



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

Elaborato Finale in Software Architecture Design

*Implementazione BossRush in  
Man vs Automated Testing Tool  
challenges*

Anno Accademico 2023/2024

Professoressa  
**Anna Rita Fasolino**

Candidati  
**Luca Antonio Scolletta matr. M63001620**  
**Erika Morelli matr. M63001616**  
**Andrea Bertolero matr. M63001657**  
**Gennaro Iannicelli matr. M63001668**  
**Giuseppe Gatta matr. M63001669**  
**Nicola Garofalo matr. N46003508**

# Introduzione

L’ingegneria del software è una disciplina in continua evoluzione che elabora approcci sistematici per la progettazione, la costruzione e la manutenzione di prodotti software. Come sostenuto dalla maggior parte dei modelli che teorizzano il ciclo vita del software, come ad esempio il modello “*a cascata*”, ogni stadio è ben definito e produce dei risultati parziali, chiamati artefatti, trasferiti a una fase successiva. La realtà, però, risulta meno lineare e più complessa rispetto alla teoria: ogni step potrebbe scoprire difetti o ambiguità derivanti da uno stadio precedente: è necessario un ritorno a fasi trascorse per correggere gli errori rilevati al fine di proseguire correttamente. Dati sperimentali, raccolti dall’osservazione di progetti industriali, dimostrano che il costo della rimozione di un errore dopo la fase di rilascio è decisamente più elevato rispetto all’eliminazione in fasi precedenti. La fase di testing del software è fondamentale, ma spesso è considerata marginale o trascurabile, non viene eseguita secondo i consolidati criteri dell’ingegneria e, di conseguenza, si rilasciano applicazioni software che non rispettano i requisiti [5].

---

Il progetto ENACTEST, promosso dal programma *Erasmus+*, ha la durata triennale 2022-2025 e si pone l’obiettivo di affrontare le problematiche legate alla fase di testing. Negli ultimi anni si è rilevato un divario notevole tra le competenze richieste dall’industria, le esigenze di apprendimento degli studenti e il modo in cui sono insegnate le tecniche di testing nelle istituzioni di istruzione superiore. ENACTEST identifica e progetta strategie di insegnamento efficaci per l’attività di testing, a tal fine è coinvolto un consorzio composto da nove partner, tra cui istituti di istruzione superiore, centri di formazione professionale e piccole e medie imprese [6].

Questo ducumento fa riferimento al lavoro svolto dai teams **A6** e **A11** in merito allo sviluppo del requisito **R10**, approfondito nel paragrafo 1.3, assegnato dalla professoressa **Anna Rita Fasolino**, docente del corso di *Software Architecture Design* durante l’anno accademico 2023/2024 presso l’*Università degli Studi di Napoli Federico II*.

# Indice

|          |                                     |           |
|----------|-------------------------------------|-----------|
| <b>1</b> | <b>Progetto</b>                     | <b>1</b>  |
| 1.1      | ENACTEST . . . . .                  | 1         |
| 1.2      | Descrizione del progetto . . . . .  | 3         |
| 1.3      | Requisito R10 . . . . .             | 4         |
| 1.4      | Strumenti . . . . .                 | 6         |
| 1.4.1    | Discord . . . . .                   | 6         |
| 1.4.2    | Trello . . . . .                    | 6         |
| 1.4.3    | Miro . . . . .                      | 7         |
| 1.4.4    | Eclipse e VSCode . . . . .          | 7         |
| 1.4.5    | Maven . . . . .                     | 8         |
| 1.4.6    | Spring Boot . . . . .               | 8         |
| 1.4.7    | Docker . . . . .                    | 9         |
| 1.4.8    | GitHub . . . . .                    | 9         |
| <b>2</b> | <b>Specifiche dei requisiti</b>     | <b>11</b> |
| 2.1      | Descrizione del requisito . . . . . | 11        |
| 2.1.1    | Requisiti funzionali . . . . .      | 11        |
| 2.1.2    | Requisiti non funzionali . . . . .  | 12        |
| 2.2      | User Stories . . . . .              | 13        |

|          |  |           |
|----------|--|-----------|
| 2.2.1    | Criteri di accettazione . . . . .                | 14        |
| 2.3      | Use Case Diagram . . . . .                       | 15        |
| 2.4      | Scenari di Utilizzo . . . . .                    | 17        |
| 2.4.1    | Scenario1: Avvio di una nuova partita "BossRush" | 17        |
| 2.4.2    | Scenario2: Svolgimento della partita . . . . .   | 19        |
| 2.4.3    | Scenario3: StoricoPartite . . . . .              | 20        |
| <b>3</b> | <b>Stato iniziale del progetto</b>               | <b>22</b> |
| 3.1      | Architettura del sistema . . . . .               | 22        |
| 3.1.1    | Task T5 . . . . .                                | 26        |
| 3.1.2    | Task T6 . . . . .                                | 28        |
| 3.1.3    | Interazione tra T5 e T6 . . . . .                | 30        |
| <b>4</b> | <b>Modifiche al task 5 e al task 6</b>           | <b>34</b> |
| 4.1      | Analisi preliminare e problematiche . . . . .    | 35        |
| 4.1.1    | Problematiche generali . . . . .                 | 35        |
| 4.1.2    | Problematiche relative alla compilazione . . . . | 36        |
| 4.1.3    | Problematiche legate al salvataggio . . . . .    | 37        |
| 4.2      | Descrizione delle chiamate API . . . . .         | 40        |
| 4.2.1    | Salvataggio iniziale . . . . .                   | 40        |
| 4.2.2    | Esecuzione RUN . . . . .                         | 41        |
| 4.2.3    | Recupero coverage giocatore . . . . .            | 41        |
| 4.2.4    | Recupero coverage robot . . . . .                | 41        |
| 4.2.5    | Aggiornamento turno, round e game . . . . .      | 42        |
| 4.3      | Modifiche a T5 . . . . .                         | 42        |
| 4.3.1    | Analisi del file editor.html . . . . .           | 44        |
| 4.3.2    | Modifiche al file editor.html . . . . .          | 50        |

|          |  |            |
|----------|--|------------|
| 4.4      | Modifiche a T6 . . . . .                       | 54         |
| 4.4.1    | Codice: RunnerHelper . . . . .                 | 65         |
| <b>5</b> | <b>Organizzazione del lavoro</b>               | <b>75</b>  |
| 5.1      | Organizzazione del lavoro . . . . .            | 75         |
| 5.1.1    | Agile: Scrum . . . . .                         | 75         |
| 5.2      | Prima iterazione . . . . .                     | 76         |
| 5.3      | Seconda iterazione . . . . .                   | 78         |
| 5.4      | Terza iterazione . . . . .                     | 80         |
| 5.5      | Quarta iterazione . . . . .                    | 84         |
| 5.6      | Quinta iterazione . . . . .                    | 85         |
| 5.7      | Considerazioni e sviluppi futuri . . . . .     | 86         |
| <b>6</b> | <b>Testing</b>                                 | <b>88</b>  |
| 6.1      | Testing dell'applicazione . . . . .            | 88         |
| <b>7</b> | <b>Guida all'installazione</b>                 | <b>94</b>  |
| 7.1      | Installazione . . . . .                        | 94         |
| 7.1.1    | Prerequisiti per l'installazione . . . . .     | 94         |
| 7.1.2    | Download di Docker Desktop . . . . .           | 95         |
| 7.1.3    | Utilizzo dell'applicazione . . . . .           | 97         |
| 7.1.4    | Ricompilazione dei codici di T5 e T6 . . . . . | 98         |
| <b>8</b> | <b>Glossario dei nomi</b>                      | <b>100</b> |
| 8.1      | Robot . . . . .                                | 100        |
| 8.2      | BossRush . . . . .                             | 100        |
| 8.3      | Classica . . . . .                             | 100        |
| 8.4      | Turno . . . . .                                | 101        |

|                         |     |
|-------------------------|-----|
| 8.5 Giocatore . . . . . | 101 |
|-------------------------|-----|

# Capitolo 1

## Progetto

Il primo capitolo è dedicato all’approfondimento del progetto ENACTEST e all’introduzione del progetto Man vs Automated Testing Tool Challenges. Inoltre, si presenta brevemente il requisito assegnato e si descrive, nel paragrafo conclusivo, l’organizzazione del lavoro svolto durante le varie iterazioni e gli strumenti utilizzati.

### 1.1 ENACTEST

Il progetto European Innovation Alliance for Testing Education, **ENACTEST**<sup>1</sup>, è un progetto internazionale nato con l’obiettivo di enfatizzare l’importanza del testing e una corretta implementazione durante lo sviluppo di prodotti software. Ad oggi, il testing è considerato una fase trascurabile e marginale, a cui dedicare tempo solo alla fine delle fasi di sviluppo. L’obiettivo di ENACTEST è incentivare il valore e

---

<sup>1</sup><https://enactest-project.eu/it/>

la parità tra la fase di sviluppo e quella di testing al fine di fornire ad entrambe la giusta importanza. In questo contesto, l'*Università degli Studi di Napoli Federico II* partecipa attivamente al progetto ENACTEST e collabora con molteplici piccole imprese per la promozione e consolidazione dell'importanza del testing. L'approccio e le idee in-

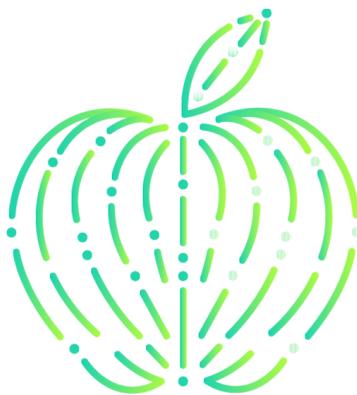


Figura 1.1: Logo ENACTEST [4]

novative di ENACTEST si sposano perfettamente con lo sviluppo di un *educational game* in cui l'utente è invitato a sfidare strumenti in grado di generare autonomamente casi di test con l'obiettivo di coprire esaustivamente tutti i possibili scenari che potrebbero verificarsi.

Dunque, il progetto ENACTEST è un metodo innovativo capace di offrire un ottimo compromesso per la formazione di una figura professionale con solide basi fornite dal classico insegnamento teorico e competenze pratiche richieste dal mondo del lavoro. In conclusione, gli studenti raggiungeranno vari obiettivi di apprendimento, tra cui: l'applicazione di diverse metodologie di progettazione dei test, l'utilizzo del framework di test JUnit per implementare i propri test e l'analisi

dell’impatto degli strumenti di test automatizzati nei processi di test, esplorandone le caratteristiche positive e negative.

## 1.2 Descrizione del progetto

**Man vs. Automated Testing Tools challenges** è il gioco educativo, promosso dal partner *UNINA*, al fine di perseguire gli obiettivi di ENACTEST: lo studente è stimolato a sfidare i tools *EvoSuite* e *Randoop*, chiamati anche **Robot**, capaci di generare automaticamente casi di test JUnit. La competizione consiste nel raggiungere un determinato livello di copertura, intesa come *Lines Of Code coverage*.

Lo sviluppo dell’applicazione ha previsto la collaborazione di più gruppi con l’obiettivo comune di soddisfare i vari requisiti dell’applicazione tramite la realizzazione di singoli componenti, detti task, successivamente integrati tra loro. La documentazione specifica dei vari task è disponibile sul sito GitHub all’indirizzo:

- <https://github.com/Testing-Game-SAD-2023>

dove è possibile consultare anche l’implementazione relativa ai diversi requisiti assegnati a ciascun gruppo. La realizzazione del requisito R10 e dei task T6 e T7 ha richiesto di esaminare le risorse presenti nella repository Github:

- <https://github.com/Testing-Game-SAD-2023/A9-R9/tree/master>

mentre, l’implementazione eseguita dai gruppi A6 e A11 è stata caricata in

- <https://github.com/Testing-Game-SAD-2023/A6-2024>

dove è presente la documentazione, il codice e un video dimostrativo.

### 1.3 Requisito R10

Nella sua versione originale, il sistema consentiva al giocatore di avviare le singole istanze di partita sfidando un solo robot per volta, eventualmente scegliendone il livello di copertura del codice, su una singola classe tra quelle disponibili. Con l’implementazione della modalità di gioco *BossRush*, l’obiettivo è quello di portare la sfida su un nuovo livello di difficoltà che possa intrattenere anche gli studenti già esperti e che possa diventare un fine da raggiungere per tutti i nuovi giocatori che si affacciano per la prima volta al mondo del testing.

Il requisito **R10** definisce le seguenti funzioni da realizzare:

- Aggiunta di una nuova modalità di gioco, chiamata “*BossRush*”, in cui il giocatore sfida tutti i Robot disponibili, eventualmente, entro un tempo massimo
- Si assume che ci sia un Robot diverso per ogni coppia Robot+Livello.  
Esempio: se ci sono 2 livelli del Robot Evosuite, si dovrà presentare ciascun livello come se fosse un Robot diverso

- Ad ogni turno giocato, ovvero ogni sfida contro i Robot, l'applicazione dovrebbe mostrare un messaggio riguardo quanti e quali Robot sono stati battuti con le relative percentuali di copertura

In sintesi, assumendo che ci siano un numero  $n$  di Robot tipo Evo-Suite disponibili e un numero  $m$  di Robot tipo Randoop, il giocatore sfiderà contemporaneamente  $n + m$  Robot. Al termine di ogni round è possibile scoprire quali Robot sono stati sconfitti e la relativa soglia di LOC raggiunta. Si riportano i seguenti esempi di un possibile messaggio di output:

- *Bravo! Hai totalizzato 75% LOC coverage e vinto contro Robot 1 65%, Robot 4 68%. Ti restano da battere Robot 2, Robot 3 e Robot 5*
- *Riprova! Non hai battuto neanche un Robot*
- *Eccellente: Hai battuto tutti i Robot!*

Infine, l'implementazione del requisito R10, relativo alla modalità di gioco *BossRush*, è stata conseguita attraverso un'approfondita analisi e comprensione dei task **T5** e **T6**, realizzata mediante uno studio accurato sia del codice sorgente sia della documentazione resa disponibile su GitHub.

## 1.4 Strumenti

### 1.4.1 Discord

Discord è un servizio di comunicazione vocale, video e testuale usato da oltre cento milioni di persone, scelto per la tecnologia video e audio a bassa latenza che ha permesso sessioni di sviluppo ben organizzate [1].

### 1.4.2 Trello

Le bacheche, le liste e le schede di Trello hanno consentito ai gruppi di passare in un batter d'occhio dalla teoria alla pratica grazie al suo carattere visivo e intuitivo, Trello aiuta i gruppi a dare vita ai progetti e proseguirne la realizzazione nel modo più efficiente possibile. La possibilità di avere sotto controllo ogni sprint ha consentito di restare in carreggiata nel raggiungimento di tutti gli obiettivi entro i tempi previsti. In aggiunta, la Timeline mostra in che modo tutti gli elementi dinamici si uniscono nel corso del tempo [3].



Figura 1.2: Icona di Trello

### 1.4.3 Miro

Miro dispone di avanzate funzionalità native pronte all'uso che consentono ai team di sviluppare la propria visione all'insegna della creatività e della collaborazione. Si semplificano le operazioni complesse ottenendo risultati migliori con la visualizzazione, riportata in figura, dei progetti costruiti, inoltre, si osservano i rapporti e le dipendenze fra le attività attraverso le Miro cards [2].



Figura 1.3: Logo Miro

### 1.4.4 Eclipse e VSCode

IDE per la programmazione Java attraverso i quali è stata possibile l'effettiva implementazione del software. In particolare, questi strumenti consentono la scrittura e il testing del codice, nonché la gestione delle dipendenze (in particolare attraverso eventuali estensioni quali Maven) e la compilazione dell'applicazione.

### 1.4.5 Maven

Maven è uno strumento di gestione del progetto e di build utilizzato nello sviluppo software. Si occupa della compilazione, del packaging e della distribuzione di progetti, semplificando la gestione delle dipendenze e standardizzando la struttura dei progetti. Il POM (Project Object Model) è un file XML fondamentale in Maven che definisce le informazioni di configurazione del progetto, inclusi gruppo di coordinamento, artefatto, versione, dipendenze, plugin e configurazioni di build. Il POM fornisce una struttura organizzata per la gestione centralizzata dei progetti in Maven.

### 1.4.6 Spring Boot

Spring Boot è un framework Java open-source progettato per semplificare lo sviluppo di applicazioni Java. La sua principale idea è quella di fornire un'infrastruttura già pronta per affrontare le sfide comuni nello sviluppo software. Questo significa che gli sviluppatori possono concentrarsi maggiormente sulla logica del proprio business anziché dedicare tempo alla configurazione e all'implementazione tecnica.

Le caratteristiche principali di Spring Boot includono la gestione delle dipendenze, la configurazione automatica e la possibilità di integrare server Web direttamente nell'applicazione. Inoltre, Spring Boot offre un ambiente di sviluppo e test molto più facile e intuitivo.

Uno dei punti di forza di Spring Boot è la sua capacità di semplificare notevolmente la creazione di applicazioni Java. Questo framework è apprezzato per la sua scalabilità, robustezza e facilità di gestione. In poche parole, Spring Boot è una scelta popolare per lo sviluppo rapido di applicazioni Java che devono essere facilmente scalabili e gestibili.

### 1.4.7 Docker

Piattaforma di containerizzazione che consente di rinchiudere i vari servizi di cui si compone l'intera applicazione in vari container indipendenti tra loro. L'utilizzo di questa piattaforma consente l'implementazione di un'architettura a microservizi e apporta numerosi vantaggi tra cui la semplicità di distribuzione dell'applicazione su server e dispositivi remoti.

### 1.4.8 GitHub

Piattaforma di hosting del codice, da esso è stato possibile biforcicare l'applicazione preesistente (versione del gruppo A3, aggiornata in gennaio 2024) e generare una nuova repository in cui completare le modifiche relative all'implementazione della nuova modalità di gioco.

# CAPITOLO 1. PROGETTO

---

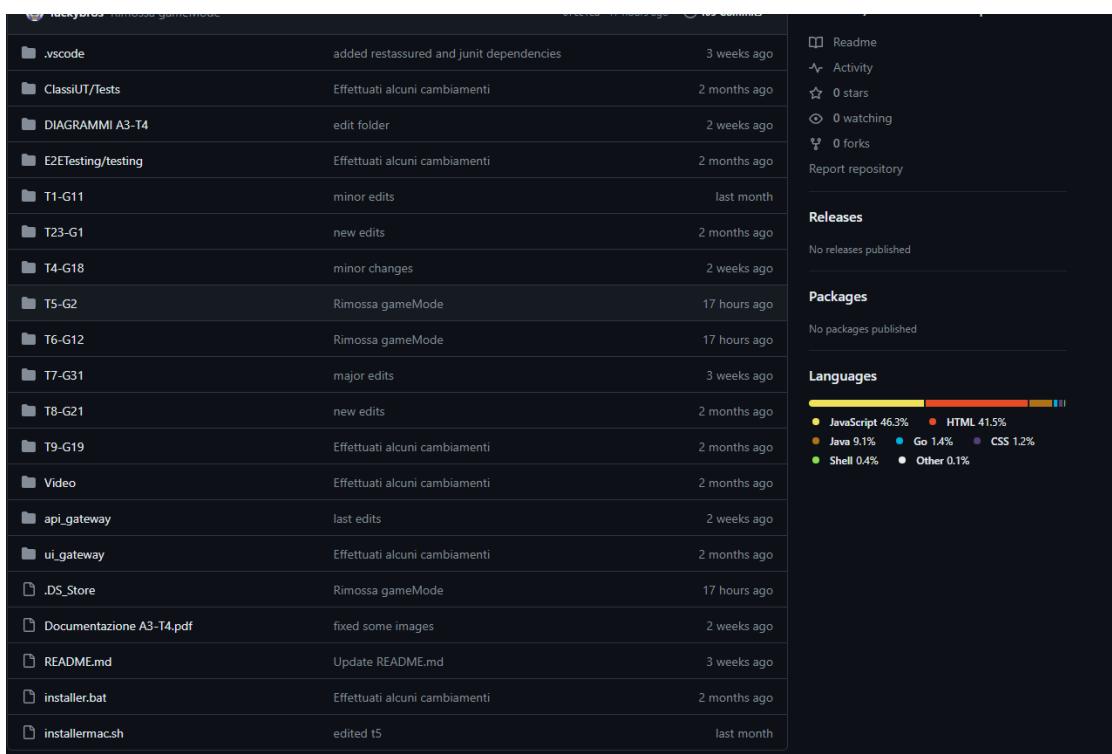


Figura 1.4: Repository del gruppo A6-A11 per l'implementazione della modalità BossRush

# Capitolo 2

## Specifiche dei requisiti

In questa sezione, verranno dettagliatamente delineati i requisiti del sistema, a partire da una rapida analisi della traccia del requisito R10.

La specifica dei requisiti è un passo cruciale per comprendere appieno le esigenze del cliente e garantire la creazione di un sistema software che soddisfi le aspettative.

### 2.1 Descrizione del requisito

#### 2.1.1 Requisiti funzionali

I requisiti funzionali rappresentano le funzionalità specifiche che il sistema deve fornire per soddisfare le esigenze degli utenti. Essi delineano le azioni e le operazioni che il software deve essere in grado di eseguire. I requisiti funzionali definiti per la modalità BossRush sono:

- **RF001 - Sfida tutti i Robot Disponibili:** gli utenti devono poter sfidare tutti i robot disponibili contemporaneamente.
- **RF002 - Calcolo delle Percentuali di Copertura:** dopo ogni turno giocato, l'applicazione deve calcolare le percentuali di copertura ottenute dall'utente e mostrare questo dato nel messaggio di risultato della partita.
- **RF003 - Visualizzazione dei Risultati:** Alla fine della partita, l'applicazione deve mostrare un messaggio con i robot battuti, le relative percentuali di copertura e i robot ancora da battere.
- **RF004 - Visualizzazione dello storico partita:** il giocatore nel visualizzare il proprio storico partite deve chiaramente poter distinguere tra dati ottenuti in modalità di gioco classica e modalità di gioco "BossRush".

### 2.1.2 Requisiti non funzionali

I requisiti non funzionali sono vincoli e caratteristiche di sistema che riguardano aspetti come la performance, la sicurezza, l'usabilità e altri attributi di qualità. Per l'implementazione di R10 sono stati definiti i seguenti requisiti non funzionali:

- **RNF001 - Performance:** L'applicazione deve rispondere in modo rapido alle azioni degli utenti, fornendo una esperienza di gioco fluida e senza ritardi.

