Scuola Politecnica e delle Scienze di Base

Corso di Laurea in Ingegneria Informatica

Elaborato di Software Architecture Design

Progetto ENACTEST: Documentazione TextEditor

Anno Accademico 2022/2023

Gruppo G12 – Task #T6:

Guarino Nicola – M63001472

Niola Vittorio – M63001429

Russo Giovanni – M63001415

Indice

Capitolo 1: Introduzione al Progetto

1.1 European iNnovative AllianCe for TESTing - ENACTEST

Il progetto di cui il Team si è occupato va contestualizzato nell'ambito dell'*ENACTEST:* un consorzio il cui obiettivo è la progettazione e la realizzazione di materiali didattici a supporto dell'insegnamento delle tecniche di testing che siano in linea con le esigenze aziendali e industriali. Il consorzio in questione, inoltre, si prefigge l'obiettivo di studiare la fattibilità dell'integrazione di suddetto materiale nell'ambito di curricula formativi universitari e professionali al fine di migliorare l'efficacia del processo di apprendimento del testing e ridurre, così, il bisogno di percorsi formativi supplementari in luoghi aziendali.

1.1.1 Obiettivi del Progetto ENACTEST

L'obiettivo del progetto è la realizzazione di un *Educational Game* in cui un giocatore – che verrà identificato con l'appellativo di *Player* – sfida un *tool di generazione automatica di test* (nella fattispecie *Unit Test*) come *Randoop* o *Evosuite*. L'applicazione finale dovrà consentire allo studente di:

- Ottenere una classe da testare, identificata con l'appellativo di *Class Under Test* (CUT) ;
- Scrivere dei casi di test JUnit per la *CUT* a partire da un template precompilato JUnit;
- Compilare ed eseguire i Test prodotti;
- Ottenere una misura di copertura tramite strumenti come *Emma* o *JaCoCo*;
- Confrontare i risultati dei casi di test generati dal *player* con quelli generati dai *tool automatici* fornendo, inoltre, delle apposite viste di riepilogo;
- Mantenere uno storico delle partite giocate da ciascun giocatore (con i relativi esiti) al fine di generare statistiche circa l'utilizzo dell'applicazione.

1.2 Task #T6: Editor di Testo

Contestualizzato il progetto *ENACTEST*, scendiamo nel dettaglio del compito assegnato al nostro Team ovvero la *realizzazione di un Editor di Test Case* che, coerentemente al backlog stilato in fase di *ideazione*, deve rispettare gli obiettivi riportati nella Figura seguente:

T6	Requisiti Sull'Editor di Test Case
	L'Editor di Test Case fornirà una finestra di editing di testo Java in cui il giocatore potrà inserire testo e fare le classiche operazioni di editing su testo, usando le modalità di presentazione del testo tipiche di un IDE Java. L'editor dovrà inoltre consentire di salvare i file di testo creati. Per ogni classe da testare potrebbe essere utile pre-caricare un template del caso di test.

Figura 1.1: Backlog Task#T6

1.2.2 Approccio Utilizzato

Per la realizzazione del Task, il Team di Sviluppo ha optato per una metodologia di lavoro *agile* basata sul paradigma ***SCRUM*** ovvero un Framework agile per la gestione iterativa ed incrementale del ciclo di sviluppo del software. La gestione iterativa ed incrementale trova sua massima espressione negli *Sprint* ovvero l'unità di base dello sviluppo di Scrum che, tipicamente, ha durata fissa da una a quattro settimane. Ogni Sprint, è importante osservare, è preceduto da una riunione di pianificazione in cui vengono identificati gli obiettivi e vengono stimati i tempi di lavoro. Al termine di ciascuno di essi il Team di Sviluppo ha il dovere di consegnare una versione potenzialmente completa e funzionante del prodotto, contenente gli avanzamenti decisi nella riunione di pianificazione dello Sprint.

1.2.3 Programmazione del Lavoro

Individuando nel giorno 08/04/2023 la data d'inizio del Progetto e nel giorno 07/06/2023 la data di conclusione, la Tabella proposta di seguito mostra la programmazione del lavoro che ha caratterizzato il periodo di sviluppo:

Iterazione	**Durata**	**Obiettivi Prefissati**	**Documenti Prodotti**
Iterazione #1	Dal 08/04 al 21/04	Individuazione dei Requisiti funzionali e non funzionali. Comprensione del Framework *CodeMirror* e realizzazione di una prima GUI dell'Editor	– Specifica dei Requisiti – Process Flow Diagram – Use Case Diagram – Scenari d'Uso – User Stories
Iterazione #2	Dal 02/05 al 06/05	Implementazione delle funzionalità di: *Find&Replace, ChangeTheme, Save* e *Save As, Undo&Redo* e realizzazione di una finestra nella GUI in cui mostrare la *ClassUnderTest*	– GUI Class Diagram – Activity Diagram – Sequence Diagram
Iterazione #3	Dal 10/05 al 07/06	Rifinitura dell'Editor, gestione ed integrazione delle dipendenze con i Task T4, T5, T7, T8, T9.	– Deployment Diagram – Communication Diagram – Component Diagram

Tabella 1.1: Programmazione del lavoro

Durante la prima iterazione, conclusasi con la Review#1 del giorno 21/04, il Team si è occupato di individuare i Requisiti funzionali e non funzionali e di approfondire la conoscenza del framework *CodeMirror*. Il Team si è occupato, inoltre, della realizzazione di un primo prototipo della GUI dell'Editor di Test così da poterlo mostrare in sede di Review con l'obiettivo di esporre l'idea concepita dal Team e ricevere feedback a riguardo. E' alla prima iterazione, inoltre, che va collocata la realizzazione dei seguenti diagrammi UML: *Process Flow Diagram*, *Use Cases Diagram*, *Use Scenarios* e, infine, le *User Stories* realizzate secondo il paradigma *agile "As – I Want to – So that"*.

Nella seconda iterazione, conclusasi, invece, con la Review#2 del giorno 06/05, il Team si è occupato dell'implementazione delle funzionalità di base dell'Editor al fine di agevolare e semplificare la sessione di Testing da parte del *player*. Sono state, infatti, implementate le funzionalità di: *Find&Replace, Undo&Redo, ChangeTheme* e le funzionalità di *Save* e *Save As*. Lato UML, invece, sono stati prodotti i seguenti diagrammi: *GUI Class Diagram, Activity Diagram* e *Sequence Diagram*.

Durante l'ultima iterazione, conclusasi con la Review#3 del giorno 07/06, il Team si è occupato dell'*integrazione* gestendo tutte le dipendenze con i servizi esterni che verranno compilativamente riportati di seguito e schematicamente mostrati nella Figura 1.2 secondo il paradigma ***SCRUM*** (filo rosso per le dipendenze) e nella Figura 1.3 mediante un *Communication Diagram*: Avvio della Partita (T5), Compilazione&Esecuzione (T7), Generatore di Test Evosuite e Randoop (T8-T9), Mantenimento delle informazioni della Partita (T4).
Il Team si è occupato, nella fattispecie, di preparare le API in formato AJAX per consentire l'invio e la ricezione di dati verso e da i servizi invocati ed invocanti. Il Team ha ultimato, infine, le operazioni di documentazione realizzando i seguenti diagrammi UML: *Deployment Diagram, Communication Diagram* e *Component Diagram*.

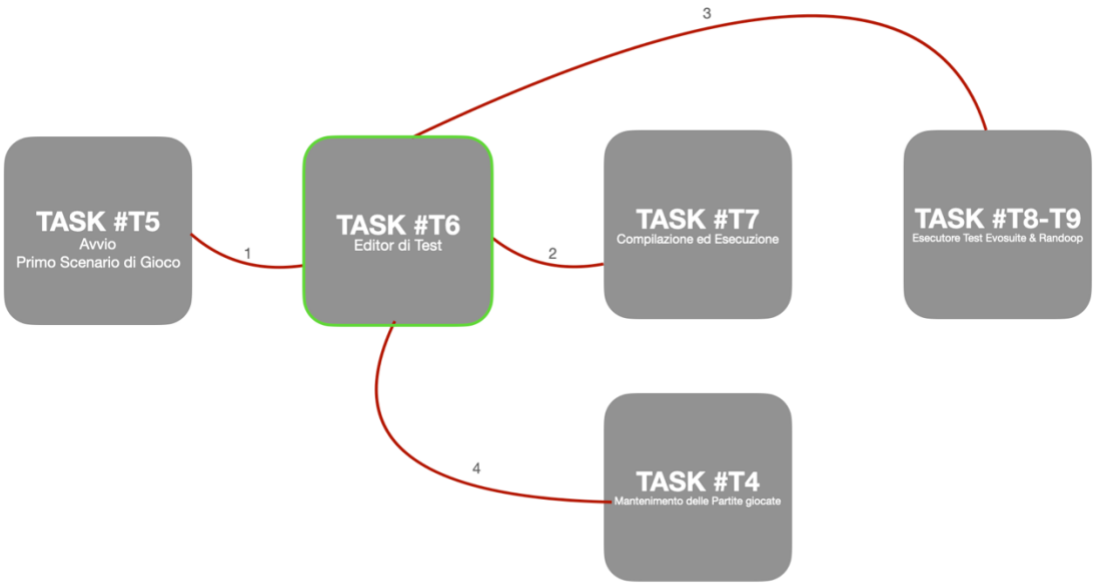

Figura 1.2: Dipendenze con servizi esterni secondo il paradigma SCRUM

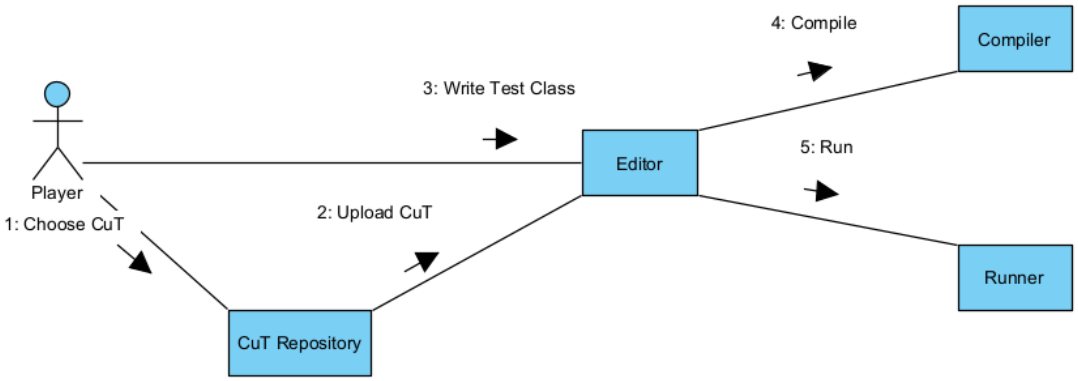

Figura 1.3: Communication Diagram

I progressi delle iterazioni sopracitate sono stati opportunamente e progressivamente resi noti tramite la piattaforma GitHub al seguente link: https://github.com/Testing-Game-SAD-2023/T6-G12 .

Capitolo 2: Specifica dei Requisiti

2.1 Requisiti Informali

Si vuole realizzare un Editor di Testo Java per la scrittura, da parte di un *player,* di Test Case da sottomettere a *strumenti di copertura* al fine di confrontare i propri risultati con quelli ottenuti da un *tool di generazione automatica di Test* come *Randoop* o *Evosuite.* Il *player* deve avere la possibilità di *scrivere i propri Test* all'interno di una finestra di editing, *consultare* la *ClassUnderTest*, *visualizzare* eventuali *warnings* o *errors* nella Console e, infine, *visualizzare* un *report* della sfida contro il Robot. Il *player* avrà, dunque, la possibilità di: *aprire* un documento dal proprio PC, *salvare* il proprio documento, *compilare* ed *eseguire* il Test realizzato, *misurare* l'efficacia del Test Case prodotto e, infine, *confrontare* i propri risultati con quelli del Robot scelto. L'esperienza di coding dovrà essere facilitata dalle funzioni di *undo*, *redo, find e replace* in aggiunta alla possibilità di *autocompletare* il codice tramite specifici suggerimenti.

