FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Serious Game to Teach Software Testing

José Nuno Barbosa Quintas

Mestrado em Engenharia de Software

Supervisor: Prof. Ana Cristina Ramada Paiva

Second Supervisor: Prof. Auri Marcelo Rizzo Vincenzi

July 30, 2026

Serious Game to Teach Software Testing

José Nuno Barbosa Quintas

Mestrado em Engenharia de Software

July 30, 2026

Resumo

O teste de mutação desempenha um papel crucial na determinação da eficácia de um conjunto de testes, tendo uma presença importante na área de testes de software. A sua prática proporciona a capacidade de avaliar o conjunto de testes, detetando testes ineficazes ou código não testado, o que contribui para a melhoria da qualidade e confiabilidade do software. No entanto, esta técnica permanece subutilizada durante testes de software, possivelmente devido ao tempo e à sobrecarga computacional que exige ou devido à falta de motivação de quem a pratica.

Outro fator que pode agravar isso é a educação insuficiente na área de teste de mutação, exacerbada pela sua maior complexidade e natureza menos tangível em comparação com outras técnicas de teste de software. Para superar estes problemas, professores e educadores frequentemente introduzem abordagens interativas ou práticas no processo de aprendizagem. Já existem diversas ferramentas e estruturas que auxiliam ou facilitam o ensino de testes de mutação, mas estas ainda podem carecer de mecanismos envolventes que possam motivar os alunos.

Outra abordagem que pode melhorar o processo de ensino de testes de mutação e, simultaneamente, manter os alunos motivados a aprender é o uso de gamificação e jogos sérios. Com esta abordagem, pretendemos expandir e aprimorar o jogo sério ***GAMFLEW***, que mantém um foco principal em objetivos de cobertura de código, com conteúdo educacional sobre testes de mutação.

Palavras-chave: Teste de Mutação, Testes de Software, Gamificação, Jogos Sérios, Educação

Abstract

Mutation Testing plays a crucial role in determining the effectiveness of a test suite, having an important presence in the software testing area. It provides the ability to evaluate the test suite, detecting ineffective tests or untested code, which contributes to the improved quality and reliability of software. However, this technique remains underused, possibly because of the perceived time and computational overhead it requires or due to a lack of motivation.

Another factor that can exacerbate this may be the insufficient education on mutation testing, aggravated by its increased complexity and less tangible nature when compared to other techniques of software testing. In order to overcome these problems, teachers and educators often introduce interactive or hands-on approaches to the learning process. There are already several tools and frameworks that assist or facilitate the teaching of mutation testing, but these may still lack engaging mechanics that can motivate students.

Another approach that can improve the teaching process of mutation testing and simultaneously keep students motivated is the use of gamification and serious games. With this approach, we aim to expand and improve the existing serious game ***GAMFLEW***, mainly focused on code coverage goals, with educational content regarding mutation testing.

Keywords: Mutation Testing, Software Testing, Gamification, Serious Games, Education

Acknowledgements

This work wouldn't have been possible without the help and company of all those who stood beside me, whom I shall thank by name.

I would like to thank the ***Lesados***, my friends from FEUP, including but not limited to Pedro, Leo, André, Letícia, Constança, and Ana, with whom I have shared the majority of this course. Thank you for being here for this long journey, for the help you've always offered, and for the long nights we have shared. None of this would have been possible without you!

I would like to thank the ***Guys***, my beloved hometown friends, for all the love and support they have given throughout this process, for the lessons, laughs, and tomfoolery we have shared, and for remaining as a central source of my motivation all these years. Love you, Guys!

I would like to thank the ***International Brigade***, my dearest friends from various corners of the world, including but not limited to Johnny, Patrick, Kiera, Pasta, Kkay, Roni, and James. Thank you for the support you've always given and the moments we have spent together. You're the proof that friendship can never be stopped by distance alone!

I would like to thank my family, including my mom, dad, and young brother, for their support throughout my years in this course. I would also like to thank my two grandmothers for their kindness and love as I completed this project.

I would like to thank Professors Ana Paiva and Auri Vincenzi for their advice and supervision throughout these semesters, whose guidance was imperative for the conclusion of this project. Last, but not least, I would like to thank all the participants for contributing to this work with their participation.

To everyone who helped me with this work, I can't thank you enough for everything you've done. Thank you so much!

José Nuno Barbosa Quintas

"Those who can imagine anything, can create the impossible"

Alan Turing

Contents

List of Figures

List of Tables

List of Acronyms

ST	Software Testing
MT	Mutation Testing
SG	Serious Game
IDE	Integrated Development Environment
GAMFLEW	GAMe For LEarning White-box testing
FRAFOL	FRAmework FOr Learning mutation testing

Chapter 1

Introduction

This chapter will cover the context, motivation, and problem to be solved in this project. Finally, it will also present the general structure that will be followed throughout this article.

1.1 Context and Motivation

Mutation Testing is a crucial yet underused technique of the software testing process, as it can be a challenging topic to teach due to its abundance of theoretical concepts, time-consuming setup and execution, and complex challenges such as the equivalent mutant problem.

In current class settings, the teaching curriculum often focuses on theory, while simultaneously neglecting practical teaching. Traditional educational techniques may not be sufficient to effectively motivate students to learn these challenging concepts, nor to apply them on real-world examples [3]. This problem seems to indicate a need for a different approach to keep them engaged and curious about the topic.

There are already examples of tools and frameworks that facilitate the teaching process and help students learn more efficiently by providing didactic or interactive content on the areas that are being taught [13, 27]. However, these frameworks may still lack motivation or outreach in the areas addressed.

Another approach that has been tried and tested is gamification, which can be defined as the introduction of game elements such as score, leaderboards, achievements, or levels to non-game contexts. Gamification presents a clear benefit when it comes to motivating and engaging someone to perform a task they may otherwise overlook [25]. It has been applied across multiple areas where gamification elements are not usually present, including the teaching of software testing.

Many existing serious games already apply these gamification elements to make learning more entertaining and appealing, but few cover the concept of Mutation Testing [1, 7, 8, 16, 19, 24].

GAMFLEW (GAMe For LEarning White-box testing), developed by Silva et al. [21] is a serious game for teaching white-box test case design techniques, where students define the test input data needed to achieve code coverage objectives defined in challenges. The game's challenges encourage students to interact with a checkers board, defining the input data. When a student

wants to submit an attempt to complete a challenge, they are also prompted to submit a comment, which their teacher or educator can then analyze. This serious game successfully applies gamification with its use of points, leaderboard, and achievements to teach players about software testing concepts, but it doesn't cover other relevant areas, such as mutation testing. Despite its proven effectiveness in engaging students with white-box testing concepts, the game's evaluation model still has some limitations.

1.2 Objectives

To guide our work in this project, we defined the following objectives:

- To review the state of the art in tools, frameworks, and serious games that support software and mutation testing education, identifying the main limitations of existing approaches.
- To design and implement a **client-side code execution engine** for the GAMFLEW platform, capable of running code asynchronously in the browser using Web Workers, without requiring any server-side infrastructure.
- To **retrofit the existing white-box testing challenges** in GAMFLEW with genuine runtime coverage analysis, replacing the previous static Boolean condition model with real statement, decision, condition, and MC/DC coverage verification.
- To design and implement a **mutation testing educational module** for GAMFLEW, in which players identify surviving mutants by providing inputs that produce divergent outputs between the original and mutated code, leveraging the execution engine developed above.
- To evaluate the effectiveness and usability of the extended platform with real users.

1.3 Research Questions

Adding to the previous objectives, this dissertation also aims to answer the following research questions:

RQ1 What tools, frameworks, and serious games currently exist to support mutation testing education?

RQ2 What are the main limitations of existing approaches?

RQ3 How can we extend GAMFLEW with challenges to teach Mutation Testing?

RQ4 Do the proposed GAMFLEW extensions improve the learning experience (motivation and knowledge acquisition)?

1.4 Document Structure

This dissertation is structured as follows:

1. Chapter 2 presents the systematic literature review, covering existing teaching methodologies, tools, frameworks, and gamified approaches.
2. Chapter 3 describes the design and implementation of the proposed mutation testing module and contributions to GAMFLEW.
3. Chapter 4 presents the evaluation methodology and results.
4. Chapter 5 summarizes the main conclusions and identifies future directions.

Chapter 2

Related Work

This chapter will cover the systematic review of articles relevant to this project by analyzing their main objectives, methodologies, and possible limitations. Finally, it will summarize the main results of this research, concluding with a final discussion on what was discovered.

2.1 Objectives

With this analysis, we aimed to answer the following questions and accomplish the related objectives:

1. What methodologies are currently used to teach Mutation Testing?
   (a) Objective: Analyze different teaching methodologies and their perceived effectiveness.
   (b) Objective: Identify the main challenges associated with teaching Software and Mutation Testing.
2. What tools and frameworks are currently integrated in the teaching process of Mutation Testing?
   (a) Objective: Analyze existing tools and how they contribute to the teaching process of Mutation Testing.
   (b) Objective: Identify the capabilities and limitations of each tool and framework.
3. How does the use of gamification and existing serious games aid the teaching process of Mutation Testing?
   (a) Objective: Analyze existing serious games and how they affect students' motivation and engagement with the education on Mutation Testing.
   (b) Objective: Identify the main characteristics, mechanics, and limitations of current applications of gamification in the teaching of Mutation Testing.

4. How can we extend GAMFLEW so that it may also teach?

   (a) Objective: Analyze the mechanisms and systems applied by gamified approaches to education on Mutation Testing and how they help students learn the topic.

2.2 Methodology

To properly structure the literature review, the research methodology was defined and presented here, focusing on the databases consulted, the search query and how it was formulated, the filtering process applied to the results, and finally the thematic grouping of the selected articles.

2.2.1 Databases

The main sources of references used in this literature review were IEEE Xplore and Scopus, with the main topics being “Mutation Testing”, “Software Testing”, “Education”, “Gamification”, and “Tools”. The following query was used to gather references:

```
(
  "All Metadata":"mutation test*"
  OR "All Metadata":"software test*"
  OR "All Metadata":"unit test*"
)
AND
(
  "All Metadata":teach*
  OR "All Metadata":learn*
  OR "All Metadata":student*
  OR "All Metadata":educat*
)
AND
(
  "All Metadata":"gam*"
  OR "All Metadata":"serious game"
  OR "All Metadata":"game-based"
  OR "All Metadata":"tool*"
  OR "All Metadata":"ide"
)
NOT
(
  "All Metadata":medicine
  OR "All Metadata":biology
  OR "All Metadata":genetics
)
```

Listing 2.1: IEEExplore/Scopus search query

2.2.2 Query Formulation

The search query was formulated with three distinct query strings, brought together by the **AND** operator. The last query string is also defined by its use of a **NOT** operator, which excludes some results. Wildcards ($\star$) were used across the query to catch variations of some of the keywords.

The first block is focused on the main topic of software testing, unit testing, and particularly mutation testing. The second block covers terms related to teaching, learning, and education in general, which ensures a result with a focus on pedagogical works, rather than just applications of mutation testing. The third block helps narrow the results to works regarding gamification and serious games, as well as tools and frameworks, ensuring a wide coverage of both approaches.

Finally, the last block was added to remove results from other unrelated fields, where the word "mutation" has a different meaning, mainly in medicine, biology, and genetics. This block was necessary to remove unnecessary results that could disturb the research.

2.2.3 Filtering Process

The query described in Listing 2.1 returned approximately 2000 sources across both databases. The following filtering steps were then applied:

1. **Year range filtering.** The publication year range was restricted to 2000–2026, removing approximately 350 sources.

2. **Title and abstract screening.** The titles and abstracts of the remaining articles were analyzed to assess their relevance to the topics of software testing, mutation testing, education, serious games, and gamification. This step removed approximately 330 articles, leaving around 70 candidates for a full analysis.

3. **Snowballing and complementary search.** The reference lists of the previous remaining candidates were examined to identify additional relevant sources not captured by the query. Adding to IEEExplore and Scopus, we also obtained some sources from **ACM Digital Library**, hoping to find relevant articles that were not covered by the other two main sources. This step was particularly important to find additional works on mutation testing and technical reviews on gamification in software engineering education. These analyses would be highly relevant to the conception of our solution, but most were not caught by the original query, possibly due to its narrower focus on the intersection of testing, education, and gamification.

4. **Final selection.** After these last steps and a complete analysis of the full texts, this review resulted in a final bibliography of twenty-five (25) sources.

A summary of the previous filtering process, alongside the associated article count of each step, can be found in the table 2.1.

Filtering Step	Scopus	IEEExplore
Initial article count	1132	1291
Year Range Filtering	943	1017
Title and Abstract Screening	34	40
Snowballing and Complementary Search	9	16
Final Selection	9	16

Table 2.1: Article count in each filtering step

We should highlight that the figures presented above are approximate numbers, as database result counts tend to vary slightly between queries. They are nonetheless representative of the general size of each filtering step.

2.2.4 Article Grouping

After obtaining the final selection of sources, they were also categorized according to the area they address, into one of three groups:

- **Teaching Methodologies**: Sources that explore, analyze, or identify the main characteristics, advantages, and drawbacks of the diverse approaches to software and mutation testing education. Relevant for identifying the main difficulties associated with the teaching process and how our contributions may remediate them.
- **Tools and Frameworks**: Sources that present practical non-gamified approaches, like tools, extensions, or frameworks to teach software and mutation testing. Although mechanically different from our main focus, this group of articles still reveals important considerations to have in the design of our project.
- **Gamification and Serious Games**: Sources that discuss and identify the main advantages and limitations of existing applications of gamification and serious games in the realms of software testing and mutation testing. This is the largest group of articles and the main focus of sources in this systematic review. Due to the nature of ***GAMFLEW***, we decided to analyze the main gaps in the existing approaches to create a more robust solution.

The articles contained within each of these categories will be discussed in the Results (Section 2.3). As mentioned before, Table 2.2 presents the distribution of sources across the identified categories, sorting them according to the database that was used.

Some works overlapped with more than one category, but they were still grouped according to the topic that seemed the most relevant. Adding to this, some external sources were also included in these numbers, as the articles were considered important enough to be added to these groups.

	Category		
Source	**Teaching Methodologies**	**Tools/Frameworks**	**Gamification/Serious Games**
Scopus	4	0	5
IEEExplore	3	2	11
Total articles	7	2	16

Table 2.2: Article grouping by topic/category

2.2.5 Main Limitations

This literature review still contains a few limitations, which we felt should be disclosed here.

Query scope and balance.

The search query was designed to capture works related to software testing, education, and gamification. However, this intersection may have been too narrow, and may have introduced an imbalance in the results: works focused on gamification and serious games were more readily available than works on teaching methodologies or non-gamified tools, which is reflected in the final distribution of sources across categories (Table 2.2). This imbalance is partially addressed but remains as a potential source of bias.

Database coverage.

The primary sources used in this review were IEEE Xplore and Scopus, complemented with sources from the ACM Digital Library. While these databases already offer good coverage of literature in the areas of software engineering and computer science, they do not cover all relevant articles. Some relevant works published outside these databases may have been missed.

2.3 Results

In this section, we will discuss the main findings of this systematic review across the previously identified topics.

2.3.1 Foundations of Mutation Testing

Before reviewing the more educational literature, we found it pertinent to explore the foundations of mutation testing itself, which helps inform both the design of the mutation testing module developed in this dissertation but also the terminology used throughout this review.

Jia and Harman [12] provide an extensive survey of the development of mutation testing over a period of around 30 years, tracing its evolution from an academic curiosity into a concrete research area within Computer Science and Software Engineering. This article characterizes mutation testing as a fault-based technique that introduces small, syntactically valid changes ("mutants") into a program's code, to evaluate whether a test suite is capable of detecting the different behavior. In this context, a test that can distinguish the mutant's output from the original program's output is

said to *kill* the mutant; a mutant that is not detected by any test in the suite is said to *survive*. The survey also discusses the main relevant challenges of the field, including the computational cost of executing large numbers of mutants and the *equivalent mutant problem* - mutants that produce the same result as the original program, despite its syntactical difference, and which can therefore never be killed by any test input.

Building on this previous study, Papadakis et al. [18] studied advances in mutation testing, organizing the literature according to the stages of the mutation testing process: mutant generation, mutant execution, and mutant analysis. Of particular relevance to this dissertation is the study's discussion of techniques and methods to reduce the computational overhead of mutation analysis. The authors also address the equivalent mutant problem as one of the field's persistent challenges, noting that automating the detection of equivalent mutants remains an unsolved challenge, relying mostly on practical approaches like human judgment.

These two articles do not address education directly, but they establish two central points that serve as the main motivation of this dissertation. Firstly, the computational cost concerns raised in both studies reinforce the architectural decision discussed in Chapter 3 to keep mutant execution lightweight and client-side, operating on small code snippets rather than attempting to do so on larger, though more realistic, codebases. Second, the equivalent mutant problem is a well-documented challenge of the technique, which justifies dedicating some effort in addressing it within the GAMFLEW module.

2.3.2 Teaching Methodologies

Chen et al. [3] analyzed the current situation and existing problems of the software testing curriculum, aiming to discuss and improve the teaching methods of software testing courses. The authors identified three general drawbacks to the current curriculum:

1. **Teacher-centered education:** Most teaching curricula focus on one-way knowledge transmission, making it difficult for students to fully understand the material or receive timely feedback.

2. **Insufficient capability for independent learning:** The one-way transmission model weakens students' ability to learn independently, reducing participation and curiosity.

3. **Lack of practicality:** The overemphasis on theory often leads to students being unable to test real-world software products after completing a software testing course.

These three drawbacks are particularly relevant to mutation testing education. The abstract nature of concepts such as mutants, mutation operators, and mutation score makes mutation testing especially vulnerable to a teacher-centered, theory-focused approach: without hands-on practice, students may be able to grasp these concepts but still struggle to apply them in real-world applications. This reinforces our belief in tools and serious games that place students in an active,

motivated, and engaging environment while learning. The general conclusion is that software testing can be taught more efficiently through a combination of theoretical education, student practice, and open discussion.

Valle et al.'s [28] also conducted a characterization of the current state of software testing education, assessing it through a variety of topics and areas like curriculum design, teaching tools, and student evaluation. Using this analysis, the authors developed *"Testing Game"*, an educational game that addresses both structural (white-box) testing and mutation testing within a single tool. The game was designed around a set of existing approaches to software testing education and was evaluated with undergraduate students, with results suggesting an increase in engagement and motivation when compared to traditional teaching methods, though the authors note that measuring the learnability of any approach is usually difficult. Because of this, *Testing Game* is one of the few prior approaches that attempt to connect white-box testing and mutation testing within a single tool or platform. Despite being a standalone game rather than an extension of an already existing platform, this study remains as a crucial guide towards the design of this dissertation's solution.

2.3.3 Tools and Frameworks

2.3.3.1 IMA-CID

Maldonado et al. [13] introduced an Educational Module focused on mutation testing, based on the platform *IMA-CID* (Integrated approach for Modelling educational content). The platform is composed of three models:

1. **Conceptual Model:** Presents a high-level description of the knowledge domain, focusing on main concepts and relationships (e.g. Mutant, Dead Mutant, Equivalent Mutant, Mutation Score, Mutant Operator).

2. **Instructional Model:** Specifies the main concepts of the knowledge domain and defines additional information through pedagogic slides. Instructional elements are classified as *Explanatory*, *Exploratory*, or *Evaluative*.

3. **Didactic Model:** Defines how topics are presented to the student, in accordance with parameters specified by the course, educators, and students.

The content of the module was organized around four types of information items: concepts (definitions and explanations of dead, equivalent, and surviving mutants), facts (historical details about the origin of mutation testing), principles (relevant phenomena within the area), and procedures (the basic steps necessary to apply mutation analysis). These items were then presented through different media, including slides, HTML pages, text documents, and integrated testing tools like *Proteum*, which allowed students to apply Mutation Testing to concrete programs.

A preliminary evaluation of the module was conducted as part of a short course on software testing. The evaluation was carried out through a survey that analyzed the students' opinions regarding the content, the interface's usability, and general impressions of the module. The results

indicated an overall positive response to the module. Despite the generally positive results, the authors still acknowledge that the evaluation was limited in its scope, mostly due to time constraints. To conclude, they suggest a future controlled experiment to properly validate the approach.

2.3.3.2 FRAFOL

Tavares et al. [27] developed *FRAFOL* (FRAmework FOr Learning mutation testing), a platform that allows students to perform Mutation Testing on existing source code without requiring external IDEs or tools. The framework's design was motivated by a gap in existing educational tools recognized by the authors: while many mutation testing tools exist for both academic and industry environments, few provide an easy setup and integration into the modern teaching curriculum. Traditionally, students may be forced to install and configure multiple complex tools, making the topic more difficult to teach and likely resulting in decreased student motivation and engagement.

FRAFOL addresses this problem by providing a unified environment where students can use two widely adopted mutation testing tools: *Major* and *PIT*. The framework is implemented as an extension of *Defects4J*, an extensive database of reproducible Java bugs commonly used in research and education. It can be deployed via Docker to simplify installation, while also ensuring a consistent behavior across different machines.

The platform is centered around a web-based interface that exposes the framework's main capabilities: importing a Java project from the *Defects4J* repository, selecting a mutation tool, and analyzing the project's source code in the integrated code editor. After compiling and running the mutation tool against JUnit tests written by students, the framework displays the resulting metrics, including the number of generated, killed, and surviving mutants. It also registers other metrics not as relevant in Mutation Testing, like code and branch coverage. For each surviving mutant, FRAFOL also displays its details: unique identifier, the mutated line, the mutated method, and the applied mutation operator.

