UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

SCUOLA POLITECNICA E DELLE SCIENZE DI BASE

DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE DELL'INFORMAZIONE

Corso di Laurea Magistrale in Ingegneria Informatica

ELABORATO DI SOFTWARE ARCHITECTURE DESIGN

Prof.ssa Anna Rita Fasolino

Studenti:

Alessandro Venturino M63001362

Luca Vastolo M63001392

Sommario

Capitolo 1: Introduzione 3
1.1 Contesto Applicativo 3
1.2 Obiettivo 3
1.3 Metodologia di sviluppo 4
1.4 Problematiche Affrontate 7
1.5 Stato dell'opera 7
1.6 Architettura a Microservizi 8
Capitolo 2: Progettazione e Sviluppo 9
2.1 Requisiti Funzionali 9
2.2 Requisiti Non Funzionali 9
2.3 Diagramma delle classi Package Game 10
2.3.1 GameController 10
2.3.2 GameLogic 11
2.3.3 TurnBasedGameLogic 11
2.3.4 ScalataGame 11
2.3 Implementazione modalità Scalata 13
2.4 Struttura Java per la modalità Scalata 16
2.4.1 Struttura e Funzionamento 17
2.5 Azioni Registrate 19
2.6 Funzionamento dettagliato GUIController 20
2.7 Selezione della Modalità e Configurazione della Pagina 21
2.8 Re-factoring front-end 24
2.9 Testing Funzionalità 25
2.9.1 Test chiamata API getScalate 25
2.9.2 Test chiamata API retrieveScalata 26
2.9.3 Test salvataggio partita in corso 27
2.9.4 Test Finale 27
Capitolo 3: Sviluppi Futuri 28
3.1 Re-factoring del back-end per la gestione della scalata 28
3.2 Miglioramento della gestione della scalata lato admin 29
3.3 Miglioramento esperienza di gioco 29

Capitolo 1: Introduzione

1.1 Contesto Applicativo

Il seguente elaborato mostra le modifiche apportate al gioco interattivo “Man vs Automated Testing Tool challenges”. Il Testing Game Web Application di cui tratteremo è un progetto che ha lo scopo di evidenziare l’importanza del testing, usando elementi interattivi quali il gioco.

In sostanza, lo studente, sfida un robot (Randoop/EvoSuite) capace di generare automaticamente dei test. Il giocatore vince la sfida se riesce ad ottenere un punteggio maggiore rispetto al robot, sulla base di diversi valori di copertura ottenuti alla fine della partita.

L’intera applicazione è basata su un’architettura a microservizi.

1.2 Obiettivo

Il nostro compito è quello di effettuare un re-factoring di una delle modalità di gioco presenti, in particolare della modalità di gioco “Scalata”.

La modalità scalata prevede un aumento graduale della difficoltà nelle sfide proposte. Quindi, una volta impostato il numero di round, è necessario superare tutte le sfide per arrivare in cima alla scalata. Nel caso in cui si perde una sfida, la scalata viene considerata persa.

Per re-factoring, si intende una completa ristrutturazione logica della modalità di gioco, adattandola ai principi di progettazione, implementati da un gruppo di colleghi, i quali a loro volta hanno effettuato modifiche significative alla logica di gioco.

Ci occuperemo quindi per intero del Task T5, integrando il più possibile la “nuova” modalità di gioco, con la logica di gioco esistente, garantendo quindi uniformità al codice complessivo.

All’interno della modalità scalata sono state introdotte nuove opzioni che consentono ai giocatori di selezionare il robot contro cui affrontare la scalata e di scegliere il livello di difficoltà. Inoltre, è stato aggiornato il front-end per uniformarlo al design delle altre modalità di gioco, garantendo una maggiore coerenza. Anche l’interfaccia grafica della leaderboard della scalata è stata rivista, poiché in precedenza non rispecchiava lo stile generale dell’applicazione.

1.3 Metodologia di sviluppo

Per la realizzazione del nostro progetto, abbiamo adottato un approccio basato sulle metodologie agili, al fine di favorire una gestione flessibile, collaborativa ed efficiente del lavoro. Dando priorità alla comunicazione, ai brevi e frequenti rilasci ed alla capacità di adattarci ai cambiamenti durante lo sviluppo.

Il nostro progetto è stato suddiviso in diversi sprint o iterazioni, durante i quali sono stati fissati obiettivi a breve termine, tipicamente di una settimana. Questa procedura ha permesso ai membri del team di avere chiaro l'obiettivo settimanale da raggiungere. In particolare, gli sprint sono stati così suddivisi:

- **Sprint #1:**
  - **Analisi generale dell'applicazione:**
    In questa fase iniziale, abbiamo ritenuto fondamentale comprendere la struttura complessiva dell'applicazione, andando oltre il nostro specifico incarico. Era infatti essenziale acquisire una visione d'insieme per avere una piena comprensione del contesto in cui si inseriva il nostro compito. Questa prima analisi si è rivelata cruciale per gettare solide basi per il lavoro successivo.
  - **Analisi approfondita del task assegnato T5:**
    Dopo aver acquisito una conoscenza globale dell'applicazione, ci siamo concentrati sulla parte a noi assegnata, relativa al task T5. Abbiamo iniziato approfondendo il funzionamento della nuova modalità prevista, analizzandone i dettagli e individuando le modifiche richieste. Successivamente, abbiamo identificato le componenti che necessitavano di un re-factoring, definendo chiaramente gli interventi necessari.
  - **Conclusione primo sprint:**
    Questo processo ci ha consentito di concludere il primo sprint con una chiara comprensione dei requisiti funzionali e non funzionali necessari per lo sviluppo. Questo approccio strutturato ci ha permesso di stabilire una base solida per le fasi successive del progetto.
- **Sprint #2:**
  - **Modifiche al codice:**
    In questa fase, abbiamo eliminato completamente i collegamenti al codice che non aderiva alla nuova logica, integrando soprattutto la modalità di selezione e gestione della scalata

all'interno della stessa struttura utilizzata dagli altri giochi. Questo perché la modalità scalata era gestita in modo diverso dalle altre, continuando a sfruttare i pattern della logica precedente
    - **Integrazione del codice:**
      Abbiamo completamente riscritto la modalità Scalata, integrando i package sviluppati dal gruppo precedente.
      Abbiamo utilizzato il **PageBuilder** per la creazione dinamica di pagine web personalizzate.
      All'interno del form di Scalata, è stato aggiunto il tipo di robot contro cui sfidarsi e la possibilità di selezionare il livello di difficoltà, seguendo la logica di sfida.
    - **Conclusione secondo sprint:**
      A seguito di queste modifiche il codice risulta più uniforme, leggero ed inoltre è stato rispettato il principio di riuso del software evitando di scrivere più volte le stesse cose.

- **Sprint #3:**
    - **Modifiche al codice:** è stata ristrutturata la logica di gestione della modalità scalata, è stato eliminato l'editor precedente ed è stato utilizzato l'editor nuovo con tutte le sue caratteristiche.
      È stato creato l'oggetto di tipo **ScalataGame** estensione della classe **GameLogic** (presente all'interno del package Game), che al suo interno ha tutti i metodi necessari per istanziare la nuova modalità di gioco correttamente.
      È stata aggiornata la logica di funzionamento della scalata.
      Inoltre, è stato implementato il flusso di gioco all'interno della scalata, che permette di passare da una sfida all'altra.
    - **Conclusione terzo sprint:**
      La creazione e l'istanziazione dell'oggetto di tipo ScalataGame, ci ha permesso di usufruire della logica di base presente.

- **Sprint #4:**
    - **Modifiche al codice:** è stato effettuato innanzitutto un re-factoring del front-end per la modalità scalata, in quanto essa risultava completamente distaccata dalle altre modalità.
      Successivamente anche LeaderboardScalata è stato modificato graficamente per utilizzare un unico template valido per tutto il

codice. Inoltre, nella nuova logica di scalata, è stato previsto che venga abilitato solo il pulsante “Play”, all’interno dell’editor, garantendo un tentativo unico per evitare che il giocatore possa ripetere la sfida più volte all’interno di questa modalità. Infine, sono stati identificati e descritti i possibili miglioramenti dell’applicazione.

    - **Conclusione quarto sprint:** alla fine di questo sprint, il codice risulta uniforme in tutta la sua logica. Anche il front-end risulta molto più uniforme.

