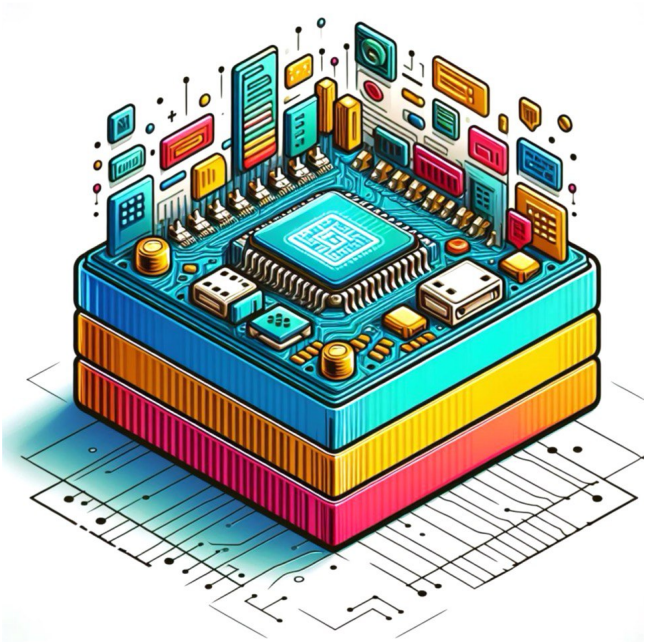
Tesina Software Testing



Nome Cognome	Matricola
Stefano Marano	M63001428
Vito Romano	M63001504
Paolo Russo	M63001426
Marco Cimmino	M63001528

Indice

1 | Introduzione

Questo elaborato presenta i risultati di un'analisi approfondita e di una serie di test condotti sui task T56 e T7 del progetto 'Testing Game'.
L'obiettivo primario del nostro lavoro di testing è stato quello di verificare la robustezza e il rispettare i requisiti specificati di queste due componenti fondamentali dell'applicazione.
Per questi motivi si è deciso di creare una suite di test modulare, affidabile e facilmente estendibile, in grado di garantire che l'applicazione risponda correttamente alle specifiche funzionali e non funzionali.
Questi approcci hanno reso possibile una copertura efficace delle funzionalità chiave del sistema, combinando test unitari e test di integrazione. Sono stati sviluppati 130 casi di test, che coprono in media tra l'80% e il 90% delle istruzioni, dei branch e delle funzioni del codice.
Il presente elaborato descrive in dettaglio le fasi di progettazione e esecuzione dei test, i risultati ottenuti e le eventuali problematiche riscontrate.

1.1 | Testing Game

Testing Game è un progetto sviluppato, in parte, dagli studenti della Federico II del corso di Software Architecture Design, che attraverso la strategia della gamification vuole valorizzare l'importanza del testing.
Il risultato dell'applicazione di tale meccanismo è stato la progettazione e conseguente sviluppo di una piattaforma di gioco che vede gli studenti competere, attraverso varie modalità di gioco, a colpi di test progettati mediante il framework JUnit, contro dei robot (per ora Randoop e EvoSuite) capaci di generare automaticamente tali test;

Il task T56 è il responsabile della gestione delle interfacce utente e del motore di gioco, è stato sottoposto a un'analisi focalizzata sui requisiti funzionali, al fine di garantire che tutte le funzionalità previste siano implementate correttamente, in modo che l'esperienza utente sia fluida. Data la sua natura, l'analisi dei requisiti funzionali del task T56 ha incluso test white-box granulari (metodo e, in alcuni casi, classe) per valutare la logica interna e test di integrazione top-down per assicurare la coesione dei componenti.

il task T7 fornisce un servizio di compilazione per file Java utilizzando Maven e JaCoCo tramite un'API RESTful, è stato valutato sia sotto l'aspetto funzionale che non funzionale. In particolare, l'obiettivo è stato verificare la conformità del servizio alle specifiche definite e la sua capacità di gestire correttamente le richieste di compilazione. Sono stati progettati casi di test che simulavano diverse situazioni, tra cui input validi, invalidi e malformati, al fine di verificare la robustezza del servizio di fronte a errori e utilizzo improprio. Inoltre, sono stati realizzati test per verificare il corretto funzionamento in un contesto concorrente, quindi in presenza di più thread, ed anche alcuni test per verificare la robustezza, in questo caso prevedendo alcune condizioni in cui si doveva trovare il filesystem.

1.2 | Approccio Scelto

Per raggiungere gli obiettivi prefissati, è stata adottata una strategia di testing principalmente di tipo white-box, che ha permesso di analizzare approfonditamente il codice sorgente e identificare eventuali difetti.
Le metriche di copertura del codice, come statements, branch e function, sono state utilizzate per valutare la completezza dei test e individuare le aree del codice meno esplorate dai test stessi.

Strumenti utilizzati Abbiamo deciso di implementare una suite di test automatizzati utilizzando le potenzialità offerte da JUnit 5 e il sistema di build Maven.
L'utilizzo di Maven ha semplificato la gestione delle dipendenze, l'esecuzione dei test e la generazione di report, integrandosi in modo fluido con il flusso di lavoro del progetto. Inoltre, essendo già ampiamente supportato e adottato nel progetto, si è rivelato la scelta più naturale per soddisfare le esigenze di automazione dei test.
La strategia di testing adottata si basa sull'uso di annotazioni come @SpringBootTest e @WebMvcTest, in particolare:

- **@SpringBootTest**: permette di caricare l'intero contesto dell'applicazione per verificare l'interazione tra i diversi componenti, garantendo che il sistema funzioni correttamente come un'unità. Necessario il suo utilizzo per poter testare quelle classi che fanno uso del RestTemplate.
- **@WebMvcTest**: è stata utilizzata per testare i controller in isolamento, simulando le richieste HTTP e verificando le risposte generate, senza caricare l'intero contesto Spring.

