

POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Informatica



Tesi di Laurea Magistrale

Gamification applied to Java development and testing

Relatori

Prof. Marco TORCHIANO

Prof. Riccardo COPPOLA

Dott. Tommaso FULCINI

Candidato

Matteo FAVRETTO

Ottobre 2023

Summary

The importance of software testing has risen enormously given the ever-increasing diffusion and importance of software in the modern world. However, this activity tends to often be overlooked and neglected, especially in didactic contexts, because of its intrinsic not creative, repetitive nature.

The goal of this thesis work is to apply gamification concepts, such as competition and the completion of goals, to the teaching of software testing concepts to students, to increase their engagement with the activity and, in turn, promote their understanding of the concepts and their importance.

The means to do so is Unit Brawl, an application designed to be employed by students and teachers in the context of the laboratory sessions that are held throughout the Object-Oriented Programming course for the Bachelor's Degree in Computer Engineering at the Politecnico di Torino. Students can use the application to track their progress in the requirements for laboratory assignments and gauge the quality of the tests they produce. Additionally, the application implements a competition among students based on their submitted implementations and tests: students are assigned points when the opponents' submissions fail on their tests, and when their submission passes an opponent's tests.

This work analyses the data regarding the usage of the platform by the students and their results during the laboratory session, assessing the impact of the gamification mechanics and planning future developments for them.

Acknowledgements

This thesis marks the end of a long, winding path that shaped the man I am today in more ways than I can express. It is therefore appropriate to take a moment to express my gratitude towards the people who have accompanied me on this journey.

Starting from my family, and specifically my mom and dad, you have been by my side all along, supporting me in all my choices in more ways than I express. Thank you for all the sacrifices you've made to help me along the way.

To my grandparents, the ones still with us as well as Nonna Rina and Nonno Cesco. I hope I have made you proud.

To my friends. There is no need for me to list all of you, you know who you are. I could have not asked for better friends, and I genuinely mean that. A lot of the person I am today comes from having such a special group of people around me. I hope you know how much you mean to me. Here's to the next sticker albums we'll make.

And of course thanks to Professors Marco Torchiano and Riccardo Coppola, and Doctor Tommaso Fulcini for the help in producing this thesis. It has been by far the largest undertaking of my life so far, and your contribution has made the difference.

Table of Contents

List of Tables	VII
List of Figures	VIII
Acronyms	X
1 Introduction	1
2 Background	3
2.1 Software Testing	3
2.1.1 Software Testing Life Cycle	4
2.1.2 Test-Driven Development	6
2.1.3 Test Classification	6
2.2 Gamification	7
2.2.1 The Octalysis framework	8
2.2.2 Gamification in Software Testing	11
3 Design	14
3.1 Context of the application	14
3.2 Overview of the gamified process	15
3.2.1 Phase 1: Developing and submitting a solution for the as- signment	15
3.2.2 Phase 2: Free-for-all	17
3.3 Gamification mechanics	19
3.3.1 Requirements List and Coverage Dashboard	19
3.3.2 Scarcity of Time	21
3.3.3 Avatars and Customization	21
3.3.4 Free-for-all, Scores and Leaderboards	21
4 Implementation	24
4.1 Server	24

4.1.1	Database	25
4.2	Front-ends	25
4.2.1	Player front-end	26
4.2.2	Admin front-end	30
4.3	Dockerization	31
4.3.1	GitLab Continuous Integration	34
5	Results	37
5.1	Experiment organization	37
5.1.1	Session 1: Control group (No platform)	37
5.1.2	Session 2: Platform Version 1	38
5.1.3	Session 3: Platform Version 2	40
5.2	Results and students' feedback analysis	41
6	Conclusions	43
6.1	Possible future developments	44
	Bibliography	46

List of Tables

2.1	The core drives of the Octalysis framework	10
3.1	Gamification mechanics	22
5.1	Participation and test quality data for Session 1	38
5.2	Participation and test quality data for Session 2	39
5.3	Participation and quality data for the Session 3	40

List of Figures

2.1	Software Testing Life Cycle phases.	5
2.2	A view of CodeDefender’s graphical user interface.	12
2.3	A view of Testable’s graphical user interface	13
3.1	Requirements table displayed upon checking progress.	16
3.2	Coverage dashboard presenting, from left to right, instruction coverage percentage, method coverage percentage, and class coverage percentage.	17
3.3	Example of a session with three players	18
3.4	Requirements table showing a problem with the first requirement.	20
3.5	Coverage dashboard presenting, from left to right, instruction coverage percentage, method coverage percentage, and class coverage percentage.	20
3.6	Octalysis analysis of the gamification features	23
4.1	UML diagram representing the core application structure	25
4.2	Navigation bar	26
4.3	Initial view of the Progress tab	27
4.4	Requirement table	27
4.5	Requirement table	28
4.6	Coverage dashboard	29
4.7	Global leaderboard displaying the complete ranking, with the top user highlighted in gold and the current user in green	29
4.8	Local leaderboard displaying only the user and the opponents immediately above and below them	30
4.9	Initial Docker stack structure for the core containers of the application	33
4.10	Complete Docker stack structure, including the Nginx proxy container	34
5.1	Test result distribution in Session 1	38
5.2	Test result distribution in Session 2	40
5.3	Test result distribution in Session 3	41

Acronyms

BPMN

Business Process Model and Notation

CI

Continuous Integration

PO

Programmazione a Oggetti

SUT

System Under Test

UML

Unified Modeling Language

Chapter 1

Introduction

In the modern world, software is becoming more and more pervasive. From Internet-of-Things to smartphones, to cloud services, and so on, code and software influence an increasing part of human life. With such an ever-increasing importance, the importance of testing such software has been growing as well. Testing software means, in general, verifying that it behaves as expected, both during its normal operations and when it needs to handle malfunctions. Consequences of inaccurate testing, or lack thereof, can be dire, and examples of this are not uncommon: from the NASA orbiter that crashed on Mars in 1999 due to its software using the Imperial system of measurement rather than the metric one [1] to the seven years in which the hole in the ozone layer over Antarctica remained undetected (from 1978 to 1985) because the data analysis software used by NASA would ignore values that deviated from the expected range, or the bug in the Therac-25 medical radiation therapy device that caused it to emit a much higher dose of radiation than the safe amount, causing at least three patients to die as a direct consequence of radiation overdose [1], the costs of poorly tested software can be enormous, both in monetary terms and in human lives. And even when it does not come to such dramatic events, producing low-quality, untested code adds to the so-called *technical debt* of a project: whenever imperfect or untested code is produced, this figurative debt grows, and sooner or later developers will have to pay it back, in the form of reviewing old code, testing it and, often, fixing problems that may have been noticed much earlier, and fixed much easier, if only the code had been tested sooner.

Despite all the reasons and arguments that can be made for software testing, it is still an activity that is often overlooked, neglected, or done poorly. This also happens in academic environments: students who are taught software testing concepts do not find the activity particularly engaging, because of its repetitive, scarcely creative nature. Furthermore, once an assignment has been completed, at least apparently, testing the implementation might seem unnecessary.

Therefore, this thesis work aims to develop a tool to promote software testing in an academic context, incentivizing students to test the software they produce. To do so, the concept of gamification has been employed. Gamification refers to *the use of design elements characteristic of games in non-game context* [3], to facilitate the engagement of users in a given activity, and it has been successfully employed in various fields, from teaching to organizational productivity, knowledge retention, crowdsourcing and more. Specifically, this work focuses on the development of Unit Brawl, a platform to promote and gamify the process of developing and testing Java code. This platform has been made available to the students of the Object-Oriented Programming course in the Bachelor Degree in Computer Engineering, for the academic year 2022-2023.

This thesis work will follow the following structure:

- Chapter 2 - Background will present the main concepts that this work is based on: from software testing, its life cycle and various types, to gamification and the specific framework that was considered for this work, while also presenting some existing examples of gamified tools for software testing
- Chapter 3 - Design will illustrate the mechanics of the application, analyzing the concepts and ratio behind them
- Chapter 4 - Implementation will explain the application's architecture, from the server in the back-end to the two different front-ends, as well as the way it was packaged in a Docker container and deployed on the Politecnico's website
- Chapter 5 - Validation will present the results of the employment of the application in the context of the course
- Chapter 6 - Conclusions will review the overall work and present some general considerations of it, as well as possible further analysis and developments to be carried out in the future

Chapter 2

Background

Unit Brawl aims at promoting high-quality software testing using gamification elements. The goal of this chapter is to delve into what software testing and gamification are. In particular, the various phases of software testing will be presented, as well as the various possible types of software testing according to the most common classifications.

2.1 Software Testing

Software Engineering is a branch of engineering that focuses on the *design, development, testing, and maintenance of software applications*. The design phase includes the planning of a solution to target the requirements and needs. Development refers to the actual implementation of the design produced in the previous step. Testing, which is not necessarily a subsequent step to development, but rather an integral part of it that allows to formally and empirically guarantee that the software being developed behaves as expected. Lastly, maintenance refers to the process of monitoring the completed system's activity to detect possible anomalies and resolve them as quickly and effectively as possible, as well as updating the system throughout its lifecycle, adding new features, refining existing ones, and correcting possible implementation errors. The focus of this work will be on the testing phase, a vital but sometimes overlooked part of Software Engineering. The IEEE Standard Glossary of Software Engineering Terminology provides the following definitions [5]:

- **test**: an activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspects of the system or component
- **verification**: the process of evaluating a system or component to determine

whether the products of a given development phase satisfy the conditions imposed at the start of that phase

- **validation:** the process of evaluating a system or a component during or at the end of the development process to determine whether it satisfies specified requirements

We can therefore define software testing as an activity carried out to verify whether a given application or one of its components satisfies the requirements that were specified for it, by recording or observing its behaviour.

2.1.1 Software Testing Life Cycle

The Software Testing Life Cycle (STLC) [10] provides a formal framework to ensure that a given piece of software meets its requirements and is free of defects. This framework is articulated in a series of phases, each of which presents specific deliverables. The overall goal across the whole process is to find and document any possible defect of the software application as early as possible: the earlier in the process a defect is discovered, the easier it is to address and resolve it with lower costs, both in terms of money and effort required.

The Software Testing Life Cycle consists of the following phases:

1. **Requirement Analysis:** this is the phase where the requirements are defined as clearly as possible, consulting with the application's stakeholders to clarify potential ambiguities.
2. **Test Planning:** during this phase, the efforts and costs of the testing activity are estimated, and a testing strategy is developed by selecting the testing methods and techniques that will be used.
3. **Test Case Development:** the goal of this phase is to produce detailed test cases, possibly also including the needed test data. The test cases produced must cover the scope that has been defined in the earlier phases.
4. **Test Environment Setup:** this activity can be carried out in parallel with the previous phase, and consists in creating the environment in which the tests will be executed. This can include hardware configuration, operating system settings, software configuration, test terminals, and more.
5. **Test Execution:** during this phase, the previously assembled test cases are executed in the selected environment, recording eventual defects found as well as their severity and priority. It is important to emphasize that this phase may need to be repeated multiple times, until all defects have been found and fixed, and the software is deemed ready for the release.



Figure 2.1: Software Testing Life Cycle phases.

