FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Framework for Teaching Mutation Testing

Xavier da Silva Costa

Master's degree in Software Engineering

Supervisor: Ana Cristina Ramada da Paiva

August 8, 2025

Framework for Teaching Mutation Testing

Xavier da Silva Costa

Master's degree in Software Engineering

August 8, 2025

Resumo

À medida que a procura por *software* robusto e escalável continua a aumentar, o papel de testes de *software* torna-se cada vez mais crítico. Em paralelo, o rápido avanço da Inteligência Artificial está a redefinir o panorama tecnológico, levantando preocupações quanto à relevância futura das profissões informáticas. Para se manterem competitivos, **é essencial que estes profissionais dominem práticas fundamentais como *software testing*** — uma área frequentemente subvalorizada e percecionada como desmotivante no contexto académico Garousi et al. (2020).

Entre as várias estratégias de teste, ***mutation testing* tem-se destacado como uma técnica poderosa** Amalfitano et al. (2022) para avaliar a eficácia das suítes de testes, através da introdução de pequenas alterações (mutações) no código do programa e da verificação da sua deteção pelos casos de teste. Apesar do seu potencial pedagógico, esta técnica permanece subutilizada em ambientes educativos devido à sua **complexidade técnica** e à **escassez de ferramentas de aprendizagem acessíveis**.

Esta dissertação investiga metodologias atuais de teste de *software* com o objetivo de expandir a proposta de uma plataforma educativa inovadora, concebida para aumentar o envolvimento com conceitos de teste, com especial foco em *mutation testing*. A plataforma, denominada **FRAFOL**, combina a técnica avançada de mutação com estratégias pedagógicas modernas, como gamificação, interfaces de utilizador intuitivas e melhorias na usabilidade do sistema. FRAFOL estabelece um ecossistema centralizado que simula um ambiente de sala de aula, onde os estudantes podem registar-se, submeter código de teste e consultar tabelas classificativas. Em simultâneo, os docentes podem acompanhar as submissões, avaliar o desempenho e fornecer feedback individualizado, colmatando assim a lacuna instrucional entre estudantes e professores. Além disso, **FRAFOL destaca-se por oferecer um ambiente de teste dinâmico que integra múltiplas ferramentas de *mutation testing***, diferenciando-se das soluções educativas existentes e promovendo um acesso facilitado a esta abordagem.

Do ponto de vista técnico, esta dissertação implementa um sistema Single Page Application (SPA), utilizando tecnologias e técnicas de *frontend* contemporâneas, como por exemplo *bundle minification*. Avalia ainda a integração do *Firebase* como uma solução **gratuita e centralizada de CRM** para autenticação de utilizadores e gestão de dados na plataforma.

Os resultados empíricos demonstram melhorias significativas tanto na **satisfação dos estudantes** com FRAFOL, como no **desempenho** do próprio sistema. Esta investigação contribui, assim, com uma solução abrangente e escalável para o ensino de *software testing*, com o objetivo mais amplo de formar profissionais informáticos altamente qualificados, preparados para enfrentar um cenário tecnológico em constante evolução.

Keywords: Software Testing, Education, Mutation Testing, Mutation Frameworks, Monolithic Architecture, CRM.

Abstract

As the demand for robust and scalable software continues to rise, the role of software testing becomes increasingly critical. In parallel, the rapid advancement of Artificial Intelligence is reshaping the technology landscape, raising concerns about the future relevance of traditional IT roles. To remain competitive, **it is essential for IT professionals to master foundational practices such as software testing**—an area that is often undervalued and perceived as demotivating within academic contexts Garousi et al. (2020).

Among the various testing strategies, **mutation testing has emerged as a powerful technique** Amalfitano et al. (2022) for evaluating the effectiveness of test suites by introducing small, systematic changes (mutations) to the program code and observing whether the test cases detect them. Despite its pedagogical potential, mutation testing remains underutilized in educational environments due to its technical complexity and lack of accessible learning tools.

This work investigates current software testing methodologies with the aim of proposing an innovative educational framework designed to enhance engagement with testing concepts, particularly mutation testing. The proposed framework, **FRAFOL**, combines this advanced testing method with modern pedagogical strategies such as gamification, intuitive user interfaces, and system usability enhancements. It establishes a centralized ecosystem simulating a classroom environment, where students can register, submit test code, and view ranked leaderboards. Simultaneously, educators can monitor submissions, assess performance, and provide individualized feedback, thereby bridging the instructional gap between teachers and learners. Moreover, **FRAFOL uniquely supports a dynamic testing environment that integrates multiple mutation testing tools**, setting it apart from existing educational solutions and reinforcing easy accessibility to mutation testing education.

From a technical standpoint, the research work explores the implementation of a Single Page Application (SPA) utilizing contemporary *frontend* technologies and techniques, such as bundle *minification*. It also evaluates the integration of Firebase as a **free centralized CRM** solution for user authentication and data management within the platform.

Empirical results demonstrate measurable improvements in both **student satisfaction** and **system performance**. The research thus contributes a comprehensive and scalable solution for teaching software testing, with the broader objective of cultivating highly skilled IT professionals equipped to navigate a rapidly evolving technological landscape.

Keywords: Software Testing, Education, Mutation Testing, Mutation Frameworks, SPA, CRM.

Acknowledgments

To my mother, father, sister, family, and friends, who put up with me throughout this journey.
To my supervisor, Professor Ana, who guided me through a difficult academic phase.
To my team leader, Lucas Rhoden, for transforming me into the professional I am today.
Lastly, to Mariana and to the church, we shall get it done.

Xavier da Silva Costa

"I gave it a go, I gave it a turn.
I gave it my heart, I'm willing to learn.
And it changed my life, I'm trying to get close with the stars tonight"

Jarad Higgins

Contents

List of Figures

List of Tables

Abbreviations and Symbols

API	Application Programming Interface
CDN	Content Delivery Network
CI/CD	Continuous Integration/Continuous Deployment
CRM	Customer Relationship Management
FOSS	Free Open-Source Software
HTML	HyperText Markup Language
IT	Information Technology
JVM	*Java* Virtual Machine
kB	*Kilobyte*
SDK	Software Development Kit
SE	Software Engineering
SPA	Single Page Application
SVG	Scalable Vector Graphics
UI	User Interface

Chapter 1

Introduction

Software is everywhere. Software systems' development is expected to grow more than 10% annually through 2029, including approximately 12% in 2025 Compton and Romanoff (2025). This means that the codependence of human life with software is ever-growing. More and more vital human tasks are being handled by simple software programs. Therefore, it is critical that these tools are **bulletproof** for use in daily life. A human being can not ever expect a fault from a software system, especially when any health constraints are at risk, these systems are expected to prevail as an aid tool for infinite possibilities.

What is the only way one can ensure that:

- These software solutions can enter the life of any human being and **be ready** to fully serve their purpose
- These tools will not disrupt, compromise, or **put at risk** the lives of those taking advantage of them.

The answer is: **testing it**. Software testing is a critical step in the software development process that aims to ensure that the functionality being created is secure, correct, and ultimately provides value for the end users IEEE Computer Society (2024). Furthermore, software testing is available in many ways: each business identifies the best procedures to be adopted in order to effectively test and ship their product into the market. With the growing complexity of systems, it is expected that robust testing is a *de facto* standard mandatory step in the development lifecycle of the program.

Research suggests, however, that there is a problem: more and more IT personnel are misinformed or not bothered at all by the adoption of testing on their software. Tramontana et al. (2024) and Cammaerts et al. (2023) found that this is due to a **lack of emphasis on teaching procedures** in the personnel's professional or academic formation, and to a perception of testing as secondary—overshadowing other software life cycle steps as more important—consequently leading to developers finding software testing as a dull and boring activity Garousi et al. (2020).

It is important to ensure that current-day developers are well-equipped with software testing skills right from the beginning of their careers. It is then fundamental to **improve how these developers are introduced to the testing topics** in an educational context.

This chapter provides a brief introduction and establishes the contextual foundation for the remainder of this research work. It explores the current landscape of software testing education, examines learners' perspectives on various tools and methodologies, and considers how a specific approach—*mutation testing*—can enhance students' understanding of the subject.

1.1 Context

As organizations seek to deliver increased competitive advantages by providing reliable software products, the software testing market is poised for sustained growth, driven by leading companies' relentless demand for innovation and excellence in acquiring professional personnel equipped with a software testing know-how skillset Wadhwani and Ambekar (2024).

Educational institutions are starting to form professionals with this growing industry demand in mind Cammaerts et al. (2023). However, these schooling approaches are often considered very generic and theoretical, not tackling the practical side of things.

Research shows that new theoretical or practical concepts ought to be integrated **at the beginning** of an IT professional's career. A skill or schema introduced early in the cognitive journey of an individual allows for better transfer and more efficient learning later on and enables more effective mastery in the field van Merriënboer et al. (2005). Schools and universities are an optimal place for this early integration as part of the professional's educational journey.

In addition, the current student generation is very tech-savvy, so educational institutions must have this in mind when defining their teaching procedures; software testing is no exception.

> "It is challenging to teach software testing in a way that is engaging for students and to ensure that they practice effective testing sufficiently."
> Fraser et al. (2018)

The most challenging dilemma in the teaching of software testing is that students do not like learning about it Garousi et al. (2020). Many scientific papers have looked at this issue, defining different solutions for this problem, such as:

1. **Gamified learning**: mix "gaming" elements with lecture.
2. **Interface Simplification**: have a concise and straightforward user interface.
3. **Freedom to the student**: give the student control, allowing them to explore the tool and learn freely.

In order to introduce students to these topics in a pedagogically effective manner, it is essential to scaffold their learning journey by starting with foundational concepts. One widely adopted strategy in software engineering education is to begin with the most granular level of testing - **unit**

testing. This approach not only reinforces good programming practices but also aligns with the principles of autonomy and interactive learning outlined above.

1.1.1 Unit testing

Unit testing is when the smallest functional unit of code is tested Amazon (2025). This means that, for example, unit tests in a login page website would go over all the smallest components one could scrutinize from the view layout, such as the `Login` button. Although these tests are one of the most basic software testing principles, they could be one of the most important Amazon (2025).

On the one hand, they provide coverage to the source code, ensuring that written code lines are executed by that specific test. On the other hand, one of the most relevant features of unit testing is the ability to reinforce the maintainability and scalability of a system. This can be explained by the fact that they are designed to ensure the system functions as expected, and in the future, if changes are made to any system component that is covered by a unit test, there is reassurance in case any new logic breaks the past desired functionality. For instance, if a developer writes new code inside a functional unit, and some unit tests that were written before fail, that not only means that the developer broke the desired business logic for that component but also avoids shipping these faults into production since the unit tests acted as a protective security layer – **ensuring system maintainability.**

Even with unit testing playing a paramount role in software testing, Bai et al. (2021) infers that "unit testing is reported as one of the skills that graduating students lack". Students believe code coverage is the most important metric outcome for test suites Bai et al. (2021). As outlined in the previous paragraphs, unit testing encompasses a varied and complex ecosystem, and Bai et al. (2021) finds that most students demonstrate limited effectiveness in identifying **known defects** within the source code, they tend to only write unit tests to achieve good coverage metrics.

To help with **defect detection**, there is a concept in software testing named **mutation** that reinforces the ability of a learner to detect faults in the written source code.

1.1.2 Mutation testing

Mutation testing is a testing method that involves intentionally introducing faults into the code to measure the ability of a test suite to detect them Balfroid et al. (2023).

This is done by replacing source code with new faulty (or not) attributes – these replacement changes are called *mutators*. If this intentional replacement is done without being detected by any test, it means that the system might be vulnerable to this type of mutation at runtime. As Amalfitano et al. (2022) mentions, mutants act as proxies for real system faults. To illustrate mutants, Table 1.1 shows a list of *mutators* introduced by a well-known mutation tool, **PIT**:

Mutator Type	Original Condition	Mutated Conditional
Conditionals Boundary Mutator	<	<=
Math Mutator	+	-
Return Values Mutator	Object	null

Table 1.1: Example of Pitest mutators used in mutation testing

If a mutated conditional (or so-called *mutant*) survives, this typically indicates that the test suite is **deficient and needs improvement** Amalfitano et al. (2022); therefore, there is a need to write more test cases to try and tackle that survivor Tavares (2024).

These survivors, however, are not always an indication of flaws in the test suite, as some may be what are known as *equivalent mutants*. Equivalent mutants are syntactically different versions of the program that, despite the introduced changes, behave identically to the original code for all possible inputs Papadakis et al. (2014). Since they do not alter the program's observable behavior, no test case—no matter how comprehensive—can distinguish them from the original implementation. Detecting equivalent mutants is notoriously challenging, as it often requires deep semantic analysis or manual inspection, making them a well-known source of noise in mutation testing results Kurtz et al. (2016).

In the context of software learning, mutation testing can be very robust. Not only does it infer a certain sense of motivation in the person writing the test cases to try and kill the greatest number of mutants, but mutation testing can also be achieved by writing simple unit tests, accomplishing two goals simultaneously.

The current mutation testing tools' landscape offers different testing environments as well as a variety of mutators to dynamize mutation conditions. In the following list there are some of the market's most used mutation tools:

- **MutPy**: Python testing tool, supports various unit test frameworks.[1]
- **PIT**: Mutation testing tool for Java, integrating with Maven, Gradle, and JUnit.[2]
- **Stryker**: Modern web development mutation testing framework for JavaScript.[3]

All these mutation tools provide useful metrics that are relevant to the analysis of the research Tavares (2024):

- **Code Coverage**: the percentage of source code that has been executed by the test suite.
- **Condition Coverage**: the percentage of a conditional code block's logical paths evaluated by the test suite.

[1] https://github.com/mutpy/mutpy
[2] https://pitest.org/
[3] https://stryker-mutator.io/

- **Mutation Score**: measures the quality of the unit tests against the mutations created by the tool's mutators. It can be defined as the ratio of killed mutants divided by the total number of mutants (survivors and dead).

Metaphorically, mutation testing serves as a game where the user **has the challenge of writing tests** for **different manipulated contexts,** trying to achieve the **greatest test suite quality** to ensure that the system is well covered by software testing. Moreover, testers are also provided with a great insight into unit testing, as well as the importance of these implementations in exporting a quality-assured software product into the market.

1.2 Motivation

It is factual that the software market is striking at a fast pace Compton and Romanoff (2025), and more qualified software testers are desired in the future technologically dependent society that the following years hold Wadhwani and Ambekar (2024). Yet, software testing is still broadly neglected as a topic in computer science education Tramontana et al. (2024).

