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 Automated 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.

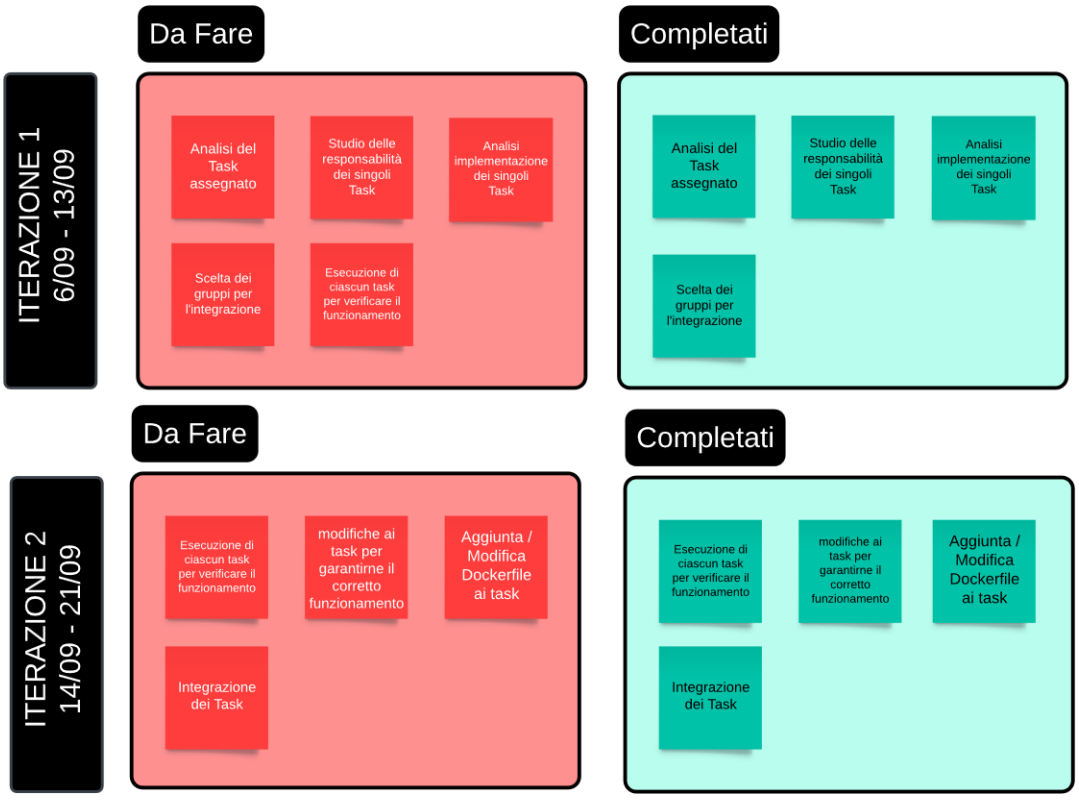
Task	Gruppo
T6	G8
T7	G31
T8	G21
T9	G19

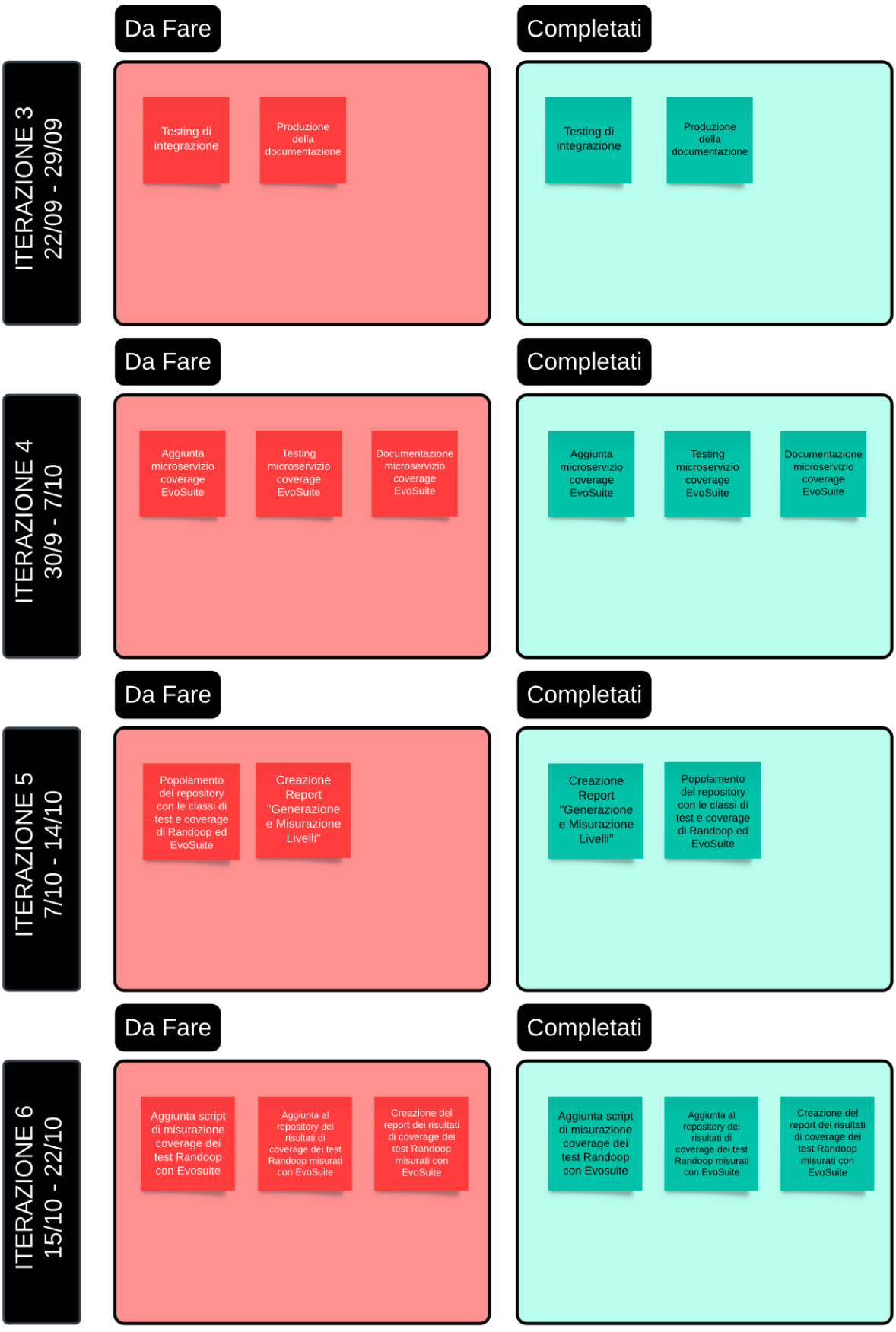
Per ognuno dei task abbiamo scelto lo specifico gruppo dopo aver consultato le documentazioni. Ad esempio per quanto riguarda T6 e T7 i gruppi G8 e G31 si erano precedentemente accordati per fornire interfacce compatibili, che hanno reso l'integrazione più semplice.

1.1 Approccio adottato

Per lo sviluppo è stata scelta 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.
Lo sviluppo poi è stato incentrato attivamente su Docker per favorire la portabilità, l'utilizzo di Docker è nato soprattutto dal fatto che ciascun task richiedeva un ambiente di sviluppo diverso e la programmazione sarebbe diventata altrimenti impossibile. Infatti, l'utilizzo di Docker favorisce anche la manutenibilità tramite il build dei servizi integrato in Docker stesso, ciò permette l'utilizzo di un semplice code editor come vscode (con eventualmente estensioni come docker e git).
Per la gestione condivisa del progetto e per il versioning si è deciso di usare GitHub.

1.2 Iteration Planning

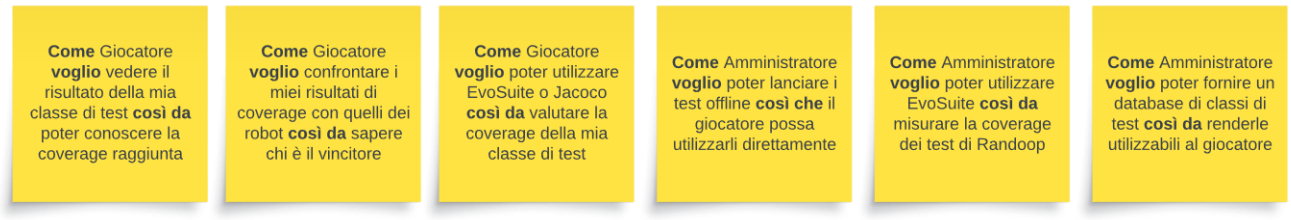



ITERAZIONE 3
22/09 - 29/09
Da Fare
Testing di integrazione
Produzione della documentazione
Completati
Testing di integrazione
Produzione della documentazione
ITERAZIONE 4
30/9 - 7/10
Da Fare
Aggiunta microservizio coverage EvoSuite
Testing microservizio coverage EvoSuite
Documentazione microservizio coverage EvoSuite
Completati
Aggiunta microservizio coverage EvoSuite
Testing microservizio coverage EvoSuite
Documentazione microservizio coverage EvoSuite
ITERAZIONE 5
7/10 - 14/10
Da Fare
Popolamento del repository con le classi di test e coverage di Randoop ed EvoSuite
Creazione Report "Generazione e Misurazione Livelli"
Completati
Creazione Report "Generazione e Misurazione Livelli"
Popolamento del repository con le classi di test e coverage di Randoop ed EvoSuite
ITERAZIONE 6
15/10 - 22/10
Da Fare
Aggiunta script di misurazione coverage dei test Randoop con Evosuite
Aggiunta al repository dei risultati di coverage dei test Randoop misurati con EvoSuite
Creazione del report dei risultati di coverage dei test Randoop misurati con EvoSuite
Completati
Aggiunta script di misurazione coverage dei test Randoop con Evosuite
Aggiunta al repository dei risultati di coverage dei test Randoop misurati con EvoSuite
Creazione del report dei risultati di coverage dei test Randoop misurati con EvoSuite

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.



