

UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

**CORSO DI LAUREA IN INGEGNERIA INFORMATICA**  
**CORSO DI SOFTWARE ARCHITECTURE DESIGN**  
**PROF. A.R. FASOLINO - A.A. 2022/2023**

***Progetto SAD***

**Requisito T6**

**Gruppo G16**

Anna  
Marta Assunta  
Giovanni

Varriale  
Sichinolfi  
Castaldi

M63001247  
M63001248  
M63001262

[anna.varriale6@studenti.unina.it](mailto:anna.varriale6@studenti.unina.it)  
[m.sichinolfi@studenti.unina.it](mailto:m.sichinolfi@studenti.unina.it)  
[giovann.castaldi@studenti.unina.it](mailto:giovann.castaldi@studenti.unina.it)

# INDICE

<b>1. Introduzione .....</b>	<b>1</b>
1.1 Descrizione del task T6 .....	1
1.2 Tecnologie .....	1
<b>2. Specifica dei requisiti .....</b>	<b>2</b>
2.1 Glossario dei termini .....	2
2.2 Classificazione dei requisiti .....	2
2.2.1 Requisiti funzionali .....	2
2.3 Modellazione dei casi d'uso .....	3
2.3.1 Attori e casi d'uso .....	3
2.3.2 Diagramma dei casi d'uso .....	2
2.3.3 Scenari .....	3
2.4 Storie Utente .....	5
2.5 Diagrammi di sequenza e diagrammi di attività .....	7
2.5.1 Diagramma di sequenza .....	8
2.5.2 Activity diagram (Compile, Run, NewFile) .....	9
<b>3. Processo di sviluppo .....</b>	<b>11</b>
3.1 Scrum .....	11
<b>4. Stima dei costi .....</b>	<b>15</b>
4.1 Unadjusted Use Case Weight (UUCW) .....	15
4.2 Unadjusted Actor Weight (UAW) .....	15
4.3 Unadjusted Use-Case Points (UUCP) .....	15
4.4 Complexity Factor .....	16
4.5 Use Case Points (UCP) .....	17
<b>5. Architettura e Progettazione .....</b>	<b>18</b>
5.1 Pattern Architetturali .....	18
5.1.1 Stile client-server .....	18
5.1.2 Pattern MVC .....	19
5.2 Componenti del sistema .....	21
5.2.1 Diagramma dei componenti .....	21
5.2.2 Diagramma di comunicazione .....	22
5.2.3 Sequence Diagram .....	23
5.3 Class e Package Diagram .....	24
5.3.1 Class Diagram .....	24
5.3.2 Package Diagram .....	25
5.4 System Context Diagram .....	26
<b>6. Implementazione .....</b>	<b>27</b>
6.1 CODE-EDITOR .....	27
6.2 code-editor.html (senza integrazioni) .....	29
.....	30
6.3 Deployment diagram .....	42
<b>7. Testing .....</b>	<b>43</b>
7.1 Test di interfaccia .....	43

# 1. Introduzione

## 1.1 Descrizione del task T6

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.

## 1.2 Tecnologie

Il requisito è stato implementato attraverso i seguenti linguaggi:

- **HTML**
- **CSS**
- **JavaScript**

In particolare, abbiamo utilizzato il framework **Spring**

L'ambiente di sviluppo che andremo ad utilizzare sarà Visual Studio Code. L'idea è quella di integrare i codici relativi al settaggio di CodeMirror (da reperire tramite GitHub) ai nostri script. **CodeMirror** è un code editor open source realizzato in JavaScript e pensato per l'inclusione all'interno delle pagine dei siti Web; grazie ad esso è possibile implementare un semplice campo di testo con diverse funzionalità, tra cui l'evidenziazione automatica della sintassi e l'auto completamento. Il campo di testo creato verrà personalizzato in base alle nostre necessità ovvero in base alla sintassi del linguaggio Java:

1. Indentazione
2. Segnalazione errori di sintassi
3. Apertura e chiusura automatica delle parentesi
4. Auto completamento (parole, funzioni parametri)

La finestra in cui lo studente può inserire il codice per la classe di test mostrerà, in base alla classe caricata, un template che varierà con la classe da testare.

## 2. Specifica dei requisiti

### 2.1 Glossario dei termini

Termine	Descrizione
Classe da testare	Classe Java per la quale lo studente deve produrre un codice di test
Classe di test	Classe Java di test per la classe da testare scritta dallo studente
IDE	Software che supporta i programmatori nello sviluppo e debugging del codice sorgente di un programma
Template del caso di test	Struttura generica della classe di test da completare

### 2.2 Classificazione dei requisiti

In questa sezione sono riportati i diagrammi di analisi realizzati. Si vuole ancora sottolineare come si è scelto di realizzare solo i diagrammi strettamente necessari alla comprensione del sistema e delle sue funzionalità più complesse

#### 2.2.1 Requisiti funzionali

##### ID      Requisito

- RF01      Il sistema deve offrire allo studente la possibilità di creare un nuovo file
- RF02      Il sistema deve offrire allo studente la possibilità di salvare la classe java creata
- RF03      Il sistema deve offrire allo studente possibilità di compilare la classe di test
- RF04      Il sistema deve offrire allo studente la possibilità di eseguire la classe di test
- RF05      Il sistema deve offrire allo studente la possibilità di cambiare tema all'editor
- RF06      Il sistema deve offrire allo studente la funzionalità di ricercare parole nei codici
- RF07      Il sistema deve offrire allo studente la possibilità di generare getter, setter e toString
- RF08      Il sistema deve offrire allo studente la possibilità di inserire la classe di test

## 2.3 Modellazione dei casi d'uso

### 2.3.1 Attori e casi d'uso

#### Attori Primari:

- Studente

#### Casi d'uso:

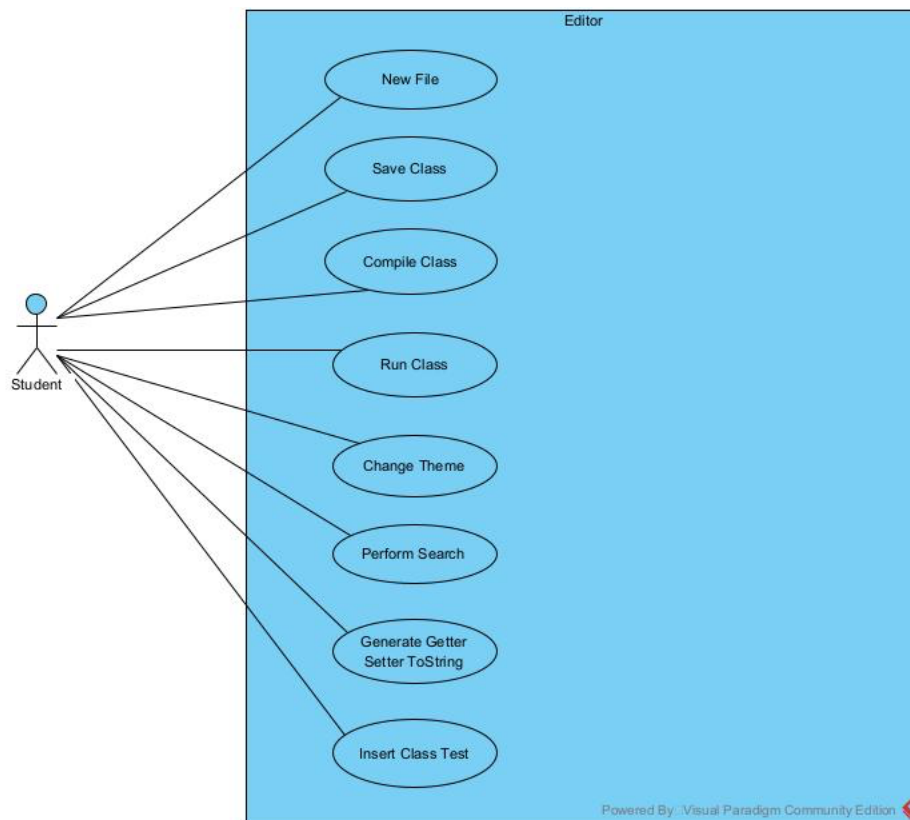
- UC1: Crea File
- UC2: Salva Classe
- UC3: Compila Classe
- UC4: Esegui Classe
- UC5: Cambia Tema
- UC6: Ricerca Parola
- UC7: Genera Getter, Setter, toString
- UC8: Inserisci classe di test

Caso d'uso	Attori Primari	Incl. / Ext.	Requisiti corrispondenti
UC1. Nuovo File	Studente	-	RF01
UC2. Salva Classe	Studente	-	RF02
UC3. Compila Classe	Studente	-	RF03
UC4. Esegui Classe	Studente	-	RF04
UC5. Cambia Tema	Studente	-	RF05
UC6. Ricerca Parola	Studente	-	RF06
UC7. Genera Getter Setter ToString	Studente	-	RF07
UC8. Inserisci Classe di Test	Studente	-	RF08

### 2.3.2 Diagramma dei casi d'uso

Il diagramma dei Casi d'Uso è un tipo di diagramma UML che abbiamo utilizzato al fine di esplicitare graficamente i casi d'uso definiti in fase di analisi e le rispettive relazioni di include che li caratterizzano, oltre all'attore primario che esegue tali casi.

Tramite il seguente diagramma si va quindi ad esplicitare il modo in cui l'attore interagisce col sistema stesso.



### 2.3.3 Scenari

Caso d'uso:	Crea File
Attore primario	Studente
Descrizione	Il sistema deve offrire allo studente la possibilità di creare un nuovo file
Pre-Condizioni	-
Sequenza di eventi principale	<ol style="list-style-type: none"> <li>1. Lo studente preme il bottone di creazione file</li> <li>2. Il sistema chiede conferma all'utente di salvare prima di creare un nuovo file</li> <li>3. Lo studente conferma l'intenzione di voler salvare il file corrente</li> <li>4. Il sistema salva il file corrente</li> <li>5. Il sistema mostra il nuovo file nell'editor</li> </ol>
Post-Condizioni	Il sistema ha creato un nuovo file per la classe di test
Casi d'uso correlati	-
Sequenza di eventi alternativi	<ol style="list-style-type: none"> <li>3. Lo studente ha già salvato il file corrente</li> <li>4. Il sistema mostra il nuovo file nell'editor</li> </ol>

Caso d'uso:	Salva Classe
Attore primario	Studente
Descrizione	Il sistema deve offrire allo studente la possibilità di salvare la classe java creata
Pre-Condizioni	-
Sequenza di eventi principale	<ol style="list-style-type: none"> <li>1. Lo studente preme il bottone di salvataggio classe</li> <li>2. Il sistema salva la classe di test</li> </ol>
Post-Condizioni	Il sistema ha salvato la classe di test
Casi d'uso correlati	-

Caso d'uso:	Compila Classe
Attore primario	Studente
Descrizione	Il sistema deve offrire allo studente la possibilità di compilare la classe di test
Pre-Condizioni	-
Sequenza di eventi principale	<ol style="list-style-type: none"> <li>1. Lo studente preme il bottone di compilazione</li> <li>2. Il sistema compila la classe di test</li> <li>3. Il sistema mostra gli eventuali errori nella console</li> </ol>
Post-Condizioni	Il sistema ha compilato la classe di test
Casi d'uso correlati	-
Sequenza di eventi alternativi	-

Caso d'uso:	Inserisci Classe Di Test
Attore primario	Studente
Descrizione	Il sistema deve offrire allo studente la possibilità di inserire la classe di test
Pre-Condizioni	-
Sequenza di eventi principale	<ol style="list-style-type: none"> <li>1. Lo studente inserisce la classe di test all'interno dell'editor</li> <li>2. Il sistema mostra l'evidenziazione delle parole, il code folding, autocompletamento, chiusura automatica delle parentesi, indentazione, dot notation.</li> </ol>
Post-Condizioni	Il sistema permette di visualizzare il codice prodotto nell'editor Class To Test
Casi d'uso correlati	-
Sequenza di eventi alternativi	-

<b>Caso d'uso: Esegui Classe</b>	
<b>Attore primario</b>	Studente
<b>Descrizione</b>	Il sistema deve offrire allo studente la possibilità di eseguire la classe di test
<b>Pre-Condizioni</b>	-
<b>Sequenza di eventi principale</b>	<ol style="list-style-type: none"> <li>1. Lo studente preme il bottone di esecuzione del test</li> <li>2. Il sistema esegue la classe di test</li> <li>3. Il sistema mostra la copertura della classe da testare nell'apposito editor</li> </ol>
<b>Post-Condizioni</b>	Il sistema ha eseguito la classe di test
<b>Casi d'uso correlati</b>	-
<b>Sequenza di eventi alternativi</b>	-

<b>Caso d'uso: Cambia Tema</b>	
<b>Attore primario</b>	Studente
<b>Descrizione</b>	Il sistema deve offrire allo studente la possibilità di cambiare il tema dell'editor
<b>Pre-Condizioni</b>	-
<b>Sequenza di eventi principale</b>	<ol style="list-style-type: none"> <li>1. Lo studente preme il pulsante per il cambio tema</li> <li>2. Il sistema mostra una lista di temi da poter selezionare</li> <li>3. Lo studente seleziona un tema dalla lista di temi visualizzati</li> <li>4. Il sistema cambia il tema dell'editor</li> </ol>
<b>Post-Condizioni</b>	Il sistema ha cambiato il tema dell'editor
<b>Casi d'uso correlati</b>	-

<b>Caso d'uso: Ricerca Parola</b>	
<b>Attore primario</b>	Studente
<b>Descrizione</b>	Il sistema deve offrire allo studente la funzionalità di ricercare parole nei codici
<b>Pre-Condizioni</b>	-
<b>Sequenza di eventi principale</b>	<ol style="list-style-type: none"> <li>1. Lo studente inserisce la parola da cercare</li> <li>2. Lo studente preme sull'apposito pulsante</li> <li>3. Il sistema mostra sugli editor la parola ricercata evidenziandola</li> </ol>
<b>Post-Condizioni</b>	Il sistema ha ricercato la parola all'interno degli editor
<b>Casi d'uso correlati</b>	-

<b>Caso d'uso: Genera Getter, Setter, toString</b>	
<b>Attore primario</b>	Studente
<b>Descrizione</b>	Il sistema deve offrire allo studente la possibilità di generare Getter, Setter e toString
<b>Pre-Condizioni</b>	-
<b>Sequenza di eventi principale</b>	<ol style="list-style-type: none"> <li>1. Lo studente preme sul bottone di Getter, Setter e toString</li> <li>2. Il sistema genera i Getter, Setter e toString relativi alle variabili definite</li> </ol>
<b>Post-Condizioni</b>	Il sistema ha generato Getter, Setter e toString
<b>Casi d'uso correlati</b>	-
<b>Sequenza di eventi alternativi</b>	-



## 2.4 Storie Utente

Si è scelto di documentare i requisiti del sistema attraverso i meccanismi delle storie utente. Ognuna di esse è caratterizzata da:

- titolo;
- descrizione: breve frase che descrive una funzionalità dell'applicazione dal punto di vista di un attore specifico, scritta utilizzando il seguente formato:
  - come [attore]
  - io voglio [funzionalità]
  - in modo da [motivo]
- priorità: indica il grado di importanza della storia utente all'interno del sistema;
- story points: unità di misura per esprimere una stima dello sforzo richiesto (effort) per implementare completamente la storia utente. Per valutare l'effort, è stato definito un insieme discreto di valori numerici da 1 a 5, dove 1 indica lo sforzo minimo e 5 lo sforzo massimo.

