

UNIVERSITÀ DEGLI STUDI DI NAPOLI
FEDERICO II

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

CORSO DI ARCHITETTURA DEL SOFTWARE

PROF. R. FASOLINO - A.A. 2022 – 23

GRUPPO G8 – TASK T6

T6	Requisiti Sull'Editor di Test Case
	<p>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.</p>

STUDENTI:

FRANCESCO	DI SERIO	M63001442	f.diserio@studenti.unina.it
ZAIRA	ABDEL MAJID	M63001494	z.abdelmajid@studenti.unina.it
DAIANA	CIPOLLARO	M63001490	da.cipollaro@studenti.unina.it
LORENZO	MANCO	N000122476	lo.manco@studenti.unina.it

Indice

1. INTRODUZIONE	4
1.1 DESCRIZIONE DEL TASK	4
1.1.1 Descrizione Informale	4
1.1.2 Descrizione Formale	4
2. SCELTE DI PROGETTO	6
2.1 APPROCCIO ADOTTATO	6
2.2 SCELTA DEL PATTERN ARCHITETTURALE	6
2.2.1 Confronto con altri pattern	8
2.3 STILE CLIENT-SERVER	8
3. TECNOLOGIE	10
3.1 REACT	10
3.2 MONACO EDITOR	10
3.2.1 Monaco Editor vs Code Mirror	11
3.3 SPRING MVC	11
3.4 DOCKER	13
4. SPECIFICA DEI REQUISITI E ANALISI DEL PROBLEMA	14
4.1 USER STORIES	14
4.2 PROGRAMMAZIONE DEL LAVORO	15
4.3 INTERAZIONI CON GLI ALTRI TASK	17
4.4 REQUISITI FUNZIONALI	18
4.5 REQUISITI NON FUNZIONALI	18
4.6 FLOW DI UTILIZZO	19
4.7 GLOSSARIO DEI TERMINI	19
5. DOCUMENTAZIONE E DIAGRAMMI DI SVILUPPO	20
5.1 SYSTEM DOMAIN MODEL	20
5.2 DIAGRAMMA DEI CASI D'USO	21
5.3 SCENARI DEI CASI D'USO	22
5.3.1 Editing	22
5.3.2 Compile & Execute	22
5.3.3 Save	23
5.3.4 Download	23
5.4 ACTIVITY DIAGRAM	25
5.4.1 Compile & Execute	25
5.4.2 Save	26
5.4.3 Download	26
5.5 DIAGRAMMA DELLE CLASSI	27
5.6 DIAGRAMMI DI SEQUENZA	29
5.6.1 Editing	29
5.6.2 Download	29
5.6.3 Compile & Execute	30
5.6.4 Save	31
5.7 STIMA DEI COSTI	32
5.7.1 Unadjusted Use Case Weight	32
5.7.2 Technical Complexity Factor	33
5.7.3 Environmental Complexity Factor	33
5.7.4 Use Case Points	34
5.8 COMPONENT DIAGRAM	35
5.9 DEPLOYMENT DIAGRAM	35
6. DOCUMENTAZIONE API	37
6.1 REST API	37
6.1.1 API updateCode	38
6.1.2 API saveTest	38

Elaborato di Architettura del Software

Studenti: ZAIRA ABDEL MAJID

DAIANA CIPOLLARO

FRANCESCO DI SERIO

LORENZO MANCO

6.1.3 API <i>getCoverage</i>	39
6.1.4 API <i>getCodiceClasse</i>	39
7. TESTING	40
7.1 TESTING DEL FRONT-END (FUNZIONALITÀ DI PRESENTAZIONE).....	40
7.2 TESTING DEL BACK-END	41
7.2.1 <i>getCodiceClasse</i>	42
7.2.2 <i>updateCode</i>	42
7.2.3 <i>saveTest</i>	42
7.2.4 <i>getCoverage</i>	43
7.3 TEST DI INTEGRAZIONE DI FRONT-END E BACK-END	43
8. INSTALLAZIONE E UTILIZZO DELL'APPLICAZIONE	45
8.1 INSTALLAZIONE SENZA L'UTILIZZO DI DOCKER.....	45
8.2 INSTALLAZIONE MEDIANTE L'UTILIZZO DI DOCKER	45
8.3 UTILIZZO DELL'APPLICAZIONE.....	46

1. Introduzione

Il lavoro di progetto svolto dal nostro team si contestualizza all'interno del progetto **ENACTEST** (European iNnovative AllianCe for TESTing): iniziativa che mira a colmare le lacune nell'educazione al testing, cercando di integrare in modo coerente e tempestivo materiali didattici per il testing, tenendo conto delle esigenze dell'industria e dei modelli cognitivi degli studenti. Il nostro team ha contribuito attivamente a questo obiettivo, focalizzandosi sull'implementazione di uno specifico task nell'ambito della realizzazione di un **Educational Game** che consentirà agli studenti di migliorare le proprie competenze nel testing e confrontare i risultati dei casi di test generati manualmente con quelli generati da strumenti automatici. In particolare, l'applicazione permette allo studente di ottenere una classe da testare, scrivere casi di test *JUnit*, compilare ed eseguire i test, ottenere una misura di copertura, tramite strumenti come *Emma* o *JaCoCo*, e confrontare i risultati con i tool automatici, come *Randoop* o *Evosuite*. Questa iniziativa si propone di migliorare le prestazioni di apprendimento degli studenti e facilitare il trasferimento di conoscenze tra l'ambiente accademico e l'industria, creando un impatto significativo nell'ambito dell'educazione al testing del software.

1.1 Descrizione del Task

1.1.1 Descrizione Informale

Il task assegnatoci prevede quanto segue:

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.

1.1.2 Descrizione Formale

Il Task consiste nella creazione di un Editor di Test Case che offra:

1. Finestra di editing di testo Java: l'applicazione deve fornire un'interfaccia utente in cui il giocatore possa inserire il testo per poter testare la classe under test, utilizzando la sintassi Java.
2. Operazioni di editing: l'editor dovrà supportare le operazioni di base per la modifica del testo, come il taglio, la copia, l'incolla e l'eliminazione del testo. Queste operazioni consentiranno al giocatore di modificare e organizzare il codice del caso di test in modo efficace.
3. Funzionalità aggiuntive per semplificare la scrittura del codice:
 - a. l'evidenziazione della sintassi

Elaborato di Architettura del Software

Studenti: ZAIRA ABDEL MAJID

DAIANA CIPOLLARO

FRANCESCO DI SERIO

LORENZO MANCO

- b. l'auto-completamento
 - c. l'indentazione automatica
- 4. Modalità di presentazione del testo: l'editor dovrà fornire diverse modalità di presentazione del testo, tipiche di un ambiente di sviluppo integrato (IDE) Java.
- 5. Salvataggio dei file di testo: l'editor dovrà consentire al giocatore di salvare i file di testo creati.
- 6. Pre-caricamento di un template dei casi di test: l'editor offrirà un template pre-caricato che il giocatore può poi personalizzare secondo le specifiche esigenze.

L'obiettivo finale del Task è sviluppare un Editor di Test Case che consenta al giocatore di creare e modificare i casi di test in modo efficiente, fornendo un'interfaccia utente intuitiva e funzionalità di editing avanzate specifiche per il linguaggio di programmazione Java.

Dei requisiti riportati e ricavati dall'analisi del task assegnato sono state implementate funzionalità aggiuntive in accordo anche con gli approcci adoperati per cercare di rendere quanto più fruibile e flessibile la web application.

2. Scelte di Progetto

2.1 Approccio Adottato

Il Team di Sviluppo ha scelto di adottare una metodologia di lavoro agile basata su **SCRUM**, framework agile utilizzato proprio per la gestione dei progetti di sviluppo software.

La scelta è stata influenzata anche sulle modalità di organizzazione dello stesso progetto che richiedeva delle review periodiche con altri gruppi di lavoro per continui aggiornamenti sui progressi fatti.

Infatti, tale framework ha come unità di base gli sprint che rappresentano periodi di tempo fissati durante i quali il team di sviluppo si impegna a lavorare per consegnare un incremento di prodotto funzionante. Gli Sprint consentono al team di concentrarsi su un insieme di obiettivi e di lavorare in modo collaborativo e focalizzato per raggiungerli in maniera incrementale ricevendo continui feedback dai relativi stakeholder (nel nostro caso i docenti che hanno supervisionato i nostri lavori).

Con questo approccio è stato possibile un costante monitoraggio dei progressi e la possibilità di adattarci alle esigenze in evoluzione durante lo sviluppo del Task.

2.2 Scelta del Pattern Architettuale

Il pattern architettuale scelto è l'**MVC** (*Model-View-Controller*) per poter fare un'adeguata separazione tra la logica di business, la presentazione e l'interazione con l'utente. Per quanto riguarda la disposizione architettuale delle risorse del sistema, l'Editor sviluppato si inserisce all'interno del **Web Front-End**.

Nel caso specifico della nostra web application che implementa un editor, la suddivisione dei ruoli tra il Modello (Model), la Vista (View) e il Controllore (Controller) può essere rappresentata come segue:

1. **Model:** rappresenta la logica di business dell'applicazione e i dati associati all'editor stesso (da noi implementato tramite file java).
2. **View:** si occupa della presentazione dell'interfaccia utente con relative finestre per la classe under test e la console di output per mostrare a video i risultati del round giocato (implementata tramite file javascript, CSS e HTML).
3. **Controller:** gestisce le interazioni tra la Vista e il Modello. Si occupa di ricevere gli input dell'utente provenienti dalla Vista, interpretarli e instradare le richieste al Modello corrispondente. Il Controllore coordina le operazioni di modifica del testo, aggiornamento della Vista in base alle modifiche apportate al Modello e gestione delle azioni dell'utente come il salvataggio del documento (implementato tramite file Java).

In termini di flusso, quando un utente interagisce con l'editor, l'input viene catturato dal Controller, che esegue le operazioni richieste sul Modello. Il Modello esegue le operazioni di business e notifica il Controllore dei cambiamenti apportati, il quale aggiorna la Vista per riflettere le modifiche apportate al Modello e permettere di

visualizzare il risultato all'utente. In questo caso la vista è rappresentata dal Front-End, i cui cambiamenti genereranno delle chiamate POST e GET indirizzate al controller; questo, a sua volta, decederà se aggiornare il model o effettuare una richiesta ad altri controller esterni tramite le API da loro esposte. Ricevute le risposte o le risorse richieste, il controller aggiornerà il model e ritornerà la risposta attesa alla view (a meno di eccezioni di funzionamento).

La scelta di tale pattern è stata spinta proprio dai vantaggi offerti come:

1. **Separazione delle responsabilità:** favorisce una separazione chiara e ben definita delle responsabilità tra il modello, la vista e il controllore, rendendo l'applicazione più modulare e scalabile, facilitando la manutenzione, il **riutilizzo del codice** e il testing.
2. **Testabilità:** dato che il modello, la vista e il controllore sono separati, è possibile testarli in modo indipendente, aumentando la facilità di creazione di test automatizzati e migliorando la copertura dei test.
3. **Agilità nello sviluppo:** in accordo anche con le scelte progettuali (agile), tale pattern promuove uno sviluppo rapido e iterativo. La separazione delle responsabilità ha consentito al team di sviluppatori di lavorare in parallelo su diversi componenti dell'applicazione senza interferire tra loro.
4. **Miglior user experience:** la separazione della logica di business dalla presentazione ha consentito di creare interfacce utente più flessibili e interattive. L'aggiornamento della vista in risposta alle azioni dell'utente può essere gestito in modo efficiente grazie al controllore.

Nonostante i numerosi vantaggi offerti esistono anche alcune limitazioni o possibili difetti associati al suo utilizzo:

1. **Complessità:** la corretta implementazione richiede una buona separazione dei ruoli tra Modello, Vista e Controllore. Questo può portare a un aumento della complessità dell'applicazione nonché del codice (che potrebbe richiedere anche maggior manutenzione), specialmente in progetti di grandi dimensioni. La gestione delle interazioni tra i componenti richiede un'attenta pianificazione e un buon design.
2. **Sovraccarico di comunicazione:** poiché il flusso di dati tra il Modello, la Vista e il Controllore avviene attraverso interfacce o eventi, ci può essere un sovraccarico di comunicazione tra i componenti, portando a una riduzione delle prestazioni. Tale problema sussiste soprattutto su applicazioni grandi e complesse, nel nostro caso il problema è ovviato alla base.
3. **Rischio di dipendenze incrociate:** in alcuni casi, il Controllore può diventare troppo complesso o eccessivamente dipendente dalla Vista o dal Modello. Ciò può portare a dipendenze incrociate e rendere difficile il riutilizzo o la modifica dei componenti indipendenti. È importante gestire attentamente le dipendenze e mantenere una separazione chiara tra i componenti, che abbiamo affrontato con un'approfondita lavorazione sui diagrammi per una buona definizione dei flussi di esecuzione.

Nonostante questi possibili difetti, la corretta applicazione e una valutazione attenta delle specifiche esigenze del progetto possono contribuire a mitigare tali difetti e sfruttare appieno i benefici offerti dall'MVC.

2.2.1 Confronto con altri pattern

Data la molteplicità di pattern esistenti, l'MVC non era necessariamente l'unica scelta possibile per la realizzazione del task richiesto. Tuttavia, a valle di un'analisi comparativa dei principali pattern si è ritenuto l'MVC quello più opportuno per lo specifico ambito applicativo e per gli attributi di qualità che si sono decisi di privilegiare.

Di seguito, dunque, viene proposto un confronto con i principali pattern, quali MVC, Sense-Compute-Control e Three-Tier:

- **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 le considerazioni qualitative, è stato scelto il pattern MVC.

2.3 Stile Client-Server

Oltre al pattern MVC, si è scelto di adottare lo stile client-server. La differenza tra pattern e stile è che il primo rappresenta una soluzione specifica a un problema comune, mentre il secondo è un insieme di principi di progettazione che può essere applicato in modo più ampio per guidare l'architettura di un sistema software, definendo, ad esempio, l'organizzazione e l'interazione dei componenti all'interno di un sistema.

Lo stile Client/Server si basa sulla definizione e distinzione concettuale di due parti dell'architettura con due ruoli diversi, di cui una funge da servitore o server caratterizzata da una o più istanze, e l'altra da client, i quali chiamano il server per invocare un servizio e ottenere un risultato. I server non conoscono il numero né l'identità dei client, ma i client conoscono l'identità del server. In questo contesto, il

Elaborato di Architettura del Software

Studenti: ZAIRA ABDEL MAJID

DAIANA CIPOLLARO

FRANCESCO DI SERIO

LORENZO MANCO

nostro Front-End, scritto in Javascript, CSS e HTML, rappresenterà il client dell'architettura, attraverso cui, tramite bottoni e modifiche dei dati – come la scrittura del codice nell'editor – si effettueranno delle richieste al Back-End server, il quale risponderà opportunamente tramite dei file JSON. Il flusso del task inizia con la restituzione della pagina HTML lato server Front-End, che avviene a seguito di una fase preliminare di autenticazione e avvio della partita (a carico degli altri task) da parte dell'utente. Di seguito è schematizzato il flusso logico di interazione e l'architettura preliminare del nostro sistema.