6. **Test Closure:** this final stage of the process consists of documenting all the results obtained, including the defects found and solved, as well as all the activities carried out during the testing process. This phase should also include the recording of feedback that can be useful to improve future testing processes.

2.1.2 Test-Driven Development

An area in which software testing is of particular importance is the test-driven development (TDD) approach [6]. Test-driven development is defined by D. Janzen and H. Saiedian as *writing automated tests prior to developing functional code in small, rapid iterations*. The idea is to convert the application's requirements into test cases that reflect them, and the development process should aim at producing code that passes such tests. In *JUnit in Action* (Manning Publications, 2003) V. Massol and T. Husted have stated that *Test-Driven Development (TDD) is a programming practice that instructs developers to write new code only if the automated test has failed and to eliminate duplications. The goal of TDD is 'clean code that works'*[8]. We can also find another definition of TDD by the Agile Alliance [2]: *"Test-Driven Development" refers to a style of programming in which these activities are tightly interwoven: coding, testing (in the form of writing unit tests), and design (in the form of refactoring). It can be succinctly described as the following set of rules:*

- *write a "single" unit test describing an aspect of the program*
- *run the test, which should fail because the program lacks that feature*
- *write "just enough" code, the simplest possible, to make the test pass*
- *"refactor" the code until it conforms to the simplicity criteria*
- *repeat, "accumulating" unit tests over time*

Overall, Test-Driven Development is reported to significantly reduce defect rates and the effort required by the final phases of the development process, at the cost of a moderate increase in the initial effort. Test-Driven Development has an integral role in Extreme Programming (XP), an Agile development methodology that aims at developing object-oriented software in very short iterations with little upfront design[6].

2.1.3 Test Classification

It is possible to classify software testing techniques according to different criteria. A first example would be the classification based on the specificity of the test:

- **unit tests** aim at verifying the correctness of single software units, e.g. methods or classes.
- **integration testing** is aimed at checking the interactions between multiple software units

- **system testing** checks the system as a whole
- **user interface (UI) testing** focuses on the user interface, testing the possible interactions between the user and the system
- **acceptance testing** is used to determine to what degree an application meets the end users' approval[4]

Another form of test classification considers the technique being adopted, and the knowledge of the element under test that is available to the tester:

- **black-box testing** is performed having no notion of the internal implementation of a software unit or application and only considers input and expected vs actual output
- **white-box testing** requires full knowledge of the implementation of the element being tested and aims at covering all possible code execution paths within it
- **mutation testing** aims at designing new tests and evaluating the quality of existing ones, by altering a program in a small way and checking whether the test suite can detect the change (or mutation)
- **exploratory testing** has individual users freely explore the application, interact with it, and hopefully discover defects that were not covered in the scope of other tests[9]

Lastly, other types of tests include:

- **performance testing** that focuses on verifying that the application satisfies performance-related requirements
- **stress testing** determines the robustness of an application when operating beyond the limits of normal operation[12]
- **penetration testing** is used to assess the system's security and its ability to withstand an attack
- **usability testing** is used to assess the effectiveness and ease of the user experience

2.2 Gamification

Gamification is defined by Deterding et al. as *the use of design elements characteristic of games in non-game contexts*[3]. It is often employed as a tool to increase

and facilitate the engagement of users, a practice that is used in various contexts, such as teaching, organizational productivity, knowledge retention, crowdsourcing, and more.

The game design elements used for gamification have been formalized by Robson et al. in the MDE (Mechanics - Dynamics - Emotions) framework[11]. *Mechanics* are the decisions made by the designers of the gamified experience to communicate to the users the goals, rules, settings, context, types of interactions, and boundaries that are in place in the experience. They are constant, meaning that they do not change from player to player or from the first time a user engages with the experience to the next time. *Dynamics* emerge from the mechanics put in place by the designer of the experience as how the user follows the mechanics to engage with the activity. They refer to the strategic decisions and interactions that occur in the experience. They pose a challenge to the designer of the experience, since they are not always predictable with absolute certainty, and can lead to unintended behaviors, which can be positive or negative in relation to the goal of the experience. Lastly, *emotions* are the mental affective states that are evoked within the participants of the gamified experience. They can be viewed as the product of the mechanics put in place, and the generators of the dynamics that occur among the participants during the experience.

2.2.1 The Octalysis framework

The framework that guided the design of the application is the Octalysis Framework, created by Yu-kai Chou and presented in his book *Actionable Gamification - Beyond Points, Labels and Leaderboards*[7]. This framework identifies eight main gamification mechanics groups, called core drives, based on the psychological aspects on which the various mechanics are based. The eight core drives are the following:

- **Epic Meaning & Calling:** the author defines this core drive as “the drive where a player believes that he is doing something greater than himself or he was ‘chosen’ to do something”. It is a drive that is often employed by adding a narrative layer to a gamified experience.
- **Development & Accomplishment:** this drive is based on the feeling of progress, developing skills, and overcoming a challenge.
- **Empowerment of Creativity & Feedback:** this drive aims at involving the player in the creative process, within the boundaries of the experience, allowing them to express themselves and receive feedback about the resulting product of his creativity.
- **Ownership & Possession:** it is the drive based on the idea of making the player feel as if they own something, such as virtual currencies or goods within

the context of the gamified experience, and making them protect and improve what they own.

- **Social Influence & Relatedness:** this drive revolves around social dynamics within the context of the gamified experience, both in a positive light, such as mentorship, alliances, and companionship, and in a negative light, such as competition and envy.
- **Scarcity & Impatience:** it is a drive based on the idea of promising something to the player that they cannot obtain right away, but they have to wait or put in effort to obtain it.
- **Unpredictability & Curiosity:** his drive is based on the sense of anticipation and curiosity generated by not knowing what is going to happen, implying random-based elements or mechanics that are undisclosed to the user to stimulate their engagement
- **Loss & Avoidance:** lastly, this drive exploits the desire to prevent something bad from happening, for example, an event that might cause all the progress to be lost.

It is possible to group the core drives according to different criteria. Note that the following classifications and the nomenclature they use do not aim at having scientific value, but only at providing a general understanding of their meaning.

Left-Brain Drives VS Right-Brain Drives

The first classification is the one that divides the drives into two groups, called left-brain and right-brain drives. The eight drives have been arranged in the octagon representing the framework according to this definition, therefore Accomplishment, Ownership, and Scarcity are left-brain drives, while Empowerment, Social Influence, and Unpredictability are right-brain drives. Left-brain drives are the ones that are more closely related to logic, calculations, and ownership, and are based on *extrinsic motivation*, meaning that they provide motivation based on the desire to obtain something. Right-brain drives, on the other hand, relate more closely to creativity, self-expression, and social aspects, and are based on *intrinsic motivation* referring to the fact that they rely on the intrinsic appeal of the activity they entail rather than on the promise of reaching a goal or obtaining an item.

We can observe how left-brain drives are the most employed because they are the easiest to implement and provide the quickest results. However, studies have shown that if the user stops being offered the extrinsic motivator, their motivation will decrease to much lower than when the motivator was first offered. Moreover, excessively employing left-brain drives might lead to stagnation and possibly drive

users away from the experience. Right-brain techniques are therefore needed, considering how they can provide the user with a much deeper motivation and connection to the experience, when used properly.

White-Hat Mechanics VS Black-Hat Mechanics

A second possible classification identifies two groups of mechanics, called *white-hat* and *black-hat*. While the previous classification was based on the psychological traits and aspects exploited by the mechanics, this classification is based on the emotions and sensations that are evoked by the mechanics. White-hat mechanics aim at producing a positive feeling in the user, such as satisfaction for having reached a goal or a sense of control and pride for the items they own, while black-hat mechanics are designed to provide negative sensations in the user, such as anxiety for a possible negative event occurring in the context of the experience.

The two groups are best suited for different purposes and contexts: black-hat mechanics are effective for scenarios like an e-commerce online store, in which the anxiety of missing out on an offer can prompt users to perform purchases that they would not otherwise have. While dealing with a longer-running experience, however, black-hat mechanics can have the opposite effect: if the user associates their experience with the product solely to feelings such as addiction or anxiety, it is only natural that, over time, they will be driven away from the product by the compounding of those sensations, or by the discovery of new and more appealing stimuli. This is where white-hat mechanics can prove effective, balancing the black-hat ones and keeping the user engaged with the product. It is worth noting that black-hat mechanics are not always and only negative: when employed in a balanced and transparent fashion they can prove extremely effective in approaching new users.

Core Drive	Left/Right Brain	Black/White Hat
Epic Meaning and Calling	-	White
Development and Accomplishment	Left	White
Empowerment of Creativity and Feedback	Right	White
Ownership and Possession	Left	-
Social Influence and Relatedness	Right	-
Scarcity and Impatience	Left	Black
Unpredictability and Curiosity	Right	Black
Loss and Avoidance	-	Black

Table 2.1: The core drives of the Octalysis framework

2.2.2 Gamification in Software Testing

As previously mentioned, gamification elements can be effectively employed in various contexts. A typical example of this is their use as an educational tool, promoting students' engagement by providing a novel approach to the topics being taught and stimulating a more active interaction with them from the students. This thesis focuses on this context specifically, analyzing the effects of gamification elements when applied in teaching software testing concepts and techniques.

As a preparatory analysis, a few tools and platforms that use gamification in the context of software testing have been selected and examined.

The first tool presented is CodeDefenders. It is a competitive tool in which two teams face each other working on a Java class: one team plays the role of the attacker, while the other plays as the defender. The goal of the attacker is to create *mutants*, which are variants of the starting class, while the defender has to write tests that discover and "kill" the mutants. The defender team can see where the attackers have introduced a mutant, but they cannot see what has been changed. The defenders' goal is, therefore, to write unit tests that pass against the original class but fail against the mutant, to discover and "kill" it. Once a mutant has been killed, the defenders can see what changes have been made. The goal of the attackers, as mentioned, is to introduce variants of the starting class that go undetected. However, it could be possible for an attacker to create a variant of the class that is semantically different from the original class, but functionally the same. In other words, attackers can write an alternative implementation of an existing class that behaves in the same way as the original. By doing so they would create a so-called *equivalent mutant*, that would be undetectable using unit tests because it would pass all the tests that pass on the original class. If the defender thinks that the attacker wrote an *equivalent mutant*, the defender can accuse the attacker, who can then submit a test that succeeds on the original class but fails on the mutant to prove that it was not, in fact, an equivalent mutant. The game is played by assigning points to the players, to the defender when they write tests that find and kill mutants and when they correctly claim that the attacker wrote an equivalent mutant, and to the attacker when they write mutants that go undetected by the tests, or when they provide tests that successfully prove that a mutant was not equivalent when accused by the defender.

We can refer to the Octalysis framework and identify the core drives employed in the design of CodeDefenders:

- **Epic Meaning & Calling:** the narrative layer referring to attackers and defenders constructed on the application leverages this core drive, allowing the players to abstract their activity from mere coding.
- **Development & Accomplishment:** by providing a scoring system and leaderboards, the player can have clear feedback on their results versus the other

players.

- **Empowerment of Creativity & Feedback:** this core drive is intrinsically tied to the coding activity in general since the way mutants and tests are written heavily depends on the single player’s methods.
- **Social Influence & Relatedness:** by splitting players into teams and having them compete against each other, social dynamics are put in place.