Di seguito sono riportate le storie utente realizzate per la specifica dei requisiti.

Storia Utente	
<b>Titolo</b>	Inserisci Classe Di Test
<b>Descrizione</b>	Come utente Io voglio inserire la classe di test In modo da poter aggiungere una classe
<b>Priorità</b>	Massima
<b>Story Points</b>	3

Storia Utente	
<b>Titolo</b>	Crea File
<b>Descrizione</b>	Come utente Io voglio creare un nuovo file In modo da poterne avere uno nuovo
<b>Priorità</b>	Media
<b>Story Points</b>	2

Storia Utente	
<b>Titolo</b>	Salva Classe
<b>Descrizione</b>	Come utente Io voglio salvare la classe di test In modo da poterla memorizzare
<b>Priorità</b>	Media
<b>Story Points</b>	2

Storia Utente	
<b>Titolo</b>	Compila Classe
<b>Descrizione</b>	Come utente Io voglio compilare la classe di test In modo da poter conoscere eventuali errori
<b>Priorità</b>	Massima
<b>Story Points</b>	3

Storia Utente	
<b>Titolo</b>	Esegui Classe
<b>Descrizione</b>	Come utente Io voglio eseguire la classe di test In modo da poter verificare la correttezza del codice
<b>Priorità</b>	Massima
<b>Story Points</b>	3

Storia Utente	
<b>Titolo</b>	Cambia Tema
<b>Descrizione</b>	Come utente Io voglio cambiare il tema In modo da poter visualizzare interfacce diverse dell'editor
<b>Priorità</b>	Media
<b>Story Points</b>	2

Storia Utente	
<b>Titolo</b>	Ricerca Parola
<b>Descrizione</b>	Come utente Io voglio ricercare le parole negli editor In modo da poter velocizzare la ricerca
<b>Priorità</b>	Minima
<b>Story Points</b>	3

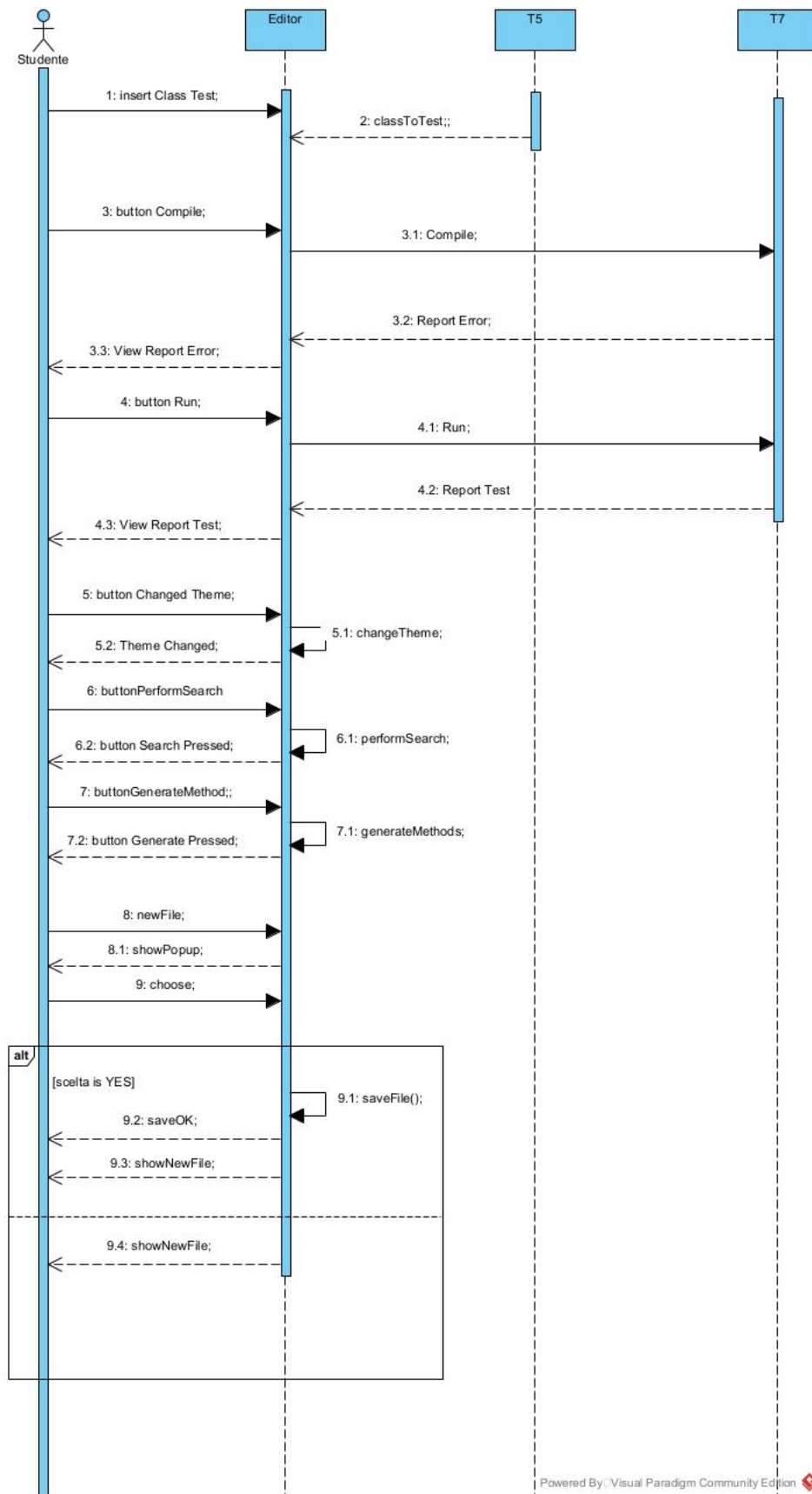
Storia Utente	
<b>Titolo</b>	Genera Getter Setter ToString
<b>Descrizione</b>	Come utente Io voglio generare i getter, setter e ToString In modo da poter velocizzare l'inserimento di queste funzioni
<b>Priorità</b>	Media
<b>Story Points</b>	2

## 2.5 Diagrammi di sequenza e diagrammi di attività

In questo paragrafo sono riportati i Sequence ed Activity Diagram di analisi più significativi per l'applicazione descritta precedentemente. Essi mostrano, per un particolare scenario, la sequenza di eventi generati dall'interazione dell'attore con il sistema.

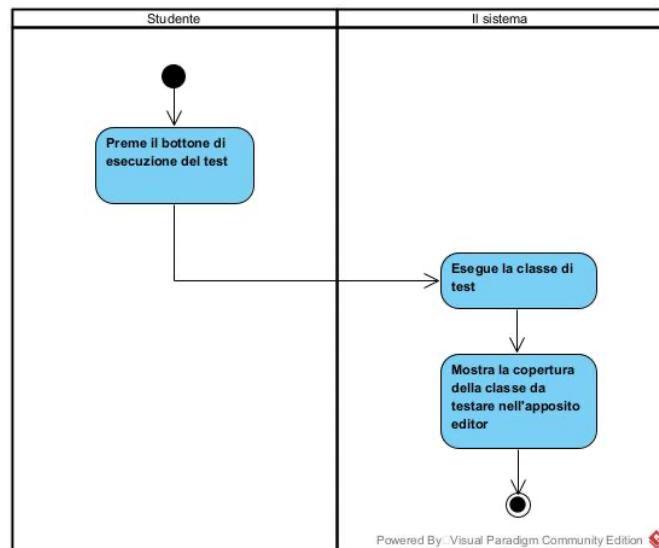
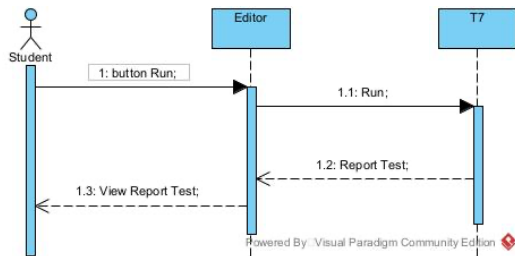
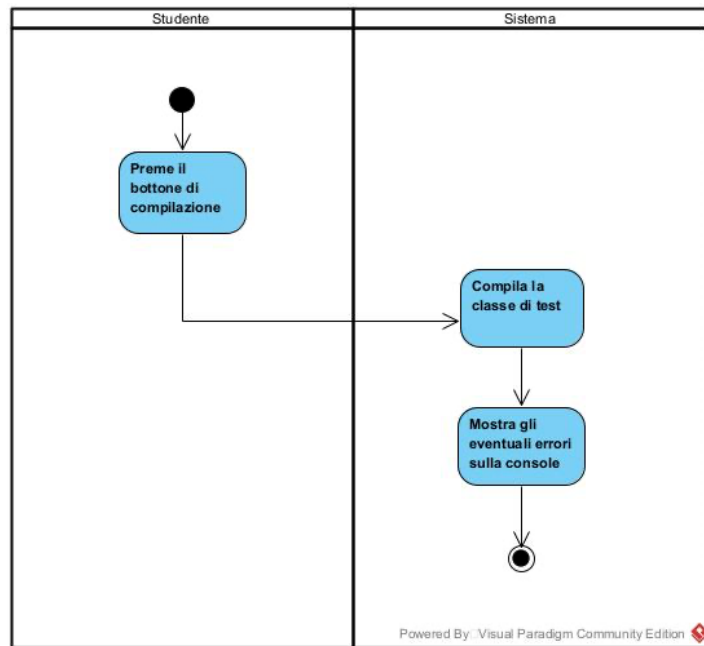
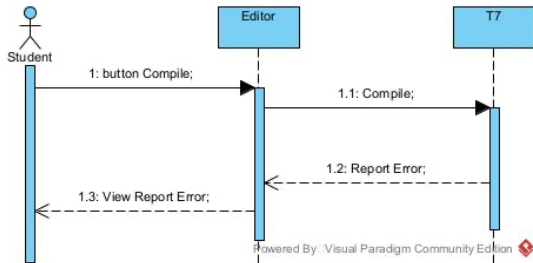
- Un **diagramma di sequenza**: diagramma UML (Unified Modeling Language) che illustra la sequenza dei messaggi tra oggetti in un'interazione. Un diagramma di sequenza consiste in un gruppo di oggetti, rappresentati da lifeline, e nei messaggi che tali istanze si scambiano durante l'interazione.
- Il **diagramma di attività**: diagramma che permette di descrivere un processo attraverso dei grafi in cui i nodi rappresentano le attività e gli archi l'ordine con cui vengono eseguite

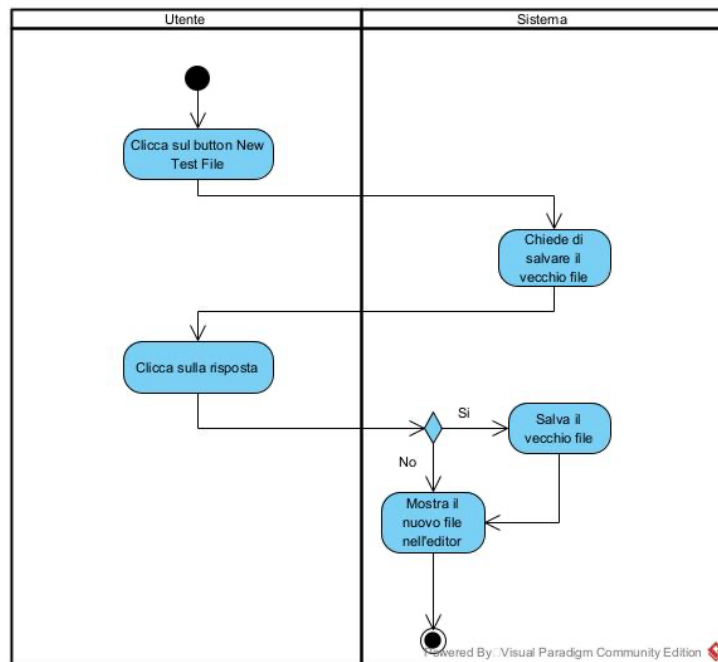
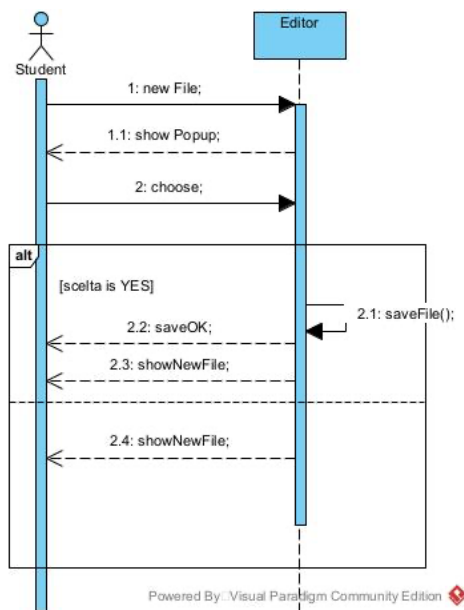
## 2.5.1 Diagramma di sequenza



Powered By: Visual Paradigm Community Edition

## 2.5.2 Activity diagram (Compile, Run, NewFile)





### 3. Processo di sviluppo

Il “Processo Software” si definisce come l'insieme delle attività necessarie allo sviluppo di un sistema software, esse si dividono in:

- Attività portanti: compiti da svolgere necessariamente.
- Attività ausiliarie: compiti che possono aumentare la qualità del software che si vuole produrre.