Al fine di migliorare e velocizzare l’interoperabilità tra i membri del gruppo, abbiamo ritenuto necessario e vantaggioso avvalerci di diversi strumenti di supporto:

- **Microsoft Teams**: questa applicazione è stata utilizzata per la comunicazione da remoto tra i membri del team. Principalmente, attraverso videochiamate sono stati discussi tutti i passi da seguire durante la progettazione.
- **GitHub**: è stata utilizzata per avere sempre a disposizione i nuovi aggiornamenti del codice, garantendo quindi una maggiore collaborazione e soprattutto di mantenere, in caso di bisogno, versioni precedenti del codice.
- **Visual Paradigm**: è stato utilizzato per la modellazione e creazione dei diagrammi. La parte relativa all’UML è stata prodotta interamente con questo strumento.
- **Docker**: è un componente fondamentale per avviare l'applicazione. Grazie ai container, permette di eseguire l'applicazione in un ambiente isolato e replicabile direttamente sulla propria macchina locale, garantendo coerenza e semplicità nell'installazione e nell'esecuzione.
- **Visual Studio Code**: è stato utilizzato VSC come IDE di sviluppo per l’intero progetto. Il motivo di questa scelta è dato dalle molteplici estensioni presenti, che agevolano la scrittura del codice. Le estensioni utilizzate sono:
    - **Spring Boot Dashboard**: ha permesso in modo semplice, di avere a portata di mano l’intero workspace del progetto.
    - **Postman**: permette direttamente da Visual Studio di effettuare e testare richieste API, quindi di vederne l’esito.
    - **Docker**: è stata usata anche come estensione in Visual Studio perché risulta più semplice orchestrare i container dall’IDE.

1.4 Problematiche Affrontate

I principali problemi affrontati sono:

- La diversità della modalità scalata rispetto alla modalità sfida ed allenamento.
- La comprensione del codice precedentemente sviluppato per quanto riguarda Scalata, che risulta poco commentata.
- L'eccessiva ridondanza del codice e l'utilizzo di funzioni non essenziali per la gestione di scalata.
- Gestione delle chiamate back-end effettuate in scalata ed il loro risultato.

1.5 Stato dell'opera

All'inizio del nostro lavoro, l'applicazione si presentava con un'architettura riorganizzata dal gruppo precedente, che aveva effettuato un re-factoring significativo per migliorare modularità, scalabilità e manutenibilità. La struttura era suddivisa in tre package principali:

- **Interface**, responsabile della gestione delle chiamate REST attraverso un Service Manager centralizzato per semplificare l'integrazione tra i servizi;
- **Component**, che organizzava le pagine web in componenti modulari per separare la logica di business dalla visualizzazione;
- **Game**, che gestiva la logica di gioco e introduceva un back-end più strutturato per supportare l'aggiunta di nuove modalità in modo più flessibile.

Inoltre, era stato avviato un re-factoring del **Task T7**, relativo alla generazione dei report di copertura dei test, con interventi mirati a migliorare la gestione della concorrenza, l'affidabilità e la sicurezza, attraverso l'introduzione di monitor, canali di comunicazione e una gestione più rigorosa degli accessi ai file.

Una delle problematiche non ancora affrontate riguardava l'integrazione della modalità **Scalata**, che rappresentava un caso d'uso complesso non supportato dall'architettura esistente. A differenza delle modalità già implementate, la Scalata introduceva una sequenza strutturata di sfide, richiedendo una gestione avanzata della progressione e del controllo dello stato di gioco.

Il nostro lavoro si è quindi focalizzato su questi aspetti, con l'obiettivo di estendere l'architettura esistente senza comprometterne la modularità e garantendo un'integrazione efficiente della nuova modalità di gioco.

1.6 Architettura a Microservizi

I **microservizi** costituiscono un’architettura software progettata per suddividere un'applicazione complessa in una serie di componenti indipendenti, ognuno con una responsabilità ben definita. Questo approccio si contrappone all’architettura monolitica, dove tutte le funzionalità risiedono in un'unica unità, spesso difficoltosa da scalare e mantenere.

Nei microservizi, ogni componente espone le proprie funzionalità attraverso API, generalmente utilizzando protocolli leggeri come REST. Questa modularità permette di distribuire e scalare i servizi in maniera indipendente, favorendo resilienza e flessibilità: un eventuale guasto di un microservizio rimane isolato senza compromettere l’intero sistema.

Un elemento distintivo è la gestione della persistenza: ogni microservizio possiede la propria base dati, eliminando accoppiamenti e dipendenze tra i servizi.

Dal punto di vista organizzativo, i microservizi agevolano il lavoro in team: ogni gruppo può sviluppare, testare e distribuire un servizio specifico in parallelo agli altri, rendendo più rapido e agile l’intero ciclo di sviluppo.

Tuttavia, questo paradigma richiede competenze avanzate per gestire la complessità di un sistema distribuito, specialmente in ambiti come il testing, il monitoraggio e l’orchestrazione.

Rispetto all’architettura SOA (Service-Oriented Architecture), i microservizi si distinguono per la semplicità dei protocolli di comunicazione, l’assenza di una persistenza globale e l’approccio "fine-grained", in cui ogni servizio è più piccolo e focalizzato su un singolo compito. Internamente, molti microservizi adottano un’architettura esagonale: la business logic è al centro, mentre gli adattatori gestiscono l’interazione con il mondo esterno, come richieste da browser, accesso a database o scambio di messaggi.

L’architettura su cui abbiamo lavorato si basa proprio sui microservizi, sfrutta i loro vantaggi di scalabilità, modularità.

Grazie a questo approccio, il sistema può evolversi rapidamente, gestire carichi elevati e rispondere efficacemente alle esigenze di un ambiente dinamico.

I microservizi rappresentano una soluzione ideale per la costruzione di applicazioni moderne, distribuite e pronte a soddisfare le richieste di un mercato in continua trasformazione, come in questo caso.

Capitolo 2: Progettazione e Sviluppo

All'interno della progettazione software, una delle parti più cruciali è la fase di definizione dei requisiti, poiché essa rappresentate la base su cui si svilupperà poi l'intera progettazione.

I requisiti permettono di stabilire cosa il software deve fare e quali problema deve risolvere. Motivo per cui essi devono esser sviluppati in modo chiaro, esaustivo e senza ambiguità.

Questo è necessario per evitare errori.

Esistono due tipologie di requisiti:

- Requisiti funzionali, ovvero cosa deve fare il sistema in risposta a determinati input.
- Requisiti non funzionali, ovvero come deve funzionare il sistema, determinando quali caratteristiche e le qualità che il sistema deve possedere.

2.1 Requisiti Funzionali

Nel nostro specifico caso, i requisiti funzionali sono:

1) La modalità scalata deve essere gestita secondo i criteri della nuova logica.
2) La modalità scalata deve utilizzare lo stesso editor delle altre modalità.
3) La modalità scalata deve utilizzare la stessa pagina di selezione delle altre modalità.
4) La modalità scalata deve permettere all'utente di poter scegliere contro quale robot sfidarsi.
5) La modalità scalata deve permettere all'utente di poter scegliere la difficoltà con cui affrontare il robot.

2.2 Requisiti Non Funzionali

1)L'interfaccia della gestione dei servizi deve essere comune.

2) Le variabili devono essere gestite correttamente.

3)La navigazione deve essere fluida.

4)L'aspetto grafico deve essere uniforme.

2.3 Diagramma delle classi Package Game

L'obiettivo principale di **Game** è gestire il ciclo di vita di una partita in modo fluido e coerente, garantendo un'esperienza di gioco bilanciata e coinvolgente per gli utenti. Il design modulare e integrabile del controller permette di estendere facilmente la logica a nuovi giochi, migliorando la flessibilità dell'architettura complessiva.

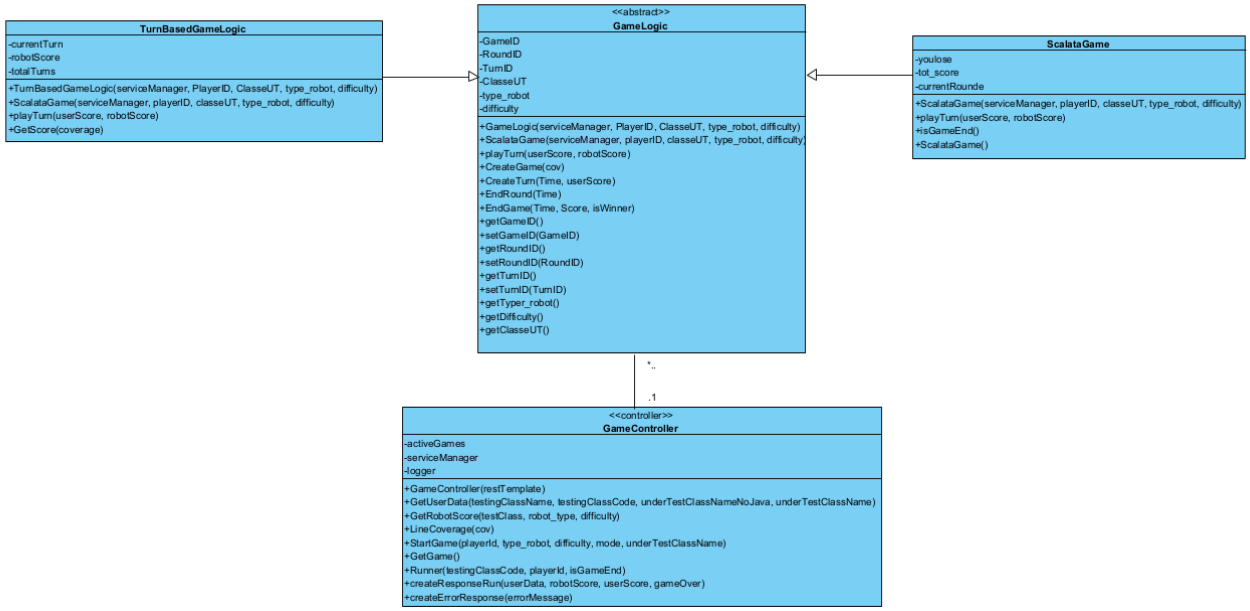


Figura 1: Diagramma delle classi Package Game.