- **RNF002 - Usabilità:** L’interfaccia utente deve essere intuitiva e facile da usare, garantendo che gli utenti possano navigare facilmente tra le diverse opzioni e comprendere le informazioni presentate.
- **RNF003 - Affidabilità:** L’applicazione deve essere affidabile e resistente agli errori, gestendo correttamente le eccezioni e garantendo che i dati dei giocatori siano protetti da perdite o danni.
- **RNF004 - Scalabilità:** L’applicazione deve essere progettata per gestire un numero crescente di giocatori e partite, senza compromettere le prestazioni o l’esperienza utente.

## 2.2 User Stories

Le User Stories sono brevi descrizioni narrative delle funzionalità del sistema dalla prospettiva dell’utente, utilizzate in metodologie agili come Scrum. Devono seguire il paradigma **INVEST** e possedere le seguenti caratteristiche:

1. **Indipendenti:** evitare stime inaccurate e rigidità nella pianificazione.
2. **Negoziabili:** descrivono l’essenza senza dettagli; i dettagli sono negoziati.

3. **Di valore:** devono portare benefici all'utente finale.
4. **Stimabili:** la stima del tempo è fondamentale; utilizzare analisi (spike) se necessario.
5. **Della giusta dimensione:** evitare storie troppo grandi o piccole.
6. **Testabili:** deve includere criteri di accettazione misurabili attraverso test.

### 2.2.1 Criteri di accettazione

In SCRUM, i criteri seguono il modello "Given-When-Then", assicurando l'allineamento tra il lavoro del team e le aspettative dell'utente. Forniscono chiarezza sulla definizione delle funzionalità, servono come metriche di valutazione e evitano fraintendimenti. Sono state create 4 User Stories in iterazioni successive per stabilire gli obiettivi del progetto. Tutte le stories utente sono state portate a termine seguendo il processo iterativo descritto nel Capitolo 5. La lista completa delle User Stories implementate è riportata in figura.

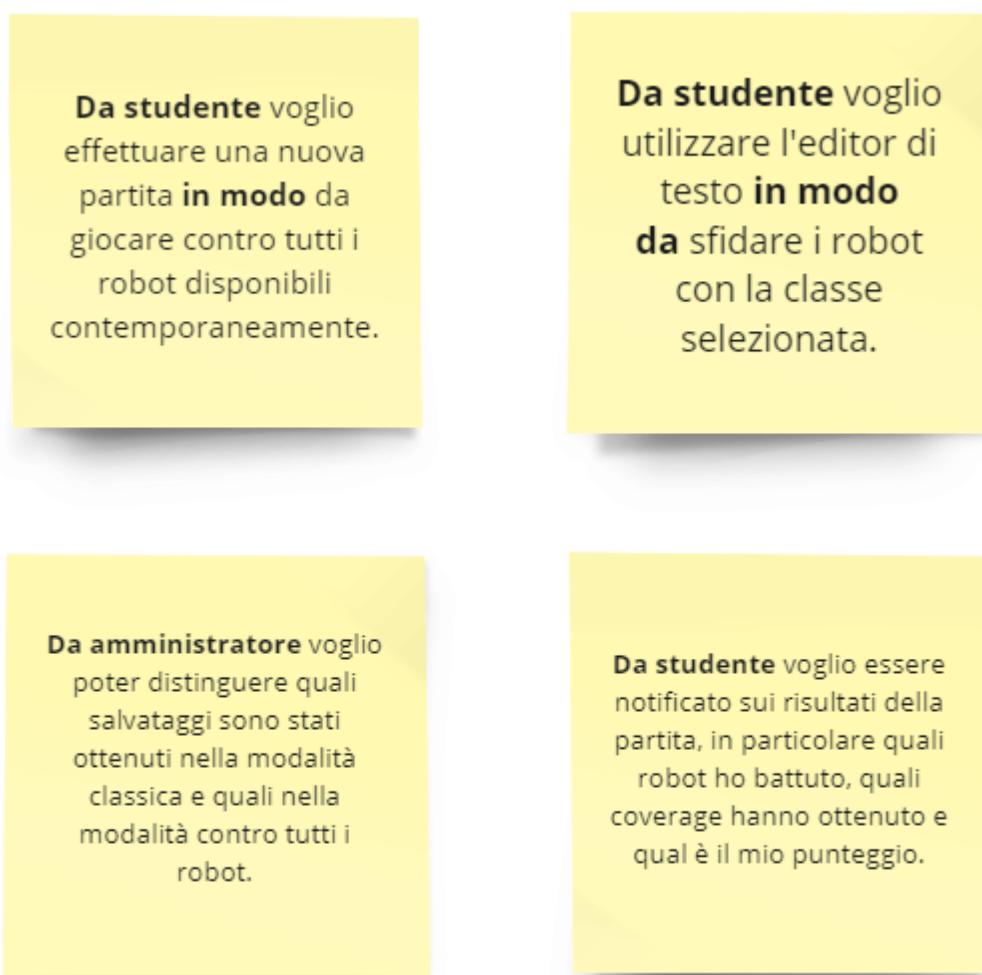


Figura 2.1: User stories definite per l'implementazione del requisito R10

## 2.3 Use Case Diagram

Il diagramma dei casi d'uso visualizza le interazioni tra attori e il sistema, offrendo una panoramica delle funzionalità. Fondamentale per la comprensione delle implementazioni, facilita la comunicazione, supporta l'analisi dei requisiti e fornisce una base per la progettazione

dell'architettura del software. In figura è possibile visualizzare lo use case diagram ideato per l'implementazione del nuovo requisito.

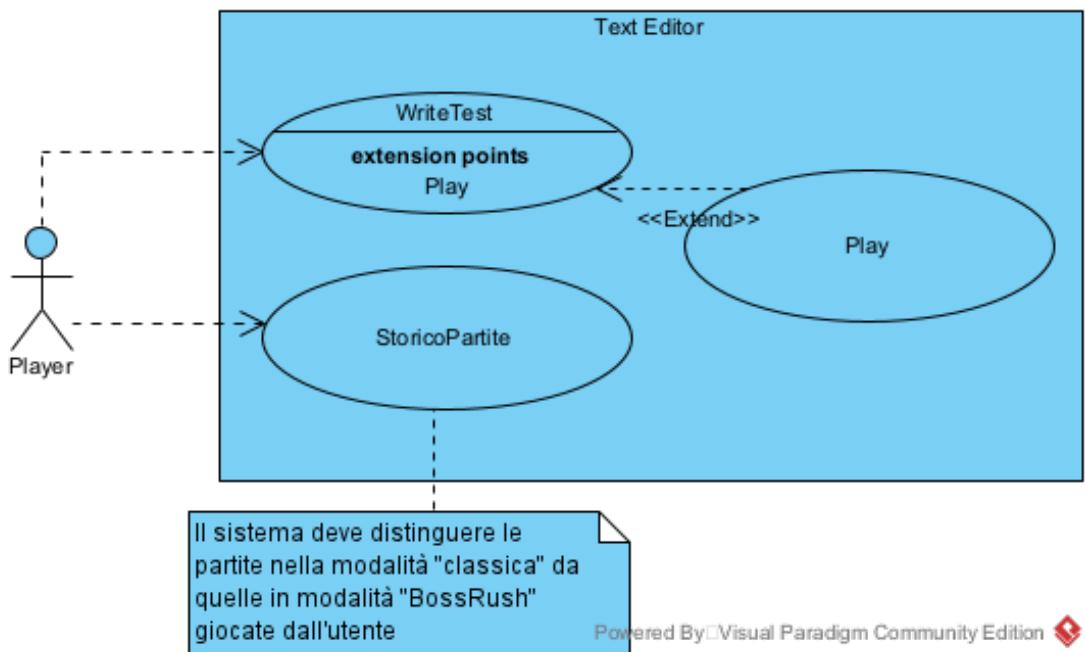


Figura 2.2: Diagramma Dei Casi D'Uso

Dal diagramma si evincono le due funzionalità base implementate nel sistema, che sono:

1. WriteTest/Play: l'utente scrive i casi di test JUnit, dopodiché premendo il bottone *Play* verifica l'eventuale vittoria o sconfitta contro tutti i robot in sfida.
2. StoricoPartite: l'utente può visualizzare lo storico delle sue partite, nel sistema devono chiaramente essere distinte le partite in base alla modalità di gioco.

## 2.4 Scenari di Utilizzo

Gli scenari di utilizzo descrivono interazioni specifiche tra utenti e sistema, cruciali durante l'analisi dei requisiti e la progettazione del software. Apportano vantaggi come l'identificazione dei requisiti funzionali, la validazione e la guida alla progettazione dell'interfaccia utente, essenziali per comprendere le esigenze degli utenti e orientare lo sviluppo del sistema. Di seguito si lasciano al lettore gli scenari di gioco definiti per il requisito implementato:

### 2.4.1 Scenario1: Avvio di una nuova partita "BossRush"

**Attore principale:** Giocatore.

**Trigger:** Il giocatore decide di avviare una nuova partita nella modalità "BossRush".

**Precondizioni:** Il giocatore ha eseguito l'accesso al sistema.

**Flusso Principale:**

1. Il giocatore seleziona la modalità di gioco "BossRush".
  - **Scenario alternativo:** Il giocatore preme "submit" prima di selezionare la classe con cui giocare.

- Il sistema risponde con un messaggio di errore.
2. Il giocatore seleziona la classe con cui giocare.
  3. Il giocatore preme il pulsante "submit".
  4. L'applicazione avvia la partita e mostra il primo turno.

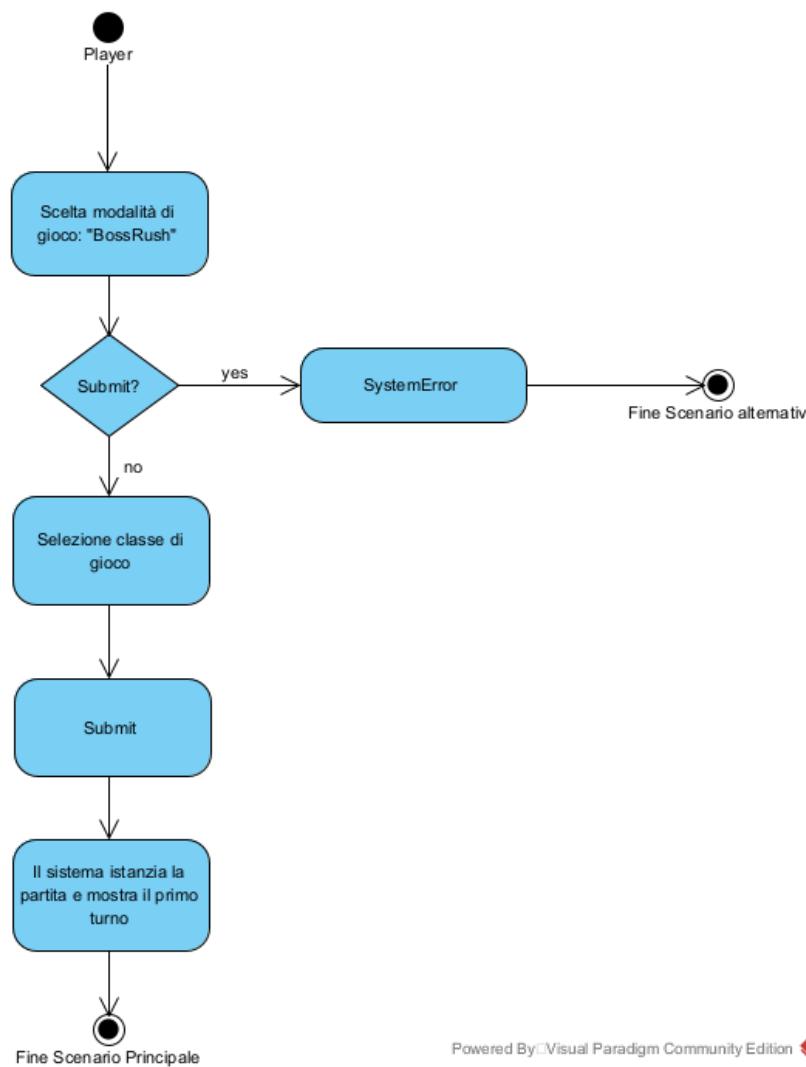


Figura 2.3: Activity diagram relativo allo scenario 1

### 2.4.2 Scenario2: Svolgimento della partita

**Attore principale:** Giocatore.

**Trigger:** Il giocatore sta svolgendo una partita nella modalità "BossRush" e preme il pulsante "*Play*".

**Precondizioni:** Il giocatore ha eseguito l'accesso al sistema, ha selezionato la modalità "BossRush" e la classe da testare.

#### Flusso Principale:

1. Il giocatore preme il bottone "Compile" per eseguire la compilazione del codice scritto da lui.

- **Scenario alternativo:** il codice dell'utente presenta errori di compilazione.
- Il sistema mostra un errore di compilazione e la partita non può essere continuata.

2. Il giocatore preme il bottone "*Play*".
3. Il sistema elabora l'azione, calcola le percentuali di copertura del codice ottenute dal giocatore e recupera quelle dei robot.
4. Il sistema mostra un messaggio con i risultati della sfida, indicando la vittoria parziale, totale o sconfitta del giocatore.

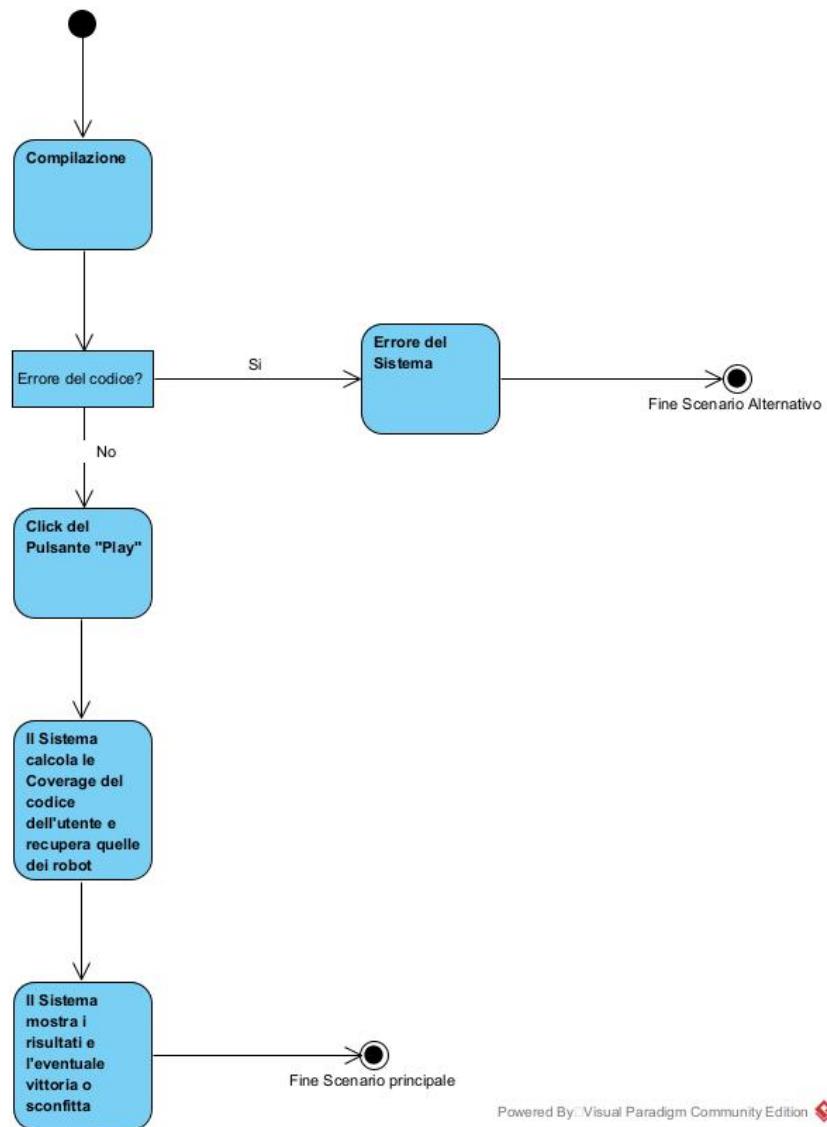


Figura 2.4: Activity diagram relativo allo scenario 2

### 2.4.3 Scenario3: StoricoPartite

**Attore principale:** Giocatore.

**Trigger:** Il giocatore è nella schermata principale del sistema e preme il pulsante di visualizzazione dello storico partite.

**Precondizioni:** Il giocatore ha eseguito l'accesso al sistema.

**Flusso Principale:**

1. Il giocatore preme il pulsante "StoricoPartite"
2. Il sistema mostra una lista, eventualmente vuota, con lo storico delle partite giocate dall'utente
3. Lo storico rende evidente la distinzione tra i dati ottenuti dalla modalità classica e la modalità contro tutti i robot.



Figura 2.5: Activity diagram relativo allo scenario 3

# Capitolo 3

## Stato iniziale del progetto

Al fine di ottenere una comprensione complessiva di tutta l'architettura software, questo capitolo tratterà una sintesi della struttura nella sua interezza analizzando ciò che ci è stato lasciato in eredità dai nostri colleghi, ponendo particolare attenzione al funzionamento dei componenti T5 e T6, la cui modifica successiva è stata fondamentale per ottenere il raggiungimento dell'obiettivo preposto.

### 3.1 Architettura del sistema

Lo studio dello stato iniziale del sistema è partito dalla comprensione dell'architettura a microservizi di cui il gioco è composto. In particolare, si è dedotto che l'architettura è suddivisa in componenti (anche detti task) implementati con uno stile client - server in cui il client

accede all'applicazione esclusivamente interfacciandosi con i due gateway disponibili detti (*UI Gateway* e *API Gateway*). Vediamo nello specifico il funzionamento dei due elementi:

- L'**UI Gateway** gestisce il routing delle richieste http relative al front - end (UI - User Interface)
- L'**API Gateway** ha il compito di gestire l'autenticazione e l'autorizzazione delle richieste relative alle API del sistema

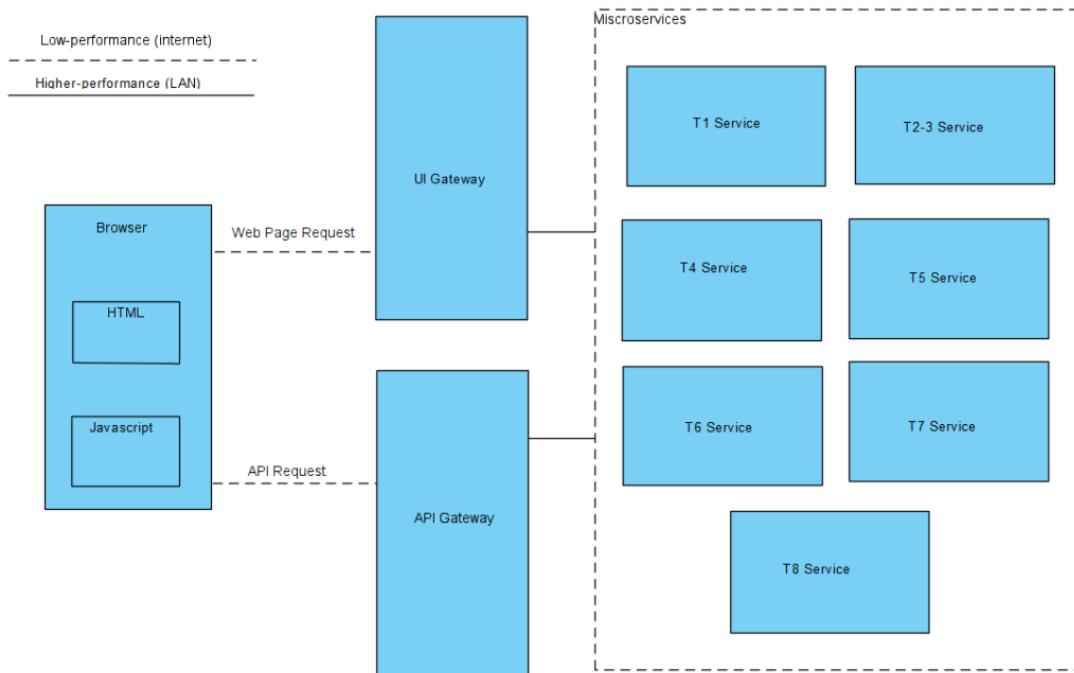


Figura 3.1: Vista dell'architettura a microservizi

L'utilizzo di questo tipologia di architettura, affiancata ad uno stile client - server, ha portato all'avvento dei seguenti vantaggi:

- Migliore scalabilità dell'applicazione;

## CAPITOLO 3. STATO INIZIALE DEL PROGETTO

- Testing del software semplificato;
- Sviluppo e implementazione, in termini di codice, semplificato.

L'architettura complessiva è osservabile nella figura in basso, in cui è mostrato il diagramma a componenti e connettori del sistema complessivo.