Furthermore, even with software testing being adopted as a subject in one's educational protocol, significant differences are found between the software testing skills required by the industry and those claimed by students Marin et al. (2022) Scott et al. (2003).

There is a need to find a solution that:

1. Strikes students with the importance of tests in the current software panorama Marin et al. (2022),
2. Does so in an engaging educational way
3. Values simplicity, reducing the initial learning curve for software testing.

> "Students need the opportunity to put what they learn into practice, using testing techniques and tools at all levels, from the individual unit to the system as a whole." Krutz and Lutz (2013)

1.3 Problem

As research shows Tramontana et al. (2024) Krutz and Lutz (2013), there is a lack of emphasis on practical testing education in comparison with theory and other educational protocols. Furthermore, students also tend to lack motivation when it comes to defining the advantages of software testing and the complex learning curve of its environment. This leads to students opting for the easiest solution – ignoring the development of tests for the software product at hand.

Moreover, Bai et al. (2021) found that in the scenario where students write test code, they tend to ignore setups and only test the happy path. This could prove to be a way of wrongly introducing testing concepts. **Software will never only be exposed to the happy path.**

There currently isn't a practical method to be introduced at the early stages of a student's education that aims to teach software fault detection, especially mutation testing (which is a strong opposition to the happy path), in a didactic and engaging way and without many struggles in the setup phase, which has been found to be demotivating for the student Garousi et al. (2020). Furthermore, no solution exists where multiple mutation tools are put up against the same environment.

The rapid evolution of software Compton and Romanoff (2025) leads to outdated research on the topic of teaching software testing. Professionals ought to be well taught to be equipped with the tools to test all the shipped digital features that are being exported at an extremely fast pace.

1.4 Goal

With the given problem, this research work aims to explore the development of a mutation testing framework where simplicity of setup, straightforwardness of user interface, and ease of use are the central focus. There is a need to ensure that the proposed tool is engaging for students and easy to integrate.

The choice of a **gamified approach** is backed up by current research, owing that it fits with the current tech-savvy generation.

Moreover, as presented before, students will **learn by killing mutants** since mutation testing is a 2-in-1 solution. It allows learners to test the smallest code units while also assuring extreme edge-case coverage.

Students and teachers will have the opportunity to adopt this solution in their course procedures, with the end goal of **changing the learners' perception** of software testing as secondary.

This research aims to reinforce and motivate the development of software testing frameworks that are used in educational protocols and shape the future generation of software professionals.

1.5 Document Structure

This dissertation contains 4 more chapters:

- **Chapter 2**: Analyzes the current state of the art and research on software testing learning, with a focus on tools and frameworks used in real-life scenarios. Based on this analysis, a proposal to enhance an existing tool is made to address identified research gaps.
- **Chapter 3**: Describes the implementation of the proposed tool. Includes technical constraints and design diagrams to illustrate the framework development process.
- **Chapter 4**: Validates the tool's requirements with external feedback and objective benchmarks that will demonstrate enhanced value and performance.
- **Chapter 5**: Presents conclusions drawn from this research and discusses future improvements to further enhance the described tool.

Chapter 2

State of the Art

There is a need to have an overview of how software testing is currently taught in order to scrutinize the lacunas and generate a bulletproof, robust solution.

First, this chapter will go over how software testing teaching has evolved through the years, as well as the methodologies that are currently adopted to introduce this concept to learners.

Secondly, the analysis will delve further into the current state of teaching tools with a special focus on mutation testing frameworks.

2.1 Research Process

To investigate the current state of the art in the domain of software testing, the following keywords were identified as central to the literature review process:

1. Software testing
2. Teaching `AND/OR` Learning
3. Mutation testing
4. Tool `AND/OR` Framework

The primary databases utilized for this investigation were *Scopus*[1] and the *IEEE Xplore Digital Library*[2]. The search strategy was designed to incrementally refine the scope of results, thereby isolating the most relevant publications for the present study. Table 2.1 summarizes the progressive narrowing of search results using *Scopus*, based on the combination of core keywords.

Keywords used	**1 and 4**	**1, 2 and 4**	**1, 2, 3 and 4**
Number of Documents	21322	2430	62

Table 2.1: Number of Documents Retrieved in *Scopus* Using Different Keyword Combinations

[1] https://www.scopus.com/
[2] https://ieeexplore.ieee.org/search/

This process served as a foundation for constructing a focused and relevant body of literature for analysis. The subsequent sections adopt a **top-down** analytical approach: beginning with a **broad overview** of software testing education and progressively exploring specific subdomains, including pedagogical practices and mutation testing methodologies and toolsets.

2.2 Teaching Methodologies

The lecturing process on the topic of software testing is not limited to educational courses. That is the tip of the iceberg. There are multiple teaching methodologies that will be analyzed in this section.

2.2.1 Education in Software Testing

Addressing the fact that there is an ever-growing need to train highly skilled software testers, universities have started integrating software testing programs into their education curriculum Tramontana et al. (2024). This methodology appears to be the most prevalent, with testing concepts blended into programming and other courses Garousi et al. (2020).

Garousi et al. (2020) and Cammaerts et al. (2024) found that testing methods were spread in different ways when it comes to testing courses:

1. **Independent Testing Course:** Course is fully focused on teaching testing concepts. It delves deeper into software testing and covers the resources in detail. However, software testing is found to be "tedious"; therefore, there is a challenge in motivating students, often making them miss out on relevant aspects of the course.

2. **Testing is integrated into one SE Course:** Testing concepts are given throughout a different programming course, that is not solely focused on testing.

3. **Testing is integrated into multiple SE Courses:** This approach spreads the teaching of software testing throughout multiple courses in the curriculum. It allows for a holistic approach that aims to easily provide **different contexts** where testing can be used, out of the box.

Garousi et al. (2020) also found that independent testing courses are not practical overall:

- What is learned in a single course can easily be forgotten, due to the **high emphasis on theory**.

- Usually, universities tend to delegate all the testing education into these individual courses, which is commonly not enough for students to absorb useful knowledge.

A proposed solution to this problem is to integrate software testing through multiple educational courses. This means that students can **implement tests given different contexts**, providing

them with a better and practical understanding of the importance of software testing. This solution also implies that these concepts are learned at an **early stage** of the student's programming journey and reinforced throughout the way.

Garousi et al. (2020) suggests that there is a tradeoff: students can easily get overwhelmed with mixing testing concepts with the actual information of the course being lectured on. In order to ensure that students do not find the subject complicated, there is a need to balance the theoretical elements with an objective practical application of these concepts.

2.2.1.1 Theory and Practice

To avoid theory dullness, there is value in making software testing courses practical and close to industrial needs. However, in doing so, multiple challenges arise:

- How to keep the protocol up to date with the most recent utilities?
- How should theory and practice be split and combined in these testing courses?

Testing is intrinsically a practical activity Garousi et al. (2020). But students can't fully dive into practice without having a theory scaffold. **These two concepts ought to complement each other**, and teachers should strive to explore the best alternative according to their university's culture and educational goals Cammaerts et al. (2024).

One thing is undeniable by research: **students ought not to get overwhelmed** Garousi et al. (2020). This rapidly decreases the learner's motivation and immediately gives the course a bad impression, which is passed on as feedback among the student community, disrupting the course's reputation and making fewer students enroll in testing activities.

Cammaerts et al. (2024) found that students are more engaged in learning software testing with **real-case studies**; however, these are not trivial to find—either by the learner or the instructor—and they usually imply a complex setup infrastructure, hindering the student's experience.

With these challenges on the horizon, investigators have devised multiple approaches to **combine the best of both theoretical and practical worlds** in straightforward, simple teaching methodologies. One of them is named gamification.

2.2.2 Gamification

Students are very fond of the gaming world. Gamification is the process of enhancing systems, services, organizations, and activities by integrating game design elements and principles in non-game contexts. The goal is to increase user engagement, motivation, competition, and participation[3].

Papers have analyzed the efficacy of software testing as a game. Gamification has *de facto* been trendy in recent research as a teaching methodology, owing to how familiar students are with

[3] https://en.wikipedia.org/wiki/Gamification

gaming activities Cammaerts et al. (2024). This method may, in fact, **increase motivation for students** learning software testing Garousi et al. (2020).

Some examples of gamification elements used by teachers in courses include Garousi et al. (2020); Scatalon et al. (2020a):

- Leaderboards
- Badges
- Reward Points
- "Quests" or hints in the format of unit tests

All of these components have one thing in common: they make the education of testing concepts more engaging and enforce a sense of "good competition" among students, which can lead individuals to strive toward better results.

However, Garousi et al. (2020)'s research presents a tradeoff: students may "game" the assessment system, adding a facetious facet to this educational method, making it possibly be perceived as less serious.

A few examples of educational testing games are *CodeDefenders*, *TestEG*, and *HALO*, some of which will be analyzed in the following sections.

2.2.3 Free Open-Source Projects

Given the courses' duration and limited teaching schema, inexperience and lack of knowledge in industry-level tools and projects become a weakness for many students Deng et al. (2020). Students may struggle to appreciate the importance of software testing when working **on small-scale coding assignments**. As a result, several studies advocate for incorporating free or open-source software (FOSS) projects into software testing education Garousi et al. (2020). These projects not only expose students to testing in a more realistic setting but also allow them to **work with code written by others**. Identifying bugs in someone else's code can be more engaging, as it avoids the discomfort of confronting one's own mistakes:

> *"When testing their own code, students are less motivated to find bugs, as bugs expose their own failure to develop a correct program."* Ochoa and Salamah (2015)

In addition, students who contribute to FOSS projects experience the effects and accomplishments of their work on **real-world software and its users**. Participating in FOSS communities also provides opportunities to learn collaboration skills from more experienced developers and to develop their professional network and relationships.

A software testing methodology that integrates FOSS may follow this structure Deng et al. (2020):

- Fork an existing project and run the unit tests
- Design new unit tests to achieve better code coverage
- Write bug reports and get familiar with issue tracking systems

By incorporating FOSS into software testing education, students are introduced to real-world software projects, including actual software defects and professional-level testing methods used in the industry. This approach provides them with **practical, hands-on experience**.

2.2.4 State of Research

With all the presented educational methods for teaching software testing, current research gives emphasis to two specific terms:

1. How to teach software testing better
2. Tools for teaching

Figure 2.1 classifies recent studies by contribution types. It is clear that investigators are focusing on pedagogical approaches and the development of tools and frameworks that enable better teaching.

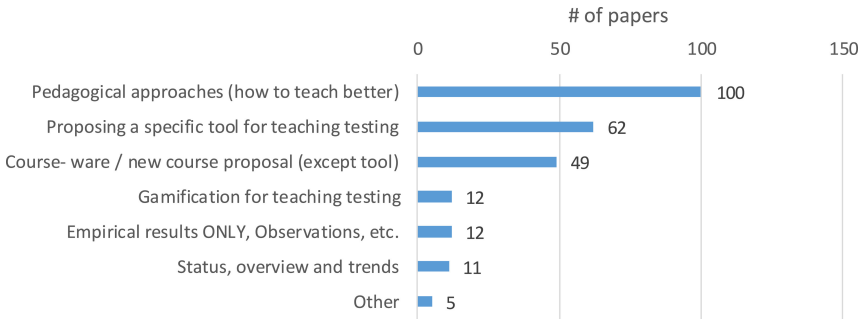

Figure 2.1: Classification of studies by contribution types [taken from Garousi et al. (2020)]

Furthermore, gamification and trends such as FOSS are also present in current research, though in a less prevalent manner.

Considering these points, it is important to recognize that a well-rounded approach—**blending aspects of various educational methods**—may offer the most effective strategy for teaching software testing. This allows the instruction to be adapted to the current cutting-edge context and the evolving needs of students Cammaerts et al. (2024), Garousi et al. (2020).

As the emphasis in recent studies shifts increasingly toward practical support mechanisms for teaching, the development of tools and frameworks has become a central theme. These resources not only facilitate more effective instruction but also expose students to **techniques used in professional environments**.

In the following section, the presented research work explores in greater detail the specific tools and frameworks designed to teach testing concepts, one of the key strategies for enhancing students' understanding of software testing practices.

2.3 Software Testing Tools

A software testing tool is a program or application that supports the process of verifying and validating that a software system behaves as expected. These tools can assist with tasks such as test case creation, execution, automation, defect tracking, and performance analysis.

An approach frequently seen in earlier research involves utilizing one or more tools to facilitate both the teaching and evaluation of software testing. It is recommended to employ widely adopted industry-standard tools to teach students proper usage and configuration within real-world environments Cammaerts et al. (2024), Garousi et al. (2020).

Moreover, engaging in testing requires students to **become familiar with new tools and libraries**, something that can be particularly challenging during the early stages of a degree program. Beginners often find it difficult to navigate these tools, especially when they are still grappling with fundamental programming concepts. As a result, students may feel overwhelmed Garousi et al. (2020) by the complexity of software testing tools.

2.3.1 Languages

Current software testing tools are written in many different programming languages.

Scatalon et al. (2020b) found a big emphasis on testing tools that use Java compared to all the other existing programming languages (Figure 2.2).

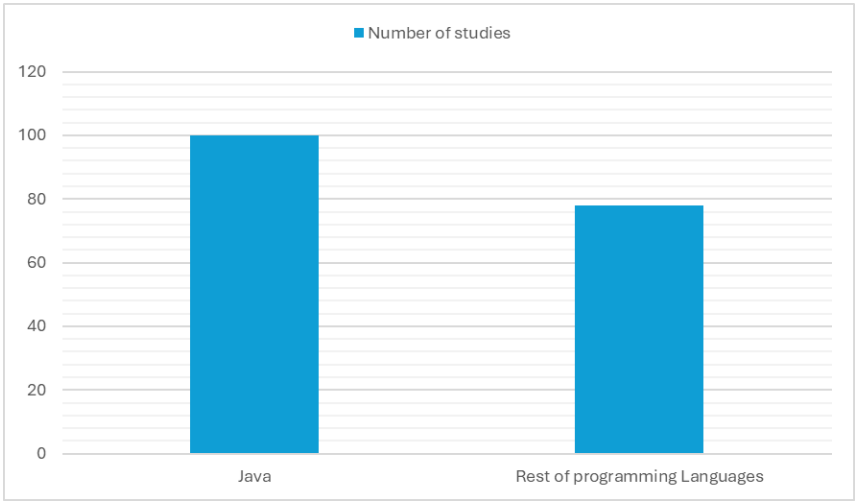

Figure 2.2: Programming languages used in software testing tools' studies

2.3.1.1 Java

Java's platform-independent nature (via the Java Virtual Machine, or JVM) allows test scripts and testing tools to run on any operating system without modification. This is crucial in software testing, as tests must be run in different environments (Windows, Linux, macOS, etc.) for validation.