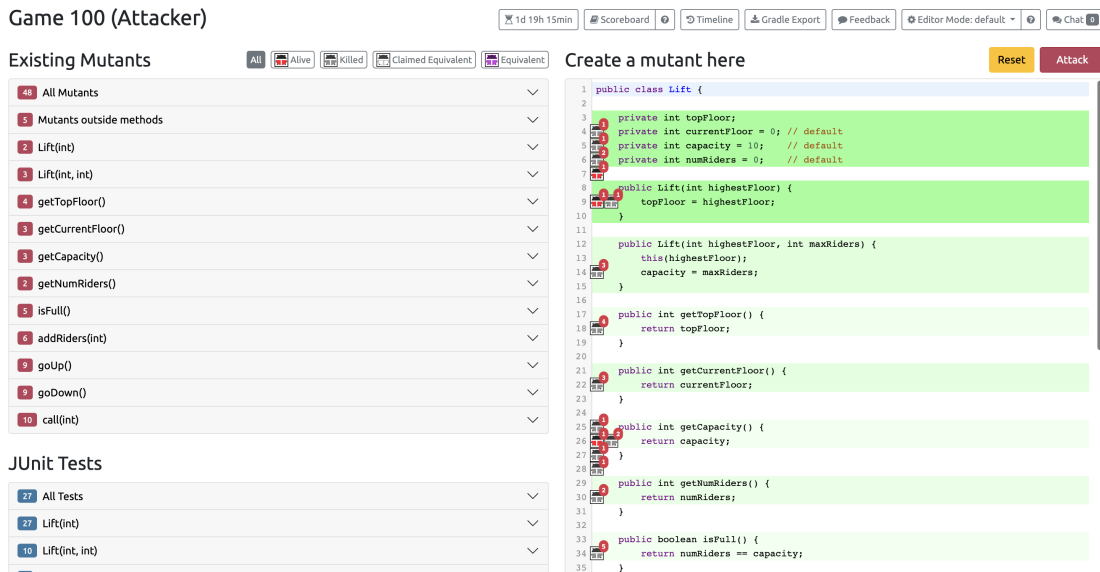


Figure 2.2: A view of CodeDefender’s graphical user interface.

Another interesting case of the application of gamification in software testing is represented by the framework developed by Costa et al., which is of particular interest for the articulated narrative layer that it employs, leveraging the Epic Meaning & Calling core drive from the Octalysis framework. Notably, this framework also represents an example of gamification in the context of exploratory testing. Participants are identified by one of three roles: a specialist, having full knowledge of the system being explored, a judge recording the actions performed by the users, and testers, who are the students themselves, freely exploring the system. Their goal is to explore the system, noting possible defects and flaws, recording them, and assigning them a priority level. A report of their activity is then generated, and in the second part of the experience, the testers can examine and grade each other’s reports, according to specified criteria and provide a justification for the scores they assign. The specialist will then proceed to examine the actions recorded by the judge in the first part of the activity and the evaluations submitted in the second part, rewarding the players based on their performance.

Another core drive leveraged by the framework is the Ownership & Possession one, providing the users with customizable avatars and the possibility to exchange the points they earn with customization items. Other notable core drives used by the framework are Social Influence & Relatedness, for the competitive component of the activity, Development & Accomplishment, introducing a scoring system based on performance, and Unpredictability & Curiosity, for the chance to obtain bonus points assigned by the specialist on the base of merit and performance.

Such a framework highlights the positive value that a strong narrative component can have on the testing experience: users reported a liking for the narrative context, which was based on the cinematographic franchise “Pirates of the Caribbean”.

Lastly, the Testable tool represents yet another virtuous example of gamification in teaching software testing concepts and techniques, focusing on unit testing. Much like the previous framework, Testable provides a strong narrative component, in which the user helps Buggy, an insect wanting to learn programming, to write quality code by creating unit tests, competing against aliens, his adversaries. The narrative layer is supported by a detailed graphical user interface, which is coherent with the narrative premises. Moreover, the tool exploits the Social Influence & Relatedness core drive in a way that helps the tool itself: users are rewarded for sharing it on social media. This represents a way in which the drive can be used by designers to help the tools and gamified experiences gain popularity, offering rewards to the user.

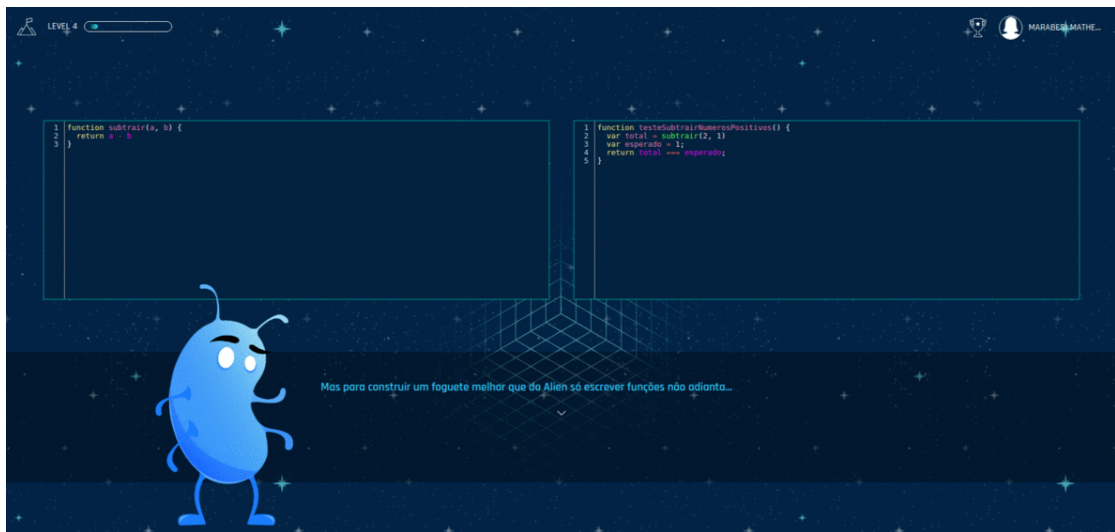


Figure 2.3: A view of Testable’s graphical user interface

Chapter 3

Design

The goal of this chapter is to present the context in which Unit Brawl is designed to be used, which is the Object-Oriented Programming course of the Bachelor Degree in Computer Engineering at Politecnico di Torino. Then the mechanics used in the application will be introduced, starting from the ones that were already in place at the start of this work, and then illustrating the ones I introduced or reworked.

3.1 Context of the application

Unit Brawl is designed to be a didactic tool helping to teach software testing concepts, with a particular focus on unit testing, during the Object-Oriented Programming course held in the Bachelor Degree in Computer Engineering at Politecnico di Torino. During the course, weekly laboratory sessions are carried out by the students, in which they receive an assignment to complete by implementing a simple Java application. Each student is given a personal repository, where they can push their solution. When the deadline for the assignment expires, a test suite is run on each student's solution, generating a report using Maven's Surefire and JaCoCo plugins, and each student receives their report by email.

This situation, however, presents some flaws and room for improvement. When students are presented with the concept of unit testing, they are encouraged to write unit tests using JUnit, to check the validity of their solution. Nevertheless writing such tests is not strictly part of the laboratories, and therefore several students either do not write them or write low-quality tests. In addition to this, due to the elective nature of the laboratories, some students opt not to participate in them.

Unit Brawl is an attempt to address these problems by introducing the unit tests as part of the requirements for each laboratory and providing the students with a platform to gamify the testing activity, promoting participation and the

production of high-quality tests.

3.2 Overview of the gamified process

This section aims to illustrate how the gamified process takes place during a session. This happens in two steps: first, the students develop and submit a solution that satisfies the requirements of the assignment, and second, after the expiration of the deadline for the session, the free-for-all competition takes place.

3.2.1 Phase 1: Developing and submitting a solution for the assignment

To achieve the main goal of the project, which is to promote students' engagement in the testing activity, the main catalyst that has been employed is competition among students. The majority of the mechanics that were introduced or reworked are aimed, directly or indirectly, to support this factor.


As previously mentioned, the goal for Unit Brawl is to create a platform to gamify the testing phase of the laboratories, concurrently promoting participation in the laboratory activities, which will last now two weeks instead of one. The gamified process will have the students write unit tests for the Java application they produce for each lab, using JUnit as a unit testing framework. The repositories for each student will be created on GitLab, whose Continuous Integration will also be used to generate intermediate reports. Finally, the students will be presented with a web platform they will be able to interact with to perform various operations, such as monitoring their progress in the laboratories.

From the publishing of the assignment, students have two weeks to implement the required classes and methods, which must have the signature specified in the assignment, and to produce unit tests for their solution, with a maximum number of tests allowed. The tests will be used not only by the students themselves to check the correctness of their solution, but they will also play a fundamental role in the competition that will take part after the deadline for the assignment expires.

After joining the assignment and providing a link to their repository, students will be able to access the "Progress" tab on the assignment page, which will allow them to check their progress so far, displaying a button. By pressing it, the students can trigger a progress check, in which the ideal tests are run on the last pushed submission, cloning it from the repository at the link that was provided upon joining the assignment. This check is designed to provide feedback to the students while they're carrying out the assignment and can be used even before the student has provided any unit test. After the check, the student is presented with the results:

- if something went wrong, for example, if the pushed solution does not compile, or the link they provided is invalid, an error message is displayed, explaining what happened
- If the tests were completed, a table containing the requirements is shown. For each requirement, the student can see whether it is completed, meaning that all the required methods for it have been implemented and have passed their tests, or if there are issues with it, such as missing methods or methods that fail their tests.

Tests report summary



Requirement	Status	Comment (Click on the failing methods for more info)
R1	Passed	All the requested methods pass their tests!
R2	Passed	All the requested methods pass their tests!
R3	Passed	All the requested methods pass their tests!
R4	Passed	All the requested methods pass their tests!

[Update requirements progress](#)

Figure 3.1: Requirements table displayed upon checking progress.

If the check is performed successfully (even in case of test failures), the student can proceed by checking the coverage of their tests. This check is designed to be used after the student has provided a significant amount of tests: if performed when no tests have been provided yet, or with a small number of them, the test will of course return a very low coverage, which would not be a significant result. By pressing the corresponding button, the coverage check is performed, running the student's tests on their solution and parsing the JaCoCo report that is generated. The results are then presented to the student as circular progress bars, showing the coverage percentage for classes, methods, and single instructions, to provide the student with more contextualized information than simply the instructions covered.

Since this check is performed on the student's tests, their number is also checked, notifying the student if it exceeds the limit specified for the assignment.

By visualizing the students' progress using progress bars and tables we are exploiting the Achievement core drive in the Octalysis framework: the visualization of the progress is a powerful tool that allows the student to have a more concrete idea of their accomplishments, and also a more precise feedback for where they need to improve.