2.3.1 GameController

Agisce come intermediario tra il client e il back-end, occupandosi della gestione delle richieste HTTP, del coordinamento del ciclo di vita delle partite e dell'interazione con servizi esterni. In particolare:

- **Gestione delle partite attive**: Mantiene una mappa (**activeGames**) delle partite in corso, associando ogni partita a un giocatore specifico (**playerId**).
- **Ricezione delle richieste**: Gestisce le richieste dal client, come l'avvio di una nuova partita o l'esecuzione di un turno, attraverso metodi annotati con **@PostMapping** e **@GetMapping**.
- **Metodi di supporto**: Include metodi come **GetUserData** e **GetRobotScore**, che recuperano informazioni necessarie mediante chiamate a servizi esterni.

2.3.2 GameLogic

Definisce i concetti fondamentali del gioco, come partite, round e turni. Offre metodi per creare, giocare e terminare le partite, e stabilisce metodi astratti che le sottoclassi devono implementare per definire la logica specifica del gioco:

- **playTurn**: Definisce la logica di esecuzione di ogni turno.
- **isGameEnd**: Stabilisce le condizioni per determinare la fine della partita.
- **getScore**: Calcola il punteggio del giocatore in base a metriche specifiche (ad esempio, la copertura del codice).

In sostanza, **GameLogic** definisce la struttura e le regole generali del gioco. Le sottoclassi che la estendono implementano i dettagli specifici del gioco stesso. Questa classe non interagisce direttamente con il client o con le richieste HTTP, ma si occupa della gestione interna della logica di gioco.

2.3.3 TurnBasedGameLogic

Estende **GameLogic** implementando la logica di gioco specifica per la modalità sfida. Questa sottoclasse definisce come vengono gestiti i turni tra i giocatori e le dinamiche specifiche della modalità sfida.

2.3.4 ScalataGame

Estende GameLogic introducendo una nuova modalità di gioco che implementa in modo diverso le tre funzioni principali:

Il metodo **playTurn(int userScore, int robotScore)** implementa la logica del confronto tra il punteggio del giocatore e quello del robot. Se il punteggio del robot supera quello del giocatore, la partita viene considerata persa e il flag youlose viene impostato a true. Diversamente, il gioco prosegue e il giocatore può affrontare il round successivo.

Il metodo **isGameEnd()** permette di verificare se la partita deve terminare. Questo metodo si basa esclusivamente sul valore della variabile youlose, stabilendo se la scalata è ancora in corso o se il giocatore è stato sconfitto.

Infine, il metodo **GetScore(int coverage)** si occupa di calcolare il punteggio totale, accumulando progressivamente il valore di coverage nei vari round.

In **Scalata Game**, l'esperienza di gioco è incentrata sul confronto diretto tra il giocatore e un avversario controllato dal computer (il robot). La semplicità della logica:

- **Confronto Diretto**: In ogni turno, il giocatore deve ottenere un punteggio superiore a quello del robot per proseguire. Questo aggiunge un elemento di tensione e competitività ad ogni fase della scalata.
- **Progressione dei Round**: Con l'incremento di currentRound, il gioco può essere strutturato in livelli successivi, ognuno con difficoltà crescente o con variazioni nel comportamento del robot.
- **Punteggio Cumulativo**: L'accumulo del punteggio attraverso GetScore incentiva il giocatore a migliorare costantemente le proprie performance, offrendo obiettivi a lungo termine oltre alla sfida immediata di ogni turno.

Questa implementazione fornisce una struttura di gioco che, pur essendo semplice, è altamente coinvolgente. Il giocatore è spinto a superare non solo l'avversario in ogni turno, ma anche a ottenere il massimo punteggio possibile nel corso dell'intera scalata.

Questa nuova modalità arricchisce l'architettura del gioco, dimostrando la flessibilità del design modulare. **Scalata Game** sfrutta l'estensibilità di **GameLogic** per offrire un'esperienza unica, incentrata sulla competizione diretta e sul miglioramento continuo, rendendo il gameplay fresco e stimolante per gli utenti.

2.3 Implementazione modalità Scalata

L'immagine sottostante presenta il workflow aggiornato per la modalità scalata. Un workflow è una rappresentazione grafica che descrive la sequenza di attività, decisioni e interazioni tra l'utente, nello specifico l'utente autenticato, ed il sistema.

Questo diagramma illustra dunque come si sviluppa il flusso operativo nella modalità scalata.

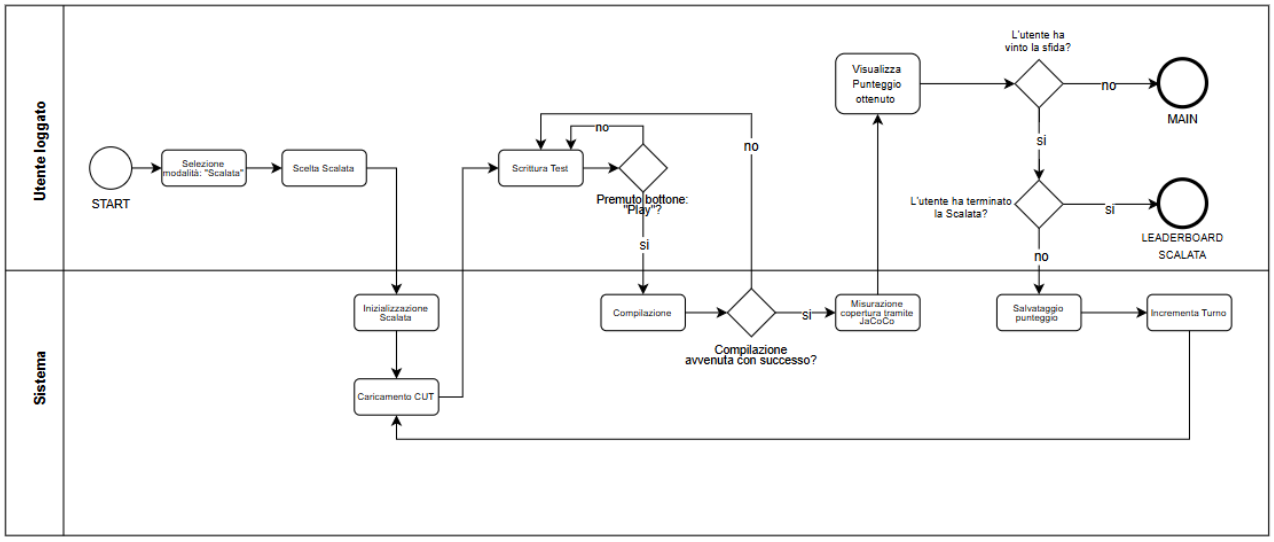


Figura 1 Workflow modalità Scalata.

L'introduzione della modalità Scalata ha richiesto una serie di modifiche strutturali e funzionali per gestire in modo efficace l'avanzamento del giocatore attraverso una sequenza predefinita di sfide.

L'obiettivo principale è quello di garantire una progressione fluida, mantenendo lo stato del gioco tra le varie fasi e assicurando la persistenza delle informazioni anche in caso di interruzioni o ricaricamenti della pagina.

Inizialmente, il nostro obiettivo era sfruttare appieno la struttura esistente del back-end per gestire la modalità scalata, integrandola in modo armonioso con le altre funzionalità del sistema. Tuttavia, poiché il back-end non era predisposto per supportare completamente questa nuova modalità di gioco, non è stato possibile realizzare un'integrazione perfetta. Abbiamo affrontato limitazioni tecniche che ci hanno impedito di implementare alcune funzionalità come previsto.