Furthermore, Java possesses **one of the most reputable testing frameworks available** – *JUnit*. In fact, this framework is commonly referred to as the de facto standard for most software testing tools available Cammaerts et al. (2024).

JUnit is widely used in testing tools due to its simplicity, reliability, and seamless integration Cammaerts et al. (2024). As a straightforward framework for unit testing, it enables developers to write and execute repeatable tests efficiently, supporting test automation and continuous integration processes. Its popularity also contributes to frequent updates and strong toolchain compatibility.

With a robust ecosystem and active community support, Java still holds the reference spot as the best option for developing a software testing tool. Consequently, it has been present in most research case studies in the past years (Figure 2.2).

Bridging mutation testing and Java, Amalfitano et al. (2022) finds there are numerous mutation tools for Java, which in turn makes it harder to elect only one suitable choice for different contexts. Furthermore, Amalfitano et al. (2022) presents a framework that compares many of these mutation tools, giving particular positive emphasis to *Major*, *PIT* and *Jumble*.

2.3.1.2 Other languages

Aside from Java and *JUnit*, Python is a highly popular technology in software testing due to its readability, simplicity, and vast library support. Frameworks like *MutPy*, `unittest`, and *Robot Framework* make it easy to write automated tests and perform both functional and acceptance testing. Python is also commonly used with tools like Selenium for web automation and integrates well with CI/CD systems.

JavaScript is another major player in the testing world, particularly for web applications. With the rise of front-end frameworks like React, Angular, and Vue, JavaScript testing tools such as Jest, Mocha, Chai, and Cypress have become essential.

Mutation testing is also expanding to mobile development languages. Vincenzi et al. (2025) present an **Android mutation testing framework** developed to execute test cases and determine mutation scores, with the aim of fostering research in mobile native mutation testing.

This research does not focus on mobile applications nor on automated CI/CD testing. It references the development of software testing tools. And for straightforward, simplistic implementation, Java and *JUnit* still hold their ground as the most viable option in the current market Scatalon et al. (2020b).

2.3.2 Tools and Frameworks

With the research on teaching software testing, the development of new tools capable of handling this educational process is ongoing.

A 2019 study found that there was a growing tendency for research on new educational tools for testing. Specifically, mutation testing has experienced a paramount increase in studies throughout the last 20 years (Figure 2.3).

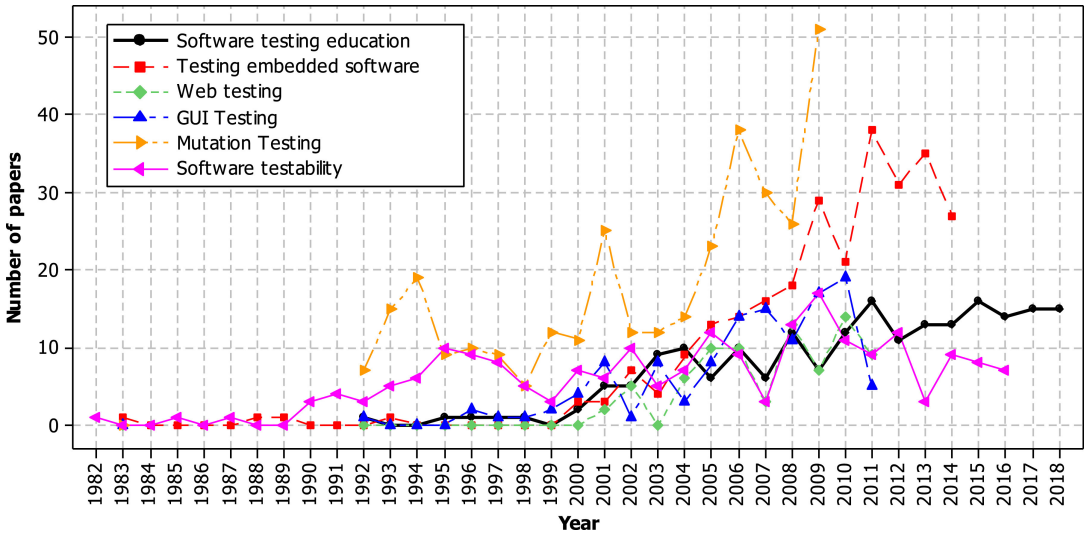

Figure 2.3: Number of research papers in different software testing topics [taken from Garousi et al. (2020)]

A common approach for these tools is to check the code coverage of the implemented tests. However, code coverage alone is not a satisfactory measure of test quality. For example, **a student could write tests without assertions on the expected outputs**, and such a significant issue would not be detected by code coverage alone. Advanced testing techniques, such as mutation testing, could help with this Garousi et al. (2020).

Mutation testing has been proven to be an effective, robust, multi-faceted solution to teach software testing Amalfitano et al. (2022), as it was displayed in the previous chapter. Therefore, the following sections will review existing mutation testing tools and analyze their flaws to find gaps that ought to be solved by the solution presented by this research.

2.3.2.1 Bacterio

Bacterio is a Java-based mutation testing tool designed to automate mutation analysis and optimize its performance. It implements efficient mutation techniques that significantly reduce both the computational cost and execution time of mutants. This makes it a practical and scalable solution for evaluating test quality while maintaining low overhead in the testing process Mateo and Usaola (2012).

Mutant generation by itself is notoriously a costly process. Firstly, all the mutants are created for the desired codebase. Next, testers **execute all the test cases against the original codebase** and verify which mutants were killed Mateo and Usaola (2012).

Bacterio aims to solve this by implementing multiple, cutting-edge techniques that reduce the costs of mutation testing. Consequently, mutation testing becomes much faster to execute, and testers benefit from a smoother experience Mateo and Usaola (2012).

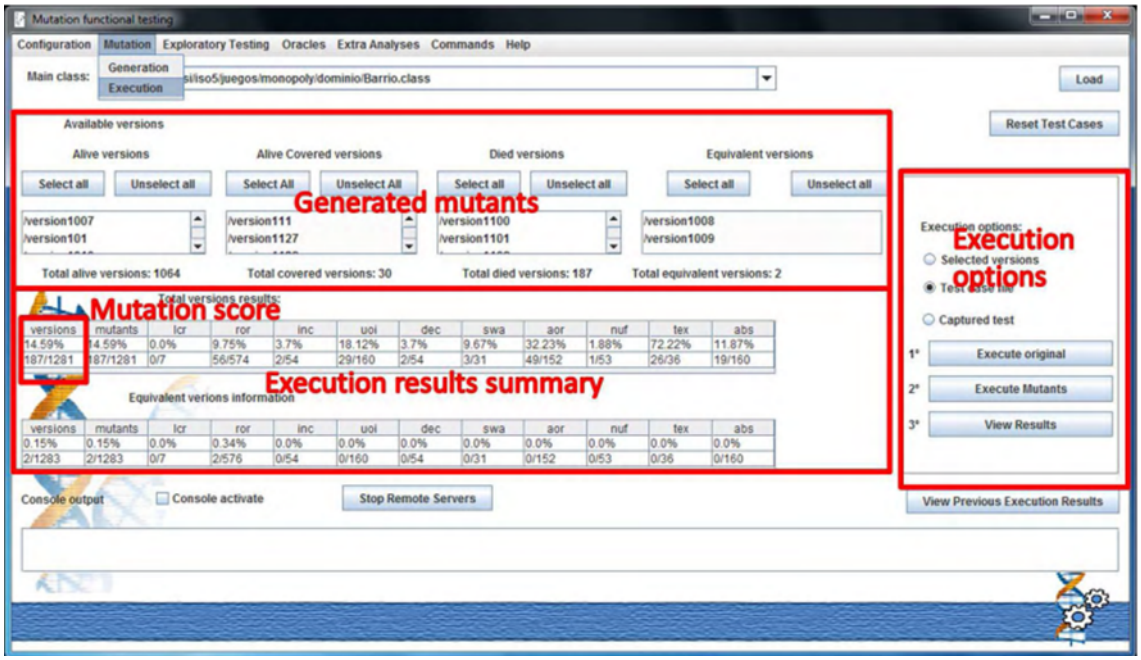

Figure 2.4: Bacterio's user interface layout [taken from Mateo and Usaola (2012)]

Figure 2.4 displays the user interface for the Bacterio tool. Immediately, the tester is shown multiple panels. A new learner may not find this experience very captivating. Passig and Nadler (2010) found that overly complex interfaces can impede effective learning.

Therefore, Bacterio's attributes can be summarized as follows Mateo and Usaola (2012):

Pros:

- A wide array of mutation strategies that support testing across various system levels.
- Parallel execution capabilities, aiming to improve mutation performance.

Cons:

- Complex user interface.
- Limited environment support and scalability.

2.3.2.2 CodeDefenders

CodeDefenders is a web-based game designed to apply gamification principles to mutation testing Rojas and Fraser (2016). In this game, two players engage in a competitive scenario centered around a single program under test: the *attacker* aims to design subtle mutants, while the *defender* focuses on developing an effective test suite. The attacker earns points for introducing mutants that evade detection, whereas the defender scores by writing tests that successfully identify and eliminate those mutants.

Using Java and JUnit, CodeDefenders emphasizes the relevance of gamification in teaching mutation testing through a competitive, score-based dueling approach Rojas and Fraser (2016).

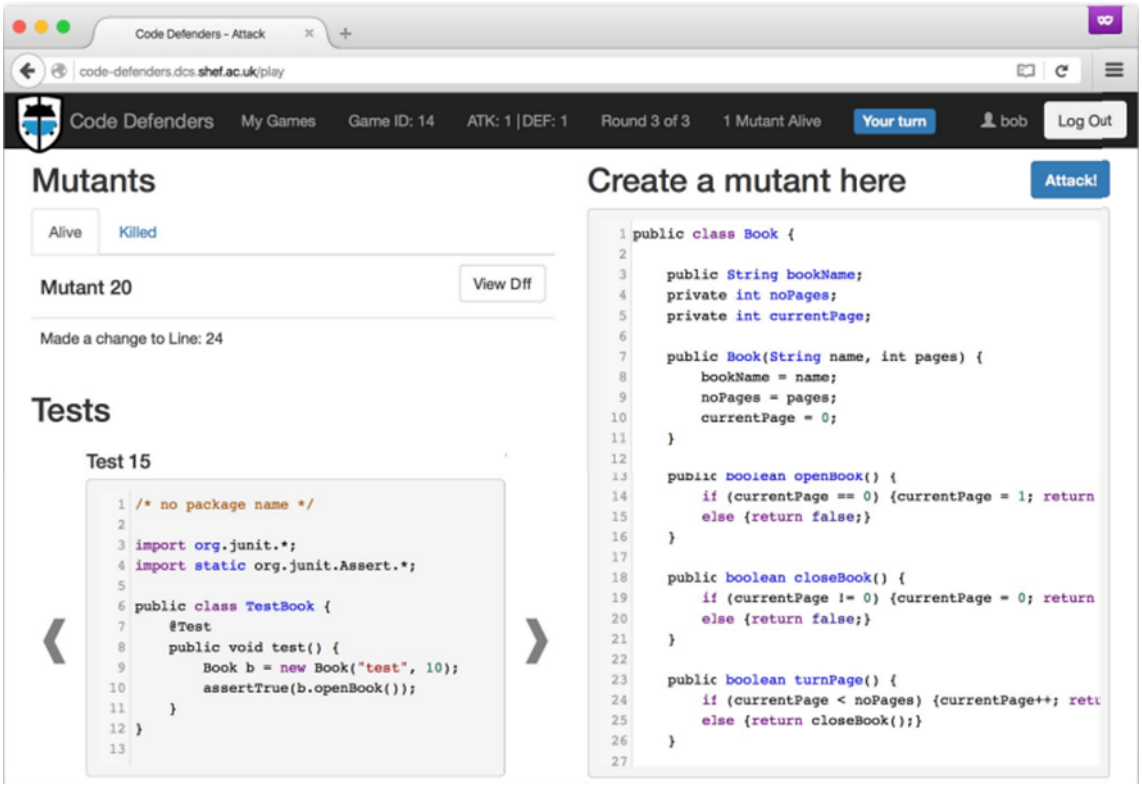

Figure 2.5: Code Defenders' user interface layout

Figure 2.5 demonstrates the program's straightforward user interface, which features a few distinct panels and views.

Concluding, this tool can be evaluated with:

Pros:

- Implementation of gamification to improve user engagement.
- Simple and straightforward interface and usage.

Cons:

- Limited infrastructure — as noted by the tool's developers Rojas and Fraser (2016): "playing CodeDefenders requires certain knowledge of Java and JUnit; this not only reduces the number of potential players of the game but also limits its playfulness."
- Monotonous learning over time due to a finite gamified approach.

2.3.2.3 Code Immunity Boost (CIB)

Code Immunity Boost (CIB) is a gamified learning system for mutation testing, inspired by vaccine development. It integrates gamification to enhance student engagement and understanding of mutation testing concepts, where students assume the roles of leukocytes (test cases) and vaccines (mutants).

Furthermore, the CIB tool automates much of the mutation testing process, which is traditionally slow and tedious when done manually Garousi et al. (2020). A fast feedback mechanism encourages students to fix errors and try new test cases, promoting a trial-and-error-based learning process Hsueh et al. (2023).

CIB is developed in Python using the `unittest` framework Hsueh et al. (2023). Its primary interface includes a *Code Editor*, which allows users to submit test cases. These are compiled and executed to generate key metrics such as the mutation score — the percentage of mutants successfully killed by the test suite.

CIB can be resumed in:

Pros:

- High motivation: CIB leverages gamification to stimulate students' interest and participation in mutation testing.
- Improved efficiency due to automation of the mutation testing process.

Cons:

- Students may not fully appreciate the intricacies of mutation testing due to the automated processes and absence of code handwriting.
- Potential for confusion due to complex domain metaphors (e.g., leukocytes, vaccines).

2.3.2.4 FRAFOL

FRAFOL is a **FRAmework FOr Learning Mutation Testing**, designed to provide learners with a simple and effective environment for practicing mutation testing. The tool emphasizes ease of integration and aims to support any educational protocol that includes software testing as part of its curriculum Tavares et al. (2020).