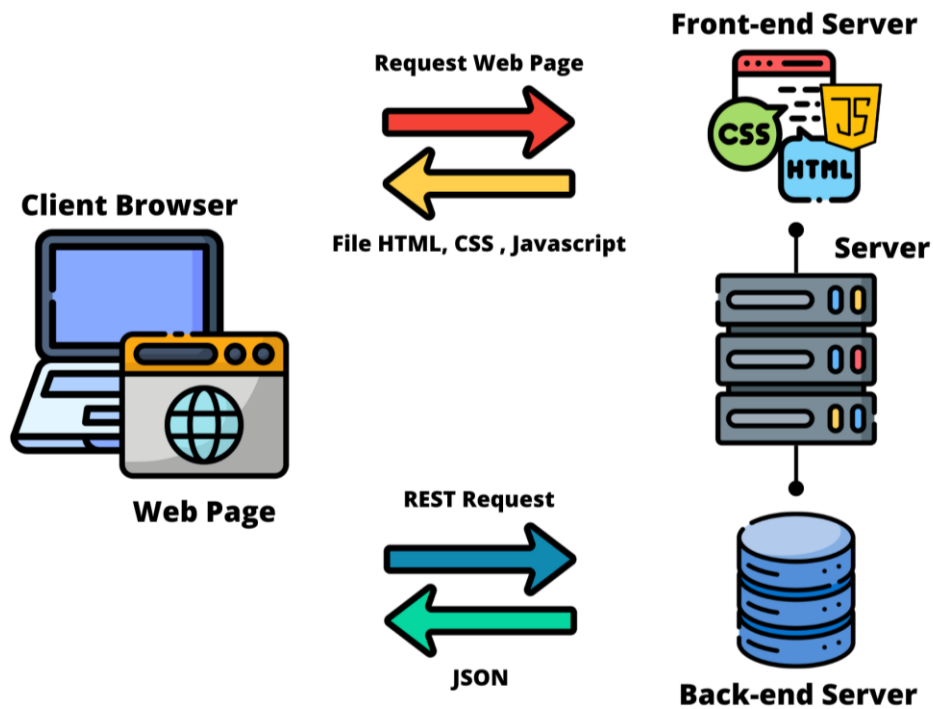


Figura 2.1 – Flusso logico dell'architettura Client-Server

3. Tecnologie

Nel corso dello sviluppo del task, abbiamo fatto affidamento ad un insieme di tecnologie consolidate e framework esistenti per ottenere risultati efficienti. Questa scelta si basa su diversi motivi fondamentali. L'efficienza e la produttività sono state fondamentali per il nostro progetto: sfruttare framework come **React**, **Monaco Editor** (basato su React) e **Spring MVC** ci ha permesso di accelerare lo sviluppo, grazie alle funzionalità predefinite, alle librerie di codice e ai modelli riutilizzabili che hanno fornito una solida base da cui partire per il nostro sviluppo.

3.1 React

React è una libreria JavaScript open-source per la creazione di interfacce utente. Si basa su un paradigma di programmazione dichiarativo e component-based, che consente di creare interfacce modulari e riutilizzabili. React utilizza un virtual DOM (Document Object Model) per ottimizzare le operazioni di rendering e offrire un'esperienza utente reattiva.

Gli elementi dell'interfaccia vengono suddivisi in componenti: unità indipendenti che incapsulano sia lo stato (state) che il comportamento (behavior) correlato a una parte specifica dell'interfaccia. I componenti possono essere composti insieme per formare interfacce più complesse.

Si utilizza un approccio unidirezionale per il flusso dei dati chiamato "props". I dati vengono passati da un componente genitore a un componente figlio tramite le props, consentendo una chiara gerarchia dei dati e un controllo preciso sul loro aggiornamento. Quando lo stato di un componente cambia, React aggiorna automaticamente solo le parti interessate dell'interfaccia, ottimizzando le prestazioni complessive.

3.2 Monaco Editor

Monaco Editor è un editor di codice open-source basato su web che fornisce funzionalità avanzate per l'editing del codice, come syntax highlighting, completamento automatico, suggerimenti, formattazione e molto altro ancora. È basato su React e può essere integrato facilmente nelle applicazioni basate sulla suddetta libreria, come nel nostro per la gestione della View e dei componenti ad esso connesso.

Utilizza una rappresentazione astratta del codice sorgente chiamata "modello" (model). Il modello è una rappresentazione strutturata del codice che consente operazioni avanzate come analisi, navigazione e manipolazione del testo. L'editor reagisce agli eventi dell'utente e mantiene sincronizzato il modello con il codice visualizzato nell'interfaccia utente.

Grazie alla sua architettura modulare, Monaco Editor offre un'ampia gamma di funzionalità personalizzabili. È possibile estendere l'editor con plug-in per adattarlo alle esigenze specifiche dell'applicazione, aggiungendo configurazioni customizzate o integrazioni con servizi esterni.

3.2.1 Monaco Editor vs Code Mirror

Monaco Editor e CodeMirror sono due popolari editor di codice open-source utilizzati per lo sviluppo di applicazioni web. Entrambi offrono funzionalità avanzate per l'editing del codice, ma presentano alcune differenze.

Monaco Editor, sviluppato da Microsoft e utilizzato in applicazioni come Visual Studio Code, è basato su linguaggi web standard come HTML, CSS e JavaScript. È altamente personalizzabile e supporta diversi linguaggi di programmazione. Grazie alla sua integrazione con React, è facile da integrare in applicazioni basate su React. Inoltre, ha una community attiva di sviluppatori, è stabile e affidabile.

CodeMirror, d'altra parte, è un editor di codice sviluppato interamente in JavaScript puro. È progettato per essere leggero e altamente personalizzabile, supportando molti linguaggi di programmazione. Può essere facilmente integrato in diverse applicazioni web e framework; inoltre, è altamente estendibile attraverso plug-in personalizzati e ha una comunità di sviluppatori attiva.

Nel nostro caso, abbiamo scelto Monaco Editor per la sua affidabilità, la vasta community attiva e la stabilità: essendo sviluppato da Microsoft e ampiamente utilizzato in applicazioni IDE, offre un'eccellente esperienza utente e continua a essere supportato e aggiornato regolarmente. La sua integrazione con React e la sua ampia gamma di funzionalità lo hanno reso la scelta ideale per le nostre esigenze di sviluppo.

3.3 Spring MVC

Spring MVC è un framework Java per lo sviluppo di applicazioni web. Si basa sul design pattern architetturale Model-View-Controller, ragion per cui ci permette di sfruttarne a pieno i suoi vantaggi.

Il flusso di lavoro inizia con la ricezione di una richiesta HTTP: il controller, che funge da punto di ingresso, elabora la richiesta e ne determina il percorso appropriato all'interno dell'applicazione. Una volta che il controller ha elaborato la richiesta, restituisce una vista, responsabile della presentazione dei dati al cliente, solitamente sotto forma di HTML generato dinamicamente o di un'altra rappresentazione visuale. Il framework offre un'ampia gamma di opzioni per la gestione delle viste, consentendo una flessibilità nel rendere l'interfaccia utente.

Spring MVC facilita anche il trattamento delle richieste e delle risposte, gestendo aspetti come la validazione dei dati e la gestione delle eccezioni. In particolare, il punto di ingresso centrale che gestisce le richieste HTTP e che coordina il flusso di elaborazione delle richieste nell'applicazione è il DispatcherServlet. Si riportano alcune delle fasi principali di response e request gestite da quest'ultimo:

1. *Ricezione della richiesta* (Request reception): alla ricezione di una richiesta HTTP, agisce come un punto di ingresso centrale dell'applicazione web. Riceve la richiesta dal client e avvia il processo di gestione.
2. *Ricerca del controller* (Controller lookup): il DispatcherServlet analizza la richiesta e determina quale controller deve essere chiamato per gestirla. Questa informazione è basata su configurazioni definite nell'applicazione, come le annotazioni @RequestMapping o le configurazioni XML.
3. *Chiamata al controller* (Controller invocation): una volta identificato il controller appropriato, il DispatcherServlet invoca il metodo che corrisponde alla richiesta. Il controller elabora la richiesta, accede ai dati necessari dal modello e prende decisioni sul flusso di esecuzione.
4. *Elaborazione della logica di business* (Business logic processing): durante questa fase, il controller esegue la logica di business necessaria per soddisfare la richiesta. Questo può includere l'interazione con servizi, database o altre componenti del sistema per ottenere i dati richiesti o effettuare operazioni specifiche.
5. *Generazione della vista* (View generation): una volta completata l'elaborazione della logica di business, il controller restituisce un oggetto ModelAndView o un oggetto di modello dati (come un oggetto Map o un oggetto DTO) al DispatcherServlet. Questi dati vengono utilizzati per generare la vista corrispondente alla richiesta.
6. *Risposta al client* (Response to the client): il DispatcherServlet prende la vista generata e utilizza un sistema di risoluzione delle viste (ViewResolver) per determinare quale template o pagina HTML deve essere utilizzato per creare la risposta. Il risultato viene quindi inviato al client come risposta HTTP.

7. *Renderizzazione della risposta* (Response rendering): Il DispatcherServlet renderizza la vista utilizzando un motore di template o un framework di rendering specificato, che può includere la compilazione dei dati nel modello della vista e la generazione di HTML o altri formati di output. La risposta viene quindi inviata al client tramite il protocollo HTTP.

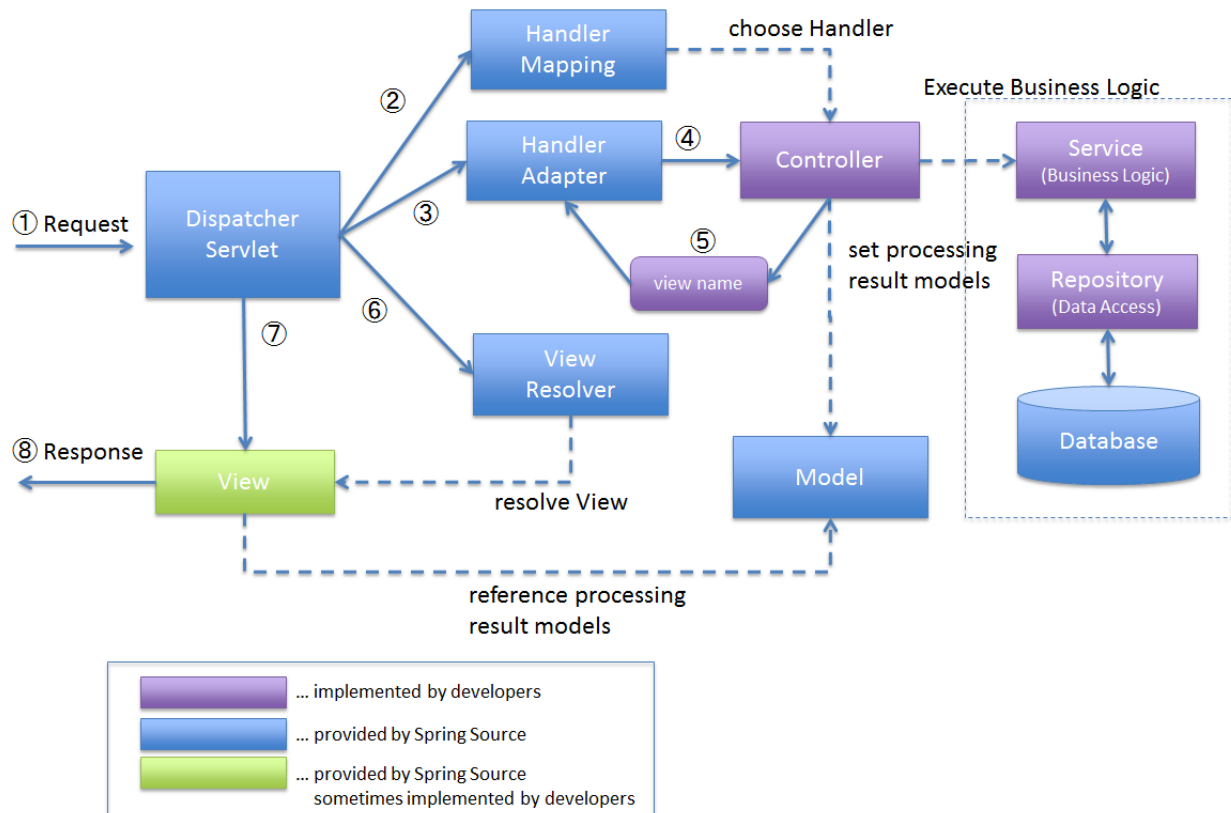


Figura 3.1 – Diagramma del flusso di elaborazione di Spring MVC, dalla ricezione della richiesta alla risposta.

3.4 Docker

Docker è una piattaforma di gestione che fornisce dei comandi semplici e standardizzati con cui creare, eliminare e manipolare container e automatizzarne la distribuzione, favorendone un deployment ottimizzato, fornendo un vero e proprio "Sistema Operativo per Container" o macchina virtuale leggera. Il Sistema ruota intorno al concetto di immagine, un file immutabile e quindi *read-only* che fornisce uno snapshot di un container, il quale sarà concretamente creato a tempo di esecuzione dell'immagine; potremmo associare le immagini a delle classi e i container alle sue istanze, per cui ogni immagine può dar vita a più container.

La gestione e definizione delle immagini è realizzata tramite un Docker file, un documento testuale che contiene tutti i comandi invocabili dall'utente per assemblare un'immagine, mentre la loro persistenza è realizzata tramite il Docker Registry. La semplicità dell'infrastruttura, la tendenza all'utilizzo di architetture a microservizi, la possibilità di definire reti isolate per i container e il suo essere Open Source, hanno fatto di Docker uno strumento estremamente popolare e utilizzato.

4. Specifica dei Requisiti e Analisi del Problema

4.1 User Stories

In Scrum, una user story è un formato comune per descrivere i requisiti di un prodotto o di una funzionalità dal punto di vista dell'utente. Rappresenta una breve, ma completa, descrizione di un'unità di lavoro che deve essere svolta dal team di sviluppo durante un'iterazione o sprint. In particolare, la struttura di una user story da noi adottata in fase di progettazione si basa sulla seguente sintassi:

- **As a** [tipo di utente]: identifica il ruolo o il tipo di utente che richiede o beneficia della funzionalità.
- **I want to** [azione/desiderio]: descrive l'azione che il tipo di utente desidera compiere o il desiderio che ha. Spesso è espressa come un verbo seguito dall'oggetto dell'azione desiderata.
- **So that** [obiettivo/risultato desiderato]: specifica il motivo o l'obiettivo che si intende raggiungere con l'azione desiderata. Indica il valore o il beneficio che l'utente si aspetta di ottenere dalla funzionalità o dalla storia.

La struttura "As a, I want to, So that" aiuta a fornire una visione chiara del contesto, delle intenzioni e degli obiettivi dei player, consentendoci di comprendere meglio le esigenze e le aspettative degli utenti durante la creazione del prodotto. Di seguito vengono proposte le storie utente per le principali funzionalità dell'Editor.



Figura 4.1 – User stories estratte, classificate per priorità

4.2 Programmazione del lavoro

Essendo lo sviluppo basato su un approccio incrementale, il lavoro è stato suddiviso in tre sprint, ciascuno della durata di circa tre settimane. Questa pianificazione è stata stabilita per garantire una gestione efficace del tempo e delle risorse, consentendo al team di concentrarsi su obiettivi specifici durante ciascuna iterazione. Durante ogni sprint, abbiamo utilizzato il **Product Backlog**: elenco prioritizzato di tutte le user stories, i requisiti e gli elementi di lavoro necessari per il progetto.

