FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Framework for Learning Mutation Testing

Pedro Diogo Veludo Tavares

Master in Informatics and Computing Engineering

Supervisor: Ana Cristina Ramada Paiva

September 18, 2024

Framework for Learning Mutation Testing

Pedro Diogo Veludo Tavares

Master in Informatics and Computing Engineering

September 18, 2024

Resumo

Software testing é um dos processos necessários durante a produção de software que permite validar se uma implementação foi bem concebida, verificando se desempenha as funções pretendidas assim como a deteção e prevenção da existência de erros. A falta ou má realização deste tipo de processo pode levar a consequências severas que resultem em perdas financeiras, falta de credibilidade no produto final, ou até perigo para a vida humana. Será, então, de interesse de todas as entidades que a devida implementação e execução de práticas de *software testing* seja levada em alta consideração. Porém, a prática de *software testing* é uma tarefa difícil, com estudos a revelar que muitos praticantes a consideram extremamente desafiante Garousi et al. (2020a). Além disso, estudos revelam que os alunos demonstram dificuldade em se envolver com a disciplina, reportando como sendo uma tarefa aborrecida e destrutiva, frequentemente tendo preferência em processos de planeamento e implementação Blanco et al. (2023).

Mutation testing é uma das técnicas de *software testing* que trascendeu para além da investigação académica e se encontra mais presente em contextos industriais e *open-source*, assim como matéria de currículo universitário em engenharia de software. Apesar de já existir um número considerável de ferramentas de *mutation testing* disponiveis, com as suas diversas vantagens e desvantagens, a sua intergração em atividades educativas permanece um problema devido à sua complexidade e necessidade de integração em processos de desenvolvimento. Adicionalmente, é desejável que os alunos tenham oportunidade de experimentar diversas ferramentas, para que, como exemplo, consigam observar diferenças na quantidade e tipo de mutantes produzidos. Exigir a instalação e familiarização de múltiplas ferramentas enfatiza a complexidade técnica da aprendizagem, resultando em inconsistências no aproveitamento dos alunos.

Este trabalho apresenta o projeto FRAFOL, uma *framework* para aprendizagem de testes de mutação, que visa criar um ambiente comum para o uso de diversas ferramentas de *mutation testing* para fins educativos, contendo várias características de qualidade de vida com o objetivo reduzir as frustrações com o *setup* e aumentar a qualidade e o foco na aprendizagem da matéria.

Keywords: Software Testing, Software Industry, Education

ACM Classification: Software and its engineering → Software creation and management → Software verification and validation

Abstract

Software testing is a critical process in the software development lifecycle, aimed at verifying and validating that a software implementation functions as intended while detecting and preventing errors. Inadequate or poorly executed software testing can lead to severe consequences, such as financial losses, diminished credibility of the final product, or even threats to human safety. Thus, ensuring the proper implementation and execution of software testing practices is of paramount importance. However, testing software remains a challenging task, with studies indicating that many practitioners still find it difficult Garousi et al. (2020a). Furthermore, research has shown that students often struggle to engage with software testing, viewing it as a tedious and destructive activity and typically favoring design and implementation tasks Blanco et al. (2023).

Mutation testing is a software testing technique that has evolved beyond academic research. It is now present in industrial and open-source settings and is increasingly a part of software engineering universities' curricula. While many mutation testing tools exist, each with different strengths and weaknesses, integrating such tools remains challenging due to the complexity of use and the need to integrate them into a development environment. Additionally, students should have the opportunity to interact with different tools to explore the differences, for example, in the types and quantities of mutants generated. However, asking students to install and familiarize themselves with multiple tools only compounds the technical complexity and likely results in unwanted differences in how and what students learn.

This paper introduces FRAFOL, a framework designed to facilitate learning mutation testing. FRAFOL offers a unified environment for using different mutation testing tools in educational settings, incorporating several quality-of-life features that alleviate setup frustrations and enhance the learning experience by allowing students to focus on the subject matter.

Keywords: Software Testing, Software Industry, Education

ACM Classification: Software and its engineering → Software creation and management → Software verification and validation

Acknowledgements

To Professor Ana Paiva and Professor Domenico Amalfitano, my heartfelt appreciation for their incredible support and dedication in assisting me in this project. Their patience and availability throughout this process made me more confident and motivated to deliver the final vision for this work.

To Professor René Just for his feedback and availability to help better understand the Defects4j framework.

To the students of "Projeto Integrador" João Lima, Pedro Estela, Rafael Aldeias, and Joaquim Cunha, for picking and choosing to contribute to this project.

To the students of TVVS and Professor José Campos for their participation and feedback with testing and experimentation, your contribution was indispensable to the success of FRAFOL.

Last but not least, to my parents for supporting me all my life and giving me all that was necessary for me to accomplish my goals.

Pedro Tavares

Contents

List of Figures

List of Tables

Abbreviations

GBL	Game-based Learning
IT	Information Technologies
TDD	Test Driven Development
TDL	Test Driven Learning
TVVS	Testes, Verificação e Validação de Software
UI	User Interface
ISSTA	International Symposium on Software Testing and Analysis

Chapter 1

Introduction

Software testing, in essence, is the continuous application of processes that verify if a software product or application performs in a desired manner. Its presence is felt throughout the entire development cycle, so different techniques and skills are demanded to exercise it properly.

The applicability can range from tests that perform checks on algorithmic outputs to tests that verify if a system satisfies certain business requirements. With the ever-increasing complexity of software systems, the demand for a qualified workforce is growing higher, and a shortage of IT professionals is still present today Phillips (2022). As such, the teaching of software testing is faced with the difficult task of introducing substantially varied testing concepts while also giving students enough practical context that prepares them to tackle the industry needs. Simultaneously, most students find testing to be a boring subject Fraser et al. (2020), which adds an additional challenge of keeping them engaged. Research, such as Gomes and Lelli (2022); Fraser et al. (2020); Clarke et al. (2010); Allowatt and Edwards (2005), has been done to explore new techniques and methods of assisting teaching through the use of courseware and gamification.

To better understand how we can improve the learning experience of software testing, we must first familiarize ourselves with this subject's current standard education methods. This chapter will explore how software testing is currently taught, how students perceive the subject, and the alignment of education with industry needs. Additionally, it will briefly highlight the principal concepts of mutation testing.

1.1 Context

From an industry standpoint, there's a great demand for software testers Phillips (2022). One problem is there are still areas in which graduates present knowledge deficiencies Scatalon et al. (2019). According to Cem Kaner Kaner (2003), the fundamental challenges in software testing are as follows: "(1) Complete testing is impossible; (2) Testers misallocate resources because they fall for the company's process myths; (3) Test groups operate under multiple missions, often conflicting, rarely articulated; and (4) Test groups often lack [enough] skilled programmers, and a vision of appropriate projects that would keep programming testers challenged", as found in

Garousi et al. (2020a). This correlates with the necessity for students to allocate multiple cognitive resources, such as the ability to make critical choices on what is the most important code to test. ENACTEST is a co-founded initiative project by the Erasmus+ Programme that aims to assist students and enterprises in this endeavor by designing new cognitive modules for testing, which help in the conception of test suites and meet the criteria necessary for dealing with challenges found in the industry ENACTEST.

From a survey performed by Scatalon et al. (2019), practitioners reported that sound knowledge of testing fundamentals helps in becoming a good tester. However, they complained that the education focus is too high on theoretical aspects and lacks more practical scenarios. Only technically strong teams in the industry seem to encourage a good testing culture; otherwise, they are often devalued. A common complaint is that courses can address testing subject topics well but don't indicate how to develop a tester mindset.

1.1.1 Mutation Testing

According to Ojdanic, "Mutation is a test adequacy criterion in which test requirements are characterized by mean of mutants obtained by performing slight syntactic modifications to the original program (for instance, the relational expression a>b can be mutated into a<b)" Ojdanic et al. (2023). The purpose is, through the artificial insertion of code modifications, it's possible to assess the effectiveness of test suites in detecting these mutations. If a mutant *survives* the fault detection, the tester is tasked to analyze the mutant and implement further test cases to *kill* it. Mutation testing falls under the category of white box testing, meaning that testers are granted complete knowledge of the application, including source code and design documentation. Many mutation testing tools support multiple programming languages and perform different operator mutations. Some may require access to source code or not, depending on what level of code mutation it supports.

The following list highlights some of the prominent mutation testing tools:

- **PITEST**: Java based mutation tool that performs byte-code level mutations. Only mutates code covered by the test suite. Produces HTML format reports. Contains 11 default mutant operators.
- **StrykerMutator**: Contains multilingual support. Performs source-code level mutations. Contains 30 mutant operators. Live report system.
- **Major**: Java based mutation tool. Performs both byte and source-code level mutations. Contains nine mutant operators. Generates log report file.

When performing mutation testing, there are certain statistics that one must be familiar with:

- **Code Coverage**: It's the measure of source code executed when a particular set of tests is run. It helps identify untested parts of a codebase that the developer test suite has not

reached. Higher code coverage indicates that more lines of code have been tested, which typically leads to fewer undetected bugs, thereby improving software quality and reliability.

- **Condition Coverage**: It evaluates whether each boolean sub-expression (condition) in the code has been tested for both true and false outcomes. This ensures that all logical paths are evaluated, helping to identify errors in the decision-making logic process of the code.
- **Mutation Score**: Software testing metric that measures the effectiveness of a test suite against the mutations created by the mutation testing tool. It's measured by the ratio of killed mutations against mutants that remain alive.

These metrics are essential because they provide insights into the effectiveness and thoroughness of a test suite. While the goal is to eliminate the highest number of mutants possible, it's also essential for testers to identify the quality of the mutant and how productive it is for the test suite. Specific mutants, despite altering the original code, don't provide an effective change in the software behavior. As such, it is impossible to write a test suite that detects such mutants and is classified as equivalent.

1.2 Motivation

Despite the ever-increasing demand and importance of software testing, it is still largely neglected as a subject in computer science education Jesus et al. (2020). In a survey by Scatalon et al. (2019), practitioners reported that only 20% had learned about software testing concepts during introductory programming courses, 50% were taught testing during a singular course, and only 17% had taken dedicated courses in the subject. The two present forms of education mentioned in most studies are dedicated lectures and regular programming courses, where only a third of software testing is done through dedicated courses Garousi et al. (2020b). Both methods present different advantages and challenges. Where dedicated courses allow for a more in-depth understanding of complex matters, they can struggle to keep students motivated as they find the topic tedious. Regular programming courses offer the opportunity to introduce topics earlier on and allow for the process of testing to maintain a presence throughout the curriculum. However, this is not always practical, as time constraints can often conflict with the opportunity for students to explore the steps necessary to guarantee that proper testing practices are being applied. As such, additional research is required to create course material that promotes the idea of engaging students with the subject of software testing that strikes the right balance between usability and realism and enable students to learn software testing techniques effectively.

1.3 Problem Statement

Many software testing tools are readily available, with active development communities and frequent use during research and investigation. Still, often, they lack comprehensive features and

consistency in their integration into curricula, requiring multiple setup procedures and extensive prior knowledge. In the context of mutation testing, multiple tools have been proposed in the literature, each offering different capabilities and approaches to generating code mutants. However, their applicability has been chiefly on research and practice Amalfitano et al. (2022). There is a lack of educational modules that allow students to interact with various mutation testing tools quickly while offering realistic contextual applications and the ability to have hands-on experience with writing test cases.

With an ever-increasing demand for software testing developers, Phillips (2022), significant resources must be applied to enrich the learning experience of these topics with a scope on assisting educators in integrating the use of testing tools during assignments while also tackling the common frustrations reported by students.

A common thread between multiple studies is students' lack of engagement with the subject of software testing. Fraser et al. Fraser et al. (2020) tell us that most students find the activity boring and destructive to the creative process of designing software applications. Part of this feeling stems from the lack of perceived value in spending resources to determine if code operates in a desired manner, often tied to the limited scope of coding projects Garousi et al. (2020b). Coincidentally, there's a higher motivation to discover bugs in software written by others, as exposing bugs can relate to failure in developing a proper program Garousi et al. (2020b); Ochoa and Salamah (2015).

1.4 Goals

The problem statement gives rise to the following objectives that this research aims to achieve. The goal is to develop a framework for education on software testing. More specifically, the method of Mutation Testing was chosen as it is an excellent subject for teaching students about the importance of software testing because it challenges them to think critically about the effectiveness of their test cases. By deliberately introducing small faults, or mutations, into a program, students can see firsthand how robust their tests are in catching errors. This process underscores the significance of thorough testing in software development and sharpens students' analytical skills as they learn to identify weaknesses in their code and improve their test suites accordingly.

Examining recent practices in software testing education, an analysis of both successful and unsuccessful cases will help identify the most effective methods to apply in a new module. In summary, this work aims to forward the research on educational tools as a means to facilitate the demanding process of learning complex software testing course material while also simplifying the setup and configuration requirements by offering a uniform environment for learning and experimenting.

