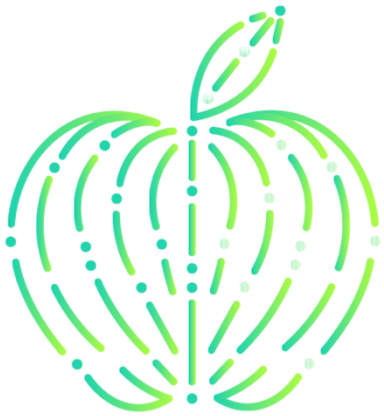
UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II
CORSO DI SOFTWARE ARCHITECTURE DESIGN
PROF. A.R. FASOLINO

ANNO ACCADEMICO 2024 - 25



TASK R4

PUNTEGGI
COPERTURE
FEEDBACK

Studenti

Emanuele Barbato M63001822
Gianluca Fabbo M63001812
Alessandro Cajafa M63001804

INDICE

0 MEMBRI DEL TEAM E REPOSITORY GITHUB 1

1 PUNTO DI PARTENZA E OBIETTIVI DEL PROGETTO 1

2 ANALISI DEI REQUISITI 3

2.1 Requisiti Funzionali 3
2.2 Requisiti Non Funzionali 3
2.3 Diagramma dei Casi d'Uso 3
2.4 Scenari 4
2.5 Storie Utente 5
2.6 Diagramma di Attività 5
2.7 Diagramma di Sequenza 6

3 ANALISI DI IMPATTO 7

4 PROGETTAZIONE E IMPLEMENTAZIONE DELLA SOLUZIONE 9

4.1 Organizzazione del lavoro 9
4.2 Requisito Visualizza Copertura Robot 9
4.2.1 Scelte di Progetto 9
4.2.2 Progettazione e Implementazione 10
4.2.3 Test Effettuati – 1 13
4.2.4 Problemi Riscontrati 14
4.2.5 Test Effettuati – 2 16
4.3 Requisito Visualizza Punteggio 17
4.3.1 Scelte di Progetto 17
4.3.2 Progettazione e Implementazione 18
4.3.3 Esempi 22
4.3.4 Test Effettuati 27
4.4 Sequence Diagram 31
4.5 Esempi di utilizzo di Trello 32

5 CONTAINER T10 33

5.1 Caratteristiche Principali 33
5.1.1 pom.xml testCompiler 33
5.2 Class Diagram 34

6 NUOVO REGOLAMENTO PUNTEGGI 36

7 ARCHITETTURA FINALE 38

8 SVILUPPI FUTURI 39

0 Membri del Team e Repository Github

Emanuele Barbato M63001822 – emanuele.barbato2@studenti.unina.it

Gianluca Fabbo M63001812– g.fabbo@studenti.unina.it

Alessandro Cajafa M63001804 – a.cajafa@studenti.unina.it

URL repository **GitHub**: https://github.com/XEmanueleBarbatoX/A13

1 Punto di Partenza e Obiettivi del Progetto

Lo stato iniziale del sistema può essere riassunto nel seguente Deployment Diagram, recuperato da documentazioni disponibili nel repository di progetto.

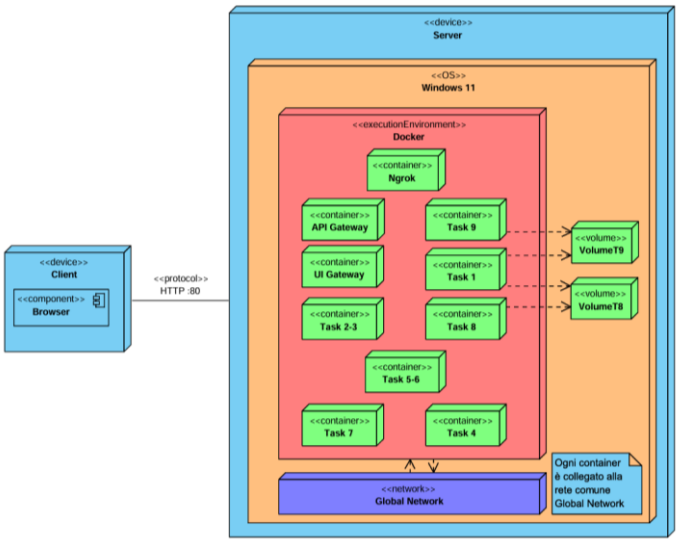


Figura 1 - Deployment Diagram

Si tratta di un sistema distribuito basato su un'architettura containerizzata e supportata da Docker. È composto da diversi container che gestiscono specifiche attività (Task 1-9), con un'interfaccia utente fornita da un **UI Gateway** e un backend centralizzato gestito da un **API Gateway**.

È possibile fornire una vista più dettagliata dei componenti facenti parti dell'applicazione ed evidenziando quelli che saranno oggetto di modifica durante lo svolgimento di questo Task.

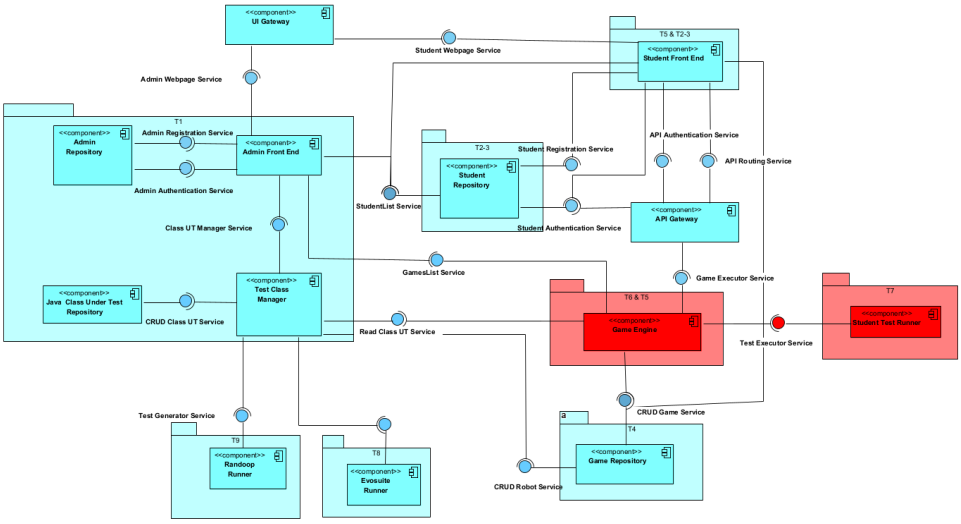


Figura 2: Component Diagram allo stato dell'arte

A livello di modifiche in tal senso, si prevede difatti che siano esclusivamente **T5** e **T7** ad essere impattati poiché la gestione del motore di gioco, coniugato ai servizi di compilazione sono ivi presenti; nondimeno, eventuali modifiche apportate all'interfaccia dell'*editor* od alla finestra *main,* sono da ricercare nel **T6**, che è stato qui accorpato a valle di una recente operazione di re-factoring.

L'obiettivo del lavoro è, infatti, quello di intervenire per riuscire ad implementare i seguenti requisiti, oggetto del Task R4:

1. Studiare e attuare i meccanismi di punteggio assegnati alle partite in base alle varie coperture di Jacoco, complessità / testabilità della classe
2. Migliorare il feedback per lo studente sui test scritti

Per farlo, si prevede di adottare un processo di sviluppo *Agile*, di tipo iterativo–prototipale, volto ad intervenire sul codice con l'intento di rilasciare rapidamente versioni dello strumento funzionanti, che possano essere opportunamente testate per verificare il soddisfacimento dei requisiti richiesti.

In altre parole, si cercherà di risolvere i problemi semplici completamente in fase di programmazione, ricorrendo alla modellazione solo quando strettamente necessario.

In una prima fase iniziale, saranno utilizzati dei modelli di alto livello esclusivamente per inquadrare il problema e le richieste, focalizzando l'attenzione sulle funzionalità che offrono il massimo valore per il cliente e che consentono, complessivamente, di realizzare i requisiti.

A partire da questi modelli preliminari, sarà sviluppata in maniera iterativa, durante le varie fasi di operatività, la documentazione integrale relativa alle diverse funzionalità implementate.

Il framework utilizzato durante lo sviluppo sarà **Scrum** e, pertanto, il lavoro sarà articolato in iterazioni (**Sprint**), della durata di alcune settimane, ciascuna finalizzata al raggiungimento di obiettivi specifici. Al termine di ogni Sprint, verranno effettuate delle **Sprint Review**, durante le quali saranno valutati i progressi complessivi, analizzate eventuali problematiche emerse e pianificate le attività da svolgere nell'iterazione successiva.

Nel corso del lavoro, inoltre, si farà largo utilizzo della piattaforma **Trello**, per organizzare il flusso operativo in modo visuale, sfruttando le bacheche per semplificare il monitoraggio delle attività e delle scadenze.

2 Analisi dei Requisiti

La trattazione può partire con una prima classificazione dei requisiti, come riportato di seguito.

2.1 Requisiti Funzionali

ID	Requisito	Origine (Punto Task)
RF01	L'applicazione deve fornire un meccanismo di punteggio all'utente che tenga conto delle caratteristiche della classe in termini di complessità e testabilità	**1**
RF02	L'applicazione deve offrire all'utente una funzionalità per visualizzare la copertura ottenuta dalla soluzione di test proposta dal robot	**2**

2.2 Requisiti Non Funzionali

ID	Requisito	Origine (Punto Task)
RNF01	La soluzione di test proposta dal robot deve essere presentata attraverso l'evidenziazione delle righe della classe sotto test, la cui visibilità deve poter essere attivata/disattivata dallo studente tramite la pressione di un apposito pulsante	**2**

2.3 Diagramma dei Casi d'Uso

Nel diagramma dei casi d'uso che segue, vengono evidenziati gli scenari specifici sui quali si andrà ad intervenire per implementare i requisiti richiesti.

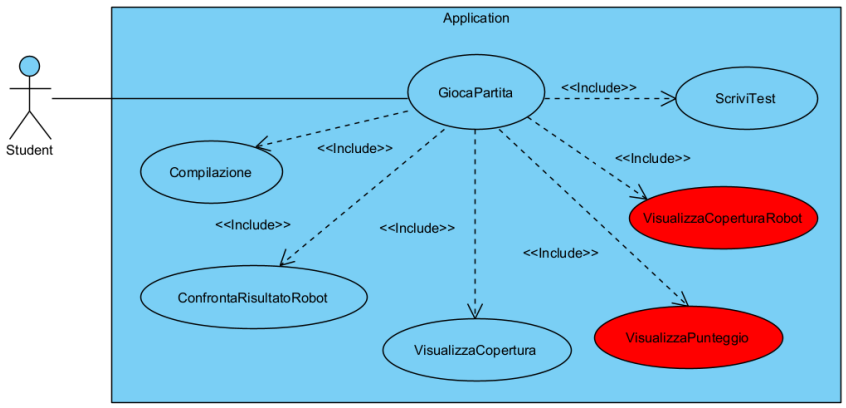


Figura 3 - Use Case Diagram

GiocaPartita è un caso d'uso generale, introdotto al fine di inquadrare il contesto in cui si realizzeranno le modifiche, che è quello in cui lo studente, dopo aver effettuato l'accesso alla piattaforma, decide di avviare una nuova partita.

Include altri casi d'uso, che rappresentano le operazioni complessive che devono essere portate a termine per giocare una partita completa. Tra questi, in rosso sono evidenziati quelli oggetto del task di lavoro in esame.

Caso d'uso	Attori Primari	Attori Secondari	Incl. / Ext.	Requisiti corrispondenti
UC1: VisualizzaPunteggio	-	-	GiocaPartita	RF01
UC2: VisualizzaCoperturaRobot	-	-	GiocaPartita	RF02

In particolare, è possibile osservare che:

1. *VisualizzaPunteggio* è un caso d'uso di **inclusione**, che calcola, attraverso una serie di metriche, il punteggio ottenuto dalla classe di test dello Studente in funzione della copertura e lo mostra a schermo.
2. *VisualizzaCoperturaRobot* è un caso d'uso di **inclusione**, che, al termine della partita, permette allo Studente di visualizzare la copertura ottenuta dal robot.

2.4 Scenari

Caso d'uso:	VisualizzaPunteggio
Attore primario	-
Attore secondario	-
Descrizione	Il sistema calcola e mostra a video il punteggio ottenuto dalla test suite scritta dallo Studente, in termini di linee di codice coperte, tenendo conto anche della complessità e della testabilità della classe
Pre-Condizioni	Il sistema ha eseguito la compilazione della classe di test e non sono occorsi errori
Sequenza di eventi principale	• Il caso d'uso ha inizio quando il sistema calcola la copertura della classe da testare • Il sistema recupera le caratteristiche di complessità e righe di codice della classe sotto test • Il sistema calcola il punteggio • Il sistema restituisce il punteggio calcolato
Post-Condizioni	Il punteggio è disponibile per il salvataggio e la visualizzazione
Casi d'uso correlati	*GiocaPartita*
Sequenza di eventi alternativi	-

Caso d'uso:	VisualizzaCoperturaRobot
Attore primario	Studente
Attore secondario	-
Descrizione	Il sistema mostra a video la copertura ottenuta dal robot
Pre-Condizioni	Il sistema ha confrontato la soluzione dello studente con quella prodotta dal robot e ne ha restituito i risultati
Sequenza di eventi principale	1. Il caso d'uso ha inizio quando lo Studente decide di visualizzare la copertura del robot 2. Il sistema recupera l'esito del test prodotto del robot 3. Il sistema mostra la copertura del robot
Post-Condizioni	-
Casi d'uso correlati	*GiocaPartita*
Sequenza di eventi alternativi	-

2.5 Storie Utente

Sono di seguito riportate le **User Stories**, che descrivono, in modo semplice, le funzionalità e i requisiti di interesse per gli utenti del sistema software, in riferimento al task in oggetto.

Le Storie Utente rappresentano uno strumento comunemente adottato nell'Extreme Programming per esprimere i requisiti.

Titolo Storia Utente 1: Lo Studente visualizza il proprio punteggio

In quanto Studente,
Voglio visualizzare il punteggio che ho ottenuto
Affinché possa determinare la copertura raggiunta.

