UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Ingegneria Informatica

Progetto in ***Software Architecture Design***

Documentazione Testing Game

Anno Accademico 2022-2023

Candidati
Domenico Chianese - M63001521
Giacomo Lisita - M63001495
Marco Lamboglia - M63001565

Indice

Analisi del Progetto

L'obiettivo del progetto è quello di integrare moduli software pre-sviluppati da altri gruppi di lavoro e che costituiscono gli ingredienti di base per un gioco che si prepone di motivare gli studenti tramite la **gamification** sullo studio delle tecniche di test su classi Java. I moduli software sviluppati sono i seguenti:

- Mantenimento delle classi da testare (T1-G20)
- Registrazione e Login (T23-G24)
- Game Repository (T4-G4)
- Game Engine (T5-G5)
- Editor di Test Case (T6-G16)
- Compilatore ed esecutore (T7-G23)
- Evosuite (T8-G21)
- Randoop (T9-G13)

I moduli sono stati sviluppati rispettando requisiti specifici che implementano le idee del gioco. Tutti le componenti sono state sviluppate secondo il pattern **MVC**, in particolare tramite il framework **Spring MVC**. Maggiori dettagli sui requisiti e sulle modalità di sviluppo sono disponibili nelle rispettive documentazioni[1], di seguito si effettua una breve descrizione dello stato dell'arte dei moduli software.

[1] Disponibili nel seguente repository *github.com/orgs/Testing-Game-SAD-2023/repositories*

Mantenimento delle classi da testare

Questo modulo si occupa della gestione delle classi da testare. Queste devono poter essere consultate e scaricate da tutti i **giocatori**. Inoltre è necessario che gli utenti **amministratori** possano caricare altre classi che verranno aggiunte al database. Il modulo espone delle ***REST API*** che permettono il soddisfacimento dei precedenti requisiti.

Registrazione e Login

Questi componenti si occupano della gestione delle credenziali di accesso degli utenti. In fase di registrazione si raccolgono anche dati generici sugli studenti che sono conservati a scopi statistici. Permette anche il logout e il **recupero** della password in caso di smarrimento. Gli amministratori potranno visualizzare un **riepilogo** di tutti gli utenti registrati.

Anche questo modulo espone un'interfaccia di ***REST API*** che permette l'integrazione con altri moduli.

Game Repository

In questo modulo è implementato il **Game Repository**. Questo consiste in un **database relazionale** che si occupa di conservare i dati delle partite e dei round, i dati di autenticazione (con transazione atomiche e interfaccia **CRUD**) e implementa in una **REST API** i metodi di ricerca di partita e round.

Game Engine

Questo è il modulo centrale. In esso è implementato il motore di gioco che si occupa di avviare il **primo scenario di gioco**. Qui il giocatore potrà scegliere la classe da testare e il robot con cui confrontarsi. Il sistema creerà la partita e verrà comunicata al Repository che la salverà.

Anche in questo caso il modulo espone una **REST API** che permette la comunicazione con il **Game Controller** e il **Player Controller**. L'engine

permette inoltre la gestione di un **sistema ad inviti** per permettere agli utenti di sfidarsi tra loro.

Editor di Test Case

Questo componente si occupa di fornire al giocatore una finestra testuale su cui poter programmare una test suite. Inoltre mette a disposizione la possibilità di caricare dei file pre-costruiti.

Compilatore ed Esecutore

Il modulo implementa le funzionalità di compilazione ed esecuzione dei casi di test scritti da giocatore e robot. Il modulo fornisce un output pulito della compilazione e dell'esecuzione in cui verranno mostrate solo le informazioni più **rilevanti**. Le **REST API** implementate permettono di, previa comunicazione della classe da compilare ed eseguire, eseguire un test.

Evosuite & Randoop

Entrambi i moduli mettono a disposizione dei generatori di test suite automatici che permettono di creare un benchmark su cui comparare le prestazioni dei giocatori. I due generatori automatici sono Evosuite e Randoop.

1.1 Analisi dell'architettura

1.1.1 Specifica dei requisiti

Questa sezione si concentra sull'analisi delle principali caratteristiche del nostro software. In particolare, sono definiti i requisiti funzionali e non funzionali. I requisiti funzionali si riferiscono al comportamento del sistema mentre i requisiti non funzionali riguardano i vincoli imposti dal sistema stesso.

Per quanto riguarda i requisiti **funzionali**, essi sono implementati all'interno dei moduli e definiscono il comportamento dell'applicativo. Lo scopo del progetto attuale non è quello di aggiungere altre funzionalità ma far coesistere i moduli pre-sviluppati e rilevare possibili errori e/o migliorie da apportare ad essi.

I requisiti **non funzionali** sono invece relativi alla specifica soluzione proposta e sono i seguenti:

ID	***Nome***
RNF1	Portabilità
RNF2	Robustezza
RNF3	Compatibilità
RNF4	Scalabilità
RNF5	Facilità di Deploy

Implementando il software all'interno di container docker risolviamo facilmente i requisiti RNF1, RNF3,RNF4 e RNF5. Difatti il **docker engine**, base su cui sono eseguiti i container docker, permette di astrarre l'applicativo dal sistema su cui esso e eseguito e inoltre permette anche l'esistenza di tutte le dipendenze necessarie all'interno dei container stessi. Per il requisito RNF2 è stato necessario implementare una corretta gestione degli error e un logging facilmente comprensibile.

1.2 Stima dei costi

La stima dei costi nell'ingegneria del software è un processo che coinvolge l'**analisi e la previsione dei costi** associati allo sviluppo di un'applicazione software. Questo processo inizia con la raccolta dei requisiti del progetto, dove si stabiliscono le funzionalità desiderate, gli obiettivi, le restrizioni e le scadenze. Successivamente, il lavoro viene suddiviso in attività più piccole e gestibili, e si stima il tempo e le risorse umane necessarie per completarle. Queste stime tengono conto delle competenze e delle esperienze del team.

Inoltre, si identificano le risorse hardware e software necessarie per lo sviluppo, il testing e la distribuzione dell'applicazione. Questi costi possono variare in base alle tecnologie e agli strumenti scelti per il progetto.

Una volta calcolati i costi associati alle risorse umane, hardware e software, si ottiene una stima complessiva dei costi del progetto. Questa stima dei costi è fondamentale per la pianificazione e il controllo del budget del progetto e aiuta a garantire che il progetto sia completato in modo efficiente e entro i limiti finanziari stabiliti. Gli **Use Case Points** (UCP) sono una tecnica di stima utilizzata nella gestione dei progetti software. Questa tecnica si concentra sulla comprensione dei requisiti funzionali di un sistema software attraverso l'analisi dei casi d'uso, degli attori coinvolti e dei fattori di complessità associati.

Nel nostro caso i casi d'uso non si riferiscono a funzionalità da implementare *ex novo* ma a come far collaborare moduli che implementano tali funzionalità e permettere quindi una corretta integrazione tra gli stessi.

1.2.1 Unadjusted Use Case Weight (UUCW):

I valori di riferimento degli Unadjusted Use Case Weight (UUCW) sono:

- Simple(1-3 transiction) Weight 5
- Average(4-7 transiction) Weight 10
- Complex(8 or more transiction) Weight 15

Valutazione dell'UUCW:

Complessità	Peso	N.UC	N.UC*Peso
Simple	5	2	10
Average	10	6	60
Complex	15	2	30
-	-	-	100

Tabella 1.1: UUCW

Use Case	Complessità
Login di un giocatore	Simple
Registrazione di un giocatore	Simple
Modalità turni	Average
Scelta classe	Average
Scelta Robot	Average
Invita Utenti	Average
Avvia nuova partita	Average
Editor	Average
Avvia compilazione	Complex
Avvia Robot	Complex

1.2.2 Technical Complexity Factor (TCF)

Il **Technical Complexity Factor** (TCF), in italiano "Fattore di Complessità Tecnica," è un elemento chiave all'interno del metodo di calcolo del Function Point Analysis (FPA). Il Function Point Analysis è una tecnica utilizzata per misurare la complessità e la dimensione di un sistema software in base alle sue funzionalità e alle sue caratteristiche. Il TCF riflette la complessità tecnica del software, prendendo in considerazione vari fattori che possono influenzare il costo e l'effort necessari per sviluppare e mantenere il software.

I valori di riferimento:

- T1 - Sistema distribuito: 2 con valore 2;
- T2 - Tempo di risposta/obiettivi prestazionali, 1 con valore 3;
- T3 - Efficienza dell'utente finale, 1 con valore 4;
- T4 - Complessità di elaborazione interna, 2 con valore 2;
- T5 - Riusabilità del codice, 2 con valore 5;
- T6 - Facile da installare, 1 con valore 4;

- T7 - Facile da usare, 1 con valore 5;
- T8 - Portabilità su altre piattaforme, 2 con valore 4:
- T9 - Manutenzione del sistema, 1 con valore 4;
- T10 - Elaborazione simultanea/parallela, 1 con valore 2;
- T11 - Funzioni di sicurezza, 1 con valore 2;
- T12 - Accesso per terze parti, 1 con valore 5;
- T13 - Formazione dell'utente finale, 1 con valore 1.

Il totale di tutti i valori calcolati e il fattore tecnico (TF), calcolato come segue:

$$TF = 2\times2+3\times1+4\times1+22+52+41+51+42+41+21+21+51+11 = 56$$

Il TF viene quindi utilizzato per calcolare il TCF con la seguente formula:

$$TCF = 0.6 + (\frac{TF}{100}) = 0.6 + (\frac{56}{100}) = 0.6 + 0.56 = 1,16$$

1.2.3 Environmental Complexity Factor (ECF)

Il **Fattore di Complessità Ambientale** (ECF, Environmental Complexity Factor) è un concetto spesso associato ai metodi di stima del software, in particolare ai modelli della famiglia **COCOMO** (COnstructive COst MOdel), come COCOMO II. L'ECF viene utilizzato per considerare vari fattori esterni che possono influenzare la complessità di un progetto software e, di conseguenza, le sue stime di costo e sforzo. Questi fattori sono legati all'ambiente e alle circostanze del progetto, piuttosto che agli aspetti tecnici. L'ECF viene generalmente espresso come un aggiustamento percentuale sull'effort complessivo del progetto.

Valori di riferimento:

- E1 - Familiarità con il processo di sviluppo utilizzato, 1 con valore 2;
- E2 - Esperienza applicativa, 1.5 con valore 3;
- E3 - Esperienza di team orientata agli oggetti, 1 con valore 1;
- E4 - Capacità di lead analyst, 1 con valore 4;
- E5 - Motivazione del team, 1 con valore 5;
- E6 - Stabilità dei requisiti, 1.5 con valore 4;

Il totale di tutti i valori calcolati e il fattore ambiente (EF), calcolato come segue:

$$EF = 2 \times 1 + 3 \times 1.5 + 1 \times 1 + 4 \times 1 + 4 \times 1 + 4 \times 1.5 + 5 \times (-0.5) = 26$$

L'EF viene quindi utilizzato per calcolare l'ECF con la seguente formula:

$$ECG = 1.4 + (-0.03 \times 26) = 0.62$$

1.2.4 Total Use Case Points (UCP)

Dopo aver determinato le dimensioni del progetto Unadjusted, calcolato il fattore tecnico TCF ed infine il fattore ambientale ECF, si può procedere al calcolo degli **UCP**:

L'UCP viene calcolato in base alla seguente formula:

$$UCP = (UUCW + UAW) \times TCF \times ECF = (100 + 0) \times 1.16 \times 0.62 = 71.92$$

