



**UNIVERSITÀ<sup>DEGLI STUDI DI</sup>  
NAPOLI FEDERICO II**

Scuola Politecnica e delle Scienze di Base  
Corso di Laurea Magistrale in Ingegneria Informatica

Elaborato progettuale in Software Architecture Design

## ***Task 07 - Gruppo 23***

Anno Accademico 2022/2023

Professoressa  
**Anna Rita Fasolino**

Studenti  
**Antonio Russo**  
**Francescopaolo Lecce**  
**Michele Savella**

# Indice

<b>1</b>	<b>Introduzione</b>	<b>1</b>
1.1	Descrizione del Task . . . . .	1
1.2	Processo di sviluppo . . . . .	1
1.3	Effort dedicato . . . . .	2
1.4	Tecnologie Utilizzate . . . . .	3
1.4.1	Docker . . . . .	3
1.4.2	Maven . . . . .	3
1.4.3	SpringBoot . . . . .	3
1.4.4	Jacoco . . . . .	4
1.5	Interazioni con altri Task . . . . .	4
<b>2</b>	<b>Documenti di analisi</b>	<b>5</b>
2.1	Analisi e specifica dei requisiti . . . . .	5
2.1.1	Requisiti . . . . .	5
2.1.2	Storie Utente . . . . .	6
2.1.3	Glossario dei termini . . . . .	7
2.1.4	Diagrammi di analisi . . . . .	7
2.1.4.1	Diagramma dei casi d'uso . . . . .	7
2.1.4.2	Diagramma delle classi . . . . .	8
2.1.4.3	Diagramma di sequenza . . . . .	9
<b>3</b>	<b>Fase di progettazione</b>	<b>11</b>
3.1	Scelta architetturale . . . . .	11
3.2	Artefatti di progettazione . . . . .	12
3.2.1	Diagramma di attività . . . . .	12

---

## INDICE

3.2.2	Diagramma di sequenza . . . . .	14
3.2.3	Diagramma dei componenti . . . . .	14
3.2.4	Diagramma delle classi . . . . .	17
3.2.5	Diagramma dei package . . . . .	19
3.2.6	Diagramma di deployment . . . . .	20
<b>4</b>	<b>Implementazione e testing</b>	<b>22</b>
4.1	Documentazione dell'implementazione . . . . .	22
4.2	Documentazione dell'interfaccia . . . . .	25
4.3	Testing . . . . .	28
4.4	Stima dei Costi . . . . .	31
<b>5</b>	<b>Guida all'installazione</b>	<b>33</b>

# Capitolo 1

## Introduzione

### 1.1 Descrizione del Task

L'applicazione deve offrire la funzionalità di compilazione e esecuzione dei casi di test scritti dal giocatore. Tale funzionalità riceverà in input due file di testo (classe da testare e classe di test) e lancerà il compilatore e l'esecutore dei test, restituendo in output le informazioni relative all'esito della compilazione (se errata) oppure l'esito dei test eseguiti. L'esito dell'esecuzione dovrà essere elaborato in maniera da estrarre da essi le informazioni rilevanti ai fini del gioco (ad esempio la presenza di errori di compilazione del test, oppure l'esito dell'esecuzione del test, come ad esempio la copertura del codice, etc.).

### 1.2 Processo di sviluppo

Nella realizzazione del progetto è stato utilizzato, in tutto il ciclo di sviluppo del software, un'approccio di tipo Agile (in contrapposizione a un approccio plan based) ed in particolare è stato utilizzato il framework Scrum durante il corso.

Ciò ci ha consentito di sviluppare in modo iterativo ed incrementale, comprendendo sempre meglio le responsabilità del nostro task e le tecnologie da utilizzare in fase di implementazione. Ad ogni Sprint (iterazione), ogni 14 giorni, sono stati sviluppati prototipi ed artefatti presentati durante ogni Sprint Reviews. Confrontandoci con altri task e grazie ai feedback dei professori siamo riusciti a portare a termine il task assegnatoci.

### 1.3 Effort dedicato

Giorno	Data	Durata	Attività
1	10 aprile	2 ore	Scrittura di storie utente, definizione dei requisiti funzionali e identificazione dei dubbi da chiarire con gli stakeholders
2	15 aprile	1 ora	Impostazione del diagramma dei casi d'uso e dei diagrammi di sequenza
3	16 aprile	2 ore	Studio delle tecnologie di compilazione ed esecuzione e creazione del mockup dell'interfaccia grafica
4	17 aprile	1 ora	Perfezionamento dei diagrammi di sequenza e dei casi d'uso in seguito alla comprensione delle tecnologie
5	18 aprile	1,5 ore	Ricerca di tool per ottenere statistiche dall'esecuzione dei test e scrittura della review dell'iterazione
6	25 aprile	2,5 ore	Comprensione di Maven e delle varie configurazioni di Junit
7	28 aprile	1,5 ore	Stesura del diagramma delle classi
8	3 maggio	1 ora	Raffinamento del diagramma di sequenza
9	4 maggio	3 ore	Comprensione dello responsabilità del modulo in coordinamento con gli altri gruppi, e stesura del diagramma dei componenti
10	6 maggio	2 ore	Progettazione preliminare di un prototipo
11	10 maggio	2 ore	Studio di Spring Boot
12	13 maggio	2 ore	Implementazione Rest Api
13	17 maggio	2,5 ore	Scrittura Codice
14	21 maggio	3 ore	Studio di Docker e Integrazione del container nel progetto
15	22 maggio	3 ore	Test di Unità dei componenti del codice
16	25 maggio	3 ore	Raffinamento dei diagrammi
17	28 maggio	3 ore	Test di Integrazione
18	31 maggio	2 ore	Modifica Codice
19	3 giugno	1 ore	Stesura della relazione e della Presentazione
20	7 giugno	2 ore	Primo tentativo di Integrazione con Task 6
21	13 giugno	3 ore	Secondo tentativo di Integrazione con Task 6
22	25 maggio	2 ore	Stesura della documentazione del progetto
23	3 luglio	2 ore	Primo tentativo di Integrazione con Task 9
24	5 luglio	3 ore	Secondo tentativo di Integrazione con Task 9
25	8 luglio	2 ore	Test Funzionali
26	10 luglio	1 ore	Stesura della relazione e della Presentazione
27	12 luglio	8 ore	Installazione del nostro servizio e tentativo di integrazione con il task T6 G16 sul server dell'università
28	13 luglio	3 ore	Revisione della documentazione del progetto

Tabella 1.1: Effort dedicato: 64 ore

## 1.4 Tecnologie Utilizzate

### 1.4.1 Docker

È Una piattaforma progettata per eseguire processi in ambienti isolabili, minimali chiamati container, semplificando i processi di deployment di applicazioni software e garantendo un elevata portabilità. È stato utilizzato per isolare il nostro servizio al fine di seguirlo indipendentemente.

### 1.4.2 Maven

È uno strumento di build automation utilizzato prevalentemente nella gestione di progetti Java. È stato utilizzato per semplificare la gestione delle dipendenze del progetto ( è possibile specificare le dipendenze del progetto in un file di configurazione (pom.xml) e Maven si occupa di scaricarle e gestirle in modo coerente). Inoltre, supporta l'uso di plugin per estendere le funzionalità del sistema. I plugin possono essere configurati nel file "pom.xml", nel nostro caso è stato utilizzato come plugin jacoco e spring-boot. Maven è stato scelto, anche perchè gestisce il processo di compilazione del codice sorgente, risolvendo automaticamente le dipendenze e generando i file compilati (classi Java) e supporta l'esecuzione dei test automatici tramite framework come JUnit. È possibile definire i test nel progetto e Maven si occupa di eseguirli durante la fase di build.

### 1.4.3 SpringBoot

È un framework open-source basato su Java che semplifica lo sviluppo di applicazioni Java. Nel nostro caso, è stato importante per la creazione di API Rest. Spring Boot offre una configurazione automatica intelligente per le API REST. Riconosce le librerie e le dipendenze correlate alle API REST e le configura in modo appropriato, riducendo la necessità di configurazioni esplicite. Esso utilizza annotazioni e convenzioni per semplificare la scrittura di codice per le API REST. Ad esempio, @RestController, @RequestMapping, @GetMapping, ecc., consentendo di definire facilmente i controller delle API e mappare i metodi ai percorsi delle richieste HTTP.

#### 1.4.4 Jacoco

È una libreria per il Code Coverage in ambito Java. JaCoCo è stato scelto perché facilmente integrabile con Maven.

### 1.5 Interazioni con altri Task

**T6-G16 Ambiente di Editor** E' stata effettuata un interazione con il gruppo G16, il quale ha avuto il compito di creare l'ambiente di Editor. E' stato tentato di collegare il button "compile and RunTest" con la nostra funzionalità di compilazione ed esecuzione dei casi di test. Ogni volta che il bottone è premuto, ci viene effettuata una richiesta di compilazione ed esecuzione e il nostro task deve fornire l'esito della compilazione nel caso in cui ci fossero errori di compilazione ed, in caso contrario, l'esito dell'esecuzione. (Tuttavia ci sono stati problemi con l'integrazione, poiché il task T6 aveva problemi a ricevere la risposta con l'esito di compilazione/esecuzione).

**T9-G27 Randoop** Si è cercato, inoltre, un interazione con il gruppo G27, il quale ha il compito di generare i test su una classe java usando il robot Randoop. Si è cercato di fare in modo che il seguente task ci fornisse una richiesta di compilazione ed esecuzione dei casi di test e che noi restituissimo in output l'esito della compilazione/esecuzione(Tuttavia ci sono stati problemi con l'integrazione, poiché il robot durante la compilazione e l'esecuzione dei casi di test si arrestava improvvisamente).

## Capitolo 2

# Documenti di analisi

### 2.1 Analisi e specifica dei requisiti

Nel seguente capitolo verranno dettagliate, chiarite e documentate le funzionalità del nostro servizio. In particolare nella fase di analisi dei requisiti verrà effettuata una descrizione completa del comportamento del nostro servizio e successivamente verranno realizzati i primi artefatti, ossia i diagrammi di analisi, che conteranno le relazioni, le interazioni e le caratteristiche del sistema durante il processo di analisi. Questi diagrammi ci serviranno per comprendere al meglio i requisiti del sistema, le entità coinvolte, le relazioni tra di esse e l'identificazione delle funzionalità.