Di conseguenza, abbiamo fatto tutto il possibile per sviluppare la modalità scalata nel miglior modo possibile all'interno dei vincoli esistenti. Alcune parti sono state gestite attraverso il localStorage del browser, poiché era l'unica soluzione praticabile per garantire la continuità e la fluidità dell'esperienza di

gioco. Questa scelta ci ha permesso di offrire agli utenti una versione funzionante della modalità scalata

Per gestire questa modalità, sono state introdotte nuove variabili globali che permettono di mantenere il riferimento alla scalata selezionata, alla sfida corrente, al punteggio accumulato e all'identificativo della scalata in corso. Questi dati vengono memorizzati nel localStorage, consentendo di riprendere la scalata senza perdita di informazioni anche in caso di refresh del browser. L'indice della sfida attuale viene costantemente aggiornato e recuperato dinamicamente, evitando così incongruenze nello stato del gioco.

Uno degli aspetti più critici della modalità Scalata riguarda la gestione dell'esito delle singole sfide: se il giocatore non riesce a superare una prova, il sistema lo informa con un messaggio di sconfitta e lo reindirizza alla homepage, viceversa se supera la sfida deve essere rimandato alla sfida successiva.

In caso di fallimento della scalata, viene eliminato ogni riferimento a quest'ultima, garantendo che non sia più possibile riprendere dallo stesso punto. Questa gestione è affidata alla funzione **handleGameLoss(),** che si occupa di mostrare un modal informativo e di eseguire una pulizia completa dei dati salvati e di riportare l'utente alla homepage.

Diversamente, se il giocatore vince la sfida, entra in gioco la funzione **handleGameProgress(userScore)**, il cui scopo è quello di determinare se ci sono ulteriori prove da affrontare o se la scalata può considerarsi completata.

Vengono aggiornati sia il punteggio totale che l'indice della sfida corrente, così da riflettere lo stato più recente della partita.

Se la scalata prevede ancora ulteriori prove, il sistema prepara l'ambiente per la sfida successiva, aggiornando le variabili di stato e mostrando all'utente un messaggio di avanzamento, con la possibilità di accedere direttamente al nuovo round.

Questo passaggio è gestito dalla funzione **proceedToNextChallenge(),** che si occupa di caricare la sfida successiva e di rimuovere eventuali dati temporanei, come il contenuto dell'editor di codice, in modo da fornire un ambiente di lavoro pulito per la nuova prova.

Quando il giocatore completa l'ultima sfida della scalata, il sistema esegue le operazioni necessarie per registrare l'esito finale attraverso la funzione **completeScalata(totalScore)** che viene invocata per chiudere ufficialmente la

scalata, aggiornando i dati di completamento e mostrando un riepilogo con il punteggio totale accumulato.

In questa fase, tutti i dati locali vengono eliminati e l'utente viene reindirizzato alla classifica generale, dove può confrontare il proprio punteggio con quello degli altri giocatori.

L'ultima funzione è **closeScalata(scalataId, isWin, finalScore)** ha il compito di chiudere una scalata e inviare i dati finali al server. Per farlo, crea un oggetto contenente lo stato di completamento della scalata, il punteggio finale e il timestamp della chiusura. Questo oggetto viene poi inviato tramite una richiesta HTTP di tipo **PUT** all'endpoint **/scalates/{scalataId}**. L'obiettivo principale è aggiornare il database con le informazioni finali della scalata.

L'intero sistema è stato progettato per garantire una transizione fluida tra le sfide, evitando interruzioni brusche e assicurando che ogni azione dell'utente abbia un effetto chiaro e immediato.

L'utilizzo del localStorage permette di mantenere le informazioni chiave senza dover appoggiarsi esclusivamente alla sessione corrente, mentre l'aggiornamento costante delle variabili di stato assicura che il flusso di gioco rimanga coerente in ogni momento.

Nonostante le problematiche riscontrate, questa implementazione ha permesso di rendere la modalità Scalata operativa e testabile, fornendo una base su cui poter lavorare per migliorarne l'architettura in una fase successiva.

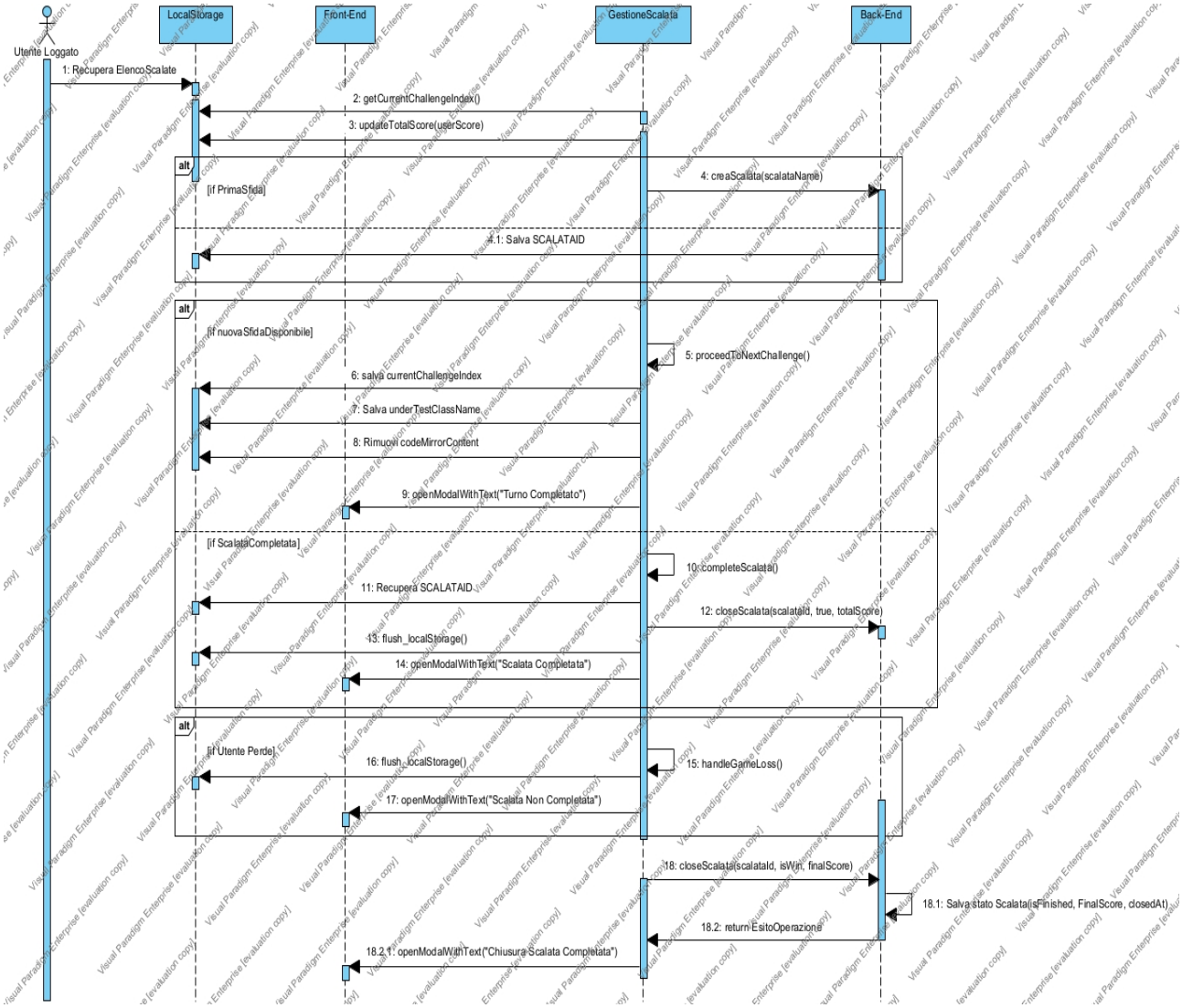


Figura 2: Sequence Diagram Gest_Scalata.

2.4 Struttura Java per la modalità Scalata

Per gestire la modalità Scalata all'interno dell'architettura del gioco, è stata sviluppata una classe Java dedicata, **ScalataGame,** che eredita dalla classe GameLogic metodi e proprietà.

L'obiettivo di questa implementazione è di fornire una gestione centralizzata della scalata, sfruttando le strutture back-end esistenti e garantendo una maggiore coerenza nei dati e nelle regole di gioco.

2.4.1 Struttura e Funzionamento

La classe ScalataGame è progettata per gestire il flusso di gioco della scalata, tenendo traccia dello stato della partita e determinando l'esito di ciascun round. Al momento dell'istanza della classe, il costruttore riceve vari parametri fondamentali, tra cui l'identificativo del giocatore (**playerID**), la classe a testare (**classeUT**), il tipo di robot con cui competere (**typeRobot**), il livello di difficoltà (**difficulty**) e la modalità di gioco (**mode**).

La gestione di questi parametri è affidata alla classe base GameLogic, da cui ScalataGame estende le sue funzionalità.

Uno degli elementi chiave dell'implementazione è la gestione dello stato della partita attraverso tre variabili fondamentali:

- **youlose**, un flag booleano che indica se il giocatore ha perso la scalata.
- **tot_score**, che rappresenta il punteggio totale accumulato dal giocatore durante la scalata.
- **currentRound**, che tiene traccia del numero di round giocati.