Ciclo di Build di Maven Maven utilizza una convenzione su configurazione (Convention over Configuration), il che significa che segue una struttura di directory predefinita. Lo sviluppatore dovrebbe rispettare questa struttura per evitare di configurare percorsi personalizzati.
Di seguito viene mostrata la struttura di un tipico progetto Maven: la directory *src/main/java* contiene il codice sorgente dell'applicazione, mentre la directory *src/test/java* ospita i test.

```
project/
├── pom.xml                                          (file di configurazione Maven)
└── src/
    ├── main/                                        (codice sorgente dell'applicazione)
    │   ├── java/
    │   │   └── com/
    │   │       └── example/
    │   │           └── App.java                     (codice sorgente classe)
    │   └── resources/
    │       └── application.properties               (file di configurazione)
    └── test/                                                (codice di test)
        ├── java/
        │   └── com/
        │       └── example/
        │           └── AppTest.java                 (test unitario della classe principale)
        └── resources/                                (risorse per i test, opzionale)
```

Il cuore di un progetto Maven è il file *pom.xml*, che definisce le dipendenze necessarie, i plugin da utilizzare e, eventualmente, il ciclo di vita del progetto.
Altro vantaggio è che Maven include una fase del ciclo di build dedicata ai test. Se un test fallisce, Maven interrompe il processo e non prosegue con la creazione del pacchetto.
In particolare, Maven organizza il ciclo di build in fasi principali, tra cui:

1. **compile**: Compila il codice sorgente nella directory *src/main/java*.
2. **test**: Compila ed esegue i test presenti nella directory *src/test/java* usando un framework di test come JUnit.
3. **package**: Prepara l'output, ad esempio un file JAR o WAR.
4. **install**: Installa l'artefatto nel repository Maven locale.
5. **deploy**: Pubblica l'artefatto in un repository remoto.

In questo modo, lo sviluppatore deve solo conoscere i comandi principali di Maven per eseguire build, test e altre operazioni. Comandi principali:

- **mvn clean**: Garantisce che ogni build inizi in un ambiente pulito, riducendo la possibilità di errori causati da artefatti obsoleti o residui.
- **mvn compile**: Compila il codice sorgente.
- **mvn test**: Esegue i test.
- **mvn package**: Compila il codice e genera un artefatto (es. un file JAR o WAR).
- **mvn install**: Installa l'artefatto generato nel repository locale.

Ogni comando è legato a una precisa fase del ciclio di vita, quindi ad es. *mvn clean install*, uno dei comandi più utilizzati nello sviluppo software, è una combinazione di due comandi separati: mvn clean e mvn install, eseguiti in sequenza.
Nell'ambito CI/CD, quest'automazione dei test con Maven offre prevenzione degli errori in produzione: l'integrazione dei test durante ogni build riduce il rischio di introdurre bug o regressioni nel codice distribuito.

```
# Esegui Tutti i Test
mvn test
# Esegui Test di una Classe Specifica
mvn -Dtest=NomeClasseTest test
# Esegui Test Specifici, in questo caso
# un singolo metodo specifico in una classe
mvn -Dtest=NomeClasseTest#metodoSpecifico test
```

Listing 1: Comandi per eseguire i test

Nello specifico del caso i comandi qui mostrati posso essere utilizzati in due precise directory:

- *A13/T7-G31/RemoteCCC/* per quanto riguarda lo sviluppo dell task T7.
- *A13/T5-G2/t5/* per quanto riguarda lo sviluppo del task T5.

Test in IDE Esploreremo i passaggi necessari per configurare Visual Studio Code (VS Code) come ambiente di sviluppo per eseguire test utilizzando JUnit.

1. **Installare l'estensione "Test Runner for Java"**

   [a] Apri VS Code e vai al Marketplace delle Estensioni (icona delle estensioni nel pannello laterale o Ctrl+Shift+X).

   [b] Cerca "Test Runner for Java" e installa l'estensione.

   Questa estensione consente di rilevare, eseguire e monitorare i test JUnit in modo semplice. Installa anche l'estensione "Java Extension Pack" se non già presente. Include strumenti essenziali per lavorare con Java, come la gestione di progetti Maven e Gradle.

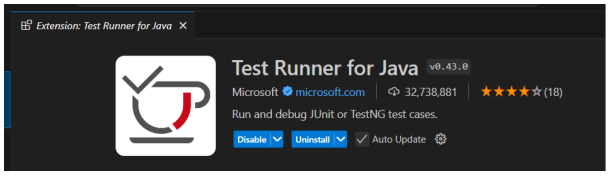


Figura 1.1: Estensione Test Runner for Java per eseguire i test Junit 4 e 5.

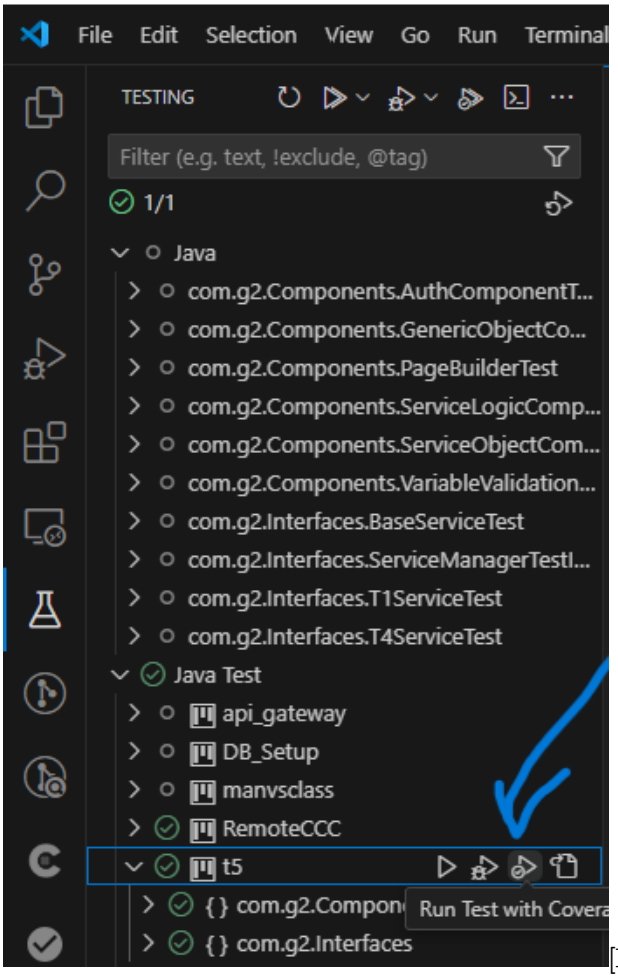
2. Scrivere i test rispettando i path di Maven.
   - **Sorgente del codice**: *src/main/java*
   - **Sorgente dei test**: *src/test/java*

```
package com.example;

import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

public class ExampleTest {
    @Test
    public void testAddition() {
        int result = 2 + 3;
        assertEquals(5, result, "La somma non     corretta");
    }
}
```

Listing 2: Esempio di Classe di Test con Juni5



[H]

Figura 1.2: Pannello di Test Explorer con opzioni di esecuzione.