#### 2.1.1 Requisiti

Questa sezione si concentra sull'analisi delle principali caratteristiche del nostro servizio. In particolare, sono definiti i requisiti funzionali e non funzionali che devono essere inclusi all'interno sistema. I requisiti funzionali sono stati realizzati per comprendere cosa deve fare il sistema per soddisfare le esigenze e le aspettative dell'utente, mentre, I requisiti non funzionali sono stati realizzati per comprendere i vincolo/requisiti imposti dal sistema.

#### Requisiti Funzionali

1. Il software deve poter ricevere in ingresso due file contenenti la classe di test e la classe da testare.

2. Il software deve essere in grado di lanciare la compilazione ed esecuzione dei casi di test.
3. Il software deve essere in grado di fornire informazioni sulla partita di gioco
4. Il software deve essere in grado di elaborare i risultati di esecuzione e/o compilazione e mandarli in uscita in un formato facilmente interpretabile

### Requisiti Non Funzionali

1. **Disaccoppiamento** dal resto del software: il servizio non deve utilizzare altri moduli del software.
2. **Portabilità**: il sistema deve funzionare su diverse piattaforme.
3. **Disponibilità**: il servizio deve essere idealmente sempre disponibile.
4. **Usabilità**: Il servizio deve essere facile da usare e da capire.

#### 2.1.2 Storie Utente

Sono state sviluppate delle user stories per migliorare la comprensione delle funzionalità del software, ponendo l'accento sulla prospettiva dell'utente finale. Questo approccio permette di visualizzare le funzionalità in modo più intuitivo e orientato alle necessità dell'utente.

Id 1. Da utente vorrei poter lanciare la compilazione e l'esecuzione dei miei casi di test per poter testare la classe. **PRIORITY : EPIC**

Id 2. Da utente vorrei poter visualizzare i risultati di compilazione ed esecuzione così da poter valutare il test effettuato. **PRIORITY : HIGH**

Id 3. Da utente vorrei poter identificare eventuali errori di compilazione per modificare il codice della classe o dei casi di test. **PRIORITY : HIGH**

Id 4. Da utente vorrei poter visualizzare informazioni sulla partita di gioco per valutare i risultati ottenuti. **PRIORITY : MEDIUM**

### 2.1.3 Glossario dei termini

- **Copertura:** la percentuale di linee di codice del progetto che sono state eseguite dai test dopo un'esecuzione.
- **Compilazione:** fase in cui il codice sorgente viene “tradotto” in linguaggio macchina.
- **Esecuzione:** processo tramite il quale un sistema di elaborazione esegue le istruzioni di un programma.

### 2.1.4 Diagrammi di analisi

Dopo aver analizzato le funzionalità del nostro servizio, sono stati realizzati diversi diagrammi con l'obiettivo di identificare i principali moduli software e le interazioni tra di essi. I diagrammi sono stati prodotti utilizzando il linguaggio di modellazione UML con il software Visual Paradigm CE.

#### 2.1.4.1 Diagramma dei casi d'uso

È stato realizzato un diagramma dei casi d'uso per descrivere le funzionalità offerte dal sistema, così come sono percepite e utilizzate dagli attori che interagiscono con esso. Nel nostro caso, l'insieme dei requisiti funzionali sono inglobati all'interno di un unico caso d'uso: “Compile and Run Tests”. L'attore primario è l'utente, esso interagisce con il servizio soltanto per chiedere la compilazione ed esecuzione dei casi di test. Non può chiedere la compilazione senza richiedere anche l'esecuzione. Non può chiedere la copertura del codice senza la compilazione.

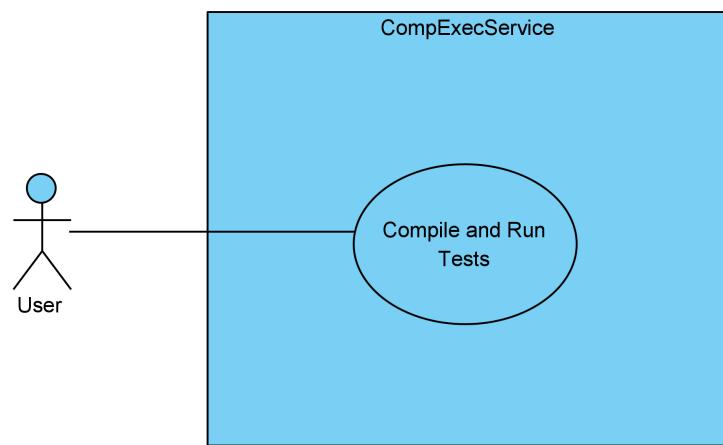


Figura 2.1: Diagramma dei casi d'uso

## Scenario

E' stato realizzato uno scenario per descrivere una situazione in cui il sistema software interagisce con l'utente con il fine di comprendere i requisiti del sistema in modo più concreto e dettagliato.

Caso d'uso:	Compile and Run Tests
Attore primario	Utente
Attore secondario	--
Descrizione	L'utente richiede compilazione ed esecuzione dei casi di test
Pre-Condizioni	Applicazione è pronta a ricevere richieste
Sequenza di eventi principale	<ol style="list-style-type: none"> <li>1. Il caso d'uso inizia quando l'utente richiede all'applicazione di compilare i test scritti da lui, inserendo due percorsi che rappresentano la classe di test e la classe da testare             <ol style="list-style-type: none"> <li>1.1. Se il percorso si riferisce ad un indirizzo web, il file viene recuperato e viene messo in una cartella.</li> <li>1.2. Altrimenti, se il percorso si riferisce a un file locale, viene direttamente spostato all'interno di una cartella</li> </ol> </li> <li>2. Il sistema compila ed esegue i test scritti dall'utente             <ol style="list-style-type: none"> <li>2.1 Se si verificano errori di compilazione, il sistema manda all'utente un file contenente gli errori di compilazione</li> <li>2.2 Altrimenti, se la compilazione è avvenuta con successo, il sistema esegue i test scritti dall'utente e manda all'utente un file contenente i risultati dell'esecuzione ed il percorso in cui è presente la coverage del codice</li> </ol> </li> </ol>
Post-Condizioni	L'utente riceve un file dov'è presente l'esito della compilazione/esecuzione dei test
Casi d'uso correlati	--
Sequenza di eventi alternativi	--

Figura 2.2: Scenario

### 2.1.4.2 Diagramma delle classi

E' stato realizzato un diagramma delle classi per descrivere i diversi tipi di oggetti all'interno di un sistema e i tipi di relazioni che esistono tra loro. In particolare nel nostro caso le Entità sono: Controller, Compilatore, EsecutoreTest, Process.

Il nostro servizio esporrà delle API di tipo REST. Al servizio verrà fornito l'url della classe da testare(urlClass) e della classe di test (urlTestClass). Il servizio manderà in output le informazioni relative all'esito della compilazione e in caso di avvenuta compilazione senza errori, l'esito dei test eseguiti in un formato interpretabile. Un ipotetico user del nostro servizio, quindi, utilizzerà l'interfaccia per mandare richieste che saranno poi gestite dal controllore.

In particolare per un interfaccia è possibile avere più controllori, ciò significa che è possibile fare più richieste contemporaneamente al controllore. Il controllore a sua volta è collegato alle entità

Compilatore ed EsecutoreTest tramite una relazione di tipo associativo. Il compilatore implementa il metodo il metodo Compile, mentre l'EsecutoreTest implementa il metodo RunTests. Il compilatore quindi, effettuerà la compilazione delle classi scritte dal giocatore mentre l'esecutore effettuerà l'esecuzione. Sia il compilatore sia L'esecutore sono collegati al Process, il quale fornisce l'esito della compilazione nel caso di errori di compilazione o l'esistenza dell'esecuzione e il report sulla copertura del codice in caso contrario.

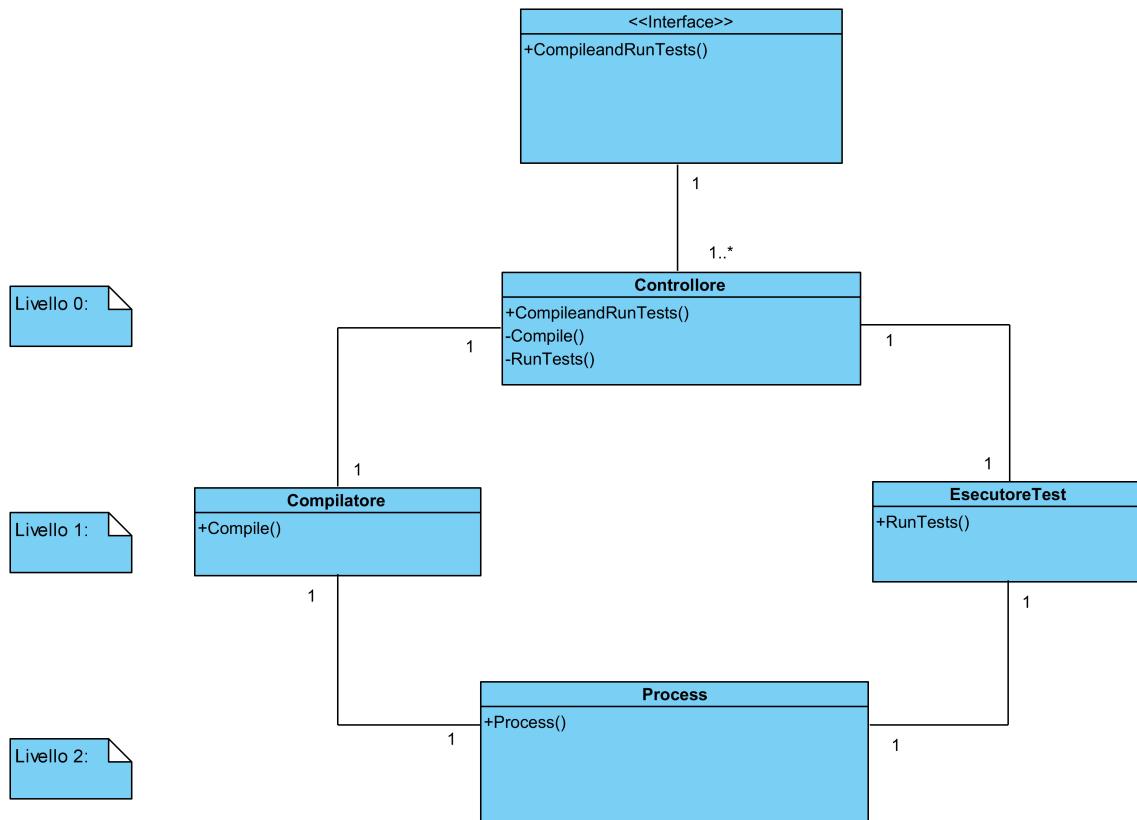


Figura 2.3: Diagramma delle classi Analisi

#### 2.1.4.3 Diagramma di sequenza

E' stato realizzato un diagramma di sequenza per illustrare la sequenza di messaggi e le interazioni necessarie per far sì che un utente possa ricevere l'esito della compilazione/esecuzione dei test a partire dalla richiesta di compilazione ed esecuzione dei test. Il diagramma di sequenza realizzato è riferito all'unico scenario individuato: Compile and Run Tests.