ID	AS A ...	I WANT TO ...	SO THAT ...	PRIORITY	SPRINT	STATUS
001	Giocatore	Inserire codice in una finestra di editing	Poter scrivere la classe di test	High	2	Complete
002	Giocatore	Apportare modifiche	Poter modificare il codice nella finestra di editing	High	1	Complete
003	Giocatore	Usufruire di funzionalità di presentazione	Facilitare l'attività di editing	Medium	3	Complete
004	Giocatore	Salvare file di testo creati	Poter mantenere la persistenza in remoto dei test scritti	Medium	1	Complete
005	Giocatore	Usufruire di un template precaricato	Essere facilitato nell'attività di coding	Low	1	Complete
006	Giocatore	Compilare e lanciare il test	Avere il risultato di esecuzione	High	1	Complete
007	Giocatore	Visualizzare la classe da testare	Per avere un riferimento nella scrittura del test e visualizzare la coverage	Medium	2	Complete
008	Giocatore	Selezionare un tema di visualizzazione	Scegliere la grafica dell'editor	Low	1	Complete
009	Giocatore	Scaricare file di testo creati	Poter mantenere la persistenza in locale dei test scritti	Medium	1	Complete
					0	
					0	
					0	
					0	
TOTAL					13	

Figura 4.2 – Product Backlog Report dell'ultimo sprint

È stato estratto uno **Sprint Backlog** per ciascuno sprint dal Product Backlog, selezionando le user stories rilevanti e estraendo i task da implementare per la specifica iterazione.

ID	USER STORY	TASKS	STATUS	EFFORT
002	Apportare modifiche al codice nella finestra di editing	Creare una finestra di editing	Complete	2
		Rendere la finestra editabile	Complete	
005	Usufruire di un template precaricato	Scrivere il template	Complete	1
		Caricarlo sulla finestra di editing	Complete	
006	Presentare una pagina di interfaccia	Creare una finestra di editing	Complete	5
		Creare i bottoni	Complete	
		Creare la finestra contenente la classe da testare	Complete	
		Creare la finestra di output	Complete	
008	Selezionare un tema di visualizzazione	Selezionare una lista di temi	Complete	1
		Implementare un menù di selezione	Complete	

Figura 4.3 – Sprint Backlog Report relativo al secondo sprint

Lo Sprint Backlog riportato è quello relativo alla seconda iterazione. È stato il punto di riferimento per organizzare, pianificare ed eseguire il lavoro durante lo sprint. In particolare, si presentano dissonanze tra i due Backlog in quanto il Product riportato fa riferimento all'ultima iterazione, dunque presenta user stories affinate. L'utilizzo di questi artefatti ci ha aiutato a gestire il lavoro in modo efficace, mantenendo il focus sulle attività prioritarie e guidando il team verso il raggiungimento degli obiettivi del progetto.

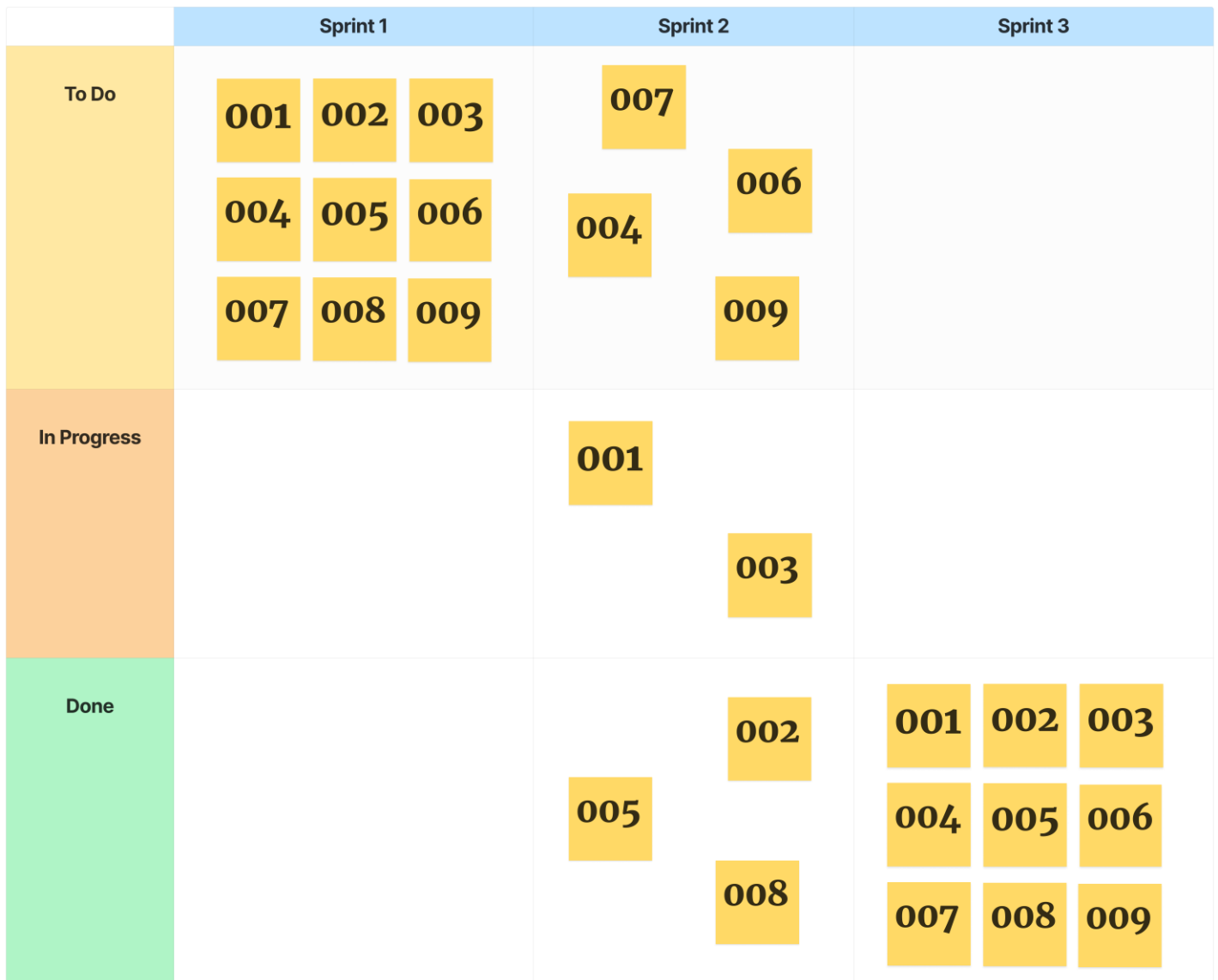


Figura 4.4 – Flusso di lavoro per lo sviluppo delle user stories, stile Kanban Board

4.3 Interazioni con gli altri Task

Per quanto riguarda le interazioni con gli altri task, occupandoci dello sviluppo dell'editor ci siamo interfacciati con:

- *Task 5*, responsabile della gestione dell'avvio della partita. Questo task ci ha fornito tutte le informazioni necessarie per lanciare un Round nell'editor.
- *Task 4*, incaricato della realizzazione di una repository per garantire la persistenza dei dati delle partite giocate.
- *Task 7*, responsabile della compilazione ed esecuzione delle classi di test. Abbiamo lavorato in stretta collaborazione con questo task per ottenere i risultati relativi alla copertura della class under test.

Di seguito si riporta il diagramma di interazione tra i task in cui si evidenzia la dipendenza dagli altri task evidenziata dal filo rosso, secondo la convenzione Agile.

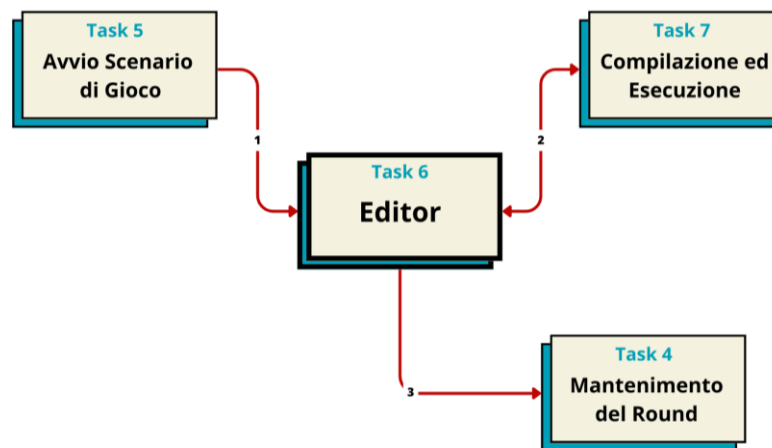


Figura 4.5 – Diagramma delle dipendenze con gli altri task

Questa interazione sinergica tra i diversi task ci ha consentito di integrare correttamente le funzionalità dell'editor con le altre componenti del sistema, così come riportato dal seguente communication diagram.

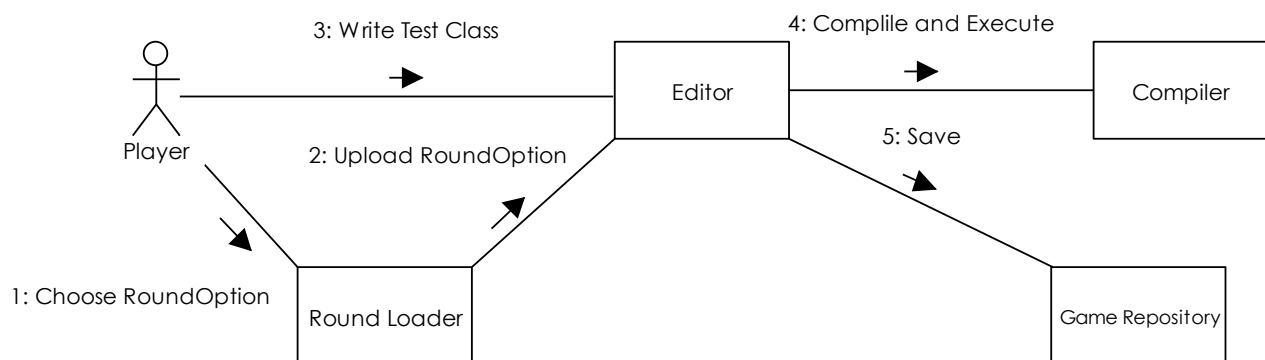


Figura 4.6 – Communication Diagram, UML

4.4 Requisiti funzionali

I requisiti funzionali sono specifici delle funzionalità e dei comportamenti che il sistema software deve fornire per soddisfare le esigenze degli utenti. Nella progettazione, abbiamo identificato questi requisiti per guidare lo sviluppo dell'Editor e garantire che il sistema offra le funzionalità richieste:

- Visualizzazione della Class Under Test in una finestra dedicata.
- Visualizzazione di output generati dalla compilazione.
- Creazione di una finestra di editing con un template predefinito.
- Salvataggio in remoto del codice prodotto in un documento di testo con un nome specificato dall'utente.
- Download del codice prodotto in un documento di testo con un nome specificato dall'utente.
- Operazioni di modifica del testo, come inserimento e cancellazione.
- Ricerca e sostituzione di testo all'interno del documento.
- Fornitura di suggerimenti e funzionalità di completamento automatico del codice.
- Syntax highlighting.
- Cambio del tema dell'editor.
- Possibilità di avviare i servizi di compilazione ed esecuzione del codice scritto.
- Highlighting delle linee di codice con colori diversi (rosso per copertura assente, giallo per copertura parziale e verde per copertura totale) in base all'esito del Coverage Test eseguito con JaCoCo.

4.5 Requisiti non funzionali

I requisiti non funzionali definiscono le qualità e le restrizioni del sistema software al di là delle funzionalità specifiche. Abbiamo identificato questi requisiti per garantire che il sistema abbia prestazioni adeguate, sicurezza e usabilità, oltre ad altre caratteristiche importanti per il successo del progetto:

- *Affidabilità*: garantire un'adeguata gestione dei file per consentire un monitoraggio completo e affidabile delle Partite giocate dal player, assicurando che i dati siano memorizzati correttamente e che non si verifichino perdite o corruzioni.
- *Compatibilità e Portabilità*: garantire che l'applicazione sia compatibile con i principali web browser (Chrome, Edge, Opera, Safari e Mozilla Firefox), consentendo agli utenti di accedere all'applicazione tramite il browser di loro scelta. La compatibilità con i sistemi operativi (Windows, macOS e Linux) è implicita, assicurando che l'applicazione funzioni correttamente su diverse piattaforme senza problemi.
- *Robustezza*: l'applicazione deve essere in grado di gestire in modo affidabile eventuali problemi o anomalie che si verificano con le interfacce, mantenendo la sua funzionalità e fornendo una risposta adeguata agli utenti.

- *Usabilità*: realizzare un'interfaccia utente intuitiva, semplice e piacevole da utilizzare, garantendo un'esperienza fluida e soddisfacente per gli utenti.

Questi requisiti di qualità sono fondamentali per garantire un'esperienza utente positiva, una funzionalità stabile e un'applicazione affidabile nel contesto dell'architettura specifica.

4.6 Flow di Utilizzo

1. A seguito dell'accesso all'applicazione web, dopo essersi loggato e aver selezionato la classe da testare, l'utente può iniziare a scrivere il codice di test.
2. L'utente può selezionare un tema di visualizzazione della finestra di editing.
3. La scrittura del codice viene facilitata mediante la colorazione delle keywords.
4. Sono fornite anche attività di supporto alla scrittura e revisione del codice come il find e il replace.
5. Una volta terminata la scrittura del codice, al giocatore è data la possibilità di salvare il codice prodotto sia in locale che in una repository di game.
6. L'utente potrà visualizzare la copertura del codice nonché i risultati prodotti da questa rispetto alla classe under test.

4.7 Glossario dei termini

Termine	Descrizione	Sinonimi
Test	Insieme di operazioni volte a determinare un prestabilito risultato	Prova
Utente	Persona fisica che interagisce con il sistema	Giocatore, Tester, Player
Sistema	Software che permette di gestire ed interfacciarsi con i servizi offerti dal prodotto	Applicazione
Sviluppatore	Persona fisica deputata alla realizzazione del codice sorgente	Developer, Creatore
Editor	Finestra che consente di scrivere e modificare un codice	Finestra di editing, Web Page
Partita	Serie di azioni definite da regole precise, che possono condurre i giocatori alla vittoria, al pareggio o alla sconfitta in base al punteggio ottenuto	Play, Match
Turno	Ciascuno dei periodi di tempo in cui viene suddivisa un'attività in base a un avvicendamento prestabilito	Round
Classe sotto test	Classe Java per la quale l'utente ha scelto di scrivere il test mediante la finestra di editing	Class Under Test, classe da testare

5. Documentazione e Diagrammi di Sviluppo

5.1 System Domain Model

Di seguito riportiamo una visione ad alto livello del sistema e delle sue entità principali con le relative relazioni ed attributi. Sono state individuate quattro entità fondamentali: *Player*, *Game*, *Round*, *Output_Round*. Il *Player* realizza una relazione di associazione di tipo *uno a molti* con l'entità *Game*, in quanto il giocatore potrebbe decidere di avviare più partite contemporaneamente. *Game* è una superclasse rispetto all'entità *Round* che invece rappresenta una sottoclasse, in quanto parte di una partita. Tra le due vige una relazione *uno a molti* perché una partita può essere costituita da più turni, ma ogni turno è associato ad un'unica partita. A sua volta, l'entità *Round* ha una relazione di dipendenza con l'entità *Output_Round* di tipo *uno a uno*; ciò è dovuto al fatto che al termine di ogni round si avrà un solo esito che, a sua volta, è associato a un turno specifico e da cui dipende la stessa entità *Round*.