3. **Eseguire i test** (singoli o di gruppo) Dopo aver scritto i test, VS Code rileverà automaticamente le classi di test. Nella barra laterale del Test Explorer (pannello Test, visibile grazie a "Test Runner for Java"), vedrai l'elenco dei test disponibili. Puoi eseguire i test in vari modi:
   - **Singolo test**: Clicca sull'icona di esecuzione accanto al nome del test nella classe direttamente nel Test Explorer.
   - **Gruppo di test**: Esegui tutti i test di una classe o un pacchetto selezionando l'opzione corrispondente nel Test Explorer.
   - **Esecuzione dall'editor**: Clicca sull'icona accanto al metodo di test nel file sorgente.

4. **Valutare la copertura del codice** (Code Coverage). Per valutare quanto del codice sorgente è stato coperto dai test, segui questi passaggi:

   [a] **Installa** uno strumento come JaCoCo (Java Code Coverage) nel progetto Maven, basta aggiungere il plugin al file pom.xml.

   [b] **Generare** il report, per farlo puoi utilizzare:

   - Il comando *mvn test jacoco:report*, il quale genera un report HTML generato da JaCoCo presente in *target/site/jacoco/index.html*.
   - Clicca sull'icona di esecuzione relativa. In questo caso, nell'editor avremo i gutters colorati indicanti la coverage (coperto, non coperto, parzialmente coperto) e relative icone sulla COV% all'interno del file explorer.

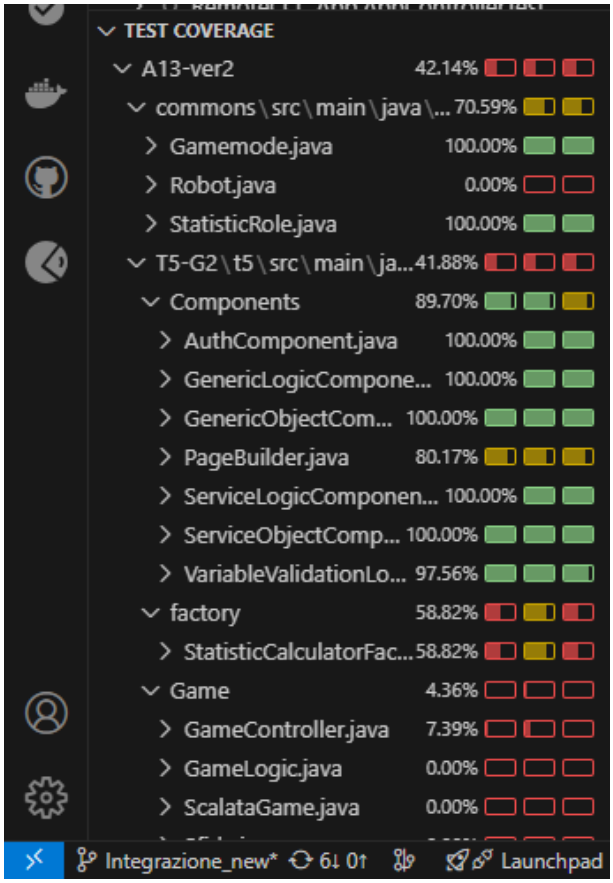


Figura 1.3: Pannello della Test Coverage.

2 | Task T56

Il task T56 è un componente fondamentale all'interno del progetto del gioco. Ha la responsabilità primaria di gestire le varie pagine web e coordinare tutte le interazioni tra il giocatore e il gioco stesso, garantendo un'esperienza utente fluida e coinvolgente.

2.1 | Struttura e Funzionalità

Per svolgere questo compito complesso, il task T56 è organizzato in tre pacchetti ben definiti:

- **Interface**: si occupa della gestione delle comunicazioni con i servizi REST, incapsulando la logica di rete per garantire un'astrazione completa. Questo pacchetto include un Service Manager, uno strumento che semplifica notevolmente:
  - l'aggiunta e la configurazione di nuovi task nel Front-end;
  - l'integrazione di nuove chiamate REST verso servizi esistenti;
  - la creazione di combinazioni flessibili e personalizzate di servizi disponibili.

  Il dispatcher centralizzato garantisce una gestione uniforme delle richieste e agevola l'espandibilità del sistema.
- **Component**: si occupa della costruzione delle pagine web, basate su componenti modulari che separano chiaramente la logica di business dai dati da visualizzare. I componenti sono disponibili in versioni generiche e integrate con il Service Manager per facilitare l'interoperabilità tra i vari task. In questo modo, è possibile definire una pagina come un insieme di componenti logici (che quindi rappresentano requisiti, prerequisiti o controlli di varia natura) e di componenti oggetto, che rappresentano i dati da inserire nella vista.
- **Game**: gestisce la logica di gioco, con un controller dedicato alla gestione della concorrenza e delle partite attive. Fornisce una classe GameLogic che consente la definizione delle diverse modalità di gioco.

Un'attenzione particolare è stata dedicata al testing approfondito dei pacchetti Interface e Component, poiché entrambi sono stati progettati per un utilizzo estensivo all'interno del progetto. Il pacchetto Game, invece, è ancora in fase di sviluppo e soggetto a modifiche, motivo per cui i test su di esso sono stati finora limitati.

