Documentazione SAD – G39

FRANCESCO CIRILLO	M63001491
ANGELO BARLETTA	M63001507
GIUSEPPE BUONOMANO	M63001506

Indice

1 Introduzione 3
1.1 Approccio adottato 3
1.2 Iteration Planning 3
2 Specifica dei Requisiti 4
2.1 Storie Utente 4
2.2 Requisiti funzionali 4
2.3 Requisiti non funzionali 4
2.4 Requisiti sui dati 5
2.5 Diagramma dei casi d'uso 5
2.6 Scenari dei casi d'uso 6
2.6.1 Compila ed Esegui 6
2.6.2 Confronto Risultati 6
2.6.3 Recupero dati dal repository 6
2.6.4 Generazione Robot Test 7
2.7 Glossario dei termini 7
3 Integrazione 8
3.1 Modifiche apportate ai Task 8
3.1.1 Task 6 8
3.1.2 Task 7 8
3.1.3 Task 8 8
3.1.4 Task 9 8
3.2 Diagrammi di Sequenza 9
3.2.1 Compila ed Esegui 9
3.2.2 Recupero dati dal repository 10
3.2.3 Confronto Risultati 11
3.2.4 Generazione Robot Test 12
3.3 Diagramma di Attività 13
3.3.1 Compila ed Esegui 13
3.3.2 Recupero dati dal repository 14
3.3.3 Confronto Risultati 14
3.3.4 Generazione Robot Test 15
3.3.5 Flusso di attività completo 15
3.4 Component Diagram 16
3.5 Deployment Diagram 17
4 Testing 18

5 Installazione ed Esecuzione 20
5.1 Installazione 20
5.2 Prova di esecuzione 21

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

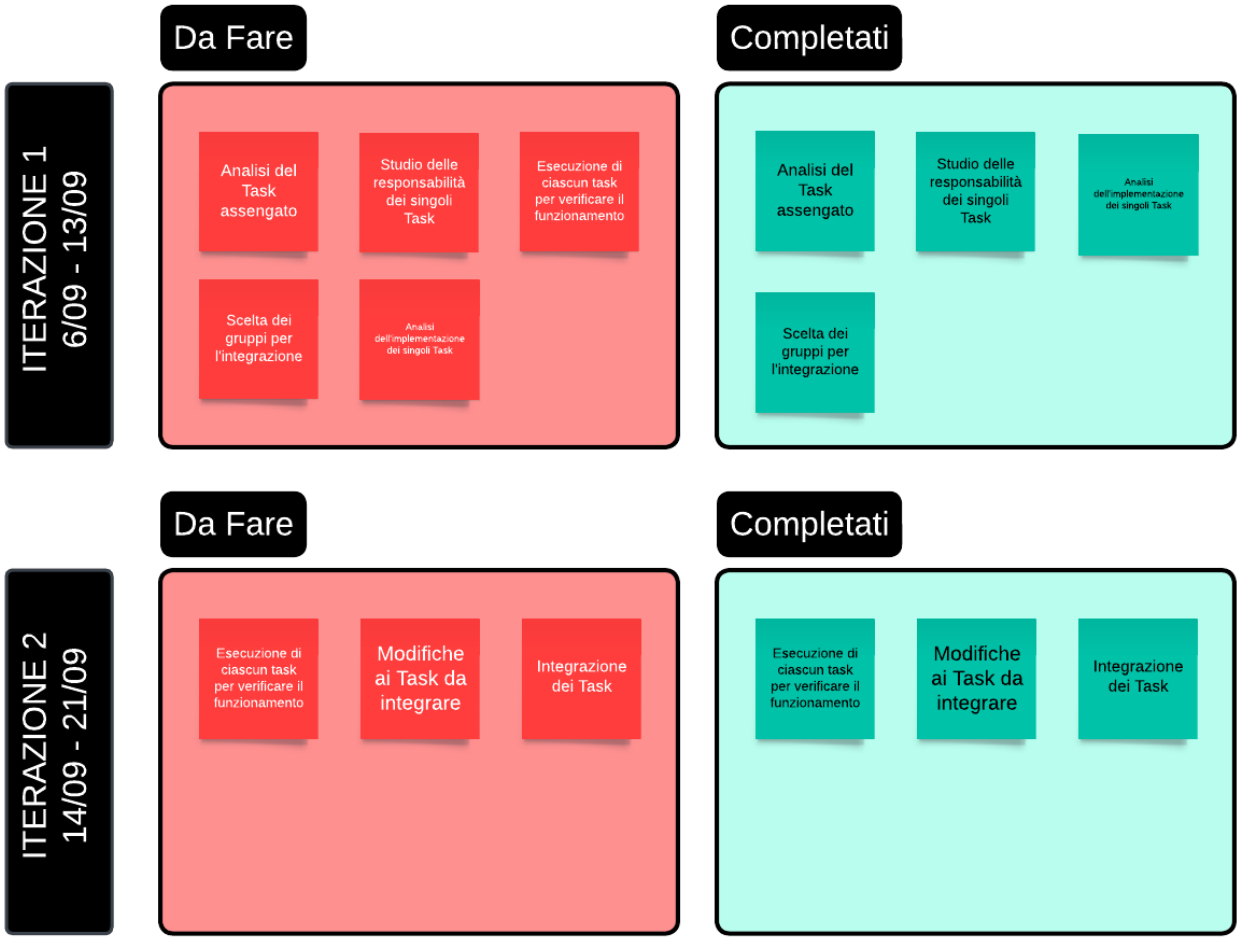
Per ognuno dei task abbiamo scelto lo specifico gruppo dopo aver consultato le documentazioni. Ad esempio per quanto riguarda T6 e T7 i gruppi G8 e G31 si erano precedentemente accordati per fornire interfacce compatibili, che hanno reso l'integrazione più semplice.

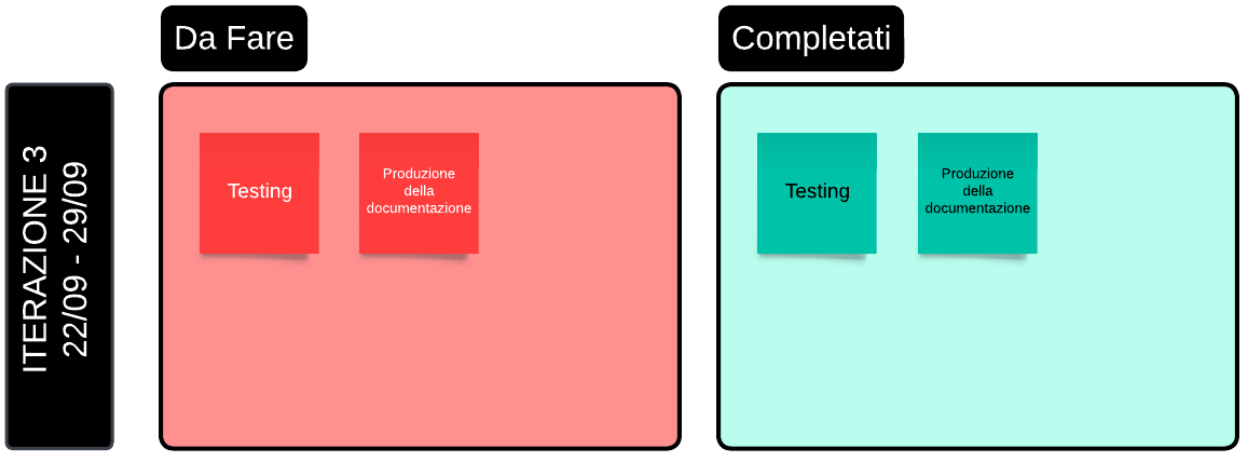
1.1 Approccio adottato

Per lo sviluppo è 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

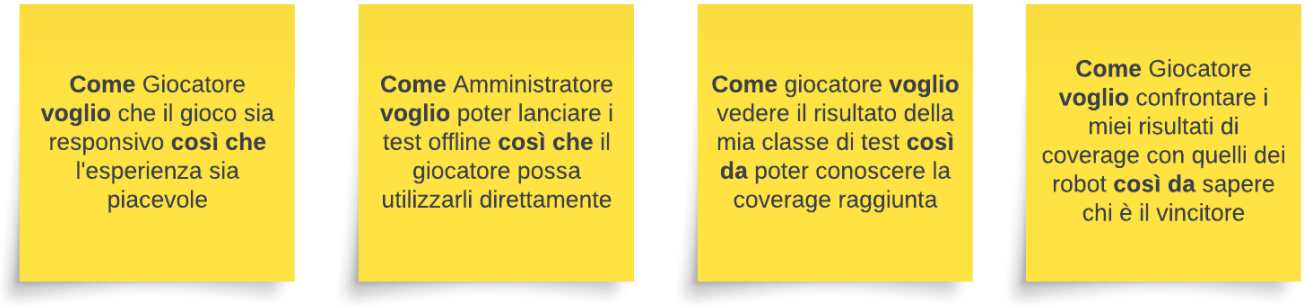




2 Specifica dei Requisiti

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

2.3 Requisiti non funzionali

I requisiti non funzionali descrivono le proprietà del sistema:

4. Portabilità
5. Compatibilità
6. Scalabilità
7. Facilità di deploy
8. 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.

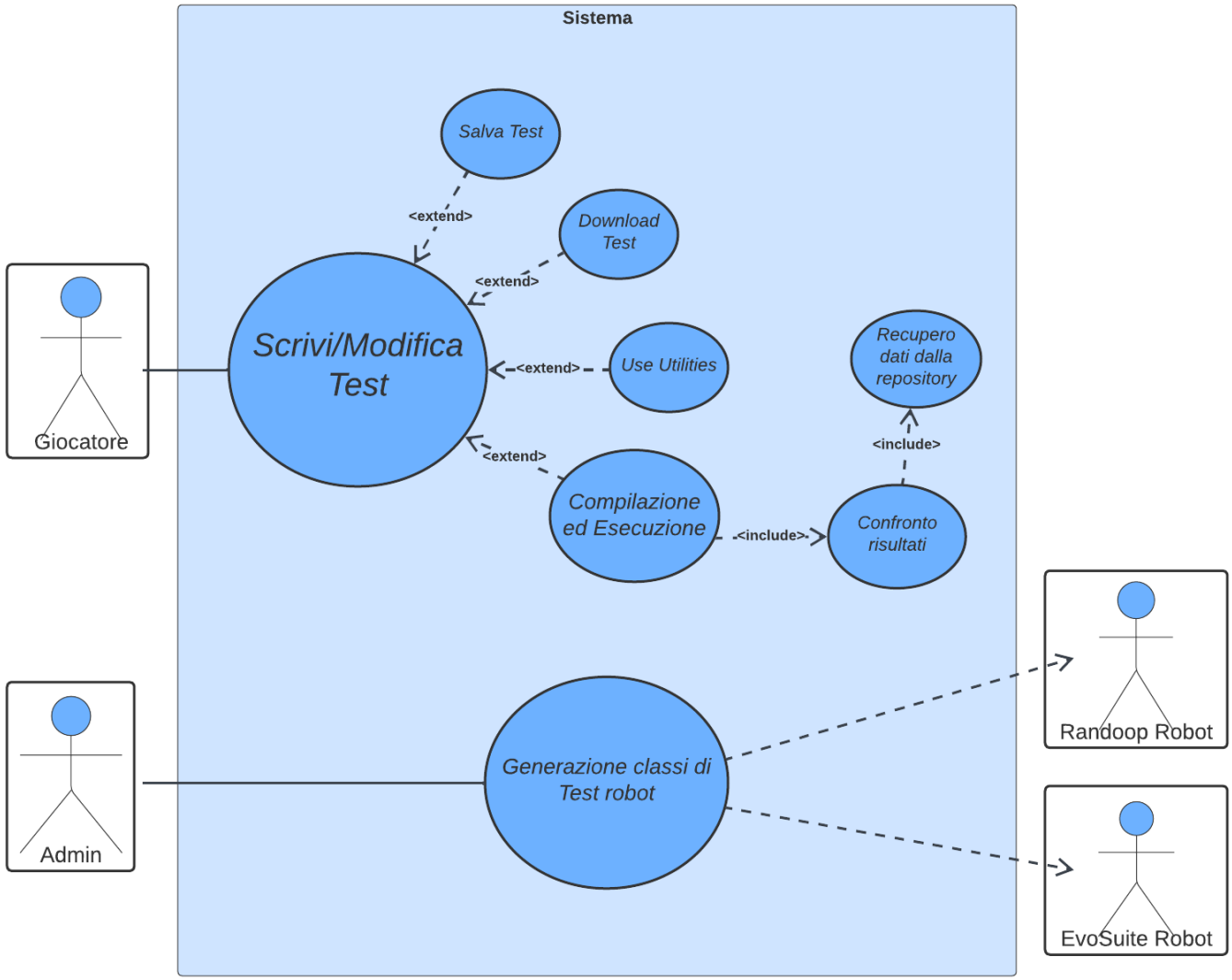
2.4 Requisiti sui dati

9. Le classi di test devono essere disponibili prima dell'avvio della partita
10. 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



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 Compila ed Esegui

Caso d'uso	Compila ed Esegui
Attore primario	Giocatore
Attore secondario	-
Descrizione	Il giocatore avvia la compilazione e l'esecuzione della propria classe di test per ottenere la copertura relativa alla classe sotto test
Pre-condizioni	La partita è stata correttamente caricata
Sequenza eventi principale	1. Il caso d'uso inizia quando il giocatore clicca il pulsante di compilazione 2. Il sistema farà una chiamata POST al server del compilatore che eseguirà il codice editato restituendo la Coverage e gli Output. 3. All'arrivo della risposta da parte del compilatore il sistema: • Prenderà la coverage e ne farà un parsing per evidenziare le righe di codice coperte. • Recupera la coverage del Robot e confronta i risultati 4. Il sistema mostra il vincitore e le coverage
Post-condizioni	Il giocatore visualizza l'esito della partita
Casi d'uso correlati	Confronto Risultati (include), Recupero dati dal repository (include)
Sequenza eventi alternativi	Errore in fase di compilazione del test

2.6.2 Confronto Risultati

2 Caso d'uso	Confronto Risultati
Attore primario	Giocatore
Attore secondario	-
Descrizione	Vengono confrontati i risultati di coverage dell'utente e di uno dei Robot
Pre-condizioni	La compilazione della classe di test dell'utente non ha generato errori
Sequenza eventi principale	1. Il sistema recupera i dati dal repository condiviso 2. Viene calcolata la copertura dell'utente 3. Viene confrontata la copertura del robot scelto con quella dell'utente
Post-condizioni	Si conosce il test con la coverage maggiore
Casi d'uso correlati	Recupero dati dal repository (include)
Sequenza eventi alternativi	-

2.6.3 Recupero dati dal repository

Caso d'uso	Recupero dati dal repository
Attore primario	Giocatore
Attore secondario	-
Descrizione	Vengono recuperati i file di copertura e la classi di test di uno dei Robot
Pre-condizioni	Sono stati generate ed eseguite le classi di test dei Robot La classe di test del giocatore non ha generato errori di compilazione
Sequenza eventi principale	1. Il sistema accede al repository condiviso 2. In base al Robot e al livello scelti, recupera la coverage corrispondente 3. Viene effettuato il parsing della coverage del robot scelto
Post-condizioni	Si hanno a disposizioni i dati per il confronto
Casi d'uso correlati	-
Sequenza eventi alternativi	-

2.6.4 Generazione Robot Test

Caso d'uso	Generazione Robot Test
Attore primario	Admin
Attore secondario	Randoop / EvoSuite
Descrizione	Vengono generate ed esguite le classi di Test dei Robot organizzate in livelli
Pre-condizioni	Esiste la classe per la quale si vogliono generare i test
Sequenza eventi principale	1. Randoop a. 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) b. Vengono eseguiti tali test per ottenere le coverage (EMMA) 2. Evosuite a. L'amministratore tramite shell esegue uno script per la generazione dei livelli (robot_generazione.sh) di una data classe b. Come argomenti vanno indicati la classe sotto test e il numero di livelli da generare 3. 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
Casi d'uso correlati	-
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

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: l'URL del coverage server permette ora di effettuare una richiesta http al Task 7 per la compilazione
- Aggiunta della classe RequestDTO utile ad interfacciarsi con il task 7
- 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 e sul livello scelti
- Modifica alla classe Coverage per memorizzare anche la coverage del Robot (Randoop o EvoSuite)
- Modifiche alla funzione getCoverage(): invia al task 7 un oggetto RequestDTO (invece di un oggetto Test); nel caso in cui non ci siano errori di compilazione viene prelevata la coverage del Robot definito in Partita relativamente alla classe sotto test in base al livello selezionato
- Modifica al landing.js che ora calcola le righe coperte dalla classe di test dell'utente, e confronta tali risultati con quelli del Robot, viene quindi mostrato un alert con il vincitore.
- 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.