2.1.1 User Stories

Una *user story* è una breve descrizione di una funzionalità o di un requisito del sistema, espressa dal punto di vista dell'utente. Le user stories, seguendo il paradigma SCRUM, seguono la seguente struttura:

- ***As a*** (tipo di utente) …
- ***I want*** (funzionalità/requisito) …
- ***So that*** (beneficio/desiderio dell'utente) …

Innanzitutto, si identifica il tipo di utente o il ruolo che richiede la funzionalità o il requisito. Successivamente si descrive la funzionalità specifica o il requisito richiesto e infine si illustra il motivo o il beneficio che l'utente ottiene dalla funzionalità o dal requisito richiesto. Di seguito vengono proposte le *storie utente* per le principali funzionalità dell'Editor*:*

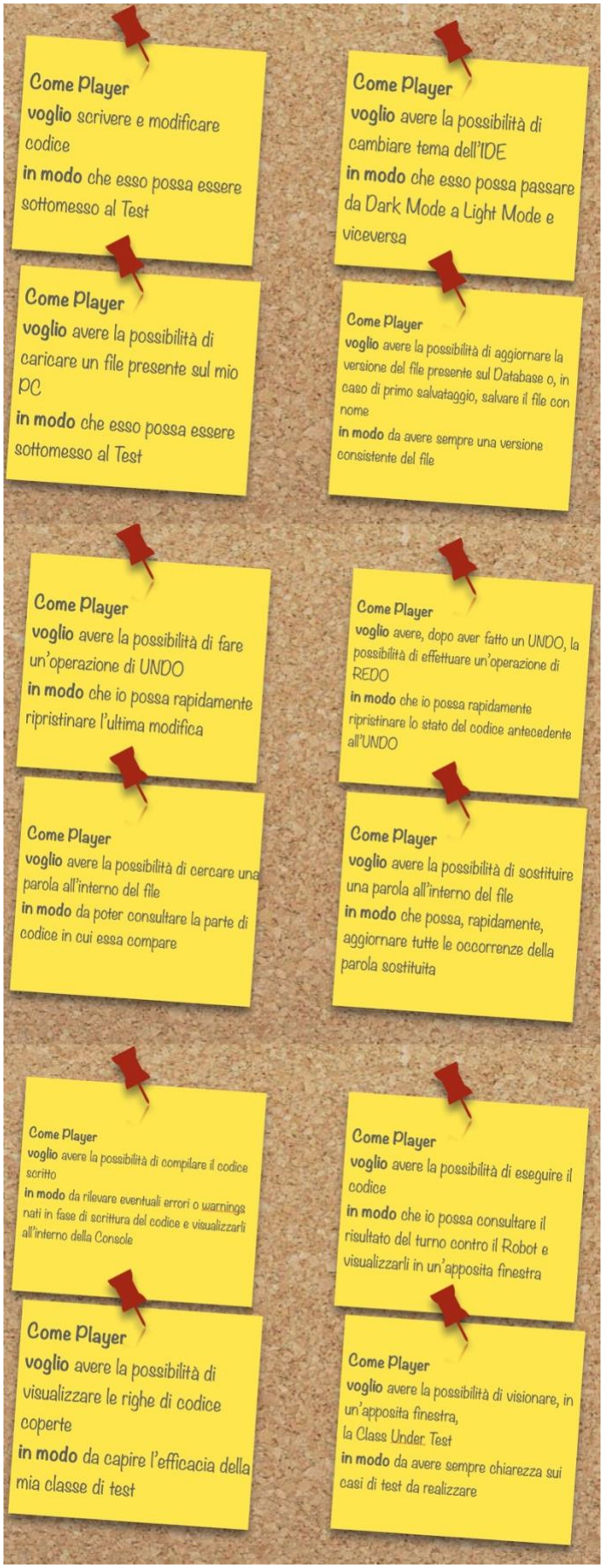

Figura 2.1: User Stories secondo il paradigma SCRUM

2.2 Requisiti Funzionali

Di seguito vengono riportati i requisiti funzionali che consentono di modellare il comportamento delle principali funzionalità offerte dall'Editor:

- Creazione di un nuovo documento di testo;
- Apertura di un documento di testo esistente presente sul proprio dispositivo;
- Salvataggio di un documento di testo con un nome specificato dall'utente;
- Operazioni di modifica del testo come inserimento e cancellazione;
- Possibilità di annullare (*undo*) e ripetere (*redo*) le azioni di modifica del testo;
- Copiare, tagliare ed incollare il testo selezionato;
- Ricerca e sostituzione di testo all'interno del documento;
- Mostrare suggerimenti ed offrire funzionalità di autocompletamento del codice;
- Colorazione sintattica;
- Cambiare tema dell'editor;
- Mostrare le *Game Info*;
- Possibilità di lanciare i servizi di compilazione ed esecuzione del codice realizzato;
- Possibilità di consultare eventuali *errors* o *warnings* all'interno della *Console;*
- Possibilità di visualizzare la *Class Under Test* all'interno di una specifica finestra;
- Possibilità di consultare i risultati raggiunti dal *Player* e dal *Robot* all'interno della finestra *Confronto Risultati*;
- Evidenziare, con i colori *rosso* (copertura assente), *giallo* (copertura parziale) e *verde* (copertura totale) le *linee di codice* secondo i risultati ottenuti dal *Coverage Test* effettuato tramite *Emma* o *JaCoCo;*

2.3 Requisiti non funzionali

Vengono definiti, di seguito, i requisiti di qualità richiesti dall'architettura:

- Usabilità: realizzare un'interfaccia utente intuitiva e semplice da utilizzare;
- Compatibilità e Portabilità con i principali Sistemi Operativi: Windows, macOS e Linux;
- Compatibilità e Portabilità con i principali Web Browser: Chrome, Edge, Opera, Safari, Mozilla Firefox;
- Robustezza: l'applicazione deve comportarsi in maniera accettabile anche quando si verificano problemi con le interfacce;
- Affidabilità: l'applicazione deve garantire un'affidabile gestione dei file per consentire un consistente monitoraggio delle Partite giocate dal *player*;
- Adeguata gestione degli errori tramite messaggi chiari e comprensibili agli utenti;

Capitolo 3: Analisi dei Requisiti

3.1 System Domain Model

Nell'ambito del progetto assegnato, il Team ha individuato quattro entità principali: *Player*, *Partita*, *Turno Partita*, *Esito Turno*. Al fine di definire i concetti principali dell'applicazione, si è scelto di modellare un System Domain Model. Come mostrato dalla Figura seguente, il *Player*, caratterizzato dagli attributi indicati, realizza una relazione di tipo *uno a molti* con l'entità *Partita*; la relazione scelta è giustificata da quanto segue: il *Player*, infatti, potrebbe decidere di avviare più partite contemporaneamente. L'entità *Partita* invece è in relazione *uno a molti* con l'entità *Turno Partita*, in quanto ogni partita può essere caratterizzata da più turni ma ogni turno è associato ad un'unica partita. A sua volta, l'entità *Turno Partita* realizza una relazione di tipo *uno ad uno* con l'entità *Esito Turno* poiché ogni turno prevede, nella sua fase conclusiva, un unico esito che, a sua volta, è associato ad uno specifico turno.

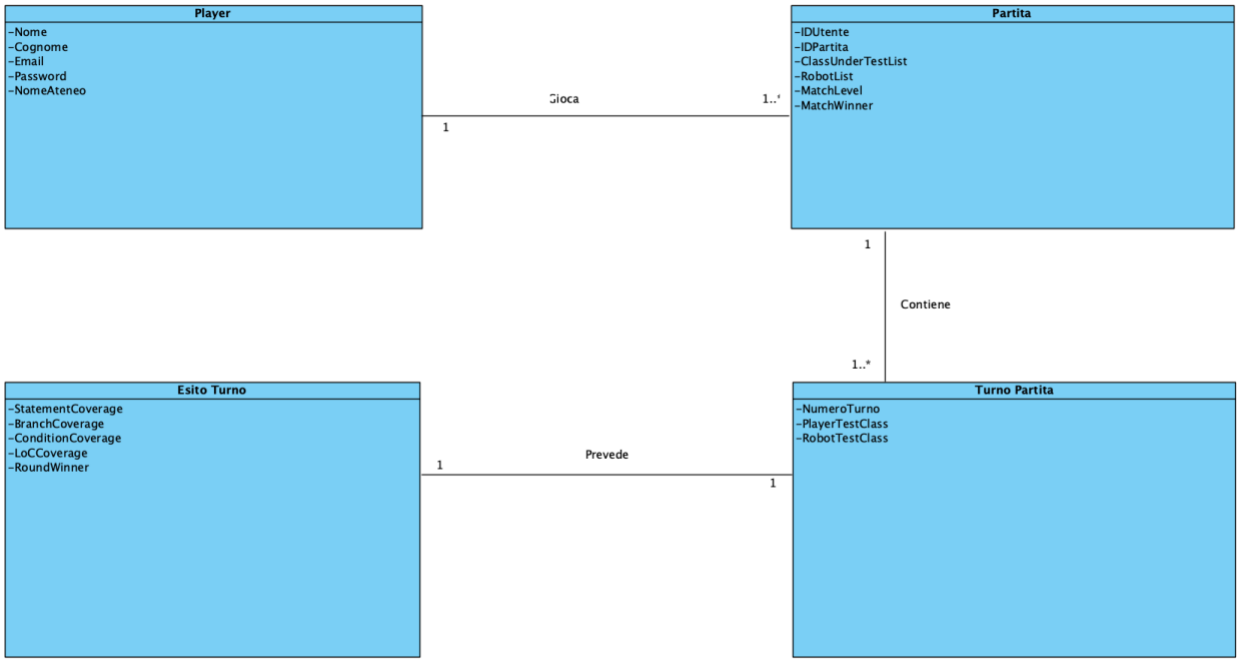

Figura 3.1: System Domain Model

3.2 Architettura Preliminare

L'applicazione creata prevede il caricamento di una pagina HTML lato server (Front-End), che viene lanciata dopo una fase preliminare di autenticazione e avvio della partita da parte dell'utente. Il client effettua, tramite il Web Browser, una richiesta iniziale per ottenere i file HTML, CSS e JavaScript. Nella Figura 3.2 è schematizzata l'architettura preliminare del nostro sistema.

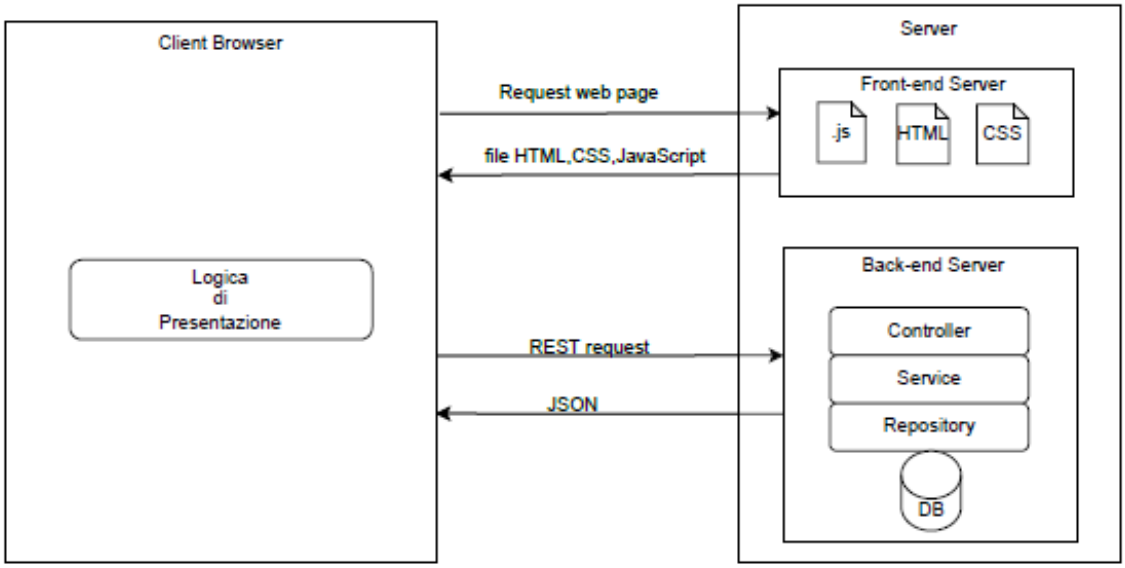

Figura 3.2: Architettura Web

3.3 Diagramma dei Casi d'Uso

La modellazione dei casi d'uso passa, necessariamente, per la contestualizzare del ruolo dell'attore all'interno del sistema che, nel nostro caso, è unico e corrisponde al *Player* che usufruisce dei servizi offerti dall'Editor. La Figura proposta di seguito mostra il diagramma in oggetto.

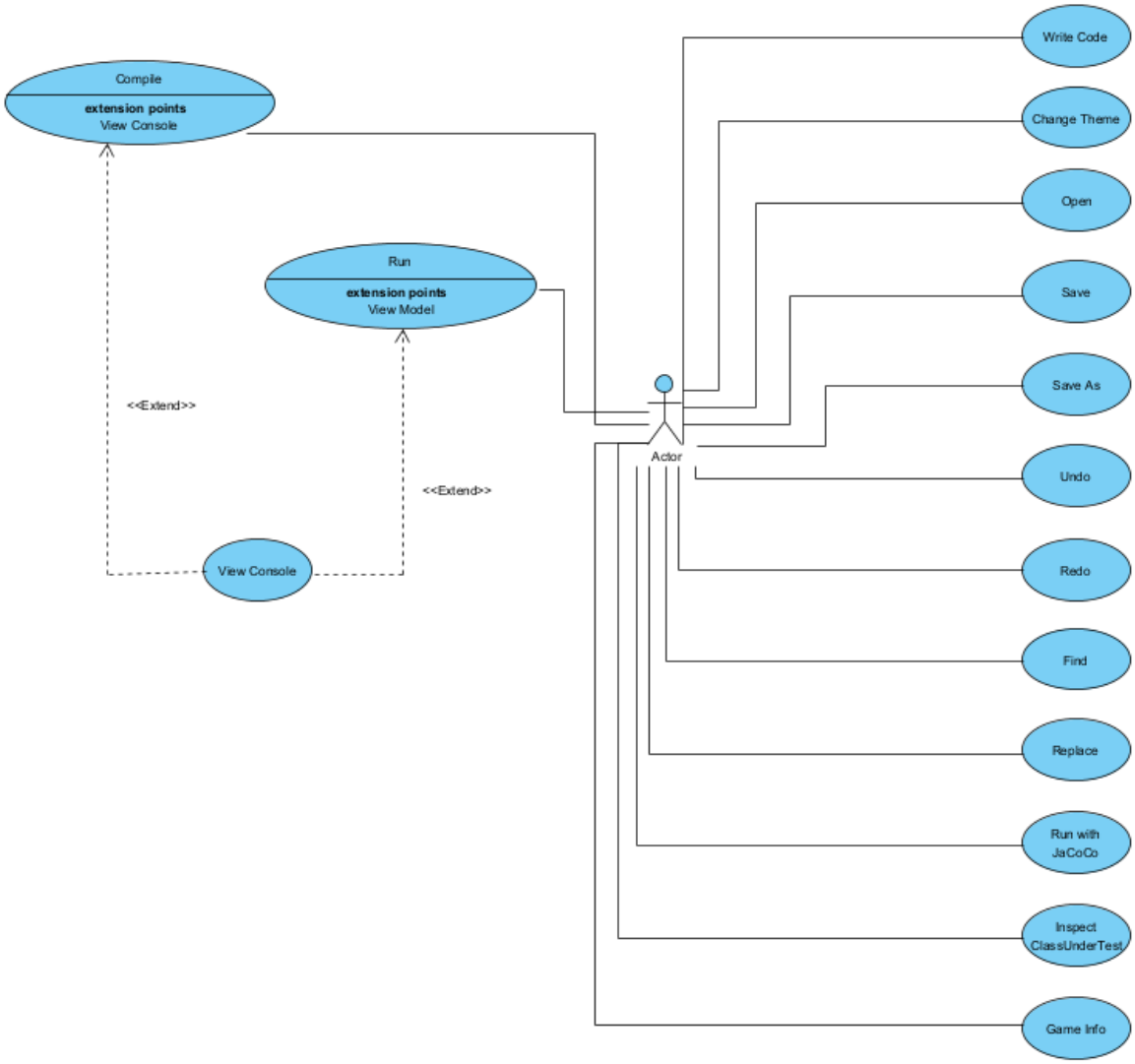

Figura 3.3: Use Case Diagram

3.4 Activity Diagrams

È stato implementato un Activity Diagram relativo ad ogni caso d'uso. Questi risultano utili in fase di analisi per descrivere il flusso delle attività che caratterizzano il caso d'uso in esame.

