UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione

Anno accademico 2022-2023

Relatore: Prof. Porfirio Tramontana

Correlatore: Prof. Luigi Sauro

Un gioco a supporto dell'apprendimento delle problematiche relative ai Test Smells

Insegnamento: Software Testing

Dario Tintore N97000335

1 Introduzione

La presente tesi si pone come obiettivo quello di descrivere la progettazione e l'implementazione di un applicativo software il cui obiettivo principale è quello di fornire uno strumento di supporto all'apprendimento della tematica di Software Testing sotto forma di gioco interattivo.
L'applicazione in questione pone le sue basi su uno studio focalizzato sulla tematica di Software Testing con un approfondimento in particolare sul concetto di Software Testing Gamification, delle sue caratteristiche peculiari, quali i benefici che questo può apportare in un ambiente accademico e lavorativo, per poi trattare in seguito lo sviluppo di un applicativo che possa sfruttare le basi teoriche descritte in precedenza in modo tale da introdurre nuove tipologie di Software Testing Gamification non considerate precedentemente in letteratura.
Lo scopo della tesi quindi consiste nello sviluppo di un sistema informatico che sia di supporto all'apprendimento di scrittura di buon codice di testing, rivolto quindi principalmente a studenti in un ambito accademico.

La tesi è strutturata nei seguenti capitoli:

1. Il primo capitolo si pone come obiettivo di introdurre la tesi
2. Il secondo capitolo si pone come obiettivo quello di descrivere i concetti fondamentali per la comprensione del sistema.
3. Il terzo capitolo si occupa dell'analisi dei requisiti del sistema che si vuole descrivere.
4. Il quarto capitolo si occupa della progettazione del software.
5. Il quinto capitolo si occupa dell'implementazione e delle scelte tecnologiche implementate.
6. Il sesto capitolo si occupa dell'installazione e della configurazione del sistema in questione.
7. Il settimo capitolo si pone come obiettivo quello di elencare dei casi d'uso dell'applicativo con tutti gli smell considerati.
8. L'ottavo capitolo si pone come obiettivo quello di una discussione conclusiva riguardo lo strumento sviluppato.

All’interno di questa tesi saranno studiati e considerati articoli provenienti da diverse fonti scientifiche, giustificando attraverso la letteratura scientifica le motivazioni dello sviluppo della tesi e delle funzionalità proposte. Nello specifico, nei seguenti capitoli, che includono anche parte dell’analisi dei requisiti, si avrà come obiettivo quello di definire e di esporre al lettore il dominio applicativo della problematica di testing e di giustificare un approccio di gamification applicato al testing.

2 Concetti di base

Al fine di fornire al lettore gli strumenti necessari per la completa comprensione del progetto è definita la sezione “Concetti di base” il cui obiettivo è quello di descrivere i concetti fondamentali di cui si fonda il sistema descritto e dei quali è necessaria l’introduzione per una corretta comprensione del sistema stesso.

I prodotti software nel corso del tempo risultano sempre più complessi [1], e questa complessità crescente si abbraccia ad un bisogno sempre maggiore di test software che siano di qualità e che garantiscano un corretto funzionamento anche a fronte di situazioni inattese [2].
Negli ultimi anni (1992-2019) gli studi riguardo il software testing education sono diventati sempre più attivi, e questo lo si può evincere da un incremento del numero di paper pubblicati riguardo la tematica in questione. Sempre più approcci di educazione al testing vengono sviluppati per trovare il miglior metodo possibile di insegnamento al testing [3]. In relazione all’insegnamento, negli anni recenti è stato introdotto il concetto di “Gamification” che va a legarsi al concetto di Software Testing.

2.1 Gamification

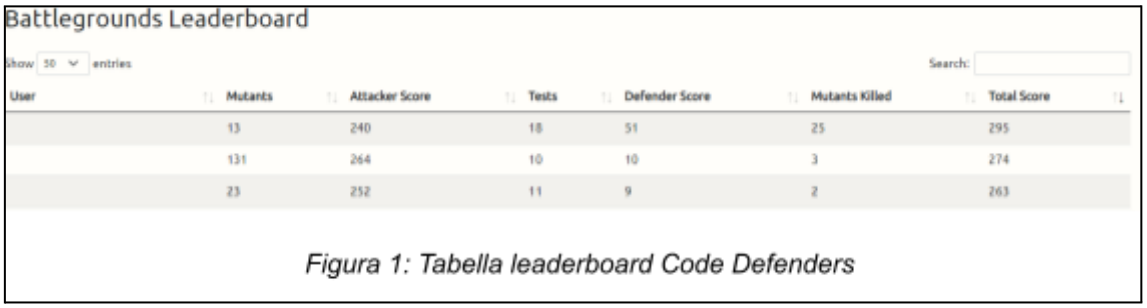
Per gamification si intende una tecnica per migliorare la qualità di un lavoro ed aumentare la motivazione dei lavoratori proponendo un task non più come un compito da portare al termine, ma organizzando il tutto sotto forma di “gioco”. [4]
Un’altra definizione di gamification può essere “L’uso di elementi di game design in contesti non di gioco” [7]. Ad esempio, il processo di prendere feature di giochi reali ed introdurli in contesti alternativi, in modo tale da rendere il task più motivante. La gamification è stata documentata come una tecnica efficace [8], una delle motivazioni si può ritrovare nel fatto che l’impegno dei giocatori è amplificato da una visione del task da portare a termine differente rispetto ad un compito statico, approcciandosi al tutto come un gioco attraverso cui sfidare gli altri “giocatori” attraverso feature come punteggi e classifiche.
Possiamo affermare che il processo di gamification relativamente all’ingegneria del software si divide principalmente in due filoni:

1. Gamification del prodotto software, ovvero implementare elementi di gioco all'interno di un prodotto software, avendo come obiettivo quello di stimolare l'utente con elementi di gioco, e di cambiare l'approccio che ha l'utente nei confronti del prodotto.
2. Gamification del processo di sviluppo del software, ovvero implementare elementi di gioco all'interno del processo di sviluppo del software, avendo come scopo il fatto di stimolare lo sviluppatore, andando a cambiare l'approccio che quest'ultimo ha nel processo di sviluppo.

Gli "elementi di gioco" [2] che possono essere presenti in un gioco sono molteplici:

- Punti: Un sistema di punteggio in grado di tenere conto del progresso del giocatore.
- Classifica: Un sistema di rappresentazione dei punteggi di ogni giocatore in modo tale da creare un ambiente di gioco competitivo.
- Badge: Dei distintivi che riconoscono il successo di un giocatore o l'aver superato determinate sfide.
- Livelli: Un modo di quantificare l'esperienza di un giocatore sotto forma di numero.
- Ricompense: Premi o bonus per i giocatori che raggiungono determinati obiettivi.
- Sfide: Sfide speciali sotto determinate condizioni di tempo o verso altri giocatori
- Sistemi di votazione: Sistema di votazione in cui vengono valutati i migliori giocatori o le migliori soluzioni proposte.
- Scommessa: Sistema di scommessa in cui l'utente scommette una certa moneta di gioco.

Esempio ricavato dal gioco Code defenders:


Battlegrounds Leaderboard

Show 50 entries Search:

User	Mutants	Attacker Score	Tests	Defender Score	Mutants Killed	Total Score
	13	240	18	51	25	295
	131	264	10	10	3	274
	23	252	11	9	2	263

Figura 1: Tabella leaderboard Code Defenders

Nella definizione di gamification possiamo trovare varie tipologie/modalità di gioco che hanno caratteristiche e problematiche diverse l'una dall'altra:

1. Single player, modalità in cui il giocatore non interagisce con altri giocatori.
   Esempio: Il giocatore deve risolvere un task in autonomia senza tener conto di una sfida o di una competizione in real-time con un altro giocatore.
2. Multiplayer, modalità in cui il giocatore interagisce/sfida altri giocatori. Questa modalità prevede le seguenti caratteristiche:
   a. Uno contro uno (Real time): Contesto in cui sono presenti due giocatori con lo stesso task ed il confronto avviene in tempo reale.
      Problematiche principali: Lag, Tempi di risposta, Connessione.
   b. Uno contro uno (A turni): Contesto in cui sono presenti due giocatori con lo stesso task, dove i giocatori si interfacciano in modo sequenziale, prima opera un giocatore, poi opera l'altro.
   c. Uno contro tutti (real time): Contesto in cui un giocatore si confronta con due o più giocatori contemporaneamente.
      Problematiche principali: Scalabilità dell'applicazione nel contesto in cui il numero di giocatori risulta particolarmente elevato.
   d. Uno contro tutti (A turni): Contesto in cui un giocatore si confronta con due o più giocatori con lo stesso task e il risultato viene valutato sulla base dei risultati degli altri giocatori.

È possibile quindi avere la presenza di più modalità di gioco che permettono di implementare all'interno dello stesso sistema diversi tipi di gameplay.

2.2 Esempi di Gamification

Un esempio di gamification in ambito Software Testing che ha riscontrato molto successo lo possiamo considerare con Code defenders.
Code defenders implementa il concetto di gamification basato sulla tecnica di Mutation Testing attraverso un testing game per classi Java su cui bisogna effettuare dei test JUnit.
L'esempio di Code Defenders nella seguente tesi è preso come principale riferimento poiché implementa il concetto di gioco basato su una tecnica che si presta molto bene ad una "gamification" e che è utilizzata in ambito pratico all'interno di progetti reali [5], e allo stesso tempo permette di avere un obiettivo di gioco ben chiaro e verificabile: "Eliminare i mutanti" [6]. Attraverso l'operazione di eliminare i mutanti è possibile ottenere un punteggio ed una metrica oggettiva.
Nel seguente caso è possibile quindi avere elementi di gioco come l'eliminazione dal codice di un determinato elemento, e più questi elementi vengono eliminati più si guadagnano punti che permettono al giocatore di scalare la classifica.

Code Defenders è un gioco in cui la modalità di gioco principale è chiamata Battleground, in cui i giocatori si dividono in Attaccanti e Difensori. In base al ruolo che il giocatore assume cambia radicalmente il gameplay.
Partendo da una classe Java predefinita gli attaccanti inseriscono dei mutanti all'interno del codice introducendo delle variazioni. Lo scopo degli attaccanti è quello di creare delle mutazioni che possono essere scoperte da un test. Più queste mutazioni sopravviveranno ai test dei difensori più questi mutanti forniranno un punteggio alto all'attaccante.
Lo scopo dei difensori invece è quello di scrivere dei test che vanno ad uccidere i mutanti. Il difensore non vede il codice mutato ma solo quello originale ed i test precedentemente realizzati. Il punteggio del difensore si basa sulla capacità di scoprire dei test mutanti. In questo caso possiamo notare alcuni elementi di gioco introdotti nei paragrafi precedenti, come un sistema di punteggio, una classifica basata sui punteggi e sul punteggio da Attaccante/Difensore, una modalità di gioco Uno vs Uno a turni in cui

prima gli attaccanti creano i mutanti e poi i difensori hanno il compito di scoprire questi mutanti.

2.3 Smell

Per introdurre il tipo di gioco su cui la tesi si basa è necessario introdurre il concetto di Bad smell. Mentre in Code Defenders possiamo vedere elementi di gioco orientati all'eliminazione di mutanti, in questo caso il fulcro principale del gioco si basa sull'eliminazione degli smell presenti all'interno di codice di test, andando quindi ad effettuare un processo di refactoring ed eventualmente a suggerire il modo in cui il codice può essere ottimizzato tramite la visualizzazione delle soluzioni degli altri giocatori.
L'obiettivo del gioco è quindi quello di insegnare strategie di scrittura del codice di test che permettano di rendere il codice di test manutenibile in modo facile e veloce.
Il concetto di Bad Smell è un meccanismo attraverso cui si riescono a riconoscere dei problemi di design riguardo la struttura interna di un prodotto software [10]. E' stato dimostrato empiricamente che la presenza di test smell influenza negativamente la comprensione del codice sorgente durante i processi di manutenzione, oltre al fatto che i test smell sono ampiamente diffusi nei sistemi informatici industriali ed open-source[11].
Partendo da questi presupposti lo sviluppo della seguente tesi si pone come obiettivo quello di suggerire ed insegnare uno sviluppo dei test orientato al corretto design tenendo conto di buone e comprovate pratiche di programmazione.
Quando si parla di Code smells si tende a pensare riguardo gli smells del codice di produzione, tuttavia così come il codice di produzione, anche il codice di testing è soggetto a problematiche di Bad Smell, che vanno ad intaccare la manutenibilità, l'estendibilità e l'efficacia delle test suite.
Nei paragrafi precedenti è stata citata l'importanza dei test all'interno del processo produttivo dello sviluppo software [3] e proprio per questo si sente il bisogno di definire degli smells che riguardano non più la scrittura di codice di produzione, ma la scrittura di codice di testing.
Una delle differenze principali tra codice di produzione e codice di test, è la struttura di base di cui si compongono. Il codice di test solitamente si divide nelle seguenti parti:

1. Codice di inizializzazione (dati utilizzati per il testing)
2. Chiamata al metodo da testare
3. Comparazione tra risultato atteso e risultato ottenuto
4. Tear down delle risorse

In questo caso possiamo affermare che il refactoring del codice di test è diverso rispetto al refactoring del codice di produzione [13]

Nelle seguenti pagine si procede ad una semplice introduzione relativa ai test smells, estratti dal sito Testsmells.org, e di una possibile soluzione da applicare al fine di migliorare la manutenibilità del codice.

2.3.1 Assertion Roulette

Lo smell assertion roulette possiamo riscontrarlo nel momento in cui in uno specifico test si presentano diverse asserzioni, senza spiegazioni, all'interno dello stesso codice di test.

Esempio:

```
@MediumTest
public void testCloneNonBareRepoFromLocalTestServer() throws
Exception {
    Clone cloneOp = new Clone(false,
integrationGitServerURIFor("small-repo.early.git"),
helper().newFolder());

    Repository repo = executeAndWaitFor(cloneOp);

    assertThat(repo,
hasGitObject("ba1f63e4430bff267d112b1e8afc1d6294db0ccc"));

    File readmeFile = new File(repo.getWorkTree(), "README");
    assertThat(readmeFile, exists());
    assertThat(readmeFile, ofLength(12));
}
```

Possibile soluzione:

Utilizzare il primo argomento dell'asserzione per restituire un messaggio di spiegazione all'utente nel momento in cui l'asserzione fallisce, in modo tale da facilitare la

comprensione riguardo quale delle tante asserzioni all'interno di un test ha riportato un problema.

2.3.2 Eager Test

Lo smell Eager Test si verifica quando un singolo test verifica più funzionalità contemporaneamente andando quindi a complicare la leggibilità e la comprensibilità del test in questione.
Esempio:

```
@Test
public void NmeaSentence_GPGSA_ReadValidValues(){
    NmeaSentence nmeaSentence = new
NmeaSentence("$GPGSA,A,3,04,05,,09,12,,,24,,,,,2.5,1.3,2.1*39");
    assertThat("GPGSA - read PDOP", nmeaSentence.getLatestPdop(),
is("2.5"));
    assertThat("GPGSA - read HDOP", nmeaSentence.getLatestHdop(),
is("1.3"));
    assertThat("GPGSA - read VDOP", nmeaSentence.getLatestVdop(),
is("2.1"));
```

Possibile soluzione:
Scompattare il codice di test in più metodi di test che si focalizzano sulle singole funzionalità testate.

2.3.3 Resource Optimism

Lo smell Resource Optimism si verifica quando un test effettua delle assunzioni ottimistiche riguardo lo stato/esistenza di una risorsa esterna al codice in questione. Il fatto che una risorsa esterna (ad esempio un Database) sia disponibile o meno è un fattore non deterministico che può causare problemi in un contesto di testing. Gli effetti di questo smell possono essere vari, e dipendono dalla risorsa in questione.

```
@Test
public void saveImage_noImageFile_ko() throws IOException {
```

```
File outputFile = File.createTempFile("prefix", "png", new
File("/tmp"));
ProductImage image = new ProductImage("01010101010101",
ProductImageField.FRONT, outputFile);
Response response = serviceWrite.saveImage(image.getCode(),
image.getField(), image.getImguploadFront(),
image.getImguploadIngredients(),
image.getImguploadNutrition()).execute();
assertTrue(response.isSuccess());
assertThatJson(response.body())
              .node("status")
              .isEqualTo("status not ok");
}
```

Possibile soluzione:

E’ possibile risolvere questa problematica con l’utilizzo di Mock oppure verificando che il test che utilizza le suddette risorse crea/alloca queste risorse prima di eseguire il codice di testing.

3 Definizione del gioco

Nel seguente capitolo verrà trattata l'ideazione del gioco da sviluppare all'interno della tesi, verranno quindi considerati diversi articoli scientifici per giustificare e confrontare le scelte di Game Design con altre soluzioni proposte in letteratura.
Sarà infine proposta una tipologia di gioco da sviluppare per le considerazioni fatte.

3.1 Obiettivi

L'obiettivo di questo elaborato è quello di realizzare uno strumento di apprendimento basato sulla tecnica di gamification citata precedentemente. Tra le principali caratteristiche di questa implementazione vi è un approccio simile al già citato Code Defenders ma con una modalità di gioco e delle regole differenti dal precedente che hanno come scopo quello dell'insegnamento della scrittura di test di qualità attraverso il concetto di smell e del confronto delle soluzioni proposte attraverso una community.
Il metodo di insegnamento di scrittura di test in questo caso fa uso dei Test Smell ottenendo un approccio molto simile a Code Defenders.
In questo caso l'obiettivo principale coincide quindi con la sensibilizzazione degli studenti riguardo la buona scrittura di codice di test. Per ottenere questo obiettivo prendiamo quindi in considerazione la letteratura scientifica, in particolare riguardo l'insegnamento attraverso la tecnica di gamification citata precedentemente.
Al fine di proporre un gioco è necessaria la definizione di un insieme di esercizi con obiettivi chiari e verificabili. Ad esempio, in una delle varie modalità prese in considerazione l'obiettivo consiste nell'eliminare lo smell dal codice sorgente proposto, in modo tale da verificare che lo studente sia in grado di comprendere quando il codice presenta degli smell, dove questi smell sono presenti e implementare delle soluzioni che siano prive di smell.

3.2 Modalità proposte

3.2.1 Refactoring game

Attraverso i Code smell è stata quindi pensata la modalità di gioco **Refactoring tests**: Al giocatore saranno presentati un insieme di esercizi che saranno categorizzati per difficoltà. Gli esercizi saranno pensati e proposti con voluti problemi di design, in modo tale da richiamare all'attenzione i potenziali problemi che sono presenti all'interno dell'esercizio. Il giocatore avrà quindi un code editor accessibile attraverso il browser in cui, da una parte sarà presente la descrizione del codice sorgente che permetterà al giocatore di comprendere qual è l'obiettivo ed il modello di dominio in questione, la seguente sezione non sarà modificabile. A destra invece sarà presente il codice di test da refattorizzare e che presenta eventualmente problemi di design. Compito del giocatore sarà quindi modificare attraverso il code editor proposto dall'applicativo il codice potenzialmente problematico.

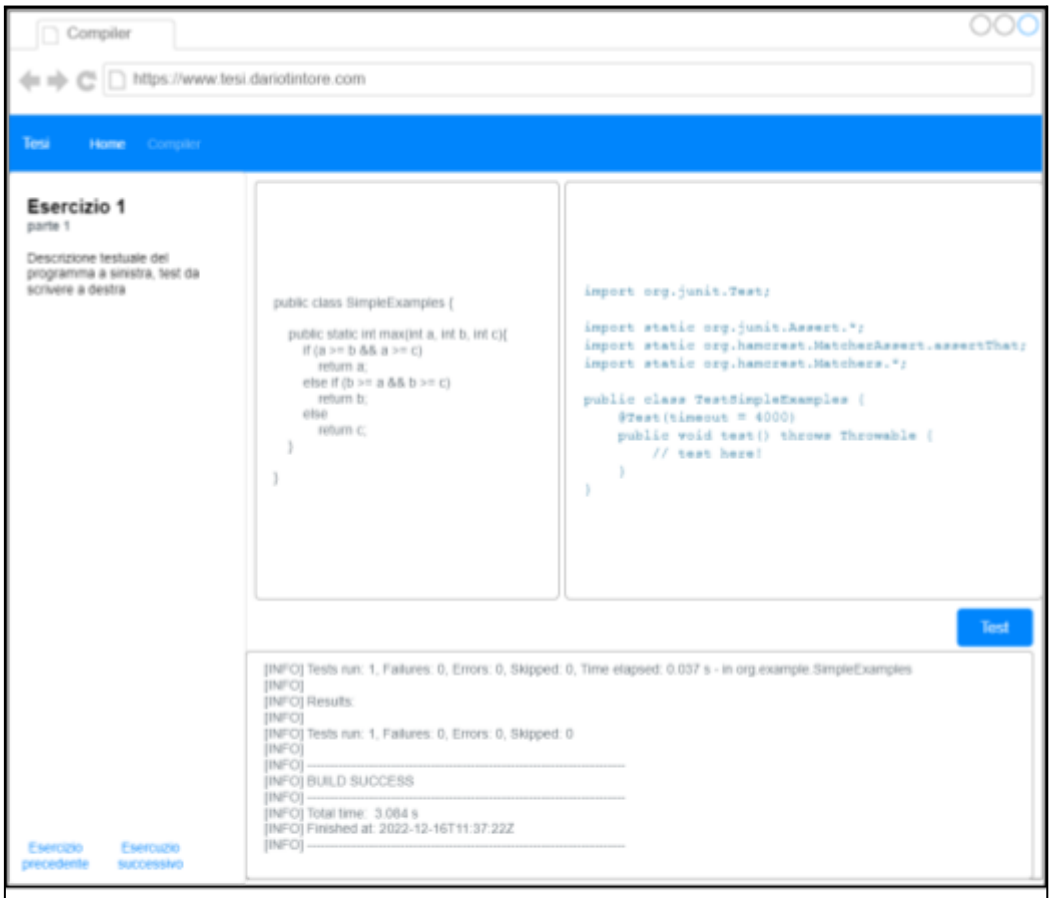


Figura 2: Mockup Refactoring game

All’invio dell’esercizio da parte del giocatore avverrà quindi un processo di verifica che analizza il codice inviato e i potenziali smell presenti. Dopo lo svolgimento dell'esercizio sarà associato un punteggio.
Poiché l’obiettivo principale dello strumento proposto è l’insegnamento di buon design di test code, minore saranno gli smell presenti all’interno del codice refattorizzato, maggiore sarà il punteggio del giocatore.
Nel momento in cui parliamo di refactoring, uno dei task più complessi è quello di verificare che il codice refattorizzato ha lo stesso comportamento rispetto al codice di test originale. In seguito allo svolgimento dell’esercizio verranno quindi eseguite diverse analisi sul codice inviato dal giocatore in modo tale da verificare il fatto che il codice abbia lo stesso comportamento rispetto al codice non refattorizzato (check comportamentale).
L’esercizio svolto quindi presenterà un warning qualora il check comportamentale non vada a buon fine, ovvero il codice refattorizzato non rispecchia correttamente il codice originale.
Per effettuare il check comportamentale ci sono varie opzioni:

1. Attraverso un’analisi di copertura del codice, il codice refattorizzato dovrebbe infatti coprire una percentuale di codice maggiore o uguale rispetto alla percentuale di codice coperta dal test non refattorizzato.
2. Considerando la crazione di codice di test che va a testare il codice di test refattorizzato, andando quindi a verificare che le richieste della descrizione dell’esercizio siano garantite.

Dopo il check comportamentale verrà avviato uno Smell Detector che verificherà la presenza di eventuali smell attraverso un processo di analisi statica del codice.
Il detector analizza il codice, e per ogni smell presente all’interno del codice il punteggio del giocatore viene influenzato.
Al termine del processo di analisi, al giocatore verranno quindi mostrati:

1) Il punteggio ottenuto.
2) Gli smell se presenti.
3) La copertura del codice
4) Il risultato del refactoring check

In seguito allo svolgimento dell'esercizio il giocatore potrà decidere se inviare la soluzione proposta in classifica oppure refattorizzare ulteriormente il codice prima di salvarlo all'interno di un database contenente le soluzioni dei giocatori.
Il giocatore ha la possibilità di visualizzare la leaderboard per l'esercizio selezionato, e se l'esercizio lo permette sarà possibile visualizzare le soluzioni svolte dagli altri giocatori, potendo quindi imparare dagli approcci dei giocatori che hanno avuto un punteggio migliore all'interno della classifica.

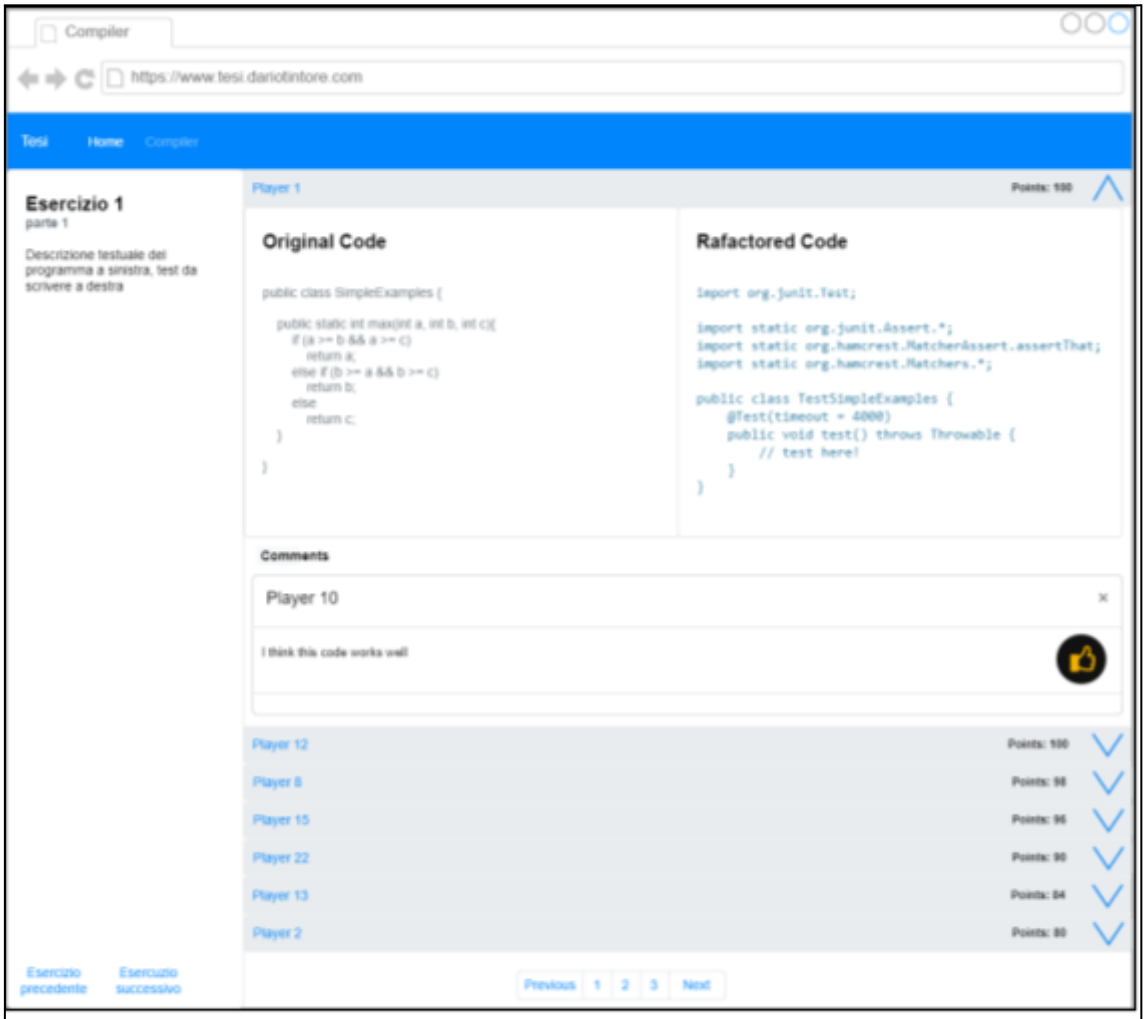


Figura 3: Mockup leaderboard

Nella classifica quindi, oltre ai punteggi, saranno presenti eventuali commenti ad ogni soluzione la possibilità di votare positivamente o meno una soluzione proposta.
Attraverso questa feature è possibile avere un approccio community-based il cui scopo è quello di apprendere il più possibile riguardo la tematica di software testing e confrontarsi.

Nella modalità proposta il processo di gamification riguarda quindi il refactoring di metodi di test, e viene considerata l'eliminazione degli smell come un punteggio riguardo la capacità dei giocatori di trattare e rifattorizzare codice problematico.
Uno scenario esemplare in cui applicare la modalità di gioco proposta potrebbe essere quello di utilizzare lo strumento in questione come supporto di insegnamento alla tematica di Software Testing, dove con la presenza di un un professore (supervisore) viene definito un insieme di esercizi ai giocatori (studenti) in modo tale da presentare degli esempi di cattivo design facili da comprendere e da risolvere. La condizione che ci permette di verificare l'affidabilità di una soluzione la possiamo considerare attraverso l'utilizzo di uno smell detector, attraverso il quale si riesce a comprendere se il giocatore ha svolto correttamente o meno l'esercizio. Lo smell detector effettuerà quindi il ruolo di arbitro il cui giudizio assegnerà o meno dei punti al giocatore andando quindi a rendere il gioco interattivo, senza richiedere necessariamente l'intervento di un supervisore esterno.
In questo scenario potenzialmente ad ogni esercizio è possibile associare più di una soluzione, quindi non è univocamente definito un metodo di refactoring per uno specifico smell e questo rappresenta anche un elemento di design del gioco.
Un ulteriore strumento per assegnare punteggi potrebbe essere l'utilizzo di uno strumento di analisi statica, in modo tale che oltre che valutare il fatto che il codice refattorizzato possiede o meno problemi di test smell, si possono valutare anche altre qualità del codice scritto. Tuttavia siccome il focus principale di questo strumento è la scrittura di buon codice di test si può pensare di assegnare un peso diverso alle valutazioni dei rispettivi strumenti, ad esempio il detector assegna 10 punti per ogni smell tolto, e lo strumento di analisi statica toglie 2 punti per ogni problema riscontrato all'interno del codice.

3.2.2 Check smell

Un'altra tipologia di gioco proposta è il **Check smell**.

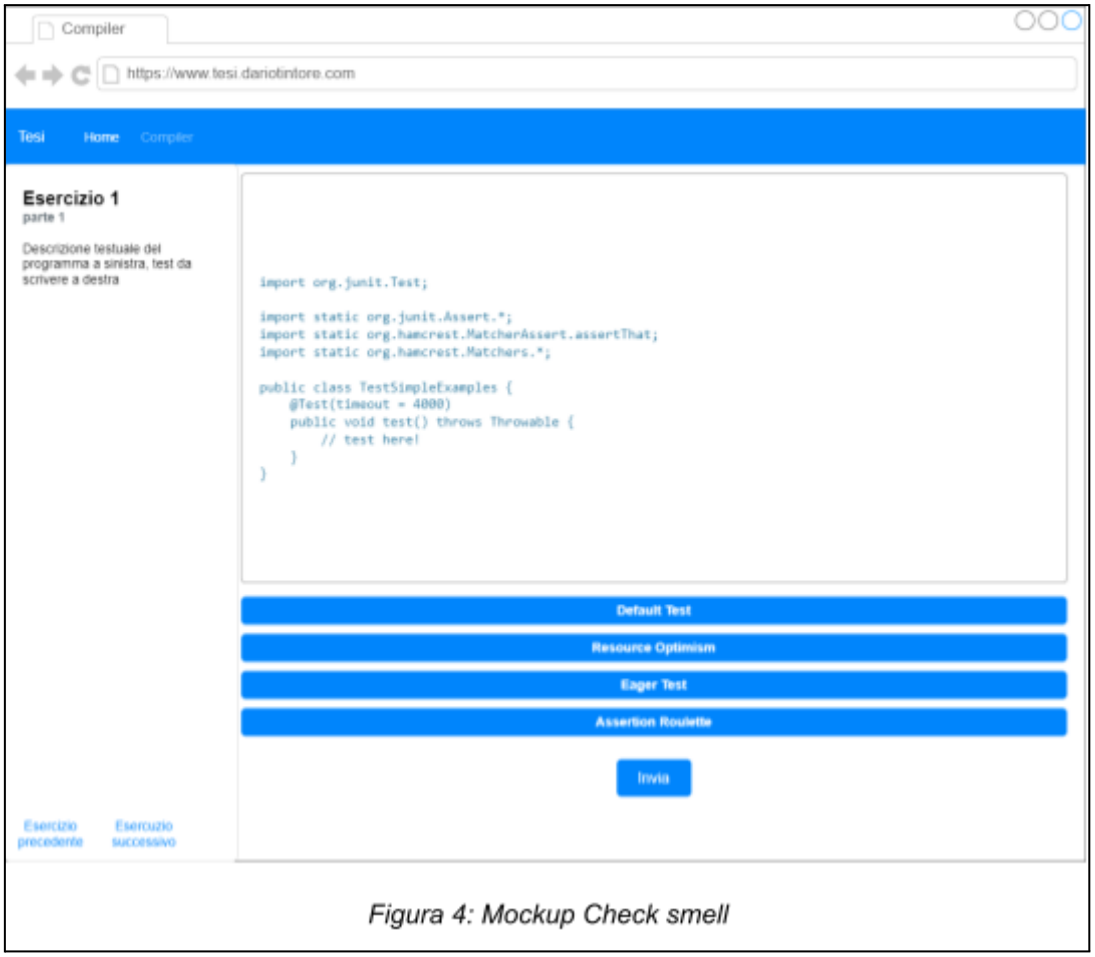


Figura 4: Mockup Check smell

In questa tipologia di gioco viene fornito un codice sorgente ed in seguito viene fornito un elenco dei possibili smell che possono trovarsi all'interno del codice fornito.
Il compito dello studente è quello di selezionare uno dei pulsanti che indica il potenziale smell presente nel codice e selezionare quali di questi risulta essere lo smell presente. In questo contesto la soluzione è univoca e verificabile poiché consiste semplicemente nel verificare che un fissato esercizio che può essere progettato per fini didattici, può essere strutturato in modo tale da avere una ed una sola soluzione. In questo contesto può essere particolarmente utile la struttura statica dell'esercizio, poiché a fronte della progettazione dello specifico esercizio si può spiegare il perché della soluzione senza dover passare necessariamente per uno smell detector.

Possiamo immaginare uno scenario esemplare in cui allo studente viene proposto un esercizio e a fronte di ogni errore può esserci una breve descrizione del perché lo specifico esercizio ha una determinata risposta.
In questo caso il sistema dei punteggi funziona in modo particolarmente semplice:
Ad ogni risposta corretta allo studente viene assegnato un punto, 0 altrimenti.