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

- Viene utilizzato solo il container per la generazione dei test. L'altro container veniva utilizzato per ricevere richieste sulla copertura, tuttavia in seguito all'integrazione è diventato inutile dal momento che ci aspettiamo di accedere direttamente al repository condiviso per il recupero dei dati di coverage.
- 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)
- Ottimizzatio il processo di installazione del task 8 nel container
- Aggiunta di un DockerFile

3.1.4 Task 9

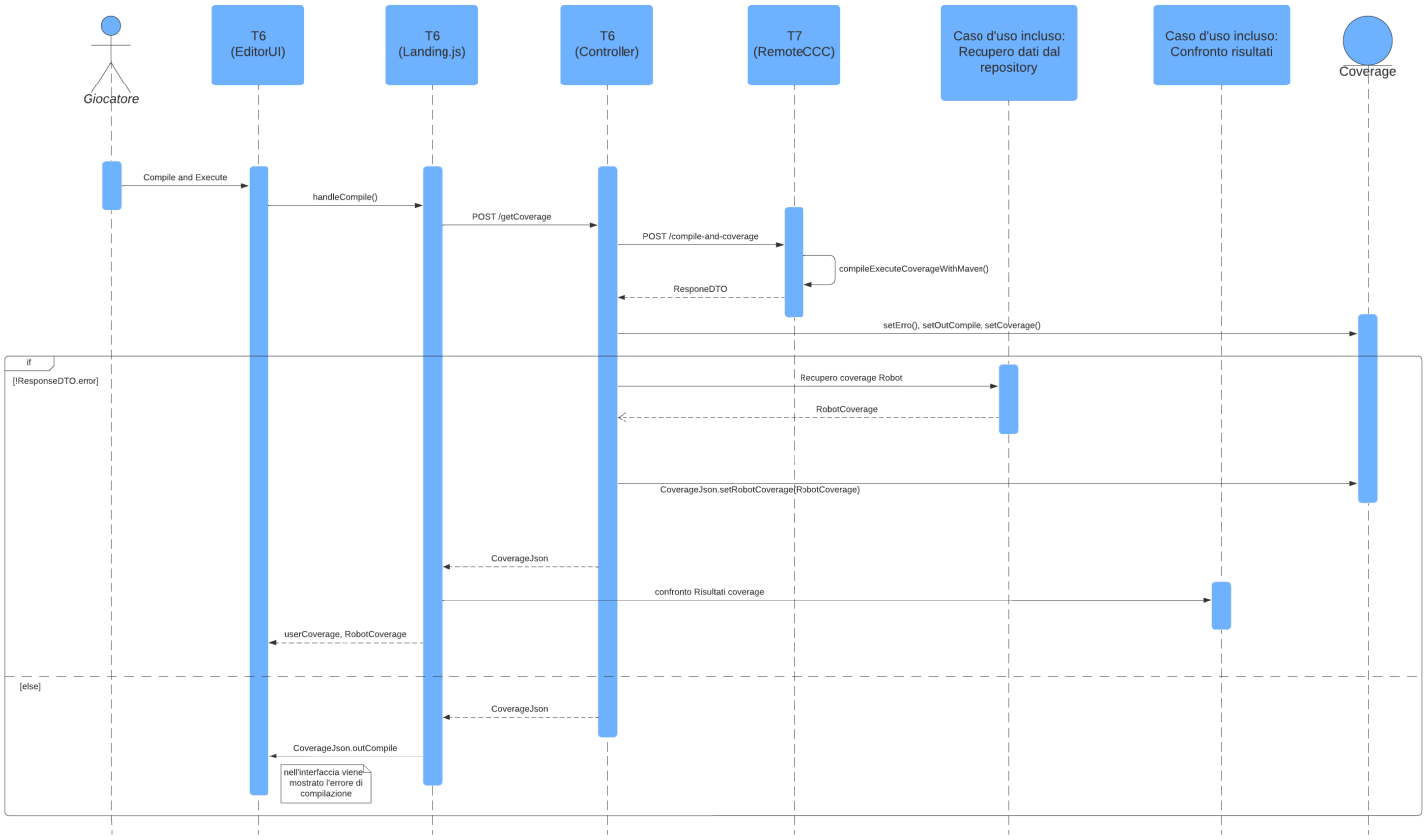
- Aggiunta del salvataggio dei file di copertura (in formato xml) per i livelli generati da Randoop
- Sostituzione del file Batch in un file Bash (per la generazione dei test e calcolo della copertura) e dei percorsi "/" in "\", ciò ci permette di eseguire il task su un container Docker
- Modifica al file pom.xml per l'aggiunta di una dipendenza JSoup

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.

3.2 Diagrammi di Sequenza

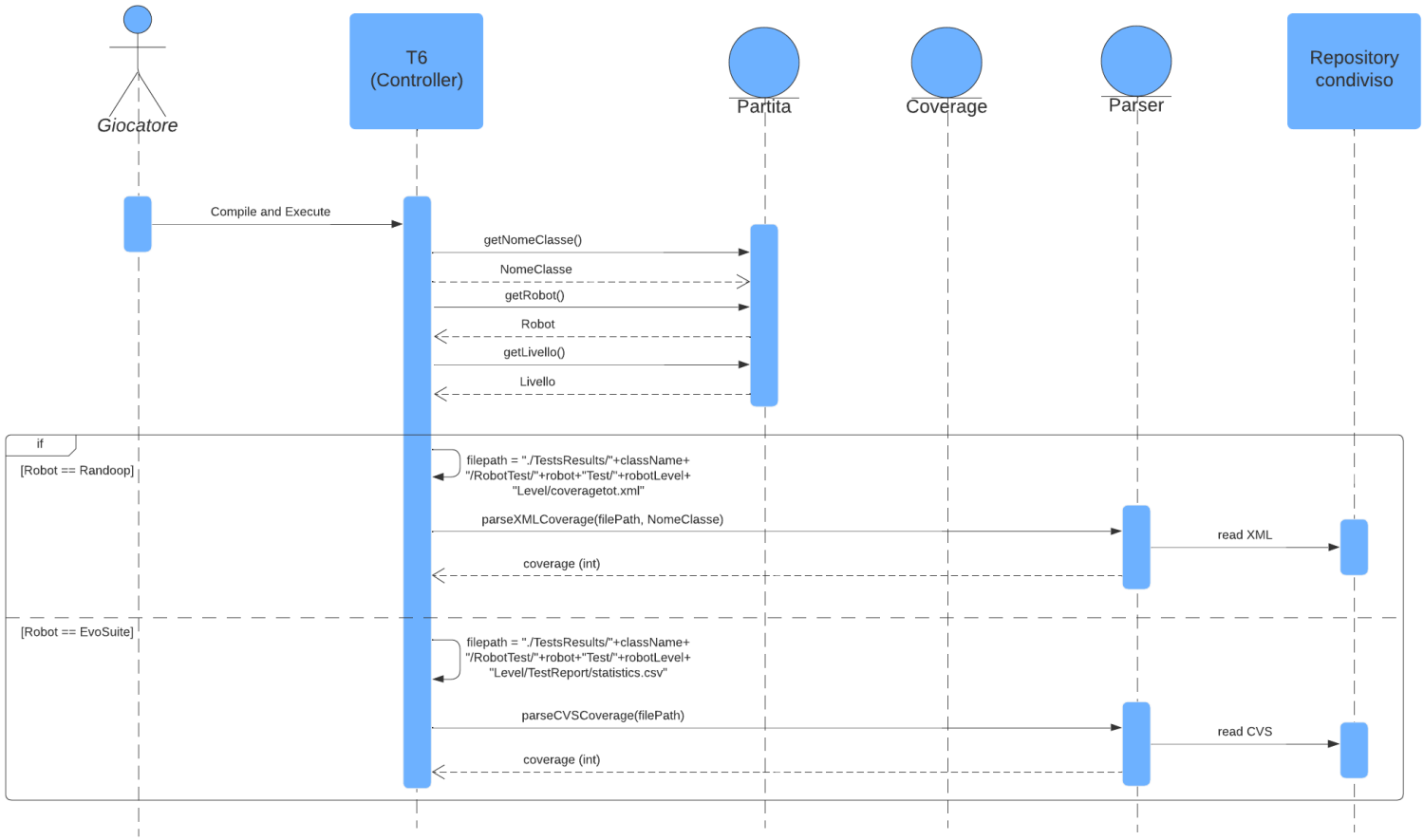
Si vogliono descrivere i diagrammi di sequenza al livello dell'interazione tra Task, rimandiamo quindi alle documentazioni dei singoli Task per diagrammi più dettagliati sulle diverse funzionalità.