Si può determinare il numero di ore di lavoro totali, considerando 8 ore in media:

$$Th = UPC \times 8 = 71.92 \times 8 = 575.36 \text{ ore}$$

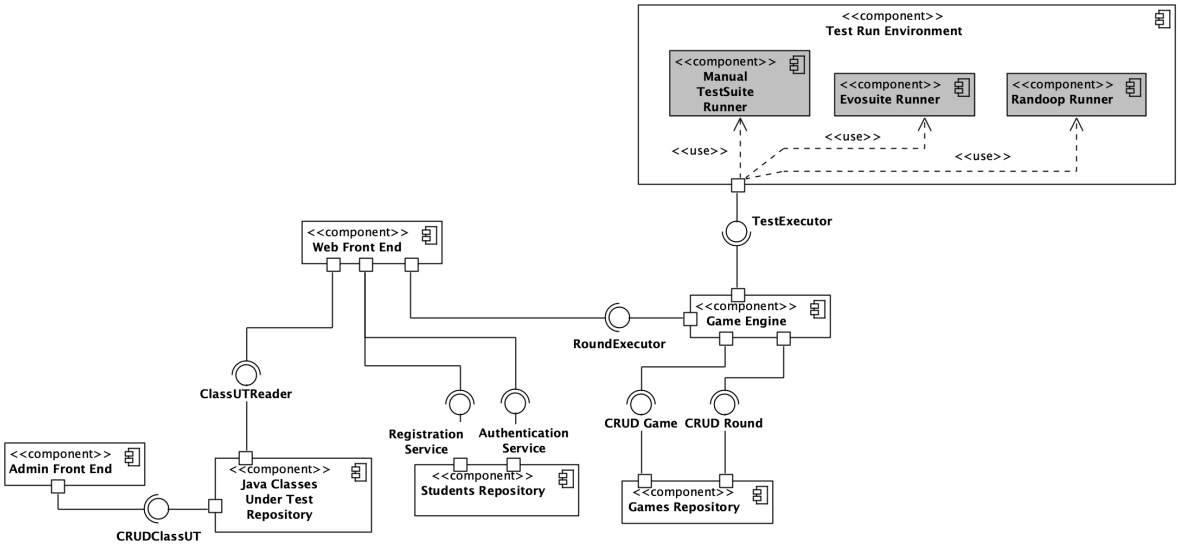

Figura 1.1: Architettura Completa

Th è considerato il numero di ore di lavoro totali per terminare il progetto, mentre ***TWh*** sono le ore settimanali di un team (composto da tre persone) che lavorano sul progetto. Facendo un rapporto, si ottengono le settimane necessarie per completare il progetto:

$$\frac{Th}{TWh} = \frac{575.36}{180} = 3.19 \text{ settimane}$$

1.2.5 Diagramma delle componenti

Lo scopo del progetto è, come detto, l'integrazione dei precedenti componenti. Può quindi essere di beneficio iniziare con una idea generale su come questi moduli possono essere effettivamente collegati tra loro.

Per facilitare l'**integrazione** tra componenti e mantenere tutti i vantaggi della modularità si è andato avanti trattando ogni componente come un servizio atomico. I componenti comunicano tra loro tramite le API messe a disposizione dagli stessi. Ogni componente, già testato e documentato singolarmente, viene considerato un **servizio** che mette a disposizione la

sua API al **Game Engine**, e cioè il quinto task, che è considerato come il centro vitale dell'applicazione.

1.3 Processo di sviluppo

Il ***Processo Software*** si definisce come l'insieme delle attività necessarie allo sviluppo di un sistema software. Le fasi invece **indispensabili** durante il processo software sono quelle di specifica del software, progettazione, implementazione, validazione e mantenimento del prodotto commissionato. Considerata la complessità relativa ad una tale opera è necessario avere un metodo ed un'organizzazione del lavoro ben studiata in modo da evitare incomprensioni tra i membri del team durante il processo dello sviluppo per mantenere alta la qualità del software che si vuole produrre.

Alla luce di ciò nasce la necessità di dover definire un metodo di lavoro che accompagnerà tutto il processo di sviluppo per far sì da prevenire e risolvere le eventuali difficoltà che si incontreranno durante il progetto. L'approccio scelto è di tipo incrementale in modo da rilasciare continuamente codice funzionante per un continuo riscontro di ciò che si sta realizzando ed essere più robusti ai possibili errori commessi.

1.3.1 SCRUM

Il framework SCRUM è un insieme di valori, principi e pratiche che i team Scrum seguono per consegnare un prodotto o un servizio. Dettaglia i membri di un team Scrum e le loro responsabilità, gli "artefatti" che definiscono il prodotto e il lavoro per creare il prodotto, e le cerimonie Scrum che guidano il team Scrum attraverso il lavoro. Ecco una panoramica più dettagliata:

- **Ruoli**: Ci sono tre ruoli chiave in un team Scrum: il Product Owner, lo Scrum Master e il Team di Sviluppo
- **Artefatti**: Gli artefatti in Scrum includono il Product Backlog, lo Sprint Backlog e l'Incremento

- **Cerimonie**: Le cerimonie Scrum includono la pianificazione dello Sprint, la Daily Stand-up, la Review dello Sprint e la Retrospettiva dello Sprint

Le fasi dello sviluppo sono tre:

- La fase iniziale, ovvero una fase di pianificazione in cui si stabiliscono gli obiettivi generali del progetto. In questa fase si progetta anche l'architettura del software.

- Fase di cicli di Sprint, in cui ogni ciclo sviluppa un incremento del sistema. È un periodo di solito 4 settimane, variabile, in cui tutto il team di lavoro si concentra per raggiungere gli obiettivi di quella iterazione. Ogni iterazione avrà i suoi obiettivi.

- La fase di chiusura del progetto impacchetta il progetto. Completa la documentazione richiesta con schermate di help del sistema e manuali utente.

Di seguito sono presenti le tabelle riassuntive delle fasi dello sviluppo.

Prima Iterazione

Obiettivi	• Studio delle tecnologie da utilizzare • Analisi della documentazione dei moduli
Attività	• Configurazione dell'ambiente di sviluppo • Revisione degli errori nei moduli
Risultati	Ambiente di sviluppo configurato
Sviluppi Futuri	• Iniziare l'integrazione dei task • Pianificare le successive iterazioni

Seconda Iterazione

Obiettivi	Integrazione dei primi cinque task
Attività	• Sviluppo del codice necessario all'integrazione dei moduli • Test di integrazione dei moduli
Risultati	Primi cinque moduli integrati
Sviluppi Futuri	• Continuare con l'integrazione dei task • Pianificare le successive iterazioni

Terza Iterazione

Obiettivi	Integrazione dei restanti quattro task
Attività	• Sviluppo del codice necessario all'integrazione dei moduli • Test di integrazione dei moduli
Risultati	• Moduli integrati • Integrazione documentata completamente
Sviluppi Futuri	Migliorare le prestazioni dell'applicativo

Integrazione

In questo capitolo saranno descritte le procedure di integrazione di ogni task all'interno del **Game Engine** e, se necessario, in altri task.

Per conservare la modularità e la facilità di modifica e manutenzione ogni task è stato assegnato ad un container **Docker** che comunica con gli altri tramite una **API**.

2.1 Task 1: Class Under Test Repository

Il *task 1* si occupa di gestire il repository delle classi che saranno oggetto del gioco. Offre anche la possibilità di registrare utenti **amministratori** che possono aggiungere ulteriori classi. Il modulo è costruito come in [figura 2.1] ed espone le interfacce presenti in figura. In particolare all'interno del modulo è progettata una **REST API** che espone le funzionalità delle interfacce.

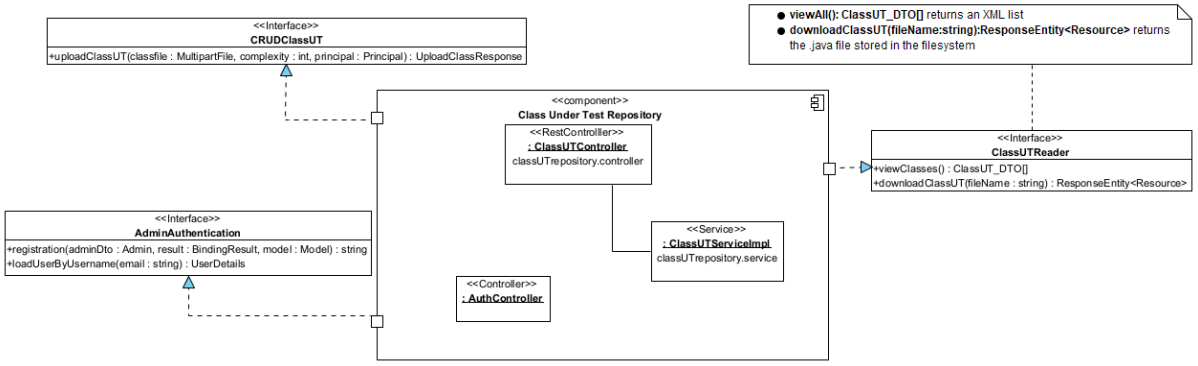

Figura 2.1: Diagramma delle componenti del Task 1

L'integrazione è stata affrontata in due parti diverse: azioni relative agli utenti **amministratori** e agli utenti **standard**.

2.1.1 Utenti Amministratori

Per quanto riguarda gli amministratori, questi devono poter registrarsi, accedere e caricare nuovi classi. Le **storie utente** relative sono:

ID	***Nome***
1	Accesso alla pagina di Registrazione
2	Accesso alla pagina di Login
3	Caricamento di una nuova Classe

Scenari

Gli **scenari** associati a queste storie sono:

ID Storia	1
Attore Primario	Amministratore
Attore Secondario	—
Descrizione	Permette la Registrazione
Pre-Condizioni	L'amministratore non deve essere registrato
Sequenza di eventi	• L'amministratore si collega alla pagina principale dell'applicativo • Clicca sul pulsante *Sei un Admin?* • Naviga alla pagina di registrazione • Inserisce i suoi dati
Post-Condizioni	L'amministratore è ora registrato
Sequenza di eventi alternativi	Se già registrato fallisce l'operazione

ID Storia	2
Attore Primario	Amministratore
Attore Secondario	—
Descrizione	Permette il Login
Pre-Condizioni	L'amministratore deve essere registrato
Sequenza di eventi	• L'amministratore si collega alla pagina principale dell'applicativo • Clicca sul pulsante *Sei un Admin?* • Naviga alla pagina di login • Inserisce i suoi dati
Post-Condizioni	L'amministratore è ora loggato
Sequenza di eventi alternativi	Se i dati sono sbagliati fallisce l'operazione

ID Storia	3
Attore Primario	Amministratore
Attore Secondario	—
Descrizione	Permette il caricamento di una nuova classe
Pre-Condizioni	L'amministratore deve essere logato
Sequenza di eventi	• L'amministratore carica il percorso di una classe da caricare nel repository • Seleziona la complessità della classe • Clicca sul pulsante *Upload*
Post-Condizioni	La classe è presente nel repository
Sequenza di eventi alternativi	—

Diagrammi di Sequenza

Per quanto riguarda i diagrammi di sequenza ci sono stati dei minimi cambiamenti rispetto alla documentazione ufficiale del modulo uno. In particolare quello relativo alla storia numero tre non è cambiato in alcun modo. Gli altri sono invece presenti in seguito.