**OSS:** i componenti di colore giallo sono quelli che sono stati modificati per l'implementazione del requisito R10.

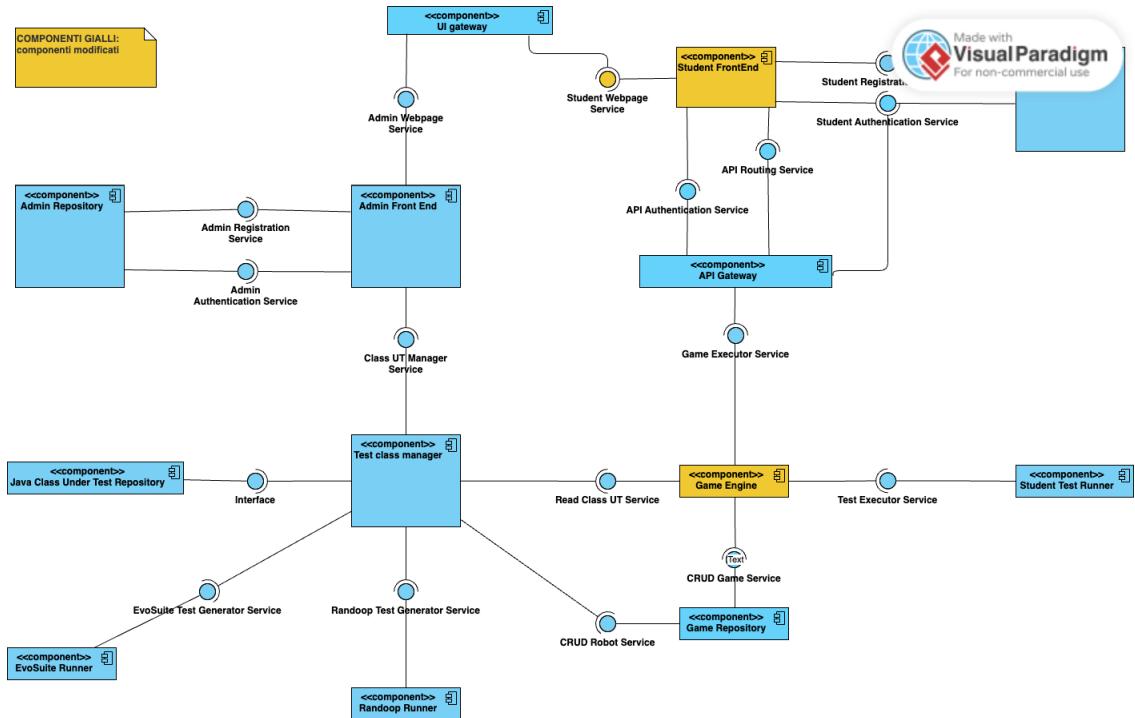


Figura 3.2: Vista dei componenti e connettori

Relativamente al diagramma dei componenti e connettori appena mostrato, il nostro lavoro si è concentrato, per quanto concerne il requisito R10, all'interno dei seguenti componenti:

- **Student Front End:** è il punto di interazione per gli studenti, costituito da Spring Boot per il back-end e CodeMirror per l'editor di testo front-end. Ha lo scopo di consentire ad un giocatore di intraprendere una partita mediante una pagina web;
- **Game Engine:** gestisce la logica di gioco, sempre utilizzando Spring Boot.

In particolare, come descritto precedentemente, ci siamo soffermati sul contenuto dei task **T5** e **T6**. Osserviamo ora il diagramma dei package relativo a T5-T6, con le dovute precisazioni, e quello dei componenti, sottolineando le osservazioni fatte in fase di analisi della documentazione.

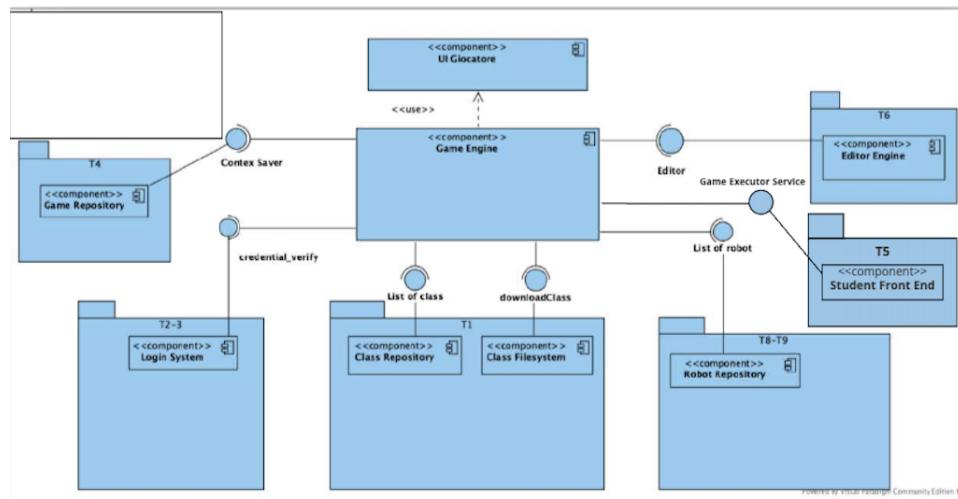


Figura 3.3: Vista dei componenti del GameEngine

Rispetto al precedente diagramma dei componenti è stato aggiunto il componente relativo a T5 **Student Front End** per specificare meglio il collegamento tra le componenti: come vedremo successivamente,

nelle precedenti documentazioni abbiamo ritrovato molta ambiguità rispetto al ruolo di T5 e T6 nel Game Engine, poiché inizialmente T6 si occupava della parte front-end, ruolo che è stato poi ceduto a T5.

Di conseguenza, analizziamo di cosa si occupano i task T5 e T6. Entrambi i task implementano il **Game Engine**, mentre il T5 di base **Student Front End**: entrambi i task seguono un pattern architettonico MVC

### 3.1.1 Task T5

Nella documentazione analizzata, il task T5 prevede l'implementazione del *Game Engine*, ovvero del motore che consente la gestione della partita e che comunica direttamente con il front end (ex T6) per lo svolgimento dei vari servizi relativi allo svolgimento del gioco: si occupa di sviluppare l'interfaccia e la logica di gestione dei dati per la creazione e l'avvio di una partita in un'applicazione. In pratica, questo task comprende sia il lato visuale che quello funzionale dell'app. Nel dettaglio, vengono create una serie di classi in Java per gestire la logica di base e il controllo del flusso dell'applicazione: queste classi si occupano di gestire il salvataggio dei dati nel database e sul file system. Inoltre, viene sviluppata una web-app utilizzando HTML, CSS e Javascript, con l'ausilio del framework Bootstrap per la parte di presentazione grafica: questa web-app consente agli utenti di autenticarsi e di avviare una partita. L'architettura segue quindi il pattern

**Model-View-Controller** (MVC), il che significa che le funzionalità dell'applicazione sono suddivise in tre parti principali: il Modello (Model) che gestisce i dati e la logica di business, la Vista (View) che si occupa della presentazione dei dati all'utente, e il Controllore (Controller) che gestisce l'interazione tra il Modello e la Vista.

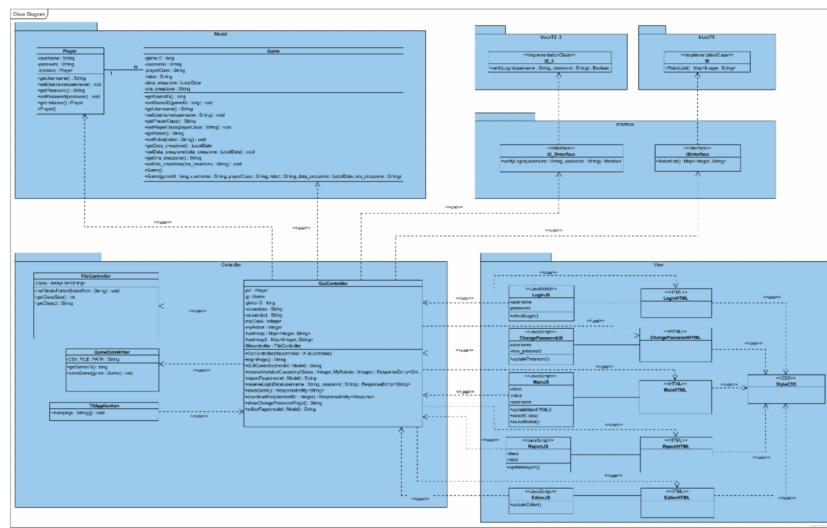


Figura 3.4: Diagramma delle classi di T5

Vediamo le varie parti cosa implementano:

- Il Modello (Model) si occupa della logica di base e delle entità principali coinvolte nel gioco. È implementato in Java e definisce tre entità chiave: lo Studente che gioca (Player), la Classe da testare (ClassUT) e la Partita stessa (Game). Il Modello permette l'accesso ai dati necessari e ne aggiorna le visualizzazioni.
- La Vista (View) gestisce l'aspetto visivo e interattivo del front-end della web-app. Questo è ciò che l'utente vede e con cui interagisce. È stata realizzata utilizzando il framework Bootstrap,

il quale semplifica la creazione di siti web moderni e adattabili utilizzando HTML, CSS e Javascript. La Vista consente agli utenti di registrarsi, effettuare il login, selezionare una classe e un robot da sfidare, e iniziare una nuova partita.

- Il Controllore (Controller) gestisce la logica di controllo, elaborando gli input dell’utente e aggiornando il Modello quando necessario. È composto da due classi principali in Java: GUIController, che utilizzando il framework Spring, gestisce le operazioni CRUD (Create, Read, Update, Delete) definendo diversi metodi per ogni operazione e route, e GameDataWriter, che definisce i metodi per interagire con Game Repository e Student Repository, due componenti che si occupano della memorizzazione dei dati relativi alle partite e agli studenti, rispettivamente.

### 3.1.2 Task T6

Inizialmente, come anche descritto nella documentazione analizzata, il task T6 prevedeva la realizzazione dell’interfaccia che implementasse il text editor in cui poter giocare la partita.

Questa descrizione tuttavia non è del tutto accurata: dobbiamo considerare in realtà attualmente il T6 come un’estensione della sezione **Controller** del pattern MVC di T5.

## CAPITOLO 3. STATO INIZIALE DEL PROGETTO

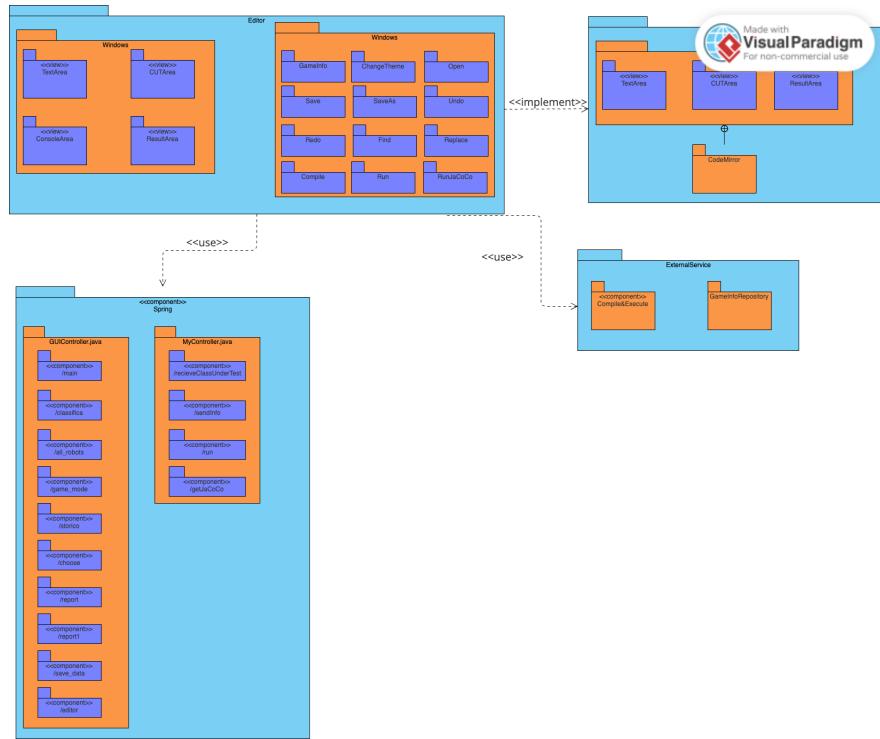


Figura 3.5: Diagramma dei package di T5 e T6

Osservato il nuovo diagramma dei package, infatti, notiamo che la sezione Controller è in realtà composta da due file Java: entrambi sono implementanti utilizzando il framework **Spring Boot** che mappa chiamate HTTP effettuate tramite interazioni con la parte front-end con dei metodi Java. Tuttavia, il loro ruolo è leggermente differente:

- **GuiController**, il controller presente in T5, mappa tutte le chiamate HTTP relative alle elaborazioni *pre-partita*, come la stampa dei robot prima della selezione, la visualizzazione dello storico, il salvataggio nel database appena la partita parte, la visualizzazione dell'editor ecc;
- **MyController**, il controller presente in T6, invece, mappa le

chiamate HTTP relative alle elaborazioni *durante la partita*, come il **RUN** che fa partire effettivamente il turno.

### 3.1.3 Interazione tra T5 e T6

Per la gestione della partita, e in particolare per la realizzazione della funzione relativa al completamento e al salvataggio dei risultati di gioco, si è osservato che mantenere i task T5 e T6 completamente disaccoppiati portava alcuni svantaggi in termini di comunicazione in quanto complicato e computazionalmente oneroso. Per questo motivo, la versione finale, realizzata dal task T11, versione G41, relativo all'integrazione di tutti i microservizi in un unico software, ha generato un sistema che unisse, almeno parzialmente, la parte backend e frontend del game engine in uno unico "macroservizio" più facilmente gestibile. In particolare, la funzionalità di interesse del nostro gruppo, relativa al bottone RUN (Play) che consente il calcolo del punteggio del giocatore e la visualizzazione a schermo dell'eventuale vittoria o sconfitta, si sviluppa come mostrato in figura dai diagrammi di comunicazione e di sequenza.

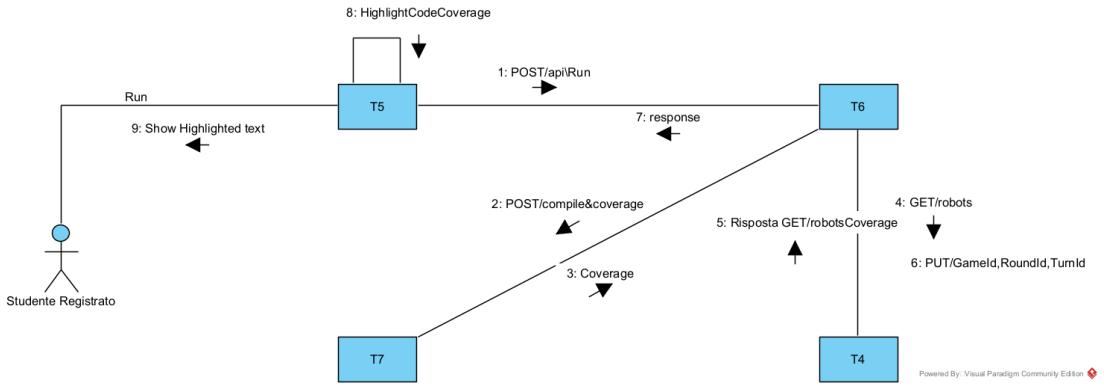


Figura 3.6: Communication diagram relativo al fine partita per il calcolo dei punteggi e dell’eventuale vittoria

Come si evince dal diagramma di comunicazione, il task T5 e T6 interagiscono attivamente per l’invio e ricezione delle informazioni della partita. Inizialmente si pensava che fosse solo il task T6, che prevedeva soltanto funzionalità di interfaccia grafica e lasciava tutto il resto agli altri componenti, a comunicare con gli altri elementi del sistema per la ricezione delle coverage dei robot relative alla classe considerata. In realtà, quello che accade è che entrambi i task si interfacciano con altri componenti del sistema, ma per scopi differenti: nel Communication Diagram si osserva che è T6 a fare effettivamente i salvataggi nel database di T4, ma lo fa attraverso una chiamata HTTP **PUT** che aggiorna la tupla relativa al turno corrente salvata precedentemente dal controller di T5 attraverso il metodo `save-data`.

Una mancanza fondamentale nella precedente documentazione erano diagrammi che permettessero di comprendere il flusso di esecuzione dei metodi studiati. Si è quindi deciso di crearne alcuni ex novo

relativamente al funzionamento dell'architettura precedente al nostro intervento, per sottolinearne le differenze dopo le nostre modifiche.

Poiché abbiamo affermato che T6 contiene un'estensione del Controller di T5 si è deciso di costruire il suo diagramma delle classi:

Infine, in un ultimo studio della documentazione del task T6, si è passati alla generazione di un diagramma delle classi che consentisse una migliore comprensione dello stato del componente.

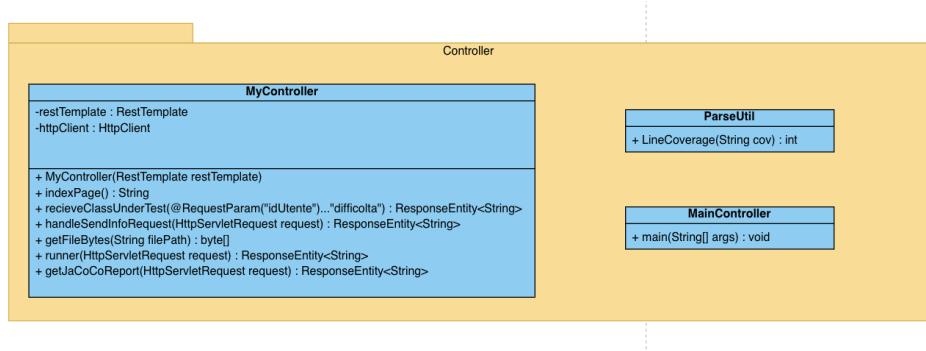


Figura 3.7: Diagramma delle classi di T6 precedente alle modifiche per la modalità "BossRush"

Inoltre, si è deciso di costruire anche un diagramma di sequenza che permettesse di comprendere meglio il flusso di esecuzione del metodo di running della partita. Come vedremo successivamente, nel capitolo 4 relativo alle modifiche, questa mancanza è stata causa di gravi rallentamenti nello sviluppo poiché non era per nulla chiaro quello che il software dovesse effettivamente fare; lasciamo quindi il diagramma di sequenza relativo al funzionamento del metodo anche per i colleghi a cui lasceremo in eredità il nostro operato.

## CAPITOLO 3. STATO INIZIALE DEL PROGETTO

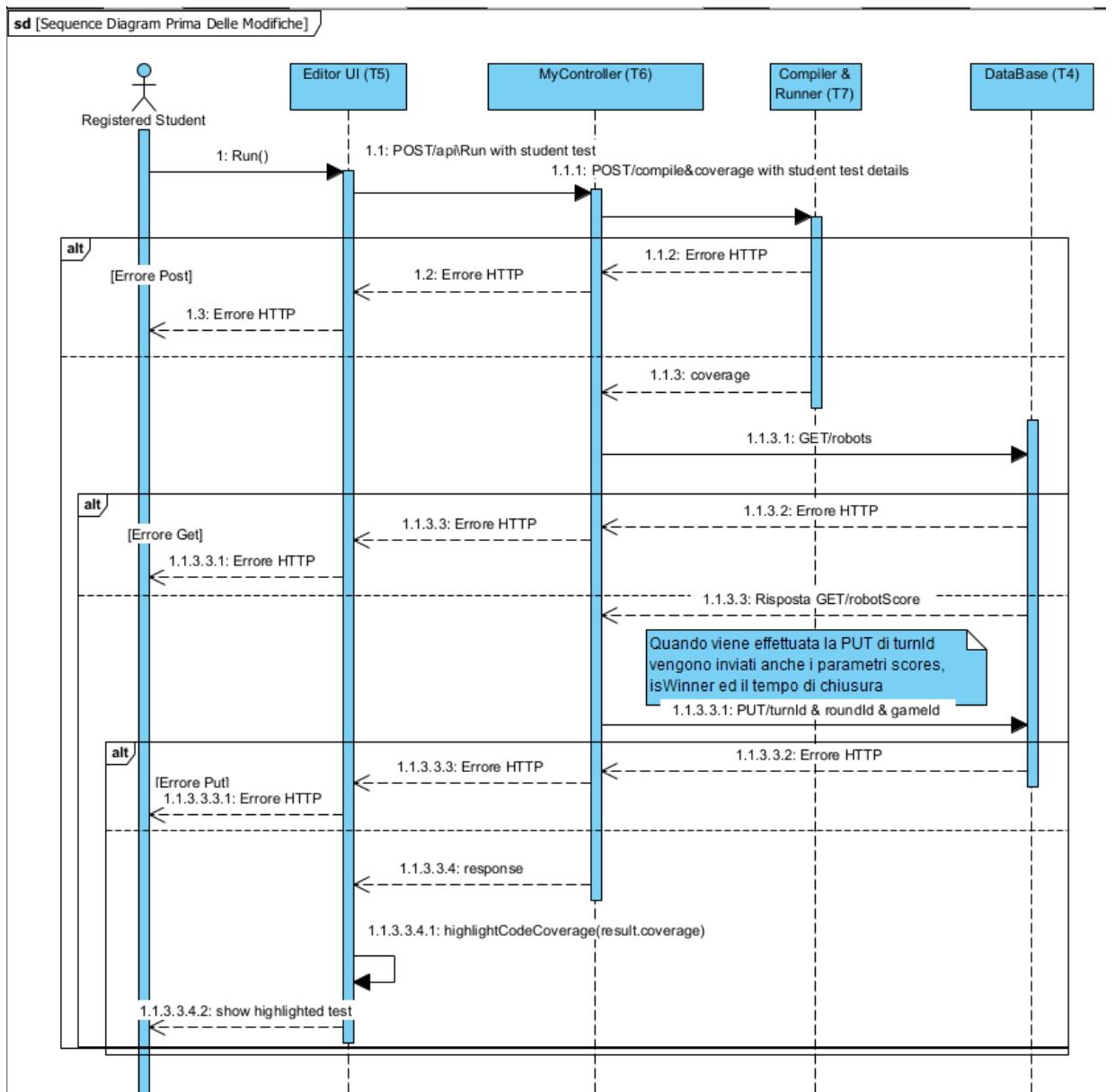


Figura 3.8: Diagramma di sequenza della vecchia versione

## Capitolo 4

# Modifiche al task 5 e al task 6

Fatta un'analisi approfondita dello stato attuale dell'architettura, in particolare dello stato di T5 e T6, ci siamo resi conto dell'alta coesione tra i due task e del loro funzionamento, essendo di fatto stati in passato un unico task il cui scopo era strettamente connesso. In questa sezione ci si presta quindi ad analizzare i problemi relativi allo stato iniziale da cui è partita la modifica e i cambiamenti effettuati.