In particolare vogliamo che l'utente possa utilizzare diversi strumenti per la valutazione della coverage della propria classe di test, EvoSuite o JaCoCo, che ci danno informazioni diverse. Lo stesso vale anche per l'admin che deve poter misurare tramite EvoSuite la coverage dei test generati dal robot Randoop, questo perché EvoSuite ci da metriche differenti ed è generalmente migliore rispetto ad EMMA (strumento utilizzato da Randoop)

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
5. Il servizio deve garantire all'admin la possibilità di misurare la coverage delle classi di test Randoop utilizzando EvoSuite
6. Il servizio deve garantire la possibilità di popolare un repository con classi di test funzionali e rispettive misure di coverage per l'utilizzo finale

2.3 Requisiti non funzionali

I requisiti non funzionali descrivono le proprietà del sistema:

1. Portabilità
2. Modificabilità
3. Facilità di deploy
4. 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.

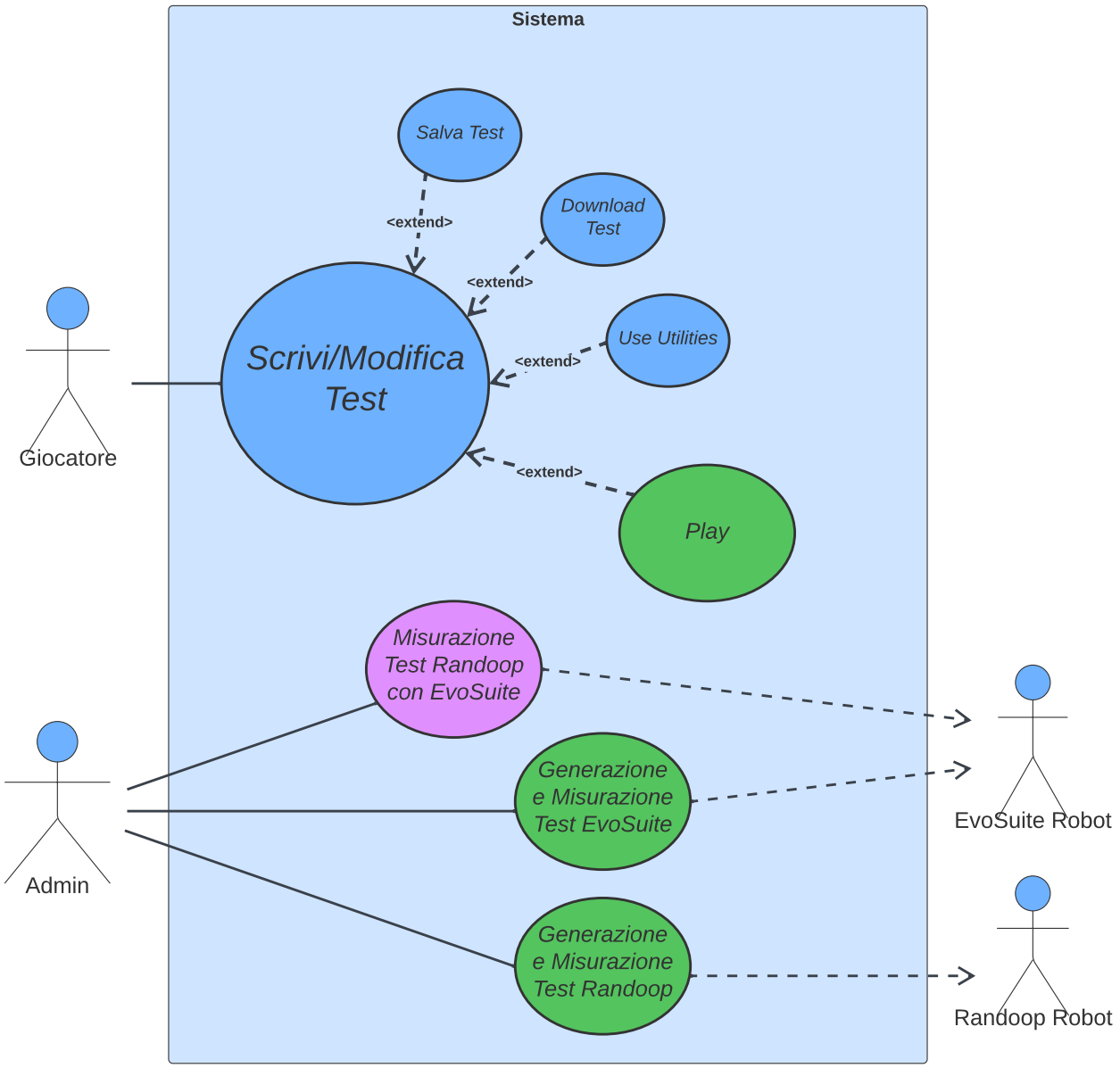
2.4 Requisiti sui dati

1. Le classi di test devono essere disponibili prima dell'avvio della partita
2. 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



Nel diagramma sono stati indicati in blu i casi d'uso che sono rimasti invariati nel processo di integrazione, in verde i casi d'uso che erano già presenti prima dell'integrazione ma sono stati modificati, mentre in viola i casi d'uso che sono stati aggiunti.

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.

Inoltre all'admin è permesso di misurare la coverage dei test Randoop anche tramite EvoSuite, otteniamo in questo modo metriche diverse e più precise rispetto ad EMMA

2.6 Scenari dei casi d'uso

2.6.1 Play

Caso d'uso	Play
Attore primario	Giocatore
Attore secondario	-
Descrizione	Il giocatore gioca 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. Il sistema fa una chiamata POST "getCoverage" al backend 3. Il backend a sua volta fa una chiamata POST al server del compilatore (Task 7) per compilare e misurare la propria classe di test 4. Il server restituisce l'output di compilazione e la coverage della classe di test valutata con JaCoCo (file XML) 5. Se si è scelto come metodo di copertura EvoSuite: 5.1. Il sistema fa una chiamata POST al server del compilatore EvoSuite 5.2. Il server restituisce l'output di compilazione e la coverage della classe di test valutata con EvoSuite (file CSV) 6. Il sistema accede al repository condiviso per il recupero della coverage del robot 7. Se robot = EvoSuite 7.1. Il sistema accede al repository e recupera il file di copertura CSV prodotto da EvoSuite, per la classe e livello scelti 8. Se robot = Randoop 8.1. Bisogna valutare il coverage Method: 8.2. Se coverage Method = JaCoCo allora il sistema recupera dal repository il file di copertura XML prodotto da JaCoCo 8.3. Se coverage Method = EvoSuite allora il sistema recupera dal repository il file di copertura CSV prodotto da EvoSuite 9. Viene creato un oggetto di tipo Coverage contenente informazioni su errori di compilazione, output di compilazione, il coverage Method, il file di coverage (XML o CSV) del robot, il file di coverage (XML e CSV) della classe dell'utente 10. Viene effettuato un parsing dei file di copertura xml e csv di utente e robot, per estrarre percentuali di copertura relative a diverse metriche 11. Viene confrontata la copertura del robot con quella dell'utente, in particolare possono essere confrontati solo risultati provenienti dallo stesso tipo di file (XML con XML e CSV con CSV) poiché considerano metriche differenti 12. Viene mostrato un alert con il vincitore e le coverage di giocatore e robot, il vincitore è decretato in base alle linee coperte 13. Vengono evidenziate le linee di codice coperte dall'utente (in verde) e non (in rosso) nella class window 14. 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

Come si può vedere da questo scenario dei casi d'uso, quando si sceglie come metodo di copertura EvoSuite viene comunque fatta una richiesta POST al compilatore del task 7, oltre che al compilatore EvoSuite. Questo dipende dal fatto che il file CSV prodotto da EvoSuite non ci permette di vedere quali linee di codice sono coperte (solo quante) e quindi non potremmo poi andare ad evidenziare tali linee nella class window. Il file XML prodotto da JaCoCo (task 7) invece ci dice anche quali linee sono coperte, quindi combinando le informazioni dei due file otteniamo sia la coverage che le linee coperte.

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	Esistono classi da testare nel repository
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 Randoop e le coverage delle classi di test sono salvati nel repository condiviso
Sequenza eventi alternativi	1. Una data classe presenta già i test Randoop 2. Non sono generati nuovi test

Se si vuole evitare che la generazione dei test Randoop parta direttamente all'avvio del container, basta scambiare gli entrypoint all'interno del Dockerfile di Randoop:

```
# ENTRYPOINT ["tail", "-f", "/dev/null"]
ENTRYPOINT /usr/lib/jvm/java-1.8-openjdk/bin/java -jar app.jar
```

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	Esistono classi da testare nel repository
Sequenza eventi principale	1. L'amministratore tramite shell esegue uno script per la generazione dei livelli (evosuite_test_script.sh), questo genera un numero di livelli prefissato (di default N=3) per ognuna delle classi del repository 2. Questo script genera i livelli richiamando lo script *"robot_generazione.sh"* per ognuna delle classi 3. I Test e le rispettive coverage sono salvati nel repository condiviso
Post-condizioni	I test EvoSuite e le coverage delle classi di test sono salvati nel repository condiviso
Sequenza eventi alternativi	• L'amministratore può generare i livelli di un'unica classe scelta utilizzando direttamente lo script *"robot_generazione.sh"* con gli opportuni parametri

2.6.4 Misurazione Test Randoop con EvoSuite

Caso d'uso	Generazione Robot Test
Attore primario	Admin
Attore secondario	EvoSuite
Descrizione	A partire dai test generati da Randoop viene misurata la coverage tramite EvoSuite (generando un file CSV)
Pre-condizioni	Sono già stati generati i test di Randoop per quella classe
Sequenza eventi principale	1. L'amministratore esegue uno script python *"randoop_coveargе.py"* 2. Questo script esegue alcune elaborazioni sui test per renderli compatibili con EvoSuite, e utilizza gli script "compilazione_test.sh" e "robot_misurazione_utente.sh" per la misurazione della coverage
Post-condizioni	Per ogni classe del repository, abbiamo anche un file CSV per i test Randoop
Sequenza eventi alternativi	1. La classe di test generata da Randoop non è compatibile con Evosuite 2. Non è generato il CSV per questo test Randoop

2.7 Scelta Classe, CoverageMethod e Robot

È possibile andare a settare questi parametri del ClassServer.js per modificare le impostazioni del gioco.

Si può scegliere la classe con la quale giocare, e quindi la classe da andare a testare, indicando il nome di questa classe nel campo "CLASS_NAME" del ClassServer. Ovviamente questa classe deve essere presente nel repository per poter essere recuperata, e deve contenere già le classi di test generate dai robot con le rispettive coverage per poterci giocare.

Possiamo poi anche scegliere contro quale robot giocare andando a scegliere Robot="Randoop" oppure Robot="EvoSuite", quindi selezioneremo i test generati dall'uno o dall'altro robot.

Infine possiamo scegliere il CoverageMethod, ovvero il metodo con il quale viene misurata la copertura della classe di test, in particolare sono possibili due scelte coverageMethod: "JaCoCo" oppure coverageMethod: "EvoSuite". Bisogna considerare che la scelta del coverageMethod è legata alla scelta del robot con il quale si intende giocare, infatti non tutte le combinazioni sono possibili:

Robot	CoverageMethod	Descrizione	PASS
Randoop	JaCoCo	La coverage dell'utente è misurata con JaCoCo mentre per il robot si prende quella di Randoop (EMMA)	OK
Randoop	EvoSuite	La coverage dell'utente è calcolata con EvoSuite, per il robot viene recuperata la misurazione EvoSuite del test generato da Randoop	OK
EvoSuite	JaCoCo	Questa combinazione non è stata realizzata	NO
EvoSuite	EvoSuite	Le coverage di utente e robot sono calcolate entrambe con EvoSuite	OK

2.8 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
EvoSuite	Robot in grado di valutare la copertura delle classi di test da lui automaticamente generate, o che gli vengono fornite (dall'utente o dal robot Randoop)	Robot EvoSuite
Randoop	Robot in grado di generare classi di test, e di valutarne la coverage con appositi strumenti	Robot Randoop
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 (usato dal task7)	-
EMMA	Strumento per la valutazione della copertura di una classe di test (usato da Randoop)	-
Metodo di coverage	Parametro che ci permettere di scegliere quale misura di copertura robot e utente mostrare (JaCoCo o EvoSuite)	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 coverage method scelto è possibile effettuare una richiesta http post al Task 7 o al task 8 per la compilazione e calcolo della 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, tale oggetto contiene campi relativi al nome e al codice della classe sotto test e della classe di test
- Modifica alla classe Partita per avere informazioni sul Robot, livello e metodo di coverage scelti
- Modifica alla classe Coverage per memorizzare la coverage dell'utente sottoforma sia di XML che CSV, la coverage del robot (XML o CSV), e il metodo di coverage necessario poi al landing.js per le sue elaborazioni
- Modifiche alla funzione getCoverage(): tramite una POST invia un oggetto RequestDTO al compilatore del task 7, e se il coverage method è EvoSuite allora lo invia anche a EvoSuite. Questo viene fatto perché quando il coverage method è EvoSuite, se non avessimo anche il file XML (ritornato dal task 7 grazie a JaCoCo) non potremmo evidenziare quali linee di codice sono coperte e quali no (EvoSuite ci da solo informazioni su quante sono coperte). Tale funzione ritorna al landing.js (che ha fatto la richiesta) un oggetto della classe Coverage dal quale potrà estrarre informazioni.
- Modifica al landing.js
    - Aggiunta una funzione di parsing (parseEvoSuiteCoverage()) per i CSV che contengono l'output di coverage utente e robot EvoSuite. Questa funzione ritorna un oggetto contenente i campi: line, weakmutation, cbranch con le rispettive coverage.
    - Aggiunta di una funzione di parsing (parseEmmaCoverage()) per gli XML che contengono la coverage del robot Randoop. Questa funzione ritorna un oggetto contenente i campi: line, block, method, class, con le rispettive coverage.
    - Modifica alla funzione parseJacocoCoverage() per il calcolo della percentuale di copertura di coverage utente relativamente alle metriche: line, branch, method, class. Tale funzione ritorna un oggetto contenente questi campi.
    - Aggiunto il confronto dei risultati dell'utente e del robot in base alle linee di codice coperte, 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.
    - Risolto 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.