The framework builds on the widely used *Defects4J* repository [4], which contains reproducible bugs from **real-world Java projects**. This infrastructure allows students to practice test case development against realistic software defects. Appendix A contains detailed information about this framework and what benefits it brings to the software world.

FRAFOL operates as follows:

1. Learners select a buggy class from one of the collections in the Defects4J repository.
2. FRAFOL mutates the buggy class and displays a list of generated mutants, along with metadata such as applied mutators and the affected lines of code (see Figure 2.6-D).
3. Students then write JUnit test cases aimed at killing the generated mutants (Figure 2.6-B).
4. The test suite is executed against the selected buggy class, and the tool reports which mutants have been killed.

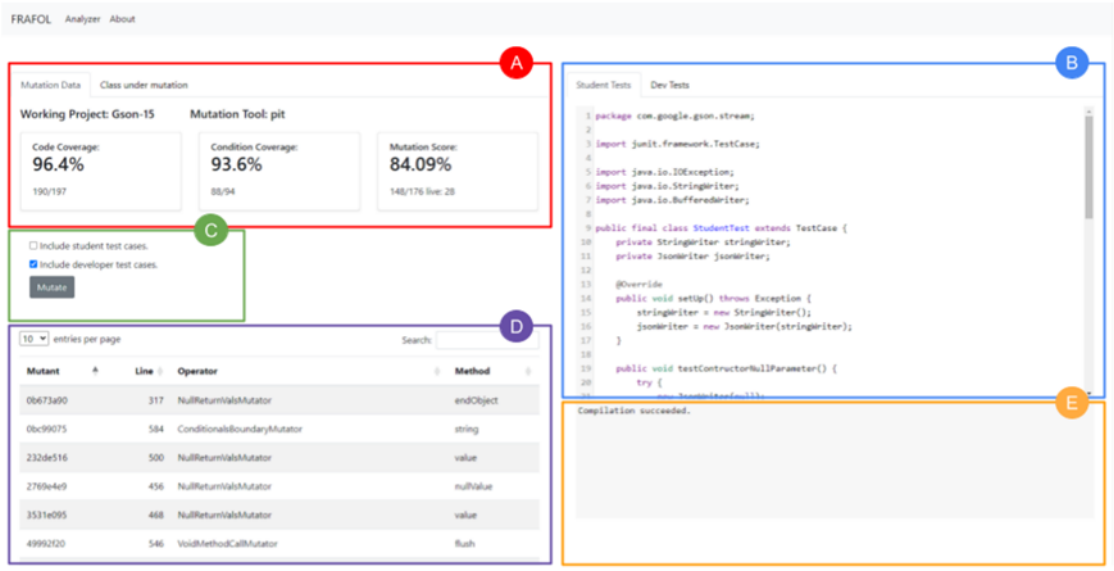

Figure 2.6: FRAFOL's user interface layout

FRAFOL's mutation testing uses Java and relies on the *JUnit* framework. It also integrates with multiple mutation tools, including **Major and PIT** (Amalfitano et al. (2022) highlighted these two tools as the standout tools for the current panorama of mutation testing), thereby offering flexibility and variety in the learning experience.

This framework was a result of research that had the goal of "*experimenting with mutation tools without the burden of complex configurations*" Tavares et al. (2020).

2.4 Discussion

This research aims to extend the work that was previously done on FRAFOL, implementing facts-based features that aim to fill any lacunas found in the current state of the art on the topic of software testing learning, especially mutation testing, as well as enhancing the current tool with innovative use cases. These are the main takeaways from the following sections, and how the FRAFOL tool, developed by this research work, tackles all of them.

2.4.1 Bridging Theory and Practice

Garousi et al. (2020) presented that a mix of theory and practice may be the best approach to teaching software testing. Current educational methods tend to over focus on theory, which can lead to high demotivation Marin et al. (2022). Research also found that students are more engaged in learning software testing with real-case studies Garousi et al. (2020).

[4]https://github.com/rjust/defects4j

Here, FRAFOL **serves as a bridge** between the knowledge that the learners grasped during theoretical classes and the engaging practice that they find in the writing of unit and mutation tests using the platform.

2.4.2 Unveiling Gamification

FRAFOL allows students to write these test cases in a gamifying manner. The tool generates a list of mutants and challenges the learner to write unit tests that will progressively reduce the number of live mutants in the list.

Gamification has been adopted by multiple research tools owing to the education of testing concepts being more engaging and enforcing a sense of "*good competition.*" Garousi et al. (2020).

Students may perceive this method as less serious Garousi et al. (2020), however, with the right amount of gamification concepts, FRAFOL tries to find a good balance in its approach.

2.4.3 FOSS for real-world context

As it has been found by Garousi et al. (2020), FOSS allows for the integration of real case studies into software testing. This not only gives the learners a perspective that they could eventually adopt in their professional career, but it also makes it more engaging since students are getting a sense of accomplishment by **solving real-world problems**.

FRAFOL encapsulates multiple real case studies by adopting the *Defects4J* framework, which contains a collection of real-life projects. This broadens the tool's horizons while keeping it up to date with trendy collections, assuring the point discussed in section 2.2.1.1, which questioned how to keep teaching protocols up to date with the most recent utilities.

2.4.4 Lacunas of current mutation tools

FRAFOL aims to fill any gaps in current research, specifically, mutation tools that already exist. On the one hand, **Bacterio** proved to be an efficient solution in increasing performance on mutation testing Mateo and Usaola (2012). On the other hand, its overly complex interface can impede effective learning Passig and Nadler (2010).

Motivation is fundamental for students in software testing Garousi et al. (2020). Therefore, **FRAFOL** aims to improve its usability by adopting a **simple interface** where users can easily navigate different tabs.

Furthermore, **CodeDefenders** notoriously enforced the gamification protocols for mutation testing. With a straightforward layout, it ensured the learner had a captivating experience while being taught the concepts of software testing.

Research found that **CodeDefenders** had a limited scalability infrastructure, which could "*limit its playfulness*" Rojas and Fraser (2016). With these findings, **FRAFOL** aims to integrate a dynamic ecosystem with the **support of multiple mutation tools** and a varied collection of real case studies, which allow for the tool to scale in an unlimited manner.

Lastly, **CIB** (Code Immunity Boost) brought yet another gamified approach to research. Findings were conclusive that gamification has a positive impact on software testing learning Hsueh et al. (2023).

However, the same study also found that students may not fully appreciate the intricacies of mutation testing due to the lack of code handwriting Hsueh et al. (2023). **FRAFOL** aims to fill this lacuna by **implementing a code editor** where students can write any test case that will target live mutants.

2.4.5 All-rounder and further improvements

This research aims to enhance the current state of FRAFOL. FRAFOL's last research document, in 2024, enforced multiple improvements in relation to the framework's future work Tavares (2024).

2.4.5.1 Mutation Enhancement

"*One key area for improvement is the live mutant report table.*" Tavares (2024). The research found that there could be a **clearer separation** between killed and live mutants, so that the user would always be synchronized with how the written tests interact with the mutants, as well as have a better overview of what tests could be more effective at killing mutants.

Furthermore, another improvement would be a more insightful approach related to mutational operands for each tool Tavares (2024). This aims to minimize the reliance on external documentation, thereby enhancing the overall user experience, by enforcing a stronger dependence on the framework.

2.4.5.2 Code Editor Experience

On the one hand, providing **color highlighting for code coverage** would significantly increase the user's engagement with the tool in a visual manner Tavares (2024). On the other hand, it would also aid the user in the task of better understanding how certain coverage conditions may be directly correlated with live mutants, enforcing the writing of cohesive tests that aim to achieve the maximum number of green lines (where green stands for a line that has been covered by a test).

2.4.5.3 Teacher View

FRAFOL was focused on student work and the usage of mutation testing to enhance software testing learning. However, to achieve the best of both worlds, the authors of FRAFOL mentioned Tavares (2024): "*Implementing a teacher's view for automated analysis of student work would greatly benefit educators.*".

Not only would this innovate the framework's educational protocols by bridging the gap between the students and the teachers, but also enhance FRAFOL's scalability and credibility, making it more suitable for **adoption across various educational methods**, thereby positioning this research as a **significant contribution to the field of software testing**.

This feature could be implemented in a class-like manner, where students would write tests, kill mutants, and submit their code. Teachers would then analyze the code and provide insightful feedback to the students. Furthermore, lecturers could also adopt a grading system according to the students' submissions or even gather useful analytics to improve the course's value in the years to come.

2.4.5.4 Standout Features

The enhanced version of FRAFOL will not only bridge the gaps mentioned in the last paragraphs but also expand the research and implement innovative software-testing related features in the system, which have not been investigated by any prior tool.

One aspect that differs FRAFOL from the rest of the mutation testing frameworks is the fact that it enables a dynamic testing ecosystem where users can take advantage of **multiple mutation testing tools** that coexist in the same system. This broadens the system's scalability, since it seamlessly encapsulates different tool sets that cater to the same objective of processing mutation testing, **while abstracting any setup or complex configuration** implied by other existing mutation tools.

Moreover, FRAFOL will be the **first mutation testing framework to have relevant features for both students and teachers**. This will dynamize the class environment, marking FRAFOL as a standout alternative to be adopted in various educational protocols around different institutions. These new attributes will be thoroughly analyzed in the next chapter.

The title of this dissertation is *"Framework for Teaching Mutation Testing"*. As the latter paragraph emphasized, teaching would be one of the main focuses of future research into FRAFOL. In the following chapter, this work will examine the **technical and implementation constraints** considered during the development and enhancement of the FRAFOL mutation framework. It will also explore how the investigation sought to realize the future improvements proposed by previous research on the state of the art of software testing, analyzed in this chapter.

Chapter 3

Restructuring FRAFOL

Extending the previous work conducted through FRAFOL was the main target of this work. This chapter will go over some key aspects of building the new version of this framework, as well as the challenges encountered in the design and implementation phases of the software.

3.1 Analysis

It is fundamental to have a clear view of how FRAFOL was structured to allow for the easing of future improvements. FRAFOL is a web system. This tool's backbone was developed using Python; more specifically, it encompasses the Flask framework. Moreover, this system integrated *Defects4J*, which served collections of mutants and buggy classes. Lastly, the application's deployment process involved Docker to support the tool's idea of ease of usage.

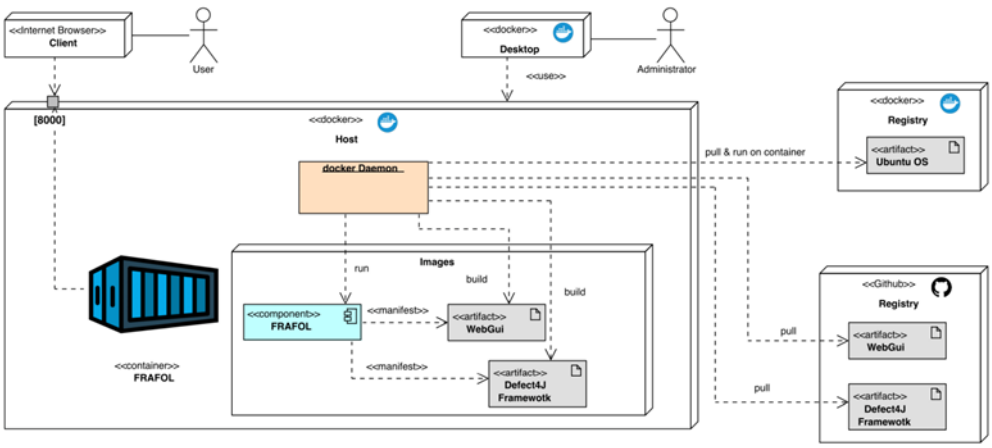

Figure 3.1: FRAFOL's old system architecture

It is worth noting that, as Figure 3.1, suggests, the view component of the application is handled entirely by Flask. Flask supports simple HTML web views that serve as frontend to the logic

handled by the framework's backend API. An example of the FRAFOL's web view can be seen in Figure 2.6.

With FRAFOL's architecture considered along with the future improvements discussed in the previous chapter, this research outlines the primary new use cases driving the restructuring of the web application:

- **Code Viewer Revamp:** The user should be able to visualize color-highlighted sections of the code to represent the current code coverage of the class under test. Furthermore, clicking on a mutant entry in the *Live Mutant Table* (see Section D in Figure 2.6) should redirect the user to the specific line of code where the mutant resides.

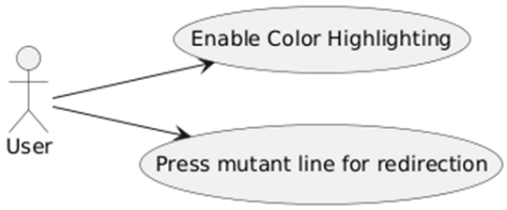

Figure 3.2: Code Viewer related use case diagram

- **Enhanced Mutation Interpretation:** To improve learning outcomes, the tool must present its output in a more digestible and informative manner. A *Kill Matrix*, mapping each test to the mutants it kills, would offer users a clearer understanding of the effectiveness and scope of their test cases.

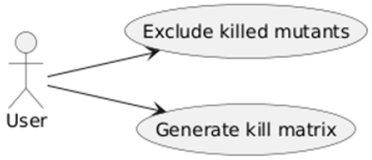

Figure 3.3: Mutation related use case diagram

- **Teacher's View:** A dedicated teacher interface should allow instructors to log in with given credentials. Once authenticated, teachers can:
  - Create or select a class.
  - View the list of enrolled students.
  - Admit or remove students from the class.
  - Inspect each student's code submissions.
  - Analyze information matrices related to the student's performance

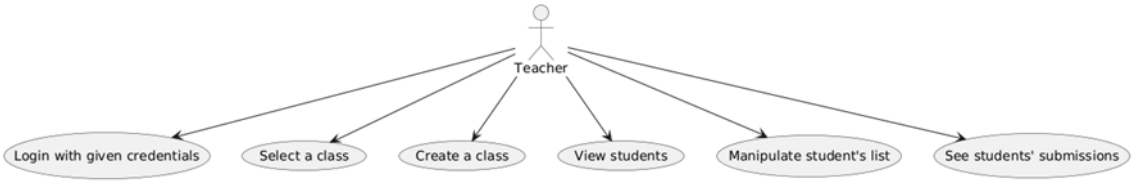

Figure 3.4: Teacher's view related use case diagram

- **Student's Submission Portal:** Students must be able to register for a new account or log in to an existing one. After joining a class and being admitted by the respective teacher, students should be able to submit code for specific mutation testing exercises or projects.