## 4.1 Analisi preliminare e problematiche

### 4.1.1 Problematiche generali

Per comprendere al meglio il lavoro da noi svolto è opportuno chiarire lo stato in cui l'applicazione ci è stata lasciata in eredità i nostri colleghi, con le opportune considerazioni. I nostri primi passi per comprendere come soddisfare il requisito assegnatoci è stato quello di consultare approfonditamente le documentazioni forniteci dei task T5-T6, relative alla versione *G-41* lasciata come base su cui cominciare a lavorare; in questa fase, purtroppo, abbiamo riscontrato non pochi problemi. L'analisi della documentazione forniteci dai task T5 e T6 si è rivelata estremamente ardua, poiché non particolarmente chiare sul loro scopo: queste infatti facevano riferimento ad una versione del gioco in cui T5 e T6 erano estremamente accoppiati e si occupavano della parte front-end del componente *Game Engine*(per esempio, nella documentazione del task T6 si fa riferimento all'editor del *Game Engine*, anche se questo non è presente nella versione finale in T6, bensì in T5). Di conseguenza la documentazione fornita non è stata di grande aiuto relativamente al nostro requisito, poiché tratta il funzionamento front-end del Game Engine e non il back-end, il quale era di nostro interesse. Pertanto, la comprensione della documentazione è stata motivo importante di rallentamento del processo software, per due motivi:

- Molto lontane dal nostro requisito, poiché descriventi il funzionamento front-end del Game Engine soffermandosi poco su quello back-end;
- Quasi nessuna documentazione relativa al funzionamento effettivo del task, dettati dalla mancanza sia di appropriati diagrammi di sequenza sia di descrizione funzionale del codice.

L’analisi del funzionamento dei task, quindi, a causa di queste problematiche, è stata fatta esclusivamente sulla base della lettura del codice. I diagrammi di sequenza presentati in precedenza, infatti, sono stati costruiti ex novo dalla comprensione del nostro gruppo relativa al funzionamento, dopo numerose iterazioni e prove.

### 4.1.2 Problematiche relative alla compilazione

Un’altra importante causa di rallentamento nello sviluppo dell’applicazione è stata la ricompilazione: ci siamo ritrovati in fase iniziale, dopo aver modificato i file nelle directory in T5 e T6 , eliminati i container ed aver avviato l’installer (che richiede comunque una buona quantità di tempo per terminare) a non notare le modifiche effettuate in fase di esecuzione.

Nella documentazione consultata non c’era nessuna indicazione riguardo questa problematica; ci siamo quindi consultati con i nostri colleghi e abbiamo appreso che le modifiche non venivano salvate a causa del

file `pom.xml` che gestisce le dipendenze del progetto Maven, il quale richiedeva di essere ricompilato ad ogni modifica attraverso i comandi:

```
mvn clean install  
mvn clean package
```

Appresa questa nozione si è deciso di creare due programmi batch per velocizzare il processo, il cui funzionamento è descritto nel capitolo 8.

### 4.1.3 Problematiche legate al salvataggio

Prima di affrontare le modifiche effettuate al software occorre però soffermarsi sulla modalità operativa effettuata dal nostro team in merito ad un'importante parte del nostro requisito: quella relativa al salvataggio.

Come specificato nelle user stories, infatti, parte del requisito era quello di avere la possibilità in fase di consultazione del database di distinguere il tipo di partita; inoltre, ci è stato richiesto anche di specificare, in fase di salvataggio, quanti robot fossero stati battuti dal giocatore.

Prima di procedere con le modifiche il team si è consultato relativamente a come procedere: si è fatta un'approfondita analisi del database, e in particolare del contenuto del task *T4* sulle entità *Turn*, *Round* e *Game*. Poiché relativamente al salvataggio il team è partito dalla versione del team *A3*, per poi migrare su quella del team *A9* si farà riferimento a queste.

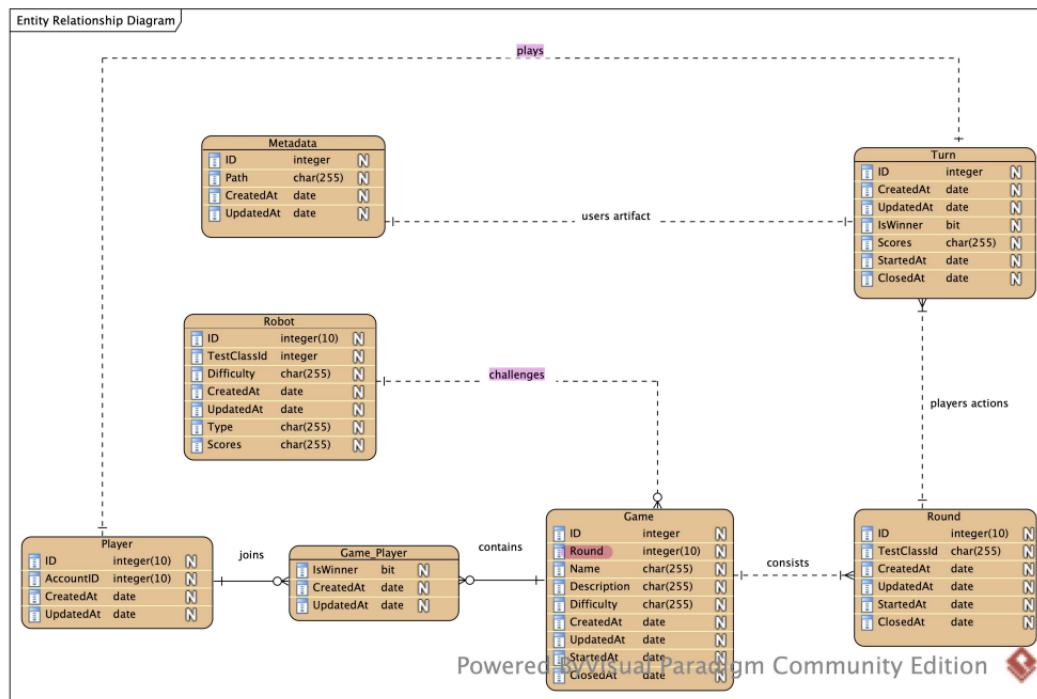


Figura 4.1: Diagramma ER del gruppo A3

Il team ha visto due possibili vie per soddisfare il requisito relativo al salvataggio:

- Rivoluzionare le entità presenti nel database, inserendo un parametro relativo alla modalità;
- Utilizzare le entità già presenti, effettuando una modifica in fase di salvataggio per distinguere le modalità.

Inizialmente si è scelto di operare seguendo il secondo approccio, poiché operare sul database del task T4 sarebbe stato troppo dispendioso in termini di lavoro e soprattutto avrebbe allontanato l'obiettivo sul nostro requisito.

Si è quindi scelto di operare sul parametro *scores* dell'entità *turn*, il quale, essendo una stringa, avrebbe salvato, oltre allo score effettivo ottenuto dal giocatore, il numero di robot battuti. Per esempio, nel caso un giocatore avesse ottenuto coverage *60%* e battuto 3 robot su 5, il valore del parametro score sarebbe stato *60(3/5)*, distinguendosi dalla modalità *Classica* in cui veniva salvato solo lo score.

Successivamente però, a causa di un problema nell'API fornитaci dal task A3 relativamente all'aggiornamento dell'entità *Turn* mediante una PUT (la PUT non funzionava correttamente, non permettendo l'aggiornamento nel database) si è deciso di migrare alla versione del gruppo *A9*, anche perché quest'ultima versione implementava la possibilità di scelta della modalità di gioco e quindi strettamente connessa al nostro task. Questa versione, oltre a risolvere il problema relativo alla PUT, proponeva un sostanziale cambiamento nella struttura delle entità, in particolare di nostro interesse sono stati l'aggiunta del parametro *Robot* nell'entità *Turn*. Questa aggiunta è stata vantaggiosa per il nostro requisito, poiché fornisce un'ulteriore distinzione con la modalità *Classica*, essendo che il valore relativo al parametro *Robot* nella modalità *bossRush* sarà sempre pari a "Tutti i robot", mentre quelli nella modalità *Classica* saranno pari a *Evosuite* o *Randoop*.

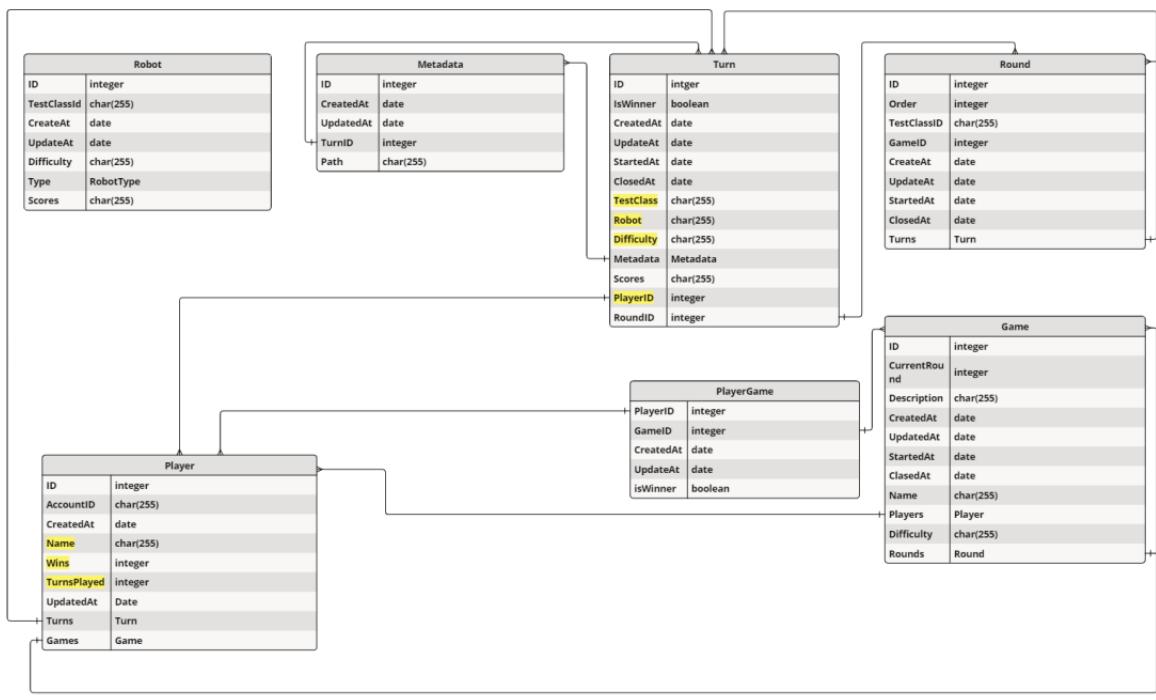


Figura 4.2: Diagramma ER del gruppo A9

## 4.2 Descrizione delle chiamate API

Prima di procedere con le modifiche, si è proceduto alla descrizione delle chiamate API utilizzate dal nostro team.

### 4.2.1 Salvataggio iniziale

- **Metodo HTTP:** POST
- **URL:** `http://localhost/api/save-data`
- **Descrizione:** viene effettuato un salvataggio parziale delle informazioni della partita, con ID del player, classe da testare e

tipo di robot nel database di T4 senza le informazioni relativa alla vittoria o sconfitta.

#### 4.2.2 Esecuzione RUN

- **Metodo HTTP:** POST
- **URL:** `http://localhost/api/run`
- **Descrizione:** passate le informazioni da lato front-end come nome della classe da testare, tipo di robot ecc.. viene avviata la funzione `runner` nel controller di T6.

#### 4.2.3 Recupero coverage giocatore

- **Metodo HTTP:** POST
- **URL:** `http://remoteccc-app-1:1234/compile-and-codecoverage`
- **Descrizione:** fornita classe di test e classe da testare, restituisce la coverage del giocatore e il valore della riuscita o meno della compilazione.

#### 4.2.4 Recupero coverage robot

- **Metodo HTTP:** GET

- **URL:** `http://t4-g18-app-1:3000/robots/[testClassId]/[type]/[difficulty]`
- **Descrizione:** Si ottiene la coverage del robot, relativamente alla classe, tipo e difficoltà fornita. C'è da sottolineare che, dati  $n$  robot Randoop e  $m$  robot Evosuite, questa chiamata verrà fatta  $n+m$  volte andando di volta in volta a cambiare il tipo e la difficoltà, fintanto che ci sono robot disponibili relativamente alla classe in ingresso.

#### 4.2.5 Aggiornamento turno, round e game

- **Metodo HTTP:** `PUT`
- **URL:** `http://t4-g18-app-1:3000/turns/[turnId]`
- **Descrizione:** Si aggiorna la tupla nel database relativa al turno specificato dall'ID con i parametri relativi allo score, alla vittoria, al nome della classe, al tipo di robot, alla difficoltà e all'orario di chiusura del turno. Si fa lo stesso con il round e il game.

### 4.3 Modifiche a T5

Il punto di partenza delle modifiche che abbiamo effettuato è stata la versione aggiornata implementata dal gruppo A9, il quale ha aggiunto, tra le varie modifiche, la schermata che prevede la scelta della moda-

lità sfida contro tutti i robot. Non ci sono state variazioni relative alla struttura del task per quanto riguarda la struttura delle classi, si è voluto però costruire un diagramma di attività relativamente alla funzione run e al suo funzionamento. Qui di seguito è presentato il diagramma di attività che illustra le operazioni eseguite nella parte front-end durante un turno di gioco:

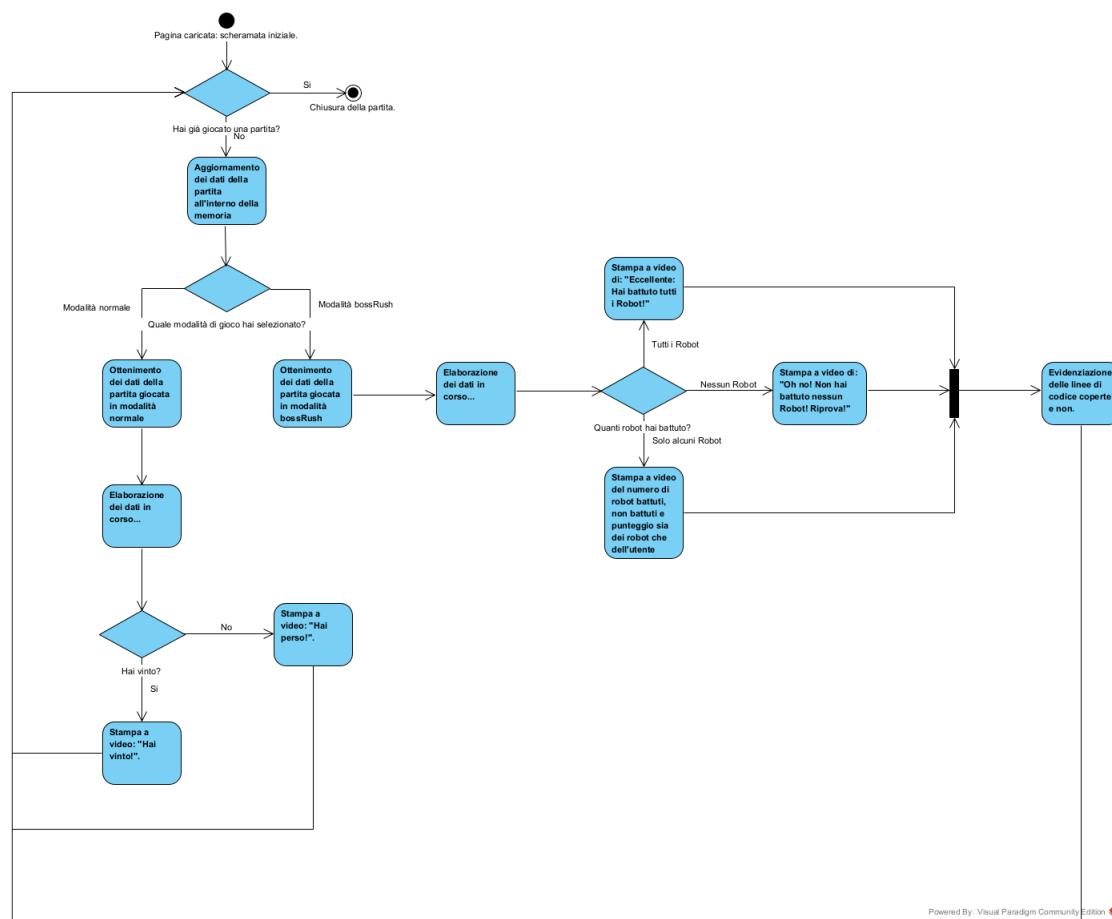


Figura 4.3: Diagramma di attività della versione aggiornata

E' possibile passare al turno successivo solo ed esclusivamente in caso di sconfitta parziale o totale.

In particolare, il punto di aggancio è stato il file **editor.html** presente all'interno del package *templates*, il quale a sua volta si trova nel package *resources* di T5, rimasto invariato rispetto alla versione *G-41*.

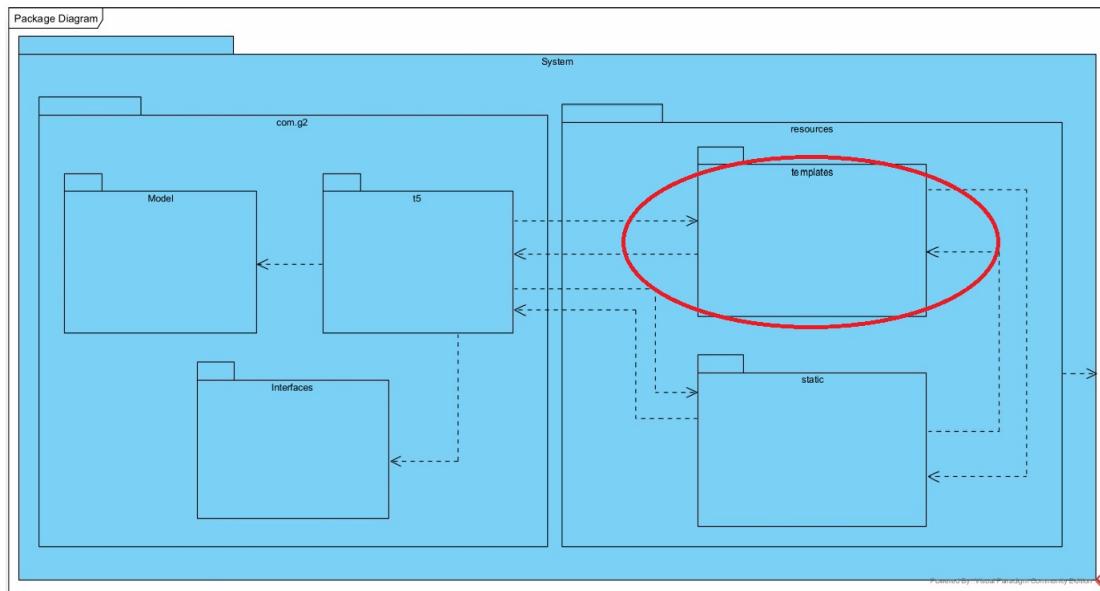


Figura 4.4: posizione di editor.html all'interno di T5.

Il documento **editor.html** costituisce l'interfaccia utente lato frontend in cui avviene la compilazione del codice relativo alla classe sottoposta al testing e l'avvio della partita. Questo processo restituisce i risultati pertinenti, inclusi il punteggio dell'utente (percentuale di linee di codice coperte) e l'esito della partita.

### 4.3.1 Analisi del file editor.html

Inizieremo analizzando il codice presente nel file **editor.html**, prima delle modifiche apportate dal nostro gruppo, concentrandoci sull'ap-

profondimento di una specifica porzione. Partiamo dal seguente spezzone, che rappresenta ciò che avviene lato front-end durante un turno di gioco:

```
775     formData.append("testingClassName", "Test" + localStorage.getItem("classe") + ".java");
776     formData.append("testingClassCode", editor.getValue());
777     formData.append("underTestClassName", localStorage.getItem("classe")+".java");
778     formData.append("underTestClassCode", sidebarEditor.getValue());
```

Figura 4.5: porzione di codice precedente alle modifiche

```
781     formData.append("turnId",localStorage.getItem("turnId"));
782     formData.append("roundId",localStorage.getItem("roundId"));
783     formData.append("gameId",localStorage.getItem("gameId"));
784     formData.append("testClassId", localStorage.getItem("classe"));
785     formData.append("difficulty", localStorage.getItem("difficulty"));
786     formData.append("type", localStorage.getItem("robot"));
```

Figura 4.6: porzione di codice precedente alle modifiche

Il codice utilizza l’oggetto *FormData* in JavaScript per generare un insieme di dati destinati a una richiesta HTTP. Ciascuna linea del codice aggiunge un campo all’oggetto *FormData*, includendo il nome della classe da testare e della classe di test, il codice della classe da testare e la classe di test, l’identificativo del turno, del round, della partita, della classe, la difficoltà e il tipo di robot, a partire dai dati presenti nel *localStorage* (oggetto in JavaScript che consente di memorizzare dati in modo persistente nel browser web). Dopo le modifiche apportate dal gruppo A9, il valore del campo *robot* può essere **Tutti i Robot**, oltre a *Randoop* ed *Evosuite*. Questa variazione ci ha per-

messo di distinguere tra la modalità classica e la modalità contro tutti i robot.

```
787     $.ajax({
788         url: "http://localhost/api/run", // con questa verso il task 6, si salva e conclude la partita e si decreta il vincitore
789         type: "POST",
790         data: formData,
791         processData: false,
792         contentType: false,
793         dataType: "json",
794         success: function(response) {
```

Figura 4.7: porzione di codice precedente alle modifiche

Il codice fornito presenta una chiamata AJAX in JavaScript mediante la libreria jQuery. La richiesta di tipo POST è indirizzata a "*http://localhost/api/run*" e include dati definiti nella variabile *formData*. I parametri *processData* e *contentType* sono impostati su ‘false’ per evitare che jQuery elabori automaticamente i dati e imponga automaticamente l’intestazione *Content-Type*. La risposta attesa è in formato JSON (*dataType: "json"*), e in caso di successo, viene eseguita una funzione di callback specificata nella sezione *success* per manipolare la risposta ricevuta. Importante notare che questa chiamata AJAX si collega all’endpoint *run* nel contesto "*/api*". È presente, infatti, una funzione denominata *runner*, scritta in Java, all’interno di T6, alla quale viene mappata la richiesta POST.