3.3 Elementi di gamification

Come descritto nella sezione precedente, un gioco si compone di varie regole ed elementi di gioco che permettano appunto di chiamare il prodotto "gioco". La scelta di questi elementi di gioco risulta fondamentale, poiché combacia sia gli elementi di divertimento sia gli elementi che permettono di ottenere l'obiettivo prefissato dal gioco. I principali elementi di gamification presi in considerazione sono stati scelti attraverso pubblicazioni scientifiche in cui esperti del settore della gamification ne approvano l'efficacia in termini di insegnamento e divertimento [14]. Nello specifico, nel gioco proposto consideriamo quindi i seguenti elementi:

- Integrazione con gli obiettivi di insegnamento
  Considerato dagli esperti [14] l'elemento di gioco più importante.
  Il seguente elemento di gioco consiste nel considerare il gioco come un elemento da abbinare ad un corso di insegnamento, in questo caso un corso di Software Testing. Il seguente gioco quindi non consiste semplicemente nell'introdurre un gioco per il semplice scopo di intrattenere ma per insegnare ed integrare ad altri strumenti. In questo caso è di fondamentale importanza quindi definire i requisiti in termini di esperienza e conoscenza del settore richiesti dal giocatore.
  Il gioco quindi non sarà adatto ad un pubblico generico, ma dovrà essere considerato solamente da studenti che sono alle prime armi relativamente alla scrittura di codice di test.
- Feedback Rapido
  La risposta rapida ed interattiva da parte dell'applicativo relativamente al risultato ottenuto è considerata un fondamentale elemento di gameplay.

Durante il corso dello sviluppo è stata pensata ad una modalità attraverso cui il professore facesse da giudice assegnando i punti ai vari esercizi, ma è stata in seguito scartata per favorire maggiore interattività dell'applicativo attraverso uno Smell Detector.

- Obiettivi chiari e verificabili
  L'obiettivo per ogni modalità è definito in modo chiaro ed univoco. Gli esercizi proposti vengono studiati in modo tale da offrire un contributo didattico ai giocatori che si interfacceranno alle varie modalità avendo un chiaro scopo.
- Difficoltà incrementale
  Gli esercizi avranno una difficoltà incrementale per stimolare l'attenzione del giocatore e dare motivazione ai giocatori a migliorarsi sempre di più. La difficoltà incrementale risulta essere particolarmente importante anche perché oltre ad essere un elemento di gioco favorisce anche l'apprendimento di casi di test più complessi.
- Approcci sperimentali
  Nel caso in cui stiamo considerando la modalità di refactoring code non esiste un unico modo per risolvere un esercizio ma diversi esercizi possono avere lo stesso punteggio massimo. Avendo effettuato queste considerazioni, risulta utile l'introduzione della funzionalità di community che permette ai giocatori di discutere riguardo le varie possibili soluzioni proposte.
- Punteggio
  Attraverso il punteggio è possibile ottenere un fattore indicativo tra performance, risultati e sforzi [4]. L'elemento di punteggio risulta essere un indicatore delle performance dei giocatori per invogliare a massimizzare i risultati.
- Classifica
  Attraverso il sistema di punteggio ed attraverso una classifica è possibile introdurre un senso di competitività all'interno del gioco, invogliando quindi i giocatori ad ottenere i risultati migliori e migliorando quindi la qualità di scrittura del codice di test.

3.4 Conclusioni e confronti

Per il suddetto testing game sono state pensate varie strutture di gioco.
La prima struttura del gioco che è stata pensata è quella di gioco community-based che si fornisse di uno smell detector e che gran parte del punteggio fosse basata su un sistema di votazione che mettesse in risalto le soluzioni più votate dai vari giocatori. Attraverso questo sistema sarebbe quindi possibile avere un approccio simile a quello presente in StackOverflow.
Tuttavia questo approccio risulta essere poco applicabile in un contesto di apprendimento universitario in cui il giocatore è da identificarsi come uno studente non esperto della tematica di testing e che quindi le migliori soluzioni non coincidono necessariamente con le soluzioni ottime. In questo caso potrebbero esserci problematiche relative al fatto che una soluzione sbagliata possa risultare più convincente rispetto ad una soluzione corretta in base al numero di voti che utenti non esperti hanno assegnato. Una modalità di gioco fortemente incentrata sulla caratteristica "community based" richiede quindi la presenza di un supervisore (es: professore) che analizzi i risultati e associ ad ogni utente un punteggio di credibilità.
Questa struttura di gioco tuttavia si presta poco alla dinamicità dei giochi di testing e per questo motivo è stata scartata.
È stata quindi presa in considerazione una variante della modalità descritta precedentemente dove il punteggio associato dalla community non influenza significativamente la posizione in classifica delle varie soluzioni proposte, ma il giudizio definitivo è effettuato dal punteggio assegnato alla soluzione da parte dello smell detector e dal refactoring check.
La modalità proposta si affida ad uno smell detector ed un compilatore ed uno refactoring checker.
Il compilatore ci permette di verificare che il codice scritto dal giocatore sia sintatticamente corretto
Il detector ci permette di valutare la soluzione proposta dal giocatore e di assegnare un punteggio.

Il refactoring checker ci permette di verificare che la soluzione proposta sia affidabile oppure bisogna tenere in considerazione un eccessivo processo di refactoring che va a modificare il comportamento del codice.

La modalità proposta non è esente da problematiche:
1) Bisogna verificare che lo smell detector sia affidabile per il task di assegnazione di un punteggio
2) Il problema di verificare che due funzioni hanno lo stesso comportamento (pre refactor/post refactor) è un problema indecidibile, quindi il refactoring check non può essere una discriminante ma solamente un elemento indicativo riguardo l'affidabilità di una soluzione.

Possiamo quindi considerare che per ogni modalità proposta sono presenti delle problematiche di cui non esiste una soluzione unica, ma che bisogna affidarsi a degli strumenti più o meno affidabili.

In passato sono stati già considerati approcci di apprendimento attraverso la gamification [15], in particolare possiamo effettuare un confronto con CodeDefenders.
In particolare, secondo numerosi studi citati [16][17][18] gli studenti risultano molto più invogliati ad utilizzare uno strumento di gamification che ad eseguire un test manuale.
Alcuni degli elementi fondamentali che invogliano il giocatore a giocare ad un gioco piuttosto che eseguire un task sono molteplici.
In questo contesto l'approccio visivo risulta essere fondamentale.
Esempio:

- L'utilizzo di strumenti di code highlighting.
- La presenza di animazioni di gioco.
- Una chiara rappresentazione delle problematiche presenti.
- Il calcolo di un punteggio che rappresenta un feedback rapido.
- Il confronto con gli altri studenti in tempo reale.
- Modalità che coinvolgono l'interazione tra più giocatori.

Secondo gli studi considerati [4], in letteratura non sono stati ancora proposti strumenti di insegnamento di scrittura di codice di test attraverso che utilizzano i test smells, pertanto la seguente tesi ha come obiettivo quello di proporre uno strumento differente che introduce nuovi elementi di game design.
Inoltre nell'ambito dell'ingegneria del software gamification gli strumenti di apprendimento relativi al testing risultano essere quelli presenti in minor parte. [17]
È possibile affermare che in letteratura, per quanto riguarda gli argomenti di refactoring testing ed i design pattern, il supporto alla gamification è assente [18], e da queste basi teoriche nasce l'idea di sviluppo dell'applicativo descritto all'interno di questa tesi.

3.5 Analisi dei requisiti

In questo capitolo vengono formalizzate le principali funzionalità del sistema presentato.

3.6 Requisiti funzionali

- Registrazione di un utente

  L'utente potrà registrare un nuovo account che gli permetterà di usufruire delle funzionalità offerte dall'applicativo.
- Login di un utente

  L'utente potrà effettuare il login con le credenziali definite nella fase precedente.
- Compilazione di codice sorgente

  L'utente potrà compilare codice sorgente scritto direttamente sull'applicativo senza dover necessariamente installare un ambiente di sviluppo al fine di verificare se il codice scritto è eleggibile per una valutazione degli smell.
- Esecuzione di un applicativo di smell detector

  L'utente potrà usufruire di uno smell detector che permetterà in modo automatico di rilevare potenziali problemi di design e scrittura del codice di testing.

- Processo di refactoring del codice

  L’utente potrà effettuare il refactoring del codice considerato problematico dallo smell detector al fine di attuare un processo di rifinitura dello stesso.
- Valutazione degli smell presenti all’interno di codice sorgente

  L’utente potrà visualizzare gli smell noti nella comunità di testing e verificare le migliori soluzioni.
- Riepilogo classifica generale per un dato esercizio

  L’utente potrà essere in grado di verificare la classifica degli utenti per l’esercizio svolto.

- Visualizzazione degli esercizi svolti da altri utenti

  L’utente potrà essere in grado di visualizzare le soluzioni prodotte dagli altri utenti al fine di poter imparare nuove tecniche e soluzioni per uno specifico esercizio già affrontato precedentemente.

3.7 Requisiti non funzionali

Il sistema di gioco implementato prevede le seguenti caratteristiche:

- Portabilità

  Il sistema dovrà essere portabile attraverso l'esecuzione in locale dell'applicativo Front end.
- Compatibilità

  Il sistema Front End dovrà essere compatibile con i principali sistemi operativi.
- Disponibilità

  Il sistema Front end deve essere sempre disponibile per la compilazione di codice e di detection degli smell anche nel caso in cui il Backend non sia disponibile.
- Usabilità

  Un utente deve riuscire ad utilizzare la piattaforma, al massimo delle sue funzionalità, dopo una media di 1 ora di utilizzo.
- Policy password

  Il sistema deve forzare l'utente ad inserire una password di almeno 6 caratteri che contiene numeri ed almeno una lettera maiuscola.
- Manutenzione

  Il sistema deve essere facilmente manutenibile ed estendibile con potenziali modalità di gioco differenti.

- Sicurezza

  Il sistema deve essere protetto da accessi non autorizzati.

3.8 Utenti previsti

Il sistema di gioco da realizzare è rivolto ad un'utenza con una conoscenza basilare del dominio applicativo (scrittura di codice di test e logica di programmazione) poiché questo strumento ha come obiettivo quello di migliorare le capacità degli utenti di scrivere codice che segua i migliori pattern. L'applicativo sviluppato si propone quindi come uno strumento di insegnamento e di miglioramento della conoscenza riguardo il dominio di software testing.

4 Progettazione

In questo capitolo verrà trattata la progettazione del sistema da sviluppare considerando le scelte e l'analisi effettuata precedentemente.

4.1 Casi d'uso

4.1.1 Diagramma dei casi d'uso

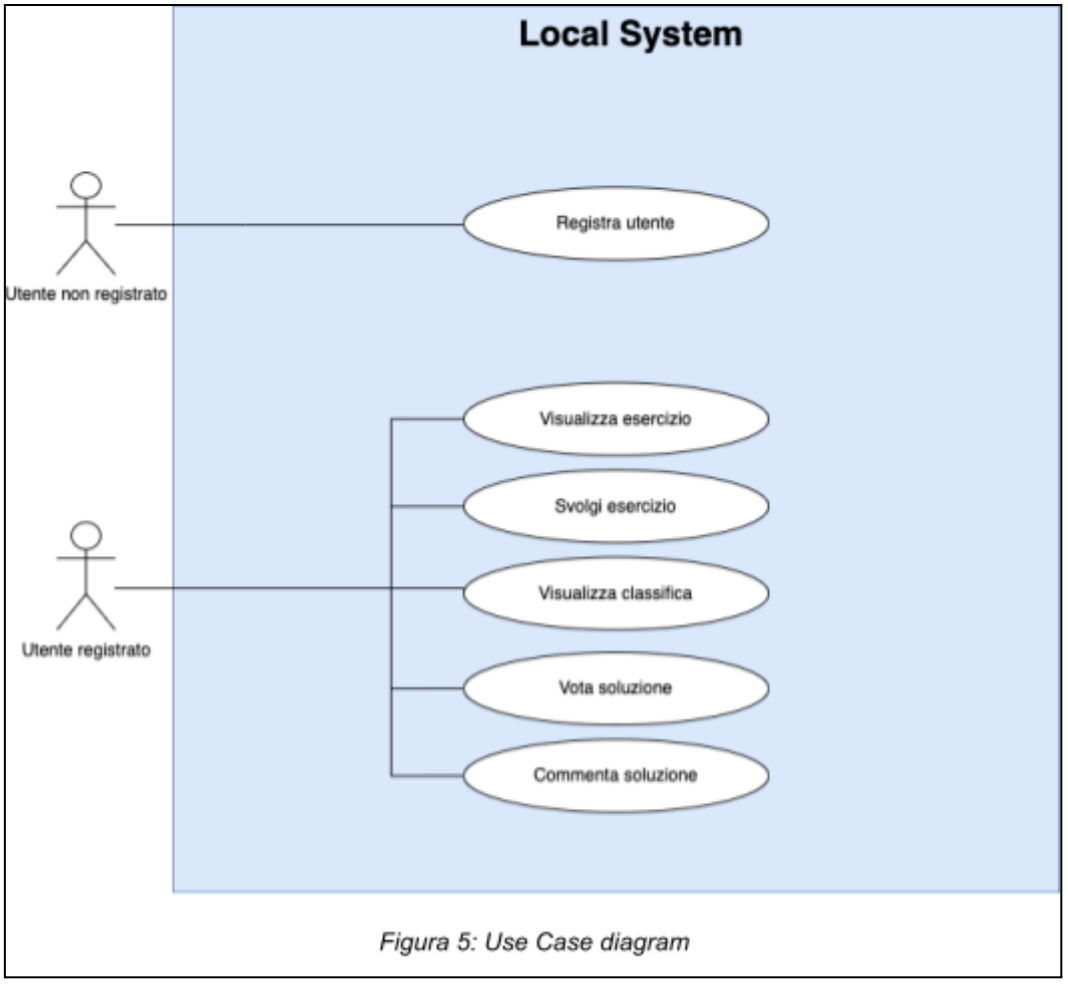


Figura 5: Use Case diagram

4.2 Tabelle di cockburn

<table>
<tr><td>USE CASE #1</td><td colspan="3">Registra utente</td></tr>
<tr><td>Goal in context</td><td colspan="3">L’utente non registrato vuole creare un nuovo account</td></tr>
<tr><td>Preconditions</td><td colspan="3">L’utente non è autenticato</td></tr>
<tr><td>Success End Condition</td><td colspan="3">L’utente riesce a creare un account</td></tr>
<tr><td>Failed End Condition</td><td colspan="3">L’utente chiude la schermata di autenticazione</td></tr>
<tr><td>Primary Actor</td><td colspan="3">Utente non registrato</td></tr>
<tr><td>Trigger</td><td colspan="3">L’utente accede all’applicativo</td></tr>
<tr><td rowspan="6">DESCRIPTION</td><td colspan="3"></td></tr>
<tr><td>Step</td><td>Utente NR</td><td>Sistema</td></tr>
<tr><td>1</td><td>L’utente avvia l’applicazione</td><td></td></tr>
<tr><td>2</td><td></td><td>Mostra la schermata di autenticazione</td></tr>
<tr><td>3</td><td>Inserisce username, password e preme “Registrati”</td><td></td></tr>
<tr><td>4</td><td></td><td>Mostra la schermata “Home”</td></tr>
<tr><td colspan="3"></td></tr>
</table>

USE CASE #2	Accesso alla piattaforma		
Goal in context	L’utente vuole effettuare l’accesso		
Preconditions	L’utente non è autenticato		
Success End Condition	L’utente riesce ad autenticarsi		
Failed End Condition	L’utente non effettua l’autenticazione oppure chiude la schermata di autenticazione.		
Primary Actor	Utente non registrato		
Trigger	L’utente accede all’applicativo		
DESCRIPTION			
	Step	Utente NR	Sistema
	1	L’utente avvia l’applicazione	
	2		Mostra la schermata di autenticazione
	3	Inserisce username, password e preme “Login”	
	4		Mostra la schermata “Home”

<table>
<tr><td>USE CASE #3</td><td colspan="3">Visualizza esercizio</td></tr>
<tr><td>Goal in context</td><td colspan="3">L’utente vuole visualizzare gli esercizi disponibili</td></tr>
<tr><td>Preconditions</td><td colspan="3">L’utente è autenticato</td></tr>
<tr><td>Success End Condition</td><td colspan="3">L’utente riesce a vedere gli esercizi disponibili</td></tr>
<tr><td>Failed End Condition</td><td colspan="3">Il sistema non recupera gli esercizi oppure l’utente chiude la schermata di autenticazione.</td></tr>
<tr><td>Primary Actor</td><td colspan="3">Utente registrato</td></tr>
<tr><td>Trigger</td><td colspan="3">L’utente seleziona una modalità dalla Topbar</td></tr>
<tr><td rowspan="4">DESCRIPTION</td><td colspan="3"></td></tr>
<tr><td>Step</td><td>Utente</td><td>Sistema</td></tr>
<tr><td>1</td><td>L’utente seleziona una modalità dalla topbar</td><td></td></tr>
<tr><td>2</td><td></td><td>Il frontend viene aggiornato con i nomi degli esercizi</td></tr>
<tr><td colspan="3"></td></tr>
</table>

<table>
<tr><td>USE CASE #4</td><td colspan="3">Svolgi esercizio</td></tr>
<tr><td>Goal in context</td><td colspan="3">L’utente vuole svolgere un esercizio</td></tr>
<tr><td>Preconditions</td><td colspan="3">L’utente è autenticato ed ha selezionato un esercizio</td></tr>
<tr><td>Success End Condition</td><td colspan="3">L’utente svolge correttamente l’esercizio</td></tr>
<tr><td>Failed End Condition</td><td colspan="3">È presente un errore di compilazione oppure il codice non passa il refactoring test</td></tr>
<tr><td>Primary Actor</td><td colspan="3">Utente registrato</td></tr>
<tr><td>Trigger</td><td colspan="3">L’utente seleziona un esercizio</td></tr>
<tr><td rowspan="8">DESCRIPTION</td><td colspan="3"></td></tr>
<tr><td>Step</td><td>Utente</td><td>Sistema</td></tr>
<tr><td>1</td><td>L’utente seleziona un esercizio dalla schermata degli esercizi</td><td></td></tr>
<tr><td>2</td><td></td><td>Recupera il codice di test e di produzione</td></tr>
<tr><td>3</td><td></td><td>Prepara la nuova schermata con gli editor riempiti dei rispettivi codici sorgente</td></tr>
<tr><td>4</td><td>Svolge l’esercizio e preme il tasto “Esegui”</td><td></td></tr>
<tr><td>5</td><td></td><td>Invia il codice al backend</td></tr>
<tr><td>6</td><td></td><td>Compila codice</td></tr>
<tr><td>7</td><td></td><td>Esegue refactoring test</td></tr>
</table>

	9		Esegue smell detector
	10		Restituisce risultato
EXTENSION #1 L'utente causa un errore di compilazione			
	Step	Attore	Sistema
	7s		Restituisce l'errore di compilazione
EXTENSION #2 Il codice refattorizzato ha un comportamento diverso rispetto al codice originale			
	Step	Attore	Sistema
	8s		Restituisce il risultato del refactoring test
EXTENSION #3 È presente una modalità di esercizio differente in cui il giocatore risponde ad un quiz sugli smell			
	Step	Attore	Sistema
	5s		Il frontend verifica le risposte e restituisce il risultato

USE CASE #5	Visualizza classifica		
Goal in context	L’utente vuole visualizzare la classifica per un esercizio		
Preconditions	L’utente ha svolto un esercizio		
Success End Condition	L’utente visualizza correttamente la classifica		
Failed End Condition	L’utente chiude la schermata dell’applicativo		
Primary Actor	Utente registrato		
Trigger	L’utente svolge un esercizio		
DESCRIPTION			
	Step	Utente	Sistema
	1	L’utente svolge un esercizio	
	2		Mostra il pulsante “Visualizza classifica”
	3	L’utente clicca il pulsante “Visualizza classifica”	
	4		Mostra la classifica dell’esercizio svolta dagli altri utenti

<table>
<tr><td>USE CASE #6</td><td colspan="3">Vota soluzione</td></tr>
<tr><td>Goal in context</td><td colspan="3">L’utente vuole votare una soluzione di un altro giocatore</td></tr>
<tr><td>Preconditions</td><td colspan="3">L’utente ha svolto l’esercizio ed è presente in classifica</td></tr>
<tr><td>Success End Condition</td><td colspan="3">L’utente vota correttamente l’esercizio</td></tr>
<tr><td>Failed End Condition</td><td colspan="3">L’utente chiude la schermata dell’applicativo</td></tr>
<tr><td>Primary Actor</td><td colspan="3">Utente registrato</td></tr>
<tr><td>Trigger</td><td colspan="3">L’utente visualizza la classifica</td></tr>
<tr><td rowspan="6">DESCRIPTION</td><td colspan="3"></td></tr>
<tr><td>Step</td><td>Utente</td><td>Sistema</td></tr>
<tr><td>1</td><td>L’utente visualizza la classifica di un esercizio</td><td></td></tr>
<tr><td>2</td><td>L’utente seleziona una soluzione dalla classifica</td><td></td></tr>
<tr><td>3</td><td>L’utente preme il tasto “vota positivamente” oppure “vota negativamente”</td><td></td></tr>
<tr><td>4</td><td></td><td>Il voto viene aggiornato</td></tr>
<tr><td colspan="3"></td></tr>
</table>

USE CASE #7	Commenta soluzione		
Goal in context	L’utente vuole commentare una soluzione di un altro giocatore		
Preconditions	L’utente ha svolto l’esercizio ed è presente in classifica		
Success End Condition	L’utente commenta correttamente l’esercizio		
Failed End Condition	L’utente chiude la schermata dell’applicativo		
Primary Actor	Utente registrato		
Trigger	L’utente visualizza la classifica		
DESCRIPTION			
	Step	Utente	Sistema
	1	L’utente visualizza la classifica di un esercizio	
	2	L’utente seleziona una soluzione dalla classifica	
	3	L’utente scrive un commento alla soluzione e preme “Invia”	
	4		Il commento viene aggiunto alla sezione dei commenti relativa all’esercizio

4.3 Modello delle informazioni

4.3.1 Progettazione concettuale delle informazioni

In questo paragrafo viene esposta la progettazione del modello delle informazioni. In questo modello saranno presenti i principali elementi che andranno quindi a definire le basi di dati che saranno poi analizzate successivamente nel capitolo di implementazione. Bisogna tenere a mente che il seguente modello delle informazioni prevede la seguente convenzione:

- Un attributo in grassetto e sottolineato rappresenta una chiave primaria.
- Le classi che hanno una relazione tratteggiata rappresentano delle classi di associazione

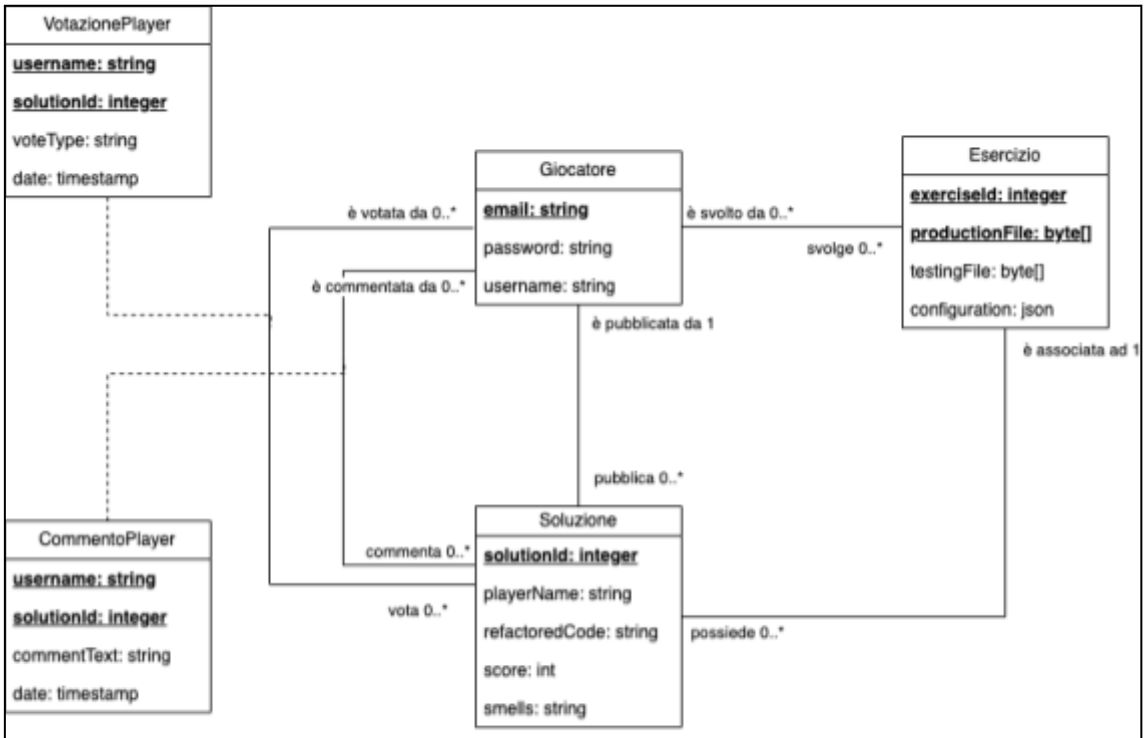


Figura 6: Modello concettuale delle informazioni

4.3.2 Analisi entità ed associazioni

Giocatore: entità che rappresenta il giocatore che utilizza il software, identificato univocamente attraverso l'intero playerId (chiave primaria).
La coppia (email, password) serviranno al giocatore per l'autenticazione. All'utente possono essere associati esercizi/soluzioni/votazioni/classifiche attraverso le rispettive relazioni:

- svolge: relazione che ci permette di comprendere quale esercizio il giocatore ha svolto, il giocatore può svolgere più esercizi.
- vota: relazione che ci permette di comprendere quale soluzione di uno specifico esercizio il giocatore ha votato, il giocatore può votare più soluzioni per uno stesso esercizio.
- pubblica: relazione che ci permette di comprendere quale soluzione il giocatore ha pubblicato all'interno della leaderboard, il giocatore può pubblicare più soluzioni all'interno della leaderboard.

Esercizio: entità che rappresenta uno dei possibili esercizi da svolgere da parte dell'utente. Un esercizio è rappresentato da un insieme di file (Produzione, Testing e Configurazione). Ogni esercizio possiede una chiave primaria (exerciseId),. All'entità Esercizio è possibile associare le seguenti relazioni:

- è svolto da: relazione che ci permette di comprendere da quali giocatori è svolto l'esercizio in questione.
- possiede: relazione che ci permette di comprendere quale classifica è associata ad uno specifico esercizio. In questo caso ogni esercizio possiede una classifica.

Soluzione: entità che rappresenta una soluzione ad un possibile esercizio, una soluzione rappresenta un elemento presente all'interno della leaderboard. Ogni soluzione possiede una chiave primaria (solutionId), una descrizione, ovvero il corpo della soluzione, ed un punteggio espresso in termini numerici che rappresenta il numero di smell che sono stati eliminati attraverso il processo di refactoring. All'entità soluzione è possibile associare la seguente relazione:

- è votata da: relazione che ci permette di comprendere quale soluzione è stata votata da un particolare utente, alla seguente relazione è associata una classe di associazione che permette di identificare la relazione tra la votazione, l'utente e la data in cui la votazione è avvenuta.

4.4 Architettura

L'architettura pensata per l'applicativo in questione è organizzata in due approcci, un approccio che definiamo "Online", ed un approccio che definiamo "Locale".
Le due architetture hanno come obiettivo quello di offrire le stesse funzionalità, avendo come scopo un bilanciamento del carico computazionale da distribuire tra i server universitari ed i calcolatori utilizzati dai giocatori.
La seguente diramazione è stata pensata per il fatto che, essendo l'applicativo destinato ad un ambiente accademico/scolastico, si è cercato di alleggerire il più possibile il carico sui server dell'università, in modo tale da rendere l'esperienza di gioco disponibile a quanti più studenti possibili. Basti infatti pensare che lo svolgimento di un singolo esercizio da parte di uno studente richiede diversi secondi, e che in una classe di decine di studenti la questione diventerebbe facilmente problematica con server dalle risorse limitate.
La differenza tra l'approccio online e l'approccio locale riguarda quindi il bilanciamento del carico relativamente allo svolgimento di un esercizio.
Nell'approccio online, la fruizione dell'applicativo sarà possibile attraverso il browser, andando quindi a rendere accessibile l'applicativo a qualsiasi calcolatore a prescindere dalle risorse possedute, in questo caso l'esecuzione di un esercizio sarà effettuata in cloud.
Nell'approccio locale invece la fruizione dell'applicativo sarà possibile attraverso una desktop application che prevede l'utilizzo di diversi requisiti, in questo scenario tutto il carico di cui il server si faceva carico nell'approccio "Online" viene spostato sul calcolatore del giocatore, andando di fatto a velocizzare l'intero processo di gioco.
Possiamo confermare quindi che l'unica differenza tra i due approcci riguarda lo svolgimento di un esercizio, ma in entrambi gli scenari il backend che verrà utilizzato sarà lo stesso.

4.4.1 Approccio online

L'obiettivo di questa architettura è di sviluppare un applicativo scalabile, che favorisca l'usabilità, la manutenibilità, la portabilità, e che possa essere utilizzata da più utenti possibili contemporaneamente.

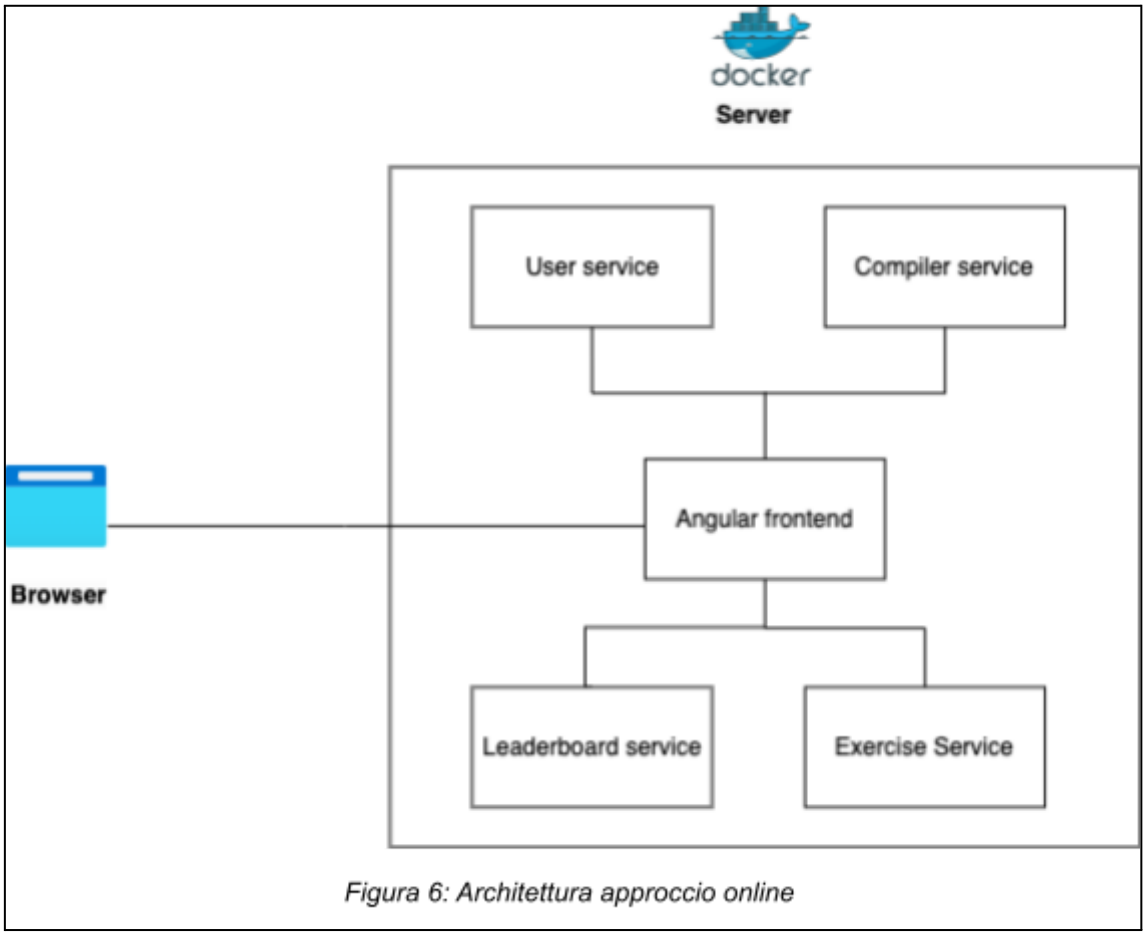


Figura 6: Architettura approccio online

Il primo approccio all'architettura nasce da una problematica presente nella seconda architettura, ovvero il fatto che per avviare l'applicativo l'utente deve sopportare un peso tecnologico dovuto alle richieste dell'architettura, e ad una gestione delle dipendenze più comoda dal punto di vista dell'utente.
Nello scenario offline infatti il giocatore deve avere in locale, al momento dell'esecuzione dell'applicativo desktop, diverse tecnologie che permettono l'avvio dell'applicativo desktop, la compilazione ed il retrieve degli esercizi da svolgere.
Nell'architettura online invece l'utente accede al frontend solamente attraverso un browser, in modo tale che il frontend, comunicando con tutte le altre componenti

dell'architettura, permette l'utilizzo del sistema attraverso un servizio online, andando quindi ad alleggerire notevolmente il carico sul calcolatore dell'utilizzatore. Di contro questo sistema risulta essere meno flessibile in termini di concorrenza, poiché il server dovrà farsi carico della compilazione, dell'esecuzione del test smell detector e di altri calcoli necessari allo svolgimento dell'esercizio. Allo stesso tempo, non è possibile svolgere più esercizi contemporaneamente sullo stesso server, ma sarà presente una coda di tipo FIFO dove i diversi studenti al momento della compilazione dovranno attendere il proprio turno.
In questo caso per ogni richiesta di un giocatore avviene una chiamata REST al microservizio chiamato "Compiler service", dove all'interno di questa chiamata saranno presenti tutti i dati necessari per effettuare la compilazione dell'esercizio in questione. La risposta che invierà il server sarà il risultato di uno svolgimento dell'esercizio. In uno scenario con un'utenza particolarmente estesa sarebbe quindi necessario un orchestratore di container (Kubernetes) che ad ogni utente associa uno specifico container permettendo una maggiore flessibilità della concorrenza, andando però ad appesantire ulteriormente il carico sul lato del server.
Quindi, considerando un ampio numero di utenti scala può essere particolarmente costoso dal punto di vista della scalabilità, dal punto di vista dell'utente risulta essere la soluzione meno impegnativa in termini di requisiti necessari al funzionamento. Tuttavia, poiché l'applicativo è pensato per un utilizzo accademico, quindi riservato ad un pubblico di studenti con esperienza riguardo la scrittura di testing nello specifico come supporto alle lezioni, una soluzione esclusivamente online potrebbe risultare particolarmente lenta.
In questa specifica architettura l'utente interagisce con l'applicativo attraverso un frontend basato su Angular, il quale comunicherà con un insieme di microservizi che permetteranno l'esecuzione di ogni funzionalità. In seguito verranno esposte le funzionalità dei microservizi.