Le fasi invece indispensabili durante il processo software sono quelle di specifica del software, progettazione, implementazione, validazione e mantenimento del prodotto commissionato. Considerata la complessità relativa ad una tale opera è necessario avere un metodo ed un'organizzazione del lavoro ben studiata in modo da evitare incomprensioni tra i membri del team durante il processo dello sviluppo per mantenere alta la qualità del software che si vuole produrre.

Alla luce di ciò nasce la necessità di dover definire un metodo di lavoro che accompagnerà tutto il processo di sviluppo per far sì da prevenire e risolvere le eventuali difficoltà che si incontreranno durante il progetto.

L'approccio scelto è di tipo incrementale in modo da rilasciare continuamente codice funzionante per un continuo riscontro di ciò che si sta realizzando ed essere più robusti ai possibili errori commessi.

#### 3.1 Scrum

Scrum è un framework agile per la gestione del ciclo di sviluppo del software, interattivo ed incrementale, concepito per gestire progetti e prodotti software. Nasce sull'idea del controllo empirico dei progetti per osservare quello che sta accadendo durante la progettazione in modo da iterare le attività. Si basa sulla trasparenza, ispezione e adattamento.

In Scrum ci sono varie fasi, innanzitutto definiamo che il framework è costituito dal team Scrum, dai ruoli, dagli eventi e dagli artefatti e dalle regole ad essi associati. Dove il team Scrum va a definire il personale per poi i ruoli e creare determinati artefatti.

Le 3 fasi di scrum sono:

- La fase iniziale, ovvero una fase di pianificazione in cui si stabiliscono gli obiettivi generali del progetto. In questa fase si progetta anche l'architettura del software.
- Fase di cicli di Sprint, in cui ogni ciclo sviluppa un incremento del sistema. È un periodo di solito 4 settimane, variabile, in cui tutto il team di lavoro si concentra per raggiungere gli obiettivi di quella iterazione. Ogni iterazione avrà i suoi obiettivi.
- La fase di chiusura del progetto impacchetta il progetto. Completa la documentazione richiesta con schermate di help del sistema e manuali utente.

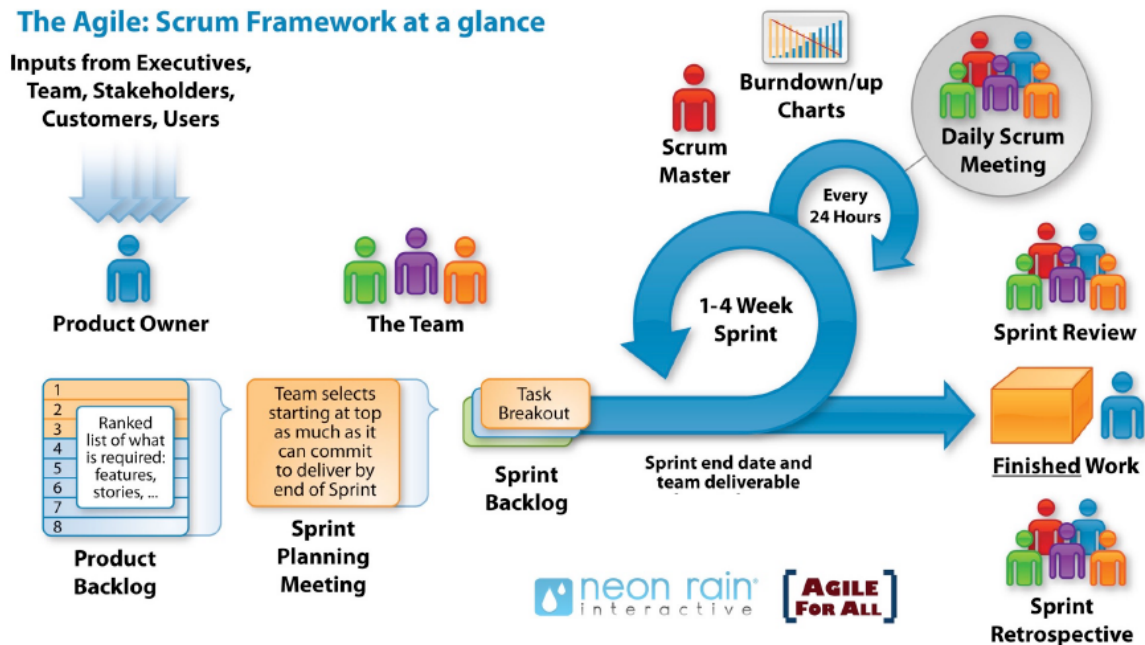
Nel sistema Scrum è presente un team composto da un product owner che prende il nome del project manager il cui obiettivo è definire il lavoro da fare e le caratteristiche del prodotto. È presente il Development Team, di solito formato da non più di 7 persone, auto-organizzate, molto autonomo.

Infine, è presente lo Scrum Master che aiuta il gruppo ad apprendere ed applicare Scrum. Il product owner ha la responsabilità di massimizzare il valore del prodotto e del lavoro svolto dal Team di Sviluppo. Deve avere, inoltre, una grande conoscenza del prodotto e dei requisiti necessari, deve saper gestire gli Stakeholder di progetto ed il suo ruolo principale è quello di produrre valore per l'utente finale e per l'azienda. Nei progetti più tradizionale questa figura è definita dal project manager che sono importanti su progetti su più larga scala. Nei progetti agili ci sarà il Product Owner.

Lo Scrum Master è colui esperto di metodi agile e si trova a supporto del team e del product owner e cerca di aiutare a mettere in pratica lo Scrum e i metodi agili.

Il team di sviluppo è costituito da un numero limitato di professionisti che lavorano per consegnare un incremento finito di prodotto potenzialmente rilasciabile. Scrum non riconosce alcun titolo ai membri del team di sviluppo, il team stesso dà indicazioni e suggerimenti al PO al fine di migliorare il prodotto e renderlo ottimale.

### The Agile: Scrum Framework at a glance



Per la gestione del ciclo di sviluppo del software sono stati utilizzati i linguaggi HTML, CSS e JAVASCRIPT, VisualStudio come editor per il codice sorgente e CodeMirror come componente JavaScript. Inoltre, sono state realizzate tre iterazioni determinando in ognuna di esse gli obiettivi prefissati, le attività svolte, i risultati ottenuti e gli sviluppi futuri:



## 1. Prima iterazione:

Gli obiettivi:

- Comprendere quali tecnologie utilizzare
- L'analisi del funzionamento di IDE online già esistenti
- Studio della documentazione online
- Realizzazione del prototipo dell'editor

Le attività svolte:

- Analisi dei requisiti
- Modellazione dei casi d'uso
- Realizzazione diagramma di sequenza
- Definizione dell'interfaccia utente

I risultati ottenuti:

- Comprensione del funzionamento dell'editor
- Esplorazione della libreria CodeMirror

Gli sviluppi futuri:

- Gestione delle funzionalità dell'editor (Syntax highlighting, Indentazione, Apertura e chiusura automatica delle parentesi, Auto completamento - parole, funzioni, parametri)
- Implementazione del codice

## 2. Seconda iterazione:

Gli obiettivi:

- Syntax highlighting
- Auto-completamento del codice
- Indentazione del codice
- Salvataggio classe di test prodotta
- Caricamento template
- Caricamento classe da testare
- Chiusura automatica delle parentesi
- Console per visualizzare messaggi dal compilatore
- CSS: Realizzazione GUI finale

Le attività svolte:

- Gestione delle funzionalità principali dell'editor (Syntax highlighting, Indentazione, Apertura e chiusura automatica delle parentesi, Auto completamento - parole, funzioni, parametri)
- Implementazione del codice

I risultati ottenuti:

- Gestione delle funzionalità principali dell'editor
- Implementazione del codice

Gli sviluppi futuri:

- Evidenziare righe di codice coperte dal test in verde, rosso (tramite Jacoco)
- Salvataggio delle diverse versioni del codice prodotto
- Code Folding
- Implementazione della dot notation
- Generazione automatica dei metodi *getter*, *setter*, *toString*
- Funzione ricerca parole nel codice

### 3. Terza iterazione:

Gli obiettivi:

- Salvataggio versioni
- dot notation
- Code folding
- Generare getter, setter e toString
- Funzione ricerca parole nei codici
- Salvataggio prima di chiudere o di fare nuovo file
- Possibilità di cambiare tema
- Interazione con gli altri task

Le attività svolte:

- Gestione delle funzionalità principali dell'editor (Salvataggio versioni ogni 30 secondi, dot notation, Code folding, Generare getter, setter, toString, Funzione ricerca parole nel codice, Salvataggio prima di chiudere o di fare nuovo file, Possibilità di cambiare tema)
- Implementazione del codice

I risultati ottenuti:

- Sono state implementate quasi tutte le features richieste

Gli sviluppi futuri:

- Interazione con gli altri task e migliorare la funzione di coverage

### 4. Quarta iterazione:

Gli obiettivi:

- Interazione con gli altri task e migliorare la funzione di coverage
- Documentazione

Le attività svolte:

- Interazione con gli altri task
- Migliorare la funzione di coverage tramite file Jacoco
- Migliorare funzione compilazione
- Realizzazione della documentazione
- Applicazione del framework Spring

I risultati ottenuti:

- L'applicazione funziona
- La documentazione è stata realizzata

## 4. Stima dei costi

### 4.1 Unadjusted Use Case Weight (UUCW)

Elenco dei casi d'uso con complessità stimata associata:

Caso D'uso	Peso	#Transazioni	Complessità
Inserisci Classe di Test	1	1	Semplice
Crea File	1	3	Semplice
Salva Classe	2	2	Semplice
Compile Classe	3	2	Media
Esegui Classe	3	2	Media
Cambia Tema	1	3	Semplice
Ricerca Parola	2	4	Semplice
Genera Getter Setter toString	1	2	Semplice

Tabella UUCW

Complessità	Peso	#Casi d'uso	Prodotto
Semplice	5	6	30
Media	10	2	20
Totale (UUCW)			50

### 4.2 Unadjusted Actor Weight (UAW)

Tipi di attori

Tipo	Esempio	Peso
Complesso	Persona tramite interfaccia grafica	3

### 4.3 Unadjusted Use-Case Points (UUCP)

$$UUCP = UUCW + UAW = 50 + 3 = 53$$

## 4.4 Compressivity Factor

### Technical Compressivity Factor (TCF)

Fattore	Peso	Valutazione	Impatto
Sistema distribuito	2	2	4
Obiettivi di performance	2	5	10
Efficienza end-user	1	4	4
Elaborazione complessa	1	1	1
Codice riusabile	1	3	3
Facile da installare	0.5	5	2.5
Facile da usare	0.5	5	2.5
Portabile	2	3	6
Facile da modificare	1	3	3
Uso concorrente	1	4	4
Sicurezza	1	4	4
Accesso a terze parti	1	3	3
Necessità di addestramento	1	0	0
Totale (TFactor)			47

$$TCF = 0.6 + (0.01 * TFactor) = 0.6 + (0.01 * 47) = 1.07$$

### Environment Complexity Factor (ECF)

Valutazione dell'impatto dei Fattori di Complessità dell'Ambiente (EFactor):

Fattore	Peso	Valutazione	Impatto
Familiare con il processo di sviluppo	1.5	2	3
Esperienza sull'applicazione	0.5	2	1
Esperienza sull'Object-Oriented	1	5	5
Capacità dell'analista	0.5	1	0.5
Motivazione	1	4	4
Requisiti Stabili	2	3	6
Staff part-time	-1	0	0
Linguaggio di programmazione difficile	-1	4	-4
Totale (EFactor)			15.5

$$ECF = 1.4 + (-0.03 * EFactor) = 1.4 + (-0.03 * 15.5) = 0.935$$

## 4.5 Use Case Points (UCP)

$$UCP = UUCP * TCF * ECF = 53 * 1.07 * 0.935 = 53.02385$$

### Calcolo delle Total hours of work (Th)

$$Th = UCP * 3 = 159.07$$

Nota supponendo che un singolo UC venga sviluppato in 2.5 ore complessive mediamente.