---

```

795         console.log(response);
796
797         risp = response;
798
799         perc_robot = response.robotScore.toString();
800
801         consoleArea.setValue(response.outCompile);
802         highlightCodeCoverage($.parseXML(response.coverage));

```

---

Figura 4.8: porzione di codice precedente alle modifiche

La funzione `console.log(response)` viene utilizzata per stampare l'intero oggetto `response` nella console del browser per scopi di debug. Successivamente il punteggio del robot selezionato, `robotScore`, presente all'interno di `response` viene convertito in una stringa e assegnata a `perc_robot`. Il parametro `outCompile` di `response` viene utilizzata per impostare il contenuto di un'area dell'interfaccia utente denominata `consoleArea`. Infine, la funzione `highlightCodeCoverage` viene chiamata con l'analisi XML dell'attributo `coverage` di `response`. Questa funzione gestisce l'evidenziazione visiva della copertura del codice nell'interfaccia utente, fornendo informazioni sulla parte del codice eseguita o coperta.

```

807         alert(response.win == true ? "Hai vinto!" : "Hai perso!");
808
809         if(response.win == true){
810             turno++; // incremento il numero di turno giocati fino ad ora
811
812             localStorage.setItem("gameId",null);
813

```

Figura 4.9: porzione di codice precedente alle modifiche

Il codice mostra un messaggio di avviso all'utente indicando se ha vinto o perso. Se l'utente ha vinto, incrementa il numero di turni

giocati e reimposta il valore associato a *gameId* nello storage locale a null.

```
815         else{
816
817
818
819
820
821     $.ajax({
822         url:'http://localhost/api/save-data',
823         data: {
824             playerId: parseJwt(getCookie("jwt")).userId,
825             classe: localStorage.getItem("classe"),
826             robot: localStorage.getItem("robot"),
827             difficulty: localStorage.getItem("difficulty")
828         },
829         type:'POST',
830         success: function (response) {
831             // Gestisci la risposta del server qui
832             localStorage.setItem("gameId", response.game_id);
833             localStorage.setItem("turnId", response.turn_id);
834             localStorage.setItem("roundId", response.round_id);
835             console.log(localStorage.getItem("turnId"));
836             turno++; // incremento il numero di turno giocati fino ad ora
837         }
838     });
839 }
```

Figura 4.10: porzione di codice precedente alle modifiche

In caso di sconfitta, nel ramo else, viene effettuata una chiamata AJAX utilizzando jQuery verso l'endpoint "*http://localhost/api/save-data*". La richiesta POST include dati come l'ID del giocatore (*playerId*), della classe (*classe*), il robot (*robot*), e la difficoltà (*difficulty*) prelevati dal *localStorage*. Questi dati vengono inviati al server per essere salvati. Nel blocco *success* della chiamata AJAX, la risposta del server viene gestita. I valori restituiti per *game\_id*, *turn\_id*, e *round\_id* vengono utilizzati per aggiornare le corrispondenti chiavi nel *localStorage*. Inoltre, la variabile turno viene incrementata.

```

838         },
839         dataType: "json",
840         error: function (error) {
841             console.error('Errore nell invio dei dati');
842             alert("Dati non inviati con successo");
843             // Gestisci l'errore qui
844         }
845     });
846
847 }

```

Figura 4.11: porzione di codice precedente alle modifiche

In caso di errore durante la chiamata AJAX interna, il blocco *error* gestisce la situazione, fornendo un messaggio di errore nella console e un avviso all’utente tramite un alert.

```

849         error: function() {
850             // Gestisci l'errore, ad esempio mostra un messaggio di errore
851             console.log("Errore durante l'invio della partita.");
852         }
853     });

```

Figura 4.12: porzione di codice precedente alle modifiche

In caso di errore durante la chiamata AJAX esterna, il blocco *error* gestisce la situazione, fornendo un messaggio di errore nella console e un avviso all’utente tramite un alert. Il blocco di codice che segue riflette le modifiche apportate al nostro programma. Analizziamole più dettagliatamente.

### 4.3.2 Modifiche al file editor.html

#### Modifiche precedenti alla versione aggiornata

Prima di avere a disposizione la versione aggiornata del lato front-end è stato modificato un ulteriore file, contenuto nel medesimo package, denominato *main.js*. **NOTA: A seguito dell'integrazione con la versione aggiornata ai file main.js ed editor.html non sono state apportate le seguenti modifiche.**

```
1 //variabili per la selezione della classe e del robot
2 var classe = null
3 var robot = null
4 var difficulty = null
5 var gameMode = null;
```

Figura 4.13: porzione di codice non soggetta alle modifiche

In primo luogo è stata introdotta un'ulteriore variabile, denominata *gameMode*, indicante la modalità di gioco scelta dall'utente (modalità classica o contro tutti i robot).

```
68 function Handlebuttonrobot (id, button, rob, size) {
69   //modificato
70   $(document).ready(function () {
71     gameMode = "bossRush";
72     robot = "Tutti i robot"
```

Figura 4.14: porzione di codice non soggetto alle modifiche

Ai fini dello sviluppo e della simulazione della logica nel back-end, sono state inizializzate le variabili *robot* e *gameMode* a *bossRush* (nome assegnato alla modalità di sfida contro tutti i robot) e *Tutti i robot*, all'interno della funzione *HandleButtonRobot*.

```
99  function redirectToPagereport () {
100    console.log(classe)
101    console.log(robot)
102    console.log(difficulty)
103
104    if (classe && robot && difficulty) [
105      localStorage.setItem('classe', classe)
106      localStorage.setItem('robot', "Tutti i robot")
107      localStorage.setItem('difficulty', difficulty)
108      localStorage.setItem('gameMode', gameMode)
```

Figura 4.15: porzione di codice non soggetta alle modifiche

Infine è stato settato il valore di *gameMode* all'interno del *localStorage*.

Nel file *editor.html*, è stata effettuata una distinzione basata sulla variabile *gameMode* per gestire la restituzione dei risultati.

```
787  $.ajax({
788    url: "http://localhost/api/run",
789    type: "POST",
790    data: formData,
791    processData: false,
792    contentType: false,
793    dataType: "json",
794    success: function(response) {
795      console.log(response);
796
797      if (localStorage.getItem('gameMode') == "bossRush") {
```

Figura 4.16: porzione di codice non soggetto alle modifiche

A partire da questo punto, le modifiche apportate sono le stesse presenti nella versione aggiornata. Non è più necessario inizializzare la variabile *robot* a *Tutti i Robot* per testare la logica di back-end, poiché l'inizializzazione di questa variabile è ora gestita correttamente direttamente dalla schermata. Inoltre, la variabile *gameMode* è stata eliminata da questa parte di codice.

## Modifiche successive alla versione aggiornata

Ora esaminiamo le modifiche effettive apportate a *editor.html*, l'unico file modificato per quanto riguarda il Task 5:

```

806 |     if (localStorage.getItem("robot") == "Tutti i Robot") {
807 |         if(response.win == "true") {
808 |             var numberofBeaten = Number(response.numberofBeaten)
809 |             var messaggio = "Eccellente! Hai battuto tutti i Robot! Hai totalizzato (" + response.score + "% LOC coverage) e hai vinto contro ";
810 |
811 |             for(var i = 1; i <= numberofBeaten; i++) {
812 |                 var chiaveBeaten = "beaten" + i;
813 |                 var datiRobotBattuto = response[chiaveBeaten];
814 |                 var dati = datiRobotBattuto.split(",");
815 |                 var index = dati[0];
816 |                 var coverage = dati[1];
817 |                 messaggio += "Robot " + index + "(" + coverage + "%)";
818 |                 if(i != numberofBeaten) messaggio += ", ";
819 |
820 |
821 |             messaggio += ".";
822 |
823 |             alert(messaggio);
824 }

```

Figura 4.17: porzione di codice successivo alle modifiche

```

825 |         else if(response.numberofBeaten == "0") {
826 |             var numberofUnbeaten = Number(response.numberofUnbeaten);
827 |             var messaggio = "Oh no! Non hai battuto nessun Robot! Hai totalizzato (" + response.score + "% LOC coverage) e devi battere ";
828 |
829 |             for(var i = 1; i <= numberofUnbeaten; i++) {
830 |                 var chiaveUnbeaten = "unbeaten" + i;
831 |                 var datiRobotNonBattuto = response[chiaveUnbeaten];
832 |                 var dati = datiRobotNonBattuto.split(",");
833 |                 var index = dati[0];
834 |                 var coverage = dati[1];
835 |                 messaggio += "Robot " + index + "(" + coverage + "%)";
836 |                 if(i != numberofUnbeaten) messaggio += ", ";
837 |
838 |             messaggio += ".";
839 |
840 |             alert(messaggio);
841 }

```

Figura 4.18: porzione di codice successivo alle modifiche

Innanzitutto, all'interno della funzione di callback *success*, situata nella prima chiamata ajax, viene effettuato un controllo sul valore memorizzato nel *localStorage* relativo alla tipologia di robot. Se il robot selezionato è **Tutti i Robot**, allora si procederà a determinare se il giocatore ha vinto o meno. Se il parametro *win*, contenuto nella risposta generata dalla funzione *runner*, assume il valore 'true', significa che il giocatore ha battuto tutti i robot e viene notificato attraverso un

alert. In caso contrario, se il parametro *numberOfBeaten*, che rappresenta il numero di robot sconfitti, è pari a 0, l'utente non ha sconfitto nessun robot e viene notificato tramite un alert.

```
811
812
813
814
815
816
817
818
819
820
821
822
823
824
```

```
else {
    var numberOfBeaten = Number(response.numberOfBeaten);
    var numberOfUnbeaten = Number(response.numberOfUnbeaten);
    var messaggio = "Bravo! Hai totalizzato (" + response.score + "% LOC coverage) e hai vinto contro ";

    for(var i = 1; i <= numberOfBeaten; i++) {
        var chiaveBeaten = "beaten" + i;
        var datiRobotBattuto = response[chiaveBeaten];
        var dati = datiRobotBattuto.split("&");
        var index = dati[0];
        var coverage = dati[1];
        messaggio += "Robot " + index + "(" + coverage + "%)";
        if(i != numberOfBeaten) messaggio += ", ";
    }
}
```

Figura 4.19: porzione di codice successivo alle modifiche

Se alcuni dei robot sono stati battuti, entriamo nell'else finale. Vengono dichiarate tre variabili: *numberOfBeaten*, che salverà il valore numerico dei robot battuti (in quanto precedentemente era una stringa), *numberOfUnbeaten*, che farà lo stesso per i robot che non sono stati battuti, e infine *messaggio*, che ci servirà a costruire il messaggio da mostrare all'utente, indicando il suo punteggio, i robot battuti, i robot non battuti e il punteggio dei robot. Nel primo ciclo for, vengono recuperati i valori dei robot battuti mediante un accesso a *response[chiaveBeaten]*. Gli attributi dell'oggetto JSON ‘response’ indicanti i dati dei robot battuti sono, infatti, memorizzati con le chiavi *beaten1*, *beaten2*, etc. Il dato recuperato viene salvato in *datiRobotBattuto*, e viene effettuato uno split per memorizzare l'indice del robot in *index* e il suo punteggio in *coverage*, per poi aggiungere il tutto al messaggio.

```
826     messaggio += ". Ti restano da battere ";
827
828     for(var i = 1; i <= numberofUnbeaten; i++) {
829         var chiaveUnbeaten = "unbeaten" + i;
830         var datiRobotNonBattuto = response[chiaveUnbeaten];
831         var dati = datiRobotNonBattuto.split("&");
832         var index = dati[0];
833         var coverage = dati[1];
834         messaggio += "Robot " + index + "(" + coverage + "%)";
835         if(i != numberofBeaten) messaggio += ", ";
836     }
837     messaggio += ".";
838     alert(messaggio);
839 }
```

Figura 4.20: porzione di codice successivo alle modifiche

Il messaggio viene poi completato effettuando le stesse operazioni per i robot che non sono stati battuti. Viene fatto un alert finale per notificare al giocatore i suoi risultati. Nel caso in cui non sia stata scelta la modalità **Tutti i Robot** si procede alla modalità classica, di cui il codice resta invariato.

## 4.4 Modifiche a T6

### Refactoring e nuove implementazioni

Si è quindi visto nei capitoli precedenti che T5 e T6 sono estremamente collegati tra loro. In particolare, osservando il precedente diagramma dei package e il diagramma delle classi di T5, questo segue un pattern **MVC**. Dobbiamo infatti considerare T6, di fatto, come un'estensione della sezione **Controller** del pattern, dove vengono mappate le chiamate HTTP sull'editor del Game Engine *durante* la partita attraverso *Spring*, mentre T5 si occupa di mappare quelle *pre-partita*.

Abbiamo lavorato quindi sulla classe dei T6 `MyController.java`, in

particolare sulla funzione che mappa la chiamata HTTP \run, ovvero `runner`. Questa effettua elaborazioni sulla base dei dati inviati da T5. Prima di descrivere i cambiamenti effettuati nel codice, bisogna aprire una parentesi sullo stato della funzione: tutte le elaborazioni mappate sulla chiamata `\run` venivano effettuate nello stesso metodo. Queste comprendevano:

- Chiamata a T7 per l'ottenimento della coverage del giocatore;
- Ottenimento della coverage dei robot da T4;
- Elaborazione sulla vittoria o meno del giocatore sulla base dei dati ottenuti;
- Costruzione della risposta verso T5;
- Salvataggio su T4 del turno, round e gioco.

Inizialmente abbiamo operato aggiungendo le elaborazioni relative alla modalità *bossRush* in `runner`, tuttavia ci siamo presto resi conto che il codice della funzione sarebbe diventato eccessivamente lungo e incomprensibile. Inoltre, essendo che, come vedremo dopo, le chiamate per l'ottenimento delle coverage dei robot *Evosuite* e *Randoop* sono sostanzialmente uguali, si è deciso di operare facendo un refactoring del codice aggiungendo una nuova classe, `RunnerHelper.java`, dividendo tutta la chiamata

*run* in sotto- funzioni secondo il loro scopo: in questo modo si è rispettato uno dei principi fondamentali dell'ingegneria del software, ovvero avere metodi con alta coesione e basso accoppiamento.

## Diagramma delle classi

Possiamo vedere le conseguenze del refactoring nel nuovo diagramma delle classi:

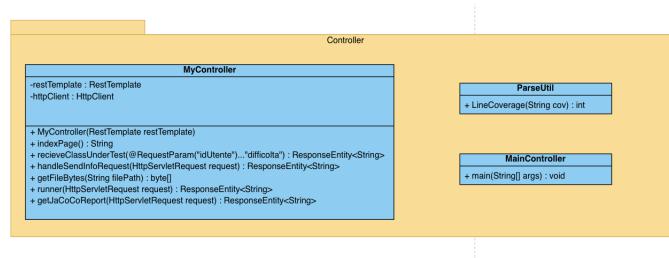


Figura 4.21: Precedente diagramma delle classi di MyController

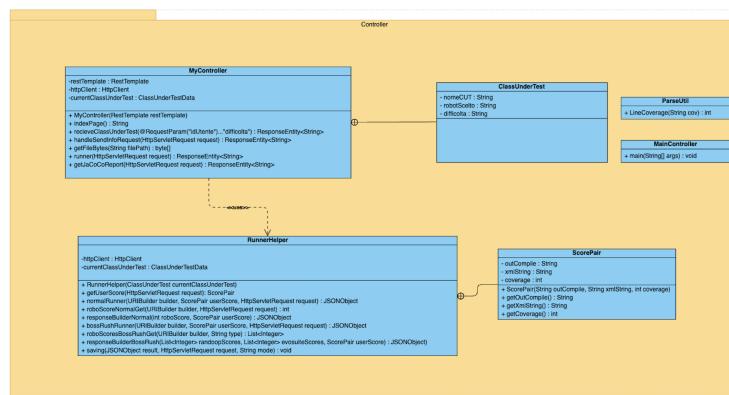


Figura 4.22: Nuovo diagramma delle classi di MyController

La modifica fondamentale è relativa all'aggiunta della classe `RunnerHelper`.java, la quale viene istanziata, come vedremo dopo, nel precedente metodo `runner`; vediamo le sue caratteristiche:

- Ha una sottoclasse `ScorePair` per la retrieve della coverage del giocatore, in modo da poter ottenere sia la percentuale di coverage sia informazioni sulla riuscita compilazione in un'unica chiamata HTTP;
- Il metodo `getUserScore`, presi in ingresso da T5 i parametri relativi alla classe da testare e quella di testing, attraverso una chiamata HTTP a T7 ottiene la percentuale di coverage del giocatore e il valore dell'*outCompile*. Questo metodo è comune ad entrambe le modalità, e viene utilizzato in `runner`;
- `normalRunner` e `bossRushRunner` sono i metodi principali chiamati in `runner`. Macroscopicamente contengono tutte le elaborazioni back-end, e restituiscono l'oggetto JSON da rimandare a T5;
- `roboScoreNormalGet` e `roboScoresBossRushGet` ottengono attraverso delle chiamate HTTP *GET* i valori di coverage del robot relativamente alla classe da testare; la differenza è che, mentre per il primo metodo otteniamo un unico valore intero, utilizzando il secondo otteniamo una lista di interi. In seguito, si approfondirà il funzionamento solo della seconda, poiché la prima è rimasta sostanzialmente invariata rispetto alla versione da cui siamo partiti;
- `responseBuilderNormal` e `responseBuilderBossRush` costrui-

scono l'oggetto JSON in risposta a T5. Il loro funzionamento è leggermente diverso, ed è legato alla filosofia scelta per determinare la vittoria del giocatore relativamente alla modalità *bossRush*;

- Infine, il metodo `saving` effettua i salvataggi, o meglio, aggiorna nel database di T4 le tuple relative al turno, round e game corrente, specificando vittoria, score ottenuto e orario di terminazione della partita.

### Diagramma di sequenza

Sulla base delle modifiche relative alla struttura delle classi, si è deciso di creare un nuovo diagramma di sequenza in cui sono preciseate tutte le chiamate tra le varie classi, così che il funzionamento del codice sia comprensibile ai futuri sviluppatori. Il diagramma di sequenza, infatti, rappresenta un elemento essenziale nel processo di sviluppo software, offrendo una visualizzazione chiara e cronologica delle interazioni dinamiche tra gli oggetti all'interno del sistema. Questo strumento riveste un ruolo cruciale nel delineare il flusso temporale delle operazioni e delle comunicazioni tra gli elementi del software, migliorando la comprensione delle sequenze di azioni e degli eventi che si verificano durante l'esecuzione del sistema.

## CAPITOLO 4. MODIFICHE AL TASK 5 E AL TASK 6

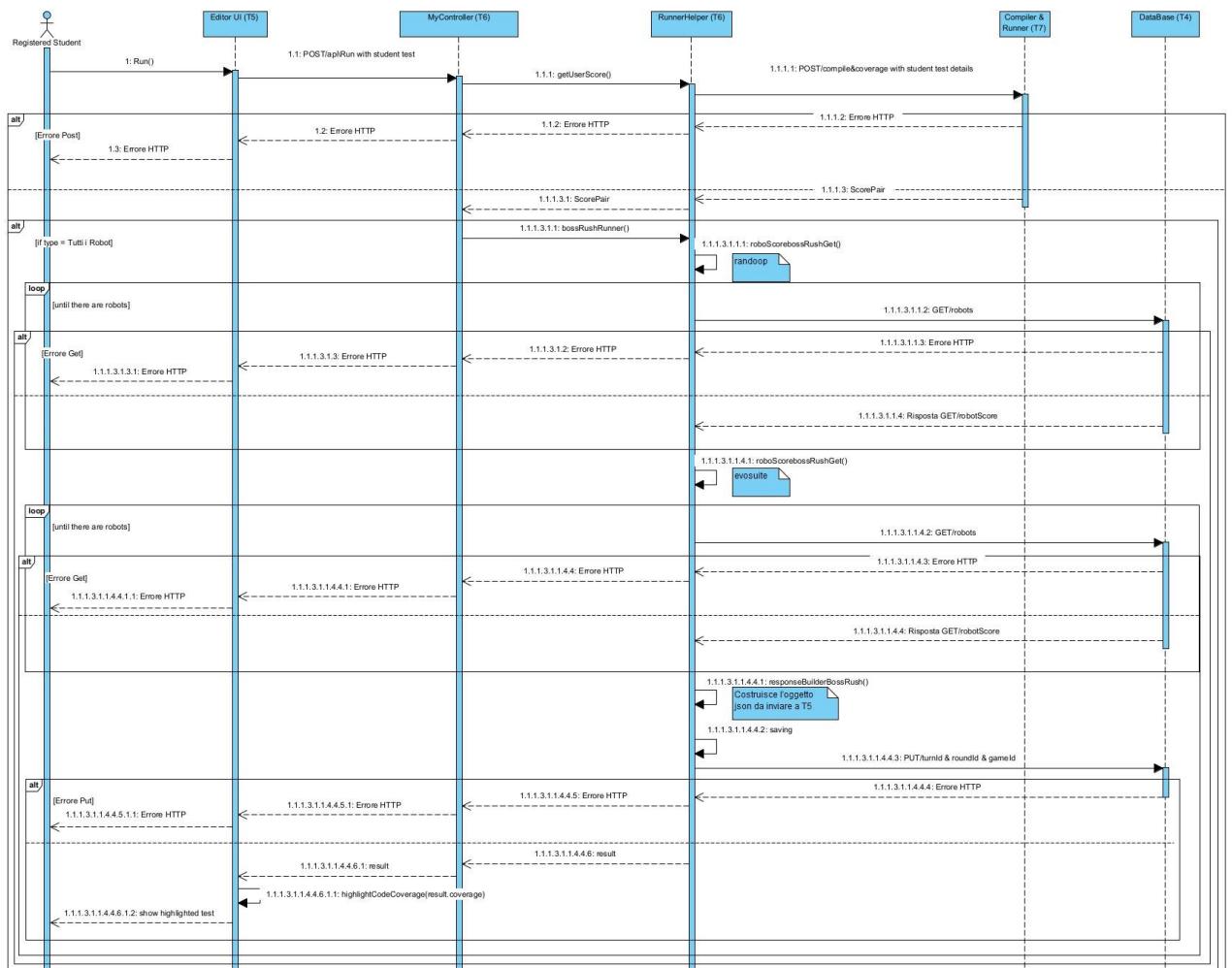


Figura 4.23: Diagramma di sequenza aggiornato, prima parte

## CAPITOLO 4. MODIFICHE AL TASK 5 E AL TASK 6

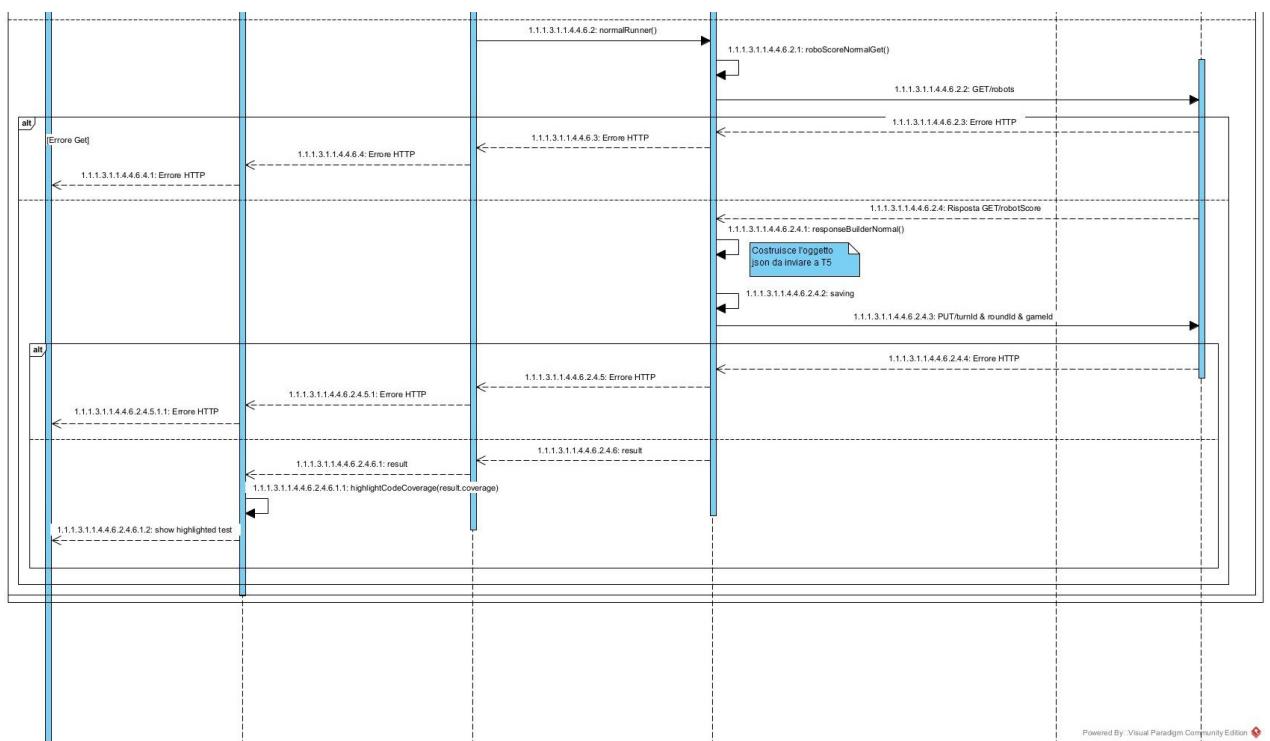


Figura 4.24: Diagramma di sequenza aggiornato, seconda parte

### Diagramma di attività

Prima di analizzare il codice del task T6, per comprendere meglio ad alto livello quello che il task T6 elabora si è deciso di costruire un diagramma di attività che chiarisca quello che il codice fa.