3.2.1 Compila ed Esegui



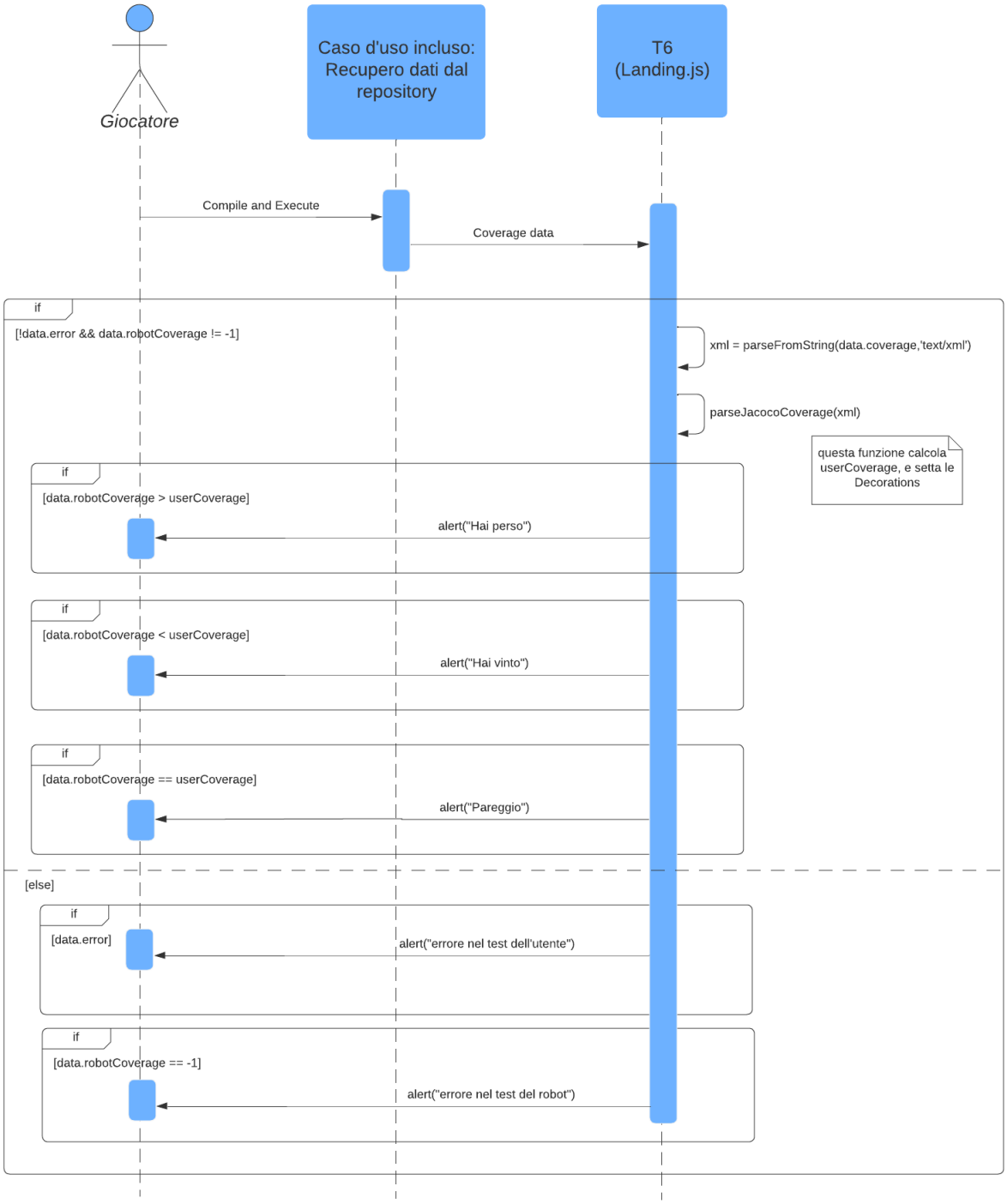
3.2.2 Recupero dati dal repository

Si consideri che il seguente caso d'uso è incluso nel caso d'uso "Compila ed Esegui", per questo motivo si è scelto di indicare come attore principale lo stesso che innesca il caso d'uso di partenza.