FRAFOL was evaluated through two user studies. A formative study with an early prototype, where several key improvements were detected, being subsequently implemented: simpler installation through the use of Docker, a refined graphical interface, and in-browser test editing. Adding to this study, a summative evaluation was also conducted, in which the participants were asked to complete a set of tasks using the platform. In general, participants rated FRAFOL positively across all assessed dimensions (Satisfaction, Learning effectiveness, Usefulness). Notably, the study reports a low value of perceived complexity, possibly indicating that students didn't find the platform difficult to understand.

Despite the encouraging results, the authors still acknowledge the limited scale of the evaluation and present suggestions for future work, including extending support to more Defects4J projects, enabling remote server deployment, introducing code coverage visualization, and developing a dedicated interface for teachers to monitor and manage students' progress.

2.3.4 Gamification and Serious Games

In a systematic literature review focused on Software Engineering education, Ngandu et al. [17] investigated how game elements are used to capture students' interest. Although the analyzed studies were inconclusive when identifying a single "best" element, the authors concluded that points and leaderboards are the most frequently implemented mechanisms, promoting flow states in the various SE topics. The study argues that the proper alignment between these elements and learning goals is crucial to expanding student engagement.

The effectiveness of reward and ranking mechanisms is corroborated by the tertiary study of Souza et al. [22], which presents the main findings on gamification within SE education. Their study revealed that the vast majority of research focuses on structural gamification, meaning the application of game elements such as leaderboards, badges, and points to modify the learning environment, while keeping the core educational content unchanged. Furthermore, competition and cooperation emerged as the most widely used social dynamics in the field, a finding we also aim to leverage.

Adding to the contributions of the previous analysis, Jesus et al. [11] also compared a gamified approach to the teaching of a software testing course against a more traditional non-gamified alternative. The results are notably mixed: while the gamified class was rated by students as more attractive and enjoyable, the measured learning was statistically similar in both approaches. Additionally, self-evaluated motivation did not seem to increase significantly in the gamified approach. The authors attribute part of this result to the inherent complexity of designing balanced and meaningful game mechanics. They also caution that gamification, if poorly designed, can lead to novel but superficial approaches that still fail to transmit genuine educational benefit. It is an important counterpoint to the generally positive results reported previously, and will be revisited in the Discussion (Section 2.4.1).

Another broad review of gamified approaches to software testing was conducted by Fulcini et al. [6], with their extensive literature review. The review characterizes gamified testing approaches according to several parameters, including the type of testing addressed (unit, integration, system) and the type of game mechanics employed. Among its conclusions, the review reports that existing gamified testing tools are predominantly applied in the phase of test *creation*, noting that unit testing and mutation testing tools form an identifiable group of contributions. The authors also defend that serious games specifically designed for mutation testing remain a small collection even within this already narrow set of articles. This finding directly guides one of the central motivations of this literature review: despite growing applications of gamification in software testing as a whole, serious games dedicated to mutation testing are still rare. Out of the few examples, even fewer attempt to integrate mutation testing with white-box coverage criteria in a single coherent platform.

2.3.4.1 Gamification in Software Testing

IntelliGame [25], a gamified plugin for the IntelliJ IDE developed by Straubinger et al., uses a variety of game-based elements to encourage developers to adopt better testing behaviours. Developers are rewarded for tasks such as writing and executing tests, achieving code coverage, and using debugging features.

This approach isn't focused on teaching new concepts; instead, it uses gamification elements such as achievements, badges, and progress indicators on top of existing tools to reinforce desirable behaviors that developers should already know how to perform. This is a different goal from the ones usually addressed by serious games, a topic which we will examine next, where the primary objective is to teach concepts that the player may not yet understand.

The validity of *IntelliGame* was subsequently examined by de Jesus et al. [10] in a study involving 174 participants that extends the original evaluation to the TypeScript programming language. The study also broadly supports the supposed motivational benefits reported in the original study, while also highlighting that the magnitude of the effect can vary across programming languages and participant populations. The authors also note the importance of evaluating gamified testing tools across multiple contexts. This was kept in consideration in the Chapter 4, aiming to recruit participants from multiple backgrounds (students, professionals, and other programmers), hopefully strengthening the validation of this dissertation's contributions.

The previous authors also conducted a broader analysis of gamification within the software testing domain [9], where they systematically categorize the gamification elements, mechanics, and characteristics addressed by existing gamified testing tools and approaches. Their characterization found that most gamified software testing approaches already covered some topics like unit testing and test execution, but few addressed white-box coverage analysis or mutation testing specifically.

Another relevant contribution to the previous reviews may be Costa and Oliveira's [5] proposed gamified framework for introducing game elements into exploratory testing instruction. While the work targets exploratory testing instead of mutation testing, its proposed solution, which involves combining clear scoring rules, incremental challenge difficulty, and mapping between game mechanics and learning objectives, still offers a methodological template that is generally applicable to the design of any gamified testing education tool or serious game. Because of this, the review of this article helped us identify key mechanics and considerations to include in the Mutation Testing educational module developed in this dissertation.

2.3.4.2 Serious Games for Software Testing

Unit Brawl [16] (Materazzo et al.) is a gamified multiplayer platform designed to support the weekly classes of a programming course, where students traditionally submit Java code without writing accompanying tests. The platform attempts to improve the classes by applying a *battle-royale* evaluation model, where each exercise requires students to submit both the implementation and a corresponding unit test suite. These submissions are all sent to a GitHub repository where

they are first validated through a continuous integration pipeline that checks if it compiles, ensures a maximum number of test cases, and finally verifies that the submitted tests do not fail. Once the submission deadline previously set by the classes' teachers or administrators expires, every player's test suite is executed against every other player's implementation. Using this logic, a player earns one point for each opponent's bug that their tests detect, and one point for each opponent's test suite that fails to find a bug in their own implementation. The article also specifies that all players get at least one point for passing the initial validation step. As seen, the game already leverages its competitive nature to keep the students engaged in the exercises, but it also applies several gamification mechanics, including a leaderboard (both a global leaderboard and a leaderboard for each class). These leaderboards show only the player's immediate competitors in rank. According to the authors, this works as a way to avoid demotivating lower-ranked students. Additionally, the platform also contains customizable avatars, purchasable with points converted into an in-game currency, and achievements tracking each player and their objectives or goals, such as test coverage. Finally, the authors conducted a tentative technical evaluation of the platform, but, at the time of publication, had not yet conducted a validation study with real students who could assess the educational impact of the approach.

GAMUT [7] (Gomes et al.) adapts the board game *Mastermind* [26] to teach unit testing concepts through a multi-stage educational approach that joins three main components: a web-based game, a recorded video lesson, and a practical homework activity, all interconnected by a single platform. In the game, the player takes on the role of a soldier who must defeat a monster by selecting the correct sequence of weapons through trial and error (the main mechanic inspired by *Mastermind*). Before attacking, the player may also test a candidate sequence against a training dummy, which provides feedback without wasting the limited number of attempts given to the player (As specified by the article, each player has three attack attempts, which are deducted in case of a wrong sequence, but also five testing attempts that are refreshed every time there is an attack with a wrong sequence). This flow directly mirrors the given (test setup), when (action), then (assertion) structure common in unit testing. The connection between trial and error and software testing is explicit in the design: a weapon sequence corresponds to the test setup (*given* phase), the attack attempt corresponds to the action (*when* phase), and the outcome of an attack or test corresponds to the assertion (*then* phase). As stated before, this mechanic is one of the more direct derivations of the original Mastermind game, where the weapon sequence is instead a sequence of four different colors. The authors justify the replacement with weapon icons by its connection to the narrative presented to the students, while also avoiding the exclusion of color-blind students. The approach also includes a hidden "developer mode" that displays the game's elements with their direct testing-domain equivalents (the monster becomes the dependency, the dummy becomes the testing double). This is a particularly important approach that allows players to more easily connect the given examples to the real testing concepts they represent. To evaluate GAMUT, the authors conducted a validation survey with 27 students, under the context of a Verification and Validation course, combining a questionnaire on the game and video lesson, a second questionnaire on the practical exercises, an online forum for qualitative feedback, and the

teacher's grading of the students' submitted unit tests. The results seemed to be mostly positive, with most participants rating the platform an 8 to 10 (on a 10-point scale). Adding to this, the majority of teams successfully applied the core mechanics of the game, but the evaluation also indicated a recurring misconception: students frequently implemented a separate stub for every test case rather than a single reconfigurable stub, and several teams omitted the spy or dummy double entirely, indicating that the distinction between test double types was not communicated with full clarity by the video lesson.

Bug Hide-and-Seek [1] (Buffardi et al.) divides players into two roles within a programming problem:

- *"Hiders"*, who must implement a solution to the problem while intentionally leaving a resilient or hard-to-detect bug that alters the behavior of the program.
- *"Seekers"*, who must implement a robust solution while writing unit tests capable of distinguishing between correct and buggy implementations across the program.

Crucially, the game's educational core lies in its evaluation methodology. Unlike other approaches, the platform doesn't treat a submission as simply correct or incorrect; instead, it isolates each function and runs only the subset of a student's tests that target that specific function against every other student's implementation of it. Each student receives a True Positive Rate (the proportion of correct implementations their tests correctly accept) and a True Negative Rate (the proportion of buggy implementations their tests correctly reject) for each function. This model allowed the authors to experiment with two variants of the game, using the game Tic-Tac-Toe for the implementation: a *between-subject* design, where each student is assigned either the Hider or Seeker role for an entire class, and a *within-subject* design, where each student attempts a correct implementation of most functions while having to act as a Hider role for exactly one random function. According to the authors, the within-subject design proved more practical, since it does not depend on a large number of students producing an entire implementation without bugs, unlike the between-subject experiment. The authors also consider that the metrics they introduced in their evaluation model, regarding both implementation and testing (True Positive Rate and True Negative Rate) seem like a better indication of a student's correctness than more traditional unit testing measures like condition/decision coverage. Even if not completely applied, this approach sets a crucial guideline in our proposed design.

These three games share a relevant point of peculiarity for this dissertation regarding how player input is evaluated. *Unit Brawl* and *Bug Hide-and-Seek* both require students to write actual code, executing it against other students' implementations or test suites. While *GAMUT* does not involve code execution within the game itself - its core mechanic is a guessing game in which the player selects weapons from a fixed set, this being evaluated against a predefined target - it does apply real code in its practical activity phase, where students write actual Python unit tests, separated from the game's main mechanic. This distinction is particularly relevant due to the preexisting GAMFLEW evaluation model described in Chapter 1, which previously relied on

predefined Boolean conditions rather than real code execution, with a game structure similar to *GAMUT*'s guessing mechanic rather than the approaches of *Unit Brawl* and *Bug Hide-and-Seek*.

2.3.4.3 Serious Games for Mutation Testing

Code Critters [24] (Straubinger et al.) is a serious game based on Tower Defense that uses a block-based programming language to teach mutation testing concepts to players with little or no coding experience. In the game, players must help healthy *"critters"* from mutant ones by placing magic mines along the path they travel. Each mine represents a test case that contains both test inputs (coordinates and terrain type of the tile it is placed on) and test assertions (blocks that define the expected behavior of the critter). The objective of the game is to place these magic mines so that healthy critters pass through them unharmed, while mutant critters become trapped. The code that influences the critters' behavior is represented as a short snippet using the block-based language. Mutant critters are variants of this snippet that contain one or more mutations, such as an incorrect initialization or a changed condition. This design allows the game to teach concepts like statement coverage, branch coverage, condition coverage, and input partitioning, thanks to its customizable level editor, where class admins can design more specialized entries. Adding to this, the game also encourages players to minimize their test suite by using as few mines as possible, reinforcing the concept of efficient test suites. The tool also includes a leaderboard and progressive difficulty levels (tutorial, beginner, and advanced), which are relevant elements to apply in our own design.

A follow-up study conducted by the same authors (Straubinger et al.) [23] extended the evaluation of *Code Critters* by experimenting with a younger audience, conducting a controlled experiment involving 40 children aged 11 to 16 in a school context. The experiment revealed that most participants actively engaged with the game as expected. The authors also observed three distinct behavioral patterns across the participants: a smaller group that engaged minimally, placing mines at random; a larger group that attempted each level methodically, aiming for a complete solution before progressing; and a third group that prioritized speed, pushing through levels as quickly as possible. The survey still revealed promising results, indicating that most of the children found *Code Critters* enjoyable, and that some still chose to continue playing the game at home. It also demonstrated that players detected an average of 90% of mutants and correctly identified around 70% of healthy critters. However, the authors acknowledge the small sample size and limited grade diversity as threats to the validity of the results, but still plan to extend the game with additional concepts and improvements.

Code Defenders [19] (Rojas et al.) is a competitive two-player web-based mutation testing game centered around Java and JUnit tests. Players take on one of two roles:

- *Attacker*, who aims to introduce discreet non-equivalent mutants into a class under test
- *Defender*, who writes JUnit tests to kill those mutants

The game proceeds through turns, with attackers scoring points for mutants that survive the test suite, and defenders scoring points for tests that kill mutants. The equivalent mutant problem is also addressed through a mechanic by the name *equivalence duel*: when a defender suspects a mutant is equivalent, they can challenge the attacker to submit a killing test (points given to the attacker), or to accept the mutant as equivalent (points given to the defender). This is an interesting mechanism that forces players to think carefully about whether the mutant is equivalent before making a challenge. The authors also introduced two difficulty modes: *Easy* and *Hard*. In *Easy* mode, the defenders can see the difference introduced by the mutant in the displayed code; in *Hard* mode, only a brief description is provided, more akin to a black-box approach. The article still recognizes some of its limitations, including the need for single and multiplayer modes, expansion of the two-person structure, a better scoring system, and the possibility of abstracting the gameplay from direct code interaction to increase accessibility.

Clegg et al. [4] subsequently explored the use of *Code Defenders* as an educational platform, suggesting a new structure of puzzles within the game that could address a wide branch of software testing concepts, including statement coverage, branch coverage, loop testing, boundary value analysis, dataflow testing, and mutation testing itself. In their approach, each concept corresponds to a level in the game, each one composed of a series of games with increasing difficulty. Players would continue to act as an attacker, creating a discreet mutant, or as a defender, writing a test that detects a pre-defined mutant. This contribution is particularly relevant to this dissertation as it presents an existing example of a mutation testing serious game already used to teach software testing concepts through its gameplay.

ModelDefenders [2] (Cammaerts et al.) is a proposed serious game that applies the same attacker-defender mechanic as *Code Defenders*. However, this approach relies on models used in Model-Driven Engineering (MDE), specifically Finite State Machines and Class Diagrams. Defenders define test cases as sequences of events with expected outcomes, while attackers introduce slight mutations to the models (this can be changing transition labels, adding or removing transitions, or altering multiplicities in class diagrams). The tool executes the test cases using the original model before accepting them, preventing the definition of incorrect tests, and then uses them to check if the mutants introduced by the attackers. At the time of publication, *ModelDefenders* is presented as a prototype in Figma, still under development, with a formal evaluation planned for future work. The authors also discuss several challenges in adapting mutation testing to MDE components and how to evaluate test cases against mutated class diagrams while accounting for object pointers and dependency relationships.

EMVille [8] (Houshmand et al.) is a slightly different approach that specifically targets the equivalent mutant problem through a quest-based gamification system. The platform presents quests to the player with an original program and one of its mutants, and the player must determine whether the mutant is equivalent or not equivalent. Quests are classified into three types according to whether the correct answer is already known (either from a previously submitted killing test case or from an automated detection technique) or still uncertain. The scoring system rewards players for providing a killing test case for a previously unsolved quest, and penalizes players declaring

mutants as equivalent when a killing test already exists. The game environment also provides several tools to support the player, including a diff visualizer, a coverage analyzer, and step-by-step execution for tracing the behavior of both the original program and its mutant. The authors conducted an initial evaluation with twelve participants, each with at least four years of experience with Java. The evaluation revealed significant benefits, with all participants reporting preferring the gamified approach over the traditional one, noting that participants frequently interacted with the leaderboard, suggesting that the competitive nature was a powerful source of motivation.

Table 2.3: Comparison of serious games for software and mutation testing education.

Game	**Focus**	**Number of Players**	**Prior Coding Knowledge**
Unit Brawl	Unit testing	2+	Yes
GAMUT	Unit testing	1	Partial
Bug Hide-and-Seek	Unit testing	2+	Yes
Code Critters	Mutation testing	1	No
Code Defenders	Mutation testing	2	Yes
ModelDefenders	Mutation testing	2	Partial
EMVille	Mutation testing	1	Yes

2.3.4.4 GAMFLEW

GAMFLEW (GAMe For LEarning White-box testing), developed by Silva et al.[21], is a serious game for teaching white-box testing, where students define the test input data needed to achieve code coverage objectives defined in challenges.

The game is centered around challenges that encourage students to interact with a checkers board, defining the input data through piece movement, placement, or removal. When a student wants to submit an attempt to complete a challenge, they are also prompted to submit a comment, which their teacher or educator can then analyze. The platform then checks if the player's input achieves the predefined conditions, and, if true, rewards them with points relative to the assigned difficulty of each challenge. Adding to this scoring system, ***GAMFLEW*** also introduces a global leaderboard that tracks each player's accumulated points, passed attempts, and failed attempts, fostering a sense of competitiveness. Another mechanic that keeps students engaged and interested in the challenges is the use of achievements. In this context, each achievement is another predefined condition associated with each challenge that can only be completed through a dedicated input. When a challenge is passed, the player can access an achievement hint, giving them a slight clue on how to complete the achievement. The player must reattempt the challenge with a passing test case that also completes the achievement. In the case of a successful attempt, the player is prompted that they passed the challenge and gained the achievement.

This serious game already successfully applies gamification with its use of points, leaderboards, and achievements to teach players about software testing concepts while boosting their motivation. Not only does it contain multiple single-player challenges, but also a multiplayer mode where players compete for who can beat the most challenges with the fewest attempts through a limited number of rounds. The platform's dual nature in allowing both single and multiplayer

gameplay remains one of its core strengths, allowing players to interact with it in more diverse situations.

The validity of this approach was tested by the authors with two experiments, one surrounding students' interactions with the platform and the other involving teachers. The first experiment was focused on students interacting with the platform, followed by a short exam to gauge how much these students had learned from interacting with the game. This was also supplemented by two questionnaires, answered by the students before and after using ***GAMFLEW***: The pre-questionnaire was used to collect data from the students regarding their personal characteristics (age, gender, course) and technological skills (knowledge of programming, software testing, coverage analysis), while the post-questionnaire was more focused on feedback regarding the platform (usability, usefulness, accessibility, productivity, efficiency, comfort). The second experiment was instead centered around teachers' experimentation with the platform, characterized by the smaller post-questionnaire that collected similar but otherwise distinct feedback from the teachers regarding questions like usability, usefulness, efficiency, and productivity. Both of these experiments allowed the authors to conclude that their approach seemed generally effective at teaching white-box testing concepts, while keeping students engaged and motivated through its different pedagogical model. They also note that there were a significant number of positive student comments regarding the platform, but that it is still limited in some areas and open to further future work.

As was mentioned previously, the serious game already contains a diverse array of validated mechanics that contribute to teaching white-box testing concepts through its extensive number of challenges, but these don't yet cover other relevant areas of the topic, such as mutation testing. We aim to leverage these advantages to model our own extension focused on Mutation Testing, alongside the strengths and limitations of the previous examples analyzed in this literature review.

2.4 Discussion

This section will summarize the results of the reviewed literature, connecting the main findings across the identified categories and identifying the gaps in them, with the intent of guiding the contributions of this dissertation.

2.4.1 Summary and Identified Gaps

The literature review carried out in this chapter covered three main areas: teaching methodologies for software and mutation testing, non-gamified tools and frameworks, and gamified approaches, including serious games. Together, these sources demonstrate the current state of mutation testing education and the main challenges associated with it.

On the topic of **teaching methodologies**, the reviewed works seem to agree on a set of recurring difficulties. Chen et al. [3] identify the predominance of teacher-centered, theory-focused instruction as an obstacle to effective software testing education, while Valle et al. [28] reinforce this idea by developing an educational game to help solve the perceived limitations. The abstract

nature of mutation testing concepts makes this field particularly vulnerable to lecture-based instructions, without the application of hands-on practice. Students may understand the concepts but still struggle to apply them in real contexts, a phenomenon that reinforces the value of interactive tools and serious games.

In the area of **tools and frameworks**, the reviewed works demonstrate that specialized educational platforms for mutation testing, such as IMA-CID [13] and FRAFOL [27], can provide meaningful support for techniques in the area. However, these tools tend to prioritize the correctness of the learning content instead of student engagement or motivation. None of these approaches incorporate gamification elements into their design, being more correctly characterized as learning environments than as games.

And finally, on the side of **gamification and serious games**, the reviewed works reveal a wide landscape of approaches dedicated to mutation testing. The studies by Fulcini et al. [6] and de Jesus et al. [9] confirm that most gamified testing approaches address broad software testing concepts like unit testing, but that mutation testing remains an underexplored area. Among the games that do address mutation testing, the reviewed works still reveal repeating patterns and trade-offs that are worth analysing.

As per the reviewed literature, the majority of serious games for mutation testing require either two players (*Code Defenders* [19], *ModelDefenders* [2]) or assume the players have prior coding knowledge (*Code Defenders*, *EMVille* [8]). *Code Critters* [23, 24] is a notable exception that lowers the skill barrier through a block-based language and a singleplayer format accessible and attractive to younger learners. However, this simplification can limit its applicability to learners with a wider programming background. *EMVille* [8] strays away from the other reviewed examples, focusing specifically on the equivalent mutant problem, a challenge that is largely absent from the other games. The scope of the game may be focused and interactive, but it may be too narrow, while also assuming the players have a prominent experience or familiarity with mutation testing.

Table 2.4 summarizes the reviewed tools, frameworks, applications of gamification and serious games along parameters relevant to this dissertation, providing a view into the gaps identified later in this section. It is worth noting that the chosen parameters (topic addressed, number of users required, and type of approach) are not exhaustive, but are still used to guide the design considerations that are most important to the development of the mutation testing module described in Chapter 3.