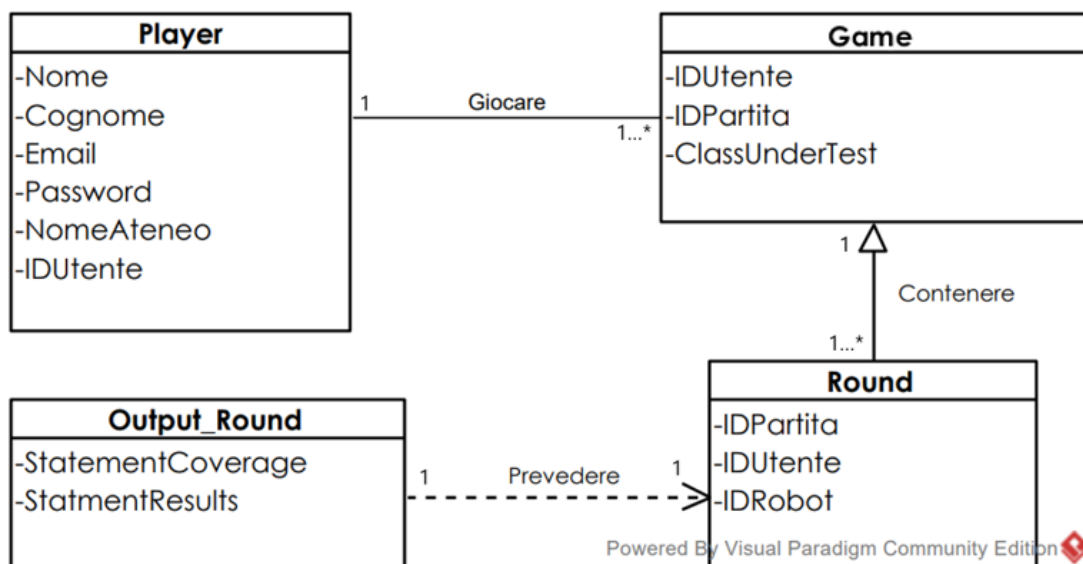


Figura 5.1 – System Domain Model, UML

5.2 Diagramma dei Casi D'Uso

Per visualizzare in modo chiaro le funzionalità del sistema dal punto di vista degli attori coinvolti e fornire una panoramica dei servizi offerti dalla nostra web application, riportiamo il diagramma dei casi d'uso, di cui notiamo che l'attore principale è proprio il giocatore che fa uso dell'applicazione.

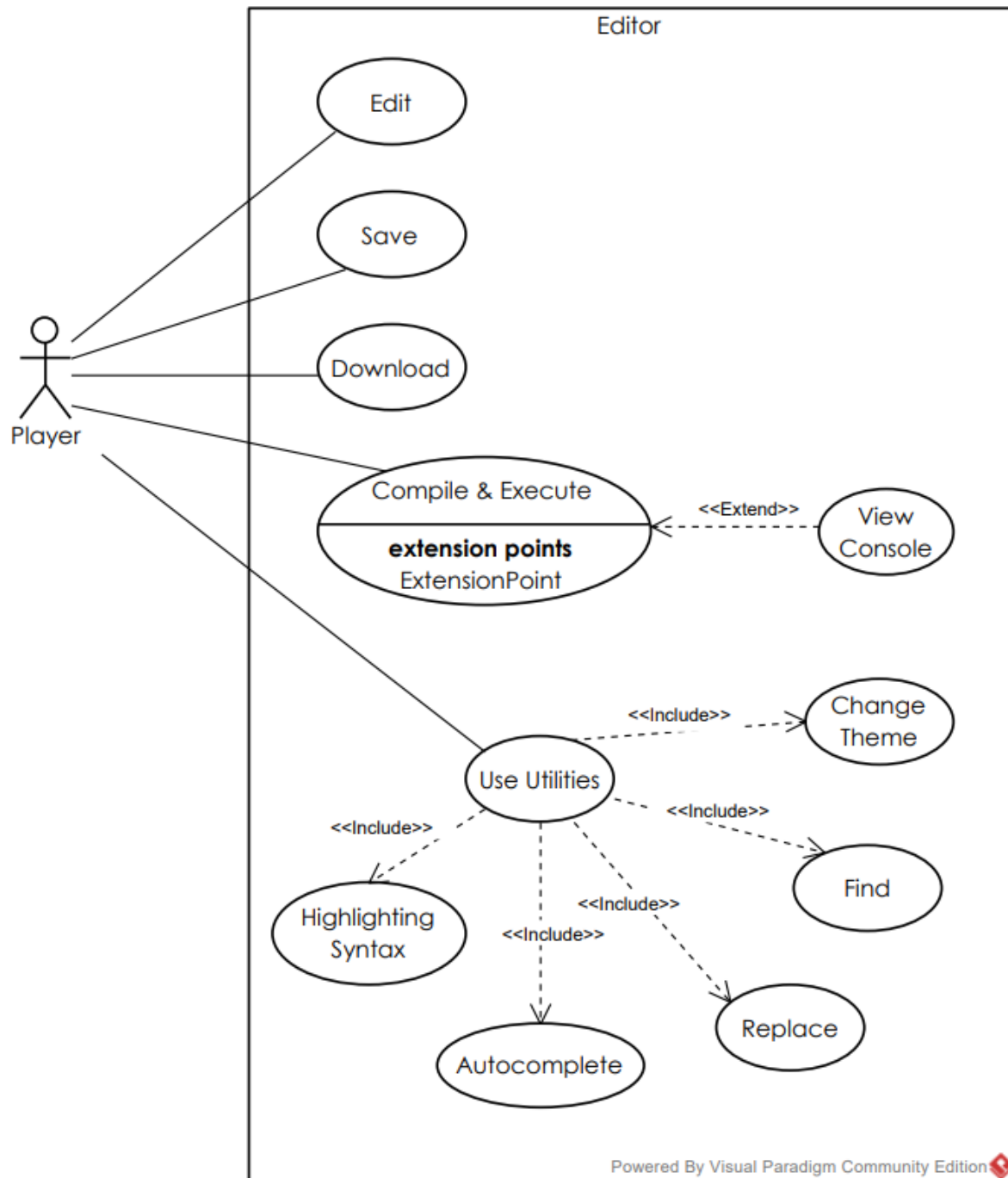


Figura 5.2 – Diagramma dei casi d'uso, UML

5.3 Scenari dei Casi D'Uso

Riportiamo la descrizione degli scenari dei principali casi d'uso evidenziati nell'ambito del nostro Task di sviluppo.

5.3.1 Editing

Si riporta lo scenario del caso d'uso *EDITING*:

Caso d'uso:	EDITING
Attore	Player
Descrizione	Un giocatore ha avviato uno scenario di gioco e inizia la scrittura di codice.
Pre-Condizioni	Il player si è autenticato e ha avviato la partita.
Sequenza di eventi	<ol style="list-style-type: none">1. Il caso d'uso inizia quando il giocatore avvia la partita.2. Il player inizia a digitare codice java per testare la classe da testare.
Post-Condizioni	Visualizzazione delle modifiche effettuate.
Casi d'uso correlati	USE UTILITIES
Sequenza di eventi alternativi	-

5.3.2 Compile & Execute

Si riporta lo scenario del caso d'uso *COMPILE & EXECUTE*:

Caso d'uso:	COMPILE & EXECUTE
Attore	Player
Descrizione	Il player vuole compilare ed eseguire il codice scritto per vedere se è riuscito a coprire interamente la classe under test.
Pre-Condizioni	È stata editata la classe di test
Sequenza di eventi	<ol style="list-style-type: none">1. Il caso d'uso inizia quando il giocatore clicca il bottone "Compile & Execute".2. Il sistema farà una chiamata POST al server del compiler che eseguirà, con apposite tecnologie (JaCoco), il codice editato restituendo la Coverage e gli Output.3. All'arrivo della risposta da parte del compiler il sistema:<ol style="list-style-type: none">a. Prenderà la coverage e ne farà un parsing per evidenziare opportunamente le righe di codice coperte e non.b. Salverà i risultati ottenuti dall'esecuzione.

Post-Condizioni	1. Visualizzazione della coverage nella finestra della classe under test. 2. Visualizzazione in console 'Output' dei risultati.
Casi d'uso correlati	-
Sequenza di eventi alternativi (1)	Si verifica un errore nella richiesta al server
Post-Condizioni Alternative (1)	Il codice editato non verrà compilato
Sequenza di eventi alternativi (2)	C'è un errore nell'esecuzione del test
Post-Condizioni Alternative (2)	La classe under test non verrà sovrascritta con i risultati di coverage

5.3.3 Save

Si riporta lo scenario del caso d'uso *SAVE*:

Caso d'uso:	SAVE
Attore	Player
Descrizione	Un giocatore vuole salvare il codice che ha editato.
Pre-Condizioni	Scrittura della classe di test.
Sequenza di eventi	1. Il caso d'uso inizia quando il giocatore clicca il bottone "Save". 2. La Web Application richiede tramite pop-up un nome e l'estensione del file da salvare. 3. L'utente digiterà nome e relativa estensione (.java). 4. Il sistema manderà il file al game Repository.
Post-Condizioni	Il file viene salvato nella game Repository.
Casi d'uso correlati	-
Sequenza di eventi alternativi	Si verifica un errore nella richiesta al server
Post-Condizioni Alternative	Il file e le relative modifiche effettuare non vengono salvate.

5.3.4 Download

Si riporta lo scenario del caso d'uso *DOWNLOAD*:

Elaborato di Architettura del Software

Studenti: ZAIRA ABDEL MAJID

DAIANA CIPOLLARO

FRANCESCO DI SERIO

LORENZO MANCO

Caso d'uso: DOWNLOAD	
Attore	Player
Descrizione	Un giocatore vuole salvare sul proprio pc il codice che ha editato.
Pre-Condizioni	Scrittura della classe di test.
Sequenza di eventi	<ol style="list-style-type: none"> 1. Il caso d'uso inizia quando il giocatore clicca il bottone "Download". 2. La Web Application richiede tramite pop-up un nome e l'estensione del file da salvare. 3. L'utente digiterà nome e relativa estensione (.java). 4. Il sistema avvierà il download del file.
Post-Condizioni	Il file viene salvato in locale.
Casi d'uso correlati	-
Sequenza di eventi alternativi	Si verifica un errore nella richiesta al server
Post-Condizioni Alternative	Il file non verrà scaricato.

Per quanto riguarda il caso d'uso *USE UTILITIES*, definito come insieme di più servizi di cui l'utente può usufruire sia tramite richiesta diretta che non, non è stato espresso in maniera esplicita per evitare ridondanze e ampollosità nella stesura della documentazione, anche in accordo alla politica adottata (agile).

5.4 Activity Diagram

Per poter descrivere il flusso delle attività di ogni caso d'uso sono stati implementati i rispettivi Activity Diagram.

5.4.1 Compile & Execute

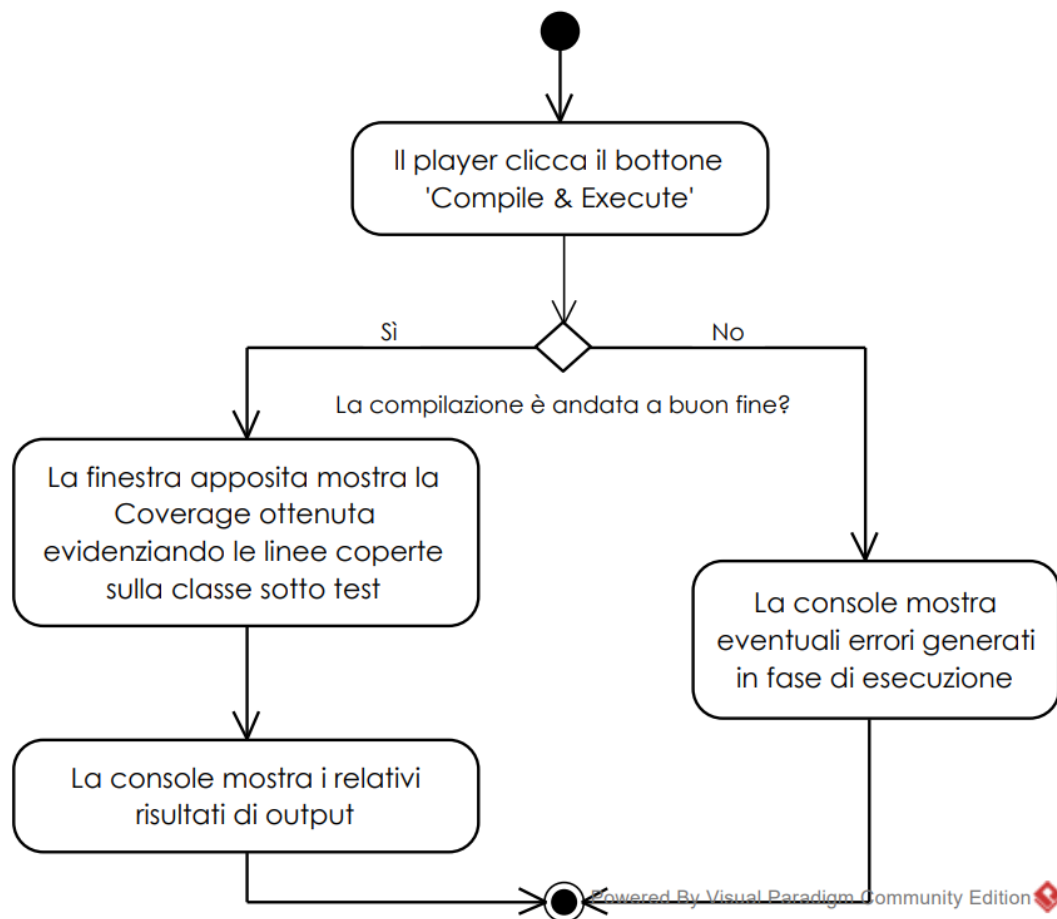


Figura 5.3 – Activity Diagram “Compile & Execute”, UML

5.4.2 Save

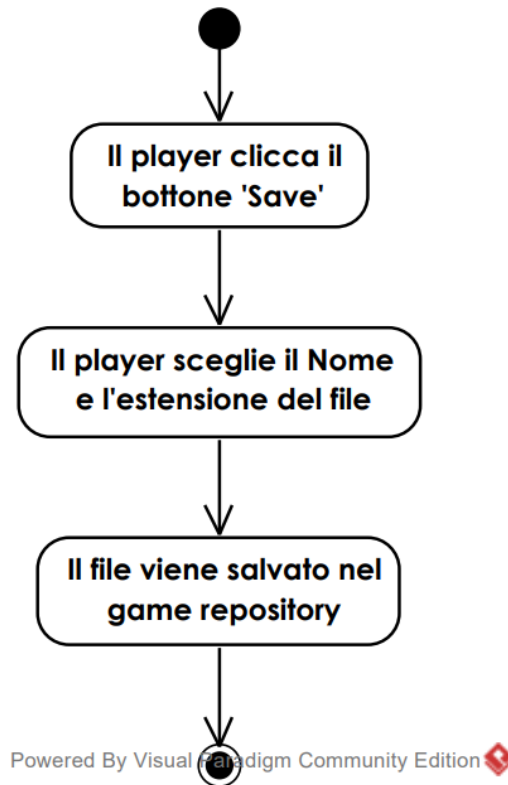


Figura 5.4 – Activity Diagram “Save”, UML

5.4.3 Download

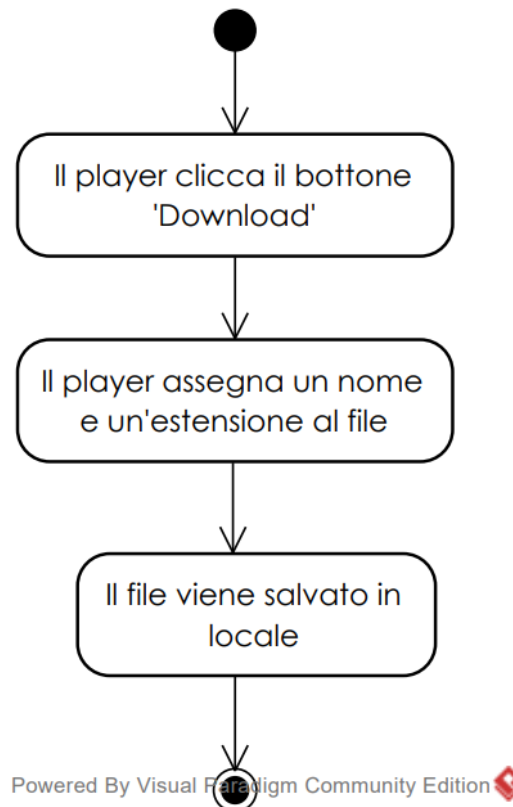


Figura 5.5 – Activity Diagram “Downlaod”, UML

5.5 Diagramma delle Classi

Dopo aver individuato i casi d'uso del sistema, e averne fatto un'analisi singolare e iterativa, si passa ad un class diagram dettagliato per individuare le rispettive entità che compongono il nostro sistema, contestualizzandolo anche al pattern scelto.