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 inglobata nel primo container, grazie al servizio di misurazione 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 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 del task (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.
- Aggiunto uno script python *"/app/utente/randoop_coverage.py"* per permettere ad EvoSuite di misurare la coverage dei test generati da Randoop.

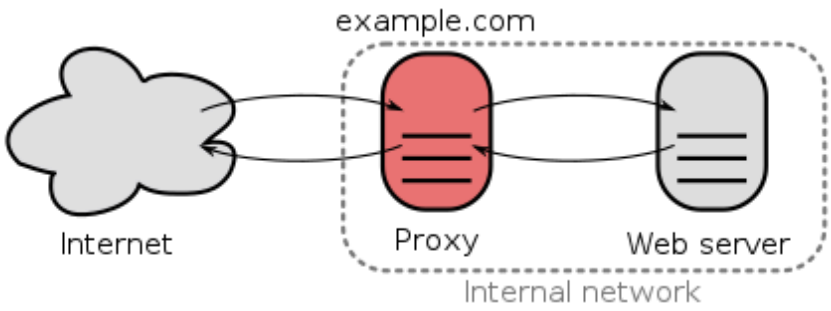
3.1.4 Task 9

- 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" (tutte le richieste che contengono "/api/" nell'URL vengono indirizzate al backend).

3.1.6 Flask e Gunicorn

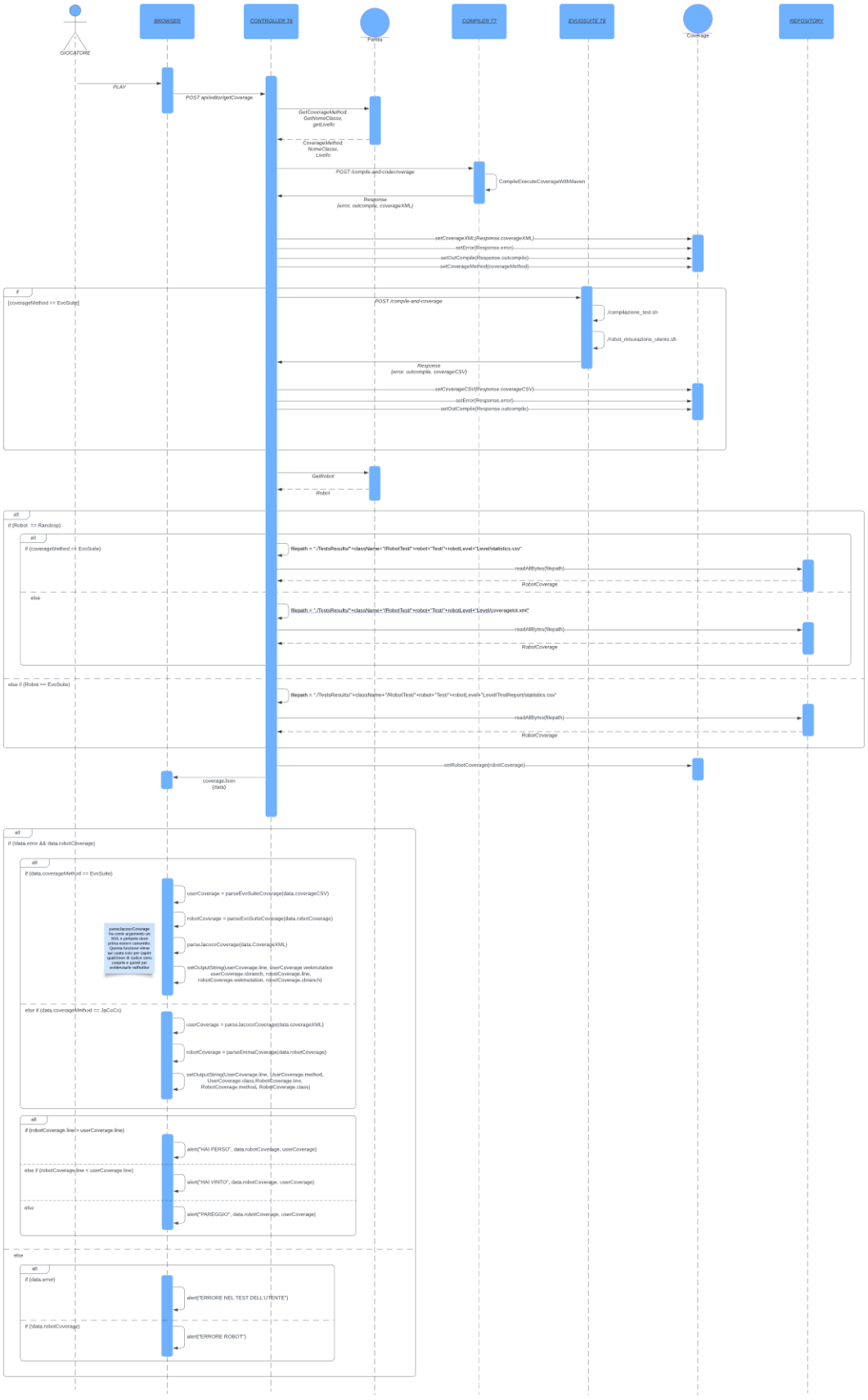
Flask è un micro-framework Web scritto in Python, basato sullo strumento Werkzeug WSGI e con il motore di template Jinja2. È ampiamente utilizzato per sviluppare applicazioni web semplici e rapide, ideali per progetti di dimensioni ridotte o medie. Flask offre un'architettura minimale e modulare, consentendo agli sviluppatori di estendere le funzionalità secondo le proprie esigenze. È stato utilizzato per la creazione dell'API del Task 8 per la misurazione dei test utente tramite EvoSuite. Data la semplicità dell'API, è stato preferito un micro-framework come Flask invece di framework più pesanti come Spring Boot.

Gunicorn (Green Unicorn) è un server HTTP WSGI (Web Server Gateway Interface) per Python. È utilizzato per ospitare applicazioni web Python, inclusi framework come Flask e Django. Gunicorn è noto per la sua scalabilità e prestazioni, in quanto è in grado di gestire più richieste simultanee in modo efficiente. Questo lo rende una scelta comune per distribuire applicazioni web Python in produzione, in particolare quando si necessita di gestire carichi di lavoro pesanti. Gunicorn, nel nostro caso, è utilizzato in combinazione con Flask per rendere l'API del Task 8 pronta per l'uso.

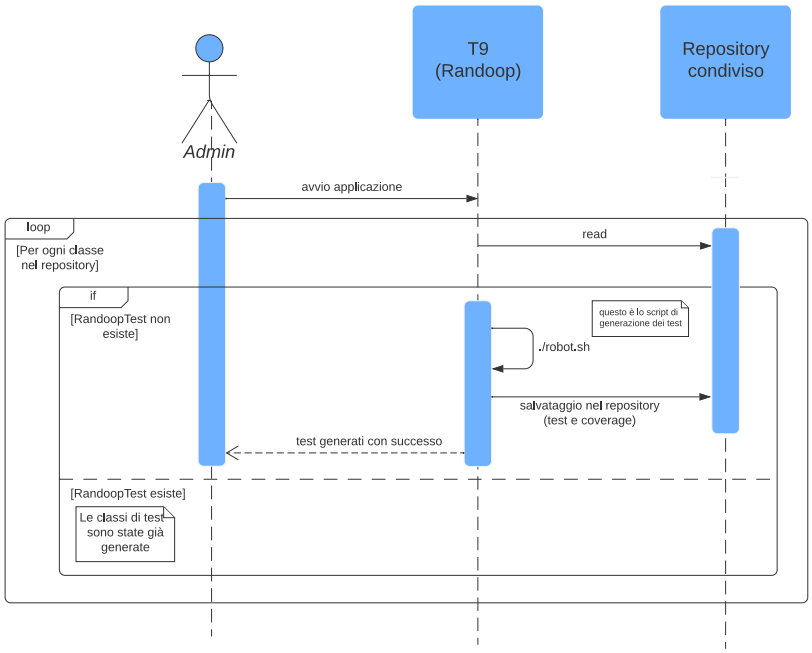
3.2 Diagrammi di Sequenza

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 specifiche funzionalità.