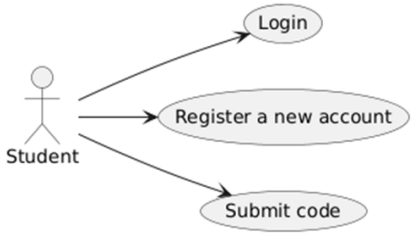

Figure 3.5: Student related use case diagram

These new functionalities are designed to enhance FRAFOL as an educational framework, empowering both students and instructors with dynamic and interactive testing protocols.

However, this research revealed that integrating such features within the current architecture **presents considerable challenges**. The next section discusses the technical strategies adopted to overcome these obstacles and introduces the new system design envisioned to support this expanded feature set.

3.2 Design

FRAFOL's limited frontend view (with *Flask*'s simple HTML environment) limits the capacity to adopt new features into the system. Furthermore, for a system to scale, it ought to have decoupled components so that these do not depend on others and can grow independently. This research work implemented a new architecture (Figure 3.6) for the system:

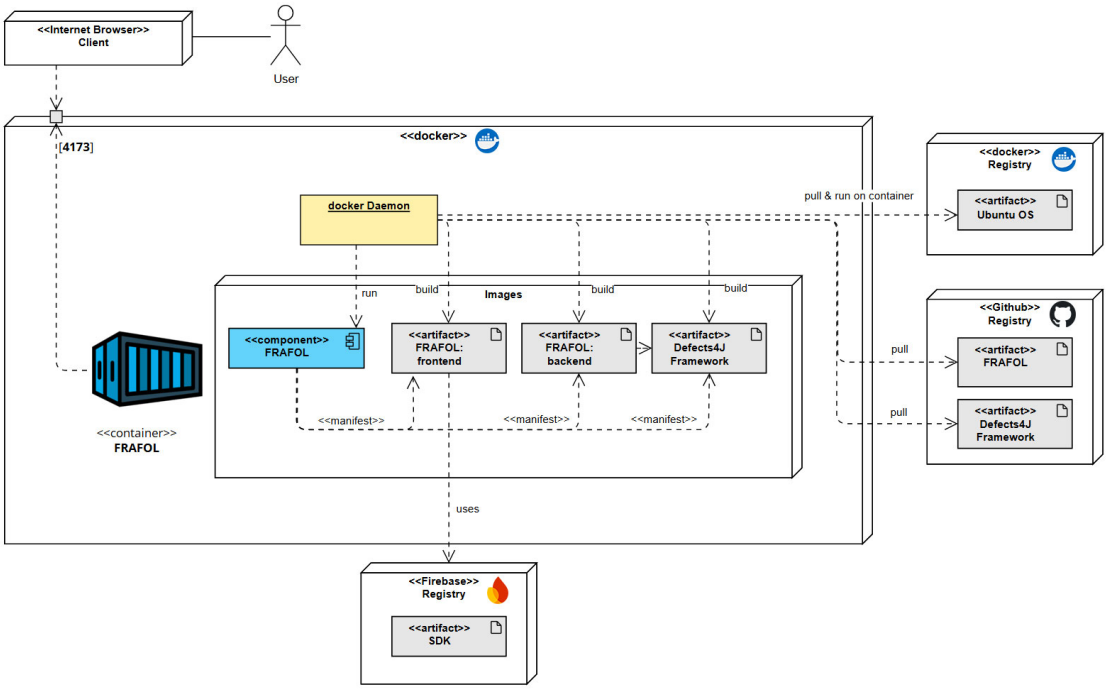

Figure 3.6: FRAFOL's renovated architecture

1. **Separated Frontend and Backend Images:** A complete refactor was performed, resulting in the decoupling of the frontend and backend components. The frontend was restructured as a Single Page Application (SPA) using *React.js*. This change enhances the system's scalability and flexibility, enabling easier integration of new features due to React's wide adoption and ecosystem support within the JavaScript community. The frontend now communicates with the backend through RESTful APIs (Figure 3.7). The backend remains implemented in *Flask*, which delegates the mutation processing tasks to the *Defects4J* framework.

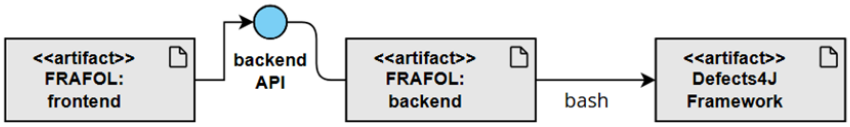

Figure 3.7: System's images inter-communication

2. **Integration of Firebase SDK:** To implement the authentication and data storage functionalities required by the proposed use cases, the system now integrates the *Firebase Software Development Kit (SDK)* - Figure 3.8. Firebase provides serverless backend services, which allow the system to handle:

   - User authentication (login and registration)
   - Secure storage of students' code submissions

   This integration **reduces the overhead** of maintaining separate authentication or database infrastructure and leverages Firebase's real-time capabilities and scalability.

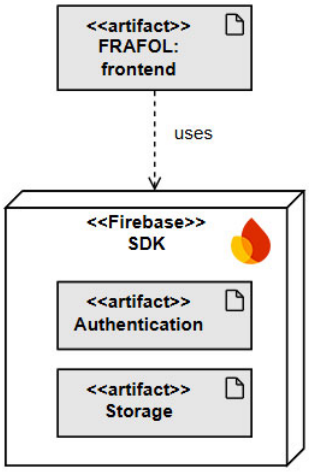

Figure 3.8: FRAFOL's usage of the Firebase SDK

3.2.1 Firebase

FRAFOL required a robust toolset capable of **handling both authentication and storage functionalities**. This study examined multiple client-side SDKs suitable for these integrations.

Clerk [1] is a renowned suite of embeddable UIs, flexible APIs, and admin dashboards that enable user authentication and management. It is widely adopted and trusted by fast-growing systems around the world. Its ease of integration makes it a promising candidate for inclusion in FRAFOL's new *React.js* frontend environment.

However, from an architectural perspective, if a **single integration can provide multiple features**, there is no benefit in coupling the system with additional entities that would unnecessarily increase dependency complexity. While Clerk provides all necessary features for managing authentication within FRAFOL, it lacks built-in support for storage.

As a result, *Firebase* emerged as the most appropriate choice for the system. Firebase's SDK provides both authentication and storage capabilities out of the box and enables direct configuration of cloud-based server properties. This proved to be essential for implementing the analyzed use cases, as it delegates critical responsibilities such as **reliability and security** to *Google*, the provider of Firebase.

This choice is particularly significant given FRAFOL's current *Docker*-based architecture. Since users have access to the source code and can manipulate the container images, delegating these security constraints to Firebase's cloud infrastructure ensures that client-side manipulation is not possible. This is enforced by Firebase's *Firestore Database Rules* and *Role-Managed Authentication*, which can **only be modified by the system administrato**r. These configurations are protected against any unauthorized or corrupt requests, enhancing the system's resilience and security.

3.3 Implementation

The first step in restructuring the FRAFOL application was to decouple the static HTML views from the *Flask* framework. These views were integrated into a brand-new, isolated frontend application developed using *React.js*. This restructuring introduced an additional Docker image corresponding to the new frontend system (Figure 3.7).

The repository was reorganized to reflect this separation of concerns, with the `api` directory representing the backend (Flask) and the `client` directory representing the frontend (React.js). Figure 3.9 illustrates this updated architecture.

With the frontend now residing in the JavaScript ecosystem, the integration of the *Firebase* SDK was seamless. A new service class (Figure 3.10) was developed to serve as an interface between the frontend application and Firebase's cloud-based infrastructure. This API service handles user authentication and data storage, fulfilling the responsibilities discussed in the system's analysis.

[1] https://clerk.com/

Figure 3.9: FRAFOL's repository structure

Key methods implemented within this service class include:

- `createStudent()` – registers a new student in the Firebase system.
- `addSubmission()` – stores a student's code submission.
- `addClass()` – registers a new class under the teacher's profile.
- `getClassSubmissions()` – retrieves submissions for all students enrolled in a specific class.

This service architecture enhances modularity and maintainability, ensuring that cloud operations are abstracted and efficiently managed within the React-based environment.

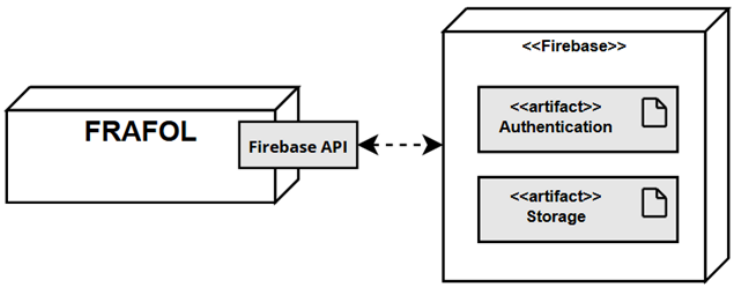

Figure 3.10: Firebase service class integration

3.4 User Interface

The user interface underwent a comprehensive styling remodel. FRAFOL now incorporates a cohesive color theme palette, which enhances the aesthetic appeal and visual consistency of the application. Figure 3.11 represents the new design which adopts a dashboard-like grid layout, promoting intuitive navigation across various tabs and sections of the system.

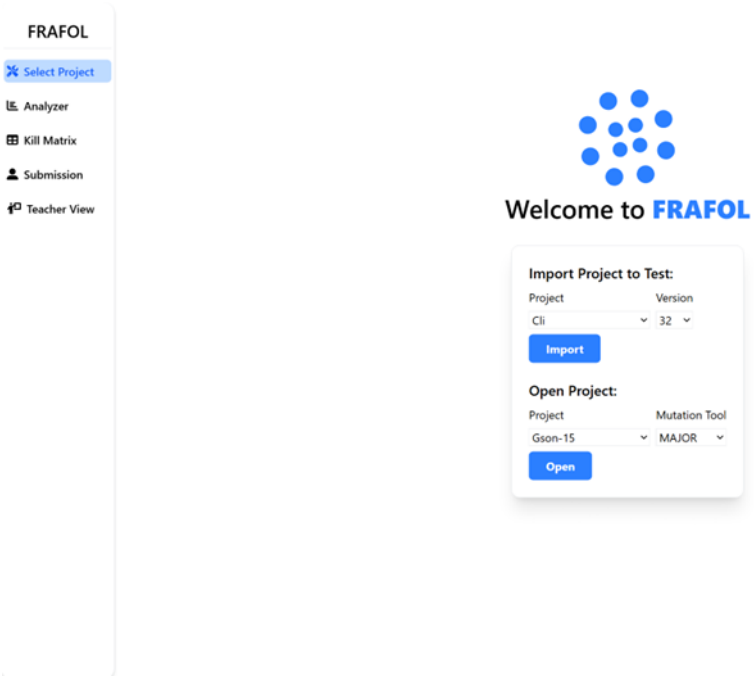

Figure 3.11: FRAFOL's new home screen.

Moreover, despite the low probability of users accessing the system via mobile devices, the interface was designed to be fully responsive. This ensures that all layout components remain accessible and properly displayed, regardless of screen size, including resolutions smaller than the standard desktop view of 1920×1080. (Figure B.2)

The redesigned interface also embraces a design-system-like methodology. Components are now modular and reusable, conforming to best practices in *React.js* componentization. This modularity simplifies the process of maintenance and future feature integration, while also ensuring a consistent user experience across different parts of the application.

FRAFOL's relevant new design views and tabs can be viewed in Appendix B.

3.5 Use Cases

This section is dedicated to detailing the implementation of the use cases discussed in the analysis phase of this chapter.

3.5.1 Code Editor

Mutation testing implicitly returns coverage metrics, which include valuable insights about the lines executed by a user's test cases. By introducing color highlighting for lines covered during test execution, the editor helps learners visually identify conditional boundaries (Figure 3.12) and guides them toward improving both mutation and coverage metrics.

Additionally, when the user clicks on a specific mutant from the Live Mutant Table, the corresponding line in the code editor is automatically highlighted and scrolled into view. This enhances usability by minimizing navigation efforts and focusing the user's attention on the mutated code.

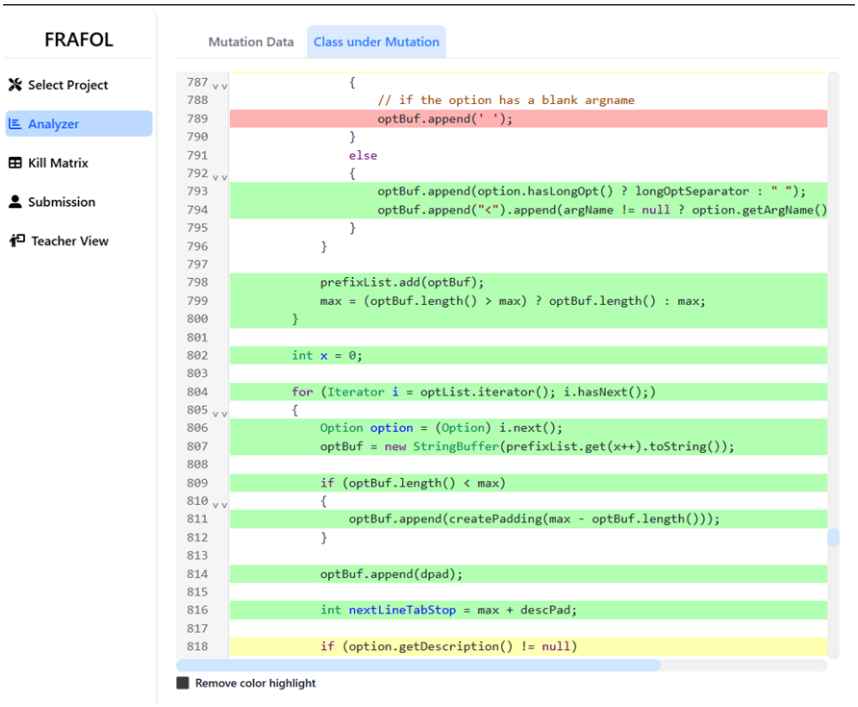

Figure 3.12: FRAFOL's code editor color highlighting.

3.5.2 Kill Matrix

One of the main contributions of this research work is the Kill Matrix feature. This matrix displays the relationship between each individual test case and the mutants it successfully kills. Unlike the Live Mutant Table, which only provides information about whether a mutant was killed, the Kill Matrix offers **granular visibility into the performance of each test method**. (Figure 3.13)

This process is computationally expensive, as it involves executing the mutation process for each test method independently. Future research could explore ways to optimize this operation.