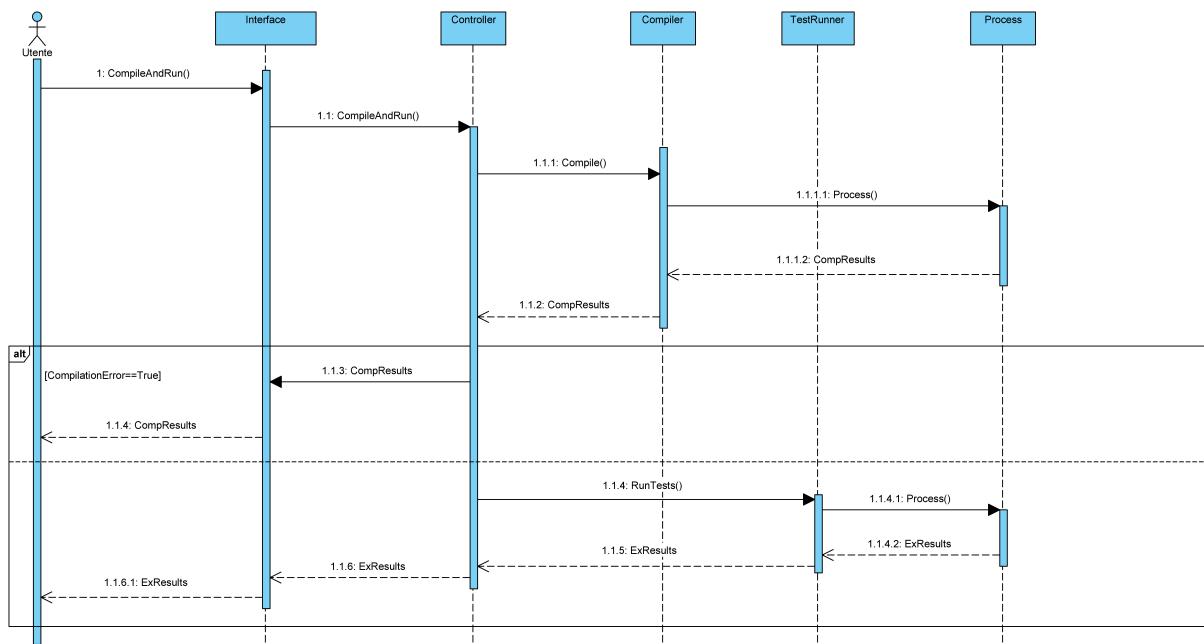


Figura 2.4: Diagramma di Sequenza Analisi

# Capitolo 3

## Fase di progettazione

Nel seguente capitolo si entra nel merito della struttura del sistema software. Viene riportata una documentazione dettagliata del sistema nel suo insieme e si descrivono le responsabilità e le interazioni di ogni modulo del sistema al fine di raggiungere gli obiettivi predisposti in fase di analisi precedentemente svolta.

Essendo il nostro software pensato per essere integrato nel progetto ENACTEST (European iNnovation AllianCe for TESting eduCaTion), che aderisce a un tipo di architettura orientata ai servizi (SOA), il nostro software sarà dunque un servizio indipendente e disaccoppiato con una precisa funzionalità che gli altri moduli potranno richiedere attraverso l'interfaccia esposta.

### 3.1 Scelta architetturale

Lo stile architetturale adottato è quello Layered, nella sua tipologia classica Closed Layer (CLA). E' stata effettuata tale scelta progettuale per dare una struttura gerarchica e ben definita del sistema, così che ciascun layer abbia una responsabilità specifica che possa esser utilizzata dal layer direttamente superiore.

In particolare, la struttura in layers del software è la seguente:

- **Layer 1:** è il layer che fornisce l'API RESTful ed intercetta richieste HTTP GET e POST;
- **Layer 2:** è il layer del Manager, che gestisce le richieste e gestisce il flusso delle operazioni successive utili per gestire la richiesta;

- **Layer 3:** è il layer del Compiler e del TestRunner, che si occupano rispettivamente della compilazione e dell'esecuzione della classe di test;
- **Layer 4:** è il layer del Processing dei risultati, costituito dai due Process che elaborano i risultati rispettivamente di compilazione ed esecuzione, che verranno forniti al manager in un formato standard;

Ulteriori vantaggi dell'utilizzo di questo stile architettonico consentono di soddisfare il requisito di manutenibilità, poiché risulta semplice modificare indipendentemente un layer dato che esso influisce al più sul layer superiore, purché l'interfaccia sia rispettata.

Pertanto, risulterà anche particolarmente semplice da testare e riutilizzare ciascun layer. Inoltre, essendo un numero esiguo di layers, l'architettura scelta non andrà ad inficiare significativamente sulle performance e sull'efficienza del sistema sebbene sia stato introdotto overhead.

## 3.2 Artefatti di progettazione

In questa sezione vengono illustrati gli artefatti prodotti in fase di progettazione. In particolare sono stati prodotti diverse tipologie di diagrammi con l'obiettivo di fornire la vista del sistema software da diverse prospettive. Tali diagrammi sono stati prodotti utilizzando il linguaggio di modellazione UML con il software Visual Paradigm CE.

Vengono elencati prima i diagrammi che ci forniscono una vista sul comportamento del servizio in esecuzione e successivamente diagrammi per una vista statica della struttura del sistema.

### 3.2.1 Diagramma di attività

L'obiettivo del seguente diagramma è delineare il comportamento del software in termini di flusso di azioni svolte durante l'esecuzione dell'attività, iniziata a seguito di una richiesta al servizio. Ogni nodo nel diagramma rappresenta un'azione che incorpora l'esecuzione di diverse operazioni. Vi sono inoltre nodi di decisione per separare i diversi rami di esecuzione al variare delle condizioni:

- il servizio, dopo aver ricevuto una richiesta di compilazione ed esecuzione dei test, deve discriminare il tipo di richiesta (GET o POST) e gestire diversamente il recupero della classe da testare della classe di test a seconda della richiesta.

- Altra biforcazione ci sarà nel caso in cui, a seguito della compilazione della classe di test, essa esibisca errori e dunque non sia possibile procedere con l'esecuzione della classe di test.
- In ogni caso, il servizio dovrà rispondere alla richiesta fornendo l'esito, un log dei risultati di compilazione e/o esecuzione coerenti ed un percorso in cui è possibile recuperare i risultati di copertura. Per dettagli riferisci alla sezione 4.2.

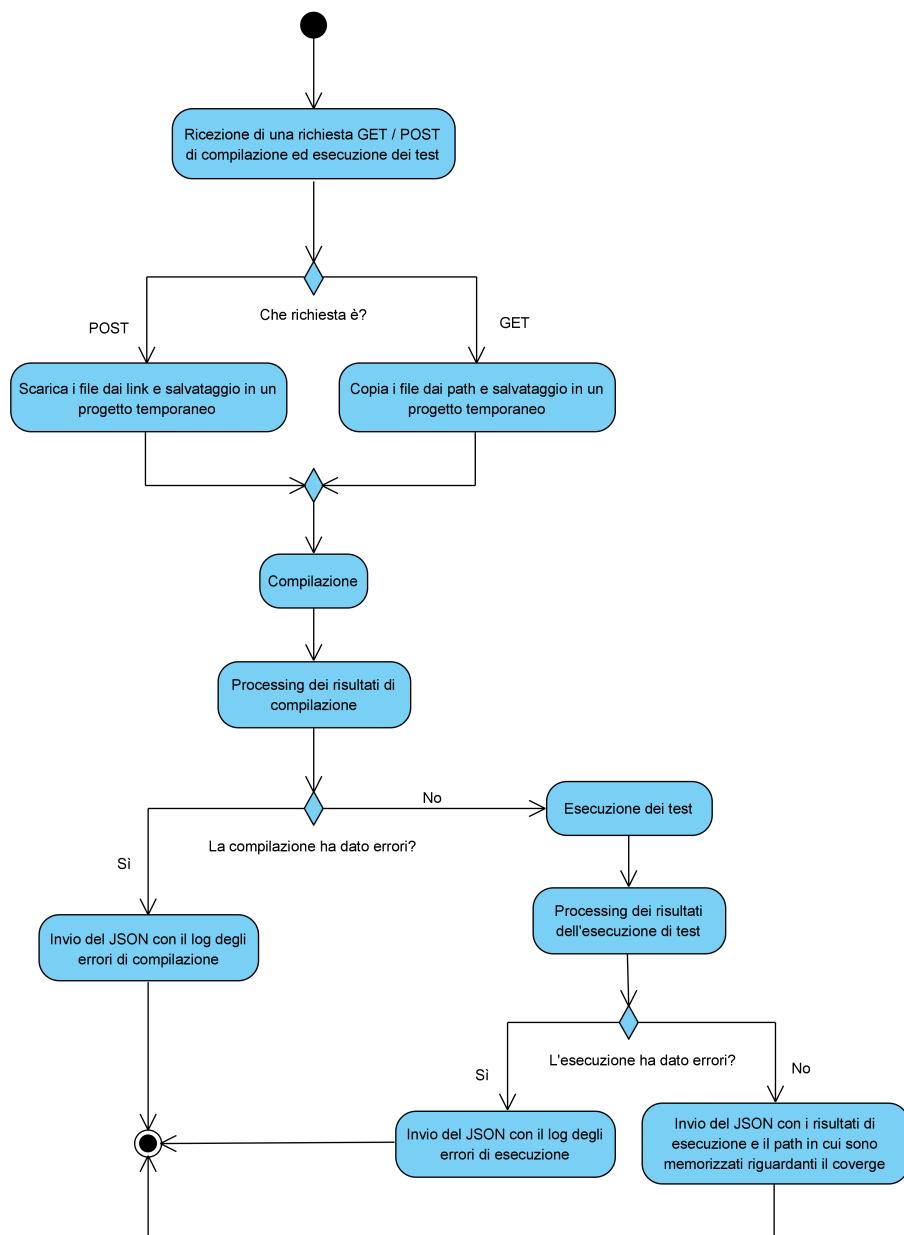


Figura 3.1: Diagramma di attività

### 3.2.2 Diagramma di sequenza

Sono stati prodotti due diagrammi di sequenza ciascuno per i due tipi di scenari possibili a seconda del tipo di richiesta.