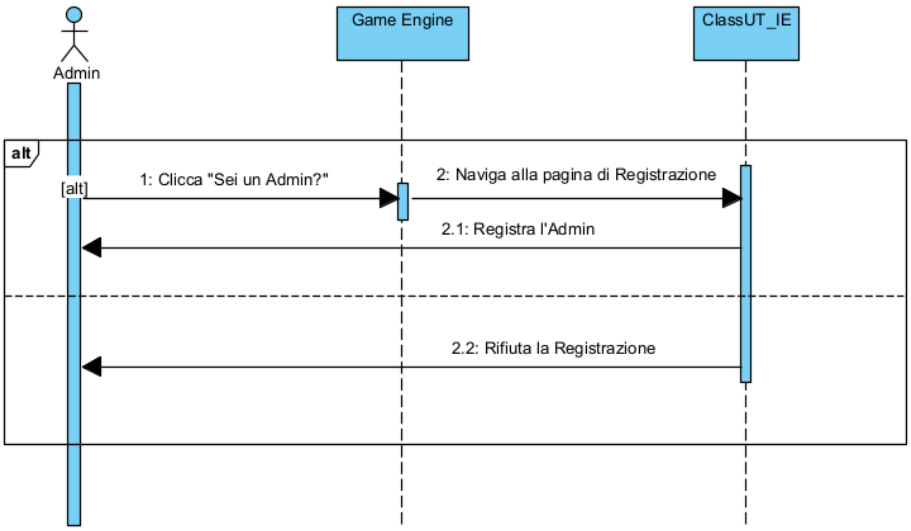

Figura 2.2: Diagramma di sequenza relativo alla storia utente 1

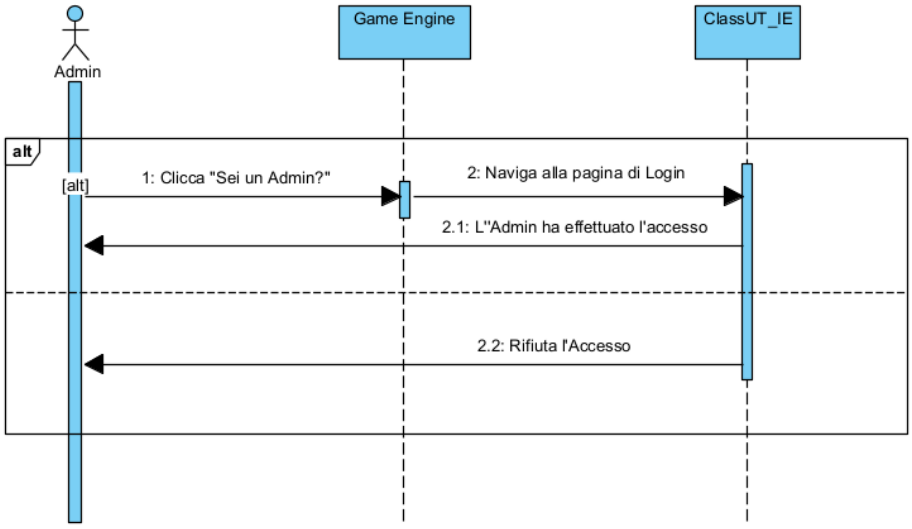

Figura 2.3: Diagramma di sequenza relativo alla storia utente 2

Non è stato necessario operare modifiche all'interno del modulo. Le uniche modifiche effettuate sono al modulo cinque per l'adattamento e la chiamata delle interfacce.

Modifiche al Game Engine

Per permettere l'accesso e la registrazione agli utenti amministratore, come si vede dai diagrammi precedenti, è stato necessario modificare il front-end del modulo cinque. La pagina di Login al gioco ora si presenta con un pulsante in più che reindirizza all'interfaccia di controllo amministratore.

All'interno del file *Login.js* è stato quindi aggiunto il codice necessario per gestire queste casistiche:

```
<Link onClick={handleAdmin} className="admin-link">
  Sei un Admin?
</Link>
```

Con relativo **handler**:

```
const handleAdmin = () => {
  window.location.href = "http://localhost:8090/login"
}
```

È presente una *preview* della nuova interfaccia di login in [figura 2.4]. In [figura 2.7] è presente il *package diagram* del modulo con evidenziato il componente che ha subito modifiche.

2.1.2 Utenti Standard

Gli utenti Standard devono invece poter scegliere la classe che sarà protagonista del gioco. Una volta effettuato il login, i cui requisiti sono specificati nella documentazione del *task 5*, è necessario sfruttare le API messe a disposizione dal modulo per permettere di generare una lista di classi e la scelta

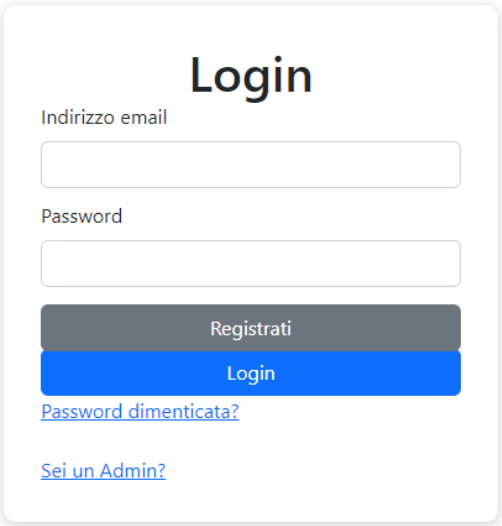

Figura 2.4: Nuovo Front-end della pagina di login

della stessa.

ID	***Nome***
1	Scelta della classe tra quelle disponibli
2	Cerca una classe

Scenari

Gli **scenari** associati sono:

ID Storia	1
Attore Primario	Amministratore/Utente
Attore Secondario	—
Descrizione	Permette la scelta di una classe tra le disponibili
Pre-Condizioni	L'utente deve essere loggato
Sequenza di eventi	L'utente seleziona una delle tre classi disponibili
Post-Condizioni	La classe è stata selezionata
Sequenza di eventi alternativi	—

ID Storia	2
Attore Primario	Amministratore/Utente
Attore Secondario	—
Descrizione	Permette la scelta di una classe tra le disponibili
Pre-Condizioni	L'utente deve essere loggato
Sequenza di eventi	Avvia una ricerca tra le classi disponibili
Post-Condizioni	La classe è stata selezionata
Sequenza di eventi alternativi	—

Diagramma di Sequenza

Di seguito sono presenti i diagrammi di sequenza relativi alla comunicazione tra task:

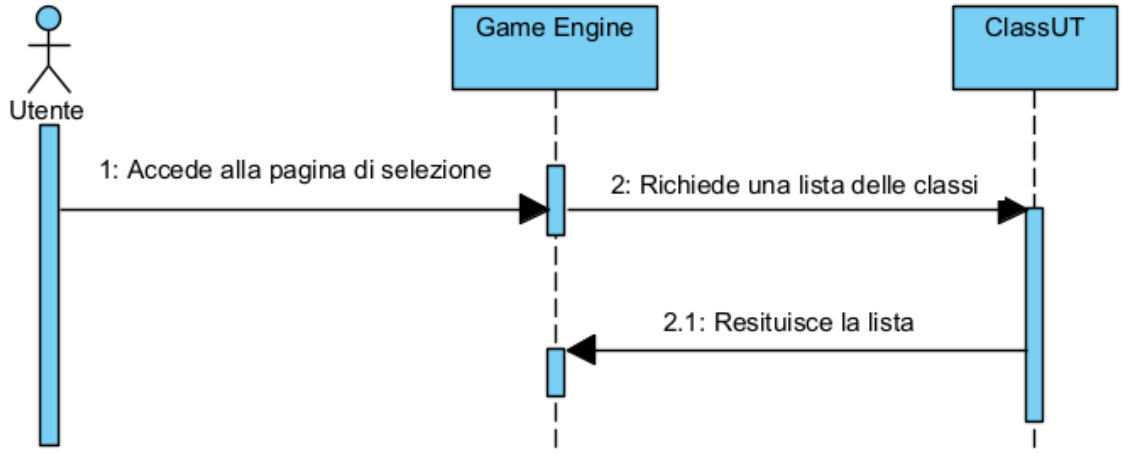

Figura 2.5: Diagramma relativo alla storia 1

Non è stato necessario operare modifiche all'interno del modulo. Le uniche modifiche effettuate sono al modulo cinque per l'adattamento e la chiamata delle interfacce.

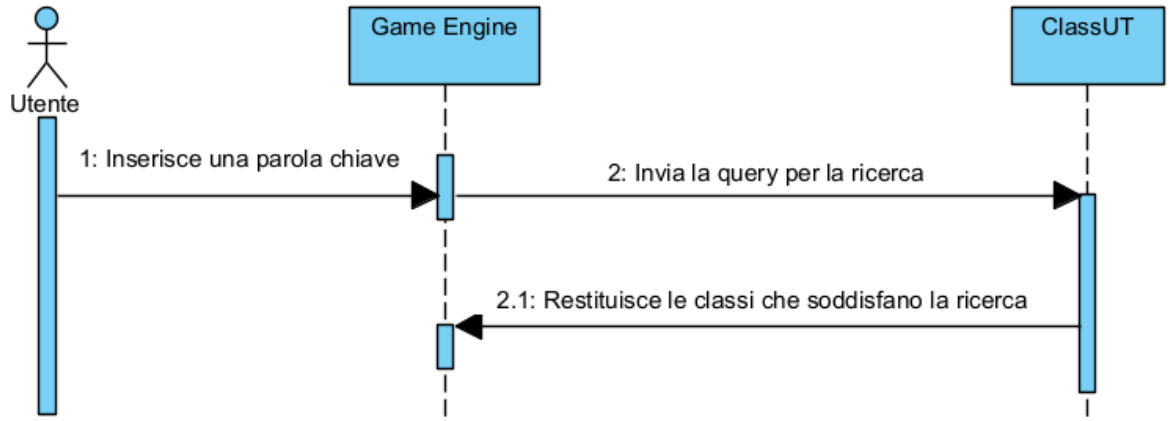

Figura 2.6: Diagramma relativo alla storia 2

Modifiche al Game Engine

Anche in questo caso è stato necessario modificare il *Front-end* del Game Engine per effettuare le richieste alle API del modulo. In particolare sono stati modificati in modo simmetrico i file *Single.js* e *Multi.js*.

All'interno di questi file è stato aggiunto il codice:

```
  useEffect(() => {
  fetch('http://localhost:8090/classut_repo/viewAll')
  .then(response => response.text())
  .then(str => new window.DOMParser().parseFromString(str, "text/xml"))
  .then(data => {
    const items = data.getElementsByTagName("item");
    const newClassList = Array.from(items).map(item => {
      const name = item.getAttribute("name");
      return { name: name, icon: faCube };
    }).filter(classItem => {
      return classItem.name.toLowerCase().includes(searchTerm.toLowerCase()
   );
    });
    setClassList(newClassList);
  });
}, [searchTerm]);
```

Questo permette, alla modifica della variabile `SearchTerm`, di effettuare una chiamata all'API del modulo per ricevere una **lista di classi filtrata** tramite la variabile stessa. Quando la variabile è vuota mostra le prime tre classi presenti nel repository.

È stato inoltre aggiunto il codice che permette di mappare **dinamicamente** i risultati della *get* ai pulsanti presenti a schermo.

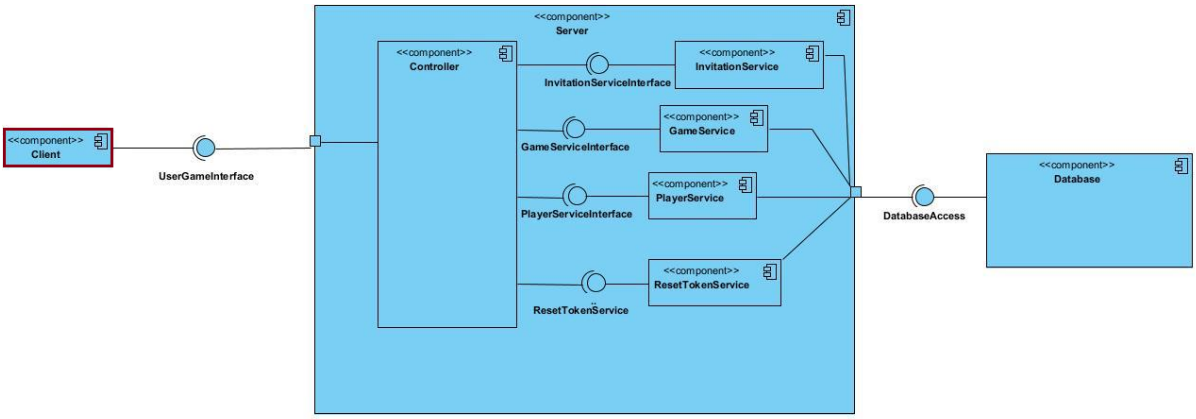

Figura 2.7: Nel *Package Diagram* è evidenziato il componente che ha subito modifiche

Il *client* è formato dai **file javascript** che sono eseguiti sul browser dell'utente.

2.2 Task 2 & 3: Login & Registrazione

Le funzionalità dei moduli due e tre si trovano già perfettamente integrate all'interno del Game Engine. Il *Login System* implementato nel modulo permette la **registrazione**, che avviene successivamente ad aver cliccato un link nell'email di conferma, il **login** e il **reset** della password. Il database che conserva i dati degli utenti è presenta il modello E-R presente in [figura 2.8].