Students are expected to respond positively to the newly developed module, as it plans to incorporate the most effective methods identified by analyzing recent software testing educational tools. The final goal is to perform a validation experiment that qualitatively measures the tool's performance in a learning environment and identifies the areas where it effectively enhances student

comprehension and engagement. Additionally, the experiment aims to uncover any weaknesses or challenges that may need further refinement, ensuring that the tool meets educational objectives.

1.5 Document Structure

The remaining document will present the following structure: Chapter 2 examines the current software testing education methods and strategies, providing an overview of the landscape in academic and professional training contexts. It reviews various software testing educational tools, including gamified approaches that enhance engagement and learning outcomes. The chapter delves into best practices, focusing on the pedagogical applications of mutation testing. It critically analyzes how these approaches are implemented and their effectiveness in fostering learning, setting the foundation for understanding FRAFOL's educational value.

Chapter 3 details the design and development cycle of the FRAFOL tool. This chapter explores the conceptualization, planning, and iterative development stages, outlining key decisions made during the tool's creation. It covers the technical architecture, user interface design, and the integration of pedagogical principles, explaining how FRAFOL addresses the educational needs identified in the literature review.

Chapter 4 presents the validation process for the FRAFOL tool, documenting two key experiments conducted to evaluate its effectiveness. It describes the experimental setup, the metrics used for evaluation, and the results obtained. The chapter includes a comprehensive analysis of the findings, discussing how FRAFOL performed in real-world educational contexts and identifying areas of success and opportunities for improvement.

Chapter 5 summarizes the dissertation's findings, offering conclusive thoughts on the success and impact of the FRAFOL tool. It goes over potential future development directions for FRAFOL, proposing enhancements and additional features that could further improve its educational value.

Chapter 2

State of the Art

Software testing requires a strong understanding of testing methodologies, attention to detail, critical thinking, communication skills, and effective resource management Kaner (2003). It's essential to understand what methodologies are studied and experimented with in academia because course programs shape the foundational knowledge of future testers. They provide structured learning on both theoretical concepts and practical applications, ensuring that graduates are well-prepared to meet industry standards. Therefore, the first section will present research methods developed to assist in learning and teaching software testing.

The next section explores various educational tools specifically designed to aid in learning software testing. These tools are evaluated for their effectiveness in providing practical, hands-on experience, which is critical for mastering testing concepts. The review includes a discussion on automatic feedback tools, interactive platforms that facilitate experiential learning, and the use of game mechanics as a source for engaging students with education.

2.1 Education Methods

Teaching software testing is crucial to preparing future engineers to use effective coding techniques and guaranteeing software quality. To this effect, the scientific community has invested interest in developing tools and teaching methods that enable students to learn testing while addressing several challenges Garousi et al. (2020a). As stated by Garousi et al. (2020b), most research is experience-based, while only a short amount had a more theoretical approach. Ben-Ari (2001) offers constructivist learning theory, which presents the idea that students must be actively involved in learning and knowledge construction. In the context of testing, this notion is very present, as the learning process often consists of being self-critical of their work or collaboratively engaging with written code by other colleagues. Furthermore, there's a problem with the traditional one-way transmission of knowledge and students' lack of in-depth thinking, as it weakens the ability to learn independently Xie (2018). A blended approach can prevent repetitive knowledge-focused teaching that can lead to student burnout, combining teacher-led instruction, student practice, and topic discussions. The software testing curriculum can be structured by separating the teaching

content into segments: the teacher handles the more challenging theoretical aspects while students engage in practical exercises and discussions in class. By adopting a "flipped classroom" approach Li Hua and Ning (2018), where students take on a central role, their active participation increases. The teacher then wraps up the session by summarizing key points, posing questions, providing additional insights, and facilitating a group discussion. This teaching model has multiple benefits since students engage in extensive self-study, including researching materials, summarizing content, and creating presentations before class. It enhances students' ability to learn independently, deepens their understanding of the course material, and improves their skills in organizing documents and delivering oral presentations. Overall, the teaching method proves to be effective Chen et al. (2020).

A study by Lambers (2020) details a *Sandwich* approach to the traditional software testing perspective, introducing static analysis as a preliminary task over testing. Static analysis techniques support the idea of students getting more familiarized with software projects, learning about their requirements and codebase, allowing them to have more thorough testing of potential complex modules, code refactoring from potential code smell detection to preserve code behavior and identifying bug alerts generated by automatic bug detection tools. Static analysis of software projects is essential for software testing because it enables developers and testers to identify potential issues in code without executing it. This preemptive approach helps pinpoint issues early in the development process, reducing the chances of defects reaching production. One of the most significant benefits of static analysis is its contribution to time efficiency. In large and complex projects, exhaustively testing every line of code is often impractical. Static analysis tools automatically highlight areas of the codebase that are more likely to contain errors or pose risks, allowing testers to prioritize their efforts effectively.

Delgado-Pérez et al. (2021) and Oliveira et al. (2015) introduce the practice of peer testing together with mutation testing, primarily focusing on analyzing the impact on learning programming techniques and allowing for collaborative learning experiences. The former study presents findings on how students perceive the use of advanced testing techniques over manual, unguided testing methods. The results show that students felt a greater appreciation of mutation testing as a testing method when faced with comparing subjective self and peer assessment on their manually written test methods against the impartial measurement of a mutation score. The peer testing approach gives students further insight into assessing their relative detection capabilities and identifying potential gaps in their test completeness. These collaborative learning experiences and peer testing are crucial in software testing education because they mirror real-world scenarios where teamwork is essential. In collaborative settings, students learn to communicate, share knowledge, and solve problems, enhancing their understanding of complex testing concepts.

Other initiatives present the opportunity for learners to familiarize themselves with more accurate industry testing practices. An observation by Eric Wong et al. (2018) shows that software testing classes are commonly lecture-oriented and lack the appropriate practical aspect of how

testing tools can be applied in real-world software. To address this, a series of industry-sponsored International Software Testing Contests (ISTC) were organized to place students in an environment where they may experience the challenges that testing practitioners face. The contests were designed to bring real-world subjects with moderate complexity and allow students to compete by attempting to achieve common industry good quality testing criteria, such as code coverage and mutation score. Such events enable students to experiment with the latest industry tool support and help raise awareness for the lack of testing practitioners.

Research by Deng et al. (2020) has also explored Free Open Source Software (FOSS) as a significant instructional medium of teaching and learning activities in software testing courses. The study offers a structure of learning activities that provides students with real-world industry-level software testing experiences. This structure follows a design with the goal of initial familiarization with the project developer test suite and then incrementally tasking students to perform contributions in the form of additional test cases, writing bug reports, practicing regression testing with continuous integration services, and finally practicing TDD. These kinds of brownfield projects often have well-documented, publicly accessible information on their source code and even active developer communities that can help students gain a better understanding of the work. Students are able to feel the impact of their accomplishments in real-world software, develop teamwork and communication skills with other senior developers, and advance their professional network.

2.2 Learning Tools

Educational tools are crucial in software testing education, bridging the gap between theoretical knowledge and practical application. Software testing, by nature, is a highly technical and hands-on discipline that requires an understanding of concepts and extensive practice to master Garousi et al. (2020a). Educational tools provide students with interactive, real-world environments where they can apply testing techniques, experiment with different scenarios, and receive immediate feedback on their actions. These tools help to solidify theoretical concepts by allowing students to engage with live code, simulate testing processes, and visualize the impact of their tests. Moreover, they cater to diverse learning styles, offering a range of functionalities such as automated feedback, cooperative learning, and even gamified experiences, which can increase student engagement and motivation.

2.2.1 Web-CAT

According to Garousi et al. (2020b), educators may use auxiliary tools for support in the learning environment. As such, many studies have been written detailing the success of using courseware. The following one has been mentioned in eight papers between the years 2003 to 2014.

Web-CAT is an automated grading system plug-in developed by Anthony Allowatt and Stephen Edwards from Virginia Polytechnic Institute and State University with the purpose of assessing student assignments Allowatt and Edwards (2005). The primary concept began with giving students early exposure to the subject of test-driven development (TDD), which forces the thought

process of explaining and understanding the expected behavior of code before actually writing. Utilizing an automated plug-in system allows for this approach to be extended to all assignments beyond testing education. The grading process provides ratings based on code validity and completeness of test cases. Validity rates how well tests conform to the task requirements, while completeness rates the coverage and gives feedback on which test suites are incorrect. Additionally, the plug-in has wide support in various submission forms, such as e-mail, FTP, HTTP, and HTTPS. Setting up requirements must also be minimal to encourage students and facilitate integration into software practical classes. To this effect, the design team extended their plugin support to Eclipse (Figure 2.1), a widely popular IDE used in academia. Eclipse provides support for both JUnit and C++ unit testing, which is integral to their testing approach.

During tool experiment assignments, students were encouraged to submit work often and respond to feedback. However, two patterns emerged: students hesitated to submit code that didn't work, preferring to fix issues on their own. This delayed feedback on errors, particularly those related to code structure and style. Additionally, students often submitted poorly formatted or undocumented code, leading to overwhelming feedback. To address these issues, students were provided with the Checkclipse and PMD plug-ins, which offered immediate feedback on style and formatting during each compilation, helping them to correct problems incrementally.

Despite the conceptual design to minimize setup requirements for students, there are still considerably observed dependencies that contradict the statement goal.

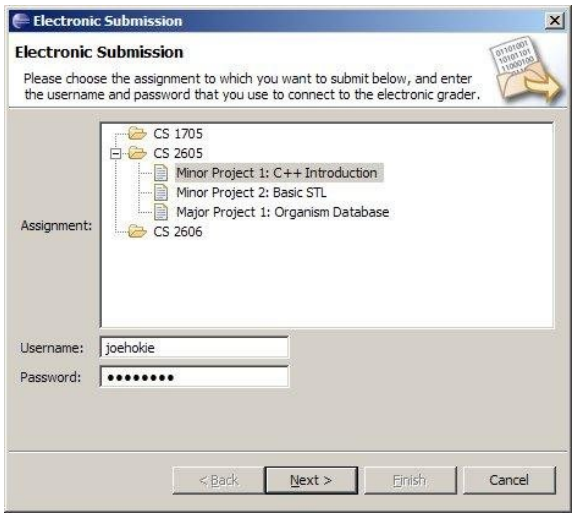


Figure 2.1: *Web-CAT* submission form for Eclipse (https://sourceforge.net/projects/web-cat/)

2.2.2 WReSTT-Cyle/SEP-CyLE

As the size and complexity of software systems continue to grow, the need for developing strategies and tools for pedagogy use increases Clarke et al. (2010). *WReSTT-Cyle* is essentially a repository of learning resources that aim to assist students and instructors in gaining the necessary knowledge on testing tools and integration. Emphasis was set on reducing the difficulty of

set-up and integration of Test-Driven Learning (TDL) into other curricula without the necessity of re-writing course material. For this approach, an online portal was developed consisting of vetted learning material, from tutorials on the use of testing tools to workshops on test integration into programming and SE courses. Later, the project saw its evolution into a collaborative social platform that can reinforce student engagement through the use of a social points system Ramasamy et al. (2018).

2.2.3 Reneri

Reneri, developed by Vera-Pérez et al. (2019), introduces the method of extreme mutation testing, an expanded approach that pushes the boundaries of traditional mutation testing by introducing more radical and aggressive mutations into the code. While standard mutation testing typically makes small, targeted changes, extreme mutation testing involves more disruptive alterations, like removing the entire body of methods and replacing them with trivial return statements. The goal of extreme mutation testing is to challenge the robustness of the test suite even further, exposing more profound flaws in the testing process. Simulating more extreme scenarios helps identify weak spots in the test suite that might go unnoticed with standard mutation techniques. The learning challenges of mutation testing can often be steep, as mutation tools can often behave unintuitively. Students may lack the appropriate guidance to understand the nature of mutants and struggle with what test to create. Most mutation testing tools offer informative feedback by presenting metrics like the mutation score and details on undetected mutants. In contrast, Reneri goes beyond simply providing raw data. It delivers suggestive feedback, offering practical insights that help users actively improve their test suite. Additionally, the drastic code changes make the effects of mutations more apparent and more straightforward to trace, as opposed to subtle, complex modifications. This simplicity helps students quickly grasp how the test suite reacts to significant code shifts, making it easier to write tests. However, an empirical evaluation by Balfroid et al. (2023) compared the use of suggestive reporting by Reneri with informative reports generated by regular mutant analysis in an education setting and concluded that both methods proved equally compelling, perhaps discrediting the notion that learners struggle with the latter form of feedback.