3.2.1 Play

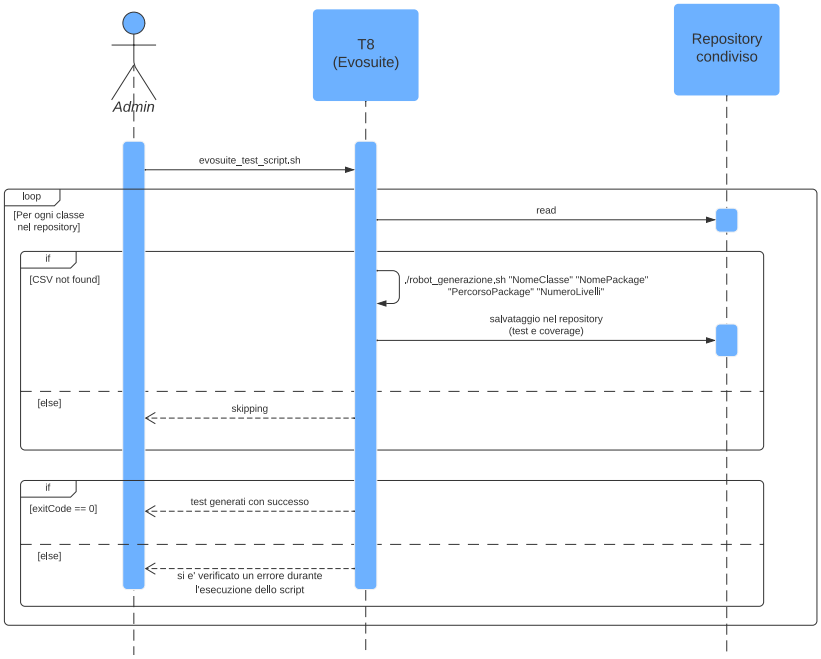


3.2.2 Generazione Test Randoop



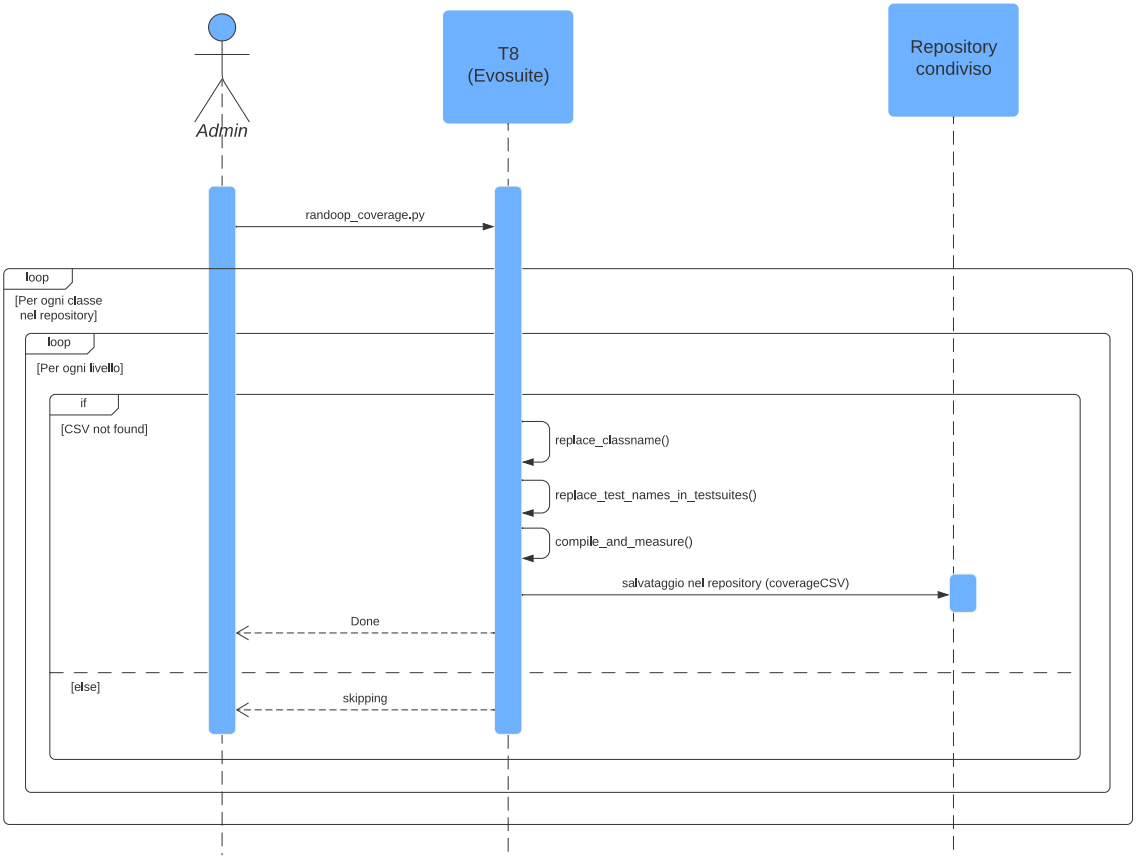
Questo caso d'uso viene avviato non appena è lanciato il container Randoop, quindi non appena viene avviata l'applicazione. Si occupa della generazione dei test e del calcolo delle coverage per tutte le classi presenti nel repository e non ancora testate (ovvero quelle classi per le quali non esiste 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 lo script *"evosuite_test_script.sh",* questo si occuperà di eseguire lo script *"robot_generazione.sh"* per la generazione dei livelli di ciascuna classe nel repository, se non sono già esistenti. A differenza di Randoop quindi i test non vengono generati automaticamente all'avvio dell'applicazione ma deve essere l'admin a dare il via.

3.2.4 Misurazione Test Randoop con EvoSuite

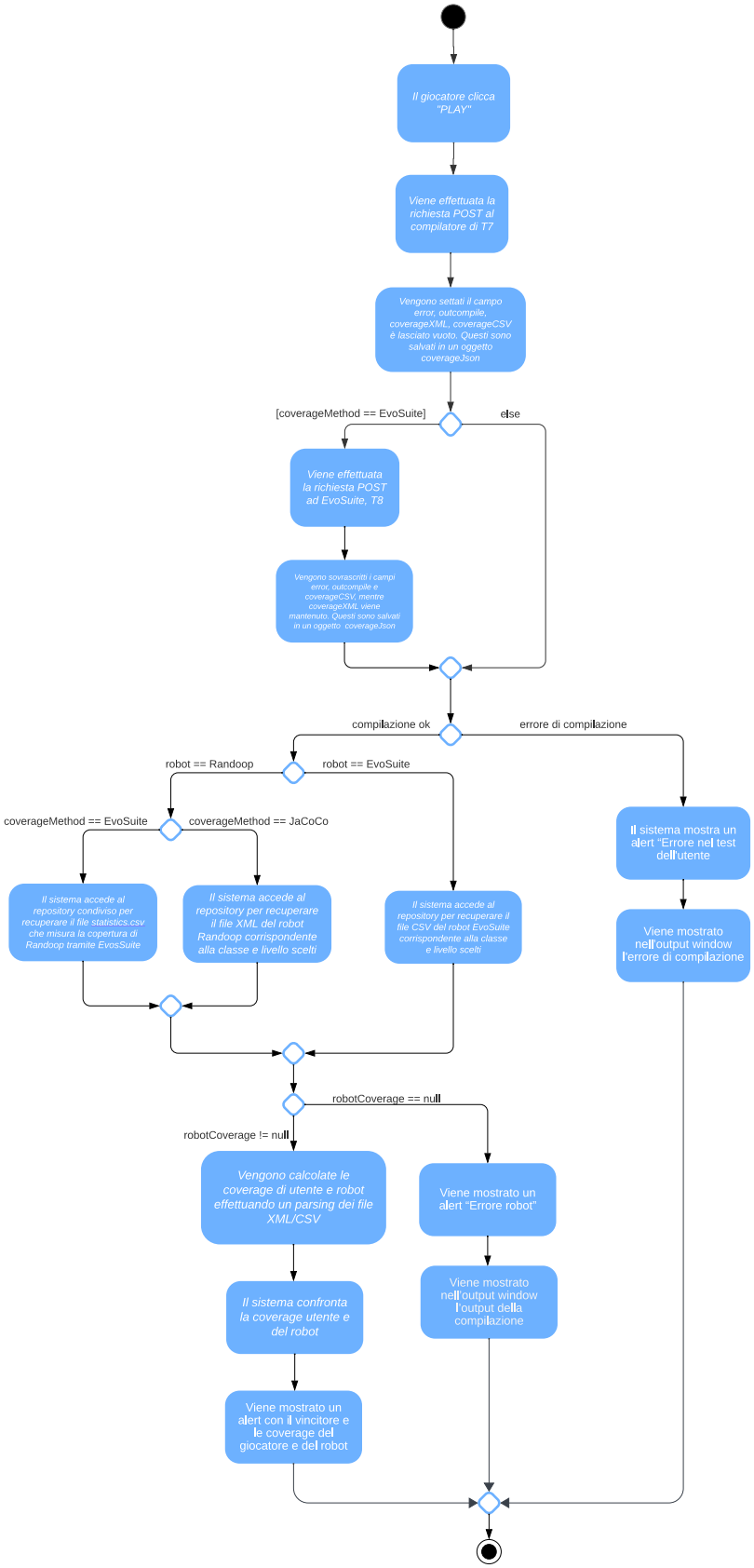


Questo caso d'uso parte nel momento in cui l'utente lancia lo script python *"randoop_coverage.py"* (va lanciato dall'amministratore all'interno del container evosuite-server), questo script eseguirà alcune elaborazioni sulle classi di test per renderle eseguibili da EvoSuite, in particolare va ad applicare una sostituzione dei nomi delle classi e delle testsuites, dopodiché esegue la funzione "*compile_and_measure()*" che va a realizzare la compilazione e misurazione della copertura con EvoSuite andando ad eseguire gli script *"compilazione_utente.sh"* e *"robot_misurazione_utente.sh"*.