- **Absence of approaches connecting white-box testing and mutation testing.** None of the reviewed tools or serious games explicitly connects the teaching of coverage analysis concepts (such as statement, branch, or condition coverage) with mutation testing within the same learning environment. The only two approaches that come close are *Code Critters*[24], which can target different coverage concepts through its level design, and Clegg et al. [4], who propose a new structure for *Code Defenders* that would map diverse coverage concepts and mutation testing across several challenges. Valle et al. [28] also attempt this integration in their *Testing Game*, but as a singular game rather than an extension of

Table 2.4: Comparison of reviewed tools and serious games against the dimensions relevant to this dissertation.

Approach	Topic addressed	Users needed	Approach Type
IMA-CID	Mutation Testing	Singleplayer	Tool
FRAFOL	Mutation Testing	Singleplayer	Framework
IntelliGame	Software Testing	Singleplayer	Gamified IDE
Unit Brawl	Software Testing	Multiplayer	Serious Game
GAMFLEW	Software Testing	Singleplayer/Multiplayer	Serious Game
GAMUT	Software Testing	Singleplayer	Serious Game
Bug Hide-and-Seek	Software Testing	Multiplayer	Serious Game
Code Critters	Mutation Testing	Singleplayer	Serious Game
Code Defenders	Mutation Testing	Two-player	Serious Game
ModelDefenders	Mutation Testing	Two-player	Serious Game
EMVille	Mutation Testing	Singleplayer	Serious Game

an existing platform with validated approaches. *GAMFLEW* is, to our knowledge, the only approach that proposes a union of white-box coverage and mutation testing within a single serious game that has already been validated.

- **Multiplayer dependency.** Many of the serious games for mutation testing, particularly *Code Defenders*[19] and *ModelDefenders*[2], require at least two players to function as designed. Although it is a good approach to increase competitiveness, this imposes a constraint on its use in a real-world setting, where organizing participants into pairs can be complex, or simply there may not be sufficient people to use the game correctly. Singleplayer alternatives exist (*Code Critters*[24], *EMVille*)[8], and even *GAMFLEW*[21] apply it partly, but these approaches either target a different audience or a narrower topic within Mutation Testing.

- **Lack of sustained motivation mechanics.** Several of the reviewed approaches note the difficulty of maintaining student engagement and motivation. Jesus et al. [11] caution that gamification, if poorly designed, can produce approaches that feel novel but fail to provide a meaningful learning benefit. Few of the reviewed tools incorporate mechanics designed to maintain engagement over multiple uses, such as progression systems, narrative, or increasing difficulty.

- **No approach integrates mutation testing into an already validated serious game.** The reviewed works either propose entirely new solutions in the form of serious games or gamified tools built from scratch. None of them extends an existing serious game to incorporate mutation testing as an additional module. Our approach would preserve ***GAMFLEW***'s mechanics and user experience while expanding its educational scope to address this topic, oftentimes forgotten or undervalued in the teaching process.

Chapter 3

GAMFLEW

This chapter will describe the design and implementation of the contributions developed in the context of this dissertation. We will analyse the existing contributions, define the functional and non-functional requirements derived from the previously identified gaps, present the conceptual design of each contribution, and describe the previous system architecture and how the new components were integrated into ***GAMFLEW***. Adding to this, the chapter will also discuss and justify the chosen technology stack, followed by the technical implementation of the main contributions.

3.1 Challenges Overview

Before detailing the technical requirements and implementation, this section introduces the different types of challenges available in ***GAMFLEW***, both the pre-existing white-box testing challenges and the mutation testing challenges introduced by this dissertation. Understanding these challenge types beforehand provides the necessary context for the more technical discussion that follows in the remainder of the chapter.

3.1.1 White-box Testing Challenges

The white-box testing challenges that were already part of ***GAMFLEW*** prior to this dissertation are grouped into five types, each corresponding to a different coverage criterion:

- **Statement Coverage** - the player must provide an input that causes every statement in the challenge's predefined line or line range to be reached at least once during execution.
- **Decision Coverage** - the player must provide inputs that exercise both the `true` and `false` outcomes of the decision contained in the predefined line or line range (for example, both branches of an `if`/`else` statement). Usually, Decision challenges contain two boards, allowing the player to hit both outcomes with one submission.
- **Condition Coverage** - the player must provide inputs such that each atomic boolean subcondition within the wider specified decision evaluates to both `true` and `false`. For

each Condition challenge, the number of boards will be equal to twice the number of sub-expressions in the condition, since each must be true and false at least once in the player's submission. Some conditions may require more or fewer boards, depending on the function structure.

- **Decision/Condition Coverage** - the player must satisfy both Decision Coverage and Condition Coverage simultaneously, ensuring that every branch is taken and every individual condition is exercised for both values.

- **Modified Condition/Decision Coverage (MC/DC)** - for each condition within a decision, the player must demonstrate, through their set of inputs, that the condition can independently influence the outcome of the overall decision, with all other conditions held constant. For this type of challenge, the number of boards can be represented by *N+1*, where N is the number of sub-expressions in the given condition.

These challenges are grouped by their associated function code, with a total of eight distinct code files. The code snippets used in the code files can be found in Section B.

Figure 3.1 is a screenshot from the platform, demonstrating some of the existing challenges and how they are organised according to type and difficulty.

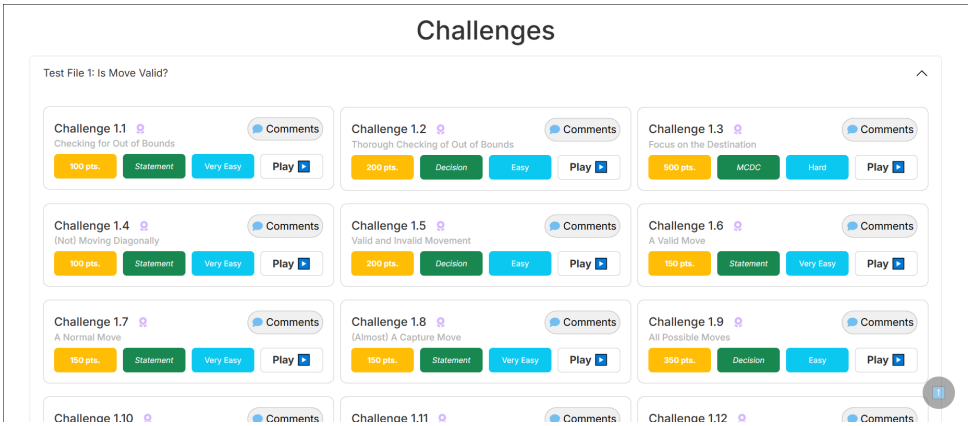

Figure 3.1: Original White-box Testing Challenges - Code File 1

3.1.2 Mutation Testing Challenges

To introduce mutation testing as a new educational topic within ***GAMFLEW***, two complementary types of mutation testing challenges were designed:

- **Kill the Mutant.** The player is presented with an original function and a mutated version. The mutation is highlighted to draw the player's attention to the modified line. The player must provide an input value such that the two versions produce different outputs. The execution engine runs both versions with the provided input and compares their return values. If they differ, the mutant is killed, and the challenge is passed.

- **Equivalent Mutant.** The player is presented with a mutant that cannot be killed by any input, because the mutation does not alter the observable behavior of the function. The player is not told in advance whether the mutant is equivalent. They may attempt as many inputs as they wish; if they are unable to find a killing input, they may explicitly mark the mutant as equivalent. If the challenge is indeed flagged as equivalent in the system, the submission is accepted, and the challenge is passed.

The two challenge types are designed to reinforce complementary aspects of mutation testing: *Kill the Mutant* develops the player's ability to reason about the behavioural impact of code mutations, while *Equivalent Mutant* develops their understanding of the limits of mutation testing and the concept of semantic equivalence.

Figure 3.2 presents a portion of the introduced mutation testing challenges, demonstrating how they are organised and displayed to the player alongside the previously developed white-box testing challenges.

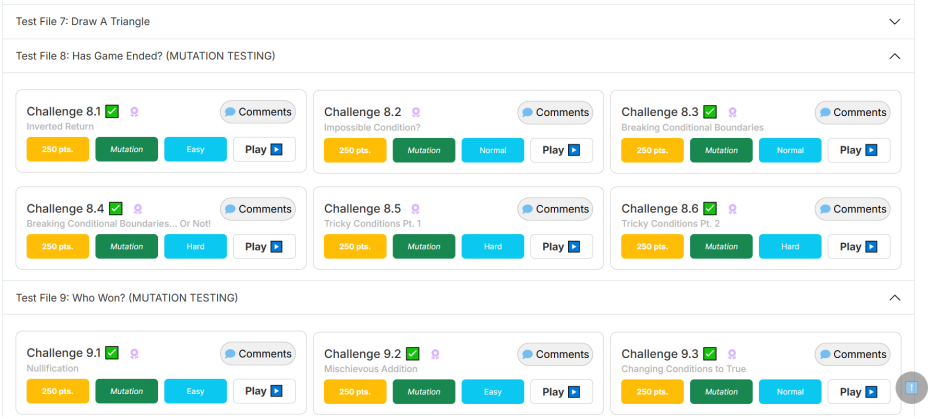

Figure 3.2: Screenshot of the List of Mutation Testing Challenges

3.2 Requirements Analysis

The requirements described in this section were derived from two sources: the gaps identified in the systematic review (Chapter 2), and the technical limitations of the ***GAMFLEW*** evaluation model described in Chapter 1.

These requirements will work towards one main contribution: a **client-side code execution engine**, enabling the implementation of **real coverage analysis**, and a **mutation testing educational module** based on it.

3.2.1 Requirements

- **R01** - The platform shall execute JavaScript code entirely on the client side, without requiring any server-side computation.

- **R03** - The platform shall track which statements, decisions, conditions, and MC/DC pairs are covered by a given player input, using this information to determine whether a challenge's objectives have been met.
- **R04** - The outcome of existing software testing challenges shall be determined by the results of real code execution and coverage analysis, replacing the previous model, which relied on Boolean conditions.
- **R05** - The platform shall support a new category of challenges in which the player is presented with an original function and a mutated version of it, asking them to provide an input that can kill the mutant(s).
- **R06** - The platform shall execute both the original and the mutated version of a function for the same input, compare their outputs, and determine whether the mutant has been killed.
- **R07** - The platform shall support a challenge variant in which the player must determine whether a presented mutant is equivalent to the original program by attempting inputs and, if unsuccessful, explicitly marking the mutant as equivalent.
- **R08** - The platform shall provide immediate feedback to the player after each submission, indicating the result of the execution.
- **R09** - The execution engine shall handle erroneous player inputs gracefully, including syntax errors and runtime exceptions, without crashing the application.
- **R10** - The execution engine shall complete the evaluation of a player's submission within a specified time frame that does not disrupt the game flow. If the code execution exceeds a defined timeout threshold, the engine shall terminate it automatically.
- **R11** - Code execution shall be performed asynchronously, without blocking the main application thread or degrading the responsiveness of the user interface.
- **R12** - Executed code shall be isolated from the main application to prevent player-submitted inputs from interfering with GAMFLEW's internal state or causing unhandled exceptions in the UI.
- **R13** - The integration of real code execution into existing white-box testing challenges shall not alter the intended scoring mechanic. Existing challenges shall continue to pass or fail under the same intended conditions as before.
- **R14** - The mutation testing module shall present both the original and mutated code in a visually clear format that allows the player to identify the difference between them.
- **R15** - The platform shall remain usable in standard modern browsers without requiring the installation of additional plugins, developer tools, or external dependencies.

3.3 Conceptual Design

The design of the contributions developed in this dissertation was structured around a single common component: a code execution engine. This engine would enable two distinct but closely related extensions to ***GAMFLEW***, which we will describe in the following subsections.

3.3.1 Coverage Goal Tracker

Before this work, each existing challenge of ***GAMFLEW*** was predefined with a set of Boolean expressions, named "preconditions" and "tests", which evaluated whether the player's inputs passed the challenge or not. This means that the platform never executes the code displayed to the players. Each submission is passed only if all the challenge's preconditions and tests are true without executing any real code. This behavior can be observed through the average game flow of a challenge presented by Figure 3.4 in Section 3.4.

With a functional code execution engine, the displayed code could be instrumented with coverage calls based on the challenge type and executed using the player's chosen input. The platform would then compare the coverage results with the coverage goals of the respective challenge, passing it if all the objectives are met.

From the perspective of a player, this approach should not change anything in the game flow. The changes should be only internal, replacing the current evaluation method with real coverage tracking. This would make future white-box testing challenges easier to create and customise, without hindering the proven functionality of the platform.

3.3.2 Mutation Testing Challenges

The second extension introduces mutation testing as a new educational topic within ***GAMFLEW***, through the two challenge types presented in Section 3.1.

The two challenge types are designed to reinforce complementary aspects of mutation testing: *Kill the Mutant* develops the player's ability to reason about the behavioural impact of code mutations, while *Equivalent Mutant* develops their understanding of the limits of mutation testing and the concept of semantic equivalence.

3.4 System Architecture

3.4.1 Previous Architecture

GAMFLEW was designed as a web-based platform and therefore follows a system architecture based on two components: a frontend and a backend.

The frontend is the component most interacted with by the players, as it handles the main game logic, displays the game area, and presents leaderboards and achievements. The player has access to multiple challenges, each associated with a function code file and a coverage goal. To solve a challenge, the player must interact with a checkers board, adding, removing, moving

or changing pieces in it, which the frontend saves as the game state and input for the challenge evaluation. on The backend serves as the storage of challenge data and submission attempts, but is also responsible for the player's authentication on the platform. To achieve the latter, it makes use of Bearer Tokens, assigning one to each player when they log in. The frontend then loads the challenge data as actual challenges to the player and submits player attempts to the backend.

Figure 3.3 presents the original system architecture of ***GAMFLEW***, as described by its authors.

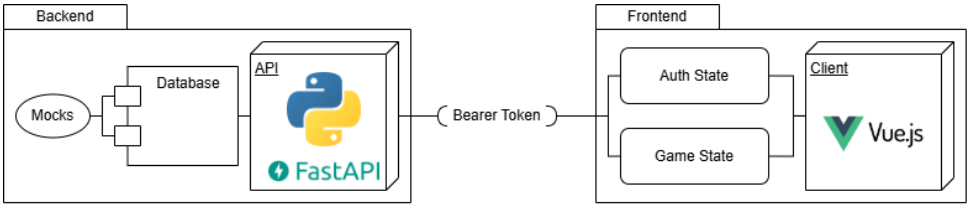

Figure 3.3: Previous GAMFLEW System Architecture

With this architecture in place, all the challenge data, including the **name**, **description**, **challenge type**, **code file**, **number of boards**, **preconditions**, and **tests**, are loaded from the backend when the player enters a specific challenge. Although these challenges rely on the same evaluation logic with their predefined boolean expressions, the challenges are assigned one of five challenge types, related to the coverage type it addresses, as described in Section 3.1

The Game Board is the central interactive component of each ***GAMFLEW*** challenge. It consists of an 8x8 square checkers grid in which each cell can have one of two types of pieces, distinguished by color: **blue** and **red**. Cells that are not occupied by any piece are considered empty. In addition to these two piece types, a cell may also be in a **stack** state, representing two pieces stacked on the same cell. Unlike real checkers, the game board doesn't force stacks to have two pieces of the same color. The board also includes a designated **out-of-bounds** cell with modifiable coordinates, which can go beyond the supported rows and columns of the board. This cell can hold a piece in any of the same states as a regular board cell and serves to represent pieces outside the game board.

The player can interact with the board through two distinct interaction modes. By default, the player can **move pieces** by clicking a cell containing a piece to select it, and then clicking a destination cell to complete the move. In the secondary **add/remove mode**, activated explicitly by the player, clicking an empty cell places a new piece on it, while clicking a cell already containing a piece either changes its color or removes it from the board. This mode allows the player to construct board configurations that would not be reachable through movement alone (for example, a board with an "illegal" number of pieces), which is necessary for challenges that require specific conditions to achieve the objective.

The input data submitted for evaluation is a structured object that represents the configuration of the board, the state of its cells and the previous moves at the moment of submission. This object includes the state of each cell (empty, blue, red or stack), the state of the out-of-bounds cell, and

a **move log** recording the sequence of interactions performed by the player during that attempt, categorising them by their types: piece placements, removals, or movements. It is this board state object that is passed to the challenge's evaluation logic, where the predefined **preconditions** and **tests** are evaluated with it.

Each challenge may contain multiple boards, each representing a separate input necessary to cover all coverage goals of the challenge function. The player only has access to one board at a time, but can cycle through them freely without losing the alterations done to them. The number of boards in a challenge is therefore directly tied to the coverage objective it addresses and determines how many distinct inputs the player must provide in a single submission. Alongside the board, the player can also see the challenge's code file and is prompted to fill in a configuration for each board before submitting all of them together in a single attempt. In this step, the players are required to leave a comment explaining their thought process, even if incorrect. The submission comments can be then analyzed by the teachers, allowing them to receive and provide feedback to the students, aiding the teaching process. The evaluation is then performed across the set of boards, with the coverage targets being assessed using the combined results of all board states.

When an attempt is submitted, the platform checks the challenge's predefined Boolean preconditions and test conditions with the submitted board states. A challenge is passed if and only if all conditions across all boards evaluate to `true`. If any condition fails, the player is informed that they did not pass the challenge and are prompted to adjust their attempt and resubmit.

With this architecture, the expected game flow for each challenge would revolve around the player choosing an input by interacting with the board, submitting an attempt with a comment, and waiting for the frontend to evaluate the challenge's conditions. If all associated conditions are given as `true`, then the player passes the challenge. Otherwise, the player is warned that their attempt is incorrect and is prompted to try again. The average game flow for one challenge is presented below in Figure 3.4

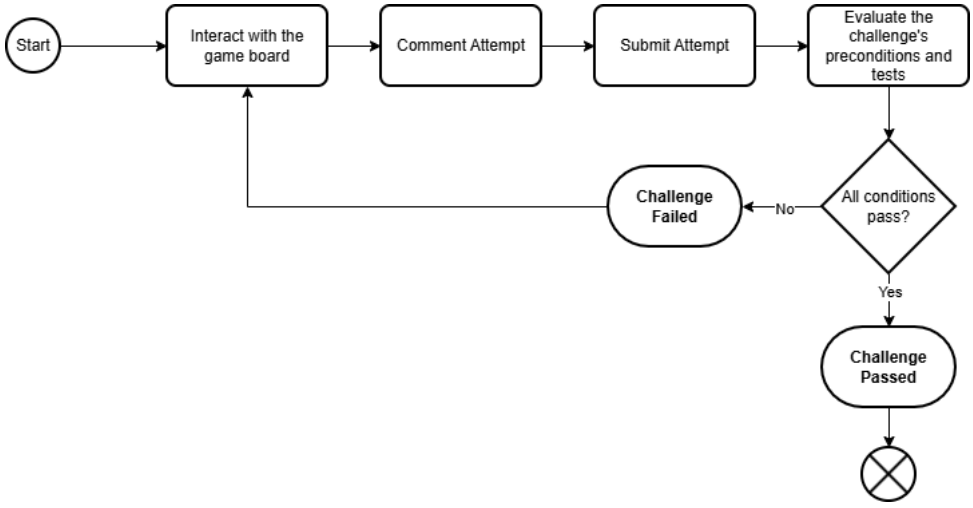

Figure 3.4: Previous game flow for a Single-player White-box Testing Challenge

3.4.2 Updated Architecture

The main contributions of this dissertation were focused on the frontend. Although we still maintained the core game-based elements and applications of gamification, as they had already proved a certain level of effectiveness, the main scoring logic had to be altered to give way to our novel approach.

Our contributions introduced three new components into the frontend:

- **Mutation Testing Challenges** - A new educational module, relying on new or existing code files, hosting a variety of mutation testing challenges. It would invoke the execution engine twice per player submission: once for the original function and once for the mutant. After this step, it would compare the outputs of each execution, passing if they are different for the same given input.
- **Code Execution Engine** - A Web Worker-based module, responsible for running JavaScript code asynchronously and returning the result of the execution. This module is common to both the coverage tracker and the mutation challenges, as they require code execution.
- **Coverage Goal Tracker** - A module that instruments challenge code before execution and, after the Worker returns the result, analyses the resulting coverage map to determine which of the challenge's coverage targets have been satisfied.

Figure 3.5 presents the updated architecture of the GAMFLEW platform after the contributions of this dissertation.

