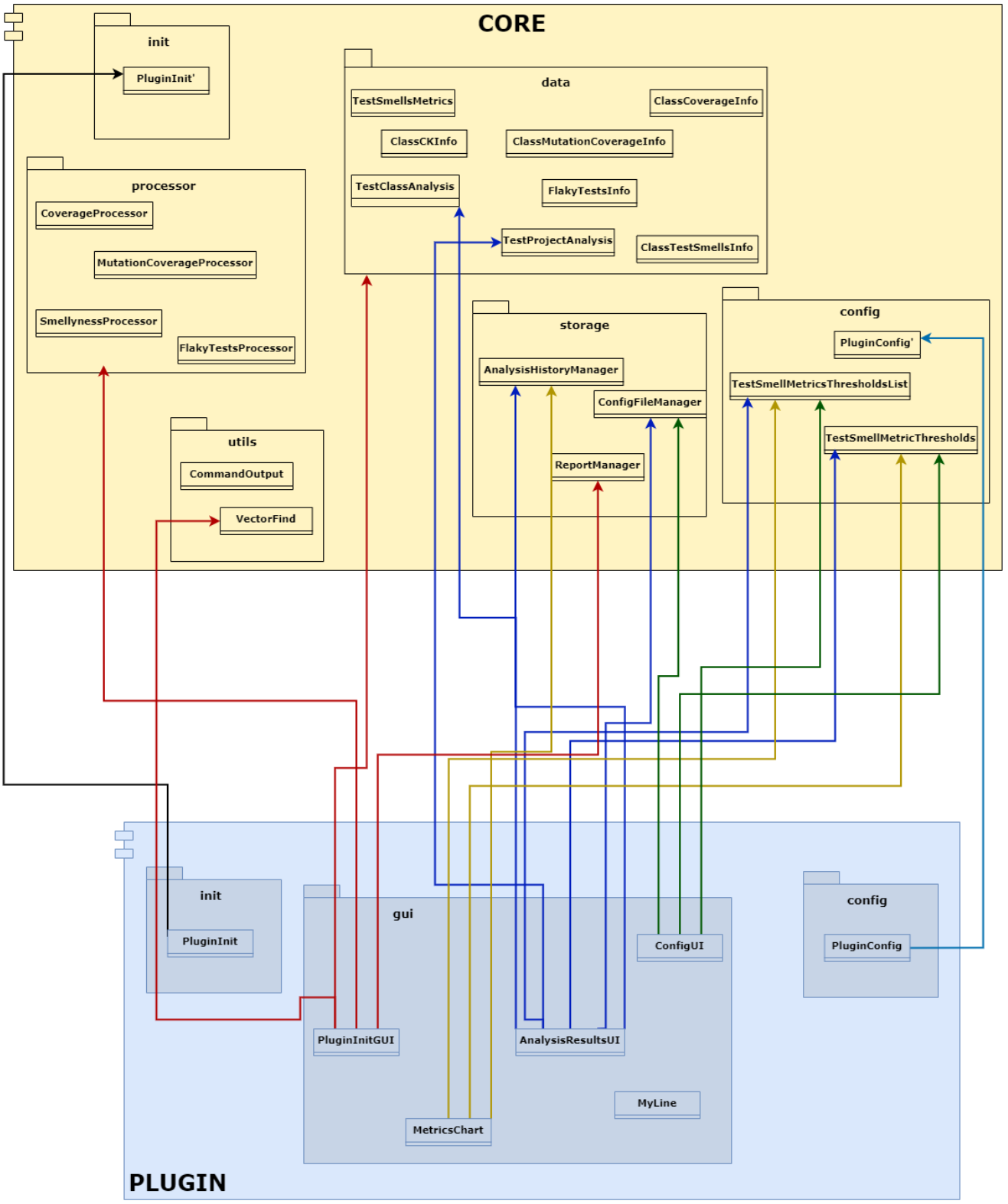
Individuazione dello Starting Impact Set

Nello starting impact set osservando la struttura del progetto si è notato che alcuni package dipendevano dalla logica del plugin intellij mentre altri facevano parte del cuore del tool, quindi nello SIS i package impattati sono tutti, ma soltanto a livello strutturale. Non si esclude però qualche modifica a qualche classe.

Per prima cosa, partendo dal progetto Gradle, si è pensato di creare due moduli separati (collegato a quanto detto prima)

Individuazione del Candidate Impact Set

Nel candidate impact set sono stati considerati i package divisi tra i due moduli nel seguente modo:



Si è deciso di analizzare in questa maniera le dipendenze poiché il progetto risulta essere di dimensioni non eccessive e pertanto è risultato più semplice e non troppo onerosa l'individuazione delle dipendenze tra le classi anche con l'aiuto della funzionalità direttamente integrate nell'IDE IntelliJ, attraverso l'opzione che mostra i class diagrams la quale ha consentito di analizzare e visualizzare le dipendenze tra le classi e tra i package source del progetto. Facendo in questo modo, inoltre, si è incrementa notevolmente la precision la quale verrà accuratamente calcolata in seguito.