3.4.1 Change Theme

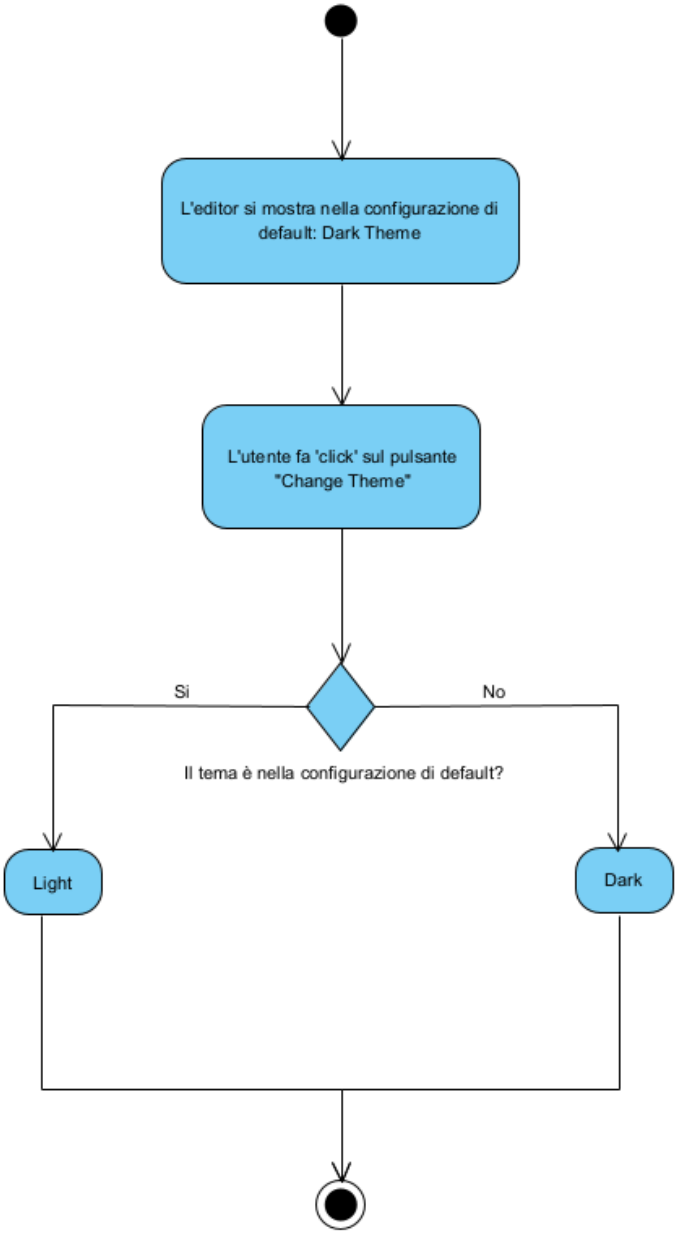

Figura 3.4: Activity Diagram - Change Theme

3.4.2 Open File

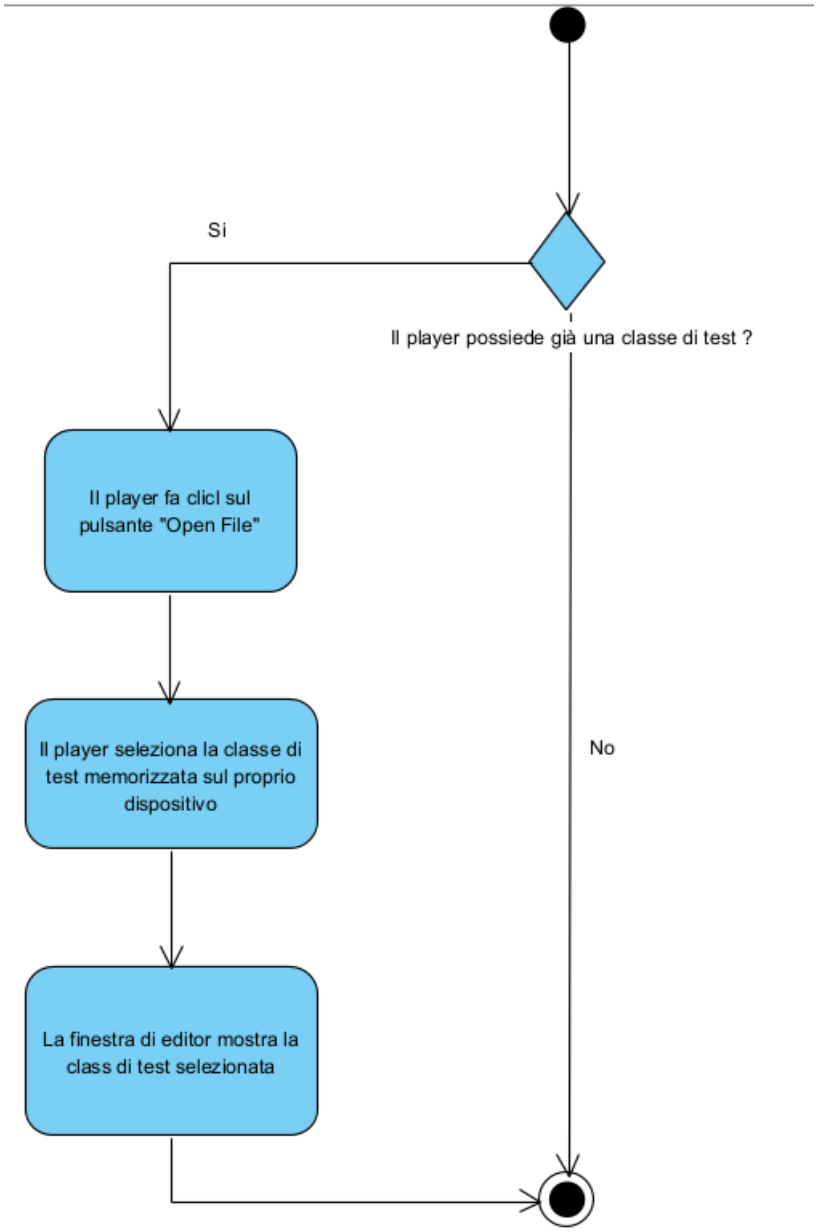

Figura 3.5: Activity Diagram - Open File

3.4.3 Save

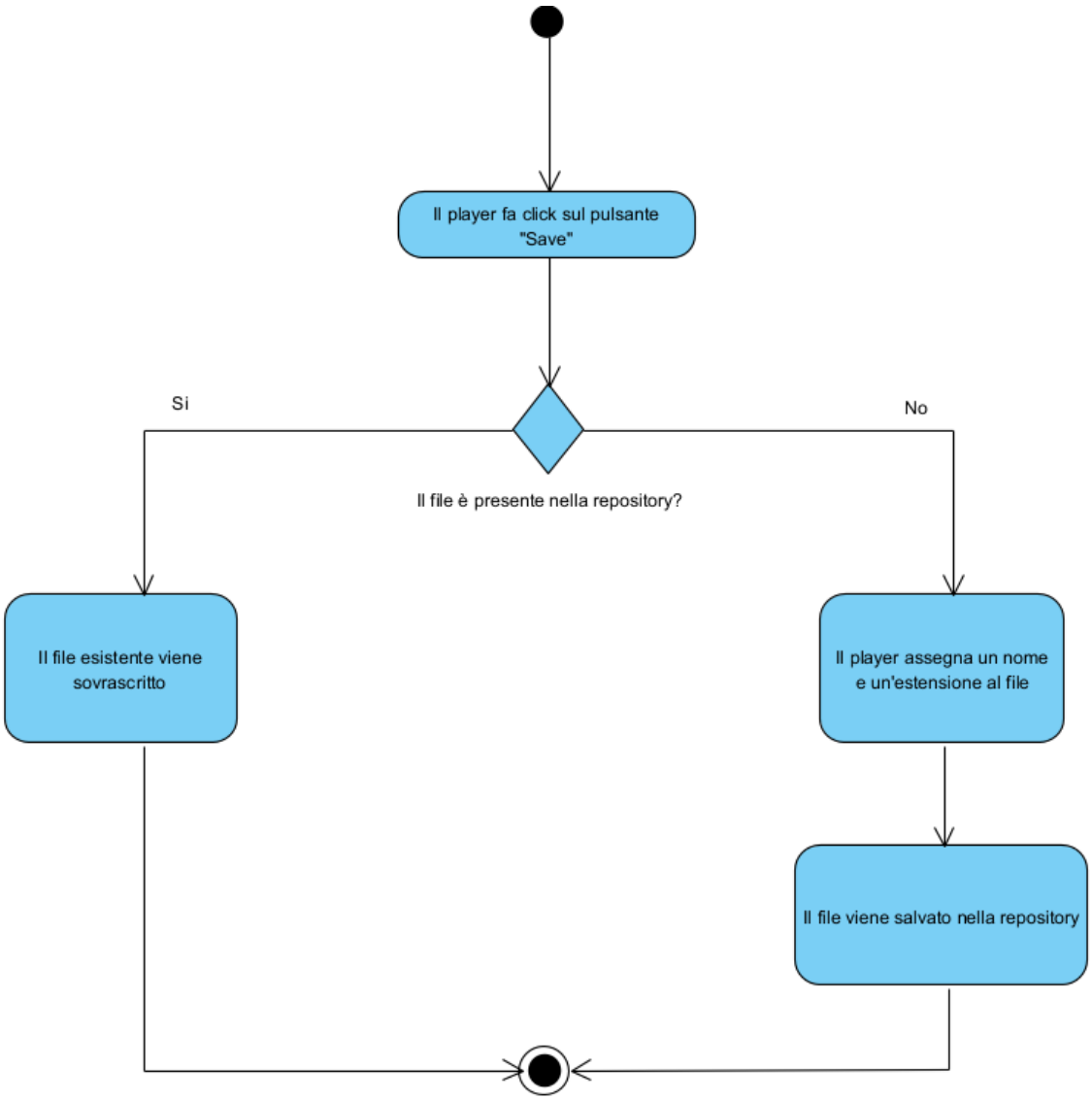

Figura 3.6: Activity Diagram - Save

3.4.4 Save As

Figura 3.7: Activity Diagram - Save As

3.4.5 Undo

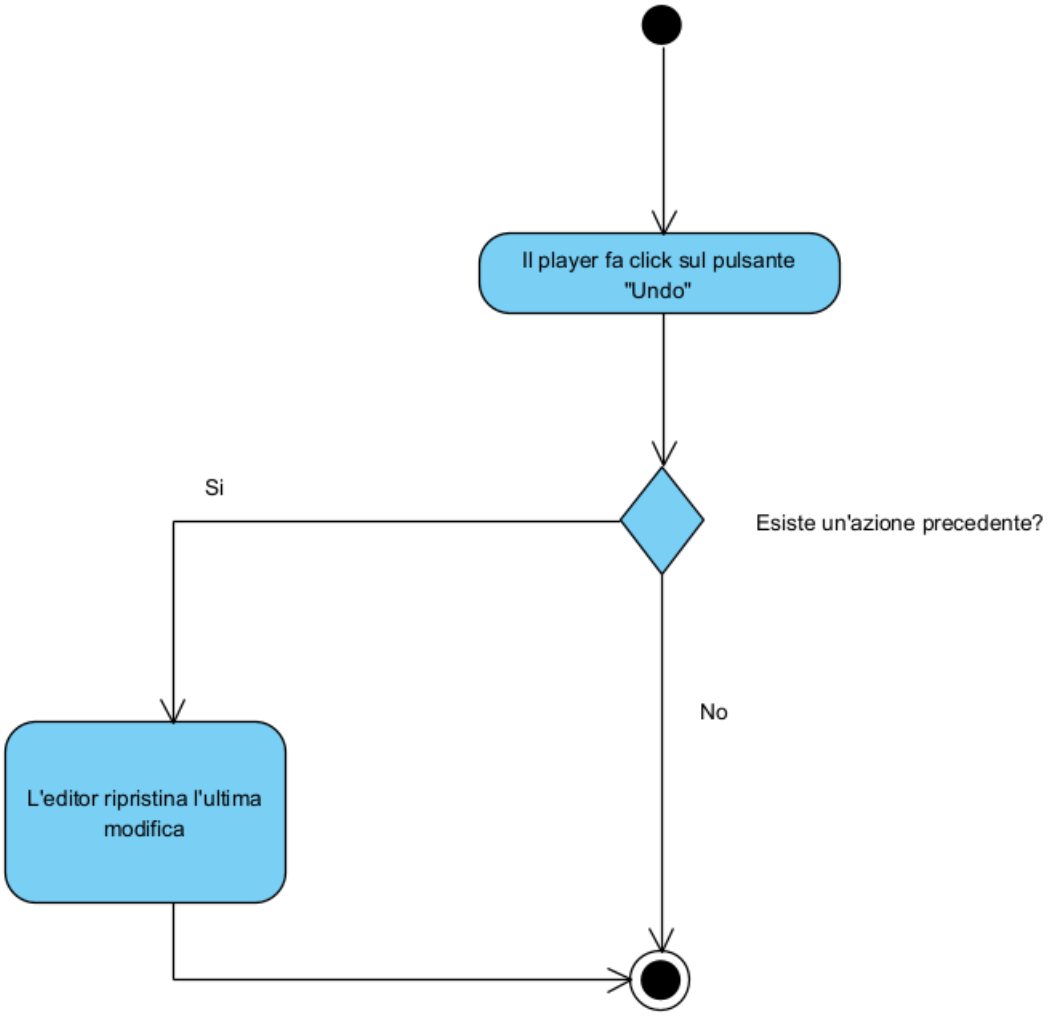

Figura 3.8: Activity Diagram - Undo

3.4.6 Redo

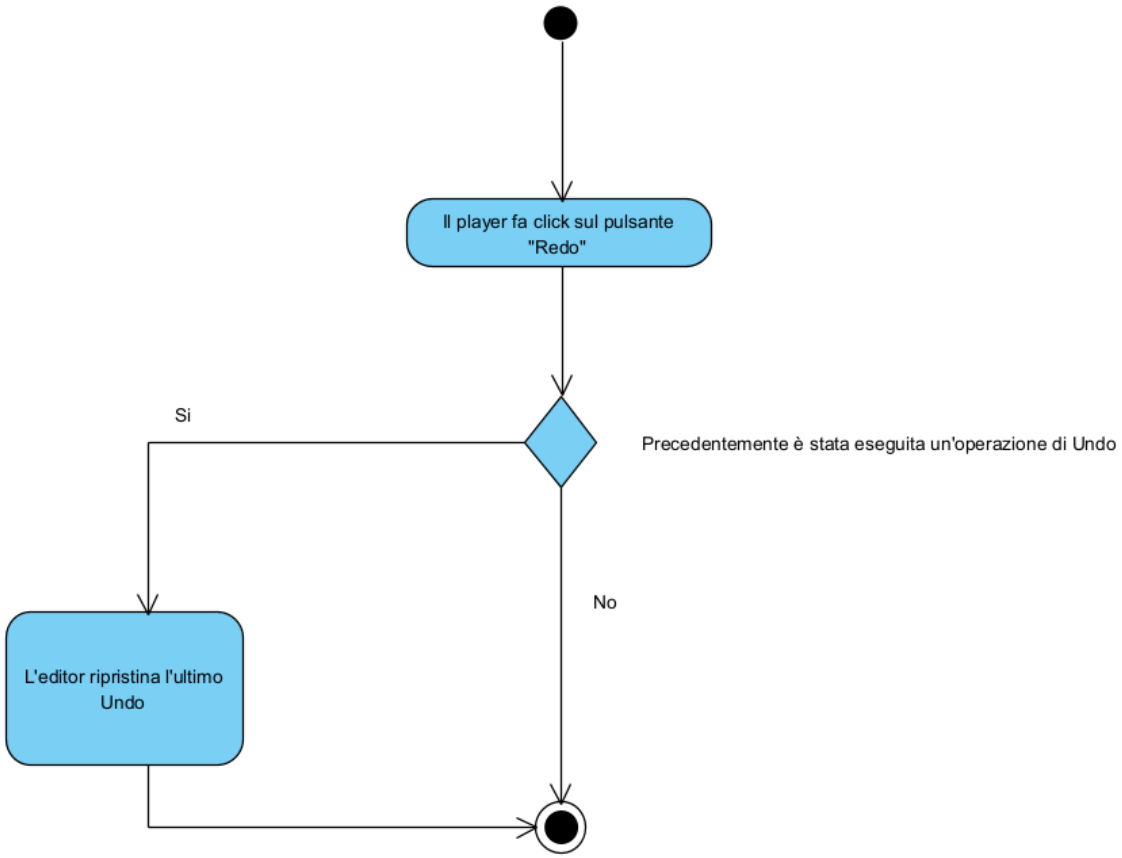

Figura 3.9: Activity Diagram - Redo

3.4.7 Find

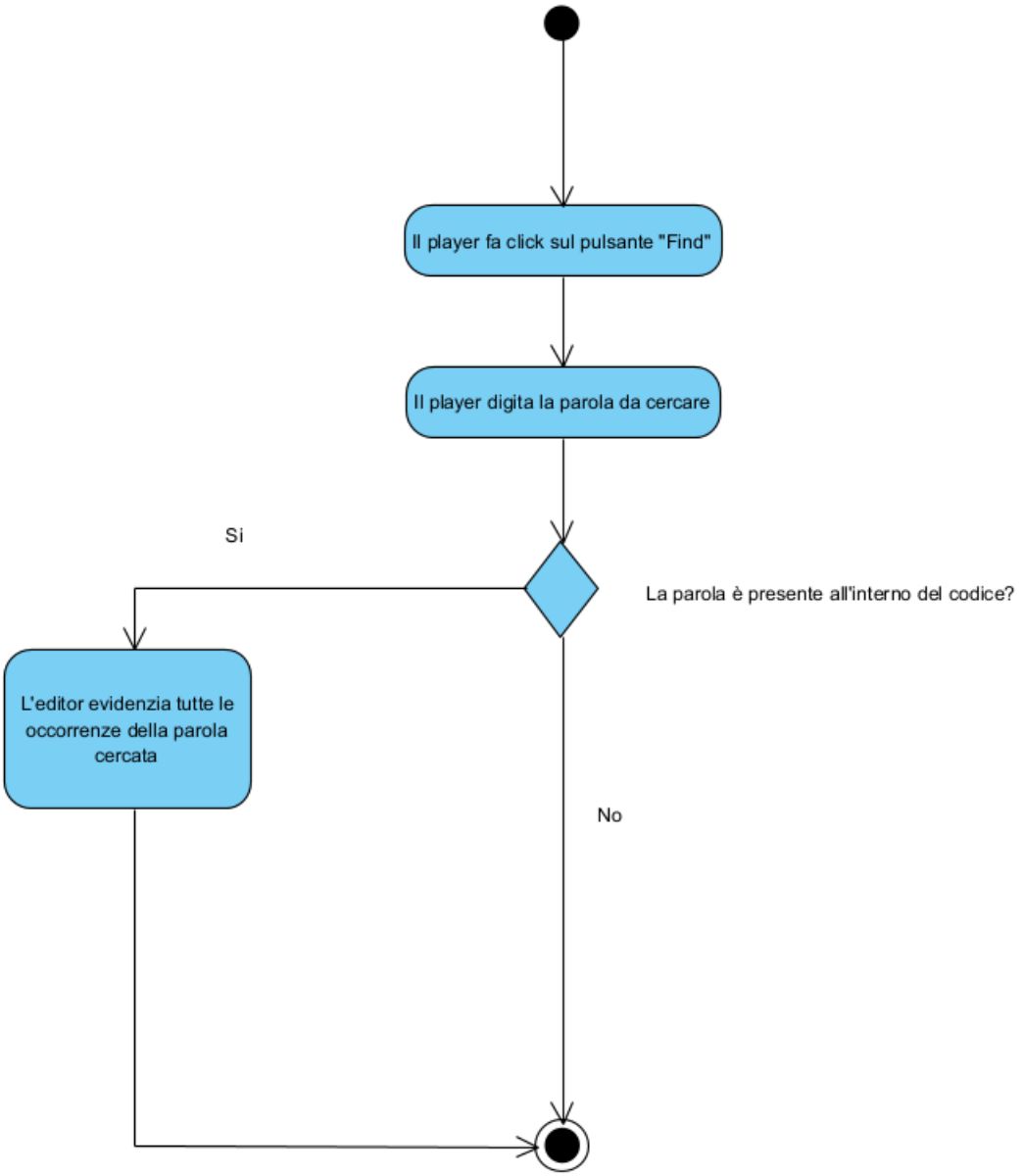

Figura 3.10: Activity Diagram - Find

3.4.8 Replace

Figura 3.11: Activity Diagram - Replace

3.4.9 Compile

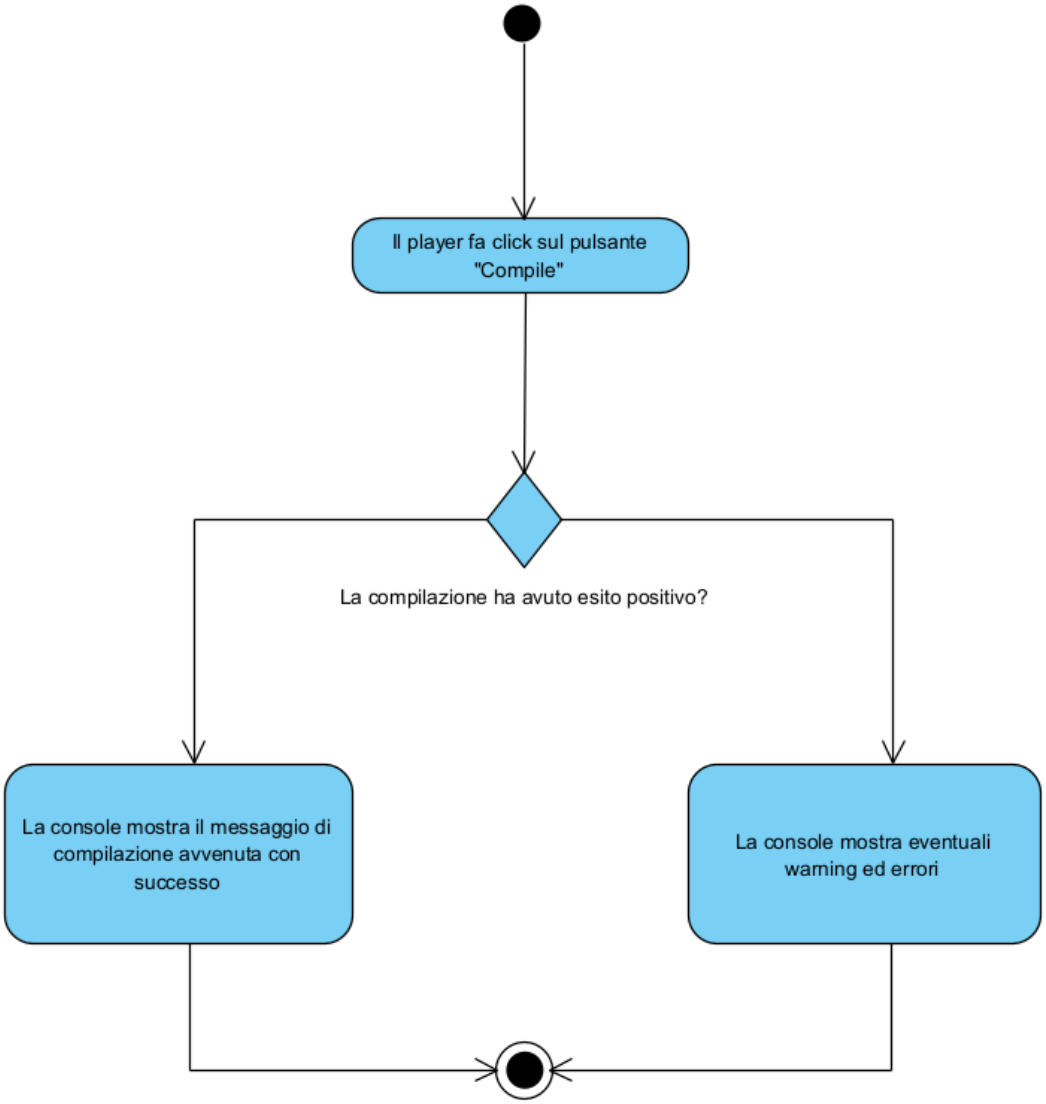

Figura 3.12: Activity Diagram - Compile

3.4.10 Run

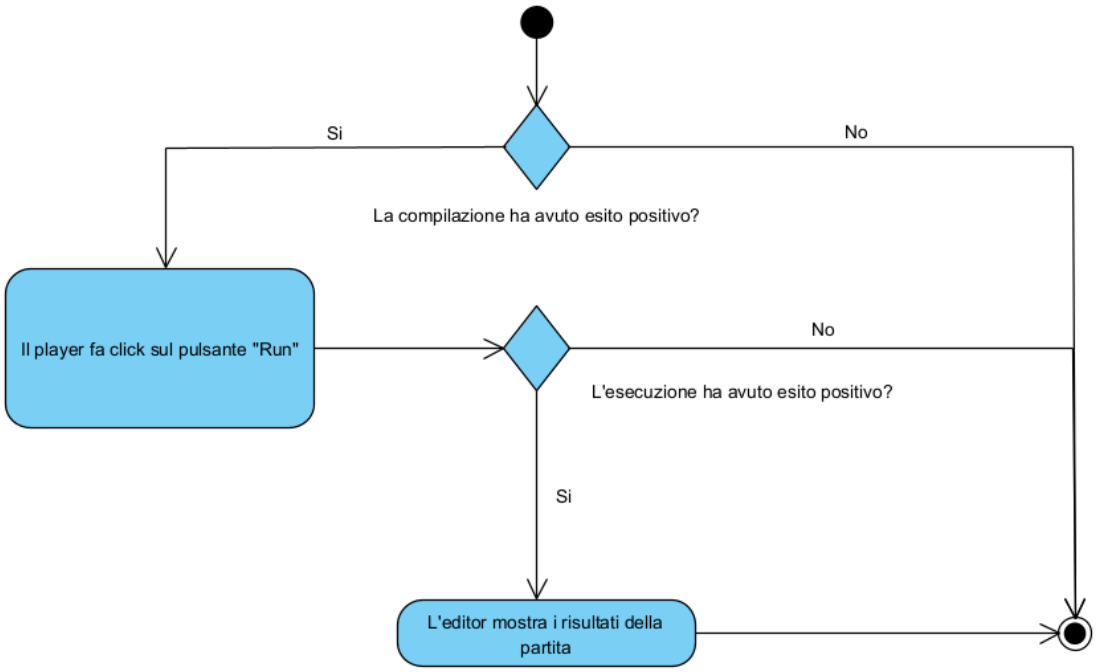

Figura 3.13: Activity Diagram - Run

3.4.11 Run with JaCoCo

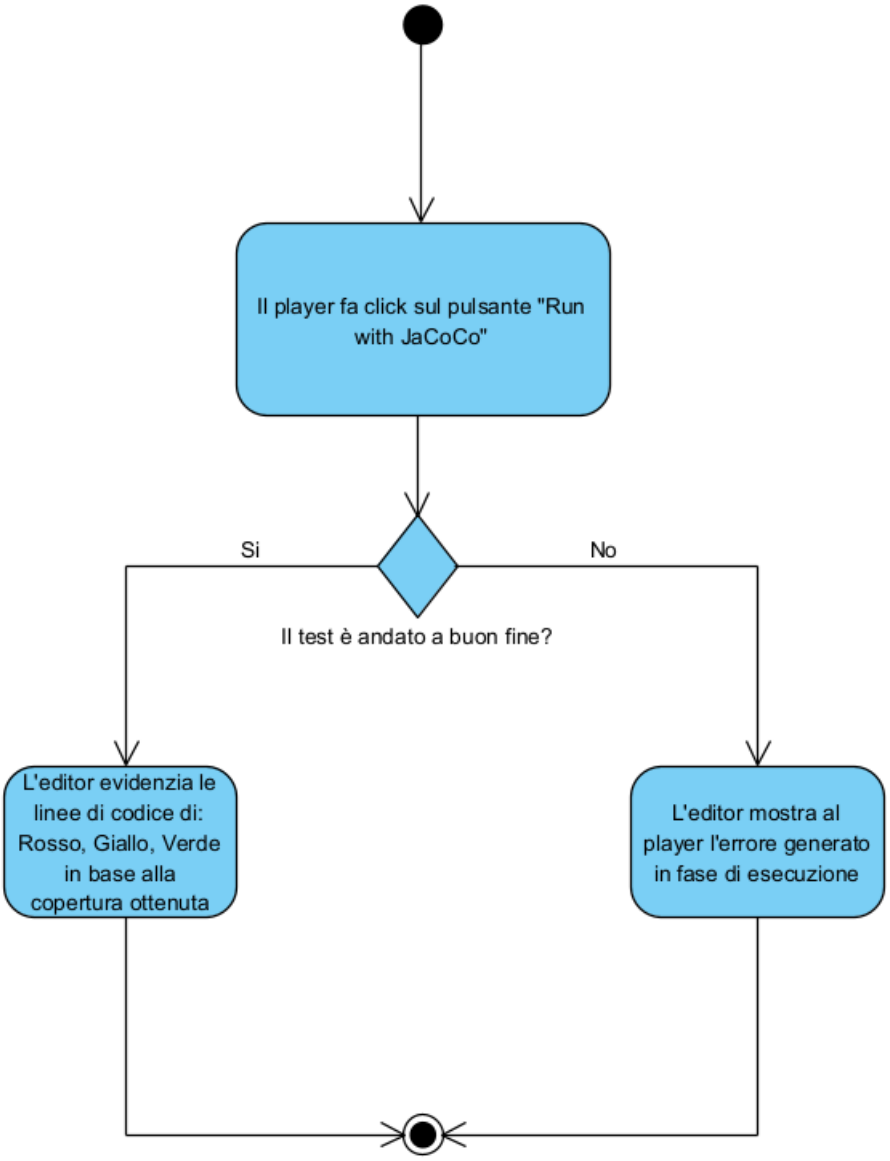

Figura 3.14: Activity Diagram - Run with JaCoCo

3.5 Scenari d'Uso

Per descrivere le sequenze di eventi che occorrono all'innesco di un caso d'uso vengono definiti gli scenari. Nei Sottoparagrafi successivi vengono illustrati tutti i possibili scenari associati ai casi d'uso precedentemente discussi.

3.5.1 Change Theme

Caso D'uso:	***Change Theme***
Attori:	Player
Precondizioni:	
Sequenza degli Eventi:	***Click*** sul pulsante relativo al ***cambio tema*** per cambiare l'aspetto dell'editor
Postcondizioni:	Light → Dark Dark → Light
Sequenza Alternativa:	Il player ***non fa click*** sul pulsante
Postcondizioni Alternative:	Tema di default (Dark)

Tabella 3.1: Scenario d'Uso - Change Theme

3.5.2 Open File

Caso D'uso:	***Open File***
Attori:	Player
Precondizioni:	Il player ***possiede già una classe di test*** per la classe da testare
Sequenza degli Eventi:	***Click*** sul pulsante relativo ***all'upload di un file.java***; ***Upload del file.java*** dal proprio dispositivo
Postcondizioni:	Il file.java viene caricato all'interno della TextArea dell'editor
Sequenza Alternativa:	Il player ***non possiede una classe di test*** già implementata
Postcondizioni Alternative:	Il player crea una classe di test inizialmente vuota

Tabella 3.2: Scenario d'Uso - Open File

3.5.3 Save

Caso D'uso:	***Save***
Attori:	Player
Precondizioni:	***Implementazione*** (parziale o totale) della ***classe di test***
Sequenza degli Eventi:	***Click*** sul pulsante relativo al ***salvataggio del file***
Postcondizioni:	Il file viene sovrascritto
Sequenza Alternativa 1:	Il player ***non fa click*** sul pulsante
Postcondizioni Alternative (1):	I progressi non vengono salvati
Sequenza Alternativa 2 :	Il player ***salva*** il file ***per la prima volta***
Postcondizioni Alternative (2):	Appare una finestra che consente al giocatore di salvare il file con il nome desiderato.

Tabella 3.3: Scenario d'Uso - Save

3.5.4 Save As

Caso D'uso:	***Save As***
Attori:	Player
Precondizioni:	***Implementazione*** (parziale o totale) della ***classe di test***
Sequenza degli Eventi:	Click sul pulsante relativo al ***salvataggio del file con nome ed estensione;*** ***Salvataggio del file.java in locale*** tramite una finestra pop-up;