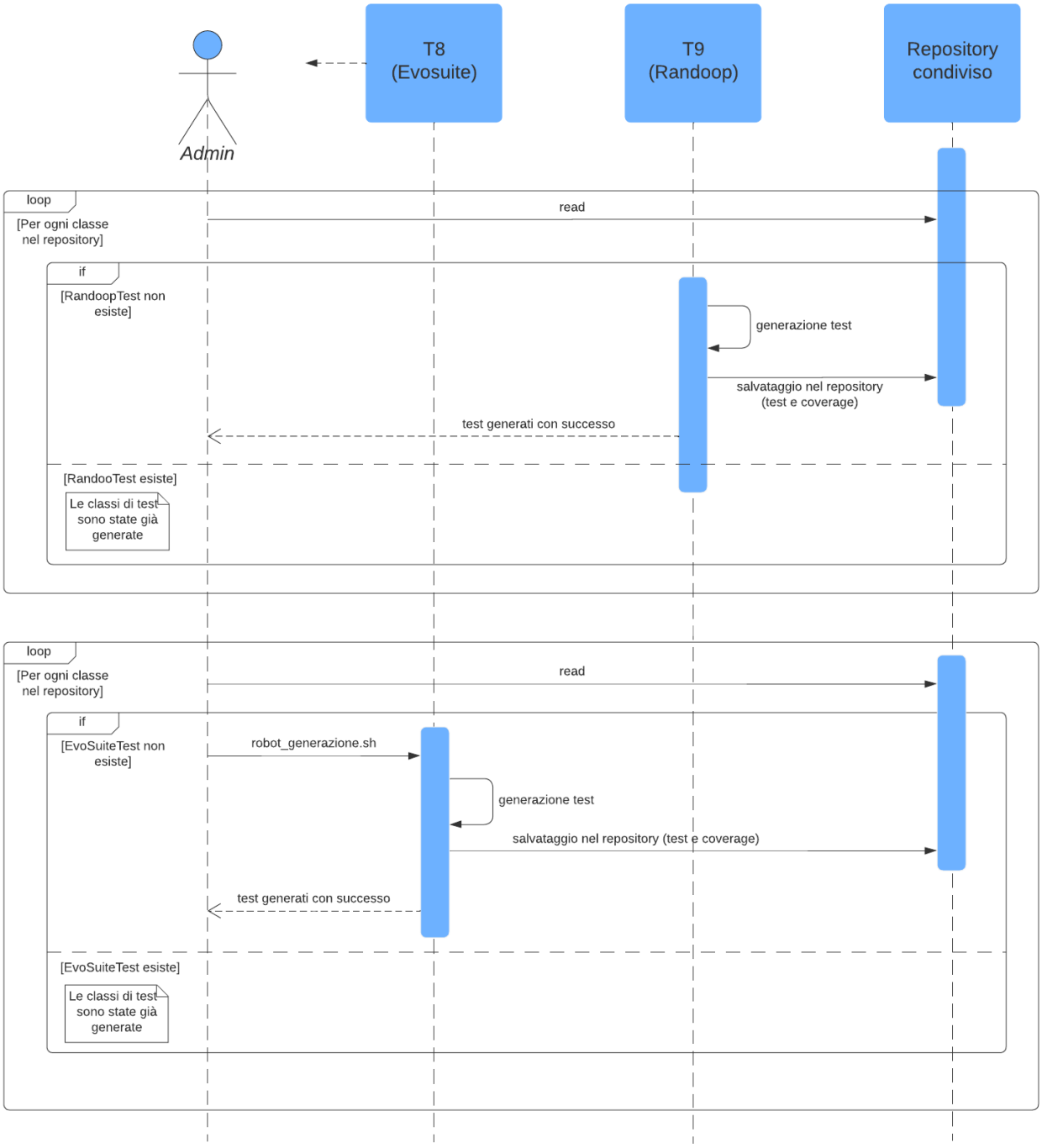


3.2.3 Confronto Risultati

Per questo caso d'uso vale lo stesso ragionamento effettuato per quello precedente, quindi è stato indicato come attore principale il Giocatore perché è quello che da il via al caso d'uso "Compila ed Esegui"