- User service: microservizio che permette la funzionalità di login e registrazione ai vari utenti
- Exercise service: microservizio che svolge il ruolo di database degli esercizi
- Compiler service: microservizio che permette di effettuare la compilazione online.

- Leaderboard service: microservizio che permette di gestire le partite gicoate e la leaderboard per un dato esercizio.

La presenza di un container per ogni funzionalità proposta risulta essere fondamentale poiché in caso di un backend monolitico, a causa dei problemi dovuti alla compilazione ed esecuzione di codice da parte di un utente a seguito dello svolgimento di un esercizio, il tutto si tradurrebbe in una inaccettabile gestione delle richieste da parte dei giocatori. Basti pensare che il processo di refactoring nello specifico richiede particolari sforzi computazionali, quindi necessita di un container separato dagli altri per evitare che altre operazioni (come ad esempio login/recupero esercizi) vengano bloccate. Importante è specificare che l'architettura prevede che ogni container sia dotato di un Database In memory (H2), basato su file interni al container stesso, questa scelta è stata intrapresa per ridurre ai minimi termini il carico di gestione di tutti questi container. In uno scenario alternativo sarebbe stato necessario gestire un container con una funzionalità di DBMS apportando ulteriore carico da gestire per il server ospitante. Questa scelta è stata implementata sia per dare maggiore flessibilità, leggerezza e minore accoppiamento tra i microservizi.

4.4.1.1 Processo di refactoring game (online)

In questo capitolo verrà spiegato il funzionamento dell'applicativo, e nello specifico il funzionamento della modalità di refactoring test.
Il seguente diagramma ha il compito di spiegare cosa avviene al livello del backend:

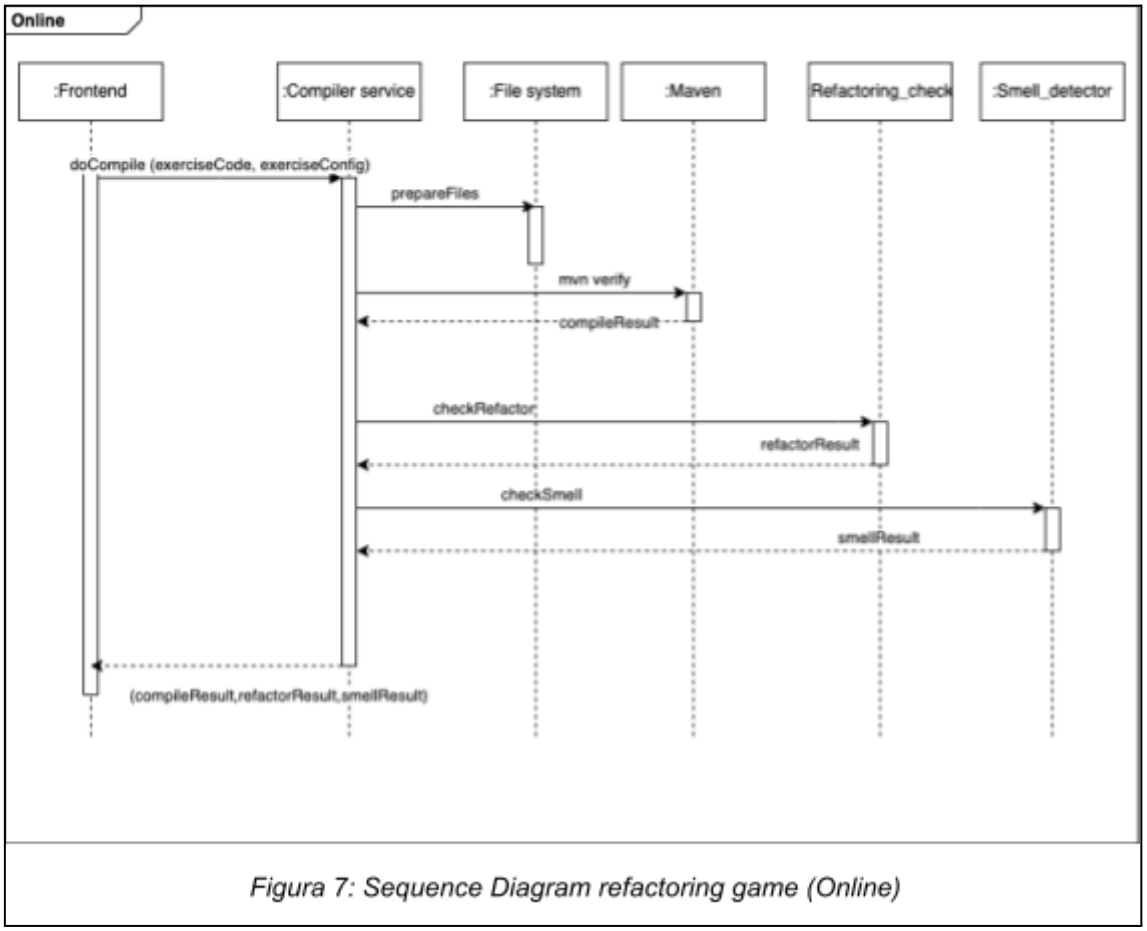


Figura 7: Sequence Diagram refactoring game (Online)

Nel momento in cui un giocatore seleziona la modalità di refactoring tests visualizzerà a schermo tutti gli esercizi presenti nel database, i quali verranno recuperati dinamicamente una chiamata REST-API che comunica con l'exercise service e restituisce tutti gli esercizi presenti nel database. L'utente quindi avrà la lista dei nomi degli esercizi dai quali potrà scegliere quali tra questi selezionare. Dopo aver selezionato l'esercizio, il frontend passerà alla schermata dell'esercizio, composta da due text editors, nel primo text editor è presente il codice di produzione, nel secondo

text editor è presente il codice di test che dovrà essere refattorizzato dal giocatore. Questa schermata è stata progettata in modo tale da far visualizzare al giocatore il codice di produzione, e quindi di permettere una maggiore comprensione dell'esercizio, e dei test che si dovranno refattorizzare. Sia il codice di produzione che il codice di testing, sono dei file presenti all'interno dell'exercise service, e che saranno recuperati all'occorrenza dal frontend attraverso una chiamata REST-API, in modo tale che, quando un giocatore nella schermata precedente seleziona un esercizio, il codice di produzione e di test popoleranno i rispettivi text editor presentati a schermo. Questo tipo di gestione degli esercizi permette facilmente sia di poter aggiungere nuovi esercizi, sia di poter effettuare delle modifiche agli esercizi, ad esempio aggiungendo nuovi tipi di test oppure nuove condizioni. Tutto ciò è possibile poiché basterà semplicemente modificare il file.java dal lato del server e tutte le modifiche saranno visualizzate automaticamente recuperate del frontend nel momento in cui questo chiamerà nuovamente l'esercizio. Per il recupero dei file di produzione e di testing, il frontend effettuerà quindi una chiamata, che passerà come argomento il nome l'esercizio selezionato dall'utente, e otterrà il codice di produzione, ed il codice di testing che serviranno per la schermata successiva.
Dopo che l'utente ha quindi effettuato il processo di refactoring, all'interno del text editor sarà ovviamente presente il codice di test refattorizzato, quindi il giocatore al termine del processo di refactoring potrà inoltrare l'esercizio svolto. Nel momento in cui l'utente inoltra quindi l'esercizio, il codice di test refattorizzato, viene inviato al compiler service insieme ad altri file di configurazione che permettono al compiler service di comprendere le regole di gioco ed altre informazioni utili. Il compiler service quindi si occuperà di iniziare il processo che porterà all'utente ad avere un punteggio, andando a valutare sia quello che sono gli smells presenti, sia quello che è il check comportamentale del refactoring. Quindi il compiler service gestisce il codice recuperato e refattorizzato attraverso il frontend, e lo predispone in modo tale da permetterne la compilazione all'interno del container. Nello specifico, nel momento in cui l'utente seleziona un esercizio richiede all'exercise service di recuperare i file di produzione e di test dell'esercizio selezionato, in seguito questi vengono predisposti all'interno dell'editor frontend, modificati ed inviati nuovamente al compiler-service. In questo stato il compiler service effettuerà, a livello di file system del container, tutte le

configurazioni necessarie ad effettuare una compilazione sia dell'esercizio originale (non refattorizzato), sia dell'esercizio refattorizzato dall'utente.

La scelta di considerare sia l'esercizio originale, sia l'esercizio refattorizzato è dovuta al fatto che necessariamente si deve confrontare il codice di test prima del processo di refattorizzaione, ed il codice di test dopo il processo di refattorizzazione.
Dopo che i file sono stati configurati nel compiler service avverrà in ordine:

1) Aggiunta delle dipendenze necessarie per il rispettivo esercizio
2) Compilazione
3) Refactoring check
4) Check smell

All'interno compiler service quindi è presente un ambiente di sviluppo, in cui è possibile effettuare la configurazione di dipendenze esterne attraverso il file pom, e quindi si tratta semplicemente dell'esecuzione dei test all'interno di un progetto configurato dinamicamente.
In questo caso bisogna tenere in considerazione che una chiamata al compiler service rappresenta uno svolgimento di un esercizio da parte di un giocatore, e che quest'ultima richiede necessariamente diversi secondi di esecuzione, dal momento che bisogna sia eseguire diversi script, sia effettuare un processo di compilazione, ed eventualmente scaricare dipendenze non presenti all'interno del container.

4.4.2 Approccio Offline

L'architettura pensata per l'applicativo è rappresentata dal seguente diagramma L'obiettivo di questa architettura è di alleggerire il più possibile il backend al fine di poter ospitare una sessione di gioco anche su server non particolarmente potenti e di favorire una fruibilità del gioco con una migliorata gestione delle operazioni concorrenti.

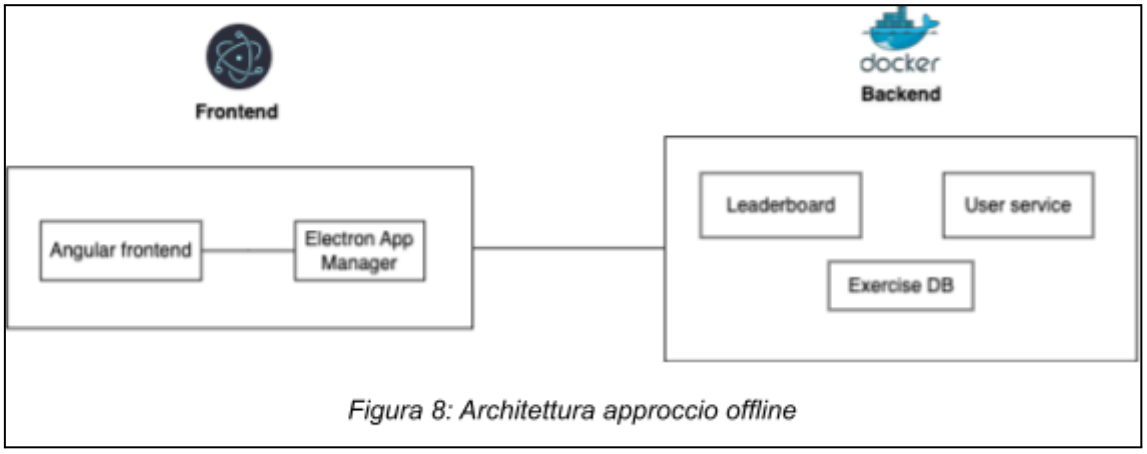


Figura 8: Architettura approccio offline

L'architettura è stata pensata in modo da scomporre il frontend, che in questo caso viene eseguito solamente in locale, dal backend, che viene eseguito su un server esterno e che permette di gestire il database e gli elementi di gamification, come ad esempio la classifica, gli esercizi e così via. Il frontend è realizzato sotto forma di desktop application grazie al framework Electron, rappresentato nell'architettura dal componente "App Manager". Electron consiste in un framework che ci permette di realizzare delle applicazioni Desktop attraverso le tecnologie Web (in questo caso è stato utilizzato Angular). Attraverso il framework Electron è possibile quindi realizzare applicativi desktop, che hanno un interfaccia grafica realizzata attraverso tecnologie web (HTML + CSS), basandosi sulla tecnologia Nodejs. Quindi avremo uno scenario in cui, tra il frontend Angular, e l'App Manager Node ci saranno delle comunicazioni interprocesso che ci permetteranno di gestire il corretto funzionamento dell'applicativo senza necessitare eccessiva potenza di calcolo lato backend. La seguente architettura è stata presa in considerazione poiché durante il corso della progettazione sono stati considerati due approcci fondamentali: un applicativo offline che si basasse sulle

risorse/dipendenze della macchina dello studente, ed un applicativo online, cloud based, che si basasse sulle risorse/dipendenze di un server che offre tutte le varie funzionalità richieste.

Grazie ad electron sarà possibile quindi gestire il File System nello stesso ed identico modo in cui avviene la compilazione sull'architettura Online, con l'unica differenza che la compilazione del codice, l'esecuzione del Test Smell Detector avvengono in locale nelle risorse dell'applicativo.. In uno scenario tipico abbiamo quindi che l'utente completa un esercizio, in seguito l'applicativo frontend invia una richiesta all'App-Manager allegando l'esercizio svolto, l'App-Manager dopo aver effettuato compilazione e check degli smell invia a sua volta il risultato al frontend. Il servizio App-Manager in questo caso ha il compito di sostituirsi a due servizi presenti nell'architettura online, ovvero il Compiler service e l'Exercise service. È possibile infatti effettuare il retrieve degli esercizi in locale senza la necessità di un exercise service, utilizzando una repository github e scaricando in locale gli esercizi che verranno considerati dal giocatore. Il seguente approccio è stato preso in considerazione poiché si vuole spostare il carico computazionale in locale, sul calcolatore del giocatore, andando ad effettuare compilazione e check degli smell senza dover appesantire il server che in presenza di molti utenti potrebbe ritrovarsi a compilare grandi quantitativi di codice, evitando quindi di rallentare il sistema. Attraverso la soluzione suggerita avremmo che l'unico compito del backend sarà quello di immagazzinare dati ed elaborare le soluzioni degli utenti, gestire il database, e gestire gli elementi di gamification attraverso il database, compito molto più semplice rispetto alla compilazione ed il check degli smell ad ogni esercizio svolto dell'utente.

L’architettura pensata per l’implementazione del progetto è **Model-view-controller,** e prevede un’architettura composta da Backend e Frontend tra essi separati.
Ogni elemento del backend è stato organizzato sotto forma di container, i quali comunicano tra di loro attraverso un database MySQL e garantiscono le principali funzionalità di base necessarie al funzionamento dell’applicativo. La scelta di utilizzare un database relazionale si deve ricercare nel fatto che i dati hanno una struttura statica e ben definita, e poiché il database non risulta essere particolarmente complesso da necessitare una distribuzione su più calcolatori.

4.4.2.1 Processo di refactoring game (offline)

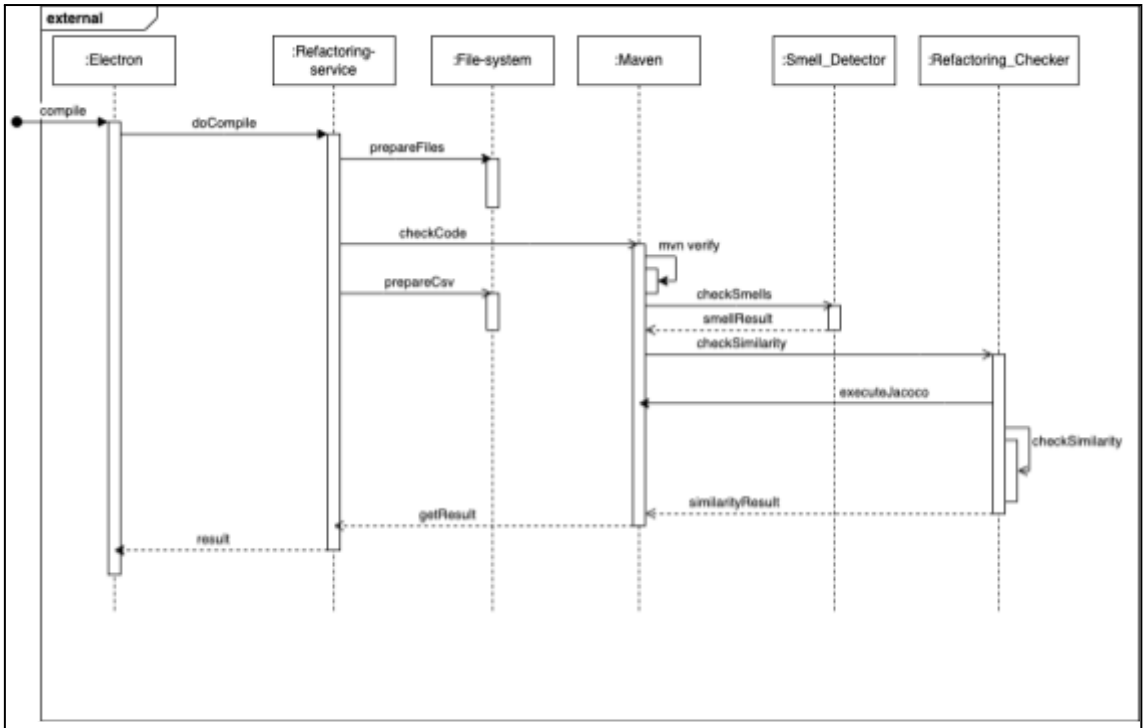


Figura 9: Sequence Diagram processo di refactoring offline

Il processo di svolgimento di un esercizio risulta essere uguale rispetto all’architettura online, con l’unica differenza che, mentre prima il frontend comunicava con il compilatore attraverso delle REST-API, adesso il frontend comunica con il compilatore attraverso dei messaggi inter-processo. In questo scenario è di fondamentale importanza tenere conto che non avviene alcun meccanismo di comunicazione in rete e che tutte le

operazioni risultano essere molto più veloci grazie al fatto di non dover aspettare il proprio turno di compilazione.

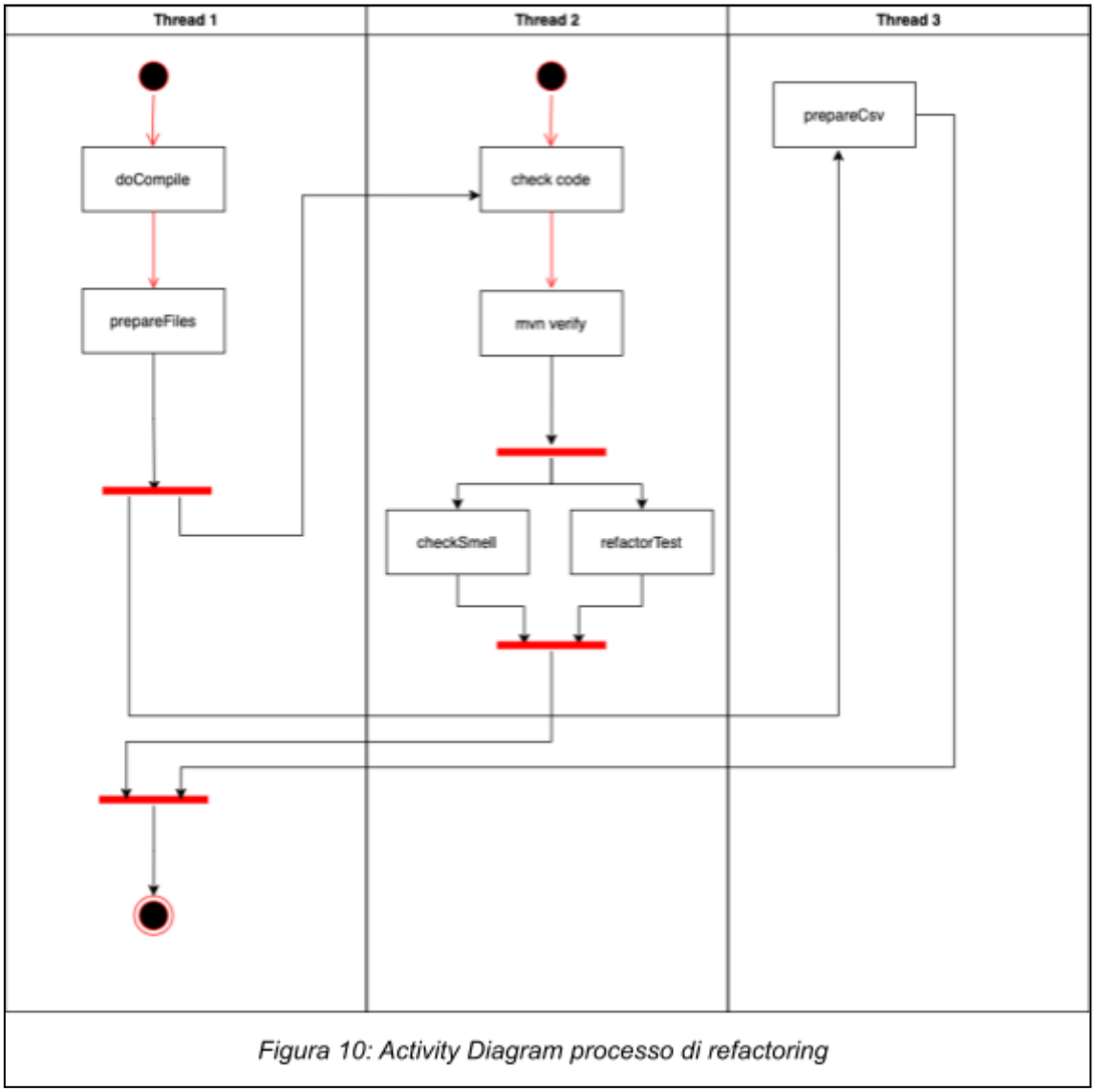


Figura 10: Activity Diagram processo di refactoring

Con il seguente diagramma viene rappresentato il modo in cui l'applicativo Electron gestisce le operazioni necessarie allo svolgimento di un esercizio, nello specifico il modo in cui vengono gestite le chiamate asincrone che sono state rappresentate nel sequence diagram precedente.

4.5 Confronti

I modelli architetturali pensati sono fondamentalmente due approcci, un approccio online, fruibile quindi da Browser (primo approccio), ed un approccio offline, fruibile sotto forma di Desktop Application (secondo approccio).

Le considerazioni per le rispettive architetture sono state fatte tenendo in conto i seguenti punti:
1) Portabilità dell'applicativo
2) Scalabilità dell'applicativo
3) Raggiungibilità dell'applicativo
4) Isolamento dalle dipendenze
5) Gestione della concorrenza

Per quanto riguarda la portabilità, un applicativo online risulta essere particolarmente portabile poiché l'unico requisito necessario è un browser funzionante che permette di accedere al servizio sviluppato, rendendo quindi possibile giocare da qualsiasi dispositivo dotato di browser senza particolari compromessi. Tuttavia grazie al framework Electron si può beneficiare ugualmente di un'eccellente portabilità con un applicativo sotto forma di Desktop Application. Attraverso Electron infatti si viene a creare una Desktop Application basata su Chromium che esegue un vero e proprio browser all'interno della Desktop Application, andando quindi ad abbattere i problemi di portabilità considerati. Nello specifico, è possibile gestire e pubblicare lo stesso applicativo, funzionante per i principali sistemi operativi, con piccole modifiche attraverso strumenti di deploy automatici come Electron-Packager. In questo caso, l'utilizzatore dell'applicativo dovrà semplicemente clonare la repository ed avviare lo script di compilazione che permetterà l'esecuzione dell'applicativo in locale. Electron-packager rappresenta quindi uno script di compilazione che genera un file eseguibile adatto al sistema operativo che si sta considerando (Linux/Windows/Mac). Quindi è possibile affermare che dal punto di vista della scalabilità non ci sono particolari differenze tra un approccio online ed offline.

Dal punto di vista della scalabilità un applicativo online risulterebbe essere più vantaggioso poiché sarebbe possibile gestire un server cloud in cui è possibile aumentare on demand le varie risorse hardware riservate ai singoli servizi utilizzati, cosa non possibile da fare in un applicativo locale.

Per quanto riguarda la raggiungibilità dell'applicativo, questo risulta essere un problema non di poco conto, la necessità di dover esporre un indirizzo IP pubblico, la necessità di dover comprare dei server o sottoscrivere un abbonamento di cloud computing su cui ospitare l'applicativo per una tesi riservata ad un uso accademico/didattico risulta essere particolarmente sconveniente e costoso. Più vantaggioso risulta invece un approccio offline in cui i principali calcoli per l'esecuzione dell'applicativo vengono effettuati offline, su una macchina locale.

L'isolamento dalle dipendenze risulta essere un punto a sfavore di un approccio offline poiché su un server online sarebbe possibile installare solo ed esclusivamente le dipendenze necessarie al funzionamento dell'applicativo da parte degli utenti. In un'architettura offline è richiesto quindi che gli strumenti necessari alla compilazione ed all'esecuzione dei file java siano già preinstallati sul sistema dell'utilizzatore del gioco. Nello specifico è richiesto:

- Node
- Maven
- Java

Per quanto riguarda la gestione della concorrenza, questa risulta essere un punto particolarmente a sfavore di un'architettura online, poiché bisogna considerare che tutte le operazioni che vengono effettuate per lo svolgimento di un singolo esercizio risultano essere bloccanti per tutti gli altri giocatori in coda a meno che non vengano impiegate un considerevole quantitativo di risorse hardware (un istanza di un container per ogni x giocatori). Spostando quindi questo carico in locale il servizio di gioco risulterà molto più scorrevole e dinamico, senza dover attendere la compilazione e l'esecuzione degli esercizi da parte degli altri giocatori.

In seguito viene quindi rappresentata una tabella riguardo i punti a favore di ogni architettura, dove con il colore verde indichiamo un vantaggio, mentre con il colore rosso indichiamo uno svantaggio dal punto di vista della realizzazione/fruibilità dell'applicativo.

	Portabilità	Scalabilità	Raggiungibilità	Dipendenze	Concorrenza
Online					
Offline					

4.5.1 Database esercizi

Un'ulteriore soluzione per alleggerire maggiormente il carico sul backend sarebbe quella di evitare di considerare di creare un servizio di database che fornisce gli esercizi ospitando questi ultimi su una repository Github per recuperare dinamicamente gli esercizi dal servizio di hosting di github. Questo si traduce nel fatto che, anche per pubblicare nuovi esercizi e renderli disponibili a tutti gli studenti basta semplicemente effettuare una push sul repository github pubblico e l'applicativo al prossimo refresh recupererà tutti i nuovi esercizi.

4.6 Processo di refactoring check

Il processo di refactoring consiste nel gestire e ristrutturare una parte di codice considerata "legacy" in modo tale da avere una struttura più comprensibile e manutenibile, mantenendo però sempre lo stesso comportamento. Algoritmicamente parlando il problema di verificare che due test, uno refattorizzato ed uno legacy, hanno lo stesso comportamento, si avvicina al problema di verificare che due funzioni sono equivalenti, quindi stiamo trattando un problema indecidibile, ovvero un problema per cui non esiste alcun algoritmo in grado di risolverlo. Poiché il processo di refactoring risulta un elemento chiave all'interno dello svolgimento dell'esercizio è stato pensato di introdurre una soluzione alternativa che potesse permettere di avere una stima

accettabile riguardo la similitudine di due test. In questo caso è stato pensato quindi di sfruttare due elementi fondamentali dei test:

1) La copertura del codice di produzione dei test
2) Il risultato dei test

Come concetto di base, bisogna considerare che la copertura di codice sorgente è un insieme, quindi date due coperture (test originale e test refattorizzato), la percentuale di copertura ottenuta dallo strumento di copertura del codice rappresenta solamente la cardinalità di uno dei due insiemi, quindi effettuare la differenza tra la percentuale di copertura del test originale, e la percentuale di copertura del test refattorizzato non rappresenta uno strumento efficace per identificare di quanto i due test differiscono tra di loro. In questo caso quindi, essendo le coperture di codice due insiemi, per verificare la differenza tra questi due insiemi è necessario effettuare un'operazione di differenza insiemistica. Consideriamo in questo caso due insiemi:

- A (Insieme che rappresenta la copertura di codice del test originale)
- B (Insieme che rappresenta la copertura di codice del test refattorizzato)

La differenza tra questi due insiemi non è pari alla differenza delle cardinalità, ovvero le percentuali di copertura dei rispettivi insiemi a meno che non sia presente una condizione di inclusione, ovvero $A \subset B$ oppure $B \subset A$, ovvero la differenza tra due insiemi è pari alla differenza delle cardinalità soltanto se un insieme include l'altro.

Per verificare che $A \subset B$ oppure $B \subset A$ bisogna verificare che $A \cup B = A$ oppure $A \cup B = B$. Per effettuare questo controllo guardiamo alle cardinalità e quindi alle percentuali.
Se questa condizione non vale, per verificare di quanto sono differenti i due insiemi andiamo a calcolare la cardinalità della differenza, e quindi la cardinalità dell'unione dei due insiemi meno la cardinalità di A più la cardinalità dell'unione dei due insiemi meno la cardinalità di B, formalmente:

$$|((A \cup B) / A)| + |((A \cup B) / B)| = differenza\ tra\ codice\ originale\ e\ refattorizzato$$

In questo caso, per verificare la differenza di copertura di due codici di test possiamo utilizzare il seguente algoritmo:

1) Se un insieme è incluso nell'altro insieme allora si effettua la differenza delle cardinalità dei due insiemi.
2) Se non vale la condizione precedente non vale andiamo a calcolare di quanto questi due insiemi differiscono tra di loro.

Per considerare l'unione dei due insiemi utilizziamo lo strumento di aggregate report. Una definizione formale di aggregate report possiamo considerarla dal sito di Eclemma, di cui lo strumento considerato, Jacoco, risulta essere un'evoluzione tecnologica dello stesso strumento Eclemma:
"Creates a structured code coverage report (HTML, XML, and CSV) from multiple projects within reactor. The report is created from all modules this project depends on, and optionally this project itself. From those projects class and source files as well as JaCoCo execution data files will be collected. In addition execution data is collected from the project itself. This also allows to create coverage reports when tests are in separate projects than the code under test, for example in case of integration tests."
L'idea per confrontare la differenza tra test refattorizzato e test non refattorizzato è stata quindi quella di creare tre moduli all'interno del container JUnit service, un modulo che contiene la classe dell'esercizio ed il test originale, un modulo che contiene la classe dell'esercizio ed il test refattorizzato, ed un modulo aggregato che considera i risultati di entrambi i moduli, ovvero l'unione.
Il senso di creare tre moduli differenti lo abbiamo perché in questo modo, con un test aggregato è facilmente possibile vedere, in ogni modulo, sia la cardinalità dell'insieme A, sia la cardinalità dell'insieme B, sia la cardinalità dell'unione di entrambi gli insiemi con un semplice comando di verifica.
Per quanto riguarda l'implementazione dell'algoritmo che calcola la differenza tra test refattorizzato e test originale è stato utilizzato uno script python.
Lo script python quindi avvia il processo di verifica con il comando mvn clean verify, in modo tale che JaCoCo per ogni modulo all'interno del container Junit service, genera un file di report in cui è rappresentata la cardinalità di ciascun insieme. Precisamente abbiamo:
Original-module (Contiene il test originale)
Refactored-module (Contiene il test refattorizzato)

Aggregate-reports (Contiene l'unione)
Dal primo modulo otteniamo quindi l'insieme A, dal secondo modulo otteniamo l'insieme B e dal terzo modulo otteniamo l'unione dei due insiemi.
Quindi per ognuno di questi moduli verrà creato un report in formato html, dove attraverso la liberia python BeautifulSoup, che permette di effettuare un processo di scraping di pagine html, in modo tale da estrarre i dati generati dallo strumento JaCoCo e renderli disponibili allo script per effettuare il calcolo della differenza insiemistica. In questo modo possiamo giudicare che i due test hanno un comportamento simile se la differenza in termini di percentuale è al di sotto del 5%. Il valore 5% è un valore simbolico utilizzato ai fini di debugging dell'algoritmo e può essere configurato a seconda delle esigenze e dell'esercizio.

4.6.1 Dipendenze dinamiche

Dal momento che il funzionamento dell'applicativo si basa su un progetto java che viene compilato ed eseguito dinamicamente attraverso l'applicativo Electron, potrebbero verificarsi delle problematiche nel momento in cui si vanno a considerare esercizi con dipendenze che non sono state considerate ed installate durante la fase di sviluppo.
L'idea di base è quella di avere un frontend che con una comunicazione interprocesso (scambio di messaggi) tra electron ed angular, angular richiede all'applicativo electron di effettuare di volta in volta i comandi e gli script necessari allo svolgimento di uno specifico esercizio, ad esempio:
Nel momento in cui un giocatore scrive all'interno del frontend l'esercizio svolto, premendo il tasto "compila" angular invia una richiesta ad electron di eseguire in locale tutti gli script necessari allo svolgimento di un esercizio. Nello specifico ciò che viene scritto all'interno del frontend viene integrato con un progetto java preesistente sul backend, in modo tale che sia possibile interfacciare maven con le rispettive librerie necessarie.