Postcondizioni:	Il file viene salvato nella cartella Download del proprio dispositivo
Sequenza Alternativa:	Il player ***non fa click*** sul pulsante
Postcondizioni Alternative:	I progressi non vengono salvati

Tabella 3.4: Scenario d'Uso - Save As

3.5.5 Undo

Caso D'uso:	***Undo***
Attori:	Player
Precondizioni:	Esiste un'azione antecedente a quella corrente
Sequenza degli Eventi:	***Click sul pulsante*** relativo all'azione di ***Undo***
Postcondizioni:	Viene ripristinato lo stato del codice antecedente all'ultima modifica
Sequenza Alternativa:	Il player ***non fa click*** sul pulsante
Postcondizioni Alternative:	Il codice non subisce variazione

Tabella 3.5: Scenario d'Uso - Undo

3.5.6 Redo

Caso D'uso:	***Redo***
Attori:	Player
Precondizioni:	Il player ha precedentemente effettuato un'operazione di Undo
Sequenza degli Eventi:	***Click sul pulsante*** relativo all'azione di ***Redo***
Postcondizioni:	Viene visualizzato il codice precedente all'azione di Undo
Sequenza Alternativa:	Il player ***non fa click*** sul pulsante
Postcondizioni Alternative:	Il codice non subisce variazione

Tabella 3.6: Scenario d'Uso - Redo

3.5.7 Find

Caso D'uso:	*Find*
Attori:	Player
Precondizioni:	
Sequenza degli Eventi:	***Click sul pulsante*** relativo all'operazione di ***ricerca***; ***Digitare parola da cercare*** all'interno della finestra pop-up;
Postcondizioni:	La parola (e tutte le sue eventuali occorrenze) viene evidenziata
Sequenza Alternativa:	La ***parola non è presente*** all'interno del codice
Postcondizioni Alternative:	Non viene evidenziato alcun testo

Tabella 3.7: Scenario d'Uso - Find

3.5.8 Replace

Caso D'uso:	*Replace*
Attori:	Player
Precondizioni:	Il player ha ***evidenziato*** la ***parola da sostituire*** all'interno del testo
Sequenza degli Eventi:	***Click sul pulsante*** relativo all'operazione di s***ostituzione***; ***Digitare parola con cui sostituire*** la parola evidenziata;
Postcondizioni:	La parola (e tutte le sue eventuali occorrenze) viene sostituita
Sequenza Alternativa:	Il player ***non fa click*** sul pulsante
Postcondizioni Alternative:	Il codice non subisce variazione

Tabella 2.8: Scenario d'Uso - Replace

3.5.9 Write Code

Caso D'uso:	*Write Code*
Attori:	Player
Precondizioni:	Partita avviata
Sequenza degli Eventi:	Il giocatore ***scrive*** all'interno dell'editor di testo un ***codice Java***
Postcondizioni:	
Sequenza Alternativa:	Il player ***possiede già una classe di test*** e vuole caricarla
Postcondizioni Alternative:	Caricamento da locale della classe di test

Tabella 3.9: Scenario d'Uso - Write Code

3.5.10 Compile

Caso D'uso:	*Compile*
Attori:	Player
Precondizioni:	Classe di test implementata
Sequenza degli Eventi:	***Click sul pulsante*** relativo all'operazione di ***compilazione***
Postcondizioni:	Visualizzazione dell'esito della compilazione
Sequenza Alternativa:	Il player ***non fa click*** sul pulsante
Postcondizioni Alternative:	Il codice non è compilato

Tabella 3.10: Scenario d'Uso - Compile

3.5.11 Run

Caso D'uso:	*Run*
Attori:	Player
Precondizioni:	Classe di test correttamente compilata