### Numero interazioni

Supponendo che ogni membro del team lavori per circa 10 ore a settimana (Wh), per 3 lavoratori otteniamo 30 ore a settimana (TWh). Se Th è il numero di ore di lavoro totali necessarie a terminare il progetto e TWh il numero di ore a settimana del gruppo di lavoro (team), allora il numero di settimane stimate necessarie per terminare il lavoro è dato da:

$Th / TWh = 159.07 / 30 = 5.30$ . Dunque, si stima che occorreranno circa 6 settimane (ovvero 3 iterazioni da 2 settimane ciascuna) per sviluppare questo progetto nella sua interezza.

## 5. Architettura e Progettazione

Dopo aver effettuato la specifica e l'analisi dei requisiti, si è proceduto con la definizione dell'architettura del sistema e la sua progettazione.

In questa fase del progetto è infatti fondamentale definire quella che è la struttura del sistema, indicando tutti gli elementi necessari per poter utilizzare il sistema, le relazioni tra di essi e le loro proprietà. Innanzitutto, sono stati definiti quali sono i principali componenti e come dovrebbe avvenire l'interazione in fase di esecuzione. È stato definito quindi il principale archivio di dati necessario all'applicazione.

Nella definizione dell'architettura si è pensato di applicare dei pattern architetturali, ossia un insieme di decisioni di progettazione architettonica applicabili ad un problema di progettazione ricorrente. Questo permette di tener conto dei diversi contesti di sviluppo software in cui tale problema si presenta. Tra i vari pattern architetturali, si è scelto di applicare il pattern MVC ed il pattern client-server.

Al fine di descrivere al meglio l'architettura sono stati realizzati i seguenti diagrammi: diagramma dei componenti, diagramma di comunicazione, diagrammi di sequenza, diagramma delle classi e dei package, diagramma di contesto.

### 5.1 Pattern Architetturali

I pattern architetturali sono una soluzione generale e consolidata a problemi comuni nell'organizzazione dell'architettura software e nel design dell'architettura software. Essi forniscono un approccio consolidato per affrontare sfide specifiche nel campo dell'architettura software. Questi pattern sono il risultato di esperienze comuni nel settore e rappresentano soluzioni collaudate e riproducibili a problemi ricorrenti. Permettono di coprire una vasta gamma di aree, come la gestione di dati, la comunicazione tra i componenti del sistema, la gestione delle interfacce utente, la sicurezza e molti altri aspetti dell'architettura software.

L'uso dei pattern architetturali può quindi portare a una serie di benefici, tra cui una maggiore modularità, una maggiore flessibilità e una migliore separazione delle responsabilità all'interno di un sistema software. Infine, essi possono facilitare la comprensione e la manutenzione del codice, nonché la collaborazione tra i membri del team di sviluppo.

#### 5.1.1 Stile client-server

Lo stile client-server è un pattern architetturale comune utilizzato nel design dei sistemi software distribuiti nel quale il sistema viene suddiviso in due componenti principali: il client ed il server.

Il client è responsabile di interagire direttamente con l'utente o l'interfaccia utente del sistema, egli invia richieste al server per ottenere informazioni o per richiedere l'esecuzione di determinate operazioni.

Il server, d'altra parte, gestisce ed elabora le richieste provenienti dai client, accede ai dati appropriati e restituisce le risposte corrispondenti. Può anche gestire la persistenza dei dati e garantire la sicurezza delle informazioni.

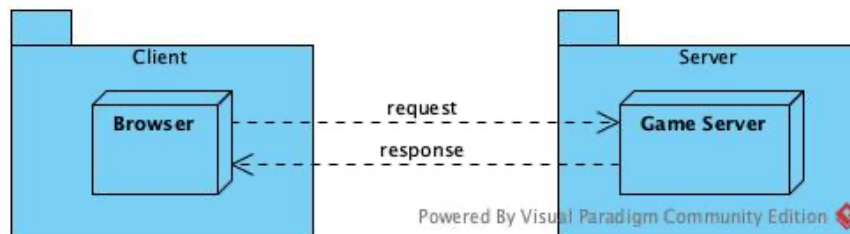
La comunicazione tra client e server avviene generalmente tramite protocolli di rete standard, come HTTP o TCP/IP.

Questo modello di comunicazione tra client e server permette una separazione chiara delle responsabilità tra le due componenti e consente la scalabilità del sistema, in quanto è possibile aggiungere o rimuovere client e server indipendentemente l'uno dall'altro.

Lo stile Client-Server offre una serie di vantaggi. Per esempio, consente una maggiore modularità, in quanto il client e il server possono essere sviluppati e modificati indipendentemente l'uno dall'altro. Inoltre, facilita la gestione delle risorse e la scalabilità, poiché i server possono essere distribuiti su macchine multiple per gestire un carico di lavoro elevato. Infine, favorisce la sicurezza, in quanto il server può applicare controlli e restrizioni sull'accesso alle risorse e sui dati sensibili.

Trattandosi di un'applicazione scritta principalmente in HTML si può affermare che si tratta di un'applicazione client-server. Nel contesto dell'architettura web, il client è il browser che richiede la pagina HTML al server tramite una richiesta HTTP. Il server, a sua volta, elabora la richiesta, recupera il file HTML richiesto e lo invia al client come risposta. Il browser interpreta quindi l'HTML e lo visualizza come una pagina web interattiva.

L'applicazione contiene, inoltre, anche codice JavaScript che aggiunge interattività e funzionalità dinamiche alla pagina HTML. In questo caso, l'applicazione web può essere considerata un sistema client-server più complesso, in cui il client (browser) comunica con il server per eseguire operazioni o richiedere risorse aggiuntive.



### 5.1.2 Pattern MVC

Il pattern architetturale Model-View-Controller (MVC) è ampiamente utilizzato nel design di applicazioni software, specialmente nelle applicazioni basate su interfacce utente. Esso divide l'applicazione in tre componenti principali:

- **Modello (Model):** il modello rappresenta la parte dei dati dell'applicazione e contiene la logica di business. Si occupa della gestione e dell'elaborazione dei dati, nonché delle regole di business e della logica di validazione. Il modello non dipende direttamente dall'interfaccia utente o dalla presentazione dei dati.
- **Vista (View):** la vista è responsabile della presentazione dell'interfaccia utente all'utente finale. Mostra i dati al client e si occupa della loro rappresentazione grafica o testuale. La vista è in genere passiva e non contiene logica di business. È in grado di visualizzare i dati forniti dal modello e di inviare le azioni dell'utente al controller.
- **Controller:** il controller funge da intermediario tra il modello e la vista. Riceve gli input dell'utente dalla vista, li elabora e interagisce con il modello per ottenere i dati richiesti o eseguire le operazioni necessarie. Successivamente, il Controller aggiorna la vista con i nuovi dati ottenuti dal modello.

L'interazione tra i tre componenti nel pattern MVC avviene secondo il seguente flusso:

1. L'utente interagisce con l'interfaccia utente fornita dalla vista.
2. La vista cattura l'azione dell'utente e la inoltra al controller.
3. Il controller elabora l'azione e interagisce con il modello per ottenere i dati necessari.
4. Il modello recupera, aggiorna o elabora i dati richiesti e li restituisce al controller.
5. Il controller aggiorna la vista con i nuovi dati ottenuti dal modello.
6. La vista si aggiorna per riflettere i dati aggiornati e attende nuove azioni dell'utente.

L'uso del pattern MVC porta a una serie di benefici, tra cui:

- Separazione delle responsabilità: Il pattern MVC consente una chiara separazione tra la logica di business, la presentazione dei dati e la gestione delle interazioni dell'utente.
- Riutilizzabilità del codice: I componenti MVC possono essere riutilizzati in diverse parti dell'applicazione, fornendo una maggiore modularità e facilitando la manutenzione.
- Testabilità: La separazione dei componenti facilita l'unit testing dei singoli componenti, consentendo un testing più efficace dell'applicazione nel suo complesso.
- Scalabilità: Il pattern MVC consente l'aggiunta o la modifica di componenti senza dover riscrivere l'intera applicazione, facilitando la scalabilità dell'applicazione.

Per lo sviluppo del progetto è stato applicato il framework **Spring**, che permette di applicare agevolmente il pattern MVC fornendone una struttura predefinita.

In particolare, viene applicato il pattern MVC in un progetto Spring con il framework Spring MVC:

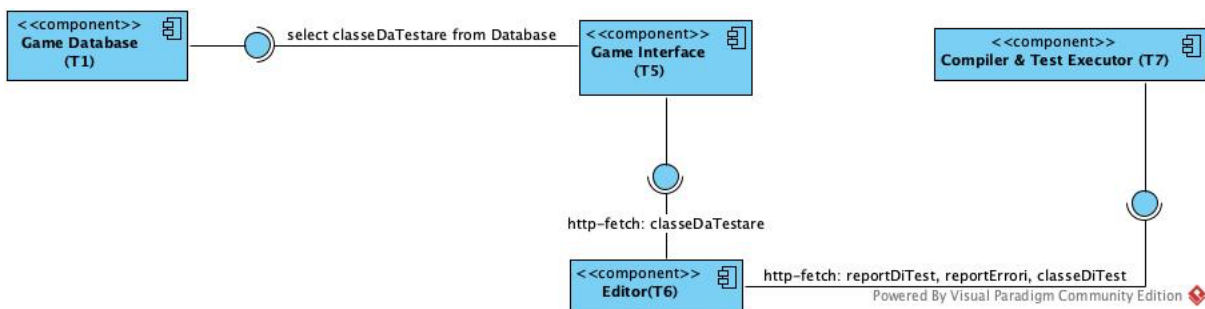
1. Modello (Model): nel contesto di un progetto Spring, il modello rappresenta le classi che rappresentano i dati e la logica di business dell'applicazione. Queste classi sono annotate come bean di Spring (solitamente con l'annotazione '@Component', '@Service' o '@Repository').
  - Le classi relative al Modello sono state implementate dal task T1 con cui è stata fatta l'interazione, tramite il task T5. Per tale ragione queste classi sono contenute nei package relativi ai loro progetti.
2. Vista (View): nella cartella "resources" del progetto Spring, è posizionato il file "code-editor.html" e le cartelle "js" e "css" contenenti rispettivamente i file JavaScript e i file CSS necessari per il funzionamento dell'applicazione. Questo file HTML definisce la struttura, il layout e gli elementi grafici dell'interfaccia utente. La vista viene visualizzata nel browser dell'utente ed è arricchita con CSS e JavaScript per ottenere un aspetto e un comportamento interattivi.
3. Controller: il controller è responsabile di gestire le richieste dell'utente e di coordinare il flusso di dati tra il modello e la vista. Nel contesto di Spring MVC, i controller sono implementati come classi annotate con '@Controller' e i metodi al loro interno vengono annotati con '@RequestMapping'. Questo è implementato tramite la classe "EditorController.java" presente nella cartella "src/main/java". All'interno di questa classe è presente il metodo "getHomePage()" che restituisce la vista per visualizzare l'editor.



## 5.2 Componenti del sistema

### 5.2.1 Diagramma dei componenti

Dopo aver definito il pattern e lo stile architetturale da adottare si è passati a definire l'architettura tramite un diagramma dei componenti. Si tratta di un tipo di diagramma strutturale utilizzato per rappresentare l'architettura del sistema software e le relazioni tra i suoi componenti principali. Questo diagramma evidenzia i componenti del sistema e le dipendenze tra di essi, consentendo di visualizzare la struttura globale dell'applicazione e come i componenti si integrano tra loro.



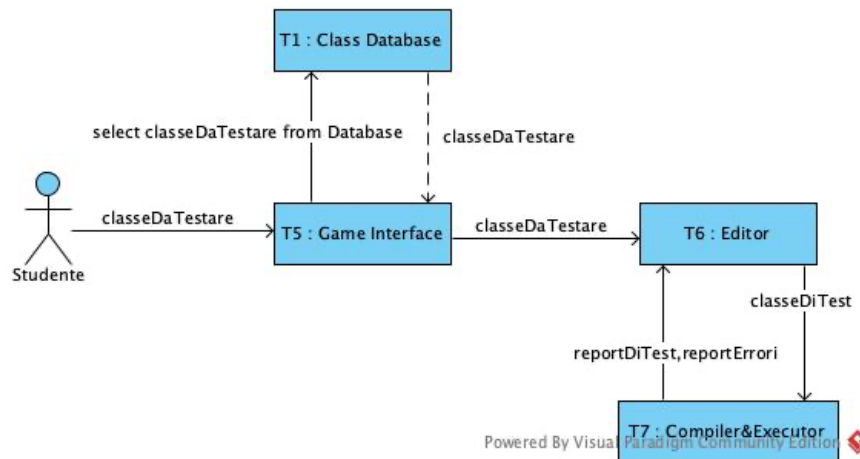
I Componenti sono:

- **Editor (T6):** pagina web dell'editor che permette di visualizzare l'editor per la visualizzazione della classe da testare e l'editor per scrivere la classe di test e di fare le operazioni ad essi correlate;
- **Game Interface (T5):** l'editor interagisce con l'interfaccia realizzata dal task T5 per caricare la classe da testare scelta dall'utente nell'editor appropriato;
- **Game Database (T1):** l'interfaccia del task T5 interagisce con il database realizzato dal task T1 per recuperare la classe da testare;
- **Compiler & Executor (T7):** l'editor interagisce con il task T7 per poter eseguire le operazioni di compilazione ed esecuzione della classe testata e per visualizzare il report del test eseguito. I file di report realizzati dal task T7 sono i seguenti:
  - reportErrori: rappresenta il file txt che contiene l'esito del test ed eventuali errori
  - reportDiTest: rappresenta il file html che contiene la copertura delle righe della classe da testare, ottenuta tramite Jacoco

Lo scambio di informazioni e file necessari deve avvenire mediante richieste http, questo non è stato completamente sviluppato ma è stato soltanto predisposto il codice per poterlo fare.