Analizzando quindi le dipendenze tra i vari package e le varie classi si può dedurre che quelle impattate dalla change request 02 saranno:

- La classe ***PluginInit*** che verrà splittata in quanto contiene sia la logica del plugin che quella core, andremo quindi a dividerla in due classi: ***PluginInit*** (che sarà nel modulo *plugin* e conterrà la logica del plugin) ed ***PluginInit'*** (che sarà nel modulo *core* e conterrà la logica core del tool, e verrà usata dalla classe ***PluginInit***);
- La classe ***PluginConfig*** che verrà splittata anch'essa in quanto contiene sia la logica del plugin che quella core, andremo dunque a dividerla in due classi: ***PluginConfig*** (che sarà nel modulo *plugin* e conterrà la logica del plugin) e ***PluginConfig'*** (che sarà nel modulo *core* e conterrà la logica core del tool, e verrà usata dalla classe ***PluginConfig***);

Per semplicità le classi ***PluginInit'*** e ***PluginConfig'*** si chiameranno rispettivamente ***Init*** e ***Config*** nel modulo core.

Le altre classi rimanenti non saranno impattate direttamente poiché gradle si occuperà di linkare le dipendenze al modulo core.

Report Post-modifica

Una volta effettuate le modifiche relative alla change request analizziamo quanto la stima sull'entità fatta nella sezione precedente sia stata precisa. Andremo, innanzitutto, a confrontare Actual Impact Set con il Candidate Impact Set e verificheremo se ci sono stati dei falsi positivi oppure delle componenti che non sono state considerate (Discovered Impact Set).

Individuazione dell'Actual Impact Set

Dopo aver effettuato le modifiche, l'Actual Impact Set contiene le seguenti classi:

- ***PluginConfig***
- ***PluginInit***
- ***CoverageProcessor***
- ***FlakyTestsProcessor***
- ***MutationCoverageProcessor***
- ***SmellynessProcessor***
- ***AnalysisHistoryManager***
- ***ConfigFileManager***
- ***ReportManager***

Individuazione dell'Discovered Impact Set

Dunque ci sono risultate classi che non avevamo considerato nel Candidate Impact Set in quanto facendo la differenza tra AIS e CIS. Le classi non considerate nel Candidate Impact Set sono dunque:

- ***CoverageProcessor***
- ***FlakyTestsProcessor***
- ***MutationCoverageProcessor***

- ***SmellynessProcessor***
- ***AnalysisHistoryManager***
- ***ConfigFileManager***
- ***ReportManager***

Ci siamo accorti di queste classi che sono state impattate dalla modifica quando siamo andati a compilare il progetto, in quanto non venivano viste più alcune dipendenze, avendo separato queste classi dal modulo *Plugin* (che conteneva le dipendenze di intellij plugin) e di conseguenza il progetto quindi non veniva compilato.

Queste classi, dunque, non erano state considerate nel Candidate Impact Set in quanto non dipendevano direttamente da altre classi del modulo *Plugin*, ma avevano una dipendenza da una classe di una delle librerie del plugin intellij (com.intellij.openapi.diagnostic.Logger) relativa all'uso del logger. Per eliminare dunque questa dipendenza senza influenzare il comportamento e le funzionalità del tool, è stato sostituito il precedente logger con il logger log4j (import org.apache.log4j.Logger).

Calcolo delle metriche

Di seguito, calcoliamo alcune metriche per verificare l'accuratezza dell' Impact Analysis.

$$\text{Recall} = \frac{|CIS \cap AIS|}{|AIS|} = \frac{2}{9} = 0{,}22$$

$$\text{Precision} = \frac{|CIS \cap AIS|}{|CIS|} = \frac{2}{2} = 1$$

$$\text{Inclusiveness} = 0 \text{ perchè AIS non è incluso in CIS}$$

La modifica è stata portata a termine con successo. Il processo di Impact Analysis non ha stimato correttamente le classi che venivano impattate dalla modifica, in quanto la Recall è stata di 0,22 poiché il nostro Discovered Impact Set non era vuoto, ciò vuol dire che alcune classi non erano state considerate nell Candidate Impact Set. Dall'altra parte la Precision è pari ad 1, quindi non abbiamo avuto dei falsi positivi. Inoltre abbiamo la Inclusiveness pari 0 perchè AIS non è incluso nel CIS avendo l'AIS delle classi in più.

1.2.3 Change request 03

Si vuole rendere il tool eseguibile da linea di comando. Deve essere possibile (1) lanciare la detection su un progetto specificati in input, (2) esportare i risultati in un formato adeguato.

Analisi della Modifica Richiesta

La change request 03 ha come obiettivo, dunque, quello di migliorare l'estendibilità dell'intero sistema. Questo perché l'implementazione di questa change request farà in modo che il sistema potrà essere utilizzato da linea di comando ricevendo in input parametri quali:

- Path del progetto da analizzare
- Path destinazione in cui verranno salvati i risultati
- Path delle librerie richieste dall'applicativo
- Opzioni per i vari metodi di analisi

Per l'implementazione di ciò, riteniamo necessario l'aggiunta di un nuovo modulo che chiameremo "*CLI*" il cui compito è quello, appunto, di gestire i parametri sopra elencati ed, inoltre, di usare le funzionalità del modulo core ottenuto dalla *Change Request 02*.
Il passo immediatamente successivo alla valutazione di problemi e soluzioni, consiste nell'individuazione degli impact set.