Verranno mostrati in seguito i sequence diagram dei metodi principali della classe ScalataGame:

- **playTurn(int userScore, int robotScore)**:
  Questo metodo rappresenta l'esecuzione di un turno all'interno della scalata. Ogni volta che viene chiamato:
  - Incrementa il contatore dei round con currentRound++, avanzando alla fase successiva della scalata.
  - Confronta il punteggio del giocatore (**userScore**) con quello del robot (***robotScore***).
    - Se il punteggio del robot è superiore a quello del giocatore (robotScore > userScore), la variabile youlose viene impostata a true, indicando che il giocatore ha perso la partita.
    - Altrimenti, youlose viene impostata a false, permettendo al giocatore di continuare la scalata.

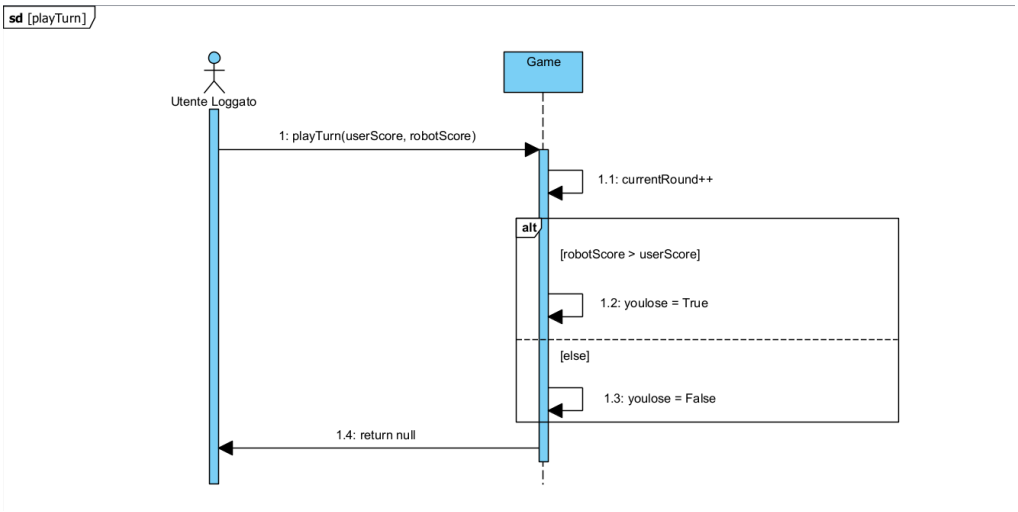


Figura 3.1 Sequence Diagram del metodo playTurn().

- **IsGameEnd():**

  Questo metodo determina se la partita è terminata:

  - Restituisce il valore della variabile youlose:
    - Se youlose è True, significa che il giocatore ha perso la sfida e la partita termina.
    - Se youlose è False, il gioco continua ed il giocatore affronta la prossima sfida della scalata.

- **GetScore(int coverage):**

  Questo metodo calcola ed aggiorna il punteggio totale del giocare:

  - Aggiunge il valore di coverage al punteggio totale tot_score, con tot_score = tot_score + coverage.
  - Restituisce il nuovo valore di tot_score, permettendo al sistema di tenere traccia del punteggio cumulativo attraverso tutti i turni.

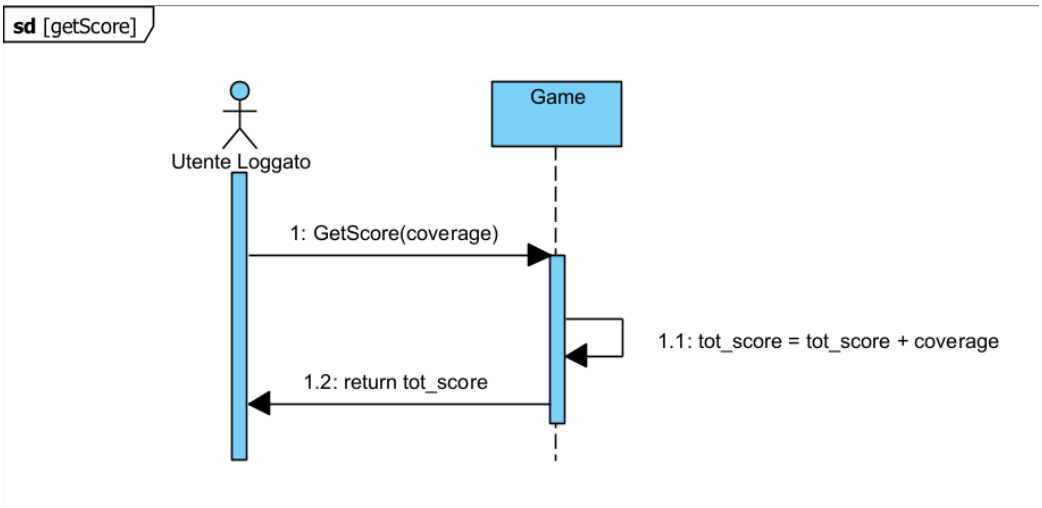


Figura 2.2 Sequence Diagram del metodo getScore().

2.5 Azioni Registrate

Nella classe **T1Service** sono stati implementati due azioni fondamentali per la gestione della modalità Scalata, con lo scopo di interagire con il back-end e ottenere le informazioni necessarie per avviare e gestire il gioco:

1) La prima azione, denominata **getScalate**, è stata progettata per recuperare l'elenco completo delle scalate disponibili nel sistema. Per farlo, effettua una richiesta di tipo **GET** all'endpoint **/scalate_list** e restituisce una lista di oggetti di tipo Scalata, contenente tutte le scalate registrate.
   Questo meccanismo permette di ottenere dinamicamente i dati necessari senza la necessità di una gestione statica lato front-end.
2) La seconda azione, **retrieveScalata** è stata sviluppato con lo scopo di recuperare le informazioni relative ad una scalata specifica, a partire dal suo identificativo. Per ottenere questi dati, viene effettuata una richiesta **GET** all'endpoint **/retrieve_scalata/{id},** che restituisce un oggetto Scalata contenente tutti i dettagli della scalata selezionata. Se la scalata esiste nel sistema, l'oggetto viene restituito correttamente, altrimenti il servizio ritorna un valore nullo.
   È importante sottolineare che i due servizi sono stati integrati seguendo la logica esistente, garantendo così coerenza e uniformità con il resto del codice.

2.6 Funzionamento dettagliato GUIController

Il metodo **gamemodePage** è responsabile della gestione della pagina di selezione della modalità di gioco, accessibile tramite il percorso **/gamemode**.

In particolare, viene utilizzato per caricare le informazioni necessarie in base alla modalità di gioco scelta dall'utente, che può essere **Sfida**, **Allenamento** o **Scalata**. Il parametro **mode** viene passato come parametro della richiesta, e viene utilizzato per determinare quale logica applicare.

Il flusso di lavoro per la modalità scalata inizia quando l'utente seleziona questa modalità attraverso il parametro mode nella richiesta.

Il codice verifica prima se la modalità è effettivamente impostata su **Scalata**. Se questo è il caso, il sistema avvia la logica specifica per questa modalità, utilizzando un oggetto **PageBuilder** che consente agli utenti di creare e personalizzare dinamicamente le loro pagine web.

L'aspetto distintivo della modalità **Scalata** è l'interazione con il back-end per recuperare e visualizzare le informazioni relative alle scalate disponibili.

Dopo aver inizializzato la pagina, viene invocato il servizio **getScalate** tramite un component **ServiceObjectComponent**, il quale esegue una chiamata al microservizio **T1** per ottenere una lista di tutte le scalate disponibili nel sistema. La risposta a questa chiamata è un oggetto contenente una lista di scalate, che viene gestita per verificare la sua validità. Se la lista non è vuota, il sistema prosegue con l'estrazione dei dettagli di ciascuna scalata.

Per ogni scalata presente nella lista, il codice invoca un secondo servizio, denominato **scalateList**, per recuperare i dettagli completi di ciascuna scalata. Questo servizio prende come parametro il nome della scalata specifica, e restituisce un oggetto di tipo Scalata che contiene tutte le informazioni rilevanti, come il nome, la descrizione e altre proprietà legate alla scalata.

Una volta ottenuti i dettagli per ogni scalata, questi vengono aggiunti a una lista separata, chiamata **dettagliScalate**, che raccoglie tutte le informazioni necessarie per la visualizzazione finale.