2.2.4 Gamification

In recent years, there's been an upward trend in the use of gamification for education in an effort to make software testing more interesting and less tedious Lauvås Jr (2018). Conceptually, gamification can increase student motivation using elements such as reward points, badges, and competitive aspects, such as leaderboards Garousi et al. (2020b). An empirical study by Blanco, R. Blanco et al. (2023) revealed that students saw a contribution in the engagement and performance of their assignments compared to those who had not experienced the effects of gamification, which fell in line with results that prior experiments had previously reported. Additionally, Gomes and Lelli (2022) details the difference between gamification and game-based learning (GBL), where the first has a passive approach to enhancing the educational experience, and the second directly

uses game mechanics to assign challenges to teach testing material. EU-funded projects such as IMPRESS have pushed for gamification to improve students' motivation at a university level Vos et al. (2020) and have given origin to multiple tools that will be discussed in this section.

2.2.4.1 Bug Hunt

Perhaps one of the earliest implementations of game mechanics in software testing education began with *Bug Hunt*, a web-based tutorial application developed in 2007 by the Computer Science and Engineering Department at the University of Nebraska-Lincoln Elbaum et al. (2007). Its design team aimed to reinforce the necessity of introducing testing concepts, like JUnit and black-/white box testing, in early programming curricula while acknowledging integration challenges, such as the diversity of student skills and goals and the already intensive presence of course material. *Bug Hunts'* notorious feature is its quality of feedback mechanisms, with text logs reporting test validity, automatic widget updates that highlight code coverage as new tests are introduced, performance indicators that report when a test case successfully exposed a fault in the source code, and a summary report after each chapter that describes all the learning objectives for that lesson. The performance indicator is complemented by a "bug jar" feature (Figure 2.2) that benchmarks the student's performance, with the number of bugs detected versus the class average. The tool successfully delivered a basic introduction to testing material; however, the limited lessons do not allow for the expansion of more advanced concepts, and the learning outcomes don't appear to contribute to the student's understanding of the realistic application of testing methods.

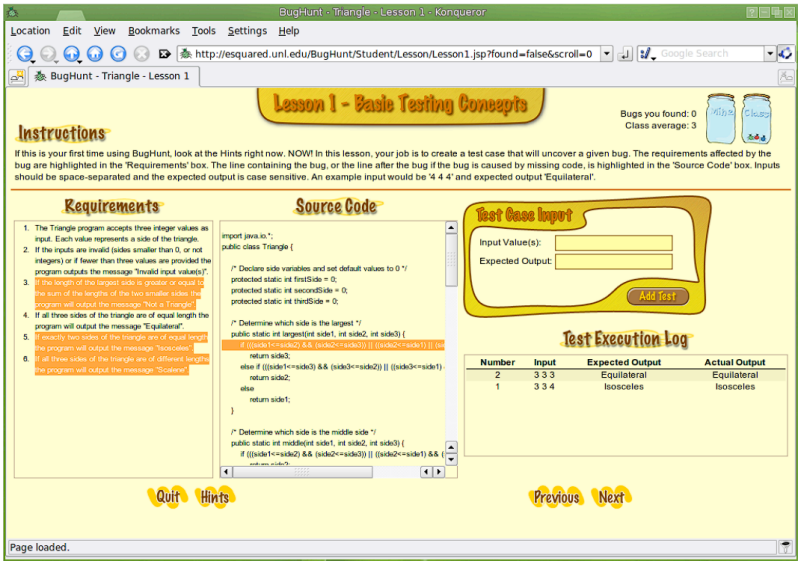


Figure 2.2: *Bug Hunt* user interface (Elbaum et al. (2007)

2.2.4.2 iLearnTEST

Some initiatives explore the concept of serious games designed without the primary focus of entertainment. Instead, a strong emphasis on storytelling elements is applied to teach players theoretical concepts and test their knowledge.

iLearnTEST, according to Ribeiro and Paiva (2015), is a framework designed to build games more easily, not intending to replace traditional education but to provide additional options in the educational process. It contains a set of frameworks that facilitates game construction by separating game content from game implementation. It also incorporates challenges to retain student engagement and feedback on correct and wrong answers provided by players. A performance comparison between two groups of students taking a software testing exam showed that the group using *iLearnTEST* as a learning tool outperformed the group relying on the internet and traditional syllabus despite having a lower average curricular grade.

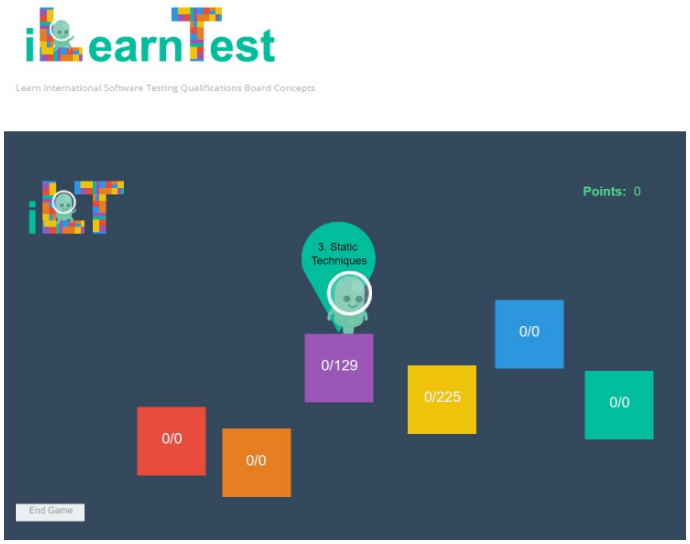


Figure 2.3: *iLearnTest* game interface (https://web.fe.up.pt/ apaiva/iLearnTest/)

2.2.4.3 Code Defenders

Arguably, the most prominent example of the application of GBL in the learning space is the game *Code Defenders*. Developed and maintained by the Chair of Software Engineering II at the Universities of Passau and Leicester Fraser et al. (2020), *Code Defenders* was created as a competitive experiment that pits two groups of students against each other, where one side plays the role of *Attackers*, which aim to mimic the behavior of a mutation testing tool by creating mutants that evade the test suites created by the *Defenders* team, that in retaliation try to create additional effective tests. The application was implemented as a web-based game. Figure 2.1 represents the user interface from the *Defenders* perspective.

Code Defenders is a prime example of a course integration aid tool that enhances active student participation in the learning process. It became an effective tool for enhancing student learning and promoting active involvement by integrating it as a graded, mandatory component of a software testing course. Fraser et al. (2020) highlights how *Code Defenders* gradually became part of software testing course curricula, with regular practice through weekly sessions, alternating roles as attacker and defender, allowing students to progressively develop their testing skills in a structured environment. Each student participated in 12 sessions, which involved creating and testing mutants, ensuring active and consistent participation. Evaluating the quality of mutants and tests throughout the semester provided students with continuous feedback, allowing them to see their improvement over time. The analysis showed a clear improvement in students' testing skills, with significant correlations between the number of sessions and their performance in terms of branch coverage and mutation scores. This demonstrates that as students played more games, their understanding and application of testing concepts improved. The moderate correlation between game performance and exam results further confirms that students who were more engaged and active in the game tended to perform better academically. Surveys indicated that students overwhelmingly liked the integration of *Code Defenders* into the course. This positive reception suggests that the gamified approach not only made learning more enjoyable but also likely increased students' intrinsic motivation to engage with the material.

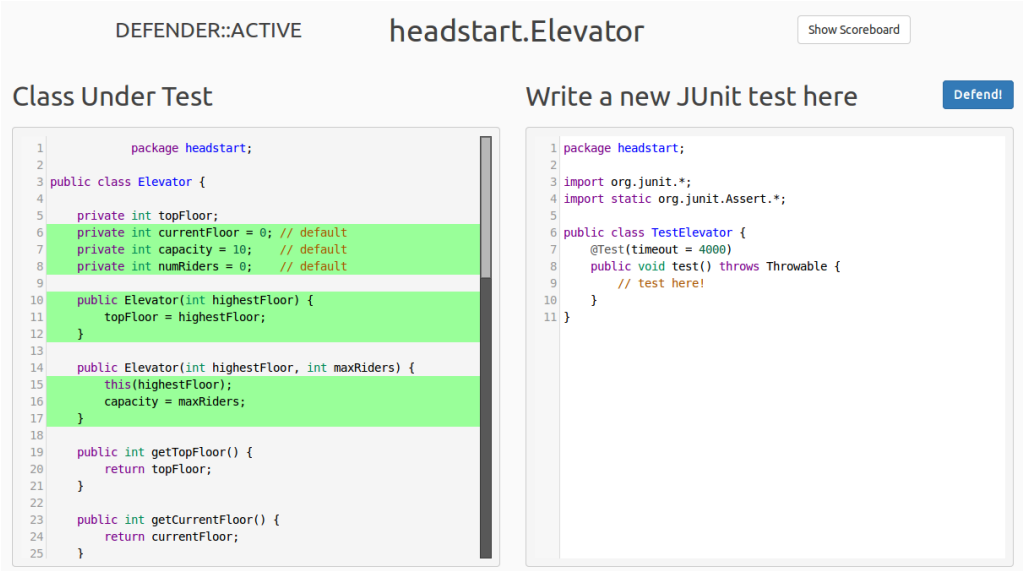


Figure 2.4: Defenders user interface (https://code-defenders.org/help)

2.2.5 Code Critters

Another game that centers around mutation testing and GBL is *Code Critters* Straubinger et al. (2023), a tower defense game that blends engaging gameplay mechanics with foundational testing

concepts. In the game, players must defend a castle from waves of critters (mutants) by strategically placing magical mines (test cases) along a path to eliminate them. Unlike *Code Defenders*, *Code Critters* targets less experienced coders by incorporating block-based programming (Figure 2.5), both as a gameplay feature and as an accessible introduction to coding. This approach helps lower the barrier to learning software testing. The game integrates narrative with educational elements, keeping students engaged while emphasizing the importance of introducing testing early in programming education. Although *Code Critters* is still a proof-of-concept, no data has yet been collected to assess its effectiveness as a learning tool.

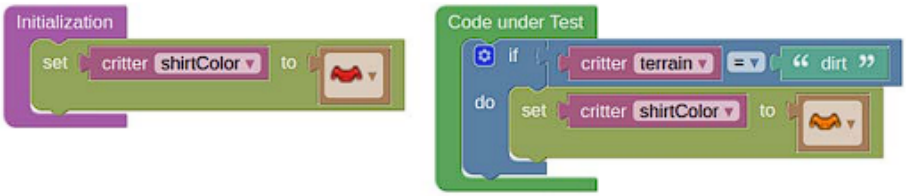


Figure 2.5: *Code Critters* test case block programming (https://code-critters.org/

2.2.5.1 CoverBot

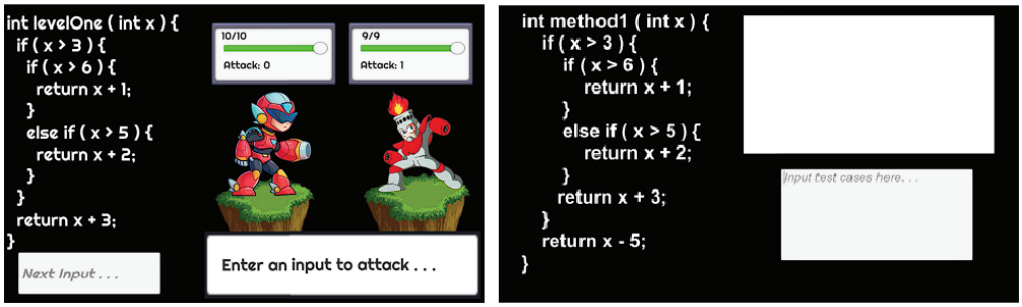


Figure 2.6: *CoverBot* standard and controlled versions (Sherif et al. (2020)

A learning tool that follows the same concepts as previous examples is *CoverBot*, which introduces code statement coverage while engaging players with a turn-based combat game. The core gameplay loop challenges players to input values that test various statements within a block of code displayed on the interface (Figure 2.6). For each successful line of code executed, the player deals health damage to the opposing enemy. An input value that does not cover any new statements will result in the player's health loss. The study by Sherif et al. (2020) states that an experiment was performed where two groups of students engaged in a learning activity where one group used a controlled version of the tool with no present gameplay features, while the second group used the standard version of *CoverBot*. After a 5-point Likert scale questionnaire, participants reported significantly higher engagement and enjoyment when using the gamified version

and a 9% increased success rate in delivering correct inputs over the other group. However, this tool does not serve as a sample for GBL because none of its gameplay elements are linked with teaching the subject matter.