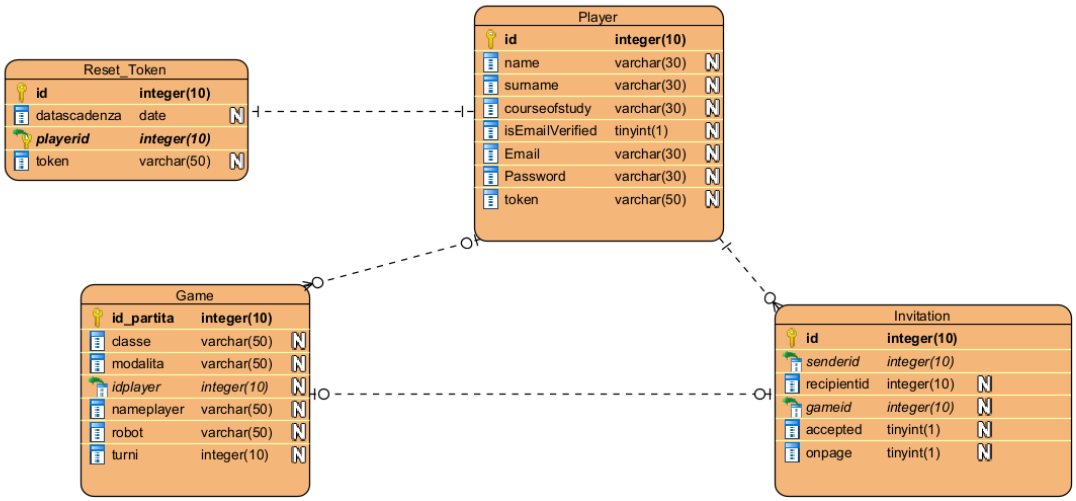

Figura 2.8: Modello E-R per la gestione degli utenti

Si è deciso di utilizzare l'implementazione del modulo cinque e non dei moduli due e tre per i seguenti motivi:

- front-end già sviluppato con l'integrazione in mente: è più coeso alla vista dell'utente finale.
- incompatibilità dei database: il modulo cinque utilizza **MySQL** mentre due e tre usano **PostgreSQL**. Questo vorrebbe dire l'utilizzo di due DBMS differenti che avrebbero rallentato il sistema su cui è eseguito l'ambiente.

- dalla documentazione del modulo cinque è possibile osservare che l'intero modulo è stato già testato e risulta funzionante (con integrazione compresa): questo permette di risparmiare tempo sul testing.

Lo svantaggio è rappresentato da un aumento dell'accoppiamento delle funzionalità e una minore facilità di manutenibilità che però sono compensati dalle precedente considerazioni.

2.3 Task 4: Game Repository

Sfruttando il database con modello presente in [figura 2.8] è stato necessario integrare le funzionalità dell'API tra il modulo quattro e cinque.

Il componente cinque, sviluppato secondo il pattern **MVC**[1], presenta già le componenti Model e View della entità *Game*. Abbiamo quindi personalizzato il Controller per esporre degli endpoint che permettessero il corretto interfacciamento tra il back-end, e quindi il database, e il front-end.

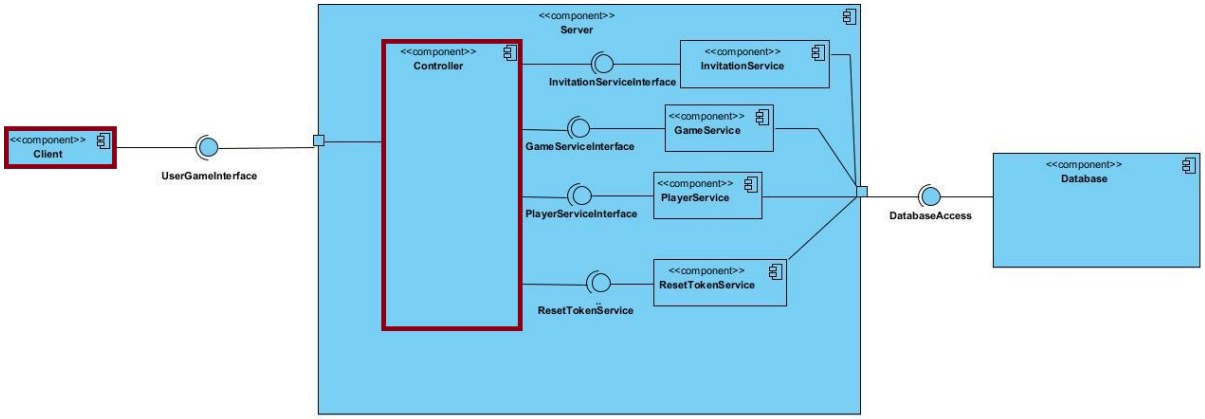

Figura 2.9: Diagramma con modifiche evidenziate

Il nuovo ***GameController*** presenta tre diversi endpoint che permettono l'aggiunta di un gioco e l'ottenimento di una lista di tutti i giochi e di un gioco per id dello stesso.

Di seguito è presente il file *GameController.java*

```
1  @RestController
2  @RequestMapping("/game") // player
3
4  @CrossOrigin
5  public class GameController {
6    @Autowired
7    private GameService gameService;
8
9    @PostMapping("/add")
10   public int add(@RequestBody Game game) {
```

[1] Anche detto Model-View-Controller

```
11      Game savedGame = gameService.saveGame(game);
12      return savedGame.getId_partita();
13    }
14
15    @GetMapping("/getAll")
16    public List<Game> list(){
17      return gameService.getAllGames();
18    }
19
20    @GetMapping("/{id}")
21    public Optional<Game> getGameById(@PathVariable int id) {
22      return gameService.getGameById(id);
23    }
24  }
```

Per l'utilizzo della API ci si può riferire alla seguente tabella:

Endpoint	***Metodo***	***Descrizione***	***Parametri***
/game/add	POST	Aggiunge una partita presente nel body.	—
/game/getAll	GET	Lista tutti le partite.	—
/game/{id}	GET	Lista le caratteristiche di una partita	*id*: id partita

In particolare per il metodo **POST** è necessario un *json* nel body che contenga tutti i dati necessari per l'aggiunta di un *game* come *classe*, *robot*, *turni* e *id*.

2.3.1 Modifiche al Front-End del Task 5

Il front-end deve poter richiamare le API agli end-point appena definiti. Sonon state infatti aggiunte nel file *Single.js* e *Multi.js*, che controllano l'inizio della partita in modalità single e multiplayer, e quindi effettivamente la creazione del singolo game le seguenti linee di codice:

```
1  const gameData = {
2    robot: robot,
```

```
3     classe: classe,
4     turni: turni,
5     modalita: modalita,
6     name: name,
7     id: nid
8   }
9   fetch('http://localhost:8080/game/add', {
10    method: 'POST',
11    headers: { 'Content-Type': 'application/json' },
12    body: JSON.stringify(gameData)
13  })
```

La variabile `gameData` viene popolata dai dati presenti all'interno della pagina web e quindi dalle scelte dell'utente. Viene poi effettuata una chiamata all'end-point *game/add* per aggiungere i dati della partita. L'end-point *game/id* viene poi utilizzato per le funzionalità multiplayer offerte dal modulo stesso.

Queste modifiche permettono ai vari task di poter comunicare in modo efficiente tra di loro utilizzando solamente tramite le API.

2.4 Task 6: Test Editor

Il Test Editor permette all'utente, dopo aver avviato un gioco e selezionato una classe, di scrivere la classe che permette di testare quella prescelta. Deve quindi esporre un front-end che verrà mostrato successivamente all'avvio del gioco. In [figura 2.10] è presente un diagramma delle componenti del sistema che interessano direttamente al modulo sei.

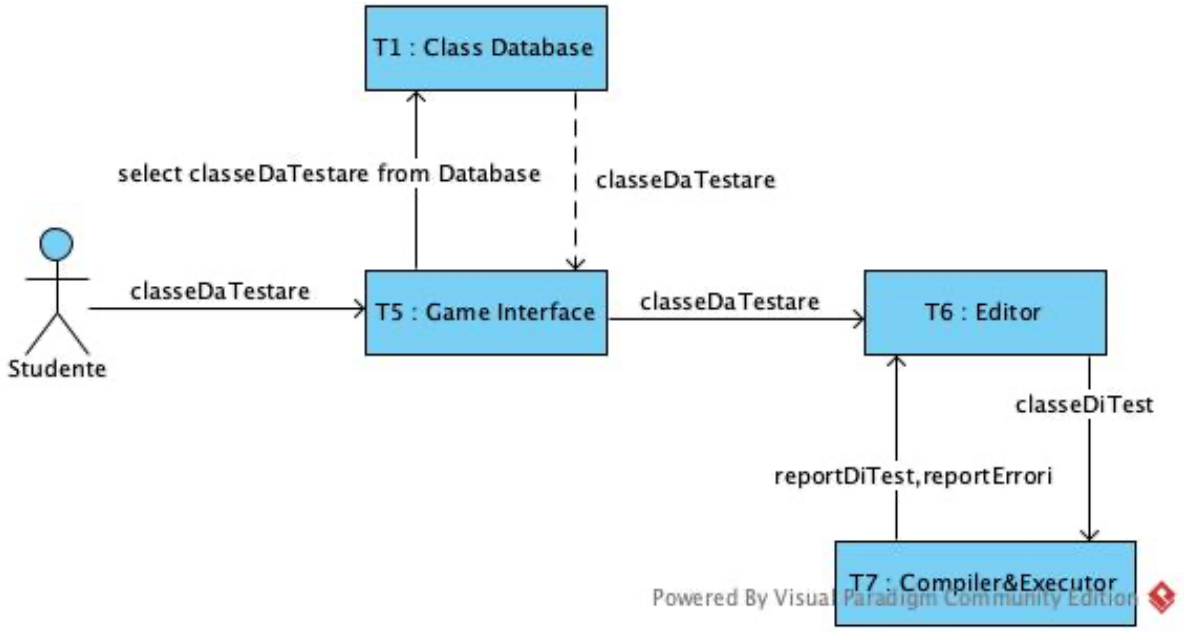

Figura 2.10: *Component Diagram* che mostra i collegamenti tra T1-5-6-7

Il modulo sei inoltre è responsabile della creazione di un File System condiviso tra gli ultimi quattro moduli e che permettono la corretta esecuzione del test alle classi. In [figura 2.11] è presente un esempio del File System.

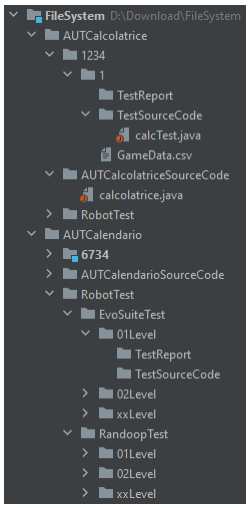

Figura 2.11: Struttura del File System condiviso

Le **storie utente** corrispondenti alle funzionalità che deve avere il modulo dopo l'integrazione sono:

ID	***Nome***
1	Accesso all'editor
2	Salvataggio Classe
3	Compilazione ed esecuzione test
4	Esecuzione del Robot

Scenari

Con **scenari**:

ID Storia	3
Attore Primario	Utente
Attore Secondario	—
Descrizione	Compilare ed eseguire la classe di test
Pre-Condizioni	L'utente deve aver scritto una classe di test
Sequenza di eventi	Clicca il tasto di esecuzione test
Post-Condizioni	È mostrato il risultato di compilazione e test
Sequenza di eventi alternativi	—

ID Storia	1
Attore Primario	Utente
Attore Secondario	—
Descrizione	Accede all'editor e mostra la classe prescelta
Pre-Condizioni	L'utente deve aver avviato una partita
Sequenza di eventi	• L'utente seleziona una classe e avvia una partita • Viene scaricata la classe prescelta
Post-Condizioni	È mostrato l'editor con la classe prescelta
Sequenza di eventi alternativi	—

ID Storia	2
Attore Primario	Utente
Attore Secondario	—
Descrizione	Salvare la classe nel *File System* condiviso
Pre-Condizioni	L'utente deve essere nell'editor
Sequenza di eventi	• L'utente scrive una classe di test • Clicca sull'icona di salvataggio
Post-Condizioni	La classe è presente nelle cartelle giuste
Sequenza di eventi alternativi	—

ID Storia	4
Attore Primario	Utente
Attore Secondario	—
Descrizione	Eseguire il robot sulla classe
Pre-Condizioni	L'utente deve essere nell'editor
Sequenza di eventi	Clicca il tasto di esecuzione Robot
Post-Condizioni	È mostrato il risultato dell'esecuzione del robot
Sequenza di eventi alternativi	—

Diagrammi di sequenza

Di seguito i diagrammi di sequenza per le storie utente.