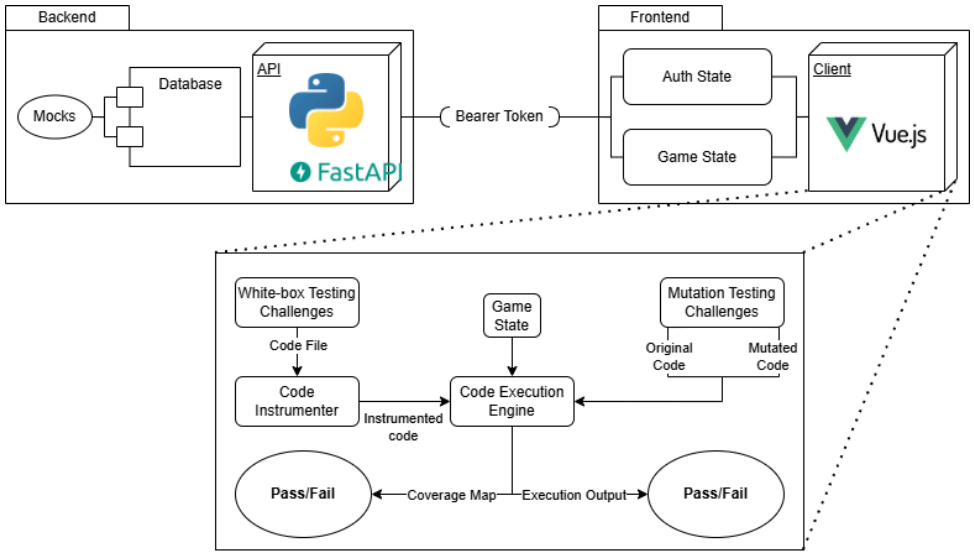

Figure 3.5: Updated GAMFLEW System Architecture

Despite sharing the same underlying execution model, the Coverage Goal Tracker and the Mutation Testing Challenges rely on **separate Web Worker instances** with diverging functionalities. For the Coverage Goal Tracker, the Web Worker has to return a coverage map which can

be analysed to determine the challenge's completion. On the other hand, in the Mutation Testing Challenges, the Web Worker must compare the output of two executions and return whether the mutant was killed or not. This separation reflects the different responsibilities of each module: the coverage Worker receives code that has already been instrumented by the main thread with tracking calls injected at the relevant statement, decision, or condition locations, returning the function's output and the resulting coverage map afterwards. The mutation Worker, by contrast, receives two code strings and is invoked twice per submission, once for the original code and once for the mutated code, and checks if they return the same output. If the functions output different results, the Web Worker informs the main thread that the mutant has been killed. Otherwise, it considers that the mutant has survived.

The three components share a common communication pattern with their respective Workers: the main thread serialises the code to be executed and the player's input, posts them to the Worker via `postMessage`, and registers a handler for the response. The Worker executes the code, collects any relevant output, and returns a structured result object. This communication pattern is illustrated in Figure 3.6.

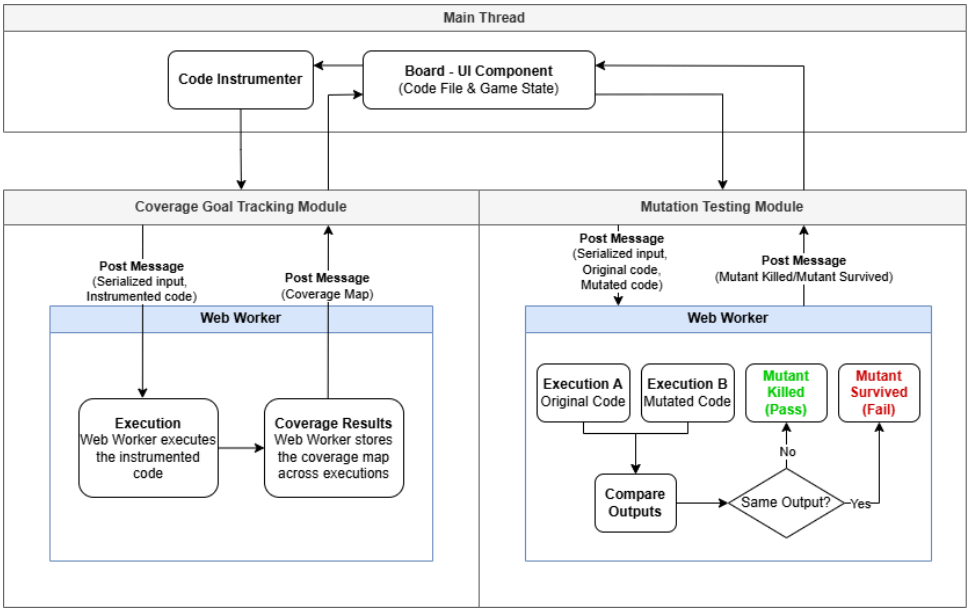

Figure 3.6: Communication sequence between the main thread and the Web Workers.

In both cases, the Worker operates as a stateless component: it receives a self-contained message, performs the execution, and terminates itself without retaining any state between submissions. This design ensures that each evaluation is independent of previous ones, which is particularly important for the coverage tracker, where the coverage map must reflect only the inputs submitted in the current attempt.

In the case of the white-box testing challenges, the game flow would not change drastically from the player's perspective, but it would be quite different internally. Rather than checking the challenge's sets of predefined Boolean conditions with the submitted board state, the platform now

instruments the challenge's code, sends it to the coverage Worker alongside the player's input, and evaluates the returned coverage map against the challenge's coverage objectives.

The backend's behaviour was not altered as drastically by our contributions, but the component was expanded with the capability of storing data related to the new mutation challenges, which we will discuss in Section 3.3.2. Crucially, the decision to perform all code execution on the client side means that the backend never directly interacts with the execution results.

The backend's main responsibilities continue to be the transfer of challenge data and the recording of player scores, but the evaluation logic itself is entirely contained within the frontend. This keeps the backend simpler and avoids introducing any complex infrastructure to the server-side, which would raise additional questions around security, scalability, and response time.

As mentioned before, from the player's perspective, the expected game flow is kept relatively unaltered, still involving the player choosing an input that may achieve the specified coverage goals and submitting an attempt with a comment until the challenge is passed. However, with these differences in implementation, the internal structure becomes significantly distinct from the previous approach. To better visualise the updated game flow, Figure 3.7 displays the updated game flow and how it differs from the original approach (found in Figure 3.4).

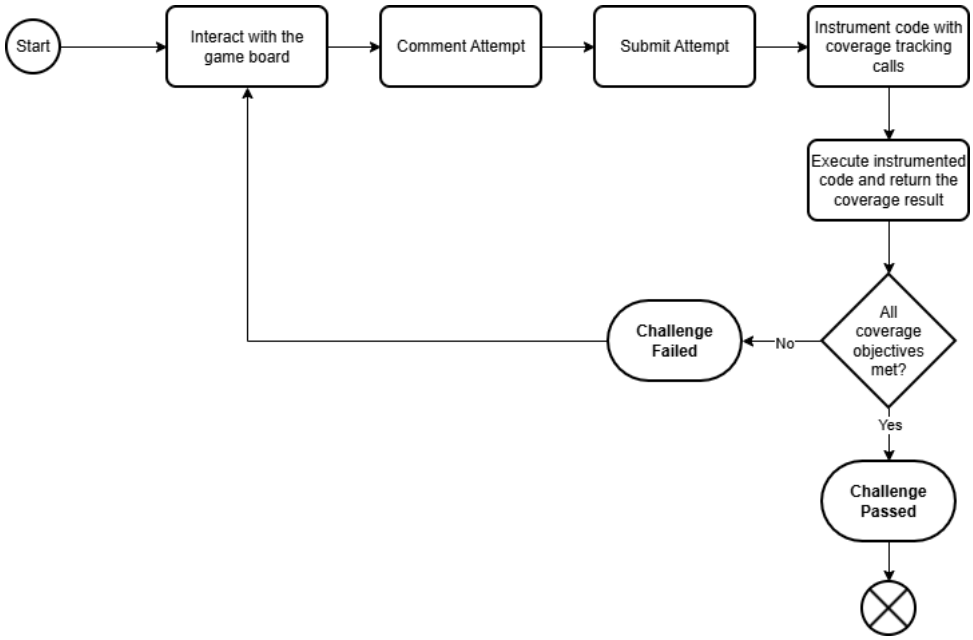

Figure 3.7: Updated game flow for a Single-player White-box Testing Challenge

3.5 User Interface

This section describes the user interface of the ***GAMFLEW*** platform, covering both the existing white-box testing challenges and the newly introduced mutation testing challenges. While the core layout and game elements were preserved from the original approach, several adjustments were made to accommodate the new execution model and the requirements of mutation testing challenges.

3.5.1 White-box Testing Challenges

The interface for white-box testing challenges was kept largely unaltered, preserving the central game board, the code block displaying the challenge's function, and the player and challenge information. From the player's perspective, the experience remains unchanged: they interact with the board, review the code, and submit an attempt with a comment.

The main visible difference introduced by our contributions is the removal of the **Condition Table** component and its associated button, which had previously allowed players to consult a truth table for a condition during the challenge. This component was removed due to its narrow application, with the intent of simplifying the interface. Additionally, the **Next** and **Previous** navigation buttons, used to cycle between boards in challenges containing multiple test cases, were removed from challenges with only a single board, where their presence was unnecessary.

Figure 3.1, presented previously in Section 3.4.1, illustrates the list of existing white-box testing challenges, remaining as a valid representation of the current state of the white-box testing module. Figure 3.8 and Figure 3.9 display, respectively, how the original interface of these challenges was originally organised and how the updated organisation differs from the original approach, according to the changes previously listed.

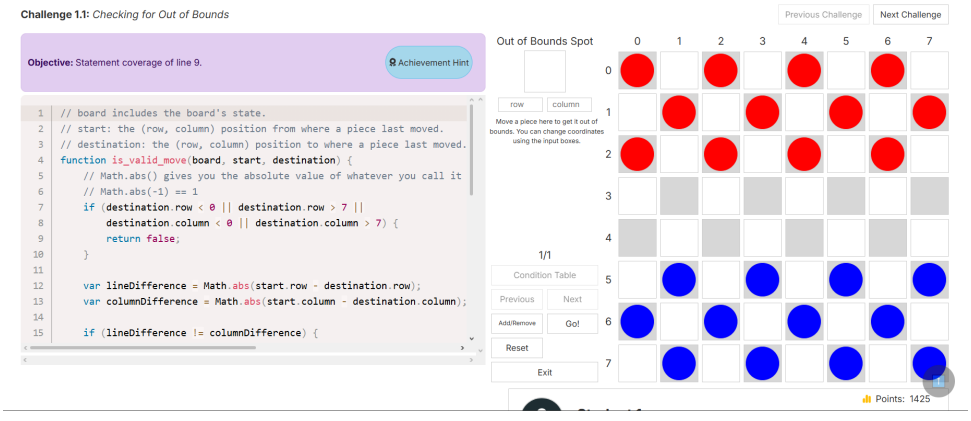

Figure 3.8: Original Challenge Board - Challenge 1.1

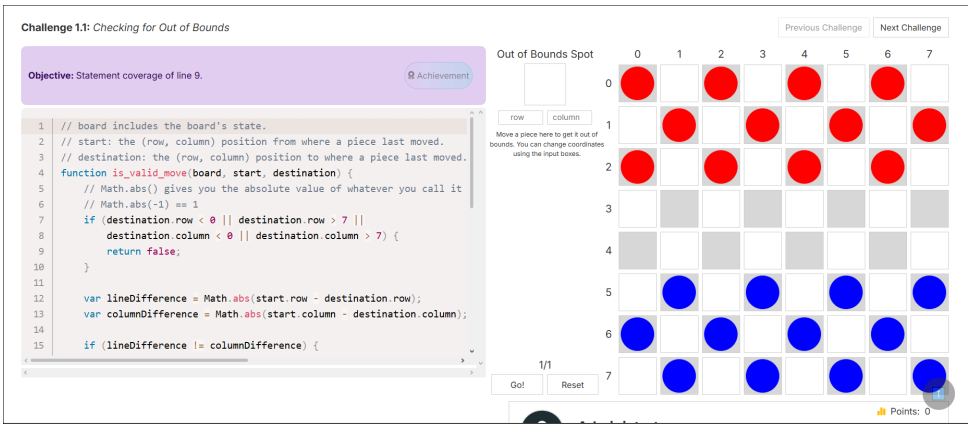

Figure 3.9: Updated Challenge Board - Challenge 1.1

3.5.2 Mutation Testing Challenges

The mutation testing module required a more significant alteration to the user interface, as it introduces a new type of challenge. Rather than interacting with a single code block, the player must be able to compare an original function with a mutated version of it, allowing them to reason about its behavior.

To support this, a **toggle button** was added above the existing code block, allowing the player to switch between the original and the mutated version of the function. The mutated line is visually highlighted, drawing the player's attention to it without requiring an independent comparison of the two versions.

For targeting **Equivalent Mutant** challenges specifically, the interface was extended with a dedicated button that allows the player to explicitly mark the presented mutant as equivalent. Although present in the interface of all new mutation challenges, it only becomes relevant when the player has been unable to find a killing input and suspects the mutant is equivalent. The feedback messages displayed upon submission were also adapted to reflect the mutation testing context, distinguishing between a killed mutant, a surviving mutant, and an explicit equivalence marking.

Two screenshots of the Mutation Testing challenges' interface can also be found later within Section 3.9 in Figures 3.11 and 3.12.

3.6 Technology Stack

The GAMFLEW platform is implemented as a web-based application using **Vue.js**, the JavaScript framework well suited to component-based UI development. Vue.js was maintained as the framework used by the pre-existing platform, and the contributions of this dissertation were implemented within the same codebase, extending the existing components without introducing a new framework or build pipeline.

Challenge code (also named "Code Files") - both for white-box testing and for mutation testing - is written in plain **JavaScript**, which is the language executed by the Web Worker at runtime. This Web Worker makes use of specific components from the ***Jasmine* Testing Framework for JavaScript** to execute the given code files. This choice is not only consistent with the existing challenge model in ***GAMFLEW***, but also an approach that would prevent drastic changes to the preexisting challenges.

The **Web Workers API**, available natively in all modern browsers, is used to achieve asynchronous, isolated code execution without any server-side implementations. No third-party library is required for this purpose, as the API is part of the standard browser environment. For each challenge, the client can create a Web Worker, provide it with the code file and input data, awaiting afterwards for the results of the execution.

Code instrumentation for coverage analysis is performed using a combination of **regular expression matching** and **string manipulation**, in accordance with the target line or line range, predetermined in each challenge. This is applied directly to the source code of each challenge before it is sent to the Web Worker. A dedicated instrumentation library or AST-based parser was considered but not used, as the structured and constrained nature of the challenge code (short, single-function snippets) made a regex-based approach sufficient and easier to implement within the existing codebase, without adding more complexity to the platform.

3.7 Code Execution Engine

The code execution engine is the foundational component introduced by this dissertation. Its purpose is to execute a JavaScript function with a given input, entirely within the browser, and return the result to the calling component in the Vue.js application.

When an attempt is submitted, the code execution is performed inside a **Web Worker**, a browser API that runs a script in a background thread, separate from the main UI thread. This provides the platform with two new main advantages:

- *Asynchronous Execution*, since the Worker runs concurrently with the UI and does not block user interactions while code is executing;
- *Isolation of Execution*, since the Worker has its own execution context and cannot access or modify the Vue.js application state.

The communication between the main thread and the Worker follows a simple message protocol. When a player submits an attempt, the calling component serialises the function code and the player's input into a JSON message object and dispatches it to the Worker via `postMessage`. After executing the function, the Web Worker posts the result back to the main thread via a second `postMessage` call. The main thread registers a one-time `onmessage` handler to receive the result and uses it to determine whether the challenge passed or failed. This behavior is demonstrated across two variations in Figure 3.6, found previously in Section 3.4.2. The final evaluation

depends on whether the challenge addresses code coverage or mutation testing, but both will be explained in the following sections 3.8 and 3.9.

The engine also handles execution failures. If the player's input causes a runtime exception inside the Worker, the error is caught, serialised, and returned to the main thread as part of the result object, allowing the UI to display an error message without crashing the application. To guard against infinite loops or other non-terminating computations, the Web Worker API allows for the implementation of a **timeout** mechanism: if the Worker does not post a response within a defined time threshold, the main thread calls `Worker.terminate()` and treats the submission as a failed execution.

As proposed, the Code Execution Engine would have three main responsibilities: it must be able to compile the given code alongside its dependencies (auxiliary functions and classes), execute the compiled function, and store its output across executions, since some challenges may require the code to be executed with more than one input. Therefore, when it receives a message, the Code Execution Engine compiles the function code, executes it, stores its output(s), and finally returns the result (a JavaScript object).

To more easily visualise how the Code Execution Engine works internally, Figure 3.10 presents a diagram explaining the data flow within the code execution engine after it receives an attempt.

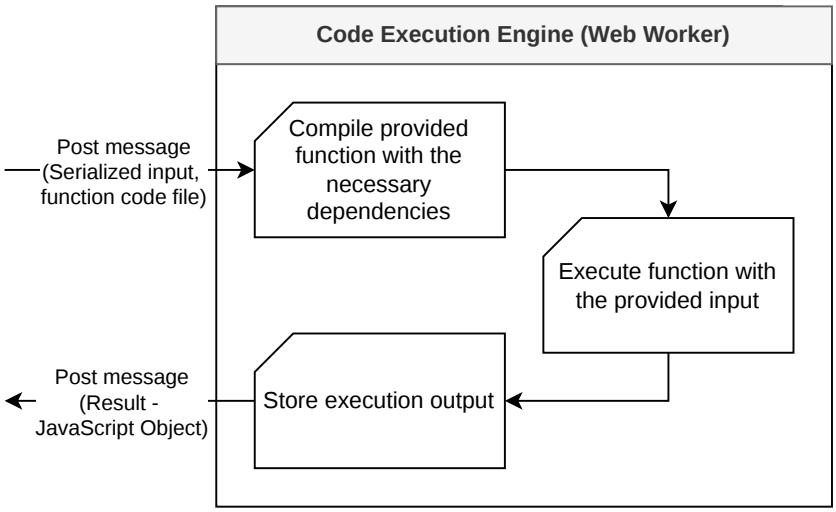

Figure 3.10: Diagram of the Internal Operations of the Code Execution Engine

3.8 Coverage Goal Tracker

The coverage goal tracker is another layer built on top of the execution engine and is responsible for determining, after a player's submission, which of a challenge's coverage goals have been achieved. This would be the central component that replaces the previous predefined Boolean condition model with an execution-based evaluation.

In order to track specific coverage goals, we needed an approach that could detect the relevant statements, decisions and/or conditions of each challenge and inject coverage tracking calls in the code, saving the results in a JavaScript object that could then be analysed by the main UI thread.

This mechanism is the first step towards achieving our proposed solution for challenge evaluation. The approach used to develop it can be found in the following subsection:

3.8.1 Code Instrumentation

Before a challenge's code is sent to the Web Worker for execution, it is **instrumented**: tracking calls are injected into the source code at the relevant locations regarding the coverage criteria specified by the challenge type and line range. This instrumentation is in the browser at runtime, using regular expressions to identify the relevant syntactic constructs and string manipulation to add the tracking calls where needed.

The following constructs are targeted by the instrumentation:

- **Return statements** - a tracking call is injected before each `return` statement, recording that the statement was reached during execution.
- **If and else branches** - a tracking call is injected at the entry of the `if` block and, where present, at the entry of the `else` block, recording which branch was taken. This still applies to branches with multiple else statements (`if, else if, else`, for example).
- **For and While loops** - a tracking call is injected inside the loop body, recording whether the loop was executed at least once, which distinguishes the case where the loop condition is never true from the case where it is.

Each tracking call records the identifier of the construct and, in some cases (Condition, Decision or MC/DC challenges), its value during execution into a **coverage map**: a plain JavaScript object maintained in the Worker's execution context. After the function finishes executing, the Worker includes this map in the result object returned to the main thread.

3.8.2 Coverage Criteria

The instrumentation described above supports the existing five challenge types and their coverage criteria, but produces a slightly different coverage map for each challenge type. These are as follows:

- **Statement Coverage** - a statement is considered covered if its associated tracking call was reached during execution.
- **Decision Coverage** - a decision (e.g. an `if` condition) is considered fully covered if both its `true` branch and its `false` branch were reached at least once in the player submission.

- **Condition Coverage** - each atomic boolean sub-expression within a compound condition must evaluate to both `true` and `false` across the set of boards submitted by the player for that challenge.
- **Condition/Decision Coverage** - a combination of condition and decision coverage, where both the decision and its sub-conditions are evaluated to both `true` and `false` with the different boards submitted by the player.
- **Modified Condition/Decision Coverage (MC/DC)** - each atomic Boolean sub-condition within a compound condition must be shown to independently influence the outcome of the overall decision. This requires the player to provide a set of inputs in which, for each sub-condition, two (or more) executions differ only in the value of that sub-condition but still produce a different outcome for the outer decision, while all other sub-conditions remain unchanged.

3.8.3 Mapping to Challenge Outcomes

Each challenge in GAMFLEW will produce a slightly different coverage map, according to its type (explored above). For **Statement challenges**, the coverage map will only record whether the target statement was covered, regardless of its value. For **Decision challenges**, the coverage map has to record whether the target decision had all its branches satisfied. In practice, this means that the coverage map will register the decision's boolean value, with the challenge passing if the decision was `true` and `false` at least once in the execution(s). **Condition challenges** work similarly, but instead, the coverage map registers the Boolean value of all the sub-conditions within a target decision. If all sub-conditions of the target decision produce a boolean value that is `true` and `false` at least once across the execution(s), then the challenge is passed. This means that the Boolean value of the target decision will be ignored, focusing only on its smaller tuples. Finally, **Modified Condition/Decision Coverage (MC/DC)** challenges record whether specific sub-conditions of the target decision cause the outer decision to change its Boolean value. For example, for the condition tuple (A && B), the coverage map would have to register whether the condition becomes false when A and B are both independently false.

In summary, if all targets are satisfied and covered according to the specified coverage criteria, the challenge passes; otherwise, the player receives feedback and is prompted to submit another attempt.

This mapping preserves full compatibility with the existing challenge model: the identifiers used in coverage targets correspond to the same logic that was previously modelled as Boolean conditions, so existing challenges required only slight alterations to the associated code files, but not a complete redesign of its structure.

3.9 Mutation Challenges

Although the two previous components were crucial, the main focus of this dissertation was to integrate a new Mutation Testing module into GAMFLEW, designed to teach players how mutations affect program behaviour, thereby achieving a better grasp of the topic. All challenges in this module were created manually, with each mutant crafted specifically for each function used in each challenge.

Due to inevitable differences between Coverage Analysis and Mutation Testing, the logic behind the proposed challenges would have to be altered accordingly. In practice, this would imply changes to both the external components, like the UI, Game board and Code Block, but also to the internal structure that determines whether a challenge is passed or not.