Individuazione dello Starting Impact Set

Per lo starting impact set abbiamo osservato che, in base alle esigenze del nuovo modulo *CLI*, le uniche differenze sono che il modulo core salva, attualmente, il file di report nella cartella del progetto, cosa che si vuole parametrizzare. Inoltre, attualmente, nello stesso file di report, non vengono salvati i flaky Test i quali riteniamo opportuno salvare.
Detto ciò, lo starting impact set è composto soltanto dalla classe ***ReportManager*** la quale contiene un metodo che si occupa del salvataggio del report del progetto ricavato dall'analisi dello stesso.

Individuazione del Candidate Impact Set

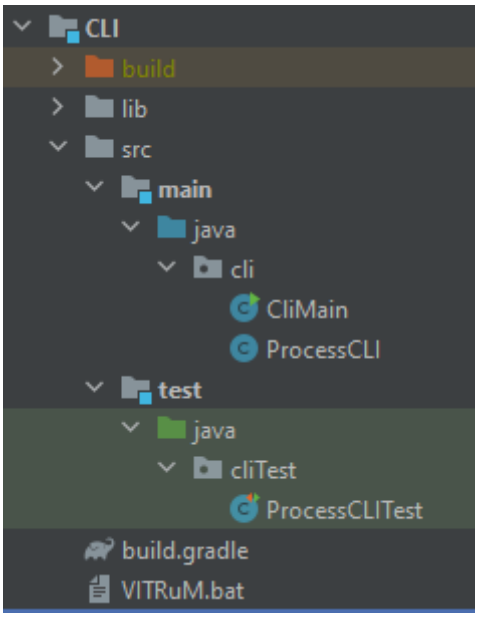
Andando ad analizzare il metodo *saveReport* della classe ***ReportManager*** è emerso che la modifica che andremo ad effettuare di questo metodo non impatterà altri metodi di altre classi e pertanto si giunge alla conclusione che anche il Candidate Impact Set è composto dalla sola classe ***ReportManager.***

Report Post-modifica

Una volta effettuate le modifiche relative alla change request si analizza quanto la stima sull'entità fatta nella sezione precedente sia stata precisa. Andremo, innanzitutto, a confrontare Actual Impact Set con il Candidate Impact Set e verificheremo se ci sono stati dei falsi positivi oppure delle componenti che non sono state considerate (Discovered Impact Set).

Le due classi che che sono state create per questo nuovo modulo CLI sono:

- ***CliMain***: che si occupa di gestire i parametri presi da linea di comando ed invocare la classe ***ProcessCLI***.
- ***ProcessCLI***: che si occupa di chiamare la logica *Core* per calcolare le metriche sui test.



Struttura del modulo CLI.

```
//flag per capire se il metodo è chiamato dal modulo CLI, in caso affermativo
// il report viene salvato al percorso indicato nella variabile static "path"
private static boolean flagCLI = false;
private static String path;

public static void saveReport(TestProjectAnalysis proj, String pathPar){
    path = pathPar;
    flagCLI = true;
    saveReport(proj);
}
```

Modifica alla classe ***ReportManager*** per parametrizzare la path di salvataggio per il nuovo modulo.

Individuazione dell'Actual Impact Set

Dopo aver effettuato le modifiche è emerso che l'actual impact set è composto dalla sola classe ***ReoportManager*** poiché sono state create delle altre classi in un nuovo modulo che pertanto sono risultate essere un'aggiunta e non una modifica delle classi dei "vecchi" moduli.

Calcolo delle metriche

Di seguito, calcoliamo alcune metriche per verificare l'accuratezza dell' Impact Analysis.

$$\text{Recall} = \frac{|CIS \cap AIS|}{|AIS|} = \frac{1}{1} = 1$$

$$\text{Precision} = \frac{|CIS \cap AIS|}{|CIS|} = \frac{1}{1} = 1$$

Inclusiveness = 1 perchè AIS è incluso in CIS

La modifica è stata portata a termine con successo. Il processo di Impact Analysis ha stimato correttamente la classe che veniva impattata dalla modifica, in quanto la Recall è stata di 1 poiché il nostro Discovered Impact Set era vuoto. Dall'altra parte anche la Precision è pari ad 1, quindi non si sono avuti dei falsi positivi. Inoltre si ha avuto la Inclusiveness pari 1 perchè AIS è incluso nel CIS.

1.2.4 Change request 04

Si vuole permettere il lancio del tool su varie versioni dei progetti. Deve essere dunque possibile lanciare la detection su uno o più progetti specificati da un file in input, analizzandone tutte le versioni e lanciando la detection per poi unire ed esportare i risultati in un formato adeguato.