Criterio di Accettazione:

Supponendo che uno Studente abbia completato un turno,
Quando il sistema valuta la copertura e il numero di partite giocate contro la stessa classe
Allora deve assegnare un punteggio in base ai criteri definiti, includendo la complessità / testabilità della classe.

Titolo Storia Utente 2: Lo Studente visualizza la copertura del robot

In quanto Studente,
Voglio visualizzare la copertura del codice ottenuta dal robot
Affinché possa comprendere come perfezionare la classe di test sviluppata.

Criterio di Accettazione:

Supponendo che uno Studente abbia completato una partita e siano stati prelevati i risultati del robot,
Quando lo Studente accede alla sezione di feedback,
Allora deve essere mostrata la copertura del codice ottenuta dal robot, con una rappresentazione chiara che evidenzi le parti coperte e non coperte dai test.

2.6 Diagramma di Attività

Si riporta di seguito il diagramma di attività che descrive il flusso principale durante una partita di gioco, allo scopo di semplificare la comprensione delle funzionalità chiave e delle interazioni tra gli attori e il sistema.

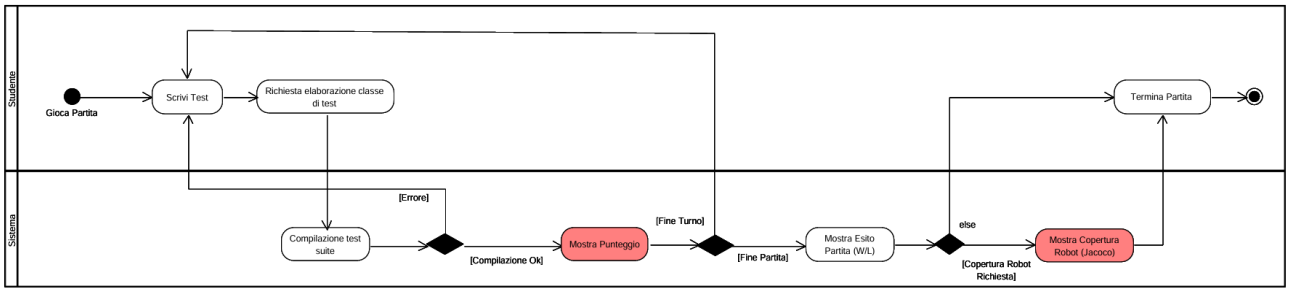


Figura 4 - Activity Diagram Gioca Partita

In rosso sono evidenziate le attività direttamente interessate dalle modifiche necessarie per l'implementazione dei requisiti richiesti, **Mostra Punteggio** e **Mostra Copertura Robot**.

2.7 Diagramma di Sequenza

Segue, come ultimo step della fase di analisi, un diagramma di sequenza di alto livello, utile per rappresentare il flusso di interazioni tra i componenti del sistema in ordine temporale. In altri termini, mostra come e quando vengono scambiati i messaggi per eseguire un processo o una funzionalità specifica.

In questo caso particolare, dato che lo scenario di entrambi i requisiti è pressoché lo stesso (ossia quello in cui l'utente richiede la valutazione dell'esito della partita), un unico diagramma è sufficiente ad evidenziare il flusso di messaggi su cui sarà opportuno intervenire per apportare le modifiche richieste.

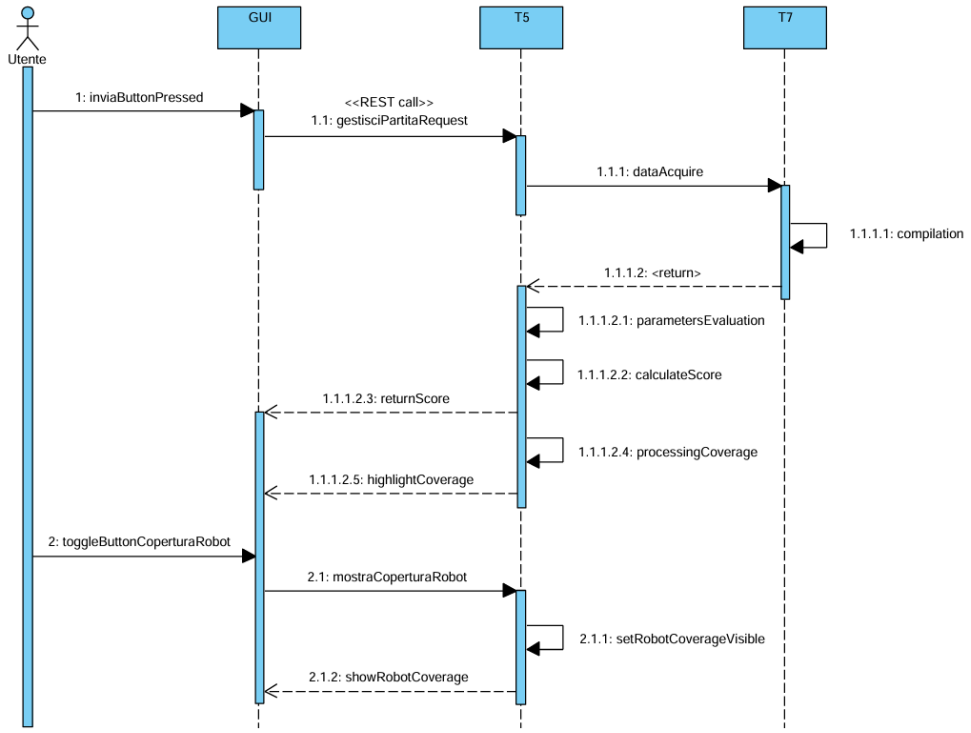


Figura 5 - Sequence Diagram di Alto livello Invia Partita

3 Analisi di Impatto

Per valutare il potenziale impatto delle modifiche sull'architettura del sistema, è stato necessario condurre un approfondito studio della documentazione e del codice esistente.

Allo scopo di agevolare la comprensione di quanto sarà affermato, si riportano dei diagrammi che rappresentano lo stato dei componenti presumibilmente impattati prima dell'intervento di modifica.

La visualizzazione della copertura ottenuta dalla test suite generata dal robot richiede la compilazione delle relative classi di test. Il componente principale del sistema deputato a tale processo è il **T7**, che, nella versione attuale dell'applicativo, si occupa della compilazione delle classi di test scritte dallo studente e della valutazione della relativa copertura della classe sotto test.

La speranza è quella di poter, quantomeno in parte, riutilizzare i servizi già disponibili per richiedere la compilazione e la copertura anche delle classi di test generate dai robot.

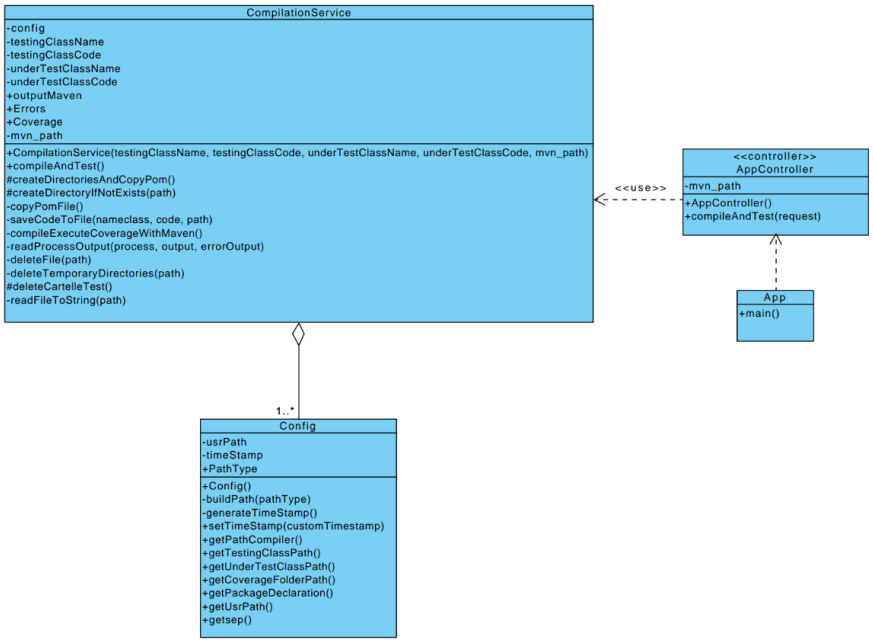


Figura 6 - Class Diagram T7

Pertanto, si può affermare con ragionevole certezza che le modifiche previste impatteranno sul task T7.

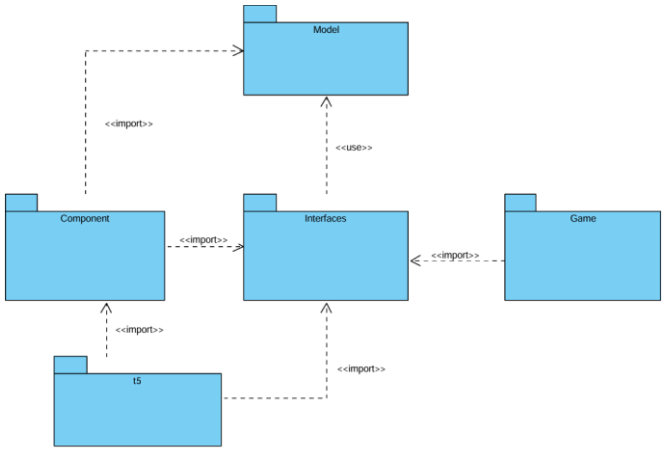


Figura 7 - Package Diagram T5

L'aggiunta del pulsante per la visualizzazione della copertura ottenuta dai robot richiede di apportare delle modifiche all'interfaccia utente, gestita dal componente **T5** attraverso il package **Component**.

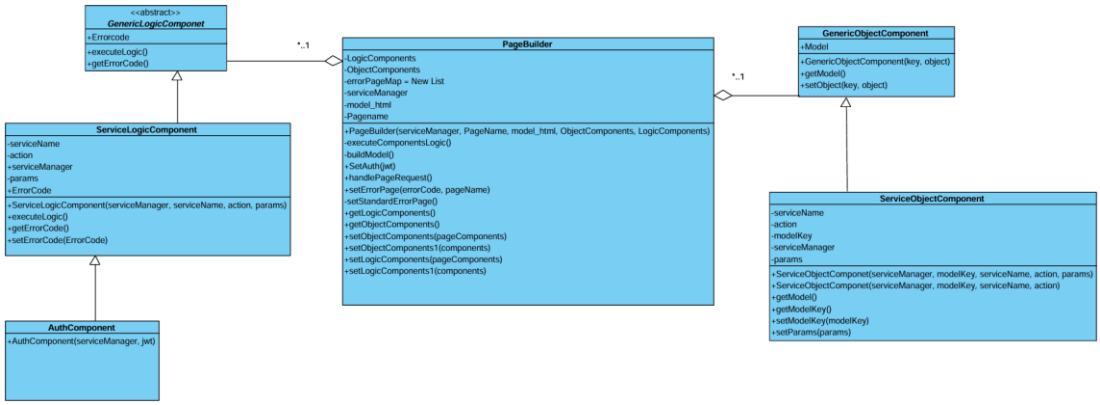


Figura 8 - Class Diagram Package Component T5

Ulteriori modifiche sul **T5** risultano necessarie per adeguare la strategia di calcolo del punteggio, attualmente implementata nella classe *Sfida* del package **Game**.

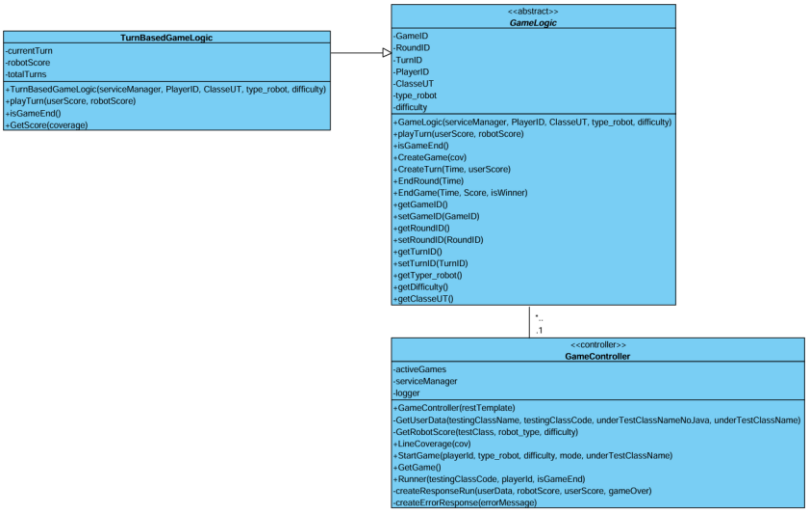


Figura 9 - Class Diagram Package Game T5

Da tali osservazioni consegue che i componenti dell'architettura sui quali sarà sicuramente indispensabile intervenire sono **T5** e **T7**. Tuttavia, è probabile che, durante l'implementazione, possano emergere ulteriori necessità che richiedano interventi su altri componenti. Qualora dovesse essere necessario, le relative motivazioni e scelte saranno adeguatamente documentate.

4 Progettazione e Implementazione della soluzione

4.1 Organizzazione del lavoro

La suddivisione delle attività tra le varie iterazioni è stata definita in base alle priorità delle modifiche. A tal proposito, nella prima fase l'interesse è stato rivolto verso l'implementazione del requisito funzionale 02, ritenuto quello di maggiore urgenza tra i task assegnati.

A titolo esemplificativo, viene illustrata l'organizzazione delle attività all'avvio della prima iterazione (Bacheca Trello).