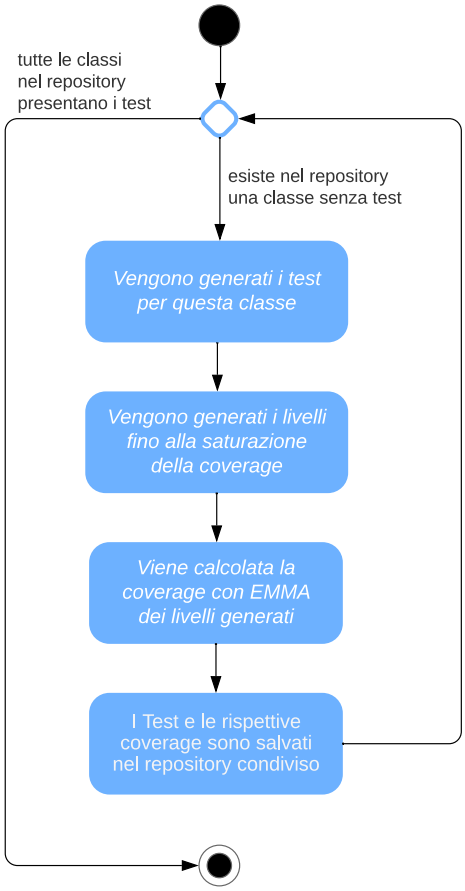
3.3 Diagramma di Attività

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

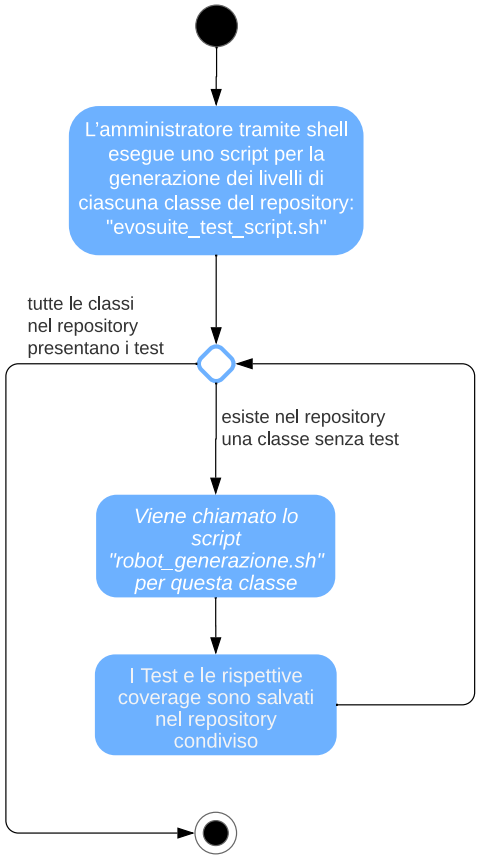
3.3.1 Play



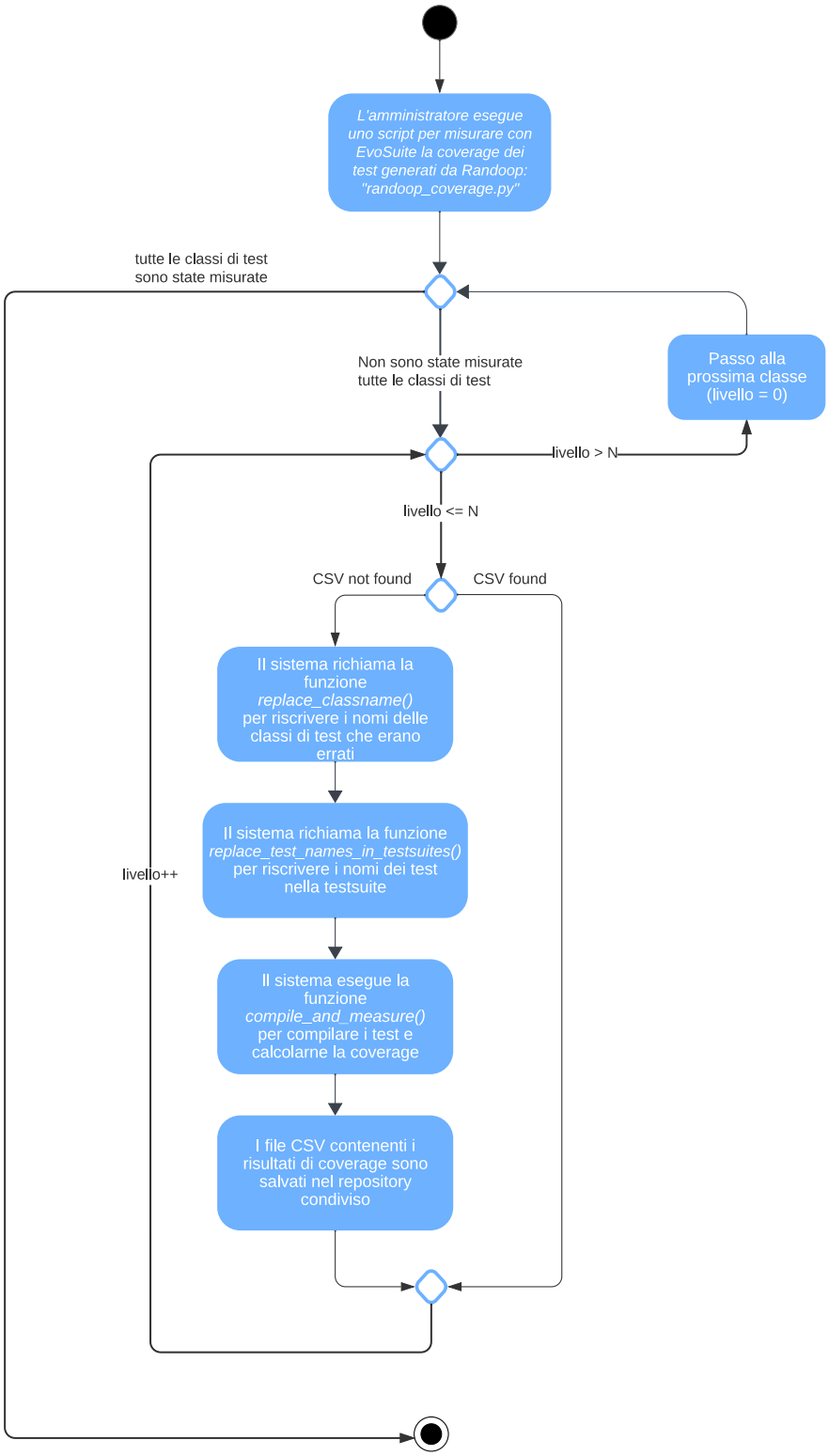
3.3.2 Generazione Test Randoop



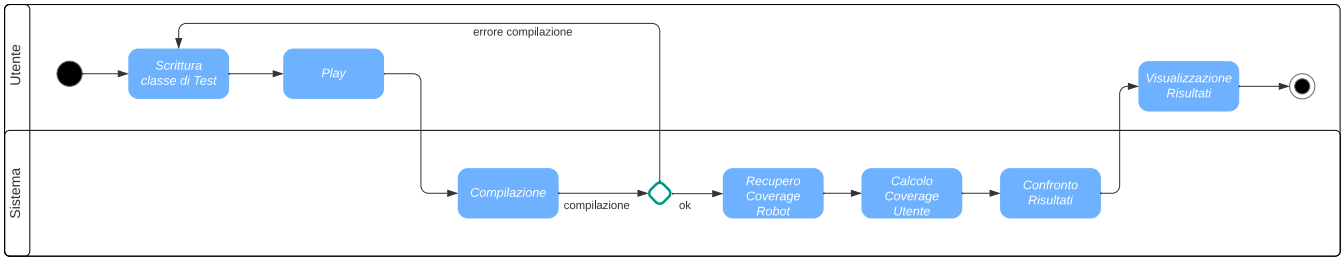
3.3.3 Generazione Test EvoSuite



3.3.4 Misurazione Test Randoop con EvoSuite

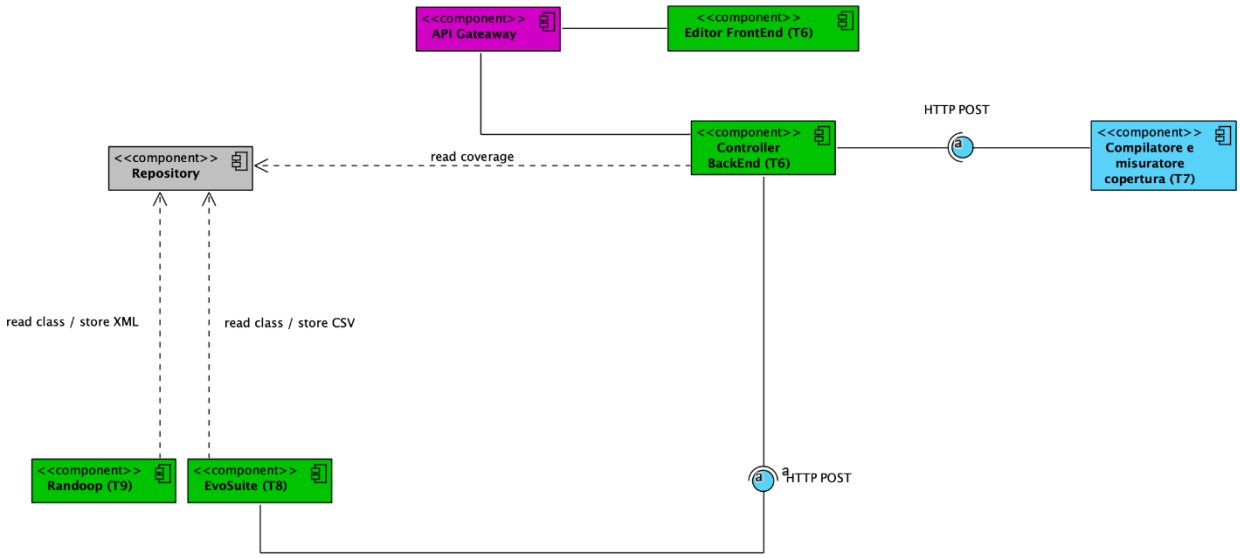


3.3.5 Flusso di attività completo



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.

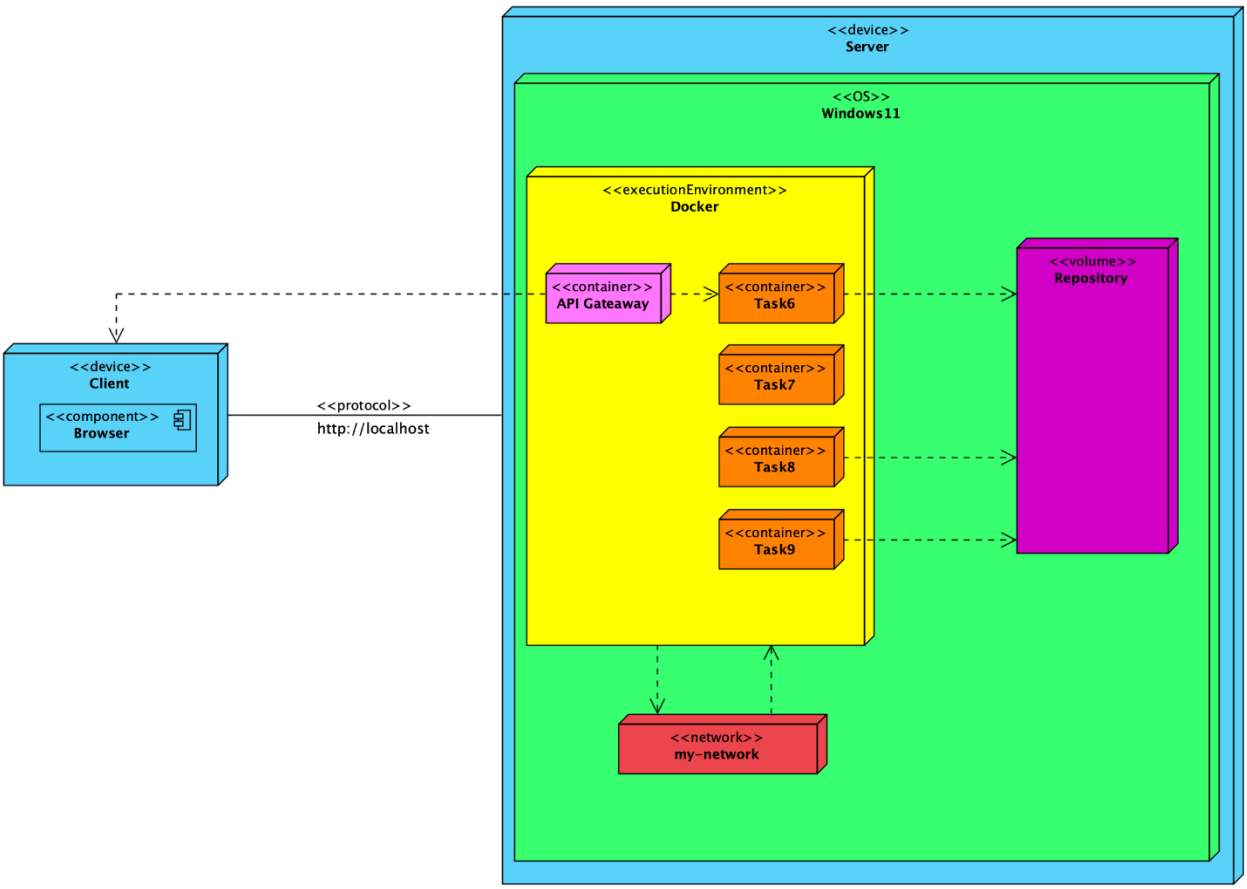


Nel diagramma sono stati indicati in blu i componenti che sono rimasti invariati nel processo di integrazione, in verde i componenti che erano già presenti prima dell'integrazione ma sono stati modificati, mentre in viola i componenti che sono stati aggiunti, infine in grigio è stato rappresentato il 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 e del task 8, 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.

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 (che è l'unico container a potersi interfacciare con gli altri che eseguono in Docker), mentre le altre al frontend.