Con tali diagrammi si vuole evidenziare il comportamento del software focalizzandosi sui messaggi scambiati tra i componenti partecipi durante la risoluzione della richiesta. Il diagramma mostra dunque la sequenza di messaggi e le interazioni necessarie per far sì che un utente possa ricevere l'esito della compilazione/esecuzione dei test a partire dalla richiesta di compilazione ed esecuzione dei test, effettuata tramite l'interfaccia.

### 3.2.3 Diagramma dei componenti

Nel seguente diagramma sono mostrati tutti i moduli del software precedentemente accennati sottoforma di componenti e le loro relazioni mediante connessioni tra le interfacce.

E' visibile la struttura a livelli del servizio, il quale espone verso l'esterno un'API di tipo REST grazie al componente RestController, che è in ascolto di richieste HTTP mediante un server TomCat (layer 1). Il componente Manager (layer 2) successivamente gestisce la richiesta e si interfaccia con il layer successivo di compilazione ed esecuzione (layer 3) dal quale otterrà i risultati di compilazione e/o esecuzione a seconda dell'esito. In prima battuta, i risultati di compilazione verranno elaborati dal componente ProcessC dell'ultimo layer, e mandati indietro al Manager che valuta (a seconda dei risultati di compilazione) se procedere con l'esecuzione. In caso affermativo il Manager procede con l'esecuzione della classe di test mediante il componente TestRunner del layer 3, il quale successivamente utilizza il ProcessE del layer 4 per elaborare i risultati di esecuzione.