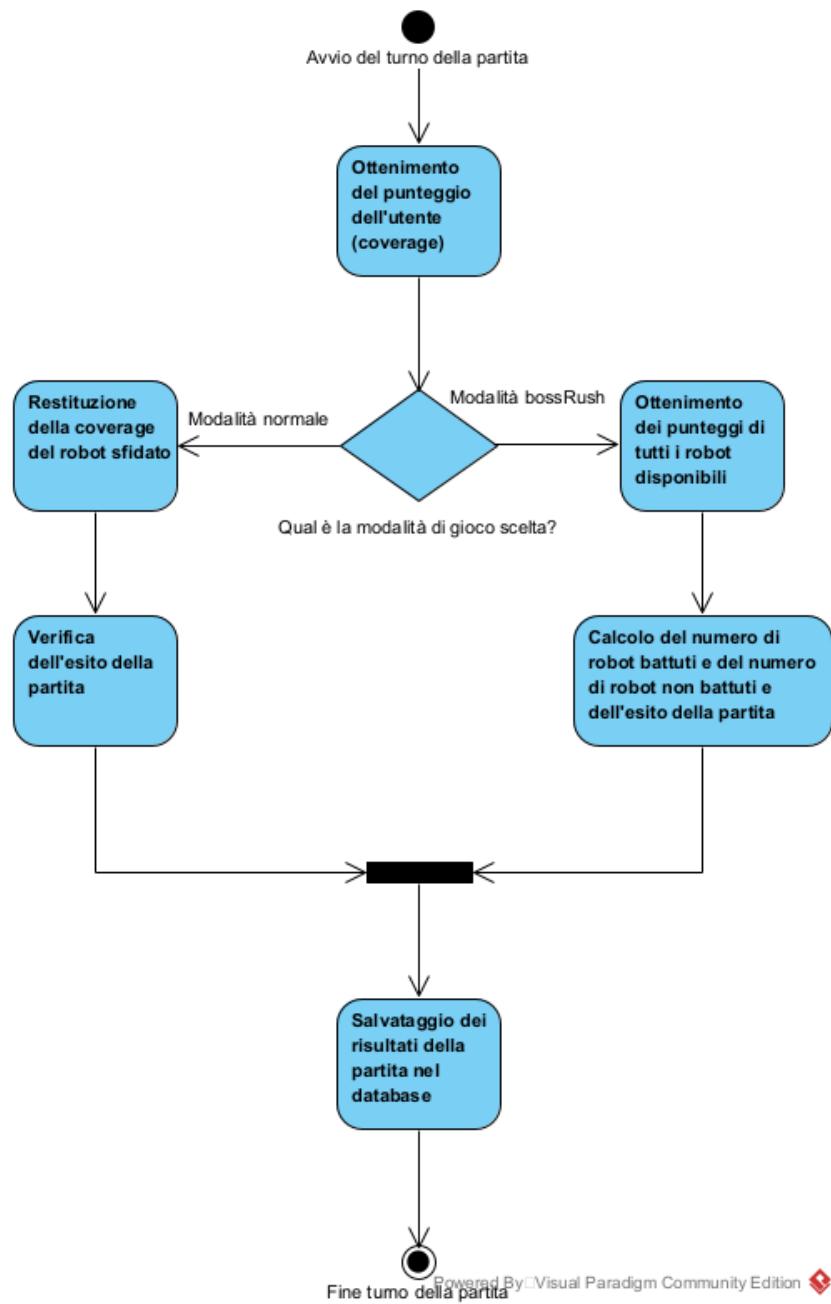


Figura 4.25: Diagramma di attività di T6

**Codice: MyController**

Vediamo la nuova versione di `MyController.java`

## CAPITOLO 4. MODIFICHE AL TASK 5 E AL TASK 6

---

```
@PostMapping("/run")
public ResponseEntity<String> runner(HttpServletRequest request) {
    try {
        RunnerHelper runnerHelper = new RunnerHelper(currentClassUnderTestData);
        // Esegui la richiesta HTTP al servizio di destinazione
        // RISULTATI UTENTE VERSO TASK 7
        ScorePair userScore = runnerHelper.getUserScore(request);

        UriBuilder builder = new UriBuilder(string:"http://t4-g18-app-1:3000/robots");
        builder.setParameter(param:"testClassId", request.getParameter(name:"testClassId"));

        JSONObject result = new JSONObject();

        // RISULTATI ROBOT VERSO TASK4
        if(request.getParameter(name:"type").equals(anObject:"Tutti i Robot")) {
            result = runnerHelper.bossRushRunner(builder, userScore, request);
        }
        else {
            result = runnerHelper.normalRunner(builder, userScore, request);
        }

        HttpHeaders headers = new HttpHeaders();
        headers.setContentType(MediaType.APPLICATION_JSON);

        return new ResponseEntity<>(result.toString(), headers, HttpStatus.OK);
    } catch (Exception e) {
        // Gestisci eventuali errori e restituisci un messaggio di errore al client
        System.err.println(e);
        return new ResponseEntity<>(HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

Figura 4.26: Codice finale del metodo *runner* di MyController

Le modifiche apportate al codice sono strettamente legate alla filosofia di refactoring descritta precedentemente. Questi accorgimenti permettono una leggibilità maggiore, e in generale permettono di avere più metodi coesi rispetto al loro scopo:

1. Viene istanziato un oggetto `runnerHelper`, usato per le elaborazioni. Questo ha come parametro passato nel costruttore il nome della classe sotto test;

2. L'ottenimento della coverage del giocatore viene effettuata attraverso il metodo `getUserScore` di `RunnerHelper`;
3. Viene poi creato un URI sull'entità `robots` e l'ID della classe da testare;
4. Il costrutto di selezione viene fatto relativamente al tipo di robot, ed a seconda di questo si avvia uno dei due metodi `runner`;
5. Infine, viene mandato il risultato dell'elaborazione a `T5`.

Di fatto, questo metodo è diventato solo un'interfaccia che ottiene le elaborazioni da `RunnerHelper`; quest'ultimo costituisce quindi un'estensione del Controller del pattern *MVC*.

Vediamo le differenze con la precedente versione, ovviamente osservando solo una parte poiché molto lungo:

## CAPITOLO 4. MODIFICHE AL TASK 5 E AL TASK 6

---

```
@PostMapping("/run") // NON ESISTE NESSUN INTERFACCIA VERSO I COMPILATORI DEI ROBOT EVOSUITE E
    // RANDOOP
public ResponseEntity<String> runner(HttpServletRequest request) {
    try {
        // Esegui la richiesta HTTP al servizio di destinazione
        // RISULTATI UTENTE VERSO TASK 7
        HttpPost httpPost = new HttpPost("http://remotecccc-app-1:1234/compile-and-codecoverage");

        JSONObject obj = new JSONObject();
        obj.put("testingClassName", request.getParameter("testingClassName"));
        obj.put("testingClassCode", request.getParameter("testingClassCode"));
        obj.put("underTestClassName", request.getParameter("underTestClassName"));
        obj.put("underTestClassCode", request.getParameter("underTestClassCode"));

        StringEntity jsonEntity = new StringEntity(obj.toString(), ContentType.APPLICATION_JSON);

        httpPost.setEntity(jsonEntity);

        HttpResponse response = httpClient.execute(httpPost);

        int statusCode = response.getStatusLine().getStatusCode();
        if (statusCode > 299) {
            System.out.println("Errore in compilecodecoverage");
            return new ResponseEntity<>(HttpStatus.INTERNAL_SERVER_ERROR);
        }

        HttpEntity entity = response.getEntity();

        String responseBody = EntityUtils.toString(entity);
        JSONObject responseObj = new JSONObject(responseBody);

        String xml_string = responseObj.getString("coverage");
        String outCompile = responseObj.getString("outCompile");
        // PRESA DELLO SCORE UTENTE
        int userScore = ParseUtil.LineCoverage(xml_string);

        // RISULTATI ROBOT VERSO TASK4
        UriBuilder builder = new UriBuilder("http://t4-g18-app-1:3000/robots");
        builder.setParameter("testClassId", request.getParameter("testClassId"))
            .setParameter("type", request.getParameter("type"))
            .setParameter("difficulty", request.getParameter("difficulty"));

        HttpGet get = new HttpGet(builder.build());
        response = httpClient.execute(get);
        get.releaseConnection();
        // Verifica lo stato della risposta
        statusCode = response.getStatusLine().getStatusCode();
    }
}
```

Figura 4.27: Codice vecchio di MyController

È stata leggermente modificata la gestione delle eccezioni: precedentemente ogni volta in cui una chiamata HTTP non andava a buon fine (veniva restituito uno statusCode maggiore di 299) veniva direttamente restituita una `ResponseEntity<>(HttpStatus.INTERNAL_SERVER_ERROR)`. Con il refactoring si è deciso di lanciare un'eccezione ogni volta in

cui una chiamata HTTP falliva, indicandone la causa nel messaggio dell’eccezione (per esempio, nella chiamata HTTP a T7, in caso di fallimento viene lanciata un’eccezione attraverso la riga `throw new HttpServerErrorException(HttpStatus.INTERNAL_SERVER_ERROR, "Errore di compile-coverage");`). Queste vengono catturate attraverso il blocco *try-catch* presente in `MyController`. Possiamo osservare come il refactoring abbia giovato alla leggibilità e alla lunghezza del codice.

#### 4.4.1 Codice: RunnerHelper

Osserviamo ora i vari metodi della nuova classe, con particolare attenzione ai nuovi.

```
public JSONObject bossRushRunner(URIBuilder builder, ScorePair userScore, HttpServletRequest request) throws
    ClientProtocolException, IOException, ParseException, URISyntaxException {
    List<Integer> radoopScores = roboScoresBossRushGet(builder, type:"radoop");
    List<Integer> evosuiteScores = roboScoresBossRushGet(builder, type:"evosuite");

    JSONObject result = responseBuilderBossRush(radoopScores, evosuiteScores, userScore);

    saving(result, request, mode:"bossRush");

    return result;
}
```

Figura 4.28: Codice di `bossRushRunner`

Il runner è molto semplice, poiché costruisce le due liste specificando il tipo di robot a `roboScoresBossRushGet` e rimanda il risultato; ancora una volta è facilmente leggibile.

## CAPITOLO 4. MODIFICHE AL TASK 5 E AL TASK 6

---

```
public List<Integer> roboScoresBossRushGet(URIBuilder builder, String type) throws
    URISyntaxException, ParseException, IOException {
    URIBuilder helper = builder;
    helper.setParameter(param:"type", type);

    List<Integer> robot = new ArrayList<>();
    HttpResponse response;
    for(int i = 1; i < 11; i++) {
        URIBuilder subHelper = helper;
        subHelper.setParameter(param:"difficulty", String.valueOf(i));

        HttpGet get = new HttpGet(subHelper.build());
        try {
            response = httpClient.execute(get);
            get.releaseConnection();

            int statusCode = response.getStatusLine().getStatusCode();
            if (statusCode > 299) {
                break;
            }
        } catch (Exception e) {
            throw new HttpServerErrorException(HttpStatus.INTERNAL_SERVER_ERROR, "Errore in " + type + " " + i);
        }

        HttpEntity entity = response.getEntity();
        String responseBody = EntityUtils.toString(entity);
        JSONObject responseObj = new JSONObject(responseBody);

        String score = responseObj.getString(name:"scores");
        Integer roboScore = Integer.parseInt(score);
        robot.add(roboScore);
    }

    return robot;
}
```

Figura 4.29: Codice di `roboScoresBossRushGet`

Vediamo il funzionamento di questo metodo:

1. Vengono utilizzati due URI "copia" di appoggio per effettuare le chiamate HTTP. Il primo, `helper` fa uno "screenshot" del URI attuale, che punta a `http://t4-g18-app-1:3000/robots/[testClassID]`, per poi aggiungergli un parametro relativo al tipo passatogli in precedenza;
2. Il secondo URI di appoggio, `subHelper` aggiunge all'URI la difficoltà attuale sulla quale si sta facendo la chiamata HTTP: l'URI diventa quindi il seguente

```
http://t4-g18-app-1:3000/robots/[testClassID]/[type]/[  
difficulty];
```

3. Si effettuano  $n$  richieste HTTP attraverso un ciclo for, ottenendo le coverage dei robot per ogni difficoltà. Nel caso non venga trovato nulla all'esecuzione della GET, si esce dal ciclo;
4. Dopo la chiamata HTTP, se non si è ancora usciti dal ciclo, ovvero se ci sono ancora robot disponibili, si aggiunge il risultato ottenuto alla lista da restituire;
5. Infine, si restituisce la lista costruita.

Questo metodo è stato aggiunto *ex-novo*: inizialmente si erano fatti due cicli for rispetto ad entrambi i tipi di robot, ma quando ci si è resi conto che i cicli erano sostanzialmente uguali si è deciso di riunire tutto in un unico metodo: la differenza tra i due è il parametro di tipo stringa `type` passato nella funzione, che, essendo chiamata in `bossRushRunner`, viene settato una volta a "randoop" ed una volta ad "evosuite".

## CAPITOLO 4. MODIFICHE AL TASK 5 E AL TASK 6

---

```
public JSONObject responseBuilderBossRush(List<Integer> randoopScores, List<Integer> evosuiteScores, ScorePair userScore) {
    JSONObject result = new JSONObject();

    int i = 0;
    boolean globalWin = false;

    int numberOfBeaten = 0;
    int numberOfUnbeaten = 0;
    for(i = 0; i < randoopScores.size(); i++) {
        if(randoopScores.get(i) <= userScore.getScore()) {
            result.put("beaten"+String.valueOf(numberOfBeaten+1), String.valueOf(i+1) + "&" + String.valueOf(randoopScores.get(i)));
            numberOfBeaten++;
        }
        else{
            result.put("unbeaten"+String.valueOf(numberOfUnbeaten+1), String.valueOf(i+1) + "&" + String.valueOf(randoopScores.get(i)));
            numberOfUnbeaten++;
        }
    }

    for(i = 0; i < evosuiteScores.size(); i++) {
        if(evosuiteScores.get(i) <= userScore.getScore()) {
            result.put("beaten"+String.valueOf(numberOfBeaten+1), String.valueOf(i+1+randoopScores.size()) + "&" + String.valueOf(evosuiteScores.get(i)));
            numberOfBeaten++;
        }
        else{
            result.put("unbeaten"+String.valueOf(numberOfUnbeaten+1), String.valueOf(i+1+randoopScores.size()) + "&" + String.valueOf(evosuiteScores.get(i)));
            numberOfUnbeaten++;
        }
    }

    if(numberOfBeaten == randoopScores.size() + evosuiteScores.size())
        globalWin = true;

    result.put("outCompile", userScore.getOutCompile());
    result.put("coverage", userScore.getXmlString());
    result.put("score", String.valueOf(userScore.getScore()));
    result.put("win", String.valueOf(globalWin));
    result.put("numberOfBeaten", String.valueOf(numberOfBeaten));
    result.put("numberOfUnbeaten", String.valueOf(numberOfUnbeaten));

    return result;
}
```

Figura 4.30: Codice di responseBuilderBossRush

Questo metodo costruisce l’oggetto JSON da restituire a T5, prendendo in ingresso le due liste di score costruite nel metodo precedente ed uno ScorePair, contenete i valori di coverage dell’utente e un messaggio relativo alla compilazione. Vediamo il suo funzionamento:

1. Vengono inizializzati tre parametri rilevanti all’esito della partita: globalWin, un booleano (inizialmente settato a falso) indicante la vittoria effettiva del giocatore (che, come specificato nei precedenti capitoli, avviene nel momento in cui si sono battuti tutti i robot sfidati), numberOfBeaten e numberOfUnbeaten, due parametri che, come si può evincere dal nome, indicano il numero di robot sconfitti e quelli ancora da sconfiggere. Questi

- ultimi due serviranno sia in fase di pop-up al giocatore, sia in fase di salvataggio;
2. Si inizializza l'oggetto JSON `result` (inizialmente vuoto) da rimandare a T5;
  3. Si scorre sulle due liste passate in ingresso al metodo, confrontando i punteggi ottenuti da ogni livello di ogni robot con quelli ottenuti dal giocatore: se il robot viene battuto si incrementa `numberOfBeaten`, altrimenti `numberOfUnbeaten`;
  4. Ad ogni confronto, viene inserito un valore all'interno dell'oggetto JSON. La chiave indica che quello è l'*i*-esimo robot battuto (o non battuto), e il suo valore comprende il suo indice (importante, poiché tra quelli Randoop ed Evosuite c'è uno shift pari al numero di robot presenti nella prima lista) e la percentuale ottenuta dal robot, divisi da un carattere (`&`) che in fase front-end permetterà lo split tra i due valori. Abbiamo scelto quest'approccio poiché un oggetto JSON con un valore sia per l'indice sia per la percentuale avrebbe portato con sé il doppio dei parametri, e si è quindi optato per una scelta più leggera;
  5. Se il numero di robot battuti è pari alla somma delle dimensioni di entrambe le liste significa che il giocatore ha sconfitto tutti i robot: il parametro `globalWin` viene quindi settato a `true`;

## CAPITOLO 4. MODIFICHE AL TASK 5 E AL TASK 6

---

6. Vengono inseriti quindi in `result` i parametri da mandare a T5, e lo si restituisce.

```
public void saving(JSONObject result, HttpServletRequest request, String mode) throws ClientProtocolException, IOException {
    // conclusione e salvataggio partita
    // chiusura turno con vincitore

    //-----Aggiunta A9-----
    String nomeCUT = currentClassUnderTestTestData.nomeCUT;
    String robotScelto = currentClassUnderTestTestData.robotScelto;
    String difficolta = currentClassUnderTestTestData.difficolta;
    //-----Aggiunta A9-----

    HttpPut httpPut = new HttpPut("http://t4-g18-app-1:3000/turns/" + String.valueOf(request.getParameter(name:"turnId")));

    JSONObject obj = new JSONObject();

    if(mode.equals(anObject:"bossRush")) {
        int numberofBeaten = Integer.parseInt(result.getString(name:"numberofBeaten"));
        int numberofUnbeaten = Integer.parseInt(result.getString(name:"numberofUnbeaten"));
        String s = result.getString(name:"score") + "(" + String.valueOf(numberofBeaten) + "/" + String.valueOf(numberofBeaten + numberofUnbeaten) + ")";
        obj.put(name:"scores", s);
    }
    else
        obj.put(name:"scores", result.getString(name:"score"));

    String win = result.getString(name:"win");
    obj.put(name:"isWinner", Boolean.parseBoolean(win));

    String time = ZonedDateTime.now(ZoneOffset.UTC).format(DateTimeFormatter.ISO_INSTANT);
    obj.put(name:"closedAt", time);
    obj.put(name:"testClass", nomeCUT);
    obj.put(name:"robot", robotScelto);
    obj.put(name:"difficulty", difficolta);

    StringEntity jsonEntity = new StringEntity(obj.toString(), ContentType.APPLICATION_JSON);

    httpPut.setEntity(jsonEntity);

    HttpResponse response = httpClient.execute(httpPut);
    httpPut.releaseConnection();

    int statusCode = response.getStatusLine().getStatusCode();
    if (statusCode > 299) {
        throw new HttpServerErrorException(HttpStatus.INTERNAL_SERVER_ERROR, statusText:"Errore in put turn");
    }
}
```

Figura 4.31: Codice di `saving`, prima parte

Il metodo `saving` si occupa del salvataggio dei dati della partita mediante richieste PUT al server di T4, sia per quanto riguarda la modalità classica, sia per quanto riguarda la modalità *bossRush*. Vediamo i vari passaggi:

1. vengono salvati i valori della classe sottoposta al testing, del robot scelto e della difficoltà selezionata dall'utente rispettivamente

- te all'interno delle stringhe *nomeCUT*, *robotScelto* e *difficoltà*;
2. viene avviata la chiusura del turno e dichiarata una richiesta  
PUT all'URI `http://t4-g18-app-1:3000/turns/turnID`;
  3. viene creato un oggetto JSON (*obj*);
  4. nel caso la *mode* dovesse essere *bossRush*, allora vengono salvati negli interi *numberOfBeaten* e *numberOfUnbeaten*, rispettivamente, il numero di robot battuti e il numero di robot non battuti (presenti nell'oggetto JSON *result* in input della funzione), parametri che saranno necessari alla costuzione della stringa *s* in cui verrà salvata la coverage ottenuta dal giocatore e in aggiunta il numero di robot battuti su quelli totali. Il punteggio viene inserito all'interno di *obj*;
  5. in caso di modalità classica nell'oggetto JSON verrà salvata semplicemente la coverage del giocatore (sempre ottenuta da *result*);
  6. nell'oggetto JSON vengono salvate informazioni addizionali, quali: esistenza della partita, orario di chiusura della partita, nome della classe under test, robot scelto e difficoltà;
  7. viene configurata un'entità JSON (*jsonEntity*) utilizzando l'oggetto JSON creato in precedenza. Questa entità verrà inviata come corpo della richiesta HTTP;

8. viene eseguita la PUT, rilasciata la connessione e controllato il codice di stato.