2.2.5.2 GoRace

In the study by Blanco, R. Blanco et al. (2023), there's an example of a tangential application of gamification. This research aimed to determine how game elements can effectively improve students' performance in software testing and compare them to non-gamified environments. For this reason, a game needed to be conceptualized, and the research team chose to use the *GoRace* platform to implement a track race where students took part in a "legendary Olympic rage for immortality" and would gain points as they completed practical tasks. This exemplifies using game elements that engage students through a reward system while not directly participating in teaching material. The design experiment implemented widely common game elements, such as points and scoreboards, and additional game dynamic elements, such as a sense of progression, interaction, narrative, and socialization. The experience was made of four exercise sessions carried out in a sequence of 14 weeks, with one controlled group of participants that did not partake in the game element and an experimental group for whom each exercise corresponded to 1000 distance units for the race track and a potential maximum score of 90 points based on participants' performance on test design, implementation, effectiveness, and execution. The results claim that in the first exercises, the level of engagement in the experimental group was higher than in the control group. However, the experiment's findings suggest that gamification's impact on student engagement and performance decreased over time. Initially, students were more engaged and motivated, particularly in the second exercise, but this engagement gradually declined in the subsequent exercises. The performance metrics followed a similar trend, with effectiveness and effectiveness increase diminishing in the latter stages. The results indicate that while gamification initially drove extrinsic motivation, it was insufficient to maintain intrinsic motivation throughout the entire seminar. As students neared the finish line and perceived diminishing rewards, their participation and effort waned, leading to increased dropout rates and a decline in performance. This suggests that the long-term effectiveness of gamification may be limited if not supplemented by other motivational strategies.

2.3 Discussion

In conclusion, this chapter underscores the importance of innovative tools and methodologies in software testing education. The review suggests that combining traditional methods with modern, interactive tools can significantly enhance the educational experience, better preparing students for the challenges of software testing in professional environments. Nonetheless, significant effort is still required to find the right combination of best methods for integrating software testing in curricula and the means to better understand current testing course practices Tramontana et al. (2024).

Traditional learning methods often fall short when fully engaging students in software testing education. These conventional approaches may lack interactive or practical components that help students understand and apply software testing principles in real-world scenarios. As a result, students might struggle to see the relevance or importance of the material, leading to disengagement. Student motivation plays a crucial role in their ability to achieve learning goals, particularly in software testing that demands theoretical knowledge and hands-on practice. Keeping students engaged requires deliberate effort and adopting methods that stimulate their interest and foster active participation in learning activities. Without this, the learning process may become a passive experience, reducing the chances of success. With modern infrastructures providing unprecedented ease of access to information, students today have more resources at their disposal than ever before. The rapid evolution of new software development and testing technologies means that staying current requires constant learning. As new tools, frameworks, and methodologies emerge, students must actively share and discuss new knowledge with their peers to remain up-to-date and proficient in their field. Collaborative learning is a vital practice in software testing education. It reinforces essential professional skills, such as peer review and teamwork, by encouraging students to work together and critique each other's work. Through collaboration, students can develop insights into their testing strategies, assess their effectiveness in detecting issues, and identify areas where their test coverage may fall short. This practice enhances their technical abilities and prepares them for real-world team-based software development environments.

There's a strong indication that using real-world projects in software testing education is crucial because it gives students practical experience and exposure to the complexities of actual software development environments. Real-world projects offer diverse challenges, including dealing with legacy code, understanding incomplete requirements, and testing under real constraints, which are often missing in theoretical or small-scale examples. This hands-on experience enhances critical thinking, problem-solving skills, and adaptability, better-preparing students for professional roles in software testing. Additionally, it opens the opportunity for learners to get involved with working projects and develop connections with other senior practitioners.

Gamification and game-based learning have shown significant potential in improving student engagement in software testing education. Incorporating game mechanics such as points, levels, and challenges creates a more interactive and enjoyable learning environment, motivating students to participate more actively. When applied effectively, game mechanics can simplify complex software testing methods, offering practical examples and scenarios that help students grasp complex concepts in a hands-on, experiential manner. However, while initial results are promising, further research is needed to fully understand how gamification influences learning outcomes and whether it consistently helps students achieve better results in mastering software testing techniques.

A compelling focus was placed on *Code Defenders*, a unique educational tool designed to teach mutation testing. *Code Defenders* stands out as one of the few tools specifically addressing mutation testing, providing an interactive, game-like environment where students learn by creating and defending against mutants. The chapter highlights how *Code Defenders* helps students grasp complex testing techniques and fosters collaborative learning through its competitive and cooper-

ative gameplay. However, while *Code Defenders* offers an engaging and interactive approach to learning mutation testing, its limitations affect its effectiveness in teaching this complex concept entirely. First, it does not utilize actual mutation testing tools; instead, students simulate the code mutations, which may not accurately represent the subtle, real-world defects that professional mutation testing tools generate. Additionally, the tool does not incorporate real-life projects, limiting learned skills' practical relevance and applicability. This lack of real-world context may prevent students from understanding how mutation testing is applied in professional settings. Finally, *Code Defenders* only provides basic metrics, such as code coverage highlights, and does not offer comprehensive mutation testing metrics like mutation score, which are critical for evaluating the effectiveness of mutation testing efforts.

Chapter 3

The FRAFOL Tool

To simplify the setup process for creating a learning environment for mutation testing, it was essential to develop a framework that focused on experimenting with mutation tools without the burden of complex configurations.

This chapter introduces the FRAFOL tool, delving into its core components, design, and implementation. It also highlights the unique advantages FRAFOL offers as a solution to the challenges outlined in the previous paragraph.

3.1 First Iteration

The development of the FRAFOL tool was an iterative process. The first implementation began as a simple UI extension of the Defects4j framework developed in Python. It was conceptualized as part of an exercise assignment where students had to run a virtual machine using Ubuntu OS and manually install all the required dependencies to execute the experiment. The primary purpose was to streamline the process by which students gathered data from mutation tools, avoiding the necessity to manually run command lines through a system terminal.

The first design iteration lets users select a subject from the Defects4j library of projects, allowing students to experiment with realistic software implementations. This helps accurately depict how students may apply software testing in a professional setting.

Figure 3.1 shows the main graphical panel where users interact with the tool. This panel is divided into multiple sections. The top section features information on the project selected for the experiment (in this example, project Lang with version 53f) and a button that prompts a system terminal window with code and condition coverage information. This data is vital because it provides complementary insights into the effectiveness of test suits. The following section includes a form to indicate the directory path to the student's developed test suite. The third section offers a selection menu for the mutation tool that will be applied to the project's source code. This operation will instruct the Defects4j framework to execute the mutation tool and run the selected test suites against the generated code. The tool supports multiple mutation tools like PIT, Major, and Judy, as this was a strategic design choice. Each tool has unique features, mutation operators, and

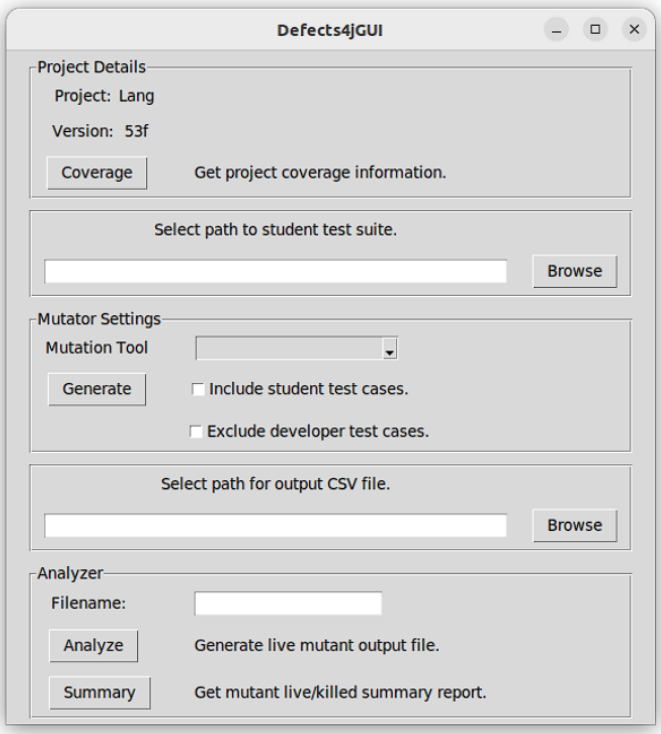
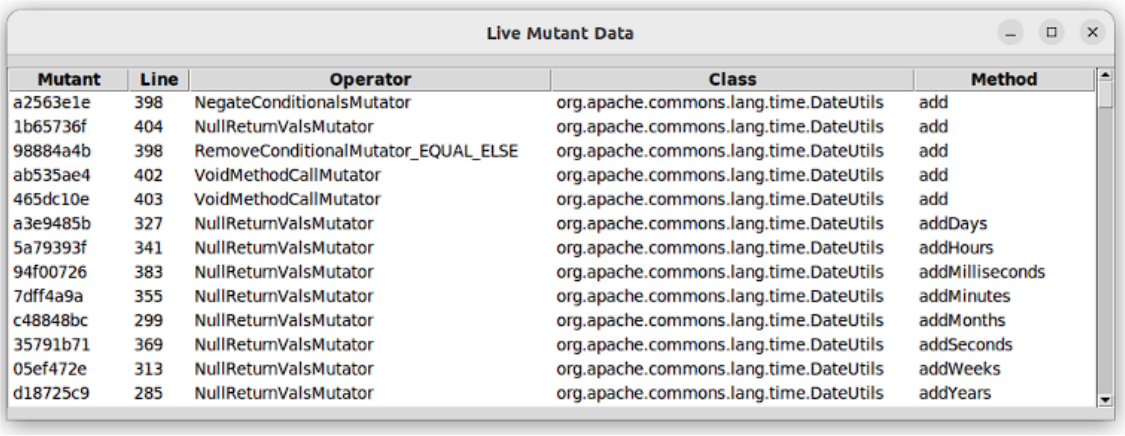


Figure 3.1: Defects4j graphical interface extension.

approaches to analyzing code, which allows students to compare and contrast different methodologies. This variety encourages deeper understanding, as learners can observe how different tools identify potential flaws in the same codebase, reinforcing key concepts in software quality assurance. Additionally, by familiarizing students with multiple tools, the learning platform equips them with practical skills that are more transferable across various professional environments where these tools might be used. The remaining features include the option to create an output CSV file that gathers data on remaining live mutants. These mutants are categorized with a mutant identifier, the source code line of their location, the mutant operator applied by the mutation tool, the object class, and the targeted class method. This information portrays the students' objective goal, which is to eliminate the highest number of live mutants still present in the table. Lastly, users may obtain a report on the mutant score, indicating the ratio of the live and dead mutants.

While this extension was a step in helping students experiment with mutation tools without having to learn the intricacies of navigating the defects4j framework, it still did not answer all the requirements for a uniform environment for learning. The following chapter will delve into the feedback provided by class students that highlights some flaws found during an assignment and how it helped formulate the design choices for the second iteration of FRAFOL.



Mutant	Line	Operator	Class	Method
a2563e1e	398	NegateConditionalsMutator	org.apache.commons.lang.time.DateUtils	add
1b65736f	404	NullReturnValsMutator	org.apache.commons.lang.time.DateUtils	add
98884a4b	398	RemoveConditionalMutator_EQUAL_ELSE	org.apache.commons.lang.time.DateUtils	add
ab535ae4	402	VoidMethodCallMutator	org.apache.commons.lang.time.DateUtils	add
465dc10e	403	VoidMethodCallMutator	org.apache.commons.lang.time.DateUtils	add
a3e9485b	327	NullReturnValsMutator	org.apache.commons.lang.time.DateUtils	addDays
5a79393f	341	NullReturnValsMutator	org.apache.commons.lang.time.DateUtils	addHours
94f00726	383	NullReturnValsMutator	org.apache.commons.lang.time.DateUtils	addMilliseconds
7dff4a9a	355	NullReturnValsMutator	org.apache.commons.lang.time.DateUtils	addMinutes
c48848bc	299	NullReturnValsMutator	org.apache.commons.lang.time.DateUtils	addMonths
35791b71	369	NullReturnValsMutator	org.apache.commons.lang.time.DateUtils	addSeconds
05ef472e	313	NullReturnValsMutator	org.apache.commons.lang.time.DateUtils	addWeeks
d18725c9	285	NullReturnValsMutator	org.apache.commons.lang.time.DateUtils	addYears

Figure 3.2: Defects4j GUI live mutant data table.

3.2 Second Iteration

FRAFOL underwent significant remodeling after evaluating its first iteration, which revealed several key issues. The initial version was hindered by a complex setup process that made it difficult for users to get started. Additionally, its graphical design was unintuitive, causing confusion and frustration for users trying to navigate the tool. The output data generated by FRAFOL was also found to be inconsistent, affecting the results' reliability. Furthermore, the overall user experience was not user-friendly enough, making the tool challenging to use effectively. These issues prompted a comprehensive redesign to enhance its functionality, ease of use, and overall effectiveness as an educational resource.