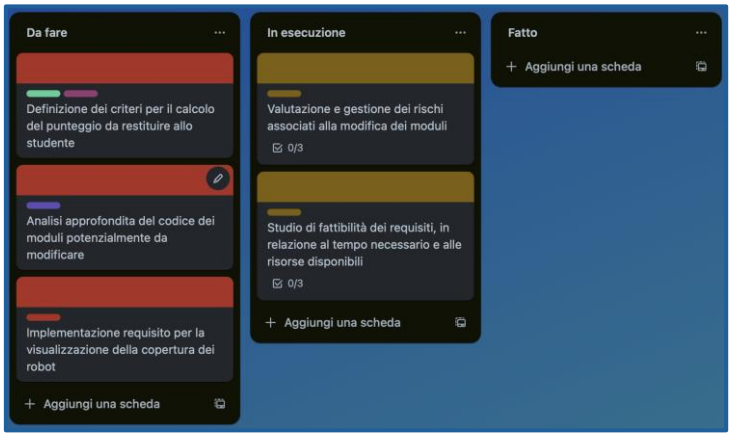


Figura 10 - Bacheca Trello iniziale

In seguito all'esito del primo Sprint, si è reso necessario intervenire per risolvere alcune criticità emerse. Di conseguenza, i membri del team hanno intrapreso attività distinte, lavorando in parallelo.

Senza entrare nei dettagli, che saranno esaminati più approfonditamente in seguito, durante il secondo Sprint una parte del team si è occupata dell'implementazione del requisito funzionale 01 relativo al calcolo del punteggio, mentre un'altra si è concentrata sulla risoluzione di alcune problematiche riguardanti la visualizzazione della copertura del robot.

La terza e ultima iterazione, infine, ha avuto come obiettivo il perfezionamento dei risultati ottenuti e la redazione di una documentazione adeguata.

A questo punto, l'obiettivo è, partendo dalle attività da svolgere, documentare le scelte progettuali adottate per la loro realizzazione. Tale documentazione verrà presentata attraverso una serie di diagrammi che illustrano le decisioni prese. Le scelte progettuali identificate saranno successivamente tradotte nelle relative implementazioni in codice.

4.2 Requisito Visualizza Copertura Robot

4.2.1 Scelte di Progetto

L'idea di base, per la funzionalità di visualizzazione della copertura delle classi di test generate dai robot, era quella di riutilizzare il servizio già messo a disposizione dal T7 per la compilazione delle classi scritte dallo Studente.

Tuttavia, a seguito dell'analisi del codice e delle cartelle contenenti i test dei robot, è emerso che questi, indipendentemente dalla tipologia (Randoop/Evosuite), tendono a generare più di una classe di test. Pertanto, la strategia di compilazione già disponibile si è rivelata inadeguata.

Ciò ha richiesto di intervenire in maniera significativa su entrambi i container T5 e T7, definendo appositi metodi specifici per la compilazione e la copertura dei test generati dai robot.

Jacoco richiede, per la compilazione, che tutte le classi di test siano conservate all'interno della directory al percorso ***.../src/test/java***; così facendo, sarà in grado di compilarle separatamente e di farne il merge per valutare la copertura complessiva della classe da testare.

La configurazione di Maven per la copertura tramite Jacoco avviene attraverso il file **pom.xml**; all'interno di questo sono state aggiunte delle istruzioni particolari che assicurano la compilazione separata delle classi di test e il merge dei relativi dati di copertura, in modo tale che Jacoco sarà in grado di restituire un unico file **jacoco.xml** contenente la copertura complessiva.

Ciò richiede che il **T7** sia in grado di individuare le classi di test generate dai robot e di spostarle, all'atto dell'avvio di Maven, all'interno della directory al percorso ***.../src/test/java***. Per farlo, l'idea è stata quella di aggiungere al controller un nuovo metodo, che gestisca una nuova rotta, la cui chiamata avviene passando, tra gli altri parametri, il path in cui andare a prelevare le classi di test, opportunamente calcolato, nel **T5**, in funzione dei parametri della partita giocata.

Inoltre, per integrare un nuovo pulsante che consenta di mostrare o nascondere la copertura del robot, è stato necessario intervenire sulla componente front-end del **T5**. A tal proposito, è stato concepito un meccanismo che prevede la richiesta di compilazione e copertura delle classi di test generate dai robot al momento in cui lo Studente conclude la partita con *Invia*. Tuttavia, la visualizzazione di tali informazioni è resa disponibile solo su richiesta, tramite la pressione del pulsante appena menzionato, che sarà visualizzato a schermo al termine della procedura.

4.2.2 Progettazione e Implementazione

Il Class Diagram di seguito riportato offre una vista delle varie classi delle principali directories del container T7, focalizzando l'attenzione principalmente sui componenti oggetto di modifiche per la realizzazione del requisito in esame.

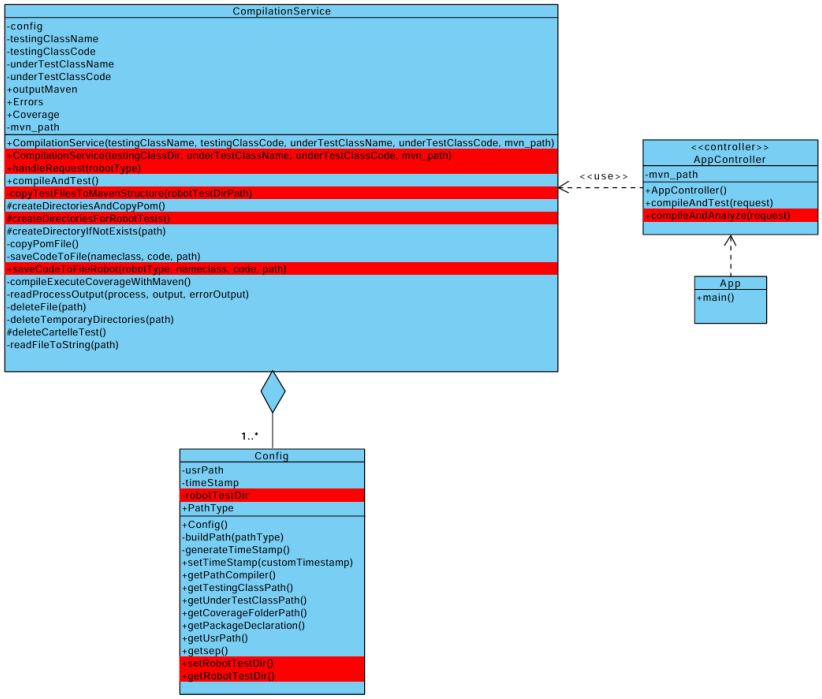


Figura 11 - Class Diagram T7 Post Modifiche

In figura sono rappresentate le classi di interesse della directory ***RemoteCCC,*** dove, in particolare, le modifiche hanno riguardato ***CompilationService***, a cui il ***T7Service*** del T5 richiede, passando attraverso il controller, la compilazione e il calcolo della copertura, e la classe ***Config***.

Per quanto riguarda ***CompilationService***, nel dettaglio:

1) ***CompilationService():*** un costruttore aggiuntivo a quello già esistente, a cui viene passato il path in cui si trova la directory delle classi di test prodotte dal robot.
2) ***handleRequest():*** un nuovo metodo aggiunto per:
    a. la creazione delle varie directories in cui inserire le classi di test prodotte dal bot, tramite invocazione del metodo ***createDirectoriesForRobotTests()***;
    b. la copia delle classi di test nell'apposita struttura Maven prevista, invocando il metodo ***copyTestFilesToMavenStructure()***;
    c. il salvataggio della classe sotto test, sfruttando il metodo ***saveCodeToFileRobot()***;
    d. la compilazione ed esecuzione dei test;
    e. il prelievo del report Jacoco.
3) ***copyTestFilesToMavenStructure***: metodo private invocato da ***handleRequest()***, aggiunto per copiare i file di test scritti dai robot in una struttura compatibile con Maven, per consentire la compilazione dei test e la valutazione della copertura del codice.
4) ***createDirectoriesForRobotTests:*** metodo protected, invocato sempre da ***handleRequest()***, aggiunto per creare e verificare l'esistenza di directory necessarie come quella per la compilazione, per le classi di test generate dal robot, per la classe sotto test e per i report di copertura.
5) ***saveCodeToFileRobot():*** metodo private aggiunto per consentire il salvataggio del codice della classe sotto test, aggiungendo ad esso anche una dichiarazione del package per renderlo visibile alle classi di test prodotte dal bot.

Per quanto riguarda la classe ***Config***, invece, è stato aggiunto:

- ***robotTestDir:*** attributo private di tipo String per memorizzare il path della directory in cui si trovano le classi di test prodotte dal bot. Insieme a questo sono stati aggiunti anche i rispettivi metodi di *Set* e *Get*.

Come già anticipato, l'implementazione della funzionalità di visualizzazione della copertura ottenuta dai robot ha richiesto di intervenire anche sul T5. A tal proposito, i Class Diagram di seguito riportati offrono una vista delle varie classi dei package del container T5, focalizzando l'attenzione principalmente sui componenti oggetto di modifiche per la realizzazione del requisito in esame.

Package Game

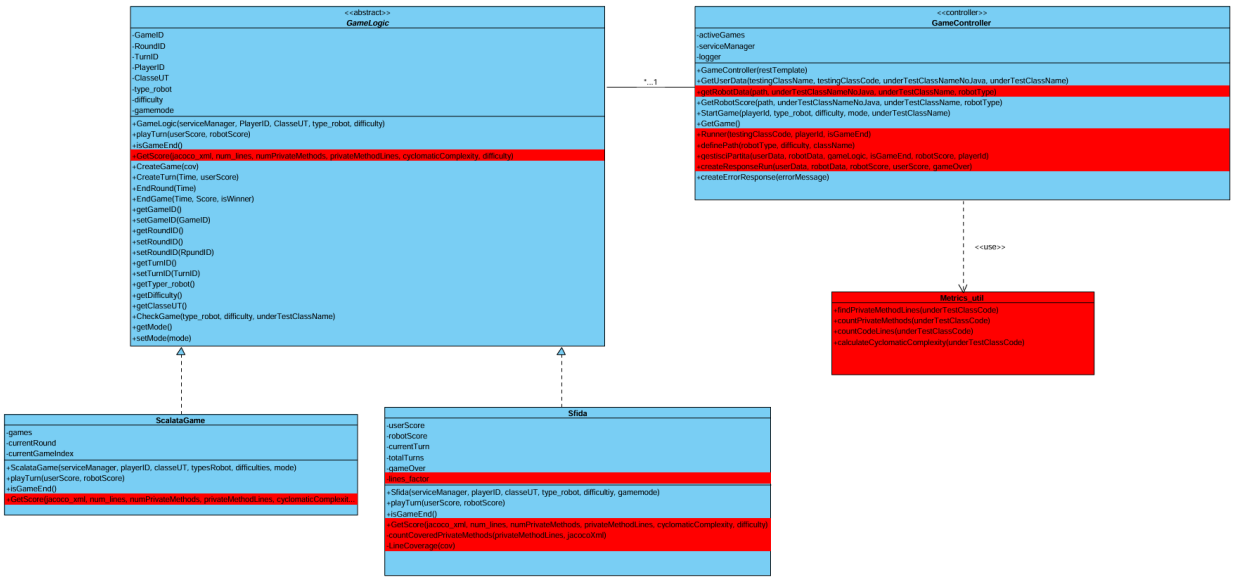


*Figura 12 - Class Diagram Package **Game** T5 post-modifiche*

Significative variazioni sono state apportate, nel package **Game**, alla classe *GameController* in termini di nuove funzionalità aggiunte e modifiche a funzioni preesistenti, per supportare l'implementazione del requisito RF03, che permette all'utente di visualizzare anche la copertura ottenuta dal robot.

Nel dettaglio:

1) ***Runner()***: metodo che corrisponde alla rotta *"/run"* invocata dal metodo ***RunGameAction()*** di ***REST_Editor.js***. È stata aggiunta al suo interno una funzionalità di calcolo dei dati del robot in termini di difficoltà e tipo, e la definizione mediante una nuova apposita funzione privata ***definePath()*** del percorso della classe di test prodotta dal robot. È stata in seguito aggiunta la possibilità di compilare tale classe prodotta con un metodo ***GetRobotData()***, che riceve in ingresso il path recuperato precedentemente e altre informazioni sulla classe under test per effettuare un confronto e produrre un apposito file di coverage.
2) ***gestisciPartita()***: questa funzione privata è stata modificata con l'aggiunta del parametro *robotData* per tenere conto anche della nuova funzionalità.
3) ***definePath():*** un metodo privato richiamato dal metodo ***Runner()*** che, sulla base dei parametri di input di tipo e difficoltà del bot e nome della classe da testare, calcola e restituisce il percorso, sottoforma di stringa, che corrisponde alla locazione della classe di test prodotta dal bot.
4) ***GetRobotData():*** un metodo aggiunto alla classe che comunica con il modulo **T7** per richiedere la compilazione e ottenere la copertura Jacoco della classe di test ottenuta dal bot.
5) ***CreateResponseRun():*** questa funzione è stata modificata con l'inserimento del nuovo parametro *robotData* che viene aggiunto all'oggetto *response* restituito.

Package Interfaces

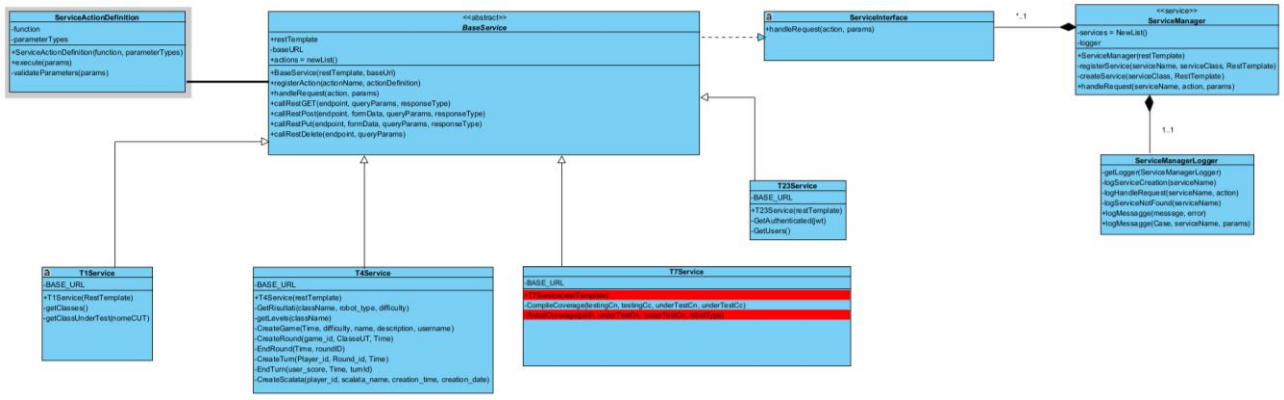


*Figura 13 - Class Diagram Package **Interfaces** T5 post-modifiche*

Per quanto concerne, invece, il package **Interfaces**, le principali variazioni hanno interessato la classe ***T7Service***, con l'obiettivo di abilitare la comunicazione con il container **T7** e richiedere la compilazione delle classi di test generate dai robot.