Abbiamo diviso l'architettura nei tre componenti principali con le relative classi che contengono:

- *Model* che si può vedere come un container dedito alla persistenza delle informazioni riguardanti l'attuale sessione di game, infatti contiene:
 - **Partita**: classe che registra le informazioni riguardanti la partita che si sta giocando.
 - **Coverage**: classe che registra le informazioni riguardanti la copertura ottenuta dall'esecuzione del codice di test editato.
 - **Test**: classe che contiene le informazioni riguardanti il test da inviare per la compilazione e l'esecuzione.
 - **Messaggio**: classe che mantiene la persistenza dei messaggi inviati dalla View al Controller.
- *Controller* che fa da tramite tra View e Model per il passaggio delle informazioni.
- *View* che modella l'interfaccia della pagina web, il cui componente principale è il **Landing** che realizza delle relazioni di composizione con:
 - **OutputWindow**: permette di modellare la finestra adibita alla visualizzazione di eventuali errori o risultati derivanti dall'esecuzione del codice editato.
 - **ThemeDropdown**: permette l'implementazione della logica del pulsante *ChangeTheme*: infatti, tale classe contiene in sé tutti i temi previsti da **monacoThemes**; ragion per cui con quest'ultima realizza anche una relazione di aggregazione.
 - **ClassWindow**: permette di modellare la finestra adibita alla visualizzazione della classe da testare, nonché dei risultati della coverage una volta eseguita la compilazione. Tale finestra ha una relazione di aggregazione con la classe **Editor** che permette di modellare il contenuto della classe, in sola lettura.
 - **CodeEditorWindow**: permette di modellare la finestra adibita alla scrittura del codice per testare la classe sotto test. Tale finestra ha una relazione di aggregazione con la classe **Editor** che permette di modellare il contenuto della classe.

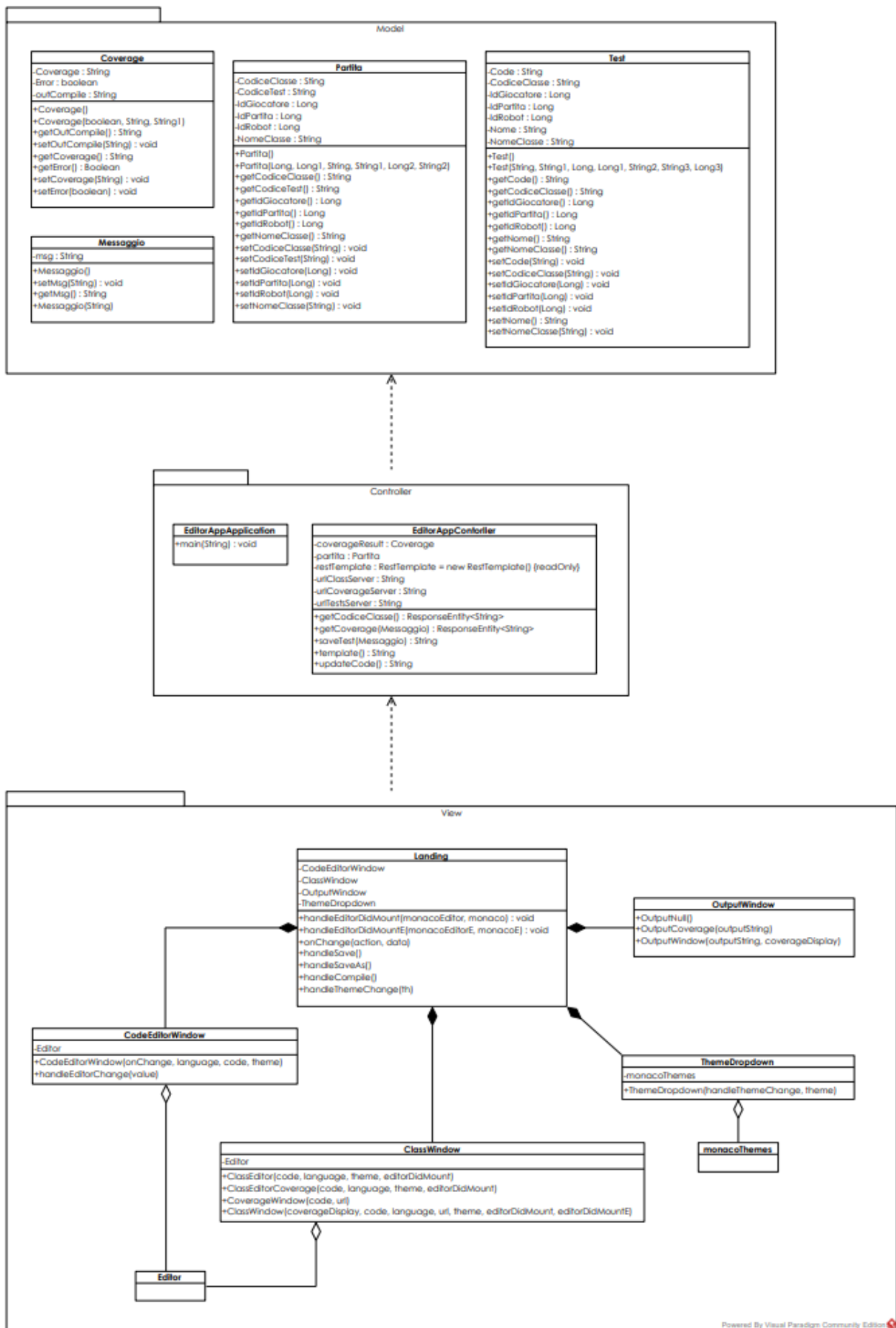


Figura 5.6 – Diagramma delle Classi, UML

5.6 Diagrammi di Sequenza

Per analizzare in maniera dettagliata il flusso di esecuzione dei casi d'uso si procede con i Sequence Diagram.