La struttura della directory ha la seguente forma:

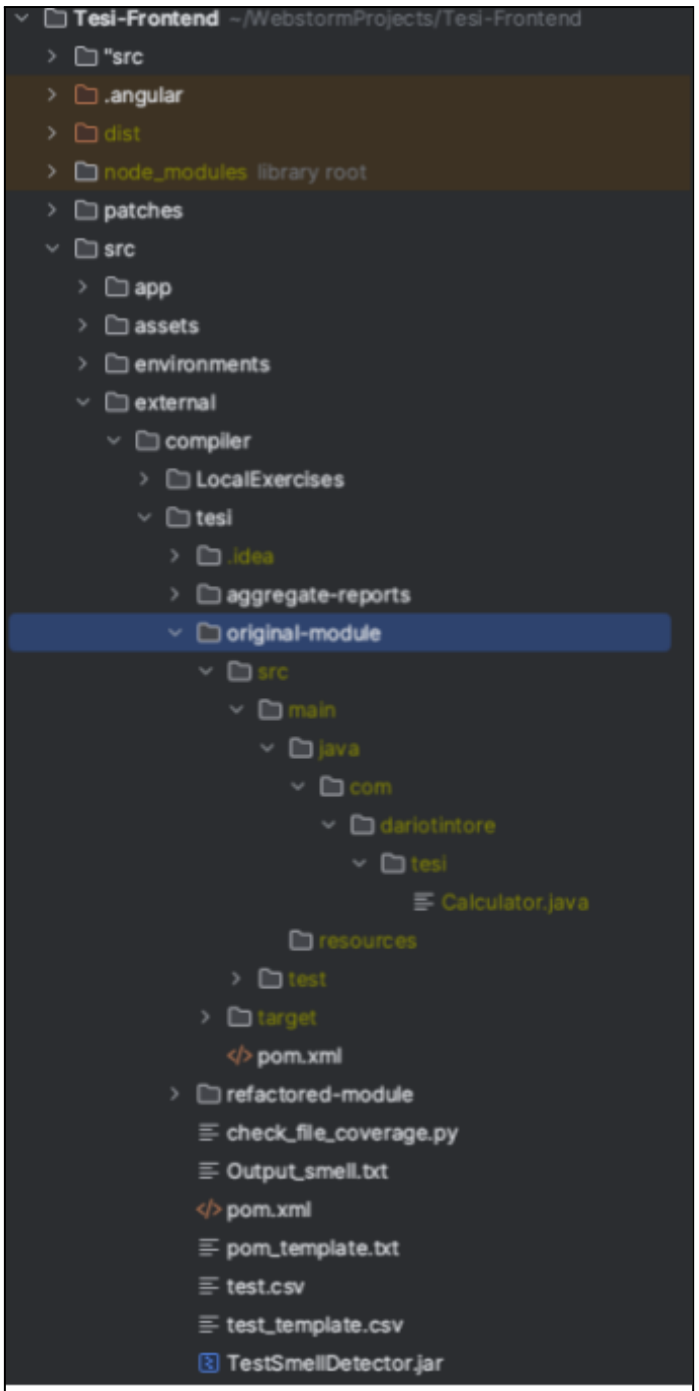


Figura 11: Struttura directory

Al fine di poter definire delle dipendenze dinamiche, definite tramite file, vengono utilizzati due file di testo ed uno script, nello specifico:

1) pom_template.txt
2) pom.xml
3) Dependency-Injector.js

Il file pom.xml è il file che viene considerato da maven nel progetto al momento della compilazione, quindi consiste del file contenente tutte le dipendenze richieste al momento della compilazione.
Il file pom_template.txt è un file di template che al posto delle dipendenze contiene la parola chiave {dependency}, in modo tale che lo script Dependency-Injector.js localizza il posto nel pom_template in cui inserire le dipendenze, per poi sostituire all'occorrenza di {dependency} tutte le dipendenze richieste dall'esercizio che sono state definite all'interno del file.

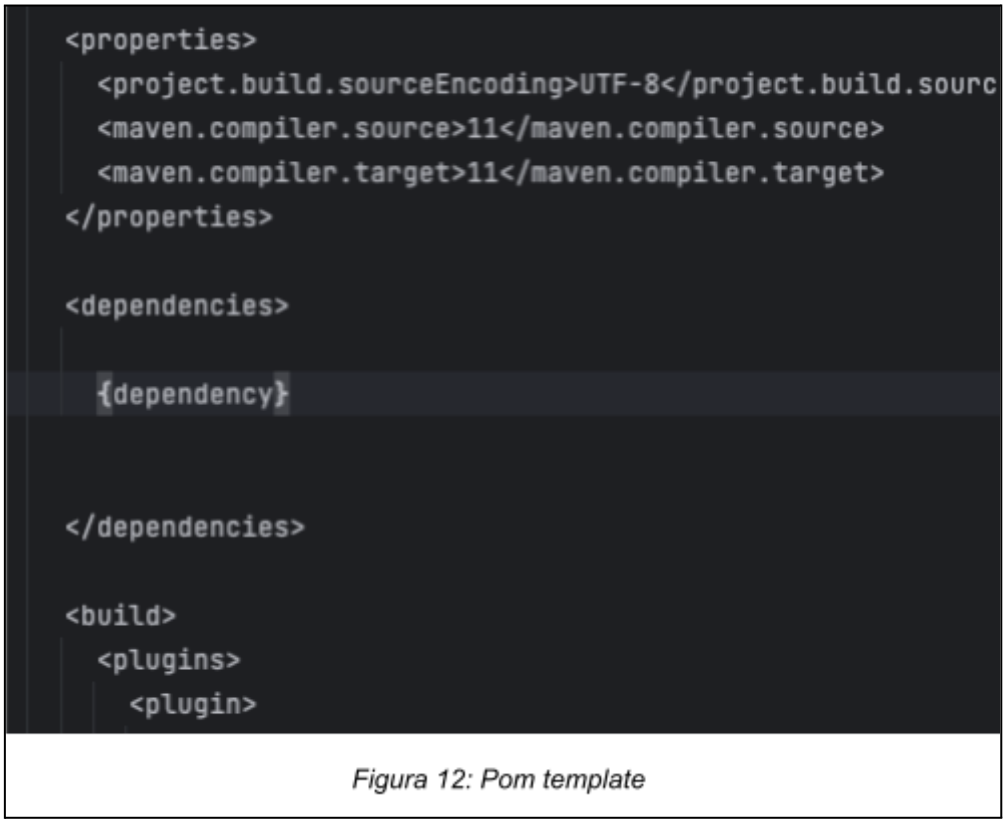


Figura 12: Pom template

In modo tale che nel momento in cui il file viene modificato, quest'ultimo sarà salvato con il nome di pom.xml e quindi al momento della compilazione le dipendenze verranno iniettate dinamicamente in base all'esercizio che stiamo considerando.
Un meccanismo simile avviene anche con l'esecuzione dello smell detector poiché quest'ultimo richiede l'utilizzo di un file csv per localizzare correttamente i file di testing e di produzione che devono essere verificati.

4.7 Opzioni esercizi

Un’ulteriore caratteristica dell’applicativo è quella di utilizzare dei file di configurazione per ogni esercizio in modo da specificare parametri aggiuntivi per uno specifico esercizio. I parametri servono quindi a configurare il modo in cui l’esercizio viene svolto in modo programmabile. Ad ogni esercizio è associato un file di configurazione che contiene i seguenti campi:

Variabili di configurazione per la modalità refactoring game:

- exerciseId: Stringa che definisce l’id di un determinato esercizio. Questo valore è utilizzato per recuperare le soluzioni all’interno della leaderboard visualizzata dal frontend. Se quindi ad esempio consideriamo lo scenario abbiamo un esercizio con exerciseId: “Calculator_01” dove all’interno del database sono presenti delle soluzioni associate a questo esercizio, nel momento in cui questo id verrà cambiato e dal frontend verrà cliccato il tasto "Leaderboard", la classifica cambierà poiché quest’ultima si basa sull’id dell’esercizio. Il senso di utilizzare un meccanismo del genere è da ricercarsi nel fatto che se si vuole effettuare una modifica al file di test o di produzione bisogna necessariamente considerare una nuova classifica, per evitare che soluzioni di più versioni dello stesso esercizio vengano recuperate nella stessa classifica.
- dependencies : Dipendenze maven da utilizzare per l’esercizio, in modo tale che queste vengano iniettate automaticamente nel compiler service
- refactoring_limit: Valore numerico che indica la percentuale di differenza tra il codice di test originale ed il codice di test refattorizzato
- smells_allowed: Valore numerico che indica il numero di smells massimo richiesto dall’esercizio in questione.
- ignored_smells: Lista di smells che verranno ignorati nel calcolo del punteggio.

Variabili di configurazione per la modalità check game:

- questions: Array che contiene le domande da presentare all’interno della modalità check game.

Le domande hanno la seguente struttura:

- questionTitle: Titolo della domanda
- questionCode: Codice che fa riferimento alla domanda

- answers: array di risposte

Le risposte hanno la seguente struttura:

- answerText: Stringa che rappresenta il valore della risposta
- isCorrect: valore booleano che rappresenta se la risposta è corretta o meno

Variabili di configurazione generiche:

- auto_valutative: Valore booleano che rappresenta se l'esercizio in questione è un test autovalutativo o meno. A seconda del valore auto_valutative il comportamento del frontend può cambiare.

Calculator.config.json


```
{
 "exerciseId": "Calculator_1",
 "refactoring_game_configuration": {
   "dependencies": [
     "<dependency>\n  <groupId>junit</groupId>\n
<artifactId>junit</artifactId>\n
<version>4.13.2</version>\n  <scope>test</scope>\n
</dependency>\n"
   ],
   "refactoring_limit": 5,
   "smells_allowed": 3,
   "ignored_smells": [
     "Assertion Roulette"
   ]
 },
 "check_game_configuration": {
   "questions": [
     {
       "questionTitle": "Qual è lo smell presente nel
seguente metodo?",
       "questionCode": "    @Test\n    public void
testAdd() { \n        int a = 15; int b = 20; \n        int
expectedResult = 35;\n        //Act \n        long result =
```

```
objCalcUnderTest.add(a, b);\n        //Assert\n
assertEquals(expectedResult, result);\n}\n",
      "answers": [
        {
          "answerText": "Assertion roulette",
          "isCorrect": true
        },
        {
          "answerText": "Magic number",
          "isCorrect": false
        }
      ]
    },
    {
      "questionTitle": "Qual è lo smell presente nel
seguente metodo?",
      "questionCode": "    @Test\n    public void
testSubtract() {\n        int a = 25; int b = 20; \n
int expectedResult = 5; \n        long result =
objCalcUnderTest.subtract(a, b);\n
assertEquals(expectedResult, result);\n    }\n",
      "answers": [
        {
          "answerText": "Resource Optimism",
          "isCorrect": true
        },
        {
          "answerText": "Unknown Test",
          "isCorrect": false
        }
      ]
    }
  ]
```

```
 },
 "auto_valutative": true
}
```

4.8 Valutazione

Uno degli scopi dell'applicativo in questione è quello di fornire agli studenti ed ai professori che utilizzeranno questo strumento, un modo di effettuare delle verifiche sotto forma di gioco.
Gli esercizi di refactoring sono quindi organizzati in modo tale da appartenere a due categorie:

1) Esercizi auto valutativi: esercizi in cui il giocatore può fare pratica con lo strumento e cominciare ad imparare quest'ultimo
2) Esercizi competitivi: esercizi in cui il giocatore invierà il risultato direttamente in forma ufficiale.

La struttura degli esercizi quindi cambia rispettivamente se ci troviamo in un tipo di esercizio competitivo oppure auto valutativo. Si può pensare ad esempio ad uno scenario in cui una classe di studenti deve effettuare un test assegnato da un professore, il cui esercizio avrà come tag nei file di configurazione la chiave auto-valutative = false.
Un esempio di gestione degli esercizi auto valutativi è il fatto che nel momento in cui viene visualizzata la classifica per un dato esercizio viene effettuata la possibilità di commentare, visualizzare l'output di un determinato utente, mentre nel momento in cui si effettua un esercizio competitivo questa possibilità viene rimossa per fare in modo che le soluzioni di diversi utenti non prendano ispirazione l'uno dall'altro.

5 Implementazione

In questo capitolo verranno trattate le principali tecnologie utilizzate nella fase di implementazione del progetto sia per quanto riguarda il lato backend sia per quanto riguarda il lato frontend. Il progetto è stato sviluppato ed implementato attraverso diversi strumenti proprietari di JetBrains, nello specifico IntelliJ per la parte backend, attraverso il linguaggio di programmazione Java, e Webstorm per la parte frontend, attraverso il linguaggio typescript.
Per il processo di versionamento del progetto è stata utilizzata la tecnologia Git insieme al servizio di hosting per repository Github.
La comunicazione tra backend e frontend avviene attraverso l'utilizzo di REST-API di cui verrà spiegato la struttura ed il funzionamento in dettaglio nel capitolo di Backend.

5.1 Frontend

Per quanto riguarda il lato frontend, nella fase preliminare di sviluppo sono stati installate ed utilizzate diverse tecnologie, tra cui ricordiamo:
Nodejs, un ambiente di runtime di Javascript, attraverso il quale è stato possibile installare ed aggiornare nuovi pacchetti all'interno del sistema, tra cui ricordiamo i principali framework:
Angular: tecnologia di sviluppo utilizzata la realizzazione del frontend
Electron: tecnologia utilizzata per la realizzazione di un backend "offline" attraverso cui realizzare l'applicativo.
Il progetto è stato avviato e generato attraverso un comando a linea di comando, nello specifico: ng new Tesi-Frontend
attraverso il quale è possibile creare un applicativo di default da cui partire per lo sviluppo di un generico progetto.
Durante tutto lo sviluppo del progetto è stata utilizzata una metodologia di sviluppo che prevede l'utilizzo di "componenti" e "servizi", dove i componenti rappresentano dei blocchi che compongono l'applicativo e che possono essere replicabili (ovvero inserire più istanze di uno stesso blocco) e modificabili. Un componente quindi si compone di una parte grafica "Html e css", e di una parte di programmazione "typescript".

Per quanto riguarda i servizi, i servizi sono classi di “supporto” che a differenza dei componenti non implementano un componente grafico ma che vengono utilizzati principalmente per questione di riusabilità del codice ed una migliore organizzazione del codice. Durante lo sviluppo della tesi i servizi sono stati utilizzati come strumento di comunicazione con il backend, in modo tale da separare la logica di renderizzazione di componenti grafici, dalla logica di comunicazione con componenti esterne ad angular, che prevedono quindi l’utilizzo di REST-API.
Lo sviluppo del lato frontend prevede lo sviluppo di due branch principali su github

- Master
- Offline

Il branch master consiste nel branch sviluppato originariamente per lo sviluppo di un applicativo online, che si interfacciasse quindi a dei servizi cloud, come AWS oppure dei servizi configurabili esternamente attraverso parametri, come ad esempio l’url da utilizzare per il microservizio di login, l’url del database con cui comunicare e così via.

Il branch offline consiste nel branch sviluppato per essere utilizzato con la tecnologia Electron, prevedendo quindi un architettura che permettesse la compilazione sulla macchina locale del giocatore.
Entrambi i branch condividono gran parte del codice, ed entrambi i branch possono essere utilizzati sia per un approccio a servizi esterni (online), sia per un approccio a servizi interni (offline), la differenza tra i due branch consiste nella generazione dell’applicativo, ovvero nel caso offline avremo una generazione di una desktop application mentre nel caso online avremo la generazione di una web application, e nei metodi propri del framework Electron che sono necessari in un approccio offline.
Nello specifico, in un approccio offline il frontend comunica con un applicativo node installato sulla macchina locale per offrire alcune funzionalità, come ad esempio la gestione degli esercizi attraverso un servizio di hosting github. Nello specifico caso il frontend invia un messaggio interprocesso all’applicativo electron, il cui applicativo electron utilizza la shell installata sul calcolatore del giocatore per clonare ed utilizzare gli esercizi recuperati da github.

5.1.1 Angular

5.1.1.1 Environment

Per lo sviluppo dell'applicativo angular sono state utilizzate le seguenti variabili di ambiente, che possono essere programmabili sia nel caso si stia provando l'applicativo in locale, sia nel caso in cui si stia provando l'applicativo nella versione online.
Path: src/environments/environment.prod.ts

```
export const environment : {production: boolean, userServ...  = {
  production: true,
  userServiceUrl: "http://localhost:8081",
  tokenServiceUrl: "http://localhost:8082",
  compilerServiceUrl: "http://localhost:8083",
  exerciseServiceUrl: "http://localhost:9090",
  compilerPath: "/src/external/compiler/tesi",
  leaderboardServiceUrl: "http://localhost:8087"
};
```

5.1.1.2 Routes

Per poter parlare del routing, è necessario parlare delle Single Page Applications, ovvero delle singole pagine HTML, che caricano e si aggiornano dinamicamente in base all'interazione con l'utente piuttosto che caricare interamente nuove pagine dal server. Questo approccio ci permette di evitare il caricamento tra lo scorrere di pagine, rendendo il comportamento dell'applicazione web molto simile ad una Desktop application. L'angular router è uno strumento presente in Angular, che permette di creare le Single Page Applications (SPA), andando a creare diversi URL in base ai contenuti mostrati nell'applicazione.

L'angular route è un oggetto di configurazione che definisce una singola Route. Per definire tutte le varie "Routes", ovvero le configurazioni per l'angular router

service, è necessario definire un array di Route Objects, nello specifico caso l'applicativo possiede i seguenti routes:

Path: src/app/app-routing.module.ts

```
const appRoutes: Routes = [
  { path: 'auth', component: AuthComponent },
  { path: 'home', component: HomeRouteComponent },
  { path: 'refactor-game', component: RefactoringGameExListRouteComponent },
  { path: 'check-game', component: CheckGameExListRoute},
  { path: 'refactor-game/:exercise', component: RefactoringGameCoreRouteComponent},
  { path: 'check-game/:exercise', component: CheckGameCoreRouteComponent},
  { path: 'refactor-game/leaderboard/:exercise', component: LeaderboardRouteComponent},
  { path: 'settings', component: SettingsRouteComponent},
  { path: "**", redirectTo:  'home'}
];
```

In seguito verranno descritte le principali Routes:

5.1.1.2.1 Auth Route

La route di autenticazione ha il compito di esporre i principali componenti di autenticazione che vengono rappresentati attraverso il componente "Auth Component".

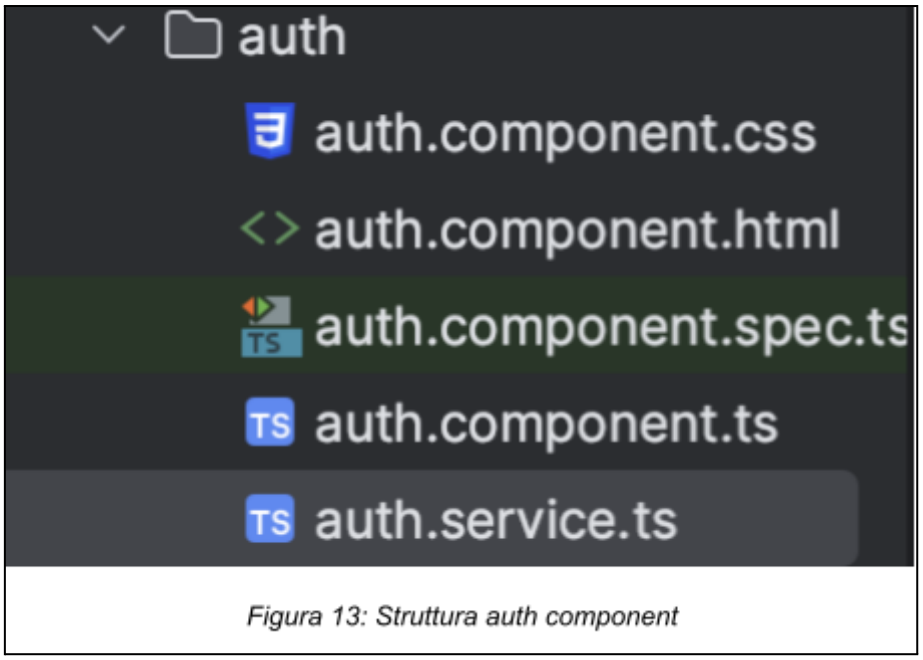


Figura 13: Struttura auth component

Affinché l'utente possa accedere alle funzionalità offerte dall'applicativo, è necessario effettuare un processo di autenticazione che prevede l'utilizzo di due informazioni per l'identificazione di un utente:

Email

Password

La realizzazione del form di login è avvenuta attraverso l'API Angular NgForm appartenente al pacchetto di strumenti che possiamo ottenere importando la libreria angular FormsModule. Attraverso l'api NgForm è possibile gestire in modo agevole i processi di validazione input e di tenere traccia delle informazioni aggregate di email e password per poi permetterne una gestione dal lato di scripting.

```
<form #authForm="ngForm" (ngSubmit)="onSubmit(authForm)">
 <div *ngIf="fail1" class="alert alert-danger"
role="alert">
   Wrong Email or Password
 </div>
 <div *ngIf="fail2" class="alert alert-danger"
role="alert">
   Server has a problem
 </div>
 <div class="form-group">
   <label for="email">E-Mail</label>
   <input
     type="email"
     id="email"
     class="form-control"
     ngModel
     name="email"
     required
     email
   />
 </div>
 <div class="form-group">
```

```
    <label for="password">Password</label>
    <input
      type="password"
      id="password"
      class="form-control"
      ngModel
      name="password"
      required
      minlength="6"
    />
  </div>
  <div>
    <button
      class="btn btn-primary"
      type="submit"
      [disabled]="!authForm.valid"
    >
      {{ isLoginMode ? 'Login' : 'Sign Up' }}
    </button>
    |
    <button class="btn btn-primary" (click)="onSwitchMode()"
type="button">
      Switch to {{ isLoginMode ? 'Sign Up' : 'Login' }}
    </button>
```

In questo caso, possiamo notare che il processo di validazione avviene attraverso gli strumenti proposti da FormsModule, nello specifico possiamo notare parole chiave nei vari input form come “email”, “required”, “minLength” che ci permettono di applicare dal punto di vista frontend un livello di validazione che permette di attivare il pulsante di Login/Sign Up solo se tutti i requisiti di validazione sono soddisfatti.

Il form di login è stato pensato in modo tale da essere utilizzato sia per la fase di ‘Sign Up’ che la fase di ‘Login’, in modo tale che premendo un pulsante, che scatena l’azione

“onSwitchMode”, le API di comunicazione col backend vengono cambiate dinamicamente al cliccare di questo tasto.
Di default avremo quindi una situazione in cui il componente si riferirà alle API di Login, se invece verrà cliccato il pulsante “Switch to Sign Up”, il form resterà identico, i campi non verranno cambiati e allo stesso tempo cambierà solamente la scritta “Switch to “Sign Up”, che in questo caso diventerà “Switch to Login”, e le che il componente utilizzerà non saranno più quelle di Login ma di Registrazione di un nuovo utente.
Sono state utilizzate inoltre direttive angular come NgIf che ci permettono di stampare a schermo messaggi di errore nel caso in cui il server abbia problemi di comunicazione con il server oppure nel caso in cui l’email e la password fornite siano errate.

Per quanto riguarda il salvataggio della sessione, è gestito attraverso un token che viene generato dal backend al momento dell’autenticazione. Il token viene salvato quindi all’interno della LocalStorage con chiave “UserData” e valore il token generato dal backend, in modo tale che al prossimo accesso dell’utente, nel momento in cui il giocatore accede all’applicazione non è necessario eseguire nuovamente l’autenticazione.
L’autenticazione automatica è infatti gestita attraverso il servizio auth.service.ts con la seguente funzione:

```
autoLogin() : void {
  // @ts-ignore
  this.userService.user.next( value: null);
  const email : string | null  = localStorage.getItem( key: "email");
  const token : string | null  = localStorage.getItem( key: "token");
  if(email == null || token == null)
    this.logout();
  else
    this.handleAuthentication(email, token);
}
```

All’interno dell’auth service è infatti presente una variabile user che all’occorrenza viene cambiata nel momento in cui avviene il login, in modo tale da poter recuperare tutti i dati agevolemente.
L’utente e il token di autenticazione dell’utente vengono conservati all’interno della local storage per permettere di conservare la sessione al prossimo accesso.

5.1.1.2.2 Exercise List Route

La route exercise list serve a permettere al giocatore di selezionare un esercizio presente tra quelli disponibili.

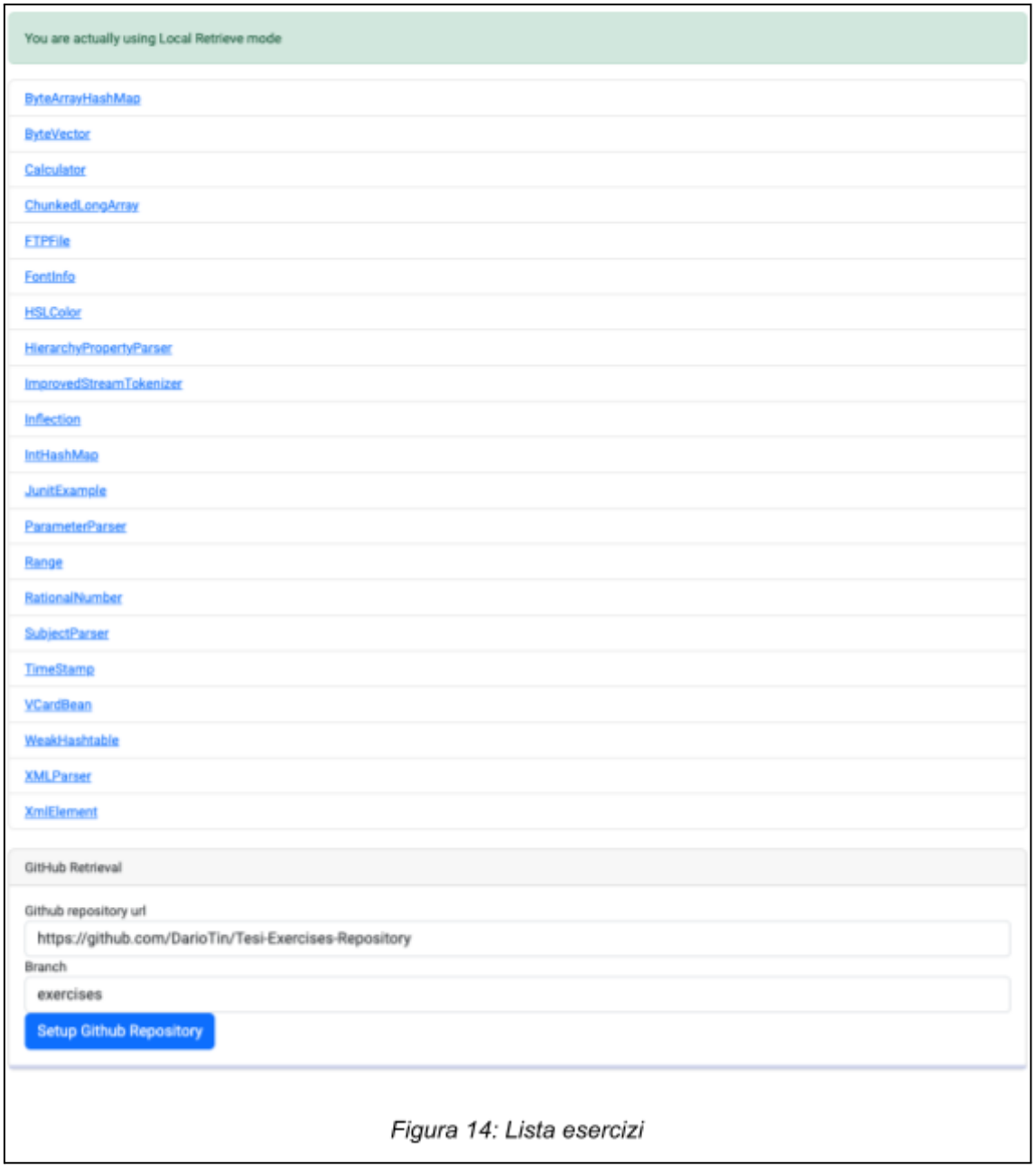


Figura 14: Lista esercizi

Il retrieve degli esercizi può avvenire attraverso due modalità: Github, Exercise service. Il retrieve tramite github è disponibile solamente per la Desktop Application, e permette attraverso un componente chiamato “Github-retriever” di scaricare in locale una repository github, recuperare gli esercizi dalla repository, e rappresentarli sotto forma di

lista all'interno della route.
Alternativamente è possibile effettuare il retrieve tramite il servizio Exercise service. In questo modo, attraverso una chiamata REST vengono recuperati tutti gli esercizi senza dover scaricare in locale la repository github.

5.1.1.2.3 Refactor Game Route

La route di refactoring game ci permette di rappresentare a schermo tutti gli elementi necessari per lo svolgimento di un esercizio. La seguente rotta utilizza il componente code editor per rappresentare un editor di testo.
Il code editor è un componente che permette la generazione di un code editor attraverso la libreria CodeMirror. CodeMirror è un componente web che permette di implementare un text editor con funzionalità utili alla scrittura di codice, come ad esempio la funzionalità di "undo" (ctrl + z), code highlighting e così via. CodeMirror è una libreria javascript modulare che in base al tipo di linguaggio che stiamo considerando possiamo importare diversi moduli che ci interessano, come ad esempio l'autosuggerimento e così via.
Il componente di code editor quindi è stato sviluppato in modo tale da avere due parametri

```
@Input() injectedCode = "";
@Input() editable = true;
```

con injectedCode si intende il codice che viene iniettato al componente CodeEditor attraverso le direttive angular html, in modo tale da renderizzare a schermo il codice correttamente formattato a seconda del linguaggio che stiamo considerando.
Quindi fondamentalmente inserendo il codice attraverso la variabile injected code in questo modo:

```
d.tintore
<app-codeeditor injectedCode="{{exerciseCode}}" #code [editable]="false"></app-codeeditor>
```

Possiamo ottenere sulla web application un code editor popolato con il codice presente nella variabile "exerciseCode".

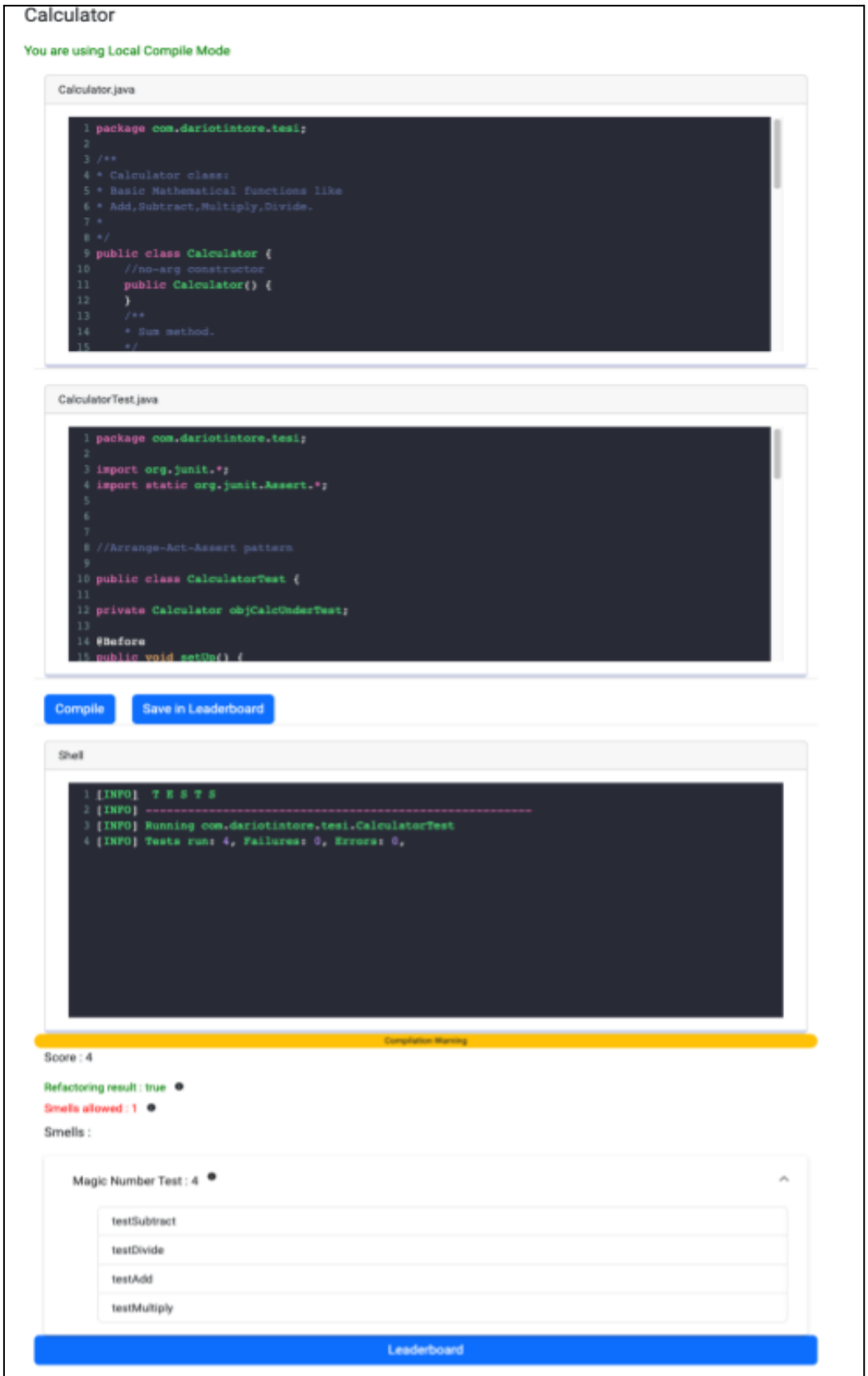


Figura 15: Esempio Refactoring Game

5.1.1.2.4 Check Game Route

La route check game è la route utilizzata per realizzare la seconda modalità di gioco prevista dall'applicativo che prevede un quiz relativamente ad un frammento di codice ed ai test smell.
La lista degli esercizi di tipo check game prevede di utilizzare un componente di "Exercise Retrieval" come avviene per la modalità refactoring game. Questo si traduce nel fatto che, per ogni esercizio di refactoring game, esiste un esercizio di check smell e viceversa. Gli esercizi vengono recuperati sempre dalla stessa repository. Le domande relative ad un esercizio sono configurate all'interno del file di configurazione json.
Il codice di inizializzazione della route infatti prevede di recuperare il file di configurazione e di popolare un insieme di domande e di risposte in rispettivamente due array.

```
questions !: Question[];
selectedAnswers: Answer[] = []
exerciseConfiguration!: ExerciseConfiguration
```

Partendo da questi dati è possibile quindi generare dinamicamente un questionario.