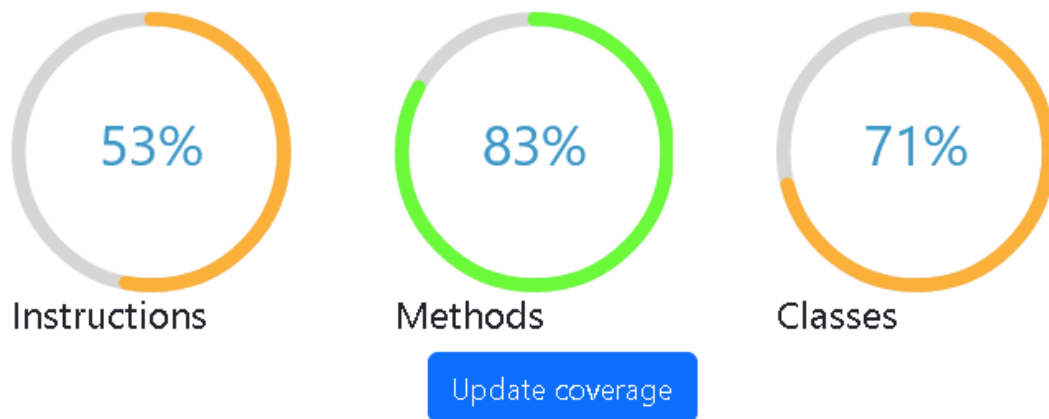


Figure 3.2: Coverage dashboard presenting, from left to right, instruction coverage percentage, method coverage percentage, and class coverage percentage.

3.2.2 Phase 2: Free-for-all

When the deadline for the assignment expires, the laboratory session is closed, and the free-for-all competition is carried out. The idea for this phase is to have students compete against each other on the quality of their solutions and tests. To do so, the application first collects the solutions and test suites submitted by all the participants in the session. Then it filters them according to several criteria:

- the solution must compile with the tests submitted by the student. A submission that does not compile cannot be admitted to the competition, because it would not be possible to run other students' tests on it
- The submitted tests must pass on the student's solution. This criteria is designed in conjunction with the following one, to prevent students from submitting tests that are specifically designed to fail, to get points from them failing on other students' submissions

- the number of submitted tests must be lower than or equal to the maximum number, specified in the assignment. This criterion is aimed at preventing students from submitting a large number of tests designed to fail, which would net them a high score by failing on all other students' submission
- The ideal solution for the session must compile correctly with the tests submitted by the student.
- the submitted tests must pass on the ideal solution used as a reference

As for the ideal tests that are part of the ideal solution, it has been deemed appropriate to assign a malus to the student if the submitted solution does not pass them, but still admit them to the final competition. After the filtering phase is complete, the remaining submissions are elaborated and a collective test suite is assembled by putting together the tests produced by all the students. This suite is then run on all the submissions, skipping, for each one, the tests written by the author of the submission. This is because each student's tests have already been executed on their own submission as part of the filtering phase. The score for each student is then computed from the results of the tests: each student gets points every time their submission passes an opponent's test and is, therefore, free from the flaw the test is designed to detect, and every time one of their tests fails on an opponent's solution, therefore identifying a flaw in the opponent's solution. As previously mentioned, failing the ideal tests results in a malus assigned to the student, which lowers the final score by 30%.

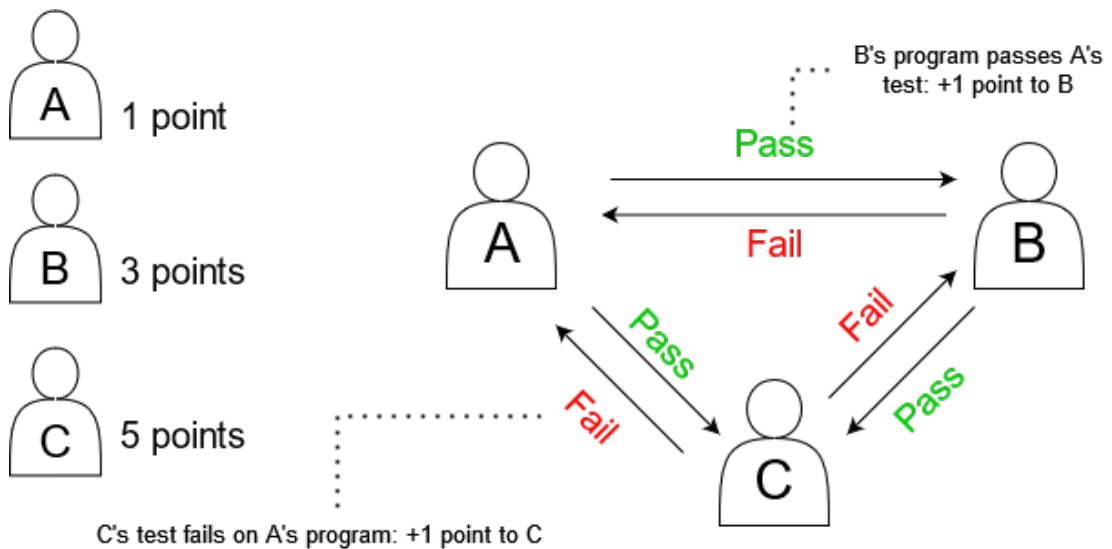


Figure 3.3: Example of a session with three players

Leaderboards are then assembled with the computed scores: a session leaderboard shows the rankings for the present session, while a global leaderboard shows the overall scores for all sessions.

3.3 Gamification mechanics

The goal of this section is to illustrate the various gamification mechanics used in the development of the application and the ratio for their design.

3.3.1 Requirements List and Coverage Dashboard

The requirement list and a progress bar that students see when checking their progress in the laboratory session leverages the Accomplishment core drive from the Octalysis frameworks: it is a sort of quest list, that visualizes the progress of the student in the assignment, showing them both their accomplishments, in the form of completed requirements, and where they need to improve, displaying a list of required methods that present issues (i.e. that fail their ideal tests). This helps the student have a more precise idea of where they made a mistake, and therefore where they need to improve. In case of incomplete requirements, the student can also click on the faulty method to see a message indicating the type of error that occurred in the test, to contextualize the mistake and get useful information about it.

Tests report summary

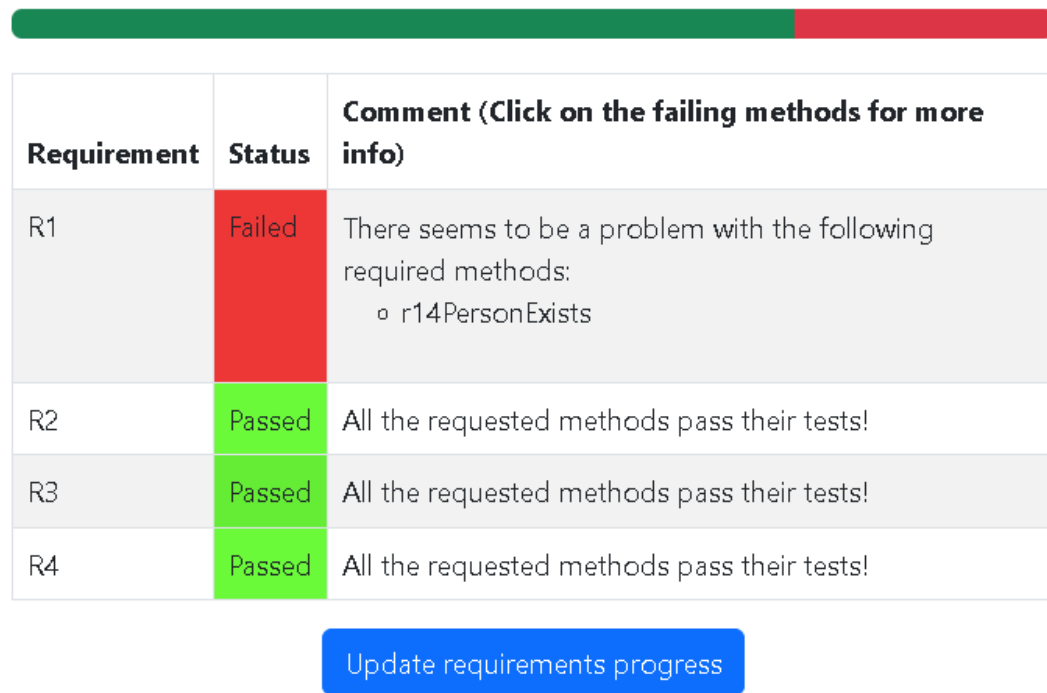


Figure 3.4: Requirements table showing a problem with the first requirement.

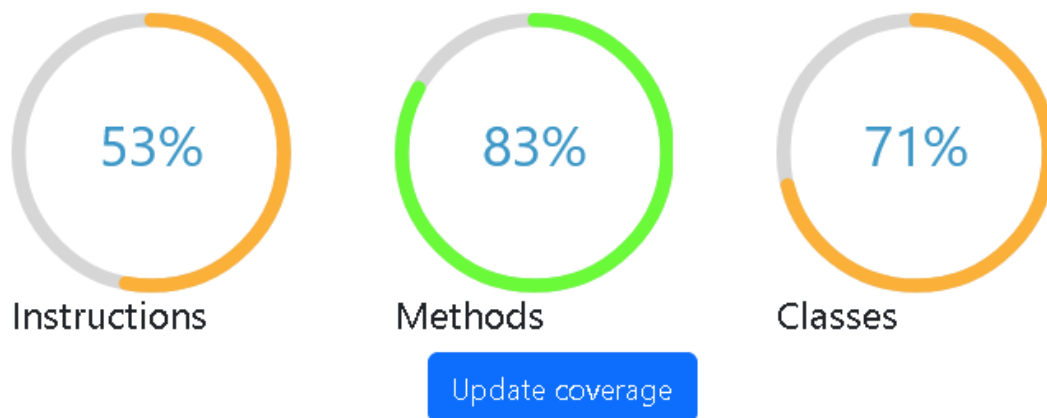


Figure 3.5: Coverage dashboard presenting, from left to right, instruction coverage percentage, method coverage percentage, and class coverage percentage.

Similarly to the requirement list, the coverage progress bars also exploit the Accomplishment core drive: they are produced by extracting and distilling data from the JaCoCo report produced when running the student's tests, allowing the student to have a clear idea of the level of coverage obtained by their tests. Students will see three coverage progress bars, showing the coverage percentage obtained for instructions, methods, and classes respectively. The idea is to provide students with a reasonable subset of information that will allow them to assess the quality of their tests, and intuitively present such information. A first iteration only showed the coverage percentage for instructions, but while such data might be the most significant, it could also be misleading, in case for example of missed classes or methods that have a small number of instructions, and that therefore would still grant a high instruction coverage percentage even if completely overlooked. Therefore the combination of the three percentages was considered to provide a more holistic view of the overall coverage.

3.3.2 Scarcity of Time

A deadline is needed to precisely constrain the students and give everyone the same time to complete the assignment. In addition to this, deadlines can be used in relation to the Scarcity core drive from the Octalysis framework as a black-hat mechanic that stimulates players to action. It is also used to trigger the free-for-all process.

3.3.3 Avatars and Customization

Students receive virtual money upon submitting assignments and participating in the free-for-all process. They can use such currency to purchase customization items for their avatar, exploiting core drives such as Social Influence & Relatedness, letting best-performing students "brag" about their achievements by sporting rare and expensive items, but also Ownership & Possession, driving them to try and obtain all possible virtual goods.

3.3.4 Free-for-all, Scores and Leaderboards

The second phase of the process, which starts when the deadline expires, implements a competition between students based on the quality of their submissions and tests. By nature of competition, this mechanic hinges on the Accomplishment core drive: students are scored and ranked based on their work, receiving rewards such as virtual money (to purchase customization items for their avatar, see later) and badges for their efforts.