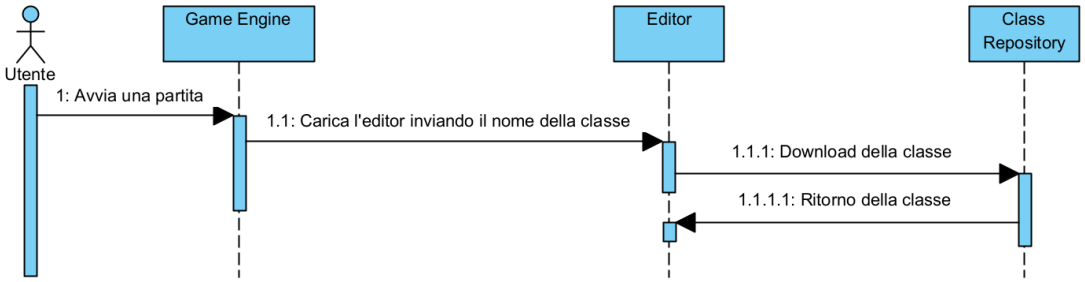

Figura 2.12: Sequenza relativa alla storia 1

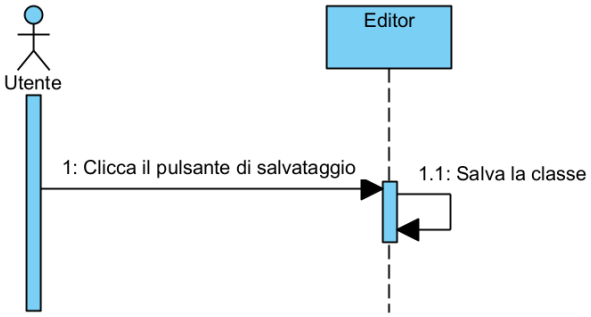

Figura 2.13: Sequenza relativa alla storia 2

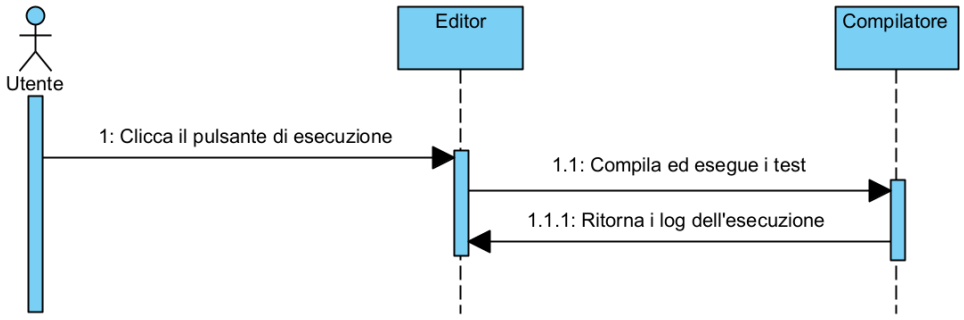

Figura 2.14: Sequenza relativa alla storia 3

2.4.1 Modifiche effettuate

Per implementare queste funzionalità è stato necessario apporre delle modifiche ai file del componente. È stato modificato sia il *controller* che la view (nella forma del file *code-editor.html*).

Controller

Per quanto riguarda le modifiche al controller sono le seguenti:

```
1  @GetMapping("/")
2  public String getHomePage(@RequestParam(name = "myParam") String class_name
       , @RequestParam(name = "id") String id, @RequestParam(name = "robot")
       String robot) {
3
4    classname = class_name;
5
6    System.out.println("Parameter value: " + class_name);
7    return "code-editor.html";
8  }
9
10 @PostMapping("/class")
11 public String postClass(@RequestBody String testClass, @RequestParam(name =
       "id") String id, @RequestParam(name = "gameid") String gameid) {
12   String filename = "AUT" + classname;
13   if (testClass != null) {
14     try {
15       String path_comp = path + filename.replace(".java", "") + "/" + id +
       "/" + gameid + "/TestSourceCode";
16       new File(path_comp).mkdirs();
17       FileWriter myWriter = new FileWriter(path_comp + "/" + filename);
18       myWriter.write(testClass);
19       myWriter.close();
20       System.out.println("Successfully wrote to the file.");
21     } catch (IOException e) {
22       System.out.println("An error occurred.");
23       e.printStackTrace();
24     }
25   }
26   return "Success";
27 }
28
29
```

```
30 @CrossOrigin(origins = "*")
31 @PostMapping("/original")
32 public String postOriginal(@RequestBody String testClass) {
33   String filename = classname;
34   if (testClass != null) {
35     try {
36       String path_comp = path + "AUT" + filename.replace(".java", "") + "/
     AUT"
37       + filename.replace(".java", "SourceCode");
38       new File(path_comp).mkdirs();
39       FileWriter myWriter = new FileWriter(path_comp + "/" + filename);
40       myWriter.write(testClass);
41       myWriter.close();
42       System.out.println("Successfully wrote to the file.");
43     } catch (IOException e) {
44       System.out.println("An error occurred.");
45       e.printStackTrace();
46     }
47   }
48   return "Success";
49 }
```

Le modifiche comprendono:

- Aggiunta alla richiesta GET all'endpoint *"/"* dei parametri `id` e anche `robot`. Queste modifiche sono necessarie all'editor per conoscere l'id della partita da caricare e quale robot far partire alla pressione del tasto corrispondente.
- Aggiunta dell'endpoint *"/class"* che permette il salvataggio della classe di test all'interno del File System condiviso e alle cartelle corrette.
- Discorso simile per l'endpoint *"/original"*.[2]

Quindi i nuovi endpoint e le funzionalità sono riassunte in questa tabella.

[2]Sul perchè sono necessari rimandiamo alla documentazione dell'integrazione del Task 7

Endpoint	***Metodo***	***Descrizione***	***Parametri***
/	GET	Restituisce la pagina principale.	*myParam*: Nome della classe, *id*: ID, *robot*: Nome del robot
/class	POST	Salva la classe di test in un file presente nel body	*testClass*: Classe di test, *id*: ID, *gameid*: ID del gioco
/original	POST	Salva la classe in un file presente nel body	*testClass*: Classe di test

View

Sono stati aggiunti i pulsanti per il **salvataggio**, **compilazione ed esecuzione dei test** e **compilazione ed esecuzione robot**. È stato inoltre necessario modificare il file per richiamare i corretti endpoint e gestire il caricamento della classe.

Con questa chiamata si richiama il servizio in esecuzione nel primo modulo per **ottenere il sorgente** della classe prescelta dall'utente.

```
1  const apiUrl = "http://localhost:8090/classut_repo/downloadClass/" +
       myParamValue;
2
3  fetch(apiUrl)
4  .then(response => response.text())
5  .then(data => {
6    testEditor.setValue(data)
7  })
8  .catch((error) => {
9    console.error("Fetch error:", error);
10 \});
```

Con questa modifica è possibile richiamare il modulo 4 per ottenere l'id dello studente che sta giocando per poi richiamare il servizio di compilazione del modulo 7 per compilare effettivamente la classe di test.

```
1     function compileJava() {
```

```
2   const errorConsole = document.getElementById('error-console');
3   var className = myParamValue
4   var GameId = id
5   var StudentId = 0
6
7   var studentApi = "http://localhost:8080/game/" + id;
8   fetch(studentApi)
9   .then(function (response) {
10    return response.json()
11  })
12  .then(function (json) {
13    StudentId = json.id_player;
14  })
15  .catch(error => {
16    console.error('Si e verificato un errore durante il recupero di ID: ',
      error);
17  });
18
19  var compileApiUrl = 'http://localhost:5000/compexec?ClassName=' + "AUT" +
      className.replace(".java", "") + '&StudentLogin=' + StudentId + '&
      GameId=' + GameId
20
21  var myHeaders = new Headers();
22  myHeaders.append("Content-Type", "application/json");
23
24  var requestOptions = {
25    method: 'GET',
26    headers: myHeaders,
27    redirect: 'follow'
28  };
29  fetch(compileApiUrl, requestOptions)
30  .then(response => response.json())
31  .then(json => {
32    console.log(json)
33    const errorElement = document.createElement('div');
34    errorElement.textContent = json.resMessage;
35    errorConsole.appendChild(errorElement);
36  })
37  .catch(error => console.log('Si e verificato un errore durante il
      recupero del file TXT:', error));
38 }
```

Infine la modifica necessaria al caricamento e il salvataggio della classe sotto test prescelta e la classe di test scritta dall'utente.

```
function saveClass() {
  var GameId = id
  var StudentId = 0
  var studentApi = "http://localhost:8080/game/" + id;

  fetch(studentApi)
  .then(function (response) {
    return response.json()
  })
  .then(function (json) {
    StudentId = json.id_player;
  })
  .catch(error => {
    console.error('Si e verificato un errore durante il recupero di ID: ',
     error);
  });

  var classUrl = "http://localhost:8190/class?id=" + StudentId + "&gameid="
     + GameId
  fetch(classUrl, {
    method: 'POST',
    body: javaEditor.getValue()
  })
  .then(response => response.text())
  .then(data => console.log(data));

  fetch('http://localhost:8190/original', {
    method: 'POST',
    body: testEditor.getValue(),
    mode: 'no-cors'
  })
  .then(response => response.text())
  .then(data => console.log(data));
}
```

2.5 Task 7: Compilatore

Il compilatore permette all'applicativo di eseguire le classi di test codificate dagli utenti. Il task 7 non ha subito modifiche sostanziali in quanto l'idea di progettazione iniziale era quella di avere un modulo compilatore completamente indipendente dal resto utilizzabile direttamente tramite un **API**. L'API esposta dal modulo comprende due **endpoint** [figura 2.15]:

- **CompExec** (**GET**): che si limita ad eseguire una classe **già** presente all'interno del File System condiviso.
- **CompExecUrl** (**POST**): che dovrebbe prendere due classi (classe sotto test e *test class*) e posizionarle nella loro posizione corretta all'interno del File System per poi offrire le stesse funzionalità dell'endpoint precedente.

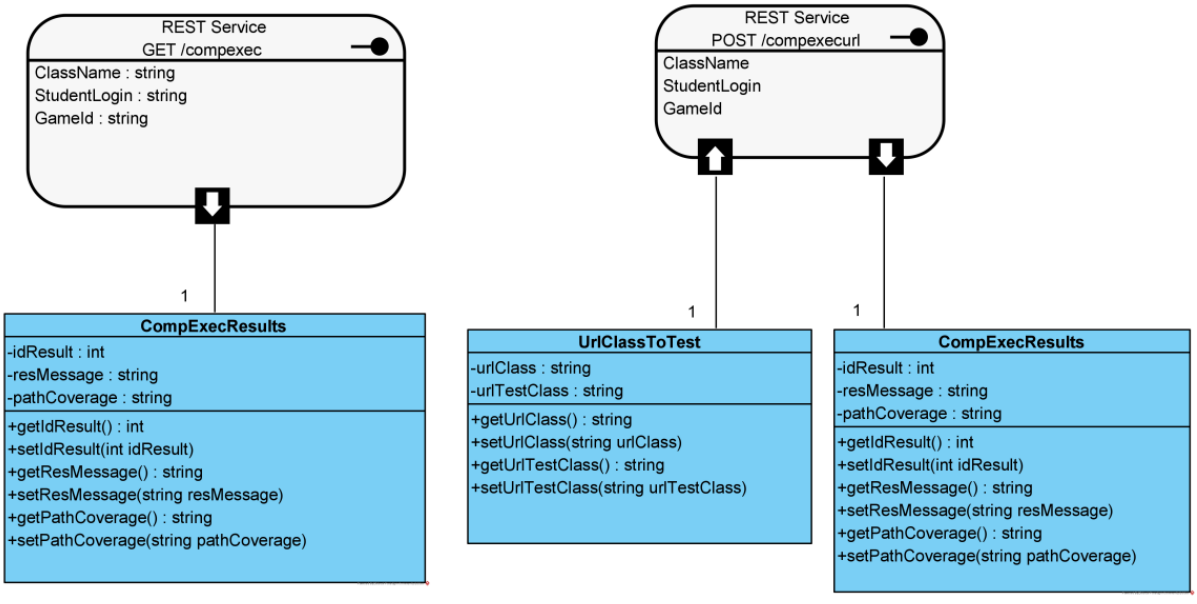

Figura 2.15: API esposta dal T7

L'utilizzo dell'API esposta però non è possibile nella sua interezza in quanto l'endpoint che permette la POST accetta la richiesta ma poi non è

in grado di posizionare effettivamente i file all'interno del File System condiviso. Un *workaround* è stato implementato come mostrato nel paragrafo precedente in modo da rendere l'applicativo funzionante. Si rimanda però alle prossime iterazione per correggere questo comportamento non corretto.