5.6.1 Editing

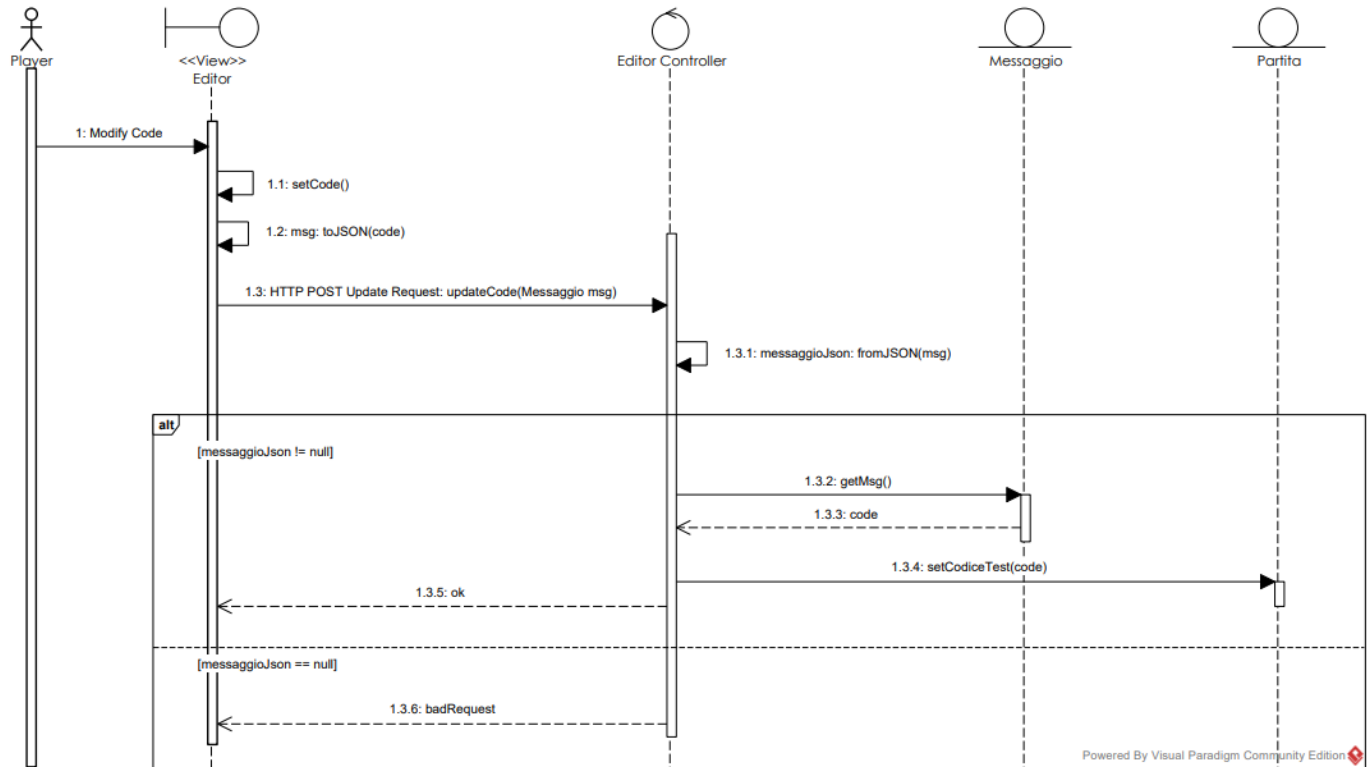


Figura 5.7 – Sequence Diagram “Editing”, UML

5.6.2 Download

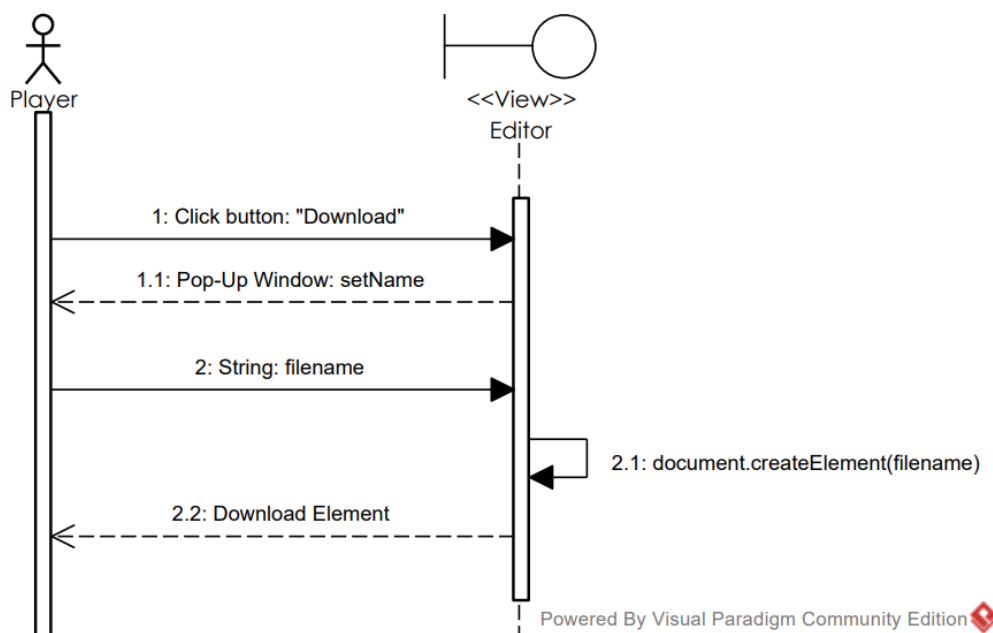


Figura 5.8 – Sequence Diagram “Download”, UML

5.6.3 Compile & Execute

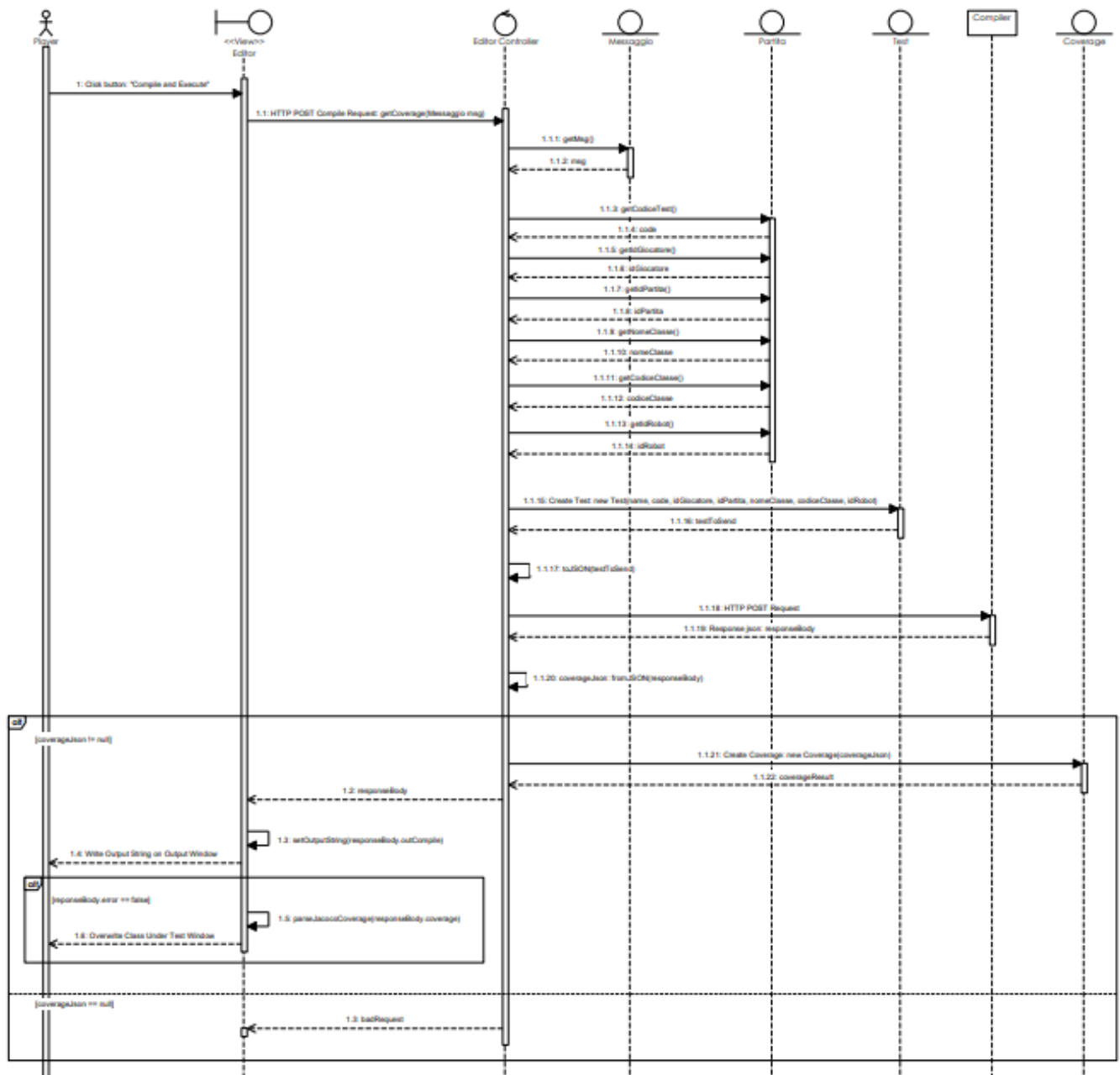


Figura 5.9 – Sequence Diagram “Compile & Execute”, UML

5.6.4 Save

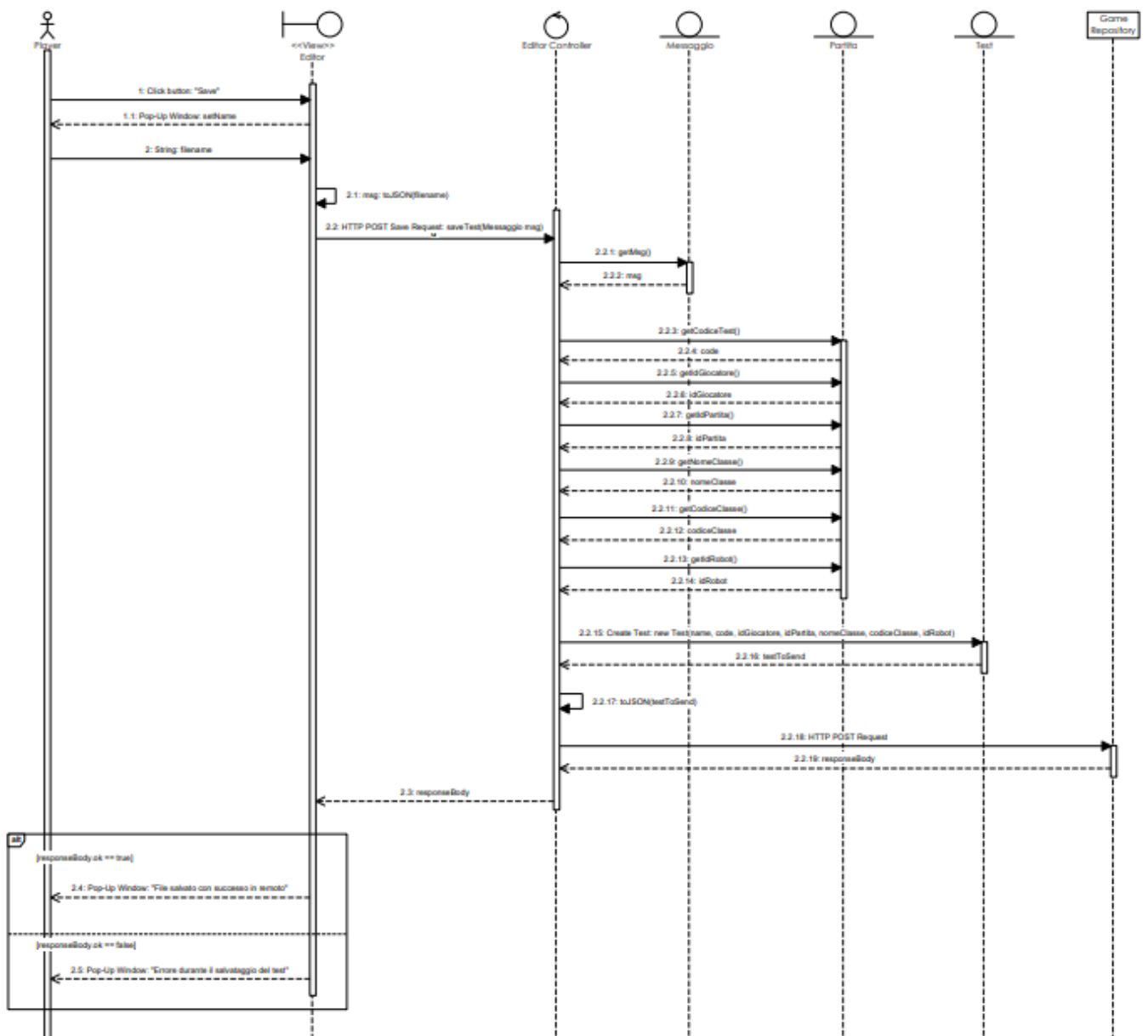


Figura 5.10 – Sequence Diagram “Save”, UML

5.7 Stima Dei Costi

Nonostante l'uso di un approccio agile, come Scrum, che si concentra principalmente sulla consegna iterativa di funzionalità di valore e sulla risposta ai cambiamenti in modo flessibile, facendo richiesta di una documentazione snella e minimal, è importante avere una buona comprensione dei costi associati allo sviluppo della web application, che può aiutare a gestire il budget, prendere decisioni informate e monitorare l'andamento del progetto in base alle risorse e tempistiche che si hanno a disposizione. 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 sulla base degli use case, ovvero le interazioni tra gli utenti e il sistema che si presta bene al nostro task, in quanto componente di Front-End.

5.7.1 Unadjusted Use Case Weight

Gli *Unadjusted Use Case Weight* (UUCW) sono un fattore chiave nel calcolo dei punti dei casi d'uso (Use Case Points) e sono utilizzati per valutare il dimensionamento e la complessità di un sistema software. Gli UUCW rappresentano la misura dei casi d'uso all'interno di un sistema. Essi riflettono il peso relativo di ciascun caso d'uso rispetto agli altri e possono variare in base alla loro complessità, frequenza di utilizzo, coinvolgimento dell'utente e altri fattori. L'assegnazione degli UUCW viene effettuata in base a una scala predefinita che attribuisce un valore numerico a ciascun caso d'uso rispetto alla sua classificazione, consentendo così il calcolo dei punti dei casi d'uso totali.

Casi D'Uso	Peso	Classificazione	#Transazioni
Editing	10	Medium	4
Compile	15	High	8
Save	10	Medium	5
Download	5	Low	2
Use Utilities			
○ Change Theme	5	Low	2
○ Find	5	Low	3
○ Replace	5	Low	4
○ Autocomplete	5	Low	1
○ Syntax Highlighting	10	Medium	4

Sommando i valori stimati, in base alla classificazione fatta per ogni caso d'uso, otteniamo:

$$\begin{aligned}UUCW &= \#casi\ Low * 5 + \#casi\ Medium * 10 + \#casi\ High * 15 = \\&= 5 * 5 + 3 * 10 + 1 * 15 = 25 + 30 + 15 = \mathbf{70}\end{aligned}$$

5.7.2 Technical Complexity Factor

I *Technical Complexity Factors* (TCF) sono utilizzati per valutare la complessità tecnica di un sistema software, considerando aspetti come l'architettura, la sicurezza e la scalabilità. Essi contribuiscono al calcolo dei punti dei casi d'uso e aiutano a stimare lo sforzo di sviluppo e il costo del progetto.

Fattori Tecnici	Peso	Valore	Peso x Valore
Sistema Distribuito	2	4	8
Prestazioni	1	3	3
Efficienza	1	4	4
Complessità	1	2	2
Riusabilità	1	2	2
Easy to Install	0.5	3	1,5
Easy to Use	0.5	5	2,5
Portabilità	2	5	10
Manutenibilità	1	2	2
Elaborazione Parallela	1	1	1
Sicurezza	1	2	2
Accessi da Terzi	1	2	2
End User Training	1	4	4

Sommando i prodotti tra i valori stimati e i pesi predefiniti otteniamo:

$$Tfactor = 44$$

Da cui il calcolo dei TCF, facendo capo alla formula empirica:

$$TCF = 0.6 + \left(\frac{Tfactor}{100}\right) = 0.6 + \left(\frac{44}{100}\right) = 1.04$$

5.7.3 Environmental Complexity Factor

L'*Environmental Complexity Factor* (ECF) valuta la complessità dell'ambiente operativo in cui il sistema sarà implementato. Questo fattore considera le interfacce utente, i requisiti di sicurezza e altri elementi ambientali. L'ECF viene combinato con i punti dei casi d'uso per stimare lo sforzo di sviluppo necessario. L'assegnazione dell'ECF viene effettuata utilizzando una scala predefinita che attribuisce un valore numerico a ciascun fattore ambientale.

Fattori D'Ambiente	Peso	Valore	Peso x Valore
Familiarità con i processi di sviluppo	1.5	2	3
Esperienza nelle applicazioni	0.5	2	1
Esperienza nella programmazione	1	3	3
Capacità di analisi	0.5	4	2
Motivazione	1	5	5
Stabilità	2	5	10
Part Time Staff	-1	0	0
Linguaggi di programmazione	-1	3	-3

$$Efactor = 21$$

$$ECF = 1.4 + (-0.03 * Efactor) = 1.4 + (-0.03 * 21) = \mathbf{0.77}$$

5.7.4 Use Case Points

Infine, abbiamo valutato gli *Use Case Points* (UCP) combinando i fattori precedentemente calcolati tramite la specifica formula. Nel nostro caso, l'Unadjusted Actor Weight (UAW) non è stato stimato in quanto l'attore unico, il player, è di tipologia Simple. Pertanto, abbiamo assegnato un valore di 1 all'UAW. Utilizzando i valori calcolati, siamo in grado di calcolare gli UCP totali per il sistema e ottenere una stima più accurata degli sforzi necessari per lo sviluppo.

$$\begin{aligned}
 UCP &= (UUCW + UAW) * TCF * ECF = \\
 &= (70 + 1) * 1.04 * 0.77 = \mathbf{56.86}
 \end{aligned}$$

Abbiamo anche stimato le ore/uomo considerando una media di 7 ore per caso d'uso, tenendo conto delle 9 settimane dedicate allo sviluppo dell'applicazione e del numero dei membri del team (4). Di conseguenza, abbiamo ottenuto una media di circa *11 ore settimanali per persona*. Utilizzando gli Use Case Points (UCP) calcolati in precedenza, abbiamo quindi stimato lo sforzo complessivo attraverso la formula per l'Estimate Effort.

$$\begin{aligned}
 UCP * 7 &= 412 \\
 \frac{412}{9 * 4} &\cong 11 \text{ ore}
 \end{aligned}$$

5.8 Component Diagram

Lo scopo principale del diagramma dei componenti è quello di rappresentare la struttura interna del sistema software modellato in termini dei suoi componenti principali, mostrare le relazioni strutturali e le dipendenze tra essi e suddividere il sistema in vari livelli di funzionalità. Per *componenti* si intendono unità autonome e incapsulate all'interno di un sistema o sottosistema che forniscono una o più interfacce e che incapsulano un comportamento. Data la presenza di diverse interdipendenze nel contesto di sviluppo, abbiamo ritenuto opportuno inserire questo diagramma all'interno della documentazione.

Il componente da noi realizzato è l'Editor che utilizza l'interfaccia esposta dal Round Loader nel momento in cui lo studente avvia una partita, per ottenere le informazioni da mantenere come l'Id della partita e la classe under test.

Analogamente, l'Editor può fare richiesta dei servizi di compilazione ed esecuzione messi a disposizione dal componente Compiler, passandogli il codice editato dall'utente e tutti gli altri dati richiesti dall'interfaccia.

Allo stesso modo possiamo descrivere l'interazione che vige tra l'Editor e la Game Repository per il salvataggio delle informazioni riguardanti la partita in corso.

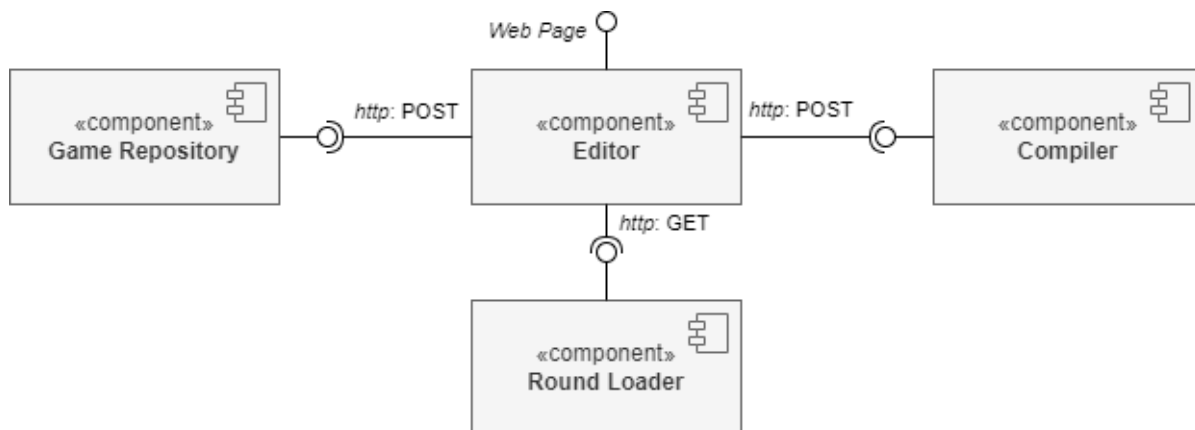


Figura 5.11 – Component Diagram, UML

5.9 Deployment Diagram

Una delle principali prospettive architetture è quella di deployment, utile per verificare se un sistema sarà in grado o meno di soddisfare i suoi requisiti una volta realizzato; essa permette di avere una vista del sistema all'atto della collocazione fisica (deployment), indicando come i vari componenti saranno distribuiti sull'hardware e di avere una rappresentazione dell'architettura. L'organizzazione del progetto prevede la divisione in due cartelle principali: *editor* e *EditorApp*.