The choice to provide two different forms of leaderboards, a global one and another restricted to a given session, is based on the following: only displaying a single, global leaderboard that encompasses all the laboratory sessions could end up demoralizing the students who are in the lower rankings, which could give up the laboratory sessions, while only producing single-session leaderboards would not allow students to learn from their mistakes and improve themselves. Including both leaderboards therefore requires some expedients to mitigate the discouragement for lower-ranked players: leaderboards have been designed to show only the top three ranked players and the ones ranked immediately above and below the current user. This narrows the match only to the immediate competitors, as well as showing the ones that are topping the charts. The idea behind this is to have students think that they have a concrete possibility to overcome their competitors, even if they happen to have a low overall score.

Mechanic	Core Drive	Left/ Right Brain	Purpose
Requirements Table and Coverage Dashboards	Development & Accomplishment	Left	Quantify progress and provide a sense of progression to players
Real-time feedback	Empowerment of Creativity & Feedback	Right	Provide players with feedback about their results
Leaderboards	Development & Accomplishment	Left	Implement competition among students
Scores	Development & Accomplishment	Left	Assign numerical values to results
Virtual currency	Ownership & Possession	Left	Purchasing customization items
Avatar	Ownership and Possession	Left	Self-expression and bragging rights
Scarcity of Time	Scarcity & Impatience	Left	Stimulate players to action
Developing assignment solutions and Unit tests	Empowerment of Creativity & Feedback	Right	Letting the player control the outcome

Table 3.1: Gamification mechanics

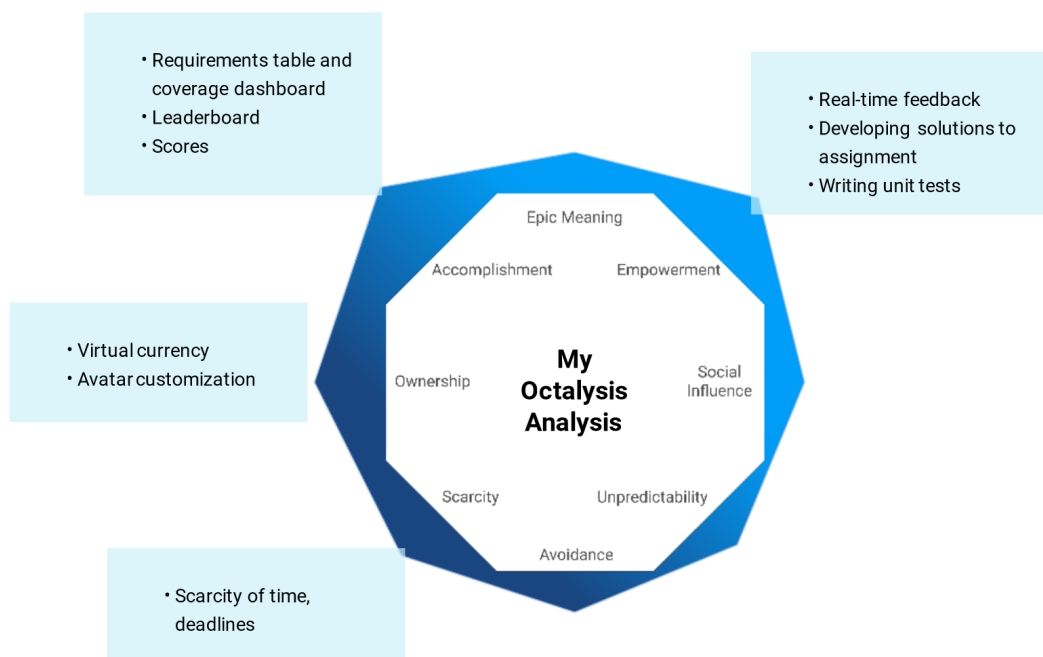


Figure 3.6: Octalysis analysis of the gamification features

Chapter 4

Implementation

The goal of this chapter is to analyze in detail the architecture of the application and its implementation, also illustrating how it has been deployed on Docker. The core of the application is constituted by four elements interacting with each other:

- the **server**, acting as the middle point connecting everything
- the **player front-end**, accessible by the students, from where they can interact with the application
- the **admin front-end**, accessible by the course faculty only, used to set up and monitor the laboratory sessions
- the **GitLab Continuous Integration tool**, which interacts with the server to produce intermediate reports for the students

The structure of the application is represented in Figure 4.1.

4.1 Server

The back-end of the application has been implemented, building on the existing infrastructure, using the Node.js runtime environment. Node.js is an open-source, JavaScript runtime environment commonly used to build web servers and real-time applications, providing an event-driven, non-blocking I/O model. A peculiarity of Node.js that makes it extremely pliant and versatile is the vast ecosystem of packages and libraries available on it using its packet manager, called *npm*. Some of these libraries have been used to implement the backbone of the system, starting from the Express library. Such a library provides a straightforward, minimal way to implement routes, middlewares, and handlers for HTTP requests, making it extremely suitable for this application. Notably, its support for middleware allowed

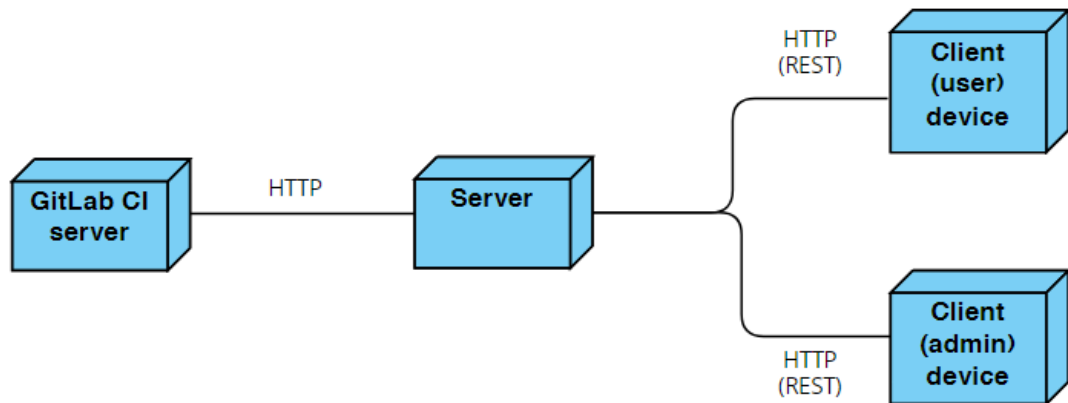


Figure 4.1: UML diagram representing the core application structure

the use of libraries to handle critical parts of the application, such as the Passport library to handle user authentication.

4.1.1 Database

Another important component of the back-end of the application consists of the database, used to store persistent information and data such as the registered users and their credentials, scores, avatar items, and so on, but also data about the laboratory sessions, links to the repositories, etc. The framework chosen to implement the database is *sqlite3*, a Node.js library that provides an interface allowing to manage a SQL database in JavaScript, performing CRUD (Create, Read, Update, Delete) operations on entries to manage the application data.

4.2 Front-ends

As previously mentioned, two different front-ends have been developed, one for the players (i.e. the students) and one for the admin. The framework used to develop both front-ends is React, an open-source JavaScript library used to build user interfaces in a declarative way. It has been chosen for its lightweight nature and the way it easily integrates with different libraries and frameworks, providing freedom in the development process.

The two front-ends have different purposes: the player front-end allows students to sign up and log in to the application. Once logged in, they can monitor the laboratory sessions and access the application's main features, consisting of the monitoring of their progress during the sessions as well as browsing the results. In

addition to this, it is also possible to view and customize the player's avatar.

The admin front-end, conversely, is destined to be accessed by faculty only. It allows them to set up laboratory sessions by providing the needed information, stop and delete ongoing sessions browse the results, as well as generating and downloading reports (in the .xls format) on the laboratory activity.

4.2.1 Player front-end

When signing up, the student is prompted to input some required information, such as their username, full name, and password. Once the form has been submitted, they can log in with the specified credentials.

Note on usernames: Since the back-end needs to work with the student's submitted solution, and they have the option to name the folders as they like, students have been asked to register using their student ID as their username, in the format sXXXXXX, X being a digit, and to use the same ID as the name for the test folder in their submissions. Test classes can be arranged freely inside such folders. Using this convention, the back-end can look for the correct folder when running the students' tests or the ideal tests, as well as assemble a comprehensive test battery during the free-for-all while also keeping track of the author of each test, to know whom to assign the points.

Once logged in, the students can navigate various sections of the application using a Navigation bar, provided with a series of tabs.



Figure 4.2: Navigation bar

Labs tab

In the *Labs* tab, students can browse the various existing laboratory sessions, and join the active one if they haven't already. By clicking on the *join* button, they will be asked to input the link to the GitLab repository they have been provided for the session, which will be stored and used to clone their solution during the various checks and the free-for-all.

On the active lab view, the student can browse the laboratory requirements, check the deadline, and trigger the progress check. On the *Progress* tab, clicking on the *Check progress* button triggers the check. If the check fails, a card with an error message is displayed, along with a button to retry the check. A different error message is displayed if the check fails because of a compilation error.

Unit Brawl

- Diet
- Social Network

Social Network

Deadline: 25-08-2023

Requirements Progress

Would you like to check your progress in the lab?

[Check progress](#)

Figure 4.3: Initial view of the Progress tab

During the check, the ideal tests are cloned in the student's submission and executed on it. The templates given to the students are set up with the Maven Surefire plugin, and executing the tests using Maven (i.e. using the command `mvn clean test -Dtest="PathToTests"`) generates a test report, both as a .txt file and a .xml file. The XML file is parsed to aggregate the test results in a suitable data structure, which is then used in the front-end to build the following requirement table:

Tests report summary

Requirement	Status	Comment (Click on the failing methods for more info)
R1	Passed	All the requested methods pass their tests!
R2	Passed	All the requested methods pass their tests!
R3	Passed	All the requested methods pass their tests!
R4	Passed	All the requested methods pass their tests!