Starting with the UI, the proposed challenges would require the player to see both the original code and the mutated code. Adding to this, the UI would have to display not only the line containing the mutant, but also its type (AOR, ROR, LCR). To tackle these requirements, a button that could switch between the Original Code and the Mutated Code was added on top of the existing Code Block, allowing players to quickly visualise the difference through the highlighted lines. The messages displayed both in case of a failing challenge or an error detection were also slightly altered to correspond more correctly to the topic being addressed.

Moving on to the internal mechanism that determines a challenge's completion, we decided to leverage the previous Code Execution Engine component, fitting it with the capability of receiving and executing both the original code and the mutated code. Additionally, it would also compare the output of both executions, returning a result to the main thread indicating whether the mutant was killed or not.

You can see these changes directly reflected in the updated challenge UI for Mutation Testing challenges, presented below in the Figures 3.11 and 3.12

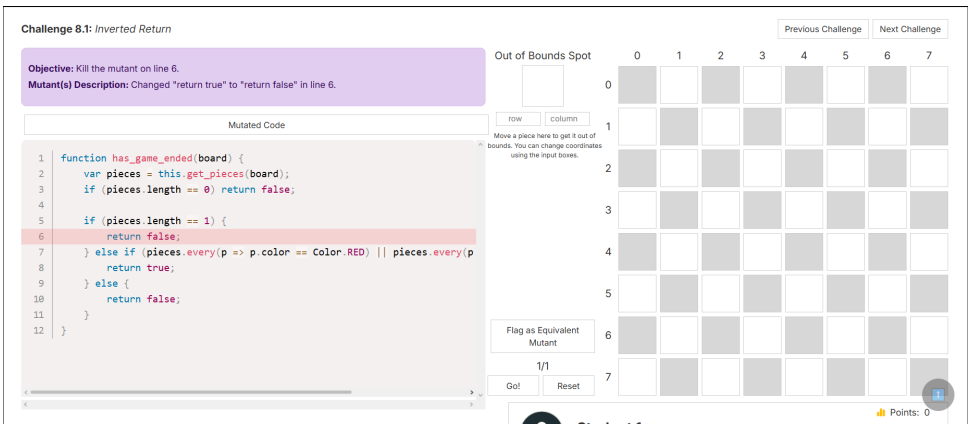

Figure 3.11: Mutation Challenge Board with the Mutated Code displayed - Challenge 8.1

3.9.1 Mutation Operators

Three classes of mutation operators were applied in the design of the mutation testing challenges:

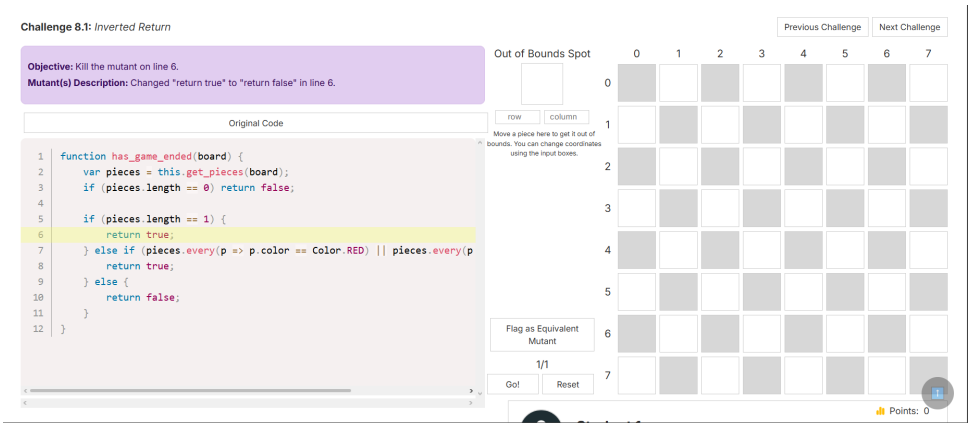

Figure 3.12: Mutation Challenge Board with the Original Code displayed - Challenge 8.1

- **Arithmetic Operator Replacement (AOR).** An arithmetic operator is replaced by a different arithmetic operator. Typical substitutions include replacing addition (+) with subtraction (−), or multiplication ($\star$) with division (/). These mutations are generally straightforward to kill, as they directly alter the computed value of an expression.
- **Relational Operator Replacement (ROR).** A relational operator is replaced by a different relational operator. For example, a less-than comparison (<) may be replaced by a less-than-or-equal comparison (<=), or an equality check (===) may be replaced by an inequality check (!==). These mutations are particularly useful for teaching boundary value analysis.
- **Logical Connector Replacement (LCR).** A logical operator is replaced by a different logical operator. The most common substitution is replacing a logical AND (&&) with a logical OR (||), or vice versa. LCR mutations are generally more challenging to solve, as they alter the value of a compound condition with multiple variations, which can be hard to identify.

As mentioned before, the mutated operator and line are visually highlighted in the interface to draw the player's attention to the modification, without requiring them to compare both versions of the code and allowing them to independently find the mutant.

3.9.2 Equivalent Mutants

As we have discussed previously, the **Equivalent Mutant** problem remains a crucial but underexplored obstacle within the Mutation Testing landscape. To address this topic and hopefully teach players about it, a small subset of the mutation testing challenges present to the player an **equivalent mutant**, which, despite the syntactical difference from the original function, produces the same output for every possible input. Because no killing test case exists, the player cannot simply complete such a challenge by providing an input in the otherwise common way.

It should also be noted that the player is not informed in advance whether a challenge contains an equivalent mutant. This is an intentional decision, which forces the player to develop the

ability to detect whether a given mutation can affect the program's observable behaviour, rather than simply trying to find a killing input by trial and error.

With this in mind, the flow for equivalent mutant challenges is as follows: The player attempts to submit an input in the same way they would in a standard *Kill the Mutant* challenge. For each attempt, the execution engine runs both the original and the mutated function, comparing their outputs afterwards, but since the mutant is equivalent, the outputs will always match. During the gameplay, the player may click a dedicated button to **mark the mutant as equivalent**. If the challenge's mutant is flagged as equivalent in the system, the submission is accepted, and the challenge is passed. If not, the submission is rejected, prompting the player to try again.

3.10 Additional Contributions

Beyond the two main contributions described in the preceding sections, several secondary improvements were made to the GAMFLEW platform during the development of this dissertation. While not as crucial to the main contributions, these still helped improve the usability of the platform, while also simplifying the player experience and the internal system of the game.

3.10.1 Color Cycling System

Before this work, adding or removing pieces from the game board required the player to explicitly toggle a secondary interaction mode, referred to as the **add/remove mode**. This mode had to be toggled on and off via a dedicated button, which introduced unnecessary complexity into the game flow, particularly if players needed to both move pieces and construct specific board configurations in quick succession.

The color cycling system replaces this interaction with a more intuitive approach that doesn't rely on a toggleable mode. Instead, clicking on an **empty cell** now places a red piece on it. Once a piece is placed, it can be selected as usual, but if selected, a **second click** on the same piece advances it through a fixed cycle of states, rather than simply deselecting it. The full cycle is as follows:

$$\texttt{EMPTY} \rightarrow \texttt{RED} \rightarrow \texttt{SELECTED} \rightarrow \texttt{BLUE} \rightarrow \texttt{SELECTED} \rightarrow \texttt{EMPTY}$$

This design allows players to place, change, and remove pieces from any cell through a single, consistent interaction, without needing to activate a separate mode on the board. The result is a more fluid game flow that facilitates board manipulation, particularly in challenges that require specific or "illegal" board configurations.

3.10.2 Interface Modifications

Several interface elements were also removed during the development of this dissertation due to redundancy, unnecessary visual complexity, or simply because they didn't provide a significant benefit to the player experience.

Condition Table. The Condition Table was a component that displayed a truth table for a condition within the challenge's function. It was accessible via a dedicated button and was intended to assist players in analysing complex Boolean expressions, mostly in Decision, Condition and MC/DC challenges. This component, along with its associated button, was removed due to its narrow focus and application, helping simplify the already extensive User Interface.

Next and Previous Buttons. In challenges containing only a single board, the **Next** and **Previous** navigation buttons previously remained disabled but visible, despite serving no functional purpose. These buttons have been removed from challenges with only one board and from the mutation challenges, reducing the interface's complexity and avoiding potential confusion for players.

Achievement Hint Button. Mutation testing challenges do not have associated achievements in the current version of the platform. As a result, the **Achievement Hint** button, which provides players with a hint towards unlocking an achievement in white-box testing challenges, was removed from the mutation testing challenge interface, providing more space for the challenge information component instead.

Chapter 4

Validation

This chapter describes the evaluation study conducted to assess the contributions of this dissertation. To validate ***GAMFLEW***, we conducted two distinct experiments.

In the first experiment, we manually evaluated each challenge, existing and new, with inputs that either passed or failed the challenges, ensuring the intended functionality of each challenge type for both correct and incorrect attempts.

In the second experiment, we recruited a set of participants to test and evaluate the platform by answering a set of questions prior to and after interacting with the game's challenges. In the following sections, we will describe and discuss the methodology, participant profiles, and limitations of this study, concluding with an analysis of the relevant results of this survey.

4.1 Manual Validation

Before conducting the practical evaluation with participants, a manual validation of all challenges was carried out. The goal of this step was to verify the functional correctness of each challenge, ensuring that the platform's evaluation model consistently accepts valid solutions and rejects invalid ones before exposing the challenges to external participants.

4.1.1 Methodology

Each challenge was tested individually, following a structured procedure applied across all challenge types. For every challenge, at least two test cases were defined and submitted:

- **Passing test case:** a correct input that satisfies the challenge's objective, expected to be accepted by the platform.
- **Failing test case:** an incorrect input that does not satisfy the challenge's objective, expected to be rejected by the platform.

For each challenge, the following attributes were recorded: the challenge name, its type, its difficulty, the passing and failing test cases used, and any additional notes relevant to the challenge's behaviour or edge cases.

This process was applied to all challenges available in ***GAMFLEW***, covering both the pre-existing white-box coverage challenges and the newly introduced mutation testing challenges. In total, 108 challenges (94 white-box challenges and 14 mutation challenges) were validated and distributed across the challenge types.

4.1.2 Results

All challenges listed were confirmed to accept the defined passing test case and reject the defined failing test case, demonstrating the correct behaviour of the platform's evaluation engine across all available challenge types and difficulties.

4.1.2.1 White-Box Coverage Challenges

The pre-existing white-box coverage challenges had already been validated by Silva et al. [21]. Although these challenges had already demonstrated their correctness in the original study, they were re-validated in this step to ensure that the integration of the new mutation testing module and the changes to the evaluation model did not introduce any errors or inconsistencies in the platform's existing behavior.

4.1.2.2 Mutation Testing Challenges

The mutation testing challenges, which represent the main contribution of this dissertation, were subject to the same level of validation as the white-box coverage challenges. The two challenge sub-types were validated as follows:

- **Kill the Mutant:** The player must provide an input that produces different outputs for the original program and the mutant. The passing test case was defined as an input that successfully distinguishes the mutant's behavior from the original; the failing test case was an input that produced identical outputs in both versions.
- **Equivalent Mutant:** The player must correctly classify whether a mutant is equivalent or non-equivalent to the original program. In this case, the passing test case corresponded to the player marking the mutant as equivalent; the failing test case corresponded to the player submitting any other input while not marking the mutant as equivalent.

4.1.3 Full Validation Record

The complete record of the manual validation, including the name, type, difficulty, passing test case, failing test case, and additional notes for each of the validated challenges were compiled in a structured spreadsheet format using Google Sheets. Figures 4.1 and 4.2 demonstrate excerpts from the spreadsheet, giving a general idea of how the validation was conducted.

GAMFLEW — Coverage Challenges: Manual Test Cases						
#	Challenge Name	Type	Difficulty	Passing Test Case (board state description)	Failing Test Case (board state description)	Notes
2.4	Matters	MC/DC	Normal	more than 12 blue pieces; Board 3: Use more than 12 red pieces.	Alternatively if all boards have more than 12 red/blue pieces, it will cause the same result.	
2.5	Win Conditions, Only When It Matters	MC/DC	Normal	Board 1: Use a valid number of red and blue pieces, not exceeding 12 of each; Board 2: Use a board with only red pieces; Board 3: Use a board with only blue pieces.	Use three valid boards with a valid number of red and blue pieces, not exceeding 12 of each. Alternatively if all boards have no red or blue pieces, it will cause the same result.	
2.6	Win Conditions, But Thoroughly	Condition/Decision	Normal	Board 1: Use an empty board; Board 2: Use a valid number of red and blue pieces, having at least 1 but not exceeding 12 of each.	Use two valid boards with a valid number of red and blue pieces, not exceeding 12 of each. Alternatively both boards with only one colored pieces will cause the same result.	
2.7	Counting Pieces, But Thoroughly	Condition/Decision	Normal	Board 1: Use more than 12 blue pieces and more than 12 red pieces; Board 2: Use a valid number of red and blue pieces, not exceeding 12 of each.	Use two valid boards with a valid number of red and blue pieces, not exceeding 12 of each. Alternatively two empty boards will cause the same result.	
2.8	Counting Pieces On Some Restrictions	Condition	Easy	Board 1: Use more than 12 blue pieces and more than 12 red pieces; Board 2: Use a valid number of red and blue pieces, not exceeding 12 of each.	Use two valid boards with a valid number of red and blue pieces, not exceeding 12 of each. Alternatively two empty boards will cause the same result.	The passing case can also work if each board only more than 12 red pieces or only more than 12 blue pieces, or vice-versa. In both situations each sub-conditions is evaluated to true and false at least once.
2.9	Winning Conditions On Some Restrictions	Condition	Easy	Board 1: Use at least one, but not more than 12 red pieces; Board 2: Use at least one, but not more than 12 blue pieces.	Use two valid boards with a valid number of red and blue pieces, not exceeding 12 of each. Alternatively two empty boards will cause the same result.	The passing case can also work if the first board has at least 1 but no more than 12 of each piece, as long as the second board is empty.
2.10	Valid Positions for Odd Rows	Statement	Normal	Place a piece on an "illegal" position. Place any piece in a cell where its row is odd and column is even.	Use an empty board, an invalid board with more than 12 pieces of each color or a valid board with all pieces in "legal" positions (row and column both even or both odd).	In checkers, a legal position means that both its row and column will have equal parity (both are even or both are odd).
2.11	Valid Positions for Even Rows	Statement	Normal	Place a piece on an "illegal" position. Place any piece in a cell where its row is even and column is odd.	Use an empty board, an invalid board with more than 12 pieces of each color or a valid board with all pieces in "legal" positions (row and column both even or both odd).	In checkers, a legal position means that both its row and column will have equal parity (both are even or both are odd).

Figure 4.1: Excerpt of the manual validation spreadsheet for Coverage Challenges

Mutation Challenges: Manual Test Cases								
#	Challenge Name	Operator Type	Difficulty	Original Code (shortened)	Mutated Code (shortened)	Killing Input (board state description)	Non-Killing Input (board state description)	Notes
8.2	Impossible Condition?	LCR	Normal	`else if (pieces.every(p => p.color == Color.RED) || pieces.every(p => p.color == Color.BLUE))`	`else if (pieces.every(p => p.color == Color.RED) && pieces.every(p => p.color == Color.BLUE))`	Place 2 or more pieces all of the same color. Original returns true; Mutant always returns false.	Use an empty board or only 1 piece. Both the original and mutated code return the same result.	Since a piece cannot be both red and blue, the mutant is in a mutually exclusive condition. This means that, in practice, the mutated condition is an unreachable branch and will always be false.
8.3	Breaking Conditional Boundaries	ROR	Normal	`if (pieces.length == 1) { return true; }`	`if (pieces.length != 1) { return true; }`	Place 2 pieces of different colors on the board. Original returns false; Mutant returns true	Use an empty board, only 1 piece or multiple pieces of the same color. Both the original and mutated code return the same result under these conditions.	
8.4	Breaking Conditional Boundaries... Or Not!	ROR	Hard	`if (pieces.length == 0) return false;`	`if (pieces.length != 0) return false;`	Place 2 or more pieces all of the same color. Original returns true; Mutant returns false.	Place 2 or more pieces of different colors on the board. Both will return false.	
8.5	Tricky Conditions Pt. 1	ROR	Hard	`if (pieces.length == 0) return false;`	`if (pieces.length <= 0) return false;`	EQUIVALENT MUTANT.	The outputs will always be the same.	pieces.length is always ≥ 0 in JavaScript (array length cannot be negative), therefore the branches are identical.

Figure 4.2: Excerpt of the manual validation spreadsheet for Mutation Challenges

4.2 Practical Validation

This evaluation was designed to address two main goals: assessing the **usability** of the extended ***GAMFLEW*** platform, and assessing the **perceived effectiveness** of the platform as an educational tool for Mutation Testing with real users.

4.2.1 Evaluation Methodology

Given the nature of the contributions, a quantitative evaluation based only on knowledge gain would be insufficient. The platform's value is derived from the players' ability to learn Mutation Testing concepts, an easy interface to navigate, the challenges' capacity to motivate students, and whether the overall experience is coherent. With this in mind, the evaluation relies on two questionnaires, one being answered before and the other after the use of the platform. The questionnaires used in this work were developed under the European project *ENACTEST* (https://enactest-project.eu/) [14, 15, 20] and covered four evaluation parameters: usability, motivation, perceived learning, and clarity of the platform's content. These questionnaires can be found in the project's repository by following the link: https://zenodo.org/records/15534588.

The study follows a **within-subjects design**: each participant uses the platform and completes both a pre-use and a post-use questionnaire. Therefore, no control group was required, as we weren't aiming to compare GAMFLEW against an alternative. With these questionnaires, we could assess the platform's effectiveness and usability across a population of participants.

The study was conducted in a **hybrid format**: some participants completed an in-person session under supervision, while the remaining ones participated remotely, accessing the platform and the questionnaires independently via a shared file and link, respectively.

The in-person sessions allowed for direct observation of participant behaviour and collection of relevant metrics directly. While the remote session lacked the benefit of direct supervision, it still helped us reach a wider population.

4.2.2 Participants

Participants were recruited from three main groups: **university students** with exposure to software engineering or programming courses, **software professionals** with industry experience, and **individuals with prior programming knowledge** who are not currently enrolled in any computer science course. This diversity allowed the evaluation to assess whether the platform is accessible and useful across different levels of prior knowledge. „ Participants were previously informed of the purpose of the study and the nature of the tasks involved in this survey. No personally identifiable information was collected beyond the age range, gender (with the option to not disclose), participant group (student, professional, or other) and their level of familiarity with software testing and mutation testing.

4.2.3 Procedure

The study procedure was structured in six sequential steps, described below in order of execution. Although the study was conducted in a hybrid format, both variations followed the same procedure, with the only difference being the level of supervision available during the session.

4.2.3.1 Step 1: Pre-use questionnaire (5-10 minutes)

Before interacting with the platform, participants completed a short questionnaire collecting their background information and familiarity with software testing, coverage analysis, and mutation testing. This step helped characterise the participants and establish a guide for the post-use responses to be compared. The following information was collected through this initial questionnaire:

- Personal Information
  - Age range
  - Gender (If the participant wishes to disclose it)
  - Qualification(s) in the area of Software Engineering - Student/Professional/Other
  - Years of experience with Programming
- Technical Skills
  - Level of familiarity with Software Testing in general
  - Level of familiarity with specific contents within White-box Testing, like coverage analysis.
  - Level of familiarity with Mutation Testing and its relevant concepts, like the Equivalent Mutant Problem
- Others (Optional)
  - Previous use of gamification/serious games in a teaching context
  - Perceived effectiveness of the previous application of gamification/serious games

4.2.3.2 Step 2: Introduction to Code Coverage and Mutation Testing (10 minutes)

Before introducing the players to the platform, they were briefly presented with an introduction to the relevant concepts, in this case, Code Coverage and Mutation Testing. This introduction relied on both an informal presentation about Mutation Testing and a section within the "How to Play" page on the platform, where the different code coverage criteria are explained, focusing on the challenge types previously implemented.

4.2.3.3 Step 3: Installing *GAMFLEW* (10-15 minutes)

For this experiment, the participants were provided with a zip file containing a Dockerized version of the platform. Because of this dependency, this step also required the users to install the Docker Desktop app. Thankfully, possibly due to the participants' experience in Software Engineering, most already had the app installed prior to the experiment, but this step allocated enough time to install the auxiliary app. This had the added benefit of allowing us to check each participant's number of successful, failed, and total attempts, as this information is directly tracked within the platform for each user.

4.2.3.4 Step 4: Platform introduction (5-10 minutes)

Participants were instructed to explore the GAMFLEW platform, starting with the "How to Play" page, which explained the average game flow of the challenges available. No detailed instructions on how to solve the challenges were given, as we also aimed to evaluate the platform's ability to guide players across their interactions, without relying on external influence.

4.2.3.5 Step 5: Platform use. (15-20 minutes)

After the previous introductions, participants were now allowed to interact with the platform freely, attempting to complete a set of ten challenges (five white-box testing, one of each coverage type, and five mutation testing challenges). The set of challenges was meant to be completed within a single session of approximately 15-20 minutes, covering a representative range of difficulty levels and challenge types. As we have discussed before, relevant information like the number of successful, failed, and total attempts could be gathered at the end of this step since it is tracked by the platform.

4.2.3.6 Step 6: Post-use questionnaire (10-15 minutes)

After completing the previous step, participants completed a second questionnaire assessing their experience with the platform regarding four main parameters: usability, motivation, perceived learning, and clarity of the platform's content. The full structure of this questionnaire is described in the following Section 4.2.4.2.