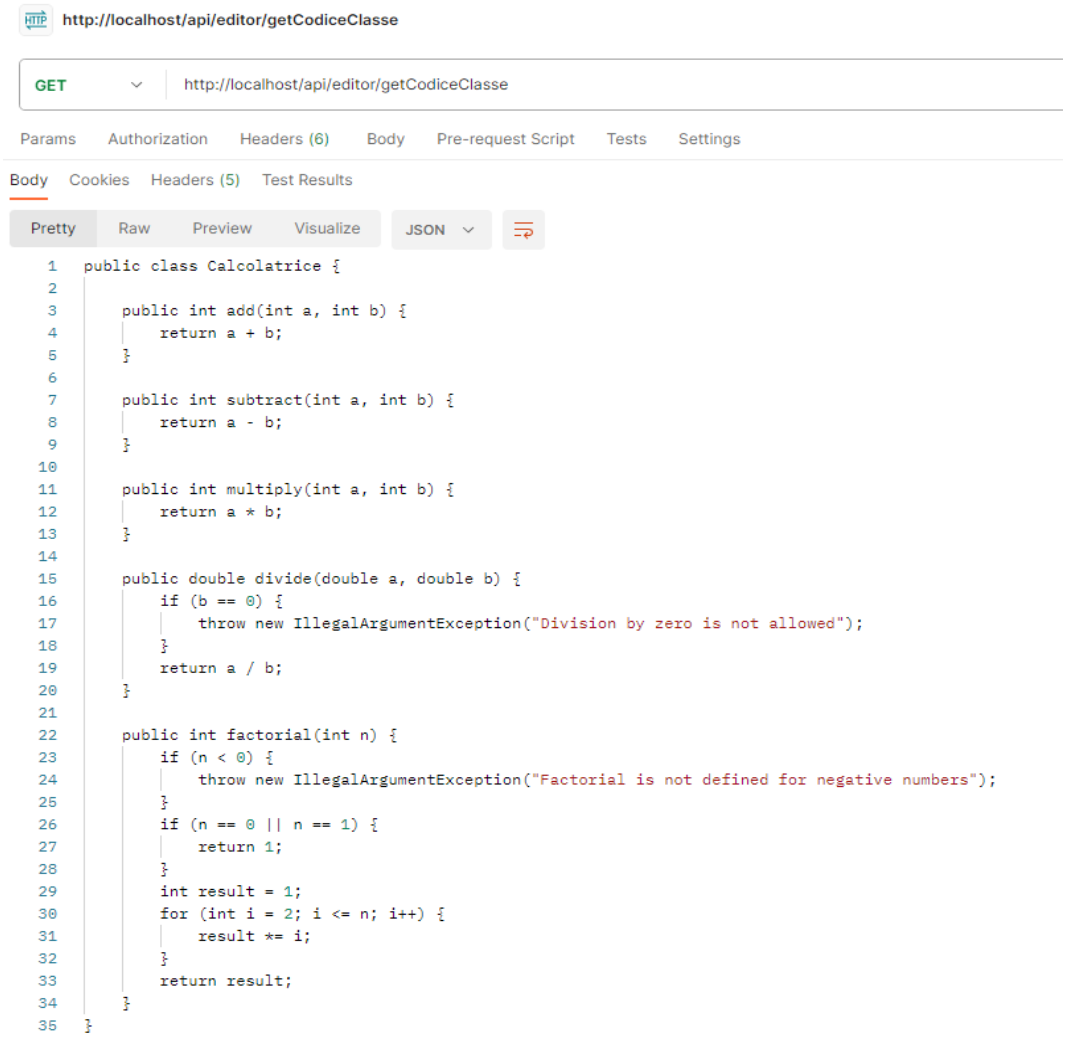
4 Documentazione API

Di seguito sono documentate le API utilizzate, queste sono anche consultabili al seguente link: https://restless-sunset-820309.postman.co/workspace/New-Team-Workspace~443703ce-70cd-4371-8bcf-e5e05bd4f9b8/collection/29616542-c98d8786-9f5d-4966-95af-4119e8ac247e?action=share&creator=29616542

4.1 getCodiceClasse

All'avvio dell'applicazione viene fatta la seguente richiesta GET dal browser (landing.js) per recuperare le informazioni riguardanti la partita corrente. In particolare il backend dopo aver gestito la richiesta e interrogato il ClassServer, risponde con il codice della classe sotto test, il quale sarà mostrato nella class window dell'editor.
L'oggetto partita presenta i seguenti campi:

- idGiocatore: identificativo del giocatore
- idPartita: valore identificativo della partita corrente
- nomeClasse: il nome della classe sotto test (con .java)
- codiceClasse: il codice della classe sotto test (verrà mostrato nella class window)
- idRobot: codice del robot con il quale si sta giocando
- codiceTest: codice della classe di test che deve scrivere l'utente, è inizializzato con un template
- livello: livello scelto della partita, ogni classe ha più livelli possibili
- robot: "Randoop" o "EvoSuite" a seconda del robot col quale si vuole giocare
- coverageMethod: scelta di quale metodo di coverage utilizzare "JaCoCo" o "EvoSuite"



4.2 getCoverage

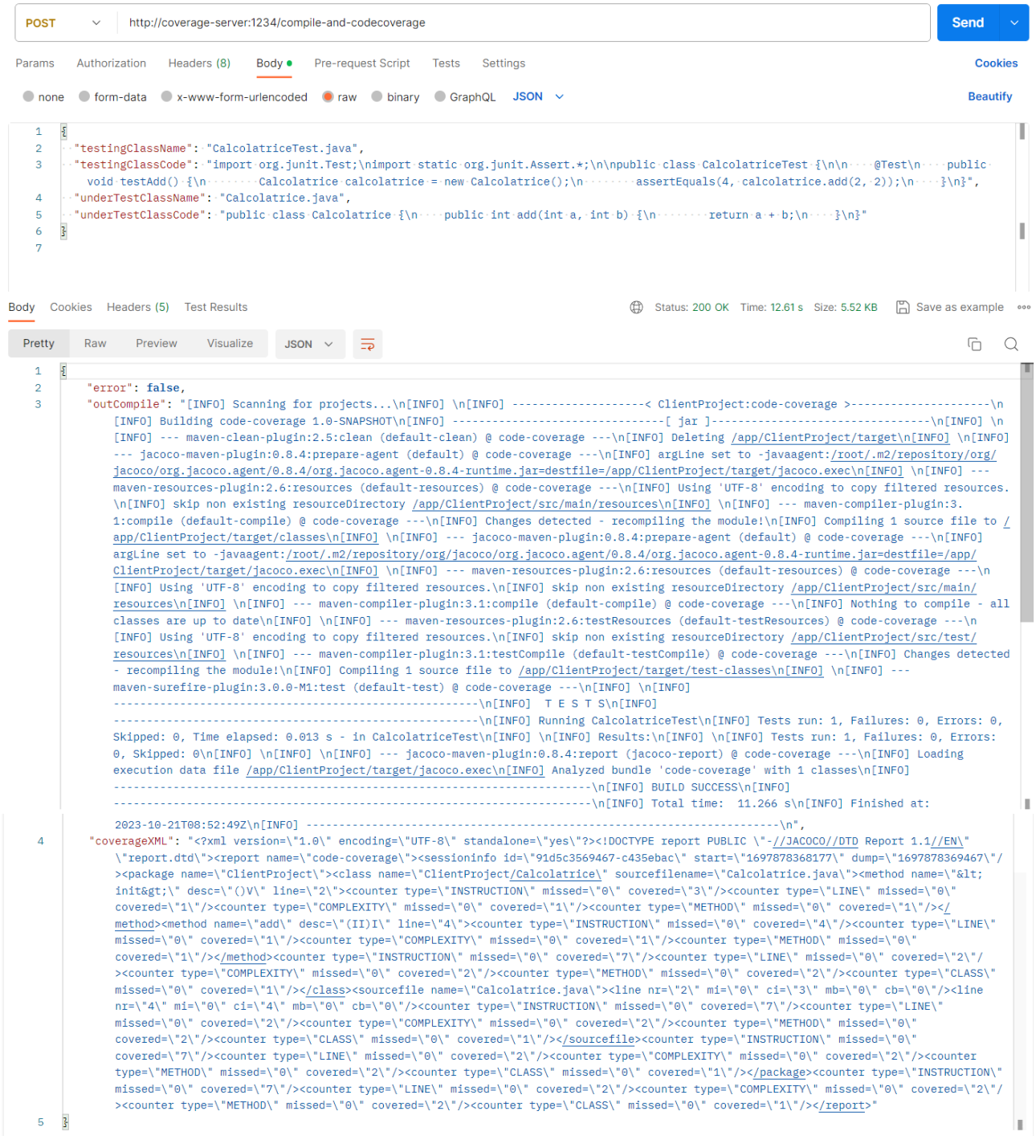
Questa API richiede il servizio di Compilazione e copertura di utente e robot.
Nel body di richiesta va indicato il nome della classe di test, mentre nel body di risposta avremo: informazioni di compilazione della classe test utente, di copertura del test dell'utente (sia XML che CSV) e il file di copertura del test del robot (XML o CSV) per la classe sotto test.

In base al valore del parametro coverageMethod in ClassServer.js viene modificato l'URL al quale viene fatta la successiva richiesta di misurazione della coverage utente, quindi essa verrà richiesta a JaCoCo o EvoSuite.

4.3 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 (coverageMethod=JaCoCo in ClassServer.js).

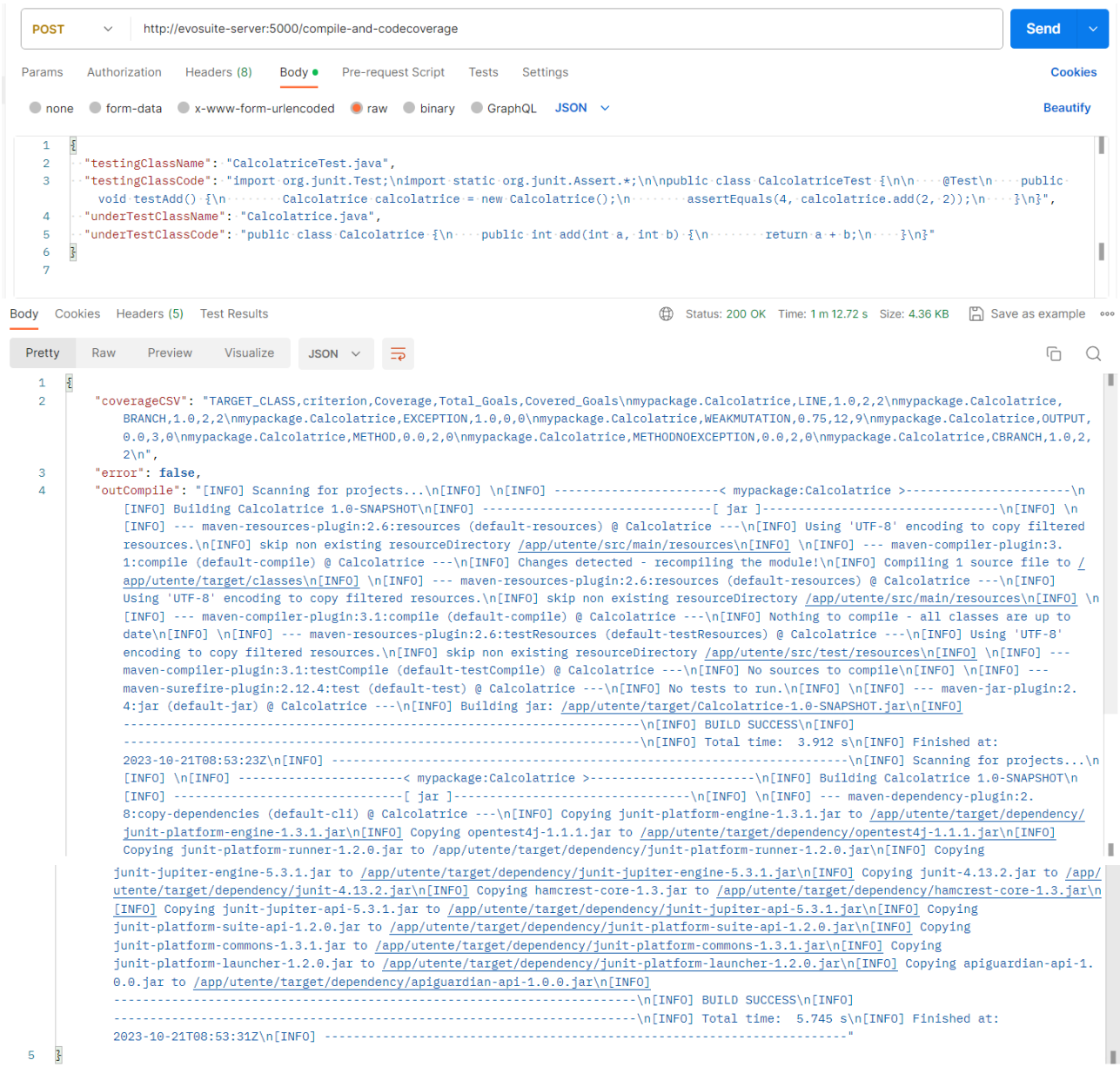
Nel body della richiesta vengono passati 4 parametri: nome e codice della classe di test e della classe sotto test. Nel body di risposta abbiamo l'output di compilazione, un flag che ci dice se ci sono stati errori di compilazione e il campo coverage sottoforma di xml (campi dell'oggetto ResponseDTO).



4.4 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 EvoSuite (coverageMethod=EvoSuite in ClassServer.js).