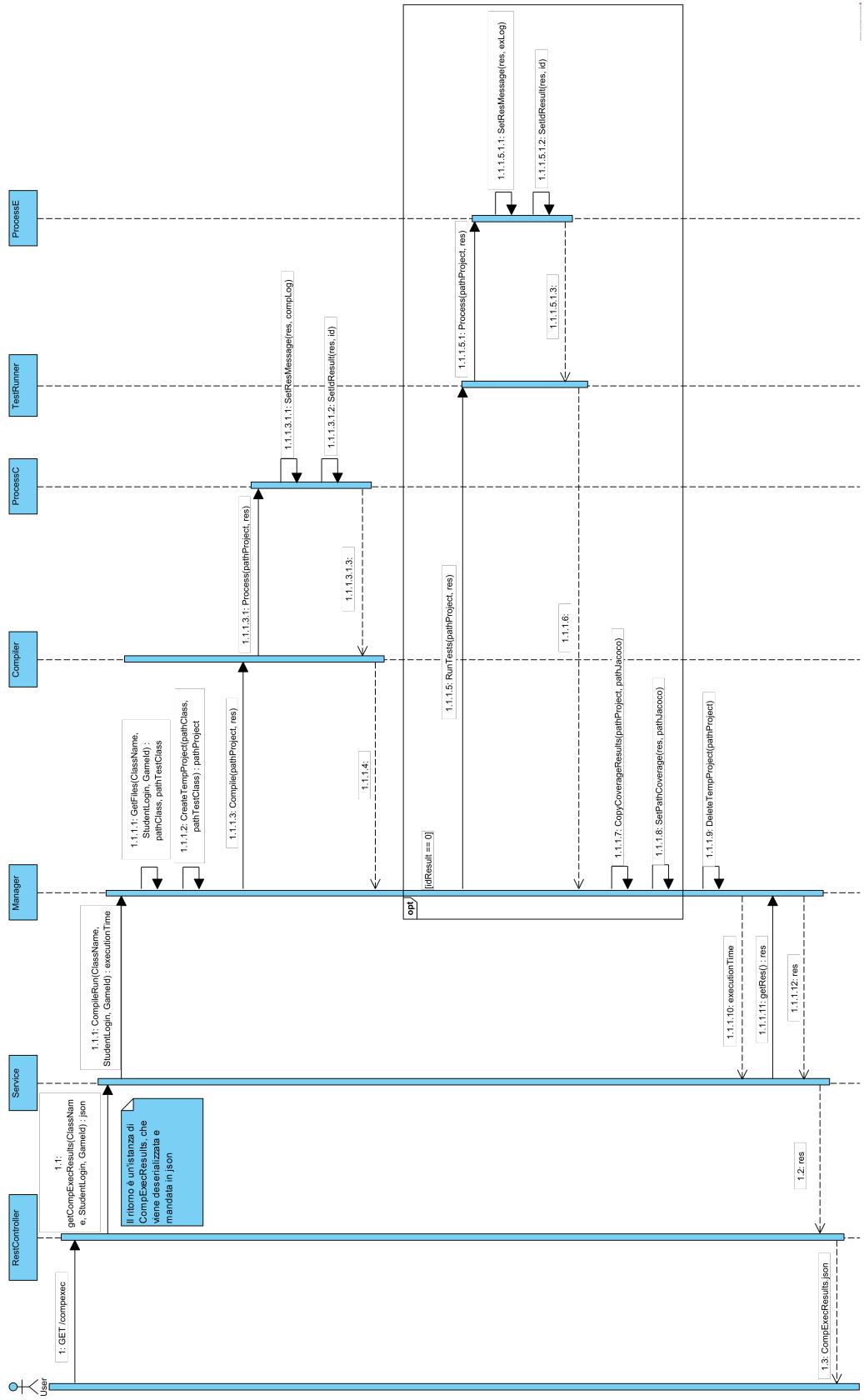


Figura 3.2: Diagramma di sequenza GET /compexec

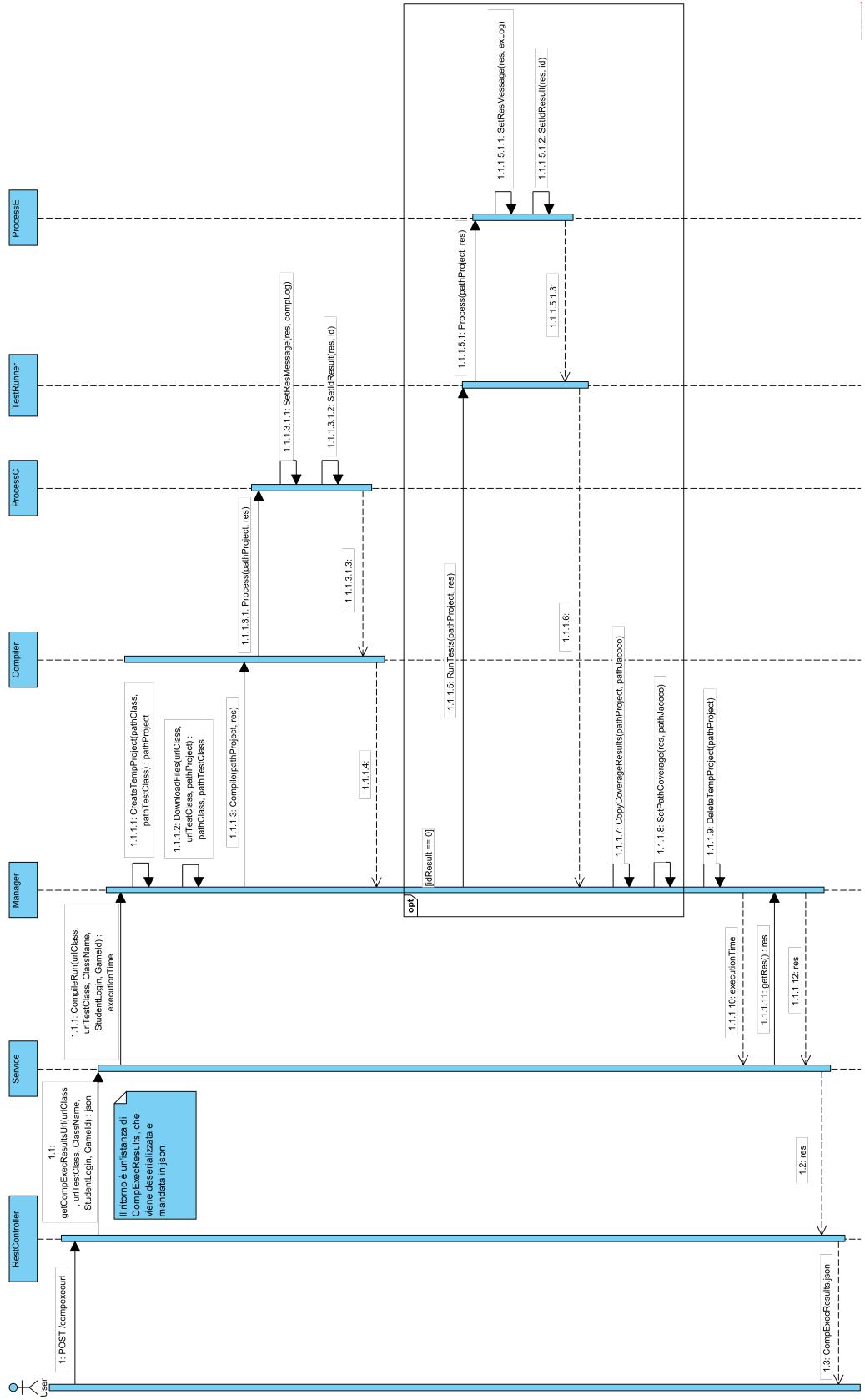


Figura 3.3: Diagramma di sequenza POST /compexecURL

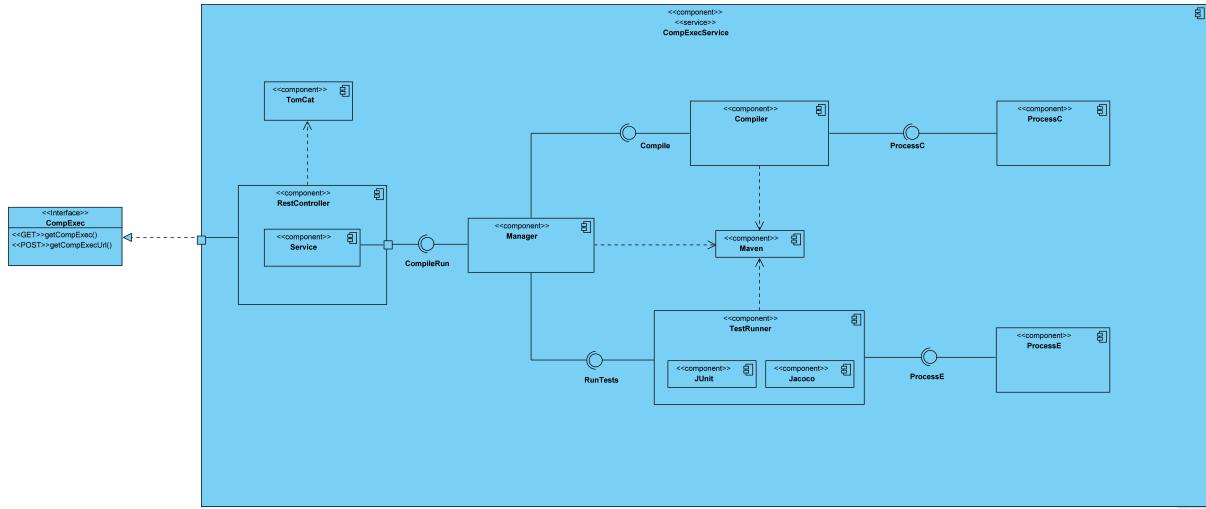


Figura 3.4: Diagramma dei componenti

### 3.2.4 Diagramma delle classi

Il seguente diagramma è la versione raffinata in fase di progettazione del diagramma delle classi esposto precedentemente in 2.1.4. Le classi ora coinvolte sono: CompExecController, CompExecService, Manager, Compiler, TestRunner, ProcessC, ProcessE e CompExecResults. CompExecController è la classe che, come già detto, espone le API REST ed è in ascolto di richieste HTTP. Essa è associata alla classe CompExecService che si occupa di interfacchiare il Controller con il Manager.

Da notare la cardinalità 1, 1..\* tra il CompExecService e il Manager, in quanto ad ogni richiesta proveniente dal CompExecService verrà istanziato un nuovo Manager per servire la richiesta, così da garantire che il servizio sia disponibile a servire altre richieste in contemporanea.

Il Manager infatti ha come attributi le directory necessarie per creare un nuovo progetto temporaneo ad ogni ricezione di una richiesta di compilazione ed esecuzione dei test, che verrà eliminato una volta servita la richiesta. Maggiori dettagli sull'implementazione disponibili nel capitolo 4.

Il Manager contiene come attributo un'istanza di CompExecResults, che è la classe utilizzata per rappresentare i risultati di compilazione ed esecuzione inerente ad una richiesta e che verrà inviata, in un opportuno formato, come risposta. Gli attributi di tale classe sono:

- **idResult**: un'intero rappresentante l'esito della compilazione e/o esecuzione dei test. 0 se è andata a buon fine, 1 se c'è stato un'errore di compilazione, 2 se c'è stato un errore in

## CAPITOLO 3. FASE DI PROGETTAZIONE

esecuzione;

- **resMessage:** contiene il log dei risultati di compilazione e/o esecuzione
- **pathCoverage:** contiene la directory dove saranno contenuti i risultati di copertura

Il CompExecResults del Manager verrà aggiornato dopo la compilazione dalla classe ProcessC e analogamente, in caso di avvenuta compilazione, lo farà il ProcessE in merito all'esito dell'esecuzione.

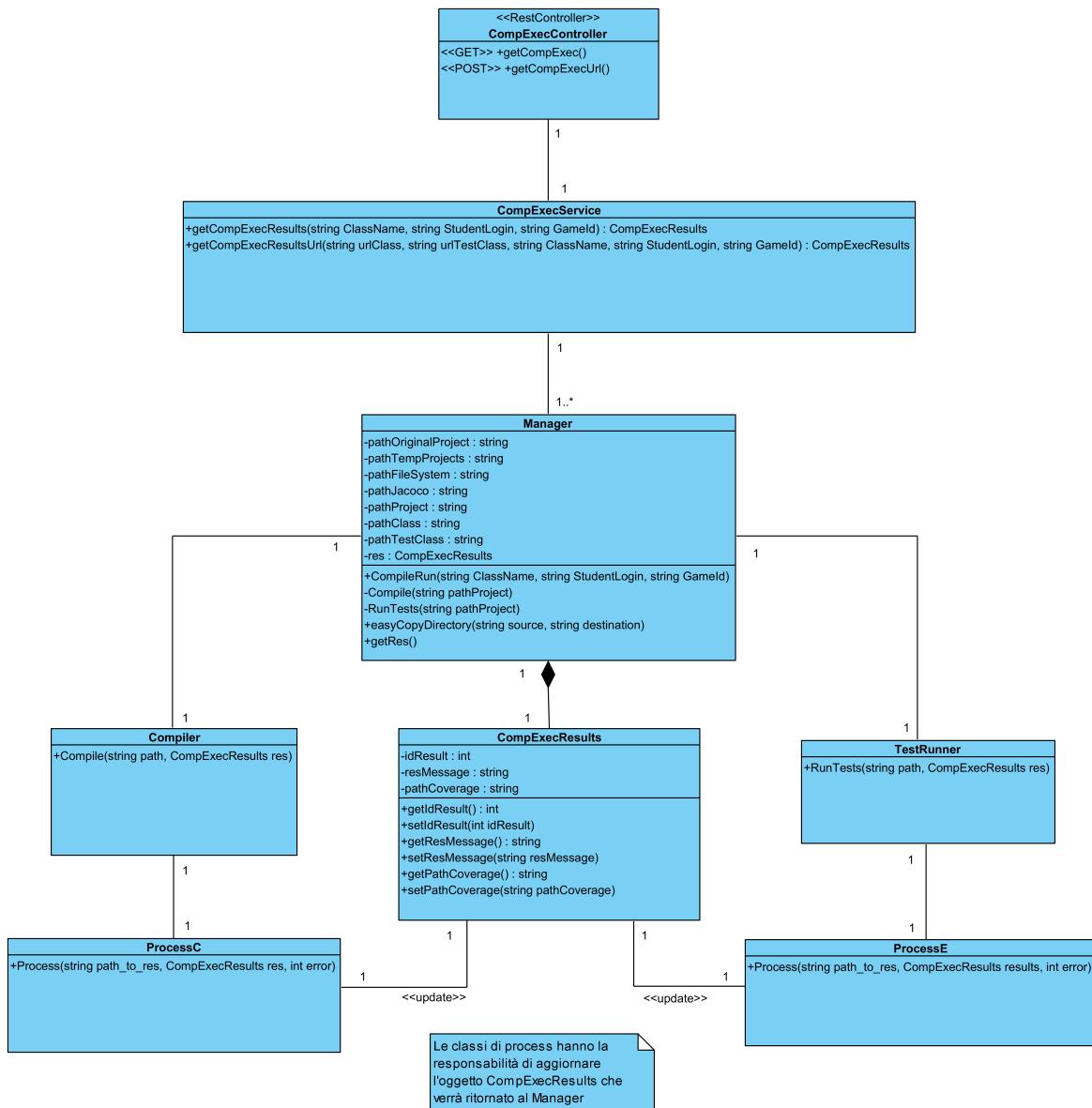


Figura 3.5: Diagramma delle classi di progettazione

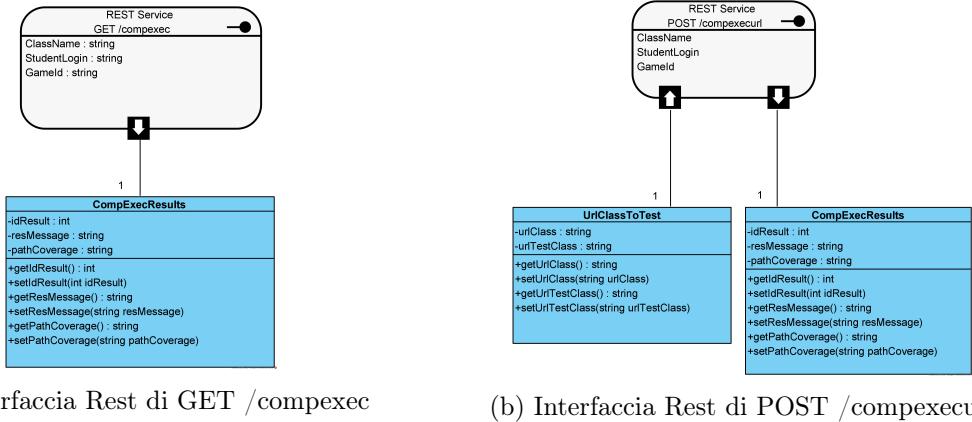


Figura 3.6: Interfacce Rest

### 3.2.5 Diagramma dei package

Nel seguente diagramma sono mostrati tutti i package utilizzati nel progetto. I package raggruppano le classi viste nel diagramma delle classi in gruppi semanticamente coerenti, rendendo più chiara la struttura dell'intero software.

Dalla figura sottostante possiamo notare come vi sia un package che racchiude quasi tutto, cioè quello del progetto principale "com.T07.compExec". All'interno di questo vi sono due package, uno che racchiude tutte le classi relative all'implementazione delle API del servizio e l'altro racchiude le classi che implementano la logica di compilazione, esecuzione dei test e processing dei risultati. Il package delle API è ulteriormente diviso in due package, uno che contiene la classe del controller responsabile della ricezione delle richieste HTTP, mentre l'altro contiene le classi che modellano i dati che il servizio scambia da/verso l'esterno. Naturalmente sono presenti delle dipendenze da questi package, che servono per implementare la corretta logica dell'intera applicazione. Infine è presente un ulteriore package "TestPackage" esterno a quello del progetto principale, che è quello che contiene le classi, da testare e di test, che il nostro servizio di compilare, proprio per questo la relazione è di tipo *use*.