Le modifiche hanno riguardato, in particolare:

- ***RobotCoverage():*** è stato aggiunto questo nuovo metodo per poter effettuare una chiamata REST al container **T7**. La funzione, nello specifico, invia una richiesta http POST all'endpoint *"/compile-and-analyze"* (endpoint all'interno del T7), costruendo l'oggetto JSON da passare, inserendo al suo interno il percorso della directory in cui si trovano i file di test prodotti dal bot, il nome e il codice della classe da testare e la tipologia di robot. Il metodo, infine, riceve e restituisce un oggetto di tipo *String* come risposta alla POST.

- ***T7Service():*** il costruttore è stato modificato per aggiungere la registrazione dell'azione *RobotCoverage()* precedentemente definita e specificando che tale azione accetta quattro parametri di tipo String.

4.2.3 Test Effettuati – 1

ID	Precond	Input	Expected Output	Output	Esito
Test_Copertura_ Randoop	Il giocatore ha selezionato una classe di test e una difficoltà (FTPFile e facile) il giocatore ha selezionato il bot Randoop; il giocatore ha inviato la classe ; bottone Visualizza CoperturaBot è stato visualizzato correttamente	Bottone Visualizza-CoperturaBot premuto	Copertura grafica prodotta da Jacoco	Copertura grafica prodotta da Jacoco	Pass
Test_Copertura_ Evosuite_I_ Versione	Il giocatore ha selezionato una classe di test e una difficoltà (FTPFile e facile) il giocatore ha selezionato il bot Evosuite; il giocatore ha inviato la classe; bottone Visualizza CoperturaBot è stato visualizzato correttamente	Null	Copertura grafica prodotta da Jacoco	Null	N/A (in quanto la precondizione della visualizzazione del pulsante non è verificata a causa di un'errata compilazione)

4.2.4 Problemi Riscontrati

Nel tentativo di sviluppare la funzionalità per consentire all'utente di visualizzare, al termine della partita, la copertura ottenuta dal robot, la configurazione del file *pom.xml* e l'individuazione del path in cui sono contenute le classi di test, come precedentemente descritto, si sono rivelate adeguate esclusivamente per i test generati dal robot Randoop.

Durante il testing dell'applicazione, avviando una nuova partita contro EvoSuite, all'atto della richiesta di visualizzazione della copertura, è emerso un errore di compilazione, dovuto all'assenza delle dipendenze necessarie a Maven per poter compilare classi di test generate da EvoSuite. L'errore riscontrato è mostrato nella seguente figura.

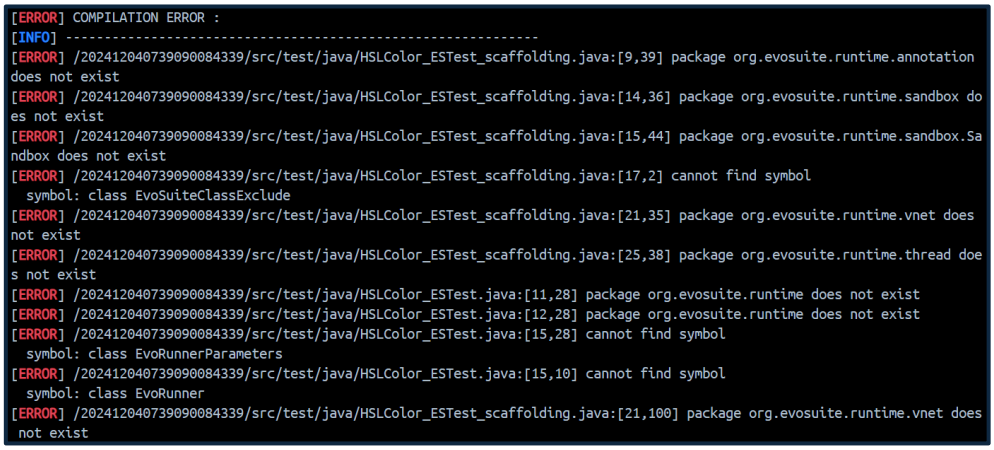


Figura 14 - Errore di compilazione EvoSuite

In particolare, Maven tentava di risolvere le dipendenze cercando la libreria **tools**, non più disponibile nelle versioni di Java più recenti, tra cui **Jdk17**, attualmente in uso nel container T7. Una versione di Jdk, invece, che non soffre di questa mancanza è la 1.8.

Tuttavia, sostituire completamente la versione di Java utilizzata dal container avrebbe potuto causare non pochi problemi di compatibilità, ragion per cui si è deciso di esplorare soluzioni alternative.

Nel tentativo di affrontare il problema, sono state valutate diverse opzioni, basandosi su fonti disponibili online che trattano errori analoghi.

Tra queste, è stato inizialmente sperimentato il **Maven Toolchains Plugin**, un plugin specifico di Maven che consente di configurare e utilizzare una versione di Java diversa da quella predefinita sul sistema, senza dover modificare manualmente la variabile **JAVA_HOME**. Tuttavia, tale soluzione si è rivelata essere inadatta, poiché richiedeva, in ogni caso, che sulla macchina host fosse installata una Jdk compatibile, aspetto che poteva rendere più complesso il processo di installazione dell'applicativo.

La soluzione ritenuta più appropriata, allora, è stata quella di spostare l'attività di compilazione in un container separato, progettato e implementato appositamente per eseguire una versione di Java compatibile, denominato **T10G724 – EvoSuite Compiler**.

La relativa documentazione di dettaglio è consultabile nel capitolo dedicato.

A seguito dell'adozione di questa soluzione, è stato aggiornato il container **T5** per consentire l'invio, in base al tipo di robot selezionato per una specifica partita, delle richieste di compilazione e copertura a container differenti (**T7 – Randoop** / **T10 – EvoSuite**).

In particolare, sono state apportate modifiche significative al package **Interfaces**, con l'introduzione di un nuovo servizio denominato ***T10Service*** e l'aggiornamento del costruttore

di ***ServiceManager***, al fine di abilitare l'invio di richieste REST verso le rotte gestite dal nuovo container. Contestualmente, sono stati adeguati i ***gateway UI*** e ***API***.

Di seguito, si riporta il Class Diagram aggiornato del package **Interfaces**:

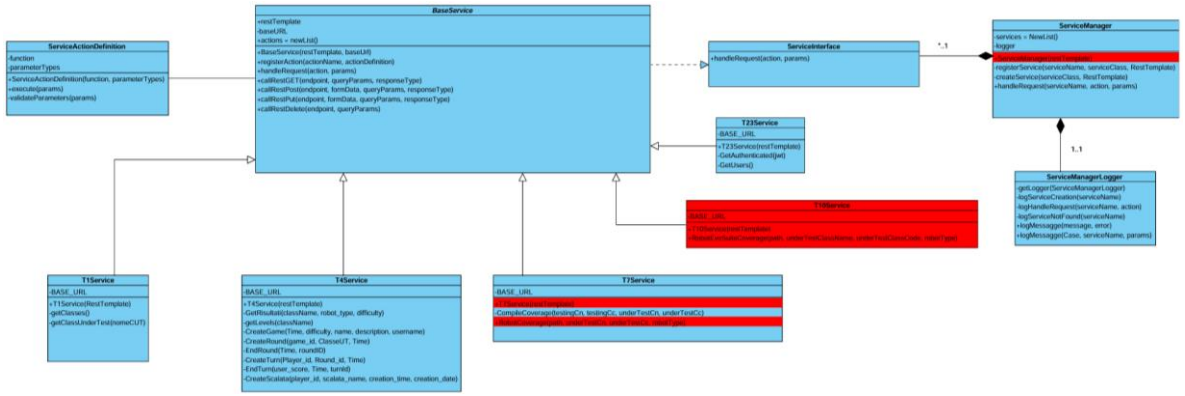


Figura 15 - Class Diagram Package Interfaces T5 Post Modifiche

Lo sviluppo di un nuovo container è sicuramente la strada da percorrere per risolvere i problemi di compilazione. Tuttavia, quest'intervento non è bastato a rendere il sistema perfettamente funzionante.

In particolare, i problemi di compilazione sono stati superati, le dipendenze vengono perfettamente risolte e i test correttamente eseguiti; purtroppo, però, **Jacoco restituisce un file di copertura all'interno del quale la coverage è sempre nulla**, nonostante la validità delle classi di test.

```
[INFO] Tests run: 22, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.425 s - in HSLColorSourceCode.HSLColor_ESTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 22, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
[INFO] --- jacoco-maven-plugin:0.8.8:report (jacoco-report) @ code-coverage ---
[INFO] Loading execution data file /202412101715488468608/target/jacoco.exec
[INFO] Analyzed bundle 'code-coverage' with 1 classes
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ code-coverage ---
```

Figura 16 - Esito Compilazione Maven

```
="245" mi="1" ci="0" mb="0" cb="0"/><counter type="INSTRUCTION" missed="572" c
overed="0"/><counter type="BRANCH" missed="58" covered="0"/><counter type="LIN
E" missed="116" covered="0"/><counter type="COMPLEXITY" missed="52" covered="0
"/><counter type="METHOD" missed="23" covered="0"/><counter type="CLASS" misse
d="1" covered="0"/></sourcefile><counter type="INSTRUCTION" missed="572" cover
ed="0"/><counter type="BRANCH" missed="58" covered="0"/><counter type="LINE" m
issed="116" covered="0"/><counter type="COMPLEXITY" missed="52" covered="0"/><
counter type="METHOD" missed="23" covered="0"/><counter type="CLASS" missed="1
" covered="0"/></package><counter type="INSTRUCTION" missed="572" covered="0"/
><counter type="BRANCH" missed="58" covered="0"/><counter type="LINE" missed="
116" covered="0"/><counter type="COMPLEXITY" missed="52" covered="0"/><counter
 type="METHOD" missed="23" covered="0"/><counter type="CLASS" missed="1" cover
ed="0"/></report>
```

Figura 17 - jacoco.xml

Molti errori dello stesso tipo sono documentati in rete, alcuni dei quali sono reperibili al link https://www.evosuite.org/documentation/measuring-code-coverage/. Sebbene in questa risorsa siano proposte anche delle possibili soluzioni, queste ultime si sono rivelate non efficaci nel caso specifico in esame.

4.2.5 Test Effettuati – 2

ID	Precondizione	Input	Expected Output	Output	Esito
Test_ Copertura_ Evosuite_ II_Versione	Il giocatore ha selezionato una classe di test e una difficoltà (FTPFile e facile) il giocatore ha selezionato il bot Evosuite; il giocatore ha inviato la classe; il bottone Visualizza CoperturaBot è stato visualizzato correttamente	Bottone Visualizza-Copertura-bot premuto	Copertura grafica prodotta da Jacoco	Copertura grafica non corrispondente a quella prodotta da Jacoco (tutte righe rosse)	Fail

4.3 Requisito Visualizza Punteggio

4.3.1 Scelte di Progetto

Il sistema di punteggio è stato totalmente ripensato, in modo tale da rendere l'esperienza di gioco e di apprendimento sempre più competitiva e coinvolgente. L'approccio precedente, con un punteggio massimo fisso, è stato sostituito da un sistema dinamico che si adatta meglio alle abilità del giocatore e alla complessità delle sfide proposte. Di seguito una panoramica sulle migliorie introdotte:

a) **Adattabilità del Punteggio**
   Il punteggio ora è calcolato considerando:
   - La difficoltà del bot avversario.
   - La complessità intrinseca della classe sotto test, basata su parametri come il numero di righe di codice totali e la complessità ciclomatica.
   - L'efficacia dei casi di test del giocatore nel coprire diverse porzioni di codice, inclusi i metodi privati.

b) **Moltiplicatori Dinamici**
   Sono stati introdotti vari moltiplicatori per premiare le prestazioni e bilanciare la sfida:
   - Bonus per una buona copertura del codice al primo tentativo.
   - Penalità per turni aggiuntivi.
   - Fattori basati sulla copertura dei metodi privati, incentivando la scrittura di test più efficaci.
   - Fattori proporzionali alla complessità della sfida, derivati da parametri come la dimensione e la complessità del codice.

c) **Calcolo Meritocratico**
   Per garantire un sistema più equo e trasparente:
   - È stato abbandonato il meccanismo di normalizzazione, favorendo un confronto diretto delle abilità.
   - Il punteggio finale è restituito con maggiore precisione, fino alla seconda cifra decimale, per valorizzare anche le differenze più sottili.

d) **Modularità e Manutenibilità**
   Per agevolare la manutenzione, tutte le funzionalità di calcolo sono state centralizzate in un modulo dedicato. Questo permette di intervenire rapidamente in caso di errori e mantiene leggibile il codice principale.

e) **Premi e Penalizzazioni**
   Il sistema incoraggia i giocatori a migliorare le loro capacità:
   - Premiando la strategia e l'efficienza nel primo tentativo.
   - Penalizzando l'inefficienza attraverso una riduzione graduale del punteggio.

Si è deciso di non introdurre un meccanismo di normalizzazione del risultato per consentire una maggiore granularità e non appiattire il risultato, visti i numerosi fattori che permettono di discriminare tra i punteggi dei giocatori.