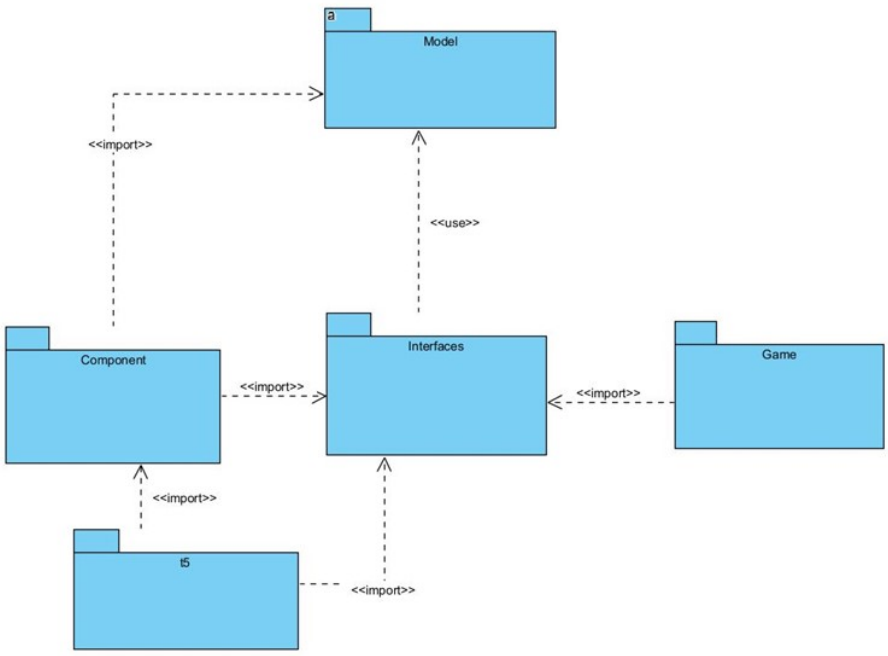


Figura 2.1: Package Diagramm Task56

2.2 | Package Interface

Nel package interface è stato implementato un dispatcher di servizi REST, denominato ServiceManager, che gestisce i vari servizi definiti estendendo la classe BaseService. In particolare:

- **BaseService**: È un'implementazione astratta che serve da base per i servizi che comunicano con API REST, offrendo un'infrastruttura condivisa per le operazioni e la gestione delle chiamate HTTP. Si basa sul **RestTemplate**, una classe fornita dal framework Spring, progettata per semplificare la comunicazione tra un'applicazione e un servizio RESTful. Offre i seguenti meccanismi:
  - La possibilità di registrare le azioni disponibili del servizio tramite il metodo **registerAction**().
  - Per gestire le richieste, implementa il metodo **handleRequest**(), che verifica l'esistenza dell'azione richiesta ed esegue le relative operazioni.

    Comprende metodi per effettuare chiamate HTTP di tipo GET, POST, PUT e DELETE, occupandosi sia della costruzione degli URI che dell'elaborazione delle risposte.

    Inoltre, gestisce le eccezioni che possono sorgere durante le chiamate REST, fornendo messaggi di errore chiari e comprensibili.
- **ServiceActionDefinition**: Rappresenta una classe immutabile che definisce le azioni, incapsulando una funzione e le specifiche dei parametri da utilizzare, un azione è sostanzialmente un operazione offerta da un microservizio attraverso un endpoint.
- **ServiceInterface**: Interfaccia che tutti i servizi devono implementare per essere incapsulati in un'unica classe per utilizzare il dispatcher.
- **ServiceManager**: Funge da gestore centrale per i servizi, consentendo la registrazione e la gestione delle istanze di servizi che implementano ServiceInterface tramite il metodo **registerService**().

  Mentre, per la gestione delle richieste, implementa il metodo **handleRequest**() per verificare l'esistenza del servizio richiesto e instrada la richiesta all'azione appropriata, occupandosi della gestione delle eccezioni.
- **T1Service**, **T23Service**, **T4Service**, **T7Service** rappresentano le interfacce per comunicare con i relativi task.

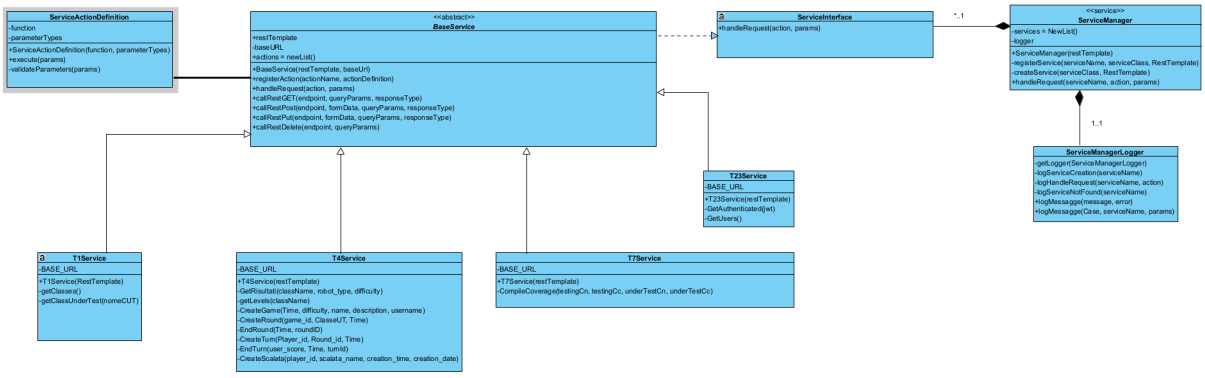


Figura 2.2: Class Diagramm del package Interfaces.

Classe di Test	Classe Testata	COV-Statement	COV-Branch	COV-functions
BaseServiceTest	BaseService	80%	70%	100%
ServiceManagerTest	ServiceManager	100%	92%	100%
ServiceManagerTestIntegration	ServiceManager	100%	92.86%	100%
	T1Service	87.5%	None	100%
	T23Service	100%	100%	100%
	T4Service	87.6%	68%	100%
	T7Service	25%	25%	33%
T1ServiceTest	T1Service	87%	None	100%
T23ServiceTest	T23Service	100%	100%	100%
T4ServiceTest	T4Service	88%	100%	100%
T7ServiceTest	T7Service	25%	25%	100%

Tabella 2.1: Classi di test per package Interfaces di T56.

2.2.1 | BaseService

Tutti i servizi devono estendere la classe **BaseService**, una classe astratta progettata per semplificare l'implementazione di un servizio reale, offre numerosi metodi per le chiamate REST e funzionalità di utilità, rendendola un elemento centrale del sistema.
Dato il suo ruolo cruciale e l'ampio utilizzo, è stata dedicata particolare attenzione al suo testing: per questo è stata sviluppata una classe di test denominata **BaseServiceImpl**, progettata specificamente per verificare il comportamento di BaseService.
Questa classe, che implementa la classe astratta, è configurata per essere utilizzata esclusivamente durante la fase di testing grazie al meccanismo dei profili.
BaseServiceImpl simula il comportamento di un servizio reale implementando una serie di azioni che rappresentano scenari di test o strumenti per verificare i meccanismi interni, queste implementazioni permettono di effettuare test mirati e approfonditi.
L'obiettivo principale è testare la capacità di BaseService di creare, gestire e organizzare in modo efficace le richieste HTTP. In particolare, viene posta attenzione sull'utilizzo dell'oggetto RestTemplate, elemento fondamentale per l'elaborazione delle chiamate REST. Per questo motivo, il testing si concentra su test di integrazione di tipo white-box, analizzando dettagliatamente il funzionamento interno della classe.