3.5.3 Student View

Students are initially prompted with a login/signup screen (Figure 3.14). Account creation is handled through Firebase Authentication and requires the following information:

- Student number (unique user identifier)
- Email address
- Password
- Class name (pre-created by a teacher)

Figure 3.13: FRAFOL's kill matrix page

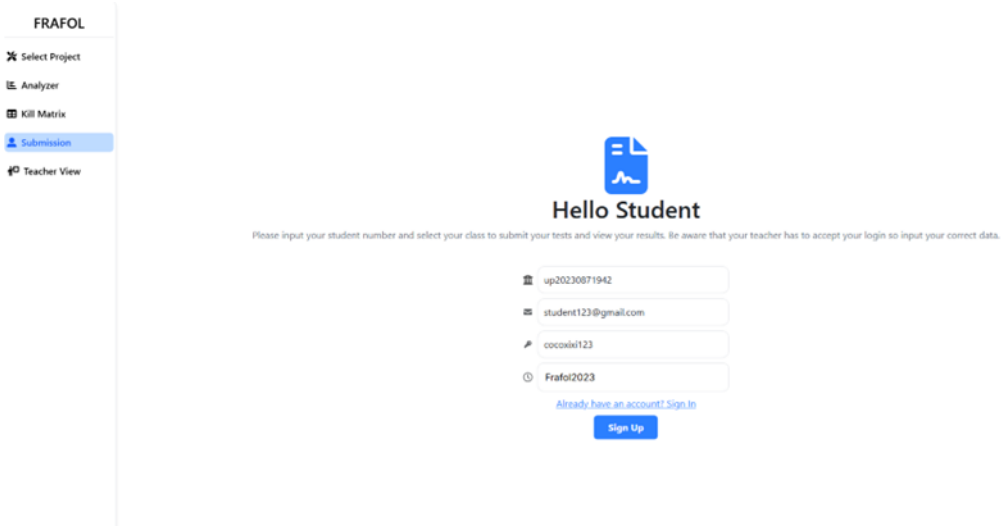

Figure 3.14: Student's login view.

After completing mutation testing on a selected project, students can make a submission (Figure B.3), which includes:

1. A list of mutant IDs killed by the student's test suite
2. The test suite code
3. Date of submission

These submissions are stored in Firebase Cloud Storage are made accessible to the teacher in the corresponding Teacher View dashboard.

3.5.4 Teacher View

The updated framework introduces a dedicated teacher view. Teachers can log in using credentials provided by the system administrator and proceed to create their own class. Upon doing so, the class becomes available for students during registration.

Teachers have full control over their classes via a dashboard interface (Figure 3.15), where they can:

- Admit new students
- Remove current students
- View statistics such as number of students, number of submissions, and pending admissions

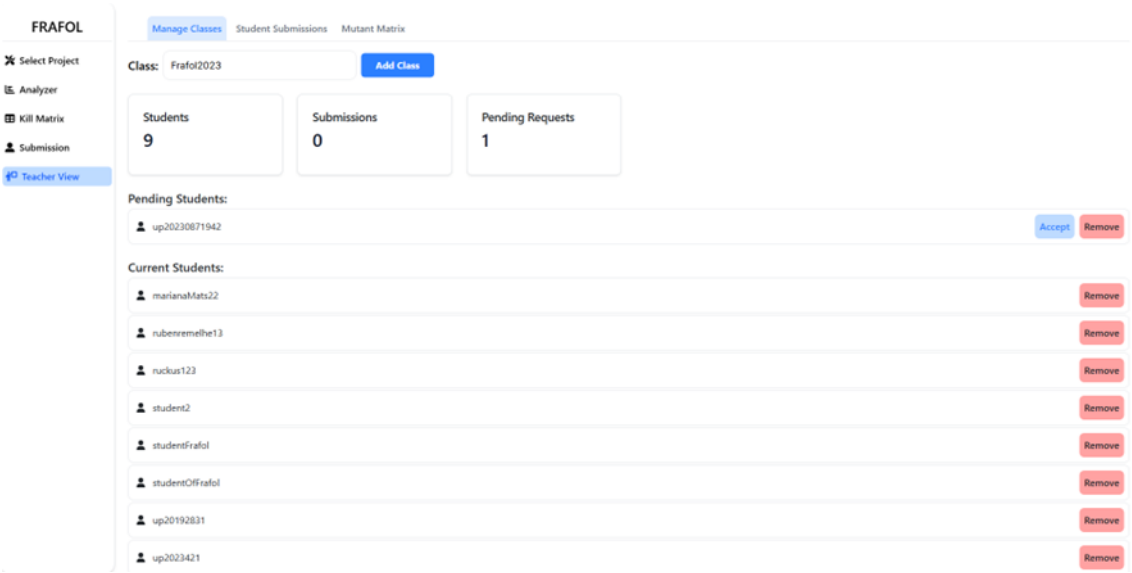

Figure 3.15: Teacher's class view

Additional tabs available to teachers include:

- **Students' Submissions**: View and manage all student code submissions.

- **Student Kill Matrix**: Displays a matrix that maps each student to the mutants they've killed per project. This enables a comprehensive evaluation of test quality and student performance.

Refer to Appendix B for figures illustrating these features.

3.5.5 Additional Contributions

The use case implementation was described in the previous section. However, this research applied additional improvements to the framework.

3.5.5.1 JUnit Support Upgrade

The previous system only supported JUnit 3 syntax for unit testing. The upgraded framework now supports JUnit 4, offering compatibility with **modern Java testing practices**.

3.5.5.2 Defects4J Update

Defects4J, responsible for providing buggy Java classes and executing mutation analysis, was updated to its latest version. This enhancement brings **improved performance**, access to a broader range of bugs, and more real-world examples.

3.5.5.3 Expansion of Available Projects

Originally, FRAFOL supported mutation testing for three projects: `Cli-32`, `Gson-15`, and `Lang-53`. This research **expanded the list by 266%**, providing users with a more diverse and realistic testing experience. The updated list includes the following.

- Cli-32
- Gson-8
- Gson-15
- Lang-12
- Jsoup-27
- Math-90
- Compress-40
- Compress-44

This expansion illustrates the framework's capacity for seamless scalability and its readiness for integration with additional real-world systems in future work.

3.5.5.4 Bundle *Minification*

FRAFOL's view now functions as a separate client of the API. As mentioned in Section 3.4, this *front-end* ecosystem is built using *React.js* and *JavaScript*, which rely on bundling tools to transform and package source code into a format **suitable for efficient delivery to web browsers**. In this project, the bundling process is handled by *Vite* [2], a modern build tool designed for speed and optimized development workflows.

Since the *front-end* view has been restructured as a Single Page Application (SPA), it is initially bundled into a package that contains the **entire application codebase** and is then built and accessed by *Docker*. To address this inefficiency, two optimization techniques were employed: *tree shaking*—automatically performed by *Vite* during the build process—and *lazy loading*. These approaches help **eliminate unused code** and **defer the loading** of certain modules, respectively. Additional theoretical background on bundle *minification* is provided in Appendix C.

Lazy loading enables specific components or modules to be loaded **only when they are needed**, rather than being included in the initial bundle. In the context of FRAFOL's SPA, dynamic `import()` statements were introduced at the routing level, allowing route-specific components to be loaded on demand.

For example, if a user only accesses "Page X", then only the necessary code for that page is imported and executed at runtime. The rest of the application's pages **are excluded** from the initial bundle, significantly reducing its size and improving load times. In contrast, without lazy loading, an user who opens FRAFOL and only interacts with the "Mutation Analysis" tab would still **incur the overhead of downloading** all other unused page components. This results in an inefficient use of resources and negatively impacts the perceived performance of the application.

This architectural change led to a significant reduction (almost 1 *megabyte* of code) in the size of the main bundle (Table 3.1), as **only essential code** for the application's initial render was included. Bundle *minification* not only improved FRAFOL's **startup performance** but also enhanced **perceived responsiveness**; these metrics were objectively validated in Chapter 4. The restructuring aligns with established best practices in modern front-end optimization, contributing to both **performance** and **maintainability** of FRAFOL.

Bundle Minification	**Before *Minification***	**After *Minification***
Main Bundle Size	1,677.65kB	799.45kB

Table 3.1: Main Bundle Size (in kilobytes) before and after bundle *minification*

This chapter presented the implementation constraints enhancements conducted by the research work. These improvements were scientifically based on the gaps found during the analysis of the state of the art of current software testing tools. The next chapter will validate these scientific contributions.

[2] https://vite.dev/

Chapter 4

Validation

In order to evaluate the results attained by this research, the validation stage was split into two parts:

1. **User Validation** - A user was given a concise **walkthrough of FRAFOL's features**, and asked to fill in a questionnaire regarding their user experience.
2. **Benchmark Validation** - This validation aims to **objectively quantify** the performance enhancements implemented throughout this research work.

Both validations will be compared with the previous version of the framework, thereby asserting the positive contribution of this document to research.

4.1 User Validation

Five Master's students were instructed to follow a set-up guideline where they would install FRAFOL and experiment with the framework. The procedure was structured as follows:

1. Install FRAFOL.
2. Import the `Cli-32` *Defects4J* project.
3. Open the project with the mutation tool PIT.
4. Try to kill additional mutants by writing new tests.
5. Check the mutation score. If the user was able to kill more mutants, the score should have increased.

These sequential actions allow for a brief overview of how the FRAFOL tool works.

Afterwards, the users were asked to fill in a questionnaire [1] that would evaluate their experience with FRAFOL. This form has multiple closed questions regarding different aspects of the

[1] https://forms.gle/GjFUjcfxB7ZfJR15A

framework that ought to be evaluated. The user is asked to give responses based on a level-5 *Likert* scale, ranging from the value 1 (strongly disagree) to 5 (strongly agree). Each of these closed questions could be encapsulated in one of the following aspects Tavares (2024):

- **Usability** - Is FRAFOL easy and concise in usage? This aspect aims to tackle the problem discussed in Chapter 2 of testing tools demanding ease of usage so that students do not get overwhelmed.
- **Satisfaction** - How does FRAFOL positively contribute to the user's learning experience? This aspect includes the interactive approaches (such as gamification) that were implemented in the tool in order to augment the user's motivation to learn.
- **Difficulties** - How does FRAFOL negatively affect the learning experience?
- **Effectiveness** - How is FRAFOL effective in improving the user's productivity in learning about mutation testing? This aspect analyzes whether the user found that the tool actually met their learning expectations and improved their knowledge about mutation testing.
- **Interface** - How pleasant, clear, and understandable is FRAFOL's graphical interface? This aspect evaluates the user's motivation and comfort with the interface's layout, and how that correlates with a better learning experience.

Table 4.1 illustrates some questions that each aspect encapsulates.

Aspect	**Questionnaire Closed Question Item**
Usability	FRAFOL is easy to use/follow.
Satisfaction	FRAFOL motivated me to learn.
Difficulties	I found FRAFOL unnecessarily complex.
Effectiveness	It is easier to learn with FRAFOL.
Interface	The interaction with FRAFOL is clear and understandable.

Table 4.1: Example of questionnaire items for each of the aspects

After the collection of the results, and in order to validate them in the same manner as previous research in FRAFOL did, for each of the aspects, the average question results were measured. Figure 4.1 illustrates the obtained values.

It is worth noting that previous research on FRAFOL conducted the same questionnaire and a similar validation group of 4 Master's students.

4.2 Discussion

The results obtained by the old FRAFOL tool in Figure 4.1 were already positive Tavares (2024). Nonetheless, the enhanced FRAFOL tool proposed by this work managed to achieve better results in all aspects.

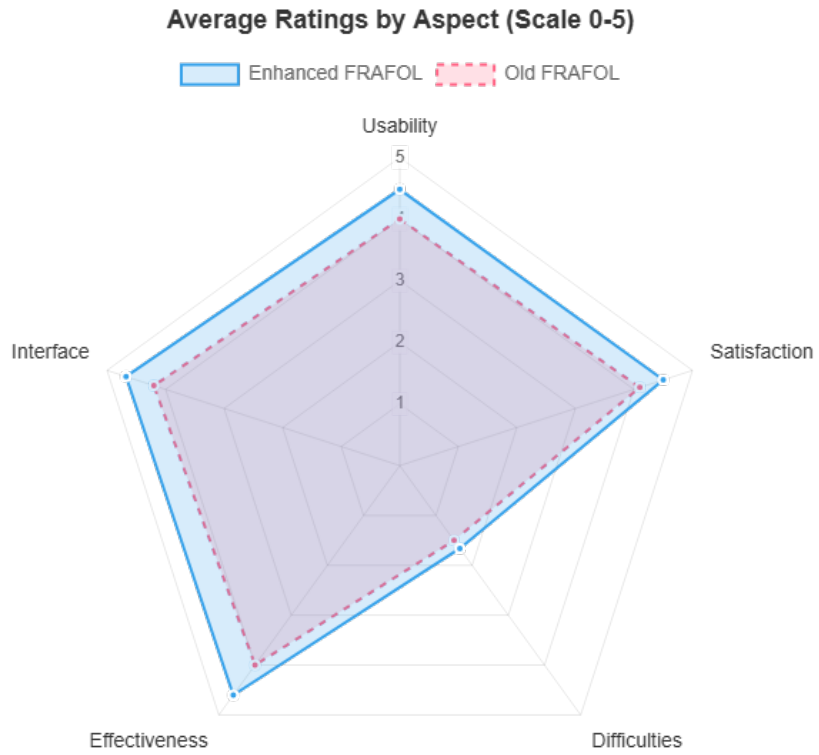

Figure 4.1: Average Ratings by aspect in a *Likert* (1-5) scale

Research noted that there was a minor increase in the "Difficulties" aspect, which could be explained by the presence of more user interface elements on screen, in comparison with the old FRAFOL's layout interface (Figure 2.6).

Moreover, the new tool successfully increased user engagement with the framework by:

1. Enhancing the interface with modern and concise components, contributing to **more user comfort** while using FRAFOL.
2. Providing more **visual feedback** on the user's actions, leading to a seamless workflow.
3. Promoting a **dashboard-like design** that enhances the **user's navigability** and **usability** of the tool.
4. Integrating more *Defects4J* collection projects, which in turn **gives the user more challenges** to further expand their mutation testing techniques.
5. Outlining **performance improvements** that provide the user with responsive, rapid feedback. (These improvements are further detailed and quantified in section 4.3.)