[Update requirements progress](#)

Figure 4.4: Requirement table

To correctly parse the test results, the ideal tests are named using the format `rX_testName`, where X is the number of the requirement. This way it is possible to parse the report, checking for each requirement if all the compulsory methods

have passed their tests. If any test for a requirement fails or produces an error, the error message returned is displayed to the student upon clicking on the defective method, as in the following image:

Tests report summary

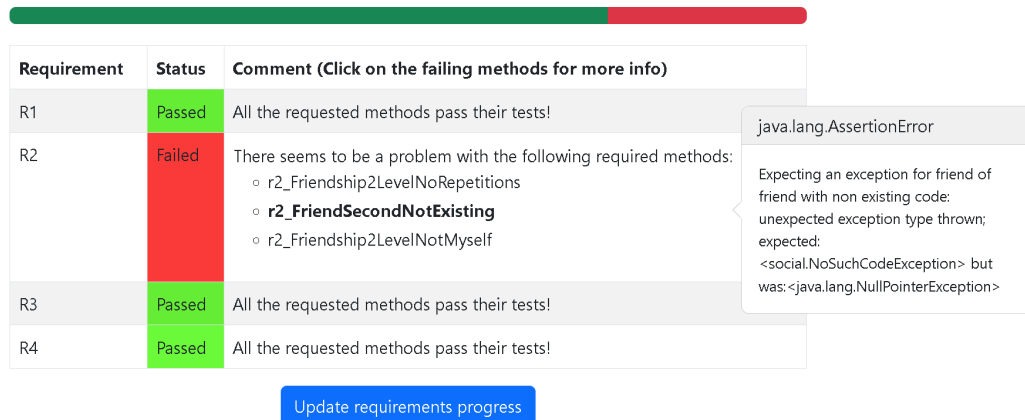


Figure 4.5: Requirement table with errors

After a successful check of the progress, the student can also require a coverage check by pressing the Check coverage button. Such a button is displayed only after a successful progress check, i.e. a check in which there are no server or compilation errors. Faulty or missing required methods do not prevent the student from checking the coverage, although they are encouraged to check the coverage only after submitting a suitable number of tests to get significant information. During a coverage check, the student's own tests are executed on their submission. The template provided to the students is set up with the JaCoCo plugin, which produces coverage reports for Java tests. The report is generated only if there are tests to be executed (i.e. if the student has written them, placed them in the correct folder, and pushed them to the remote repository) and if they succeed. If any of those conditions are not met, a message informs the student, also reminding them of the correct folder location and name where to put the tests. If the report is correctly generated, it is parsed to assemble a dashboard for the student, displaying the coverage percentage for the instructions, methods, and classes in the form of circular progress bars. A message is shown if the number of tests submitted by the student exceeds the maximum threshold set for the laboratory session.

If the laboratory session is expired, the student can access the My Results tab, where they can check the leaderboard for the session.

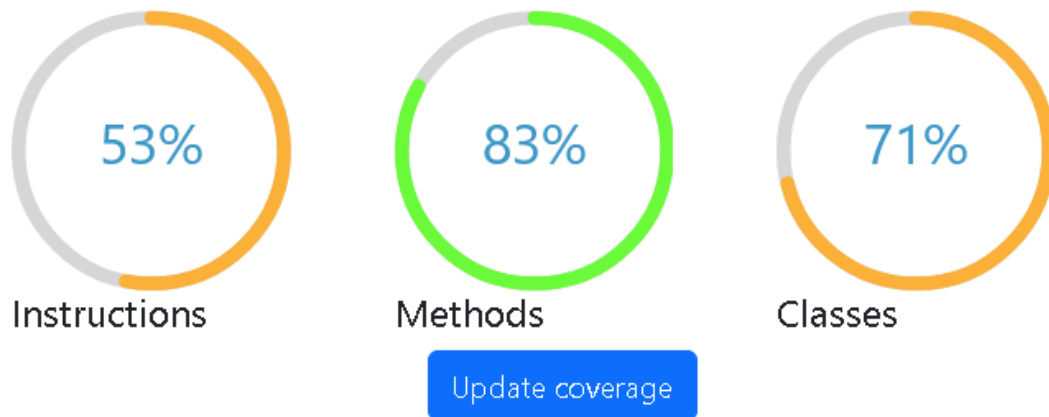


Figure 4.6: Coverage dashboard

0.4






#	Player	Points
1	 s236507	15.8
2	 s292488	14.9
3	 s000000	14.0
4	 s111111	13.1
5	 s222222	11.9

Figure 4.7: Global leaderboard displaying the complete ranking, with the top user highlighted in gold and the current user in green

Leaderboard tab

The Leaderboard tab contains two distinct leaderboards. The first one is a *local* leaderboard, which only displays the student's position and the opponents immediately above and below them, or the second and third-ranked if the student ranked first. The idea behind this is to employ *urgent optimism*, a powerful dynamic where the player feels as if they have the actual and concrete opportunity to obtain immediate results: in this instance by reaching and surpassing their immediate competitors, while fending off pursuers. Showing only the complete leaderboard can be discouraging for players who have struggled in a given session, possibly causing them to abandon the laboratory sessions altogether. Nevertheless, a complete leaderboard is a valuable piece of information and is therefore included in the

Leaderboard tab as well.

0.4




#	Player	Points
1	 s236507	15.8
2	 s292488	14.9
3	 s000000	14.0

Figure 4.8: Local leaderboard displaying only the user and the opponents immediately above and below them

Shop tab

The Shop tab allows the user to browse available avatar items and purchase them using the money earned from the laboratory sessions. There are items with prices ranging from free to quite expensive, to offer both customization options even at early stages and the “bragging rights” associated with boasting a rare and valuable item earned through good results in the sessions.

Profile tab

The Profile section lets the user browse their profile, check their information, browse their results for the sessions they took part in, select their avatar from the ones they own, and check the achievement list.

4.2.2 Admin front-end

Faculty can access their reserved front-end to set up and manage laboratory sessions, and download reports regarding the activities on the application.

After logging in, the administrator can view the existing laboratory sessions or create a new one in the *Labs* tab. When creating a session the admin is prompted to input the needed information, such as the title, deadline, requirements, link to the ideal solution, and the maximum number of tests. The link to the ideal solution is immediately tested upon submitting the session form by attempting to clone the ideal solution and returning an error message should the operation fail. Nevertheless, the link is editable at a later time as well.

If there is an active session, the admin can view the number of participants (i.e. the number of students who have joined it), stop it manually, or delete it. If there

is no active session, the admin can start the free-for-all process for any sessions in which it has not been performed already, and browse the results.

The Leaderboard and Profile tabs in the navigation bar are the same as the player front-end.

The Report tab allows the admin to download reports, in .xls format, about various aspects of the laboratory activity, organized into four types of report:

- the *general* report contains the aggregate numbers of laboratory sessions and players and the average number of participants and points per lab
- the *users* report contains, for each registered user, the credentials, the number and percentage of sessions joined, and the average and best score
- the *labs* report contains, for each session, the number of participants, the percentage of participants in relation to the total number of users, and the average score
- the *userLabs* report offers a more detailed analysis of the performance of each student in each session they took part in, including the score, number of submitted tests, how many of their tests failed on an opponent's submission, and how many tests by an opponent passed on theirs.

4.3 Dockerization

To deploy it and make it accessible to the students from the Politecnico di Torino website, Docker was used as the tool to encapsulate the application environment, including its dependencies and configuration. This approach allows to streamline the process of integrating the tool on the institution's website, eliminating potential compatibility issues.

I approached the task of Dockerizing the application first by creating a Docker container for each part of the core application itself, i.e. the server and the two front-ends. The server Docker image was created using the latest Alpine Linux distribution available at the time, and then installing the needed libraries for the server. Crucially, the server requires the installation of Node and npm to run the Node Express server, Git to clone the students' and ideal solutions, and Maven to run the tests. In addition to this, the content of the package.json folder was copied into the container and the command `npm install` was run to install all the needed dependencies. Lastly, port 3001 was exposed to be able to send API requests to the server. The content of the Dockerfile for the server is as follows:

```
1 FROM alpine:latest
2 RUN apk update
3 RUN apk add git
```

```
4 RUN apk add --update nodejs npm
5 RUN apk add maven
6 WORKDIR /server
7 COPY package*.json ./
8 RUN npm install
9 COPY . .
10 EXPOSE 3001
11 CMD ["node", "index.js"]
```

Listing 4.1: Server Dockerfile

For both front-ends the starting point was a Node base image, then the content of the respective package.json folders was copied into the containers, and the npm install command was run, as follows:

```
1 FROM node:latest
2 WORKDIR /admin_frontend
3 COPY package*.json ./
4 RUN npm install
5 COPY . .
6 EXPOSE 3000
7 CMD ["npm", "start"]
```

Listing 4.2: Admin front-end Dockerfile

```
1 FROM node:latest
2 WORKDIR /player_frontend
3 COPY package*.json ./
4 RUN npm install
5 COPY . .
6 EXPOSE 3000
7 CMD ["npm", "start"]
```

Listing 4.3: Player front-end Dockerfile

As reported in the two Dockerfiles, port 3000 was exposed on both containers to have them communicate with the server. At this point, the Docker stack was organized as shown in Figure 4.9.

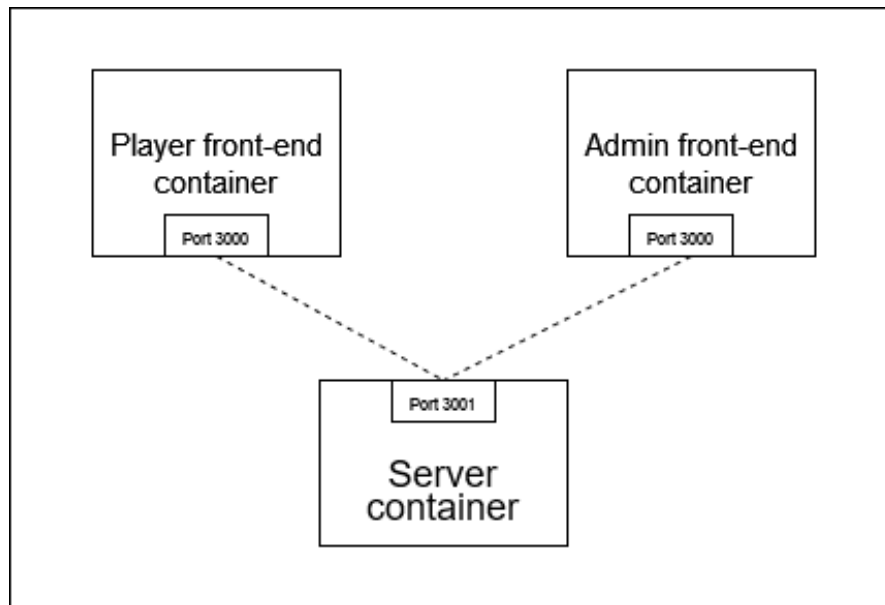


Figure 4.9: Initial Docker stack structure for the core containers of the application

From here, Nginx has been used as a proxy to comply with Politecnico’s firewall policy. This was necessary because, while the communication between the front-ends and the server happens internally within the Docker stack and would work correctly with the exposed ports specified in the Dockerfiles, requests coming from the outside (e.g. from a student’s computer) would be blocked by the University’s firewall, which prevents connections to such ports.

To address this, a proxy has been set up as a single access point for users, granting access to the services provided by the Docker stack by exposing a port that is accepted by the firewall and redirecting requests to the correct container depending on the URL. Nginx is the tool used for this purpose. It is a simple, open-source server, reverse proxy server, and load balancer, which has proved suitable for the task at hand given its capability to handle a large number of simultaneous connections.

The overall Docker structure is depicted in Figure 4.10.