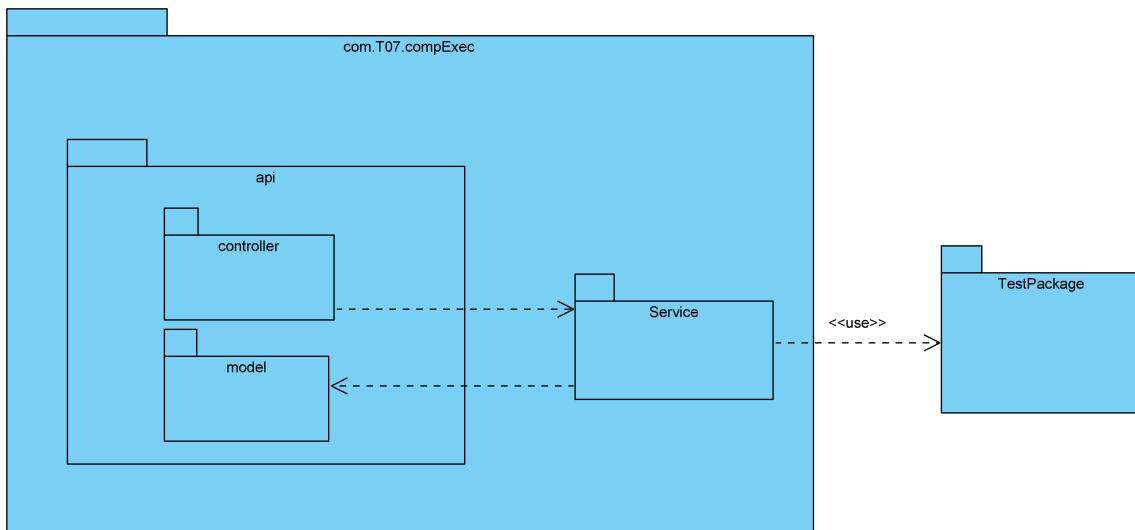


Figura 3.7: Diagramma dei package

### 3.2.6 Diagramma di deployment

Il seguente diagramma mostra l'architettura in cui si inserisce il nostro sistema in esecuzione. Il diagramma di deployment mostra la configurazione dei nodi di elaborazione a tempo di esecuzione e dei componenti che vi risiedono, in genere offre una vista statica del sistema che mostro la topologia dell'hardware nello stesso.

Dalla figura sottostante possiamo notare come l'interezza del sistema modellato è in esecuzione su un nodo, chiamato Host System. Il nostro servizio, modellato dal componente CompExecService, si manifesta come un file JAR ottenuto dal build del nostro progetto, ed è in esecuzione su un nodo che vedremo nel capitolo 4 essere un container docker che ci permette di isolare le nostre dipendenze dal resto del sistema. Il nostro progetto utilizza inoltre un progetto accessorio per svolgere compilazione ed esecuzione dei test, che nel diagramma è il ProjectUt. Abbiamo modellato la parte del software che interagisce con noi e su cui non abbiamo lavorato direttamente come un nodo, il cui nome TestingGame. L'interazione con noi avviene attraverso un collegamento diretto per le richieste di compilazione ed esecuzione di test, ma i file su cui lavorare non vengono inviati su quel collegamento ma sono condivisi attraverso lo Shared FileSystem su cui insistono entrambi gli ambienti di esecuzione. Il modo in cui avviene la comunicazione tra Testing Game e Compexec Container tramite anche lo Shared FileSystem verrà affrontata con più dettaglio nella sezione 4.2

Seguendo poi le direttive degli stakeholder abbiamo previsto un FileSystem condiviso a cui accede sia il nostro servizio che il resto del software del testing game per memorizzare la

Con il nodo Testing Game abbiamo deciso di modellare la parte del software che interagisce con noi e su cui non abbiamo lavorato direttamente.

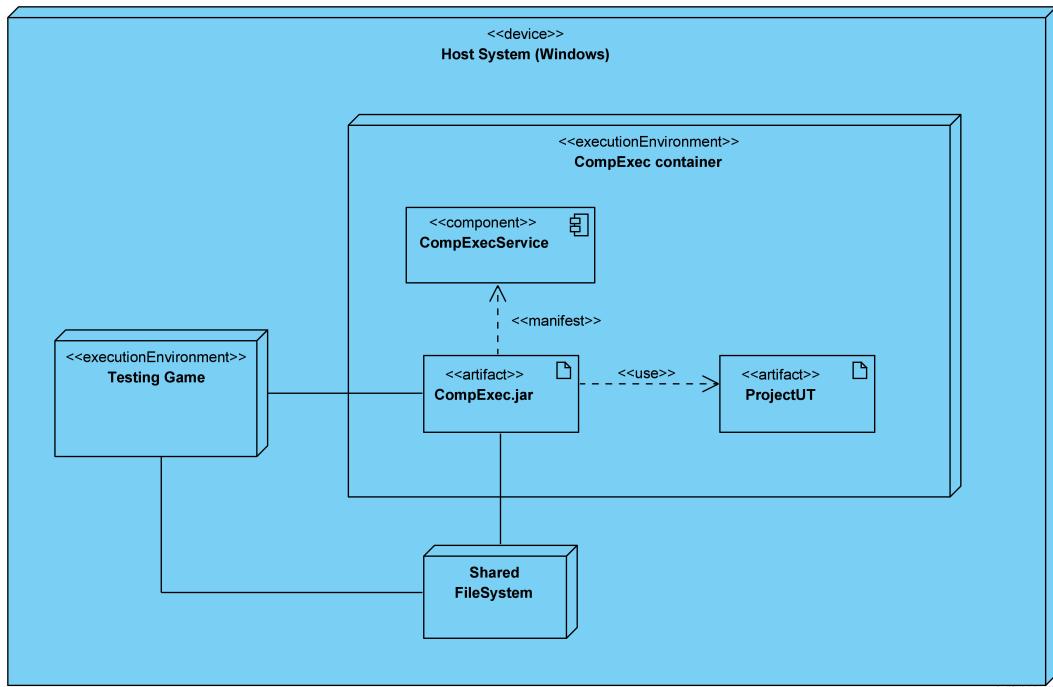


Figura 3.8: Diagramma di deployment

# Capitolo 4

## Implementazione e testing

In questo capitolo ci occuperemo di descrivere come abbiamo implementato il software descritto nei due capitoli precedenti e come abbiamo testato il suo corretto funzionamento. Inoltre forniremo anche una descrizione precisa dell’interfaccia del modulo software per permettere un utilizzo semplice dello stesso.

### 4.1 Documentazione dell’implementazione

Partire dal riferimento all’architettura e al diagramma dei componenti Dire che classi abbiamo utilizzato e cosa fanno in breve Dire attraverso cosa il servizio si interfaccia con l’esterno e come è stato implementato il server e tutte cose Spiegare come il servizio è stato tutto ridotto ad un jar e come si interfaccia con il projectUT (è un cartella) Dire che abbiamo usato docker per il deploy, spiegare dockerfile Spiegare alla fine come ci si interfaccia con il servizio mentre sta runnando

Abbiamo deciso di codificare il nostro servizio utilizzando il Java (versione 17) utilizzando Apache Maven (versione 3.8.3) per gestire il progetto e la sua build automation. Maven sfrutta un Project Object Model (POM) per gestire tutte le informazioni del progetto, le dipendenze e le direttive per il building del progetto Java, che opportunamente modificato ci ha permesso di creare l’applicazione nella sua interezza.

Per realizzare la struttura base del nostro servizio abbiamo usato lo strumento Spring Boot che semplifica e velocizza lo sviluppo di applicazioni web e microservizi basati sul framework Spring di Java. Per realizzare l’interfaccia del servizio, invece, abbiamo utilizzato Spring Web

MVC, un framework molto utile che, attraverso l'utilizzo del pattern architetturale Model View Controller (MVC), permette di implementare in modo semplice delle API REST. Per fare ciò, apre un server Apache Tomcat in grado di ricevere richieste HTTP, che vengono poi accettate (o rifiutate) e gestite dal controller.

Il controller Rest è quindi essenziale per l'implementazione dell'interfaccia, ed è anche il punto in cui vengono definiti gli endpoint disponibili, il tipo di richieste supportate ed i parametri necessari da associare alle richieste. Il punto di collegamento tra l'interfaccia ed il resto del modulo è una classe Service, che inoltra la richiesta ricevuta dal controller al Manager. In particolare abbiamo previsto un meccanismo che prevede la creazione di più istante del Manager nel caso in cui arrivino altre richieste mentre se ne sta servendo una, ogni istanza di Manager lavorerà con un progetto diverso per eseguire compilazione ed esecuzione dei test, maggiori informazioni più avanti.

Importanti per il funzionamento dell'interfaccia sono anche delle classi Model che modellano il formato di input ed output della nostra interfaccia, entrambe vengono utilizzate dal controller per la conversione tra JSON e classe e viceversa.

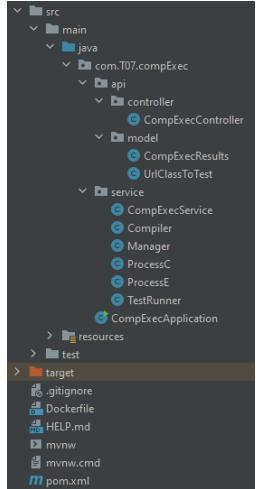
Di seguito riportiamo in breve la struttura del progetto, che rispecchia quanto visto nel diagramma dei package 3.2.5.

Il nostro servizio è incaricato di compilare ed eseguire i casi di test di una classe, per farlo abbiamo creato un altro progetto Java a cui ci riferiremo utilizzando il termine ProjectUT.

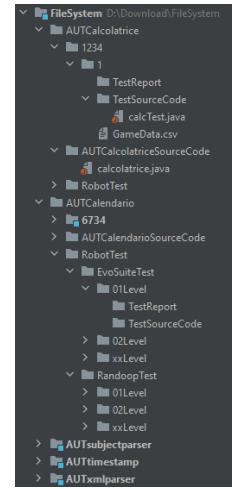
ProjectUT è un progetto generato attraverso l'archetipo di Maven Quickstart il cui POM è stato modificato per includere JUnit, necessario per eseguire i test, e JaCoCo, necessario per generare la copertura del codice in seguito ai test. Il servizio principale sarà quindi in grado di lanciare comandi Maven per raggiungere gli obiettivi di compilazione ed esecuzione dei test. In particolare il nostro servizio è progettato per creare un progetto uguale a ProjectUT ad ogni ricezione di una richiesta di compilazione ed esecuzione dei test, questo verrà eliminato una volta servita la richiesta. Questo meccanismo dei progetti temporanei è anche essenziale per servire più richieste di compilazione ed esecuzione in contemporanea, infatti ogni richiesta sarà associata ad un progetto temporaneo diverso che verrà gestito da un'istanza della classe Manager diversa, come accennato poco sopra.