Inoltre attualmente il modulo non supporta funzionalità multiplayer. Un ulteriore sviluppo richiesto è rappresentato dall'affinamento dell'output delle richieste API: potrebbe essere più utile inviare direttamente anche il **valore di copertura** del codice all'interno del messaggio di risposta piuttosto che scrivere un file che poi deve essere letto.

È stato inoltre previsto, e aggiunto, un ulteriore **endpoint** che permette l'avvio dei robot Randoop ed Evosuite. Cosi facendo è presente in un unico container tutto il software che dovrà eseguire, compilare e testare classi. In [figura 2.16] sono presenti i nuovi endpoint.

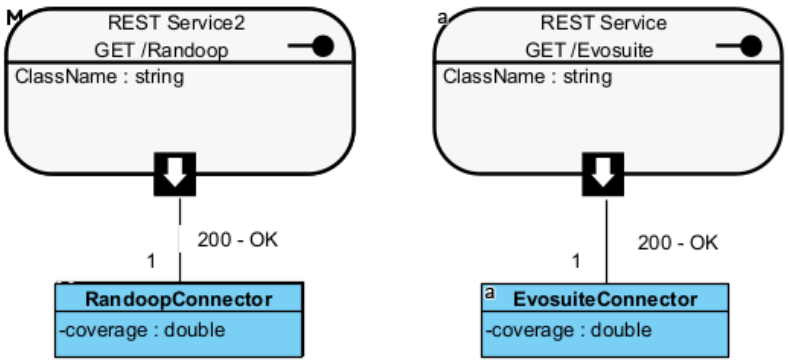

Figura 2.16: Nuovi endpoint T7

Endpoint	***Metodo***	***Descrizione***	***Parametri***
/Evosuite	GET	Restituisce la coverage del robot Evosuite.	*ClassName*: Nome della classe
/Randoop	GET	Restituisce la coverage del robot Randoop.	*ClassName*: Nome della classe

2.6 Task 8: EvoSuite

Il modulo 8 dovrebbe implementare uno dei due *sfidanti* del gioco: il robot EvoSuite. Questo non è altro che un generatore di test automatico che l'utente punta a battere nel livello di **copertura del codice** dei test.

Il modulo offre un API, implementata tramite un server **NodeJS**, per la misurazione della coverage dell'utente. Una funzionalità già implementata dal precedente modulo che però soffre delle stesse problematiche: non restituisce direttamente le statistiche del test ma le scrive all'interno di un file condiviso. Questa funzionalità non è stata quindi utilizzata.

Il robot vero e proprio è implementato come in [figura 2.17].

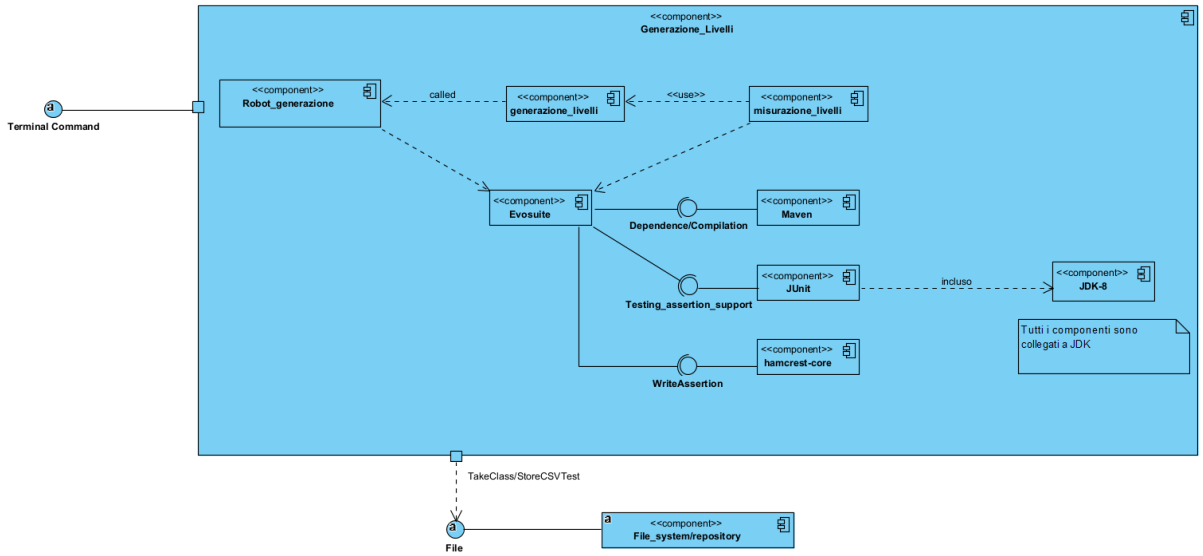

Figura 2.17: Diagramma delle componenti

Il modulo non espone quindi nessuna API e anzi utilizza una serie di file batch per la sua esecuzione. Questi file hanno bisogno per parametro il nome della classe, del package e il percorso del package. Facendo prove con i file presenti all'interno dello stesso repository il funzionamento del robot è risultato molto erratico: in base alla quantità di carico e al posizionamento o meno del file risultato non è possibile prevedere a priori se il robot riuscirà o meno nel suo compito.

Inoltre lasciare o meno un file di statistiche dei risultato del robot non è un sistema adatto al funzionamento dell'applicazione per come è stata intesa: container docker con servizi indipendenti che comunicano tra loro. Un funzionamento più adatto sarebbe stato quello di prevedere un'**API** che a cui poter inviare il nome della classe da testare e da cui arrivano i valori di **copertura** del robot.

Un'altra soluzione potrebbe essere di imitare il funzionamento del robot Randoop: esso è stato implementato in un unico file JAR che viene poi importato all'interno del componente **compilatore**. Il progetto dovrebbe quindi esporre una interfaccia semplice che garantisca l'esecuzione dei test richiamando un unica funzione con parametro il nome della classe da testare.

I tentavi di integrazione del modulo non sono andati a buon fine. Una sua integrazione avrebbe richiesto una riscrittura profonda di un modulo abbastanza confusionario. Si rimanda alle prossime iterazione per implementare tali suggerimenti e per l'integrazione completa.

Una volta effettuate le correzioni, il robot potrà essere richiamato direttamente tramite il pulsante presente all'interno dell'editor di codice.

```
1  function execRobot() {
2    var className = myParamValue
3    if (robot == "R") {
4      robotUrl = 'http://localhost:5000/Randoop?ClassName=' + className.
   replace(".java", "")
5      fetch(robotUrl)
6      .then(response => response.text())
7      .then(text => {
8        console.log(text)
9        const coverageElement = document.getElementById('covdisp');
10       coverageElement.textContent = "Coverage: " + text;
11     })
12     .catch(error => console.log('Si e verificato un errore durante il
   recupero del file TXT:', error));
13   } else {
14     robotUrl = 'http://localhost:5000/Evosuite?ClassName=' + className
15     fetch(robotUrl)
16     .then(response => response.text())
17     .then(text => {
18       console.log(text)
```

```
19          const coverageElement = document.getElementById('covdisp');
20          coverageElement.textContent = "Coverage: ";
21          coverageElement.textContent = "Coverage: " + text;
22        })
23        .catch(error => console.log('Si e verificato un errore durante il
      recupero del file TXT:', error));
24      }
25    }
```

2.7 Task 9: Randoop

Il secondo robot è implementato nel task 9, Randoop fornisce uno strumento utile alla generazione di test in maniera automatica e contribuisce a rendere la sfida dell'utente il più competitiva possibile fornendo il valore di **copertura del codice** dei test. L'accesso al servizio avviene tramite una API sullo stesso porto del compilatore all'endpoint */Randoop*. L'implementazione è avvenuta in questo modo per favorire l'accessibilità: integrando il robot all'interno del task 7 a nostro avviso si rende più chiaro il legame tra la compilazione e l'esecuzione del robot ed inoltre l'utilizzo di un Dockerfile in comune semplifica l'esecuzione del codice. L'integrazione è avvenuta tramite l'**artifact** generato dal modulo: `RandoopManager.jar`

Il Dockerfile è stato quindi modificato per la corretta fruizione del servizio, viene modificata la work directory dopo aver generato le directory utili per la compilazione e viene eseguito il file "install.sh", in seguito a queste modifiche abbiamo un'impostazione corretta dell'ambiente di sviluppo.

```
1  RUN mkdir projectUT
2  RUN mkdir tempRun
3  RUN mkdir App
4  + RUN mkdir T9
5  + COPY T9-G13-main/ T9
6  + WORKDIR "/T9"
7  + RUN bash install.sh
8  + WORKDIR "/"
9  COPY projectUT/ ProjectUT/
10 COPY compExec-0.0.1-SNAPSHOT.jar App/CompExec.jar
11 EXPOSE 8080
```

Listing 2.1: Dockerfile modificato

L'obiettivo dell'integrazione è permettere la sequenza prevista in [figura 2.18].

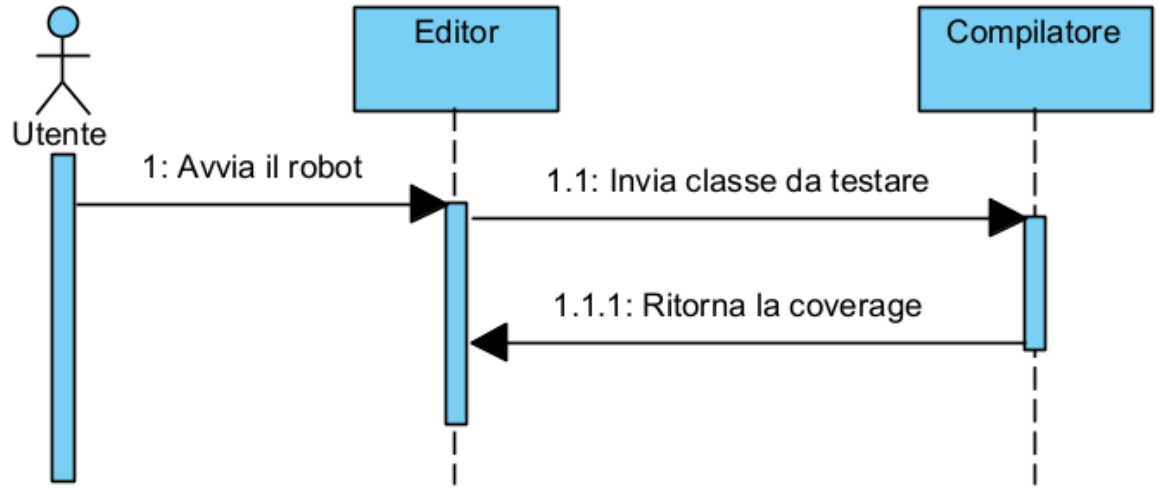

Figura 2.18: Sequenza per eseguire il robot

2.7.1 Modifiche effettuate

Il modulo nove ha però subito alcuni cambiamenti per adattare la sua esecuzione al suo ambiente. Sono stati modificate le variabili relative ai **path** per adattarle all'esecuzione del nuovo FileSystem. È stato inoltre corretto un bug che impediva l'esecuzione corretta del robot. Difatti esso rischiava di saturare il buffer output e questo di torno faceva in modo da non terminare l'esecuzione del robot. Questa modifica è presente nel file *RandoopTestGenerator.java*, in particolare alla riga 16 (si svuota il buffer in due file).

```
1  private void randoop(int timeLimit, String nomeRegr, String nomeErr, int
       seed) throws IOException,
2    String separator = ":";
3    String[] commands = {"/bin/bash","-c",""};
4    logger.info("Project in: " + PROJECT_DIR);
5    String cmd = "cd " + PROJECT_DIR + " && mvn compile && java -classpath "
6    + "../randoop-all-4.3.2.jar"
7    + separator
8    + "./target/classes/"
9    + " randoop.main.Main gentests"
10   + " --testclass=TestPackage." + INPUT_CLASSNAME
```

```
11    + " --time-limit=" + timeLimit
12    + " --regression-test-basename=" + nomeRegr
13    + " --error-test-basename=" + nomeErr
14    + " --randomseed=" + seed
15    + " --junit-output-dir=" + TEST_DIR
16    + " > stdout.log 2> stderr.log && mvn test";
17    logger.info("command: " + cmd);
18    commands[2] = cmd;
19    logger.info("Waiting for randoop");
20    Process process = Runtime.getRuntime().exec(commands);
21    //process.getInputStream().transferTo(System.out);
22    process.waitFor();
23    logger.info("Randoop Over");
24  }
```