I campi del body della richiesta sono gli stessi di ***4.3***, 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
  Postcondizioni: Si conosce il vincitore della partita

- 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
  Postcondizioni: Si conosce 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"
  Postcondizioni: Il calcolo della copertura con JaCoCo non è completato con successo. L'utente può tornare a editare la classe di test

- 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"
  Postcondizioni: Il calcolo della copertura con EvoSuite non è completato con successo. L'utente può tornare a editare la classe di test

- Test5:
  Descrizione: Compilazione, Esecuzione e Calcolo Coverage di una classe di test con errori del robot (sia per Randoop che EvoSuite)
  Precondizioni: non è stata calcolata la coverage del robot per quel livello
  Input: clicca sul bottone "Play"
  Output atteso: viene mostrato un messaggio di errore: "Errore robot"
  Postcondizioni: Il recupero della copertura del robot e il confronto non vanno a buon fine

- Test6:
  Descrizione: Generazione dei test Randoop
  Precondizioni: Sono state aggiunte nuove classi nel repository
  Input: Avvio dell'applicazione
  Output atteso: Vengono generati i livelli contenenti test e coverage per ogni classe non ancora testata
  Postcondizioni: I dati prodotti sono salvati nel repository condiviso

- Test7:
  Descrizione: Generazione dei test EvoSuite
  Precondizioni: Sono state aggiunte nuove classi nel repository
  Input: *./app/robot/evosuite_test_script.sh*
  Output atteso: Vengono generati i livelli contenenti test e coverage per ogni classe non ancora testata
  Postcondizioni: I dati prodotti sono salvati nel repository condiviso

- Test8:
  Descrizione: Generazione dei test EvoSuite con errori nelle classi sotto test (vale anche per Randoop)
  Precondizioni: Sono state aggiunte nuove classi con errori sintattici nel repository
  Input: *./app/robot/evosuite_test_script.sh*
  Output atteso: Errore nella generazione (per le classi con errori)
  Postcondizioni: Non sono generati i test per le classi sotto test che presentano errori, mentre sono salvati nel repository i dati prodotti dalle classi senza errori

- Test9:
  Descrizione: Misurazione della coverage tramite EvoSuite dei test generati da Randoop
  Precondizioni: Sono stati generati i test Randoop per le classi nel repository
  Input: esecuzione di /app/utente/*randoop_coverage.py*
  Output atteso: Vengono generati i file di coverage CSV per ogni livello
  Postcondizioni: I file di coverage CSV per ogni test Randoop sono salvati nel repository

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	*./app/robot/evosuite_test_script.sh*	Vengono generati i livelli contenenti test e coverage	PASS
Test8	*./app/robot/evosuite_test_script.sh*	Errore nella generazione (per le classi con errori)	PASS
Test9	./randoop_coverage	Vengono generati i file di coverage CSV per ogni livello	PASS

5.2 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 (Reverse 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 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. Ciò ha anche permesso di popolare il repository con classi di test utilizzabili poi durante il gioco.
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 quindi realizzato uno script python *"report/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;

Si osservi che nel report Randoop i risultati di coverage sono compresi nell'intervallo [0-100] e rappresentano quindi un valore percentuale. Nel report EvoSuite invece la coverage è compresa nell'intervallo [0-1], quindi andrebbe moltiplicato per 100 per ottenere un valore percentuale.

6.1 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.

6.1.1 Misurazione coverage con Emma

Questo è il metodo standard utilizzato da Randoop (in questa implementazione) per misurare la coverage dei test generati, In questo caso la copertura è valutata con EMMA, questo genera un file xml contente alcune metriche di copertura, quelle che a noi interessano sono BLOCK e LINE.
Per la generazione di questo report di Randoop è stato utilizzato uno script python *"report/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.

6.1.2 Misurazione coverage con EvoSuite

È possibile anche misurare la coverage dei test Randoop utilizzando il misuratore di EvoSuite, in questo modo misuriamo la coverage di questi test utilizzando le stesse metriche di EvoSuite, ovvero WAEKMUTATION e CBRANCH, quindi sarà possibile confrontare direttamente le classi di test generate dai due robot e vedere generalmente quale si comporta meglio.

Prima di passare al report vero e proprio è necessario fare alcune osservazioni circa la struttura dei file Randoop. Le cartelle dei test Randoop contengono file del tipo:
”RegressionTest0_it0_livello1_it1_livello2.java”
“RegressionTest0_it1_livello2.java”
“RegressionTest1_it1_livello2.java”
“RegressionTest_it0_livello1_it1_livello2.java”
“RegressionTest_it1_livello2.java”
Alcuni di questi sono TEST SUITE, che raccolgono gli altri. È possibile distinguere le test suite dal fatto che non hanno una cifra dopo la parola "Test", in questo caso, “RegressionTest_it0_livello1_it1_livello2” e “RegressionTest_it1_livello2” sono test suite.
Ad ogni test suite corrisponde una serie di test, che è possibile riconoscere aggiungendo cifre dopo "Test" nel nome della test suite, ad esempio “RegressionTest_it0_livello1_it1_livello2” usa come test “RegressionTest0_it0_livello1_it1_livello2” e nient'altro, mentre “RegressionTest_it1_livello2” usa come test “RegressionTest0_it1_livello2” e “RegressionTest1_it1_livello2”.

Per far si che questi test Randoop fossero effettivamente utilizzabili dal misuratore EvoSuite, è stato quindi necessario effettuare delle elaborazioni. In particolare è stato realizzato uno script python *“/app/utente/randoop_coverage.py”*. Il compito dello script è:

1. Correggere ogni public class <nome> {} all'interno dei test con il nome corretto del file
2. All'interno di ogni test suite, importare correttamente i test (i nomi di default sono errati)
3. Creare un file test suite che racchiude tutte le test suite
4. Compilare e misurare la coverage della test suite completa con EvoSuite (utilizzando gli script già visti in precedenza *“./compilazione_test.sh”* e *“./robot_misurazione_utente.sh”*)

Dopo aver eseguito questo script, per ciascuna classe del repository, avremo un file “statistics.csv” in ogni livello di Randoop (repository/”NomeClasse”/RobotTest/RandoopTest/0xLevel/statistics.csv), questo file conterrà le misure di coverage relativamente alle metriche richieste.

Un volta prodotti i file csv è stato possibile generare il report grazie ad un semplice script *“report/randoop_report_evosuite_coverage.py”*, il quale esegue le stesse operazioni dello script usato per il report precedente, le uniche differenze riguardano il fatto che le informazioni sono estratte da un file csv (invece che xml), e che le metriche di copertura di interesse sono WAEKMUTATION e CBRANCH (invece di LINE e BLOCK)
Di seguito è mostrato l’output del report (contenuto anche nel file randoop_report_evosuite_coverage.txt)

ByteArrayHashMap: TEST PASSED
Weak mutations: [0.6453, 0.8486, 0.8486]
Cbranches: [0.5542, 0.7108, 0.7229]

ByteVector: TEST PASSED
Weak mutations: [0.6969]
Cbranches: [0.4783]

Calcolatrice: TEST PASSED
Weak mutations: [0.9908]
Cbranches: [1.0]

FontInfo: TEST PASSED
Weak mutations: [0.4591]
Cbranches: [0.7333]

FTPFile: TEST PASSED
Weak mutations: [0.8736, 0.8989, 0.9157, 0.9354, 0.9494]
Cbranches: [0.6308, 0.6615, 0.6769, 0.7077, 0.7231]

HierarchyPropertyParser: TEST PASSED
Weak mutations: [0.4458, 0.4717, 0.5283]
Cbranches: [0.5455, 0.5909, 0.6091]

HSLColor: TEST PASSED
Weak mutations: [0.8374, 0.8416, 0.849]
Cbranches: [0.8169, 0.8873, 0.9014]

ImprovedStreamTokenizer: TEST PASSED
Weak mutations: [0.3168]
Cbranches: [0.102]

ImprovedTokenizer: TEST PASSED
Weak mutations: [0.8759, 0.9087, 0.9204]
Cbranches: [0.7778, 0.8025, 0.8025]

Inflection: TEST PASSED
Weak mutations: [0.5479]
Cbranches: [0.7037]

OutputFormat: TEST PASSED
Weak mutations: [0.8603]
Cbranches: [0.791]

ParameterParser: TEST PASSED
Weak mutations: [0.5277, 0.5566, 0.5566, 0.5566]
Cbranches: [0.4568, 0.5309, 0.5556, 0.5679]

Range: TEST FAILED, file not found

RationalNumber: TEST PASSED
Weak mutations: [0.5136, 0.7557, 0.7738, 0.7919]
Cbranches: [0.3676, 0.6029, 0.6176, 0.6471]

ResultSetColumnReader: TEST PASSED
Weak mutations: [0.0183]
Cbranches: [0.0125]

ResultSetHelper: TEST PASSED
Weak mutations: [0.2143]
Cbranches: [0.4962]

StringParser: TEST PASSED
Weak mutations: [0.2628]
Cbranches: [0.3718]

SubjectParser: TEST PASSED
Weak mutations: [0.3631]
Cbranches: [0.3226]

TimeStamp: TEST PASSED
Weak mutations: [0.9295]
Cbranches: [0.9302]

VCardBean: TEST PASSED
Weak mutations: [0.0873, 0.0952, 0.746]
Cbranches: [0.2679, 0.2857, 0.6696]

WeakHashtable: TEST PASSED
Weak mutations: [0.5714, 0.6522, 0.6584]
Cbranches: [0.3924, 0.4684, 0.481]

XmlElement: TEST PASSED
Weak mutations: [0.3955, 0.5159, 0.5568, 0.5795, 0.5886]
Cbranches: [0.4421, 0.5789, 0.6421, 0.6947, 0.7053]

XMLParser: TEST PASSED
Weak mutations: [0.1597]
Cbranches: [0.225]

TEST PASSED : 22
TEST FAILED : 1

Per la classe Range c'è stato un errore in fase di calcolo della coverage tramite EvoSuite (test/java/mypackage/RegressionTest0_it1_livello2.java:56: error: ComparableComparator has private access in Range\n   org.junit.Assert.assertTrue("\'" + comparator_b10 + "\' != \'" + Range.ComparableComparator.INSTANCE + "\'", comparator_b10.equals(Range.ComparableComparator.INSTANCE));\n), ciò ha portato alla mancata creazione dei file csv e quindi all'errore nel report.
Ovviamente vale lo stesso ragionamento del caso precedente per quanto riguarda il numero di livelli generati.

6.2 Report EvoSuite

Per avviare la generazione dei test EvoSuite è necessario lanciare il comando *./app/robot/robot_generazione.sh* con i parametri corretti per ciascuna classe, per evitare questa operazione ripetitiva è stato realizzato uno script */app/robot/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 non esiste allora lancia il comando *./app/robot/robot_generazione.sh* seguito dai parametri per quella classe, altrimenti la salta (già testata). 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. 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 *"report/evosuite_report.py"* per la creazione del report dei test generati. Questo script è molto simile a quello utilizzato nel caso precedente, infatti estrae da un file CSV le informazioni riguardanti le 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.

7 Installazione ed Esecuzione

7.1 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.
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.

A questo punto il frontend sarà accessibile su ***http://localhost/***.