Sequenza degli Eventi:	***Click sul pulsante*** relativo all'operazione di ***esecuzione***
Postcondizioni:	Visualizzazione dei risultati all'interno della Console
Sequenza Alternativa:	Il player ***non fa click*** sul pulsante
Postcondizioni Alternative:	Non vengono mostrati in console i risultati della partita

Tabella 3.11: Scenario d'Uso – Run

3.5.12 Run with Jacoco

Caso D'uso:	*Run with Jacoco*
Attori:	Player
Precondizioni:	Classe di test compilata
Sequenza degli Eventi:	Click sul bottone
Postcondizioni:	Colorazione delle linee di codice coperte dalla classe di test attraverso Jacoco
Sequenza Alternativa:	Il player **non fa click** sul pulsante
Postcondizioni Alternative:	Assenza di Test di Copertura con Jacoco

Tabella 3.12: Scenario d'Uso - Run with JaCoCo

3.5.13 Inspect ClassUnderTest

Caso D'uso:	***Inspect ClassUnderTest***
Attori:	Player
Precondizioni:	Scelta della ClassUnderTest dalla Repository
Sequenza degli Eventi:	La Class Under Test viene caricata all'interno dell'editor di testo
Postcondizioni:	La Class Under Test viene visualizzata nella apposita finestra di gioco
Sequenza Alternativa:	
Postcondizioni Alternative:	

Tabella 3.13: Scenario d'Uso - Inspect ClassUnderTest

3.7 Stima dei Costi

Per l'Analisi dei Costi si è scelto di utilizzare il metodo degli *Use Case Points* che permette di stimare dimensione e durata del progetto.

3.7.1 UUCW – Unadjusted Use Case Weight

Calcolando il numero di transazioni di ognuno dei casi d'uso analizzati, è possibile stimarne la complessità ed il peso. La Tabella 3.14 riporta, schematicamente, la *complessità*, il *peso* ed il *numero di transazione* assegnate ad ogni caso d'uso.

CASO D'USO	COMPLESSITÀ	# TRANSAZIONI	PESO
Game Info	Semplice	1	5
Change Theme	Semplice	1	5
Open	Media	3	10
Save	Semplice	2	5
Save As	Media	4	10
Undo	Semplice	1	5
Redo	Semplice	1	5
Find	Media	3	10

Replace	Media	4	10
Compile	Media	4	10
Run	Media	4	10
Run with JaCoCo	Elevata	6	15

Tabella 3.14: UUCW

Moltiplicando i casi d'uso per il loro peso e sommandoli otteniamo il parametro *UUCW*.

$$\boldsymbol{UUCW = 5 * 5 + 6 * 10 + 15 = 100}$$

3.7.2 Fattori di complessità tecnica

I requisiti non funzionali complicano lo sviluppo di un sistema software. La stima dei costi, per affrontare anche questi vincoli, definisce alcuni fattori di complessità tecnica che verranno analizzati nella seguente tabella.

FATTORE	PESO	VALUTAZIONE	IMPATTO
Sistema distribuito	2	3	6
Prestazioni	2	3	6
Efficienza	2	2	2
Elaborazioni complesse	1	2	2
Riusabilità	2	3	3
Facilità d'installazione	0.5	3	1
Facilità d'uso	0.5	2	2
Portabilità	2	2	4
Facilità di cambiamento	1	2	2
Utilizzo concorrente	2	1	2

Sicurezza	1	2	3
Accesso da terze parti	1	1	1
Necessità di aggiornamenti	1	1	1

Tabella 3.15: Report dei fattori di complessità tecnica

Sommando i valori d'impatto di tutti i fattori si ottiene il Tfactor, che nel nostro caso risulta essere

$$\boldsymbol{Tfactor = 35}$$

Effettuiamo adesso il calcolo del Tecnical Complexity Factor:

$$\boldsymbol{TCF = 0.6 + (0.01 * Tfactor) = 0.95}$$

3.7.3 Fattori di complessità dell'ambiente

Eseguiamo adesso il calcolo del peso dei fattori di complessità ambientali.