Analisi della Modifica Richiesta

La change request 04 ha come obiettivo, dunque, quello di migliorare l'estendibilità dell'intero sistema. Questo perché l'implementazione di questa change request farà in modo che il sistema potrà essere utilizzato per lanciare anche analisi su larga scala; il sistema prenderà come input da linea di comando i seguenti parametri:

- Path del file di testo che contiene una lista di progetti Github da analizzare
- Path destinazione in cui verranno salvati i risultati

Individuazione dello Starting Impact Set

Per lo starting impact set l'unica classe che sarà impattata è la classe ***ReportManager*** la cui modifica consisterà nel cambiare il nome del file in cui verrà salvato il report rinominandolo come *"resultTest.csv"*.

Individuazione del Candidate Impact Set

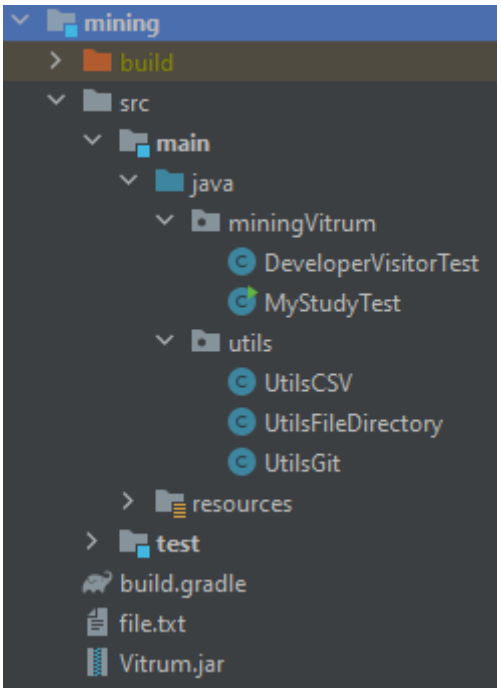
Andando ad analizzare il metodo *saveReport* della classe ***ReportManager*** è emerso che la modifica che andremo ad effettuare di questo metodo non impatterà altri metodi di altre classi e pertanto si giunge alla conclusione che anche il Candidate Impact Set è composto dalla sola classe ***ReportManager*.**

Report Post-modifica

Una volta effettuate le modifiche relative alla change request analizziamo quanto la stima sull'entità fatta nella sezione precedente sia stata precisa. Andremo, innanzitutto, a confrontare Actual Impact Set con il Candidate Impact Set e verificheremo se ci sono stati dei falsi positivi oppure delle componenti che non sono state considerate (Discovered Impact Set).

Le due classi che abbiamo creato per questo nuovo modulo *CLI* sono:

- ***MyStudyTest***: che si occupa di gestire i parametri presi da linea di comando e di lanciare il mining con Repodriller sulle varie versioni della repository.
- ***DeveloperVisitorTest***: che si occupa di chiamare il Vitrum sulla versione della repository ad un certo commit.
- Ulteriori classi utils che contengono funzioni utilitarie per fare il merge dei vari file e la loro gestione.



Struttura e classi del modulo mining

```
public static void saveReport(TestProjectAnalysis proj){
// LOGGER.info("Starting report");
String fileName = new SimpleDateFormat( pattern: "yyyyMMddHHmm'.csv'").format(new Date());
String outputDir = "";
if(flagCLI){
    fileName= "resultTest.csv";
    outputDir = path;
    flagCLI = false;
} else {
    outputDir = proj.getPath() + "\\reports";
}
```

Istruzione per il per rinominare il report

Alla fine tutti i file ottenuti verranno uniti in un unico file finale come report.

Individuazione dell'Actual Impact Set

Una volta effettuate le modifiche è emerso che l'actual impact set è composto dalla sola classe ***ReoportManager*** poiché sono state create delle altre classi in un nuovo modulo che pertanto sono risultate essere un'aggiunta e non una modifica delle classi dei "vecchi" moduli.

Calcolo delle metriche

Di seguito, si sono calcolate alcune metriche per verificare l'accuratezza dell' Impact Analysis.

$$\text{Recall} = \frac{|CIS \cap AIS|}{|AIS|} = \frac{1}{1} = 1$$

$$\text{Precision} = \frac{|CIS \cap AIS|}{|CIS|} = \frac{1}{1} = 1$$

Inclusiveness = 1 perchè AIS è incluso in CIS

La modifica è stata portata a termine con successo. Il processo di Impact Analysis ha stimato correttamente la classe che veniva impattata dalla modifica, in quanto la Recall è stata di 1 poiché il nostro Discovered Impact Set era vuoto. Dall'altra parte anche la Precision è pari ad 1, quindi non ci sono stati dei falsi positivi. Inoltre la Inclusiveness è risultata pari ad 1 perchè AIS è incluso nel CIS.

1.2.5 Change request 05

L'obiettivo della change request è quello di far scegliere all'utente se vuole utilizzare le funzionalità dell'attuale modulo *CLI* o del modulo mining in base ai parametri che passa come argomenti.