È stato inoltre modificato il file *RandoopConnector.java* per permettere il ritorno della ***coverage*** direttamente dalla funzione che esegue i test.

```
1   private synchronized double execRandoopTest(String className, int
      maxNumberLevel) {
2    if (numberThreads < N_MAX) {
3      numberThreads++;
4      int threadIndex = 0;
5      while (busyThreads[threadIndex]) {
6        threadIndex++;
7      }
8      busyThreads[threadIndex] = true;
9      RandoopTestGenerator thread = new RandoopTestGenerator(className, this,
        maxNumberLevel, threadIndex + 1,
10     repositoryPath);
11     thread.start();
12     double coverage = -3;
13     try {
14       logger.info("Waiting for threads");
15       thread.join();
16       logger.info("Thread Over");
17       return thread.getCoverage();
18     } catch (InterruptedException e) {
19       e.printStackTrace();
20       return coverage;
21     }
22   } else {
23     requests.add(new RandoopRequest(className, maxNumberLevel));
```

```
24      return -1;
25    }
26  }
```

La funzione attende l'attesa del *thread* per restituire la **coverage** risultante. In caso di errore si restituiscono dei valori di errore (**-3**: impossibile eseguire test, **-1**: richiesta in attesa).

È stata infine migliorata la capacità di scrivere log utili da parte del modulo aggiungendo informazioni durante l'esecuzione delle procedure.

2.7.2 Cambiamenti al Task 7

L'esecuzione del robot è fatta esguire all'interno del code editor, è infatti possibile tramite l'utilizzo del bottone Run Robot che esegue la richiesta API all'endpoint /Randoop per eseguire lo script relativo alla generazione dei task ricevendo in risposta il valore di coverage da battere che viene poi mostrato a schermo.

All'interno del file *CompExecController.java* è stato aggiunto l'endpoint che effettua la chiamata alle funzione del *Randoop Manager* tramite il *Connector*:

```
1  @GetMapping("/Randoop")
2  public String Randoopest(@RequestParam String ClassName){
3    logger.info("Get Randoop Received");
4    RandoopConnector con = new RandoopConnector();
5    double coverage = -2;
6    try {
7      coverage = con.generateRandoopTest(ClassName, 3);
8    } catch (RandoopException e) {
9      e.printStackTrace();
10   }
11   return String.valueOf(coverage);
12 }
```

2.7.3 Cambiamenti al Task 6

Per poter richiamare le API appena implementate nel task 7 è stato necessario modificare il file *code-editor.html*. È stato aggiunto un pulsante che esegue il robot prescelto dall'utente. Con questo pulsante viene effettuata la GET all'endpoint del robot corrispondente. Di seguito la modifica:

```
1  function execRobot() {
2   var className = myParamValue
3   if (robot == "R") {
4     robotUrl = 'http://localhost:5000/Randoop?ClassName=' + className.
     replace(".java", "")
5     fetch(robotUrl)
6     .then(response => response.text())
7     .then(text => {
8       console.log(text)
9       const coverageElement = document.getElementById('covdisp');
10      coverageElement.textContent = "Coverage: " + text;
11    })
12    .catch(error => console.log('Si e verificato un errore durante il
     recupero del file TXT:', error));
13  } else {
14    robotUrl = 'http://localhost:5000/Evosuite?ClassName=' + className
15    fetch(robotUrl)
16    .then(response => response.text())
17    .then(text => {
18      console.log(text)
19      const coverageElement = document.getElementById('covdisp');
20      coverageElement.textContent = "Coverage: ";
21      coverageElement.textContent = "Coverage: " + text;
22    })
23    .catch(error => console.log('Si e verificato un errore durante il
     recupero del file TXT:', error));
24  }
25 }
```

Questa funzione, che verrà richiamata alla pressione del pulsante, effettua le chiamate GET al modulo sette per l'esecuzione del robot.

Guida all'utilizzo

L'utilizzo di un software di testing è essenziale per garantire la qualità e l'affidabilità del software che sviluppiamo. In questo paragrafo viene offerta una guida dettagliata sull'uso di questo strumento.

3.1 Caricamento classi

Il caricamento delle classi è un servizio riservato agli utenti dotati di diritti di amministratore, dopo aver eseguito il login come admin si può dunque procedere al caricamento delle classi da testare desiderate. Le classi in questione dovranno avere dei requisiti particolari:

- Nome della classe scritto con lettera maiuscola
- Non essere già presente all'interno delle classi inserite

3.2 Compilazione

Il task di compilazione risulta quello a cui si deve prestare maggiore attenzione al fine di un corretto funzionamento. La classe da compilare dovrà avere lo stesso nome della classe inserita e presa in analisi(con lettera maiuscola) preceduta dalla stringa "AUT". Il package da importare dovrà essere `TestPackage`. Prima di iniziare la compilazione è necessario salvare i file all'interno della cartella condivisa che indica, in primo luogo l'id del giocatore, e poi l'id della partita, tramite un sistema di sotto cartelle presente in [figura 3.1].

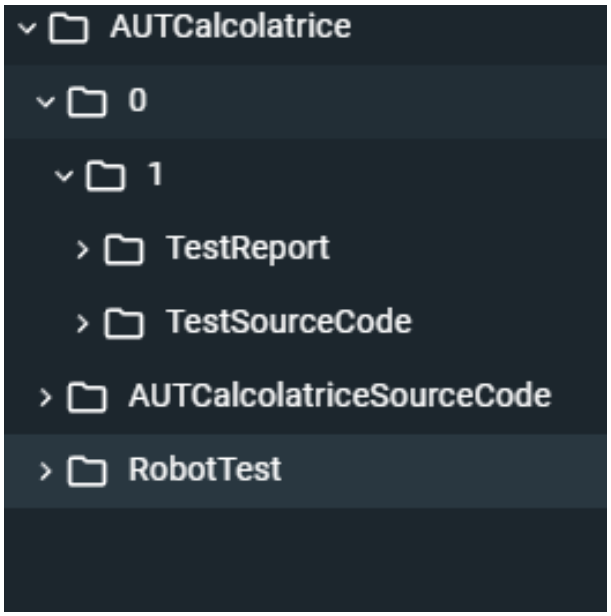

Figura 3.1: Percorso in questione

Dopo aver eseguito il salvataggio sarà possibile compilare la classe scritta, in caso di una build avvenuta con esito positivo, il risultato sarà scritto all'interno del console log del code-editor. In caso di esito negativo troveremo i file con gli errori di compilazione all'interno di una cartella nel filesystem condiviso, in particolare il percorso da seguire è: `\tempRun\numeroRunTemporanea\logexe.txt`. In [figura 3.2] un esempio di build fallita.

3.3 Randoop

La generazione dei casi di test scritti da Randoop e la misura della coverage degli stessi è implementata utilizzando il bottone di run test all'interno del code-editor. Dopo averli generati questi ultimi si troveranno all'interno della cartella del compiler nel file docker al percorso: `root\.t9_projects_test\project_n` con n numero del thread preso in considerazione.

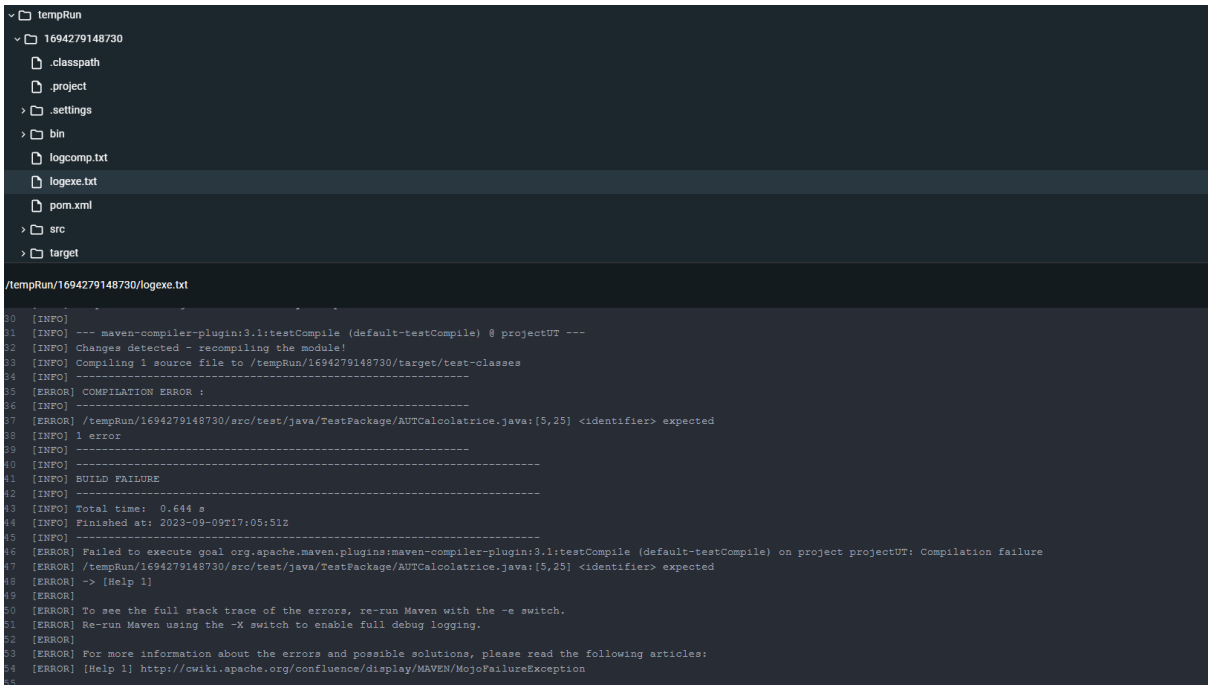

Figura 3.2: Esempio di una build fallita

3.4 Troubleshooting

In seguito verranno analizzati alcuni dei problemi più comuni riguardo l'esecuzione e l'installazione del software.

3.4.1 Problemi con l'utilizzo di Docker

I principali problemi nell'inizializzazione del progetto rigurdanti Docker risiedono nell'avvio del task 1. Nel caso in cui i due container che rappresentano il task non dovessero avviarsi sarà necessario rimuovere il contenuto della cartella al percorso: `.\Integration_SAD\T1-G20-main\T1-G20-main\classUT-repository\database` nella repository di installazione

3.4.2 Caricamento classi e visione delle stesse

Il caricamento delle classi può rappresentare un ostacolo per la corretta esecuzione del software, questo è legato principalmente agli stringenti requisiti delle classi da caricare. Nel caso ci fossero problemi nel caricamento è utile andare a rimuovere il contenuto della cartella `.\Integration_SAD\T1-G20-main\T1-G20-main\classUT-repository\ClassiUT` per poter caricare le classi in modo pulito.