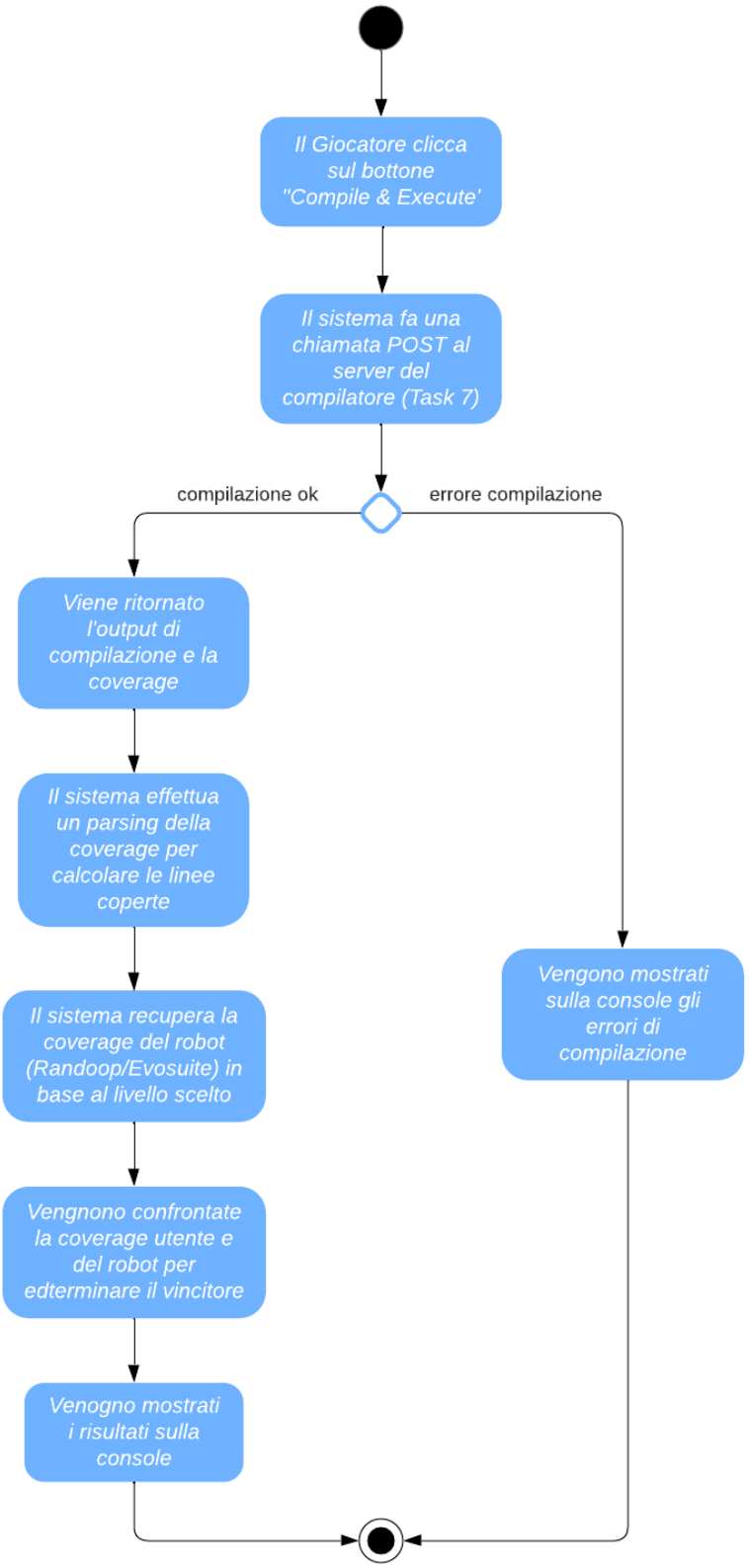
3.2.4 Generazione Robot Test



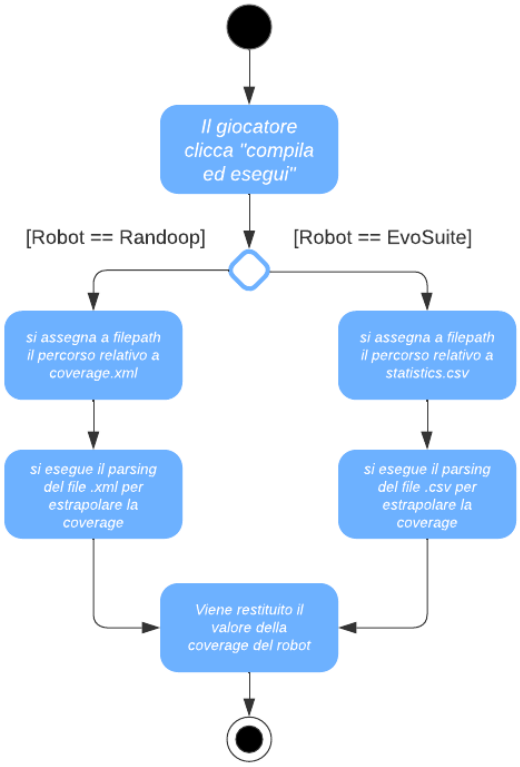
3.3 Diagramma di Attività

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

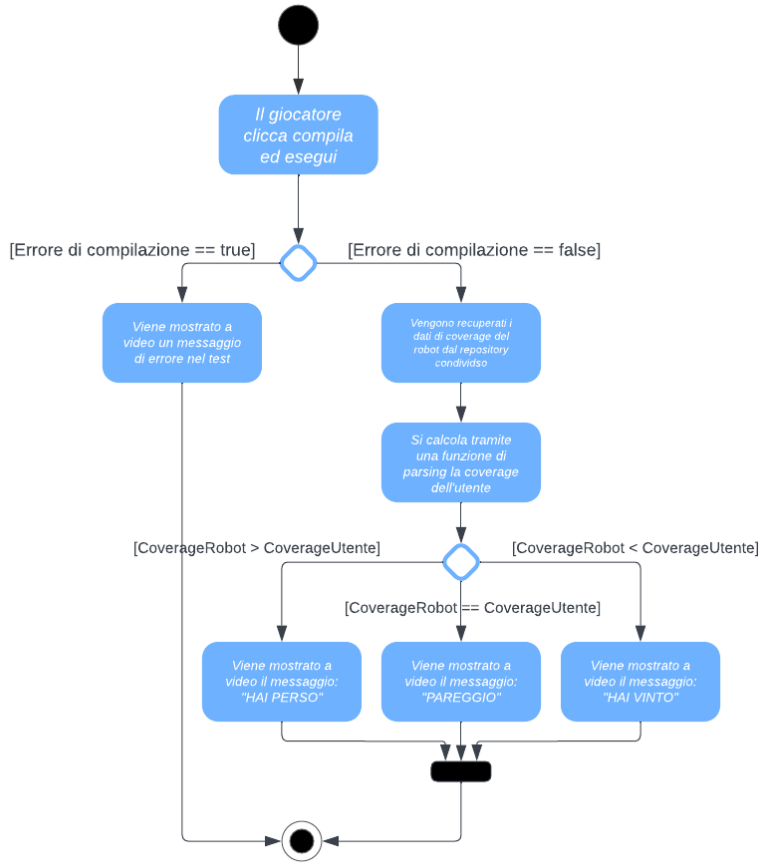
3.3.1 Compila ed Esegui



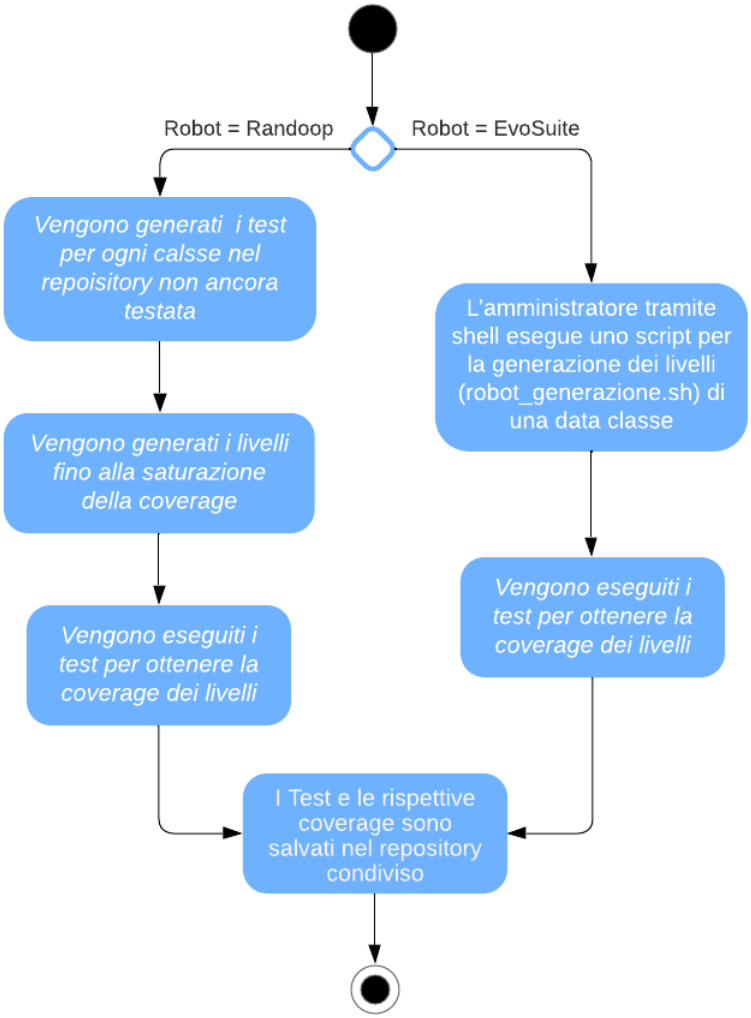
3.3.2 Recupero dati dal repository



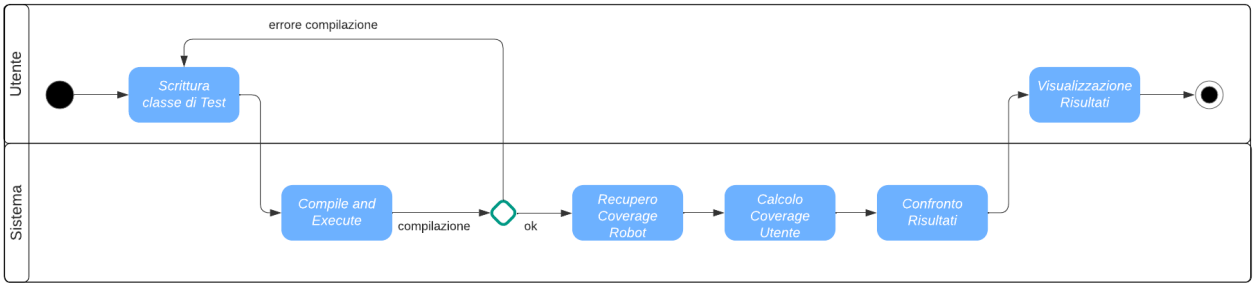
3.3.3 Confronto Risultati



3.3.4 Generazione Robot Test

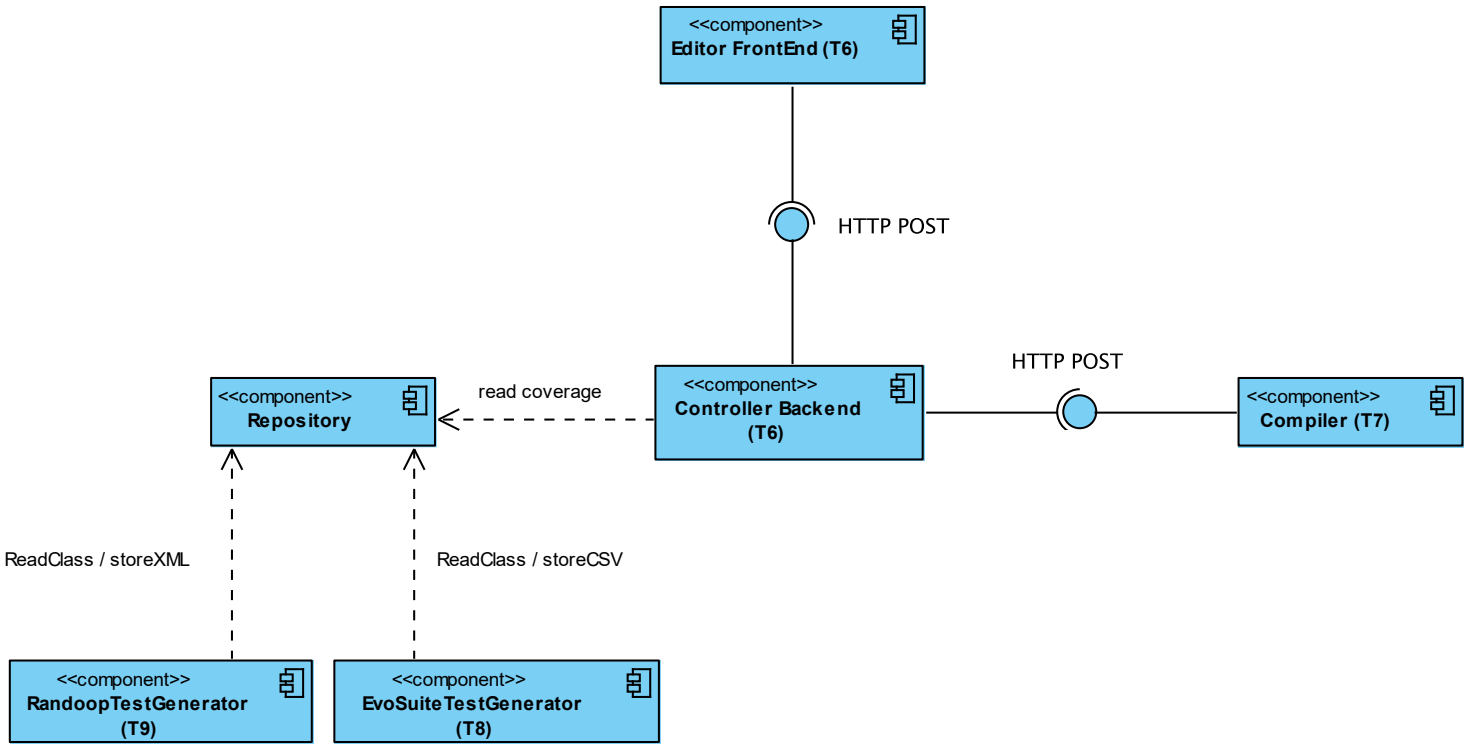


3.3.5 Flusso di attività completo



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.