Analisi della Modifica Richiesta

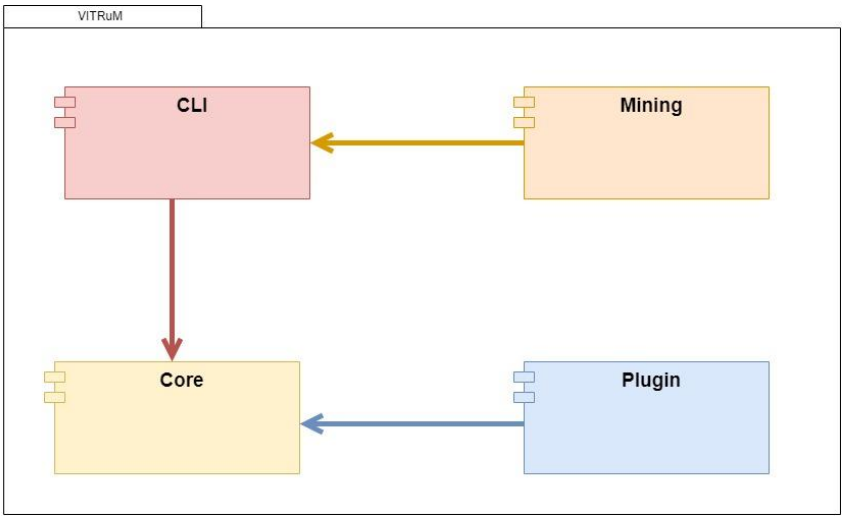
Inizialmente si era pensato di aggiungere nel modulo *CLI* questa opzione, tuttavia dopo l'analisi del modulo mining si è scoperto che con questa modifica ci sarebbe stata una dipendenza ciclica in quanto mining già dipendeva da *CLI*, e con questa si sarebbero aggiunte anche delle dipendenze a mining dal modulo *CLI* (creando così un ciclo, in quanto *mining* usa già le funzionalità di *CLI* per lanciare Vitrum sulle singole versioni del progetto).

A questo punto abbiamo i moduli *CLI* e *mining* che permettono rispettivamente: l'invocazione di Vitrum su un singolo progetto e l'invocazione di Vitrum su tutte le versioni di una lista di progetti facendo il mining.

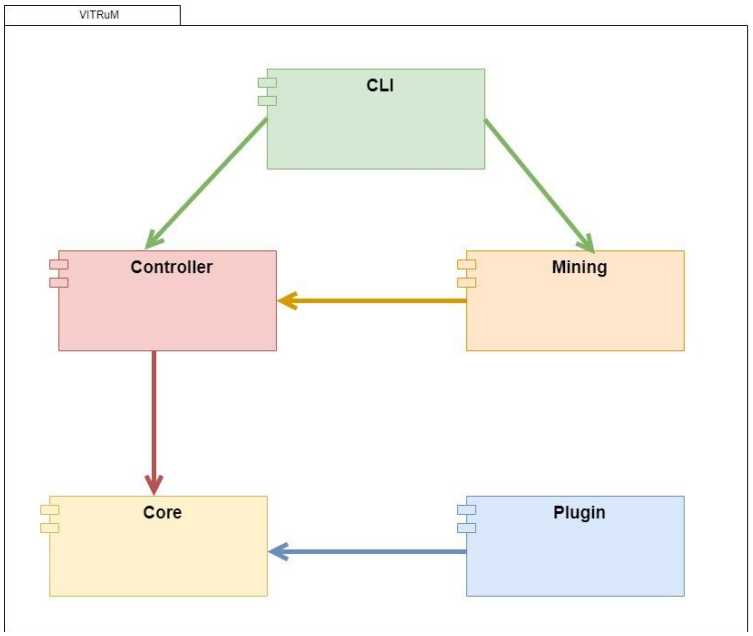
L' implementazione di questa change request ci porterà a creare un nuovo modulo padre che dipenderà sia dall'attuale *CLI* che da mining e si occuperà della gestione dei parametri e delle options passati dall'utente andando a riconoscere, in base ad essi, quale modulo invocare.

La change request 05 vuole quindi migliorare l'estendibilità dell'intero sistema. Questo perché l'implementazione di questa change request farà in modo che il sistema potrà essere utilizzato sia per lanciare l'analisi su un solo progetto, sia per lanciare anche analisi su larga scala.

Dopo avere analizzato il modulo *CLI* è risultato che l'implementazione comporterebbe un refactor dell'attuale classe ***CliMain*** in quanto questa non avrà più la responsabilità di gestire la command line interface (CLI); per questo motivo, quindi, il nuovo modulo padre che si andrà a creare, implementerà questa responsabilità e sarà lui a prendere il nome di "*CLI*", facendo sì che il vecchio modulo *CLI* venisse rinominato in "*controller*" (poiché appunto, dopo la modifica, non farà altro che richiamare i giusti metodi delle classi del modulo core anche in base ai parametri passati al costruttore della classe del "nuovo" modulo *core*).



Prima della modifica



Dopo la modifica

Individuazione dello Starting Impact Set

Per lo starting impact set saranno impattate due classi: **CliMain** dal vecchio modulo *CLI* e ***MyStudyTest*** del modulo mining le cui modifiche consistono nell'aggiungere dei metodi che prendono dei parametri passati dal nuovo modulo ed invocano la loro logica.