Profili di Test Il meccanismo dei profili di test in Maven consente di eseguire diverse configurazioni o set di test in base al contesto o all'ambiente. Ad esempio, puoi configurare un profilo per eseguire solo test unitari, un altro per i test di integrazione o per ambienti specifici (come sviluppo, staging o produzione). I profili si definiscono nel file pom.xml e possono essere attivati in base a criteri specifici, nel nostro caso è stato usato il profilo di default già attivo in automatico durante la base di testing di Maven.
Vantaggi dei Profili di Test

- **Flessibilità**: Permettono di separare e gestire diversi tipi di test (unitari, integrazione, carico).
- **Contesto Specifico**: Puoi adattare i test all'ambiente o alle necessità di build.
- **Automazione CI/CD**: Utili per eseguire set di test specifici nei pipeline di Continuous Integration.

Mock RestService Server è uno strumento utilizzato per simulare le risposte di un servizio REST senza dover interagire con un servizio reale.
Questo è particolarmente utile nei test di applicazioni che consumano API REST, permettendo di verificare il comportamento dell'applicazione client in modo isolato e controllato; infatti, evita dipendenze da servizi reali, riducendo i problemi legati alla disponibilità o alla configurazione del servizio remoto. Inoltre evita di inviare dati sensibili o critici a un servizio esterno durante i test.
Nel nostro contesto, sarebbe stato possibile effettuare un mock dell'oggetto RestTemplate. Tuttavia, questa strategia avrebbe testato solo l'interazione con il metodo chiamato (ad esempio, getForObject), senza simulare un vero flusso HTTP. Questo approccio limita il contesto del protocollo HTTP, rendendo impossibile validare aspetti come la struttura delle richieste inviate o l'analisi delle risposte ricevute.
MockRestServiceServer permette di simulare scenari più completi e vicini alla realtà, in particolare:

- Intercetta le richieste HTTP effettuate tramite il RestTemplate e restituisce risposte simulate, replicando uno scenario reale.

- Consente di testare non solo la chiamata al metodo, ma anche l'interazione completa con l'endpoint remoto, consente di verificare:
  - Che l'URL richiesto sia corretto.
  - Che il metodo HTTP utilizzato sia appropriato (GET, POST, ecc.).
  - Che le intestazioni e il corpo della richiesta siano conformi.
  - Offre un meccanismo nativo per confrontare le richieste effettuate dall'applicazione con le aspettative definite.

Inoltre, MockRestServiceServer offre un meccanismo nativo per confrontare le richieste effettuate dall'applicazione con le aspettative definite.
Questo strumento consente una validazione più precisa del contenuto delle richieste, superando le limitazioni del semplice mock di RestTemplate, dove tali controlli non sono possibili data la natura del mock stesso. Grazie a queste caratteristiche, si rivela particolarmente adatto per i test di integrazione, in questo modo permette di simulare un'interazione reale con un servizio remoto, consente di verificare non solo il corretto funzionamento delle singole chiamate, ma anche l'interazione tra i componenti dell'applicazione client e il protocollo HTTP.

```
@Autowired
private RestTemplate restTemplate;
private BaseServiceImpl baseService;
private MockRestServiceServer mockServer;
private String Base_URL = "http://mock_url:123";

@BeforeEach
public void setUp() {
    mockServer = MockRestServiceServer.createServer(restTemplate);
    baseService = new BaseServiceImpl(restTemplate);
}
```

Listing 3: Setup dei test con Mock Rest Service Server

Questo approccio ha reso possibile una campagna di testing di integrazione, focalizzata sui seguenti obiettivi principali

- **Testing negativo dell'interfaccia "handleRequest"**: Sono stati realizzati test per verificare la robustezza dell'interfaccia chiamando azioni non esistenti o utilizzando parametri malformati.
- **Verifica della gestione di endpoint e query parameter delle chiamate REST**: Questi test verificano che la costruzione e l'elaborazione delle chiamate REST siano corrette e aderenti alle specifiche.
- **Gestione degli errori di rete**: Sono stati simulati scenari di errori di rete, come le risposte HTTP 501 (Not Implemented) e 504 (Gateway Timeout), per testare il comportamento e l'affidabilità del servizio in condizioni avverse.
- **Testing dei metodi di supporto**: Particolare attenzione è stata data ai metodi di supporto forniti dalla classe, come quelli utilizzati per conversioni o operazioni accessorie, per garantirne la correttezza.

Grazie a questa configurazione, è stato possibile testare l'interfaccia HandleRequest, responsabile dell'esecuzione di specifiche azioni con parametri, iniettando azioni personalizzate per sollecitare determinati metodi o porzioni di metodi della BaseService. Questo approccio ha garantito un'ampia copertura (sia di tipo statements che branch) e una maggiore affidabilità del sistema.

```
/*
*  T2 - Eseguo una Get con parametri
*/
@Test
public void testGetWithParameters() {
    // Definisci il parametro di test
    String testParam = "123";
    // Risposta simulata
    String expectedResponse = "Success";
    // Costruisci l'endpoint atteso
    String endpoint = Base_URL + "/TestGetParams?Test_Query_Params=123";
    // Aspettati una chiamata GET e restituisci una risposta simulata
    mockServer.expect(requestTo(endpoint))
              .andExpect(method(HttpMethod.GET))
              .andRespond(withSuccess(expectedResponse, MediaType.TEXT_PLAIN));

    // Chiama il metodo da testare
    String response = (String) baseService.handleRequest("TestGetParams", testParam);
    // Verifica la risposta
    assertEquals(expectedResponse, response);
    // Verifica che il server simulato abbia ricevuto la richiesta
    mockServer.verify();
}
```

Listing 4: Esempio di Test usando il Mock.

2.2.2 | Service Manager

Per verificare il corretto funzionamento del Service Manager, sono state implementate diverse tipologie di test, ognuna mirata a validare specifici aspetti del sistema:

- **Test unitari a livello di metodo**: Questi test verificano la corretta creazione e registrazione dei servizi, includendo anche scenari negativi.