3.2.1 Design

The core concept is creating a standard environment where students can experiment with different mutation tools. For the initial stage, choosing which mutation tools were best adapted to be used in a learning environment was necessary. For this purpose, we opt for Major and PIT because of their extensive use in research and practice while having complementary features (e.g., source code vs bytecode mutations). These mutation tools are not specific to FRAFOL, meaning further tool extensions can be integrated. While the intent is to shield students from the technical complexities of utilizing these tools, FRAFOL still offers visual distinction from the output quality, allowing them to observe different types of mutant operands and which part of code gets altered. A decision was made for Judy not to be included in the new version because of its inconsistent output, which could lead to unreliable results in mutation testing. Additionally, since Judy is no longer actively maintained, it lacks updates and support, making it less dependable for modern testing. These issues undermine its effectiveness as a teaching tool, prompting the decision to focus on more reliable and up-to-date mutation testing tools.

FRAFOL works as an extension of Defects4j (Figure 3.3), which in itself is a framework that offers a collection of projects with different versions of reproducible bugs, an infrastructure of

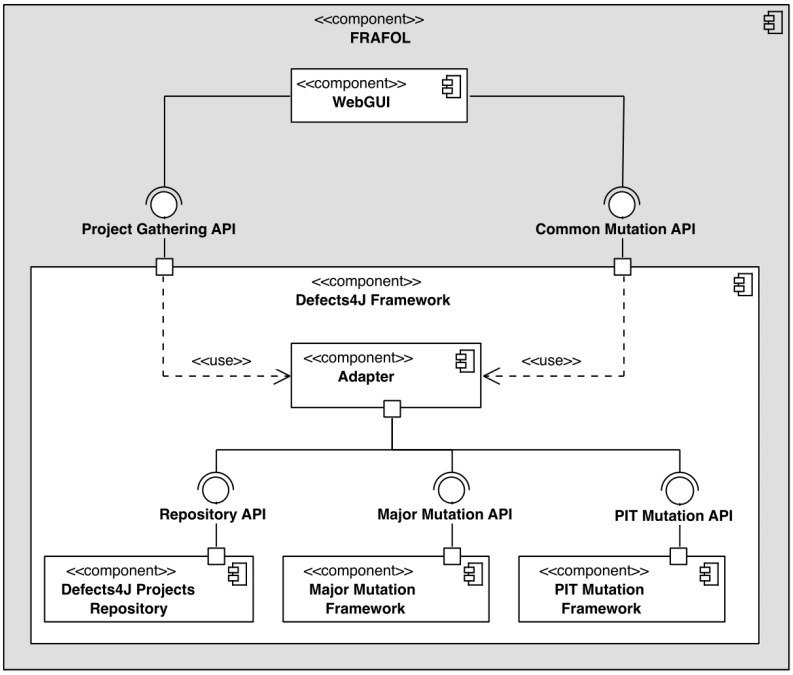


Figure 3.3: UML Component Diagram of the FRAFOL tool.

tools and scripts and is widely used for research and education. FRAFOL uses the Defects4j benchmark as a core component of its design, which natively interfaces with the Major mutation testing tool. Defects4j was the optimal choice because it offered valuable features, such as compilation and testing scripts, while providing realistic subjects with which to experiment. The framework was later updated to support additional mutation tools, such as PIT, and is planned for potential future integrations.

The primary design philosophy is to create an environment that is both easy to install and user-friendly, enabling students to quickly immerse themselves in the subject of mutation testing. This approach allows them to observe mutation effects and write test codes that enhance the quality of the test suite. To achieve this goal, the following features were implemented:

- **Interface** — A web interface was required to encompass all the available framework components and enhance accessibility and usability.
- **Real-world project selection** — Access to the Defects4j database of real-world projects to experiment with.
- **Mutation tool selection** — Provide the option to work with different mutation tools for each experiment (PIT or Major).
- **Integrated Development Environment** — Provide users an IDE with basic functionalities for writing test cases.
- **Error feedback** — Provide the user with compilation error information regarding their test suite.

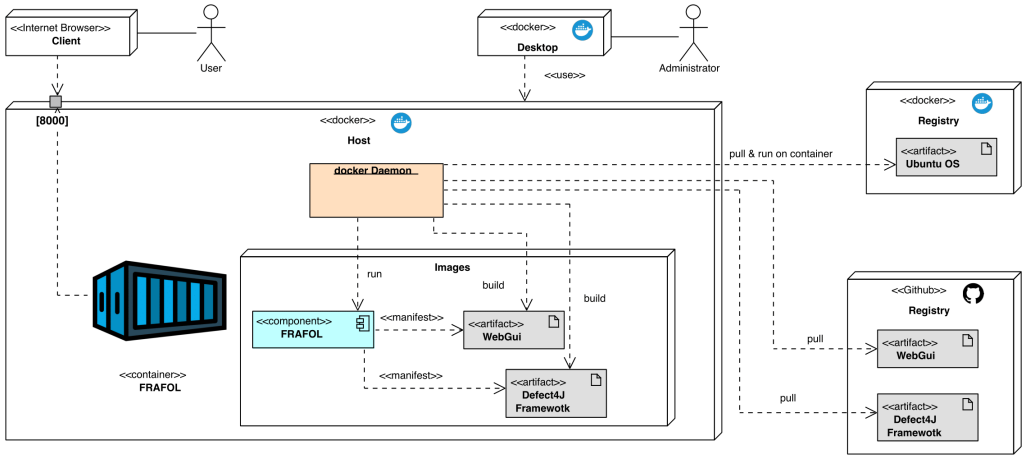


Figure 3.4: UML Deployment Diagram of the FRAFOL tool.

- **Mutation tool execution** — Allow a means to execute the mutation tool while providing the option to run the test suites against the generated code.
- **Mutant data analysis** — Provide detailed information regarding the live mutants after running the mutation tool (mutantID, operand, source code line, mutated method).
- **Code metrics** — Detailed statistics on code coverage, condition coverage, and mutation score related to the source code under mutation.
- **Additional information** — Provide visual, easy access to the targeted source code of the class being mutated and the base developer test suite related to that class.

3.2.2 Implementation

FRAFOL is integrated into a Docker-based, containerized architecture to facilitate end-user experience by simplifying setup and ensuring consistency in the working environment and control of dependencies.

As illustrated in Figure 3.4, Docker implements a client-server architecture, whereby communicating with the docker daemon allows for the build and creation of containers. For this implementation, we build an image that runs an instance of Ubuntu with the FRAFOL tool while also exposing port 8000.

FRAFOL also provides a WebGUI component implemented in Python using Flask. The Flask application provides an API allowing the end-user to interface with the tool using their local browser with the URL `http://localhost:8000/`.

3.2.3 User Interface

The user interface is a core component of the FRAFOL tool design. As stated, FRAFOL implements a WebGUI, which is the best option for interfacing with a docker container. The landing page of FRAFOL (Figure 3.5) presents a start menu that contains two selection forms. The first

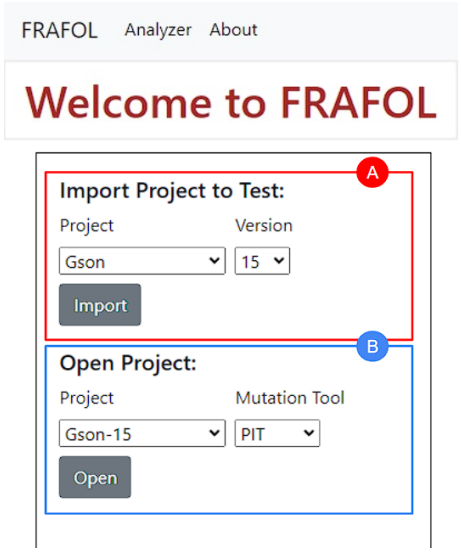


Figure 3.5: FRAFOL start menu page.

selection (A) allows the user to import a project version, which results in the tool obtaining and compiling the source code. The second selection form (B) includes all the previously imported projects the user has begun working on, including the recently imported ones, and allows them to be open.

Figure 3.6 presents the main view for FRAFOL mutation analysis integrated environment. In sections (A) and (B), there are four navigation tabs: the two present in (A) have one displaying a dashboard featuring infographics related to code coverage, condition coverage, and mutation score data, and the other displaying the source code for the class under mutation; the tabs present in (B) have one displaying the base written developer test cases for the class under mutation, and the other has an integrated Codemirror IDE for users to write their own test cases. Section (C) features a button to execute the mutation tool and run the test suites, with the additional options of including or excluding the developer and student test suites. Compilation data will be displayed in section (E) if any errors are detected. After invoking a mutation tool, section (D) will display a table containing details about the active mutants. This information includes the mutant's identifier, the source code line where the mutation occurred, the type of operand applied by the tool, and the affected class method. If a mutant is successfully killed by the test suite, it will be removed from the table during subsequent executions of the mutation tool. Each table entry, when clicked, hyperlinks to the code line in the project source code tab to facilitate users in rapidly finding the contextual information necessary to understand the mutation effect.

3.2.4 Use Case Story

To better comprehend the end-user experience, this subsection will detail a possible use case scenario of FRAFOL. The user must start by theoretically understanding mutation testing and its core

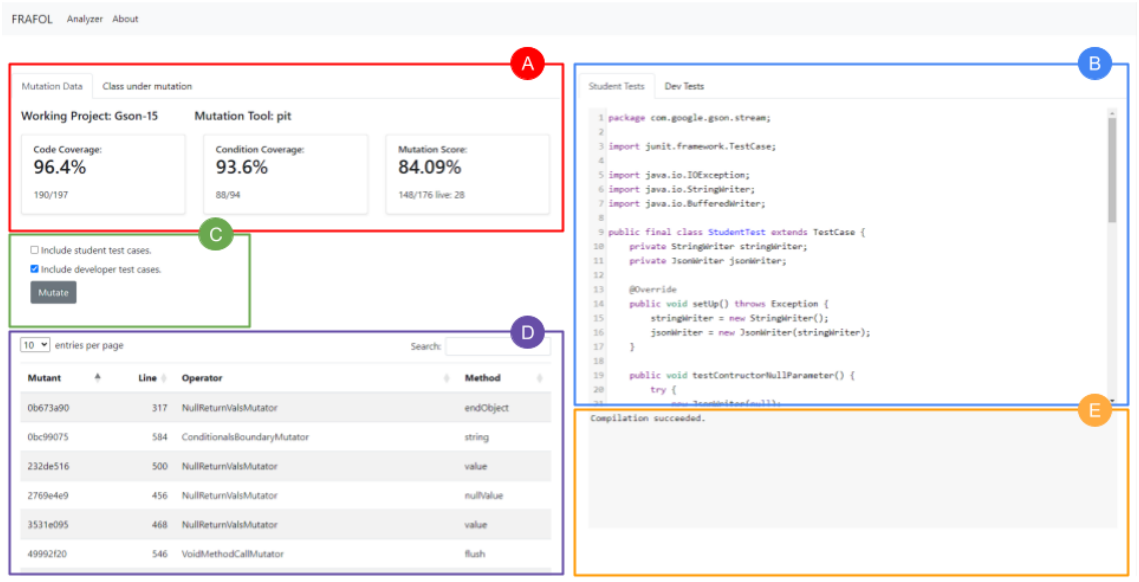


Figure 3.6: FRAFOL Mutation analysis integrated environment.

concepts. With the appropriate knowledge requirements, practical study may begin by launching FRAFOL and picking a project and mutation tool for experimentation.

The first user task is to familiarize himself with the project source code by opening the *Class Under Mutation* tab (A) (Figure 3.7) and reading the comment lines to gain a brief understanding of the class methods functionality. Second, he will analyze the statistics in the Mutation Data tab and check how much code and condition statements are being covered by the base developer test cases by default. Once the user is ready to advance, an initial run of the mutation tool is required to populate the live mutant table by clicking the *Mutate* button and including the developer test suite. This operation will generate code mutations and execute the test suite against the generated code. At this stage, the user begins analyzing the remaining live mutants present in table (D) (Figure 3.6), which were not killed by the developer test cases. For this example, the user will attempt to eliminate the specific mutant `0082f1cf` (Figure 3.8), starting with the following steps:

- Identify the mutant operator `VoidMethodCallMutator`.
- Consult the tool documentation for information on the mutant behavior, as observed in figure 3.9. The identified `VoidMethodCallMutator` is a PIT mutation operator that removes method calls to void methods.
- Observe method `printHelp` in line 382 and identify which void method has been removed.

Following the previous analysis, the user begins conceptualizing a test case to eliminate the specific mutant and may check the developer's test suite (B) (Figure 3.7) for reference on existing tests that potentially cover the altered method. Once the user completes writing a test, a re-run of the mutation tool, including the student test cases, is required to update the live mutant table.