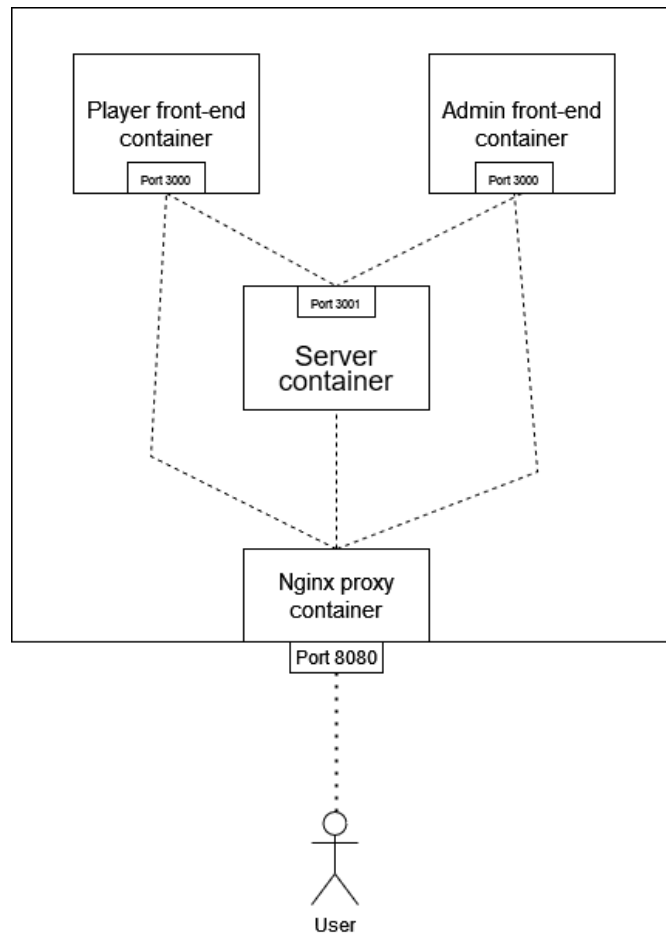


Figure 4.10: Complete Docker stack structure, including the Nginx proxy container

The resulting Docker stack has then been deployed on the Politecnico server, granting students access to the platform.

4.3.1 GitLab Continuous Integration

In addition to the functionalities offered by Unit Brawl, students can also access the full JUnit reports in XML format on GitLab. This is because the templates they're provided with are set up with GitLab's Continuous Integration (CI) tool, a part of the GitLab platform that allows developers to automate the testing and integration of software changes on a project. By placing a `.gitlab-ci.yml` file in the root folder of the project, we can specify a script to be executed every time a change is pushed to the repository. In our case, we can automate the execution of the JUnit tests present in the student's pushed solution and collect the resulting

XML files containing the test report. This way, should a student want to access the full original reports for their files they can, while Unit Brawl offers a more concise and gamified representation of the same information.

The content of the `.gitlab-ci.yml` is as follows:

```
1 variables:
2   MAVEN_OPTS: >-
3     -Dhttps.protocols=TLSv1.2
4     -Dorg.slf4j.simpleLogger.showDateTime=true
5
6   MAVEN_CLI_OPTS: >-
7     --batch-mode
8     --fail-at-end
9     --show-version
10
11 verify:
12   stage: test
13   tags:
14     - oop
15   script:
16     - 'DISPLAY=:1 mvn $MAVEN_CLI_OPTS test'
17   artifacts:
18     when: always
19     reports:
20       junit:
21         - target/surefire-reports/TEST-*.xml
22
23
```

Listing 4.4: Script in the `.gitlab-ci.yml` file

This script specifies options to be used when executing the tests using Maven. Specifically, the following options are present:

- `-batch-mode` specifies to run Maven in *non-interactive mode*, avoiding asking for manual intervention or user interaction. This mode is usually used in CI tools, where user interaction is not feasible and the goal is to have processes run automatically
- `-fail-at-end` specifies that the execution should continue even in case of failures, which should be reported at the end. This way all tests will be executed even in case of test failures, which will be included in the final report, rather than causing the execution to stop
- `-show-version` displays the Maven version being used

The `verify` job contains a single stage, named `test`, that is set to execute the content of the `script` section, which is `DISPLAY=:1 mvn $MAVEN_CLI_OPTS test`.

The meaning of such a script is to run the command `mvn test` using the previously defined options, to produce the test reports. The final section, `artifacts`, specifies which files should be saved at the end of the process. Here we specify that the files to be saved are `.xml` files whose name starts with `TEST-` located in the folder `target/surefire-reports`. We also specify the files should be collected regardless of the job's success or failure, using the `when: always` option.

Chapter 5

Results

This chapter aims to analyze the results of the gamification experiment, as well as discuss the feedback gathered from students.

5.1 Experiment organization

The experiment involved the last three laboratory sessions for the Object-Oriented Programming course for the Bachelor's Degree in Computer Engineering at Politecnico di Torino for the 2022-2023 academic year, during May and June 2023. The selected sessions are the ones in which the students start to apply the concepts of unit testing firsthand, after being introduced to them in class. The sessions will be referred to in the following as *Session 1*, *Session 2*, and *Session 3* respectively.

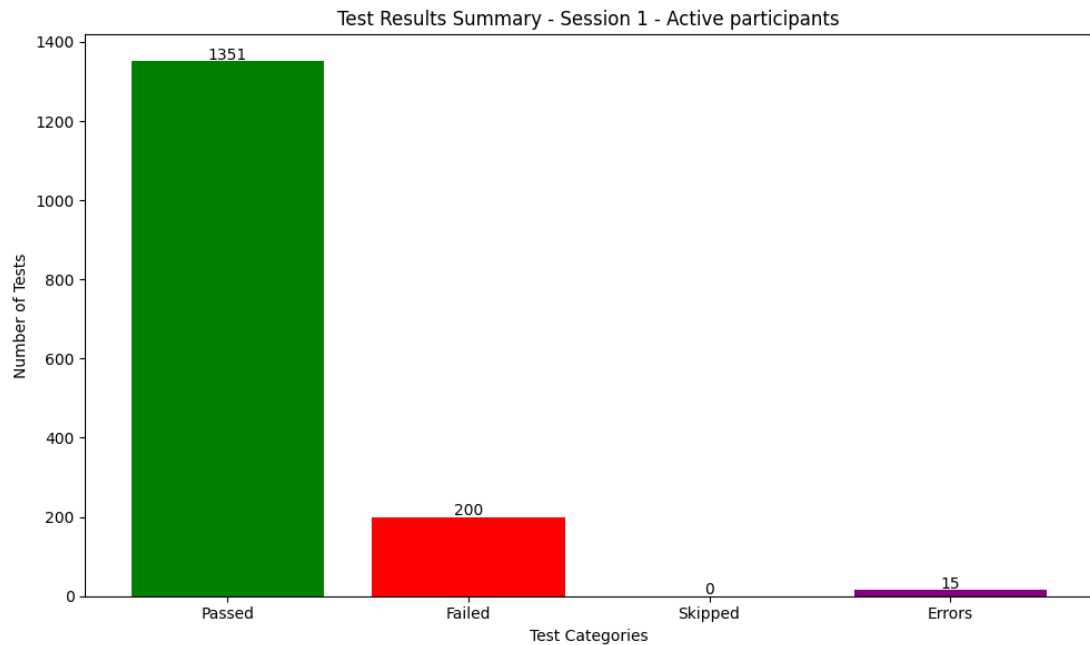
5.1.1 Session 1: Control group (No platform)

The first session was utilized as a control group to gather preliminary data regarding the students' normal behavior and approach to the laboratory activities, without the use of the platform. For this session, students were not given any specific indication other than to submit an implementation that satisfied the requirements of the assignment and that included unit tests to verify its correctness. To submit their implementation, students simply had to push it to the remote GitLab repository they were provided with. There were 112 active participants in this session out of the 213 students officially enrolled in the course. To be considered active participants, students had to have pushed at least one commit to the remote repository they were assigned. Of these 112 participants, 59 submitted a solution that managed to pass the acceptance tests provided by teachers, while the remaining 53 provided a submission that failed at least one acceptance test. Data from session 1 are presented in Table 5.1:

Table 5.1: Participation and test quality data for Session 1

		% on enrolled students	% on active participants
Enrolled students	213	-	-
Active participants	112	52.58%	-
Submissions passing acceptance tests	59	27.69%	52.67%
Submissions failing acceptance tests	53	24.88%	47.32%

The bar chart in Figure 5.1 presents the test result distribution considering only the active participants in Session 1:

**Figure 5.1:** Test result distribution in Session 1

5.1.2 Session 2: Platform Version 1

For the second session, students were provided with an initial version of the platform, which included the following features:

- Students could sign up and log in to the application.

- They could check the requirements for the session.
- They could monitor their progress throughout the session by checking which of the requirements were satisfied and which were not, while also getting information about the type of error, if any.
- They could assess the coverage of their tests in terms of instructions covered, methods covered, and classes covered, to gauge the quality of their tests.

This session recorded an overall decrease in student participation: as reported in Table 5.2, the number of active participants for this session was 100. However, the data showed an increase in the percentage of students' submissions that passed the acceptance tests relative to the total number of active participants: 55 students provided a solution that passed the acceptance tests, while 45 submissions failed the acceptance tests.

Table 5.2: Participation and test quality data for Session 2

		% on enrolled students	% on active participants
Enrolled students	213	-	-
Active participants	100	46.95%	-
Submissions passing acceptance tests	55	25.82%	55.00%
Submissions failing acceptance tests	45	21.13%	45.00%

Figure 5.2 presents the test result distribution in Session 2:

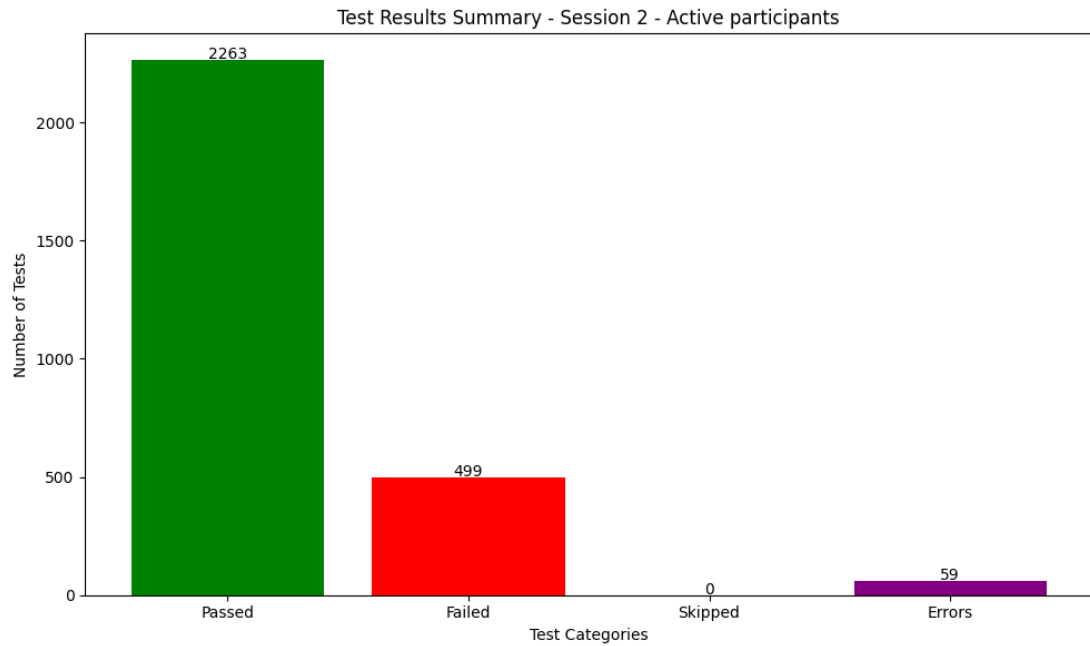


Figure 5.2: Test result distribution in Session 2

The test result distribution is aligned with the distribution in the Session 1.