  Per far ciò è stato esposto con una apposita classe Stub, essendo privato, il metodo registerService e sono stati utilizzati mock per simulare servizi malformati. Ad esempio, sono stati testati casi di servizi non validi, come l'uso di un ID errato. L'obiettivo è verificare che il sistema sia in grado di gestire correttamente tali situazioni e che risponda in modo robusto agli errori.

- **Test di integrazione**: Questi test utilizzano le implementazioni reali dei servizi e del RestTemplate, ma è stato simulato tramite mock il server remoto per generare diversi tipi di situazioni.

  Questa configurazione ha permesso di validare non solo la capacità del Service Manager di gestire correttamente gli oggetti dei servizi, ma anche la sua abilità nel trattare adeguatamente le eccezioni generate durante le operazioni.

2.2.3 | T1Service, T23Service, T4Service, T7Service

In questo caso, l'obiettivo è verificare che ciascuna di queste classi implementi correttamente il comportamento atteso, attraverso test unitari a livello di metodo.
Poiché estendono tutte la BaseService, e quest'ultima è già stata ampiamente testata, i test specifici per ciascuna di queste classi si concentrano principalmente sulle azioni individuali registrate in ogni servizio, ma anche che l'interazione tra i vari metodi della BaseService.
Anche in questo caso avviene il mocking delle risposte: i test usano MockRestServiceServer per simulare le risposte del server, senza fare richieste reali a un server esterno.
Ogni test verifica che il servizio gestisca correttamente le risposte, che siano esse di successo, vuote, o di errore, e che il comportamento del sistema sia come previsto.
In questo modo, si cerca di validare che ogni servizio gestisca correttamente il flusso delle informazioni e rispetti le logiche di business definite.

```
/*
* Test3: testGetClassUnderTest
* Precondizioni: Il server mock    impostato per restituire un contenuto byte
* specifico per la classe richiesta.
* Azioni: Invocare handleRequest per la classe specifica.
* Post-condizioni: Verificare che il risultato corrisponda al contenuto atteso.
*/
@Test
public void testGetClassUnderTest() {
    String nomeCUT = "Calcolatrice";
    String endpoint = "/downloadFile/" + nomeCUT;
    String Base_URL_t = Base_URL + endpoint;

    String fileContent = "public class Calcolatrice {}";
    byte[] byteArray = fileContent.getBytes(StandardCharsets.UTF_8);

    mockServer.expect(once(), requestTo(Base_URL_t))
            .andExpect(method(HttpMethod.GET))
            .andRespond(withSuccess(byteArray, MediaType.APPLICATION_OCTET_STREAM));

    String result = (String) T1Service.handleRequest("getClassUnderTest", nomeCUT);
    assertNotNull(result);
    // Verifica che il risultato sia quello atteso
    assertEquals(fileContent, result);
    // Verifica che il server mock abbia ricevuto la richiesta
    mockServer.verify();
}
```

Listing 5: Esempio di Test sviluppato per T1Service

2.3 | Package Component

Ogni componente all'interno del package Component rappresenta una parte della pagina, con responsabilità ben definite e facilmente estendibili. La logica di ciascun componente è separata dal rendering della vista, permettendo un'integrazione efficace con il sistema di templating (es. Thymeleaf).
il **PageBuilder** è lo strumento principale, progettato per semplificare la creazione e la gestione delle pagine web in un'applicazione Java Spring. Questa componente funge da ponte tra i vari componenti dell'interfaccia utente e i servizi backend, permettendo una costruzione dinamica e modulare delle pagine. Grazie alla sua architettura flessibile, PageBuilder consente agli sviluppatori di progettare esperienze utente interattive e reattive in modo efficiente. Le caratteristiche principali sono:

- **Struttura Dinamica delle Pagine**: PageBuilder consente di assemblare pagine web in modo dinamico, permettendo l'aggiunta di componenti in base alle necessità della speficifica situazione in cui si trova la pagina.
- **Integrazione con Componenti**: Supporta l'integrazione di diversi componenti, sia per la visualizzazione dei dati che per la logica di business, facilitando l'organizzazione del codice.
- **Utilizzo di Thymeleaf**: Sfrutta il motore di template Thymeleaf per generare HTML in modo efficiente, garantendo che le pagine siano ottimizzate per l'interazione con i dati provenienti dai servizi.
- **Gestione dei Prerequisiti**: Permette di definire prerequisiti per le pagine, assicurando che tutte le condizioni necessarie siano soddisfatte prima di caricare il contenuto, migliorando così l'esperienza dell'utente.

2.3.1 | Struttura del package

PageBuilder è il fulcro del sistema di costruzione dinamica delle pagine. È stato progettato per combinare logic componets e object componets, rispettivamente dati e logica di business della pagina, consentendo

un'elevata flessibilità e modularità. Si occupa della gestione e del rendering delle pagine, orchestrando l'esecuzione di vari componenti e gestendo eventuali errori che possono sorgere durante il processo.
GenericObjectComponent funge da classe base per la gestione degli oggetti destinati al modello della pagina. Inserimento dati nel modello ,sfruttando una mappa chiamata Model per contenere oggetti associati a chiavi specifiche. Questi oggetti possono poi essere passati al template per il rendering della pagina Consente l'estensione da parte di componenti oggettuali specifici, come il ServiceObjectComponent, per implementare logiche più complesse.
ServiceObjectComponent estende il GenericObjectComponent ed è specializzato nel recupero di dati da servizi esterni.Il recupero dinamico dei dati avviene tramite il ServiceManager, che effettua richieste verso servizi esterni per ottenere le informazioni necessarie al modello.Questi dati vengono poi inseriti nel modello utilizzando una chiave specifica, permettendo una corretta visualizzazione delle informazioni nella pagina
GenericLogicComponent è una classe astratta che stabilisce la struttura per i componenti logici nel sistema. Pertanto, il componente offre due funzionalità principali: il metodo executeLogic(), che si occupa di eseguire la logica del componente restituendo true in caso di successo o false in caso di errore, e il metodo getErrorCode(), il quale fornisce un codice di errore utile per identificare eventuali problemi, come un'autenticazione fallita o la mancanza di dati necessari.
AuthComponent è un'estensione di ServiceLogicComponent e si occupa della logica di autenticazione degli utenti. Per verificare l'autenticazione, utilizza il ServiceManager inviando una richiesta al servizio per controllare se l'utente dispone delle autorizzazioni necessarie. In caso di autenticazione fallita, il componente restituisce un codice di errore specifico (come Auth_error), che può essere utilizzato per reindirizzare l'utente a una pagina di login.