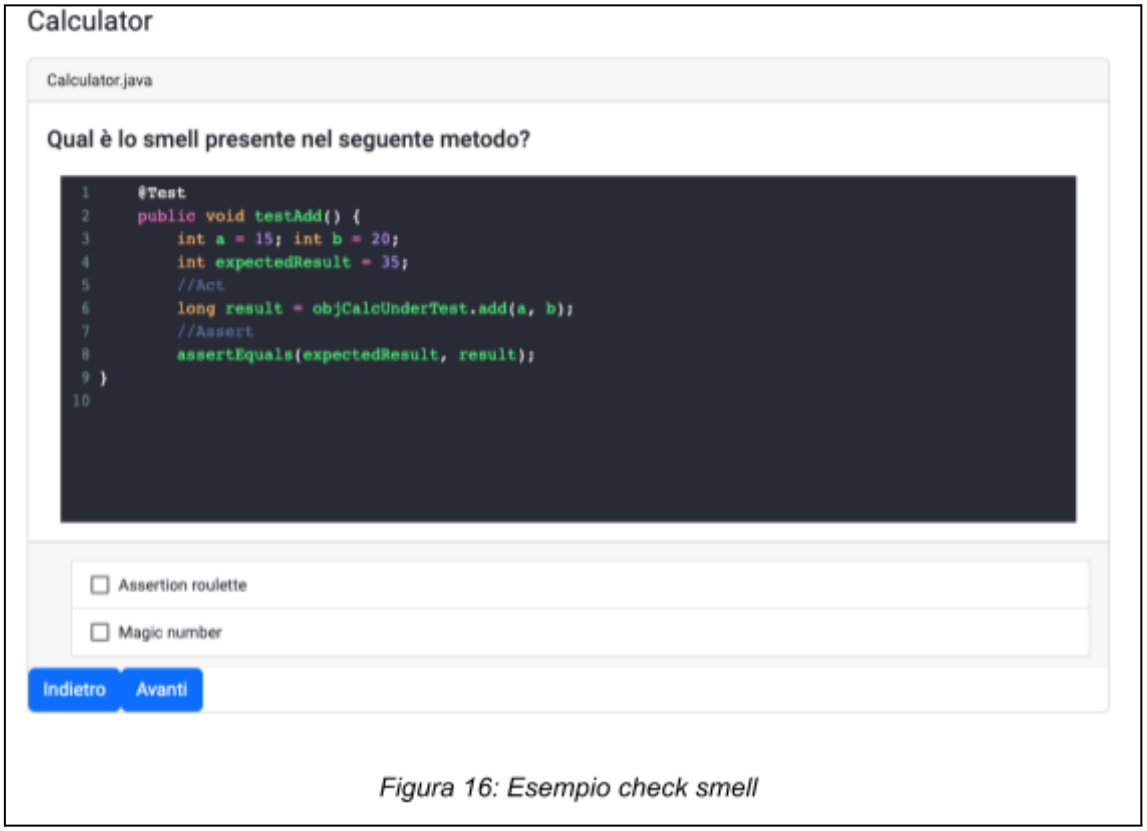


Figura 16: Esempio check smell

Leaderboard Route

La Leaderboard route è la route utilizzata per rappresentare la classifica di uno specifico esercizio. La leaderboard route quindi rappresenta un insieme di componenti chiamati “Solution”, dove una solution rappresenta una soluzione proposta da un giocatore, insieme alle statistiche di compilazione ottenute.

Il componente Solution è il componente utilizzato per definire una soluzione proposta all’interno della leaderboard. Solution quindi si compone di due code editor, uno per il codice di produzione, uno per il codice refattorizzato ed un insieme di commenti degli utenti che hanno commentato la soluzione in questione. Nel momento in cui stiamo renderizzando le soluzioni recuperiamo i dati attraverso il backend, nello specifico dall componente “leaderboard-service”. Il frontend angular quindi utilizza una chiamata Rest-API per recuperare i dati attraverso il servizio leaderboard.service.ts

```
getSolutionsByExerciseName(exercise: string) : Observable<Solution[]> {
  return this.http.get<Solution[]>( url: environment.leaderboardServiceUrl + '/leaderboard/' + exercise);
}
```

Dove il frontend passa semplicemente il nome dell’esercizio, e al frontend vengono restituite tutte le soluzioni che in questo caso corrispondono alle soluzioni degli altri giocatori che hanno già affrontato precedentemente lo stesso esercizio.

In questo caso il modello dati utilizzato per una soluzione è il seguente:

```
export class Solution {
  solutionId!: number;
  exerciseName!: string;
  playerName!: string;
  refactoredCode!: string;
  commentList!: SolutionComment[];
  creationTimestamp!: Date;
  upVotes!: number;
  downVotes!: number;
  score!: number;
  refactoringResult!: boolean;
  smells!: string;
```

Questo modello si rifà alla tabella delle soluzioni presenti all’interno del backend leaderboard-service, ed utilizza gli stessi campi per effettuare una conversione 1:1.

Il modello di un generico commento invece è definito come segue

```
export class SolutionComment {
  commentId!: number;
  commentAuthor!: string;
  commentText!: string;
  solutionId!: number;
  creationTimestamp!: Date;
```

Questo modello si rifà alla tabella dei commenti presenti all'interno del backend leaderboard-service, ed utilizza gli stessi campi per effettuare una conversione 1:1.

In seguito alla chiamata REST-API otterremo quindi un insieme di Soluzioni, attraverso le quali utilizzeremo la direttiva NgFor per generare e rappresentare dinamicamente le soluzioni che verranno recuperate di volta in volta.

```
<div *ngFor="let solution of solutions; let i = index">
  <app-solution
                playerName="{{solution.playerName}}"
                [comments]="solution.commentList"
                exerciseCode="{{testingCode}}"
                editedCode="{{solution.refactoredCode}}"
                [solutionId]=solution.solutionId
                creationTimestamp="{{solution.creationTimestamp}}"
                [upVotes]=solution.upVotes
                [downVotes]=solution.downVotes
                [score]="solution.score"
                [refactoringResult]="solution.refactoringResult"
                [smells] = "solution.smells"
                [isAutoValutative] = "this.isAutoValutative">
  </app-solution>
</div>
```

Vengono quindi iniettate, per ognuna delle proprietà interessate dal nostro componente, le informazioni che andiamo a recuperare attraverso la chiamata REST-API, in modo tale da ottenere un risultato simile:

Figura 17: Leaderboard

Ovvero che per ogni soluzione viene generata una vista che permette di visualizzare la rispettiva soluzione in modo agevole, di commentare e votare questa stessa soluzione. Tutte le soluzioni sono rappresentate sotto forma di accordion.

In questo caso bisogna considerare che, nel momento in cui è disattiva la modalità di auto-valutazione, il comportamento dell'applicativo cambia, in particolare nella modalità di refactor-game. La modalità auto-valutazione è destinata agli studenti che vogliono esercitarsi con l'applicativo ed imparare riguardo i vari test smell. Nel momento in cui la modalità di auto-valutazione è disattivata, si presuppone che il giocatore stia affrontando un test "ufficiale". Questo comporta il fatto che, nel momento in cui visualizza la classifica non vengono più mostrate le soluzioni degli altri giocatori, ma solamente informazioni relative al punteggio e l'ora di invio. Questo è stato pensato poiché in una modalità "ufficiale" il giocatore potrebbe "barare" considerando le risposte inviate dagli altri giocatori.

Esempio di leaderboard con auto-valutazione disattivata:

Figura 18: Leaderboard con autovalutazione false

5.1.1.2.5 Settings Route

La settings route è una rotta definita per impostare dinamicamente le variabili di ambiente ed i server di riferimento a runtime.
Nella versione Desktop è possibile impostare il tipo di compilazione, che può avvenire in cloud o in locale, ed il tipo di exercise retrieval, che può avvenire tramite github (locale) oppure tramite il rispettivo microservizio (exercise service).
Nella versione browser la compilazione in locale ed il retrieve tramite github sono disabilitati.

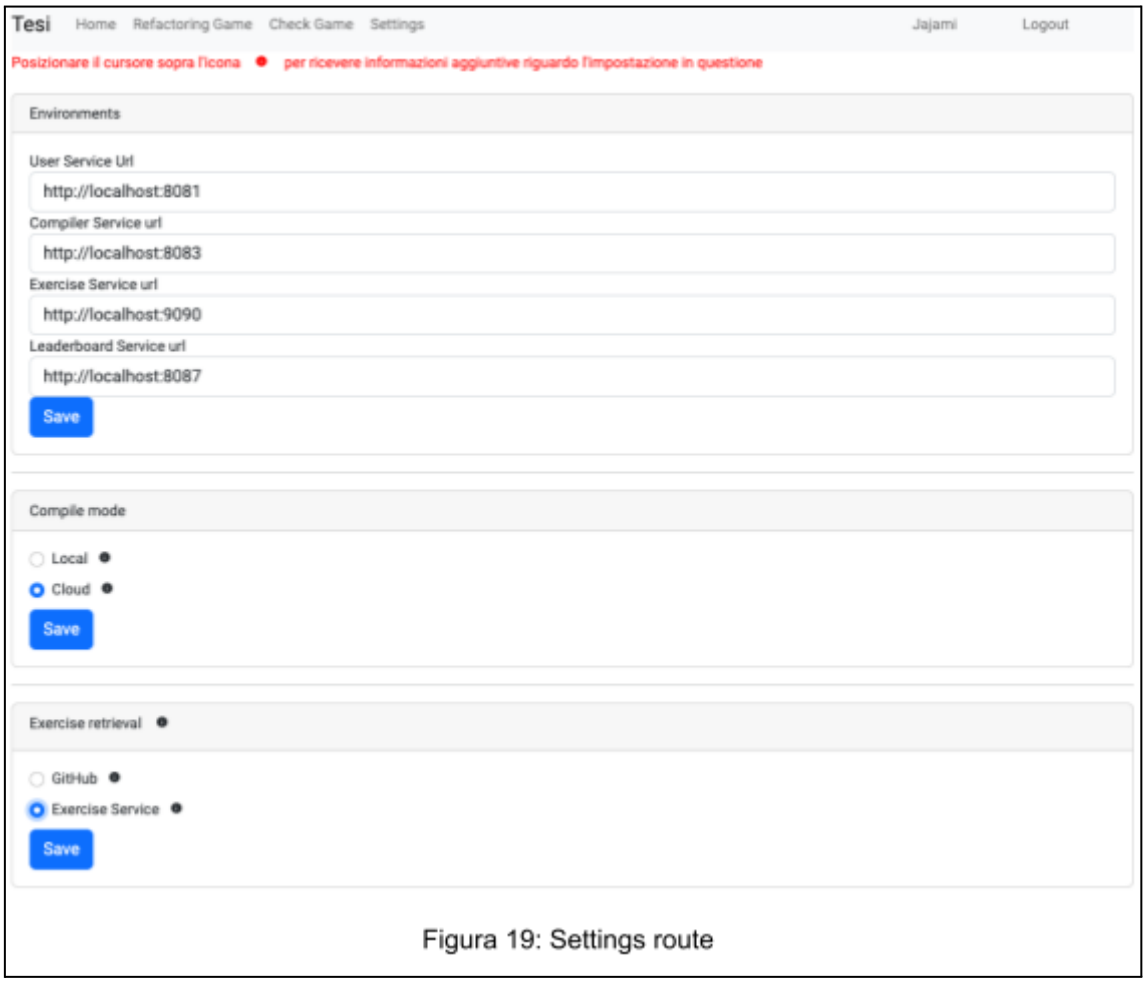


Figura 19: Settings route

5.1.2 Electron

Con electron si intende un framework per sviluppare applicazioni desktop utilizzando tecnologie web. Gli applicativi sviluppati con electron sono embedded con Chromium, un browser open source su cui è basato il noto browser Google Chrome.
Gli applicativi electron vengono eseguiti all'interno di un ambiente Node.js, il che vuol dire che è possibile importare e creare altri moduli installabili attraverso il package manager npm. Nello specifico bisogna considerare il modulo BrowserWindow proprio del pacchetto electron che rappresenta il processo principale e che ci permette di gestire e creare l'applicativo desktop che visualizziamo a schermo.
Nel momento in cui si vuole iniettare l'applicativo web sviluppato con angular, è necessario quindi buildare quest'ultimo, e caricare i file generati all'interno di una variabile mainWindow

```
mainWindow.loadURL(
  url.format( urlObject: {
    pathname: path.join(process.env.ROOT_PATH, `/dist/dariotintore_frontend/index.html`),
    protocol: "file:",
    slashes: true
  })
);
```

in seguito, attraverso mainWindow è possibile creare il BrowserWindow che consiste nel processo principale.

```
function createWindow () : void  {
  mainWindow = new BrowserWindow({
    width: 1920,
    height: 1080,
    webPreferences: {
      nodeIntegration: true,
      enableRemoteModule: true,
      contextIsolation: false,
      devTools: true
    }
  })
```

Una delle caratteristiche più importanti per quanto rigaurda electron è la Inter-Process Communication (IPC).
Attraverso la IPC è possibile realizzare il funzionamento dell'applicativo sulla macchina locale. Nello specifico con chiamate IPC avviene la comunicazione tra Electron e la shell locale, quindi il funzionamento si basa attraverso uno scambio di messaggi tra Angular ed Electron, dove Angular ha il compito di renderizzare graficamente l'applicativo, mentre electron gestisce il file system e la shell, ed esegue attraverso dei comandi locali gli applicativi necessari allo svolgimento di un esercizio.
Esempio di comunicazione tra Angular ed Electron per il recupero degli esercizi da repository github scaricata in locale:

```
this._electronService.ipcRenderer.send( channel: 'getProductionClassFromLocal',exercise);
```

Come prima cosa avviene un messaggio da angular ad electron con etichetta "getProductionClassFromLocal"
In seguito l'applicativo electron cattura il messaggio grazie alla stessa etichetta ed invia nuovamente un messaggio al frontend con etichetta "receiveProductionClassFromLocal"

```
ipcMain.on( channel: 'getProductionClassFromLocal',  listener: async(event : IpcMainEvent ,data) : Promise<void> => {
  let result : string  = await repo.getProductionFilesFromLocal(data);
  mainWindow.webContents.send( channel: 'receiveProductionClassFromLocal', result)
})
```

Ed infine per concludere il giro, l'applicativo angular riceve la risposta da electron attraverso la stessa etichetta utilizzata da electron "receiveProductionClassFromLocal"

```
// GET PRODUCTION CLASS FROM ELECTRON
this._electronService.ipcRenderer.on( channel: 'receiveProductionClassFromLocal',
   listener: (event : Electron.IpcRendererEvent ,data) : void =>{
  this.zone.run(  fn: () : void => {
    this.userCode = data
    this.originalProductionCode = data
  })
})
```

Struttura applicativo electron

Per quanto riguarda la struttura dell'applicativo electron, che consiste in tutti i file che ci permettono di eseguire l'applicativo, possiamo considerare il seguente albero:

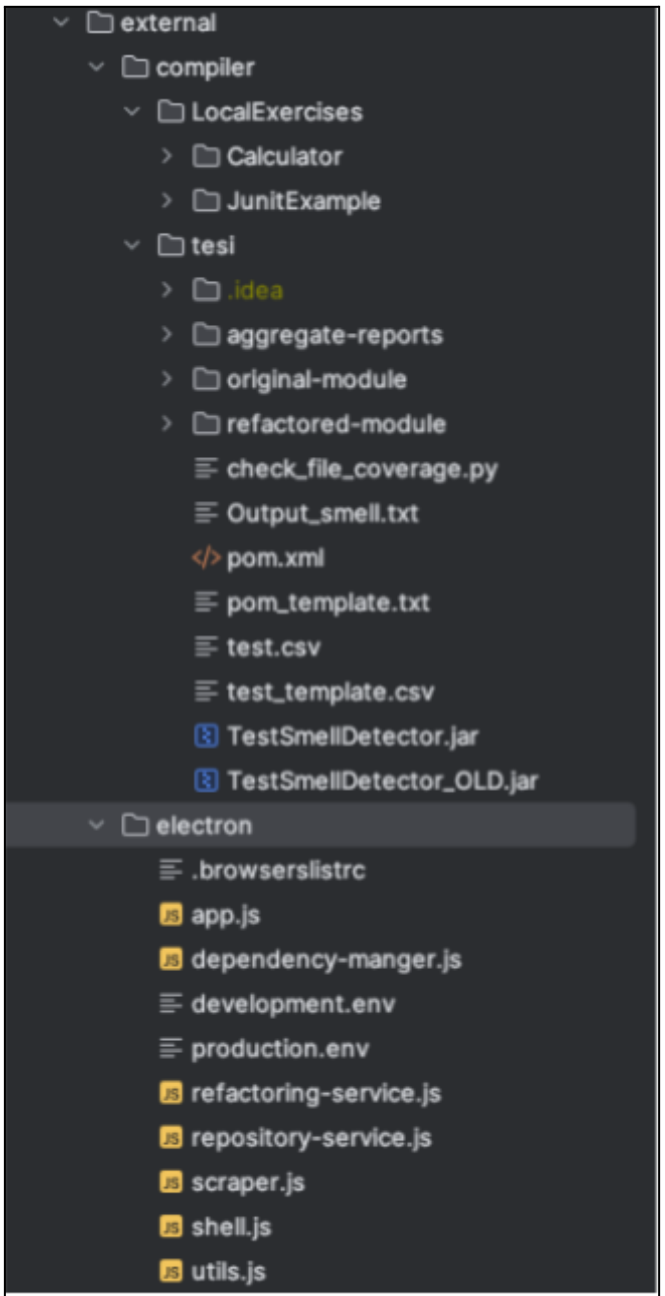


Figura 20: Struttura electron

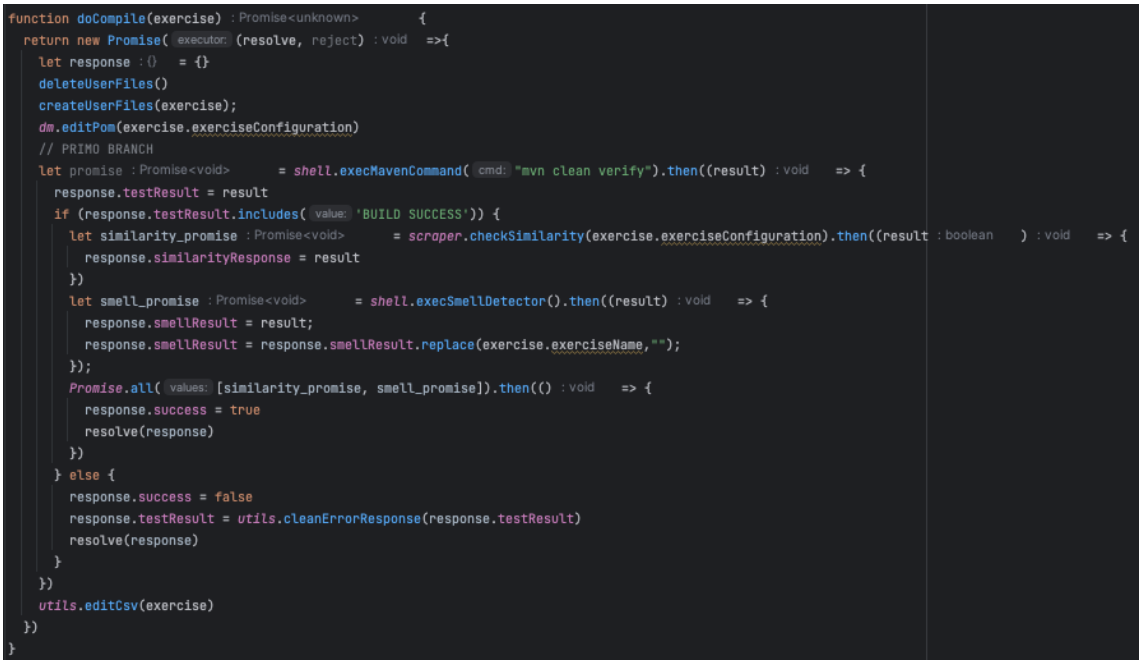
Dove external consiste nella cartella contenente tutti i file "esterni" alla web application, ed in cui possiamo trovare la sezione "Compiler", corrispondente al servizio Junit-service nell'architettura online, ed "Electron", che si compone dei file javascript

che ci permettono sia di gestire il file system, sia di gestire la shell, sia di gestire tutte le altre funzionalità.
In seguito vengono elencati i file ed i obiettivi:

app.js : file javascript che consiste nel main dell'applicativo electron, ci permette di creare una finestra per la web application, importare le principali dipendenze utili al funzionamento dell'intero applicativo.

dependency-manager.js : file javascript che permette la gestione dinamica delle dipendenze del file pom.xml. All'interno del dependency-manager abbiamo due metodi. Il primo che consideriamo (`getRefactoringGameConfiguration`) ci permette di recuperare le dipendenze di un esercizio, che in questo caso sono salvate all'interno della cartella "Local Exercises", ovvero la cartella che viene creata e popolata nel momento in cui tramite angular avviene il download degli esercizi tramite repository github.
Il secondo metodo (`editPom`) effettua invece la modifica vera e propria del file pom.xml all'interno della cartella compiler, che contiene il progetto che stiamo considerando nel momento in cui avviamo un esercizio.

refactoring-service.js : file javascript che rappresenta il cuore alla base dell'esercizio di refactoring test. All'interno del refactoring-service è presente il metodo doCompile che riceve l'oggetto "exercise", il quale oggetto rappresenta un'istanza di un esercizio di refactoring ricevuta attraverso il frontend, ed effettua a sua volta tutte le modifiche al file system, l'esecuzione dello smell detector e degli script ulteriori. Il metodo DoCompile:



```
function doCompile(exercise) : Promise<unknown>        {
  return new Promise( executor: (resolve, reject) : void   =>{
    let response : {}   = {}
    deleteUserFiles()
    createUserFiles(exercise);
    dm.editPom(exercise.exerciseConfiguration)
    // PRIMO BRANCH
    let promise : Promise<void>       = shell.execMavenCommand( cmd: "mvn clean verify").then((result) : void    => {
      response.testResult = result
      if (response.testResult.includes( value: 'BUILD SUCCESS')) {
        let similarity_promise : Promise<void>       = scraper.checkSimilarity(exercise.exerciseConfiguration).then((result : boolean    ) : void    => {
          response.similarityResponse = result
        })
        let smell_promise : Promise<void>       = shell.execSmellDetector().then((result) : void    => {
          response.smellResult = result;
          response.smellResult = response.smellResult.replace(exercise.exerciseName,"");
        });
        Promise.all( values: [similarity_promise, smell_promise]).then(() : void    => {
          response.success = true
          resolve(response)
        })
      } else {
        response.success = false
        response.testResult = utils.cleanErrorResponse(response.testResult)
        resolve(response)
      }
    })
    utils.editCsv(exercise)
  })
}
```

Si occupa quindi di:

1. Cancellare i file dell'esercizio precedente (se presente) all'interno della cartella Compiler
2. Creare i file dell'esercizio che vogliamo creare all'interno della cartella Compiler
3. Modificare il pom.xml con le dipendenze richieste dall'esercizio
4. Avviare la maven clean verify per verificare eventuali errori di compilazione ed avviare lo strumento JaCoCo
5. Avviare il check-refactor
6. Eseguire lo Smell Detector
7. Restituire il risultato sotto forma di Json Object in modo che il frontend possa poi elaborarlo.

Repository-service.js: file javascript che offre diverse funzioni che ci permettono di gestire gli esercizi attraverso GitHub. Il seguente file ci permette quindi di scaricare con una git clone gli esercizi dal servizio di hosting, e di estrarre dai file di configurazione dell'esercizio richiesto tutte le opzioni in base alla tipologia di esercizio che stiamo considerando.

Esempio di esecuzione della funzionalità di clone presente nel repository-service:

```
async function cloneRepository(data) : Promise<void>  {
  utils.deleteDirectoryFiles( path: process.env.ROOT_PATH + "src/external/compiler/LocalExercises")
  await sh.execGitCloneCommand(data._branchName, data._url)
}
```

scraper.js: file javascript che effettua il processo di data-scraping dai file autogenerati da jacoco all'esecuzione del comando "maven clean verify". Nel momento in cui lo strumento di jacoco viene eseguito, vengono generati dei report per tre progetti contemporaneamente:

1) Il progetto originale, contenente quindi la classe java di produzione con il test di partenza non modificato.
2) Il progetto refattorizzato, contenente quindi la classe java di produzione con il test di partenza refattorizzato dall'utente.
3) L'aggregazione dei due progetti (jacoco-aggregate)

Tutti questi elementi sono necessari per comprendere la natura del refactoring, ed in tutti e tre i progetti viene generato un file di report riguardo la copertura del codice di test della classe di testing sulla classe di produzione.

il processo di data-scraping avviene quindi estraendo dai report html che possiamo trovare in:

{nome-progetto}/target/site/jacoco/index.html

Quindi prendendo i tre file html generati in Original-module, Refactored-module e Aggregate-module è possibile quindi estrarre e calcolare la differenza insiemistica attraverso la quale viene considerato il refactoring check.

shell.js: file javascript che offre funzioni che permettono di eseguire una shell sul calcolatore locale, nello specifico abbiamo i seguenti metodi:

execMavenCommand(cmd) : esegue un comando maven con riferimento al pom presente nella cartella compiler
execSmellDetector() : esegue lo smell detector all'interno della cartella compiler
execGitCloneCommand(url,branch) : esegue l'operazione di clone git clone in locale per l'esecuzione degli esercizi
execGenericCommand(cmd) : esegue un generico comando passato come parametro

utils.js: file javascript che contiene funzionalità generiche che vengono utilizzate all'interno del progetto, come ad esempio la scrittura su file, o la cancellazione di un file.

5.2 Backend

Nella fase di sviluppo del Backend è stato pensato di realizzare un'architettura a microservizi, i quali microservizi sono indipendenti l'uno dall'altro, e con cui è possibile comunicarvi attraverso delle REST-API.
Le motivazioni che hanno portato alla scelta di utilizzare un'architettura a microservizi per la parte di Backend riguarda il fatto che i microservizi possono essere facilmente containerizzati con Docker, facilmente distribuiti, fornendo una singola responsabilità ad ogni modulo e facilitando l'esposizione di un API comprensibile e flessibile.
Un'altra motivazione per cui è stato scelto di utilizzare un'architettura a microservizi è stata quella di rendere l'applicativo il più modulare ed intercambiabile possibile, un esempio possiamo trovarlo nello scenario in cui viene utilizzata sia la versione Desktop che la versione Browser, dove grazie ad un'architettura a micro servizi è possibile contattare solo il microservizio di autenticazione per poter accedere all'applicativo, senza sovraccaricare il resto dell'infrastruttura, rendendo l'esperienza di gioco degli utilizzatori dell'applicativo versione browser più fluida.
Lo sviluppo del backend è stato pensato in modo tale da poter essere adattato a diversi casi d'uso, dove ogni caso d'uso può essere rappresentato da un branch della rispettiva repository, si può pensare infatti ad uno scenario in cui ad ogni repository è associato un cloud differente (AWS, Azure ecc…).
La natura dell'applicativo è stata pensata per due contesti (online/locale), i quali contesti tra di loro contrapposti sviluppando il tutto in modo tale da poter rendere le componenti riutilizzabili.
Per quanto riguarda il lato backend, nella fase preliminare di sviluppo sono state installate ed utilizzate diverse tecnologie, tra cui ricordiamo:

Java: La scelta dietro la tecnologia Java per quanto riguarda il backend è dovuta dall'indipendenza del sistema operativo, un'alta scalabilità ed un insieme di framework che rendono il processo di sviluppo molto più agevole.

Spring Boot: Framework utilizzato per lo sviluppo di microservizi con tecnologia Java

Docker: Piattaforma che permette di sviluppare, testare e distribuire prodotti software con maggiore facilità ed indipendenza dalla macchina di riferimento

Jenkins: Strumento di sviluppo per un approccio al CI/CD, attraverso lo sviluppo di Pipeline che prevedono un insieme di step ad ogni versione rilasciata del software.

Sonarqube: Strumento di analisi statica che ha permesso di sviluppare codice pulito e sicuro

Nodejs: Piattaforma di sviluppo utilizzata per lo sviluppo del Compiler service.

5.2.1 Spring Boot

Al fine di poter descrivere i micro servizi sviluppati per il backend risulta necessaria un'introduzione su Spring Boot ed i suoi componenti tipici.
Tutti i micro servizi sono stati sviluppati utilizzando la tecnologia Spring Boot.
Per Spring Boot si intende un framework che contiene un insieme di strumenti raggruppati per favorire lo sviluppo di micro servizi ed applicazioni web. Tra le tante tecnologie presenti in Spring Boot ricordiamo Hibernate.
Hibernate permette l'utilizzo della pratica di sviluppo Object-Relational Mapping (ORM). Attraverso l'utilizzo della pratica di programmazione ORM viene facilitata l'integrazione tra la programmazione orientata agli oggetti e i database relazionali.Un esempio dell'integrazione citata precedentemente possiamo notarla in questo breve

esempio:

```java
@Entity
@Table(name = "user")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "userid")
    private Long userId;
    5 usages
    @Column(name = "firstname")
    private String firstName;
    5 usages
    @Column(name = "lastname")
    private String lastName;
    5 usages
    @Column(name = "email")
    private String email;
    4 usages
    @Column(name = "password")
    private String password;
```

Il seguente esempio considera l'entità user presente all'interno del microservizio user-service, ovvero una classe java che viene trasposta all'interno del database nello stesso identico modo in cui è stata definita all'interno dell'applicativo java, senza necessitare quindi operazioni come la creazione di una tabella e così via.
All'esecuzione del microservizio user-service infatti verrà creata automaticamente la tabella user con tutti i campi definiti all'interno della classe java come mostrato nell'esempio precedente.

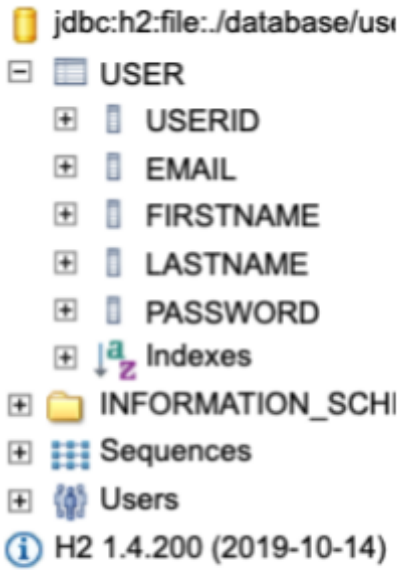


Figura 21: Database user service

5.2.1.1 Componenti

Nei micro servizi sviluppati è stato utilizzato il pattern Controller-Service-Repository, che ci permette di implementare il pattern “Separation of Concerns”.

Rest controller: un REST Controller è un componente di Spring Boot che ci permette di accedere all’applicativo dall’esterno tramite degli endpoint da raggiungere attraverso applicativi esterni. Il componente controller è il primo componente al livello più alto del pattern “Controller-Service-Repository” poiché espone le funzionalità che vengono utilizzate dai servizi esterni. Un esempio in particolare può essere il seguente:

1) Definiamo un endpoint all’interno del microservizio exercises-service che ci permette di recuperare tutti gli esercizi disponibili dal backend, mettendo a disposizione una route al seguente indirizzo “http://localhost:8080/exercises
2) Il frontend effettua una richiesta GET all'indirizzo “http://localhost:8080/exercises”
3) Il backend riceve la chiamata e restituisce gli esercizi
4) Il frontend riceve la risposta del backend e renderizza a schermo ciò che ha ottenuto dal backend.

All’interno del controller possiamo avere diverse annotazioni che ci permettono quindi di definire degli endpoint in base al metodo HTTP che richiediamo.
@GetMapping : espone un endpoint con metodo HTTP Get
@PostMapping : espone un endpoint con metodo HTTP POST
@DeleteMapping : espone un endpoint con metodo HTTP Delete
@PutMapping : espone un endpoint con metodo HTTP Put
@PatchMapping : espone un endpoint con metodo HTTP Patch

Service: Per service si intende un componente Spring che viene utilizzato per implementare la Business Logic. Il componente Service è presente ad un livello intermedio all’interno del pattern C-S-R ed è invocato dal Controller. Il flusso quindi prevede:

1) Il backend riceve la chiamata dall’esterno e cattura questa chiamata attraverso l’endpoint.
2) Il metodo definito all’interno del controller chiama il servizio che effettua le operazioni necessarie.
3) Il metodo del servizio chiama un repository service per implementare business logic
4) Il controller restituisce il risultato ottenuto dal return del metodo del servizio.

Repository: Il componente repository ci permette di gestire ed effettuare operazioni sui dati, quindi sullo storage. Il repository ci permette quindi di interfacciarsi con il Database ed eseguire query. Il flusso con il componente Repository può essere il seguente:

1) Il backend riceve la chiamata dall’esterno e cattura questa chiamata attraverso l’endpoint.
2) Il metodo definito all’interno del controller chiama il servizio che effettua le operazioni necessarie.
3) Il metodo del servizio chiama un repository service per implementare business logic
4) Il service chiama i metodi presenti sul repository service che ci permettono di gestire i dati presenti sul database

5) Il controller restituisce il risultato ottenuto dal return del metodo del servizio.

Entity: per entity si intende un componente che rappresenta il modello dell'informazione dei dati che stiamo manipolando sul database

5.2.2 Microservizi

Per quanto riguarda lo sviluppo dei microservizi è stato utilizzato il design pattern Data Transfer Object. Il Design Pattern DTO è utilizzato in sistemi software per limitare il quantitativo di informazioni esposte dal microservizio riguardo una specifica entità, quindi senza dover necessariamente mappare una classe con una tabella del Database, permettendoci di recuperare solamente le informazioni necessarie al caso d'uso specifico. L'utilizzo del design pattern DTO è stato suggerito dallo strumento di analisi statica Sonarqube per motivi di sicurezza del codice Spring boot.

5.2.2.1 Database

Siccome ogni microservizio avrà un proprio database, e non tutti i Database saranno relazionali (Ad esempio quello degli esercizi), avremo uno scenario in cui i diversi database saranno separati tra loro, la comunicazione tra questi avverrà quindi attraverso delle foreign key, che in questo caso sono definite attraverso la notazione di sottolineatura, mentre le chiavi primarie saranno definite attraverso uno stile grassetto.