Di seguito un diagramma che esprime il punteggio sottoforma di classi:

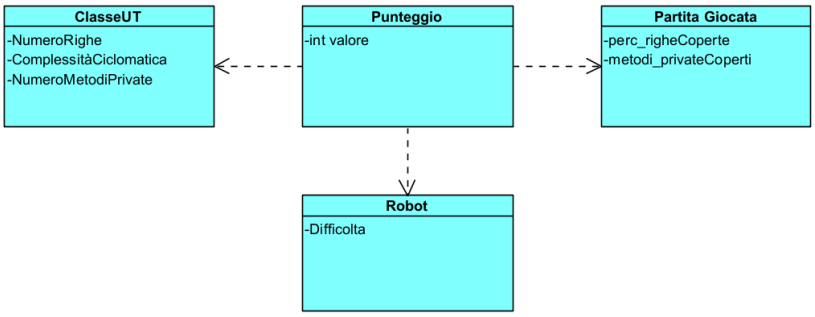


Figura 18 - Diagramma Punteggio

Le metriche del nuovo sistema di punteggio sono state rese accessibili all'utente mediante un'accattivante finestra *pop-up* che compare alla pressione del relativo bottone di ***info*** presente nella schermata di selezione della modalità, in particolar modo nella modalità *Sfida.*

4.3.2 Progettazione e Implementazione

Al fine di definire questo articolato sistema di moltiplicatori e penalizzazioni, sulla base delle considerazioni precedenti, sono state individuate come principali oggetto di modifiche le classi ***GameController*** e ***Sfida*** del **T5**.

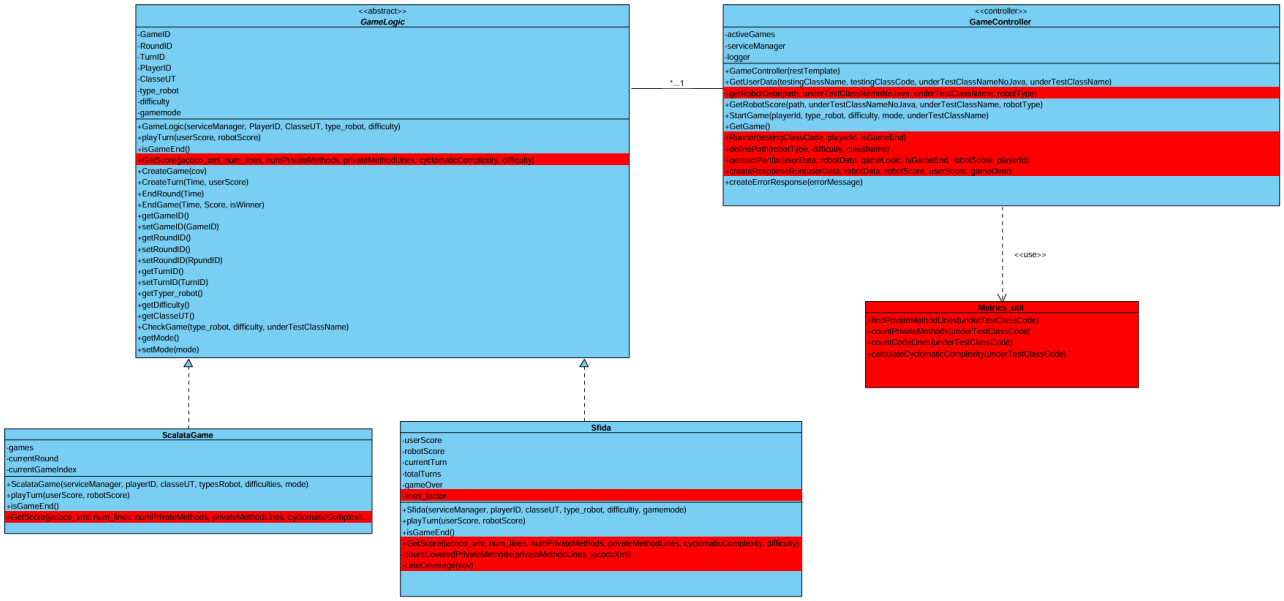


*Figura 19 - Class Diagram Package **Game** T5 Post Modifiche*

È sembrato sin dal primo momento necessario, a tal fine, implementare una serie di funzionalità aggiuntive e pertanto si è deciso di introdurre una nuova classe all'interno del package *Game* del T5, chiamata ***Metrics_util***. Qui, infatti, sono stati definiti tutti e soli i metodi che operano sulla classe da testare e che permettono di ottenere da essa valori utili per la logica del calcolo del punteggio, funzioni richiamate pertanto esclusivamente all'interno della classe *GameControlle*r, lasciando alla classe *Sfida*, il calcolo di parametri dinamici legati alla partita stessa, come il numero di metodi privati coperti o il valore di coverage ottenuto.

Questa classe aggiuntiva è stata introdotta per facilitare scenari di manutenzione ricorrenti, permettendo di avere ben presente un punto preciso in cui intervenire in caso di malfunzionamenti o di logica errata nel calcolo del punteggio e per mantenere una buona leggibilità della funzione ***GetScore()*** della classe *Sfida*, dove viene effettivamente calcolato e restituito il punteggio.

Le principali modifiche, nel package **Game**, sono state apportate a:

- ***gestisciPartita():*** funzione definita in *GameContoller* che richiama al suo interno la funzione *GetScore()* per ottenere il punteggio. Qui viene effettuato, oltre alla logica già presente, il calcolo del numero di metodi privati della classe e della complessità ciclomatica. Per questi calcoli è stato necessario aggiungere varie funzionalità in ***Metrics_util***, quali:
  - ***findPrivateMethodLines***: funzione che dal codice sorgente della classe da testare permette di trovare le righe in cui vi è una dichiarazione di un metodo private;
  - ***calculateCyclomaticComplexity***: funzione che opera sul codice sorgente e permette di calcolarne la complessità cilomatica basandosi sulla teoria di McCabe;
  - ***countPrivateMethods***: funzione che sul codice sorgente della classe da testare permette di trovare il numero di metodi private al suo interno.
  - ***countCodeLines:*** funzione che dal codice sorgente permette di estrarre il numero di righe di codice effettivo

  Questo conteggio dei metodi private e di quelli coperti viene qui ottenuto per non appesantire eccessivamente la funzione GetScore e per mantenerne alta la leggibilità.
- ***getScore():*** questa funzione definita nell'interfaccia *GameLogic* e poi definita sia in *Sfida* che in *Scalata* è stata profondamente rivisitata in quanto è effettivamente in essa che viene calcolato e restituito al *GameController* il punteggio. Sostanziali modifiche hanno riguardato la firma funzione stessa: sono stati infatti aggiunti, oltre al solo parametro relativo alla copertura ottenuta dal test, **numlines**, ossia il numero di righe di codice della classe, **numprivateMethods,** ossia il numero di metodi private nella classe da testare, **privateMethodsLines,** ossia un ArrayList contenente le linee dove presenti i metodi privati, **ciclomaticComplexity**, ossia la complessità ciclomatica della classe**,** ***underTestClassCode*** ossia il codice effettivo della classe sotto test, ***difficulty*** ossia il parametro intero da 1 a 3 che rappresenta la difficoltà del bot.
  Nella implementazione della funzione all'intero di *Sfida* è stato definito il complesso sistema di moltiplicatori e penalizzazione richiamando una serie di funzioni private che permettono di calcolare i parametri dinamici legati alla partita.
  Oltre alla penalità, che viene moltiplicata per la percentuale di coverage, già definita come potenza di 0.9 per ogni turno aggiuntivo, come anticipato, si è introdotto un semplice controllo sulla base della difficoltà del bot e sulla percentuale di copertura ottenuta al primo tentativo, definendo così un bonus di 1.1 o 1.3 nel caso di bot rispettivamente medio o difficile, questo bonus viene poi moltiplicato per la percentuale di coverage ottenuta facendo così crescere il punteggio.
  È stato poi ulteriormente definito il moltiplicatore statico al punteggio parziale ottenuto fino ad adesso, che tiene conto della difficoltà generica della sfida, parametro dato dalla somma della difficoltà del bot e della classe.
  La difficoltà della classe è stata intesa come somma delle righe di codice totali, della complessità ciclomatica e del numero di metodi private passato come parametro alla funzione.
  Il punteggio fino ad adesso ottenuto viene, infine, moltiplicato per un altro fattore dinamico, che va da un minimo di 1 a un massimo di 2, che tiene conto del rapporto del numero di metodi private coperti dal giocatore e di quelli effettivamente presenti nella classe. Il punteggio ottenuto come moltiplicazione della percentuale di righe di codice coperte per tutti questi moltiplicatori viene arrotondato alla seconda cifra decimale e restituito.

In particolare, gli innovativi moltiplicatori introdotti sono stati implementati nel modo seguente:

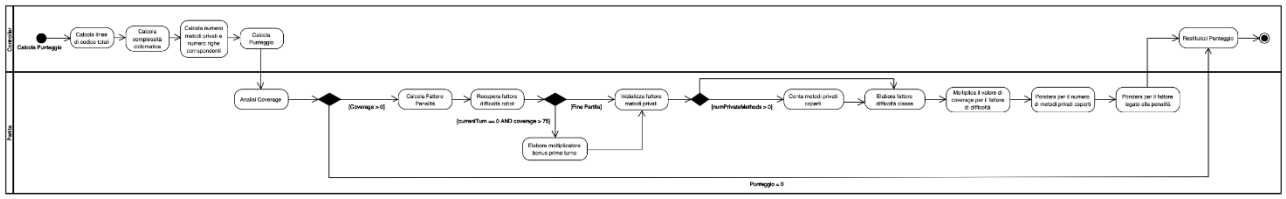
- **Bonus primo turno e penalizzazioni:** per definire questo bonus per il primo turno e penalizzazioni per turni aggiuntivi si è operato all'interno del contesto stesso della funzione *GetScore() e* in quanto la definizione del bonus è un semplice controllo con un *if* sul turno corrente e sulla copertura che viene passata come parametro alla funzione e con un ulteriore controllo sulla difficoltà del bot.
- **Complessità:** il fattore della complessità, come detto, è dato dalla somma della complessità del bot (passata come parametro alla funzione stessa) e la complessità ciclomatica, il numero di righe della classe e il numero di metodi private, tutti e 3 parametri ottenuti richiamando apposite funzioni che operano sul codice sorgente della classe e definite in *score_util*.
- **Numero di metodi privati coperti:** il numero di metodi privati effettivamente coperti dal codice di test è stato ottenuto all'interno di *Sfida*. Il moltiplicatore effettivo del punteggio è stato calcolato all'interno di *getScore()* come 1+ il rapporto tra il numero di metodi coperti e il numero di metodi. È stato sommato 1 in quanto il rapporto in questione è un numero che varia tra 0 e un massimo di 1 e lasciandolo così non avrebbe costituito un fatto moltiplicativo, bensì un fattore decrementale.

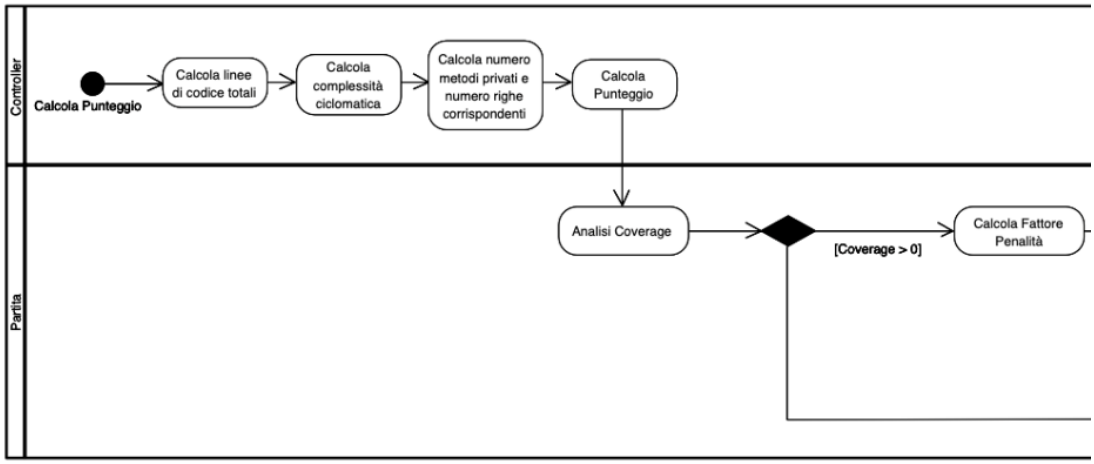
Questi moltiplicatori verranno integrati in un unico fattore ***difficulty_factor*** che sarà pari alla somma di:

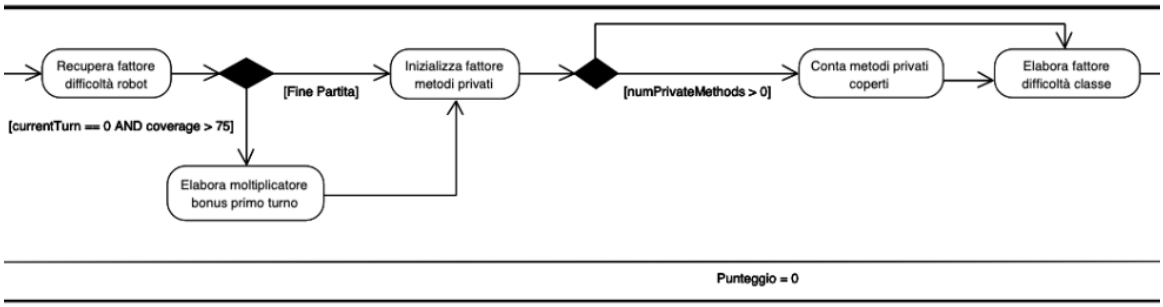
- ***difficultyBot_factor***: rappresenta la difficoltà del bot ed è un parametro che può assumere valori da 1 a 3.
- ***difficultyClass_factor:*** rappresenta la difficoltà della classe ed è ottenuta dalla combinazione del *CyclomaticComplexityFactor* ed il numero di righe di codice totali della classe.