Classe di Test	Classe Testata	COV- Statement	COV- Branch	COV - functions
PageBuilderTest	PageBuilder	80%	80%	80,77%
GenericObject ComponentTest	GenericObjectComponent	100%	100%	100%
GenericLogicComponentTest	GenericLogicComponent	100%	None	100%
ServiceLogicComponentTest	ServiceLogicComponent	100%	None	100%
ServiceObjectComponentTest	ServiceObjectComponent	100%	100%	100%
VariableValidation LogicComponentTest	VariableValidation LogicComponent	100%	90%	100%

Tabella 2.2: Classi di Test per il package Component di T56

2.3.2 | Page Builder

Il testing del componente PageBuilder si concentra su due livelli principali: test di unità a livello di metodo e di classe. Questi approcci permettono di verificare rispettivamente la robustezza delle singole funzionalità e l'efficacia dei meccanismi di gestione degli errori e della creazione delle pagine.

Test di Unità a Livello di Metodo Questi test di unità sono stati progettati per valutare l'affidabilità dei metodi individuali del PageBuilder. L'obiettivo principale è verificare che ogni metodo funzioni correttamente in isolamento, affrontando scenari specifici e limitati. Ad esempio:

- Validazione della corretta inizializzazione di un componente della pagina.
- Verifica della manipolazione dei dati e del corretto passaggio di informazioni.
- Controllo del comportamento in presenza di parametri non validi o nulli.

Test di Unità a Livello di Classe I test di classe si concentrano sulla funzionalità complessiva del PageBuilder, simulando scenari in cui vengono aggiunti uno o più componenti alla pagina.
Questi test valutano un flusso di operazioni su un istanza dell'oggetto, come ad esempio:

- L'interazione tra diversi componenti aggiunti al page builder.
- La gestione degli errori durante l'esecuzione di componenti
- una corretta gestione del modello HTML.

Per isolare il PageBuilder e testarlo adeguatamente, sono stati creati **mock** per simulare le dipendenze:

1. **Mock del ServiceManager**: Realizzato da noi, simula le chiamate a servizi REST per il recupero dei dati. Permette di testare il comportamento del PageBuilder senza dipendere da servizi reali.
2. **Mock della classe Model**: La classe Model di Spring, utilizzata per passare dati ai template HTML, è stata simulata usando mockito ed è stata usata per verificare che i dati vengano istanziati e inseriti correttamente. Si è valutato, ad esempio, se l'aggiunta di un componente comporta l'aggiornamento del modello con le informazioni corrette.

2.3.3 | GenericObjectComponent

La classe GenericObjectComponent è responsabile della gestione di un modello interno rappresentato come una mappa (MapString, Object) che rappresenta una risorsa generica da inserire in un determinato punto di un modello HTML.
L'obiettivo principale dei test è validare che la classe gestisca correttamente la creazione e la manipolazione della risorsa, mantenendo un comportamento coerente anche in presenza di input particolari o situazioni limite.
Questi test si concentrano su due aspetti fondamentali. Da un lato, si valuta il comportamento dei metodi principali della classe, come **getModel** e **setObject**, assicurandosi che restituiscano i risultati attesi e che permettano di modificare il modello in modo appropriato.
Parallelamente, i test sui costruttori analizzano la capacità della classe di inizializzarsi correttamente in diversi scenari, come nel caso in cui vengano forniti parametri nulli o validi. In questi casi, si controlla che il modello interno sia coerente con le aspettative e che la classe non generi errori imprevisti.
Nel complesso, questi test mirano a verificare che la classe GenericObjectComponent sia robusta e affidabile, offrendo comportamenti prevedibili e adeguati sia in condizioni normali che in situazioni eccezionali. Questo approccio contribuisce a costruire una base solida per ulteriori integrazioni e utilizzi più complessi della classe all'interno del sistema.

2.3.4 | ServiceObjectComponent

ServiceObjectComponent è un'estensione di GenericObjectComponent, dove la risorsa da gestire proviene da un servizio REST e si ottiene tramite ServiceManager.
I casi di test pensati per esser d'unità e testare i metodi, sono stati sviluppati per coprire diversi aspetti del comportamento della classe, sia in scenari ordinari che in condizioni limite o di errore.
Un primo gruppo di test riguarda il metodo **getModel**, responsabile di restituire un modello popolato con i dati elaborati dal servizio. I test analizzano vari scenari come ad esempio, nel test testGetModelException si assicura che un'eccezione venga gestita in modo adeguato, restituendo un modello nullo.
Un secondo gruppo di test si concentra sugli aspetti più tecnici e accessori, come l'inizializzazione dei campi e la configurazione dinamica. Ad esempio, il test testServiceObjectComponentFieldsWithReflection utilizza la riflessione per accedere ai campi privati e verificare che i valori passati al costruttore siano stati inizializzati correttamente. Allo stesso modo, il test testGetSetModelKey verifica il corretto funzionamento dei metodi getModelKey e setModelKey, assicurandosi che la chiave di modello possa essere recuperata e aggiornata in modo coerente.

2.3.5 | ServiceLogicComponent

La classe ServiceLogicComponent rappresenta un modulo che implementa logica di business utilizzando l'interazione con altri task attraverso un gestore di servizi, il ServiceManager, anche in questo caso abbiamo usato il mock da noi definito. I test, principalmente di unità a livello di metodo, che sono stati descritti mirano a verificare il comportamento della classe in vari scenari, valutandone la robustezza e la capacità di gestire correttamente esecuzioni di successo, errori e condizioni limite.
Il primo aspetto testato riguarda il metodo **executeLogic**, che è al centro delle operazioni di ServiceLogicComponent e se ne valida il comportamento.
Un altro aspetto cruciale analizzato riguarda la gestione degli errori tramite il metodo setErrorCode e il relativo getter getErrorCode. Il test dedicato verifica che sia possibile impostare e recuperare correttamente il codice di errore, confermando che l'attributo venga aggiornato come previsto.