### 5.2.2 Diagramma di comunicazione

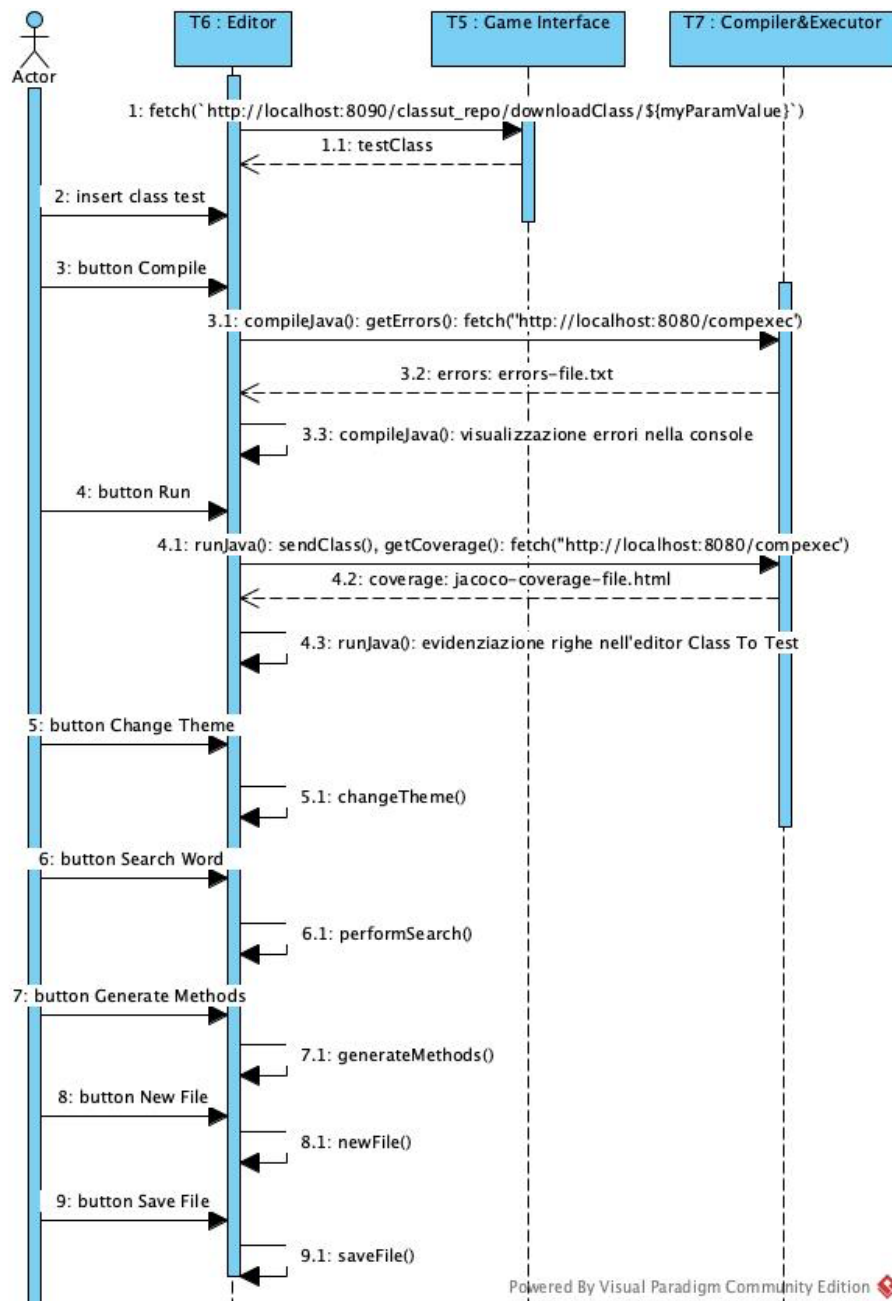
Al fine di comprendere meglio quali sono le relazioni tra i componenti è stato realizzato un diagramma di comunicazione. Si tratta di un tipo di diagramma che viene utilizzato per visualizzare le interazioni tra gli oggetti o le entità all'interno di un sistema, evidenziando come gli oggetti comunicano tra loro per raggiungere uno specifico obiettivo. Esso mostra le connessioni tra gli oggetti attraverso le linee di comunicazione e gli eventi scambiati tra di essi.



Essendo il task T6, e quindi la realizzazione dell'editor, un task intermedio ci si trova nella fase in cui lo studente ha già effettuato il Login e deve tastare la classe che ha scelto (classeDaTestare). Il task T5 ha implementato l'interfaccia che permette allo studente di scegliere la classe, selezionare il robot e quindi avviare la partita. A questo punto la classe da testare scelta dall'utente (classeDaTestare) viene caricata nell'editor appropriato. Una volta che lo studente ha scritto la classe di test (classeDiTest), per poter eseguire le funzioni di compilazione ed esecuzione del codice realizzato, l'editor deve interagire con il task T7 inviandogli il codice prodotto (classeDiTest). Una volta eseguito il codice, il task T7 invia all'editor un report relativo alla copertura della classe da testare (reportDiTest) ed un report relativo ad eventuali errori di sintassi (reportErrori). A questo punto sull'editor che mostra la classe da testare verranno evidenziate le righe di codice coperte (in verde) e quelle non coperte (in rosso) e nella console verranno visualizzati l'esito del test ed eventuali errori di sintassi.

### 5.2.3 Sequence Diagram

È stato quindi realizzato un Sequence Diagram di progettazione, che è un tipo di diagramma di interazione che descrive la sequenza di azioni e messaggi scambiati tra gli oggetti all'interno del sistema software per realizzare una particolare funzionalità. Tramite questo diagramma è possibile visualizzare come gli oggetti dovrebbero interagire tra loro, quali messaggi devono scambiarsi e in quale ordine.



Nel diagramma riportato si mostra come avviene l'interazione tra i vari task, riportando in maniera più dettagliata quali sono le funzioni JavaScript che vengono utilizzate per effettuare l'interazione che verranno descritte più appropriatamente nel capitolo sull'Implementazione.

- *testClass* rappresenta la classe da testare (classeDaTestare) che viene inviata dal task T5 tramite la fetch;
- *javaClass* rappresenta la classe di test (classeDiTest) realizzata dall'utente che l'editor invia al task T7 per ricevere i report;

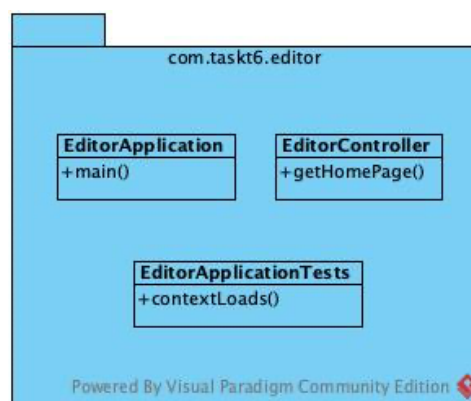
Anche questo diagramma mostra come effettivamente dovrebbe avvenire il collegamento in base alle funzioni JavaScript presenti nel codice, ma l'integrazione definitiva non è stata realizzata. La funzione *runJava()* ha una chiamata al metodo *sendClass()*, che è quello che dovrebbe implementare la fetch per inviare la classeDiTest al T7 ed la funzione *getCoverage()* per la fetch necessaria a ricevere dal T7 il reportDiTest. Allo stesso modo la funzione *compileJava()* effettua una chiamata al metodo *getErrors()*, che a sua volta deve contenere l'implementazione della fetch per recuperare da T7 il file reportDiTest.

## 5.3 Class e Package Diagram

Sono stati quindi realizzati il class diagram per il componente Editor ed il package diagram per l'intero sistema al fine di indicare quali sono le classi necessarie dai vari package per poter realizzare l'interazione con i vari task.

### 5.3.1 Class Diagram

Il Class Diagram è uno dei principali strumenti di modellazione utilizzato per descrivere la struttura statica di un sistema orientato agli oggetti. Esso rappresenta le classi del sistema, le loro relazioni e le proprietà e i metodi associati a ciascuna classe. Questo diagramma fornisce una panoramica chiara e concisa della struttura del sistema e delle relazioni tra le classi.



Il Class Diagram mostra solamente le classi del framework Spring. Il file che contiene l'implementazione della pagina html dell'editor è il file "code-editor.html" localizzato nella cartella "/spring/editor/src/main/resources/static/code-editor.html".

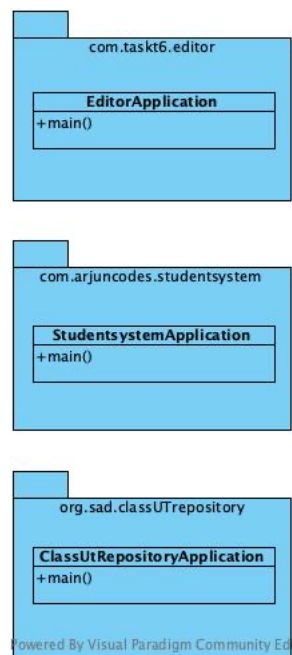
Per rappresentare le funzioni JavaScript presenti si è pensato di utilizzare comunque la notazione uml:



Vengono così mostrate tutte le funzioni JavaScript presenti nel file html, la cui implementazione è descritta nel capitolo successivo.

### 5.3.2 Package Diagram

Il Package Diagram è un tipo di diagramma utilizzato per visualizzare la struttura organizzativa dei package, le loro dipendenze e le relazioni tra di essi. I package sono usati per utilizzare e raggruppare gli elementi di un sistema, come classi, interfacce, file o altri pacchetti.

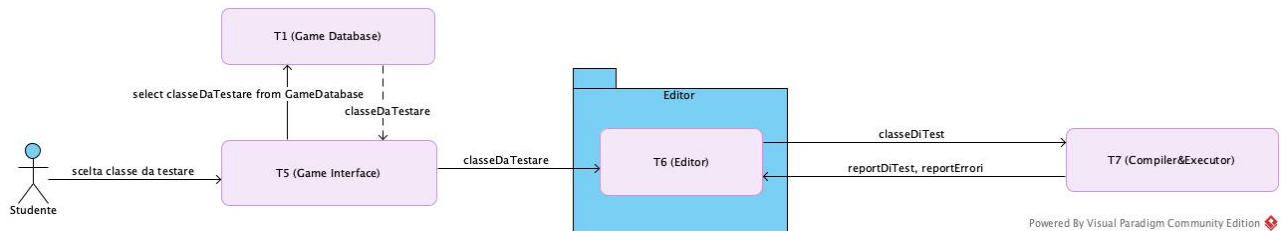


Nel package sono state inserite soltanto le classi Application del framework Spring e i relativi package, esse sono le classi principali per far eseguire l'applicazione e poter effettuare l'interazione con gli altri task.

Il package diagram non contiene il progetto relativo al task T7 in quanto non è stata realizzata l'integrazione vera e propria con questo task, ma sono state realizzate soltanto le funzioni *runJava()* e *compileJava()* che partendo dai file realizzati dal gruppo T7-G23 eseguono rispettivamente l'evidenziazione delle righe e la visualizzazione degli errori e del report di test.

## 5.4 System Context Diagram

Un System Context Diagram, anche noto come diagramma di contesto del sistema, è uno strumento di modellazione utilizzato per visualizzare le relazioni tra un sistema e le entità esterne con cui interagisce. Esso fornisce una panoramica ad alto livello di un sistema, mettendo in evidenza le sue interazioni con gli attori esterni, come utenti, altri sistemi, dispositivi hardware, database, reti e servizi esterni.



Anche in questo diagramma si nota come dovrebbe avvenire l'interazione tra il task T6 ed i task T5 e T7.

## 6. Implementazione

A seguito della fase di progettazione vi è quella di implementazione vera e propria dell'applicazione. In questo capitolo si riporta una descrizione dettagliata del codice realizzato con particolare enfasi sugli aspetti implementativi dei file ritenuti di maggior interesse.