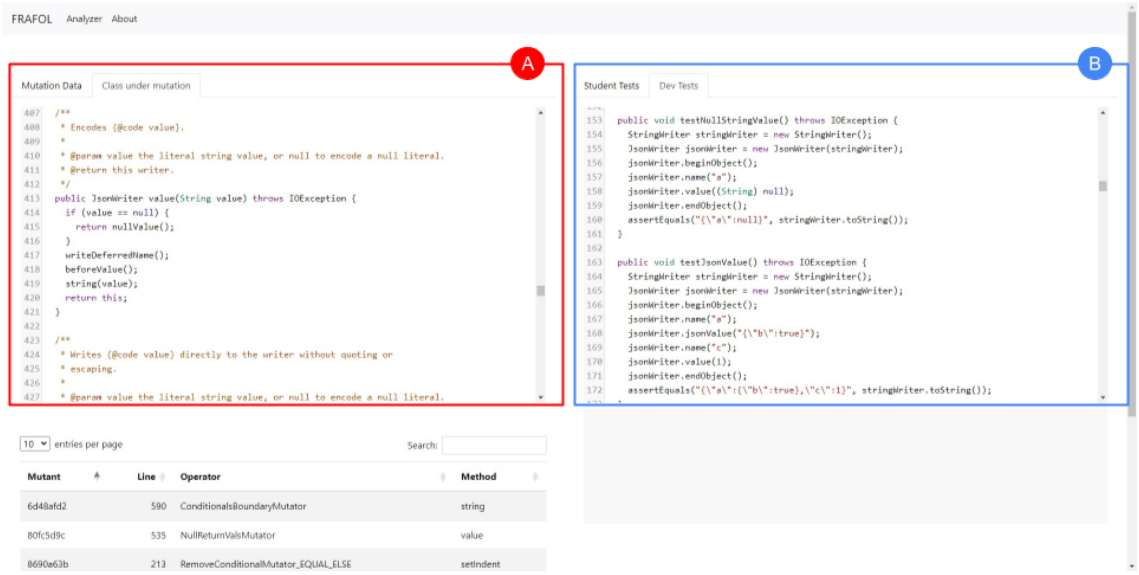


Figure 3.7: FRAFOL Mutation analysis integrated environment 2.

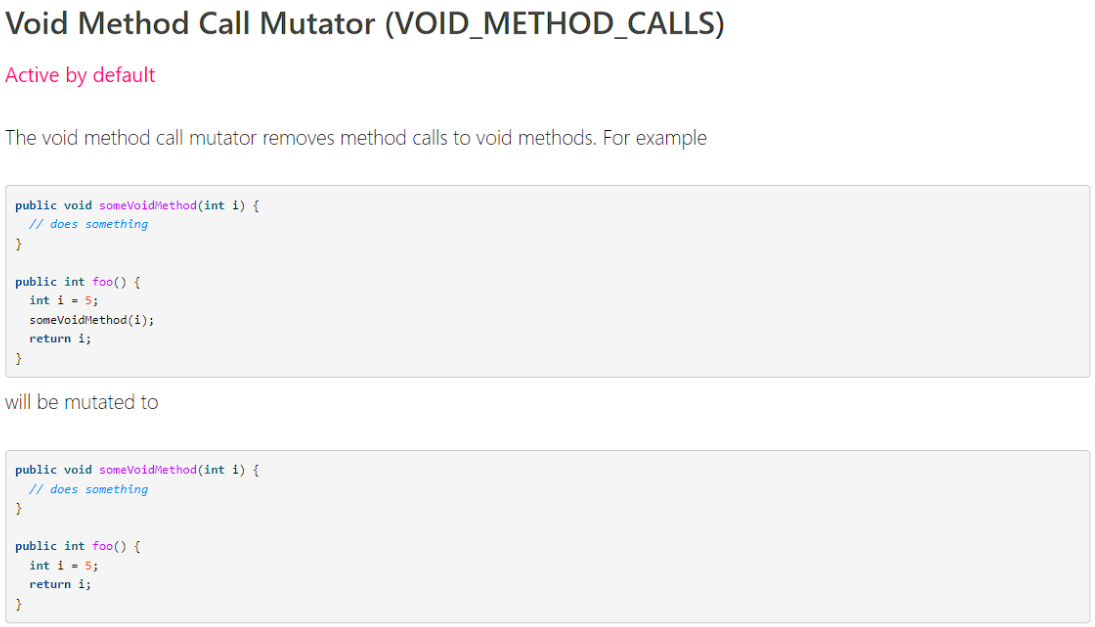
If the test case is successful, the target mutant will no longer be present, and the mutation data tab should reflect a change in the coverage and mutation score. The user may continue exercising further test cases until the same coverage and mutation score criteria are satisfied. This exemplifies the expected use case scenario with the FRAFOL tool.

3.2.5 Tool Availability

FRAFOL is available at `https://github.com/projFRAFOL/projFRAFOL` (see README.md for details), and a video demonstration is available at: `https://youtu.be/JMvGskRQre8`. As of writing, the current version of FRAFOL only supports the Gson-15 and Cli-32 project versions. An updated working tool version is being developed with a generalization that will include other Defects4J project versions.

Mutant	Line	Operator	Method
0082f1cf	382	VoidMethodCallMutator	printHelp

Figure 3.8: Sample entry line of live mutant table.



Void Method Call Mutator (VOID_METHOD_CALLS)

Active by default

The void method call mutator removes method calls to void methods. For example

```
public void someVoidMethod(int i) {
  // does something
}

public int foo() {
  int i = 5;
  someVoidMethod(i);
  return i;
}
```

will be mutated to

```
public void someVoidMethod(int i) {
  // does something
}

public int foo() {
  int i = 5;
  return i;
}
```

Figure 3.9: PIT documentation on mutant operator (https://pitest.org/quickstart/mutators/)

Chapter 4

Experiments and Validation

As mentioned in the previous chapter, FRAFOL underwent an iterative development cycle. It is critical to validate theoretical models in practical, real applications to understand if all objective criteria are being satisfied. Experimenting with FRAFOL is essential to ensure that the conceptualized model performs as expected under stipulated conditions, such as classroom assignments or personal self-taught initiatives. Furthermore, experimental validation helps identify unforeseen issues or limitations that may not be evident during development.

The primary objective of this experiment was to assess users' learning quality experience, how effectively the tool is integrated into a learning environment, and overall tool usability satisfaction. Two user studies were conducted: a formative study that informed the design of FRAFOL and a summative study that evaluated the most recent version of the tool.

This chapter details how both studies were conducted, what validation analysis was performed, and some concluding thoughts on the experiments.

4.1 Experiments

Two experiments were performed to validate the initial and final iterations of the software tool application to ensure a comprehensive evaluation of its performance and reliability. The first experiment focused on the initial iteration to identify any potential flaws, limitations, or areas for improvement, providing a baseline for further development. After an extensive update of the tool, a second experiment was necessary to confirm that the improvements effectively addressed the issues identified in the initial test, enhanced the tool's functionality, and met the desired requirements.

4.1.1 First design experiment

The Defects4j graphical interface extension functioned as the first design prototype for what later came to be the FRAFOL tool. To understand the applicability of its core design features, it was necessary to test it in an active learning environment. The initial prototype was introduced as a

part of an assignment designed to teach students the importance of mutation testing. This experiment was performed in a course dedicated to teaching all aspects related to software testing, with a classroom composed of 35 students in their first year of master's degree in software and computation engineering, in the optional class of TVVS.

Students were informed to follow a set-up guideline that required using a virtual machine with Ubuntu OS and a series of command instructions to install all dependencies necessary to run the Defects4j framework. Students were then given a specific Java project from within the Defects4j library, for which they would apply a designated mutation tool and develop an extended test suite to kill all non-equivalent mutants. The test suites must be written using JUnit 3 or 4, depending on the project base requirements. Lastly, students were tasked to fill out a spreadsheet in Microsoft Excel format, as shown in Figure 4.1, detailing their analysis of each mutant regarding their equivalency, productivity, and respective rationales for each qualification. This experiment took place during November and December of 2023. At the end of it, students were given a questionnaire[1] which consisted of closed questions with responses based on a level-5 Likert scale, ranging from 1 (strongly disagree) to 5 (strongly agree). Questions were divided into multiple categories that related to specific aspects of the tool:

- **Usability** — How easy was it to use/follow the tool and to find all the necessary information to complete the task?
- **Learnability** — How practical was the tool as a learning assistant, and what level of prior knowledge is required?
- **Satisfaction** — How does the tool positively contribute to the learning experience?
- **Difficulties** — How does the tool affect the learning experience negatively?
- **Effectiveness** — Does the tool satisfy its design criteria for improving learning productivity?
- **Usefulness** — How does the tool satisfy the student's criteria for learning about mutation testing?
- **Feedback** — How complete/helpful is the information relayed by the tool?
- **Interface** — How pleasant, clear, and understandable is the graphical interface for the tool?

4.1.1.1 Results and analysis

Before looking at the resulting data, it's important to remark that both experiments had considerably different stages. The first experiment, conducted throughout a month-long assignment as part of a course lecture, provides a more accurate depiction of the target environment for this type

[1] https://forms.gle/m8FJTGbev8odTNTe9

ID of analyzed mutant	Is the mutant equivalent?	Provide a rationale for equivalence	Is the mutant productive?	Provide a rationale for why you think it is productive or unproductive	Test inputs (if mutant is not equivalent)
1000	No	-	Yes	It is productive because ...	1, 2, 3
5000	Yes	It is equivalent because...	No	-	-
10000	No	-	No	-	-1, 2, 2
14000	Yes	It is equivalent because...	Yes	It is productive because ...	-

Figure 4.1: Mutant equivalency and productivity analysis table

of tool application rather than an isolated, more controlled experiment, as was the second experiment. Nevertheless, the results obtained proved helpful in understanding the differences in how users perceived all the categories related to the tool as it evolved from its initial design.

The results of the first design experiment were mainly not favorable. As shown in figure 4.2, while the average rating for `Usability` (3.0) and `Learnability` (2.7) was slightly moderate, suggesting that while some users could use the tool and learn it with effort, the overall experience was not smooth or intuitive. The remaining categories present ratings under 2.5, and the highest score for the category `Difficulties` (3.8) represents how much users felt the tool negatively affected their learning experience. Understanding the points of frustration was critical in identifying the necessary changes and, as such, by reflecting upon the interactions with students during the assignment, it was possible to determine the following key improvements:

1. **Simplify setup requirements** — The installation process was too extensive and demanding, as many students reported issues setting up the framework, often due to a lack of understanding or mistakes while following the guidelines. This resulted in a considerable time loss when configuring individual user computers and contributed to a negative learning experience.

2. **Streamline and unified UI** — While the purpose of the Defects4j extended graphical interface was to abstract users from requiring to learn how to operate with the Defects4j framework, it was not enough. Students were still required to navigate multiple components to access all the necessary information. Data metrics on code coverage, condition coverage, and mutation score were detached from the main UI interface. Users had to navigate the project folder to access the mutated class's source code and the developer test suite. Lastly, a 3rd party IDE software was required for students to write their own test cases. The purpose of this module is to remove barriers to the learning experience and maximize engagement, which led to the decision to unify and streamline the platform.

3. **Common working environment** — Although students were required to use a virtual machine and follow the same setup guidelines for the experiment, noticeable inconsistencies

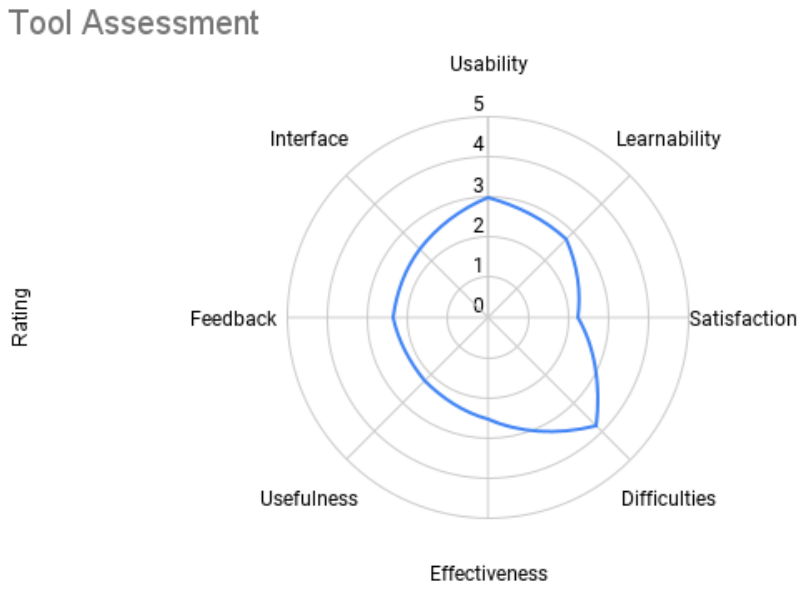


Figure 4.2: First experiment tool assessment results.

were observed in the output data across different machines. To ensure consistency in the tool performance, a solution was necessary to achieve a common environment with efficient dependency management.