2.3.6 | VariableValidationLogicComponent

La classe VariableValidationLogicComponent è progettata per verificare la validità di una variabile sulla base di criteri predefiniti, come la corrispondenza a valori consentiti o la verifica che non sia nulla.
I test coprono diverse combinazioni di scenari possibili, garantendo che il comportamento della classe rispetti la specifica in condizioni normali, in caso di errori e in assenza di vincoli.
Un aspetto fondamentale di questi test è il metodo executeLogic, che valuta la validità della variabile in base ai controlli configurati. Tra i principali scenari testati troviamo:

- **Controllo di valori consentiti**: si verifica che la variabile sia correttamente validata rispetto a una lista di valori consentiti. Se la variabile rientra nei valori consentiti, il metodo restituisce true; altrimenti, false, con l'impostazione di un codice di errore appropriato (es. VALUE_NOT_ALLOWED).
- **Gestione dei controlli null**: Test che verificano il comportamento della classe quando la variabile è null. In particolare, si testa l'attivazione esplicita del controllo checkNull e la gestione di errori associati (es. codice di errore NULL_VARIABLE).
- **Configurazione dinamica dei vincoli**: Test che illustrano la possibilità di configurare i vincoli dinamicamente tramite metodi come setCheckAllowedValues, verificando che l'elenco di valori consentiti venga applicato correttamente durante l'esecuzione della logica.
- **Comportamenti in condizioni limite**: si esamina il comportamento della classe quando si impongono combinazioni particolari di valori e controlli, ad esempio: la variabile è null o contiene un valore non consentito, ma nessun controllo è stato configurato.

3 | Task T7

Il task T7 prevede un servizio REST che automatizzi il processo di compilazione di file Java, esecuzione dei test e analisi della copertura del codice. Per raggiungere questo obiettivo, vengono utilizzati Maven come strumento di build e JaCoCo come framework per il calcolo della copertura.

3.1 | Struttura del Progetto

Il task è organizzato attorno a tre classi principali:

- **AppController**: Espone il controller per la chiamata REST fornita dal servizio. La sua responsabilità è gestire l'interazione con gli utenti.
- **CompilationService**: Fornisce il servizio principale per la compilazione e il testing del codice.
- **Config**: Supporta le operazioni del servizio di compilazione, gestendo la configurazione dei percorsi del filesystem e centralizzando la logica relativa ai path.

Il cuore del servizio è rappresentato dal metodo compileAndTest() della classe CompilationService. Questo metodo esegue una sequenza di operazioni per compilare e testare il codice Java, utilizzando Maven per orchestrare il processo. Il metodo è progettato per gestire eventuali errori in modo robusto e garantire la pulizia delle risorse temporanee.
Il flusso operativo è suddiviso in cinque fasi principali, ognuna implementata attraverso un metodo dedicato:

1. **Creazione ambiente**: Viene creata la struttura delle directory necessarie e copiato il file pom.xml di configurazione.
2. **Scrittura codice**: Il codice sorgente e i relativi test vengono salvati nei file appropriati all'interno della struttura di progetto..
3. **Avvio compilazione**:Si avvia Maven per eseguire la compilazione del codice, i test automatici e il calcolo della copertura del codice.
4. **Successo o errore**: In base all'esito del processo, vengono impostati i risultati. Questi includono i dettagli di copertura generati da JaCoCo oppure gli errori rilevati durante la compilazione o l'esecuzione dei test. I log vengono registrati per scopi di debugging.
5. **Pulizia**:Tutti i file temporanei e le directory create durante il processo vengono eliminati, garantendo che il servizio non lasci tracce inutili.

3.2 | Definizione e Tipologie di Test

Per garantire la qualità del servizio REST, sono state definite diverse tipologie di test, ognuna mirata a coprire specifici aspetti funzionali e comportamentali del sistema. In particolare:

Test di Tipo Black Box Questi test si concentrano sulla validazione dell'interfaccia REST del servizio, senza conoscere i dettagli dell'implementazione interna.
L'obiettivo è simulare scenari di utilizzo realistici, utilizzando file esterni con una struttura parametrizzata. All'interno di questa categoria sono stati sviluppati:

- **Test di Concorrenza**: Questi test verificano il comportamento del sistema in condizioni di carico multi-thread, validando la capacità del servizio di gestire correttamente richieste parallele senza compromettere l'affidabilità o la consistenza.
- **Test Negativi**: Mirano a verificare il comportamento del sistema in presenza di input errati o scenari di errore. Sono stati sviluppati test per due fasi differenti:
  - **Fase statica** (compilazione): Validano che il sistema sia in grado di rilevare e segnalare errori di compilazione.
  - **Fase dinamica** (esecuzione dei test): Controllano che il servizio gestisca correttamente situazioni di fallimento durante l'esecuzione dei test, segnalando errori in modo appropriato.

Classe di Test	Classe Testata	COV- Statement	COV- Branch	COV - functions
PageBuilderTest	PageBuilder	80%	80%	80,77%
GenericObject ComponentTest	GenericObjectComponent	100%	100%	100%
GenericLogicComponentTest	GenericLogicComponent	100%	None	100%
ServiceLogicComponentTest	ServiceLogicComponent	100%	None	100%
ServiceObjectComponentTest	ServiceObjectComponent	100%	100%	100%
VariableValidation LogicComponentTest	VariableValidation LogicComponent	100%	90%	100%

Tabella 3.1: Classi di test per il Task T7.

Test di tipo white box Sulla Classe CompilationService sono stati implementati test di tipo white box, progettati per verificare il corretto funzionamento di specifici metodi interni.
Questo approccio consente di analizzare in dettaglio il comportamento del codice e di identificare eventuali anomalie nella logica implementativa.

```
private ResultActions Mock_perform(JSONObject requestJson,
    Boolean expectedError, Boolean expectCoverage) throws Exception {
    // Metodo per eseguire la richiesta di compilazione e testing
    return mockMvc.perform(post("/compile-and-codecoverage")
            .contentType(MediaType.APPLICATION_JSON)
            .content(requestJson.toString()))
            .andExpect(status().isOk())
            .andExpect(content().contentType(MediaType.APPLICATION_JSON))
            .andExpect(jsonPath("$.outCompile").isNotEmpty())
            .andExpect(jsonPath("$.error").value(expectedError))
            .andExpect(expectCoverage
                    ? jsonPath("$.coverage").isNotEmpty()
                    : jsonPath("$.coverage").doesNotExist());
}
```

Listing 6: Esempio di Test parametrizzato BlackBox.