### 6.1 CODE-EDITOR

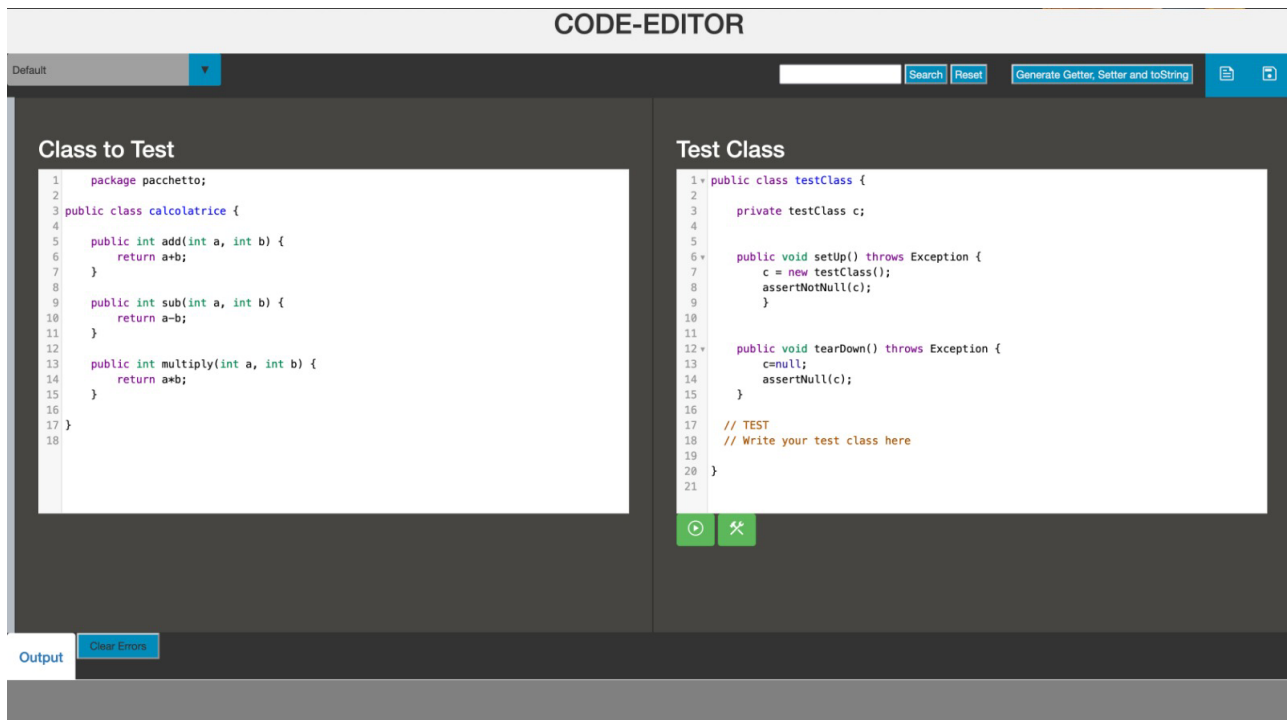


Figura 1-GUI code-editor

Attraverso l'uso del linguaggio HTML combinato con CSS e JavaScript abbiamo realizzato l'interfaccia grafica del nostro editor. Ad esempio, il button per creare un nuovo file è stato associato, tramite l'attributo "onclick", alla funzione JavaScript "showPopup();" che consente di scegliere se salvare il file corrente oppure no: se l'utente dovesse scegliere di salvare la classe di test corrente, verranno richiamate le funzioni "saveFile();", "newFile();" e "hidePopup();" per, rispettivamente, salvare il file, creare un nuovo file nell'editor in cui verrà visualizzato un template, e nascondere il pop-up. Si è proceduto allo stesso modo per tutti i button presenti nell'interfaccia grafica.

Per realizzare entrambe le finestre di editor con le relative funzioni, è stata utilizzata la libreria JavaScript *CodeMirror* che offre una vasta gamma di funzionalità per la modifica e la visualizzazione del codice. Abbiamo implementato la libreria includendo nel nostro file .html i file (.css e .js) relativi alle funzionalità che abbiamo realizzato:

- **<link rel="stylesheet" href="css/codemirror.css">**: Questo link importa un file CSS specifico per lo stile di Codemirror.
- **<link rel="stylesheet" href="css/dialog.css">**: Questo link importa un file CSS per lo stile dei dialoghi nell'editor.
- **<link rel="stylesheet" href="css/foldgutter.css">**: Questo link importa un file CSS per lo stile del gutter di piegatura nell'editor.
- **<link rel="stylesheet" href="css/error-mark.css">**: Questo link importa un file CSS per lo stile dei segni di errore nell'editor.
- **<link rel="stylesheet" href="css/style-editor.css">**: Questo link importa un file CSS per lo stile generale dell'editor.

- **<link rel="stylesheet" href="css/show-hint.css">**: Questo link importa un file CSS per lo stile delle funzionalità di suggerimento nell'editor.
- **<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.1/css/bootstrap.min.css">**: Questo link importa il framework CSS Bootstrap, specificamente la versione 3.4.1, che fornisce stili predefiniti per elementi di interfaccia utente.
- **<link rel="stylesheet" href="css/index.css">**: Questo link importa un file CSS specifico per lo stile dell'indice o della pagina principale del tuo progetto.

I successivi script importano i file JavaScript necessari per il funzionamento di CodeMirror e alcune delle sue funzionalità:

- **<script src="js/codemirror.js"></script>**: Questo script importa il file principale di Codemirror che contiene la logica principale dell'editor.
- **<script src="js/show-hint.js"></script>**: Questo script importa il file JavaScript per le funzionalità di suggerimento nell'editor.
- **<script src="js/java.js"></script>**: Questo script importa il supporto specifico per il linguaggio Java nell'editor.
- **<script src="https://unpkg.com/split.js/dist/split.min.js"></script>**: Questo script importa il file JavaScript per la suddivisione dell'interfaccia utente in pannelli.
- **<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>**: Questo script importa la libreria jQuery, una popolare libreria JavaScript per semplificare la manipolazione degli elementi HTML e l'interazione con il DOM.
- **<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.1/js/bootstrap.min.js"></script>**: Questo script importa il file JavaScript di Bootstrap, che fornisce funzionalità interattive e di stile per elementi di interfaccia utente.
- **<script src="js/closebrackets.js"></script>**: Questo script importa la logica per la chiusura automatica delle parentesi nell'editor.
- **<script src="js/lint.js"></script>**: Questo script importa la logica per il linting o la validazione del codice nell'editor.
- **<script src="js/foldcode.js"></script>, <script src="js/brace-fold.js"></script>, <script src="js/comment-fold.js"></script>, <script src="js/foldgutter.js"></script>**: Questi script importano la logica per la piegatura del codice nell'editor.
- **<script src="js/clike.js"></script>**: Questo script importa il supporto per i linguaggi di programmazione simili a C nell'editor.
- **<script src="js/dialog.js"></script>**: Questo script importa la logica per i dialoghi nell'editor.
- **<script src="js/search.js"></script>, <script src="js/searchcursor.js"></script>**: Questi script importano la logica per la ricerca di testo nell'editor.
- **<script src="js/jump-to-line.js"></script>**: Questo script importa la logica per il salto a una linea specifica nell'editor.
- **<script src="js/match-highlighter.js"></script>**: Questo script importa la logica per evidenziare le corrispondenze di testo nell'editor.
- **<script src="/java-hint.js"></script>**: Questo script importa il supporto per i suggerimenti specifici del linguaggio Java nell'editor.



Inoltre, abbiamo importato anche i file .css per i temi. I file importati sono stati reperiti dalla repository GitHub di CodeMirror.

## 6.2 code-editor.html (senza integrazioni)

I seguenti codici sono relativi al funzionamento del solo nostro editor, senza integrazioni con i task T5 e T6.

### 6.2.1 Finestra editor classe under test

```
//editor per visualizzare classe under test
var testClass = `    package pacchetto;

    public class calcolatrice {

        public int add(int a, int b) {
            return a+b;
        }

        public int sub(int a, int b) {
            return a-b;
        }

        public int multiply(int a, int b) {
            return a*b;
        }
    }
`;

var testEditor = CodeMirror(document.querySelector(".editor .code .class-test"), {
    mode: "text/x-java",
    value: testClass, // Utilizza il valore di default inizialmente
    tabSize: 4,
    lineNumbers: true,
    autoCloseBrackets: true,
    theme: "themeSelect.value",
    readOnly: true,
});
testEditor.setSize("100%", "450px");
```

Figura 2-Creazione finestra editor per classe di test

Il codice JavaScript riportato si riferisce alla creazione della finestra di editor di sinistra per la visualizzazione della classe under test. In particolare, la variabile “testClass” è stata definita come una stringa di testo in cui è stata salvata la classe calcolatrice da mostrare nella finestra appena viene richiamato l’editor. Successivamente, è stata utilizzata la libreria CodeMirror per inizializzare l’editor di codice nella pagina HTML, tramite la variabile “testEditor”. Il linguaggio del codice nell’editor viene impostato su “text/x-java”, il valore iniziale dell’editor viene impostato sulla stringa “testClass” di cui sopra e vengono specificate alcune opzioni come la *tabSize* impostato su 4 spazi, la visualizzazione dei numeri di riga sul lato sinistro della finestra, la chiusura automatica delle parentesi, il tema dell’editor il quale deve essere selezionato dallo studente.

Inoltre, dato che il testo Java di questo editor non può essere modificato, con l'opzione *readOnly*, è stato reso l'editor di sola lettura. L'ultima istruzione riguarda la dimensione della finestra di editor ed imposta la larghezza sul valore 100% (ovvero occuperà tutto lo spazio disponibile in orizzontale) mentre l'altezza su 450px.

### 6.2.2 Finestra editor test class

```
//editor per scrivere classe di test
//template
var javaClass = `public class testClass {
private testClass c;
public void setUp() throws Exception {
    c = new testClass();
    assertNotNull(c);
}
public void tearDown() throws Exception {
    c=null;
    assertNull(c);
}
}
// TEST
// Write your test class here
`;

var javaEditor = CodeMirror(document.querySelector(".editor .code .java-code"), {
    mode: "text/x-java",
    value: javaClass,
    tabSize: 4,
    lineNumbers: true,
    autoCloseBrackets: true,
    theme: "themeSelect.value",
    gutters: ["CodeMirror-linenumbers", "CodeMirror-foldgutter", "CodeMirror-lint-markers"],
    lint: true,
    foldGutter: true,
    foldOptions: {
        rangeFinder: new CodeMirror.fold.combine(CodeMirror.fold.brace, CodeMirror.fold.comment)
    },
    extraKeys: {
        "Ctrl-Q": function (cm) {
            cm.foldCode(cm.getCursor());
        }
    }
});
```

In modo analogo è stata creata la finestra dell'editor di destra per scrivere la test class. Anche in questo caso, con la variabile "JavaClass", viene caricato un template che è possibile modificare aggiungendo il proprio codice Java. La maggior parte delle opzioni combaciano con la finestra di editor per la classe under test, tranne che per alcune come *lint* per abilitare l'evidenziazione delle parole chiave e delle strutture del linguaggio Java nel codice; *foldGutter*, per abilitare il code folding, ovvero nascondere delle porzioni di codice, usato insieme a *foldOptions* il quale ci consente di specificare le azioni per il *foldGutter*, che nel nostro caso viene abilitato quando viene riconosciuto del codice tra parentesi graffe e tra i commenti. Infine, abbiamo l'opzione *extraKeys* che specifica tasti aggiuntivi e le rispettive funzioni associate. In questo caso, viene specificato che premere "Ctrl-Q" nasconderà il codice nella posizione del cursore.

### 6.2.3 enableAutocomplete();

```
// Funzione per abilitare l'autocompletamento Java
function enableJavaAutocomplete() {
    CodeMirror.commands.autocomplete = function (cm) {
        cm.showHint({ hint: CodeMirror.hint.java });
    };
}
```

La funzione “enableJavaAutocomplete()” consente di impostare l'autocompletamento del codice Java come funzionalità di default nell'editor ottenuto tramite CodeMirror. Quando l'utente scrive il codice, verranno visualizzati i suggerimenti di completamento specifici per il linguaggio Java, facilitando la scrittura e la navigazione del codice Java all'interno dell'editor. In particolare, all'interno della funzione “autocomplete” viene assegnata una nuova implementazione alla funzione “cm.showHint()”, propria di CodeMirror, che ha come oggetto l'opzione `{ hint: CodeMirror.hint.java }`. Questo indica che verrà utilizzato l'autocompletamento specifico per il linguaggio Java. “cm”, invece, viene usato comunemente per riferirsi all'oggetto CodeMirror creato per il nostro editor, quindi è un'istanza dell'oggetto CodeMirror che viene utilizzata come riferimento per chiamare i metodi e le proprietà specifiche dell'editor.

#### 6.2.4 runJava();

```
async function runJava() {
  //Run test
  // Leggi il contenuto del file HTML di report Jacoco
  fetch('calcolatrice.java.html')
    .then(response => response.text())
    .then(html => {
      // Parsing del contenuto HTML
      const parser = new DOMParser();
      const doc = parser.parseFromString(html, 'text/html');

      // Trova l'elemento <pre> che contiene il codice sorgente Java
      const preElement = doc.querySelector('pre.source.lang-java.linenums');

      // Recupera tutte le righe di codice
      const codeLines = preElement.querySelectorAll('span');

      // Inizializza due array per le righe coperte e non coperte
      const coveredLines = [];
      const notCoveredLines = [];

      // Itera sulle righe di codice e identifica quelle coperte e non coperte
      codeLines.forEach(line => {
        const lineId = line.getAttribute('id');
        const lineNumber = parseInt(lineId.substring(1));

        if (line.classList.contains('fc')) {
          coveredLines.push(lineNumber);
        } else if (line.classList.contains('nc')) {
          notCoveredLines.push(lineNumber);
        }
      });

      // Evidenzia le righe coperte nell'editor CodeMirror
      coveredLines.forEach(lineNumber => {
        testEditor.addLineClass(lineNumber - 1, 'background', 'covered-line');
      });

      // Evidenzia le righe non coperte nell'editor CodeMirror
      notCoveredLines.forEach(lineNumber => {
        testEditor.addLineClass(lineNumber - 1, 'background', 'not-covered-line');
      });
    })
    .catch(error => {
      console.error('Si è verificato un errore durante il recupero del file HTML:', error);
    });
}
```

Questa funzione è stata implementata per eseguire la test class e evidenziare le righe della classe under test che sono state coperte dal test, quindi deve eseguire una serie di istruzioni per leggere ed analizzare il file HTML contenente il report di copertura del codice generato da Jacoco, che ci è stato inviato dal task T7 e che noi, in questo caso senza integrazioni, abbiamo nella cartella `spring\editor\src\main\resources\static`.