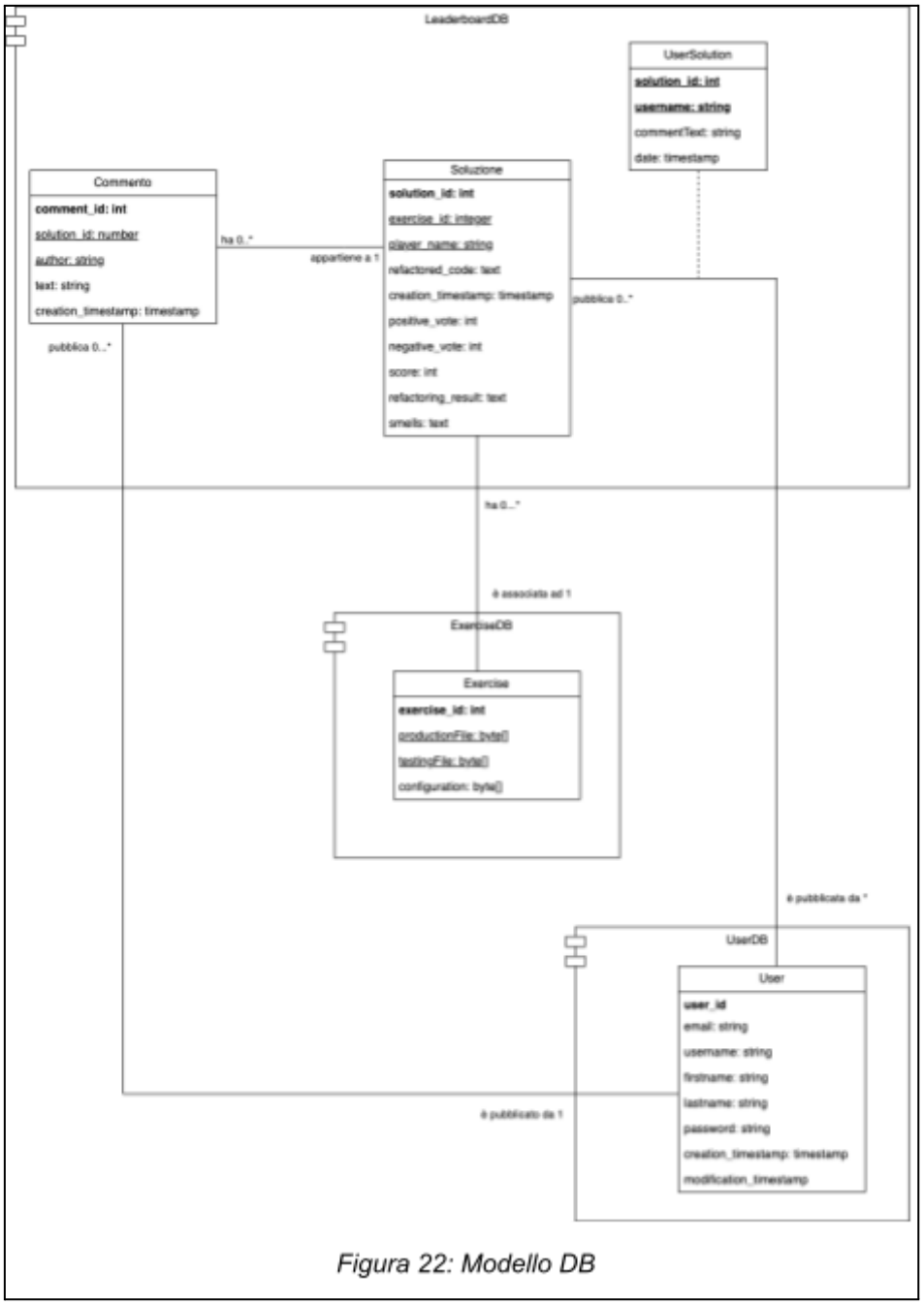


Figura 22: Modello DB

5.2.2.2 User Service

Lo User service consiste nel microservizio che fornisce le principali funzionalità per la gestione degli utenti, quindi di login/sign up per giocatori. La porta utilizzata per il microservizio è 8081 ed espone i seguenti endpoint:

```
@PostMapping("/")
public ResponseEntity<JSONObject> saveUser(@RequestBody
AuthUserDTO user) {
    return userService.saveUser(user);
}
```

Endpoint per salvare un utente sul database dopo aver effettuato la criptazione della password

```
@GetMapping("/{id}")
public Optional<UserModelDTO> getUserById(@PathVariable
Long id) {
    return userService.getUserDTOById(id);
}
```

Endpoint che utilizza una PathVariable per recuperare i dati di un utente attraverso il suo id univoco

```
@GetMapping("/")
public List<UserModelDTO> getAllUsers() {
    return userService.getAllUsersDTO();
}
```

Endpoint che permette di recuperare i dati di tutti gli utenti registrati sul database, utilizzato per fini di debugging.

```
@DeleteMapping("/")
public ResponseEntity<JSONObject> deleteUser(@RequestBody
AuthUserDTO user) {
    return userService.deleteUserById(user);
}
```

Endpoint che utilizza una PathVariable per cancellare un utente dal database attraverso il suo id univoco.

```
@PostMapping("/authenticate")
public ResponseEntity<JSONObject> login(@RequestBody
AuthUserDTO user) {
    return userService.login(user);
}
```

Endpoint che permette di effettuare le operazioni di autenticazione attraverso un token

Il microservizio in questione prevede l'utilizzo di funzionalità di crittografia per la registrazione di un utente. Le password infatti sono cifrate attraverso il componente di Springboot security BCrypt.

```
5.2.2.3 import org.springframework.security.crypto.bcrypt.BCrypt;
```

5.2.2.3 Leaderboard service

Il microservizio Leaderboard-service consente di tenere traccia delle partite giocate dai giocatori e di gestire i dati relativi alla leaderboard/commenti. La porta utilizzata per il microservizio è 8087 ed espone i seguenti endpoint:

```
@GetMapping("/{exerciseName}")
public List<Solution> getSolutionsByExercise(@PathVariable("exerciseName") String exerciseName){
  return solutionService.getSolutionsByExerciseName(exerciseName);
}
```

Endpoint per ottenere le soluzioni associate ad uno specifico esercizio. Il seguente endpoint viene chiamato dal frontend nel momento in cui il giocatore vuole accedere alla leaderboard di un esercizio.

```
@PostMapping("/postComment")
public void postCommentByExerciseName(@RequestBody Comment comment){
  commentService.saveCommentForExerciseName(comment);
}
```

Endpoint definito per permettere ad un utente di postare un commento relativo ad una soluzione. Il seguente endpoint viene chiamato dal frontend nel momento in cui un giocatore effettua il submit di un commento sotto una soluzione

```
@PostMapping("/")
public void saveSolutionByExercise(@RequestBody SolutionSaveRequestDTO solution){
  solutionService.saveSolutionForExerciseName(solution);
}
```

Endpoint definito per permettere ad un utente di pubblicare la propria soluzione nel momento in cui esegue un esercizio. Il seguente endpoint viene richiamato dal frontend allo svolgimento di un esercizio.

```
@PostMapping("solution/{solutionId}")
public void voteSolution(@PathVariable("solutionId") Long id, @RequestBody JSONObject body){
  solutionService.voteSolution(id, body);
}
```

Endpoint definito per permettere ad un utente di votare una soluzione. Il seguente endpoint viene chiamato da un giocatore nel momento in cui quest'ultimo clicca su un

pollice (su/giù) all'interno della leaderboard. Questo endpoint aggiorna automaticamente i voti relativi ad una specifica soluzione.

Per quanto riguarda le entity, ovvero i modelli dei dati utilizzati all'interno del leaderboard-service abbiamo due entities:
Solution:

```java
@Entity
@Table(name = "solution")
public class Solution {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  @Column(name = "solution_id")
  private Long solutionId;

  2 usages
  @Column(name = "exercise")
  private String exerciseName;
  2 usages
  @Column(name = "player_name")
  private String playerName;
  2 usages
  @Column(name = "refactored_code")
  private String refactoredCode;

  2 usages
  @OneToMany
  @JoinColumn(name="solution_id")
  private List<Comment> commentList;
```

Entità che consiste nel modello dati che rappresenta l'informazione relativa ad una soluzione di un giocatore per uno specifico esercizio. Ad una soluzione sono associati più commenti, e questa informazione è modellata attraverso un'associazione "OneToMany" fornita da Spring Boot.

```java
public class Comment {
  @Id
  @GeneratedValue(strategy = GenerationType.AUTO, generator="seq")
  @GenericGenerator(name = "seq", strategy="increment")
  @Column(name = "comment_id")
  private Long commentId;

  2 usages
  @Column(name = "author")
  private String commentAuthor;

  2 usages
  @Column(name = "text")
  private String commentText;

  2 usages
  @Column(name = "vote")
  private int commentVote;

  2 usages
  @Column(name = "solution_id")
  private Long solutionId;
```

Entità che consiste in un commento

5.2.2.4 Exercise service

Il microservizio Exercise-service consente di fornire una repository per gli esercizi che saranno utilizzati dai giocatori. Il seguente microservizio è stato definito per come supporto ulteriore per la distribuzione degli esercizi utilizzabili sull'applicativo. Il senso del seguente microservizio è il seguente: Se il microservizio exercise-service risponde, allora gli esercizi vengono presi attraverso una chiamata REST-API, altrimenti se il servizio non è attivo gli esercizi vengono recuperati attraverso GitHub.
Il seguente microservizio presenta degli esercizi salvati in locale in una cartella di nome "ExerciseDB", quindi le operazioni di retrieve e di creazione di esercizi si limitano alla creazione di file sul filesystem in cui è installato il microservizio. È stata pensata una soluzione in cui gli esercizi fossero salvati sotto forma di file su un database relazionale, tuttavia in corso di sviluppo è stata optata per una soluzione più "leggera" dal punto di vista computazionale, andando a minimizzare il numero di database che devono essere gestiti dall'intero applicativo. La porta utilizzata per il microservizio è 9090 ed espone i seguenti endpoint:

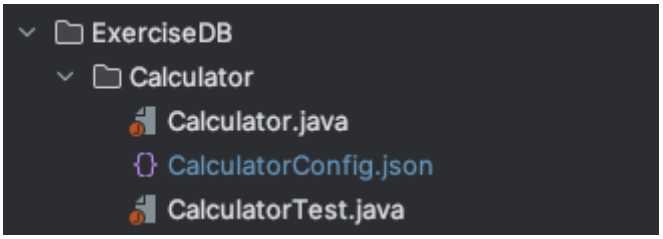
```
@GetMapping("/")
public ResponseEntity<List<String>> getListFiles() {
  return ResponseEntity.status(HttpStatus.OK).body(storageService.getAllFiles());
}
```

Endpoint che ci permette di recuperare tutti i nomi degli esercizi attualmente presenti sull'Exercise-service. La seguente chiamata è la chiamata che viene effettuata nel momento in cui l'applicativo frontend si trova nella schermata di exercise-list, sia per quanto riguarda la modalità di refactoring-test che per quanto riguarda la modalità check smell. In questo caso otteniamo quindi una lista di esercizi che verranno utilizzati dal frontend per comprendere il numero di esercizi attualmente disponibili.

```
@GetMapping("/{exercise}/{type}")
public ResponseEntity<byte[]> retrieveFile(
        @PathVariable String exercise,
        @PathVariable String type) throws IOException {
  byte[] file = storageService.getFile(exercise, type);

  return ResponseEntity.ok()
      .header(HttpHeaders.CONTENT_DISPOSITION, ...headerValues: "attachment; filename=\"" + exercise + "\"")
      .body(file);
}
```

Endpoint che permette di recuperare un file dalla cartella ExerciseDB. In questo endpoint abbiamo due parametri, il primo è il nome dell'esercizio che ci permette di selezionare l'esercizio attraverso il nome della cartella, il secondo parametro è il tipo di file che stiamo andando a cercare, ovvero un file di Test, di Produzione oppure di Configurazione.
La cartella ExerciseDB possiede la seguente struttura



Questo si traduce nel fatto che:
Una chiamata GET all'indirizzo http://localhost:9090/Calculator/Production Restituisce il file Calculator.java
Una chiamata GET all'indirizzo http://localhost:9090/Calculator/Test Restituisce il file CalculatorTest.java
Una chiamata GET all'indirizzo http://localhost:9090/Calculator/Config/ Restituisce il file CalculatorConfig.java

5.2.2.5 Compiler Service

Il microservizio compiler service è il componente che ci permette di compilare online e di svolgere di fatto gli esercizi di nostro interesse.
Questo microservizio è stato sviluppato attraverso la tecnologia Node.js, e risulta una trasposizione lato server di ciò che avviene con electron per la compilazione in locale. Questo si traduce nel fatto che il microservizio compiler-service ed i servizi electron sono tra loro estremamente simili.
Il framework utilizzato per lo sviluppo del microservizio è ExpressJS, che consiste in un framework estremamente leggero e minimalista, le cui caratteristiche si adattano bene allo scopo proposto dal nostro micro servizio che coincide in una compilazione il più veloce possibile.

Questo micro servizio è stato sviluppato in modo tale da gestire le richieste di compilazione attraverso una coda, dove le richieste vengono soddisfatte in modo sequenziale attraverso il pacchetto npm express-queue

```
app.use(helmet());
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));

app.use(queue( config: { activeLimit: 1, queuedLimit: 10 }));
app.use(cors());
```

Il microservizio in questione espone un unico endpoint atto alla compilazione di uno specifico esercizio:

```
app.post( path: '/compiler/refactoring', handlers: async (req : Request<P, ResBody, ReqBody, ReqQuery, LocalsObj> , res : Response<ResBody, LocalsObj> ) : Promise<void>  => {
    try{
        let result = await refactoring_service.doCompile(req.body)
        result.testResult = utils.cleanSuccessResponse(result.testResult)
        result.smellResult = utils.cleanSmells(result.smellResult);
        console.log(result);
        res.status( code: 200);
        res.json( body: {testResult: result.testResult, similarityResponse: result.similarityResponse, smellResult: result.smellResult, success: result.success})
        res.end();
    }catch(error){
        console.log(error);
    }
});
```

Il body richiesto per la corretta compilazione è composto dai seguenti campi:

1) exerciseName : Nome dell'esercizio che coincide con la cartella
2) originalProductionCode : Codice di produzione, contenuto nel file {{nomeEsercizio}}.java
3) originalTestCode : Codice di test, contenuto nel file {{nomeEsercizio}}Test.java
4) refactoredTestCode : Codice di test refattorizzato dal giocatore
5) exerciseConfiguration : Contenuto del file Json di configurazione presente nel file {{nomeEsercizio}}Config.json

5.2.3 Containers

Lo sviluppo del backend è avvenuto attraverso lo strumento dei container attraverso l'utilizzo della tecnologia Docker. Per descrivere quindi la struttura del backend e delle sue funzionalità vengono elencati e descritti i container realizzati durante la fase di implementazione. La coordinazione tra questi container è realizzata attraverso un file docker-compose che permette di gestire dal punto vista dell'archiviazione e della sincronizzazione le varie dipendenze di ogni container.

5.2.4 docker-compose

Per la gestione di un'architettura dotata di vari container è stato utilizzato lo strumento definito come "Docker compose".
Attraverso docker compose è possibile definire un file yaml di configurazione che ci permette di gestire con un singolo comando un insieme di container attraverso le configurazioni definite nello stesso file di configurazione.
Il tool docker-compose permette quindi con un singolo comando "docker-compose up" di scaricare da remoto, le immagini dei container presenti sull'hosting service Docker Hub, e di avviare il backend con tutte le configurazioni necessarie.

Figura 23: Struttura Docker

è possibile quindi raggruppare tutti i servizi necessari all'interno del folder "tesi".
Il file docker-compose.yaml utilizzato al fine di sviluppo possiede la seguente forma:

```
version: "3"
services:
   userservice:
       image: dariotintore/user-service
       container_name: user-service
       ports:
           - '8081:8081'
       environment:
           -
SPRING_DATASOURCE_URL=jdbc:h2:file:./database/userdb
   compilerservice:
       image: dariotintore/compiler-service-node
       container_name: compiler-service
       ports:
           - '8083:8083'
   leaderboardservice:
       image: dariotintore/leaderboard-service
       container_name: leaderboard-service
       ports:
           - '8087:8087'
       environment:
           -
SPRING_DATASOURCE_URL=jdbc:h2:file:./database/leaderboard
db
   exerciseservice:
       image: dariotintore/exercise-service
       container_name: exercise-service
       ports:
           - '9090:9090'
   frontend:
       image: dariotintore/frontend
       container_name: frontend
       ports:
           - '4200:80'
```

In questo caso, il docker-compose.yaml considera la creazione di tutti i servizi necessari all'esecuzione del backend, insieme al microservizio di frontend che permette ai giocatori di accedere all'applicativo tramite browser.

5.3 Test Smell Detector

All'interno del progetto viene utilizzato uno strumento di smell detection che possiamo recuperare alla seguente repository GitHub.
https://github.com/TestSmells/TestSmellDetector/releases
Lo strumento in questione è stato già discusso nei capitoli precedenti, tuttavia bisogna considerare che durante le fasi dello sviluppo sono state apportate diverse modifiche al progetto in questione dal punto di vista della rappresentazione dei dati.
È stato cambiato infatti il modo in cui gli smell vengono restituiti in modo tale da formare un output JSON utilizzando la repository seguente:

```
<dependency>
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
    <version>2.10.1</version>
</dependency>
```

In questo modo è stato possibile effettuare la creazione di un oggetto JSON che ci permettesse di recuperare i metodi che presentano gli smell, ed i rispettivi tipi di smell in modo agevole. Il risultato finale è il seguente:

```
JunitExample
{"Eager Test":{"methods":["testRemove","testSize","testAdd"]},"Unknown Test":{"methods":["removeAll"]},"Magic Number Test":{"methods":["removeAll","testRemove","testSize","testAdd"]},"Lazy Test":{"methods":["add",
 "sizeOfStudent"]}}

Process finished with exit code 0
```

Dove la prima riga corrisponde al file Java che stiamo considerando, seguita dall'oggetto JSON che sarà poi elaborato dal compiler service ed inviato successivamente al Frontend. Successivamente il frontend scompone i componenti di questo oggetto JSON per rappresentare agevolmente sotto forma di Accordion tutti gli smell e tutte le informazioni utili allo svolgimento dell'esercizio.

5.3.1 Thresholds

In questo caso bisogna tenere presente che lo strumento di smell detector considerato all'interno della tesi prevede che uno smell si verifichi se viene oltrepassata una specifica soglia di tolleranza, rappresentata attraverso un numero. Ad esempio nel caso in cui stiamo considerando lo smell di Assertion Roulette, la soglia numerica consiste nel numero di asserzioni necessarie a far verificare la presenza dello smell dallo smell detector.

All'interno del file Thresholds.kt è presente quindi la definizione di una soglia per un certo insieme di smell, i valori settati di default sono i seguenti:

```
open class DefaultThresholds : Thresholds() {
    override val eagerTest: Int
        get() = 1
    override val assertionRoulette: Int
        get() = 1
    override val conditionalTestLogic: Int
        get() = 0
    override val magicNumberTest: Int
        get() = 0
    override val mysteryGuest: Int
        get() = 0
    override val resourceOptimism: Int
        get() = 0
    override val sleepyTest: Int
        get() = 0
    override val emptyTest: Int
        get() = 0
    override val printStatement: Int
        get() = 0
    override val redundantAssertion: Int
        get() = 0
    override val sensitiveEquality: Int
        get() = 0
```

eagerTest = numero di metodi dell'oggetto di produzione invocati necessari a triggerare lo smell, in questo caso se ci sono un numero di chiamate a metodi dell'oggetto di produzione maggiore rispetto al valore di eagerTest viene rilevato lo smell.

assertionRoulette = numero di asserzioni necessarie a triggerare lo smell, in questo caso se ci sono un numero di asserzioni maggiore rispetto al valore di assertionRoulette viene triggerato lo smell.

conditionalTestLogic = numero di loops e conditional statements presenti nel test necesari a triggerare lo smell, in questo caso se ci sono un numero di loops/conditional statements maggiore rispetto al valore di conditionalTestLogic viene triggerato lo smell.

magicNumberTest = numero di letterali numerici nelle asserzioni necessari a triggerare lo smell, in questo caso se ci sono un numero di letterali numerici nelle asserzioni maggiore rispetto al valore di magicNumberTest viene triggerato lo smell.

mysteryGuest = numero di risorse esterne tollerate, in questo caso se ci sono un numero di risorse esterne maggiore rispetto al valore di mysteryGuest viene triggerato lo smell.

resourceOptimism = numero di risorse esterne tollerate, in questo caso se ci sono un numero di risorse esterne maggiore rispetto al valore di mysteryGuest viene triggerato lo smell.

sleepyTest = numero di sleep tollerate nel metodo di test, in questo caso se il numero di sleep presenti nel test è maggiore rispetto al valore di sleepyTest viene triggerato lo smell.

emptyTest = numero di statement necessari a triggerare lo smell, in questo caso se il numero di statement del metodo è uguale al valore di emptyTest viene triggerato lo smell.

printStatement = numero di print statement necessari a triggerare lo smell, in questo caso se il numero di print presenti nel test è maggiore rispetto al valore di printStatement viene triggerato lo smell.

redundantAssertion = numero di asserzioni ridondanti necessarie a triggerare lo smell, in questo caso se ci sono un numero di asserzioni ridondanti maggiori rispetto al valore di reduntantAssertion allora lo smell viene verificato

sensitiveEquality = numero di sensitive equality presenti nel test necessarie a triggerare lo smell, in questo caso se ci sono un numero di sensitive equality maggiore rispetto al valore di sensitiveEquality allora lo smell viene identificato.

5.3.2 Smell considerati

- Assertion Roulette
- Conditional Test Logic
- Constructor Initialization
- Default Test
- Duplicate Assert
- Eager Test
- Empty Test
- General Fixture
- Ignored Test
- Lazy Test
- Magic Number Test
- Mystery Guest
- Redundant Print
- Redundant Assertion
- Resource Optimism
- Sensitive Equality
- Sleepy Test
- Unknown Test

5.4 Pipelines

Per rendere lo sviluppo più agevole, sono state progettate ed implementate delle pipeline attraverso lo strumento Jenkins

Figura 24: Pipelines presenti su Jenkins

Le seguenti pipeline hanno come scopo quello di effettuare la compilazione, un'analisi statica attraverso lo strumento Sonarqube, e di generare i container che poi verranno pubblicati automaticamente su Docker Hub

Figura 25: Esempio di build Jenkins

6 Installazione ed utilizzo

6.1 Installazione server

Per installare l'infrastruttura server in locale è necessario clonare il seguente repository:
https://github.com/DarioTin/Tesi-Server-Setup
Dopo aver clonato il repository è necessario aprire un terminale all'interno della directory creata ed inviare all'interno del terminale
docker-compose up
Ottenendo un risultato del genere:

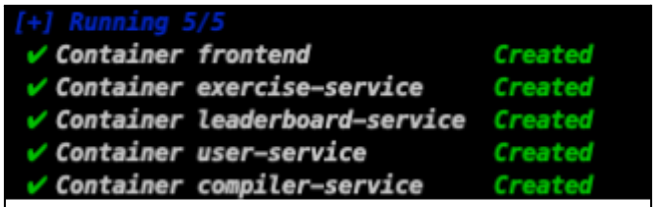
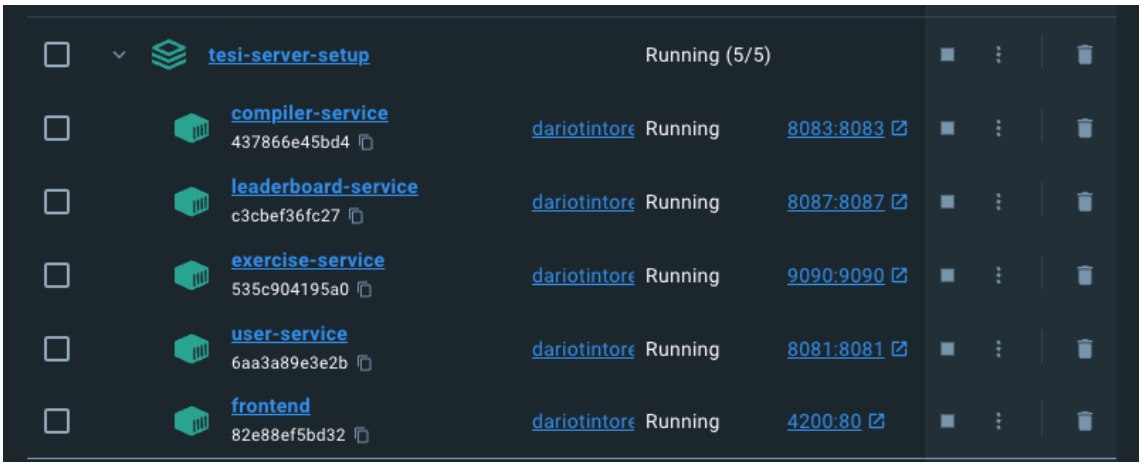


Figura 26: Risultato docker-compose up

In questo scenario avverrà la creazione di tutti i container necessari alla creazione dell'infrastruttura necessaria ad ospitare una sessione di gioco.



In seguito all'avvio di tutti i container è possibile accedere all'applicativo, partendo dal calcolatore server, al seguente indirizzo
http://localhost:4200/
In questo scenario avremo quindi installato correttamente il server in locale.

6.2 Installazione client

Per installare il client in locale è necessario clonare la seguente repository oppure scaricare l'ultima release dal seguente indirizzo:
https://github.com/DarioTin/Tesi-Client-Setup/releases
Dopo aver estratto il tutto, è necessario aprire un terminale all'interno della directory creata ed inviare all'interno del terminale
npm install
In questo caso, siccome l'installazione è indipendente dal sistema operativo, il nome della cartella e l'estensione dell'eseguibile che verrà creato varierà in base alla piattaforma su cui il comando sarà avviato.
Al termine dell'installazione verrà installato all'interno della directory "DarioTintore-Tesi-..." l'applicativo in locale, che potrà essere avviato con la seguente procedura:

1) Aprire un terminale all'interno della directory "DarioTintore-Tesi-..."
2) Eseguire il comando di avvio per l'eseguibile creato all'interno della directory "DarioTintore-Tesi"
3) Inizia a giocare 🙂

6.3 Installazione esercizi

L’aggiunta degli esercizi può essere effettuata su due piattaforme, su GitHub oppure tramite l’exercise service.
Gli esercizi recuperabili da github possono essere recuperati solo tramite la desktop application, mentre gli esercizi inseriti sull’exercise service possono essere recuperati da tutte le versioni dell’applicazione, ma richiedono che il microservizio “exercise service” sia attivo.
Sia nel caso dell’aggiunta di esercizi sulla repository GitHub che dell’aggiunta di esercizi sul microservizio, le regole da seguire sono identiche, l’unica cosa che cambia è il luogo in cui gli esercizi risiedono.

6.3.1 GitHub

Per aggiungere un nuovo esercizio recuperabile tramite github è necessario creare una nuova repository, in cui ad ogni esercizio è associata una cartella, ed in ogni cartella ci sono 3 file, dove i file java devono avere lo stesso nome dell’esercizio in questione.

6.3.2 ExerciseRepository

Per aggiungere un nuovo esercizio recuperabile tramite microservizio Exercise Repository è necessario creare una cartella all’interno della cartella ExerciseDB, in cui ad ogni esercizio è associata una cartella, ed in ogni cartella ci sono 3 file, dove i file java devono avere lo stesso nome dell’esercizio in questione

6.3.3 Struttura esercizio

Ad esempio se stiamo considerando di aggiungere un nuovo esercizio nominato “StringFormatter” avremo una struttura del genere

Per aggiungere un esercizio al microservizio ExerciseRespository è necessario

StringTester (Nome cartella)

StringFormatter.Java (Nome file di produzione)

StringFormatterTest.java (Nome file di testing originale)

StringFormatterConfig.json (Nome file di configurazione)

Quindi si richiede una configurazione del genere

{Nome Esercizio} (Directory)

{Nome Esercizio}.java (File java di produzione)

{Nome Esercizio}Test.java (File java di test)

{Nome Esercizio}Config.java (File di configurazione)

La struttura dei file è la seguente:

Nel file di produzione e di test deve esserci sempre il seguente package:

```
package com.dariotintore.tesi;
```

All'interno dei file di configurazione devono esserci sempre definite le seguenti chiavi:

```
{
 "refactoring_game_configuration": {
   "dependencies": [
     ""
   ],
   "refactoring_limit": number,
   "smells_allowed": number
 },
 "check_game_configuration": {
   "questions": [
     {
       "questionTitle": "",
       "questionCode": "",
       "answers": [
         {
           "answerText": "",
           "isCorrect": true/false
         }
       ]
```

```
    }
 },
 "auto_valutative": true/false
}
```

Le chiavi di refactoring_game riguardano le impostazioni riguardo la modalità refactoring game, mentre le chiavi di check_game riguardano la modalità di check smell. Nel momento in cui ci saranno chiavi mancanti in uno dei due casi l'esercizio potrebbe non funzionare correttamente.
Per ogni esercizio è quindi necessario configurare correttamente, come nell'esempio proposto in precedenza, tutte le configurazioni necessarie.
La spiegazione delle chiavi è riportata al capitolo "Opzioni esercizio".

6.4 Utilizzo database

I database presenti nei vari microservizi, come già citato in precedenza, sono dei database in memory, il che vuol dire che avviando il microservizio verrà avviato anche il rispettivo database a cui il microservizio è legato.
Per accedere al database di uno specifico microservizio è necessario considerare la seguente regola:
http://{urlmicroservizio}:{portamicroservizio}/h2
Se ad esempio stiamo considerando di avviare il microservizio user-service all'url 192.168.1.1 con porta 8081, per raggiungere il database sarà necessario aprire il browser ed inserire il seguente url:
http://192.168.1.1:8081/h2
Una volta raggiunto il seguente indirizzo avremo la seguente schermata:

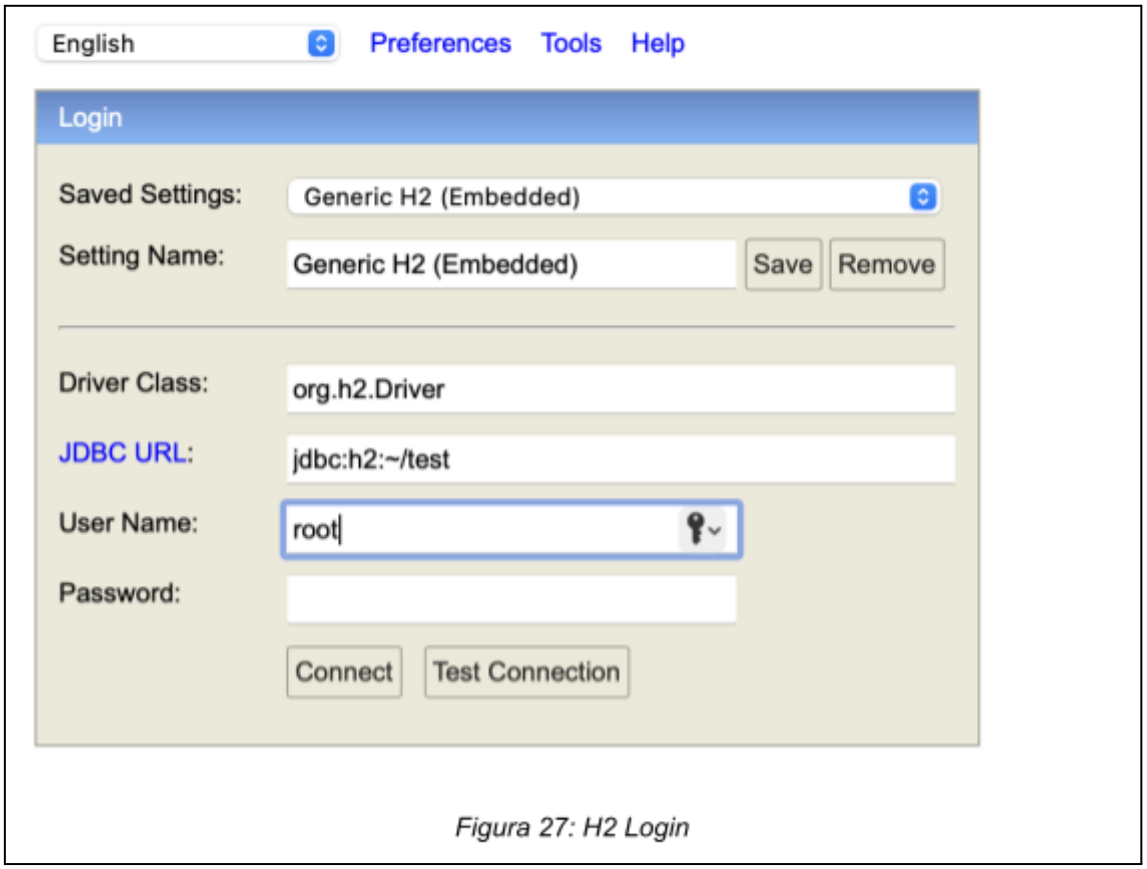


Figura 27: H2 Login

Per ottenere le informazioni necessarie, ovvero il JDBC URL, lo username e la password dobbiamo considerare il microservizio a cui ci vogliamo connettere, ed aprire il file application.properties presente nella cartella resources

All’interno di questo file troveremo tutte le informazioni necessarie da inserire per accedere al database. Considerando lo scenario in cui vogliamo accedere al microservizio user-service avremo la seguente configurazione:

```
# H2
spring.datasource.url=jdbc:h2:file:./database/userdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=root
spring.datasource.password=password
```

Quindi una volta inseriti i dati:

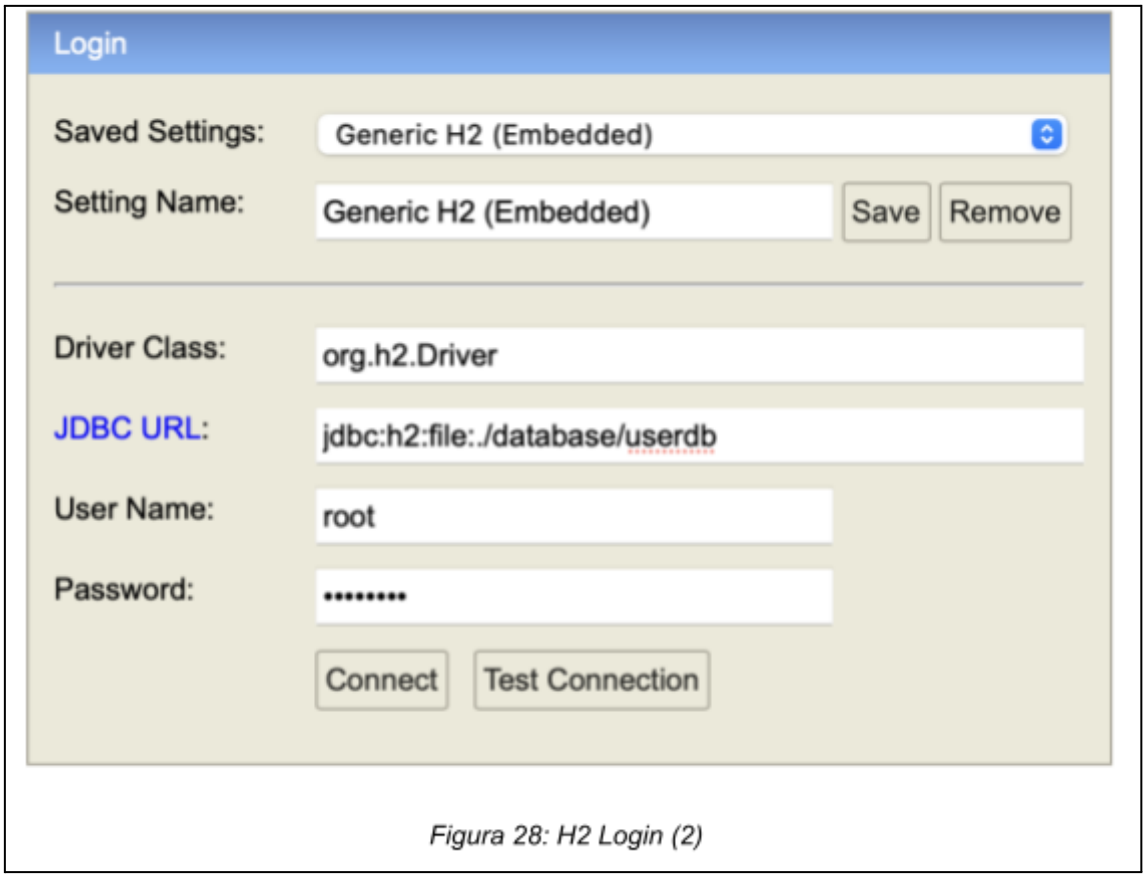


Figura 28: H2 Login (2)