Un altro componente importante per l'implementazione del nostro servizio è l'utilizzo di un

FileSystem, infatti, conformandoci alle richieste degli stakeholders abbiamo deciso di prelevare le classi da compilare ed eseguire da un filesystem condiviso, su cui verranno anche memorizzati i risultati riguardanti la coverage prodotti dal nostro servizio. Il FileSystem ha una struttura ben precisa, è diviso in base a 3 informazioni, classe da testare, id dello studente che svolge i test ed id della partita giocata dallo studente. Di seguito la struttura del FileSystem condiviso:



(a) Struttura del progetto



(b) Struttura del Shared FileSystem

Quindi abbiamo utilizzato Maven per ottenere un file JAR relativo al nostro progetto e che, messo in esecuzione, permettesse di eseguire stabilmente il nostro servizio. Di supporto all'esecuzione è necessario vi sia, nella stessa directory del JAR del servizio, la cartella del ProjectUT ed un link al FileSystem condiviso.

In conclusione abbiamo deciso di encapsulare il nostro servizio e quindi anche le sue dipendenze (Java e Maven) utilizzando Docker, una piattaforma per eseguire e condividere in modo semplice applicazioni di qualsiasi tipo. Per creare un container che ospitasse il nostro servizio è stato necessario scrivere un dockerfile, che descrive i passi da seguire per creare il container che contenga il nostro servizio. Di seguito riportiamo il suddetto dockerfile:

```
FROM maven:3.8.3-openjdk-17
RUN mkdir projectUT
RUN mkdir App
COPY projectUT/ projectUT/
COPY compExec-0.0.1-SNAPSHOT.jar App/CompExec.jar
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "/App/CompExec.jar"]
```

Figura 4.2: Dockerfile

In breve partiamo da un'immagine di container base con Java 17 e Maven 3.8.3 installati, vengono create alcune cartelle di supporto all'esecuzione e viene copiata la cartella del ProjectUT. Successivamente, all'avvio del container verrà effettuato il collegamento tra la cartella FileSystem interna al container con il FileSystem condiviso del sistema host.

Avviato il container, il servizio può essere acceduto tramite una semplice richiesta HTTP (i cui dettagli saranno discussi nella sezione precedente) effettuata sul porto definito nella proprietà dell'applicazione Spring Boot. In risposta vi sarà il log di errore e il path, riferito al FileSystem condiviso, in cui vi sono i risultati riguardanti la copertura ottenuti tramite JaCoCo.

## 4.2 Documentazione dell'interfaccia

spiegare come i requisiti hanno influito sull'interfaccia e sul fatto che ora ci vengono passate le classi in modo indiretto Dire in generale cosa compone l'interfaccia, riferendosi anche ai diagrammi e sugli assunti che facciamo per lavorare così (forse da dire nella sezione dell'implementazione)  
Presentare swagger e spiegare come abbiamo realizzato la documentazione OpenAPI con swagger  
Mostrare la documentazione e dire dove è accessibile

Abbiamo impiegato molto tempo a riflettere sull'interfaccia del nostro servizio, era necessario sia semplice, generale e fornisca le tutte le informazioni richieste. Tenendo in considerazione ciò e tenendo in considerazione i requisiti funzionali presentati in precedenza abbiamo elaborato una prima interfaccia che prevedesse in ingresso il path della classe da testare e quello della classe di test, quindi un riferimento diretto alle classi. Con l'introduzione del FileSystem condiviso, che ha una struttura ben precisa, abbiamo pensato di generalizzare l'interfaccia, quindi non riferendoci in modo diretto alle classi, ma utilizzando un riferimento indiretto. Infatti è possibile identificare univocamente una classe da testare e il codice di test scritto dall'utente con solo 3 informazioni:

- **ClassName:** Il nome della classe sotto test
- **StudentLogin:** L'identificativo univoco dell'utente
- **GameId:** L'identificativo della partita che l'utente sta giocando

Quindi, fatte queste considerazioni, abbiamo definito un primo endpoint. Inoltre abbiamo previsto la possibilità di fornire, oltre i tre parametri visti sopra, anche degli url a cui trovare i files di

## CAPITOLO 4. IMPLEMENTAZIONE E TESTING

classe da testare e classe di test. Questo potrebbe essere utile nel caso in cui entrambi i files non siano presenti nel FileSystem condiviso, in tal caso il nostro servizio recupera i files dagli url, li memorizza nelle directory corrette utilizzando i parametri forniti ed esegue poi compilazione ed esecuzione dei test.

In risposta il nostro servizio fornisce sempre un JSON con tre campi:

- **idResult:** un intero che indica l'esito dell'operazione, 0 indica che tutto è andato a buon fine, 1 indica errori di compilazione e 2 indica errori nell'esecuzione dei test;
- **resMessage:** una stringa contenente il log di compilazione o esecuzione dei test (a seconda dell'idResult);
- **pathJacoco:** una stringa contenente il path riferito al FileSystem condiviso in cui trovare i risultati della coverage forniti da JaCoCo.

```

JSON Raw Data Headers
Save Copy Collapse All Expand All Filter JSON
idResult: 0
resMessage: "[INFO] Scanning for projects...[WARNING] Some problems were encountered while building the effective model for T07:projectUT:jar:1.0-SNAPSHOT[WARNING] 'build.plugins.plugin.version' for org.apache.maven.plugins:maven-surefire-plugin is missing. @ line 53, column 15[WARNING] 'build.plugins.plugin.version' for org.apache.maven.plugins:maven-failsafe-plugin is missing. @ line 58, column 15[WARNING] It is highly recommended to fix these problems because they threaten the stability of your build.[WARNING] [WARNING] For this reason, future Maven versions might no longer support building such malformed projects.[WARNING] [INFO] [INFO] .....< T07:projectUT >.....[INFO] Building projectUT 1.0-SNAPSHOT[INFO] .....[jar].....[INFO] [INFO] --- jacoco-maven-plugin:0.8.6:prepare-agent (prepare-agent) @ projectUT ---[WARNING] The artifact xml-apis:xml-apis:jar:2.0.2 has been relocated to xml-apis:xml-apis:jar:1.0.b2[INFO] argline set to -javaagent:/root/.m2/repository/org/jacoco/org.jacoco.agent/0.8.6/org.jacoco.agent-0.8.6-runtime.jar=destfile:/tempRun/1689356864793/target/jacoco.exec[INFO] [INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ projectUT ---[INFO] Using 'UTF-8' encoding to copy filtered resources.[INFO] skip non-existing resource directory /tempRun/1689356864793/src/main/resources[INFO] [INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ projectUT ---[INFO] Changes detected - recompiling the module![INFO] Compiling 1 source file to /tempRun/1689356864793/src/test/[target/classes][INFO] [INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ projectUT ---[INFO] Using 'UTF-8' encoding to copy filtered resources.[INFO] skip non-existing resource directory /tempRun/1689356864793/src/test/[resources][INFO] [INFO] --- maven-compiler-plugin:3.1:testCompile (default-testCompile) @ projectUT ---[INFO] Changes detected - recompiling the module![INFO] Compiling 1 source file to /tempRun/1689356864793/target/test-classes[INFO] [INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ projectUT ---[INFO] Surefire report directory: /tempRun/1689356864793/target/surefire-reports[INFO] .....[INFO] [INFO] \nT E S T S \n[INFO] [INFO] \nRunning TestPackage.calcTest\nTests run: 2, Failures: 0, Errors: 0, Skipped: 0. Time elapsed: 0.037 sec\n[INFO] [INFO] Results :\n[INFO] [INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0\n[INFO] [INFO] --- jacoco-maven-plugin:0.8.6:report (report) @ projectUT ---[INFO] Loading execution data file /tempRun/1689356864793/target/jacoco.exec[INFO] Analyzed bundle 'projectUT' with 1 classes[INFO] [INFO] BUILD SUCCESS[INFO] [INFO] Total time: 1.161 s\n[INFO] Finished at: 2023-07-14T17:47:47Z\n[INFO] ....."
pathCoverage: "FileSystem/AUTCalcolatrice/1234/1/TestReport/"

```

Figura 4.3: Esempio di JSON che il servizio restituisce in risposta

I risultati prodotti da Jacoco riguardo la coverage sono organizzati in una cartella con la struttura presentata in figura 4.4. In particolare sfogliando il file index.html è possibile osservare tutti i risultati prodotti, un esempio riguardante la classe calcolatrice è riportato nelle seguenti figure.

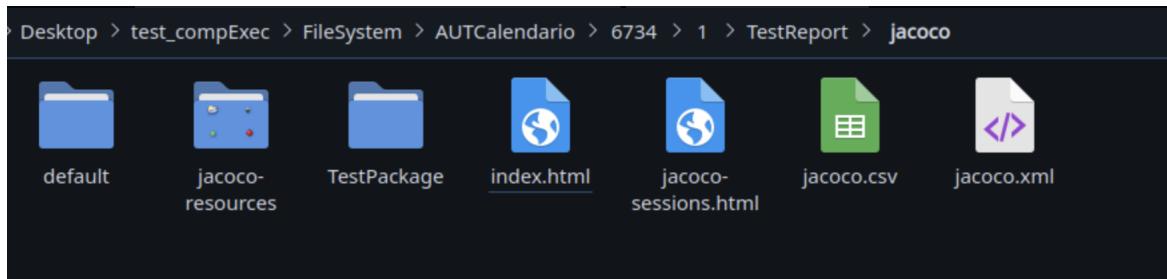


Figura 4.4: Jacoco\_struct

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Ctry	Missed	Lines	Missed	Methods	Missed	Classes
<code>calcolatrice</code>	73%		n/a		1	4	1	4	1	4	0	1
Total	4 of 15	73%	0 of 0	n/a	1	4	1	4	1	4	0	1

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Ctry	Missed	Lines	Missed	Methods	Missed	Classes
<code>sub(int, int)</code>	0%		n/a		1	1	1	1	1	1	1	1
<code>add(int, int)</code>	100%		n/a		0	1	0	1	0	1	0	1
<code>multiply(int, int)</code>	100%		n/a		0	1	0	1	0	1	0	1
<code>calcolatrice()</code>	100%		n/a		0	1	0	1	0	1	0	1
Total	4 of 15	73%	0 of 0	n/a	1	4	1	4	1	4	0	1

```

1. package TestPackage;
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. }

```

Inoltre in risposta ad una richiesta possono presentarsi anche gli errori 400 e 500 HTTP, questo perchè non è stato ancora implementato un mapping degli errori preciso e puntuale per la nostra applicazione.