In particolare, la funzione `fetch('calcolatrice.java.html')` effettua una richiesta HTTP per ottenere il contenuto del file HTML denominato `calcolatrice.java.html`. Se la risposta alla richiesta si risolve correttamente allora con l'istruzione `.then(response => response.text())` viene estratto il testo del file HTML. Per estrarre le informazioni dal file Jacoco di nostro interesse sono state eseguite le seguenti azioni:

- è stato effettuato il *parsing del contenuto HTML*, creando un'istanza dell'oggetto `DOMParser` che consente di analizzare il testo HTML (la *const parser*) e su di esso è stato analizzato il testo HTML e convertito (la *const doc*);
- è stato utilizzato il metodo `querySelector()` dell'oggetto `doc` per trovare l'elemento `<pre>` che ha le classi `"source"`, `"lang-java"` e `"linenums"`. Questo elemento contiene il codice sorgente Java;
- è stato utilizzato il metodo `querySelectorAll()` chiamato sulla *const codeLines* per selezionare tutti gli elementi `<span>` all'interno dell'elemento `<pre>`. Questi elementi rappresentano le singole righe di codice;
- è stato utilizzato il metodo `forEach()` chiamato sulla *const codeLines* (che ora contiene tutte le righe di codice) per iterare su tutte le righe di codice selezionate nell'elemento `<pre>`. Per ogni riga, viene estratto l'ID dell'elemento e il numero di riga associato. Se la riga contiene la classe CSS `"fc"` (che indica che è coperta) o la classe `"nc"` (che indica che non è coperta), viene aggiunta la corrispondente riga nell'array `coveredLines` o `notCoveredLines` rispettivamente.

Dopo aver identificato le righe coperte e non coperte del codice sorgente Java nel file HTML di report di copertura, devono essere evidenziate le linee di codice coperte nella relativa classe under test, quindi con il metodo `forEach()` si itera l'array `coveredLines()` e su ogni riga viene chiamato il metodo `addLineClass()` sull'oggetto `testEditor` che aggiunge una classe CSS chiamata `"covered-line"` (definita nel file `style-editor.css` incluso nell'head del codice). La riga viene identificata utilizzando il numero di riga diminuito di 1, poiché gli indici delle righe in `CodeMirror` partono da 0.

In modo analogo si procede per le *notCoveredLines*.

```
1  package pacchetto;  
2  
3  public class calcolatrice {  
4  
5      public int add(int a, int b) {  
6          return a+b;  
7      }  
8  
9      public int sub(int a, int b) {  
10         return a-b;  
11     }  
12  
13     public int multiply(int a, int b) {  
14         return a*b;  
15     }  
16  
17 }  
18
```

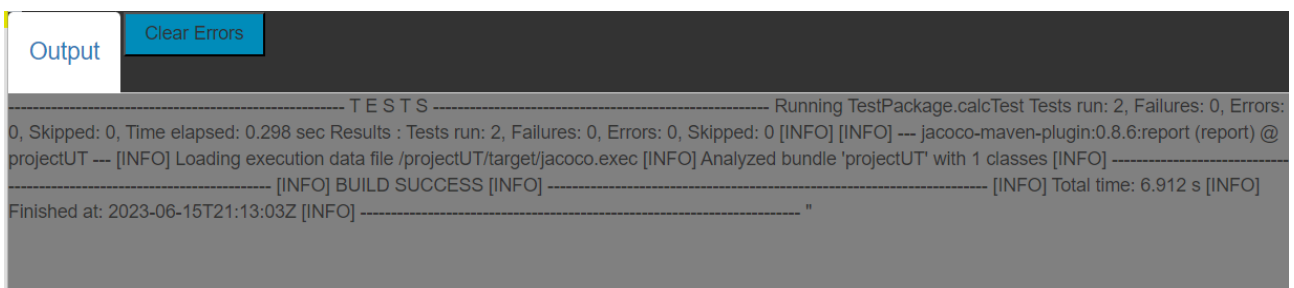
### 6.2.5 compileJava();

```
function compileJava() {
  // Recupera l'elemento della console di errori
  const errorConsole = document.getElementById('error-console');

  fetch('ex_res.txt')
    .then(response => response.text())
    .then(text => {
      const errorElement = document.createElement('div');
      errorElement.textContent = text;
      errorConsole.appendChild(errorElement);
    })
    .catch(error => {
      console.error('Si è verificato un errore durante il recupero del file TXT:', error);
    });
}

function clearErrors() {
  const errorConsole = document.getElementById('error-console');
  errorConsole.innerHTML = '';
}
```

La funzione esegue la compilazione della classe under test e mostra nella console di output eventuali errori. Anche in questo caso, il file contenente il risultato della compilazione ci è stato inviato dal task T7 e lo troviamo nella cartella `spring\editor\src\main\resources\static`. Analogamente alla funzione “`runJava()`”, andiamo a leggere il contenuto del file “`ex_res.txt`”. In particolare, viene creato un nuovo elemento `<div>` per visualizzare il contenuto del file che viene passato con `text` all’interno del `<div>`, mentre il `<div>` viene aggiunto all’elemento `errorConsole`, che rappresenta la console degli errori nel DOM.



The screenshot shows the IDE's Output console. At the top, there is a tab labeled "Output" and a button labeled "Clear Errors". The console displays the following text:

```
----- T E S T S ----- Running TestPackage.calcTest Tests run: 2, Failures: 0, Errors:
0, Skipped: 0, Time elapsed: 0.298 sec Results : Tests run: 2, Failures: 0, Errors: 0, Skipped: 0 [INFO] [INFO] --- jacoco-maven-plugin:0.8.6:report (report) @
projectUT --- [INFO] Loading execution data file /projectUT/target/jacoco.exec [INFO] Analyzed bundle 'projectUT' with 1 classes [INFO] -----
[INFO] BUILD SUCCESS [INFO] ----- [INFO] Total time: 6.912 s [INFO]
Finished at: 2023-06-15T21:13:03Z [INFO] ----- "
```

### 6.2.6 saveFile();

```
//Salvataggio classe java
function saveFile() {
    var textToWrite = javaEditor.getValue();
    var textFileAsBlob = new Blob([textToWrite], {
        type: "text/plain;charset=utf-8"
    });
    var currentdate = new Date();
    var datetime = "testClass" + currentdate.getDate() + "_"
        + (currentdate.getMonth() + 1) + "_"
        + currentdate.getFullYear() + "@"
        + currentdate.getHours() + "_"
        + currentdate.getMinutes() + "_"
        + currentdate.getSeconds();

    var fileNameToSaveAs = datetime + ".java";

    var downloadLink = document.createElement("a");
    downloadLink.download = fileNameToSaveAs;
    downloadLink.innerHTML = "Download File";
    if (window.webkitURL != null) {
        // Chrome allows the link to be clicked
        // without actually adding it to the DOM.
        downloadLink.href = window.webkitURL.createObjectURL(textFileAsBlob);
    } else {
        // Firefox requires the link to be added to the DOM
        // before it can be clicked.
        downloadLink.href = window.URL.createObjectURL(textFileAsBlob);
        downloadLink.onclick = destroyClickedElement;
        downloadLink.style.display = "none";
        document.body.appendChild(downloadLink);
    }

    downloadLink.click();
}
```

La funzione "saveFile()" viene richiamata quando si preme il button "Save as Java Class" e quando si preme il button "New Test Class" perché in questo caso verrà mostrato un pop-up in cui verrà chiesto allo studente di scegliere se salvare la test class corrente o meno. In particolare, nella variabile **textToWrite** viene recuperato il valore corrente dell'editor della test class e viene creato l'oggetto Blob per rappresentare il contenuto del file di testo da salvare. Il nome del file da salvare viene generato a partire dalla stringa **datetime** concatenata con ".java" nel formato "testClassDD\_MM\_YYYY@HH\_MM\_SS.java". Nella variabile **downloadLink** è stato creato l'elemento "a" che fungerà da collegamento per il download del file e viene impostato l'attributo "download" dell'elemento "a" con il nome del file da salvare. Questo attributo specifica il nome del file quando viene scaricato. Chrome non richiede che l'elemento sia nel DOM per poterlo cliccare, quindi viene creato un URL (che rappresenta il contenuto del file di testo da scaricare) passando l'oggetto "textFileAsBlob".

Nel caso in cui avessimo un altro browser, ad esempio Firefox che richiede che il collegamento per il download sia effettivamente nel DOM per poterlo cliccare, verrà sempre creato l'URL con oggetto "textFileAsBlob" ma questo verrà aggiunto al DOM come elemento nascosto (display: none) e verrà distrutto dopo che è stato cliccato. L'ultima istruzione `downloadLink.click()` genera un evento di click sul collegamento per avviare il download del file.

### 6.2.7 Autosave

```
//Autosalvataggio (ogni 30s)
var timer;
javaEditor.on("change", function () {
    // Debounce the save function to avoid frequent saving
    clearTimeout(timer);
    timer = setTimeout(saveFile, 30000);
});
```

Questa funzione consente di salvare il codice prodotto ogni 30 secondi. Viene impostato un gestore di eventi sull'evento "change" dell'editor Java. Questo significa che la funzione specificata verrà eseguita ogni volta che il contenuto dell'editor viene modificato dall'utente. "clearTimeout(timer)" cancella il timer precedente per evitare che il salvataggio venga eseguito troppo frequentemente. Questo previene l'autosalvataggio se l'utente sta continuamente modificando il codice. Infine viene settato un nuovo timer che eseguirà la funzione "saveFile()" di cui sopra, ogni 30000 millisecondi.

### 6.2.8 performSearch()

```
function performSearch() {
    var searchTerm = searchInput.value;
    javaEditor.focus();
    testEditor.focus();

    if (searchOverlay) {
        javaEditor.removeOverlay(searchOverlay);
        testEditor.removeOverlay(searchOverlay);
        searchOverlay = null;
    }

    if (searchTerm) {
        searchOverlay = {
            token: function (stream) {
                if (stream.match(searchTerm)) return "searching";
                while (stream.next() && !stream.match(searchTerm, false)) { }
            }
        };

        javaEditor.addOverlay(searchOverlay, { opaque: true });
        testEditor.addOverlay(searchOverlay, { opaque: true });
    }
}
```



La funzione consente di implementare la funzionalità di ricerca di parole in entrambi gli editor tramite l'apposito campo di input. In particolare, il primo if si occupa di verificare se già esiste un overlay di ricerca (meccanismo fornito da CodeMirror) negli editor, ovvero se è stato già evidenziato un token (parola). Se esiste, viene rimosso per prepararsi alla nuova ricerca. Il secondo if controlla se è stato inserito un termine di ricerca. Se è presente, si crea l'oggetto `searchOverlay` per evidenziare i risultati della ricerca e su di esso viene definita la funzione "token" che viene richiamata per ogni parola all'interno dell'editor; se il token corrisponde ai termini di ricerca, viene restituita la classe "searching" per evidenziarlo. Il metodo "addOverlay()" richiamato su entrambi gli editor serve per garantire che l'overlay copra il testo.

### 6.2.9 resetSearch();

```
function resetSearch() {
  javaEditor.focus();
  testEditor.focus();
  javaEditor.removeOverlay("search");
  testEditor.removeOverlay("search");
  searchInput.value = "";
  if (searchOverlay) {
    javaEditor.removeOverlay(searchOverlay);
    testEditor.removeOverlay(searchOverlay);
    searchOverlay = null;
  }
}
```

La funzione viene utilizzata per reimpostare la ricerca negli editor ed è collegata al button "Reset". In particolare, vengono rimossi entrambi gli overlay di ricerca associati al token e il valore dell'elemento di input della ricerca viene reimpostato ad una stringa vuota, cancellando il termine di ricerca precedentemente inserito. L'if, invece, si riferisce all'evidenziazione, quindi analogamente, se è presente l'overlay, viene rimosso.

### 6.2.10 generateMethods();

```
for (var i = 0; i < lines.length; i++) {
  var line = lines[i].trim();
  if (line !== "") {
    var parts = line.split(" ");
    if (parts.length === 2) {
      var variableName = parts[1].replace(";", "");
      var capitalizedVariableName = variableName.charAt(0).toUpperCase() + variableName.slice(1);

      var getterCode = "public get" + capitalizedVariableName + "() {\n  return " + variableName + ";\n}";
      var setterCode = "public void set" + capitalizedVariableName + "(" + parts[0] + " " + variableName + ") {\n  this." + variableName + " = " + variableName + ";\n}";

      generatedCode += getterCode + "\n\n" + setterCode + "\n\n";
    }
  }
}

generatedCode += "public String toString() {\n";
generatedCode += "  return \"here is the string to print\";\n";
generatedCode += "};";

var currentCode = javaEditor.getValue();
javaEditor.setValue(currentCode + "\n\n" + generatedCode);
```

```

generatedCode += "public String toString() {\n";
generatedCode += "    return \"here is the string to print\";\n";
generatedCode += "}";

var currentCode = javaEditor.getValue();
javaEditor.setValue(currentCode + "\n\n" + generatedCode);
}

```

Questa funzione è collegata al button “generateButton” che, quando viene premuto, genera nella finestra di editor della test class i metodi Getter e Setter da completare. In particolare, viene salvato il contenuto dell’editor e viene diviso in righe così da poterlo iterare; se la riga non è vuota, viene suddivisa in parti in base agli spazi bianchi che vengono salvate in parts; se la riga è composta da due parti, si presuppone sia composta dalla dichiarazione di variabile e dal suo nome e vengono creati i codici per il getter e setter utilizzando le parti estratte dalla riga corrente `generatedCode += getterCode + "\n\n" + setterCode + "\n\n"`; concatenando il codice del getter, due linee vuote e il codice del setter. Dopo aver ripetuto il procedimento per tutte le righe, viene aggiunto il metodo `toString()` al codice generato. Infine viene aggiunto il codice generato al codice corrente dell’editor.