FATTORE	PESO	VALUTAZIONE	IMPATTO
Familiarità coi processi di sviluppo	1.5	2	3
Esperienza nelle applicazioni	0.5	1	1
Esperienza nella programmazione	1	2	3
Capacità da analista	0.5	2	1
Motivazione	1	4	4
Requisiti di stabilità	2	2	2
Part-time staff	-1	0	0
Linguaggi di programmazione difficili	-1	1	-1

Tabella 3.16: Report dei fattori di complessità dell'ambiente

Sommando i valori d'impatto di tutti i fattori si ottiene l'Efactor, che nel nostro caso risulta essere

$$\boldsymbol{Efactor = 1.4 + (-0.03 * 13) = 1.01}$$

3.7.4 Calcolo finale degli Use Case Points e stima

Moltiplicando tutti i risultati ottenuti è possibile calcolare gli UCP (Use Case Point):

$$\boldsymbol{UCP = UUCW * TCF * EF \ = \ 100 * 0.95 * 1.01 = 95.95}$$

È stato considerato, inoltre, un fattore di 6 ore per use case point per cui le ore di lavoro totali dedicate allo sviluppo del software risultano essere definite dalla seguente relazione:

$$\boldsymbol{UCP \ * \ 6 \ = \ 575.7}$$

Considerato un arco di tempo di 9 settimane e poiché il team è composto da 3 persone, le ore settimanali assegnato ad ogni membro risultano essere circa 22.

Capitolo 4: Documenti di Sviluppo

4.1 Scelte Architetturali

Il pattern architetturale adottato è di tipo *Model – View – Controller* (MVC) applicato ad uno stile architetturale di tipo client – server. Coerentemente alla Figura 3.2 proposta nel Capitolo precedente, il Web Browser, quando richiede una pagina web, contatta il front-end del server ottenendo l'applicazione costituita dai file HTML, CSS e JavaScript che vengono restituiti al Web Browser quando viene richiesta una pagina web. In questo caso, il front-end è sviluppato utilizzando il framework CodeMirror. L'interazione tra il player e l'interfaccia utente (GUI) avviene attraverso l'esecuzione di codice JavaScript all'interno del browser. Questo codice JavaScript è responsabile di gestire gli eventi generati dagli utenti, come ad esempio il clic di un pulsante o la digitazione di testo in un campo di input. Per ottenere o inviare dati al server, il codice JavaScript esegue chiamate REST (REpresentational State Transfer) utilizzando API esposte dal server. Le chiamate REST possono essere utilizzate per recuperare dati da servizi esterni con cui l'Editor deve integrarsi. Il server, nel contesto dell'architettura scelta, gestisce le richieste REST provenienti dal front-end. Il server elabora queste richieste, esegue le operazioni necessarie per ottenere o inviare i dati richiesti e restituisce le risposte al front-end.

Il modello corrisponde alla rappresentazione dei dati e della logica associata ad essi. La vista è responsabile di visualizzare i dati nel modo appropriato e di gestire l'interfaccia utente. Il controllore, infine, gestisce gli eventi generati dagli utenti e coordina l'interazione tra il modello e la vista.

Questa architettura client-server con pattern MVC offre una separazione chiara delle responsabilità e permette un'organizzazione modulare e scalabile del codice, consentendo all'applicazione di essere sviluppata e mantenuta in modo più efficace.

4.2 Confronto con Pattern Architetturali alternativi

Il Team di Sviluppo ha ritenuto necessario giustificare la scelta del pattern architetturale MVC. Di seguito, dunque, viene proposta un'analisi comparativa con i principali pattern architetturali alternativi quali: *Three-Tiered* e *Sense-Compute-Control*.

- **Model-View-Controller (MVC)**
  - **Vantaggi:**
    - Separazione chiara delle responsabilità tra il modello (logica dei dati), la vista (presentazione) e il controller (gestione degli eventi e del flusso dell'applicazione);
    - Maggiore riusabilità del codice grazie alla modularità dei componenti;
    - Consente una gestione più facile delle modifiche e dell'evoluzione dell'applicazione;
  - **Svantaggi:**
    - Può diventare complesso gestire le dipendenze tra i componenti, specialmente in applicazioni di grandi dimensioni;
    - Può richiedere un'attenta progettazione iniziale per definire correttamente le responsabilità dei componenti;
- **Architettura a tre livelli (Three-Tiered)**
  - **Vantaggi:**
    - Separazione chiara delle responsabilità tra il livello *display* (User Interface), il livello di *Business Logic* e il livello di persistenza dei dati *State* (data persistence);
    - Favorisce la modularità e la scalabilità dell'applicazione;
    - Migliora la manutenibilità del codice grazie alla separazione dei livelli;

        - Consente la riusabilità dei componenti, poiché ogni livello è indipendente dagli altri;
    - Svantaggi:
        - Possibile overhead di comunicazione tra i livelli, specialmente in applicazioni complesse;
        - Potenziale duplicazione dei dati tra i livelli, se non gestita correttamente;
- Sense-Compute-Control:
    - Vantaggi:
        - Fornisce una separazione chiara delle responsabilità tra i componenti che catturano i dati iniziali (Sense), eseguono i calcoli o l'elaborazione (Compute) e controllano il flusso dell'applicazione (Controllori o Attuatori);
        - Facilita la gestione dello stato dell'applicazione, poiché il flusso di dati avviene attraverso i componenti sopra individuati;
        - Promuove una progettazione modulare e una migliore gestione delle dipendenze tra i componenti;
    - Svantaggi:
        - Richiede una progettazione attenta per definire correttamente le responsabilità dei componenti;
        - Potrebbe non essere adatto per tutte le applicazioni, specialmente quelle con requisiti complessi o troppo specifici;

Alla luce delle caratteristiche individuate per ognuno dei pattern architetturali, è possibile fornire un'analisi comparativa sui principali punti d'interesse:

- **Separazione delle responsabilità:** MVC offre una separazione chiara tra modello, vista e controller. Three-Tiered separa l'applicazione in tre livelli distinti: presentazione, business logic e persistenza dei dati. Sense-Compute-Control, infine, separa le responsabilità in componenti di acquisizione dei dati (Sense), elaborazione (Compute) e controllo del flusso (Controllori o Attuatori).
- **Modularità:** Tutti e tre i modelli promuovono la modularità del codice e consentono la riusabilità dei componenti.
- **Gestione dello stato:** MVC richiede una gestione esplicita dello stato, poiché il controller gestisce il flusso dei dati tra modello e vista. Three-Tiered può gestire lo stato attraverso il livello di business logic o utilizzando un framework esterno. Sense-Compute-Control, invece, fornisce una gestione dello stato più esplicita, in quanto il flusso dei dati avviene attraverso i componenti Sense, Compute e Control.
- **Complessità:** MVC e Three-Tiered possono entrambi diventare complessi in applicazioni di grandi dimensioni, richiedendo un'adeguata gestione delle dipendenze e delle interfacce tra i componenti. Sense-Compute-Control richiede una progettazione attenta per definire correttamente le responsabilità dei componenti, ma può ridurre la complessità gestendo in modo più esplicito il flusso dei dati.

Per i motivi sopra elencati e al fine di allineare il modello architetturale scelto dal Team con il modello scelto dagli altri Team coinvolti nel progetto, è stato scelto il pattern MVC.

4.3 Framework Spring

4.3.1 Spring Back-End

Spring è un framework Java ampiamente utilizzato per lo sviluppo di applicazioni enterprise. Si basa su una struttura modulare e offre una serie di funzionalità fondamentali attraverso il modulo Core. Queste funzionalità includono la gestione delle transazioni, la struttura per le applicazioni web e l'accesso ai dati. Due principi chiave su cui si basa il framework Spring sono l'*Inversion of Control* (IoC) e la *Dependency Injection* (DI).

- *Inversion of Control* (IoC) delega al framework la responsabilità della gestione del flusso di controllo all'interno dell'applicazione. Ciò significa che il framework si occupa della creazione degli oggetti, dell'inizializzazione e dell'invocazione dei metodi. Questo permette agli sviluppatori di concentrarsi sulla business logic dell'applicazione senza dover preoccuparsi degli aspetti tecnici della gestione degli oggetti.
- *Dependency Injection* (DI) è una specifica implementazione dell'IoC all'interno di Spring. La DI prevede che gli oggetti all'interno dell'applicazione accettino le loro dipendenze (ovvero gli altri oggetti di cui hanno bisogno) tramite il costruttore o i metodi setter. Invece di creare le loro dipendenze, gli oggetti ricevono le dipendenze dall'esterno. Questo approccio permette una maggiore modularità e facilità di gestione delle dipendenze all'interno dell'applicazione.

Gli oggetti istanziati all'interno di Spring vengono chiamati *bean* e vengono dichiarati all'interno del progetto. L'*IoC Container* di Spring si occupa di reperire i bean dichiarati e di iniettare tutte le dipendenze associate ad essi. Questo offre un alto livello di flessibilità e facilità di configurazione delle dipendenze all'interno dell'applicazione.

Menzioniamo, per completezza, altre caratteristiche importanti che rendono Spring un Framework largamente diffuso:

- Supporto per diversi livelli di astrazione;
- Integrabilità con altri framework e tecnologie come Hibernate o JPA (Framework di Persistenza), JUnit o Mockito (Framework di Testing) e Angular o React (Tecnologie di Front-End);
- Semplice e robusto supporto per la gestione delle transazioni;

In seno alla breve analisi proposta, dunque, possiamo affermare che l'utilizzo di Spring permette una migliore gestione delle dipendenze, una maggiore modularità del codice e una maggiore facilità di sviluppo e manutenzione delle applicazioni Java.

4.3.2 Spring Boot

Spring Boot è un'estensione del framework Spring che mira a semplificare lo sviluppo delle applicazioni Spring e velocizzarne il processo di configurazione. Una delle caratteristiche chiave di Spring Boot è l'autoconfigurazione che riduce, notevolmente, la quantità di codice di configurazione necessario. Questo significa che molte delle configurazioni standard vengono gestite automaticamente dal framework, consentendo, così, agli sviluppatori di concentrarsi maggiormente sulla logica dell'applicazione. Nella versione precedente di Spring, infatti, era necessario configurare manualmente i bean, definire i file di configurazione XML o utilizzare le annotazioni per dichiarare i componenti dell'applicazione. Con l'introduzione di Spring Boot, invece, gran parte di questa configurazione è gestita automaticamente. Il framework include una serie di classi di autoconfigurazione che identificano le dipendenze necessarie all'interno del progetto e creano automaticamente i bean corrispondenti. Uno dei bean più importanti all'interno di Spring Boot è il DispatcherServlet. Il DispatcherServlet è un componente centrale all'interno dell'architettura delle applicazioni web basate su Spring ed è responsabile dell'indirizzamento delle richieste HTTP in

ingresso a tutti gli altri controller presenti all’interno dell'applicazione. E’ importante osservare, infine, che esso viene automaticamente configurato e gestito dal framework per cui gli sviluppatori saranno esonerati dalla sua configurazione. Questo semplifica notevolmente lo sviluppo delle applicazioni web e riduce la complessità dell'architettura.

4.3 Libreria CodeMirror

La realizzazione dell'Editor, in linea con le assegnazioni fornite al Team, ha previsto l'utilizzo di *CodeMirror:* una libreria JavaScript open-source che consente di realizzare un editor di testo altamente personalizzabile per l'integrazione nelle applicazioni web. L'analisi del Framework condotta nella fase iniziale del progetto ha permesso al Team di individuare i seguenti punti chiave che lo caratterizzano:

- **Funzionalità avanzate di editing:** CodeMirror offre un'ampia gamma di funzionalità di editing come: evidenziazione della sintassi, completamento automatico, indentazione intelligente, ricerca e sostituzione, selezione multipla, evidenziazione delle parentesi e molto altro ancora. Queste funzionalità (implementate all'interno del progetto) permetteranno al *player* di scrivere codice in modo più efficiente e accurato, migliorando la produttività e riducendo gli errori;
- **Personalizzazione:** CodeMirror è altamente personalizzabile e offre molte opzioni di configurazione. È possibile personalizzare lo schema dei colori, i temi, i comportamenti di indentazione e molto altro ancora;
- **Supporto per diversi linguaggi di programmazione:** CodeMirror supporta una vasta gamma di linguaggi di programmazione e markup, tra cui JavaScript, HTML, CSS, Python, Java e molti altri. La libreria offre evidenziazione della sintassi per i linguaggi supportati, consentendo agli sviluppatori di visualizzare il codice in modo chiaro e di individuare facilmente gli errori di sintassi;
- **Integrabilità:** CodeMirror può essere facilmente integrato in diverse applicazioni web. Fornisce un'API semplice e ben documentata che consente agli sviluppatori di interagire con l'editor in modo programmatico.