Individuazione del Candidate Impact Set

Andando ad analizzare la classe **ProcessCLI** si è riscontrato che non verrà impattata dalla modifica così come ***DeveloperVisitorTest***, dunque il candidate impact set sarà formato solo dalle classi: **CliMain** e **MyStudyTest**.

Report Post-modifica

Una volta effettuate le modifiche relative alla change request è stata fatta l'operazione di refactoring in CLI:

- Nome del modulo rinominato in controller;
- Nome della classe ***CliMain*** rinominata in ***CoreController***;
- Nome della classe ***ProcessCli*** rinominata in ***ProcessManager***;
- Nome del package rinominato in *controllerLogic*;

Nella classe ***CoreController*** nel modulo controller è stata tolta dunque la logica che si occupava di prendere i parametri da *CLI* lasciando quella di inizializzazione e configurazione del modulo core.

Nella classe ***MyStudyTest*** nel modulo *mining* è stato aggiunto un metodo che prende i parametri dal nuovo modulo e gestendoli ne invoca la logica.

Analizzando quanto la stima sull'entità fatta nella sezione precedente sia stata precisa si andrà innanzitutto, a confrontare Actual Impact Set con il Candidate Impact Set per verificare se ci sono stati dei falsi positivi oppure delle componenti che non sono state considerate (Discovered Impact Set).

Individuazione dell'Actual Impact Set

Una volta effettuate le modifiche è emerso che l'actual impact set è composto dalle classi ***CoreController*** e ***MyStudyTest*** in quanto sono le uniche classi impattate dalla modifica.

Calcolo delle metriche

Di seguito, calcoliamo alcune metriche per verificare l'accuratezza dell'Impact Analysis.

$$\text{Recall} = \frac{|CIS \cap AIS|}{|AIS|} = \frac{2}{2} = 1$$

$$\text{Precision} = \frac{|CIS \cap AIS|}{|CIS|} = \frac{2}{2} = 1$$

$$\text{Inclusiveness} = 1 \text{ perchè AIS è incluso in CIS}$$

La modifica è stata portata a termine con successo. Il processo di Impact Analysis ha stimato correttamente la classe che veniva impattata dalla modifica, in quanto la Recall è stata di 1 poiché il nostro Discovered Impact Set era vuoto. Dall'altra parte anche la Precision è pari ad 1, quindi non si sono avuti dei falsi positivi. Inoltre si ha avuto la Inclusiveness pari 1 perchè AIS è incluso nel CIS.

1.3 Testing

Il Test Plan ha lo scopo di analizzare e gestire le attività di testing riguardanti il progetto VITRuM, per verificare il corretto funzionamento della piattaforma. In questa sezione di documento vengono riportate le strategie di testing da adottare, le funzionalità da testare e gli strumenti utilizzati per rilevare errori all'interno del codice prodotto, in maniera pianificata per evitare che essi si presentino agli utenti finali durante l'utilizzo del sistema. I risultati di questi test saranno utilizzati per capire dove bisognerà intervenire per correggere gli eventuali errori presenti all'interno del codice. Le attività di testing sono state pianificate per i seguenti moduli:

- *core;*
- *controller;*
- *mining;*
- *CLI;*

Obiettivi del testing

Lo scopo del testing è quello di dimostrare la presenza di faults (errori) all'interno del sistema. Le attività di testing saranno mirate all'identificazione dei faults e ad un successivo intervento per eliminarne la presenza. Lo scopo del testing, infatti, è quello di dimostrare la presenza di faults (errori) all'interno del sistema. Un test avrà successo se, dato un input al sistema, l'output osservato sarà diverso dall'output atteso dall'oracolo, quindi è stata identificata una failure. In tal caso questa verrà analizzata e si procederà alla correzione. Al contrario, un test fallirà quando l'output osservato sarà uguale a quello presente nell'oracolo. Per oracolo si intende il risultato atteso, in base ai requisiti, dall'esecuzione di un caso di test. I nostri obiettivi per la fase di testing saranno quindi:

• Testare le funzionalità del sistema;

• Effettuare test di regressione ogni volta che si introducono nuove caratteristiche al sistema o vengono modificate quelle presenti;

• Massimizzare, per quanto sia possibile, attraverso la comprensione del codice, la coverage (line e branch), eseguendo il testing con un tool che permetta di calcolare la coverage dei casi di test che va ad eseguire. Si andrà poi a vedere la percentuale raggiunta, ed in base a ciò verranno coperte altre parti di codice con nuovi casi di test, finchè possibile.

Approccio

Per il sistema VITRuM si effettuerà il testing per ogni modulo man mano che verrà aggiunto. Nella prima fase verranno eseguiti i test di unità dei singoli componenti, in modo da testare nello specifico la correttezza di ciascuna unità andando a constatare il corretto funzionamento di tutte le singole unità di codice. Questa fase verrà effettuata al completamento di ogni unità realizzata per poter individuare tempestivamente gli errori presenti nel codice. Nella seconda fase verrà fatto il testing di integrazione in cui si andrà a testare l'integrazione dei vari sottosistemi. Inoltre per accertarci che le funzionalità che precedentemente funzionavano sul sistema sarà effettuato il testing di regressione ogni volta che sarà modificata qualche componente già esistente nel sistema, concludendo con il testing di sistema.