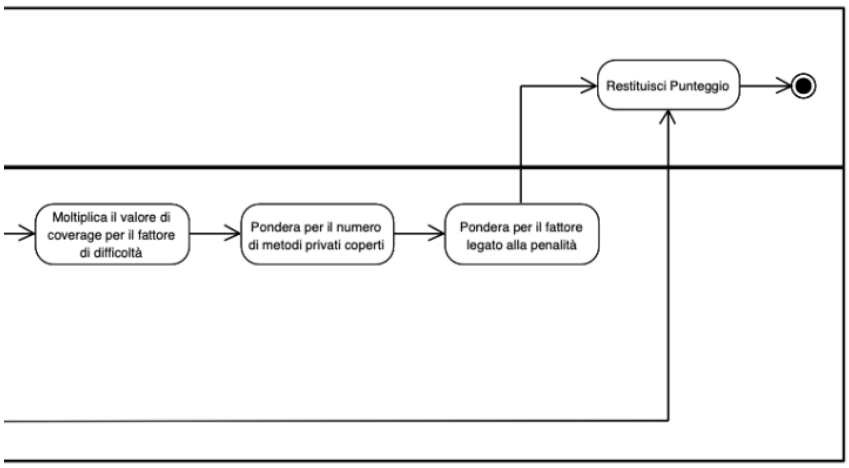
Questo fattore sarà moltiplicato per il valore di **coverage** ottenuto dall'utente proveniente dal file di coverage **jacoco.xml**, per il moltiplicatore *lines_factor (relativo alla copertura nel primo turno)*, per il moltiplicatore *privateMethodsFactor* e per il moltiplicatore *penaltyFactor*, ottenendo il ***final_score***.

Di seguito un **Activity Diagram** che ripercorre l'iter necessario al calcolo del punteggio.




Controller
Calcola Punteggio
Calcola linee di codice totali
Calcola complessità ciclomatica
Calcola numero metodi privati e numero righe corrispondenti
Calcola Punteggio
Partita
Analisi Coverage
[Coverage > 0]
Calcola Fattore Penalità


Recupera fattore difficoltà robot
[Fine Partita]
Inizializza fattore metodi privati
[numPrivateMethods > 0]
Conta metodi privati coperti
Elabora fattore difficoltà classe
[currentTurn == 0 AND coverage > 75]
Elabora moltiplicatore bonus primo turno
Punteggio = 0


Restituisci Punteggio
Moltiplica il valore di coverage per il fattore di difficoltà
Pondera per il numero di metodi privati coperti
Pondera per il fattore legato alla penalità

4.3.3 Esempi

Al fine di chiarire il modo in cui cresce il punteggio in maniera direttamente proporzionale alla difficoltà della classe testata, vengono proposti in forma tabellare i risultati ottenuti a seguito dell'esecuzione di test su 3 classi a complessità crescente: *MyClass1*, *MyClass2*, *MyClass3*.

Per una maggiore leggibilità della tabella vengono rimossi i dati relativi alla copertura ottenuta dal test prodotto, al turno corrente e alla difficoltà del bot, in quanto costanti e sempre corrispondenti rispettivamente a:

- 100% di copertura
- Primo turno giocato
- Difficoltà pari a 2.

Classe	Numero righe (senza commenti e righe vuote)	Complessità ciclomatica	Num. Metodi private	Punteggio
MyClass1	11	1	0	221
MyClass2	33	8	0	225
MyClass3	36	8	1	450

La scelta delle 3 classi *MyClass1*, *MyClass2*, *MyClass3* è stata dettata dall'esigenza di mostrare con tre esempi giocattolo come la complessità della classe contribuisca alla crescita del punteggio e quali parametri hanno più peso rispetto ad altri. Per meglio comprendere tutto ciò viene di seguito fornita una panoramica più dettagliata sul codice delle classi oggetto di esempio e sulla crescita dei vari parametri.

MyClass1

- **Codice**

```
public class MyClass1 {

    // Metodo 1: Saluta con un messaggio
    public String greet(String name) {
        return "Ciao, " + name + "! Benvenuto.";
    }

    // Metodo 2: Somma due numeri e restituisce il risultato
    public int addNumbers(int a, int b) {
        return a + b;
    }

    // Metodo 3: Calcola il doppio di un numero
    public int doubleValue(int number) {
        return number * 2;
    }
}
```

- **Fattori**
  - Il currentTurn è 0 (siamo al primo turno), quindi il fattore **penaltyFactor =1** (non ci sarà penalizzazione per il turno aggiuntivo).
  - Il codice della classe MyClass contiene **11** righe escluse quelle vuote o di commento, quindi il fattore **num_lines = 11**.
  - La difficoltà del bot è 2, quindi **difficultyBot_factor = 2.0**.
  - Siccome la copertura al primo turno è stata del 100% e la difficoltà del bot è 2, ci sarà un fattore moltiplicativo **lines_factor = 1.1**
  - La classe MyClass non ha metodi privati, quindi **numPrivateMethods = 0.**
  - Poiché non ci sono metodi privati, il valore privateMethodsCovered è irrilevante. Ma per la logica, sarebbe 0. Quindi il fattore **privateMethodsFactor = 1.**
  - La classe non ha if o else quindi **CyclomaticComplexity = 1.**
  - Il fattore di difficoltà statica della classe difficultyClass è dato dalla somma di CyclomaticComplexity + num_lines + numPrivateMethods, pertanto sarà pari a 12. Il fattore **difficultyClassFactor** sarà a tal proposito 0.012.
  - Il fattore di difficoltà che tiene conto della somma di difficultyBot_factor + difficultyClassFactor sarà pari a **difficulty_factor = 2.012.**

- **Punteggio**
  - Il **primo prodotto parziale** è dato dal prodotto della percentuale di copertura per il difficulty_factor, quindi sarà pari a **201,2**.
  - Il **secondo prodotto parziale** è dato dal prodotto del primo prodotto parziale *lines_factor*privateMethodsFactor, quindi sarà **221.32**
  - Il **prodotto finale** è dato da questo secondo prodotto parziale moltiplicato per il fattore di penalità e convertito in intero per eccesso o difetto, pertanto sarà pari a **221**.

MyClass2

- **Codice**

```
public class MyClass2 {
  // Metodo 1: Saluta con un messaggio
  public String greet(String name) {
    if (name == null || name.isEmpty()) {
      return "Ciao! Non hai fornito un nome.";
    }
    String greeting = "Ciao, " + name + "! Benvenuto.";
    if (name.length() > 10) {
      greeting += " Hai un nome lungo!";
    }
    return greeting;
  }
```

```
    // Metodo 2: Somma due numeri e restituisce il risultato
    public int addNumbers(int a, int b) {
        int sum = a + b;

        if (sum > 100) {
            System.out.println("La somma è maggiore di 100.");
        } else if (sum < 0) {
            System.out.println("La somma è negativa.");
        }
            else if (sum <10){

        return sum;
    }

    // Metodo 3: Calcola il doppio di un numero
    public int doubleValue(int number) {
        int result = 0;

        // Usa un ciclo for per calcolare il doppio tramite somme ripetute
        for (int i = 0; i < 2; i++) {
            result += number;
        }

        if (result % 2 == 0) {
            System.out.println("Il doppio è pari.");
        } else {
            System.out.println("Il doppio è dispari.");
        }

        return result;
    }
}
```

- **Fattori**
  - Il currentTurn è 0 (siamo al primo turno), quindi il fattore **penaltyFactor =1** (non ci sarà penalizzazione per il turno aggiuntivo).
  - Il codice della classe contiene **33** righe escluse le righe vuote o di commento, quindi il fattore **num_lines = 33**.
  - La difficoltà del bot è 2, quindi **difficultyBot_factor = 2.0**.
  - Siccome la copertura al primo turno è stata del 100% e la difficoltà del bot è 2, ci sarà un fattore moltiplicativo **lines_factor = 1.1**
  - La classe MyClass non ha metodi privati, quindi **numPrivateMethods = 0.**
  - Poiché non ci sono metodi privati, il valore privateMethodsCovered è irrilevante. Ma per la logica, sarebbe 0. Quindi il fattore **privateMethodsFactor = 1.**
  - La classe ha 7 tra if, else, for e else if, quindi **CyclomaticComplexity = 8.**
  - Il fattore di difficoltà statica della classe difficultyClass è dato dalla somma di CyclomaticComplexity + num_lines + numPrivateMethods, pertanto sarà pari a 41. Il fattore **difficultyClassFactor = 0.041**.
  - Il fattore di difficoltà che tiene conto della somma di difficultyBot_factor+difficultyClassFactor sarà pari a **difficulty_factor = 2.041.**

- **Punteggio**
    - Il **primo prodotto parziale** è dato dal prodotto della percentuale di copertura per il difficulty_factor, quindi sarà pari a **204,1**.
    - Il **secondo prodotto parziale** è dato dal prodotto del primo prodotto parziale *lines_factor*privateMethodsFactor, quindi sarà **224.51**.
    - Il **prodotto finale** è dato da questo secondo prodotto parziale moltiplicato per il fattore di penalità e convertito in intero per eccesso o difetto, pertanto sarà pari a **225**.

MyClass3

- **Codice**

```
public class MyClass3 {

  // Metodo 1: Saluta con un messaggio
  public String greet(String name) {
    if (name == null || name.isEmpty()) {
      return "Ciao! Non hai fornito un nome.";
            }
    String greeting = "Ciao, " + name + "! Benvenuto.";
    if (name.length() > 10) {
      greeting += " Hai un nome lungo!";
    }
    return greeting;
          }

  // Metodo 2: Somma due numeri e restituisce il risultato
  public int addNumbers(int a, int b) {
            int sum = a + b;
    if (sum > 100) {
      System.out.println("La somma è maggiore di 100.");
    } else if (sum < 0) {
      System.out.println("La somma è negativa.");
    }
              else if (sum <10){
    return sum;
          }

  // Metodo 3: Calcola il doppio di un numero
  public int doubleValue(int number) {
            int result = 0;
    // Usa un ciclo for per calcolare il doppio tramite somme ripetute
    for (int i = 0; i < 2; i++) {
      result += number;
            }
```

```
        if (result % 2 == 0) {
            System.out.println("Il doppio è pari.");
        } else {
            System.out.println("Il doppio è dispari.");
                    }
        return result;
    }
}
```

- **Fattori**
  - Il currentTurn è 0 (siamo al primo turno), quindi il fattore **penaltyFactor =1** (non ci sarà penalizzazione per il turno aggiuntivo).
  - Il codice della classe MyClass contiene **36** righe escluse le righe vuote o di commento, quindi il fattore **num_lines = 36**.
  - La difficoltà del bot è 2, quindi **difficultyBot_factor = 2.0**.
  - Siccome la copertura al primo turno è stata del 100% e la difficoltà del bot è 2, ci sarà un fattore moltiplicativo **lines_factor = 1.1**
  - La classe MyClass non ha 1 metodo privato, quindi **numPrivateMethods = 1.**
  - Il valore privateMethodsCovered, ossia dei metodi private coperti dal test, è 1. Quindi, il fattore che tiene conto del rapporto di metodi private coperti e di quelli esistenti + 1 è **privateMethodsFactor = 2.**
  - La classe ha 7 tra if , else, for e else if quindi **CyclomaticComplexity = 8.**
  - Il fattore di difficoltà statica della classe difficultyClass è dato dalla somma di CyclomaticComplexity + num_lines + numPrivateMethods, pertanto sarà pari a 45. Il fattore **difficultyClassFactor = 0.045**.
  - Il fattore di difficoltà che tiene conto della somma di difficultyBot_factor + difficultyClassFactor sarà pari a **difficulty_factor = 2.045.**

- **Punteggio**
  - Il **primo prodotto parziale** è dato dal prodotto della percentuale di copertura per il difficulty_factor, quindi sarà pari a **204,5**.
  - Il **secondo prodotto parziale** è dato dal prodotto del primo prodotto parziale *lines_factor*privateMethodsFactor, quindi sarà **449.9**.
  - Il **prodotto finale** è dato da questo secondo prodotto parziale moltiplicato per il fattore di penalità e convertito in intero per eccesso o difetto, pertanto sarà pari a **450**.

4.3.4 Test Effettuati

Nella scelta dei test da realizzare per quanto riguarda il punteggio si è cercato di coprire tutte le casistiche possibili affinché potessero entrare in gioco i vari moltiplicatori.

Per tenere conto del fattore della difficoltà del bot è stata testata la stessa classe *FTPFile* nel caso del robot Randoop in tutte e 3 le difficoltà.

La scelta della classe *FTPFile* è stata dettata anche dalla presenza di metodi private al suo interno per verificare l'effettivo peso del moltiplicatore del numero di metodi private in cui si riesce a entrare.

Per tenere conto del fattore di penalizzazione per turni aggiuntivi si è pensato di testare *FTPFile* nel caso di bot facile giocando due turni.

Per tenere conto e verificare il fattore moltiplicativo al primo turno si è deciso di includere un caso di test con la classe *FTPFile* nel caso di bot medio giocando due turni, il primo con una copertura <75%, il secondo con uno maggiore.

Per verificare che effettivamente i parametri di difficoltà non fossero influenzati dalla tipologia di bot, ma solo dalla sua difficoltà, si è deciso di includere anche un caso di test della classe *FTPFile* nel caso di bot EvoSuite alla difficoltà difficile.

Per tenere conto dell'effettiva influenza della difficoltà della classe si è effettuato anche un test con una classe differente ossia *OutputFormat*.