Architettura Finale

L'architettura finale è presente all'interno del diagramma seguente. Esso mostra l'architettura in cui si inserisce il nostro sistema in esecuzione. Il **diagramma di deployment** mostra la configurazione dei nodi di elaborazione a tempo di esecuzione e dei componenti che vi risiedono.

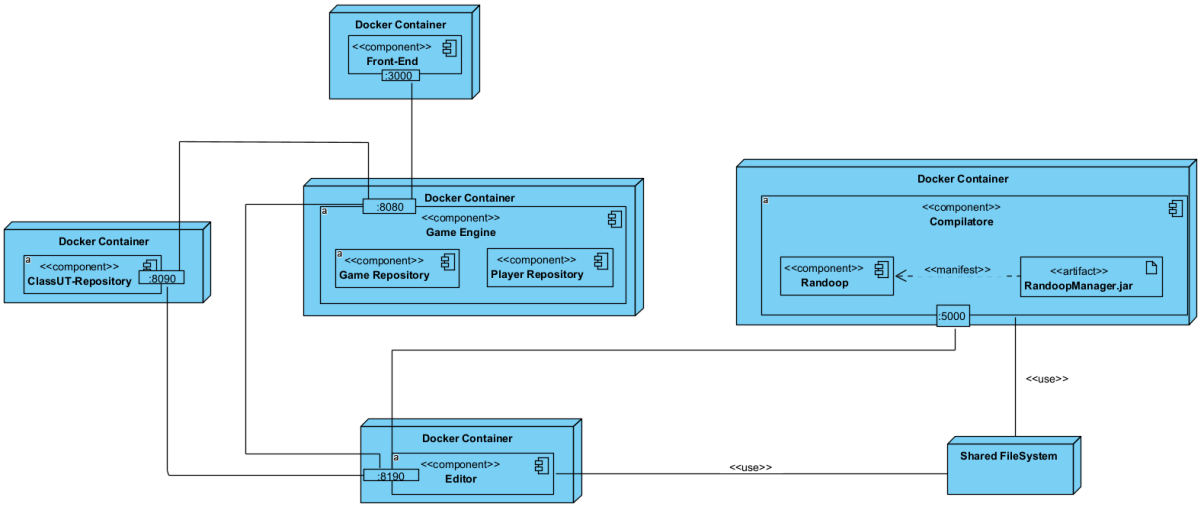

Figura 4.1: Diagramma di deployment

4.1 Installazione

Il software è pensato per essere anche facile da installare e fare il *deploy*. All'interno del repository è fornito il file **install.bat**. È necessario che la macchina su cui si intende installare il sistema rispetti questi requisiti:

- **Sistema Operativo**: Windows 11

- **CPU**: Quad Core da 2 GHz o superiore
- **RAM**: 16 Gb o superiore
- **Disco Rigido**: 20 Gb di spazio disponibile

Inoltre è necessario che sulla macchina sia installato **Docker Desktop** e **Windows Subsystem for Linux 2** (WSL 2). Una volta scaricato l'intero repository, il file batch di occuperà quindi di generare e avviare i container necessari all'applicativo.

Il contenuto del file **install.bat** è riportato di seguito:

```
::creazione del volume condiviso
docker volume create shared_data

::reset dei file necessari al container di mantenimento delle classi
cd T1-G20-main\T1-G20-main\classUT-repository
mkdir tmp
robocopy /mir tmp .\ClassiUT\
robocopy /mir tmp .\database\
rmdir tmp
::build e compose up dell'immagine
docker build -t t1 . --no-cache
docker-compose up -d

::reset dei file necessari al game engine
cd ..\..\..\T5-G5-main\T5-G5-main
xcopy build \T5-G5-main\ /s /Y
cd T5-G5-main
::build e compose up dell'immagine del front-end
docker build -t t5front . --no-cache
docker-compose up -d
cd backend\
::build e compose up dell'immagine del back-end
docker build -t t5back . --no-cache
docker-compose up -d

::creazione di immagine e container editor
cd ..\..\..\..\T6-G16-main\T6-G16-main\spring\editor
docker build -t t6 . --no-cache
::esecuzione del container su porta :8190 e montaggio del volume
 condiviso
docker run -p 8190:8080 -d -v shared_data:/FileSystem --name editor t6
```

```
31
32  ::creazione di immagine e container compilatore
33  cd ..\..\..\..\T7-G23-main\T7-G23-main
34  docker build -t t7 . --no-cache
35  ::esecuzione del container su porta :5000 e montaggio del volume
     condiviso
36  docker run -p 5000:8080 -d -v shared_data:/FileSystem --name compiler t7
```

Per informazioni sul singolo *dockerfile* per ogni modulo si rimanda alla documentazione specifica.

Testing

L'obiettivo principale del testing è di garantire la **qualità** e l'**affidabilità** del sistema o del prodotto in questione. Ciò implica la creazione di una serie di test o casi di prova che vengono eseguiti in modo sistematico per individuare eventuali difetti o comportamenti indesiderati. I risultati dei test vengono quindi analizzati per correggere gli errori e migliorare il sistema o il prodotto. Sono stati effettuati due test per l'applicazione: test di **integrazione** e l'**alphatest**.

5.1 Test di integrazione

Un **test di integrazione** è una pratica fondamentale nello sviluppo software. Serve a verificare che diverse parti o moduli di un'applicazione interagiscano correttamente tra loro. Quando si sviluppa un software complesso, spesso è necessario suddividerlo in moduli o componenti più piccoli, ognuno dei quali svolge una funzione specifica. Questi moduli devono lavorare insieme in modo sinergico affinché l'applicazione funzioni correttamente.Il test di integrazione è il passo successivo dopo il testing delle singole parti. Qui, l'attenzione si concentra sull'interazione tra questi moduli. L'obiettivo è scoprire se ci sono problemi o errori quando i moduli vengono combinati. Questi errori potrebbero emergere a causa di discrepanze nelle interfacce tra i moduli o a causa di comportamenti inaspettati quando vengono utilizzati insieme.

Per eseguire un test di integrazione, gli sviluppatori creano scenari specifici in cui i moduli interagiscono tra loro. Possono anche utilizzare strumenti

di automazione per riprodurre queste interazioni in modo sistematico. L'idea è rilevare problemi come errori di comunicazione, conflitti di dati o funzioni che non si comportano come previsto quando i moduli operano congiuntamente. Nel nostro caso il test di integrazione è stato effettuato a posteriori dello sviluppo dei moduli. In seguito è riportato il test di integrazione:

T1:

Task 3

Descrizione del Test: Login del giocatore

Azioni: Clicco sul pulsante di Login .

Output Atteso: Il login è avvenuto con successo.

Postcondizione: Avvio dell'interfaccia del task 5.

Risultato: PASS

T2. Registrazione di un amministratore

Task 2

Descrizione del Test: Test di registrazione di un amministratore con dati corretti quando l'amministratore è già registrato.

Precondizioni: L'amministratore è già registrato.

Azioni: Clicco sul pulsante di registrazione.

Output Atteso: La registrazione fallisce, l'amministratore è già registrato.

Postcondizione: Ritorno al Task 2.

Risultato: PASS

T2. Registrazione di un amministratore

Task 2

Descrizione del Test: Test di registrazione di un amministratore con dati corretti.

Precondizioni: L'amministratore non è registrato.

Azioni: Clicco sul pulsante di registrazione.

Output Atteso: La registrazione ha successo.

Postcondizione: Ritorno al Task 2.

Risultato: PASS

T3. Creazione di una partita

Task 5

Descrizione del Test: Test di creazione di una nuova partita con dati validi.

Precondizioni: Nessuna partita in corso.

Azioni: Clicca su Conferma selezione.

Output Atteso: La partita viene creata correttamente.

Postcondizione: Avvio dell'editor del Task 6.

Risultato: PASS

T4. Creazione di una partita

Task 5

Descrizione del Test: Test di creazione di una nuova partita con dati della classe testata vuoti.

Precondizioni: Nessuna partita in corso.

Azioni: Clicca su Conferma selezione.

Output Atteso: Partita non generata.

Postcondizione: Ritorno al Task 5.

Risultato: PASS

T5. Creazione di una partita

Task 5 Descrizione del Test: Test di creazione di una nuova partita con dati incompleti.

Precondizioni: Nessuna partita in corso.

Azioni: Clicca su Conferma selezione

Output Atteso: Partita non generata.

Postcondizione: Ritorno al Task 5.

Risultato: PASS

T6. Esecuzione di un caso di test

Task 6

Descrizione del Test: Test di esecuzione di un caso di test valido.

Precondizioni: Classe di test e caso di test validi.

Azioni: Salva la classe e avvia il debug.

Output Atteso: I risultati dell'esecuzione sono corretti.

Postcondizione: Avvio del compilatore del Task 7.

Risultato: PASS

T6. Esecuzione di un caso di test

Task 6

Descrizione del Test: Test di esecuzione di un caso di test con errore di sintassi nel codice.

Precondizioni: Classe di test con errore di sintassi.

Azioni: Salva la classe e avvia il debug.

Output Atteso: L'esecuzione del test fallisce.

Postcondizione: Avvio del compilatore del Task 7, con errori.

Risultato: PASS

T7. Robot Randoop

Task 5

Descrizione del Test: Avvio del robot Randoop per la generazione dei test.

Precondizioni: Nessuna partita giocata.

Azioni: Clicca su Randoop.

Output Atteso: Partita generata.

Postcondizione: Avvio del Task 9.

Risultato: PASS

T7. Storico delle partite

Task 6

Descrizione del Test: Test di consultazione dei dettagli di una partita passata da parte degli studenti.

Precondizioni: Lo studente ha effettuato almeno una partita nel passato, e ci sono partite registrate nello storico.

Azioni: Clicca sul pulsante di ricerca.

Output Atteso: Visualizzazione dei dettagli della partita selezionata, compresi i risultati, la classe testata, il robot avversario e altre informazioni rilevanti.

Postcondizione: Avvio dello storico del Task 4.

Risultato: PASS

5.2 AlphaTest

L'AlphaTest rappresenta una tappa cruciale nel ciclo di sviluppo di qualsiasi software, applicazione o prodotto destinato al pubblico. È una sorta di laboratorio virtuale in cui il software o il prodotto è sottoposto a una serie di rigorosi esami, condotti da un gruppo selezionato di utenti interni o beta tester, spesso composti da membri stessi del team di sviluppo o da figure direttamente coinvolte nel progetto.

L'obiettivo principale dell'AlphaTest è smascherare quei bug insidiosi, quei problemi di funzionalità nascosti nel codice, e tutti quei difetti che potrebbero compromettere l'esperienza dell'utente finale. Questo processo non solo riguarda la scoperta dei problemi, ma anche la loro risoluzione tempestiva. Inoltre, consente di mettere alla prova le caratteristiche chiave del software o del prodotto, garantendo che tutto funzioni come previsto.

I dati e le informazioni raccolte durante l'AlphaTest forniscono un quadro critico per la prossima fase di Beta Testing e per il processo di sviluppo nel suo complesso.È riportata la tabella dell'alphatest effettuato sull'intera applicazione:

Test Case	**Azione**	**Input**	**Risultato**	**Esito**
TC001	Avvia applicazione	N/D	Schermata iniziale	*Pass*
TC002	Registrazione utente	Dati di registrazione	Registrazione completata con successo	*Pass*
TC003	Autenticazione utente	Credenziali di autenticazione	Accesso riuscito	Pass
TC004	Consulta classi disponibili	N/D	Elenco classi visualizzato	Pass
TC005	Download del codice di una classe	Classe selezionata	Download avvenuto con successo	Pass
TC006	Aggiorna insieme di classi disponibili	Classe da aggiungere	Classe aggiunta e salvata	Pass
TC007	Avvia nuova partita del Primo Scenario	Scelte fatte per la partita	Partita creata e salvata con successo	Pass
TC008	Editor di Test Case	Classe da testare e codice	Test case creato e salvato	Pass
TC009	Compilazione ed esecuzione dei casi di test	File di test compilati ed eseguiti	Risultati ottenuti correttamente.	Pass
TC010	Esecuzione del Robot Randoop	Classe da testare	Risultati ottenuti correttamente	Pass