La prima contiene il codice relativo alla realizzazione della View, mediante tre sotto-cartelle e due artefatti principali:

- la cartella "src" contiene i file CSS e Javascript utilizzati per l'implementazione dell'interfaccia grafica della nostra applicazione

- la cartella "node_modules" contiene i moduli importati all'interno dei file Javascript
- la cartella "public" contiene il file *index.html* in cui si trova il codice della pagina web
- nel file "package.json" sono definite le dipendenze del progetto React
- il file "Dockerfile" serve per l'installazione del Front-End su Docker

La seconda cartella contiene invece il codice relativo alla realizzazione di Controller e Model, suddiviso tra i seguenti:

- la cartella "src" contiene i file Java utilizzati per l'implementazione dei package *Controller* e *Model* e delle rispettive classi
- la cartella "target" contiene il file *EditorApp-0.0.1-SNAPSHOT.jar* che corrisponde al risultato della compilazione del progetto Spring MVC
- nel file "pom.xml" sono definite le dipendenze del progetto
- il file "Dockerfile" serve per l'installazione del Back-End su Docker

Inoltre, nella cartella principale sono presenti altri due artefatti importanti:

- il file ".env" che permette di modificare l'url dei tre server esterni con i quali si interfaccia il nostro Back-End
- il file "docker-compose.yml" che consente di installare l'intera applicazione su Docker (compresi tre server fittizi creati da noi per simulare i servizi offerti dai task con cui la nostra applicazione si interfaccia)

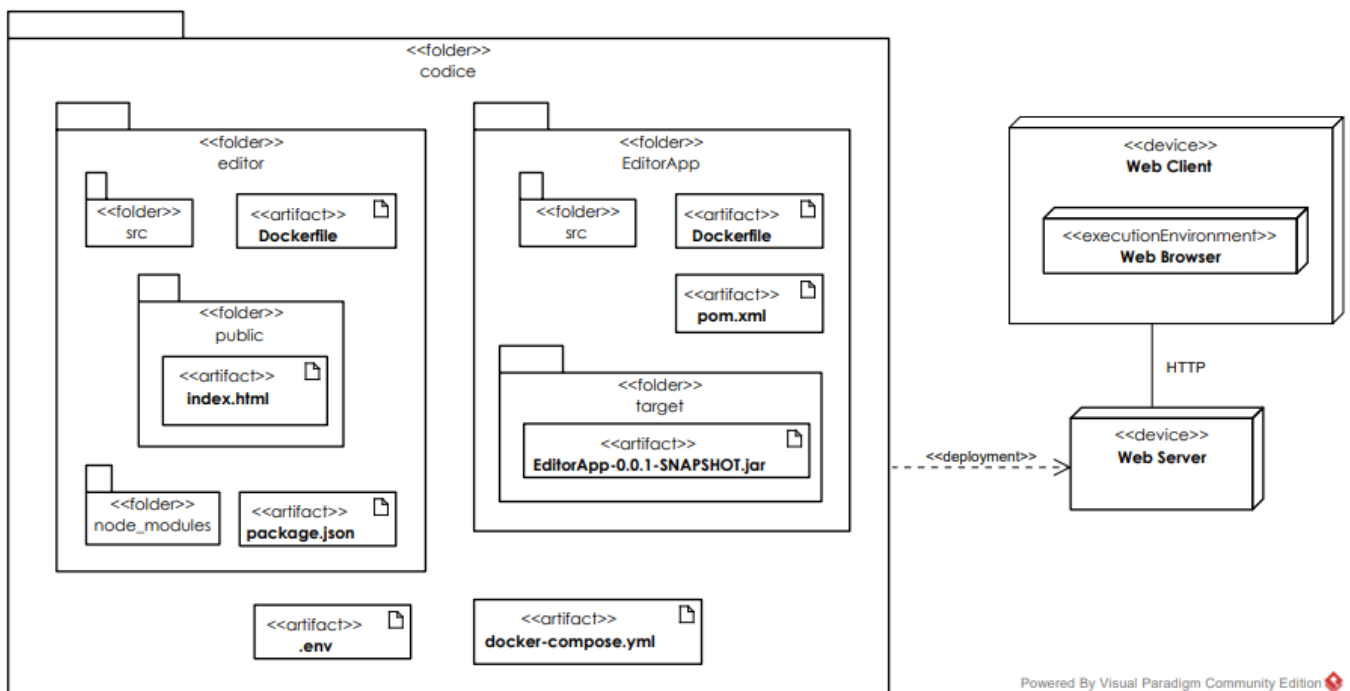


Figura 5.12 – Deployment Diagram, UML

L'applicazione viene infine distribuita in un Web Server che mediante richieste HTTP si interfaccia con un Web Client. Al suo interno è in esecuzione un Browser che permette all'utente di usufruire dei servizi di editing implementati.

6. Documentazione API

6.1 Rest API

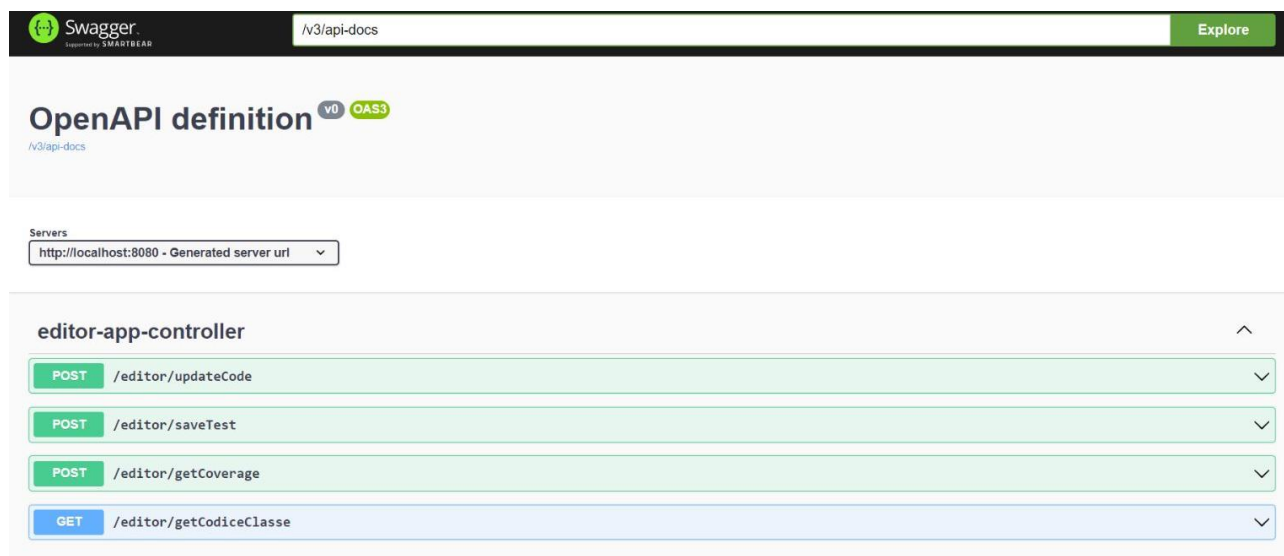


Figura 6.1 – API docs relative all'editor, Swagger

REST è uno **stile architetturale** client-server based organizzato su un piccolo insieme di operazioni di creazione, lettura, aggiornamento e cancellazione (CRUD) e un unico schema di indirizzamento. Lo stile RESTful si basa sul trasferimento delle rappresentazioni delle risorse da un server ad un client, ma la risorsa (un elemento di dati) esiste indipendentemente dalla sua rappresentazione (ad esempio una risorsa Capitolo Libro può avere 3 diverse rappresentazioni: pdf, word e RTF). In REST ogni risorsa ha un identificativo univoco, la URI e ad ogni risorsa sono associate 4 operazioni di creazione, lettura, aggiornamento e modifica (POST, GET, SET, DELETE), tramite cui ogni client HTTP può parlare con qualsiasi server HTTP senza ulteriori configurazioni e accedere direttamente ai dati sfruttando i protocolli http o https; inoltre, REST è pensato per essere autodescrittivo e, nel migliore dei casi, è un protocollo stateless. Grazie ai pochi vincoli che REST impone, è caratterizzato da una grande semplicità e i servizi RESTful comportano un sovraccarico inferiore rispetto ai cosiddetti "grandi servizi web"; inoltre, i servizi RESTful non sono vincolati ad usare la rappresentazione XML dei dati, ma possono anche usarne altre, come ad esempio JSON (Javascript Object Notation) che sono ottimizzate e danno minore overhead.

Tuttavia, REST è caratterizzato da alcune problematiche, come il fatto che non esistono standard per la descrizione di interfacce RESTful (bisogna usare la documentazione informale delle interfacce) e che quando si usano servizi RESTful bisogna creare una propria infrastruttura per monitorare e gestire la qualità e affidabilità dei servizi.

A valle del contesto del nostro applicativo e delle esigenze di performance del task, abbiamo deciso di implementare delle API RESTful, che abbiamo documentato tramite l'utilizzo della piattaforma Swagger, visibili al seguente link <https://app.swaggerhub.com/apis/ZAIRAABDELMAJID/EditorG8/1.0.0> e di cui si riportano gli screen nelle seguenti sezioni.

6.1.1 API updateCode

POST /editor/updateCode

Try it out

Parameters

No parameters

Request body required

application/json

Example Value | Schema

```
"string"
```

Responses

Code	Description	Links
200	OK	No links

Media type

/

Controls Accept header.

Example Value | Schema

```
string
```

Figura 6.2 – API updateCode, Swagger

6.1.2 API saveTest

POST /editor/saveTest

Try it out

Parameters

No parameters

Request body required

application/json

Example Value | Schema

```
{  "msg": "string"}
```

Responses

Code	Description	Links
200	OK	No links

Media type

/

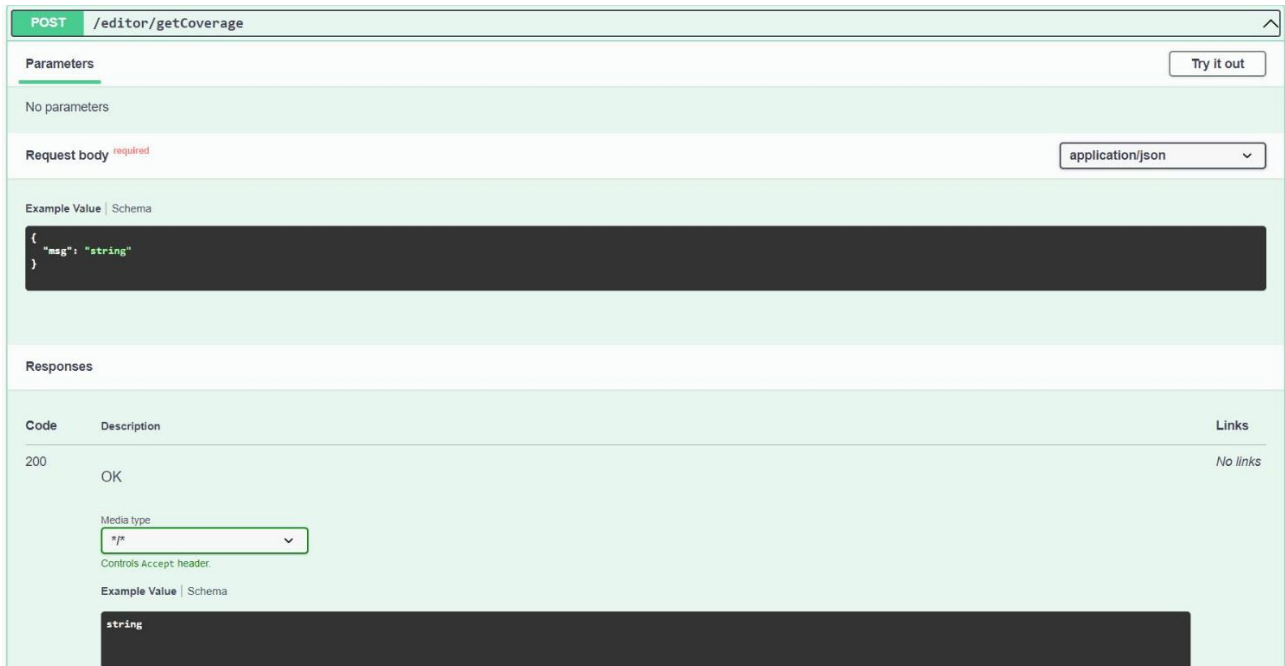
Controls Accept header.

Example Value | Schema

```
string
```

Figura 6.3 – API saveTest, Swagger

6.1.3 API getCoverage



The image shows the Swagger UI for the POST endpoint `/editor/getCoverage`. The interface is divided into several sections:
1. **Method and Path:** A green header bar at the top shows the method `POST` and the path `/editor/getCoverage`.
2. **Parameters:** A section below the header with a 'Try it out' button. It states 'No parameters'.
3. **Request body:** A section labeled 'Request body' with a red 'required' tag. It has a dropdown menu set to 'application/json'.
4. **Example Value:** A dark box containing a JSON object: `{ "msg": "string" }`.
5. **Responses:** A table with columns 'Code', 'Description', and 'Links'. It lists a 200 status code with the description 'OK' and 'No links'.
6. **Media type:** A dropdown menu set to '*/*' with a note 'Controls Accept header.'
7. **Example Value:** A dark box containing the text 'string'.

Figura 6.4 – API getCoverage, Swagger

6.1.4 API getCodiceClasse



The image shows the Swagger UI for the GET endpoint `/editor/getCodiceClasse`. The interface is divided into several sections:
1. **Method and Path:** A blue header bar at the top shows the method `GET` and the path `/editor/getCodiceClasse`.
2. **Parameters:** A section below the header with a 'Try it out' button. It states 'No parameters'.
3. **Responses:** A table with columns 'Code', 'Description', and 'Links'. It lists a 200 status code with the description 'OK' and 'No links'.
4. **Media type:** A dropdown menu set to '*/*' with a note 'Controls Accept header.'
5. **Example Value:** A dark box containing the text 'string'.

Figura 6.5 – API getCodiceClasse, Swagger

7. Testing

L'attività di testing è stata suddivisa in diverse fasi per poter provare il corretto funzionamento dei diversi moduli.

7.1 Testing del Front-End (funzionalità di presentazione)

Le funzionalità testate nel Front-End sono essenzialmente quelle che non richiedono alcuna comunicazione con il controller e che riguardano le operazioni di interazione diretta con l'utente quali:

- Cambio del tema di presentazione
- Download in locale del test
- Scrittura e modifica del codice
- Find
- Replace
- Syntax Highlighting
- Autocompletamento

Ciò richiederà quindi un test del corretto funzionamento di bottoni e finestre di testo. Ovviamente per effettuare tutti i test è necessario preliminarmente avviare il Front-End come in figura

```
C:\Users\franc\Desktop\T6-G8\codice\editor>npm start
> start
> react-scripts start

Browserslist: caniuse-lite is outdated. Please run:
  npx browserslist@latest --update-db
  Why you should do it regularly: https://github.com/browserslist/browserslist#browsers-data-updating
(node:25528) [DEP_WEBPACK_DEV_SERVER_ON_AFTER_SETUP_MIDDLEWARE] DeprecationWarning: 'onAfterSetupMiddleware' option is deprecated. Please use the 'setupMiddlewares' option.
(Use 'node --trace-deprecation ...' to show where the warning was created)
(node:25528) [DEP_WEBPACK_DEV_SERVER_ON_BEFORE_SETUP_MIDDLEWARE] DeprecationWarning: 'onBeforeSetupMiddleware' option is deprecated. Please use the 'setupMiddlewares' option.
Starting the development server...
Compiled successfully!
```

Figura 7.1 – Esecuzione del Front-End

Successivamente si procede con Postman a testare le diverse API.