The following segment will reveal some collected data on student performance related to the experiment. Due to the lack of reference data on exercises in previous years, it is not possible to qualitatively analyze how well students accomplished their tasks. However, some observations can still be made on the results of each individual mutation tool.

4.1.1.2 Mutation Score Analysis

The data in table 4.1 shows the number of mutations generated by each tool in the framework. It's possible to observe that Judy produces, on average, a more significant number of mutants than other tools while also demonstrating an inconsistent variation between each project, as seen by the value of project `Cli`. This disparity in number can also be explained by Judy's tendency to produce an increased number of equivalent mutations.

Project	PIT	Major	Judy
Cli	173	340	1058
Gson	176	275	283
Lang	238	466	488

Table 4.1: Mutants produced by each tool.

Table 4.2 demonstrates the reported mutation score the base project developer test suites achieved. The scores reveal that many mutants are already eliminated by the developer test cases, leaving students to eliminate an average of 19.63% of the remaining mutants on each project. A particular distinction in Judy's results shows that the test suites have a higher difficulty in eliminating mutants, attributed mainly to the superior equivalency rate.

Project	PIT	Major	Judy
Cli	87.28%	80.60%	67.20%
Gson	84.09%	84.07%	80.57%
Lang	89.92%	77.25%	72.34%

Table 4.2: Default mutation scores.

As such, the expected percentage increase of mutants killed by students' test suites for Judy will be lower than other tools. This is verifiable by the results present in table 4.3, which shows a minor increase for projects `Cli` and `Gson`, with the outlier of 10.18% increase on project `Lang`. Students had an easier time eliminating mutants while using Major, with an average rise of 8.94%, but not significantly more than PIT (average rise of 8.06%).

Project	PIT	Major	Judy
Cli	94.63% (+7.35%)	87.60% (+7.00%)	72.46% (+5.26%)
Gson	95.59% (+11.50%)	92.06% (+7.99%)	84.54% (+3.97%)
Lang	95.24% (+5.32%)	89.10% (+11.85%)	82.51% (+10.18%)

Table 4.3: Average mutation score with student test suites.

The last table (table 4.4) completes the information with the average number of mutants killed. Students were able to eliminate an average of 41.54% of the remaining live mutants. This can be considered a good number, as a portion of live mutants can be attributed to equivalent mutants, and generally, depending on the project and mutation tool, although hard to determine, the percentage of equivalent mutants can fall between 10% and 30% of total mutants (Ayad et al. (2019), Grün et al. (2009)).

Project	PIT	Major	Judy
Cli	12.71	29.33	53.20
Gson	20.25	22.50	11.40
Lang	13.86	59.00	49.67

Table 4.4: Average mutants killed.

4.1.1.3 Line and Condition Coverage Analysis

Project	PIT	Major	Judy	Default
Cli	99.07%	97.03%	99.67%	96.50%
Gson	99.75%	99.25%	99.50%	96.40%
Lang	99.23%	98.63%	97.85%	97.10%

Table 4.5: Line coverage scores.

Line coverage is an essential metric in software testing as it measures the percentage of code lines executed during testing. High code coverage indicates that most of the code has been tested, reducing the likelihood of hidden bugs. It's also one of the primary criteria for test suite quality by industry practitioners (Eric Wong et al. (2018)). The column `Default` on table 4.5 demonstrates that coverage levels were already significantly high, with an average of 96.67% coverage achieved by the developer test suite. However, the extended student test cases raised that percentage to nearly full coverage in some cases.

Lastly, condition coverage is crucial in software testing because it ensures that all logical conditions within the code are evaluated for both true and false outcomes. Unlike line coverage, which only verifies that each line of code is executed, condition coverage focuses on testing the decision points in the software, such as if-else statements and loops. Also, achieving condition coverage usually requires more test cases than code coverage because it involves testing multiple combinations of boolean conditions, even within the same line of code. As shown in table 4.6, the same number of test cases could not reach equivalent coverage levels as the previous report.

Project	PIT	Major	Judy	Default
Cli	96.70%	95.80%	94.30%	93%
Gson	98.90%	98.95%	98.57%	93.60%
Lang	94.30%	98.90%	92.60%	90.30%

Table 4.6: Condition coverage scores.

4.1.2 Second design experiment

This summative study comes after the tool re-design development phase, which saw significant changes to the original concept of FRAFOL. Following such redesign, a new tool validation experiment is essential to ensure the changes have not compromised the tool's functionality. Even if the redesign is intended to improve upon its original iteration, such changes can introduce new variables, and it's crucial to identify that the latest version still meets its intended purpose and performs tasks as expected.

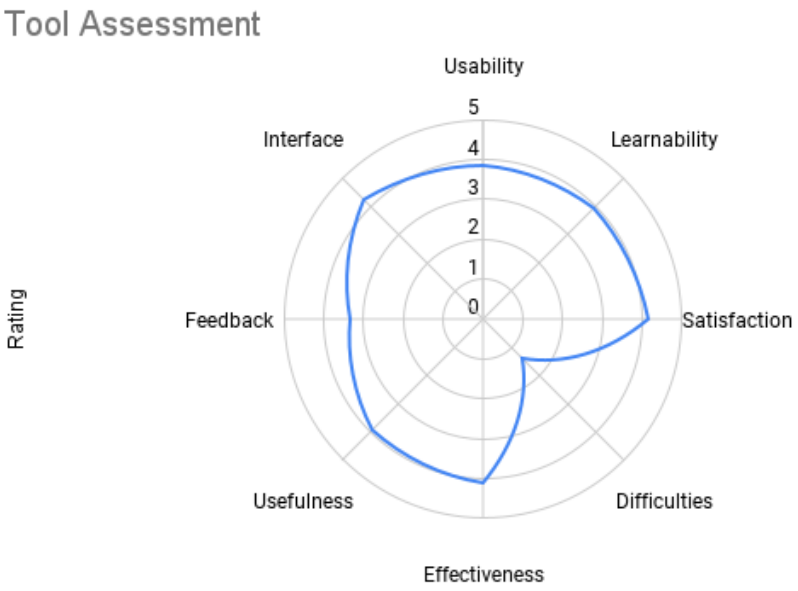


Figure 4.3: Second experiment tool assessment results.

This study asked four MSc students to evaluate various aspects of FRAFOL, such as usability and complexity (see figure). All students had prior knowledge of software testing, including mutation testing methodology. This experiment was performed remotely, with each participant's personal computer, and involved the following steps:

First, participants were given an installation guideline for the FRAFOL tool, which aimed to take approximately 15 minutes to complete. Second, after a brief presentation of the tool, participants were given a list of four tasks to complete within the span of 35 minutes:

1. Select the Gson project, specifically version 15, and choose the PIT mutation testing tool.
2. Analyze the base developer test suite of the project and assess their code coverage and mutation score (**C** in Figure 3.4).
3. Analyze the results (**D** in Figure 3.4) and write a `JUnit test class` (**B** in Figure 3.4) to target a specific mutant from the list of live mutants (**D** in Figure 3.4).
4. Reevaluate the mutation coverage and score, this time including developer and student test cases (**C** in Figure 3.4) and verify the elimination of specified mutant.

Finally, the participants completed a questionnaire[2], which consisted of closed questions with responses based on a level-5 Likert scale, ranging from 1 (strongly disagree) to 5 (strongly agree), with categories following the same description as the first experiments questionnaire.

[2] https://forms.gle/V2C3tqaZWK4YsYkK9

4.1.2.1 Results and analysis

For the second experiment, the resulting data was considerably more optimistic. All participants completed the setup process within the expected time frame. This is an important benchmark that validates the choice of Docker as an appropriate tool for staging FRAFOL for future use. The results in figure 4.3 indicate that users were more appreciative of the new `Interface` (4.3) design, with more clean and clear labels. Users were able to quickly familiarize themselves with all components of the tool. They believed that FRAFOL could significantly enhance their learning `Effectiveness` (4.1) and found it to be `Useful` (4.1). The `Satisfaction` (4.2) had a significant boost compared to the first design, revealing a positive outlook on tools' impact on the learning experience. In high contrast to the first study, the `Difficulties` (1.4) score is strikingly lower. However, it's relevant to consider that these scores reflect upon a much shorter interaction time frame and should be analyzed as a first impression rather than a prolonged use case. FRAFOLs' `Usability` and `Learnability` both received a score of 3.9, demonstrating that users were quick to understand how to use the tool with minimal effort. This is crucial because the faster users can grasp the tool's features and functionalities, the sooner they can focus on actual learning content. Despite the positive results, there is still clear room for improvement. Although positive, the `Feedback` (3.4) score still reveals a demand for additional contextual information and performance indicators. Some comments highlighted the need to visualize the impact of tests on code by identifying all the eliminated mutants by a specific test and which lines of code achieved coverage through additional test cases. Every participant was able to apply their knowledge and complete the designated task successfully within the estimated time.

4.2 Conclusions of Results

The FRAFOL tool was evaluated through two distinct experiments that differed significantly in their context and duration. The first experiment was embedded within a two-month-long academic assignment, providing a realistic environment but yielding unfavorable results. The second experiment, a shorter and more controlled study, resulted in a more positive user evaluation. The differences in these outcomes provide valuable insights into the tool's evolution and the impact of the modifications made between the two experiments. The initial study highlighted significant issues with the initial design of FRAFOL. Most user ratings were below 2.5, indicating general dissatisfaction. This accentuates the importance of providing a well-thought-out process when designing a learning aid component, lest it becomes the antithesis of the principal goal of enhancing the education experience and instead contributes to it negatively. While it wasn't possible to perform a qualitative analysis of the student's results, it is hopeful that in future editions of TVVS, additional data will be added to provide a better understanding of the impact of FRAFOL on practical performance.

The transition from the first to the second experiment highlights the importance of iterative design and user feedback in developing educational tools. The significant improvements (Figure 4.4)

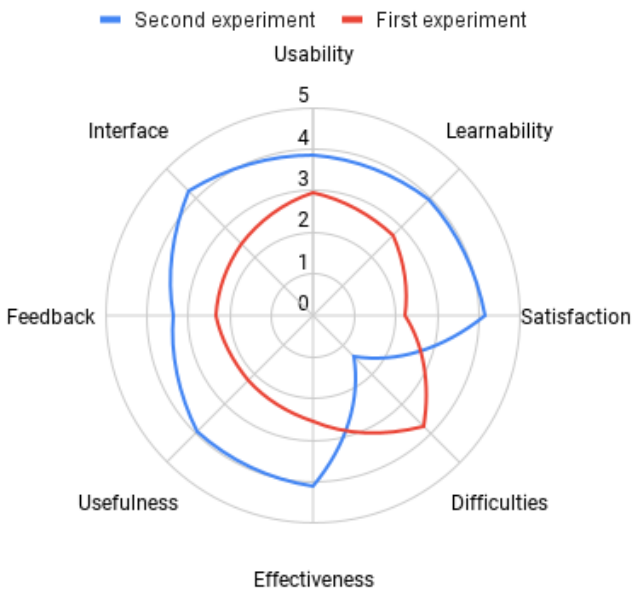


Figure 4.4: Tool assessment comparison graph.

in user satisfaction, usability, and overall experience between the two experiments suggest that the changes implemented were successful. However, the analysis indicates that there is still room for enhancement, particularly in providing more detailed and contextualized data within the tool. There's a continuous strong indication that giving feedback and contextual information significantly contributes to the engagement and motivation of learners, particularly when tied to visual stimulation. Immediate, visually engaging feedback helps learners understand their progress, reinforcing correct actions, prompting course corrections when needed, and serving as a reward system when performing positive results. Future iterations should focus on these aspects to refine FRAFOL and ensure it fully meets user needs in various educational settings.

Chapter 5

Conclusions and Future Work

Continuing research on software testing education is vital to the software industry's advancement and future professionals' preparation. Ongoing research in this field allows for developing innovative educational approaches, such as gamified learning and testing aid tools, which engage students and improve learning outcomes. Additionally, continuously exploring pedagogical methods tailored to software testing ensures that education remains aligned with industry trends, technological advancements, and evolving best practices.

This document introduced FRAFOL, a tool designed to provide a user-friendly platform that supports educators and helps students master mutation testing techniques. FRAFOL was developed with three main objectives: simplifying setup and configuration requirements, offering a unified testing environment, and providing real software project samples for practical testing.

The tool underwent two design iterations. During the first iteration, a summative study identified several areas for improvement. These issues stemmed from an overly complicated installation process that required too many steps and dependencies and an unintuitive UI design that detracted from the focus on performing testing tasks. After careful consideration, it was concluded that incremental fixes were insufficient, prompting the need for a comprehensive redesign.