Le attività di testing saranno eseguite in parallelo con l'implementazione del sistema, man mano che viene aggiunto un nuovo modulo, dal momento che si è scelto di utilizzare un metodo di sviluppo incrementale, quindi testando prima le componenti unitarie e man mano integrare.

- **Testing di unità**

  Per effettuare il testing delle singole componenti del sistema, verrà utilizzata la tecnica "White-Box testing", in quanto non vi è una documentazione contenente specifiche formali e ben precise delle singole classi e pertanto queste sono state ricavate analizzando la struttura del codice. Come strumento sarà usato il framework JUpiter, il quale comprende JUnit. In questa fase saranno analizzate le funzionalità dell'applicazione ed il comportamento delle singole componenti tenendo conto della loro struttura interna. I risultati del testing verranno analizzati e usati per correggere gli errori che causano il fallimento del sistema. Nel caso in cui si verifichi un errore con dei risultati inattesi si interverrà in maniera tempestiva sulla componente in modo da renderla correttamente funzionante per poi procedere alle successive le fasi di testing.

- **Testing di integrazione**

  Dopo il testing di unità delle componenti, si procederà con il test di integrazione in cui andremo a verificare l'integrazione delle componenti.

- **Testing di regressione**

  Effettuato sulle componenti che sono state modificate per verificarne il corretto funzionamento anche dopo che sono state aggiunte/modificate delle parti. Come testing di regressione essendo pochi metodi abbiamo optato per l'opzione **retest-all**, in quanto richiede pochi minuti. Per le classi dei moduli che erano già presenti è stato effettuato il testing di regressione poiché prima di effettuare le modifiche si è ricavato un oracolo (risultati di un progetto analizzato) che è stato poi utilizzato per confrontare i risultati ottenuti dal testing di queste classi. Pertanto ci siamo assicurati che il funzionamento pre-modifica sia lo stesso di quello post-modifica.

- **Testing di sistema**

  Successivamente i test di integrazione e regressione si procede al testing di sistema per verificare che il sistema svolga in maniera corretta tutte le funzionalità che ci si era prefissati di implementare.

Una volta pianificato il testing dunque man mano che è stata sviluppata una nuova funzionalità in un modulo nuovo è stato effettuato il testing di unità sulle singole componenti e poi quello di integrazione sulle altre componenti.

- Modulo *core*

  Il modulo *core* essendo il modulo principale del sistema, che contiene la logica cuore del tool, questo sarà testato per primo in quanto tutti gli altri moduli del sistema dipendono

da questo modulo, pertanto andando a testare dapprima questi metodi si potrà poi fare il testing di integrazione con gli altri moduli, ed anche il testing di regressione per poter verificare che dopo eventuali modifiche sia garantito ancora il corretto funzionamento del sistema.

- Modulo *controller*

  Una volta aggiunte le funzionalità in questo modulo saranno fatti i relativi casi di test. In particolare, in questa fase, sarà effettuato il testing di regressione sulla classe **ReportManager,** coinvolta in una modifica, per un accertamento del corretto funzionamento delle precedenti funzionalità.

- Modulo *mining*

  In questo modulo saranno effettuati i test delle singole unità, in particolare di tutti i metodi utils, e poi la loro integrazione nell'esecuzione del mining con il framework *RepoDriller*.