5.1.3 Session 3: Platform Version 2

The version of the platform that was deployed for the last session included the free-for-all feature in addition to the previous ones. For this session, the recorded data presented 83 active participants out of the 213 officially enrolled students, with 24 submissions successfully passing the acceptance tests and 59 submissions failing the acceptance tests. The data are presented in Table 5.3:

Table 5.3: Participation and quality data for the Session 3

		% on enrolled students	% on active participants
Enrolled students	213	-	-
Active participants	83	38.96%	-
Submissions passing acceptance tests	24	11.26%	28.91%
Submissions failing acceptance tests	59	27.69%	71.08%

The test results distribution for Session 3 is presented in Figure 5.3:

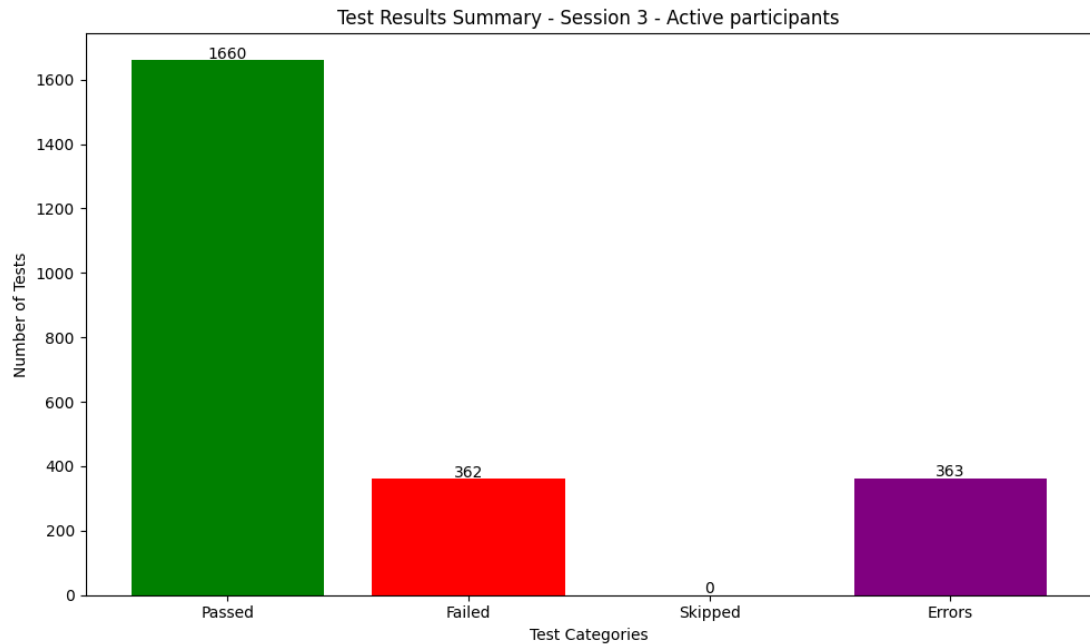


Figure 5.3: Test result distribution in Session 3

5.2 Results and students' feedback analysis

The experiment's results have highlighted areas for improvement. A first aspect is laboratory attendance: taking part in the laboratory sessions is of course highly beneficial for the students' learning since it allows them to experiment with the concepts they have been taught in a firsthand fashion, without being guided by the teacher, and therefore gaining a much clearer understanding of the notions they're presented. Moreover, they are bound to run into some issues and errors in their implementation, which can prove a valuable source of knowledge and insight. Nevertheless, the non-mandatory nature of the laboratories entails a percentage of students not taking part in them. This is even further enhanced by the progressively increasing difficulty of the laboratories throughout the course, which may discourage students who have been struggling with earlier ones. Additionally, the deadlines that are placed on each session might also cause some students' participation not to be recorded, if they happen to be catching up with the laboratories after they have expired.

However, the recorded data can also support a more positive analysis: while the percentage of active participants during the sessions has gone down throughout

the experiment, laboratory session 5 has shown a slight increase in the percentage of students who have managed to produce a solution that passed all the acceptance tests, in comparison with the control group session. This can indicate the effectiveness of the requirement tracking feature of the platform: being able to see clearly when a requirement has been completed can act as a powerful motivator for students, who can see a visual representation of their progress. Moreover, being able to see which of the required methods present issues, as well as the possibility to read the error messages for each one, can help students pinpoint the problem in their code, especially in the case of submissions that compile correctly but fail the tests. Having an initial indication of "where to look" can ease the burden of having to go over the code again to find the problem, and therefore increment the number of students who do not give up in case of failed tests.

The participation percentage dropped significantly from session 5 to session 6. This can be dependent on several factors: from the increased difficulty of the session to its proximity to the exam session, which might cause some students to set the sessions aside and return to them later on, possibly after taking other exams. In this context, assigning points as a participation bonus of sorts would prove a powerful incentive for students to take part in all the sessions. Additionally, a further bonus could be assigned to the students who have performed best across the various sessions and have topped the leaderboards, to stimulate students to produce quality submissions and tests.

Another way to increase students' participation would be the implementation of additional gamification mechanics: from customizable avatars, to allow students to personalize their own and express themselves, to a list of achievements to obtain during the course, such as participating in the first session, all of them, as well as obtaining a given coverage percentage in one or more sessions, and more, which would prove a good way to provide students with long-term goals and objectives that would keep them going.

Chapter 6

Conclusions

This thesis work revolves around the design, implementation, and deployment of a gamification platform whose main purpose is to promote the learning of concepts of unit testing for Java applications to students enrolled in the Object-Oriented Programming course for the Bachelor Degree in Computer Engineering in Politecnico di Torino. The platform is designed to support the laboratory sessions carried out throughout the course, in which students can apply the concepts learned in class.

The platform leverages gamification mechanics to promote students' engagement. The main features of the application are two: the first one is a tracking feature, that allows users to check whether the various requirements defined in the session's assignment are completed, and, if not, what type of error has risen. It is also possible for users to check the coverage percentage of their tests, in terms of instructions covered, methods covered, and classes covered, through a dashboard.

The second main feature of the platform is a form of competition based on the students' submissions and tests. When the deadline for a given session expires, submissions are filtered according to specific criteria, and the ones that fit all of them take part in the competition, in which the tests submitted by all students are executed on each other's submissions, assigning points accordingly. The resulting scores are used to build a session-specific leaderboard, rewarding students with virtual currency based on their position, and a global leaderboard, spanning all the various sessions, implementing a championship of sorts among students.

The overall goal of promoting unit testing stems from the fact that this activity is often overlooked and disregarded. By gamifying it, this work intends to improve both the quantity and quality of unit tests produced by students.

6.1 Possible future developments

Upon analysis of the platform and its usage by students, some areas for improvement have emerged.

A first aspect worth considering is an extension of the requirement tracking feature, especially on the messages being displayed when a requirement is not completed. Currently, the user is presented with the same error messages they would get on an IDE when running the tests. While being able to see exactly which errors refer to which requirement is indubitably useful, it would be beneficial to provide the students with more specific information: in case of successful compilation but unsatisfied requirement, for instance, a message with some information about the type of error encountered, with an explanation of what typically causes such an error, could be beneficial to help the students understand better the nature of their mistakes.

Another possible useful change would be allowing students to use the tracking features after the expiration of the deadline for a session. It is not uncommon for students to delay tackling a laboratory session in favor of more pressing matters, such as an imminent exam, to come back to them later on. Preventing them from utilizing what is effectively one of the core features of the application after the deadline would cut off these students altogether while allowing them to make use of the platform anyway would let them benefit from the gamification approach to their learning, even if it deferred. Of course, this cannot be done for the free-for-all aspects, for reasons of fairness of competition, but since the tracking feature provides a purely individual service, it could be reasonable to extend its availability past the deadline.

Other future developments can be directed to features that are only in their early stages of development, such as the achievement features. These achievements are objectives that students can complete on a medium-to-long term, and include goals such as taking part in their first session, or all of the sessions for the course, ranking in the top 10 for a session, clearing all requirements for a session, and so on. This mechanic can be motivating for students and can be integrated with the competition aspect, by assigning points based on the achievements of students. The avatar customization feature can be expanded as well, possibly also broadening the customization to other areas, such as a user profile of sorts for students.

Lastly, the next possible step may be allowing students to use the application for more sessions, possibly all of the sessions in the course. This would allow for gathering more data about the application's usage by students, but it would require planning a restriction of the features initially accessible by users: since they are presented with the concept of unit testing only after a few weeks in the course, initially restricting the accessible features to only the requirement tracking would be reasonable. This would allow them to still be guided throughout the first

assignments, without exposing them to concepts that have not been introduced in class, which may cause confusion among the students and undermine the purpose of the application itself. Then, once the needed concepts have been presented, additional features such as the coverage tracker and the free-for-all can be unlocked, also giving the students a sense of progression.

Bibliography

- [1] SolarWinds Worldwide, LLC. 2009. URL: <https://www.pingdom.com/blog/10-historical-software-bugs-with-extreme-consequences/> (visited on 10/05/2023).
- [2] Agile Alliance. 2023. URL: <https://www.agilealliance.org/glossary/tdd> (visited on 07/24/2023).
- [3] Sebastian Deterding et al. «From Game Design Elements to Gamefulness: Defining Gamification». In: vol. 11. Sept. 2011, pp. 9–15. DOI: 10.1145/2181037.2181040.
- [4] Alexander S. Gillis. *What is acceptance testing?* 2021. URL: <https://www.techtarget.com/searchsoftwarequality/definition/acceptance-test> (visited on 07/24/2023).
- [5] «IEEE Standard Glossary of Software Engineering Terminology». In: *IEEE Std 610.12-1990* (1990), pp. 1–84. DOI: 10.1109/IEEESTD.1990.101064.
- [6] D. Janzen and H. Saiedian. «Test-driven development concepts, taxonomy, and future direction». In: *Computer* 38.9 (2005), pp. 43–50. DOI: 10.1109/MC.2005.314.
- [7] Yu kai Chou. *Actionable Gamification: Beyond Points, Badges and Leaderboards*.
- [8] Vincent Massol and Ted Husted. *JUnit in Action*. USA: Manning Publications Co., 2003. ISBN: 1930110995.
- [9] Deepak Parmar. *Exploratory testing*. 2023. URL: <https://www.atlassian.com/continuous-delivery/software-testing/exploratory-testing> (visited on 07/24/2023).
- [10] pp_pankaj. *Software Testing Life Cycle (STLC)*. 2023. URL: <https://www.geeksforgeeks.org/software-testing-life-cycle-stlc/> (visited on 07/24/2023).
- [11] Karen Robson et al. «Is it all a game? Understanding the principles of gamification». In: *Business Horizons* (Apr. 2015). DOI: 10.1016/j.bushor.2015.03.006.

BIBLIOGRAPHY

- [12] Wikipedia contributors. *Stress testing (software)* — *Wikipedia, The Free Encyclopedia*. [Online; accessed 24-July-2023]. 2023. URL: [https://en.wikipedia.org/w/index.php?title=Stress_testing_\(software\)&oldid=1143328664](https://en.wikipedia.org/w/index.php?title=Stress_testing_(software)&oldid=1143328664).