This validation stage proved that all of these latter affirmations contributed positively to an increase in **usability, satisfaction, and effectiveness** of the FRAFOL framework.

4.3 Benchmarks

This study presents significant performance enhancements to FRAFOL's processing capabilities.

To validate these improvements, benchmark experiments were conducted, comparing the performance of equivalent use cases in both the original and the optimized versions of FRAFOL. The experiments were executed under the following hardware and software configuration:

- **Machine** – MacBook Pro
- **Processor** – Apple M2 Pro
- **Memory** – 16 GB RAM
- **Java Version** – `OpenJDK 17.0.14`
- **Docker Engine Version** – `27.5.1`

4.3.1 Mutation Analysis Performance

Mutation analysis constitutes the core functionality of the FRAFOL tool. For benchmarking purposes, two buggy projects from the *Defects4J* dataset: `Cli-32` and `Gson-15` were selected. The experiment followed these steps:

1. Load the project.
2. Execute mutation analysis.
3. Record the execution time.

The timing was measured using the following pseudocode structure:

Listing 4.1: Mutation Analysis Timing Code

```
beginTime = getCurrentTime();

asynchronousMutationAnalysis();

endTime = getCurrentTime();

executionTime = endTime - beginTime;
```

For each of the projects, the analysis **was run 4 times** in both FRAFOL's versions. Table 4.3 presents the obtained results:

On average, the optimized version of FRAFOL demonstrated a performance improvement of approximately *51.8%* across the tested projects. Specifically, the execution time for mutation analysis in the `Cli-32` project decreased from 17.12 seconds to 9.95 seconds (a 41.9% reduction),

Tool Version	Cli-32 (s)	Gson-15 (s)
FRAFOL (original)	17.12	83.45
FRAFOL (after improvements)	9.95	31.51

Table 4.2: Average Execution Time (in seconds - from a total of 4 runs) for Mutation Analysis in Two Defects4J Projects

while the time for `Gson-15` dropped from 83.45 seconds to 31.51 seconds (a 62.2% reduction). These results highlight the substantial efficiency gains achieved through the enhancements introduced in the updated version of the tool.

4.3.2 Caching Mechanisms

Caching is one of the best software enhancements in order to achieve paramount application performance Zanini (2021). With the usage of the browser's *local storage* and *Docker's* file system, some caching features were explored in this research that sought to improve the overall performance of FRAFOL.

4.3.2.1 Project Importation

FRAFOL's current architecture works by importing a project from the *Defects4J* environment to the machine's Docker file system. With the project's code on the Docker machine, the tool can now collect the buggy classes and tests and run the mutation analysis.

For importing a project into the tool in the main screen menu (Figure 3.11), the user selects the desired project version and the system would:

1. Import the project to the Docker file system (if it does not exist yet).
2. Compile the project.
3. Execute the project's code coverage analysis
4. Project is opened

This study work identified that when a user attempted to revisit a project that **had already been imported** into the system, the import procedures were unnecessarily re-executed, resulting in redundant operations.

To improve this, once a project is imported, its **compilation and coverage results are cached**, thereby, in subsequent runs, if the user tries to import the project again, the system will recognize whether that information is already cached and in case it is, the project is opened immediately, skipping the redundant steps.

With this, the following benchmarks were taken for the `Cli-32` project:

The enhanced version of FRAFOL demonstrated a notable performance improvement in the import process for the `Cli-32` project. While the time required for the first import remained

Project Imports	**First import (s)**	**Subsequent imports (s)**
FRAFOL (original)	7.12	7.11
FRAFOL (after improvements)	7.09	0.86

Table 4.3: Average Import Time (in seconds - from a total of 4 runs) for the `Cli-32` project

nearly the same—7.12 seconds in the original version compared to 7.09 seconds after improvements—the time for subsequent imports was significantly reduced, dropping from 7.11 seconds to just 0.86 seconds. This represents an **87.9% reduction in subsequent import time**, highlighting the effectiveness of caching in optimizing repeated import operations.

4.3.2.2 Session Persistence

This research work improved the user's session persistence on the client side (browser). With the usage of the browser's *local storage*, the following features are cached:

1. The student's test code for each project.
2. The project's coverage and line highlighting metrics.
3. The project's killed and live mutants.
4. The user's login session.

These optimizations not only contribute positively to the overall performance of FRAFOL, but also greatly enhance the user's experience by allowing for a consistent and concise workflow, which persists between different usages of the tool.

4.4 Results and Conclusions

FRAFOL's validation process encompassed two complementary perspectives: a qualitative assessment centered on user experience, and a quantitative evaluation focused on system performance.

The first validation phase involved a group of users—five MSc students—interacting directly with the tool. Their feedback provided valuable insights into FRAFOL's user interface, overall usability, and its potential as a pedagogical aid. Participants highlighted the **clarity of the visual layout**, the intuitive navigation, and the tool's ability to foster a deeper understanding of mutation testing concepts. This feedback proved instrumental in improving the tool's assessment regarding its scientific value in the learning of mutation testing, in comparison with the old FRAFOL tool (Figure 4.1).

The second phase of validation adopted a benchmark-driven methodology to **objectively measure** the impact of the improvements introduced in this work. Comparative experiments between the original and the optimized versions of FRAFOL demonstrated significant performance gains in key operations such as mutation analysis and project importation. Caching mechanisms, session

persistence strategies, and bundle *minification* techniques proved particularly effective, contributing to reductions in redundant operations and enhancing overall efficiency.

In summary, the results of both validation approaches confirm that the enhancements introduced in this research not only **improved the system's technical performance** but also **strengthened its usability and value**. FRAFOL is now better positioned to serve as an efficient educational tool for teaching and practicing mutation testing.

Chapter 5

Conclusion and Future Work

Research allows for the ever-growing scientific knowledge in the field of software testing, which has been proven to be more and more critical, as the **world turns to technology as a scaffold** in a number of processes. Equipping IT professionals with testing concepts is essential, as no refined software product is released **without rigorous testing** to prevent real-time failures. Mutation testing proved to be one of the most important concepts when it comes to simulating these failures in software programs. With a bridge between **effective teaching and learning of mutation testing**, students are provided with an indispensable skill set that broadens their understanding of software testing.

5.1 Objectives Satisfaction

This study introduces FRAFOL, a tool designed to bridge the gap in effective mutation testing education. FRAFOL offers several key features:

1. **An innovative, modern user interface** that ensures a smooth and intuitive learning experience, allowing seamless navigation between the framework's features.

2. A broad ecosystem **containing multiple mutation testing tools**, which encapsulates all setup and installation complexities into one simple framework functionality. Its support for multiple mutation testing tools is a **significant distinguishing feature**.

3. **A diverse collection of real-world projects** suitable for mutation testing. Utilizing actual shipped source code motivates users and hones their cognitive testing capabilities to align with industry standards essential for future professional careers.

4. **A class-based learning platform** where students and teachers can collaborate to make better use of FRAFOL. By introducing project code submissions, leaderboards, and feedback mechanisms, position FRAFOL as an effective framework for fostering mutation testing education in universities and educational institutions.

Previous research on FRAFOL identified several ambiguities, indicating **substantial room for enhancement**. This work explored and implemented new functionalities, including code color highlighting, mutant kill matrix generation, and student and teacher learning systems. Furthermore, the architectural restructuring of the tool **yielded notable performance improvements**, both in user experience and quantifiably in processing speed.

The validation phase of this research confirmed the positive impact of FRAFOL's enhancements. It demonstrated an improvement in user experience compared to the previous version as well as objective performance gains of approximately ***51.8%*** in the duration of mutation analysis, which is FRAFOL's primary use case.

The tool is free, open-source, and available for use at `https://github.com/apaivafeup/projFRAFOL/tree/FRAFOL2.0`.

In summary, this research successfully **achieved all proposed objectives** and expanded into additional areas, ensuring maximum growth potential and **paving the way for future substantial improvements**.

5.2 Future Work

The improved version of FRAFOL proved to be a successful experiment, enhancing the state of research on software testing teaching and learning. The author of this work believes this research scaffold can be further expanded with future contributions.

5.2.1 *JUnit* Upgrade

As of 2025, *JUnit*'s framework is ever-growing with constant new version releases. In order to keep FRAFOL up to date with industry standards—which has already been proven to be a significant advantage for learners— future research ought to explore the integration of *JUnit* 5+ into the system. This will not only significatively improve FRAFOL's ecosystem but also ensure that users are **applying the latest Java unit test syntax**. It is important to still maintain **retro compatibility** with older versions of *JUnit*. This can be achieved, for instance, using various engines such as *JUnit* Vintage.

5.2.2 Kill Matrix Processing Speed

Chapter 3 mentioned that the process of generating the kill matrix is long, since it runs one unit test at a time against the entire buggy class. This feature has room for optimization in the algorithm that processes this use case. Multiple improvements such as **parallel test analysis** or a more **lightweight approach** to the data that encapsulates the creation of the kill matrix can be investigated. Furthermore, keeping the framework up to date with the latest *Defects4J* release can always bring speed optimizations, as it was proved in Table X.

5.2.3 Equivalent Mutant Suppression

Equivalent mutants, as outlined in Chapter 1, pose a significant challenge to the validity of mutation analysis results. Their presence can lead to misleading mutation scores, thereby compromising the accuracy of the overall evaluation. To mitigate this issue, future research on software testing should focus on developing more effective techniques for detecting equivalent mutants before the calculation of mutation metrics. One promising direction is the integration of machine learning models trained on datasets of previously identified equivalent mutants. Such an approach could **enable the automated classification of future mutants with a higher degree of precision**, reducing the need for extensive manual inspection and enhancing the reliability of mutation testing outcomes.

5.2.4 Continuous Integration - Automated Testing

Reinforcing software testing importance, the addition of **automated unit and integration tests** would prove to be of great value in ensuring FRAFOL's maintainability and scalability for eventual future implementations. This pipeline would serve as a shield to ensure all of FRAFOL's different functionalities are thoroughly secure and ready for use.

FRAFOL's room for growth is evident with the validation results. Future contributions will foster the tool's potential for broadening the educational virtues that it provides to software testing learning and teaching as a whole, once more enduring and providing insightful scientific findings towards an emphasis on better and enhanced research in the world of software testing.

References

Domenico Amalfitano, Ana Paiva, Alexis Inquel, Luís Pinto, Anna Fasolino, and Rene Just. How do java mutation tools differ? *Communications of the ACM*, 65:74–89, 11 2022. doi: 10.1145/3526099.

Amazon. What is a unit test? AWS, 2025. https://aws.amazon.com/what-is/unit-testing/.

Gina R. Bai, Justin Smith, and Kathryn T. Stolee. How students unit test: Perceptions, practices, and pitfalls. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, ITiCSE '21, page 248–254, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450382144. doi: 10.1145/3430665.3456368. URL https://doi.org/10.1145/3430665.3456368.

Martin Balfroid, Pierre Luycx, Benoît Vanderose, and Xavier Devroey. An empirical evaluation of regular and extreme mutation testing for teaching software testing. In *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 405–412, 2023. doi: 10.1109/ICSTW58534.2023.00074.

Felix Cammaerts, Monique Snoeck, and Ana C. R. Paiva. Collecting cognitive strategies applied by students during test case design. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, EASE '23, page 455–459, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400700446. doi: 10.1145/3593434.3593954. URL https://doi.org/10.1145/3593434.3593954.

Felix Cammaerts, Porfirio Tramontana, Ana C. R. Paiva, Nuno Flores, Fernando Pastor Ricós, and Monique Snoeck. Exploring students' opinion on software testing courses. *ACM International Conference Proceeding Series*, page 570 – 579, 2024. doi: 10.1145/3661167.3661276. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-85197454050&doi=10.1145%2f3661167.3661276&partnerID=40&md5=abc03c4515cbe144ffa0e9a027ea7449. Cited by: 0.

Eric Compton and Dan Romanoff. Future remains bright for software firms. Morningstar, 2025. https://www.morningstar.com/stocks/future-remains-bright-software-firms.

Lin Deng, Josh Dehlinger, and Suranjan Chakraborty. Teaching software testing with free and open source software. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 412–418, 2020. doi: 10.1109/ICSTW50294.2020.00074.

Gordon Fraser, Alessio Gambi, and José Miguel Rojas. A preliminary report on gamifying a software testing course with the code defenders testing game. ACM Digital Library, 2018. https://dl.acm.org/doi/10.1145/3209087.3209103.

Vahid Garousi, Austen Rainer, Per Lauvås Jr., and Andrea Arcuri. Software-testing education: A systematic literature mapping. *Journal of Systems and Software: Volume 165, July 2020, 110570*, 2020. URL https://www.sciencedirect.com/science/article/pii/S0164121220300510#fig0008.

Nien-Lin Hsueh, Zeng Hung Xuan, and Bilegjargal Daramsenge. Design and implementation of gamified learning system for mutation testing. *International Journal of Information and Education Technology*, 13(7):1150 – 1155, 2023. doi: 10.18178/ijiet.2023.13.7.1916. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-85164394001&doi=10.18178%2fijiet.2023.13.7.1916&partnerID=40&md5=7c0823e81b384600f4162f879b4a61d0. Cited by: 2; All Open Access, Gold Open Access.

IEEE Computer Society. The importance of software testing. Computer.org, 2024. https://www.computer.org/resources/importance-of-software-testing/.

Daniel E. Krutz and Michael Lutz. Bug of the day: Reinforcing the importance of testing. In *2013 IEEE Frontiers in Education Conference (FIE)*, pages 1795–1799, 2013. doi: 10.1109/FIE.2013.6685147.

Bob Kurtz, Paul Ammann, Jeff Offutt, and Mariet Kurtz. Are we there yet? how redundant and equivalent mutants affect determination of test completeness. pages 142–151, 04 2016. doi: 10.1109/ICSTW.2016.41.

Beatriz Marin, Vos Tanja, Ana Paiva, Anna Fasolina, and Monique Snoeck. Enactest - european innovation alliance for testing education. *CEUR - RCIS Workshops 2022*, 2022.

Pedro Reales Mateo and Macario Polo Usaola. Bacterio: Java mutation testing tool: A framework to evaluate quality of tests cases. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 646–649, 2012. doi: 10.1109/ICSM.2012.6405344.

Omar Ochoa and Salamah Salamah. An approach to enhance students' competency in software verification techniques. pages 1–9, 10 2015. doi: 10.1109/FIE.2015.7344050.