7.2.1 *getCodiceClasse*

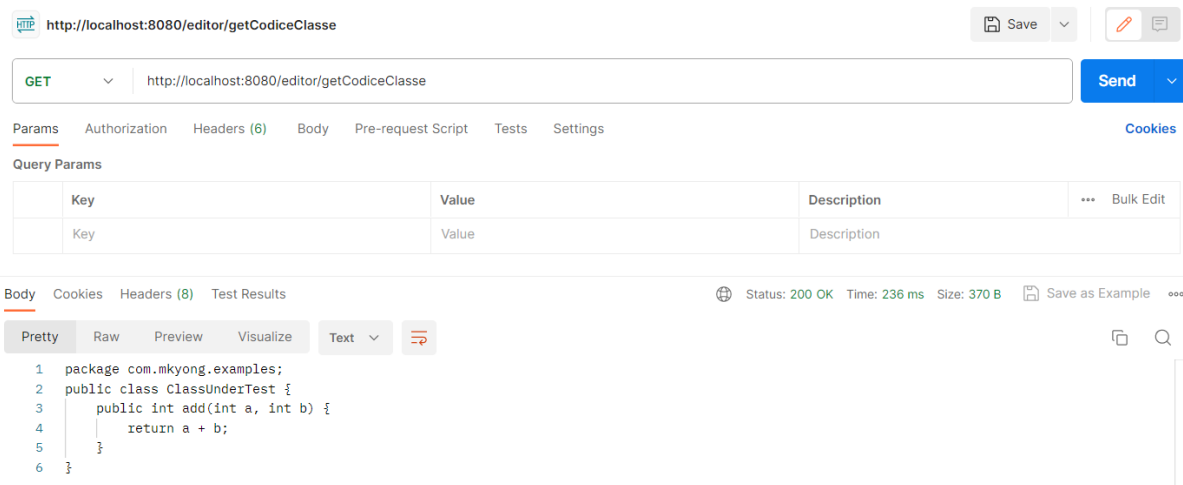


Figura 7.3 – Test Postman API *getCodiceClasse*

7.2.2 *updateCode*

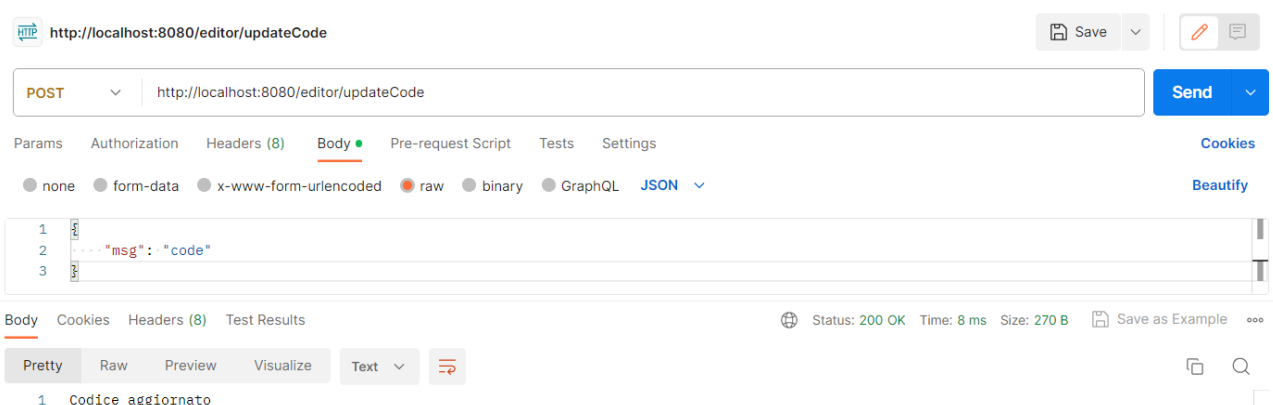


Figura 7.4 – Test Postman API *updateCode*

7.2.3 *saveTest*

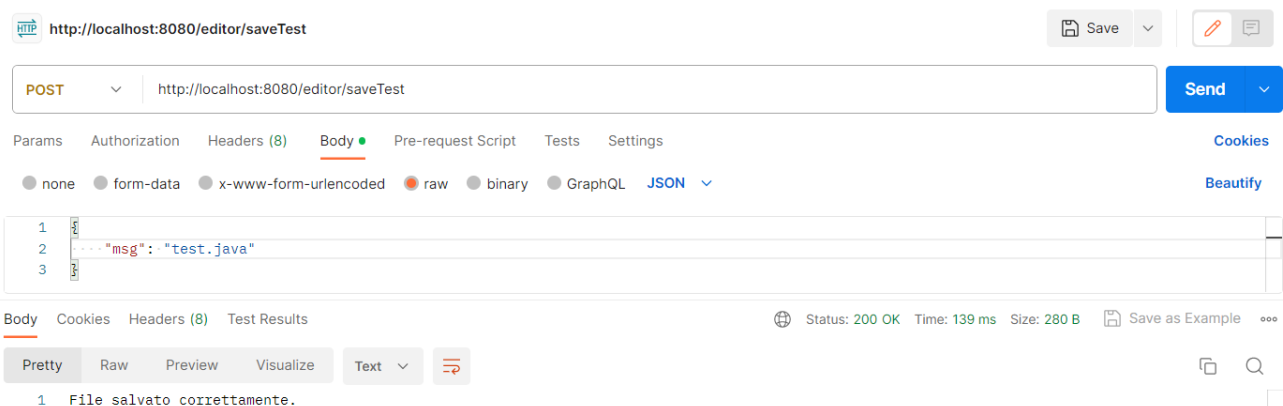


Figura 7.5 – Test Postman API *saveTest*

7.2.4 *getCoverage*

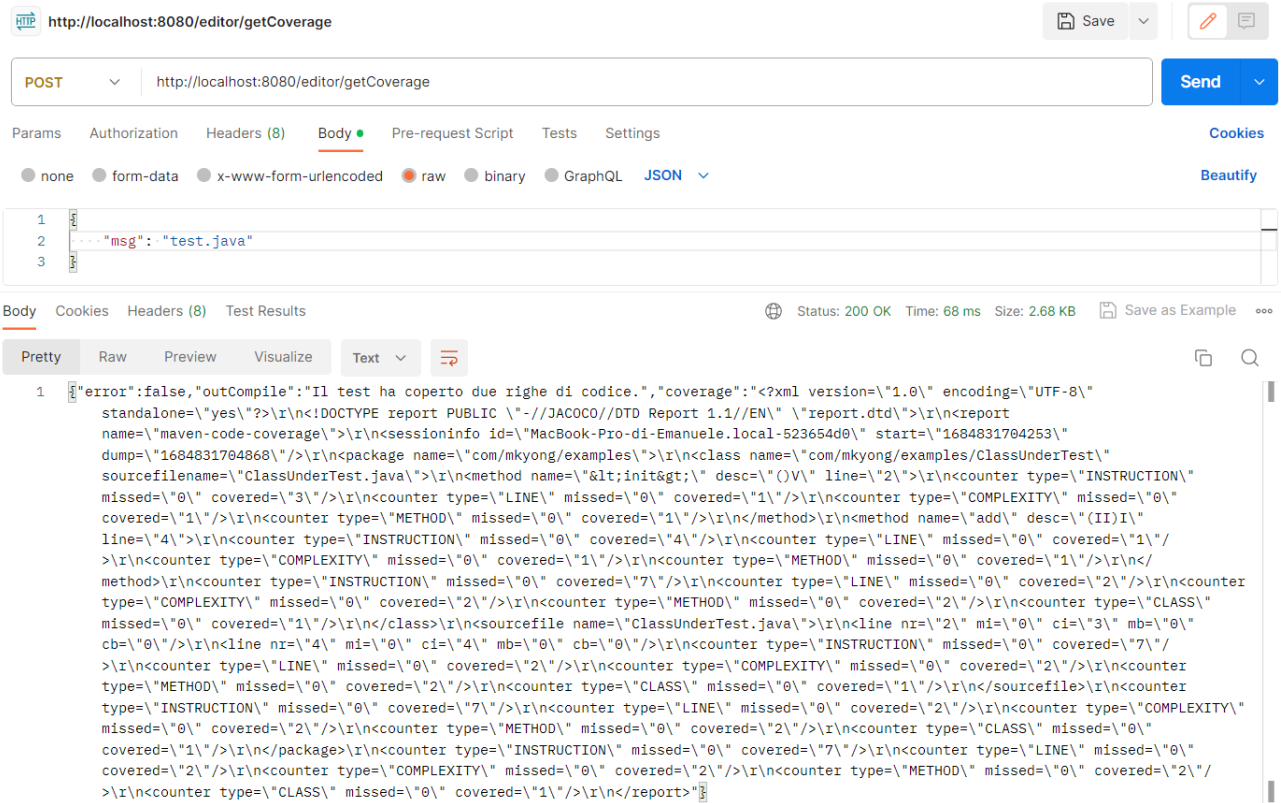


Figura 7.5 – Test Postman API getCoverage

7.3 Test di integrazione di Front-End e Back-End

Si collegano adesso Front-End e Back-End, testando se la pressione dei tasti attiva le funzionalità attese e dà gli output desiderati. Sarà quindi necessario avviare preliminarmente il Back-End, il Front-End e i server fittizi relativi agli altri task

```
C:\Users\franc>cd Desktop
C:\Users\franc\Desktop>cd T6-G8
C:\Users\franc\Desktop\T6-G8>cd codice
C:\Users\franc\Desktop\T6-G8\codice>cd ClassServer
C:\Users\franc\Desktop\T6-G8\codice\ClassServer>node ClassServer.js
Server in ascolto sulla porta 3002
|
```

Figura 7.7 – *Esecuzione ClassServer*

```

C:\Users\franc>cd Desktop
C:\Users\franc\Desktop>cd T6-G8
C:\Users\franc\Desktop\T6-G8>cd codice
C:\Users\franc\Desktop\T6-G8\codice>cd CoverageServer
C:\Users\franc\Desktop\T6-G8\codice\CoverageServer>node CoverageServer.js
Server in ascolto sulla porta 3001

```

Figura 7.8 – Esecuzione CoverageServer

```

C:\Users\franc>cd Desktop
C:\Users\franc\Desktop>cd T6-G8
C:\Users\franc\Desktop\T6-G8>cd codice
C:\Users\franc\Desktop\T6-G8\codice>cd TestsServer
C:\Users\franc\Desktop\T6-G8\codice\TestsServer>node TestsServer.js
Server in ascolto sulla porta 3003

```

Figura 7.9 – Esecuzione TestsServer

Si procede poi con il test delle varie funzionalità. La seguente foto rappresenta la situazione prima e dopo la pressione del bottone compile & execute.

Class Under Test

```

1 package com.mkyong.examples;
2 public class ClassUnderTest {
3     public int add(int a, int b) {
4         return a + b;
5     }
6 }

```

Output



Class Under Test

```

1 package com.mkyong.examples;
2 public class ClassUnderTest {
3     public int add(int a, int b) {
4         return a + b;
5     }
6 }

```

Output

Il test ha coperto due righe di codice.

Figura 7.10 – Esito del test di pressione del bottone "Compile & Execute"

8. Installazione e utilizzo dell'applicazione

8.1 Installazione senza l'utilizzo di Docker

1. Effettuare il clone della cartella del progetto: [T6-G8](#)
2. Scaricare l'ultima versione di [Node.js](#)
3. Aprire il command prompt di Node.js e digitare il comando:
 - `npm install`
4. Per eseguire il backend, da shell spostarsi nella cartella *codice/EditorApp/target* e digitare il comando:
 - `java -jar EditorApp-0.0.1-SNAPSHOT.jar`
5. Per eseguire il Front-End, dal prompt di Node.js spostarsi nella cartella *codice/editor* e digitare il comando:
 - `npm start`
6. Se l'applicazione non dovesse partire automaticamente, aprire una pagina del browser e collegarsi all'indirizzo <http://localhost:3000>

Volendo testare l'applicazione con i tre server fittizi creati da noi, bisogna lanciare ognuno di essi singolarmente eseguendo i seguenti passaggi:

7. Per eseguire il server che restituisce la classe da testare, dal prompt di Node.js spostarsi nella cartella *codice/ClassServer* e digitare il comando:
 - `node ClassServer.js`
8. Per eseguire il server che restituisce i risultati di coverage, dal prompt di Node.js spostarsi nella cartella *codice/CoverageServer* e digitare il comando:
 - `node CoverageServer.js`
9. Per eseguire il server che effettua il salvataggio del test, dal prompt di Node.js spostarsi nella cartella *codice/TestsServer* e digitare il comando:
 - `node TestsServer.js`

8.2 Installazione mediante l'utilizzo di Docker

1. Effettuare il clone della cartella del progetto: [T6-G8](#)
2. Scaricare l'ultima versione di [Docker Desktop](#)
3. Aprire Docker Desktop
4. Per costruire ed eseguire il container del Back-End, da shell spostarsi nella cartella *codice/EditorApp* e digitare i comandi:
 - `docker build -t backend .`
 - `docker run -p 8080:8080 backend`
- 4.1. In caso si volessero cambiare gli url dei server esterni con cui si interfaccia il Back-End, bisogna sostituire l'ultimo comando con il seguente:
 - `docker run -p 8080:8080 -e COVERAGE_SERVER_URL=http://my-coverage-server:3001/ -e CLASS_SERVER_URL=http://my-class-server:3002/ -e TESTS_SERVER_URL=http://my-tests-server:3003/ backend`

5. Per costruire ed eseguire il container del Front-End, da shell spostarsi nella cartella *codice/editor* e digitare i comandi:
 - `docker build -t frontend .`
 - `docker run -p 3000:3000 frontend`
6. Aprire una pagina del browser e collegarsi all'indirizzo <http://localhost:3000>

Volendo testare l'applicazione con i tre server fittizi creati da noi, dopo aver eseguito i primi tre passaggi bisogna effettuare le seguenti operazioni:

7. Cambiare eventualmente l'url dei server esterni modificando il file *.env* situato all'interno della cartella *codice*
8. Per costruire ed eseguire tutti i container, da shell spostarsi nella cartella *codice* e digitare il comando:
 - `docker-compose up`
9. Aprire una pagina del browser e collegarsi all'indirizzo <http://localhost:3000>

È possibile cambiare la porta esposta sul container Docker con l'opzione "-p" quando si lancia il comando: "`docker run -p <porta_origine>:<porta_destinazione> ...`".

8.3 Utilizzo dell'applicazione

Dopo aver eseguito l'installazione e avviato l'applicazione, sarà possibile modificare il codice presente all'interno della finestra di editing ed effettuare le operazioni di cambiamento del tema di scrittura e salvataggio in locale del test mediante la pressione del pulsante *Download Test*. Eseguendo anche i tre server fittizi, si potrà visualizzare la classe da testare nella finestra in alto a destra, salvare in remoto il codice prodotto attraverso l'utilizzo del pulsante *Save Test*, e avviare le operazioni di compilazione ed esecuzione del test. Verrà generato il risultato di una compilazione simulata che sarà visualizzato come esito di coverage, sovrascrivendo la classe under test, e stringa di output, mostrata all'interno della finestra in basso a destra.

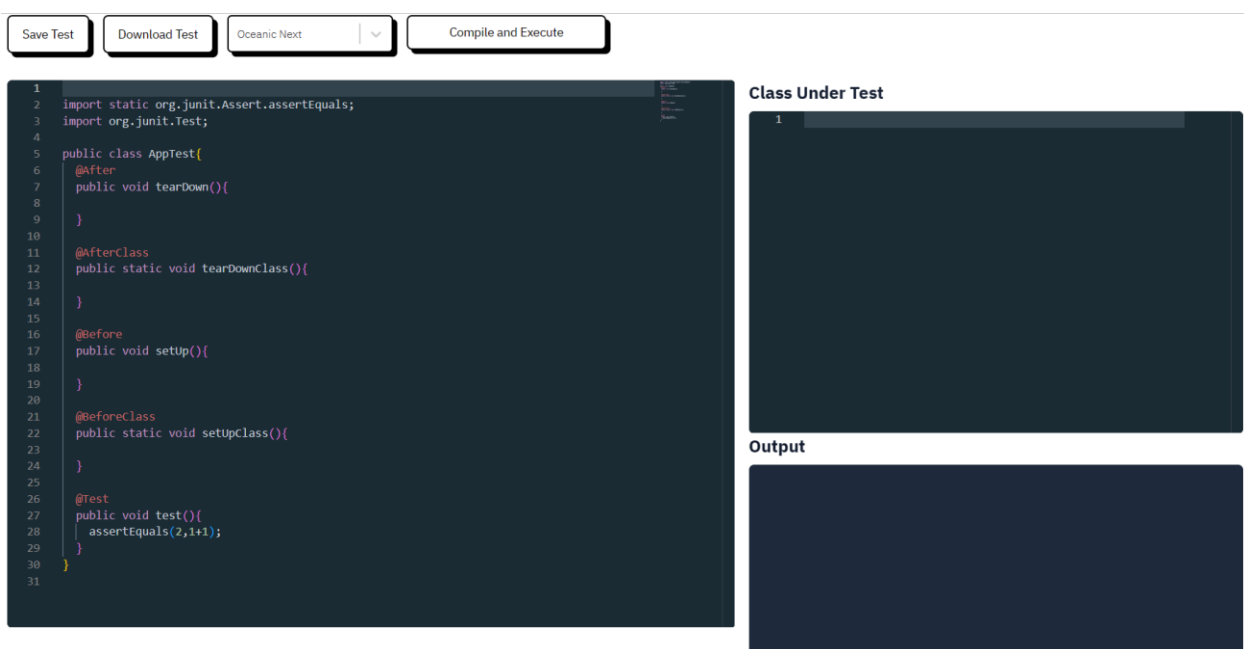


Figura 8.1 – Aspetto della Web Page sviluppata