4.3.1 Class Diagram – Graphic User Interface

Il Team ha ritenuto opportuno dettagliare le principali classi che costituiscono l'interfaccia utente e le dipendenze che vi intercorrono. A tal proposito, dunque, è stato prodotto un Class Diagram che viene riportato nella Figura 4.1. Si consiglia, tuttavia, di visualizzare il diagramma in oggetto tramite il progetto Visual Paradigm in modo da poterlo consultare con maggior chiarezza. Il Class Diagram proposto prevede:

- **Editor GUI:** è una classe che modella l'intera interfaccia della pagina Web. Nello specifico, esso rappresenta il container più esterno che, a sua volta, è costituito dalle seguenti finestre:
    - **ConsoleWindow:** la seguente classe permette di modellare la finestra adibita alla visualizzazione di eventuali *errors* o *warnings* commessi dal *player* e rilevati dal servizio di compilazione. Tale finestra ha una relazione di *composizione* con la classe *EditorGUI* e richiama, a sua volta, la seguente classe:
        - **Console**: permette di modellare il contenuto della classe ConsoleWindow e, per tal motivo, ha una relazione di *aggregazione* con la suddetta classe;
    - **EditorTextArea:** la seguente classe permette di modellare la finestra di editing. Presenta una relazione di *composizione* con la classe *EditorGUI* e si compone, a sua volta, delle seguenti classi:
        - **CodeMirror:** permette di modellare l'utilizzo della libreria Javascript *CodeMirror* e prevede una relazione di *composizione* con la classe *EditorTextArea;*
        - **Theme:** individuata per l'implementazione del pulsante *ChangeTheme*, la seguente classe contiene in sé tutti i temi previsti da *CodeMirror.* Osserviamo, infine, che essa rispetta una relazione di *aggregazione* con la classe *EditorTextArea;*

- **ReportWindow:** la seguente classe consente di modellare la finestra adibita alla visualizzazione dei risultati della partita tra il *player* e il Robot scelto. Essa prevede, così come le finestre precedenti, una relazione di *composizione* con la classe *EditorGUI* e si caratterizza, a sua volta, per la seguente classe:
  - **Report:** permette di modellare il contenuto che viene mostrato all'interno della finestra di report. Per tale classe, dunque, è stata individuata una relazione di *aggregazione* con la classe *ReportWindow;*
- **ClassUnderTestWindow:** consente la modellazione della finestra in cui viene caricata la *ClassUnderTest* scelta dal *player* in fase di inizializzazione della partita contro il Robot. Essa osserva una relazione di *composizione* con la classe *EditorGUI* ed è caratterizzata, a sua volta, da:
  - **ClassUnderTest:** permette di modellare il contenuto della finestra sopra analizzata e presenta una relazione di *aggregazione* con la classe *ClassUnderTestWindow;*

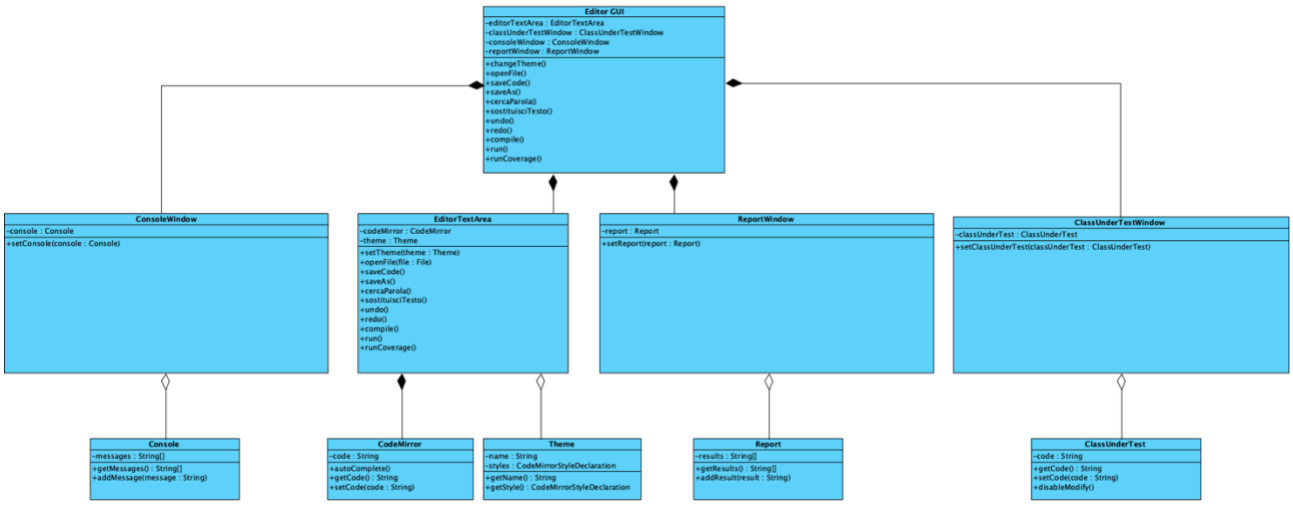

Figura 4.1: Class Diagram per la GUI

4.4 Sequence Diagram

Per dettagliare ulteriormente le operazioni svolte dalle varie funzionalità, si presentano alcuni diagrammi di sequenza. Il Team ha scelto di sviluppare quelli che si ritenevano più simbolici e necessari.

4.4.1 Find

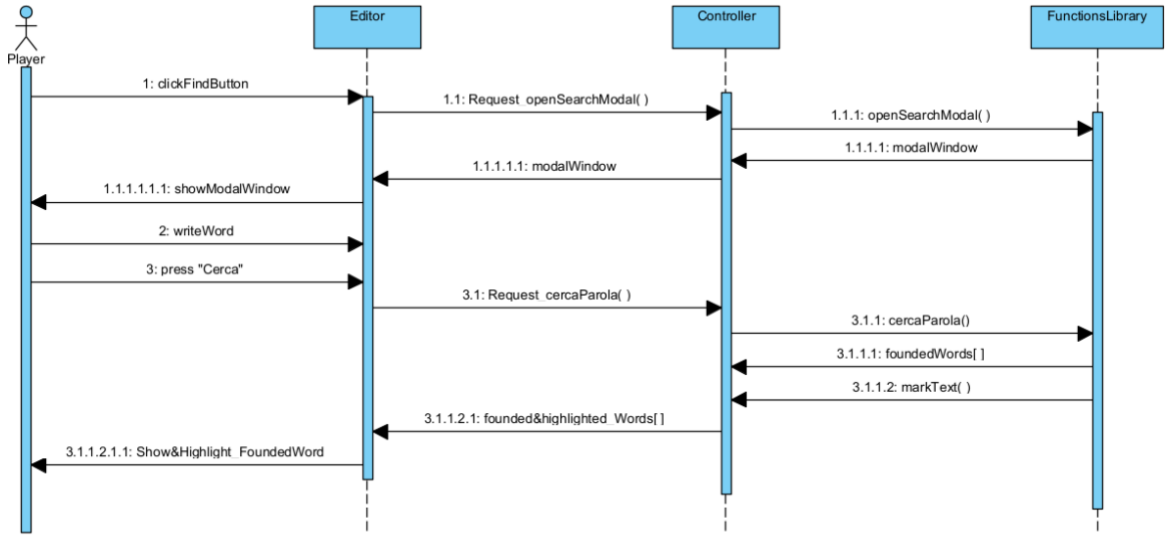

Figura 4.2: Sequence Diagram - Find

4.4.2 Compile

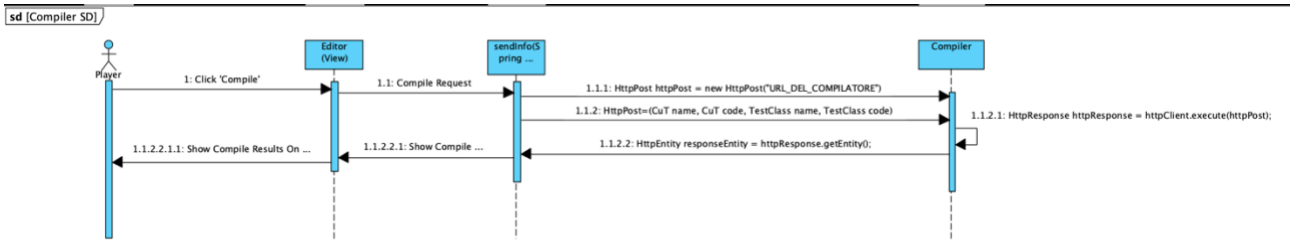

Figura 4.3: Sequence Diagram - Compile

4.4.3 Run

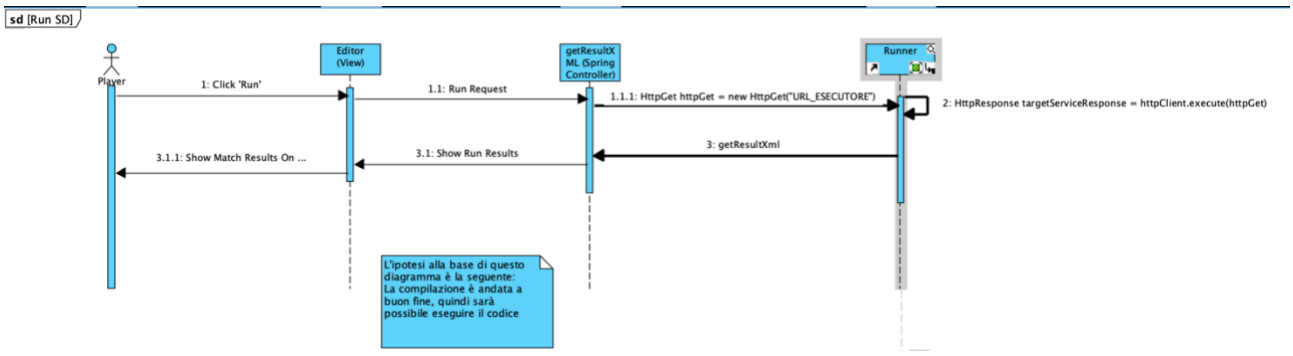

Figura 4.4: Sequence Diagram -.Run

4.4.4 Run with JaCoCo

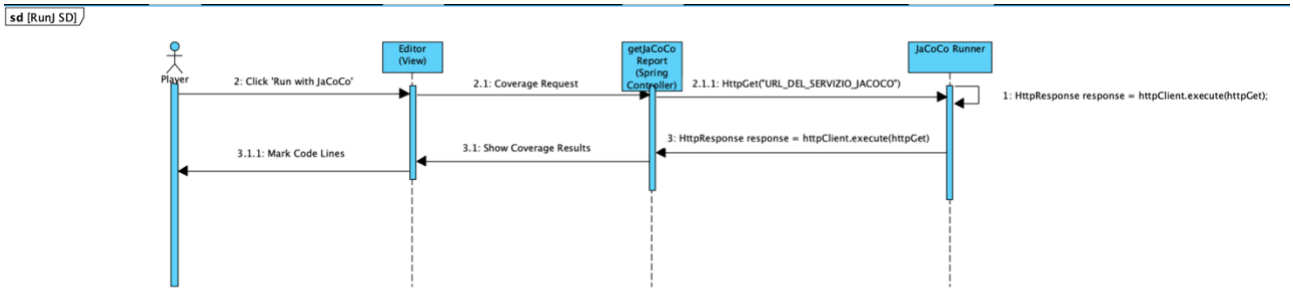

Figura 4.5: Sequence Diagram – Run with JaCoCo

4.4.5 Autocomplete function

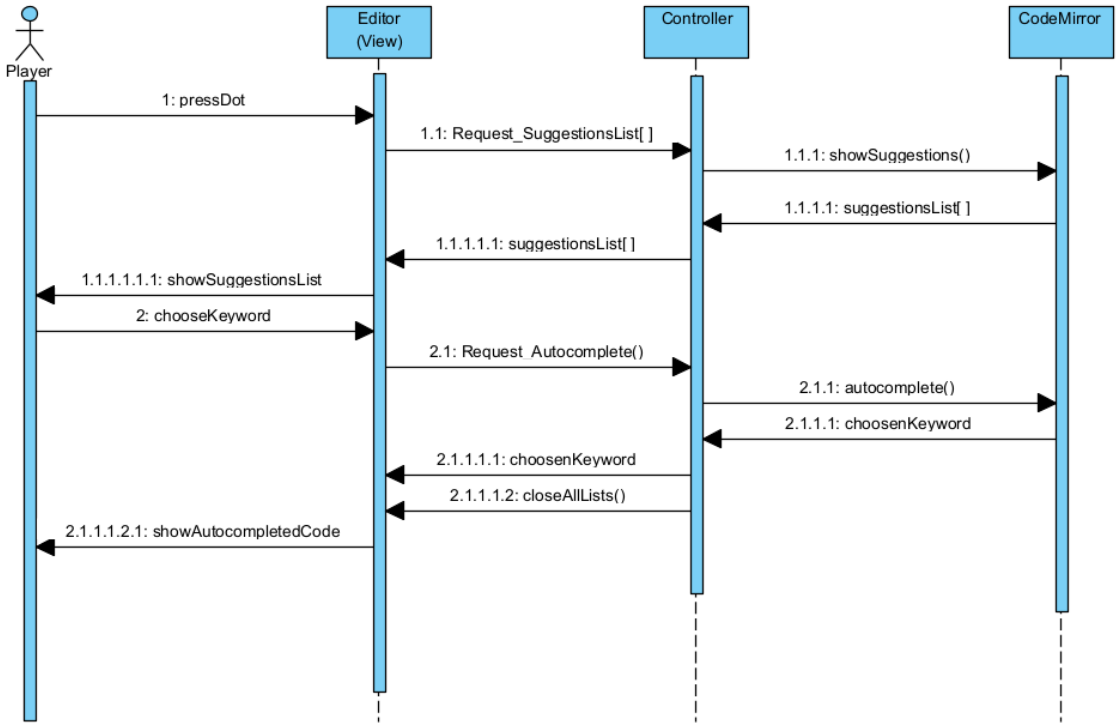

Figura 4.6: Sequence Diagram - Autocomplete

4.5 Component Diagram

Questo diagramma mostra la gerarchia e le dipendenze dell'Editor con gli altri componenti del progetto. La Figura mostra un'entità chiamata *GameInfo* che invoca l'*Editor* che mostra la propria socket per l'interazione. L'entità in esame contiene, al suo interno, tutte le informazioni necessarie allo svolgimento della partita quali: *UserID*, *GameID*, *ClassUnderTest* scelta, *numero del turno* della partita, etc…

Allo stesso modo il componente *Editor*, attraverso due socket, sfrutta i servizi di compilazione ed esecuzione messi a disposizione dal componente *Compile&Run* tramite i due rispettivi lollipop. Nella fattispecie, l'Editor manda al componente invocato sia la classe di test prodotta dal player che la classe da testare. Dal componente invocato riceverà, dunque, un file contenente i risultati da mostrare, rispettivamente, nella finestra *Confronto Risultati* (risposta del Runner) e nella finestra *Console* (risposta del Compiler). Medesimo discorso può essere esteso all'interazione che vive tra l'Editor ed il componente *Evosuite&Randoop Robot* da cui l'Editor riceve i risultati (*XML Files*) dei test eseguiti dai suddetti tool di generazione automatica di test. Vi è, infine, l'interazione tra l'Editor e il componente *MantenimentoDatiPartita* a cui, l'Editor, vengono inviate le informazioni utili alla memorizzazione dei dati della partita giocata.