Nel caso in cui l'installazione non sia locale ma su un server remoto, sarà necessario modificare due variabili con il dominio o IP corretto:

- ***server_name*** in nginx/nginx.con
- ***DOMAIN_NAME*** in T6-G8-main/codice/editor/src/components/Landing.js

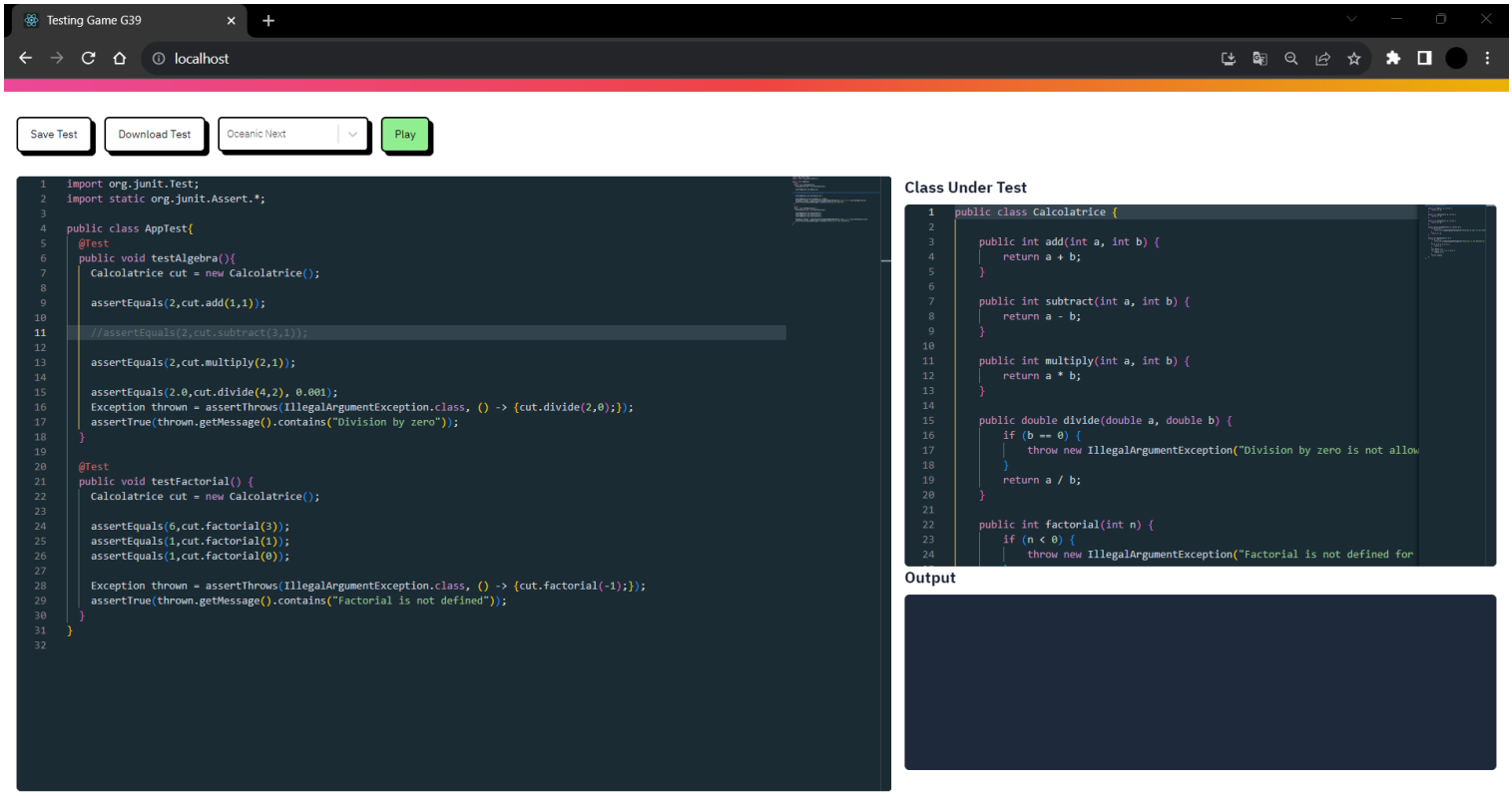
Nel caso in cui si volesse aggiungere una nuova classe da testare bisognerà svolgere i seguenti step:

1. l'admin deve inserire il file .java nel repository rispettando la struttura del Filesystem specificata in ***repository/FolderTree_v1.txt***
2. Rilanciare il comando: ***docker compose up –build -d***
3. I test del robot Randoop e le rispettive coverage saranno automaticamente generati
4. Per aggiungere ai test Randoop le misure di coverage valutate con EvoSuite è necessario recarsi nel container evosuite-server ed eseguire lo script python:
   - cd /app/utente/
   - python3 randoop_coverage.py
5. Per avviare la generazione automatica dei test EvoSuite per tutte le classi del repository, recarsi nel terminale del container evosuite-server ed eseguire lo script:
   /app/robot/evosuite_test_script.sh
6. Nel caso in cui si invece voglia lanciare la generazione dei test EvoSuite per una specifica classe è possibile eseguire all'interno del container evosuite-server i seguenti comandi:
   - cd /app/robot
   - CLASS_NAME=<nomeclasse>
   - ./robot_generazione.sh $CLASS_NAME ${CLASS_NAME}SourceCode
   /repository/${CLASS_NAME}/${CLASS_NAME}SourceCode <numero livelli>

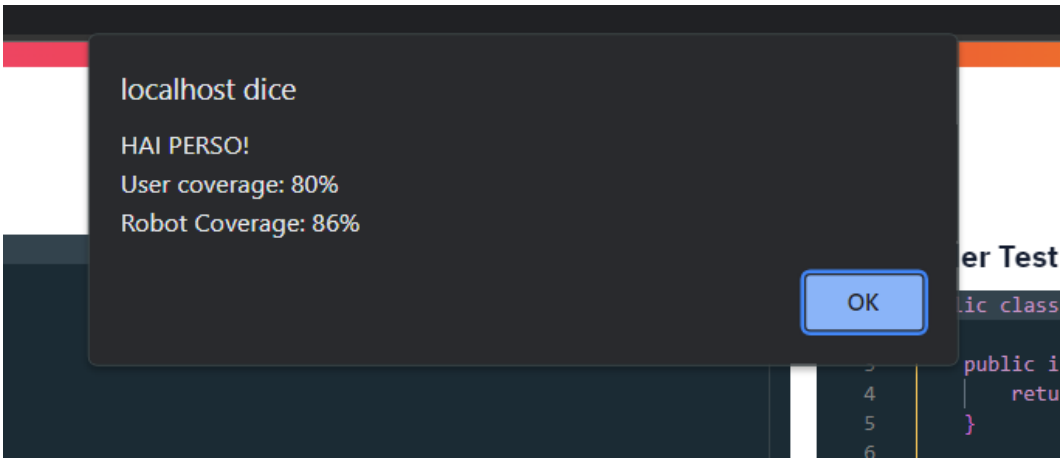
7.2 Prova di esecuzione

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:

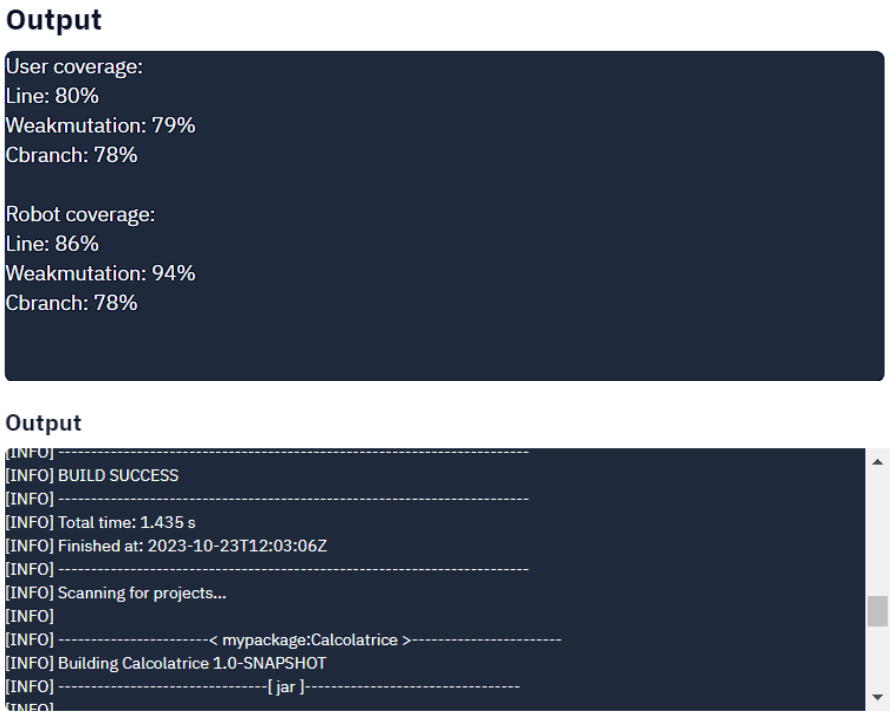
NB: a titolo di esempio è stato commentato il test della funzione subtract()



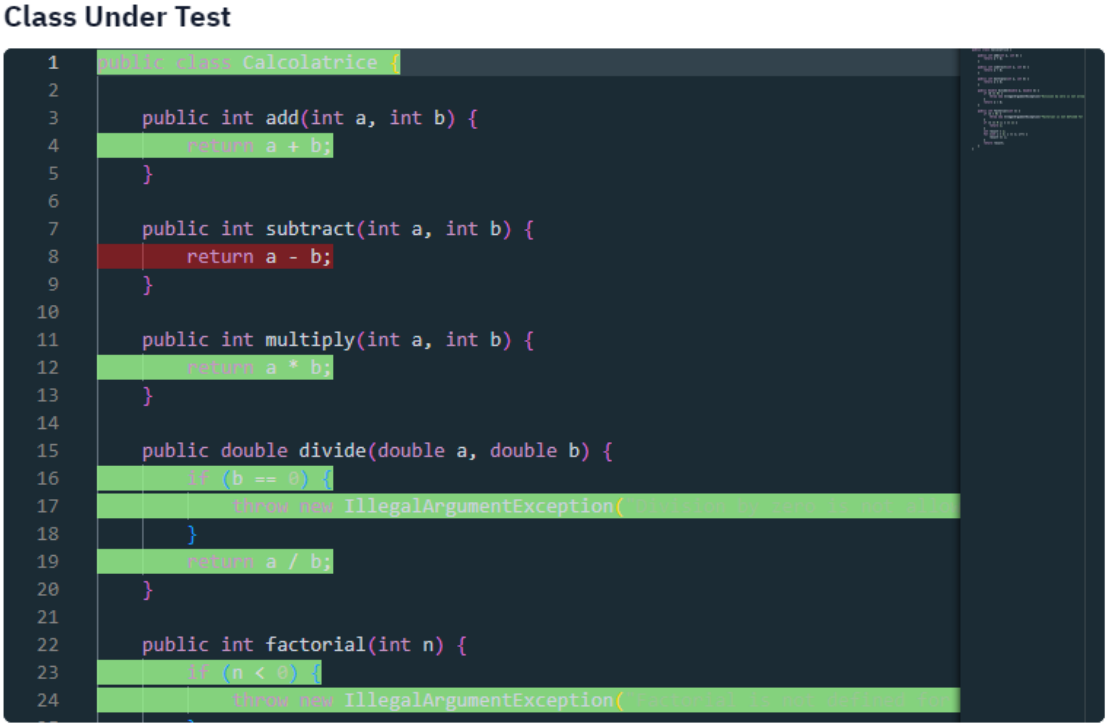
Nell'editor di testo è già presente un template per la classe under test, quindi sarà possibile modificare la classe di test oppure passare direttamente alla compilazione.
Cliccare quindi il bottone "Play", al termine della compilazione, esecuzione, e confronto del test con quello del robot, verrà mostrato 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, in particolare in questo caso sono mostrate le seguenti metriche (cambiano se cambia il coverageMethod):



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



8 Sviluppi Futuri

E’ possibile in futuro migliorare ed estendere l’applicazione sotto diversi aspetti:

1. Il backend non ha sessione, ciò non permette l’utilizzo concorrente della web app
2. Miglioramento sicurezza dell’applicazione
3. Integrazione con gli altri task di creazione della partita
4. Velocizzare la compilazione con misuratore EvoSuite
5. Migliorare il criterio di generazione dei livelli di Randoop ed EvoSuite