### 6.2.11 EditorApplication.java

```

package com.taskt6.editor;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class EditorApplication {

    Run | Debug
    public static void main(String[] args) {
        SpringApplication.run(EditorApplication.class, args);
    }

}

```

Dato che è stato utilizzato il framework Spring, sono state importate le classi necessarie da Spring Boot per avviare l’applicazione. L’annotazione `@SpringBootApplication` indica che questa classe è l’entry point principale dell’applicazione; in particolare, il metodo `main` viene eseguito quando viene avviata l’applicazione ed è responsabile di avviare il contesto di Spring Boot chiamando il metodo `run` di `SpringApplication`. Questo avvia l’applicazione Spring Boot, inizializza i componenti necessari e avvia il server web incorporato che consente all’applicazione di essere eseguita come una web app.

### 6.2.12 EditorController.java

```
package com.task6.editor;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class EditorController {

    @GetMapping("/")
    public String getHomePage() {
        return "code-editor.html"; // Nome del file HTML da restituire
    }
}
```

La classe EditorController è stata annotata con @Controller, che indica che è un componente di Spring responsabile di gestire le richieste HTTP e restituire una risposta. @GetMapping, invece, indica che il metodo “getHomePage();” gestirà le richieste HTTP GET alla homepage (/). Quando viene chiamato questo metodo, restituisce una stringa che rappresenta il nome del file HTML code-editor.html. questo significa che quando un utente accede all’URL dell’applicazione, il controller restituirà il file code-editor.html come risposta.

## 6.3 code-editor.html (integrazione con il task 5)

I codici che seguiranno sono relativi alla preparazione per l'integrazione del task 5, in particolare per il caricamento della classe under test che l'utente sceglie. Innanzitutto, con le istruzioni "const urlParams = new URLSearchParams(window.location.search);" e "const myParamValue = urlParams.get('myParam');" estraiano il valore del parametro specifico dalla stringa di query dell'URL che sarà il parametro *myParam*.

### 6.3.1 Finestra editor classe under test

```
//editor per visualizzare classe under test
var testClass = ""; // Valore di default

var testEditor = CodeMirror(document.querySelector(".editor .code .class-test"), {
  mode: "text/x-java",
  value: testClass, // Utilizza il valore di default inizialmente
  tabSize: 4,
  lineNumbers: true,
  autoCloseBrackets: true,
  theme: "themeSelect.value",
  readOnly: true,
});

fetch(`http://localhost:8090/classut_repo/downloadClass/${myParamValue}`)
  .then(response => response.text())
  .then(data => {
    testClass = data; // Aggiorna il valore di testClass con i dati della risposta fetch
    testEditor.setValue(testClass); // Aggiorna il valore dell'editor con i nuovi dati
  })
  .catch(error => console.error(error));

CodeMirror.commands["selectAll"](testEditor);
testEditor.setSize("100%", "450px");
```

L'editor della classe under test viene creato allo stesso modo con la differenza che il valore di default *testClass* è inizialmente vuoto ma verrà sostituito nel momento in cui lo studente sceglierà la classe negli altri task. In particolare, con il metodo "fetch(http://localhost:8090/classut\_repo/downloadClass/\${myParamValue})" viene fatta richiesta all'endpoint specificato per scaricare il contenuto del file di classe specificato dal parametro "myParamValue". In questo modo vengono aggiornati i valori della variabile *testClass* e dell'editor in cui verrà visualizzato la classe selezionata dallo studente nelle pagine precedenti dell'applicazione.

## 6.4 code-editor.html (integrazione con il task 7)

Nel caso di integrazione con il task 7, dobbiamo recuperare i dati relativi alla coverage e agli errori di compilazione.

### 6.4.1 getCoverage();

```
function getCoverage() {
  fetch('http://localhost:8080/compexec?urlClass=test&urlTestClass=test2')
    .then(response => response.json())
    .then(data => {
      // Accedi al valore specifico desiderato
      return pathCoverage = data.pathCoverage;
    })
    .catch(error => {
      console.error('Errore durante la lettura del file JSON:', error);
    });
}
```

La funzione consente di recuperare l'URL del file di copertura dal file JSON con campo "pathCoverage", essa verrà richiamata dalla funzione "runJava()" per leggere i dati relativi alla coverage.

### 6.4.2 getErrors();

```
function getErrors() {
  fetch('http://localhost:8080/compexec?urlClass=test&urlTestClass=test2')
    .then(response => response.json())
    .then(data => {
      // Accedi al valore specifico desiderato
      return resMessage = data.resMessage;
    })
    .catch(error => {
      console.error('Errore durante la lettura del file JSON:', error);
    });
}
```

La funzione opera allo stesso modo della funzione "getCoverage();" però si riferisce al campo "resMessage" contenente il file con gli errori di compilazione.

### 6.4.3 runJava();

```
async function runJava() {  
  sendClass();  
  var url = getCoverage();  
  //Run test  
  // Leggi il contenuto del file HTML di report Jacoco  
  fetch(url)  
    .then(response => response.text())  
    .then(html => {
```

La funzione è uguale alla funzione “runJava();” specificata nel paragrafo “code-editor.html (senza integrazioni)”: è stata aggiunta la funzione “sendClass();” per inviare la classe under test al task 7 per ricevere il file della coverage Jacoco.xml e degli errori di compilazione relativo alla classe.

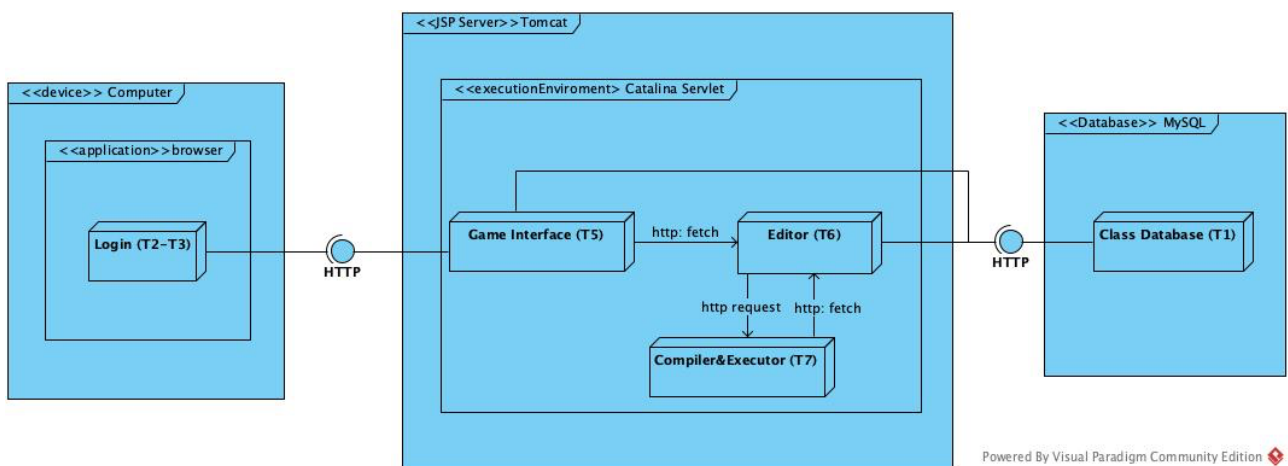
## 6.3 Deployment diagram

Un diagramma di deployment è un tipo di diagramma UML (Unified Modeling Language) che descrive come un sistema software viene distribuito su una serie di nodi hardware o di infrastruttura. Questo diagramma mostra la configurazione fisica del sistema e illustra la disposizione dei componenti software all'interno dell'ambiente di esecuzione.

Il diagramma di deployment definisce i nodi come rappresentazioni astratte delle risorse fisiche o virtuali su cui il sistema è eseguito. Questi nodi possono essere server fisici, server virtuali, computer desktop, dispositivi mobili, dispositivi di rete, etc. Ogni nodo può contenere uno o più componenti software.

I componenti software possono includere applicazioni, database, librerie, file di configurazione e altri elementi che costituiscono il sistema. Le relazioni tra i nodi e i componenti software sono mostrate mediante frecce o linee tratteggiate che indicano le connessioni di rete o i canali di comunicazione.

Il diagramma di deployment può inoltre mostrare dettagli aggiuntivi, come le specifiche dei protocolli di comunicazione utilizzati, i vincoli di risorse hardware (ad esempio, capacità di storage o di elaborazione richiesta da un nodo), la configurazione di rete, la topologia di rete e altre informazioni pertinenti al sistema in fase di deployment.



## 7. Testing

Non essendo stata completamente realizzata l'integrazione tra i vari task non è stato eseguito il test di integrazione, ma solo quello di interfaccia.

### 7.1 Test di interfaccia

Il test di interfaccia, noto anche come test di UI (User Interface) o test di front-end, è una pratica di testing software che si concentra sulla validazione e verifica del corretto funzionamento dell'interfaccia utente di un'applicazione o di un sistema. Questo tipo di test è principalmente finalizzato a garantire che l'interfaccia utente sia intuitiva, facile da usare, esteticamente gradevole e che risponda correttamente alle interazioni dell'utente.

Il test di interfaccia coinvolge l'esecuzione di una serie di azioni sull'interfaccia utente, come clic sui pulsanti, inserimento di dati in campi di input, selezione di opzioni da menu a discesa, scorrimento di liste, navigazione tra pagine e così via. L'obiettivo è verificare se l'interfaccia risponde correttamente a queste azioni e se i risultati corrispondono alle aspettative dell'utente.

Durante il test di interfaccia, vengono solitamente verificati diversi aspetti, tra cui:

1. **Funzionalità:** verifica che tutte le funzionalità dell'interfaccia utente siano accessibili e si comportino come previsto. Ciò include la validazione di pulsanti, link, menu, formulari, ecc.
2. **Navigazione:** verifica che la navigazione tra le diverse pagine o sezioni dell'applicazione funzioni correttamente, inclusi i collegamenti, i menu di navigazione e i controlli di paginazione.
3. **Layout e design:** verifica che l'aspetto grafico dell'interfaccia sia coerente, esteticamente gradevole e che si adatti correttamente a diversi dispositivi e risoluzioni.
4. **Responsività:** verifica che l'interfaccia si adatti e funzioni correttamente su diversi dispositivi e dimensioni dello schermo, come desktop, tablet e dispositivi mobili.
5. **Feedback e messaggi di errore:** verifica che vengano visualizzati feedback appropriati all'utente, come messaggi di conferma, avvisi o messaggi di errore, quando necessario.
6. **Usabilità e accessibilità:** verifica che l'interfaccia sia facile da usare, intuitiva e che rispetti le linee guida di accessibilità per garantire l'accesso a utenti con disabilità.

Per eseguire il test di interfaccia, possono essere utilizzati diversi approcci, come test manuali in cui un tester umano esegue manualmente le azioni sull'interfaccia utente, o test automatizzati in cui vengono utilizzati strumenti e framework specifici per eseguire gli script di test. L'automazione del test di interfaccia può essere vantaggiosa per garantire una copertura più ampia e una maggiore efficienza nei test ripetitivi.

Test Case ID	Titolo test	Scopo	Descrizione	Passi del test	Dati di input	Aspettative	Risultati	Conclusioni
1	Compila Codice	Valutare la funzionalità del bottone	Il bottone "Compile" permette di visualizzare gli errori	Il bottone viene premuto	-	Visualizzazione risultati	I risultati sono visualizzati	PASS
2	Esegui Codice	Valutare la funzionalità del bottone	Il bottone "Run" permette di visualizzare gli errori	Il bottone viene premuto	-	Visualizzazione copertura	Le righe coperte dal test sono evidenziate di verde, quelle non coperte di rosso	PASS
3	Cambia Tema	Valutare la funzionalità del selettore tema	Il selettore del tema permette di cambiare il tema	Il bottone viene premuto	Tema selezionato	Applicazione nuovo tema	Il tema viene cambiato	PASS
4	Ricerca parola	Valutare la funzionalità del bottone "Search" e del bottone "Reset"	Il bottone "Search" permette di evidenziare la parola richiesta ed il bottone "Reset" permette di inizializzare la barra di ricerca	Viene inserita la parola da cercare, viene premuto il bottone "Search" e infine viene premuto il	Parola da ricercare	Parola ricercata evidenziata negli editor	Premendo su "Search" la parola cercata viene evidenziata, poi premendo su "Reset" la barra di ricerca viene inizializzata	PASS



				bottoni "Reset"				
5	Generare Metodi Getter, Setter, toString	Generazione dei metodi Getter, Setter, toString	Il bottone "Generate Getter, Setter, toString" permette di generare i metodi nell'editor Class To Test	Il bottone "Generate Getter, Setter, toString" viene premutato	-	Generazione dei metodi Getter, Setter, toString	I metodi Getter, Setter, toString sono generati all'interno dell'editor Class To Test	PASS
6	Nuovo File	Creazione di un nuovo file	Il bottone "New File" permette di creare un nuovo file	Il bottone "New File" viene premutato	-	Generazione di un nuovo file nell'editor Class To Test e richiesta di salvataggio	Il nuovo file è generato nell'editor Class To Test e viene visualizzato il popup per la scelta sul salvataggio	PASS
7	Salva File	Salvataggio della classe presente nell'editor Class To Test	Il bottone "Save File" permette di salvare il file in locale	Il bottone "Save File" viene premutato	-	Salvataggio della classe presente nell'editor Class To Test	Il file viene salvato correttamente	PASS