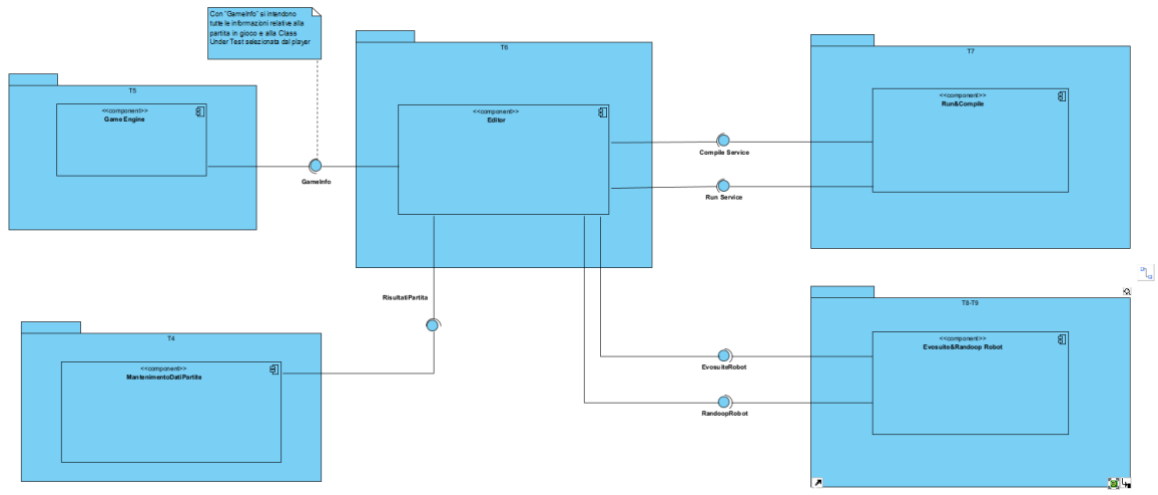

Figura 4.7: Component Diagram

4.6 Package Diagram

La Figura 4.8 mostra i Package individuati per il compito assegnato al Team. Sono stati realizzati, a tal proposito, packages che consentono di modellare l'*Editor*, il *Code,* i servizi offerti dal Framework *Spring* e gli *External Services* con cui il Team prevede di collaborare.

Il package *Editor* si struttura, coerentemente a quanto mostrato in Figura, di un package *Windows* in cui sono modellate le quattro finestre che compongono l'editor e di un package *Buttons* in cui sono riportati tutti i pulsanti presenti all'interno della *barra degli strumenti* dell'Editor.

Il package in oggetto viene realizzato (da qui deriva lo stereotipo <<*implement by*>>) dal package *Code* caratterizzato dai linguaggi di programmazione utilizzati per la realizzazione della GUI e, naturalmente, dalla libreria di *CodeMirror.*

L'Editor, inoltre, utilizza (stereotipo <<*use*>>) i servizi di *Spring* per realizzare l'interfacciamento con gli *External Services*. Il package *Spring* si compone di un modulo Java relativo ai Controller che consentono l'interazione con i vari servizi individuati e di un secondo modulo Java (*mainController*) che permette l'avvio del Framework.

Gli *External Services* individuati, coerentemente ed in linea con quanto già descritto nel *Component Diagram*, sono: servizio di *compilazione* ed *esecuzione*, servizio di *avvio partita*, servizio di *mantenimento dei dati della partita* e, infine, i servizi che offrono i risultati dell'esecuzione dei test da parte del *robot* scelto in fase di inizializzazione del match.

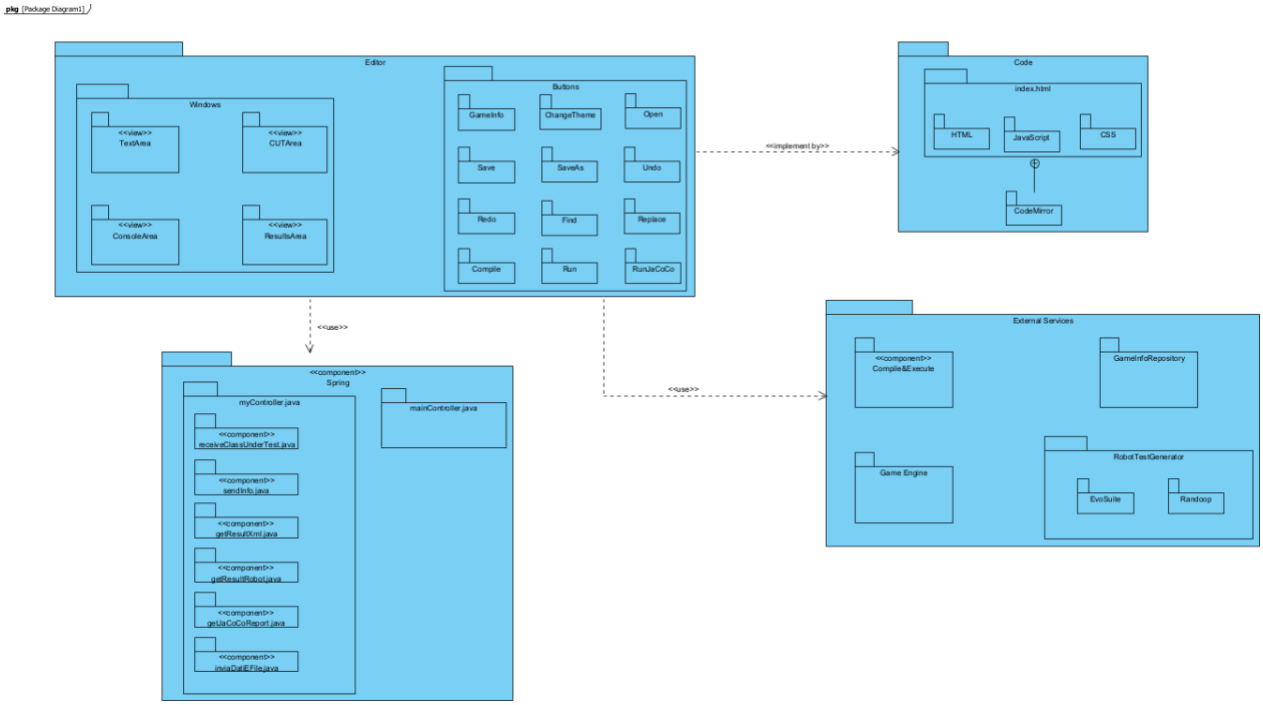

Figura 4.8: Package Diagram

4.7 Deployment Diagram

Nel diagramma proposto nella Figura seguente è mostrata la configurazione di deploy del sistema. Esso contiene, oltre alla libreria CodeMirror, la cartella contenente il progetto Spring. Quest'ultimo conserva i file necessari per l'interfacciamento con gli altri servizi previsti dal progetto. Sono anche presenti dei file .png che saranno utilizzati come icone per i pulsanti creati all'interno della pagina HTML dell'Editor. I codici HTML, Javascript e CSS sono contenuti nel file index.html che, insieme al resto del sistema, viene "*deployato*" su un Web Server Apache che prevederà, inoltre, l'interfacciamento con un Web Browser che, a sua volta, consentirà all'utente di visualizzare correttamente l'applicazione realizzata.

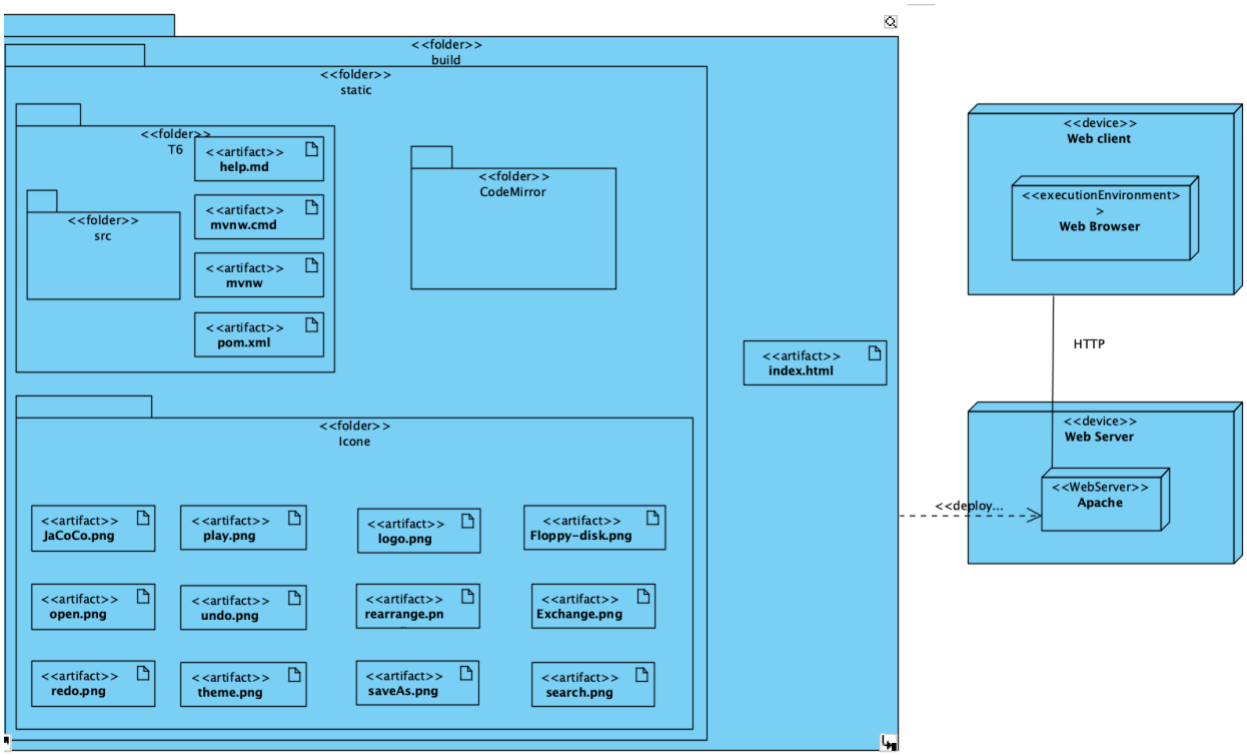

Figura 4.9: Deployment Diagram

Capitolo 5: Sviluppi Futuri

Il Progetto realizzato, dal punto di vista del Team, presenta ancora margini di miglioramento. Sarebbe auspicabile, infatti, che la gestione delle dipendenze individuate possa essere migliorata al fine di realizzare un servizio di Editing ancor più competitivo al suo rilascio. Il Team ritiene che la GUI realizzata possa offrire all'utente un'esperienza di Testing piacevole ed intuitiva. E' auspicabile, inoltre, che la libreria portante del progetto – *CodeMirror* – possa avere, in futuro, delle release che permettano il continuo aggiornamento dell'Editor realizzato.

La Figura 5.1, in conclusione, mostra la GUI relativa alla data di consegna del progetto.

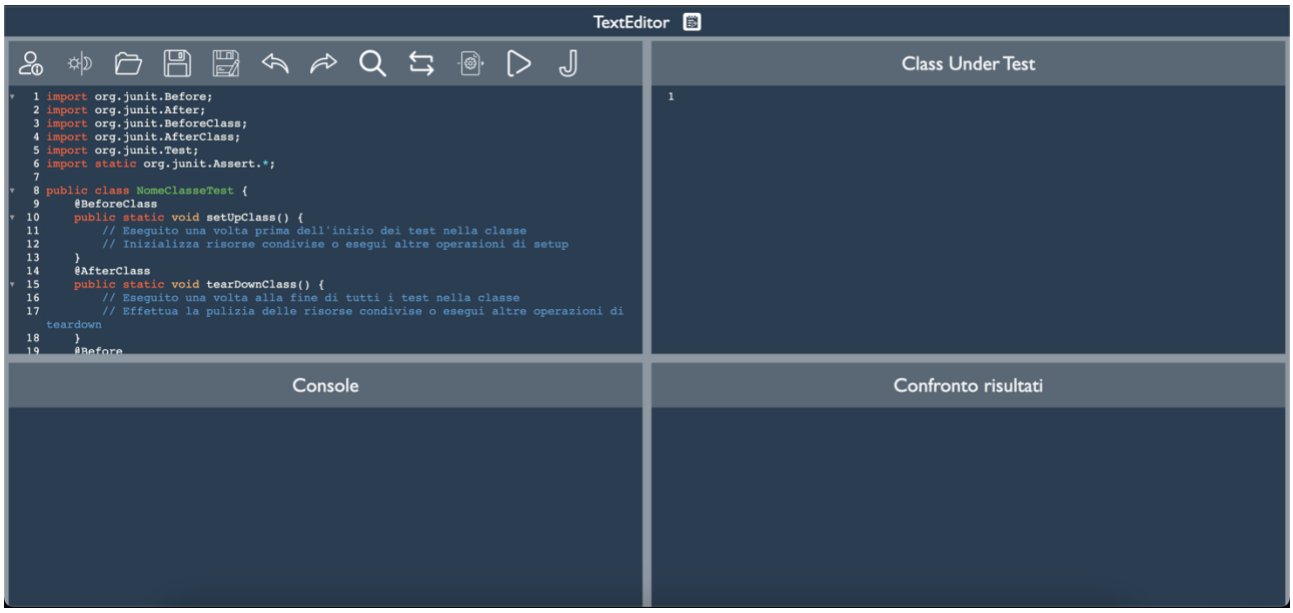

Figura 5.1: GUI TextEditor

Indice delle Figure

Indice delle Tabelle