4.2.4 Metrics and Instruments

As previously demonstrated, this validation relied on two primary instruments: the pre-use questionnaire and the post-use questionnaire. Together, these instruments capture not only details about each participant, like their background, but also their experience with the platform, and their perception of its educational value.

In addition to the questionnaires, a collection of platform usage metrics was automatically collected in each participant's deployment of the platform, like the number of successful, failed, and total attempts, complementing the subjective responses given in the previous questionnaires.

This section describes each instrument in detail, explaining the specific information each section of questions is intended to gather.

4.2.4.1 Pre-Use Questionnaire

The pre-use questionnaire serves two distinct purposes. First, it collects **demographic and background information** that allows the participant population to be characterised for analysis. Second, it establishes a **baseline of prior knowledge** for the concepts addressed by the platform, which is crucial to contextualize the post-use responses. For example, a participant who reports high confidence in mutation testing after the session can mean different things, depending on whether they started the session with no previous knowledge or with years of experience. You can find the full questionnaire in the section A.1 in the Appendix.

The questionnaire is organised into three sections presented below:

Section 1 - Background The first section collects five pieces of background information from each participant: their age and gender (if they wish to disclose it), their current situation (university student, software professional, researcher, or other), their years of programming experience, and whether they have previously taken a formal or informal course on software testing.

Other than the **age and gender** variables, which serve only to gauge the diversity of the population, the other variables offer baselines of familiarity within the addressed topics. For example, the **current situation** variable (university student, software professional, researcher, or other) allows the results to be broken down by group. This enables comparisons between students who may have recently learned the testing concepts in a course and professionals who apply them more frequently in the industry context. These two groups may have different initial levels of familiarity, different expectations for a learning tool, and even different reactions to the gamification elements of the platform.

Adding to this, the **years of programming experience** variable provides a more quantifiable measure of technical background than the current situation alone. A student in their final year of a software engineering degree most likely will have more practical programming experience than a first-year computer science student. Grouping by experience rather than simply by affiliation can reveal details that would otherwise be missed.

Finally, the **prior software testing course** variable is relevant for interpreting the results on the learnability and perceived learning during the post-use questionnaire. A participant who has studied software testing formally will most likely find the platform's interface, mechanics, and addressed concepts more familiar than someone who may have learned the same topics informally. Potentially, the first participant may rate the platform's learnability with a higher value, but the perceived learning with a lower one, since they may have less unknown context to explore.

Section 2 - Familiarity with Relevant Concepts The second section presents five concepts and asks participants to rate their familiarity with each on a five-point Likert scale, from *No familiarity*

(1) to *Very familiar* (5). The five concepts addressed in the section are: software testing in general, white-box testing, code coverage criteria, mutation testing, and the equivalent mutant problem.

These five items are carefully ordered from the most general to the most specific, mirroring the organization of the topics addressed by the platform. Software testing is the broadest concept with the largest number of challenges; white-box testing is a subcategory within it; code coverage criteria is the central topic explored in GAMFLEW's existing challenges; mutation testing is the new topic introduced by this dissertation's contributions; and finally, the equivalent mutant problem is the most advanced and specific concept within mutation testing. This separation allows us to gauge how knowledgeable each participant is not only in software testing in general, but also in more domain-specific concepts within this area.

The responses to this section are therefore correlated with the post-use responses regarding perceived effectiveness and learnability, helping to determine whether prior familiarity influences the platform's perceived educational impact. For instance, one might expect participants with no prior knowledge of mutation testing to report a greater level of perceived learning after the session than those who were already familiar with the concept, assuming that the platform succeeds in introducing it efficiently.

Section 3 - Prior Experience with Serious Games The third section asks whether participants have previously used a serious game or gamified platform to learn a technical concept, and, if they answered yes, whether they found it helpful.

This means that a participant who has never used a serious game before and still rates the GAMFLEW experience as positive is providing a particularly strong argument for the platform's approachability. At the same time, a participant who has used similar tools before and still finds GAMFLEW engaging is providing meaningful evidence that its quality is relatively equal to or even better than the alternatives they have experienced. Both patterns are greatly helpful in different ways, and this background variable allows the analysis to distinguish between them.

4.2.4.2 Post-Use Questionnaire

The post-use questionnaire is the main source of direct feedback for this study. It is similar in structure, but larger in content, being organised into seven sections, each targeting a different aspect of the user experience, and concluding with an eighth section of open-ended questions. The questionnaire uses a combination of the same five-point Likert scale, multiple-choice questions, and text fields, particularly for the open-ended questions.

The seven aspects were chosen to cover a full analysis of the user experience in an educational serious game. These ranged from technical aspects of usability and learnability, levels of satisfaction and confidence, to perceived learning, and finally, its purposefulness as a vehicle for education. With this, we hoped to achieve a nuanced evaluation of the platform through a diverse set of perspectives.

You can find the full questionnaire in the section A.2 in the Appendix.

Section 1 - Usability The first section evaluates how accessible the platform was to use from a player's perspective, independently of the difficulty of the content it delivers. In this context, usability is the measure of how easily an user can achieve their intended actions without confusion or error.

The five questions in this section cover the intuitiveness of the interface, the ability to find information within the platform, the transparency of the layout and organisation, the clarity of feedback, and the presence of technical errors.

Any moderately negative feedback on this section could suggest the need for interface re-design, even if the new educational content didn't require any changes.

Section 2 - Learnability The second section assesses how quickly participants were able to become familiar with the platform and its mechanics independently. Learnability is, in this context, distinct from usability: a platform can be usable once learned but still have a steep learning curve, or it may be approachable but reveal usability issues over time. For a serious game designed to be used in educational settings, learnability is a particularly important consideration.

The four items in this section evaluate whether participants could understand the platform's mechanics without external influence, whether challenge objectives were clear from their descriptions alone, whether they felt comfortable exploring the platform, and whether the transitions between different challenges felt natural.

The last item is particularly important to this dissertation: one of our intentions for the platform is that the connection between coverage-based testing and mutation testing should be coherent and easily understood. If participants find the transition between challenge types complex or confusing, this suggests that this connection is not being communicated effectively through the platform's design.

Section 3 - Satisfaction and Comfort The third section addresses the user experience: how participants felt while using the platform. This dimension seemed important since it captures aspects of the experience outside the areas of usability or learnability. Nevertheless, this is still an element that can influence whether a user would return to the platform or recommend it to others.

The five questions cover the confidence in interacting with the platform, comfort in experimenting with inputs, overall satisfaction, ease of use of the game board and its mechanics, and enjoyment of the experience. Students should feel free to try different inputs without excessive hassle, learn from the feedback, and enjoy the overall experience, not being limited by the fear of making a wrong attempt.

If participants do not feel comfortable experimenting with the platform, this could suggest a limitation in the learning process that goes beyond interface design or technical implementations.

Section 4 - Difficulty and Complexity The fourth section gauges how difficult and complex the challenges are from the participant's perspective. This section is concerned particularly with the

cognitive difficulty of the challenges: how hard participants found it to reason about and solve the challenges, regardless of how easy the interface was to operate.

The first four questions assess how appropriate the platform's overall difficulty level is, how clear the challenge descriptions are, how difficult mutation testing challenges are compared to coverage challenges, and finally, how difficult Equivalent Mutant challenges are in comparison to the other challenge types.

The last question, regarding the Equivalent Mutant Problem, is particularly important since, as discussed in Chapter 3, this challenge type is intentionally more demanding: it requires the participant to think about the equivalent behavior of two functions, which is fundamentally different from other Mutation Challenges where the players must find an input that distinguishes them. High difficulty ratings are therefore expected and not necessarily problematic. However, if participants also consider the challenge unclear or frustrating, this could suggest a need to improve how the challenge type is contextualised within the platform.

Some questions in this questionnaire are worded in a reverse direction, where higher agreement corresponds to greater difficulty or confusion. This should, in theory, prevent the participants from answering with a positive bias, while also eliciting a more honest acknowledgement of problems.

Section 5 - Effectiveness, Productiveness, and Efficiency The fifth section assesses whether participants felt that the platform was effective as a learning tool, whether it helped them understand the addressed concepts, and whether it helped them gain new capabilities. Therefore, this section of the questionnaire measures learnability, even though this was conducted through a self-evaluation and not another objective approach.

The six questions in this section mostly cover understanding of mutants, the relationship between inputs and program behavior, perceived value of time spent, efficiency of the platform as a practice environment, and confidence in reasoning about mutation testing after the session.

One of the two main contributions of this dissertation is the replacement of GAMFLEW's Boolean condition evaluation model with real code execution and coverage analysis. The educational value of this contribution depends on whether the engine produces correct results, but also on whether the feedback it generates is perceived as useful by the learner. A negative response to questions regarding feedback would suggest that the engine may be failing to communicate its results in a pedagogically useful way.

The last question of this section also asks participants to assess their confidence in reasoning about Mutation Testing in general, which evaluates the effectiveness of the mutation testing challenges specifically. The question acts as the main indicator of the platform's educational impact on the participant's capability of reasoning about Mutation Testing.

Section 6 - Usefulness The sixth section checks if participants perceive GAMFLEW as a useful tool for its intended educational purpose.

The four questions are mostly focused on determining the usefulness of the platform for learning mutation testing, its suitability for new players and/or students recently introduced to the topic,

the willingness of the participants to use it as a study tool, and the probability of recommending it to others. These questions are particularly informative at the individual level: a participant who rates the platform as useful but who would not recommend it may have had a positive experience that they still consider too complex or challenging for a general audience.

The section also prompts the participants to give an overall rating of ***GAMFLEW***, using a five-point scale stretching from *Very poor* to *Excellent*. This was mostly meant to obtain a single general measure of participant satisfaction, allowing us to easily use it as an indicator of the platform's overall reception.

Section 7 - Serious Game Experience The seventh section addresses the game-based elements and experience of the platform, gauging if the gamification elements and game mechanics were positive motivators of the participant's engagement and learning.

The five questions cover the game elements used throughout the platform and its challenges (points, achievements, leaderboard) and their effects on engagement, sense of progress, and accomplishment.

The last of these five questions is arguably the most important in this section, since it asks participants directly whether the game format made mutation testing concepts easier to understand than a theoretical approach would have. A positive response would indicate a high educational value through the serious game approach, supporting the design decisions taken in this dissertation and the findings regarding serious games in Mutation Testing education, explored previously in Chapter 2.

Section 8 - Open-Ended Feedback The eighth and final section consists of four optional open-ended questions inviting participants to describe what they liked most about the platform, what they found most difficult, what they would change or improve, and any additional comments or suggestions they wish to offer.

While the previous sections mostly relied on quantitative items that allow the identification of general patterns and trends across the participant population, open-ended responses allow participants to give their thoughts, perspectives, and suggestions on the platform. This is something the questionnaire would otherwise not gauge without these questions. They also carry the added benefit of being more explicit when reporting problems or inconsistencies the participant may have detected. The questions are not specific to technical errors either, allowing the participants to report their feelings and emotional evaluation, or concrete improvements and suggestions that may shape the future work on the platform.

4.2.5 Platform Usage Metrics

In addition to the questionnaire responses, the following usage metrics were automatically collected by the platform during each participant's session or through a personal inquiry:

- **Challenge completion rate** - the proportion of assigned challenges completed by the participant within the session.
- **Successful attempt rate** - the proportion of successful attempts to total attempts submitted by the participants within the session.
- **Failed attempt rate** - the proportion of failed attempts to total attempts submitted by the participants within the session.

These metrics provide an important complement to the subjective questionnaire responses, allowing cross-validation between the perceived difficulty and the actual performance of each participant.

4.3 Results

This section will analyse and discuss the results and main findings of the previous experiment.

4.3.1 Participant Profile

The Practical Validation was conducted with 9 volunteers in total. The previously mentioned pre-use questionnaire revealed a demographic with the following characteristics:

- 8 participants were within the 18-25 age range. The remaining one was within the 26-39 age range.
- 6 participants identified themselves as Male. The remaining three identified as Female.
- 5 participants were students enrolled in Computer Science, Software Engineering, or another related area. Three of the remaining participants considered themselves industry professionals, while the final participant answered "Other".
- 7 participants had 3-5 years of programming experience. Of the two remaining participants, one had more than 5 years, and the other had 1-2 years.
- 8 participants had taken a formal software testing course during their education. The remaining one had not.

Adding to the personal attributes of the participants, the questionnaire also gathered their self-perceived technological skills with Software and Mutation Testing Concepts, asking the participants to rate them on a 1-5 scale. These results are demonstrated in Table 4.1, where the average reported familiarity level of each topic is displayed.

In general, this revealed a generally high level of familiarity with software testing, white-box testing, and code coverage concepts, with most questions regarding these areas being answered with at least a level 3. Despite this, the questionnaire still detected a slightly lower level of familiarity with Mutation Testing and its concepts, like the Equivalent Mutant problem, reinforcing our point regarding its teaching in Chapter 2.

Topic	Average Level of Familiarity
Software Testing	3.89
White-box Testing	3.78
Code Coverage	3.33
Mutation Testing	2.78
Equivalent Mutant Problem	2.56

Table 4.1: Participants' Average Level of Familiarity with the addressed areas

4.3.2 Platform Usage Metrics

As was mentioned previously in Section 4.2.5, ***GAMFLEW*** is capable of registering relevant information regarding player submission attempts within the platform. This information was gathered for each participant and documented in Table 4.2 alongside their average.

Participant	Successful Attempts	Failed Attempts	Total Attempts	Completed Challenges
A	7	15	22	7
B	10	9	19	10
C	8	13	21	8
D	10	7	17	10
E	8	12	20	7
F	6	16	22	6
G	6	18	24	6
H	9	9	18	8
I	11	10	21	9
Average	8.33	12.11	20.44	7.88

Table 4.2: Number of completed challenges and of successful, failed, and total attempts of each participant

This information is then aggregated and used to calculate the relevant metrics that we presented previously, being displayed in Table 4.3, grouped by each participant. Much like the previous table, this one is also accompanied by the average of each metric. Since we previously defined that each participant would solve no more than 10 challenges, the **Challenge Completion Rate** metric was always calculated with 10 as the maximum achievable number of completed challenges.

These results suggest a mostly positive engagement with the platform across the participants. An average challenge completion rate of 78% indicates that most participants were able to complete most of the assigned challenges within the allotted session time, with two participants (B and D) completing all challenges and the remaining participants completing between six and nine.

The average successful attempt rate of 42% is expected due to the exploratory nature of the platform. Participants submitted an average of 20.44 attempts to complete an average of 8 challenges, implying that most required multiple tries before finding a correct solution. This is the expected behaviour of a serious game that encourages experimentation and learning through trial and error. Additionally, the Failed Attempt rate was also higher than the Successful Attempt rate

Participant	Challenge Completion Rate	Successful Attempt Rate	Failed Attempt Rate
A	0.7	0.32	0.68
B	1.0	0.52	0.48
C	0.8	0.38	0.62
D	1.0	0.59	0.41
E	0.7	0.40	0.60
F	0.6	0.27	0.73
G	0.6	0.25	0.75
H	0.8	0.50	0.5
I	0.9	0.52	0.48
Average	0,78	0.42	0.58

Table 4.3: Platform Usage Metrics of each participant

for most participants. By itself, this does not mean the platform was too difficult to understand, just that participants submitted multiple attempts before completing them. These results will complement the discussion of the qualitative results in Section 4.3.4, where participant feedback on the platform's difficulty may help explain any patterns found in this data.

4.3.3 Quantitative Results

This section presents the results gathered from the post-use questionnaire, organised according to the seven evaluation parameters described in Section 4.2.4. As mentioned previously, most questions relied on a five-point Likert scale, ranging from 1 (*Completely Disagree*) to 5 (*Completely Agree*). Some sections contain negatively worded questions, where a "low level" actually is considered a positive answer.

The following sections will discuss and sum up the results of the post-questionnaire, grouping them according to their relevant section:

Usability This section revealed a mostly positive trend when it came to the participants' ability to use, navigate, and explore the interface easily. The first three questions had no answers below level 3 (Neutral), supporting the idea that the interface is well organized and easy to understand. However, the second-to-last question, regarding the feedback returned for each submission, seems to reveal a need to clarify and improve the message(s) returned when a player submits an attempt.

Learnability In this section regarding learnability, most answers were positive, with only two at a level 2 (Disagree). We did expect more positive answers in the last question regarding transitions between challenge types. Although mostly positive, the high number of level 3 answers (Neutral) could reveal a need for the platform to progress through its challenges more intuitively.

Satisfaction and Comfort The results in this section seem to indicate a positive level of satisfaction with the platform and its mechanisms. This section only had two answers equal to a

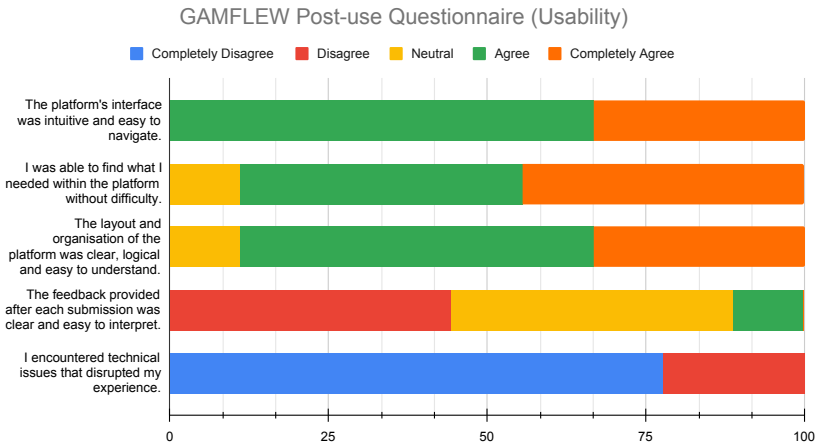

Figure 4.3: Results of the Post-use Questionnaire. Section 1 - Usability

level 3 (Neutral), with the rest being higher (Agree or Completely Agree), which suggests that ***GAMFLEW*** allows players to fully explore, interact, and experiment with the platform and its challenges, without fear of failing, being wrong, or being evaluated on their performance.

Difficulty and Complexity This section, regarding the difficulty and complexity of ***GAMFLEW***, provides the most mixed results in the validation, with answers ranging from a level 1 (Completely Disagree) to a level 5 (Completely Agree).

In the first question, 5 participants did not consider the challenges appropriately difficult, rating this question below a level 3 (Neutral). The second question's answers complement this idea, since most participants showed some level of doubt when being instructed by the platform. This behavior, although unusual, may be a direct cause of the diverse levels of participants' knowledge within the Software Testing sphere.

Nevertheless, these results still demonstrate a general direction for future work, possibly suggesting a need for better explanations or additional educational content that may complement the current challenges.

Effectiveness, Productiveness, and Efficiency

This section, mainly dedicated to measuring the learning effectiveness of the platform, indicates a good direction towards an efficient teaching tool, with an added caveat that should guide important future decisions.

Although the results were mostly positive, the answers to the second-to-last question, regarding feedback on the code coverage (white-box testing) challenges, were mostly comprised of the levels 1, 2, and 3 (Completely Disagree, Disagree, Neutral), which seems to indicate that the platform needs clearer feedback messages regarding which coverage goals have been achieved and which haven't.

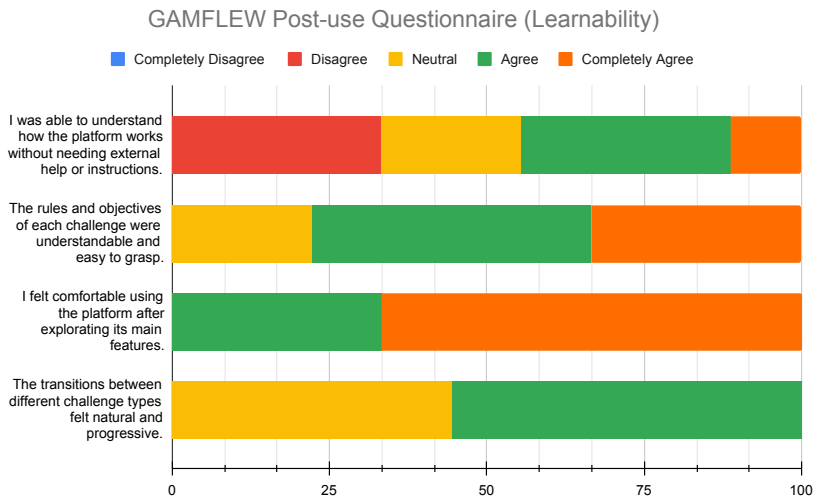

Figure 4.4: Results of the Post-use Questionnaire. Section 2 - Learnability

Usefulness

This sixth section regarding the usefulness and purposefulness of ***GAMFLEW*** indicates that the platform is appropriate to aid teaching efforts on Software Testing and Mutation Testing, with some important considerations. The first consideration comes from the first two questions, where some participants considered the platform suitable, but not complete, to teach players unfamiliar with Mutation Testing. The second comes from the last question, where participants could rate the platform in general. Although most participants rated it with at least a level 3, we still believe more participants would be needed to get an accurate average of the platform's rating.

Serious Game Experience

The final section of this questionnaire was concerned specifically with the gamification and serious game facet of ***GAMFLEW***.

Once again, the results were mostly positive, with most participants sharing the opinion, with different levels of similarity, that the game format, elements, and mechanics contributed towards their feelings of accomplishment, focus, motivation, and engagement when learning about the different topics, with a higher level of success than what would be expected from a theoretical explanation.

These answers directly corroborate our main findings regarding gamification and serious games previously discussed in Chapter 2, supporting the idea that these approaches can have beneficial effects on students' interest and curiosity in the topics being taught.