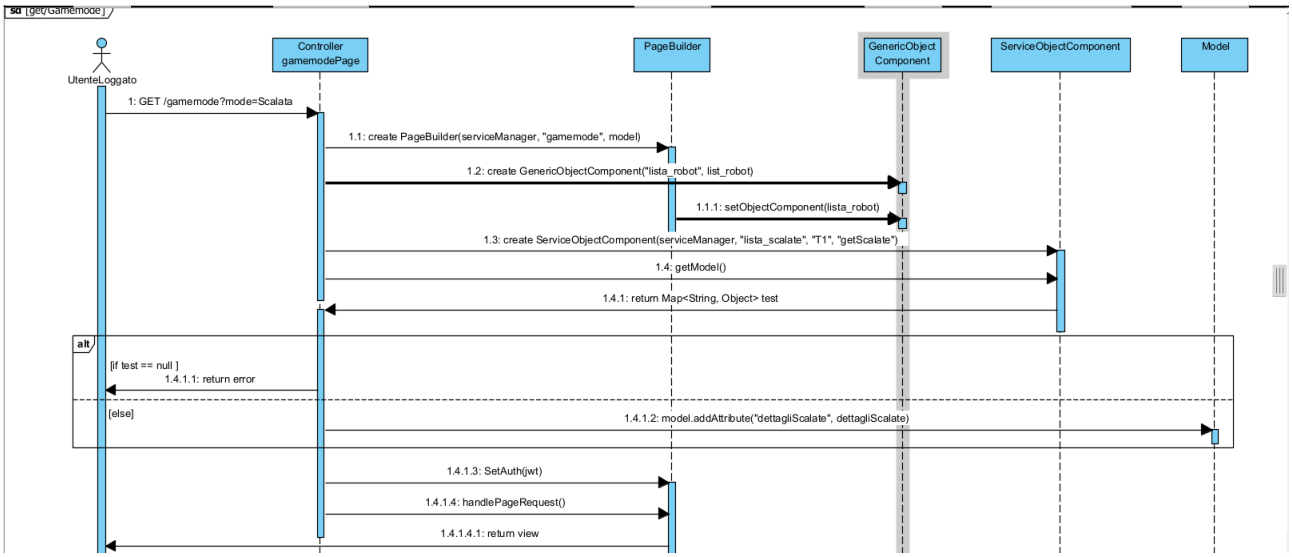
Se i dettagli per una scalata non sono disponibili o se si verificano errori durante il recupero dei dati, il sistema stampa un messaggio di errore, ma non interrompe il flusso di esecuzione.

Una volta che tutte le scalate sono state processate, la lista dettagliScalate viene passata al modello della pagina tramite **model.addAttribute(),** consentendo di visualizzare le informazioni sulle scalate nell'interfaccia utente.

Questa logica consente agli utenti di visualizzare un elenco di scalate disponibili, con informazioni dettagliate su ciascuna, rendendo la modalità scalata interattiva e dinamica.

La gestione dei dati è centralizzata nel back-end, che fornisce le informazioni necessarie tramite le chiamate ai servizi, e viene visualizzata dinamicamente sulla pagina web, offrendo una buona esperienza utente.

Ecco di seguito mostrato il sequence relativo alle modifiche implementate.



2.7 Selezione della Modalità e Configurazione della Pagina

La gestione della modalità Scalata si articola in diverse fasi, dalla selezione iniziale alla memorizzazione delle informazioni necessarie per il gioco.

Ogni fase è progettata per gestire specificamente il flusso di gioco della scalata, in modo che l'utente possa selezionare la modalità, scegliere la scalata desiderata e passare alla sfida vera e propria.

Quando l'utente accede alla pagina, la modalità di gioco viene determinata tramite un parametro presente nell'URL. La funzione **GetMode()** esamina questo parametro e identifica se la modalità selezionata è Scalata. Se la modalità è Scalata, il codice si occupa di aggiornare l'interfaccia utente in modo che siano visibili solo gli elementi relativi alla scalata. Ad esempio, il selettore delle classi viene nascosto, poiché in questa modalità non è necessario scegliere una

classe specifica, mentre il selettore per le scalate diventa visibile, permettendo all'utente di scegliere tra le diverse opzioni di scalata disponibili.

Una volta che l'utente ha selezionato la modalità Scalata, il codice si occupa di memorizzare le informazioni relative alla selezione. Il codice estrae dalla stringa di selezione della scalata un elenco di classi che saranno utilizzate nel gioco.

La funzione salva nel localStorage l'elenco delle classi (**ElencoScalate**), il nome della scalata, il round corrente ed il punteggio totale.

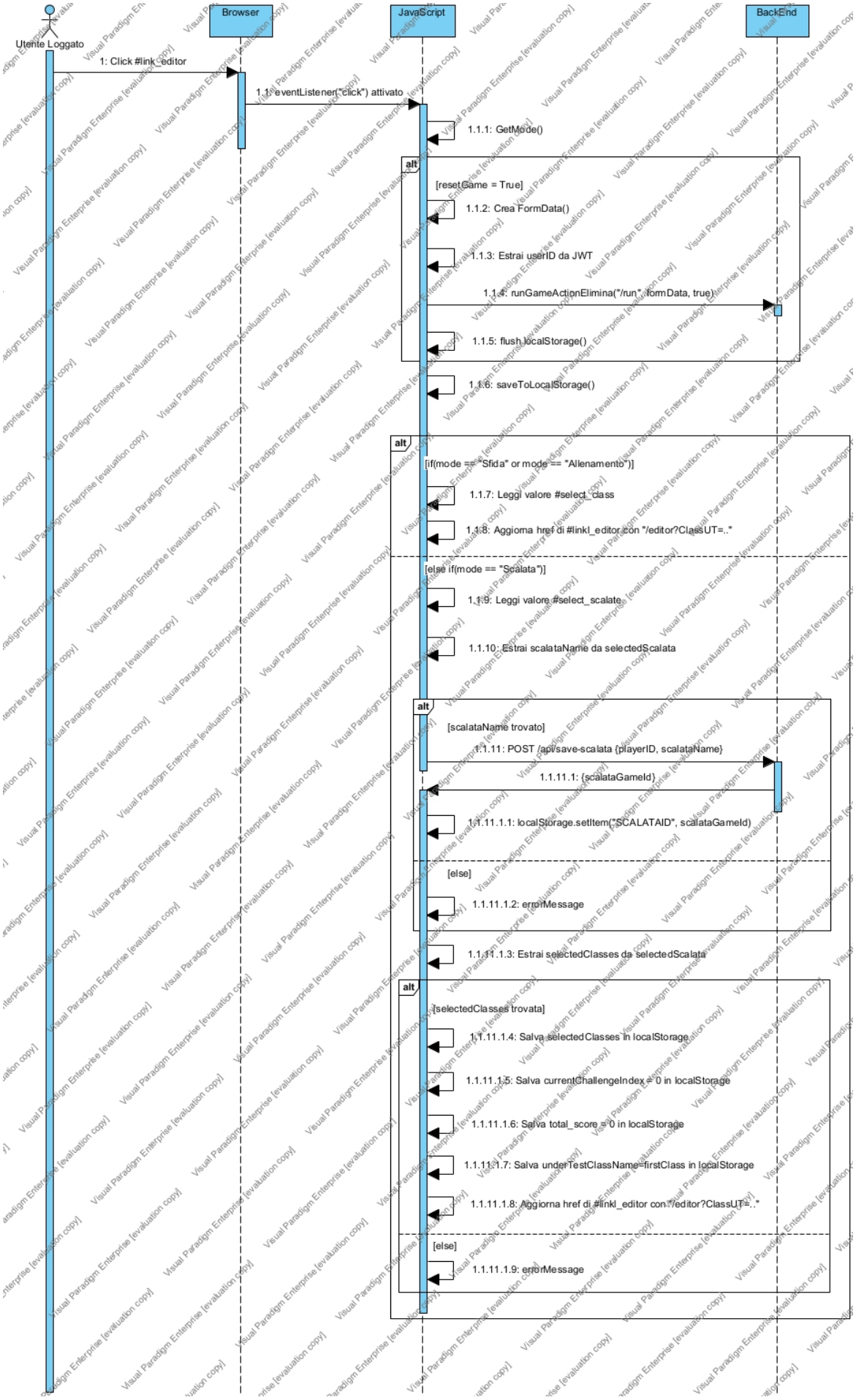
Questo processo di memorizzazione garantisce la persistenza dei dati tra le sessioni, consentendo all'utente di riprendere il gioco in qualsiasi momento, gestire i diversi round della scalata e proseguire senza dover reimpostare i parametri.

Infine, il codice esegue una richiesta al back-end per salvare le informazioni relative alla scalata selezionata.

Oltre alla gestione della selezione della scalata, il codice si occupa anche di abilitare o disabilitare alcuni elementi dell'interfaccia utente in base alla modalità di gioco. Ad esempio, nella modalità Scalata, il pulsante di invio (associato al link per entrare nell'editor) viene abilitato solo se tutte le informazioni necessarie sono state selezionate, come la scalata, il robot e la difficoltà.

Questo garantisce che l'utente non possa procedere senza aver completato tutte le scelte obbligatorie.