ID	Precond	Input	Expected Output	Output	Esito
FTP_facile_primo_ tentativo_Randoop	Il giocatore ha selezionato la classe FTPFile; il giocatore ha selezionato il bot Randoop; il giocatore ha selezionato difficoltà del bot facile; il giocatore al primo tentativo ha scritto nell'apposito editor una classe di test che copra il 100% del codice	Bottone Invia premuto	Punteggio: 149pt	Punteggio: 149pt	Pass
FTP_facile_secondo_ tentativo_Randoop	Il giocatore ha selezionato la classe FTPFile; il giocatore ha selezionato il bot Randoop; Il giocatore ha selezionato	Bottone Invia premuto	Punteggio: 134pt	Punteggio: 134pt	Pass

	difficoltà del bot facile; Il giocatore ha **giocato** il primo tentativo con una classe di test che copra il 100% del codice al secondo tentativo; Il giocatore al secondo tentativo ha scritto nell'apposito editor una classe di test che copra il 100% del codice				
FTP_medio_primo_ tentativo_Randoop	Il giocatore ha selezionato la classe FTPFile; Il giocatore ha selezionato il bot Randoop; Il giocatore ha selezionato difficoltà del bot medio; Il giocatore al primo tentativo ha scritto nell'apposito editor una classe di test che copra il 100% del codice	Bottone Invia premuto	Punteggio: 273pt	Punteggio: 273pt	Pass
FTP_medio_secondo_ tentativo_Randoop	Il giocatore ha selezionato la classe FTPFile; il giocatore ha selezionato il bot Randoop; Il giocatore ha selezionato difficoltà del bot medio; Il giocatore ha **giocato** il primo tentativo con una classe di test che non copra più del 75%	Bottone Invia premuto	Punteggio: 224 pt	Punteggio: 224 pt	Pass

	(precisamente copre il 42.7%); Il giocatore al secondo tentativo ha scritto nell'apposito editor una classe di test che copra il 100% del codice				
FTP_difficile_primo_tentativo_Randoop	Il giocatore ha selezionato la classe FTPFile; Il giocatore ha selezionato il bot Randoop; Il giocatore ha selezionato difficoltà del bot difficile; Il giocatore al primo tentativo ha scritto nell'apposito editor una classe di test che copra il 100% del codice	Bottone Invia premuto	Punteggio: 453pt	Punteggio: 453pt	Pass
FTP_difficile_primo_tentativo_Evosuite	Il giocatore ha selezionato la classe FTPFile; Il giocatore ha selezionato il bot Evosuite; Il giocatore ha selezionato difficoltà del bot difficile; Il giocatore al primo tentativo ha scritto nell'apposito editor una classe di test che copra il 100% del codice	Bottone Invia premuto	Punteggio: 453pt	Punteggio: 453pt	Pass
Output_facile_primo_tentativo_Randoop	Il giocatore ha selezionato la classe OutputFormat;	Bottone Invia premuto	Punteggio: 146pt	Punteggio: 146pt	Pass

	il giocatore ha selezionato il bot Randoop; il giocatore ha selezionato difficoltà del bot facile; il giocatore al primo tentativo ha scritto nell'apposito editor una classe di test che copra il 100% del codice				

4.4 Sequence Diagram

Viene di seguito presentato un Diagramma di Sequenza di basso livello per evidenziare il flusso di esecuzione dinamico e l'interazione tra i task T5 e T7 nell'ambito dell'invio della partita e della visualizzazione della copertura del bot, al fine di giustificare ulteriormente le modifiche descritte da un punto di vista statico nei Class Diagram precedentemente proposti.

Tale flusso si riferisce a una partita giocata contro **Randoop**, l'unico scenario per il quale è garantito il corretto funzionamento.

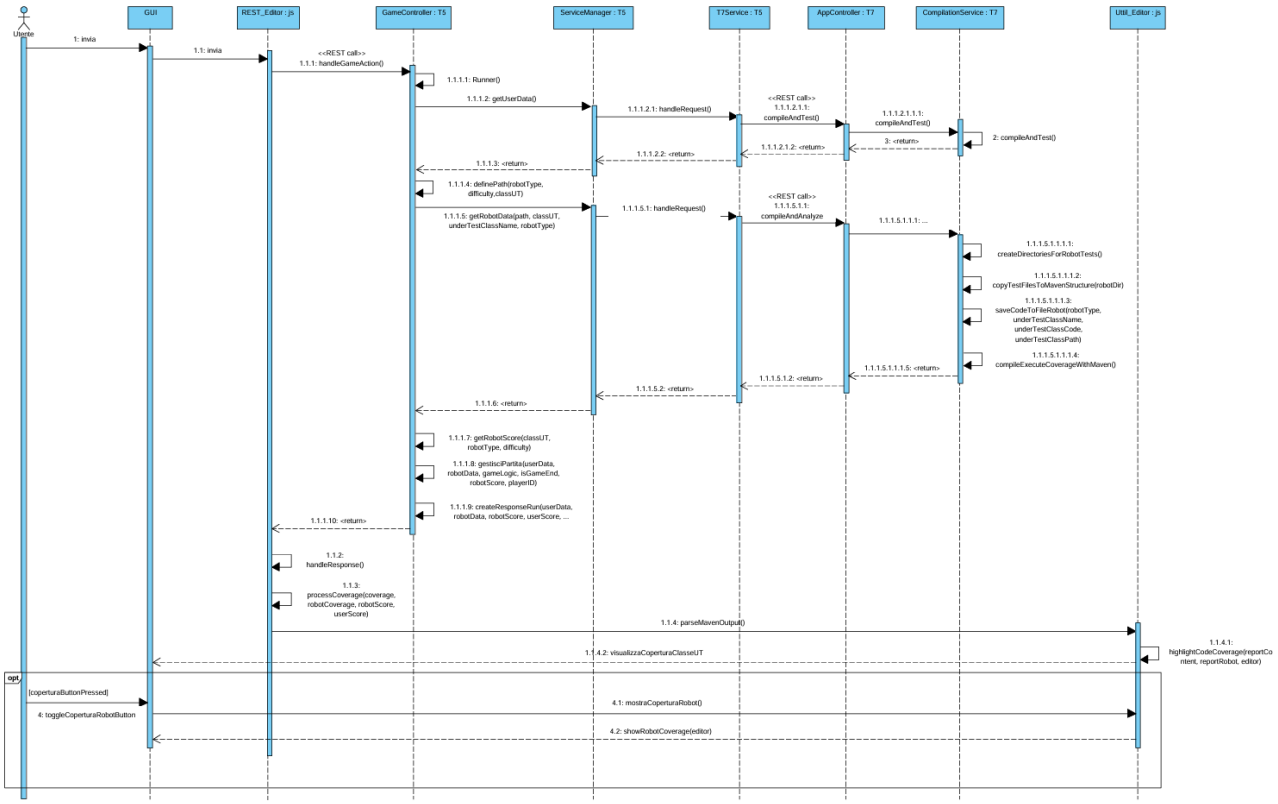
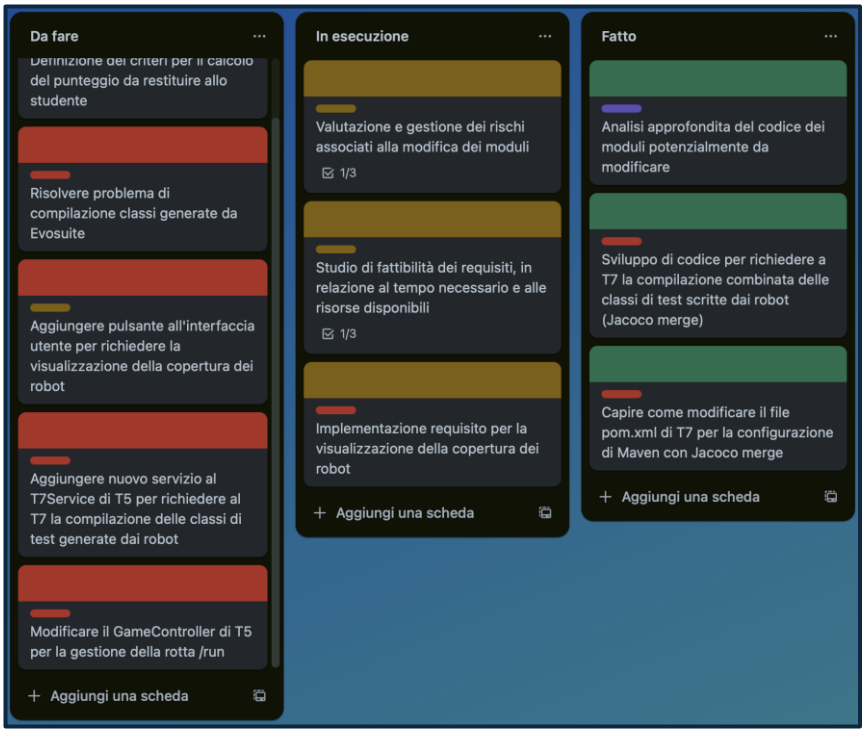
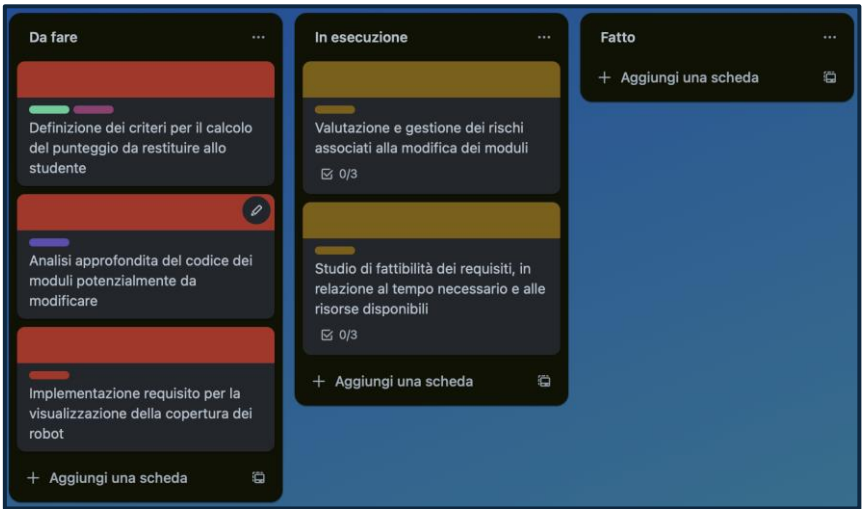


Figura 20 - Sequence Diagram di dettaglio Invia Partita

4.5 Esempi di utilizzo di Trello

Si riportano, a scopo esemplificativo, i diversi stati della bacheca di **Trello** nel corso delle varie attività.





5 Container T10

Sulla falsa riga dei container già presenti, la costruzione del **T10G724 – EvoSuite Compiler** è avvenuta utilizzando lo strumento online **Spring Initializr**, che semplifica la creazione di progetti basati su **Spring Boot**; questa è un'estensione di Spring che facilita la configurazione e il deployment di applicazioni Spring, attraverso l'utilizzo di moduli, tra cui **Spring MVC**.

Lo strumento restituisce una configurazione iniziale, includendo solo le dipendenze necessarie per avviare lo sviluppo, che è stata opportunamente modificata per abilitare l'utilizzo della versione 1.8 di Jdk (Java 8).

5.1 Caratteristiche Principali

Evosuite Compiler funge da servizio remoto di compilazione per classi di test generate da EvoSuite. In particolare, ha la funzione di valutare l'efficacia dei test generati, determinando il livello di copertura ottenuto sulle classi oggetto di verifica. Progettato per operare in modalità remota, come il T7, questo componente è facilmente adattabile a scenari che adottano un modello di interazione orientato ai servizi.

Il processo di compilazione si basa sull'utilizzo di **Maven**. È importante evidenziare che questo strumento viene ovviamente impiegato anche nella fase di compilazione del codice sorgente, che precede il processo di build del container con Docker. Durante la costruzione del container, il **Dockerfile** specifica l'adozione di Java 8 come runtime, risolvendo il problema delle dipendenze precedentemente descritto.

Per tale ragione, sono presenti due file di configurazione **pom.xml**. Quello a cui far riferimento per la configurazione di Maven per i test Evosuite è contenuto nella directory *testCompiler*.

5.1.1 pom.xml testCompiler

La configurazione si basa su una serie di dipendenze e plugin. In particolare, Maven scarica automaticamente le dipendenze dichiarate dai repository remoti, le aggiunge al classpath e le rende disponibili per la compilazione e i test. Inoltre, configura ed esegue i plugin specificati durante le fasi del ciclo di vita, come la compilazione del codice o l'esecuzione dei test.

Dipendenze principali

- **JUnit (v4.13.1):** Fornisce il framework necessario per la scrittura e l'esecuzione dei test unitari. È una libreria ampiamente utilizzata in ambito Java per definire test modulari e automatizzati.
- **XMLUnit Core (v2.9.0)**: Una libreria dedicata al confronto e alla verifica di file XML. Risulta particolarmente utile nei test che richiedono il controllo di output strutturati in formato XML.
- **EvoSuite Runtime (v1.0.6):** Una dipendenza essenziale per eseguire i test generati automaticamente da EvoSuite.

Tutte le dipendenze hanno lo scope test, quindi vengono utilizzate esclusivamente durante la fase di testing, garantendo un impatto minimo sulle dimensioni e le prestazioni del prodotto finale.

Plugin configurati

- **Maven Surefire Plugin:** Plugin responsabile dell'esecuzione dei test durante la fase test del ciclo di vita di Maven. Lavora in combinazione con JUnit per garantire che i test siano eseguiti in modo affidabile e riproducibile.
- **JaCoCo Maven Plugin:** Utilizzato per misurare la copertura del codice durante i test.
- **EvoSuite Maven Plugin:** Integra EvoSuite nel processo Maven per generare automaticamente classi di test.

5.2 Class Diagram

Il **Class Diagram** complessivo del container, che rappresenta la sua struttura statica, è riportato in figura. Come si può osservare, la configurazione è molto simile a quella del T7, essendo la funzione, in generale, pressoché la stessa.