```
// chiusura round
httpPut = new HttpPut("http://t4-g18-app-1:3000/rounds/" + String.valueOf(request.getParameter(name:"roundId")));
obj = new JSONObject();
obj.put(name:"closedAt", time);
jsonEntity = new StringEntity(obj.toString(), ContentType.APPLICATION_JSON);
httpPut.setEntity(jsonEntity);
response = httpClient.execute(httpPut);
httpPut.releaseConnection();

statusCode = response.getStatusLine().getStatusCode();
if (statusCode > 299) {
    throw new HttpServerErrorException(HttpStatus.INTERNAL_SERVER_ERROR, statusText:"Errore in put round");
}

// chiusura gioco
httpPut = new HttpPut("http://t4-g18-app-1:3000/games/" + String.valueOf(request.getParameter(name:"gameId")));
obj = new JSONObject();
obj.put(name:"closedAt", time);
jsonEntity = new StringEntity(obj.toString(), ContentType.APPLICATION_JSON);
httpPut.setEntity(jsonEntity);
response = httpClient.execute(httpPut);
httpPut.releaseConnection();

statusCode = response.getStatusLine().getStatusCode();
if (statusCode > 299) {
    throw new HttpServerErrorException(HttpStatus.INTERNAL_SERVER_ERROR, statusText:"Errore in put game");
}
```

Figura 4.32: Codice di saving, seconda parte

1. viene avviata la chiusura del round e viene dichiarata una richiesta PUT all'URI `http://t4-g18-app-1:3000/rounds/roundID`;
2. viene reinizializzato l'oggetto JSON `obj` ed inserito l'orario di terminazione;

3. viene configurata un'entità JSON (`jsonEntity`) utilizzando l'oggetto JSON creato in precedenza. Questa entità verrà inviata come corpo della richiesta HTTP;
4. viene eseguita la PUT, rilasciata la connessione e controllato il codice di stato.
5. viene avviata la chiusura della partita e viene dichiarata una richiesta PUT all'URI `http://t4-g18-app-1:3000/games/gameID` ;
6. viene reinizializzato l'oggetto JSON `obj` ed inserito l'orario di terminazione;
7. viene configurata un'entità JSON (`jsonEntity`) utilizzando l'oggetto JSON creato in precedenza. Questa entità verrà inviata come corpo della richiesta HTTP;
8. viene eseguita la PUT, rilasciata la connessione e controllato il codice di stato.

```
public JSONObject normalRunner(URIBuilder builder, ScorePair userScore, HttpServletRequest request)
    throws ClientProtocolException, IOException, URISyntaxException {
    int roboScore = roboScoreNormalGet(builder, request);

    JSONObject result = responseBuilderNormal(roboScore, userScore);
    saving(result, request, mode:"classic");
    return result;
}
```

Figura 4.33: Codice di `normalRunner`

. Infine, per quanto riguarda i metodi della modalità **Classica**, di fatto non sono stati modificati nella loro logica, ma ne è stato effettuato

solo un refactoring rispetto a quello che facevano. Di base, lo schema del loro funzionamento è lo stesso per quelli della **BossRush**:

1. Chiamata del runner in `MyController`;
2. Ottenimento dello score;
3. Costruzione dell'oggetto JSON da mandare a T5;
4. Salvataggio.

Non riportiamo quindi tutti i codici relativi a questa modalità poiché sostanzialmente identici alla versione precedente.

# Capitolo 5

## Organizzazione del lavoro

### 5.1 Organizzazione del lavoro

L'organizzazione del lavoro per la comprensione, implementazione e testing del requisito R10 si è basata su Daily Scrum, un framework di gestione di progetti facente parte delle metodologie agili. Scrum fonda le sue basi sulla gestione e lo sviluppo di progetti software ed è basato su un set di principi e valori definiti nel "Manifesto Agile" che offrono una struttura organizzativa ai team di sviluppo affinché possano lavorare in maniera efficiente e collaborativa.

#### 5.1.1 Agile: Scrum

Nel contesto del progetto i team hanno adottato il framework Scrum come metodologia di gestione e sviluppo del requisito R10. In particolare, il progetto è stato sviluppato in **cinque iterazioni** lunghe circa

due settimane, per un totale complessivo di circa due mesi e mezzo di lavoro. L'inizio di ogni iterazione ha previsto l'ottenimento di un *Iteration Planning* che consentisse di definire gli obiettivi, suddivisi in *task* di piccole dimensioni distribuiti tra i vari membri dei team, da portare a compimento nelle successive due settimane.

### 5.2 Prima iterazione

Nel corso della prima iterazione, sono state svolte attività preliminari di avvio al lavoro di gruppo, quali l'analisi e la comprensione dei task T5 e T6 (la cui profonda comprensione è stata fondamentale per l'implementazione successiva del requisito), l'organizzazione e la cooperazione dei team, la valutazione della documentazione già esistente realizzata dai gruppi G12 e G16, e la visione di video tutorial al fine di comprendere maggiormente le varie funzionalità già implementate. A seguito di questo primo lavoro, si è compreso con chiarezza la struttura logica del componente interessato e si è passati alla definizione delle user stories, qui lasciate in figura ma meglio approfondite nel capitolo 2 di questo documento.

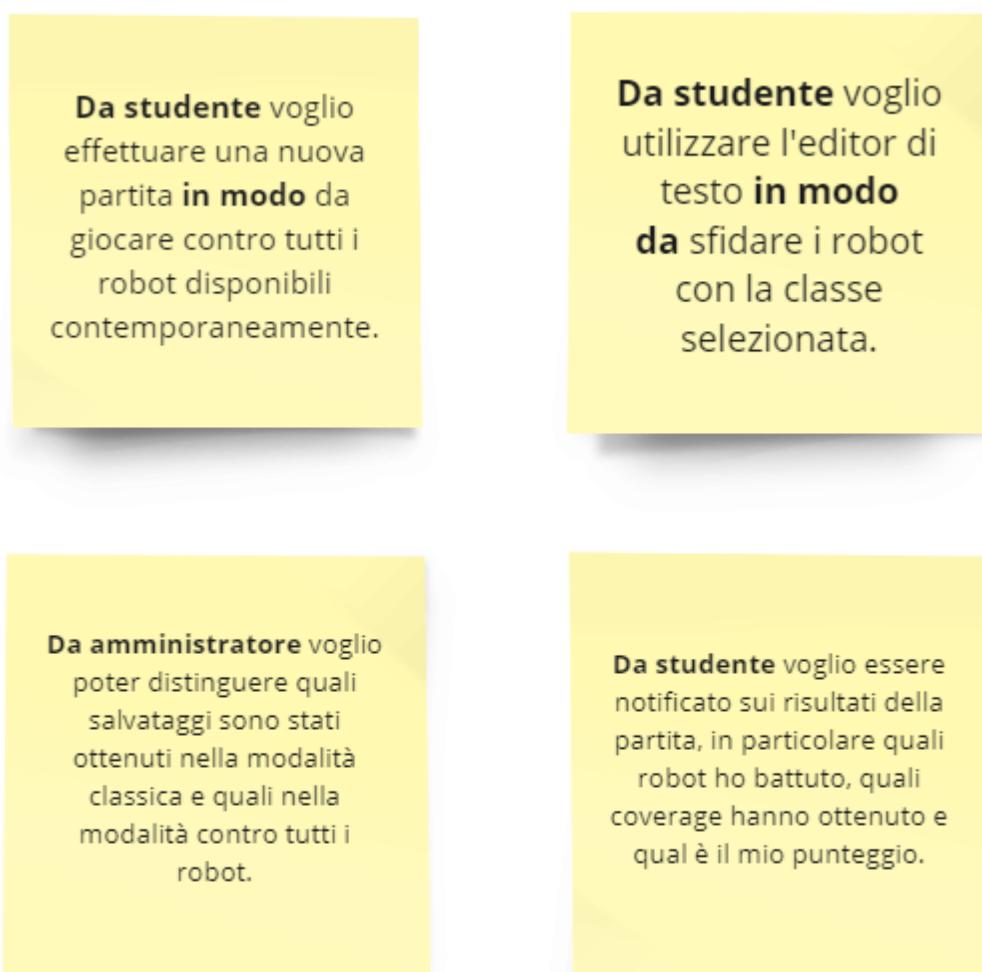


Figura 5.1: User Stories per l'implementazione del requisito R10

**N.B.:** l'implementazione delle User Stories è stata così suddivisa:

- **US1 e US2:** Iterazione n.4
- **US3 e US4:** Iterazione n.5

In seguito, si è proceduto con la definizione dell'elenco degli obiettivi e delle modifiche da apportare al sistema, inclusi quelli opzionali, mediante l'utilizzo della piattaforma Trello.

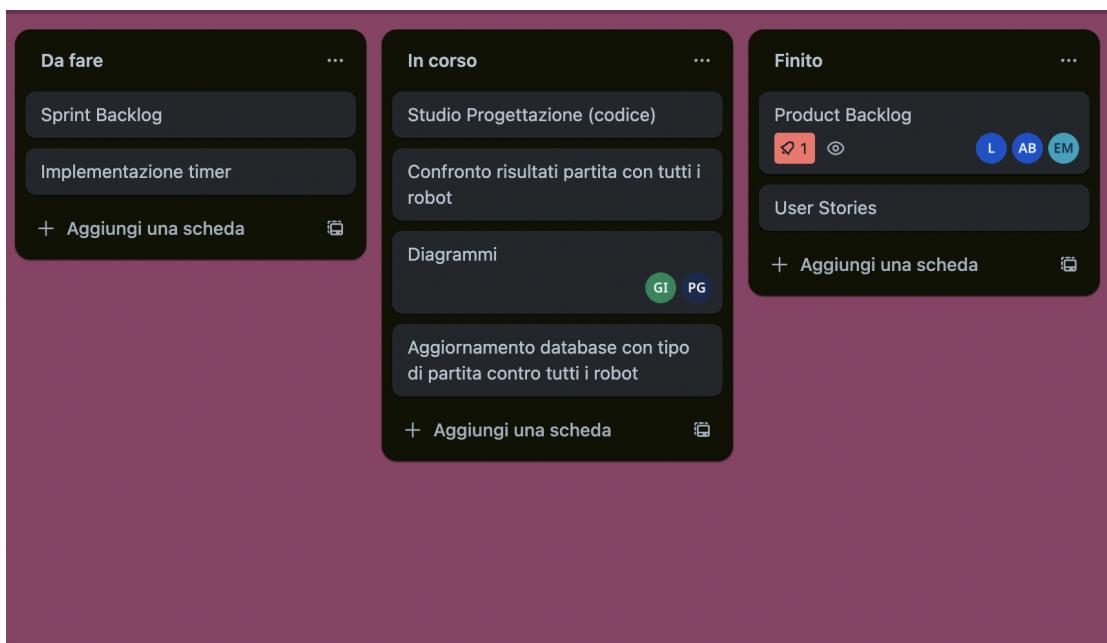


Figura 5.2: Trello board definita durante la prima iterazione

In conclusione della prima iterazione, gli obiettivi posti su trello sono stati riportati su una board della piattaforma Miro col fine di generare una prima forma di product backlog, da rifinire poi nelle successive iterazioni.

### 5.3 Seconda iterazione

La seconda iterazione ha prodotto una comprensione più accurata dei task da modificare per l'implementazione del requisito e una definizione precisa dell'organizzazione del lavoro tra i vari membri dei team. Nel pieno rispetto delle metodologie agili e di Scrum in particolare, si è passati alla definizione di un *product backlog* rifinito, partendo dall'elenco già esistente in Miro. Ciò che si è ottenuto è la riorganizzazione

dei task definiti nella prima iterazione a cui viene assegnata una priorità in base all'importanza che tale obiettivo aveva rispetto al resto.

In particolare, la priorità di un task può salire in base a:

- Priorità del cliente.
- Urgenza di ricevere feedback.
- Difficoltà nell'implementazione.
- Relazioni simbiotiche tra gli elementi (es. l'implementazione di B è più semplice se A esiste già).

Sulla base di tali considerazioni, si è quindi generato il seguente product backlog:

## Backlog

| Order | ID | Item  | Type        | Status      |
|-------|----|---|-------------|-------------|
| 1     | 1  | User stories  | Progettuale | Completed   |
| 2     | 2  | Stima delle complessità   | Progettuale | Not started |
| 3     | 3  | Studio progettazione (codice)                                   | Progettuale | Started     |
| 4     | 4  | Aggiornamento documentazione (diagrammi e Sprint Backlog)       | Progettuale | Started     |
| 5     | 5  | Confronto risultati partita con tutti i robot                   | Funzionale  | Started     |
| 6     | 6  | Aggiornamento database con tipo di partita contro tutti i robot | Funzionale  | Started     |
| 7     | 7  | Implementazione timer   | Funzionale  | Not started |

Figura 5.3: Product backlog ridefinito a partire dall'elenco di task ottenuto nella prima iterazione

Al termine dell'iterazione, si è realizzato un communication diagram relativo ai componenti T5 e T6 per la migliore comprensione delle modifiche da apportare per l'implementazione del requisito da realizzare. La spiegazione del funzionamento dei componenti e la descrizione dettagliata del communication diagram sono presenti al paragrafo 3.6.

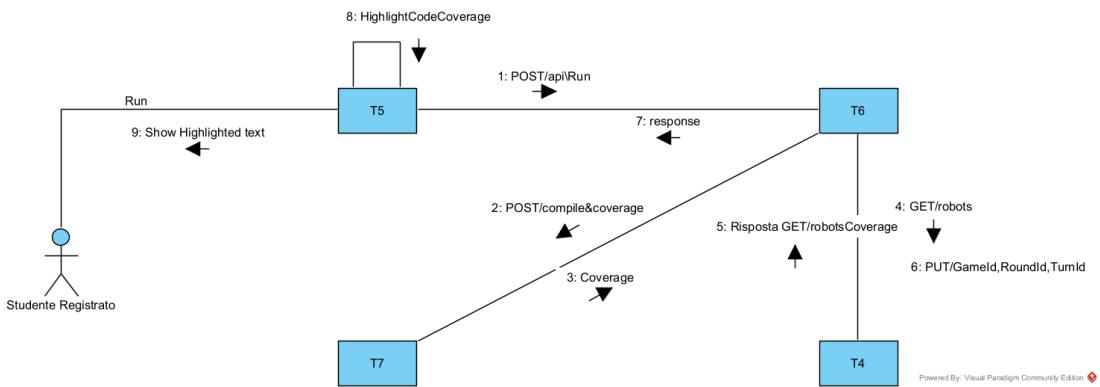


Figura 5.4: Communication diagram relativo al fine partita per il calcolo dei punteggi e dell'eventuale vittoria

## 5.4 Terza iterazione

La terza iterazione ha visto i due team entrare nel vivo del lavoro, generando codice nuovo e disegnando nuovi diagrammi che aiutassero a comprendere nel dettaglio il funzionamento del sistema. Prima di passare alle modifiche effettive è risultato comunque fondamentale effettuare una stima dei costi attraverso il metodo degli *Story Points*, ovvero una tecnica di stima dei costi che consente di definire la complessità di implementazione di una user story in termini di tempo ed effort, pesandone la complessità rispetto alle seguenti informazioni:

- **Rischio:** la quantità di rischio o incertezza associata con il task (rientra in questo caso la necessità di interfacciarsi con terzi, in questo caso gli altri team di sviluppo).
- **Ripetizione:** esperienza del team nell'implementazione di task simili. Questo punto ha una applicabilità solo se considerato su larga scala in quanto è più preciso quante più sono grandi le esperienze pregresse del team stesso.
- **Complessità:** complessità effettiva dell'implementazione del task anche in base alla chiarezza di definizione del obbiettivo preposto.

Sulla base di tali considerazioni si è dunque potuto eseguire una stima dei costi a partire dai seguenti punti:

1. Determinare la sequenza degli story points: si tratta di determinare una sequenza numerica o pseudonumerica che consenta di assegnare un valore di complessità al task considerato. Solitamente si sfrutta una versione modificata della sequenza di Fibonacci (0,0.5,1,2,3,5,8,...) oppure si usa il metodo del **t-shirt sizing** che sfrutta come sequenza quella relativa alle taglie delle t-shirt (XS, S, M, L, XL ed XXL). In questo contesto si è preferito usare quest'ultima.
2. Creazione della matrice degli story points: consiste nel creare una matrice che, a partire dalla sequenza definita nel pun-

## CAPITOLO 5. ORGANIZZAZIONE DEL LAVORO

---

to precedente, consenta la visione anche grafica del peso della user story e una migliore comprensione della quantità di effort che l'implementazione del task comporta. Di seguito è possibile visualizzare la matrice generata dal team.

| Story Point | Amount of effort required | Amount of time required | Task Complexity   | Task risk or uncertainty |
|-------------|---------------------------|-------------------------|-------------------|--------------------------|
| XS          | Minimum Effort            | A few minutes           | Little complexity | None                     |
| S           | Minimum Effort            | A few hours             | Little complexity | None                     |
| M           | Mild Effort               | A day                   | Low complexity    | Low                      |
| L           | Moderate Effort           | A few days              | Medium Complexity | Moderate                 |
| XL          | Sever Effort              | A week                  | Medium Complexity | Moderate                 |
| XXL         | Maximum Effort            | A month                 | High Complexity   | High                     |

Figura 5.5: Matrice degli story points a partire dalla sequenza delle taglie di t- shirt

3. Planning poker meeting: termina la fase apriori della stima dei costi effettuando un meeting tra tutti i componenti del team che assegnano un peso ad ogni user story sulla base di quanto lavoro ognuno crede di poter apportare e quanto effort ciò può richiedere. Il peso finale di ogni user story viene assegnato eseguendo una sorta di media tra tutti i pesi che sono stati assegnati dai singoli membri del team. Nel caso specifico del requisito R10, note le user stories già discusse nel Capitolo 2 e riportare nel paragrafo precedente, il costo finale assegnato ad ogni task è stato il seguente:

- US01: L.
- US02: XS.
- US03: XL.
- US04: M.

Alla luce di ciò si è potuto quindi definire gli obiettivi delle iterazioni ed è stato possibile immaginare una deadline per ogni user story. Si osservi che tali obiettivi non sono ultimi nel senso che, in completa simbiosi con le metodologie agili, gli story points possono essere ridefiniti iterativamente ad ogni iterazione al fine di avere sempre una chiara stima di ciò che, volta per volta, bisogna andare a fare. Completata la stima dei costi, l'iterazione si è conclusa con l'avviamento delle modifiche ai componenti T5 e T6 al fine di implementare la prima user story. In particolare, le modifiche apportate sono le seguenti:

- **T5:** aggiunta del parametro *"gameMode"* all'interno del file *main.js*, che può assumere i valori di *"normal"* e *"bossRush"*, in modo tale da consentire la distinzione tra le due modalità di gioco. Come già sottolineato in precedenza, questa modifica nell'iterazione finale non è stata apportata.
- **T6:** implementata la biforcazione delle partite in base alla modalità di gioco, modificata la richiesta verso T4 (componente gestore del database) al fine di ottenere le coverage dei robot (la richie-

sta di coverage diventa sostanzialmente una richiesta iterativa sulle due tipologie di robot disponibili, EvoSuite e Randoop).

## 5.5 Quarta iterazione

La quarta iterazione è finalizzata al compimento delle user story 1 e user story 2.

La documentazione ereditata relativa ai task T5 e T6 si è dimostrata difficile da interpretare poiché non era chiara riguardo al loro obiettivo. Questi task facevano riferimento a una versione del gioco in cui le componenti T5 e T6 erano strettamente integrate e si occupavano principalmente della parte front-end del componente Game Engine. Di conseguenza, la documentazione fornita non è stata di grande aiuto per il nostro requisito, poiché trattava principalmente il funzionamento del front-end del Game Engine anziché il back-end, che era di nostro interesse. A causa delle problematiche riscontrate, lo studio del funzionamento è stato condotto esclusivamente attraverso l'analisi del codice sorgente. In seguito a un problema con l'API fornita dal task A3, che non permetteva l'aggiornamento corretto dell'entità "Turn" nel database tramite il metodo PUT, abbiamo deciso di passare alla versione del gruppo A9. Questa decisione è stata influenzata anche dal fatto che la versione A9 offre la possibilità di selezionare la modalità di gioco, un aspetto strettamente legato al nostro compito. Inoltre, questa nuova versione ha risolto il problema relativo alla PUT e ha

introdotto significative modifiche nella struttura delle entità, tra cui l'aggiunta del parametro "Robot" nell'entità "Turn". Questa modifica è stata particolarmente vantaggiosa per il nostro requisito, poiché ha permesso una ulteriore distinzione tra le modalità di gioco. Ad esempio, nella modalità "bossRush", il parametro "Robot" sarà sempre "Tutti i robot", mentre nella modalità "Classica" sarà specificato come "Evosuite" o "Randoop".

## 5.6 Quinta iterazione

La quinta e ultima iterazione conclude la trattazione del requisito assegnato e si concentra sullo sviluppo delle user story 3 e user story 4, inoltre si pone l'obiettivo di integrare il lavoro svolto dal gruppo A9. Il punto di partenza delle nostre modifiche è stata la versione aggiornata sviluppata dal gruppo A9. Questa versione ha introdotto diverse migliorie, inclusa una schermata che permette di scegliere la modalità di sfida contro tutti i robot. Non abbiamo apportato modifiche alla struttura delle classi del task, ma abbiamo deciso di creare un diagramma di attività per comprendere meglio il funzionamento della funzione "run". Sono stati modificati i seguenti file: *editor.html* (T5) e *MyController.java* (T6). Inoltre, è stato aggiunto il file *RunnerHelper.java* come conseguenza del refactoring.

## 5.7 Considerazioni e sviluppi futuri

Durante gli ultimi mesi ci siamo resi conto di quanto una buona organizzazione del lavoro sia fondamentale per il raggiungimento dei propri obiettivi; in particolare, questo è ancora più vero quando si parla dell'implementazione di un componente su un sistema software già implementato di cui si conosceva poco. In questi termini dunque, l'approccio di una metodologia agile quale SCRUM è risultato fondamentale per la buona riuscita di questo lavoro e ha notevolmente semplificato tutte quelle che sono le necessarie interazioni tra gli sviluppatori (in questo caso i due team A6 e A11) e il cliente (rappresentato dal docente) in quanto ha consentito di avere sempre ben chiaro quali fossero gli obiettivi e come il sistema dovesse comportarsi, senza aver mai rischiato di apportare implementazioni inutili, non richieste o addirittura sbagliate.

Si conclude questo capitolo con alcune delle idee di implementazione opzionali che c'eravamo proposti e che per motivi di tempo non è stato possibile realizzare, certi del fatto che potranno essere utili come idea di progetto per i gruppi futuri che arriveranno:

- Per rendere il gioco più stimolante, oltre all'implementazione della modalità "BossRush", si era pensato alla possibilità di introdurre un **timer di gioco** che desse un tempo limite al giocatore entro il quale completare la scrittura dei suoi test. Eventualmente si potrebbe pensare di introdurre l'implementazione

di un pulsante nella schermata iniziale del gioco che consenta l'eventuale attivazione/disattivazione del timer stesso.

- Ad oggi, il sistema esegue i confronti di copertura del codice soltanto attraverso le linee di codice (LOC), una possibile nuova implementazione sarebbe quella di introdurre ulteriori modalità di gioco che consentano di battere i robot tramite **diverse metriche di copertura del codice**. Alcuni esempi di metriche di copertura, oltre a quella della copertura delle linee di codice, sono:
  1. Decision Coverage
  2. Weakmutation coverage
  3. Method Coverage
- Analogamente alla tipologia di coverage scelta si potrebbe pensare di implementare più modalità di gioco che obblighino il giocatore a **testare una classe seguendo un metodo specifico** (si pensi al metodo delle classi di equivalenza) oppure una nuova modalità di gioco in cui non viene fornito il codice della classe che incoraggi il **testing blackbox**

# Capitolo 6

## Testing

### 6.1 Testing dell'applicazione

Testare le applicazioni è una fase cruciale dello sviluppo software e assume un ruolo fondamentale quando si adottano metodologie di tipo agile, in cui il testing viene eseguito costantemente durante ogni iterazione di sviluppo. Per il testing del funzionamento della modalità "BossRush" abbiamo deciso di applicare la tecnica di testing white-box della copertura delle decisioni, applicata al testing delle GUI, in modo da coprire tutti gli scenari di utilizzo che sono stati ideati nei capitoli precedenti. In particolare, affinché fosse possibile coprire tutti gli scenari ed i flussi di utilizzo dell'applicazione (relativi alla nuova modalità di gioco), abbiamo ideato il seguente albero di decisione:

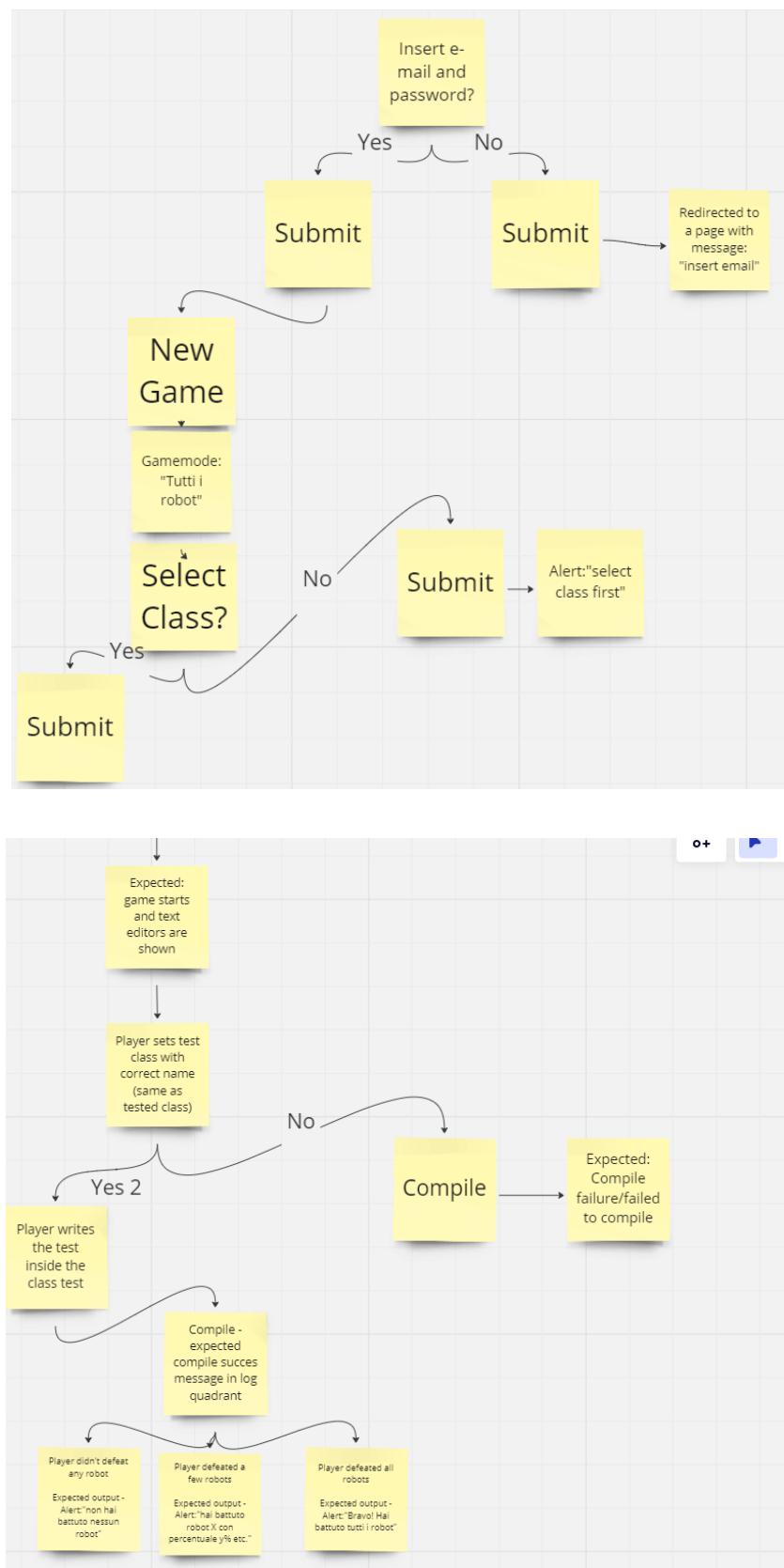


Figura 6.1: Decision Tree per la copertura degli scenari di utilizzo

Considerata la natura a microservizi dell'applicazione, molte delle decisioni presenti nell'albero potrebbero sembrare inutili in quanto funzioni di componenti non direttamente modificati dal gruppo; tuttavia, per completezza, abbiamo preferito testare anche alcune funzioni di altri componenti in modo da essere sicuri che le modifiche apportate non disturbassero gli altri funzionamenti. In particolare, i test case completi sono schematizzati in maniera tabellare come segue.

**OSS:** sul GitHub del nostro gruppo sono disponibili dei video dimostrativi che mostrano i risultati dei test, coprendo tutte le possibili decisioni che sono state definite nell'immagine precedente. Qui si lasciano al lettore una serie di screenshot relativi al funzionamento di fine partita.

| Nome del Test         | Precondizioni  | Passi da eseguire  | Output atteso   | Esito Test |
|-----------------------|--|--|---|------------|
| Login no e-mail       | User has registered and is on login page   | Press button "Accedi"  | A page with message "Email"   | OK         |
| BossRush no class     | User has logged in   | Press button New Game -> Press button "Second Mode (1VSAll)" -> Press  | Alert:"Seleziona una classe"  | OK         |
| BossRush CompileError | User has logged in, class Calcolatrice and class VCardBean have been uploaded        | Press button New Game -> Press button "Second Mode (1VSAll)" -> Select class Calcolatrice -> Press button "Play against all Robots" -> press "Submit" -> set test class with | In console a build error is shown   | OK         |
| BossRush Lose         | User has logged in, uploaded the class and started the match on class "Calcolatrice" | Change test class name to "testCalcolatrice" -> press button "Compile" -> press "play/submit"  | Alert:"Oh no! Non hai battuto nessun Robot! Hai totalizzato (0% LOC coverage) e devi                          | OK         |
| BossRush PartialWin   | User has logged in, uploaded the class and started the match on class "VCardBean"    | Copy and paste file "TestVCardBean2" in the text editor->press button "Compile" -> press "play/submit"   | Alert:"Bravo! Hai totalizzato (32% LOC coverage) e hai vinto contro Robot 1(30%), Robot 2(31%). Ti restano da | OK         |
| BossRush Win          | User has logged in, uploaded the class and started the match on class "Calcolatrice" | Copy and paste file "TestCalcolatrice" in the text editor -> press button "Compile" -> press   | Eccellente: Hai battuto tutti i Robot! Hai totalizzato (100% LOC coverage) e                                  | OK         |

Figura 6.2: Tabella dei test delle UI effettuati