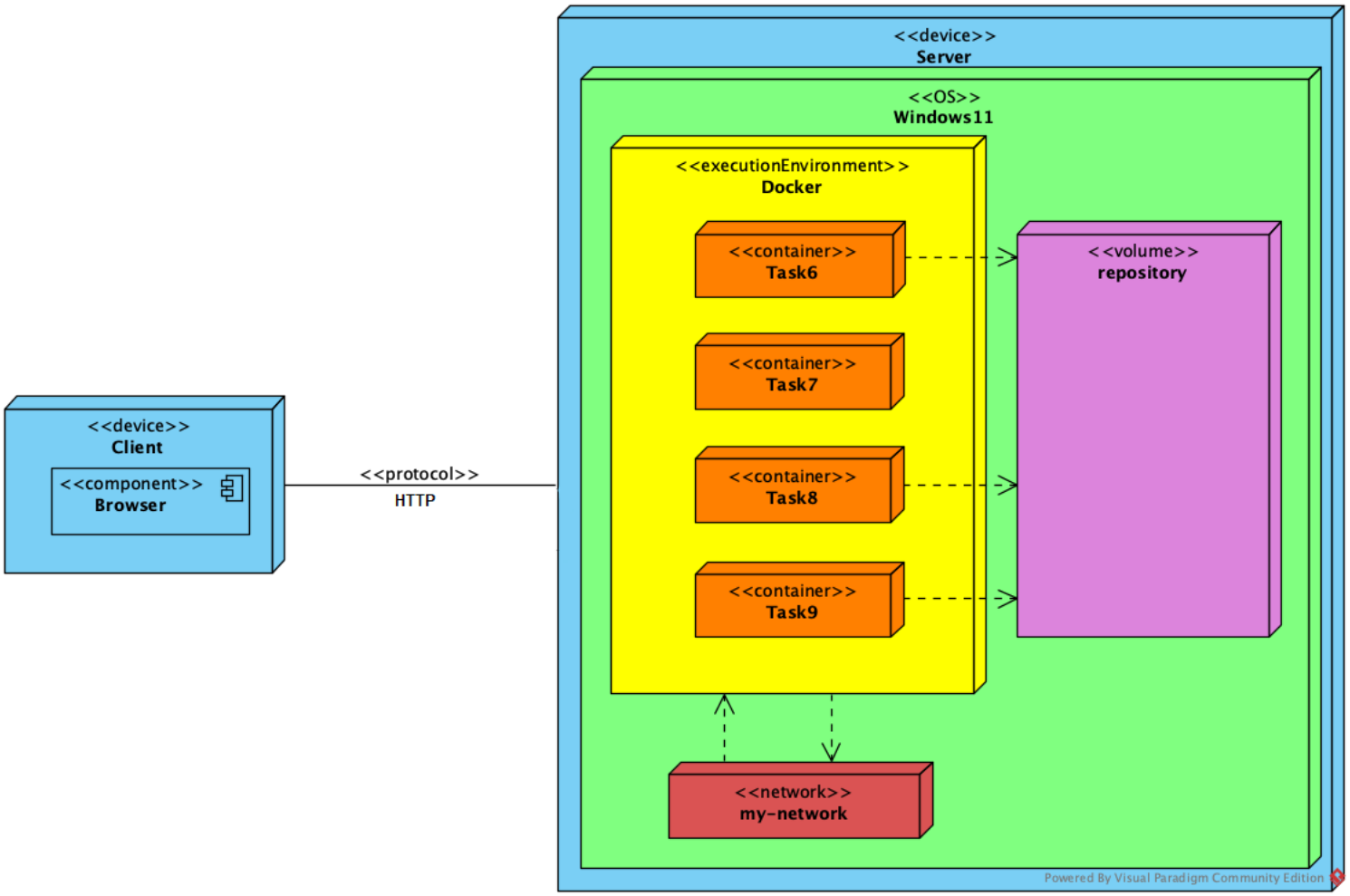


Il componente “Editor Frontend” richiede tramite richiesta POST http i servizi del Backend, questo a sua volta si interfaccia con il componente “Compiler” del Task 7 ed utilizza il repository condiviso per recuperare i valori di coverage. Sul repository agiscono anche i componenti di generazione dei 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.



4 Testing

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

- Test1:
  Descrizione: Compilazione ed Esecuzione 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 "Compile & Execute"
  Output atteso: viene mostrato il vincitore e i dati di coverage di utente e robot

- Test2:
  Descrizione: Compilazione ed Esecuzione di una classe di test con errori dell'utente
  Precondizioni: errore sintattico nella classe di test dell'utente
  Input: clicca sul bottone "Compile & Execute"
  Output atteso: viene mostrato un messaggio di errore: "Errore nel test dell'utente"

- Test3:
  Descrizione: Compilazione ed Esecuzione di una classe di test con errori del robot
  Precondizioni: ci sono stati problemi nel recupero della coverage del robot (RobotCoverage = -1)
  Input: clicca sul bottone "Compile & Execute"
  Output atteso: viene mostrato un messaggio di errore: "Errore robot"

- Test4:
  Descrizione: Generazione dei test Randoop
  Precondizioni: E' 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

- Test5:
  Descrizione: Generazione dei test EvoSuite
  Precondizioni: E' 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

- Test6:
  Descrizione: Generazione dei test EvoSuite con errore nell'input (classundertest non esiste)
  Precondizioni: E' 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

- Test7:
  Descrizione: Generazione dei test EvoSuite con errore nella classe sotto test (vale anche per Randoop)
  Precondizioni: E' 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 "Compile & Execute"	viene mostrato il vincitore e i dati di coverage di utente e robot	PASS
Test2	clicca su "Compile & Execute"	viene mostrato un messaggio di errore: "Errore nel test dell'utente"	PASS
Test3	clicca su "Compile & Execute"	viene mostrato un messaggio di errore: "Errore robot"	PASS
Test4	Avvio dell'applicazione	Vengono generati i livelli contenenti test e coverage	PASS
Test5	./robot_generazione.sh ClassUnderTest ClassUnderTestSourceCode /repository/ClassUnderTest/ClassUnderTestSourceCode 3	Vengono generati i livelli contenenti test e coverage	PASS
Test6	./robot_generazione.sh classundertest ClassUnderTestSourceCode /repository/ClassUnderTest/ClassUnderTestSourceCode 3	Errore nella generazione	PASS
Test7	./robot_generazione.sh ClassUnderTest ClassUnderTestSourceCode /repository/ClassUnderTest/ClassUnderTestSourceCode 3	Errore nella generazione	PASS

5 Installazione ed Esecuzione

5.1 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.