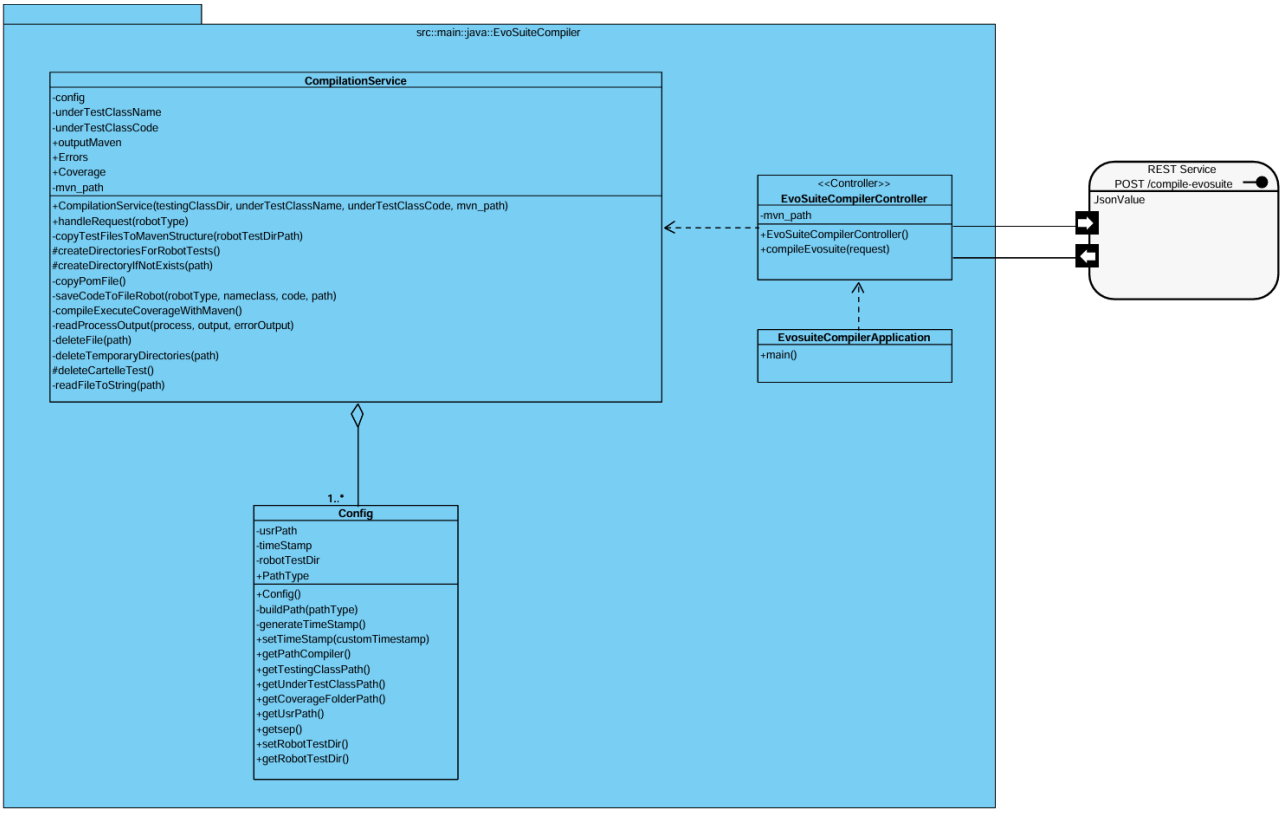


Figura 21 - Class Diagram T10

EvosuiteCompilerController

È il componente Spring Boot che espone l'endpoint REST *"/compile-evosuite"* per la compilazione delle classi di test generate da EvoSuite. La ricezione di una chiamata per tale rotta manda in esecuzione il metodo **compileEvoSuite(request)**, che al suo interno costruisce un'istanza di **CompilationService**, responsabile dell'attività di compilazione vera e propria.

EvoSuiteCompilerApplication

È il punto di ingresso principale per l'applicazione Spring Boot **EvoSuiteCompiler**. La sua funzione principale è avviare l'applicazione e configurare il contesto Spring.

CompilationService

È la classe che si occupa della compilazione, dell'esecuzione dei test e della generazione dei report di copertura del codice per i test generati da EvoSuite. Include al suo interno metodi per preparare le directory, copiare file, gestire processi Maven e pulire risorse temporanee, tra cui:

- **CompilationService()**: costruttore; configura il servizio con i dettagli delle directory dei test, della classe sotto test e del percorso di Maven.
- **handleRequest()**: è il metodo che gestisce l'intera pipeline; prepara le directory, copia i file, salva la classe sotto test, compila ed esegue i test, legge i report di copertura e, infine, rimuove le directory temporanee. Lancia, inoltre, eccezioni se qualcosa dovesse andare storto.

- **copyTestFileToMavenStructure()**: copia i file di test generati dai robot in una struttura compatibile con Maven, **src/test/java**. Lancia un'eccezione se la directory dei test non esiste o non contiene file validi.
- **createDirectoriesForRobotTests()**: crea e verifica l'esistenza delle directory per la compilazione, dei test generati dai robot, della classe sotto test, del report di copertura, e richiede la copia del file *pom.xml* con una chiamata a…
- **copyPomFile()**: copia il file *pom.xml* presente nella cartella **testCompiler** (necessaria per separare il pom per la compilazione dei test da quello del codice dell'intero container) nella directory di compilazione.
- **saveCodeToFileRobot()**: aggiunge una dichiarazione di package alla classe da testare per garantirne la visibilità alle classi di test e salva il file nella directory specificata. Lancia eccezioni se il nome della classe o il percorso sono invalidi.
- **compileExecuteCoverageWithMaven()**: esegue un processo Maven per compilare ed eseguire i test; in particolare, registra l'output del processo e gestisce errori o timeout. Ritorna *true* se la compilazione è avvenuta con successo, altrimenti lancia un'eccezione.

Config

Si tratta di una classe responsabile di gestire le informazioni riguardanti i path dove salvare i file java e di impostare e recuperare la directory contenente i test generati dai robot, con validazioni per garantire che il valore sia configurato correttamente.

6 Nuovo Regolamento Punteggi

Di seguito una breve presentazione della nuova regolamentazione relativa ai punteggi, visionabile dall'utente direttamente dall'applicazione mediante un apposito pulsante di info:

Benvenuto nella nostra sfida di **testing gamification**! Scopri come puoi ottenere il massimo del punteggio e scalare la classifica. Abbiamo ideato un sistema di calcolo che premia la tua strategia e abilità, bilanciando le difficoltà delle classi da testare. Ecco i criteri:

---

1. Copertura dei Metodi Privati

Più riesci a coprire i metodi privati di una classe, maggiore sarà il tuo vantaggio.

- Ogni metodo privato coperto viene rapportato al totale dei metodi privati della classe.

(Fai attenzione ai dettagli e otterrai bonus nascosti!)

---

2. Complessità Ciclomatica

Affrontare classi complesse non è per tutti: se riesci a gestirle bene, il punteggio rifletterà il tuo impegno.

- Valutiamo quanti costrutti iterativi e di selezione hai coperto rispetto al totale della classe.

(The greater the complexity, the bigger the reward)

---

3. Penalità per Turni Aggiuntivi

Ogni turno extra ti costerà punti.

- Per ogni turno, il tuo punteggio viene ridotto di un fattore autoincrementante

(Vuoi evitare penalità? Copri il massimo al primo turno!)

---

4. Bonus Copertura al Primo Turno

Se superi il **75% di copertura** al primo turno, otterrai un moltiplicatore extra basato sulla difficoltà:

- **Difficoltà Media**: bonus del 10%.
- **Difficoltà Difficile**: bonus di 30%.

(Un primo turno esplosivo vale doppio!)

---

5. Fattore di Difficoltà Totale

Il punteggio tiene conto della difficoltà del bot e della classe sotto test:

- Bot più difficili ti premiano di più.
- Classi più complesse, con tanti metodi privati e maggiore numero di righe, aumentano il valore del tuo moltiplicatore.

(Affronta il livello giusto per sfidare te stesso e guadagnare di più!)

Punteggio Finale

Il tuo punteggio viene calcolato sommando la tua copertura, i bonus ottenuti e i moltiplicatori applicati, scala la classifica!

Gioca Strategico e Vinci

Punta a coprire il massimo al primo turno, scegliendo con attenzione il livello di difficoltà del bot e delle classi. Ogni decisione conta per conquistare il podio!

7 Architettura Finale

Segue il Deployment Diagram raffigurante l'architettura complessiva dell'applicativo, in cui è evidenziato il nuovo componente **T10** aggiunto.

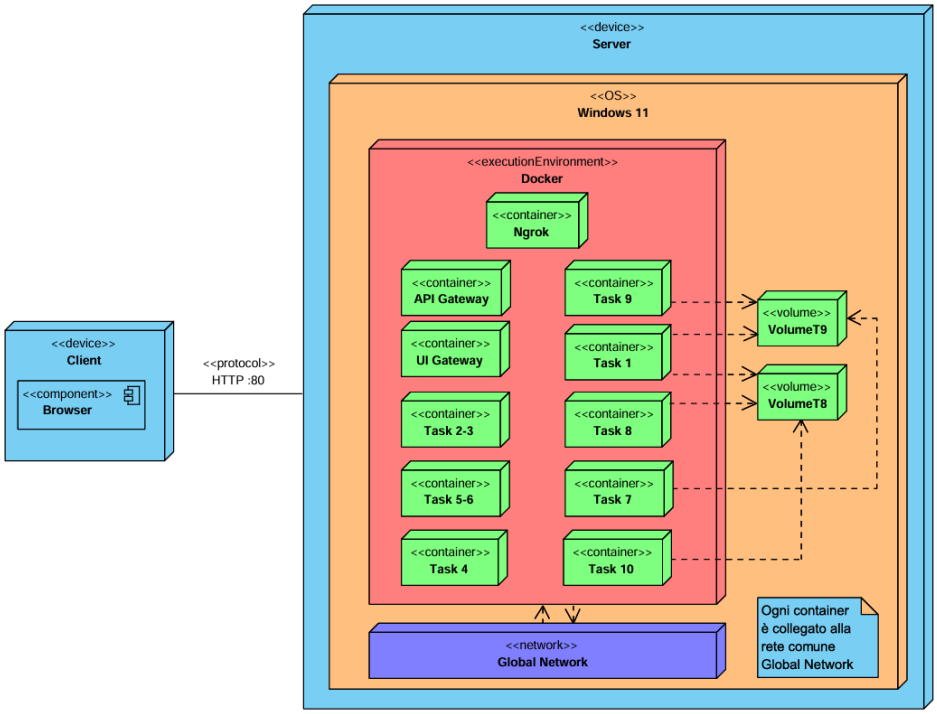


Figura 22 - Deployment Diagram Finale

Dalla figura è possibile notare anche le dipendenze di **T7** e **T10** dai volumi **T8** e **T9** per il recupero delle classi di test prodotte dai robot per la compilazione.

A conclusione, si presenta il Component Diagram aggiornato con l'inclusione del nuovo volume.

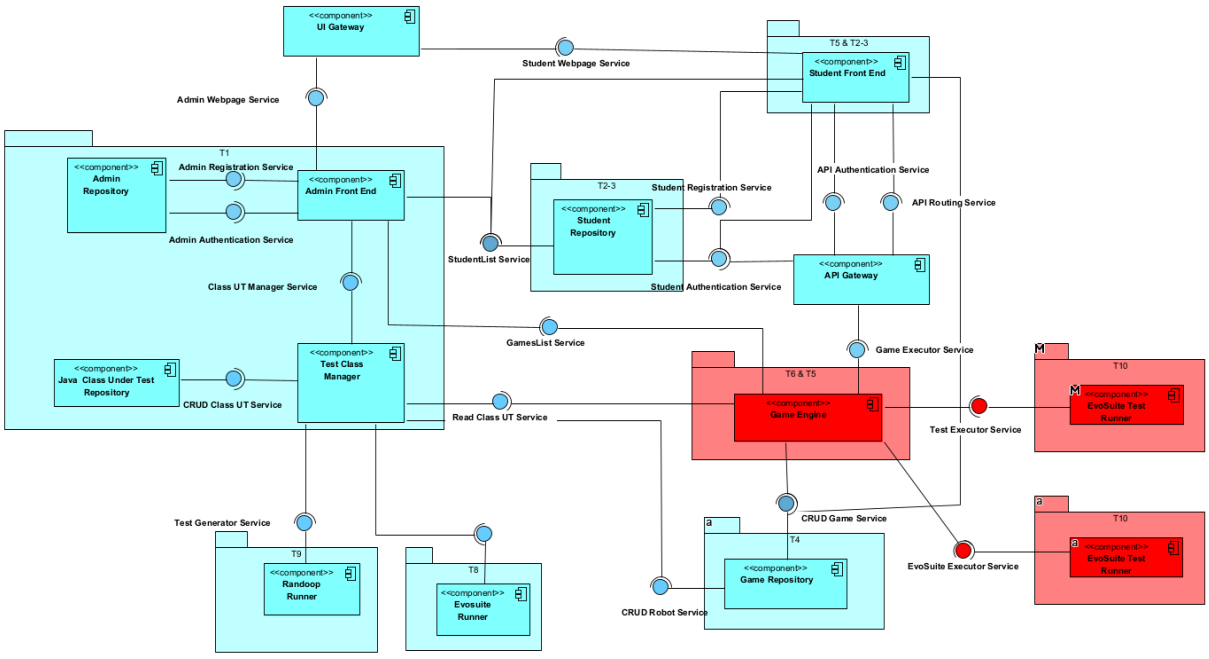


Figura 23: Component Diagram con l'inclusione di T10

8 Sviluppi futuri

Tra le proposte che il team di progetto intende avanzare, figura quella di affrontare il problema relativo alla copertura di Maven per classi di test EvoSuite. Con ogni probabilità, la questione risiede in una configurazione non corretta; pertanto, eventuali interventi futuri dovranno essere orientati alla modifica e ottimizzazione del file **pom.xml**.

In merito alla valutazione della copertura ottenuta dalle classi di test generate dai robot, per alleggerire il carico sull'applicazione, potrebbe essere utile considerare l'inserimento del file di copertura **jacoco.xml** direttamente nelle cartelle che vengono caricate dall'admin insieme alla classe da testare; l'integrazione del file di copertura permetterebbe di evitare il processo completo di compilazione, riservandolo invece ai casi in cui venga caricata unicamente la classe da testare e si richieda la generazione dei relativi test, presumibilmente ai task T8 e T9.

In riferimento a questi ultimi, potrebbe essere opportuno prevedere un processo di re-factoring per migliorarne la struttura e allinearla con l'architettura complessiva dell'applicazione.