```

localhost dice
Eccellente! Hai battuto tutti i Robot! Hai totalizzato (100% LOC coverage) e
hai vinto contro Robot 1(100%), Robot 2(100%).
OK

Class Under Test
public int divide(int a, int b) {
    if(b == 0) {
        throw new ArithmeticException("Cannot divide by zero");
    }
    return a / b;
}

Console
INFO: Running CLIENTPROJECT/TEST CALCULATOR
INFO: Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.118 s - in
ClientProject.TestCalculator
INFO: Results:
INFO: Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
INFO: -- jacoco-0.8.8-report [jacoco-report] @ code-coverage --
INFO: Loading execution data file /ClientProject/target/jacoco.exec
INFO: Analyzed bundle 'code-coverage' with 1 classes
INFO: BUILD SUCCESS
INFO: Total time: 4.088 s
INFO: Finished at: 2024-02-18T17:17:32Z
INFO:

Confronto risultati
Fatto Risultati (percentuale di linee coperte)
Il tuo punteggio EvoSuite: 75% LOC
Il tuo punteggio Jacoco: 75% LOC
Informazioni aggiuntive di copertura:
Il tuo punteggio EvoSuite: 100% Branch
Il tuo punteggio EvoSuite: 100% Exception
Il tuo punteggio EvoSuite: 70% WeakMutation
Il tuo punteggio EvoSuite: 6% Input
Il tuo punteggio EvoSuite: 0% Method
Il tuo punteggio EvoSuite: 0% MethodNotFoundException
Il tuo punteggio EvoSuite: 66% CBranch

```

Figura 6.3: Test Vittoria contro tutti i robot

```

localhost dice
Oh no! Non hai battuto nessun Robot! Hai totalizzato (75% LOC coverage) e
devi battere Robot 1(100%), Robot 2(100%).
OK

Class Under Test
public int divide(int a, int b) {
    if(b == 0) {
        throw new ArithmeticException("Cannot divide by zero");
    }
    return a / b;
}

Console
INFO: Running CLIENTPROJECT/TEST CALCULATOR
INFO: Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.243 s - in
ClientProject.TestCalculator
INFO: Results:
INFO: Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
INFO: -- jacoco-0.8.8-report [jacoco-report] @ code-coverage --
INFO: Loading execution data file /ClientProject/target/jacoco.exec
INFO: Analyzed bundle 'code-coverage' with 1 classes
INFO: BUILD SUCCESS
INFO: Total time: 7.052 s
INFO: Finished at: 2024-02-18T17:15:01Z
INFO:

Confronto risultati
Fatto Risultati (percentuale di linee coperte)
Il tuo punteggio EvoSuite: 75% LOC
Il tuo punteggio Jacoco: 75% LOC
Informazioni aggiuntive di copertura:
Il tuo punteggio EvoSuite: 100% Branch
Il tuo punteggio EvoSuite: 100% Exception
Il tuo punteggio EvoSuite: 70% WeakMutation
Il tuo punteggio EvoSuite: 6% Input
Il tuo punteggio EvoSuite: 0% Method
Il tuo punteggio EvoSuite: 0% MethodNotFoundException
Il tuo punteggio EvoSuite: 66% CBranch

```

Figura 6.4: Test modalità BossRush in cui il player viene sconfitto da tutti i robot

```

localhost dice
Bello! Hai totalizzato (77% LOC coverage) e hai vinto contro Robot 1(50%).
Robot 2(50%) Ha Robot 4(52%). Ti restano da battere Robot 3(65%), Robot
5(60%), Robot 6(60%).
OK

Class Under Test
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;
public class VCardBean {
    private VCardBean vCardBean;
    @Before
    public void setup() {
        vCardBean = new VCardBean();
    }
    @Test
    public void testSetAndGetFirstName() {
        vCardBean.setFirstName("John");
        assertEquals("John", vCardBean.getFirstName());
    }
}

Console
INFO: Running CLIENTPROJECT/TEST VCARDBEAN
INFO: Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.001 s - in
ClientProject.TestVCardBean
INFO: Results:
INFO: Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
INFO: -- jacoco-0.8.8-report [jacoco-report] @ code-coverage --
INFO: Loading execution data file /ClientProject/target/jacoco.exec
INFO: Analyzed bundle 'code-coverage' with 1 classes
INFO: BUILD SUCCESS
INFO: Total time: 7.052 s
INFO: Finished at: 2024-02-18T17:15:01Z
INFO:

Confronto risultati
Fatto Risultati (percentuale di linee coperte)
Il tuo punteggio EvoSuite: 77% LOC
Il tuo punteggio Jacoco: 77% LOC
Informazioni aggiuntive di copertura:
Il tuo punteggio EvoSuite: 100% Branch
Il tuo punteggio EvoSuite: 100% Exception
Il tuo punteggio EvoSuite: 70% WeakMutation
Il tuo punteggio EvoSuite: 6% Input
Il tuo punteggio EvoSuite: 0% Method
Il tuo punteggio EvoSuite: 0% MethodNotFoundException
Il tuo punteggio EvoSuite: 66% CBranch

```

Figura 6.5: Test vittoria parziale

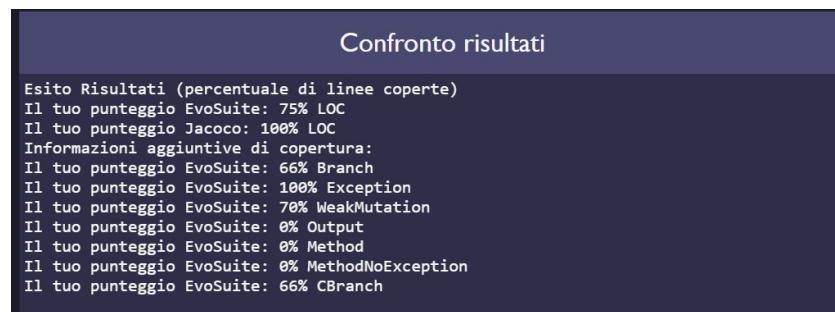


Figura 6.6: Test Confronto risultati di fine partita

# Capitolo 7

## Guida all'installazione

### 7.1 Installazione

#### 7.1.1 Prerequisiti per l'installazione

Prima di procedere con l'installazione dell'applicazione, assicurarsi di avere i seguenti componenti installati sul sistema:

- **Java** : verificare la presenza di Java eseguendo il seguente comando nel terminale o prompt dei comandi:

```
java --version
```

se Java non è installato o la versione è inferiore a quella richiesta, scaricarlo da

"[https://www.java.com/download/ie\\_manual.jsp](https://www.java.com/download/ie_manual.jsp)" e seguire le istruzioni di installazione.

- **Maven** : verificare la presenza di Maven eseguendo il seguente comando:

```
mvn --version
```

se Maven non è installato, scaricarlo da

"<https://maven.apache.org/download.cgi>" e seguire le istruzioni di installazione.

### 7.1.2 Download di Docker Desktop

Docker è una piattaforma open-source che utilizza contenitori (*"container"*) per facilitare il deployment di applicazioni. I contenitori offrono un ambiente isolato e portatile, consentendo alle applicazioni di essere eseguite in modo coerente su diversi ambienti. Gli sviluppatori definiscono le configurazioni degli ambienti di esecuzione attraverso file chiamati "Dockerfile". Docker semplifica il packaging, la distribuzione e la gestione delle applicazioni, contribuendo alla portabilità e alla coerenza in diversi ambienti di sviluppo e produzione. Bisogna procedere all'installazione di Docker Desktop al link "<https://www.docker.com/products/docker-desktop/>"

Avviare i file "ricompilazione-T5.bat" e "ricompilazione-T6.bat". E' necessario poi avviare il file batch "*installer.bat*", il quale porterà alla creazione di una rete (*global-network*) condivisa tra tutti i container, del volume VolumeT9 comune ai Task 1 e 9 e del VolumeT8 comune ai Task 1 e 8, oltre che i singoli container. Al termine dell'installazione bisognerà attivare tutti i container.

Successivamente, è necessario configurare il container *manvsclass-mongo db- 1* seguendo le indicazioni riportate di seguito:

1. posizionarsi all'interno del terminale del container mediante il tasto denominato *exec*.
2. lanciare il comando "*mongosh*".
3. digitare i seguenti comandi in maniera sequenziale:

```
use manvsclass

db.createCollection("ClassUT");

db.createCollection("interaction");

db.createCollection("Admin");

db.createCollection("Operation");

db.ClassUT.createIndex({ difficulty: 1 })

db.Interaction.createIndex({ name: "text", type: 1 })

db.interaction.createIndex({ name: "text" })

db.Admin.createIndex({username: 1})
```

A questo punto l'applicazione è completamente configurata ed è raggiungibile sulla porta :80.

### 7.1.3 Utilizzo dell'applicazione

In prima battuta è necessario procedere alla **registrazione dell'admin** all'URL `http://localhost/registraAdmin` mediante l'inserimento di alcuni dati, quali: nome, cognome, username, e password. Una volta terminata la registrazione è possibile caricare le classi da testare, in modo tale da procedere all'avvio del gioco.

Successivamente è possibile procedere con la **registrazione dell'utente** all'URL `http://localhost/register`, nel caso in cui non si possiedano ancora le credenziali. In alternativa, è possibile passare direttamente alla fase di **login**, mediante l'URL `http://localhost/register`. Durante la registrazione bisogna inserire il proprio nome, cognome, email e password (almeno 8 caratteri, una maiuscola, una minuscola, e un carattere speciale).

Una volta effettuato l'accesso, sarà possibile scegliere la modalità di gioco, ovvero classica o contro tutti i robot. Successivamente si potrà selezionare la classe da testare e, nel caso si sia scelta la modalità classica, anche il robot da sfidare. Infine, sarà possibile procedere all'avvio della partita.

### 7.1.4 Ricompilazione dei codici di T5 e T6

Una volta effettuate delle modifiche sui codici dei task T5 e T6, all'interno dei loro percorsi sono stati lanciati i seguenti comandi da terminale:

```
mvn clean install
```

```
mvn clean package
```

questi comandi sono spesso utilizzati in concomitanza con il file pom.xml per garantire che tutte le dipendenze dichiarate nel file pom.xml siano scaricate e che il progetto sia compilato con successo.. Essi semplificano il processo di gestione delle dipendenze e di creazione degli artefatti durante lo sviluppo del software.

Abbiamo creato due file batch, uno per T5 e uno per T6, per semplificare e velocizzare la ricompilazione del progetto. Il batch di T5 è il seguente:

```
rem Percorso del primo progetto
```

```
set PROJECT_PATH=T5-G2\t5
```

```
rem Esegui mvn clean install per il primo progetto
```

```
cd %PROJECT_PATH%
```

```
mvn clean install
```

```
mvn clean package
```

Mentre quello di T6:

```
rem Percorso del secondo progetto  
set PROJECT_PATH=T6-G12\T6
```

```
rem Esegui mvn clean install per il secondo progetto  
cd %PROJECT_PATH%  
mvn clean install  
mvn clean package
```

# Capitolo 8

## Glossario dei nomi

### 8.1 Robot

Per Robot si intendono i tool di generazione automatica di test che il giocatore deve sfidare durante una partita.

### 8.2 BossRush

Modalità di gioco implementata relativa al requisito R10 che consente di sfidare tutti i robot disponibili per una determinata classe in contemporanea.

### 8.3 Classica

Quando ci si riferisce alla modalità *classica* si intende quella relativa alla sfida contro un solo robot, ovvero la modalità originale.

## 8.4 Turno

Nel contesto della modalità di gioco "BossRush" ogni volta che un giocatore tenta di iniziare una nuova partita (Game) seleziona la modalità di gioco corretta e la classe da testare (round). Da tali precondizioni, ogni tentativo di submit è da definirsi **turno** o tentativo.

## 8.5 Giocatore

Utente attivo che gioca la partita, anche detto "Player".

# Bibliografia

- [1] discord.com. <https://discord.com/>. Visitato il 10 gennaio 2024.
- [2] miro.com. <https://miro.com/app/board/uXjVNKE26R4=/>. Visitato il 10 gennaio 2024.
- [3] trello.com. <https://trello.com/it>. Visitato il 10 gennaio 2024.
- [4] enactest project.eu. Enactest. <https://enactest-project.eu/it/>. Visitato il 10 gennaio 2024.
- [5] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Ingegneria del software. Fondamenti e principi*. Pearson, 2004.
- [6] linkedin.com. Enactest project. <https://www.linkedin.com/company/enactest-project/about/?viewAsMember=true>. Visitato il 10 gennaio 2024.