Possiamo accedere al DBMS interno

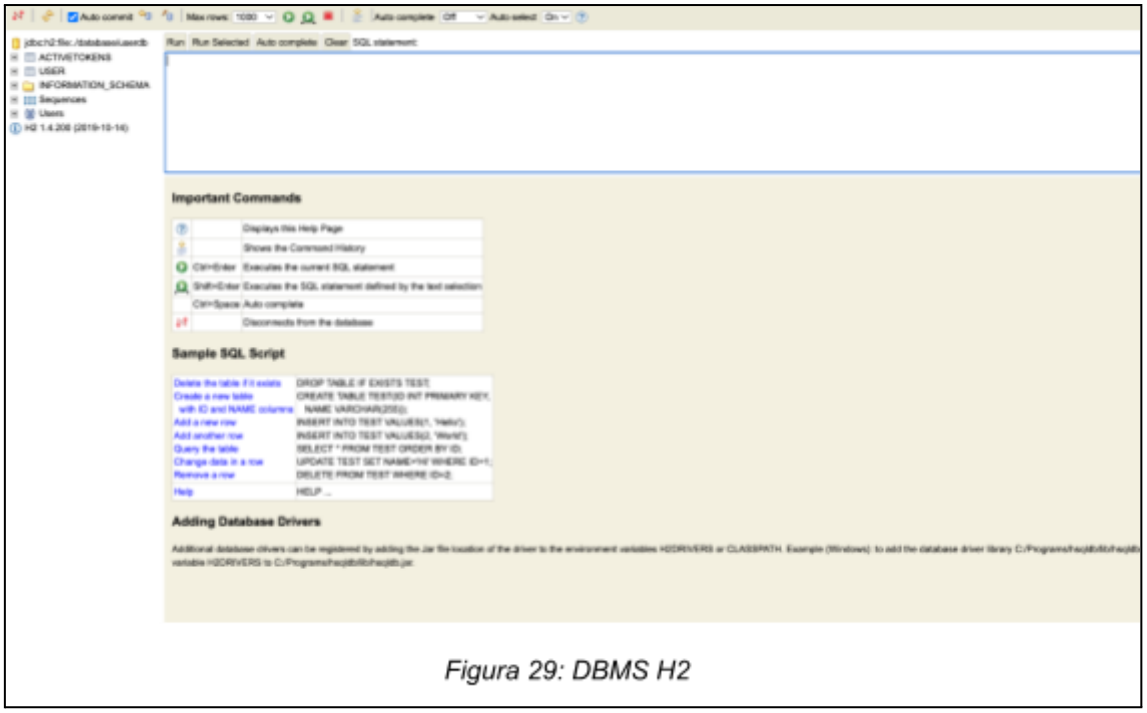


Figura 29: DBMS H2

6.5 Repositories

User Service	https://github.com/DarioTin/Tesi_User_service
Exercise Service	https://github.com/DarioTin/Tesi_Exercise_service
Compiler Service	https://github.com/DarioTin/Tesi_Compiler_service
Leaderboard Service	https://github.com/DarioTin/Tesi_Leaderboard_service
Frontend Online	https://github.com/DarioTin/Tesi_Frontend
Server-Setup	https://github.com/DarioTin/Tesi-Server-Setup
Client-Setup	https://github.com/DarioTin/Tesi-Client-Setup

7 Esempio d’uso

In questo capitolo verranno esposti degli esempi d’uso dell’applicativo descritto.
Le funzionalità e gli esempi che verranno esposti all’interno di questo capitolo rappresentano le funzionalità fondamentali che richiedono una comprensione maggiore dell’applicativo, verranno evitati ad esempio casi d’uso di funzionalità generiche come Login, Registrazione e così via.
Per gli esempi d’uso, viene considerato l’esercizio Calculator, scelto poiché risulta essere particolarmente semplice ed intuitivo. In seguito vengono esposti i file di cui questo esercizio si compone

Calculator.java

```
package com.dariotintore.tesi;


/**
* Calculator class:
* Basic Mathematical functions like
* Add,Subtract,Multiply,Divide.
*
*/
public class Calculator {
   //no-arg constructor
   public Calculator() {
   }
   /**
   * Sum method.
   */
   public int add(int a, int b) {
   return a + b;
   }
   /**
   * Subtract method.
   */
   public int subtract(int a, int b) {
```

```
    return a - b;
    }

    /**
    * Multiply method.
    */
    public long multiply(int a, int b) {
    return a * b;
    }

    /**
    * Divide method.
    * Note that this method throws an exception when
    * b is zero.
    */
    public double divide(int a, int b) {
        double result;
        if (b == 0) {
            throw new IllegalArgumentException("Divisor
cannot divide by zero");
        }
        else {
            result = Double.valueOf(a)/Double.valueOf(b);
        }
    return result;
    }
}
```

CalculatorTest.java

```
package com.dariotintore.tesi;

import org.junit.*;
import static org.junit.Assert.*;
```

```
//Arrange-Act-Assert pattern

public class CalculatorTest {

private Calculator objCalcUnderTest;

@Before
public void setUp() {
//Arrange
objCalcUnderTest = new Calculator();
}

   @Test
   public void testAdd() {
       int a = 15; int b = 20;
       int expectedResult = 35;
       //Act
       long result = objCalcUnderTest.add(a, b);
       //Assert
       assertEquals(expectedResult, result);
       assertEquals(result, result);
}

   @Test
   public void testSubtract() {
       int a = 25; int b = 20;
       int expectedResult = 5;
       long result = objCalcUnderTest.subtract(a, b);
       assertEquals(expectedResult, result);
   }
```

```
@Test
public void testMultiply() {
int a = 10; int b = 25;
long expectedResult = 250;
long result = objCalcUnderTest.multiply(a, b);
assertEquals(expectedResult, result);
}

@Test
public void testDivide() {
int a = 56; int b = 10;
double expectedResult = 5.6;
double result = objCalcUnderTest.divide(a, b);
assertEquals(expectedResult, result,0.00005);
}
}
```

CalculatorConfig.json

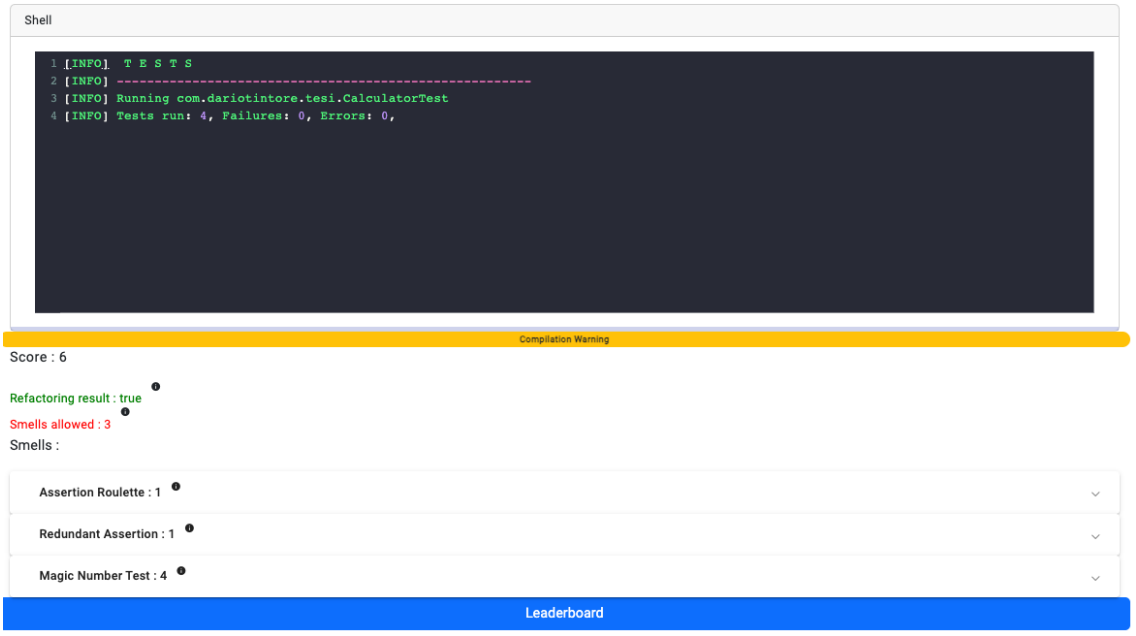
```
{
 "exerciseId": "Calculator_1",
 "refactoring_game_configuration": {
   "dependencies": [
     "<dependency>\n  <groupId>junit</groupId>\n
<artifactId>junit</artifactId>\n
<version>4.13.2</version>\n  <scope>test</scope>\n
</dependency>\n"
   ],
   "refactoring_limit": 5,
   "smells_allowed": 3
 },
 "check_game_configuration": {
   "questions": [
```

```
    {
      "questionTitle": "Qual è lo smell presente nel
seguente metodo?",
      "questionCode": "    @Test\n    public void
testAdd() { \n        int a = 15; int b = 20; \n        int
expectedResult = 35;\n        //Act \n        long result =
objCalcUnderTest.add(a, b);\n        //Assert\n
assertEquals(expectedResult, result);\n}\n",
      "answers": [
        {
          "answerText": "Assertion roulette",
          "isCorrect": true
        },
        {
          "answerText": "Magic number",
          "isCorrect": false
        }
      ]
    },
    {
      "questionTitle": "Qual è lo smell presente nel
seguente metodo?",
      "questionCode": "    @Test\n    public void
testSubtract() {\n        int a = 25; int b = 20; \n
int expectedResult = 5; \n        long result =
objCalcUnderTest.subtract(a, b);\n
assertEquals(expectedResult, result);\n    }\n",
      "answers": [
        {
          "answerText": "Resource Optimism",
          "isCorrect": true
        },
        {
```

```
            "answerText": "Unknown Test",
            "isCorrect": false
          }
        ]
      }
    ]
  },
  "auto_valutative": true
}
```


7.1 Esecuzione esercizio con Warning

In questo caso stiamo considerando di non apportare alcuna modifica all'esercizio in questione, cliccando semplicemente sul tasto di compilazione andando ad ottenere il seguente risultato:

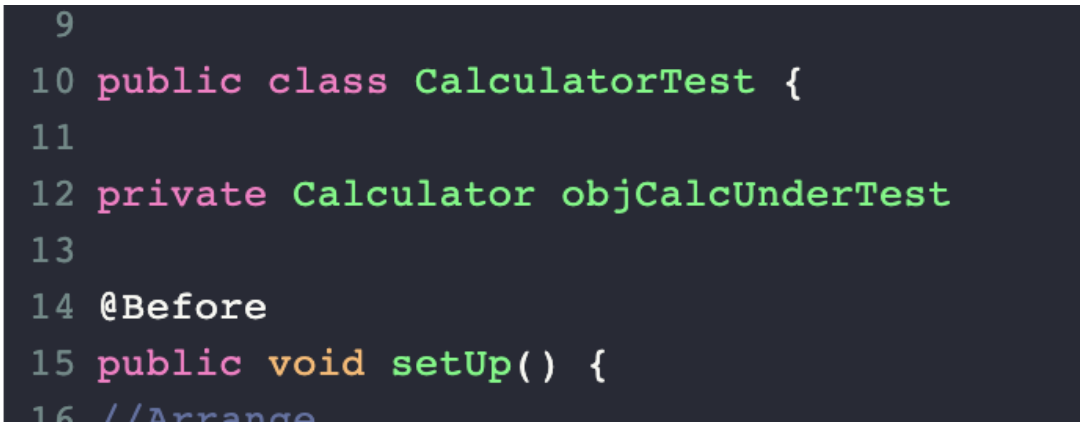


In questo caso possiamo notare che il refactoring test ha come risultato true poiché il codice non è cambiato, quindi la copertura del codice originale e la copertura del codice refattorizzato coincidono.

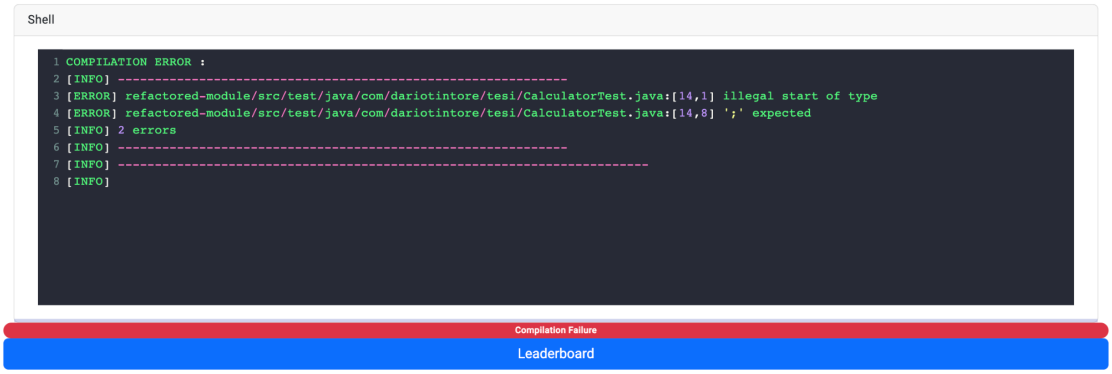
Gli smells allowed sono settati a 3, mentre il numero di smells ottenuti è 4, come rappresentato dallo score, quindi si ottiene una compilazione con Warning.

7.2 Esecuzione esercizio con Errore

Mostriamo in questo caso un esempio di errore di compilazione iniettato artificialmente attraverso un errore.



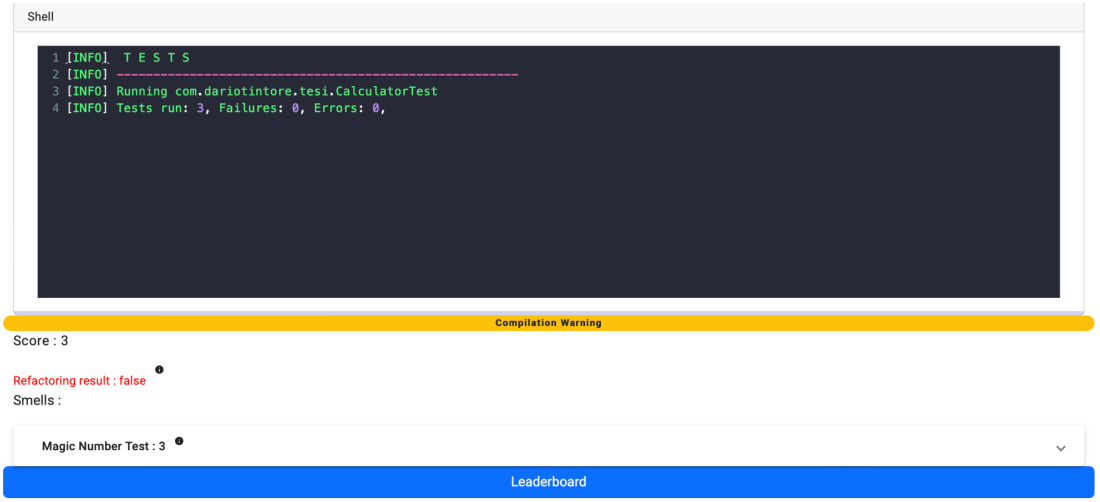
L'errore in questione risulta l'aver cancellato il punto e virgola a riga 12 del file di testing. In questo caso andando a premere il tasto di compilazione avremo la seguente schermata:



Con la spiegazione riguardo l'errore di compilazione.

7.3 Esecuzione esercizio con Refactoring test false

In questo caso modifichiamo il file di test in modo tale da verificare il funzionamento del refactoring check. In questo caso quindi rimuoviamo un metodo di test e vediamo che il refactoring check fallirà. Nello specifico rimuoviamo il testAdd:



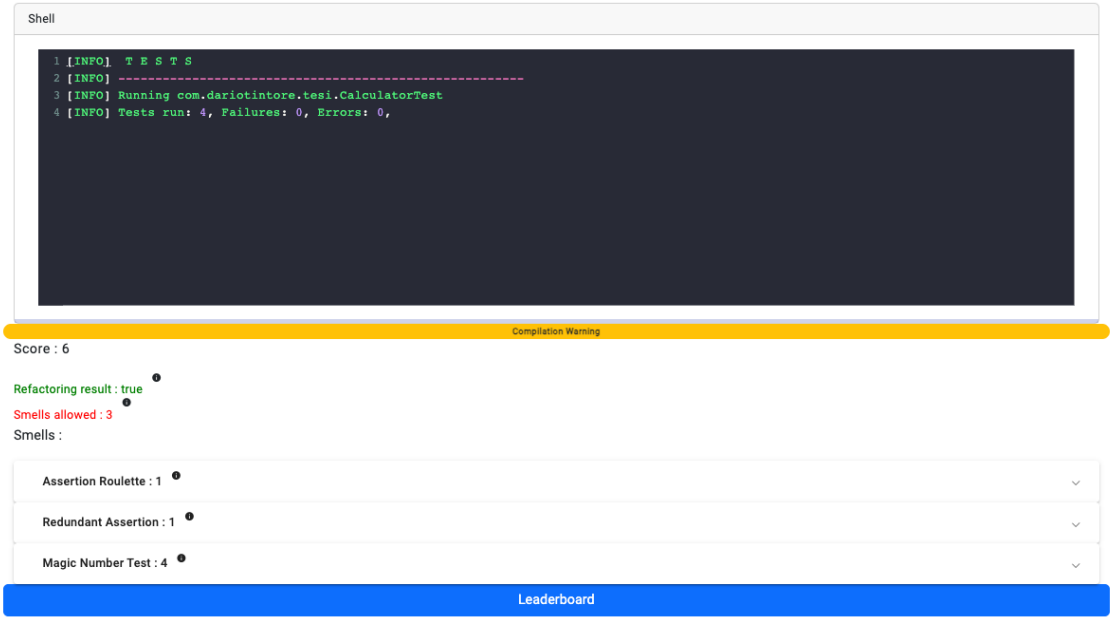
In questo caso ci troviamo di fronte ad uno scenario in cui, essendo stato rimosso un tests, abbiamo 3 test smell invece che 4, quindi non ci troviamo più a violare il vincolo relativo al numero di test smell accettati all'interno del test, ma avendo cambiato drasticamente il comportamento dei test, con la rimozione di un test, il refactoring result darà come risultato falso.

7.4 Rimozione di uno smell da un esercizio

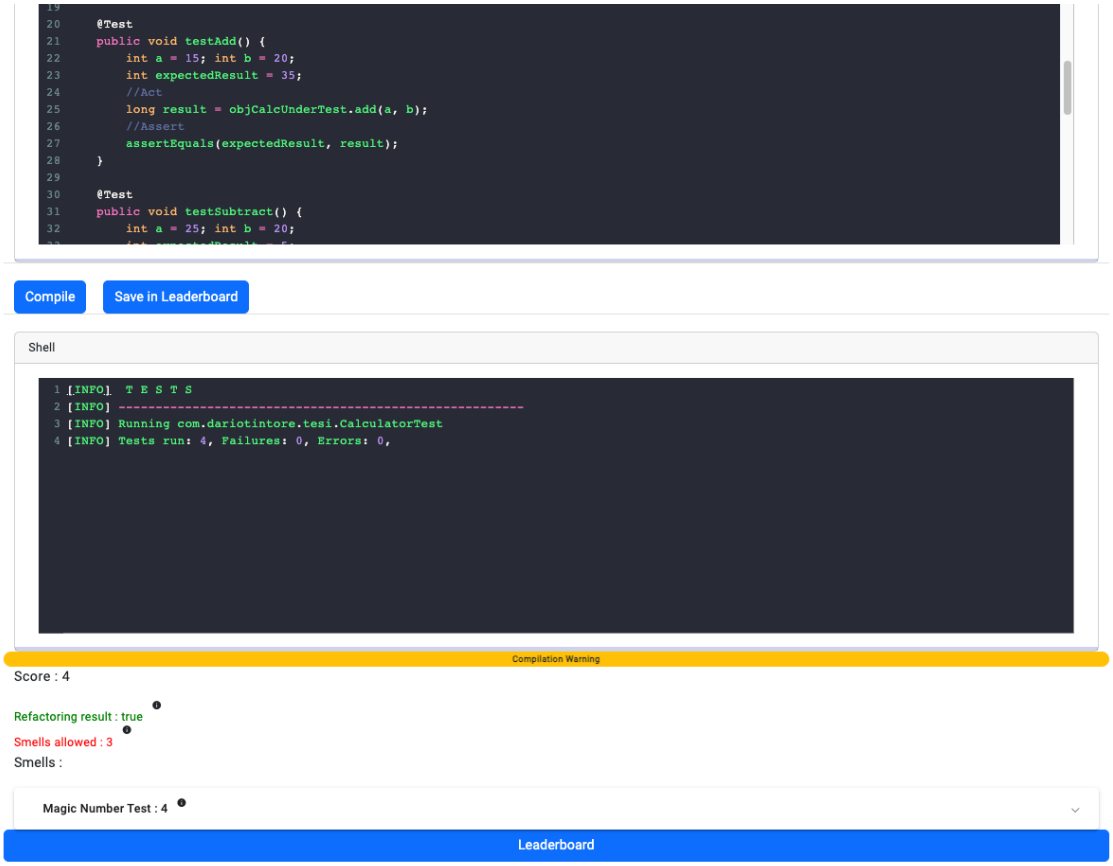
In questo caso vediamo la rimozione di uno smell all'interno dell'esercizio Calculator. Il test su cui opereremo sarà testAdd,

```
@Test
public void testAdd() {
    int a = 15; int b = 20;
    int expectedResult = 35;
    //Act
    long result = objCalcUnderTest.add(a, b);
    //Assert
    assertEquals(expectedResult, result);
    assertEquals(result, result);
}
```

infatti senza modificare il test, al momento della compilazione avremo il seguente risultato:

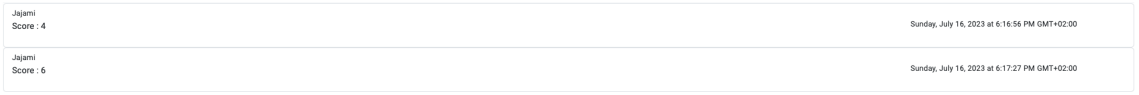


Mentre andando a rimuovere l'ultima asserzione il risultato diventerà il seguente:

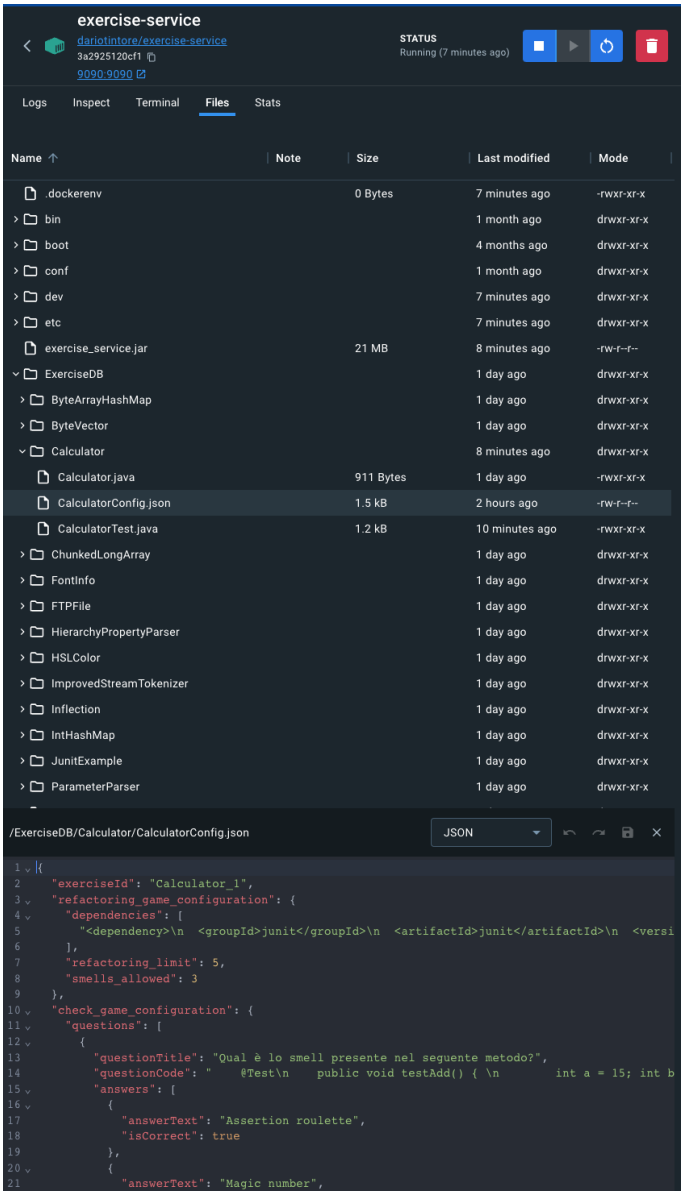


7.5 Modifica esercizio

Nel caso in cui stiamo considerando lo scenario in cui abbiamo un esercizio già pubblicato tra gli studenti, in cui la classifica è già popolata, ad esempio:



Nel caso in cui si volesse effettuare una nuova sessione dello stesso esercizio, ad esempio una nuova classifica, oppure nel caso in cui si volesse modificare l'esercizio in questione e quindi considerare una nuova classifica, bisogna modificare il campo "exerciseId" nel container docker:



Ovvero è possibile modificare l'exerciseId all'interno del container, per poi salvare il file con un nuovo id, in modo tale che nel momento in cui i giocatori andranno a selezionare l'esercizio in questione, faranno riferimento ad una classifica diversa. Infatti cliccando sul pulsante della leaderboard avremo il seguente risultato

Ovvero una classifica vuota.

7.6 Casi esemplari

I seguenti refactor sono stati verificati con lo strumento sviluppato prima e dopo il refactoring per verificare che lo smell non sia più presente. In seguito vengono proposti dei semplici esempi riguardo i test smell al fine di permettere al lettore la comprensione.

7.6.1 Assertion roulette

Lo smell di Assertion Roulette si verifica nel momento in cui ci sono differenti asserzioni nello stesso test, e queste non sono documentate. Lo smell testimonia una problematica di documentazione e comprensibilità, andando a rendere il test più difficile da gestire.

Pre refactor:

```
    @Test
    public void testAdd() {
        int a = 15; int b = 20;
        int expectedResult = 35;
        //Act
        long result = objCalcUnderTest.add(a, b);
        //Assert
        assertEquals(expectedResult, result);
        assertEquals(result, result);
    }
```

Post refactor:

```
    @Test
    public void testAdd() {
        int a = 15; int b = 20;
        int expectedResult = 35;
        //Act
        long result = objCalcUnderTest.add(a, b);
        //Assert
        assertEquals(expectedResult, result);
    }
```

7.6.2 Magic number test

Lo smell si verifica nel momento in cui un'asserzione contiene un "Magic Number", ovvero un numero letterale come parametro di un'asserzione. Questa pratica è sconsigliata ed è preferibile utilizzare delle costanti o delle variabili da utilizzare come input delle asserzioni.

Pre refactor:

```
    @Test
    public void testAdd() {
        obj.add("Emma");
        obj.add("Ronan");
        obj.add("Antonio");
        obj.add("Paul");
        assertEquals(4, obj.sizeOfStudent());
    }
```

Post Refactor:

```
    @Test
    public void testAdd() {
        obj.add("Emma");
        obj.add("Ronan");
        obj.add("Antonio");
        obj.add("Paul");
        int size = 4;
        assertEquals(size, obj.sizeOfStudent());
    }
```

7.6.3 Ignored Test

I test con tag @Ignored risultano essere dei test che apportano un overhead all'esecuzione di una test suite a livello di compilazione, senza apportare alcun beneficio dal punto di vista dell'affidabilità dei test. È consigliabile quindi rimuovere questi test per rimuovere lo smell in questione.

Pre refactor:

```
    @Test
    @Ignore
    public void testAdd() {
        obj.add("Emma");
        obj.add("Ronan");
        obj.add("Antonio");
        obj.add("Paul");
        assertEquals(4, obj.sizeOfStudent());
    }
```

Post refactor:

```
    @Test
    public void testAdd() {
        obj.add("Emma");
        obj.add("Ronan");
        obj.add("Antonio");
        obj.add("Paul");
        assertEquals(4, obj.sizeOfStudent());
    }
```

7.6.4 Eager Test

Lo smell Eager Test si verifica nel momento in cui un metodo di test invoca un certo numero di metodi dell'oggetto di produzione. Questo smell è "configurabile" all'interno delle configurazioni dello smell detector, che è di default fissato a 4. Il numero 4 non è fissato a caso ma è stato proposto nell'articolo "Investigating Severity Thresholds for Test Smells" da Davide Spadini [19].

Pre refactor:

```
    public void testAdd() {
        obj.add("Emma");
        obj.add("Ronan");
        obj.add("Antonio");
        obj.add("Paul");
        int max = 4;
        assertEquals(max, obj.sizeOfStudent());
    }
```

Post refactor:

```
    public void testAdd() {
        obj.add("Emma");
        obj.add("Ronan");
        obj.add("Antonio");
        int max = 3;
        assertEquals(max, obj.sizeOfStudent());
    }
```

7.6.5 Empty Test

Lo smell Empty Test avviene nel momento in cui non è presente codice eseguibile all'interno di un test. Test del genere potrebbero essere creati eventualmente per scopi di debugging oppure essere frutto di dimenticanze all'interno della test suite.

Pre refactor:

```
    public void testAdd() {
        // empty
    }
```

Post Refactor:

```
    public void testAdd() {
        obj.add("Emma");
        obj.add("Ronan");
        obj.add("Antonio");
        obj.add("Paul");
        assertEquals(4, obj.sizeOfStudent());
    }
```

7.6.6 Duplicate Assert

Lo smell Duplicate Assert si verifica nel momento in cui abbiamo più asserzioni con gli stessi parametri, andando di fatto a creare un'asserzione identica all'altra. Questo è il caso in cui è necessario testare un metodo ma con parametri differenti. In questo caso è consigliabile utilizzare un metodo di test separato dall'altro con un nome differente.

Pre refactor:

```
@Test
public void testRemove() {
    obj.add("Antonio");
    obj.remove("Paul");
    assertEquals(1, obj.sizeOfStudent());
    assertEquals(1, obj.sizeOfStudent());
}
```

Post refactor:

```
@Test
public void testRemove() {
    obj.add("Antonio");
    obj.remove("Paul");
    assertEquals(1, obj.sizeOfStudent());
}
```

7.6.7 Unknown Test

Lo smell Unknown Test si verifica nel momento in cui ci sono dei test senza alcun asserzione presente all'interno del metodo di test.

Pre refactor:

```
@Test
public void removeAll() {
    obj.removeAll();
}
```

Post refactor:

```
@Test
public void removeAll() {
    obj.removeAll();
       assertEquals(0, obj.sizeOfStudent());
}
```

7.6.8 Redundant Assertion

Questo smell occorre nel momento in cui un'asserzione è sempre verificata. Ad esempio il caso in cui un'asserzione, a prescindere dal codice che la precede, restituisce sempre true, oppure sempre false. Queste problematiche si verificano solitamente in casi di debugging/scrittura di codice di test.

Pre refactor:

```
    @Test
    public void testAdd() {
        int a = 15; int b = 20;
        int expectedResult = 35;
        //Act
        long result = objCalcUnderTest.add(a, b);
        //Assert
        assertEquals(expectedResult, result);
        assertEquals(result, result);
    }
```

Post refactor:

```
    @Test
    public void testAdd() {
        int a = 15; int b = 20;
        int expectedResult = 35;
        //Act
        long result = objCalcUnderTest.add(a, b);
        //Assert
        assertEquals(expectedResult, result);
    }
```

7.6.9 Sensitive Equality

Lo smell di sensitive equality si verifica nel momento in cui un metodo di test invoca il metodo toString di un oggetto che non ha effettuato override di toString.

Pre refactor:

```
	@Test(timeout = 4000)
	public void test_55_1058() throws Throwable {
		VCardBean v = new VCardBean();
		v.setEmail("abdullah@shaffield.ac.uk");
		v.setFax("00000000");
		v.setFirstName("Abdullah");
		v.setLastName("Alsharif");
		v.setMiddleName("None");
		v.setFormattedName("Abdullah Alsharif");
		v.setOrganization("SHef");
		v.setPhone("1011112");
		v.setTitle("Mr");
		assertEquals("BEGIN:VCARD\nFN:Abdullah
Alsharif\nN:Alsharif;None;Abdullah;\nTITLE:Mr\nORG:SHef\nEMAIL
;TYPE=INTERNET:abdullah@shaffield.ac.uk\nTEL;TYPE=VOICE:101111
2\nTEL;TYPE=FAX:00000000\nEND:VCARD", v.toString());
	}
```

Post refactor:

```
	public void test_55_1058() throws Throwable {
		VCardBean v = new VCardBean();
		v.setEmail("abdullah@shaffield.ac.uk");
		v.setFax("00000000");
		v.setFirstName("Abdullah");
		v.setLastName("Alsharif");
		v.setMiddleName("None");
		v.setFormattedName("Abdullah Alsharif");
		v.setOrganization("SHef");
		v.setPhone("1011112");
		v.setTitle("Mr");
		assertEquals("BEGIN:VCARD\nFN:Abdullah
```

```
Alsharif\nN:Alsharif;None;Abdullah;\nTITLE:Mr\nORG:SHef\nEMAIL
;TYPE=INTERNET:abdullah@shaffield.ac.uk\nTEL;TYPE=VOICE:101111
2\nTEL;TYPE=FAX:00000000\nEND:VCARD",
                              v.getVcard() + v.getFirstName() +
v.get ... );
     }
```

7.6.10 Redundant Print/Print Statement

I print statement negli unit test risultano inutili essendo i test eseguiti come parte di un processo automatico con una scarsa o nulla interazione umana.

Pre refactor:

```
    @Test
    public void testAdd() {
        int a = 15; int b = 20;
        int expectedResult = 35;
        //Act
        long result = objCalcUnderTest.add(a, b);
        //Assert
        System.out.println("OK");
        assertEquals(expectedResult, result);
        assertEquals(result, result);
    }
```

Post refactor:

```
    @Test
    public void testAdd() {
        int a = 15; int b = 20;
        int expectedResult = 35;
        //Act
        long result = objCalcUnderTest.add(a, b);
        //Assert
        assertEquals(expectedResult, result);
        assertEquals(result, result);
    }
```

7.6.11 Lazy Test

Lo smell Lazy Test avviene nel momento in cui viene testato due volte lo stesso metodo ma in due test differenti. Questo si verifica quando il risultato del metodo ricorrente viene relazionato in un'asserzione

Pre refactor:

```
@Test
public void lazy1(){
      long result = objCalcUnderTest.add(-1, -2);
      assertEquals(-3, result);
}
@Test public void lazy2(){
      long result = objCalcUnderTest.add(1, 2);
      assertEquals(3, result);
}
```

Post refactor:

```
@Test public void lazy1(){
  long resultPositiveAdd = objCalcUnderTest.add(1, 2);
  long resultNegativeAdd = objCalcUnderTest.add(-1, -2);
  assertEquals("Positive add" ,3, resultPositiveAdd);
  assertEquals("Negative add" ,-3, resultNegativeAdd);
}
```

7.6.12 Conditional Test Logic

Lo smell “Conditional Test Logic” si verifica nel momento in cui sono presenti control statements, come ad esempio “if, switch” ed espressioni simili.