Il sequence diagram in basso mostra in modo dettagliato la sequenza di azioni che vengono effettuate quando si seleziona la modalità scalata.


Utente Loggato
Browser
JavaScript
BackEnd
1: Click #link_editor
1.1: eventListener("click") attivato
1.1.1: GetMode()
alt
[resetGame = True]
1.1.2: Crea FormData()
1.1.3: Estrai userID da JWT
1.1.4: runGameActionElimina("/run", formData, true)
1.1.5: flush localStorage()
1.1.6: saveToLocalStorage()
alt
[if(mode == "Sfida" or mode == "Allenamento")]
1.1.7: Leggi valore #select_class
1.1.8: Aggioma href di #linkI_editor con "/editor?ClassUT=.."
[else if(mode == "Scalata")]
1.1.9: Leggi valore #select_scalate
1.1.10: Estrai scalataName da selectedScalata
alt
[scalataName trovato]
1.1.11: POST /api/save-scalata {playerID, scalataName}
1.1.11.1: {scalataGameId}
1.1.11.1.1: localStorage.setItem("SCALATAID", scalataGameId)
[else]
1.1.11.1.2: errorMessage
1.1.11.1.3: Estrai selectedClasses da selectedScalata
alt
[selectedClasses trovata]
1.1.11.1.4: Salva selectedClasses in localStorage
1.1.11.1.5: Salva currentChallengeIndex = 0 in localStorage
1.1.11.1.6: Salva total_score = 0 in localStorage
1.1.11.1.7: Salva underTestClassName=firstClass in localStorage
1.1.11.1.8: Aggiorna href di #linkI_editor con "/editor?ClassUT=.."
[else]
1.1.11.1.9: errorMessage

2.8 Re-factoring front-end

È stato eseguito un re-factoring della pagina leaderboardScalata per renderla più coerente con il resto del progetto, poiché in precedenza presentava un layout e una struttura completamente diversi.

Grazie a questa revisione, la pagina ora offre un'interfaccia più uniforme, migliorando la chiarezza e l'esperienza utente. Nello specifico, per ogni scalata creata, è ora possibile visualizzare in modo chiaro e organizzato le seguenti informazioni:

- Nome del giocatore (player),
- Punteggio ottenuto durante la scalata,
- Tempo impiegato per completare la sfida,
- Posizione in classifica, per confrontare le proprie prestazioni con quelle degli altri partecipanti

Questo aggiornamento rende la leaderboard più leggibile e in linea con il design e le funzionalità delle altre sezioni dell'applicazione.

Per rendere l'esperienza di gioco più accattivante, è stata inserita una barra di progresso. A questo scopo è stata modifica la funzione **openModalWithText**, che serve ad aprire un modal con un titolo, un corpo ed una serie di pulsanti dinamici, in particolare la funzione:

- Aggiorna il titolo ed il corpo con i valori forniti,
- Gestisce la logica dei pulsanti nel footer,
- Crea una barra di progresso animata: se la modalità "Scalata" è attiva, calcola il progresso dell'utente (in percentuale) ed assegna un colore alla barra in base allo stato di avanzamento dell'utente. L'aggiunta di questo elemento tra una sfida e conferisce un maggiore entusiasmo all'utente, il quale è cosciente dei progressi fatti e anche di quelle che saranno le successive sfide.
- Impedisce la chiusura accidentale del modal.

2.9 Testing Funzionalità

Nel processo di sviluppo software, il testing è una fase cruciale per individuare e correggere eventuali errori. Ha lo scopo di garantire che i requisiti iniziali siano rispettati e che il sistema funzioni correttamente.

Nel nostro caso, i test sono stati condotti per verificare la correttezza di specifiche funzionalità, tra cui:

- Le chiamate API,
- Il corretto svolgimento di tutte le fasi di scalata.

Per l'esecuzione dei test, è stato utilizzato Postman, specificando il metodo, GET e l'URL a cui indirizzare la richiesta.

2.9.1 Test chiamata API getScalate

In questo test viene verificata la chiamata all'endpoint **/scalate_list** utilizzando il metodo GET. L'obiettivo è controllare cosa viene effettivamente restituito, ovvero una lista di oggetti di tipo scalata, che rappresentano tutte le scalate disponibili nel sistema.

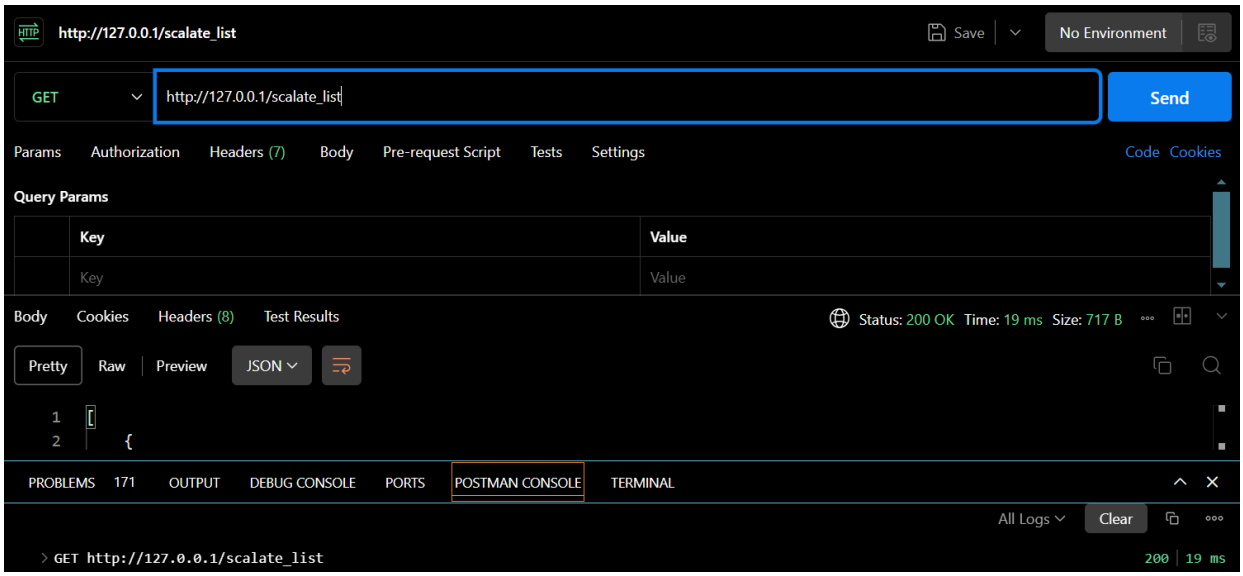


Figura 1: Creazione chiamata test per endpoint /scalate_list.

Il risultato restituito mostra i parametri necessari per ogni scalata, tra cui:

- Il **nome** della scalata,
- L'**autore**,
- La **descrizione** fornita in fase di creazione,

- Il **numero di round**,
- Le **classi** associate alla scalata.

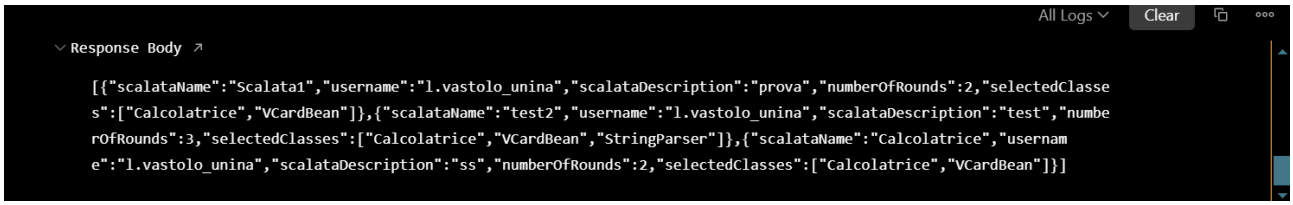


Figura 2: Risultato chiamata endpoint /scalate_list.

2.9.2 Test chiamata API retrieveScalata

Per verificare il corretto funzionamento del recupero delle informazioni di una scalata specifica, è stato testato l'endpoint **/retrieve_scalata/{id}** (nel test è stato usato come id “Scalata1”).