It should be noted that the last question, regarding how the game-based approach made Mutation Testing concepts easier to learn in comparison to more traditional approaches, still had a relevant number of participants answer with a level 2 (Disagree). Complemented with the results

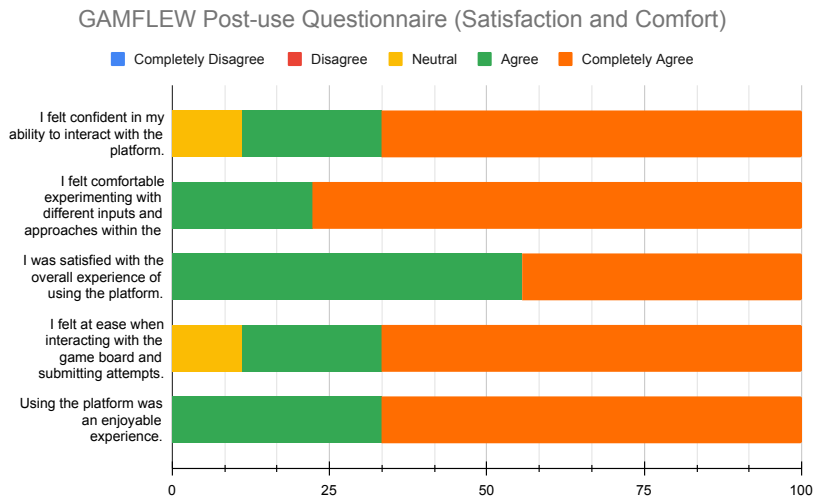

Figure 4.5: Results of the Post-use Questionnaire. Section 3 - Satisfaction and Comfort

in section 4.3.4, we believe this supports the previous belief that Mutation Testing would benefit from structured lessons or educational content that may help the players grasp the objectives more easily.

4.3.4 Qualitative Results

This section presents the qualitative findings gathered from the last section of the post-use questionnaire, which consisted of four optional open-ended questions that invited participants to reflect on what they liked most, what they found most difficult, what they would improve, and any additional comments they may wish to give. Although all nine participants completed the questionnaire, not all chose to answer the questions within this section.

Out of the nine participants, only three replied to at least one of the questions in this final section.

The first participant reported a technical issue that they encountered during a challenge, stating that an empty board caused the challenge to display an error instead of a challenge passed/failed message. This erroneous behavior was discovered and corrected in the project's code at a later date. Despite this, they still commented positively on the platform, stating that the variety of challenges kept them wanting to play.

The second participant commented specifically on the complexity of the MC/DC challenges, stating that some felt hard to understand based on the concept alone, suggesting that these challenges should be solvable with independent attempts, allowing players to complete the challenge through trial and error while not being forced to meet all coverage goals with one singular submission.

The third and final participant also reported a smaller technical issue on the Mutation Testing module, where a non-equivalent mutant was treated like an equivalent one by the system. Much

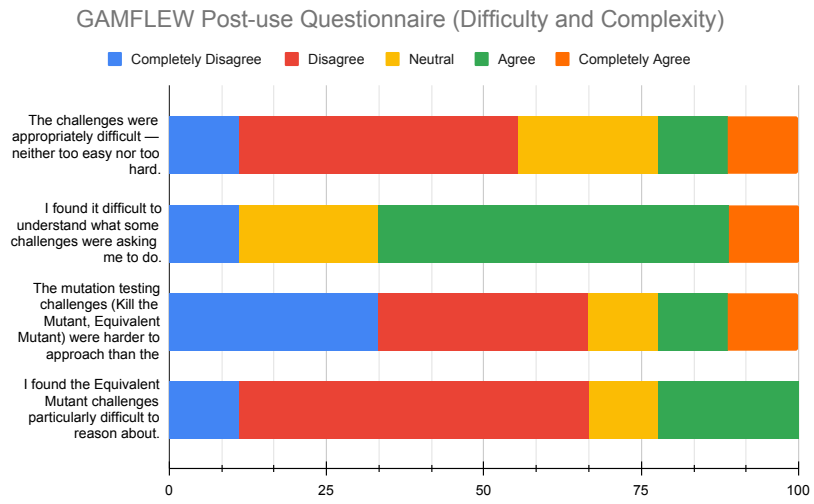

Figure 4.6: Results of the Post-use Questionnaire. Section 4 - Difficulty and Complexity

like the first fault, this one was also corrected shortly after the experiment was concluded. Adding to this, the participant also left a comment suggesting that the platform should offer a wider explanation of the more relevant topics of Mutation Testing.

4.4 Discussion

This section discusses the main findings of the practical validation in consideration of the research questions defined in Chapter 1. While **RQ1** and **RQ2** were addressed through the systematic literature review conducted in Chapter 2, this chapter focuses mostly on **RQ3** and **RQ4**, concerned with the design and evaluation of the extensions proposed in this dissertation.

4.4.1 RQ3 - Can we extend GAMFLEW with challenges to teach Mutation Testing?

RQ3 is primarily addressed by the design and implementation described in Chapter 3, but the validation results confirm whether the proposed extension is coherent, functional, and accessible to real users.

The manual validation confirmed the functional correctness of all 108 challenges across all challenge types, demonstrating that the client-side execution engine works as intended. The two technical issues identified during the practical evaluation were both minor and were corrected shortly after the experiment concluded. Neither seems to have negatively affected the overall experience, as the errors were not critical, and satisfaction seems to have remained mostly positive despite their occurrence.

The learnability results further support the viability of the proposed extensions. Most participants were able to understand the platform's mechanics with minimal or no external guidance, and challenge objectives were mostly clear from their descriptions alone. The transition between

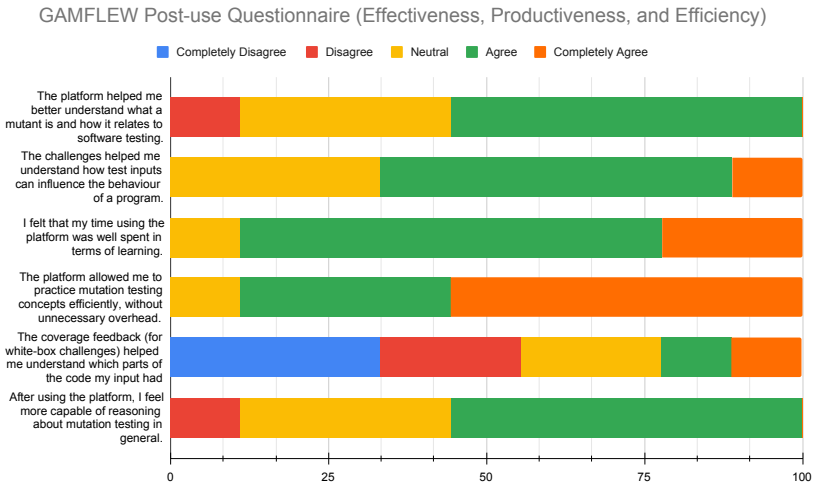

Figure 4.7: Results of the Post-use Questionnaire. Section 5 - Effectiveness, Productiveness, and Efficiency

white-box coverage challenges and mutation testing challenges was rated as generally positive, but the relatively high number of neutral responses on the corresponding question suggests that this connection could be communicated more explicitly within the platform.

One concern raised by both the quantitative and qualitative results was the absence of structured introductory content within the platform, mostly on the topic of mutation testing. Several participants, particularly those with lower familiarity of the topic, indicated that the challenges alone were not always sufficient to teach its concepts. One participant explicitly suggested that the platform would benefit from wider explanatory content on mutation testing. This finding suggests that the extension, while functionally correct, would benefit from an additional educational layer, like tutorials or topic summaries, to fully support the teaching of mutation testing to players encountering it for the first time.

4.4.2 RQ4 - Do the proposed GAMFLEW extensions improve the learning experience (motivation and knowledge acquisition)?

RQ4 is the central question of this evaluation, addressing both the motivational and educational facets of the platform. The results across the seven sections of the post-use questionnaire allow us to examine each of them individually.

4.4.2.1 Motivation and Engagement

The Serious Game Experience section produced the most generally positive results of the entire questionnaire. Most participants agreed that the game format, elements, and mechanics (points, achievements, leaderboard) contributed positively to their feelings of accomplishment, motivation, and engagement throughout the session. The majority also considered the game-based approach

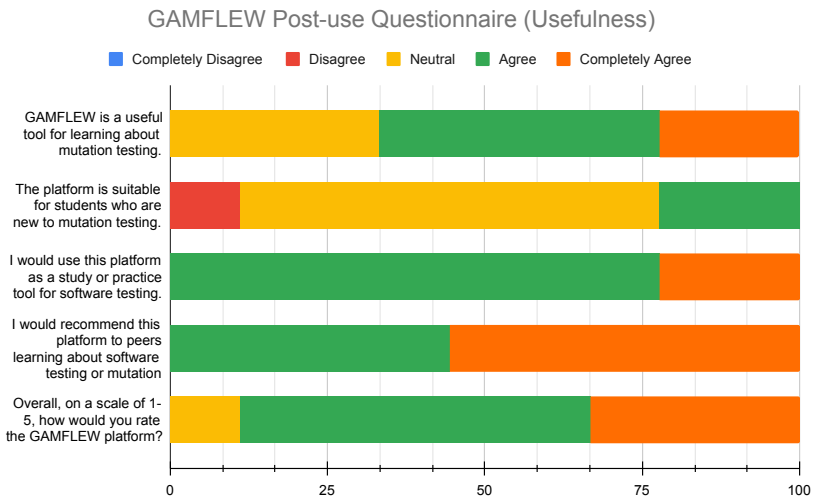

Figure 4.8: Results of the Post-use Questionnaire. Section 6 - Usefulness

more effective at teaching the addressed concepts than a purely theoretical explanation would have been.

However, a significant number of participants still rated the last item of this section, asking whether the game format made mutation testing easier to understand compared to traditional approaches, at a level 2 (Disagree). This is consistent with the warning by de Jesus et al. [11] in Chapter 2, who observed that gamification alone does not guarantee meaningful learning gain. In our case, the result most likely reflects the same idea presented above: the game format is motivating, but not explanatory enough to allow players with little to no knowledge to effectively take advantage of said motivation.

4.4.2.2 Knowledge Acquisition and Perceived Effectiveness

The Effectiveness section indicated a generally positive direction, with most participants agreeing that the challenges helped them grasp important relations within Mutation Testing. The final item of this section, asking participants to rate their confidence in reasoning about Mutation Testing, also received predominantly positive responses. Considering that the pre-use questionnaire revealed a relatively low average level of familiarity with Mutation Testing (2.78 out of 5) and the Equivalent Mutant problem (2.56 out of 5), this increase in perceived confidence is a meaningful outcome.

However, the second-to-last item of the Effectiveness section, which asked about the clarity of the coverage feedback returned after each submission, received predominantly low responses, around the levels 1, 2, and 3. This pattern suggests that, while the execution engine produced correct results, the feedback messages presented to players were not sufficiently informative to help them achieve the coverage goals. One participant's comment reinforced this finding, noting that MC/DC challenges in particular were difficult to understand from the concept description

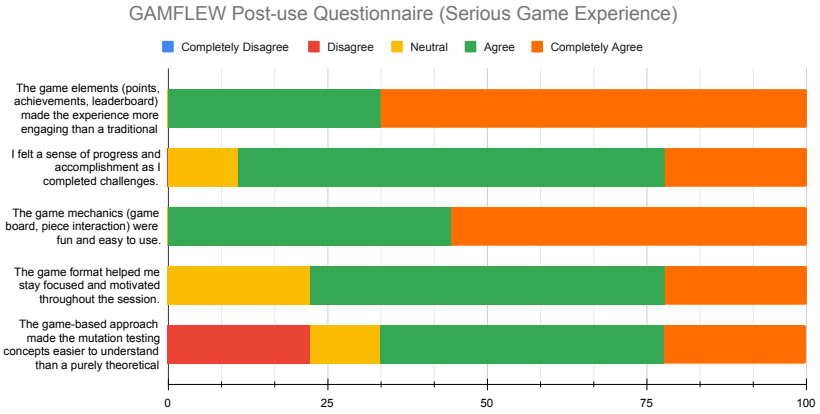

Figure 4.9: Results of the Post-use Questionnaire. Section 7 - Serious Game Experience

alone. This means that the correctness of the execution engine is a necessary but not sufficient condition for an effective education, suggesting that the feedback given to the player still requires further refinement.

4.4.2.3 Difficulty

The Difficulty and Complexity section provided the most mixed results of the entire questionnaire. Five participants considered that the challenges were not appropriately calibrated in difficulty. As discussed previously in Section 4.2.4, higher difficulty ratings for the *Equivalent Mutant* challenges were expected, given the extra layer of complexity required by this challenge type. Therefore, these mixed results reflect the complexity of the topic rather than a flaw in the platform's design, but they still reinforce the need for additional educational content.

4.4.3 Threats to Validity

The validity of the conclusions drawn from this evaluation is subject to a number of threats, which are disclosed here.

Sample size. The total of nine participants is the most significant limitation of this evaluation. This number may be insufficient to draw more robust conclusions. A larger and more diverse sample would be necessary to validate the results more accurately.

Selection bias. Participants were recruited voluntarily, but mostly through convenience sampling. Most individuals already had experience with software testing, mutation testing, serious games, or gamification, potentially increasing positive responses on the motivation and perceived learning, which can be considered a bias.

Short exposure time. The practical validation relied on a relatively short exposure to the platform (around 20 minutes). The motivational impact of some elements, such as the leaderboard or achievement system, could not be fully assessed.

Perceived vs. actual learning. The evaluation relies on self-reported measures of perceived learning. A participant who felt they learned from the experience may not have actually acquired correct knowledge. Future evaluations should consider adding a short exam or test before and after the exposure to the platform.

Undetected technical issues. Two technical issues were identified through participant feedback and corrected after. It is possible that additional issues went undetected.

4.4.4 Final Comments

The results of both evaluations offer a broadly encouraging picture of the extended ***GAMFLEW*** platform. The manual validation confirmed the functional correctness of all 108 challenges, but the practical evaluation, conducted with nine participants from diverse backgrounds, indicated that the platform is usable, satisfying to interact with, and perceived as a meaningful environment for learning software and mutation testing concepts.

At the same time, the evaluation identified two main areas for improvement that should guide future development. The first concerns the clarity of the coverage and mutation testing feedback: while the execution engine produces correct results, the messages communicated to players after each submission are not yet sufficiently informative to fully support the learning process. The second concerns the lack of structured introductory content: several participants indicated that the challenges alone were not always sufficient to convey mutation testing concepts to players encountering them for the first time.

These findings, combined with the threats to validity discussed in Section 4.4.3 suggest that the current evaluation should be interpreted as an initial validation rather than an assessment of the platform's educational impact. A larger and more diverse participant population, combined with an extended experimentation period, would be necessary to draw more accurate conclusions.

Nonetheless, the results seem to provide sufficient evidence to support the viability of the approach taken in this dissertation: extending an existing, validated serious game with an educational module on mutation testing is a promising direction for the teaching of the topic.

Chapter 5

Conclusions

5.1 Summary of Contributions

In summary, this dissertation set out to extend ***GAMFLEW***, an existing, previously validated serious game for white-box testing education, by adding a mutation testing module. The work was motivated by a recognised gap in the current literature regarding gamified approaches to software testing education and serious games designed to teach mutation testing. As previously discussed, no existing approaches integrate mutation testing with white-box coverage analysis within a single, already validated platform.

The contributions of this dissertation can be summarised as follows:

5.1.1 Client-side code execution engine.

A code execution engine was designed and implemented to run JavaScript code asynchronously in the browser using Web Workers. This engine serves as the main foundation of all new challenge types introduced in this dissertation (Mutation Challenges), since it enables real-time execution of player inputs, which can be used to achieve coverage goals or mutation testing conditions in a lightweight, self-contained environment.

5.1.2 Adaptation of existing white-box testing challenges.

The pre-existing white-box testing challenges in ***GAMFLEW*** were adapted to utilise genuine coverage analysis at runtime, replacing the platform's original static Boolean condition evaluation model with real statement, decision, condition, and MC/DC coverage verification.

5.1.3 Mutation testing educational module.

New mutation testing challenges were designed and implemented within ***GAMFLEW***, introducing two new challenge types: *Kill the Mutant*, where players must provide an input that produces different outputs between an original program and one of its mutants, and *Equivalent Mutant*,

where players must correctly classify whether a mutant is equivalent when compared to the original code. These challenges make use of the newly introduced execution engine while remaining coherently integrated within the platform's existing gamification elements, like the score points, leaderboard, and achievement mechanics.

5.1.4 Empirical evaluation.

The extended platform was evaluated through two experiments: a manual validation that confirmed the functional correctness of all 108 challenges, and a practical evaluation with nine participants from diverse backgrounds that indicated the platform is usable, satisfying to interact with, and perceived as a meaningful environment for learning mutation testing concepts.

5.2 Limitations

Despite the encouraging results of the evaluation, this dissertation is subject to a number of limitations that should be acknowledged.

5.2.1 Scale of the evaluation

The practical evaluation was conducted with nine participants, which may be insufficient to draw robust conclusions. Future work should seek to replicate the evaluation with a larger and more diverse participant population.

5.2.2 Feedback clarity

The quantitative results revealed that the feedback communicated to players after each submission, particularly for white-box coverage challenges, was not sufficiently informative to fully guide their understanding of the coverage goals. While the execution engine produces correct results, the messages require further refinement to achieve a higher level of clarity.

5.2.3 Lack of structured introductory content

Several participants, particularly those with lower prior familiarity with mutation testing, indicated that the challenges alone were not always sufficient to teach the relevant concepts. The platform currently lacks educational content specifically for mutation testing, which could allow players to learn the necessary foundation before attempting the challenges.

5.2.4 JavaScript dependency

All challenges in GAMFLEW are implemented in JavaScript. This limits the platform to learners with a relative knowledge or experience with JavaScript and may disadvantage users with different programming backgrounds.

5.2.5 Perceived vs. actual learning

The practical validation relied on self-reported measures of perceived learning rather than objective knowledge assessments. Future evaluations should consider complementing the questionnaire with pre- and post-use tests to assess actual learning gains more accurately.

5.3 Future Work

The limitations identified above, combined with the findings of the validation, give us a general guide for future work.

5.3.1 Larger and more diverse evaluation

A more robust evaluation of the platform should be conducted with a larger and more diverse participant sample, ideally spanning multiple sessions over the course of a semester. Such a study would allow the long-term motivational effects of the gamification elements to be assessed and would provide a more reliable basis to draw conclusions about the platform's educational impact. The inclusion of a pre- and post-knowledge test would also allow perceived learning to be correlated with actual knowledge gain.

5.3.2 Improved feedback

The most immediately actionable direction is the improvement of the feedback messages returned to players after each submission. In particular, the coverage feedback for white-box challenges should be made more explicit, indicating not only whether a coverage goal was met but also which specific statements, branches, or conditions were or were not exercised by the submitted input. Similarly, the mutation testing feedback could be enriched with more detailed explanations of why a given input did or did not distinguish the mutant's behavior from the original.

5.3.3 In-platform introductory content

A natural complement to the existing challenges would be a set of dedicated introductory materials integrated directly into the platform, covering the key concepts of mutation testing — mutants, mutation operators, mutation score, and the equivalent mutant problem — before players attempt the challenges. This could take the form of interactive tutorials, concept summaries within the challenge interface, or a structured onboarding flow that guides new players through the platform's mechanics progressively.

5.3.4 Expanded challenge set

The current mutation testing module comprises a representative but necessarily limited set of challenges. Future work should expand this set to cover a wider range of mutation operators, programming constructs, and difficulty levels, ensuring that the platform remains engaging and educational

across multiple sessions. The platform's existing challenge editor, available to teachers and administrators, could also be extended to support the creation of mutation testing challenges, allowing educators to design their own content tailored to their specific course requirements.

In conclusion, we consider that we have achieved the main goals of this research work, having produced valuable and meaningful contributions to an already validated platform, and we believe that ***GAMFLEW*** can positively and accurately contribute towards software and mutation testing education. Several ideas were identified as plausible future work, and if possible, we would like to continue expanding the platform with these suggestions.

References

[1] Kevin Buffardi and Pedro Valdivia. “Bug Hide-and-Seek: An Educational Game for Investigating Verification Accuracy in Software Tests”. In: *2018 IEEE Frontiers in Education Conference (FIE)*. San Jose, CA, USA, Oct. 2018, pp. 1–8. DOI: 10.1109/FIE.2018.8658748.

[2] Florian Cammaerts and Monique Snoeck. *ModelDefenders: A novel gamified mutation testing game for model-driven engineering*. 2023. URL: https://ceur-ws.org/Vol-3645/tools4.pdf.

[3] Hongwei Chen, Xin Wang, and Limin Pan. “Research on Teaching Methods and Tools of Software Testing”. In: *2020 15th International Conference on Computer Science & Education (ICCSE)*. Delft, Netherlands, 2020, pp. 760–763. DOI: 10.1109/ICCSE49874.2020.9201788.

[4] Benjamin S. Clegg, José Miguel Rojas, and Gordon Fraser. “Teaching software testing concepts using a mutation testing game”. In: *Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering and Education Track (ICSE-SEET)*. 2017, pp. 33–36. DOI: 10.1109/ICSE-SEET.2017.1.

[5] Igor Ernesto Ferreira Costa and Sandro Ronaldo Bezerra Oliveira. “A Systematic Strategy to Teaching of Exploratory Testing using Gamification”. In: *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*. SciTePress, 2019, pp. 307–314. DOI: 10.5220/0007711603070314.

[6] Tommaso Fulcini et al. “A Review on Tools, Mechanics, Benefits, and Challenges of Gamified Software Testing”. In: *ACM Computing Surveys* 55.14s (2023), 310:1–310:37. ISSN: 0360-0300. DOI: 10.1145/3582273. URL: https://dl.acm.org/doi/10.1145/3582273.