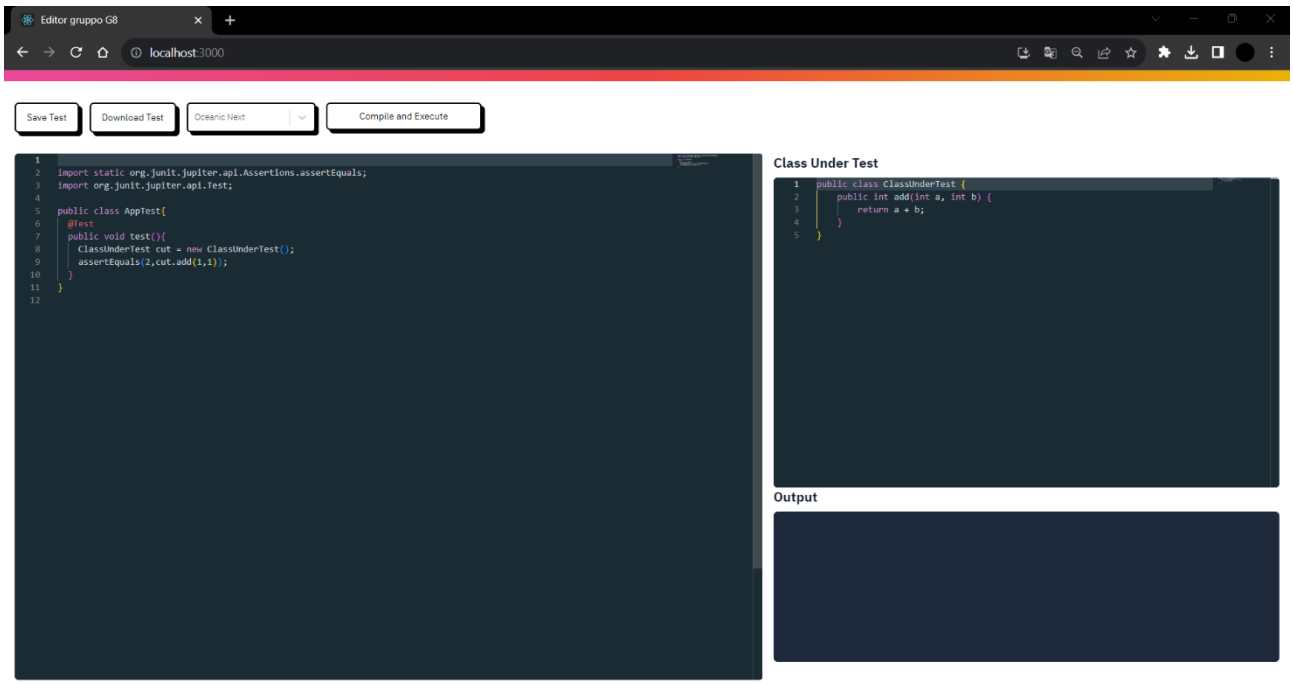
Nel caso in cui si volesse aggiungere una nuova classe da testare bisognerà svolgere i seguenti step:

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: Prototipo2.0/Prototipo2.0
6. Digitare il seguente comando per la generazione dei test EvoSuite:
   ./robot_generazione.sh "NomeClasse" "NomePackage" "PercorsoPackage" "NumeroLivelli"

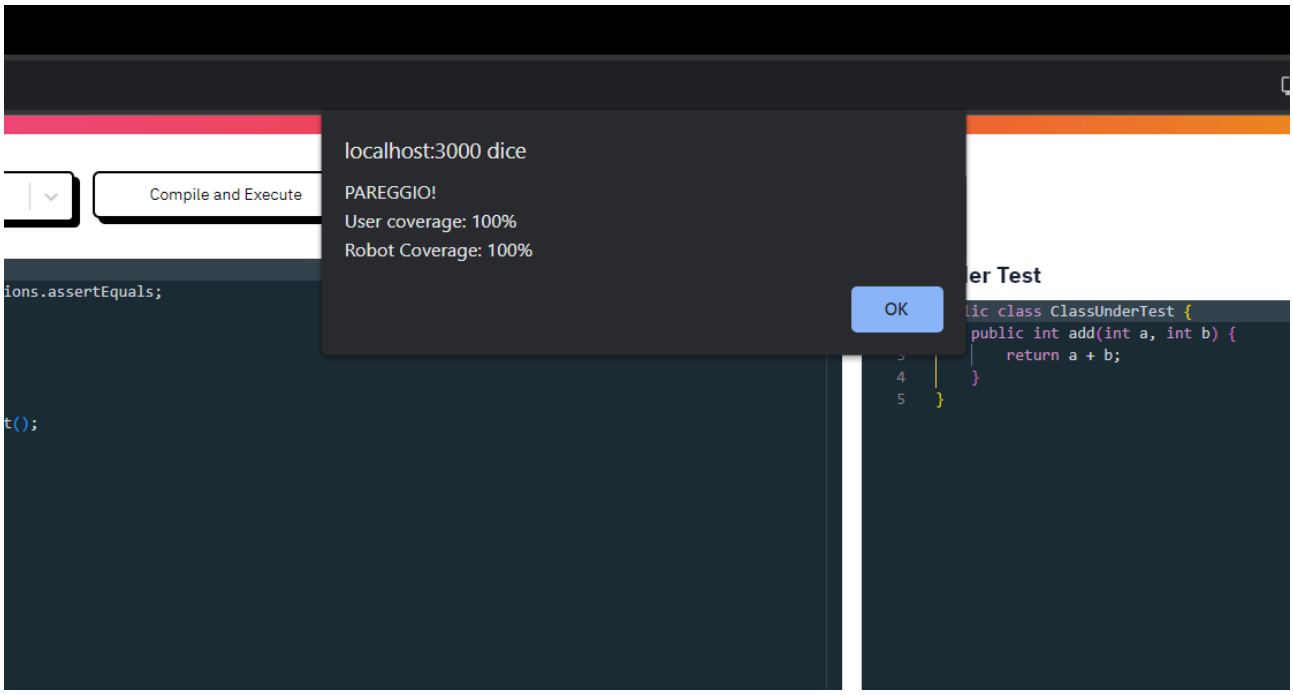
A questo punto è possibile recarsi tramite browser all'indirizzo: ***localhost:3000*** per acceder all'editor di testo e utilizzare l'applicazione.

5.2 Prova di esecuzione

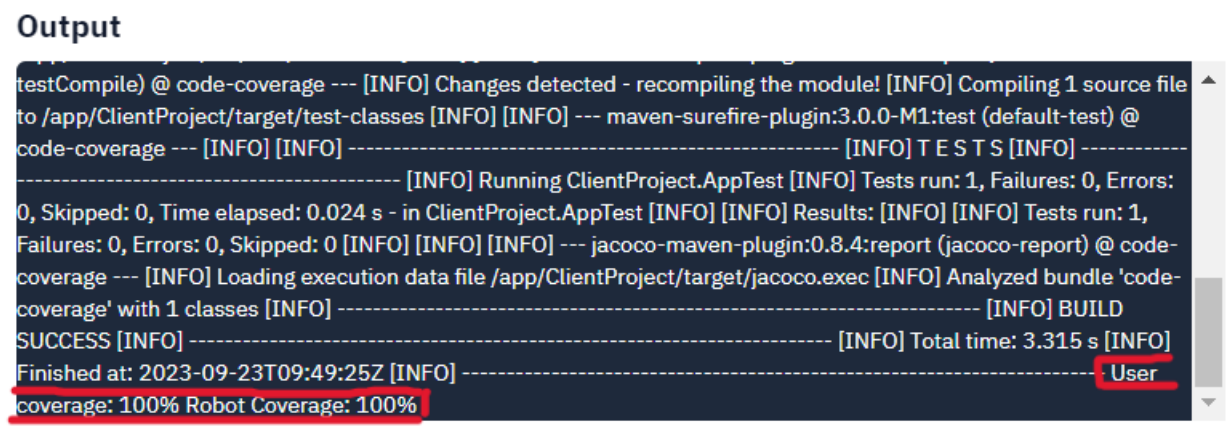
Una volta realizzata la procedura di installazione è possibile recarsi all'indirizzo localhost:3000, sarà mostrata la seguente schermata contenente l'editor per la classe di test:



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 "Compile & Execute", 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:



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