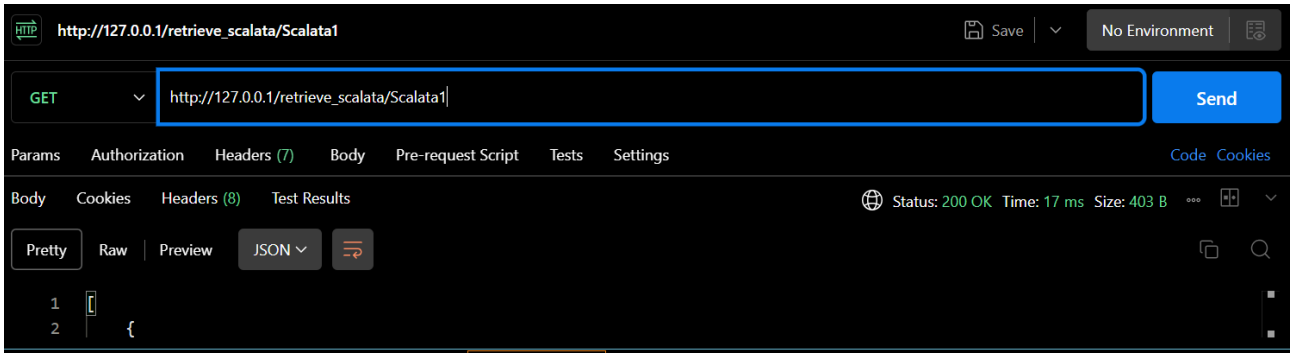


Figura 3: Creazione chiamata test per endpoint /retrieve_scalata/{id}

L'azione **retrieveScalata** è stata sviluppata per restituire i dettagli di una scalata a partire dal suo identificativo univoco (**ID**). Utilizzando una richiesta **GET**, il sistema dovrebbe restituire un oggetto **Scalata**, contenente tutte le informazioni associate, come:

- Il **nome** della scalata,
- L’**autore**,
- La **descrizione**,
- Il **numero di round**,
- Le **classi** della scalata.

L'obiettivo è verificare che la risposta contenga correttamente i dettagli della scalata richiesta e che il sistema gestisca eventuali errori, come ID inesistenti o non validi.

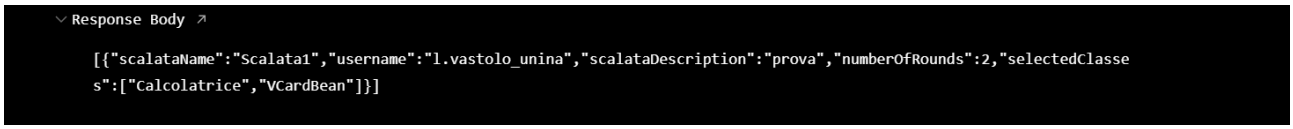


Figura 4: Risultata chiamata endpoint /retrieve_scalata/Scalata1.

2.9.3 Test salvataggio partita in corso

Questo test verifica la capacità del back-end di recuperare i dati dell'ultima partita in corso in caso di interruzione del gioco. Viene analizzato il comportamento del sistema nel ripristinare lo stato della partita precedente dopo un'interruzione. Inoltre, il test verifica se il sistema salva correttamente i caratteri inseriti nell'editor di testo durante un'interruzione.

Clicca per accedere al video dimostrativo.

L'immagine mostra le chiamate che vengono fatte al back-end per recuperare i dati della partita in corso.

Figura 5: chiamate back-end per riprendere i dati della partita in corso.

2.9.4 Test Finale

Il test finale è stato eseguito per verificare il corretto funzionamento della modalità scalata nella sua totalità. Durante il test, viene mostrato un video di un'intera scalata, consentendo di valutare ogni fase del processo e assicurarsi che il sistema operi senza problemi. Questo permette di analizzare il comportamento del software in condizioni reali, verificando la fluidità dell'esperienza, la corretta registrazione dei dati e l'eventuale presenza di anomalie. L'obiettivo è garantire che la modalità scalata funzioni in modo ottimale e senza interruzioni.

Clicca qui accedere al video dimostrativo pt.2.

Capitolo 3: Sviluppi Futuri

A seguito delle attività svolte all'interno del progetto, sono emerse alcune criticità che necessitano di un intervento strutturato per migliorare la stabilità, l'usabilità e la manutenibilità dell'applicazione. In particolare, si rende necessaria una revisione approfondita di alcune componenti chiave per garantire un'integrazione più efficiente tra i vari servizi e una gestione più fluida delle operazioni lato amministratore.

3.1 Re-factoring del back-end per la gestione della scalata

Uno degli interventi più critici e impegnativi riguarda il re-factoring del back-end per la modalità Scalata. Inizialmente, la Scalata era l'unica modalità rimasta separata dalle altre a causa della sua complessità e della difficoltà di unificarla con il resto del sistema. Il nostro lavoro ha permesso di integrare completamente questa modalità con le altre, superando le difficoltà architetturali che in precedenza ne impedivano l'uniformità.

Tuttavia, nonostante l'enorme lavoro svolto, non abbiamo potuto sfruttare appieno il back-end esistente per completare il lavoro, in quanto l'implementazione attuale risulta totalmente disorganizzata e non compatibile con la nuova logica di gestione che abbiamo sviluppato.

L'obiettivo del re-factoring è garantire che il back-end gestisca in maniera centralizzata e strutturata tutte le operazioni relative alla Scalata, eliminando logiche ridondanti e fornendo dati essenziali in modo coerente al front-end.

Questo intervento prevede una riprogettazione completa della struttura esistente, ridefinendo le API e i meccanismi di comunicazione tra i microservizi. Sarà necessario un lavoro approfondito sull'architettura dati, garantendo che le transizioni tra le sfide avvengano in modo fluido, senza incongruenze e con una gestione dello stato chiara e affidabile.

Abbiamo già definito una base solida per questa revisione, ma sarà essenziale rivedere l'interconnessione tra i vari componenti per assicurare un'integrazione efficiente. La correzione degli errori presenti e l'allineamento con la nuova logica di gestione rappresentano passaggi fondamentali per garantire la piena funzionalità della Scalata e una migliore esperienza utente.

3.2 Miglioramento della gestione della scalata lato admin

Parallelamente al re-factoring del back-end, è indispensabile un'ottimizzazione della gestione della Scalata all'interno del pannello amministrativo. Alcuni problemi emersi riguardano:

- **Gestione delle Scalate**: Attualmente, l'aggiunta di una nuova Scalata richiede un aggiornamento manuale della pagina per essere visualizzata correttamente. Questo comportamento deve essere corretto attraverso una gestione più efficiente dello stato nel front-end, evitando ricaricamenti superflui e garantendo una reattività più immediata.
- **Selezione delle Classi**: L'attuale processo di selezione delle classi risulta poco intuitivo e macchinoso. Sarà necessario riprogettare l'interfaccia utente per migliorare l'esperienza dell'utente amministratore, adottando soluzioni più ergonomiche come liste filtrabili, suggerimenti automatici e una visualizzazione più chiara delle classi disponibili.
- **Coerenza Grafica**: La grafica della sezione Scalata non è perfettamente allineata con il resto dell'applicazione, creando una discontinuità visiva che potrebbe impattare negativamente sull'esperienza utente. È opportuno uniformare lo stile, adottando lo stesso design system utilizzato per le altre sezioni dell'applicazione.

3.3 Miglioramento esperienza di gioco

La gamification è sempre più diffusa nel mondo del game design per migliorare l'esperienza di gioco e renderla più accattivante.

Questo approccio consiste nell'inserire elementi tipici del gioco, come premi, livelli, missioni e sfide, per coinvolgere maggiormente i giocatori e aumentare il loro livello di immersione e divertimento.

Uno degli obiettivi principali della gamification è incrementare il senso di gratificazione e progressione nel giocatore. Tecniche come il sistema di punti, badge e classifiche incentivano la competizione e spingono l'utente a migliorare continuamente le proprie prestazioni. Inoltre, la progressione a livelli e l'introduzione di ricompense inaspettate mantengono alta la motivazione, evitando che l'interesse per il gioco diminuisca nel tempo.

Offrire la possibilità di scegliere il proprio percorso, sbloccare contenuti esclusivi o ottenere vantaggi basati sulle proprie scelte crea un senso di controllo e autonomia che aumenta il coinvolgimento.

La gamification favorisce l'interazione sociale, infatti la modalità multiplayer, le sfide tra amici e sistemi di ricompensa condivisi incentivano la collaborazione e il senso di comunità all'interno del gioco.

È possibile quindi introdurre alla fine di ogni sfida, in caso di vittoria, ad esempio, un messaggio di congratulazioni, animazioni e/o immagini che aumentano il grado di soddisfazione dell'utente a fine partita e in caso di sconfitta un messaggio di incoraggiamento che spinge l'utente a riprovarci.

Anche nel caso della scalata potrebbe servire a questo scopo, introdurre dei messaggi tra una sfida e l'altra che ricordano all'utente le sfide restanti e magari anche dei messaggi e/o animazioni per aumentare l'entusiasmo nel gioco. Inoltre si potrebbe implementare una logica che prevede l'aumento della difficoltà in cui il robot migliora progressivamente le proprie performance, il giocatore quindi, è sfidato ad adattarsi e migliorare di conseguenza ed introdurre delle logiche di bonus e potenziamenti, ottenute dal giocatore se supera determinate soglie di punteggio o completa obiettivi specifici.

L'implementazione di questi miglioramenti rappresenterà un passo fondamentale per rendere user-friendly l'applicazione, garantendo un'esperienza di gioco e gestione più fluida ed efficace.