Mike Papadakis, Marcio Delamaro, and Yves Le Traon. Mitigating the effects of equivalent mutants with mutant classification strategies. *Sci. Comput. Program.*, 95(P3):298–319, December 2014. ISSN 0167-6423. doi: 10.1016/j.scico.2014.05.012. URL https://doi.org/10.1016/j.scico.2014.05.012.

D. Passig and L. Nadler. Structural and conceptual user interfaces and their impact on learning. *duc Inf Technol 15, 51–66*, 2010.

José Miguel Rojas and Gordon Fraser. Code defenders: A mutation testing game. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 162–167, 2016. doi: 10.1109/ICSTW.2016.43.

Lilian Passos Scatalon, Rogério Eduardo Garcia, and Ellen Francine Barbosa. Teaching practices of software testing in programming education. In *2020 IEEE Frontiers in Education Conference (FIE)*, pages 1–9, 2020a. doi: 10.1109/FIE44824.2020.9274256.

Lilian Passos Scatalon, Rogério Eduardo Garcia, and Ellen Francine Barbosa. Teaching practices of software testing in programming education. In *2020 IEEE Frontiers in Education Conference (FIE)*, pages 1–9, 2020b. doi: 10.1109/FIE44824.2020.9274256.

Elsje Scott, Alexander Zadirov, Sean Feinberg, and Ruwanga Jayakody. The alignment of software testing skills of is students with industry practices - a south african perspective. 01 2003. doi: 10.28945/2681. https://jite.org/documents/Vol3/v3p161-172-040.pdf.

P. Tavares, A. Paiva, D. Amalfitano, and R. Just. FRAFOL: Framework for learning mutation testing. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 1846–1850, 2020.

Pedro Diogo Veludo Tavares. Framework for learning mutation testing. Master's thesis, Universidade do Porto - Faculdade de Engenharia da Universidade do Porto, 2024.

Porfirio Tramontana, Beatriz Marín, Ana Paiva, Alexandra Mendes, Tanja E. J. Vos, Domenico Amalfitano, Felix Cammaerts, Monique Snoeck, and Anna Rita Fasolino. State of the practice in software testing teaching in four european countries. Google Scholar, 2024. https://scholar.google.pt/citations?view_op=view_citation&hl=pt-PT&user=gkpFe5sAAAAJ&sortby=pubdate&citation_for_view=gkpFe5sAAAAJ:Z5m8FVwuT1cC.

van Merriënboer, J.J.G., Sweller, and J. Cognitive. Cognitive load theory and complex learning: Recent developments and future directions. *Educational Psychology Review: Volume 17, pages 147–177, (2005)*, 2005. URL https://link.springer.com/article/10.1007/s10648-005-3951-0#author-information.

Auri M.R. Vincenzi, Pedro H. Kuroishi, João Bispo, Ana R.C. da Veiga, David R.C. da Mata, Francisco B. Azevedo, and Ana C.R. Paiva. Metford – mutation testing framework for android. *Journal of Systems and Software*, 222:112332, 2025. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2024.112332. URL https://www.sciencedirect.com/science/article/pii/S0164121224003765.

Preeti Wadhwani and Aishvarya Ambekar. Software testing market size. Global Market Insights, 2024. https://www.gminsights.com/industry-analysis/software-testing-market.

Antonello Zanini. What is caching and how it works. Auth0, 2021. https://auth0.com/blog/what-is-caching-and-how-it-works/.

Appendix A

Defects4J Framework

A.1 Purpose

Defects4J is a collection of real-world, reproducible software bugs accompanied by a standardized infrastructure designed to support and advance empirical research in software engineering.[1] By providing access to well-documented bugs from widely used open-source Java projects, Defects4J enables researchers and educators to systematically evaluate fault detection techniques, mutation analysis, automated program repair, and regression testing strategies.

The framework's core aim is to facilitate reproducibility and comparability in software testing experiments by offering controlled environments for bug reproduction, test execution, and coverage assessment. Defects4J's widespread adoption and consistent versioning make it an essential benchmark suite in modern software testing research.

A.2 Project Composition

As of the latest version, Defects4J includes 854 active bugs (plus 10 deprecated ones) drawn from 17 open-source Java projects, such as *Jsoup*, *JacksonDatabind*, *Lang*, and *Math*. Each bug follows a strict set of criteria to ensure empirical consistency: it must originate from an issue tracker, be fixed in a single commit, and possess a triggering test case that fails before the fix and passes after.

Each bug is uniquely identified and associated with two revisions: a buggy version (`<id>b`) and its corresponding fixed version (`<id>f`). The framework avoids renumbering bug identifiers to preserve backward compatibility with existing research.

A.3 Tooling Interface

Defects4J emphasizes a command-line interface (CLI)-centric workflow. The CLI provides structured access to key metadata, such as source and test directories, classpath configurations, and

[1] https://github.com/rjust/defects4j

valid test sets. For users operating outside the native Defects4J environment, the `defects4j export` command is essential for obtaining these resources.

By adhering to this interface, users can ensure the reproducibility of experimental setups and integrate Defects4J seamlessly into broader mutation testing or automated debugging pipelines, such as those implemented in frameworks like FRAFOL.

Appendix B

FRAFOL's web views

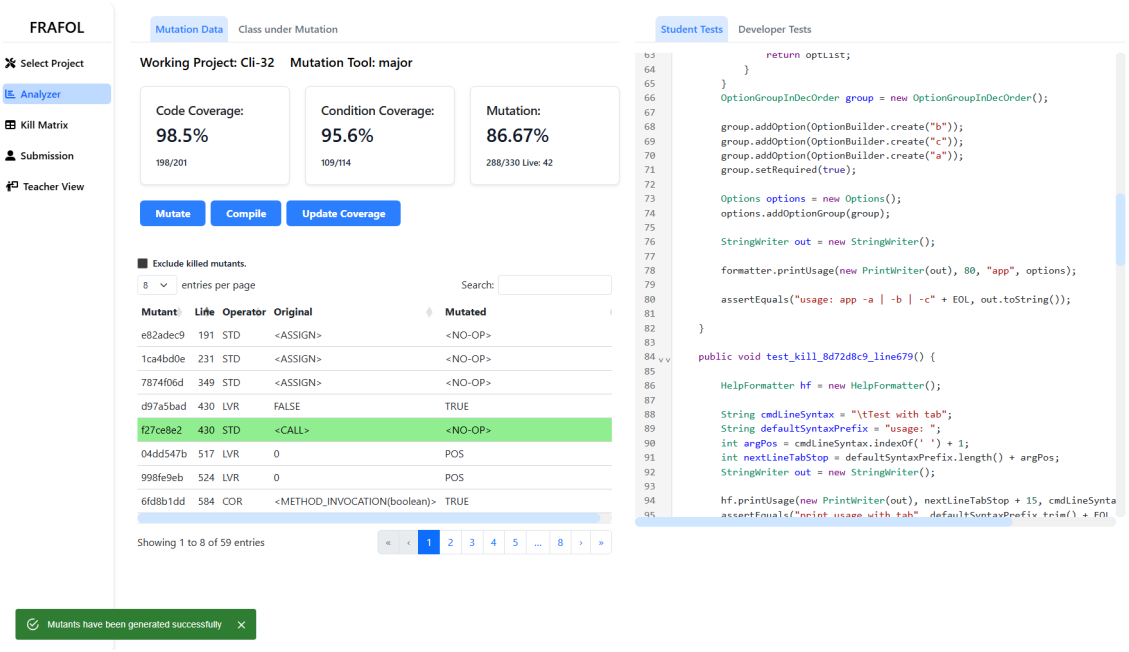

Figure B.1: Mutant Analyzer Tab

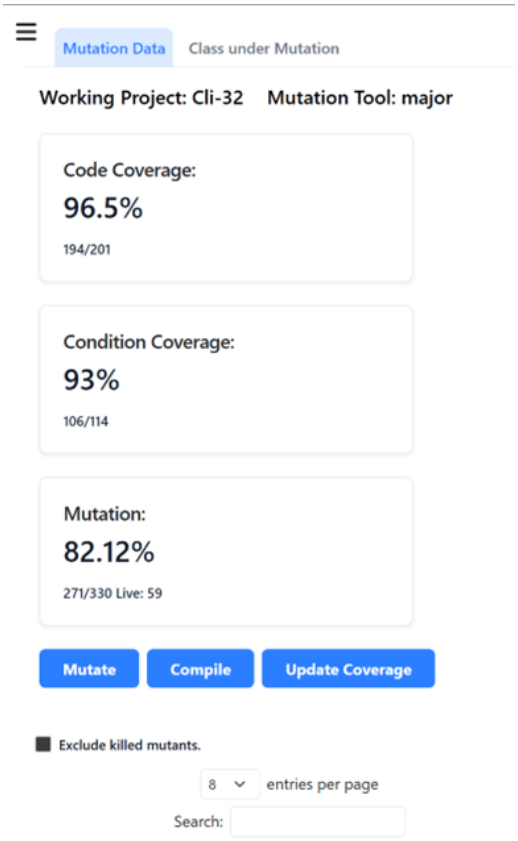

Figure B.2: FRAFOL's mobile responsive view.

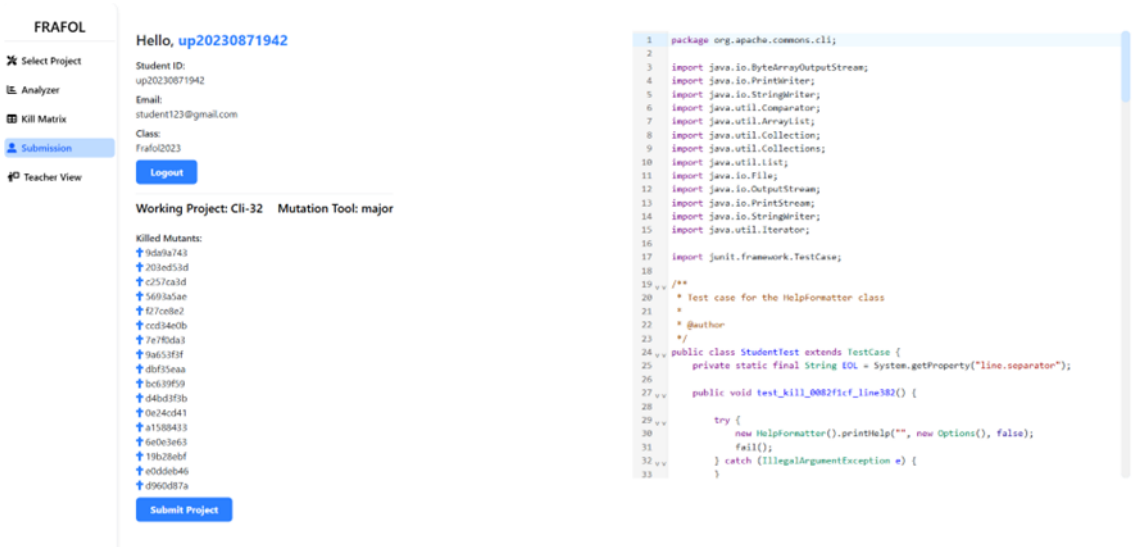

Figure B.3: Student view - Submission tab

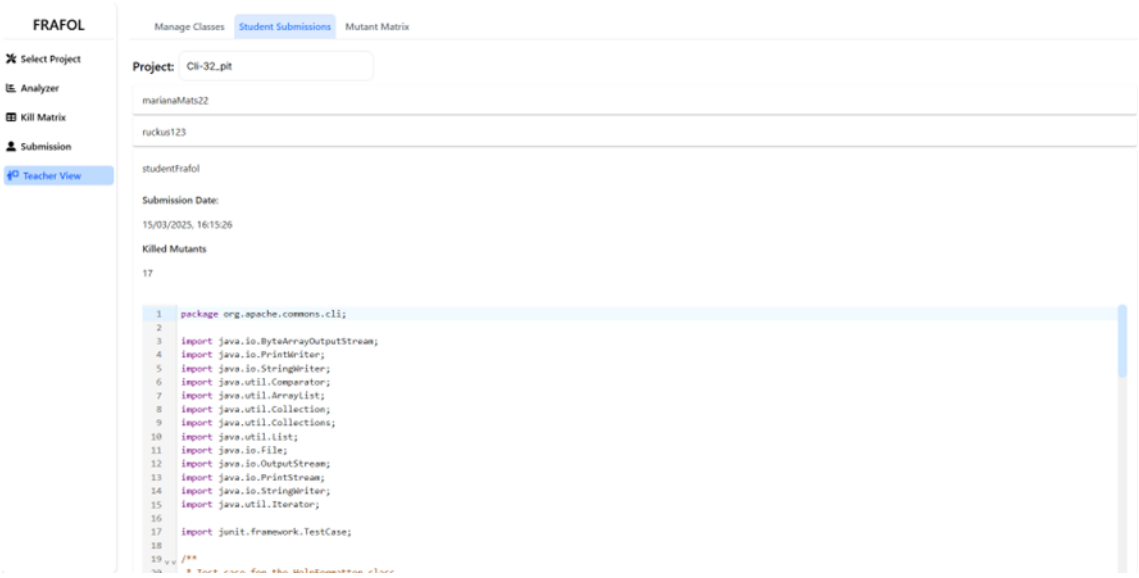

Figure B.4: Teacher view - Student submissions tab

Figure B.5: Teacher view - Mutant matrix tab

Appendix C

Bundle *Minification*

C.1 Purpose

The primary purpose of bundling is to consolidate and optimize a web application's JavaScript, CSS, and static assets into one or more output files, thereby reducing the number of HTTP requests required to the server at runtime. Additionally, bundlers often apply various forms of *minification* and *tree shaking*—removing unused code and compressing file sizes—to enhance loading performance and reduce bandwidth usage.

C.2 Problem

In Single Page Applications (SPAs), the packaging of multiple assets into an output file can result in the generation of a large initial bundle that contains the entirety of the application code, even if only a subset is needed at startup.

C.3 Solution

To address the inefficiencies of large initial bundles, there are multiple techniques that can be employed:

- **Lazy Loading** - deferring the importation of unused files
- **Tree shaking** - removal of dead and unused code from the bundle
- **Dependency Optimization** - each dependency and library that the project uses will be included in the final bundle. Therefore it is vital to optimize the usage of these third parties.
- **Media Optimization** - usage of remote assets —- for example, with origin in *CDNs* or *servers*— and utilization of optimized SVG icons can drastically improve bundle size.