Pre refactor:

```
@Test
public void conditionalTest(){
    int a = 5;
    int b = 6;
    int expectedResult = 11;
    if(a == 5){
      if(b == 6){
      assertEquals(objCalcUnderTest.add(a,b),expectedResult);
      }
    }
}
```

Post refactor:

```
@Test
public void conditionalTest(){
    int a = 5;
    int b = 6;
    int expectedResult = 11;
    assertEquals(objCalcUnderTest.add(a,b),expectedResult);
}
```

7.6.13 Default test

Lo smell “Default Test” si verifica nel momento in cui la classe di test ha uno specifico nome. In questo caso stiamo parlando non più di uno smell di un metodo, ma di uno smell di una Test Suite.

Lo smell detector si accorge di questo smell quando la nostra classe ha un nome di default generato da un IDE, come ad esempio ExampleInstrumentedTest oppure ExamplerUnitTest.

Pre refactor:

```
public class ExampleInstrumentedTest {
```

Post refactor:

```
public class CalculatorTest {
```

7.6.14 General Fixture

General Fixture è uno smell che si verifica nel momento in cui un metodo di test non accede ai campi inizializzati all'interno del metodo "SetUp()".

Pre refactor:

```
    @Test
    public void testAdd(){
           int i = 0;
           int j = 0;
           int result = 0;
           assertEquals(result,j);
     }
```

Post refactor:

```
    @Test
    public void testAdd() throws IOException{
           int i = 0;
           int j = 0;
           int result = objCalcUnderTest.add(i,j);
           assertEquals(result,j);
     }
```

7.6.16 Mystery Guest

Lo smell Mystery Guest si verifica nel momento in cui il test fa riferimento ad una risorsa esterna, ad esempio un database, un file ed elementi simili. La soluzione consiste nel rimuovere l'approccio a risorse esterne rimuovendo le relative funzioni oppure utilizzando dei Mock.

Pre refactor:

```
@Test
public void exampleMysteryTest() {
  int a = 56; int b = 10;
  double expectedResult = 5.6;
  double result = objCalcUnderTest.divide(a, b);
  File f = new File("");
  assertEquals(expectedResult, result,0.00005);
}
```

Post refactor:

```
@Test
public void exampleMysteryTest() {
  int a = 56; int b = 10;
  double expectedResult = 5.6;
  double result = objCalcUnderTest.divide(a, b);
  assertEquals(expectedResult, result,0.00005);
}
```

7.6.17 Sleepy Test

Lo smell “Sleepy Test” occorre nel momento in cui all’interno di un test è presente un thread wait che attende i risultati di un

Pre refactor:

```
    @Test
    public void testAdd() {
        int a = 15; int b = 20;
        int expectedResult = 35;
        //Act
        long result = objCalcUnderTest.add(a, b);
        //Assert
           try{
             Thread.sleep(100);
           }catch(Exception e){
             e.printStackTrace();
           }
        System.out.println("OK");
        assertEquals(expectedResult, result);
        assertEquals(result, result);
    }
```

Post refactor:

```
    @Test
    public void testAdd() {
        int a = 15; int b = 20;
        int expectedResult = 35;
        //Act
        long result = objCalcUnderTest.add(a, b);
        //Assert
         System.out.println("OK");
        assertEquals(expectedResult, result);
        assertEquals(result, result);
    }
```

7.6.18 Constructor Initialization

Lo smell Constructor initialization avviene nel momento in cui il test utilizza un costruttore invece che un metodo setUp per inizializzare i campi.

Pre refactor:

```
public class CalculatorTest {

  private Calculator objCalcUnderTest;

  public CalculatorTest(){
      objCalcUnderTest = new Calculator();
  }
```

Post refactor:

```
public class CalculatorTest {

private Calculator objCalcUnderTest;
  @Before
public void setUp() {
//Arrange
objCalcUnderTest = new Calculator();
   }
}
```

7.6.19 Resource Optimism

Lo smell “Resource Optimism” avviene nel momento in cui si effettua un’assunzione ottimistica riguardo una risorsa esterna utilizzata dal metodo di test. È possibile quindi eliminare lo smell utilizzando un metodo che ci permette che la risorsa effettivamente sia accessibile con metodi come “exists(), isFile()” e simili.

Pre refactor:

```
@Test
public void testResource() throws IOException{
        File f = new File("prefix.png");
}
```

Post refactor:

```
@Test
public void testResource() throws IOException{
        File f = new File("prefix.png");
        if(f.exists()){
        }
}
```

8 Validazione

Il seguente capitolo rappresenta il capitolo conclusivo, di validazione riguardo il lavoro presentato precedentemente attraverso uno studio empirico che ha come oggetto i test smells e lo strumento sviluppato. Lo strumento non è stato ancora implementato in un ambito accademico con una classe di studenti, ma attraverso degli utenti che avevano lo scopo di completare lo studio in questione.

8.1 Obiettivi

Gli obiettivi della validazione sono di valutare la correttezza e usabilità dello strumento di supporto all'insegnamento realizzato, al fine di tracciare una modalità di utilizzo ottimale nell'ambito di un task di learning (la cui esecuzione reale non può essere oggetto della tesi per questioni di tempistica e disponibilità dei soggetti sperimentali).

8.2 Research questions

RQ1) Quanti e quali tipi di test smells possono essere adeguatamente riconosciuti e corretti con lo strumento proposto?

Lo scopo della prima domanda di ricerca è quello di valutare se lo strumento utilizzato riesce correttamente a riconoscere test smells e la loro correzione. Le tipologie di test smells per le quali si riconoscono oggettive problematiche nel riconoscere l'avvenuta correzione dovrebbero essere filtrate dagli esercizi proposti.

RQ2) Quale soglia di differenza di copertura può essere ritenuta accettabile?

Una soglia di differenza di copertura è accettabile se effettivamente il test rifattorizzato può essere considerato concettualmente equivalente a quello originale. Misurando la media delle differenze di copertura ritenute valide si può proporre un valore operativo ottimale per questo parametro.

RQ3) Quanto tempo è necessario per rifattorizzare gli smell?

Un'attività sperimentale che coinvolge studenti deve avere dei limiti temporali ben precisi, al di sopra dei quali non possono essere esclusi effetti di stanchezza. Un valore accettabile per attività individuali da svolgere in classe può essere di una o due ore. Nel caso di homework invece potrebbero essere considerati valori più elevati, nell'ordine delle dieci ore (ma possiamo concentrarci solo sulla prima categoria). Misurare il tempo medio necessario per risolvere uno smell può fornire un'indicazione operativa sul numero di smell che potrebbe essere risolta in ogni esercizio.

8.3 Metriche

Per RQ1, la metrica utilizzata è il numero di smell risolti per tipologia, rispetto a quelli presenti, nelle prove svolte.

Per RQ2 la metrica è il tempo medio necessario per risolvere gli smell, possibilmente calcolato sia in forma aggregata che per tipologia di smell (si immagina che alcuni smell possano essere più complessi di altri da togliere, anche se sappiamo che questa misura non tiene conto di effetti di apprendimento e similarità tra le soluzioni, ma per misurarli sarebbero necessarie molte persone e un vero studio empirico sperimentale anziché una validazione).

8.4 Oggetti sperimentali

Le classi considerate per la sperimentazione sono 5, di cui consideriamo alcune caratteristiche fondamentali, come il numero di righe del file di produzione, il numero di righe del file di test ed il numero di test presenti all'interno di ciascun esercizio.
La motivazione con cui sono state considerate queste caratteristiche è da ricercarsi nel fatto che, maggiore sarà il numero di righe e di test presenti all'interno dei file, maggiore potrebbero essere i potenziali smells presenti all'interno dei file, maggiore sarà la complessità del refactoring da effettuare per ottenere dei test privi di smell.

File Produzione	File Test	#LOC Produzione	#LOC Test.	#Test
WeakHashTable	WeakHashTableTest	393	127	11
SubjectParser	SubjectParserTest	164	236	17
Inflection	InflectionTest	193	174	26
Range	RangeTest	474	339	35
XmlElement	XmlElementTest	442	512	49

Tabella 1: File utilizzati

Le classi sono ottenute dal seguente sito: https://study.code-defenders.org/

8.5 Procedura sperimentale

Il TestSmell Detector utilizzato all'interno della procedura di sperimentazione è la versione 2.2 ottenibile al seguente link.
L'esperimento è stato effettuato su una macchina con processore Apple Silicon M1 con un clock di 3.1 GHz, 8 GB di Ram, con un disco a stato solido (SSD) di 256 GB, utilizzando il sistema operativo macOS Ventura 13.4.1.
La versione dell'applicativo utilizzata per l'esperimento è stata la versione Desktop, quindi avendo il backend in esecuzione su Docker ed il frontend in esecuzione su browser Safari 16.5.2
L'esperimento è stato svolto interamente dallo studente scrittore della tesi, Dario Tintore N97000335 applicando la seguente metodologia:

1) Viene avviata una prima compilazione per identificare gli smell presenti all'interno dell'esercizio.
2) Viene scelto uno smell tra quelli presenti.
3) Viene applicata la correzione affinché lo smell non sia più presente.

I test smell da rimuovere sono stati scelti in modo casuale e per ogni modifica è stata effettuata una compilazione per verificare la correttezza del refactoring e della correzione applicata.
I tempi richiesti per lo svolgimento delle soluzioni sono stati considerati cronometrando con un applicativo esterno la risoluzione di un singolo smell, andando a considerare la ricerca dello smell all'interno del codice e la risoluzione dello stesso. Prima di effettuare l'operazione di registrazione del tempo è stato testato l'applicativo in modo tale da prendere dimestichezza con lo strumento e le varie tipologie di test smell.
Gli smell sono stati affrontati studiando sia il codice di produzione che il codice di test, andando di fatto a comprendere l'origine dell'impossibilità della risoluzione di alcuni smell che richiedessero una modifica non più sul codice di test ma sul codice di produzione.

8.6 Premesse

8.6.1 Tuning dell'applicativo

In questo caso bisogna tenere presente che lo strumento di smell detector considerato all'interno della tesi prevede che uno smell si verifichi se viene oltrepassata una soglia di tolleranza, rappresentata attraverso un numero, si ricorda al lettore che le soglie di tolleranza sono state trattate nel paragrafo 5.3.1. Nel caso in cui stiamo considerando lo smell di Assertion Roulette, la soglia numerica consiste nel numero di asserzioni necessarie a far verificare la presenza dello smell dallo smell detector.
È necessario quindi verificare che le soglie settate dallo smell detector siano adeguate al corretto svolgimento degli esercizi.
Nei paragrafi successivo vengono elencate le modifiche proposte per le soglie che ci permettono un funzionamento ottimale dell'applicativo, e di evitare individuare uno smell dove questo non dovrebbe essere presente.+

8.6.1.1 Magic Number Test

Questo specifico smell all'interno dei file di configurazione presenta una soglia settata a 0. Questo comporta la problematica che ogni singolo metodo di test che viene eseguito dallo smell detector rappresenta uno smell di Magic Number Test.
Impostando la soglia di Magic Number ad 1 abbiamo lo scenario in cui è presente almeno un parametro numerico all'interno di un'asserzione lo smell verrà segnalato dal nostro smell detector.
Quindi la seguente tabella ci permette di avere una visione complessiva degli effetti di questa modifica.

File Produzione	File Test	#MN pre-modifica	#MN post-modifica
WeakHashTable	WeakHashTableTest	11	1
SubjectParser	SubjectParserTest	17	0
Inflection	InflectionTest	26	0

Range	RangeTest	35	9
XmlElement	XmlElementTest	49	7

Tabella 2: Modifica MN

Gli smell rilevati dallo Smell Detector risultano adesso calibrati correttamente e si riferiscono ai corretti metodi di test.

8.6.1.2 Lazy Test

Per quanto riguarda gli smells che possono essere adeguatamente riconosciuti e corretti bisogna considerare che ci sono alcuni tipi di smells che difficilmente possono essere corretti in tempi brevi e senza modificare totalmente la suite test. Uno di questi smells è Lazy Test.

Per quanto riguarda lo smell Lazy Test questo occorre nel momento in cui uno o più metodi di test invocano lo stesso metodo della classe che si vuole testare, tuttavia possiamo notare che non sempre è facile refattorizzare una test suite in modo tale da rimuovere questo smell. Se consideriamo per esempio la classe WeakHashTableTest possiamo notare che gli smell Lazy Test sono i seguenti:

- containsKey
- isEmpty
- put

Questi tre metodi quindi vengono invocati in più metodi di test.

Se vogliamo prendere in esame il metodo “containsKey”, possiamo notare che questo metodo occorre nei seguenti casi di test:

- test_3_1178()
- test_4_1179()
- test_5_1187()
- test_7_1264()
- test_9_1272()

Questo si traduce nel fatto che, abbiamo 5 test e dobbiamo refattorizzare tutti questi in modo tale da ottenere una situazione in cui il metodo containsKey venga impiegato in un solo metodo. Per quanto riguarda la calibrazione all’interno del TestSmellDetector

non è presente alcun valore configurabile, quindi partiamo col dire che non è possibile effettuare un cambio di configurazione dal punto di vista del TSDetect. Quindi è necessario effettuare un refactor che prevede di compattare cinque metodi in un singolo test. Tuttavia questa situazione potrebbe essere particolarmente sconveniente in test suite di grandi dimensioni, sia dal punto di vista della complessità dei test, che potrebbero avere un gran numero di righe, sia dal punto di vista della coerenza di un singolo test, che si ridurrebbe a fare più cose e particolarmente diverse tra loro.
Si ricavano le conclusioni che questo smell risulta pertanto più problematico che benefico dal punto di vista di una buona progettazione degli smell.
Volendo analizzare i test abbiamo i seguenti test:

```
@Test(timeout = 4000)
public void test_3_1178() throws Throwable {
   // test here!
   WeakHashtable wh = new WeakHashtable();
   wh.put("a",new Integer(1));
   wh.put("b",new Integer(2));
   wh.put("c",new Integer(3));
   wh.put("d",new Integer(4));

   assertTrue(wh.containsKey("a"));
}

@Test(timeout = 4000)
public void test_4_1179() throws Throwable {
   // test here!
   WeakHashtable wh = new WeakHashtable();
   wh.put("a",new Integer(1));
   wh.put("b",new Integer(2));
   wh.put("c",new Integer(3));
   wh.put("d",new Integer(4));
     assertFalse(wh.containsKey("e"));
}
```

```
@Test(timeout = 4000)
public void test_5_1187() throws Throwable {
   WeakHashtable wilko = new WeakHashtable();
   assertFalse(wilko.containsKey("123"));
}

@Test(timeout = 4000)
public void test_7_1264() throws Throwable {

   java.util.HashMap foo = new java.util.HashMap();
   WeakHashtable w = new WeakHashtable();
   foo.put("a","b");
   w.putAll(foo);
   assertTrue(w.keySet().contains("a"));
   assertTrue(w.containsKey("a"));

}

@Test(timeout = 4000)
public void test_9_1272() throws Throwable {
   WeakHashtable wh = new WeakHashtable();
   wh.put("a",new Integer(1));
   wh.put("b",new Integer(2));
   wh.put("c",new Integer(3));
   wh.put("d",new Integer(4));

   wh.remove(new String("c"));
   assertFalse(wh.containsKey("c"));
}
```

Una possibile soluzione quindi consiste nel rimuovere tutti questi casi di test ed aggiungere un singolo caso di test che li contiene tutti, in questo caso abbiamo il seguente caso di test:

```
@Test(timeout = 4000)
public void testLazyTestKiller() throws Throwable{
   WeakHashtable wh = new WeakHashtable();
   wh.put("a",new Integer(1));
   wh.put("b",new Integer(2));
   wh.put("c",new Integer(3));
   wh.put("d",new Integer(4));
   assertTrue(wh.containsKey("a"));
   assertFalse(wh.containsKey("e"));
   assertFalse(wh.containsKey("123"));
   java.util.HashMap foo = new java.util.HashMap();
   foo.put("a","b");
   wh.putAll(foo);
   assertTrue(wh.keySet().contains("a"));
   assertTrue(wh.containsKey("a"));
   wh.remove(new String("c"));
   assertFalse(wh.containsKey("c"));
}
```

Con questa operazione lo smell detector riesce correttamente a verificare che il Lazy Test smell per il metodo “containsKey” è effettivamente rimosso, tuttavia è stata di molto aumentata la complessità di un singolo caso di test con un gran numero di

asserzioni e con un gran numero di task. Ne consegue che il LazyTest in classi particolarmente complesse sarebbe meglio ignorarlo inserendolo all'interno della configurazione dei TestSmell ignorati.

8.6.2 Eager Test

All'interno del Test Smell Detector è presente una soglia che ci permette di capire quando uno smell è presente o meno. Nel caso dello Eager Test Smell questa soglia è settata ad 1. Questo vuol dire che, nel momento in cui all'interno di un caso di test viene chiamato più di un metodo della classe che stiamo testando, allora lo smell detector restituisce la presenza dello smell. Tuttavia questo valore risulta essere non adeguato, basti pensare allo scenario in cui si vuole testare il corretto funzionamento di una struttura dati all'interno di una classe di test. In questo caso anche solamente testando che le operazioni di base della struttura dati (rimozione, aggiunta) funzionino correttamente, è necessario effettuare almeno due chiamate alla classe in questione, ovvero una chiamata per inserire/rimuovere un oggetto all'interno della struttura dati, una chiamata per verificare attraverso un'assert che l'operazione è stata effettuata correttamente.
Si propone quindi di cambiare la soglia da 1 a 4 in modo tale da rendere lo Detector più adatto ai refactor che verranno effettuati. Il numero 4 non è fissato a caso ma è stato proposto nell'articolo "Investigating Severity Thresholds for Test Smells" da Davide Spadini [19].

File Produzione	File Test	#ET pre-modifica	#ET post-modifica
WeakHashTable	WeakHashTableTest	8	4
SubjectParser	SubjectParserTest	17	17
Inflection	InflectionTest	14	0
Range	RangeTest	31	16
XmlElement	XmlElementTest	32	7

Tabella 3: Modifica ET

8.7 Risultati

I seguenti risultati sono stati ottenuti attraverso l'esecuzione degli esercizi in locale con lo scopo di rispondere alle Research Questions.

8.7.1 RQ1

Come citato nei paragrafi precedenti, la metrica utilizzata è il numero di smell risolti per tipologia, rispetto a quelli presenti, nelle prove svolte. I risultati vengono quindi esposti attraverso la seguente tabella che prevede la seguente legenda:

Legenda:
# = Numero
AR = Assertion Roulette
ET = Eager Test
DA = Duplicate Assert
MN = Magic Number
SE = Sensitive Equality
In ogni colonna è rappresentato il numero di smell della rispettiva tipologia, e tra parentesi tonde è rappresentato il numero di smell risolti della rispettiva tipologia

Classe	#AR	#ET	#DA	#MN	#SE	#Smell presenti	#Smell risolti
WeakHash Table	3 (3)	4 (4)	1 (1)	1 (1)		9	9
Subject Parser	15 (15)	17 (0)	0	0		32	15
Inflection	11 (11)	0	0	0		11	11
Range	21 (21)	16 (0)	0	9 (9)	2 (2)	48	32
XMLEleme nt	16 (16)	9 (1)		7 (7)		32	24

Tabella 3: Risultati RQ1

$$\frac{\#Smell\ risolti}{\#Smell\ totali} = \frac{91}{132} = 68\%\ smell\ risolti$$

Analizzando la tabella precedente è possibile notare una problematica in particolare con la classe “SubjectParser”. Per quanto riguarda la classe SubjectParser per come è stata sviluppata possiede alcuni problemi con il test smell EagerTest. Lo smell Eager Test si verifica nel momento in cui un metodo di test invoca un certo numero di metodi dell’oggetto di produzione. Dopo questa considerazione iniziale, la struttura dei metodi di test di SubjectParser è la seguente:

```
public void test_2_1849() throws Throwable {
   String input = " [hello] (world) , test";
   SubjectParser sp = new SubjectParser(input);
   String verify = "" + sp.getUpperRange() +
sp.getThisRange() + sp.getId() + sp.getTitle() +
sp.getRangeString();
   assertEquals("11-1[hello] (world) , testnull", verify);
}
```

Ovvero abbiamo una stringa verify che viene generata attraverso n chiamate di metodi dell’oggetto SubjectParser, ed è esattamente ciò che non vogliamo che si verifichi. Per modificare questo smell quindi sarebbe necessario effettuare una modifica dal punto di vista del codice di produzione per ottenere ciò che ci interessa senza dover necessariamente chiamare un numero così alto di metodi. Tuttavia questa modifica va oltre la classe di test e quindi non è possibile effettuare alcun fix.

Per quanto riguarda la classe XmlElement bisogna effettuare delle considerazioni riguardo lo smell Eager Test. In alcuni metodi sappiamo che è possibile operare sullo smell per rimuoverlo, in altri è necessario invece un cambiamento strutturale del metodo. Questo cambiamento strutturale richiede solitamente che uno stesso test venga scomposto in due o più test in modo tale da diminuire il numero di chiamate che vengono effettuate all'interno del metodo di test.
Se consideriamo ad esempio il test "test_22_1384()" abbiamo la seguente situazione:

Pre refactoring:

```
@Test(timeout = 4000)
public void test_22_1384() throws Throwable {
   XmlElement a = new XmlElement();
   XmlElement b = new XmlElement();
   a.addAttribute("p","a");
   assertFalse(a.equals(b));
   b.addAttribute("p","a");
   b.setData("abc");
   a.setData("abc");

   XmlElement z = new XmlElement();
   a.addSubElement(z);
   b.addSubElement(z);
   assertTrue(a.equals(b));

}
```

Post refactoring:

```
@Test(timeout = 4000)
public void test_22_1384() throws Throwable {
   XmlElement a = new XmlElement();
   XmlElement b = new XmlElement();
   a.addAttribute("p","a");
```

```
    b.addAttribute("p","a");
    assertTrue(a.equals(b));
}

public void test_22_1385() throws Throwable {
    XmlElement a = new XmlElement();
    XmlElement b = new XmlElement();
    XmlElement z = new XmlElement();
    a.addSubElement(z);
    b.addSubElement(z);
    assertTrue(a.equals(b));
}
```

8.7.2 RQ2

Per quanto riguarda il cambio di copertura, bisogna considerare che al variare dello smell il refactoring può avere un impatto più o meno significativo. Se stiamo considerando ad esempio lo smell Assertion Roulette il refactoring non risulta essere particolarmente impattante, questo perché lo smell in questione è possibile risolverlo semplicemente andando ad aggiungere un messaggio all'asserzione, quindi se stiamo considerando di voler rifattorizzare il seguente metodo:

```
	@Test(timeout = 4000)
	public void test_8_1271() throws Throwable {
		WeakHashtable ww = new WeakHashtable();
		try{
			ww.put(null,null);
			fail("Expected exception");
		}catch(NullPointerException npe){
			assertEquals("Null keys are not
allowed",npe.getMessage());
		}
		try{
			ww.put("shit",null);
			fail("Expected exception");
		}catch(NullPointerException npe){
			assertEquals("Null values are not
allowed",npe.getMessage());
		}
	}
```

Un refactoring che non avrà alcun impatto dal punto di vista della copertura sarà:

```
	@Test(timeout = 4000)
	public void test_8_1271() throws Throwable {
		WeakHashtable ww = new WeakHashtable();
		try{
			ww.put(null,null);
			fail("Expected exception");
		}catch(NullPointerException npe){
			assertEquals("First assert","Null keys are not
allowed",npe.getMessage());
		}
		try{
			ww.put("shit",null);
			fail("Expected exception");
		}catch(NullPointerException npe){
			assertEquals("Second assert","Null values are
not allowed",npe.getMessage());
		}
	}
```

Dopo aver considerato l'ipotesi che ci sono degli smell che si prestano in modo meno invasivo ad un refactoring e degli smell che necessitano un cambiamento strutturale, andiamo a considerare il il cambiamento in termini di percentuale di copertura apportato dopo il refactoring di ogni smell per le classi considerate precedentemente:

Classe	% Copertura pre refactoring	%Copertura post refactoring	Metodi refattorizzati
WeakHashTable	54	54	8
Subject Parser	77	77	18
Inflection	98	98	11
Range	98	94	24
XMLElement	94	94	22

Tabella 4: Risultati RQ2

Possiamo notare che nella maggior parte dei casi la copertura è rimasta identica, quindi non è stato possibile trovare elementi interessanti durante la sperimentazione per quanto riguarda una soglia di copertura accettabile. L'unico caso interessante riguarda la classe Range, in cui lo smell che risulta apportare un cambiamento dal punto di vista della copertura è Sensitive Equality, di cui in seguito consideriamo il test:

```
	@Test(timeout = 4000)
	public void test_21_1320() throws Throwable {
		Range<Integer> r = Range.between(new Integer(10),
new Integer(50));
		assertEquals("{10:50}", r.toString("{%1$s:%2$s}"));
	}
```

In questo caso la soluzione allo smell di "Sensitive Equality" sappiamo essere l'implementazione di un nuovo metodo che permetta di effettuare il confronto che prima avveniva attraverso il metodo toString. Possiamo quindi considerare un cambiamento della copertura poiché il metodo toString, appartenente alla classe che stiamo testando, non viene più chiamato, con conseguente riduzione di copertura. Se ne deduce il fatto che per quanto riguarda l'assegnazione di una soglia in termini numerici di copertura accettabile questa può variare di molto in base all'esercizio considerato. Un'ipotesi sarebbe quella di far settare allo scrittore dell'esercizio di testing una soglia cucita su misura sull'esercizio in questione in modo tale che vengano considerati casi particolari come quello del refactoring di smell che comportano un cambiamento strutturale del codice.

8.7.3 RQ3

In questo caso viene considerato il tempo necessario alla rimozione di tutti gli smell presenti all'interno delle varie classi di test.

- Quando una cella è colorata di nero vuol dire che lo smell in questione non è presente nella classe.
- Quando una cella è colorata di colore rosso vuol dire che lo smell all'interno della classe non può essere risolto.
- Quando una cella è colorata di colore giallo vuol dire che lo smell all'interno della classe può essere risolto parzialmente con modifiche che necessitano un refactoring più profondo.

All'interno di ogni cella è presente il tempo richiesto, e tra parentesi tonde è presente il numero di occorrenze dello specifico smell.

Class/Smell	Assertion Roulette	Eager Test	Duplicate Assert	Magic Number Test	Sensitive Equality
WeakHashTable	1,22 m (3)	1,31 m (4)	1,36 m (1)	20 s (1)	
Subject Parser	6,57 m (15)	(17)			
Inflection	2,22 m (11)				
Range	10 m (21)	(16)		10,06 m (9)	1,45 m (2)
XmlElement	10,06 m (16)	(8)		4,06 m (7)	

Tabella 5: Risultati RQ3

Nel caso in cui stiamo considerando le tabelle di colore rosso o giallo il tempo oppure il numero di smell non è presente siccome non è stato possibile risolvere correttamente tutti gli smell.

In questo caso stiamo considerando che il tempo medio per smell è una variabile aleatoria di cui gli smell provati nelle singole applicazioni sono i campioni, avendo quindi in questo caso cinque campioni.
I tempi esposti considerano il tempo impiegato per identificare lo smell all'interno della classe. Questa caratteristica si traduce nel fatto che il tempo può variare leggermente nel caso in cui stiamo considerando classi di grandi dimensioni, in cui una dimensione importante della classe da rifattorizzare incide sulla corretta comprensione dello smell da risolvere.
Analizzando i dati presenti nella tabella precedente è possibile quindi affermare che non ci sia una stretta correlazione tra numero di metodi di test, numero di righe di cui la classe si compone, e tempo necessario richiesto per effettuare il refactoring della classe di test, poiché molto dipende dal tipo di smell presente, dalla complessità del test in questione e dal fatto se lo smell è possibile fixarlo o meno. Risulta quindi difficile poter affermare di trovare un criterio di difficoltà di un esercizio di test.

Classe di test	LOC	#Metodi di test	# Test smells	Tempo impiegato (cumulativo)
WeakHashTable	133	11	9	4,49 m
SubjectParser	236	17	32	6,57 m
Inflection	173	26	11	2,22 m
Range	339	35	48	21,51 m
XmlElement	512	49	32	14.12 m

Tabella 6: Valutazione complessità Classi di test

Le linee di codice (LOC) considerate all'interno della tabella comprendono le righe vuote.

8.8 Conclusioni

Dopo lo studio di validazione considerato nei paragrafi precedenti possiamo notare una forte correlazione tra la tipologia di smell, il tempo necessario a risolvere lo smell, ed il cambiamento in termini di copertura del metodo di test. È stato possibile quindi considerare che alcuni smell non impattano particolarmente in termini di tempo rispetto alle tipologie di smell discusse nei capitoli precedenti. Tra gli smell di più facile risoluzione possiamo possiamo ricordare:

- Assertion Roulette
- Magic Number

Dove la risoluzione consiste in un approccio meccanico che non va ad impattare né sulla struttura né sulla complessità del test considerato, nello specifico:
1) Nel caso dell'assertion roulette basta considerare un messaggio di asserzione come primo parametro per risolvere lo smell.
2) Nel caso del Magic Number basta definire una variabile al posto del letterale per risolvere lo smell.
Andando ad analizzare la tabella 6, è stato possibile considerare inoltre che la complessità dei test non è direttamente riconducibile al numero di metodi presenti all'interno della classe da testare, né al numero di Linee di codice.
Lo strumento in questione, che si propone come uno strumento di supporto all'insegnamento, necessita quindi di una strutturazione adeguata degli esercizi e dei parametri relativi agli esercizi da parte di un docente esperto riguardo la tematica di software testing, in modo tale da impostare correttamente i parametri e scrivere dei test esemplari per un'adeguata comprensione degli smell e dei casi particolari che possiamo riscontrare riguardo la tematica dei test smells.

Bibliografia

1. Algaze, B. “Software Is Increasingly Complex. That Can Be Dangerous.” *Https:// Www.Extremetech.Com/Computing/259977-Software-Increasingly-Complex- Thats-Dangerous*, https:// www.extremetech.com/computing/259977-software-increasingly-complex-thats-dangerous.
2. Garousi, Vahid, et al. “Software-Testing Education: A Systematic Literature Mapping.” *Journal of Systems and Software*, vol. 165, July 2020, p. 110570, doi:10.1016/j.jss.2020.110570.
3. Uddin, Azeem, and Abhineet Anand. “Importance of Software Testing in the Process of Software Development.” *Unknown*, 1 Jan. 2019, https://www.researchgate.net/publication/331223692_Importance_of_Software_Testing_in_the_Process_of_Software_Development.
4. Patrone, Davide, and Michele Viscardi. *Gamification of Software Testing Activities: A Systematic Mapping*.
5. Sánchez, Ana B., et al. “Mutation Testing in the Wild: Findings from GitHub.” *Empirical Software Engineering*, vol. 27, no. 6, July 2022, doi:10.1007/s10664-022-10177-8.
6. Hamimoune, Soukaina, and Bouchaib Falah. “Mutation Testing Techniques: A Comparative Study.” *2016 International Conference on Engineering & MIS (ICEMIS)*, IEEE, 2016, http://dx.doi.org/10.1109/icemis.2016.7745368.
7. García, Félix, et al. “A Framework for Gamification in Software Engineering.” *Journal of Systems and Software*, vol. 132, Oct. 2017, pp. 21–40, doi:10.1016/j.jss.2017.06.021.
8. Werbach, Kevin, and Dan Hunter. *For the Win, Revised and Updated Edition*. Wharton School Press, 2020, http://dx.doi.org/10.2307/j.ctv2hdrfsm.
9. Aljedaani, Wajdi, et al. “Test Smell Detection Tools: A Systematic Mapping Study.” *Evaluation and Assessment in Software Engineering*, ACM, 2021, http://dx.doi.org/10.1145/3463274.3463335.
10. Bamizadeh, Lida, et al. “An Analytical Study of Code Smells.” *Tehnički Glasnik*, vol. 15, no. 1, Mar. 2021, pp. 121–26, doi:10.31803/tg-20210205095410.

11. Bavota, Gabriele, et al. “An Empirical Analysis of the Distribution of Unit Test Smells and Their Impact on Software Maintenance.” *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, IEEE, 2012, http://dx.doi.org/10.1109/icsm.2012.6405253.
12. Baker, P., et al. “TRex - The Refactoring and Metrics Tool for TTCN-3 Test Specifications.” *Testing: Academic & Industrial Conference - Practice And Research Techniques (TAIC PART’06)*, IEEE, http://dx.doi.org/10.1109/taic-part.2006.35. Accessed 24 Jan. 2023.
13. van Deursen, Arie, et al. “Refactoring Test Code.”, 29 Aug. 2001, https://www.researchgate.net/publication/2534882.
14. Wang, Yung-Fu, et al. “The Key Elements of Gamification in Corporate Training – The Delphi Method.” *Entertainment Computing*, vol. 40, Jan. 2022, p. 100463, doi:10.1016/j.entcom.2021.100463.
15. Fraser, Gordon, et al. “Gamifying a Software Testing Course with Code Defenders.” *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, ACM, 2019, http://dx.doi.org/10.1145/3287324.3287471.
16. Clegg, Benjamin S., et al. “Teaching Software Testing Concepts Using a Mutation Testing Game.” *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering Education and Training Track (ICSE-SEET)*, IEEE, 2017, http://dx.doi.org/10.1109/icse-seet.2017.1.
17. Pedreira, Oscar, et al. “Gamification in Software Engineering – A Systematic Mapping.” *Information and Software Technology*, vol. 57, Jan. 2015, pp. 157–68, doi:10.1016/j.infsof.2014.08.007.
18. Porto, Daniel de Paula, et al. “Initiatives and Challenges of Using Gamification in Software Engineering: A Systematic Mapping.” *Journal of Systems and Software*, vol. 173, Mar. 2021, p. 110870, doi:10.1016/j.jss.2020.110870.
19. *ACM Digital Library*, https://dl.acm.org/doi/10.1145/3379597.3387453.
20. *ACM Digital Library*, https://dl.acm.org/doi/abs/10.1145/3350768.3352490.