The results of the second summative study showed significant improvements over its initial design. Docker significantly simplified the setup requirements for the tool by providing a consistent and isolated environment where the tool can run without the complexities of manual configuration. By containerizing the tool and its dependencies, Docker ensures that users can avoid issues related to differences in operating systems, software versions, or system settings. Instead of installing and configuring various components individually, users can run a Docker container, which includes everything needed for FRAFOL to function correctly. Positive user satisfaction is highly attributed to the new interface, which presents a cleaner layout and provides all the necessary components for experimentation in a single frame. However, the study also highlighted the need for further work on enhancing the feedback mechanism.

The tool successfully met the study's goals and demonstrates strong potential for future use of FRAFOL in classroom settings. Additionally, a tool demonstration paper presenting FRAFOL has been submitted and accepted for ISSTA 2024, the leading symposium on software testing and

analysis.

5.1 Future Work

The current version of the educational tool FRAFOL is fully prepared for live deployment, offering a functional and efficient platform for users. However, this iteration presents opportunities for further enhancement. One key area for improvement is the live mutant report table. While it successfully lists live mutants, it could be improved by retaining the display of killed mutants without removing them, which would provide a clearer representation of how the test suite interacts with the mutation tool.

Additionally, the tool's code coverage presentation can be enhanced by highlighting the lines of code that are currently covered. This would provide users with a more intuitive understanding of which sections of their code are effectively tested.

Another improvement is the incorporation of more detailed information about mutation operands directly within the tool. This would reduce the need for users to consult external documentation, improving the overall user experience.

The current code editor offers basic functionality. While the goal is not to compete with industry-level IDE software, additional quality-of-life features would offer a more pleasant experience such as auto-complete code for Junit testing methods.

Lastly, implementing a teacher view for automated analysis of student work would greatly benefit educators. This feature would enable instructors to provide more timely feedback and access valuable analytics for grading, improving both student outcomes and teacher efficiency.

References

A. Allowatt and S. Edwards. Ide support for test-driven development and automated grading in both java and c+. In *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange, eclipse'05*, pages 100–104, 2005. doi: 10.1145/1117696.1117717. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-40749108722&doi=10.1145%2f1117696.1117717&partnerID=40&md5=0f452e6574de2df28062183bf8e8ec6e. Export Date: 03 July 2023; Cited By: 21.

Domenico Amalfitano, Ana C. R. Paiva, Alexis Inquel, Luís Pinto, Anna Rita Fasolino, and René Just. How do java mutation tools differ? *Commun. ACM*, 65(12):74–89, 2022. ISSN 0001-0782. doi: 10.1145/3526099. URL https://doi.org/10.1145/3526099.

Amani Ayad, Imen Marsit, JiMeng Loh, Mohamed Nazih Omri, and Ali Mili. Estimating the number of equivalent mutants. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 112–121, 2019. doi: 10.1109/ICSTW.2019.00039.

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.

Mordechai Ben-Ari. Constructivism in computer science education. *Journal of Computers in Mathematics and Science Teaching*, 20(1):45–73, 2001.

R. Blanco, M. Trinidad, M. J. Suárez-Cabal, A. Calderón, M. Ruiz, and J. Tuya. Can gamification help in software testing education? findings from an empirical study. *Journal of Systems and Software*, 200, 2023. doi: 10.1016/j.jss.2023.111647. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-85149184996&doi=10.1016%2fj.jss.2023.111647&partnerID=40&md5=3c0e790ba7c186de4508455845f46310. Export Date: 28 March 2023; Cited By: 0.

Honghong Chen, Xu Wang, and Liangguang Pan. Research on teaching methods and tools of software testing. In *2020 15th International Conference on Computer Science Education (ICCSE)*, pages 760–763, 2020. doi: 10.1109/ICCSE49874.2020.9201788.

P. J. Clarke, A. A. Allen, T. M. King, E. L. Jones, and P. Natesan. Using a web-based repository to integrate testing tools into programming courses. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion, SPLASH '10*, pages 193–200, 2010. doi: 10.1145/1869542.1869573. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-78650104227&doi=10.1145%2f1869542.1869573&partnerID=40&md5=36c5eb238bb200ffea0001c78930372f. Export Date: 03 July 2023; Cited By: 11.

Pedro Delgado-Pérez, Inmaculada Medina-Bulo, Miguel Ángel Álvarez García, and Kevin J. Valle-Gómez. Mutation testing and self/peer assessment: Analyzing their effect on students in a software testing course. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, pages 231–240, 2021. doi: 10.1109/ICSE-SEET52601.2021.00033.

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.

Sebastian Elbaum, Suzette Person, Jon Dokulil, and Matt Jorde. Bug hunt: Making early software testing lessons engaging and affordable. In *29th International Conference on Software Engineering (ICSE'07)*, pages 688–697, 2007. doi: 10.1109/ICSE.2007.23.

ENACTEST. Learning capsules for he and vet providers to prepare future professionals for testing and improving software production. Available at https://enactest-project.eu/,Access Date: 08 July 2023;.

W. Eric Wong, Linghuan Hu, Haoliang Wang, and Zhenyu Chen. Improving software testing education via industry sponsored contests. In *2018 IEEE Frontiers in Education Conference (FIE)*, pages 1–5, 2018. doi: 10.1109/FIE.2018.8658960.

G. Fraser, A. Gambi, and J. M. Rojas. Teaching software testing with the code defenders testing game: Experiences and improvements. In *Proceedings - 2020 IEEE 13th International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2020*, pages 461–464, 2020. doi: 10.1109/ICSTW50294.2020.00082. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-85091769617&doi=10.1109%2fICSTW50294.2020.00082&partnerID=40&md5=57eda767afbc587819e2e66335c65466. Export Date: 03 July 2023; Cited By: 4.

V. Garousi, M. Felderer, M. Kuhrmann, K. Herkiloğlu, and S. Eldh. Exploring the industry's challenges in software testing: An empirical study. *Journal of Software: Evolution and Process*, 32(8), 2020a. doi: 10.1002/smr.2251. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-85079420650&doi=10.1002%2fsmr.2251&partnerID=40&md5=41c1fb7751e75df33c759427b9c0f1e7. Export Date: 28 March 2023; Cited By: 17.

V. Garousi, A. Rainer, P. Lauvås, and A. Arcuri. Software-testing education: A systematic literature mapping. *Journal of Systems and Software*, 165, 2020b. doi: 10.1016/j.jss.2020.110570. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-85082879194&doi=10.1016%2fj.jss.2020.110570&partnerID=40&md5=d2ea0d3167a2817188efc650fc3548d3. Export Date: 28 March 2023; Cited By: 25.

R. F. Gomes and V. Lelli. Gamut: Game-based learning approach for teaching unit testing. In *ACM International Conference Proceeding Series*, 2022. doi: 10.1145/3493244.3493263. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-85147652236&doi=10.1145%2f3493244.3493263&partnerID=40&md5=97b1cba24f8473fde0e3e2b3f36eba57. Export Date: 18 April 2023; Cited By: 0.

Bernhard J. M. Grün, David Schuler, and Andreas Zeller. The impact of equivalent mutants. In *2009 International Conference on Software Testing, Verification, and Validation Workshops*, pages 192–199, 2009. doi: 10.1109/ICSTW.2009.37.

Gabriela Martins de Jesus, Fabiano Cutigi Ferrari, Leo Natan Paschoal, Simone do Rocio Senger de Souza, Daniel de Paula Porto, and Vinicius Humberto Serapilha Durelli. Is it worth using gamification on software testing education? an extended experience report in the context of undergraduate students. *Journal of Software Engineering Research and Development*, 8(0):6:1 – 6:19, 2020. doi: 10.5753/jserd.2020.738. URL https://sol.sbc.org.br/journals/index.php/jserd/article/view/738.

C. Kaner. Fundamental challenges in software testing. *Butler University*, 2003. URL http://kaner.com/pdfs/FundamentalChallenges.pdf.

Leen Lambers. How to teach software testing? experiences with a sandwich approach. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 425–428, 2020. doi: 10.1109/ICSTW50294.2020.00076.

Arcuri A. Lauvås Jr, P. Recent trends in software testing education: A systematic literature review. *The Norwegian Conference on Didactics in IT education* ., 2018.

Xia Huang Li Hua, Qiong Gu and Bin Ning. The blended teaching mode of software testing course in the internet plus context. pages pp. 95–99, 02 2018. doi: 10.2991/icem-17.2018.55.

Major. Easy and scalable mutation analysis for java! Available at https://mutation-testing.org/, Access Date: 08 July 2023;.

O. Ochoa and S. Salamah. An approach to enhance students' competency in software verification techniques. In *Proceedings - Frontiers in Education Conference, FIE*, volume 2015, 2015. doi: 10.1109/FIE.2015.7344050. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-84960462085&doi=10.1109%2fFIE.2015.7344050&partnerID=40&md5=37338eca606fbf10e37e828f77a8e53a. Export Date: 04 July 2023; Cited By: 3.

M. Ojdanic, E. Soremekun, R. Degiovanni, M. Papadakis, and Y. L. Traon. Mutation testing in evolving systems: Studying the relevance of mutants to code evolution. *ACM Transactions on Software Engineering and Methodology*, 32(1), 2023. doi: 10.1145/3530786. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-85152596004&doi=10.1145%2f3530786&partnerID=40&md5=f6264ac9345ba49fe0db2b48328ee945. Export Date: 06 July 2023; Cited By: 0.

Rafael A. P. Oliveira, Lucas B. R. Oliveira, Bruno B. P. Cafeo, and Vinicius H. S. Durelli. Evaluation and assessment of effects on exploring mutation testing in programming courses. In *2015 IEEE Frontiers in Education Conference (FIE)*, pages 1–9, 2015. doi: 10.1109/FIE.2015.7344051.

Tracy Phillips. Is there a shortage of developers? developer shortage statistics in 2022. *CodeSubmit*, 2022. Available at https://codesubmit.io/blog/shortage-of-developers/.

PITEST. Real world mutation testing. Available at https://pitest.org/, Access Date: 08 July 2023;.

V. Ramasamy, H. W. Alomari, J. D. Kiper, and G. Potvin. A minimally disruptive approach of integrating testing into computer programming courses. In *Proceedings - International Conference on Software Engineering*, pages 1–7, 2018. doi: 10.1145/3194779.3194790. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-85051112940&doi=10.1145%2f3194779.3194790&partnerID=40&md5=791b8be8295b56533dee0e2529bb3111. Export Date: 06 July 2023; Cited By: 10.

T. P. B. Ribeiro and A. C. R. Paiva. Ilearntest: Educational game for learning software testing. In *2015 10th Iberian Conference on Information Systems and Technologies, CISTI 2015*, 2015. doi: 10.1109/CISTI.2015.7170608. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-84943340009&doi=10.1109%2fCISTI.2015.7170608&partnerID=40&md5=b998eae2182f96b4cf3a2afaae0f1527. Export Date: 09 July 2023; Cited By: 4.

L. P. Scatalon, M. L. Fioravanti, J. M. Prates, R. E. Garcia, and E. F. Barbosa. A survey on graduates' curriculum-based knowledge gaps in software testing. In *Proceedings - Frontiers in Education Conference, FIE*, volume 2018-October, 2019. doi: 10.1109/FIE.2018.8658688. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-85063495090&doi=10.1109%2fFIE.2018.8658688&partnerID=40&md5=7aad288d22b347d063edbb25868e27a5. Export Date: 15 June 2023; Cited By: 1.

Eman Sherif, Andy Liu, Brian Nguyen, Sorin Lerner, and William G. Griswold. Gamification to aid the learning of test coverage concepts. In *2020 IEEE 32nd Conference on Software Engineering Education and Training (CSEET)*, pages 1–5, 2020. doi: 10.1109/CSEET49119.2020.9206224.

Philipp Straubinger, Laura Caspari, and Gordon Fraser. Code critters: A block-based testing game. In *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 426–429, 2023. doi: 10.1109/ICSTW58534.2023.00077.

StrykerMutator. Stryler visualmutator fetter. Available at https://stryker-mutator.io/, Access Date: 08 July 2023;.

Porfirio Tramontana, Beatriz Marín, Ana C. R. 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. In *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 59–69, 2024. doi: 10.1109/ICST60714.2024.00015.

Oscar Luis Vera-Pérez, Benjamin Danglot, Martin Monperrus, and Benoit Baudry. Suggestions on test suite improvements with automatic infection and propagation analysis. 2019. URL https://arxiv.org/abs/1909.04770.

Tanja E. J. Vos, Gordon Fraser, Ivan Martinez-Ortiz, Rui Prada, António Rito Silva, and I. S. W. B. Prasetya. Tutorial on a gamification toolset for improving engagement of students in software engineering courses. In *2020 IEEE 32nd Conference on Software Engineering Education and Training (CSEET)*, pages 1–3, 2020. doi: 10.1109/CSEET49119.2020.9206212.

QiaoLing Xie. Research and exploration on some problems in software testing course in colleges and universities. 01 2018. doi: 10.2991/icem-17.2018.55.