[7] Rodrigo F. Gomes and Valéria Lelli. “GAMUT: GAMe-based learning approach for teaching Unit Testing”. In: *ACM International Conference Proceeding Series*. Nov. 2022. DOI: 10.1145/3493244.3493263.

[8] Mahdi Houshmand and Samad Paydar. “EMVille: A gamification-based approach to address the equivalent mutant problem”. In: *2017 7th International Conference on Computer and Knowledge Engineering (ICCKE)*. Mashhad, Iran, Oct. 2017, pp. 326–332. DOI: 10.1109/ICCKE.2017.8167900.

[9] Gabriela Martins de Jesus et al. “Gamification in Software Testing: A Characterization Study”. In: *Proceedings of the III Brazilian Symposium on Systematic and Automated Software Testing (SAST)*. 2018, pp. 39–48.

[10] Gabriela Martins de Jesus et al. “Gamifying Testing in IntelliJ: A Replicability Study”. In: *Proceedings of the ACM on Software Engineering* (2025). DOI: 10.1145/3728983. URL: https://dl.acm.org/doi/10.1145/3728983.

[11] Gabriela Martins de Jesus et al. “Is It Worth Using Gamification on Software Testing Education? An Experience Report”. In: *Proceedings of the XVIII Brazilian Symposium on Software Quality (SBQS)*. Association for Computing Machinery, 2019, pp. 178–187. DOI: 10.1145/3364641.3364661. URL: https://doi.org/10.1145/3364641.3364661.

[12] Yue Jia and Mark Harman. “An Analysis and Survey of the Development of Mutation Testing”. In: *IEEE Transactions on Software Engineering* 37.5 (Sept. 2011), pp. 649–678. ISSN: 1939-3520. DOI: 10.1109/TSE.2010.62.

[13] José Carlos Maldonado and Elisa Fabiana Barbosa. “Establishing a Mutation Testing Educational Module based on IMA-CID”. In: *Second Workshop on Mutation Analysis (Mutation 2006 — ISSRE Workshops 2006)*. Raleigh, NC, USA, 2006, p. 14. DOI: 10.1109/MUTATION.2006.4.

[14] Beatriz Marín et al. “ENACTEST Project - European Innovation Alliance for Testing Education”. In: *Proceedings of the Research Projects Exhibition Papers Presented at the 35th International Conference on Advanced Information Systems Engineering (CAiSE 2023)*. Ed. by Joel Font et al. Vol. 3413. CEUR-WS, 2023, pp. 91–96. URL: https://ceur-ws.org/Vol-3413/paper13.pdf.

[15] Beatriz Marín et al. “ENACTEST: European Innovation Alliance for Testing Education”. In: *CEUR Workshop Proceedings*. Vol. 3144. CEUR-WS, 2022. URL: https://ceur-ws.org/Vol-3144/RP-paper5.pdf.

[16] Andrea Materazzo et al. “Unit Brawl: Gamified Unit Testing Inspired by Battle Royale”. In: *Proceedings of the 2023 IEEE/ACM 7th International Workshop on Games and Software Engineering (GAS)*. 2023, pp. 1–7. DOI: 10.1109/GAS59301.2023.00008.

[17] Matipa Ricky Ngandu et al. “Capturing student interest in software engineering through gamification: a systematic literature review”. In: *Discover Education* 10 (2023). Article 200. DOI: 10.1007/s44217-023-00069-4. URL: https://doi.org/10.1007/s44217-023-00069-4.

[18] Mike Papadakis et al. “Mutation Testing Advances: An Analysis and Survey”. In: *Advances in Computers*. Vol. 112. Elsevier, 2019, pp. 275–378. DOI: 10.1016/bs.adcom.2018.03.015.

[19] José Miguel Rojas and Gordon Fraser. “Code Defenders: A Mutation Testing Game”. In: *2016 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 2016, pp. 162–167. DOI: 10.1109/ICSTW.2016.43.

[20] Mehrdad Saadatmand et al. “Software Testing Education and Industry Needs - Report from the ENACTEST EU Project”. In: *Product-Focused Software Process Improvement. Industry, Doctoral-Symposium, Tutorial, and Workshop Papers: 26th International Conference, PROFES 2025, Salerno, Italy, December 1–3, 2025, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2025, pp. 53–68. DOI: 10.1007/978-3-032-12092-2_4.

[21] M. Silva, A. C. R. Paiva, and A. Mendes. “GAMFLEW: serious game to teach white-box testing”. In: *Software Quality Journal* 33.1 (2025). ISSN: 1573-1367. DOI: 10.1007/s11219-024-09706-z. URL: https://doi.org/10.1007/s11219-024-09706-z.

[22] Marcos Souza et al. “Gamification in Software Engineering Education: a Tertiary Study”. In: *Proceedings of the XXXVII Brazilian Symposium on Software Engineering (SBES)*. 2023. DOI: 10.1145/3613372.3614193. URL: https://dl.acm.org/doi/10.1145/3613372.3614193.

[23] Peter Straubinger, Lena Bloch, and Gordon Fraser. “Engaging Young Learners with Testing Using the Code Critters Mutation Game”. In: *2024 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 2024, pp. 322–330. DOI: 10.1109/ICSTW60967.2024.00063.

[24] Peter Straubinger, Lena Caspari, and Gordon Fraser. “Code Critters: A Block-Based Testing Game”. In: *2023 IEEE 16th International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 2023, pp. 426–429. DOI: 10.1109/ICSTW58534.2023.00077.

[25] Peter Straubinger and Gordon Fraser. “Improving Testing Behavior by Gamifying IntelliJ”. In: *Proceedings of the International Conference on Software Engineering (ICSE)*. May 2024. DOI: 10.1145/3597503.3623339.

[26] Jeff Stuckman and Guo-Qiang Zhang. *Mastermind is NP-complete*. arXiv:cs/0512049. 2005. DOI: 10.48550/arXiv.cs/0512049. URL: https://doi.org/10.48550/arXiv.cs/0512049.

[27] Pedro Tavares et al. “FRAFOL: FRAmework FOr Learning mutation testing”. In: *ISSTA 2024 — Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2024, pp. 1846–1850. DOI: 10.1145/3650212.3685306.

[28] Pedro Henrique Dias Valle et al. “Educational games: A contribution to software testing education”. In: *2017 IEEE Frontiers in Education Conference (FIE)*. Indianapolis, IN, USA: IEEE, Oct. 2017, pp. 1–8. DOI: 10.1109/FIE.2017.8190470.

Appendix A

Validation Questionnaires

This appendix presents the full text of the two questionnaires used in the validation study described in Chapter 4. The pre-use questionnaire (Appendix A.1) was administered before participants interacted with the platform. The post-use questionnaire (Appendix A.2) was administered immediately after the platform session.

Both questionnaires were distributed digitally via Google Forms. All responses were collected anonymously; no personally identifiable information was recorded.

A.1 Pre-Use Questionnaire

Section 1 - Background

1. **How old are you?**
   (Select one)

   ☐ 18-25

   ☐ 26-39

   ☐ 40-69

   ☐ 70+

2. **What is your gender?**
   (Select one)

   ☐ Male

   ☐ Female

   ☐ Would rather not disclose

   ☐ Other

3. **What best describes your current qualification in the area of Software Testing?**
   (Select one)

- ☐ University student (Computer Science, Software Engineering, or related)
- ☐ University student (other area)
- ☐ Software professional (industry)
- ☐ Teacher/ Researcher / Academic
- ☐ Other

4. **How many years of programming experience do you have?**
   (Select one)

   - ☐ Less than 1 year
   - ☐ 1–2 years
   - ☐ 3–5 years
   - ☐ More than 5 years

5. **Have you ever taken a course or module on software testing?**
   (Select one)

   - ☐ Yes, as part of a formal degree or programme
   - ☐ Yes, through self-study or online courses
   - ☐ No

Section 2 - Familiarity with Relevant Concepts

Rate your familiarity with each of the following concepts on a scale from 1 to 5, where 1 means "No familiarity" and 5 means "Very familiar"

4. **Software testing in general**

1	**2**	**3**	**4**	**5**
○	○	○	○	○

5. **White-box testing** (testing based on internal code structure)

1	**2**	**3**	**4**	**5**
○	○	○	○	○

6. **Code coverage criteria** (e.g. statement, decision, condition coverage)

1	**2**	**3**	**4**	**5**
○	○	○	○	○

7. **Mutation testing** (introducing small changes to code to evaluate test quality)

1	2	3	4	5
○	○	○	○	○

8. **The equivalent mutant problem**

1	2	3	4	5
○	○	○	○	○

Section 3 - Prior Experience with Serious Games

9. **Have you ever used a serious game or gamified platform to learn a technical concept?**
   (Select one)

   ☐ Yes, frequently

   ☐ Yes, occasionally

   ☐ Once or twice

   ☐ Never

10. **If yes, did you find it helpful for learning?**
    (Select one or skip if you answered "Never" above)

    ☐ Very helpful

    ☐ Somewhat helpful

    ☐ Neutral

    ☐ Not very helpful

    ☐ Not helpful at all

A.2 Post-Use Questionnaire

Rate how much you agree or disagree with the following statements, using the following scale:

1 = Strongly disagree 2 = Disagree 3 = Neutral 4 = Agree 5 = Strongly agree

Section 1 - Usability

1. The platform's interface was intuitive and easy to navigate.

1	2	3	4	5
○	○	○	○	○

2. I was able to find what I needed within the platform without difficulty.

1	2	3	4	5
○	○	○	○	○

3. The layout and organisation of the platform were clear and logical.

1	**2**	**3**	**4**	**5**
○	○	○	○	○

4. The feedback provided after each submission was clear and easy to interpret.

1	**2**	**3**	**4**	**5**
○	○	○	○	○

5. I did not encounter technical issues that disrupted my experience.

1	**2**	**3**	**4**	**5**
○	○	○	○	○

Section 2 - Learnability

6. I was able to understand how the platform works without needing external help or instructions.

1	**2**	**3**	**4**	**5**
○	○	○	○	○

7. The rules and objectives of each challenge were easy to grasp.

1	**2**	**3**	**4**	**5**
○	○	○	○	○

8. I felt comfortable using the platform after exploring its main features.

1	**2**	**3**	**4**	**5**
○	○	○	○	○

9. The transitions between different challenge types felt natural and progressive.

1	**2**	**3**	**4**	**5**
○	○	○	○	○

Section 3 - Satisfaction and Comfort

10. I felt confident in my ability to interact with the platform.

1	**2**	**3**	**4**	**5**
○	○	○	○	○

11. I felt comfortable experimenting with different inputs and approaches within the challenges.

1	**2**	**3**	**4**	**5**
○	○	○	○	○

12. I was satisfied with the overall experience of using the platform.

1	**2**	**3**	**4**	**5**
○	○	○	○	○

13. I felt at ease when interacting with the game board and submitting attempts.

1	**2**	**3**	**4**	**5**
○	○	○	○	○

14. Using the platform was an enjoyable experience.

1	**2**	**3**	**4**	**5**
○	○	○	○	○

Section 4 - Difficulty and Complexity

15. The challenges were appropriately difficult. Neither too easy nor too hard.

1	**2**	**3**	**4**	**5**
○	○	○	○	○

16. I found it difficult to understand what some challenges were asking me to do.

1	**2**	**3**	**4**	**5**
○	○	○	○	○

17. The mutation testing challenges (Kill the Mutant, Equivalent Mutant) were harder to approach than the coverage challenges.

1	**2**	**3**	**4**	**5**
○	○	○	○	○

18. I found the Equivalent Mutant challenges particularly difficult to reason about.

1	**2**	**3**	**4**	**5**
○	○	○	○	○

Section 5 - Effectiveness, Productiveness, and Efficiency

20. The platform helped me better understand what a mutant is and how it differs from the original code.

1	**2**	**3**	**4**	**5**
○	○	○	○	○

21. The challenges helped me understand how test inputs can influence the behaviour of a program.

1	**2**	**3**	**4**	**5**
○	○	○	○	○

22. I felt that my time using the platform was well spent in terms of learning.

1	**2**	**3**	**4**	**5**
○	○	○	○	○

23. The platform allowed me to practice mutation testing concepts efficiently, without unnecessary overhead.

1	**2**	**3**	**4**	**5**
○	○	○	○	○

24. The coverage feedback (for white-box testing challenges) helped me understand which parts of the code my input had exercised.

1	**2**	**3**	**4**	**5**
○	○	○	○	○

25. After using the platform, I feel more capable of reasoning about Mutation Testing in general.

1	**2**	**3**	**4**	**5**
○	○	○	○	○

Section 6 - Usefulness

26. GAMFLEW is a useful tool for learning about mutation testing.

1	**2**	**3**	**4**	**5**
○	○	○	○	○

27. The platform is suitable for students who are new to mutation testing.

1	**2**	**3**	**4**	**5**
○	○	○	○	○

28. I would use this platform again as a study or practice tool for software testing.

1	**2**	**3**	**4**	**5**
○	○	○	○	○

29. I would recommend this platform to peers learning about software testing or mutation testing.

1	**2**	**3**	**4**	**5**
○	○	○	○	○

30. **Overall, how would you rate the GAMFLEW platform?**
(Select one)

    ☐ Excellent

    ☐ Good

    ☐ Neutral

    ☐ Poor

    ☐ Very poor

Section 7 - Serious Game Experience

31. The game elements (points, achievements, leaderboard) made the experience more engaging than a traditional exercise would have.

1	**2**	**3**	**4**	**5**
○	○	○	○	○

32. I felt a sense of progress and accomplishment as I completed challenges.

1	**2**	**3**	**4**	**5**
○	○	○	○	○

33. The game mechanics (game board, piece interaction) were fun and easy to use.

1	**2**	**3**	**4**	**5**
○	○	○	○	○

34. The game format helped me stay focused and motivated throughout the session.

1	**2**	**3**	**4**	**5**
○	○	○	○	○

35. The game-based approach made mutation testing concepts easier to understand than a purely theoretical explanation would have.

1	**2**	**3**	**4**	**5**
○	○	○	○	○

Section 8 - Open-Ended Feedback

Please answer the following questions in your own words. These questions are optional but greatly appreciated.

38. What did you like most about the platform?

39. What did you find most confusing or difficult to understand?

40. What would you change or improve about the platform?

41. Do you have any additional comments or suggestions?

Appendix B

Source Code Excerpts

This appendix presents the source code of the functions used in the challenges available in GAMFLEW.

Test File 1: Is Move Valid?

```
// board includes the board's state.
// start: the (row, column) position from where a piece last moved.
// destination: the (row, column) position to where a piece last moved.
function is_valid_move(board, start, destination) {
    // Math.abs() gives you the absolute value of whatever you call it
        with.
    // Math.abs(-1) == 1
    if (destination.row < 0 || destination.row > 7 ||
        destination.column < 0 || destination.column > 7) {
        return false;
    }

    var lineDifference = Math.abs(start.row - destination.row);
    var columnDifference = Math.abs(start.column - destination.column);

    if (lineDifference != columnDifference) {
        return false;
    } else {
        if (lineDifference == 1) {
            if (board.state[destination.row][destination.column].color
                    == Color.EMPTY) {
                return true;
            } else {
                return false;
            }
```

```
        } else if (lineDifference == 2) {
            if (board.state[destination.row][destination.column].color
                    == Color.EMPTY) {
                var middlePiece = board.state[
                    Math.round((start.row + destination.row) / 2)][
                    Math.round((start.column + destination.column) / 2)];
                if (middlePiece.color ==
                        board.state[start.row][start.column].color) {
                    return false;
                } else if (middlePiece.color != Color.EMPTY) {
                    return true;
                }
            } else {
                return false;
            }
        } else {
            return false;
        }
    }
}
```

Listing B.1: Test File 1: Is Move Valid?

Test File 2: Is Board Valid?

```
function is_board_valid(board) { // board is the board's state.
    var bluePieces = this.count_blue_pieces(board);
    var redPieces = this.count_red_pieces(board);

    if (bluePieces > 12 || redPieces > 12) {
        return false;
    } else if (bluePieces == 0 || redPieces == 0) {
        return false;
    }

    var pieces = this.get_pieces(board);
    var odd = [1, 3, 5, 7], even = [0, 2, 4, 6];

    for (const p of pieces) {
        if (p.position.x % 2 != 0) { // x is equal to the piece's row
            if (!odd.includes(p.position.y)) { // y is equal to the piece'
                s column
```

```
                return false;
            }
        } else {
            if (!even.includes(p.position.y)) {
                return false;
            }
        }
    }

    return true;
}
```

Listing B.2: Test File 2: Is Board Valid?

Test File 3: Can Piece Move?

```
function can_piece_move(board) {
    const piece = this.get_pieces(board)[0]
    // Quadrants are: top-right, bottom-right, bottom-left, top-left.
    if (piece.color == Color.STACK) return;

    var directions = [
        { row: 1, column: 1 },
        { row: 1, column: -1 },
        { row: -1, column: -1 },
        { row: -1, column: 1 },
    ];

    // Kings can move in all directions.
    if (piece.king) {
        for (var d of directions) {
            var destination_row = Number(piece.position.x) + d.row,
                destination_column = Number(piece.position.y) + d.column;

            if ((board.state[destination_row][destination_column].color
                    == Color.EMPTY) ||
                (board.state[destination_row][destination_column].color
                    != piece.color &&
                board.state[destination_row + d.row]
                    [destination_column + d.column].color == Color.EMPTY))
                        {
                return true;
```

```
            }
        }
    }

    // Normal pieces can only move in half the directions.
    // Blue and red pieces move in opposite directions!
    if (piece.color == Color.RED) {
        for (var d of [directions[0], directions[1]]) {
            var destination_row = Number(piece.position.x) + d.row,
                destination_column = Number(piece.position.y) + d.column;

            // Color.EMPTY represents an empty space.
            if (board.state[destination_row][destination_column].color
                    == Color.EMPTY ||
                (board.state[destination_row][destination_column].color
                    != piece.color &&
                board.state[destination_row + d.row]
                    [destination_column + d.column].color == Color.EMPTY))
                        {
                return true;
            }
        }
    } else {
        // Blue pieces only move 'up', which means
        // all moves have a negative column component.
        // They can, however, have a positive or negative row component.
        for (var d of [directions[2], directions[3]]) {
            var destination_row = Number(piece.position.x) + d.row,
                destination_column = Number(piece.position.y) + d.column;

            if (board.state[destination_row][destination_column].color
                    == Color.EMPTY ||
                (board.state[destination_row][destination_column].color
                    != piece.color &&
                board.state[destination_row + d.row]
                    [destination_column + d.column].color == Color.EMPTY))
                        {
                return true;
            }
        }
    }
}
```

Listing B.3: Test File 3: Can Piece Move?

Test File 4: Has Game Ended?

```
function has_game_ended(board) {
    var pieces = this.get_pieces(board);
    if (pieces.length == 0) return false;

    if (pieces.length == 1) {
        return true;
    } else if (pieces.every(p => p.color == Color.RED) ||
               pieces.every(p => p.color == Color.BLUE)) {
        return true;
    } else {
        return false;
    }
}
```

Listing B.4: Test File 4: Has Game Ended?

Test File 5: Who Won?

```
function who_won(board) {
    var blue = this.count_blue_pieces(board),
        red = this.count_red_pieces(board);

    if (blue == red) {
        if (this.count_empty_spaces(board) != 64) {
            // Tie
            return [Color.RED, Color.BLUE];
        } else {
            // Invalid board.
            return null;
        }
    } else {
        if (blue > red) {
            return Color.BLUE;
        } else {
            return Color.RED;
        }
    }
}
```

Listing B.5: Test File 5: Who Won?

Test File 6: Build A Triangle

```
function build_a_triangle(board) {
    var a = this.count_red_pieces(board);   // Counts the number of red
        pieces.
    var b = this.count_blue_pieces(board);  // Counts the number of blue
        pieces.
    var c = this.count_empty_spaces(board); // Counts the number of empty
        spaces.

    // Tests if the lengths of the sides of the triangle are valid.
    if (!this.is_triangle(a, b, c)) {
        return false;
    } else {
        if (b == c && b == a) {
            return true;
        } else if (b == a || b == c || a == c) {
            return true;
        } else {
            return true;
        }
    }
}
```

Listing B.6: Test File 6: Build A Triangle

Test File 7: Draw A Triangle

```
function draw_a_triangle(board) {
    // This function finds all blue pieces in the board and
    // returns them in a list. The final list is ordered by
    // row and column - respectively, (row, column).
    var vertices = this.find_blue_pieces(board)

    if (vertices.length != 3) {
```

```
        return;
    }

    // This function calculates the Cartesian distance between
    // two (row, column) vertices. It rounds up to 2 decimal places.
    var length_1 = this.distance(vertices[0], vertices[1]),
        length_2 = this.distance(vertices[1], vertices[2]),
        length_3 = this.distance(vertices[2], vertices[0]);

    // Ordered from the shortest (1st) to longest (3rd).
    var ordered_sides = [length_1, length_2, length_3].sort();

    // side_a is always the shortest,
    // side_c is always the biggest.
    // side_b may be the same length as either the above, but NOT BOTH.
    var side_a = ordered_sides[0],
        side_b = ordered_sides[1],
        side_c = ordered_sides[2];

    // Math.sqrt() calculates the square root of a number.
    // Math.pow(number, power) calculates number^power.
    // Sets are like lists, but they contain no duplicates.
    if (Math.sqrt(Math.pow(side_a, 2) + Math.pow(side_b, 2)) == side_c) {
        return true; // Rectangle Triangle (Pythagoras)
    } else if (new Set([side_a, side_b, side_c]).size == 3) {
        return true; // Scalene Triangle
    } else if (new Set([side_a, side_b, side_c]).size == 2) {
        return true; // Isosceles Triangle
    }
}
```

Listing B.7: Test File 7: Draw A Triangle