L'interfaccia è realizzata utilizzando delle API Rest, che abbiamo deciso di documentare utilizzando Swagger che è uno strumento per la documentazione di questo tipo di API. Swagger

permette di realizzare una specifica chiara, dettagliata ed interattiva delle API, inclusi endpoint, i metodi di richiesta e i codici di risposta, nonché le informazioni sulle strutture e i tipi di dati utilizzati dall'API. La specifica fatta da Swagger segue il formato di documentazione open-source definito dallo standard OpenAPI.

In particolare Swagger è in grado di generare questa documentazione in modo automatico, in particolare abbiamo utilizzato SpringDoc che sfrutta Swagger per creare la specifica della API ed hosta la pagina generata da Swagger sul server Tomcat creato da Spring.

Di seguito riportiamo come si presenta la documentazione con cui si può interagire e che è raggiungibile e consultabile una volta che il servizio è in esecuzione all'endpoint /swagger-ui:

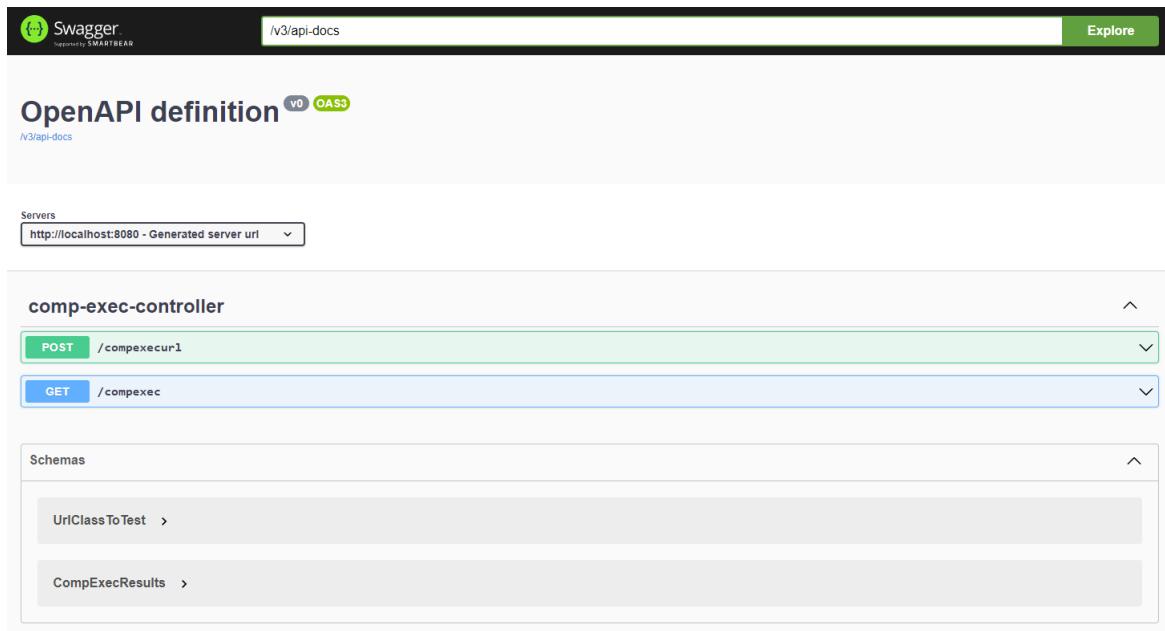


Figura 4.5: Documentazione delle API con Swagger

### 4.3 Testing

Prima di costruire la nostra applicazione nella sua interezza abbiamo testato, tramite test funzionali, le singole unità per verificarne il corretto funzionamento. Abbiamo poi testato l'integrazione delle unità per verificare che le interazioni si svolgessero in modo corretto. Infine, avendo ottenuto l'applicazione nella sua interezza abbiamo deciso di testarne il funzionamento riportando anche i tempi di risposta.

Tutti i test sono stati effettuati utilizzando una macchina con processore un i7 9700k, 32 gb di

## CAPITOLO 4. IMPLEMENTAZIONE E TESTING

**Parameters**

Name	Description
ClassName * required	string (query)
StudentLogin * required	string (query)
GameId * required	string (query)

**Responses**

Code	Description	Links
200	OK	No links

Media type: application/json

Example Value | Schema

```
{
  "idResult": 0,
  "resMessage": "string",
  "pathCoverage": "string"
}
```

Figura 4.6: Documentazione delle API con Swagger

**Parameters**

Name	Description
ClassName * required	string (query)
StudentLogin * required	string (query)
GameId * required	string (query)

**Request body** required

Media type: application/json

Example Value | Schema

```
{
  "urlClass": "string",
  "urlTestClass": "string"
}
```

**Responses**

Code	Description	Links
200	OK	No links

Media type: application/json

Example Value | Schema

```
{
  "idResult": 0,
  "resMessage": "string",
  "pathCoverage": "string"
}
```

Figura 4.7: Documentazione delle API con Swagger



Figura 4.8: Documentazione delle API con Swagger

Classe da testare	Classe di test	Errori attesi	idResult	Tempo
Calcolatrice	calcTest	Nessuno	0	2.98 s
Calendario	TestCalendario	Nessuno	0	3.25 s
Calendario1	TestCalendario	Err compilazione	1	1.2 s
SubjectParser	TestSubjectParser	Err test	2	3.22 s
TimeStamp	TestTimeStamp	Nessuno	0	3.15 s
XMLParser	TestXMLLoader	Nessuno	0	3.54 s

Tabella 4.1: Test con classi diverse

ram con Ubuntu 20.04s come sistema operativo. È stato utilizzato Postman per fare le richieste HTTP al nostro servizio in esecuzione su un container docker.

Notiamo che utilizzando questa test breve suite il nostro servizio si comporta come dovrebbe garantendo tempi di risposta sempre al di sotto dei 3 secondi, quindi garantisce in genere buoni tempi di risposta.

Abbiamo inoltre testato il nostro servizio in caso di richieste multiple simultanee. Abbiamo notato come il servizio riuscisse a rispondere in modo corretto a tutte le richieste fatte, ma i tempi di risposta del server si allungassero. In particolare abbiamo notato come il tempo di risposta fosse direttamente proporzionale al numero di richieste simultanee effettuate. Di seguito i tempi medi di risposta ad un bulk di richieste simultanee di dimensione crescente.

Dimensione del bulk di richieste	Tempo medio di risposta per richiesta
1	3.4 s
2	4.95 s
3	6.5 s
4	8.4 s
5	9.9 s
6	11.7 s

Tabella 4.2: Test con richieste simultanee

## 4.4 Stima dei Costi

- Tabella di riferimento per le complessità di dati e transazioni

	SEMPLICE	MEDIO	COMPLESSO
<b>NILF</b>	7	10	15
<b>NEIF</b>	5	7	10
<b>NEI</b>	3	4	6
<b>NEO</b>	4	5	7
<b>NEQ</b>	4	4	6

CompileandRunTests

	VALORE	SEMPLICE	MEDIO	COMPLESSO	TOT
<b>NILF</b>	0				
<b>NEIF</b>	0				
<b>NEI</b>	5	3	4		17
<b>NEO</b>	3	4	5		13
<b>NEQ</b>	0				

NEI : ClassName, GameId, StudentLogin, UrlClasse,UrlClasseTest [3 semplici , 2 medi]  
 NEO : ResultId, resMessage ,pathJacoco [2 semplici , 1 medio]

Fattori Correttivi

<b>COMUNICAZIONE DATI</b>	4
<b>DISTRIBUZIONE ELABORAZIONE</b>	0
<b>PRESTAZIONI</b>	1
<b>UTILIZZO INTENSIVO CONFIGURAZIONE</b>	1
<b>FREQUENZA DELLE TRANSAZIONI</b>	2
<b>INSERIMENTO DATI INTERATTIVO</b>	0
<b>EFFICIENZA PER L'UTENTE FINALE</b>	3
<b>AGGIORNAMENTO INTERATTIVO</b>	0
<b>COMPLESSITÀ ELABORATIVA</b>	0
<b>RIUSABILITÀ</b>	3
<b>FACILITÀ INSTALLAZIONE</b>	3
<b>FACILITÀ GESTIONE OPERATIVA</b>	1
<b>Molteplicità DI SITI</b>	0
<b>FACILITÀ DI MODIFICA</b>	1

19

Stima finale

---

## CAPITOLO 4. IMPLEMENTAZIONE E TESTING

UFP	LLOC/FP
30	1590
FP	LLOC/FP
25,2	1336

## Capitolo 5

# Guida all'installazione

Come già accennato in precedenza abbiamo utilizzato docker per fare il deploy del nostro servizio.

Grazie a questa scelta l'installazione del nostro servizio richiede ben pochi requisiti, che sono:

1. Un'installazione di docker sull'host su cui si vuole installare il servizio, riferirsi ai requisiti di sistema indicati da docker;
2. Una directory con la struttura del FileSystem condiviso con la struttura di figura 4.1b.

Ottenute entrambe queste cose si può procedere all'installazione vera e propria. Basta scaricare la repository del progetto, posizionarsi nella cartella ed eseguire il comando:

```
$ docker build . -t -name-
```

sostituendo a -name- il nome che si vuole dare al container. Una volta che l'immagine di base viene correttamente scaricata e che il container viene buildato basta eseguire il seguente comando per mettere in esecuzione il nostro servizio:

```
$ docker run -p XXXX:8080 -v /pathToFilesystem/:/FileSystem/ -name-
```

sostituendo a XXXX il porto su cui si vuole che il servizio sia in ascolto, a /pathToFilesystem/ il path sull'host della directory richiesta al punto 2 e a -name- il nome assegnato al container al passo precedente.

Fatto ciò il nostro servizio sarà in esecuzione ed in attesa di richieste HTTP del tipo descritto nella sezione 4.2.