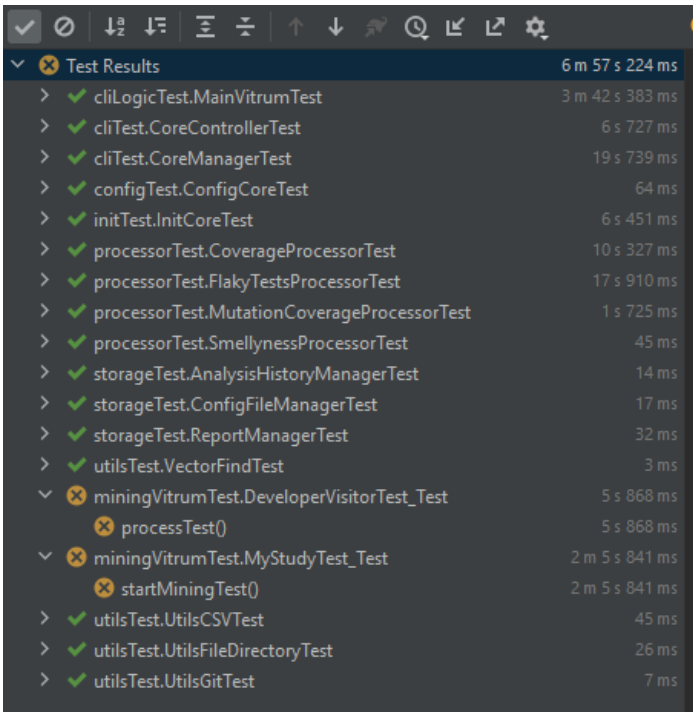
- Modulo *CLI*

  In questo modulo sarà effettuato il test di integrazione che comprende tutto il funzionamento del sistema, in quanto questo sarà il test di integrazione più alto nella gerarchia. Sarà inoltre necessario il testing di regressione dei moduli *controller* e *mining,* poiché l'invocazione della loro logica richiederà la modifica di essi attraverso l'aggiunta di metodi che ne permettono la loro esecuzione. Dovremmo quindi accertarci del loro corretto funzionamento anche dopo la modifica. Da questo modulo inoltre verrà lanciato anche il testing di sistema in quanto da qui ci sarà la possibilità di andare a lanciare l'intera logica applicativa del sistema.

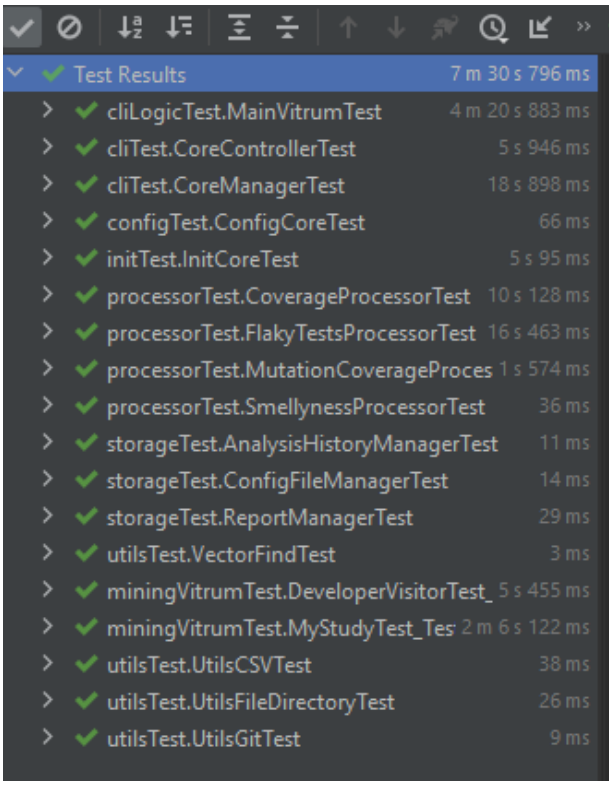
  Per ogni classe di test, al fine di migliorare la leggibilità e la comprensibilità, verranno aggiunti dei commenti che spiegheranno, in linea generale, i file usati in quella specifica classe di test.

Test report

Durante il testing di regressione sono stati riscontrati alcuni problemi che sono stati immediatamente risolti, uno di questi è stato la conseguenza della modifica alla classe **ReportManager** il che ha causato le seguenti failure agli altri test, in particolare a quelli del package *miningVitrumTest*:



Dopo averli individuati è stata effettuata la correzione dei bug attraverso un'operazione di debug mirata e successivamente è stato avviato nuovamente il testing di regressione:



Dopo aver effettuato questa fase di testing è stato ottenuto il seguente report con le relative misure di coverage:

34% classes, 41% lines covered in 'all classes in scope'

Element	Class, %	Method, %	Line, %	Branch, %
cliLogic	100% (1/1)	100% (11/11)	85% (61/71)	100% (0/0)
data	100% (8/8)	63% (97/153)	55% (164/295)	100% (0/0)
META-INF	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
processor	100% (4/4)	100% (8/8)	89% (315/353)	72% (16/22)
storage	100% (5/5)	100% (13/13)	89% (209/234)	62% (17/27)
config	80% (4/5)	74% (20/27)	60% (39/64)	50% (1/2)
utils	80% (4/5)	94% (17/18)	68% (151/219)	29% (19/65)
controllerLogic	100% (2/2)	100% (4/4)	96% (78/81)	27% (3/11)
init	50% (1/2)	50% (2/4)	58% (35/60)	21% (3/14)
miningVitrum	100% (2/2)	85% (6/7)	81% (80/98)	20% (2/10)
gui	0% (0/31)	0% (0/77)	0% (0/764)	0% (0/190)

100% classes, 89% lines covered in package 'storage'

Element	Class, %	Method, %	Line, %	Branch, %
AnalysisHistoryManager	100% (3/3)	100% (8/8)	87% (78/89)	100% (0/0)
ReportManager	100% (1/1)	100% (3/3)	89% (82/92)	71% (15/21)
ConfigFileManager	100% (1/1)	100% (2/2)	92% (49/53)	33% (2/6)

E' stato infine estratto il seguente report in formato HTML il quale è stato caricato al seguente link:

https://github.com/armandoC27/VITRUM/tree/main/report%20test%20Vitrum

Conclusioni

In conclusione, gli obiettivi del testing che ci eravamo prefissati sono stati raggiunti in quanto abbiamo testato le funzionalità principali del sistema come avevamo pianificato ed inoltre, grazie al testing di regressione, abbiamo anche preservato le vecchie funzionalità del sistema, garantendone sempre il corretto funzionamento.